TECHNISCHE
UNIVERSITÄT
DARMSTADT

Modellbasierte Generierung und Reduktion von Testsuiten für Software-Produktlinien

VOM FACHBEREICH ELEKTROTECHNIK UND INFORMATIONSTECHNIK
DER TECHNISCHEN UNIVERSITÄT DARMSTADT
ZUR ERLANGUNG DES AKADEMISCHEN GRADES EINES
DOKTOR-INGENIEURS (DR.-ING.)
GENEHMIGTE DISSERTATION

VON

DIPL.-INF. HARALD CICHOS

GEBOREN AM 9. JANUAR 1981 IN KÖNIGS WUSTERHAUSEN

REFERENT:	PROF. DR. RER. NAT. A. SCHÜRR
KORREFERENT:	PROF. DR. RER. NAT. G. ENGELS

TAG DER EINREICHUNG:	22.APRIL 2013
TAG DER MÜNDLICHEN PRÜFUNG:	8.JULI 2013

D17

DARMSTADT 2013

ERKLÄRUNG

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, 22.*April* 2013

Harald Cichos

Die Mutter ist der Genius des Kindes.
— Georg Wilhelm Friedrich Hegel (1770–1831)

DANKSAGUNG

Für die fachlich konstruktive und menschliche Unterstützung während meiner Promotion möchte ich mich von ganzem Herzen bei meinem Doktorvater, Prof. Dr. Andy Schürr, sowie Malte Lochau, Siamak Haschemi und Thomas Heinze bedanken.

ZUSAMMENFASSUNG

Software-Produktlinienentwicklung ist ein Paradigma zur kostengünstigen Entwicklung vieler individueller aber sich ähnelnder Softwareprodukte aus einer gemeinsamen Softwareplattform heraus. Beispielsweise umfasst im Automotive-Bereich eine Software-Produktlinie (SPL) für ein Auto der Oberklasse typischerweise mehrere hunderttausend Softwaresystemvarianten. Um sicherzustellen, dass jede einzelne Produktvariante einer SPL in ihrer Funktionalität der Spezifikation entspricht, kann Testen verwendet werden. Da separates Testen jeder einzelnen Produktvariante meistens zu aufwändig ist, versuchen SPL-Testansätze die Gemeinsamkeiten der Produktvarianten beim Testen auszunutzen. So versuchen diese Ansätze geeignete Testartefakte wiederzuverwenden oder nur eine kleine repräsentative Menge von Produktvarianten stellvertretend für die ganze SPL zu testen. Da Software-Produktlinienentwicklung erst seit einigen Jahren verstärkt eingesetzt wird, sind im SPL-Test noch einige praxisnahe Probleme ungelöst. Beispielsweise existiert bisher kein Testansatz, mit dem sich eine gewisse Abdeckung bezüglich eines gewählten Überdeckungskriteriums auf allen Produktvarianten einer SPL effizient erreichen lässt.

In dieser Arbeit wird ein Black-Box-Testfallgenerierungsansatz für Software-Produktlinien vorgestellt. Mit diesem Ansatz lassen sich für alle Produktvarianten einer SPL eine Menge von Testfällen aus einer formalen Spezifikation (Testmodell), die mit Variabilität angereichert wurde, effizient generieren. Diese Testfallmenge, im Folgenden als *vollständige SPL-Testsuite* bezeichnet, erreicht auf jeder Produktvariante der SPL eine vollständige Abdeckung bzgl. eines strukturellen Modell-Überdeckungskriteriums. Die Effizienz des Ansatzes beruht auf der Generierung von Testfällen, die variantenübergreifend wiederverwendbar sind. Dadurch müssen mit dem neuen Ansatz weniger Testfälle generiert werden als wenn dies für jede Produktvariante separat geschieht. Um bei Bedarf die Anzahl der generierten Testfälle reduzieren zu können, werden außerdem drei Algorithmen zur Testsuite-Reduktion vorgestellt. Die Neuerung der vorgestellten Algorithmen liegt im Vergleich zu existierenden Reduktionsalgorithmen für Testsuiten von Einzel-Softwaresystemen darin, dass die Existenz von variantenübergreifend verwendbaren Testfällen in einer SPL-Testsuite berücksichtig wird. Dadurch wird sichergestellt, dass trotz Testsuite-Reduktion die vollständige Testmodellabdeckung einer jeden Produktvariante durch die SPL-Testsuite erhalten bleibt. Sollte es aufgrund limitierter Ressourcen nicht möglich sein jede Produktvariante mit den in der vollständigen SPL-Testsuite enthaltenen Testfällen zu testen, kann mittels einer SPL-Testsuite eine kleine repräsentative Produktmenge aus der SPL bestimmt werden, deren Testergebnis (im begrenzten Rahmen) Rückschlüsse auf die Qualität der restlichen Produktvarianten zulässt. Zur Evaluation des Ansatzes wurde dieser prototypisch implementiert und auf zwei Fallbeispiele angewendet.

ABSTRACT

Software product line engineering is a paradigm for low-cost development of many individual but similar software products from a common software platform. For example, in automotive industry a software product line (SPL) of a software system for a luxury vehicle comprises typically more than hundreds of thousands variants. Testing can be used to ensure that the functionality of each single product variant of an SPL complies with its specification. Since testing each product variant separately is too expensive, existing approaches exploit the commonalities of these product variants. Therefore, these approaches reuse appropriate test artifacts or test only a small representative set of variants in place of the entire SPL. Since software product line engineering is a young discipline, some practical problems in SPL testing remain unresolved. For example, so far, no approach exists that achieves a certain coverage with respect to a chosen test coverage criterion for each product variant of an SPL efficiently.

In this thesis a black-box test case generation approach is presented for software product lines. This approach can be used to efficiently generate a set of test cases for all variants of an SPL based on a formal specification (test model) that is enriched with variability. This set of test cases represents a *complete SPL test suite* that achieves for each product variant of this SPL a complete coverage with respect to structural test model coverage criteria. The efficiency of this approach based on the generation of test cases that are applicable to various product variants. Because of this re-use of test cases, fewer test cases have to be generated compared to generating these separately for each product variant. Additionally, three algorithms are presented for test suite reduction to further reduce the number of generated test cases. The novelty of these algorithms compared to existing reduction algorithms for test suites of single software systems is the consideration of re-useable test cases, which are applied to more than one variant, in a test suite. During test suite reduction, this consideration prevents the unintended decrease of the achieved test model coverage of each product variant. If testing each product variant with appropriate test cases of the complete SPL test suite is not possible due to limited resources, a complete SPL test suite can be used to identify a small representative set of product variants of the SPL. The test results of this representative set allow (to a certain degree) to draw inferences about the quality of the remaining variants. The approach has been implemented as a prototype and applied to two case studies for evaluation.

VERÖFFENTLICHUNGEN

Einige in dieser Arbeit vorgestellten Ideen und Texte wurden bereits teilweise in folgenden wissenschaftlichen Arbeiten publiziert:

Cichos, H. ; Oster, S. ; Lochau, M. ; Schürr, A.: Model-based Coverage-Driven Test Suite Generation for Software Product Lines. In: *(MoDELS'11) Proceedings in the International Conference Model Driven Engineering Languages and Systems* Bd. 6981, Springer, 2011 (LNCS), S. 425–439

Cichos, H. ; Heinze, T.: Efficient Test Suite Reduction by Merging Pairs of Suitable Test Cases. In: *Proceedings of Models in Software Engineering: Reports and Revised Selected Papers of Workshops and Symposia at MoDELS'10* Bd. 6627, Springer, 2011 (LNCS), S. 244–258

Cichos, H. ; Lochau, M. ; Oster, S. ; Schürr, A.: Reduktion von Testsuiten für Software-Produktlinien. In: *(SE'12) Beiträge zur Software Engineering Konferenz*, 2012 (GI-Edition Lecture Notes in Informatics), S. 143–154

Cichos, H. ; Heinze, T.: Efficient Reduction of Model-based Generated Test Suites through Test Case Pair Prioritization. In: *(MoDeVVa'10) Workshop on Model-Driven Engineering, Verification and Validation*, IEEE, 2010, S. 37–42. – Best-Paper-Award

Cichos, H. ; Schürr, A.: Dynamische Äquivalenzklassen im Klassifikationsbaum für zustandsbehaftete Systeme. In: *(Informatik'10) Beiträge der 40. Jahrestagung der Gesellschaft für Informatik e.V.* Bd. 176. Bonn : Gesellschaft für Informatik, 2010 (LNI 2), S. 345–350

Lucio, L. ; Weissleder, S. ; Fondement, F. ; Cichos, H.: Models in Software Engineering. In: *MoDeVVa 2011 Workshop Summary* Bd. 7167. Springer, 2012, S. 183–186

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

AKRONYME

CTL Computation Tree Logic

DNF Disjunktive Normalform

FODA Feature-orientierte Domänenanalyse

LTL Linear Temporal Logic

MBT Modellbasiertes Testen

OCL Object Constraint Language

OMG Object Management Group

SPL Software-Produktlinie

SUT System Unter Test

UML Unified Modeling Language

1

EINLEITUNG

Software spielt heute eine immer größere Rolle und ist aus vielen Bereichen, wie Medizin, Telekommunikation und Automotive, nicht mehr weg zu denken. Beispielsweise wird Software im Automotive-Bereich bereits bei einem Großteil der technischen Komponenten zur Steuerung eingesetzt [SL10]. Ein Auto der Oberklasse enthält typischerweise zwischen 80 bis 100 solcher Komponenten [MG08, FM12]. Damit nicht für jede Konfiguration, die sich aus den verwendeten Komponenten ergibt, das dazu passende Steuerungssystem von Grund auf neu entwickelt werden muss, wird Software-Produktlinienentwicklung betrieben.

Software-Produktlinienentwicklung ist ein Paradigma um viele individuelle aber dennoch ähnliche Softwareprodukte aus einer gemeinsamen Softwareplattform für ein bestimmtes Marktsegment zu entwickeln [PBL05]. Eine Softwareplattform stellt eine Menge wiederverwendbarer Entwicklungsartefakte zur Verfügung, deren Zusammenspiel durch eine gemeinsame Softwarearchitektur beschrieben wird. Durch Wiederverwendung der Entwicklungsartefakte lassen sich kundenindividuelle Softwareprodukte effizient entwickeln [VDLSR07]. Je nach Konfiguration eines Softwareprodukts werden unterschiedliche Entwicklungsartefakte bei der Entwicklung wiederverwendet. Das hat zur Folge, dass die Produkte untereinander Gemeinsamkeiten aber auch Unterschiede aufweisen, weshalb sie auch als Produktvarianten bezeichnet werden. Alle Produktvarianten zusammen bilden eine Software-Produktlinie (*SPL*). Aufgrund der Wiederverwendung von Entwicklungsartefakten lässt sich die Entwicklungszeit der einzelnen Produktvarianten verkürzen und zugleich deren Qualität steigern [OWES11].

Von einem im Auto installierten Softwaresystem wird eine sehr hohe Zuverlässigkeit erwartet, da dieses für sicherheitskritische Funktionen (z.B. ABS) verantwortlich ist, bei denen bereits ein einziger Fehler großen Schaden verursachen kann. Damit ein Softwaresystem korrekt funktioniert, wird während der Entwicklung Qualitätssicherung betrieben. Eine Form der Qualitätssicherung stellt das dynamische Testen dar. Beim dynamischen Test wird das zu testende Softwaresystem (*SUT*) mit Testfällen zur Ausführung gebracht. Durch den Vergleich der erwarteten Ausgabedaten mit den erzeugten Ausgabedaten können Fehler im SUT entdeckt werden. Da in der Praxis die Kosten im Testprozess beschränkt sind, muss eine Auswahl getroffen werden, welche Testfälle erstellt und auf dem SUT ausgeführt werden [Lig02]. Die ausgewählten Testfälle bilden eine *Testsuite*.

Aufgrund beschränkter Ressourcen können in der Praxis nicht alle Produktvarianten einer großen SPL (bestehend aus sehr vielen Produktvarianten) separat (Product-by-Product-Vorgehen [TTK04]) getestet werden. Aus diesem Grund wird nach geeigneten Ansätzen gesucht, mit denen sich die Qualität einzelner Produkt-

varianten selbst bei großen SPLs sicherstellen lässt. Zu diesem Zweck verfolgen bereits existierende Ansätze in der Regel eine der folgenden zwei Strategien [ER11], die darauf aufbauen, dass die von der Plattform bereitgestellten Entwicklungsartefakte bereits ausreichend getestet wurden.

- Die erste Strategie verfolgt das Ziel geeignete Testartefakte beim Test weiterer Produktvarianten wiederzuverwenden statt diese für jede Produktvariante erneut zu erstellen. Zu diesem Zweck kombinieren viele Ansätze [Was04, RKPR05, Oli08, WSS08, OMR10, LSKL12, LLSG12] den SPL-Test mit dem Modellbasierten Test (MBT). Beim MBT liegt die Spezifikation formal als Testmodell vor, sodass aus diesem systematisch Testfälle generiert werden können. Durch die Verwendung von Modellen können in der Regel Modellfragmente für geeignete Produktkonfigurationen ohne viel Aufwand wiederverwendet werden, wenn die Gemeinsamkeiten und Unterschiede dies zulassen. Ein Beispiel für Wiederverwendung von Modellfragmenten ist der Einsatz eines 150%-*Testmodells*. Ein 150%-Testmodell ist ein mit Variabilität angereichertes Testmodell. In diesem sind einige Modellelemente mit Bedingungen annotiert, die erfüllt sein müssen, damit diese Elemente im produktspezifischen Testmodell einer Produktkonfiguration enthalten sind. Durch Binden der Variabilität des 150%-Testmodells lässt sich aus diesem das produktspezifische Testmodell einer jeden Produktvariante herleiten. Demzufolge besteht jedes produktspezifische Testmodell aus einer Teilmenge von Modellelementen des 150%-Testmodells.

- Da es in der Regel selbst durch Wiederverwendung von Testartefakten nicht möglich ist alle Produktvarianten einer SPL zu testen, verfolgt die zweite Strategie das Ziel, nur eine kleine Teilmenge aller Produktvarianten zu testen [McG01]. Das *Produkt-Auswahlkriterium* muss dabei so gewählt werden, dass das Testergebnis der kleinen Menge Rückschlüsse auf die Qualität der restlichen Produktvarianten erlaubt. Wie groß die repräsentative Teilmenge wird, hängt maßgeblich vom gewählten Produkt-Auswahlkriterium ab. Üblicherweise erfordert das von der repräsentativen Menge zu erfüllende Kriterium eine Überdeckung von Anforderungen [Sch07a] oder basiert auf der Auswahl unterschiedlicher Kombinationen von Konfigurationsparametern der Produktvarianten einer SPL, ähnlich wie beim kombinatorischen Test [OMR10].

Da die Software-Produktlinienentwicklung und somit auch der SPL-Test erst seit einigen Jahren aufgrund des Trends zur individuellen Massenproduktion verstärkt in den Fokus der Wissenschaft gerückt ist [PBL05], existieren noch viele ungelöste, praxisnahe Probleme. Beispielsweise existiert bisher kein Testansatz, mit dem sich eine vollständige strukturelle Überdeckung bzgl. eines gewählten Modell-Überdeckungskriteriums auf dem produktspezifischen Testmodell einer jeden Produktvariante effizient erreichen lässt. Um das zu erreichen, müsste jedes durch das Überdeckungskriterium vorgegebene, erfüllbare Testziel (z.B. ein Modellelement) in jedem produktspezifischen Testmodell durch mindestens einen Testfall abgedeckt werden. Als Testmodelle können bei zustandsbehafteten Systemen einfache Automaten, bestehend aus Zuständen und Transitionen, verwendet

werden. Ein Testziel in solch einem Testmodell wäre folglich erfüllt, wenn ein Testfall zum Testen verwendet wird, dessen Pfad im Automaten das entsprechende Element bzw. Modellfragment traversiert. Wenn jedes erfüllbare Testziel durch mindestens einen Testfall in der Testsuite abgedeckt wird, erfüllt diese Testsuite das gewählte Überdeckungskriterium auf diesem Softwaresystem.

Wurde eine Menge von Testfällen erstellt, die auf dem produktspezifischen Testmodell einer jeden Produktvariante eine vollständige Abdeckung der erfüllbaren Testziele erreicht, handelt es sich dabei um eine *vollständige SPL-Testsuite*. Eine vollständige SPL-Testsuite lässt sich auf einfache, aber ineffiziente Weise erstellen, indem aus dem produktspezifischen Testmodell einer jeden Produktvariante eine produktspezifische Testsuite hergeleitet wird, die auf diesem Testmodell eine vollständige Überdeckung erreicht. Werden anschließend alle Testfälle aus diesen produktspezifischen Testsuiten zusammengefasst, erfüllt diese Menge von Testfällen die Eigenschaft einer vollständigen SPL-Testsuite. Allerdings ist das beschriebene Vorgehen zur Generierung einer vollständigen SPL-Testsuite für große SPLs ineffizient, da es sich aufgrund der separaten Bearbeitung um einen Product-by-Product-Ansatz [TTK04] handelt, bei dem sehr viele Testfälle generiert werden. Hinzu kommt, dass bei der Verwendung eines 150%-Testmodells zur Herleitung der produktspezifischen Testmodelle Testfälle erstellt werden können, die identisch sind mit Testfällen, die zuvor bereits für andere Produktvarianten erstellt wurden. Der Grund für diese identischen Testfälle ist der gemeinsame Ursprung der aus einem 150%-Testmodell hergeleiteten produktspezifischen Testmodelle. Aufgrund dieser identischen Testfälle und der separaten Bearbeitungsweise fällt die so erstellte vollständige SPL-Testsuite unnötig groß aus, was sich negativ auf die Kosten bei der Generierung, der Wartung und der Ausführung der Testfälle auswirkt. Damit die Generierung einer vollständigen SPL-Testsuite weniger Kosten verursacht, müssen effizientere Ansätze entwickelt werden.

Forderung 1 *Entwicklung eines Ansatzes zur Generierung einer vollständigen SPL-Testsuite, welcher die Erstellung von identischen Testfällen für unterschiedliche Produktvarianten einer SPL verhindert.*

Selbst bei Erfüllung von Forderung 1 kann die Anzahl der Testfälle und die damit einhergehende Anzahl an Testfallausführungen in einer vollständigen SPL-Testsuite sehr groß ausfallen. Um die Kosten für die Ausführung und Wartung der erstellten Testfälle bei Bedarf verringern zu können, bieten sich Testsuite-Reduktionstechniken an. Bei der Testsuite-Reduktion wird versucht, die Anzahl der Testfälle und die Anzahl der Testfallausführungen zu reduzieren, unter Rücksichtnahme darauf, dass die erreichte Abdeckung erhalten bleibt. Existierende Testsuite-Reduktionsansätze sind jedoch für Testsuiten von Einzel-Softwaresystemen ausgelegt und nicht für SPL-Testsuiten. Dadurch arbeiten diese Ansätze gegebenenfalls nicht effizient oder verringern sogar ungewollt die mit einer SPL-Testsuite erreichte vollständige Abdeckung auf einer Produktvariante der SPL.

Forderung 2 *Entwicklung eines Ansatzes zur Reduktion von SPL-Testsuiten, bei dem sowohl die Anzahl der Testfälle als auch die Anzahl der Testfallausführungen reduziert wird, ohne jedoch die erreichte Abdeckung bzgl. eines Überdeckungskriteriums auf irgendeiner Produktvariante zu senken.*

Wie bereits erwähnt können in der Praxis meistens nicht alle Produktvarianten einer großen SPL aufgrund zu hoher Kosten ausreichend getestet werden. Unter der Annahme, dass auf all diesen Varianten eine vollständige Modellüberdeckung angestrebt wird, könnte diesem Ziel unter bestimmten Zugeständnissen dennoch nahe gekommen werden. Dafür müsste eine repräsentative Menge von Produktvarianten identifiziert werden, die stellvertretend für die restlichen Produktvarianten getestet wird. Bisher existiert jedoch noch kein geeignetes Produkt-Auswahlkriterium zur Auswahl solch einer repräsentativen Produktmenge.

Forderung 3 *Identifikation eines Produkt-Auswahlkriteriums, mit dem sich eine Menge von Produktvarianten identifizieren lässt, deren Testergebnis Rückschlüsse auf die Qualität der restlichen Produktvarianten erlaubt, unter der Annahme, dass auf all diesen Varianten eine vollständige Modellüberdeckung angestrebt wird.*

1.1 WISSENSCHAFTLICHER BEITRAG

Mit dem in dieser Arbeit vorgestellten Ansatz lassen sich alle drei zuvor vorgestellten Forderungen erfüllen (siehe Forderung 1, 2 und 3). Jedoch wurden bei der Entwicklung des Ansatzes gewisse Einschränkungen bzw. Annahmen getroffen.

Da es sich bei dem neuen Ansatz um einen modellbasierten Black-Box-Testansatz für den SPL-Test handelt, der von einem 150%-Testmodell Verwendung macht, ist es zur Erfüllung der Forderungen 1 und 2 notwendig, dass dieses Modell bereits vorliegt oder aus der Spezifikation der SPL herleitbar ist. Außerdem wird zur Erfüllung der Forderung 3 angenommen, dass eine hohe Korrelation zwischen den Implementierungsartefakten und Testmodellartefakten bzgl. der variantenübergreifenden Gemeinsamkeiten existiert, damit die durch den Test der ausgewählten repräsentativen Produktmenge gewonnenen Testergebnisse auch Rückschlüsse auf die Qualität der ungetesteten Produktvarianten erlauben. Des Weiteren werden als Produktvarianten einer SPL zustandsbasierte, reaktive Softwaresysteme mit einem deterministischen Verhalten angenommen.

Zur Erklärung und Evaluation des Ansatzes werden in dieser Arbeit zwei an den SPL-Kontext angepasste Fallbeispiele verwendet, die in Abschnitt 1.2 vorgestellt werden. Im Folgenden wird der wissenschaftliche Beitrag dieser Arbeit zusammengefasst.

- In dieser Arbeit wird ein effizienter Ansatz zum Generieren einer vollständigen SPL-Testsuite vorgestellt. Diese SPL-Testsuite enthält für jede Produktvariante der SPL eine Menge an Testfällen, die auf dem jeweiligen produktspezifischen Testmodell eine vollständige Abdeckung erreicht bzgl. eines strukturellen Modell-Überdeckungskriteriums. Zu diesem Zweck werden direkt aus einem 150%-Testmodell Testfälle generiert, die in der Regel variantenübergreifend verwendbar sind. Durch den Einsatz variantenübergreifend verwendbarer Testfälle müssen weniger Testfälle generiert werden als wenn für jede Produktvariante separat Testfälle erstellt werden. Außerdem wird dadurch die Generierung identischer Testfälle (identisch bzgl. der abgedeckten Testziele) verhindert. Somit erfüllt der neue Ansatz Forderung 1.

- Um Forderung 2 nachzukommen, werden außerdem drei Testsuite-Reduktionsalgorithmen vorgestellt, die speziell für SPL-Testsuiten entwickelt wurden. Diese Algorithmen berücksichtigen variantenübergreifend verwendbare Testfälle in einer SPL-Testsuite und können dadurch sowohl die Anzahl der in einer Testsuite enthaltenen Testfälle als auch die Anzahl der benötigten Testfallausführungen reduzieren, ohne dabei die erreichte Überdeckung zu senken, wie es bei Testsuite-Reduktionsansätzen für Einzel-Softwaresysteme der Fall sein kann.

- Um Forderung 3 zu erfüllen, wird eine mit dem neuen Ansatz generierte vollständige SPL-Testsuite als Produkt-Auswahlkriterium verwendet. Mittels dieser SPL-Testsuite lässt sich eine in der Regel kleine Teilmenge von Produktvarianten in der SPL identifizieren, die repräsentativ dafür ist, dass jeder in der SPL-Testsuite enthaltene Testfall mindestens auf einer dieser Produktvarianten verwendbar ist. Werden anschließend all die in der SPL-Testsuite enthaltenen Testfälle auf den entsprechenden repräsentativen Produkten ausgeführt, lassen deren Testergebnisse Rückschlüsse auf die Qualität der restlichen Produktvarianten zu.

- Zur Realisierung des Ansatzes wurde Azmun [Has] verwendet. Azmun ist ein in der Programmiersprache Java entwickeltes Rahmenwerk für den modellbasierten Test. Die standardmäßig eingebundenen und zum Test von Einzel-Systemen ausgelegten Komponenten wurden durch eigene, an den SPL-Kontext angepasste Komponenten ersetzt. Diese ermöglichen unter anderem die Verarbeitung von 150%-Testmodellen und den Zugriff auf den Model-Checker SPIN [Holo3], der in dieser Arbeit als Testfallgenerator verwendet wird. Die während der Implementierung gewonnenen Erkenntnisse werden vorgestellt und diskutiert.

1.2 FALLBEISPIELE

Zur Erklärung und Evaluation des Ansatzes werden in dieser Arbeit zwei an den Kontext angepasste Fallbeispiele verwendet. Im ersten Fallbeispiel *Fahrkartenautomat* wird eine fiktive Steuerungssoftware für Fahrkartenautomaten betrachtet. Die Steuerungssoftware wird als einzelnes Softwaresystem eingeführt und im weiteren Verlauf der Arbeit zu einer Software-Produktlinie erweitert. Da dieses Fallbeispiel in dieser Arbeit durchgängig zur Vorstellung der Testbegriffe und Testtechniken verwendet wird, ist die Software-Produktlinie bestehend aus 8 Produktvarianten sehr klein gehalten. Das zweite Fallbeispiel *Body Comfort System* basiert auf einer realen Fallstudie [MLD+09] aus dem Automotive-Bereich, die während einer Kooperation zwischen dem Automobilhersteller Volkswagen [VW] und der TU Braunschweig entstanden ist. Das in dieser Studie verwendete Einzel-Softwaresystem steuert die Komfortfunktionen in einem Auto. In einer anschließenden Arbeit [OLZG11] wurde dieses Einzel-Softwaresystem zu einer komplexen Software-Produktlinie erweitert, die auch in dieser Arbeit verwendet wird (siehe Abbildung A.4).

1.3 GLIEDERUNG

Die vorliegende Arbeit ist wie folgt gegliedert:

- Da das Thema der vorliegenden Arbeit den Forschungsgebieten *Software-Produktlinien-Test* und *Testsuite-Reduktion* zugeordnet werden kann und diese beiden dem Forschungsgebiet *Softwaretest* angehören, wird zuerst in Kapitel 2 in dieses Forschungsgebiet eingeführt. Dabei werden die in dieser Arbeit verwendeten und die zum Verständnis des neuen Ansatzes notwendigen Begriffe vorgestellt.
- Anschließend wird in Kapitel 3 näher auf den modellbasierten Test, eine spezielle Variante des Softwaretests, eingegangen. Dabei liegt der Fokus insbesondere auf der Generierung von Testfällen aus formalen Modellen.
- In Kapitel 4 wird näher auf die Testsuite-Reduktion eingegangen. Dabei werden existierende Ansätze für Einzel-Softwaresysteme vorgestellt mit denen sich die Anzahl der in einer Testsuite enthaltenen Testfälle reduzieren lässt.
- In Kapitel 5 wird näher auf den Software-Produktlinien-Test eingegangen, da sich der in dieser Arbeit vorgestellte Ansatz hauptsächlich diesem Thema zuordnen lässt. In diesem Kapitel steht der Test mehrerer Produkte, welche auf derselben Softwareplattform basieren und dadurch Gemeinsamkeiten aufweisen, im Fokus und nicht, wie in den Kapiteln zuvor, der Test eines einzelnen Softwaresystems. Neben der Einführung in die Thematik und der Definition relevanter Begriffe werden auch themenverwandte Forschungsarbeiten vorgestellt. Durch diese lässt sich der Beitrag der vorliegenden Arbeit von bereits existierenden Arbeiten abgrenzen.
- In Kapitel 6 werden Begriffe und Methoden eingeführt, die eigens für den neuen Ansatz entwickelt wurden und in den darauffolgenden Kapiteln benötigt werden.
- In Kapitel 7 wird der neu entwickelte Ansatz zur Generierung eine Testsuite vorgestellt, die auf allen Produktvarianten einer SPL eine vollständige Überdeckung bzgl. eines gewählten Überdeckungskriteriums erreicht. Der dafür verwendete Algorithmus wird anhand der zwei Fallbeispiele evaluiert und die daraus gewonnenen Erkentnisse diskutiert.
- In Kapitel 8 werden verschiedene Algorithmen zur Reduktion solch einer mit dem vorgestellten Ansatz generierten Testsuite hinsichtlich der Testfallanzahl, der benötigten Anzahl an Testfallausführungen als auch der Gesamtlänge aller Testfälle vorgestellt und anhand der zwei Fallbeispiele evaluiert.
- In Kapitel 9 wird näher auf die prototypische Implementierung des Ansatzes und die dabei gelösten Herausforderungen eingegangen.
- Abschließend wird in Kapitel 10 die Arbeit zusammengefasst und ein Ausblick auf weitere Forschungsmöglichkeiten gegeben.

2

SOFTWARETEST

Software spielt heute eine immer größere Rolle und ist aus vielen Bereichen, wie Medizin, Telekommunikation und Automotive, nicht mehr weg zu denken. Beispielsweise werden im Automotive-Bereich bereits viele der eingesetzten technischen Komponenten durch Software gesteuert und überwacht [SL10]. Aufgrund des bisherigen Erfolgs dieser Technik ist davon auszugehen, dass sich der Trend auch in Zukunft fortsetzen und der Funktionsumfang von Software weiter zunehmen wird. Größerer Funktionsumfang geht in der Regel aber auch mit einem länger dauernden Entwicklungsprozess einher, was sich meistens negativ auf die Entwicklungskosten auswirkt. Um dennoch wirtschaftlich zu arbeiten, wird von Entwicklerseite aus versucht den Entwicklungsprozess so effizient wie möglich zu gestalten. Dabei besteht die Herausforderung darin, in möglichst kurzer Zeit eine Software zu entwickeln, die möglichst fehlerfrei ist und ihrer Spezifikation entspricht.

Definition 2.1 (Spezifikation) *„Ein Dokument, das die Anforderungen [...] des Systems bzw. der Komponente beschreibt, idealerweise genau, vollständig, konkret und nachprüfbar. [...]“ [Ger]*

Definition 2.2 (Anforderung) *„Eine [...] Eigenschaft, die eine Software [...] besitzen muss, um [...] eine Spezifikation [...] zu erfüllen.“ [Ger]*

2.1 SOFTWAREQUALITÄT

Die softwareseitige Realisierung der in einer Spezifikation beschriebenen Anforderungen wirkt sich auf die Qualität einer Software aus [Bal98, S.257]. Nach der internationalen Norm ISO 25000 [ISOa], die seit 2005 die Norm ISO 9126 [ISOb] ersetzt, lässt sich die Qualität einer Software auf die folgenden sechs Qualitätsmerkmale zurückführen:

- Funktionalität (functionality) beschreibt, inwieweit die in den Anforderungen beschriebene Funktionalität in der Software realisiert wurde.
- Zuverlässigkeit (reliability) „... beschreibt die Fähigkeit eines Systems, ein Leistungsniveau unter festgelegten Bedingungen über einen definierten Zeitraum zu bewahren.“ [SL10]

- Benutzbarkeit (usability) beschreibt den „... Aufwand, der zur Benutzung der Software erforderlich ist unter Berücksichtigung von unterschiedlichen Benutzergruppen.“ [SL10]
- Effizienz (efficiency) beschreibt „... die benötigte Zeit und den Verbrauch an Betriebsmitteln für die Erfüllung einer Aufgabe.“ [SL10]
- Änderbarkeit (maintainability) beschreibt den benötigten Aufwand, der erforderlich ist, um Änderungen an der Software durchzuführen.
- Übertragbarkeit (portability) beschreibt den Aufwand, der erforderlich ist, um die Software in eine andere Hardware- oder Software-Umgebung zu übertragen.

Wurde eine Anforderung fehlerhaft umgesetzt, kann solch ein Fehler die Qualität der Software beeinträchtigen. Im Allgemeinen wird unter dem Begriff *Fehler* die Nichterfüllung einer definierten Anforderung verstanden [SL10, S.249]. Für den Softwaretest ist dieser Begriff zu allgemein gehalten, weshalb in *Fehlhandlung*, *Fehlerzustand* und *Fehlerwirkung* unterschieden wird. Eine Fehlhandlung (error) findet immer dann statt, wenn eine in der Spezifikation definierte Anforderung von einem Entwickler nicht korrekt implementiert wird. Durch diese Fehlhandlung entsteht ein Fehlerzustand (fault) in der Software. Wird der Fehlerzustand während der Ausführung der Software erreicht, kann das eine Fehlerwirkung (failure) hervorrufen, die sich durch eine Abweichung im Ist-/Sollverhalten zeigt.

Eine besonders hohe Bedeutung hat die Softwarequalität bei sicherheitskritischen Softwaresystemen bzw. Komponenten, da in diesen bereits ein einziger Fehler zu Personenschäden bzw. hohen Kosten führen kann [Ger].

Definition 2.3 (Kosten) *Kosten sind eine Maßzahl, die den zu erbringenden Aufwand für eine Lösung eines Problems quantifiziert. Kosten können in verschiedenen Einheiten, wie zum Beispiel Geldwert, ausgedrückt werden.*

Die Kosten beim Testen beziffern sich beispielsweise in der Anzahl der Testfälle oder in der Anzahl der Testfallausführungen. Der in dieser Arbeit vorgestellte Ansatz bezieht sich auf die Qualitätssicherung der Funktionalität einer Software durch Testen.

2.2 QUALITÄTSSICHERUNG

Bereits während der Entwicklung von Software wird versucht darin enthaltene Fehlerzustände durch Qualitätssicherungsmaßnahmen zu finden und zu korrigieren, um die Softwarequalität zu steigern. Nach der ISO 9000 [Beu] sind unter dem Begriff *Qualitätssicherung* systematisch durchführbare Tätigkeiten zu verstehen, die eingesetzt werden, um Vertrauen in die korrekte Umsetzung der in der Spezifikation definierten Anforderungen zu schaffen. Dabei lassen sich Qualitätssicherungsmaßnahmen zur Validierung und Verifizierung von Software einsetzen.

- Bei der Validierung wird überprüft, ob ein Softwareprodukt die Anforderungen, die im Lastenheft des Kunden stehen, befriedigt. (nach [Lig02, S.514])
- Bei der Verifizierung wird überprüft, ob die Ergebnisse einer Entwicklungsphase die Anforderungen, die im Pflichtenheft des Arbeitnehmers stehen, erfüllen. (nach [Lig02, S.514])

Die Qualitätssicherungsmaßnahmen zur Validierung und Verifizierung von Software lassen sich in zwei Klassen unterteilen:

- Die konstruktive Qualitätssicherung versucht die Entstehung von Fehlern noch während der Entwicklung zu verhindern [RBGW10, S.16].
- Die analytische Qualitätssicherung versucht bereits existierende Fehler möglichst früh zu entdecken und die erreichte Qualität zu bewerten [RBGW10, S.16]. Die dafür eingesetzten Techniken lassen sich der *statischen Analyse* oder dem *dynamischen Test* zuordnen, wobei nur bei letzterem die Software zur Ausführung gebracht wird [Lig02, S.33].

Die Kosten für Maßnahmen zur Qualitätssicherung im Allgemeinen und das Prüfen und Testen im Speziellen belaufen sich bezogen auf die Entwicklungskosten auf 25% bis 50% oder höher [LL07, SL10]. Da der in dieser Arbeit vorgestellte Ansatz sich dem dynamischen Test zuordnen lässt, wird auf diesen im Folgenden näher eingegangen.

2.3 DYNAMISCHER TEST

Definition 2.4 (Testen) *„Testing is the process of operating a system or component under specified conditions, observing or recording the results and making an evaluation of some aspects of the system or component.“ [IEE]*

Beim dynamischen Test wird das Testobjekt (das Softwaresystem oder eine Komponente davon) mit konkreten Eingabedaten zur Ausführung gebracht, um das Verhalten des Testobjekts auf Fehlerwirkungen zu untersuchen. Durch die so gewonnenen Ergebnisse lässt sich die Qualität des Testobjekts quantifizieren. Sollten bei der Ausführung des Testobjekts Fehlerwirkungen aufgedeckt werden, lassen sich die entsprechenden Fehlerzustände nach dem Testen durch Debugging lokalisieren und entfernen.

In der Regel ist es durch Testen nicht möglich die Abwesenheit aller Fehlerzustände zu garantieren [Lig02]. Die einzige Ausnahme stellt der vollständige Test des Testobjekts dar, bei dem jeder Zustand im Zustandsraum des Testobjekts mit allen möglichen Kombinationen von Eingabedaten (ein Datum pro Eingabeparameter) auf Fehlerwirkungen hin überprüft wird [SL10].

Definition 2.5 (Zustandsraum) *Der Zustandsraum eines Automaten ist die Menge aller diskreten Zustände, die ein Automat annehmen kann.*

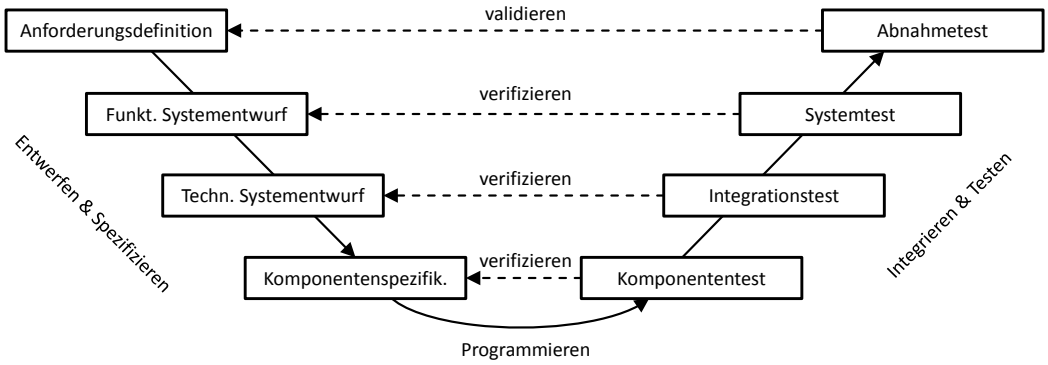


Abbildung 2.1: V-Modell
nach [Bal98, S.101] und [SL10, S.42]

Da Testobjekte in der Praxis meistens einen sehr großen Zustandsraum besitzen, ist vollständiges Testen meistens nicht empfehlenswert aufgrund der damit einhergehenden hohen Kosten. Stattdessen wird stichprobenartig überprüft, ob die definierten Anforderungen vom Testobjekt zumindest soweit erfüllt werden, dass im typischen operativen Betrieb keine kostspieligen Fehlerwirkungen zu erwarten sind. Diese Vorgehensweise ist in der Regel einfacher und schneller durchführbar als ein vollständiger Test. Durch systematisches Vorgehen im Testprozess sind die Ergebnisse reproduzierbar, was zur Folge hat, dass sich die Qualität einer Software objektiv messen lässt [LL07].

2.4 TESTSTUFEN

Je später im Softwareentwicklungsprozess Fehler entdeckt werden, umso höher fallen die zur Korrektur benötigten Kosten aus [SL10, S.185]. Aus diesem Grund sollte im Entwicklungsprozess so früh wie möglich mit dem Testen der bereits fertiggestellten Softwarekomponenten begonnen werden. Da sich die Testobjekte während des Entwicklungsprozesses stark ändern, wird der gesamte Testprozess typischerweise in Teststufen unterteilt. Die Teststufen orientieren sich dabei in der Regel an den Entwicklungsstufen des V-Modells [IB] (siehe Abbildung 2.1). Die Testobjekte der jeweiligen Teststufe werden dabei gegen die Spezifikation der gegenüberliegenden Entwicklungsstufe getestet [SL10, S.42]. Dadurch wird überprüft, ob die in der Entwicklungsstufe gestellten Anforderungen vom Testobjekt erfüllt werden. Voraussetzung für eine korrekte Auswertung ist, dass die Spezifikationen der vorangegangenen Entwicklungsstufen immer aktualisiert wurden, wenn Änderungen an der Spezifikation in späteren Entwicklungsstufen vorgenommen wurden. Nach [Bal98] wird Testen in den früheren Teststufen häufiger zur Verifizierung und in späteren Teststufen häufiger zur Validierung eingesetzt (siehe Abbildung 2.1). Der in dieser Arbeit vorgestellte Ansatz ist für die Teststufe *Systemtest* vorgesehen. Da auf dieser Teststufe ein Softwaresystem als Testobjekt verwendet wird, wird im Folgenden statt des Begriffs *Testobjekt* konkret vom zu testenden System (System Unter Test (SUT)) gesprochen.

2.5 TESTBEGRIFFE

Beim dynamischen Testen wird das zu testende System mit konkreten Testfällen zur Ausführung gebracht.

Definition 2.6 (Konkreter Testfall) *Ein Testfall für ein deterministisches Softwaresystem ist eine endliche Sequenz von Paaren, wobei sich jedes Paar aus einer Menge von konkreten Eingabedaten und einer Menge erwarteter konkreter Ausgabedaten zusammensetzt. (nach [GS05], [Ger, Glossar] und [Broo5, S.607])*

Bei den Daten kann es sich sowohl um Werte als auch um Signale handeln. Die erwarteten Ausgabedaten werden aus einem Testorakel [Wey82] abgeleitet, das in der Regel die Spezifikation des Testobjekts ist. Welchen Zweck bzw. welche Ziele ein Testfall erfüllen soll, wird noch vor dessen Erstellung in der entsprechenden Testfallspezifikation festgelegt. Der erstellte Testfall muss diese in der Testfallspezifikation beschriebenen Testziele unbedingt erfüllen, damit zwischen Testfall und Testfallspezifikation keine Inkonsistenz entsteht.

Definition 2.7 (Testfallspezifikation) *Eine Testfallspezifikation ist eine Beschreibung, in der die von einem Testfall zu erfüllenden Testziele festgelegt sind.*

Durch die Erstellung, Ausführung und Wartung (Aktualisierung der Testfallspezifikation) der Testfälle entstehen Kosten. Diese Kosten sind neben dem zeitlichen Aspekt ein Grund dafür, warum beim dynamischen Test nur ein Teil aller möglichen Testfälle ausgeführt werden kann. Solch eine endliche Teilmenge von Testfällen wird als Testsuite bezeichnet.

Definition 2.8 (Testsuite) *Eine Testsuite ist eine endliche Menge von Testfällen.*

Die Testfälle einer Testsuite sollten wenn möglich repräsentativ, fehlersensitiv, redundanzarm und ökonomisch sein [Lig02, S.36]. Um bei der Auswahl solcher Testfälle für die Testsuite systematisch vorzugehen, kann ein Testfall-Auswahlkriterium verwendet werden.

Definition 2.9 (Testfall-Auswahlkriterium) *Ein Testfall-Auswahlkriterium ist eine Eigenschaft, welche die ausgewählten Testfälle aus einer möglicherweise unendlich großen Menge wählbarer Testfälle erfüllen müssen.*

Welches Auswahlkriterium gewählt wird, ist abhängig davon, welches Szenario überprüft werden soll [GS05]. Ob das zu testende System (SUT) nach der Ausführung der ausgewählten Testfälle genügend getestet wurde, wird durch das Testendekriterium festgelegt.

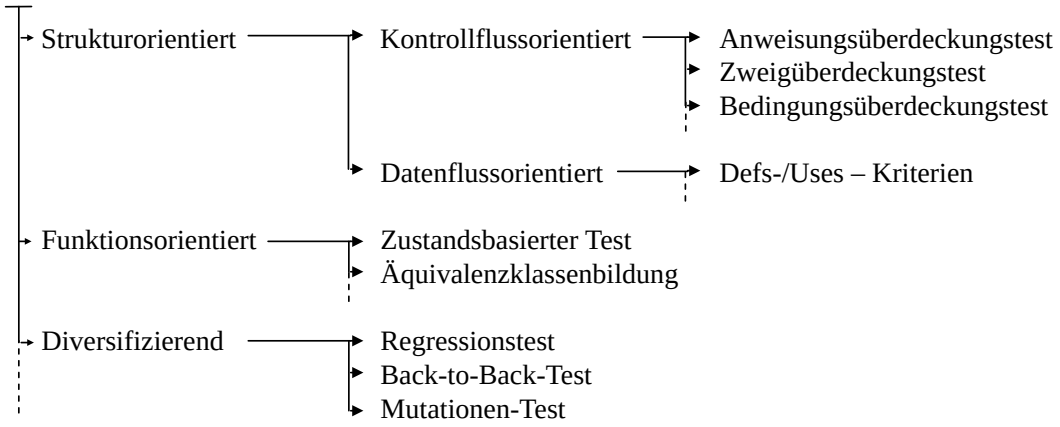


Abbildung 2.2: Klassifikation der dynamischen Testtechniken
gekürzt nach [Lig02, S.34]

Definition 2.10 (Testendekriterium) *Ein Testendekriterium ist eine Bedingung, deren Erfüllung zur Beendigung des Testprozesses führt.*

2.6 KLASSIFIKATION DER TESTTECHNIKEN

Die Beschränkung auf Stichproben beim Testen hat zu Testtechniken geführt, die unterschiedliche Ziele verfolgen und somit für bestimmte Situationen besser geeignet sind. Eine detaillierte Klassifikation der dynamischen Testtechniken ist in Abbildung 2.2 dargestellt. In der Praxis kommen den funktionsorientierten und einigen der kontrollflussorientierten Testtechniken eine besonders hohe Bedeutung zu. Im Allgemeinen werden alle funktionsorientierten Testtechniken den Black-Box-Testverfahren zugeordnet [Lig02, S.37]. Black-Box-Testverfahren verwenden zum Erstellen der Testfälle nur die Spezifikation aber nicht die Programmstruktur des Testobjekts. Auch bei der Bewertung der Testergebnisse wird nur das von außen sichtbare Verhalten verwendet. Folglich eignen sich Black-Box-Testverfahren zum Aufdecken von Fehlern in der Spezifikation, und sind deshalb prädestiniert für die Teststufen *Integrations-* und *Systemtest*. Die strukturorientierten Testtechniken lassen sich im Allgemeinen den White-Box-Testverfahren zuordnen [Lig02, S.37]. White-Box-Testverfahren verwenden die Programmstruktur des Testobjekts sowohl bei der Erstellung der Testfälle als auch bei der Bewertung der Testergebnisse. Somit eignen sich White-Box-Testverfahren besonders zur Aufdeckung von Fehlerzuständen in Komponenten und werden somit im Komponententest angewendet. Im folgenden Kapitel wird näher auf die funktionsorientierte Black-Box-Testtechnik *Zustandsbasierter Test* in Kombination mit *modellbasiertem Testen* eingegangen.

3

MODELLBASIERTES TESTEN

Das manuelle Erstellen von Testfällen durch einen Menschen basiert in der Regel auf informell definierten Anforderungen. Diese sind meistens in natürlicher Sprache verfasst und daher leicht verständlich. Dadurch konstruiert der Mensch typischerweise bereits beim Lesen der Anforderungen gedanklich ein mentales Modell des zu testenden Systems, welches er anschließend zur Herleitung von Testfällen verwendet.

Definition 3.1 (Modell) *Ein Modell ist ein für seinen Zweck abstrahiertes Abbild eines Originals (nach [Sta73]).*

Die durch ein Modell ermöglichte abstrahierte Sicht auf bestimmte Aspekte der Anforderungen reduziert die Komplexität bei der Testfallerstellung. Aufgrund der Nutzung eines mentalen Modells kann solch ein Vorgehen im wörtlichen Sinne bereits als modellbasiertes Testen verstanden werden [Binoo]. Jedoch lässt sich dieses Vorgehen nicht der Testtechnik *Modellbasiertes Testen* zuordnen, da dafür im Allgemeinen ein formales, eindeutig reproduzierbares und von Anderen einsehbares (z.B. grafisch spezifiziertes) Modell gefordert wird. In der Regel werden dafür dieselben Modelle verwendet, die auch in der eigentlichen Softwareentwicklung eingesetzt werden.

„Modellbasiertes Testen (MBT) hat zum Ziel, Prinzipien der modellbasierten Softwareentwicklung auf den Test zu übertragen. Es umfasst die Nutzung von Modellen für die Automatisierung von Testaktivitäten und die Modellierung von Testartefakten.“ [RBGW10]

Definition 3.2 (Testartefakt) *Ein Testartefakt ist ein Teilergebnis aus dem Testprozess (z.B. Testfall, Testmodell, Testfallspezifikation oder Testergebnis).*

Im Testprozess kann modellbasiertes Testen unterschiedlich stark ausgeprägt zur Anwendung kommen. Nach [RBGW10, S.122] existieren folgende drei, aufeinander aufbauende Ausprägungen:

1. Modellorientiertes Testen - Nutzung von Modellen als Diskussionsgrundlage, zur Visualisierung von Systemeigenschaften und als Orientierungshilfe bei der manuellen Testfallerstellung.

2. Modellgetriebenes Testen - Nutzung von Modellen zur Generierung von Testartefakten.

3. Modellzentrisches Testen - Alle für den Testprozess relevanten Informationen (z.B. für das Testmanagement) und Testartefakte liegen zwecks einheitlicher Darstellung in Modellform vor oder werden nach deren Erstellung in ein Modell übertragen.

In dieser Arbeit kommt MBT in der zweiten Ausprägung (modellgetriebenes Testen) zur Anwendung, weshalb im Folgenden näher darauf eingegangen wird.

3.1 MODELLGETRIEBENES TESTEN

Beim modellgetriebenen Testen werden Modelle einerseits als Leitfaden für Testaktivitäten und andererseits zur Generierung von Testartefakten, beispielsweise Testfälle, benutzt. Für den Test nicht benötigte Informationen über das zu testende System werden bei der Modellierung abstrahiert, sodass schließlich nur die für den Test relevanten Informationen im Modell enthalten sind. Eine zu starke Abstraktion kann jedoch dazu führen, dass sich wichtige Testfälle nicht mehr aus dem Modell herleiten lassen.

Bei der Erstellung der formalen Modelle muss darauf geachtet werden, dass die Modelle verständlich, zweckmäßig, korrekt und formal sind [RBGW10, S.224]. Sind die letzten zwei Eigenschaften erfüllt, lassen sich aus solchen Modellen Testfälle mittels Testfallgeneratoren automatisch erstellen. Aus formalen Modellen können sowohl konkrete als auch abstrakte Testfälle generiert werden. Abstrakte Testfälle entstehen unter anderem dadurch, dass bei der Modellierung des Modells für den Test relevante Informationen abstrahiert wurden. In diesem Fall müssen die abstrakten Testfälle vor ihrer Ausführung mit den dafür benötigten Informationen beispielsweise durch Testtreiber angereichert werden [PP05].

Die beim MBT verwendeten formalen Modelle werden ebenfalls wie die mentalen Modelle aus den informellen Anforderungen abgeleitet, die aufgrund der in natürlicher Sprache verfassten Texte widersprüchlich, redundant und mehrdeutig sein können [RBGW10, S.49]. Aufgrund der Notwendigkeit im MBT die Modelle formal, und somit maschinenlesbar, spezifizieren zu müssen, werden bereits während der Erstellung der formalen Modelle häufig Fehler/Unstimmigkeiten in der informellen Spezifikation aufgedeckt und Unvollständigkeiten häufiger erkannt als bei mentalen Modellen.

Nach der Erstellung eines formalen Modells lassen sich aus diesem systematisch und ohne menschliche Fehlhandlung Testfälle generieren, was zu einer Effizienzsteigerung im Testprozess führt. Beispielsweise können im MBT Testfälle im Vergleich zur manuellen Testfallerstellung ohne viel Aufwand erneut generiert werden, wenn Änderungen an der Spezifikation bzw. am formalen Modell vorgenommen werden.

3.2 MODELLKATEGORIEN

Die im MBT verwendeten Modelle lassen sich in *Testmodelle*, *Systemmodelle* und *Umgebungsmodelle* kategorisieren [RBGW10, MGEW12]. Je nach Ursprung der Modelle handelt es sich beim MBT um ein Black-, Grey- oder White-Box-Testverfahren [UL07]. Bei der Modellierung eines Modells ist darauf zu achten, dass nur Modelliertes getestet werden kann. Sollen neben dem eigentlichen Verhalten auch noch mögliche Fehl-Anwendungsfälle getestet werden, müssen diese ebenso modelliert werden. Damit Modelle auch als Testorakel dienen können, müssen diese auch das zu erwartende Verhalten des SUT beschreiben [RBGW10, S.193].

Testmodelle werden speziell für den Test aus der informellen Spezifikation hergeleitet. Bei der Herleitung eines Testmodells entsteht gewollt Redundanz (beispielsweise bezüglich der Funktionalität) zu dem ebenfalls aus der Spezifikation erstellten SUT. Wenn anschließend trotz Herleitung aus derselben Spezifikation an einigen Stellen des Testmodells und der Implementierung keine Redundanz vorliegt, kann das auf Fehler im SUT oder im Testmodell hinweisen. Testmodelle werden in der Regel auf der Teststufe *Systemtest* zur Validierung funktionaler Anforderungen eingesetzt, da die für den Test verantwortliche Abteilung in der Regel keinen Zugriff auf bereits existierende Entwicklungsartefakte (z.B. Systemmodelle) hat [RBGW10]. Sollte der Systemtest von einer externen Organisation durchgeführt werden, wie im Automotive-Bereich üblich, kann das Testmodell sogar aus einer völlig anderen Spezifikation erstellt werden [PP05]. Der Einsatz eines Testmodells im Testprozess ist nur dann sinnvoll, wenn bereits zu Beginn sichergestellt werden kann, dass die Initialkosten zum Erstellen des Testmodells und die laufenden Kosten für dessen Wartung geringer ausfallen, als die Kosten für das manuelle Erstellen aller über den Testprozess hinweg benötigten Testartefakte. Sollte das Erstellen eines eigenständigen Testmodells zu hohe Kosten verursachen, besteht auch die Möglichkeit ein Testmodell aus einem Systemmodell herzuleiten.

Bei Systemmodellen handelt es sich um Modelle, die bereits bei der Entwicklung des SUT als Implementierungsvorlage und/oder zur Codegenerierung verwendet wurden. Wenn Systemmodelle im MBT zur Testfallgenerierung unverändert eingesetzt werden, können die daraus generierten Testfälle lediglich zum Test des Codegenerators oder der Installationsumgebung eingesetzt werden, da der Code und die Testfälle aus demselben Modell generiert werden. In dem Fall entspricht das Ist-Verhalten immer dem erwarteten Soll-Verhalten, was zur Folge hat, dass die so erstellten Testfälle keine Fehler im SUT finden können. Um mit Systemmodellen Fehler entdecken zu können, müssen diese zumindest in Teilen überarbeitet werden, sodass Redundanz entsteht, die zum Aufdecken von Fehlern im SUT benötigt wird. Am ehesten eignen sich Systemmodelle im MBT für den Komponententest, um durch White-Box-Tests Implementierungsfehler zu finden [RBGW10].

Umgebungsmodelle werden zur Modellierung der Struktur und des Verhaltens der Umgebung eingesetzt. Umgebungsmodelle beschreiben Nutzungsszenarien der Systemschnittstelle bzw. die Interaktion mit der Systemumgebung, die vom Testmodell oder Systemmodell berücksichtig werden können (siehe Abbil-

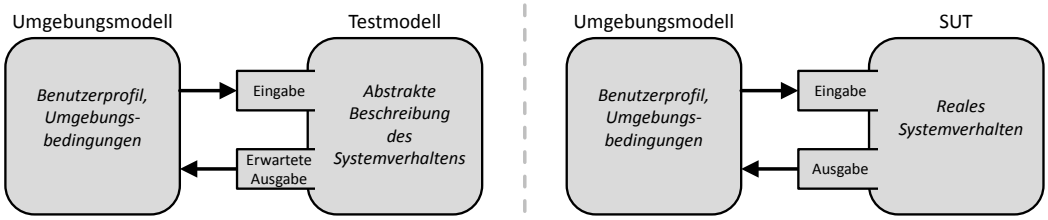


Abbildung 3.1: Anwendungsmöglichkeiten des Umgebungsmodells im MBT
angelehnt an 4-Variablen-Modell von [PM95]

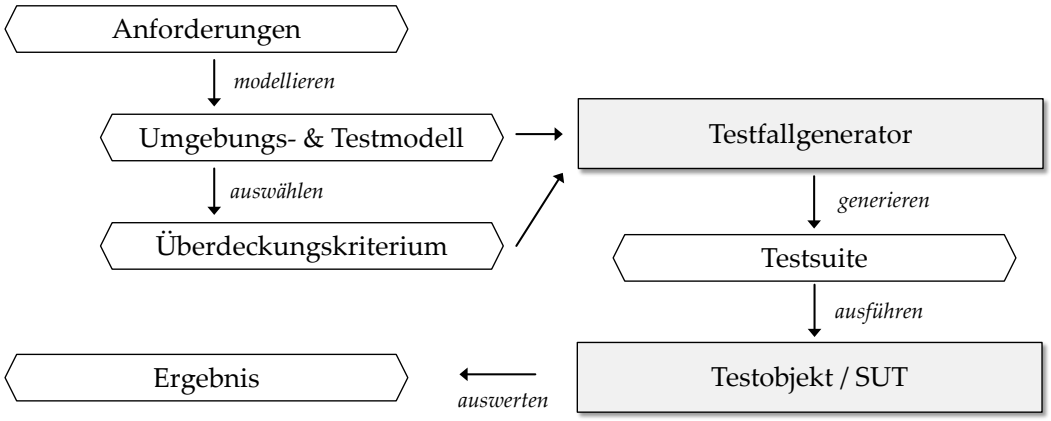


Abbildung 3.2: Modellgetriebener Testprozess

dung 3.1). Beschreiben lässt sich das Umgebungsverhalten beispielsweise durch Zustandsdiagramme, Ablaufdiagramme oder stochastische Aussagen. Letztere geben die Wahrscheinlichkeiten für das Auftreten von Ereignissen an [RBGW10, S.91]. Aus diesen Nutzungsprofilen lassen sich Anwendungsfälle ableiten, die Testfällen entsprechen.

Wenn heutzutage in der Praxis Modelle zum Testen verwendet werden, kommen häufiger Systemmodelle als Testmodelle zum Einsatz [RBGW10, S.216]. Dies ist darauf zurückzuführen, dass die Wiederverwendung eines existierenden Systemmodells weniger Kosten verursacht, als das Erstellen eines neuen Testmodells durch einen Experten. In dieser Arbeit wird davon ausgegangen, dass der in Abbildung 3.2 dargestellte modellgetriebene Testprozess angewendet wird, bei dem die Testfälle aus einem Testmodell hergeleitet werden. Prinzipiell lassen sich für den in dieser Arbeit vorgestellten Ansatz aber auch Systemmodelle verwenden.

3.3 MODELLIERUNGSSPRACHEN

Die im MBT verwendeten Modellierungssprachen geben Regeln für die Spezifikation und Konstruktion von Modellen vor. Durch grafische Elemente kann Verhalten und Struktur modelliert werden. Eine international standardisierte Modellierungssprache ist die Unified Modeling Language (UML) der Object Management Group (OMG), die sowohl in der Entwicklung als auch beim Test von Software eingesetzt wird. Die von der UML bereitgestellten Diagramme lassen sich unterteilen

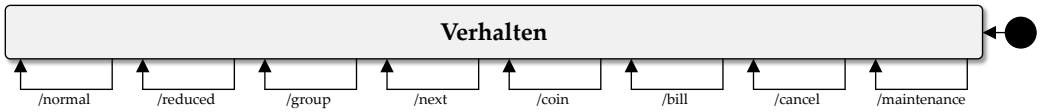
in Strukturdiagramme und Verhaltensdiagramme. Durch Strukturdiagramme lassen sich die statische Struktur und der modulare Aufbau eines Softwaresystems modellieren. Zur Beschreibung der möglichen Zustände einer Softwarekomponente können Attribute benutzt werden. Das Verhalten einer Komponente lässt sich durch Verhaltensdiagramme beschreiben. Verhalten findet statt, wenn durch Eingaben Operationen innerhalb der Komponente ausgelöst werden, die sich auf die Attribute und somit den Zustand der Komponente auswirken. Die UML bietet Aktivitätsdiagramme für Ablaufverhalten, Sequenzdiagramme für Interaktionsmodellierung und Zustandsdiagramme für Zustandsverhalten an. Informationen, die sich nicht mit grafischen Elementen der UML darstellen lassen, können mit der Object Constraint Language (OCL) formal beschrieben werden [UL07]. So lassen sich mit OCL beispielsweise Wertebereiche von Attributen einschränken oder logische Bedingungen für Transitionen angeben. Mögliche Werkzeuge zum Modellieren von UML-Zustandsdiagramme sind MagicDraw [NoM] oder Rhapsody [IBM]. Dabei wurde ersteres Werkzeug in dieser Arbeit zum Erstellen der verwendeten Modelle eingesetzt.

3.4 ZUSTANDSAUTOMATEN

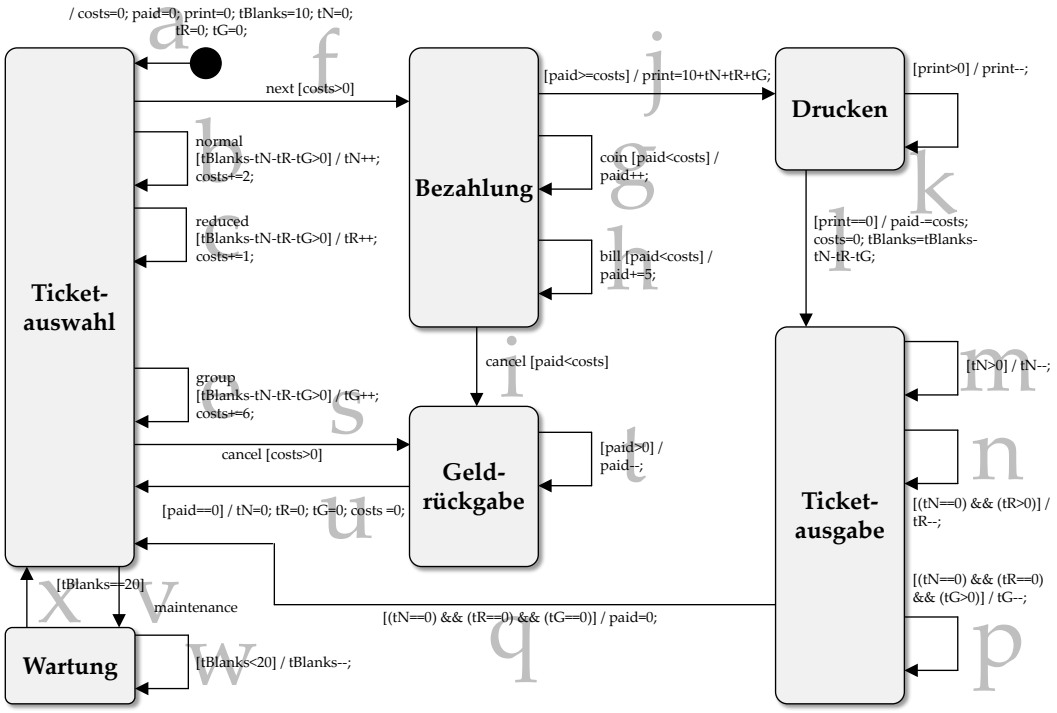
Der in dieser Arbeit vorgestellte Ansatz wurde vorrangig für den Test zustandsbasierter Softwaresysteme entwickelt. Zum Test dieser Systeme bietet sich die Testmethode *Zustandsbasierter Test* an, bei der die Testfälle aus einer in Form eines Zustandsautomaten vorliegenden Spezifikation hergeleitet werden [Bal98, S.56]. Aufgrund dieser grafischen Spezifikation liegt der Einsatz des Zustandsbasierten Tests im MBT-Kontext nahe.

Zustandsautomaten bestehen aus einer Menge von Zuständen und eine Menge von Transitionen (Zustandsübergänge). Eine Transition überführt den Automaten von einem Zustand in einen anderen, wenn ein gewisses Ereignis (z.B. ein von der Umwelt gesendetes Signal) unter gewissen Bedingungen eintritt. Für Zustandsautomaten existieren unterschiedliche Erweiterungen, die z.B. hierarchische oder orthogonale Zustände zulassen. Hierarchische Zustände erlauben die Unterteilung eines Zustands in mehrere Unterzustände wohingegen orthogonale Zustände Nebenläufigkeit ermöglichen. Der in dieser Arbeit vorgestellte Ansatz betrachtet nur einen Teil des Sprachumfangs der UML-Zustandsdiagramme. Unterstützt wird beispielsweise die Beschriftung von Transitionen mittels einer Standardgrammatik (z.B. PROMELA [Holo3]), die Aussagen und Zuweisungen über Attribute, die Werte aus einem bestimmten Wertebereich speichern können, ermöglicht. Die unterschiedlichen Belegungsmöglichkeiten der Attribute führt dazu, dass die grafisch dargestellten Zustände eines UML-Zustandsautomaten nur Äquivalenzklassen über den Zuständen des eigentlichen Zustandsraums (siehe Definition 2.5) des modellierten Testobjekts darstellen.

Zustandsautomaten lassen sich prinzipiell zur Modellierung eines zustandsbasierten Systems auf allen Teststufen anwenden [Sch07b], eignen sich aber insbesondere für die Teststufe *Systemtest* [Bal98, S.60]. Werden Zustandsautomaten als Testmodelle im Systemtest eingesetzt, modellieren diese in der Regel nicht nur mögliche Testabfolgen, sondern auch das zu erwartende Verhalten des zu testen-



(a) Umgebungsmodell



(b) Testmodell

Abbildung 3.3: Umgebungs- und Testmodell

den Systems (wie dieses auf Ereignisse unter Berücksichtigung seines aktuellen Zustands reagieren soll) bzw. die erwarteten Ausgaben. Somit kann das im Modell spezifizierte Verhalten als Testorakel eingesetzt werden [Bro05, S.609]. Da auch in dieser Arbeit der Systemtest im Vordergrund steht, wird das Verhalten des Testmodells und das Verhalten des Umweltmodells durch einen endlichen Zustandsautomaten beschrieben.

Beispiel

Im Fallbeispiel *Fahrkartenautomat* wird davon ausgegangen, dass eine Steuerungssoftware für Fahrkartenautomaten des öffentlichen Nahverkehrs modellbasiert getestet werden soll. In Abbildung 3.3 ist das Umgebungs- und Testmodell der zu testenden zustandsbehafteten Steuerungssoftware dargestellt. Das im Testmodell modellierte Verhalten ist abstrakt gehalten (siehe Abbildung 3.3b).

Der Automat startet im Zustand *Ticketauswahl*, in dem die Menge der zu erwerbenden Fahrkarten (normale, reduziert und für Gruppen) zusammengestellt werden kann. Im Zustand *Bezahlung* können die ausgewählten Fahrkarten durch Geldmünzen aber auch Geldscheine bezahlt werden. Sobald der ausstehende Geldbetrag beglichen oder mehr als nötig eingezahlt wurde, werden die Fahr-

karten gedruckt und ausgegeben. Sollte zuviel Geld eingezahlt worden sein, wird dieses nicht ausgegeben. Sollte der Bezahlvorgang abgebrochen werden, wird der bisher eingezahlte Geldbetrag im Zustand *Geldrückgabe* zurückgezahlt. Bei Bedarf können die zum Drucken der Fahrkarten benötigten Fahrkarten-Rohlinge im Zustand *Wartung* aufgefüllt werden. Generell kann der Automat aus jedem Zustand in den Zustand *Ticketauswahl* zurückkehren.

Insgesamt besteht das Testmodell aus 7 Zuständen (inkl. Startzustand) und 25 Transitionen (inkl. Initialtransition). Jede Transition ist so modelliert, dass ein deterministisches Verhalten im Automaten gewährleistet ist. Das Testmodell lässt sich auch als Testorakel verwenden, da das zu erwartende Verhalten der zu testenden Steuerungssoftware durch die Aktionen der Transitionen dargestellt wird.

Das Umgebungsmodell in Abbildung 3.3a besteht aus 2 Zuständen (inkl. Startzustand) und 9 Transitionen (inkl. Initialtransition). Alle 8 aus dem Zustand *Verhalten* ausgehenden Self-Transitionen benötigen kein Ereignis und werden durch keine Bedingung eingeschränkt. Somit können aus dem Umgebungsmodell beliebige Eingabesequenzen hergeleitet werden.

3.5 STRUKTURELLE ÜBERDECKUNGSKRITERIEN

Im MBT wird als Testendekriterium (siehe Definition 2.10) und als Testfall-Auswahlkriterium (siehe Definition 2.9) häufig ein strukturelles Überdeckungskriterium verwendet [RBGW10].

Definition 3.3 (Strukturelles Überdeckungskriterium) *Ein strukturelles Überdeckungskriterium ist ein Merkmal, das Elemente in einer Struktur aufweisen können. Es wird zur Auswahl einer Teilmenge von Elementen aus einer Struktur verwendet.*

Bei der Verwendung eines strukturellen Überdeckungskriteriums als Testendekriterium gilt das Testendekriterium als erfüllt, wenn die erstellten Testfälle bei ihrer Ausführung eine gewisse Überdeckung der mit dem Überdeckungskriterium ausgewählten Elemente erreichen. In diesem Kontext werden die Elemente auch als Testziele bezeichnet.

Definition 3.4 (Testziel) *Ein Testziel (engl. test goal) ist im MBT in der Regel ein Modellfragment (z.B. eine Transition). Ein Testziel gilt als erfüllbar, wenn ein Testfall erstellt werden kann, der das Testziel erreicht. Sobald solch ein Testfall erstellt wurde und in der Testsuite enthalten ist, gilt das Testziel als erfüllt.*

Im Zustandsbasierten Test kann ein Testfall für ein Testziel erstellt werden, der dieses Testziel erfüllt bzw. abdeckt, wenn im Zustandsraum der Automaten ein Pfad existiert, der zum Testziel führt.

Wenn ein Überdeckungskriterium als Testendekriterium verwendet wird, kann dieses Testendekriterium auf effiziente Weise erfüllt werden, indem dasselbe Über-

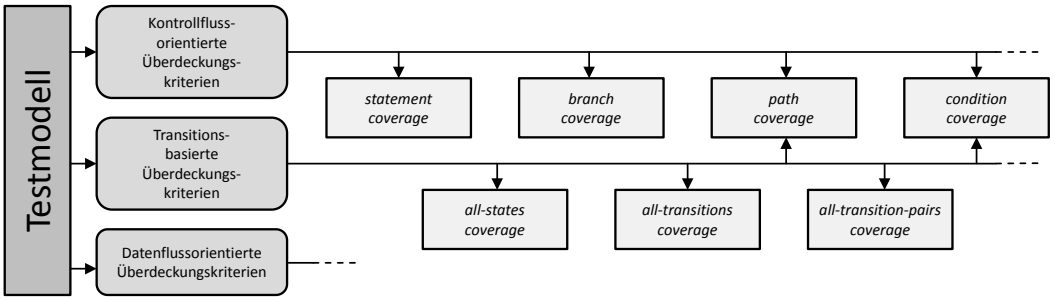


Abbildung 3.4: Auswahl struktureller Überdeckungskriterien
basierend auf [RBGW10, S.241]

deckungskriterium auch als Testfall-Auswahlkriterium verwendet wird. In diesem Fall wird für jedes durch das Überdeckungskriterium definierte Testziel mindestens ein Testfall erstellt bzw. ausgewählt, sodass anschließend eine vollständige Überdeckung der erfüllbaren Testziele erreicht wird, was zur Erfüllung des Testendekriteriums führt. In der vorliegenden Arbeit werden strukturelle Überdeckungskriterien als Testfall-Auswahlkriterium verwendet.

Die strukturellen Überdeckungskriterien lassen sich in datenflussorientierte, kontrollflussorientierte und zustandsübergangsbasierte Überdeckungskriterien unterteilen (siehe Abbildung 3.4). Im Folgenden wird auf diese näher eingegangen; eine darüber hinausgehende, ausführliche Übersicht und Erklärung ist in [UL07, S.107] zu finden.

Datenflussorientierte Überdeckungskriterien betrachten den Zugriff auf Variablen in der Modellstruktur und deren Manipulation. Kontrollflussorientierte Überdeckungskriterien beziehen sich auf den Kontrollfluss und die darin vorkommenden Bedingungen und Verzweigungen, wie sie beispielsweise in Aktivitätsdiagrammen vorkommen. Zustandsübergangsbasierte Überdeckungskriterien hingegen beziehen sich auf die Zustände und Transitionen eines zustandsbasierten Systems. Teilweise lassen sich kontrollflussorientierte Überdeckungskriterien aber auch auf die Transitionen in Zustandsdiagrammen anwenden, da diese sowohl Bedingungen als auch einen Kontrollfluss in ihren Aktionen besitzen können. In dieser Arbeit werden die transitionsbasierten Überdeckungskriterien *all-transitions* und *all-transition-pairs* als Auswahlkriterium für Testfälle bzw. zur Bestimmung von Testzielen eingesetzt.

- all-transitions : Alle Transitionen in einem Zustandsautomaten müssen abgedeckt werden.
- all-transition-pairs : Alle aneinander angrenzenden Transitionspaare in einem Zustandsautomaten müssen abgedeckt werden.

Das Überdeckungskriterium *all-transition-pairs* gilt gegenüber dem Überdeckungskriterium *all-transitions* als stärkeres Kriterium, da ersteres zweiteres subsumiert [UL07, S.120]. Als schwächstes in Abbildung 3.4 aufgeführtes transitionsbasiertes Überdeckungskriterium gilt *all-states*, wohingegen die *Pfadüberdeckung* als stärkstes gilt. Jedoch ist Pfadüberdeckung bei Testmodellen mit unendlich

tc#	Signal-Eingabesequenz	Transitionspfad
1		*a*
2	normal	*a b*
3	reduced	*a c*
4	group	*a e*
5	normal, next	*a b f*
6	normal, next, coin	*a b f g*
7	normal, next, bill	*a b f h*
8	normal, next, cancel	*a b f i*
9	reduced, next, coin	*a c f g j*
10	reduced, next, coin	*a c f g j k*
11	reduced, next, coin	*a c f g j k k k k k k k k k k l*
12	reduced, next, cancel, normal, next, coin, coin	*a c f i u b f g g j k k k k k k k k k k l m*
13	reduced, next, coin	*a c f g j k k k k k k k k k k l n*
14	reduced, next, cancel, group, next, coin, coin, bill	*a c f i u e f g g h j k k k k k k k k k k l p*
15	reduced, next, coin	*a c f g j k k k k k k k k k k l n q*
16	normal, cancel	*a b s*
17	reduced, normal, next, coin, cancel	*a c b f g i t*
18	reduced, next, cancel	*a c f i u*
19	maintenance	*a v*
20	maintenance	*a v w*
21	reduced, maintenance	*a c v w w w w w w w w w w x*

Abbildung 3.5: Testsuite mit redundanten Testfällen

vielen Pfaden nicht realisierbar, weshalb es in der Praxis wenig Relevanz hat. Eine ausführliche Übersicht über Überdeckungskriterien für Zustandsautomaten ist in [SMFMoo] zu finden.

Beispiel

Aus dem in Abbildung 3.3 dargestellten Testmodell lassen sich unendlich viele unterschiedliche Testfälle herleiten, da es aus jedem Zustand in seinen Ausgangszustand zurückkehren kann. Wird das Überdeckungskriterium *all-transitions* als Auswahlkriterium verwendet, stellt jede der 21 Transitionen ein Testziel dar, welches durch mindestens einen Testfall in der Testsuite abgedeckt werden muss. Dafür werden maximal 21 Testfälle benötigt. In Abbildung 3.5 sind 21 exemplarisch hergeleitete Testfälle dargestellt, von denen jeder einzelne konkret für ein Testziel erstellt wurde. Auf der linken Seite der Abbildung sind die konkreten Testfälle durch jeweils eine Signal-Eingabesequenz dargestellt. Auf der rechten Seite der Abbildung ist für jeden Testfall dessen Transitionspfad als abstraktere Form des Testfalls dargestellt. Um einen Transitionspfad nachvollziehen zu können, wurde jede Transition im Testmodell mit einem Bezeichner annotiert (siehe Abbildung 3.3b).

Bei der Testfallerstellung wird nach Testfällen gesucht, die mindestens die in ihrer Testfallspezifikation festgelegten Testziele abdecken (siehe Definition 2.7). Auf dem Pfad eines Testfalls zu dessen vorgegebenen Testzielen im Testmodell können sich jedoch noch weitere, nicht von der Testfallspezifikation geforderte Testziele befinden, die dann beiläufig („en passant“) abgedeckt werden.

Definition 3.5 (Beiläufig abgedecktes Testziel) *Ein beiläufig abgedecktes Testziel ist ein Testziel, welches von einem Testfall erfüllt wird, obwohl das nicht von der Testfallspezifikation gefordert wird.*

3.6 MODEL-CHECKER ALS TESTFALLGENERATOREN

Testfallgeneratoren erstellen automatisch Testfälle für Testziele aus formalen Modellen (z.B. Verhaltensmodelle) [RBGW10, S.111]. Zur Herleitung der Testziele wird dem Testfallgenerator neben dem eigentlichen Modell in der Regel auch ein Überdeckungskriterium als Auswahlkriterium übergeben. Als Testfallgenerator können unterschiedliche Werkzeuge eingesetzt werden. Wichtig ist letztendlich nur, dass das gewählte Werkzeug mit all den im Modell benutzten Konstruktionen wie z.B. Parallelisierung und hierarchische Verfeinerung umgehen kann. Bei größeren Testmodellen ist es außerdem wichtig, dass das gewählte Werkzeug auch mit der Zustandsraumexplosion umgehen kann, die in der Regel bei größeren, praxisnahen Testmodellen eintritt [Cla08].

Zur Evaluation des in dieser Arbeit vorgestellten Ansatzes wurde das Testframework Azmun [Has] eingesetzt, in das die Model-Checker SPIN [Hol03] und NuSMV [BCTT] als Testfallgeneratoren eingebunden werden können. Model-Checker wurden ursprünglich für die automatische Verifikation von Hardware-Zustandssystemen konzipiert [Cla08] und erst später zur modellbasierten Testfallgenerierung eingesetzt [CSE96, ABM98, CG08, GH99, HRV01, HLSC01, HCL+03, HU05, ZML07]. Model-Checker sind Werkzeuge, mit denen sich bestimmte Eigenschaften für einen endlichen Automaten vollautomatisch durch formale Verifikation beweisen lassen. Zu diesem Zweck wird der Zustandsraum (siehe Definition 2.5) des Automaten vollständig untersucht, um sicherzustellen, dass die zu verifizierende Eigenschaft in allen möglichen Zuständen erfüllt ist. Sollte während der formalen Verifikation eine Verletzung der Eigenschaft festgestellt werden, wird ein Gegenbeispiel erstellt. Dieses Gegenbeispiel stellt einen konkreten Pfad im Automaten dar, der vom Anfangszustand des Automaten zu dem Zustand führt, für den die Verletzung der Eigenschaft ermittelt werden konnte. Auf Grund dieser Fähigkeit können Model-Checker auch zur Generierung von Testfällen eingesetzt werden. Dazu werden Trap-Properties [GH99] verwendet.

Definition 3.6 (Trap-Property) *Eine Trap-Property ist eine vom Modell erfüllte Eigenschaft (z.B. „Ein Testziel ist erfüllbar"), die negiert wird, damit deren formale Verifikation fehlschlägt.*

Eine Trap-Property für ein erfüllbares Testziel φ könnte durch „Es existiert im Automaten kein Pfad, der das Testziel φ erreicht." beschrieben werden. Während der formalen Verifikation dieser Eigenschaft wird der Model-Checker erkennen, dass dieses Testziel erfüllbar ist, woraufhin ein Gegenbeispiel erstellt wird, welches einen entsprechenden Ausführungspfad zum Testziel φ enthält.

Wie jede andere mittels Model-Checking zu verifizierende Eigenschaft auch, werden Trap-Properties in Temporaler Logik beschrieben, die spezielle Operatoren für Aussagen über das zeitliche Verhalten des Modells besitzt. Zu den bekanntesten Temporalen Logiken gehören die Linear Temporal Logic (LTL) [Pnu77] und die Computation Tree Logic (CTL) [CE08].

Da Model-Checker ursprünglich für die formale Verifikation von Zustandsautomaten konzipiert wurden, sind ihre Fähigkeiten auf dem Gebiet der automatischen Testfallgenerierung begrenzt. So besteht die größte Schwäche von Model-Checkern bei der Testfallgenerierung darin, dass diese den Zustandsraum mit einer für die formale Verifikation und nicht für die Testfallgenerierung geeigneten Strategie untersuchen [FWA09]. Für die formale Verifikation wird eine Strategie eingesetzt, welche den vollständigen Zustandsraum untersucht und prüft, ob eine zu verifizierende Eigenschaft erfüllt ist oder nicht. Bei der Testfallgenerierung mit einem Model-Checker hingegen ist es in der Regel gar nicht notwendig den vollständigen Zustandsraum zu durchsuchen, wenn eine Trap-Property verwendet wird, für die sich mehrere Gegenbeispiele finden lassen. Folglich ist eine Strategie zum effizienten Durchsuchen des vollständigen Zustandsraums bei der Testfallgenerierung nicht optimal. Da in der Praxis die Zustandsräume der zur Testfallgenerierung verwendeten Modelle sehr groß ausfallen können, kann die Verwendung einer für die formale Verifikation ausgelegten Strategie bei der Untersuchung des Zustandsraums zu sehr langen Wartezeiten und hohen Speicherverbrauch führen. Diese Schwäche lässt sich jedoch relativieren, wenn bei der Modellerstellung darauf geachtet wird, dass der Zustandsraum nicht zu groß wird bzw. es zu keiner Zustandsraumexplosion kommt. Eine andere Möglichkeit diese Schwäche zu beseitigen besteht darin, die in Model-Checkern verwendeten Techniken in einem zur Testfallgenerierung konzipierten Werkzeug zu reimplementieren [JJ05].

Eine weitere Schwäche bei der Verwendung von Model-Checkern als Testfallgeneratoren stellen die generierten Testsuiten dar [FWA09]. Diese beinhalten häufig Testfälle, die identische Prefix-Sequenzen von Eingabedaten aufweisen oder sogar vollständig durch anderen Testfälle subsumiert werden. Das hat den Nachteil, dass sich durch die erneute Ausführung diese Sequenz, obwohl diese zu einem anderen Testfall gehört, keine neuen Fehler entdecken lassen.

4

TESTSUITE-REDUKTION

Während der Entwicklung einer Software werden ständig weitere Funktionen hinzugefügt oder bestehende modifiziert [RHRH02]. Da zum Testen der neuen oder modifizierten Funktionen neue Testfälle erstellt werden müssen, nimmt über die Zeit die Anzahl der in einer Testsuite enthaltenen Testfälle zu. Da die Ausführung eines jeden Testfalls Kosten verursacht (z.B. durch das Zurücksetzen eines technischen SUT in den Startzustand) kann ab einem gewissen Zeitpunkt die Ausführung aller in der Testsuite enthaltenen Testfälle hohe, nicht mehr vertretbare Kosten verursachen. Um die Anzahl der auszuführenden Testfälle gering zu halten bzw. mit einer großen Anzahl an Testfällen umgehen zu können, wurden unterschiedliche Verfahrenstechniken entwickelt von denen im Folgenden drei vorgestellt werden.

- Testfall-Selektion: Aus einer Menge existierender Testfälle werden nur diejenigen zur Ausführung ausgewählt, die benötigt werden, um den veränderten Teil einer Software zu testen (z.B. im Regressionstest). Die anderen, dafür nicht benötigten Testfälle werden nicht ausgeführt, was zur Folge hat, dass die Ausführungskosten sinken.

- Testfall-Priorisierung: Vorhandene Testfälle werden vor ihrer Ausführung nach einem Kriterium sortiert. Beispielsweise können die Testfälle so angeordnet werden, dass zuerst Testfälle mit einer potentiell hohen Fehlererkennungsrate ausgeführt werden, damit die Qualität der Software frühzeitig eingeschätzt werden kann.

- Testsuite-Reduktion: Die Anzahl der enthaltenen Testfälle soll reduziert werden, ohne die durch die Testfälle erreichte Abdeckung der Testziele zu senken. Die Reduktion ist vor, nach und während der Erstellung einer Testsuite möglich.

Eine ausführliche Übersicht über diese Verfahrenstechniken und eine Diskussion ihrer ungelösten Probleme bietet [YH08]. Im Folgenden wird näher auf die Testsuite-Reduktion, auch Testsuite-Minimierung genannt [FW07], eingegangen.

In der vorliegenden Arbeit besteht ein Teil des Beitrags in der Vorstellung einer neuen Testsuite-Reduktionstechnik, die speziell für Testsuiten im Software-Produktlinien-Test konzipiert wurde. Da bisher keine vergleichbaren Arbeiten auf dem Gebiet *Software-Produktlinien-Test* existieren, wird im Folgenden auf Testsuite-Reduktionstechniken eingegangen, welche für Testsuiten von Einzel-Softwaresystemen entwickelt wurden.

Arbeiten zur Testsuite-Reduktion lassen sich in zwei Kategorien unterteilen. Die Arbeiten der ersten Kategorie beschäftigen sich mit neuen Reduktionstechniken [PK10, CXZN08, FW07, TG05, HMR04, JH03, CL95, HGS93], wohingegen die anderen Arbeiten empirische Studien zu bereits existierenden Techniken vorstellen [ZZM06, RHRH02, RHOH98, WHLM95].

4.1 TESTSUITE-REDUKTIONSPROBLEM

Forschungsarbeiten zur Testsuite-Reduktion beschäftigen sich mit der Entwicklung geeigneter Ansätze zum Lösen des Testsuite-Reduktionsproblems, das erstmalig im Jahr 1993 in [HGS93] formal beschrieben wurde.

Definition 4.1 (Testsuite-Reduktionsproblem) *Es existieren eine Testsuite* $TS = \{tc_1, tc_2, ..., tc_m\}$ *und eine Menge von Testzielen* $TG = \{tg_1, tg_2, ..., tg_n\}$*, die durch Testfälle* $tc \in TS$ *abgedeckt sein müssen, um die geforderte Überdeckung zu erreichen.*

Finde eine Teilmenge von TS*, sodass die Anzahl der enthaltenen Testfälle minimal ist und jedes Testziele* $tg \in TG$ *von mindestens einem Testfall aus dieser Teilmenge abgedeckt wird.*

Folglich soll in einer Menge von Testfällen eine minimale Teilmenge identifiziert werden, die genau dieselben Testziele abdeckt wie die ursprüngliche Menge [CXZN08]. Um eine reduzierte, aber dennoch repräsentative Testsuite zu erlangen, müssen redundante Testfälle aus der ursprünglichen Testsuite entfernt werden. Im Folgenden werden zwei übliche Interpretationen des Begriffs *Redundanz* vorgestellt.

4.1.1 *Redundanz hinsichtlich der abgedeckten Testziele*

Nach dem Testsuite-Reduktionsproblem ist ein Testfall redundant, wenn dieser aus einer Testsuite entfernt werden kann und sich trotzdem die Anzahl der abgedeckten Testziele nicht ändert [JH03]. Durch das Entfernen eines redundanten Testfalls aus einer Testsuite ist somit garantiert, dass die Abdeckung der Testziele trotz Reduktion erhalten bleibt.

Beispiel

Die in Abbildung 3.5 dargestellte Testsuite enthält 21 Testfälle, von denen einige redundant hinsichtlich der abzudeckenden Testziele sind. Beispielsweise ist Testfall *tc*9 redundant, da all seine Testziele bereits von anderen Testfällen abgedeckt werden. Beispielsweise deckt bereits Testfall *tc*12 all die Testziele von Testfall *tc*9 alleine ab. Wird diese Testsuite reduziert indem redundante Testfälle entfernt werden, kann die in Abbildung 4.1 dargestellte Testsuite hergeleitet werden.

tc#	Transitionspfad
12	*a c f i u b f g g j k k k k k k k k k k k l* ***m***
14	*a c f i u* ***e*** *f g g* ***h*** *j k k k k k k k k k k k l* ***p***
15	*a c f g j k k k k k k k k k k k l* ***n q***
16	*a b* ***s***
17	*a c b f g i* ***t***
21	*a c* ***v*** *w w w w w w w w w w* ***w x***

Abbildung 4.1: Reduzierte Testsuite ohne redundante Testfälle

Durch das Entfernen eines redundanten Testfalls *tc* aus einer Testsuite TS, kann es passieren, dass ein zuvor ebenfalls redundanter Testfall tc' zu einem essentiellen Testfall wird, da dieser der einzige verbleibende Testfall in Testsuite TS ist, der ein gewisses Testziel abdeckt, welches zuvor auch von Testfall *tc* abgedeckt wurde.

Definition 4.2 (Essentieller Testfall) *Ein in einer Testsuite enthaltene Testfall ist essentiell, wenn dieser ein oder mehrere Testziele als einziger abdeckt (nach [CL95]).*

Folglich muss in einer Testsuite, die keine redundanten Testfälle mehr enthält, jeder enthaltene Testfall ein essentieller Testfall sein.

Beispiel
Beispielsweise ist der Testfall *tc*16 in der in Abbildung 4.1 ein essentieller Testfall, da, wenn dieser aus der Testsuite entfernt werden würde, kein anderer Testfall mehr das Testziel *s* abdecken würde.

Auch wenn eine Testsuite soweit reduziert wurde, dass diese keine redundanten Testfälle mehr enthält, muss es sich dabei nicht um die optimale Lösung, eine minimale Testsuite, handeln. Bei den praxisrelevanten Ansätzen zu diesem Thema handelt es sich lediglich um Heuristiken, die nicht zwangsläufig eine optimale Lösung finden, da das Testsuite-Reduktionsproblem NP-vollständig ist [HCL+03].

4.1.2 *Redundanz hinsichtlich der Eingabedatensequenz*

In empirischen Studien [RHOH98, WHLM95] konnte gezeigt werden, dass die Fehlersensitivität einer Testsuite sinken kann, selbst wenn nur solche Testfälle entfernt werden, die redundant hinsichtlich bereits abgedeckter Testziele sind.

Definition 4.3 (Fehlersensitivität einer Testsuite) *Die Fehlersensitivität einer Testsuite ergibt sich aus der Anzahl der Fehlerwirkungen, die durch die Testfälle der Testsuite aufgedeckt werden können.*

Dieser Umstand ist darauf zurückzuführen, dass durch das Abdecken aller Testziele meistens nur ein kleiner Teil aller möglichen Fehlerquellen im SUT überprüft

wird. Die Eingabedatensequenz, die ein Testfall benötigt um die von der Testfallspezifikation geforderten Testziele zu erreichen und somit abzudecken, überprüft meistens beiläufig noch andere Fehlerquellen. Aus diesem Grund existiert noch eine stärkere Forderung, die nur das Entfernen derjenigen Testfälle aus der Testsuite erlaubt, die redundant hinsichtlich ihrer Eingabedatensequenz sind. Somit ist ein Testfall tc gegenüber Testfall tc' nur dann redundant, wenn die gesamte Eingabedatensequenz von tc mit der Eingabedatenanfangssequenz von tc' identisch ist.

Beispiel

Beispielsweise ist in Abbildung 3.5 der Testfall $tc5$ redundant (hinsichtlich der Eingabesequenz) gegenüber dem Testfall $tc6$, da die gesamte Eingabesignalsequenz von Testfall $tc5$ der Eingabesignal-Anfangssequenz von Testfall $tc6$ entspricht.

Es ist offensichtlich, dass ein Testfall, der redundant bzgl. der Eingabedatensequenz ist, ebenfalls redundant bzgl. der abgedeckten Testziele sein muss, wenn Determinismus vorliegt. Die zweite, einschränkendere Interpretation des Begriffs *Redundanz* hat den Nachteil, dass damit oft weniger Testfälle aus der Testsuite entfernt werden können. Somit muss bei einer Testsuite-Reduktion abgewogen werden, welche der beiden Forderungen die geeignetere ist [RHOH98]. Die zweite Interpretation wird in Arbeiten [BR01, HMR04, FW07] verwendet, wenn die Fehlersensitivität einer Testsuite trotz Reduktion beibehalten werden soll.

In [FW07] wird darüber hinaus noch versucht, die Redundanz zu entfernen, die entsteht, wenn zwei oder mehr Testfälle dieselbe Eingabedatenanfangssequenz besitzen. Da durch das mehrmalige Ausführen derselben Anfangssequenz keine weiteren Fehlerwirkungen aufgedeckt werden können, versucht dieser Ansatz, die nicht identischen Endsequenzen durch eine neu generierte Verbindungssequenz miteinander zu verknüpfen. Anschließend kann ein Testfall, ohne die Fehlersensitivität der Testsuite zu senken, entfernt werden, wenn darauf geachtet wird, dass die Verbindungssequenz in denselben Zustand des Zustandsraums führt, von dem aus die Eingabedatenendsequenz des zu entfernenden Testfalls aus startet.

Beispiel

In Abbildung 4.2 sind zwei Testfälle tc' und tc'' dargestellt, die aufgrund einer identischen Testfalleingabedatenanfangssequenz dieselben Fehlerwirkungen aufdecken. Vom Zustand X des Zustandsraums des Testmodells ausgehend besitzen beide Testfälle eine unterschiedliche Eingabedatenendsequenz, dadurch weisen diese das Potential auf unterschiedliche Fehlerwirkungen aufzudecken. Um die durch die identischen Teilabschnitte eingeführte Redundanz zu entfernen und zugleich die Anzahl der Testfälle zu verringern, ohne die Fehlersensitivität der Testsuite zu senken, kann Testfall tc' um eine Verbindungssequenz erweitert werden, die vom Zustand Y zum Zustand X führt, sodass von dort aus mit der Endsequenz von tc'' fortgefahren wird. Anschließend ist Testfall tc'' vollständig redundant gegenüber Testfall tc', was zur Folge hat, dass tc'' aus der Testsuite entfernt werden kann.

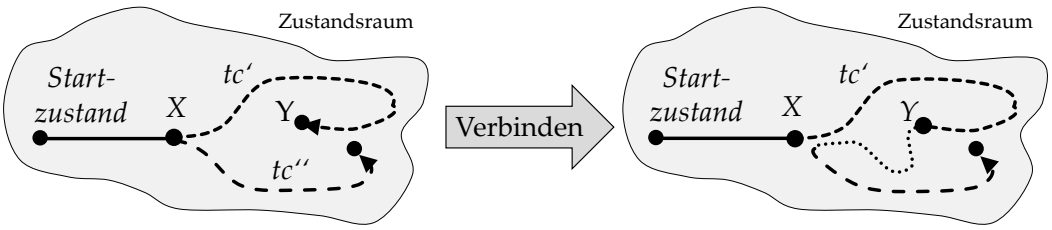


Abbildung 4.2: Reduktion mittels einer Verbindungssequenz

Dieses Vorgehen ist jedoch nur zulässig, wenn die Testfälle aus einem Modell generiert werden, das ebenfalls für die Implementierung (z.B. Systemmodell) verwendet wird. Diese Forderung ist notwendig, da ansonsten nicht garantiert ist, dass der Endzustand der Anfangssequenz des einen Testfalls auch dem Anfangszustand der Endsequenz des anderen Testfalls im Zustandsraum des SUT entspricht.

4.2 ZEITPUNKT DER REDUKTION

Reduktionsmaßnahmen können vor, während und nach der Erstellung einer Testsuite durchgeführt werden:

- In [KKT07] wird ein Testsuite-Reduktionsansatz vorgestellt, der *vor* der Erstellung einer Testsuite angewendet wird. So werden die Testziele, für welche die Testfälle erstellt werden sollen, so angeordnet, dass zuerst Testfälle für diejenigen Testziele erstellt werden, die mit hoher Wahrscheinlichkeit gleich noch weitere Testziele beiläufig mit abdecken (siehe Definition 3.5). Durch diesen Ansatz können Kosten eingespart werden, da weniger Testfälle erstellt werden, die später aufgrund ihrer Redundanz wieder entfernt werden.

- Bei Reduktionsansätzen, die *während* der Testsuite-Erstellung angewendet werden, wird nach jedem erstellten Testfall überprüft, welche Testziele bereits durch einen zuvor erstellten Testfall beiläufig (siehe Definition 3.5) abgedeckt werden. Existiert solch ein Testfall, muss für solche Testziele kein weiterer Testfall erstellt werden.

- Ansätze, die erst *nach* der Erstellung einer Testsuite und somit auf eine bereits existierende Testsuite angewendet werden, funktionieren so, wie im Testsuite-Reduktionsproblem beschrieben (siehe Abschnitt 4.1).

Der in der vorliegenden Arbeit vorgestellte Ansatz verhindert die Generierung von identischen (und somit redundanten) Testfällen im SPL-Kontext. Zusätzlich dazu werden Reduktions-Algorithmen für den SPL-Kontext vorgestellt, mit denen sich die Testfallanzahl einer bereits existierenden Testsuite reduzieren lässt ohne dabei die Anzahl der abgedeckten Testziele zu senken.

5

SOFTWARE-PRODUKTLINIEN-TEST

Die Techniken des Softwaretests sind im Allgemeinen für ein einzelnes, separat entwickeltes Softwareprodukt konzipiert. Diese Techniken gehen von der Annahme aus, dass die für dieses Produkt erstellten Testartefakte (siehe Definition 3.2) im Testprozess zukünftiger Produkte nicht wiederverwendet werden können. Jedoch trifft diese Annahme in der Regel beim Test der Produktvarianten einer Software-Produktlinie nicht zu.

Definition 5.1 (Software-Produktlinie) *Eine Software-Produktlinie besteht aus einer Produktlinienarchitektur, einer Menge wiederverwendbarer Entwicklungsartefakte und einer Menge an Produktvarianten, die sich aus diesen mehrfach wiederverwendbaren Artefakten heraus entwickeln lassen. (nach [Bos01])*

Da die Produktvarianten aufgrund der mehrfach wiederverwendeten Entwicklungsartefakte häufig untereinander viele Gemeinsamkeiten aufweisen, können Testtechniken eingesetzt werden, die Testartefakte von ähnlichen, bereits getesteten Produktvarianten wiederverwenden. Dadurch sind solche Techniken effizienter als wenn Testtechniken für Einzel-Softwaresysteme auf die Produktvarianten einer SPL angewendet werden.

5.1 ENTWICKLUNG VON SOFTWARE-PRODUKTLINIEN

Softwaresysteme lassen sich unterscheiden in Standardsoftware und kundenindividuelle Software. Kundenindividuelle Software wird für einen einzigen Kunden entwickelt und speziell auf dessen Anforderungen zugeschnitten, wohingegen Standardsoftware für den Massenmarkt entwickelt wird. Da die Kosten für die Reproduktion eines fertigentwickelten Softwaresystems im Vergleich zu dessen Entwicklungskosten kaum ins Gewicht fallen, kann Software, die für den Massenmarkt konzipiert wurde, wesentlich günstiger vertrieben werden als kundenindividuelle Software. Einen Kompromiss zwischen günstiger und individueller Software stellt die kundenindividuelle Massenproduktion für ein zuvor festgelegtes Marktsegment (Domäne) dar [PBL05]. Um kundenindividuelle Massenproduktion zu ermöglichen, wird eine Softwareplattform benötigt [PBL05].

Definition 5.2 (Softwareplattform) *„Eine Softwareplattform besteht aus einer Menge von Softwareteilsystemen und Schnittstellen, die ein gemeinsames Gerüst bilden, aus dem heraus eine Menge an Produktvarianten effizient entwickelt werden kann." (nach [VDLSR07, S.315])*

Die Entwicklung einer Softwareplattform verursacht gerade zu Beginn hohe Kosten. Um keine Fehlinvestition zu tätigen, sollten vor deren Entwicklung die Wünsche des Marktes (die Gemeinsamkeiten und Unterschiede, die potentielle kundenindividuelle Software aufweisen würde [CN01]) gewissenhaft analysiert werden [CHW98]. Ist dies geschehen, können die Teilsysteme, die viele Produkte gemeinsam haben, in jeder später zu entwickelnden Produktvariante kostengünstig wiederverwendet werden, wohingegen die zuvor identifizierten Unterschiede zur Kundenindividualität beitragen. Die Kosten für die Entwicklung einer Softwareplattform fallen geringer aus, wenn bereits zuvor entwickelte Individualprodukte aus derselben Domäne existieren, die sich als kostengünstige Informationsquelle verwenden lassen. Die Investition in eine Softwareplattform amortisiert sich gewöhnlich nach zwei bis drei daraus entwickelten Produkten [PBL05, WL99].

5.2 ENTWICKLUNGSRAHMENWERK FÜR SOFTWARE-PRODUKTLINIEN

Die Entwicklung einer Software-Produktlinie unterscheidet sich hauptsächlich von der Entwicklung eines einzelnen Softwaresystems darin, dass eine gemeinsam genutzte Softwareplattform existieren muss, bevor die kundenindividuellen Produkte entwickelt werden können. Diese Gegebenheit wird im Rahmenwerk zur Software-Produktlinienentwicklung [PBL05] durch folgende zwei ineinandergreifende Entwicklungsprozesse realisiert (siehe Abbildung 5.1):

- Domänenentwicklung (Domain Engineering)
- Produktentwicklung (Application Engineering)

Der Domänenentwicklungsprozess wird genutzt um eine robuste Softwareplattform zu erstellen, die auf eine bestimmte Domäne bzw. auf ein zuvor festgelegtes Marktsegment beschränkt ist [CN01]. Der Domänenentwicklungsprozess setzt sich aus fünf Teilprozessen zusammen (siehe Abbildung 5.1 oben), die in [PBL05] ausführlich beschrieben werden. In jedem der fünf Teilprozesse werden Domänenartefakte erstellt (z.B. variable Softwarekomponenten, variable Modelle und variable Testartefakte). Die in einem Teilprozess erstellten Domänenartefakte werden sowohl vom anschließenden Teilprozess als auch im Produktentwicklungsprozess genutzt. Aufgrund ihrer Variabilität können Domänenartefakte im Entwicklungsprozess geeigneter Produkte wiederverwendet werden. Damit die in den einzelnen Domänenartefakten eingefügte Variabilität auch konsistent zur Variabilität anderer Domänenartefakte anderer Teilprozesse bleibt, werden Traceability-Links zwischen diesen erstellt. Ebenso gibt jeder Teilprozess seinem vorherigen Teilprozess Feedback, damit die Qualität und die eingearbeitete Variabilität zueinander

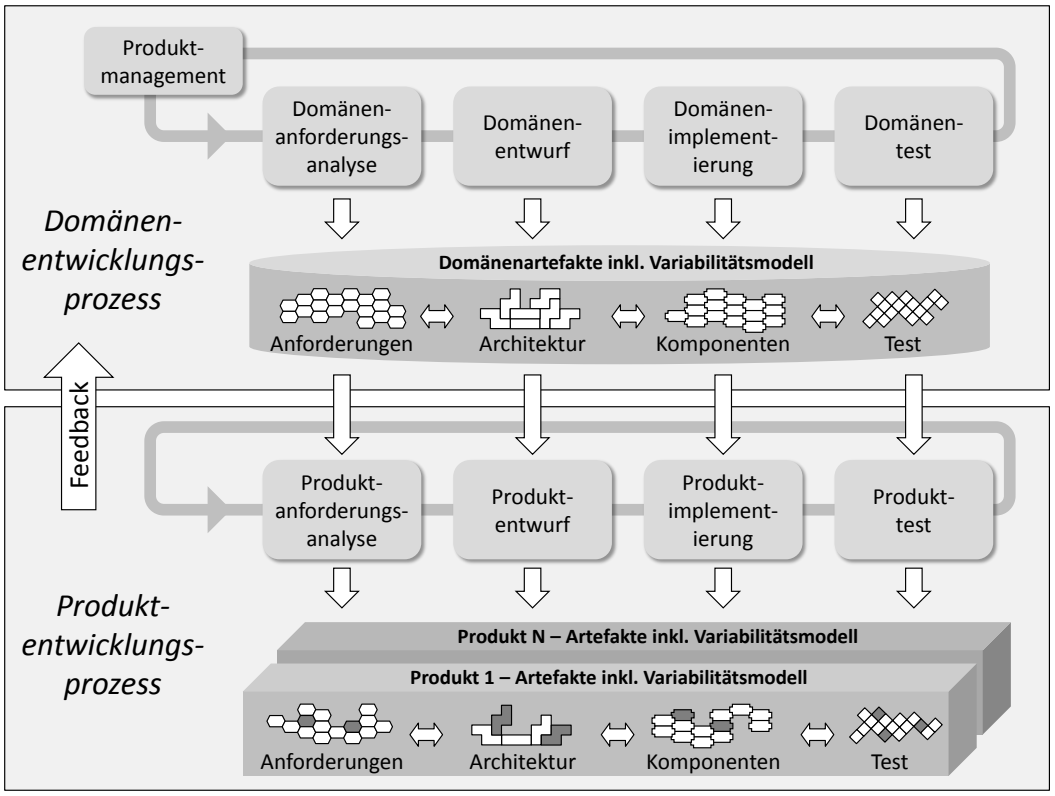


Abbildung 5.1: Entwicklungsrahmenwerk für Software-Produktlinien
übersetzt aus [VDLSR07, S.48]

konsistent bleiben. Da die vorliegende Arbeit einen Beitrag auf dem Gebiet des SPL-Tests leistet, wird in Abschnitt 5.4 näher auf den Teilprozess *Domänentest* eingegangen.

Der Produktentwicklungsprozess befasst sich mit der effizienten Entwicklung der einzelnen kundenspezifischen Produkte. Dafür werden die während der Domänenentwicklung erstellten Artefakte aus der Softwareplattform bestmöglich wiederverwendet. Die von der Plattform gebotene Variabilität wird gebunden, sobald die zur Erstellung des kundenspezifischen Produkts benötigten Artefakte ausgewählt wurden. Der gesamte Produktentwicklungsprozess setzt sich aus vier Teilprozessen zusammen (siehe Abbildung 5.1 unten) [PBL05]. Genauso wie beim Domänenentwicklungsprozess gibt auch hier jeder Teilprozess im Produktentwicklungsprozess Rückmeldung an seinen vorherigen Teilprozess. Zusätzlich zu dem Feedback zwischen den Teilprozessen ist auch Feedback vom Produktentwicklungsprozess zum Domänenentwicklungsprozess möglich [VDLSR07, S.48].

Zusammengefasst lässt sich sagen, dass während des Domänenentwicklungsprozesses die Variabilität explizit festgelegt und eine Softwareplattform entwickelt wird, welche dann während des Produktentwicklungsprozesses verwendet wird, um Produkte nach Wünschen des Kunden zu erstellen. Dabei ist das Ziel, die Entwicklungskosten und die Entwicklungszeit für die einzelnen Produktvarianten zu senken und dennoch deren Softwarequalität aufgrund der wiederverwendeten Artefakte zu steigern. Die zwei Entwicklungsprozesse des Rahmen-

werks können mit existierenden Entwicklungsmethoden (z.B. V-Modell [IB] oder Rational Unified Process (RUP) [Vero0]) kombiniert werden [PBL05]. Weitere Varianten eines SPL-Entwicklungsprozesses werden in [Gom04] und [CN01] vorgestellt.

5.3 MODELLIERUNG DER VARIABILITÄT DURCH FEATUREMODELLE

Variabilität bezieht sich in der vorliegenden Arbeit auf die zur selben Zeit existierenden unterschiedlichen Varianten eines Domänenartefakts und nicht auf dessen Versionen, die über die Zeit entstehen [PBL05, S.66]. Zur Darstellung der Variabilität einer SPL lassen sich Variabilitätsmodelle verwenden [PBL05]. Variabilitätsmodelle können zur Konsistenzerhaltung der Variabilität mittels Traceability-Links über alle Teilprozesse hinweg verwendet werden, aber auch zur Herleitung gültiger Produktkonfigurationen [PBL05, CHE04, CA05, CHE05b]. Als Variabilitätsmodelle lassen sich beispielsweise die aus der Softwareentwicklung bekannten Anwendungsfalldiagramme [BHP03, ML02] aber auch Featuremodelle [KLD02, FFB02] und Orthogonale Variabilitätsmodelle [PBL05] einsetzen, wobei die letzten zwei zueinander semantisch äquivalent sind [MHP+07]. In dieser Arbeit werden zur Darstellung der Variabilität einer SPL durchgängig Featuremodelle verwendet.

Definition 5.3 (Feature) *Ein Feature ist eine für den Kunden relevante, funktionale Eigenschaft eines Systems [KCH+90].*

Die für eine Softwaredomäne relevanten Features lassen sich durch Feature-orientierte Domänenanalyse (FODA) [KCH+90] identifizieren. Die identifizierten Features können durch Featuremodelle, die erstmalig in [KCH+90] vorgestellt wurden, zueinander in Beziehung gesetzt und hierarchisch angeordnet werden. Durch die Auswertung der in einem Featuremodell dargestellten Variabilität lassen sich alle Produktkonfigurationen herleiten.

Definition 5.4 (Produktkonfiguration) *Eine Produktkonfiguration entspricht einer Feature-Auswahl, die nicht die im Featuremodell angegebenen Abhängigkeiten (Beziehungen und Einschränkungen) zwischen den Features einer SPL verletzt.*

Das Featuremodell einer SPL stellt alle vorgesehenen Produktkonfigurationen einer SPL und somit auch die daraus entwickelbaren Produktvarianten grafisch kompakt dar. Ein Featuremodell lässt sich durch ein oder mehrere Featurediagramme beschreiben [KCH+90, S.64]. Jedes Featurediagramm besteht dabei aus einer grafisch dargestellten Baumstruktur mittels der die einzelnen Features zueinander in Beziehung gesetzt werden. Jeder Knoten im Baum entspricht dabei einem Feature. Eine Ausnahme stellt der Wurzelknoten dar, welcher nur den Konzeptnamen des Featurediagramms enthält [CA05].

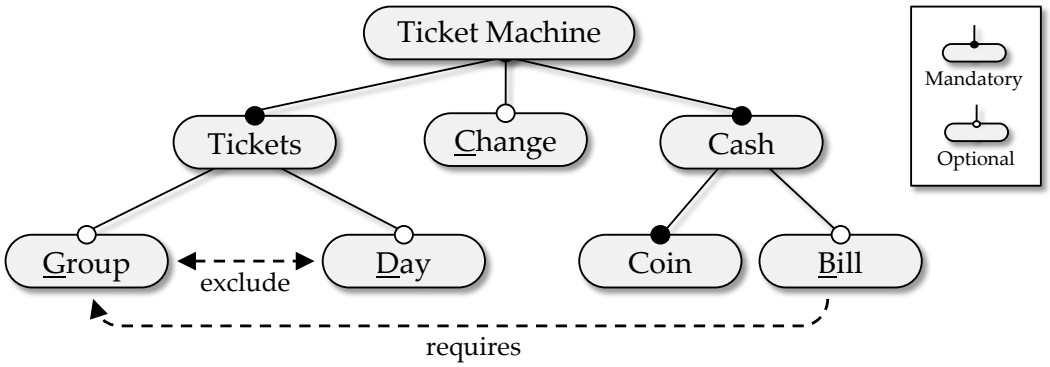


Abbildung 5.2: Featuremodell der FA-SPL

Feature* \ PC	*fa1*	*fa2*	*fa3*	*fa4*	*fa5*	*fa6*	*fa7*	*fa8*
Group	0	1	0	0	1	0	1	1
Day	0	0	1	0	0	1	0	0
Change	0	0	0	1	1	1	0	1
Bill	0	0	0	0	0	0	1	1

* 0=nicht enthalten 1=enthalten

Abbildung 5.3: Produktkonfigurationen der FA-SPL

Beispiel

Im Folgenden wird das in Abschnitt 3.4 eingeführte Fallbeispiel *Fahrkartenautomat* von einem einzelnen Softwaresystem zu einer SPL erweitert. Es wird angenommen, dass diese Fahrkartenautomaten-SPL (FA-SPL) modellbasiert getestet werden soll. In Abbildung 5.2 ist das Featuremodell der FA-SPL dargestellt. Die Funktionalität der optionalen Features lässt sich wie folgt beschreiben:

- (G)roup : Ermöglicht den Kauf von Gruppenfahrkarten
- (D)ay : Ermöglicht den Kauf von Tagesfahrkarten
- (C)hange : Wechselgeld wird ausgegeben
- (B)ill : Die Bezahlung mit Geldscheinen ist möglich

Aus dem Featuremodell lassen sich unter Berücksichtigung aller Abhängigkeiten 8 Produktkonfigurationen herleiten (siehe Abbildung 5.3). Je nachdem welche Features in einer Produktkonfiguration an- bzw. abwesend sind, hat das Einfluss auf die Funktionalität der dazu passenden Produktvariante. Beispielsweise entspricht die Produktkonfiguration *fa7* in Abbildung 5.3 dem zuvor in Abschnitt 3.4 vorgestellten Einzel-Softwaresystem.

Im Featuremodell wird jedes Feature durch einen Namen repräsentiert. In der Regel stellt der Name eines Eltern-Features einen abstrakten Oberbegriff für die Namen der diesem untergeordneten Kind-Features dar. Nicht sichtbar im Featurediagramm sind die mit einem Feature über interne Traceability-Links verbunden Domänenartefakte [Ost12], beispielsweise Modelle mit Variabilität [Gom04, Gom05, CA05, Was04, GKPR08]. Durch diese Verbindung zu anderen im Entwicklungsprozess verwendeten Modellen bekommen die im Featuremodell grafisch dargestellten Namen der Features eine tiefergehende Bedeutung [CA05]. Bei besonders komplexen Featuremodellen können auch abstrakte Features zur besseren Strukturierung eingesetzt werden. Abstrakte Features besitzen kein Mapping auf Domänenartefakte. Die Menge an möglichen Produktkonfigurationen, die sich aus dem Featuremodell herleiten lassen, lässt sich mit der FODA-Notation durch folgende Eltern-Kind-Beziehungen einschränken:

MANDATORY: Das Kind-Feature muss in der Produktkonfiguration enthalten sein, wenn das Eltern-Feature in der Produktkonfiguration enthalten ist.

OPTIONAL: Das Kind-Feature kann in der Produktkonfiguration enthalten sein, wenn das Eltern-Feature in der Produktkonfiguration enthalten ist.

ALTERNATIVE (XOR): Genau eins der Kind-Features muss enthalten sein, wenn das Eltern-Feature in der Produktkonfiguration enthalten ist.

OR: Mindestens eins der Kind-Features muss enthalten sein, wenn das Eltern-Feature in der Produktkonfiguration enthalten ist.

Zusätzlich zu den an der Baumstruktur ausgerichteten Eltern-Kind-Beziehungen können im Featuremodell auch hierarchieübergreifende Einschränkungen zwischen zwei Features angegeben werden:

F_1 REQUIRES F_2: Wenn Feature F_1 in der Produktkonfiguration enthalten ist, muss auch Feature F_2 enthalten sein.

F_1 EXCLUDES F_2: Wenn Feature F_1 in der Produktkonfiguration enthalten ist, darf Feature F_2 nicht enthalten sein (und umgekehrt).

Die abstrakte Syntax der Featuremodelle nach FODA-Notation ist in einer vereinfachten Variante des Metamodells in Abbildung 5.4 dargestellt. Eine weitere mögliche Variante des Metamodells ist in [WE09] zu finden.

Da die Abhängigkeiten zwischen den Features durch die grafische Darstellung leichter erkennbar sind, wird die Kommunikation zwischen Entwicklern und Kunden bei der Auswahl einer Produktkonfiguration vereinfacht [KCH+90, HST+08]. Das ursprüngliche FODA-Featuremodell wurde in vielen Forschungsarbeiten erweitert, um dessen Ausdrucksmächtigkeit zu erhöhen. Eine ausführliche Zusammenfassung dieser Erweiterungen ist in [CHE05b] zu finden. Einige dieser Erweiterungen ergänzen die im Featuremodell dargestellten Features um Kardinalität, Gewichtung und Wahrscheinlichkeiten sowie komplexere Aussagen über zulässige Featurekombinationen. So bezieht sich die Kardinalität eines Features darauf, wie oft dieses in einer Produktkonfiguration vorkommen

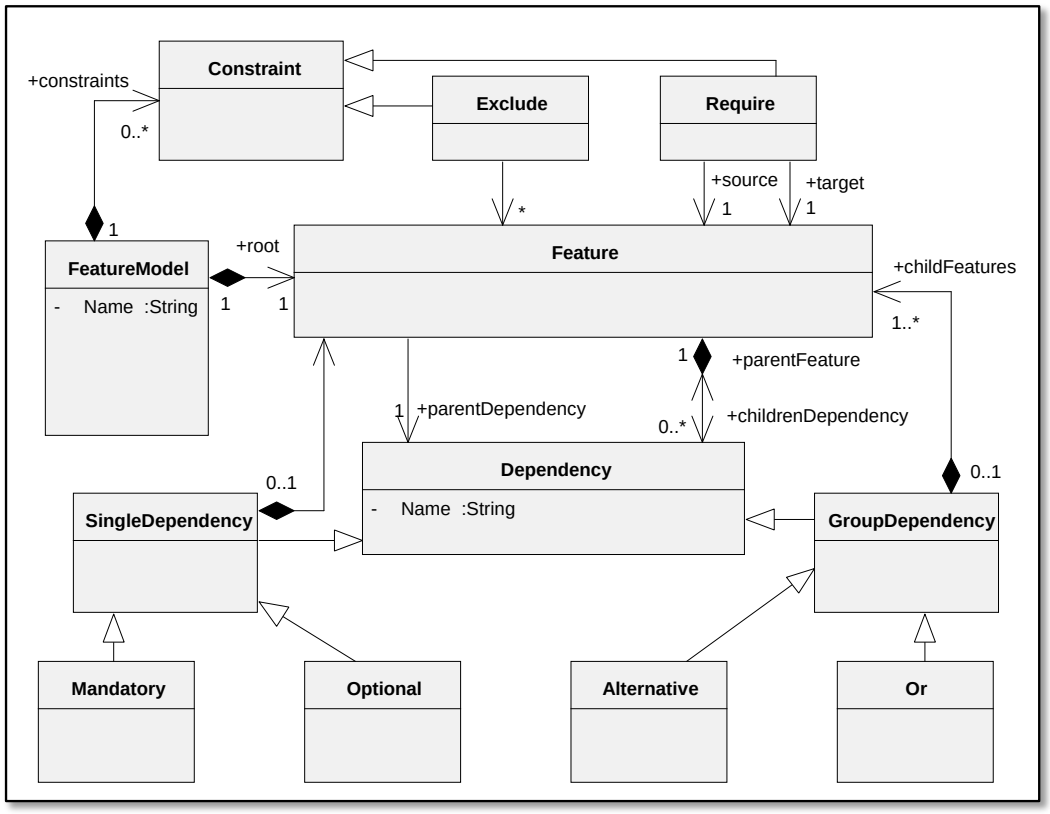


Abbildung 5.4: Vereinfachtes Metamodell für (FODA-)Featuremodelle
nach [Ost12]

kann [CHE05a, RBSP02]. In [WDS09] werden Features mit einem Wert gewichtet, um anschließend basierend auf diesen Werten Produktkonfigurationen zu erstellen, die gewissen Kostenbeschränkungen unterliegen (z.B. Ressourcenverbrauch). Auch gibt es Erweiterungen, die zusätzlich zu den harten Einschränkungen wie *exclude* oder *requires* noch weiche Einschränkungen (Fuzzy-Logik) erlauben, was durch die Angabe von Wahrscheinlichkeiten an den Features erreicht wird [CSW08]. Eine weitere Erweiterung erlaubt komplexe Aussagen für hierarchieübergreifende Einschränkungen zwischen Features (z.B.: F_1 requires F_1 or F_2) [Bat05]. Durch das Hinzufügen von mehr Informationen in Featuremodellen wird deren Auswertung komplexer. Eine Übersicht über Literatur bzgl. computergestützter Auswertung von Informationen in Featuremodellen wird in [BSRC10] gegeben.

5.4 TESTPROZESS EINER SPL

Bezogen auf das SPL-Rahmenwerk unterscheidet sich das Testen von Software-Produktlinien vom individuellen Testen einzelner Softwaresysteme hauptsächlich in den folgenden zwei Punkten:

- Zwei Testprozesse (Domänen- und Produkttestprozess)
- Umgang mit der Variabilität im Domänentest und Produkttest

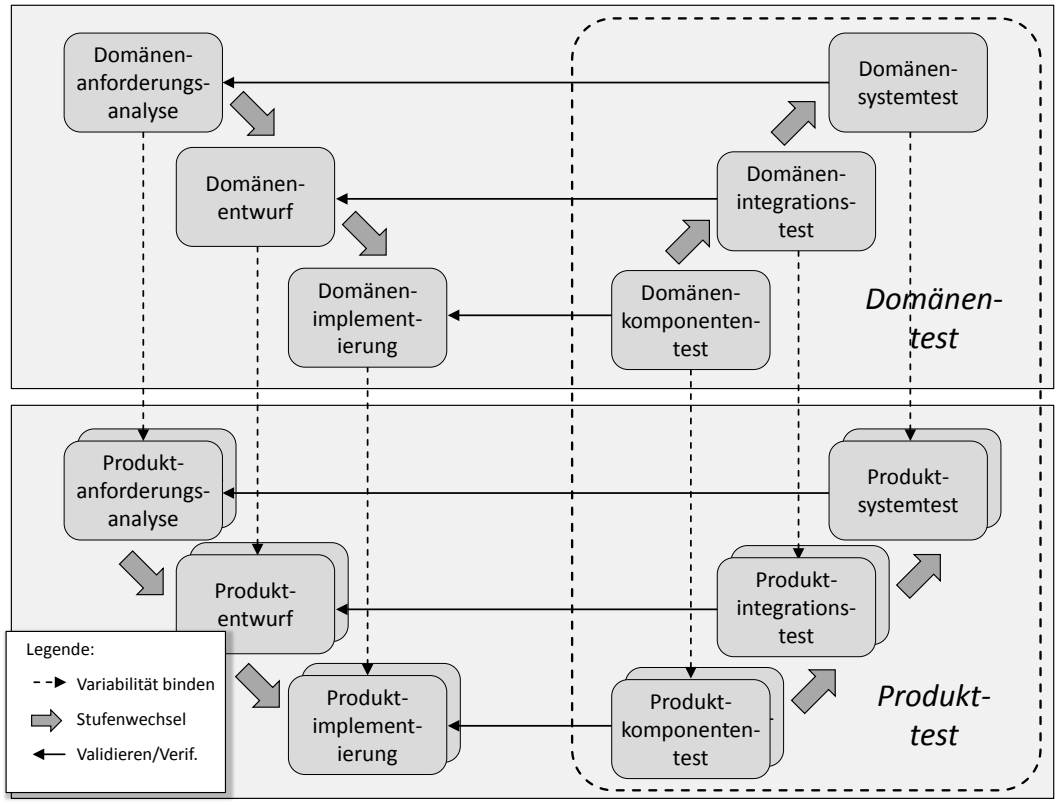


Abbildung 5.5: SPL-Testprozess
vereinfacht nach [OWES11]

Beide Testprozesse können in die Teststufen *Komponententest*, *Integrationstest* und *Systemtest* unterteilt werden, wobei dann jede Teststufe im Domänentestprozess mit ihrem Äquivalent im Produkttestprozess verzahnt ist (siehe Abbildung 5.5). Viele der existierenden SPL-Testansätze lassen sich der Teststufe *Systemtest* zuordnen [ER11].

Im Domänentest werden die während der zuvor durchlaufenen Teilprozesse (Domänenanforderungsanalyse, -entwurf und -implementierung) erstellten, variablen Domänenartefakte entweder selber getestet [PBL05, S.258] oder zur Erstellung von Testartefakten verwendet, die sich im Produkttest wiederverwenden lassen [PBL05, S.30]. Dafür werden die Testartefakte mit Variabilitätsinformationen versehen. Da im Domänentest noch keine vollständigen Produkte existieren, sondern nur einzelne oder zusammengesetzte variable Komponenten, lassen sich diese auch nur in der zum aktuellen Entwicklungsstand passenden Teststufe testen [ER11]. Aufgrund der fehlenden vollständigen Produkte im Domänentestprozess kann die Teststufe *Systemtest* nicht zum dynamischen Testen verwendet werden. Dennoch können in dieser Teststufe vorbereitende Maßnahme für den Produkt-Systemtest getroffen werden. Beispielsweise können wiederverwendbare Testartefakte (z.B. Testmodelle oder Testfälle) erstellt werden [PBL05, S.269].

Im Produkttest werden die mithilfe der SPL-Plattform erstellten Produktvarianten oder deren Komponenten getestet. Dabei werden nicht wie beim Test eines einzelnen Softwaresystems die Testartefakte für jede Produktvariante neu erstellt,

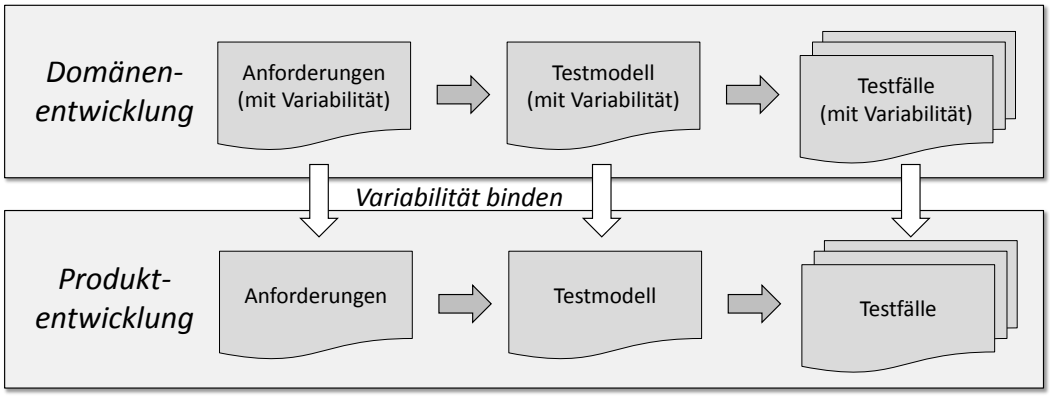


Abbildung 5.6: Modellbasiertes Testen in der SPL-Entwicklung
übersetzt aus [RKPR05]

sondern die zur Produktvariante passenden, variablen Testartefakte aus der Domänenentwicklung wiederverwendet, nachdem deren Variabilität entsprechend der gewählten Produktvariante gebunden wurde.

5.5 THEMENVERWANDTE SPL-TESTANSÄTZE

Im Folgenden werden Arbeiten aus dem SPL-Kontext vorgestellt, die themenverwandt mit dem Beitrag dieser Arbeit sind. Insbesondere wird auf Arbeiten mit dem Thema *Wiederverwendung von Testartefakten* oder *Test ausgewählter Produkte* eingegangen. Eine ausführliche Übersicht über darüber hinaus gehende Themen wie Testprozess, Testmanagement, Testbarkeit, Testautomatisierung und Teststufen (Systemtest, Integrationstest sowie Komponententest) im SPL-Kontext wird in [ER11, McG01] gegeben.

5.5.1 *Wiederverwendung von Testartefakten*

Damit die zur Verfügung stehenden Ressourcen zum Testen einer großen SPL ausreichen, existieren SPL-Testansätze, die Testartefakte im Domänentest mit Variabilität versehen, um diese im Produkttest für ähnliche Produktvarianten wiederverwenden zu können. Die im Folgenden vorgestellten Testansätze reduzieren die Kosten im SPL-Test durch die Wiederverwendung von Testartefakten.

5.5.1.1 *Modelle*

Aufgrund der guten Wartbarkeit und Maschinenlesbarkeit von formalen Modellen, eignet sich modellbasiertes Testen (siehe Kapitel 3) für den Test von Software-Produktlinien [PBL05]. Es existiert bereits eine Vielzahl von modellbasierten SPL-Testansätzen, von denen die meisten für den Systemtest ausgelegt sind [ER11]. In diesen Ansätzen werden zumeist im Domänentest variable Modelle erstellt, die zur Erstellung von Testfällen (gegebenenfalls mit Variabilität) verwendet werden. Diese Testfälle werden anschließend im Produkttest wiederverwendet (siehe Abbildung 5.6).

In [Gom04, Gom05, CA05, Was04, GKPR08] werden Modelle vorgestellt, die Variabilität besitzen. Wird diese Variabilität hinsichtlich einer Produktkonfiguration gebunden, lässt sich das entsprechende produktspezifische Modell herleiten. Meistens beinhalten diese Testmodelle bereits die gesamte zu testende Funktionalität der SPL. In [OMR10, Ost12, WSS08] werden solche Testmodell zur Generierung einer produktspezifischen Testsuite für ausgewählte Produktvarianten der SPL verwendet. In der vorliegenden Arbeit wird ebenfalls ein mit Variabilität angereichertes Testmodelle zur Generierung von Testfällen eingesetzt. Für variable Testmodelle eignen sich beispielsweise Kontrollflussdiagramme, Sequenzdiagramme und Zustandsautomaten. In der vorliegenden Arbeit wird ein Zustandsautomat als Testmodell verwendet, aus dem sich die produktspezifischen Testmodelle herleiten lassen.

In [OMR10, Ost12, LG08] werden Ansätze beschrieben, die ein variables Zustandsautomat als Testmodell verwenden. Die Variabilität wird erreicht, indem die Elemente des Testmodells mit Bedingungen über Features annotiert werden. Jedes Element, dessen Bedingungen für eine gewählte Produktkonfiguration erfüllt ist, wird bei der Konstruktion des produktspezifischen Testmodells verwendet. Die Bedingung eines jeden Elements wird dabei aus dem zur SPL gehörenden Featuremodell hergeleitet. Dadurch entsteht eine Bindung zwischen der Variabilität im Testmodell und der Variabilität im Featuremodell, die in [LG08] durch eine Mapping-Funktion realisiert wurde. In der vorliegenden Arbeit wird ebenfalls eine Mapping-Funktion zwischen Featuremodell und variablen Zustandsautomat verwendet.

In [Was04, WSS08] wird ebenfalls ein variabler Zustandsautomat als Testmodell verwendet, welcher bereits die gesamte zu testende Funktionalität aller Produktvarianten einer SPL beschreibt. Jedoch werden die produktspezifischen Testmodelle nicht extra aus dem variablen Testmodell konstruiert. Stattdessen werden die Zustandsübergänge im variablen Testmodell entsprechend der gewählten Produktkonfiguration durch Bedingungen eingeschränkt. In [WSS08] werden die Bedingungen durch OCL-Expressions ausgedrückt. Beide Arbeiten verwenden zur Herleitung der Bedingungen kein Featuremodell. In der vorliegenden Arbeit wird das variable Testmodell ebenfalls durch Bedingungen eingeschränkt.

In [Oli08, OG08, RKPR05, HVR04] werden mit Variabilität angereicherte Aktivitätsdiagramme statt Zustandsautomaten zur Generierung von Testfällen verwendet. In diesen Diagrammen werden die Aktivitäten durch Bedingungen annotiert, die aus einem Featuremodell hergeleitet werden. Die in [RKPR05, HVR04] vorgestellten Ansätze sind für den Systemtest einer Software-Produktlinie konzipiert. Wie in der vorliegenden Arbeit auch, verwenden beide Ansätze strukturelle Überdeckungskriterien als Auswahlkriterium für die generierten Testfälle. Im Gegensatz zu den in [HVR04] generierten Testfällen, weisen die in [RKPR05] generierten Testfälle, die durch Sequenzdiagramme dargestellt werden, jedoch selbst Variabilität auf.

In [LSKL12, LLSG12] werden die produktspezifischen Testmodelle nicht durch ein mit Variabilität angereichertes Testmodell hergeleitet. Stattdessen wird ein Basis-Testmodell als Ausgangspunkt verwendet, aus dem sich durch die Anwendung von Deltas produktspezifische Testmodelle herleiten lassen. Jedes Delta be-

steht dabei aus einer Modellelement-Operation (Entfernen oder Hinzufügen), die durchgeführt wird, wenn die dazugehörende Anwendungsbedingung über Features erfüllt ist.

Andere Ansätze verzichten auf das Erstellen eines Testmodelles und erstellen Testfälle direkt aus den formalen Anforderungen, die mit Variabilität angereichert wurden. So werden in [BG03, BFGL06] für den anforderungsbasierten Test textuell beschriebene Anwendungsfälle mit bestimmten Markierungen versehen, welche die Variabilität beschreiben. Soll eine Produktvariante getestet werden, wird die Variabilität gebunden und die Anwendungsfälle als Testfallspezifikation verwendet. In [NFLTJ04, NLTJ06] wird ähnlich verfahren, nur dass dort UML-Anwendungsfalldiagramme mit Variabilität verwendet werden, die nach dem Binden der Variabilität zur Generierung von produktspezifischen Testfällen verwendet werden.

Zur Erstellung von Modellen mit Variabilität sowie das Binden dieser Variabilität für den Produkttest eignet sich, wie in [OMR10, Ost12] gezeigt, das kommerzielle Varianten-Management-Werkzeug pure::variants [PS] in Verbindung mit dem Modellierungswerkzeug IBM Rational Rhapsody [IBM] und ATG [BTC].

5.5.1.2 *Testfälle*

In [KBK11] wird ein White-Box-Testansatz für die Teststufe *Komponententest* vorgestellt, der für einen existierenden Testfall die Features identifiziert, die in einer Produktvariante notwendigerweise anwesend oder abwesend sein müssen, um den Testfall darauf anwenden zu können. Dafür wird eine statische Analyse auf der Implementierung durchgeführt. Anschließend kann für einen Testfall eine Menge von Produktvarianten bestimmt werden, auf denen dieser Testfall bei seiner Ausführung dasselbe Ergebnis liefern muss. In der vorliegenden Arbeit wird eine ähnliche Analyse auf einem variablen Testmodell für die Teststufe *Systemtest* durchgeführt. Da ein Testmodell verwendet wird und nicht die Implementierung, handelt es sich bei dem in der vorliegenden Arbeit vorgestellten Ansatz um ein Black-Box-Testverfahren. Dadurch kann nicht garantiert werden, dass ein auf unterschiedlichen Produktvarianten verwendbarer Testfall dasselbe Testergebnis erzeugt. Darüber hinaus wird in [Wüb10] ein Variabilitätsmanagement-Ansatz für Testfallspezifikationen einer SPL beschrieben, mit dem sich ebenfalls Kosten im SPL-Test einsparen lassen.

5.5.1.3 *Testergebnisse*

In [ESR08, ERW10, ER10] wird die dem Regressionstest zugrundeliegende Idee auf den SPL-Kontext übertragen. Beim Regressionstest eines Einzel-Softwaresystems werden nach einer Änderung am System nur die Teilfunktionalitäten erneut getestet, die von der Änderung betroffen sind [Ger, Glossar]. Dies soll sicherstellen, dass durch die Änderung keine Fehlerzustände hinzugefügt oder freigelegt wurden. Die nicht geänderten Teilfunktionalitäten werden nicht erneut getestet, da keine Veränderung der Testergebnisse erwartet wird. Diese Vorgehensweise lässt sich auch auf die unterschiedlichen Produktvarianten einer SPL anwenden. In diesem Fall werden nur die neuen Teile in einer Produktvariante getestet, wel-

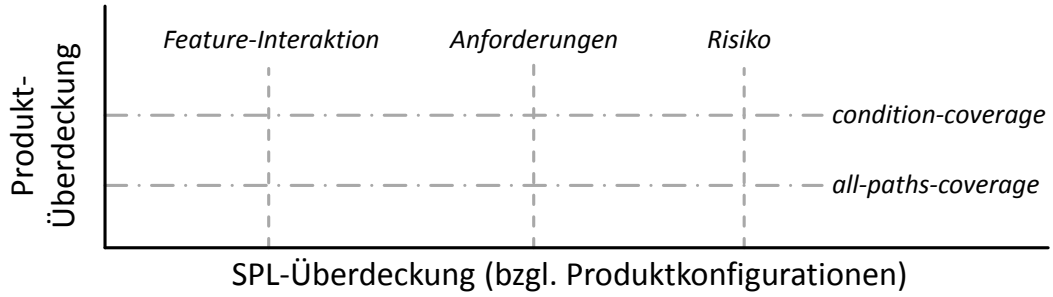


Abbildung 5.7: Überdeckung für Produktinstanzen und SPL
angelehnt an [CDS06]

che noch nicht in einer vorherigen Produktvariante getestet wurden. Folglich werden in neuen Produktvarianten die Testergebnisse von bereits zuvor getesteten Teilen wiederverwendet. Bei dieser Vorgehensweise muss sichergestellt werden, dass die nicht erneut ausgeführten Testfälle, wenn sie denn ausgeführt werden würden, die gleichen Testergebnisse erzeugen würden, da ansonsten Fehler unentdeckt blieben.

5.5.2 *Test ausgewählter Produkte*

Software-Produktlinien können aus sehr vielen Produktkonfigurationen bestehen. Folglich kann es aufgrund beschränkter Ressourcen passieren, dass selbst durch Wiederverwendung von Testartefakten nicht mehr alle Produktvarianten einzeln getestet werden können. In [McG01] wurde erstmals auf das Problem hingewiesen, Produktvarianten gezielt auswählen zu müssen, wenn zu viele Varianten in der SPL existieren, die getestet werden müssten. Vergleichbar ist dieses Vorgehen mit der Anwendung eines Auswahlkriteriums auf eine Menge potentiell erstellbarer Testfälle (siehe Abschnitt 3.5), nur dass es sich hier um eine Menge von potentiell testbaren Produktvarianten und nicht um Testfälle handelt.

Nach [CDS06] lässt sich der Begriff *Überdeckung* im SPL-Kontext nicht allein auf Überdeckungskriterien für Testfälle einzelner Produktvarianten beschränken, sondern muss auch auf die möglichen Produktvarianten einer SPL erweitert werden (siehe Abbildung 5.7). Dafür wird aus allen zu testenden Produktvarianten einer SPL eine Teilmenge identifiziert, welche bezüglich eines bestimmten Kriteriums repräsentativ für alle anderen Varianten ist. Anschließend wird die repräsentative Teilmenge stellvertretend für alle anderen Produktvarianten getestet. Die Größe der Teilmenge hängt maßgeblich von der Anzahl der in einer SPL enthaltenen Produktvarianten und dem gewählten Produkt-Auswahlkriterium ab. Auf den Test der verbleibenden Produktvarianten der SPL wird verzichtet, da davon ausgegangen wird, dass die Testergebnisse bzgl. des Kriteriums die gleichen sind. In [Ost12, OMR10, CDS07, PSK+10] wird eine zu testenden Produktmenge durch die Auswahl von Produktvarianten mit bestimmten Featurekombinationen gebildet. Durch dieses kombinatorische Vorgehen lassen sich Testfälle erstellen, die auf der gebildeten Produktmenge mindestens eine paarweise Feature-Interaction-Abdeckung erreichen. Beim Bilden dieser Produktmenge werden in [OMR10] al-

le Abhängigkeiten und Hierarchien im Featuremodell berücksichtigt. Abgesehen von solch kombinatorischen Ansätzen, lässt sich mit dem Ansatz aus [Kol03] eine zu testenden Produktmenge aus allen Produktvarianten der SPL auf der Grundlage einer Risikoanalyse auswählen. In [Sch07a] wird wiederum eine Heuristik benutzt, um eine Produktmenge zu generieren, die alle Anforderungen einer SPL abdeckt. Keiner der bisher existierenden Ansätze verwendet jedoch als Produkt-Auswahlkriterium eine Menge von Testfällen. Mit dem in der vorliegenden Arbeit vorgestellten Ansatz lassen sich Testfälle generieren, welche eine gewählte strukturelle Überdeckung auf allen produktspezifischen Testmodellen einer SPL garantieren und welche die Auswahl einer Menge von Produktvarianten vereinfachen, auf denen diese Testfälle ausführbar sind.

6

BEGRIFFSDEFINITIONEN UND BENÖTIGTE METHODEN

Im Folgenden werden Begriffe definiert und Methoden vorgestellt, die für den im anschließenden Kapitel 7 vorgestellten Ansatz entwickelt wurden und für diesen benötigt werden.

6.1 ZUSTANDSAUTOMATEN ALS TESTMODELLE

In dieser Arbeit werden Zustandsautomaten zur Darstellung von Testmodellen verwendet.

Definition 6.1 (Zustandsautomat)
Ein Zustandsautomat *ist ein Tripel* (S, s_0, T)*, wobei gilt*

- S *ist eine endliche Menge von Zuständen,*
- $s_0 \in S$ *ist der Initialzustand,*
- $T \subseteq S \times L \times S$ *ist eine Menge von beschrifteten Transitionen über einer Menge von Transitionsbeschriftungen* L*, jeweils bestehend aus* $event[guard]/action$*.*

Eine Transition $t \in T$ beschreibt durch deren Transitionsbeschriftung der Form $event[guard]/action$ [Har87] einen möglichen Übergang von einem Zustand in einen anderen. Zur Beschriftung der Transitionen kann eine Standardgrammatik (z.B. Promela [Holo3]) verwendet werden, die Aussagen und Zuweisungen bezogen auf Attribute unterstützt. Von den Details solch einer Grammatik wird im Folgenden abstrahiert. Es gilt:

- Ereignis (event) : Wenn das Ereignis (z.B. ein von der Umwelt gesendetes Signal) eintritt und die Bedingung (guard) erfüllt ist, wird ein Zustandsübergang vom Ausgangszustand in den entsprechenden Zielzustand ausgelöst.
- Bedingung (guard) : Die Bedingung stellt einen Ausdruck über Attributwerten des Automaten dar und kann verwendet werden, um Determinismus sicherzustellen (wenn mehrere Transitionen auf dasselbe Ereignis reagieren) oder um sonstige Einschränkungen zu realisieren.
- Aktion (action) : Sobald das Ereignis eingetreten und die Bedingung erfüllt ist, wird die Aktion atomar ausgeführt. Eine Aktion kann zu Wertänderungen von Attributen führen und/oder neue Ereignisse auslösen.

Im Folgenden beschreibt $TM(L)$ die Menge aller als Testmodell verwendeten *wohlgeformten* und *vollständigen* Zustandsautomaten über der Menge aller Transitionsbeschriftungen.

- *wohlgeformt*: Im Zustandsautomaten $tm \in TM(L)$ ist jeder Zustand $s \in S$ und jede Transition $t \in T$ vom Initialzustand s_0 aus erreichbar.
- *vollständig*: Ein Ereignis wird verworfen bzw. ignoriert, wenn dieses Ereignis im aktuellen Zustand des Automaten keinen Zustandsübergang auslöst.

6.2 TESTFÄLLE AUS ZUSTANDSAUTOMATEN

Ein aus einem deterministischen Zustandsautomat $tm \in TM(L)$ erstellter konkreter Testfall tc lässt sich als eindeutiger Transitionspfad darstellen. Um diesen herzuleiten, müssen die Testfall-Eingabedaten von tc auf den Zustandsautomaten tm angewendet werden. Da bei der Überführung eines konkreten Testfalls in dessen Transitionspfad die konkreten Eingabe- und Ausgabedaten verloren gehen, stellt der Transitionspfad eine Abstraktion eines konkreten Testfalls dar. Im Folgenden wird ein Testfall tc als eine endliche Sequenz von Transitionen dargestellt.

Definition 6.2 (Testfall) *Ein* Testfall *ist eine endliche Sequenz* $(t_0, t_1, \ldots, t_{k-1}) \in T^*$ *mit k Transitionen aus einem Zustandsautomaten* $tm \in TM(L)$.

Die Überführung von einem als Transitionspfad dargestellten abstrakten Testfall in einen konkreten Testfall ist nur dann zulässig, wenn

1. die Sequenz von Transitionen als zusammenhängender Transitionspfad im Zustandsautomaten tm enthalten ist und
2. die Transitionsbedingung einer jeden Transition im vorausgehenden Zustand erfüllt ist.

Folgende Relation wird angenommen:

$valid \subseteq T^* \times TM(L)$, wobei gilt $valid(tc, tm) :\Leftrightarrow$ Ein Testfall tc ist *valide* für ein Testmodell $tm \in TM(L)$, wenn dessen Transitionssequenz $(t_0, t_1, \ldots, t_{k-1})$ einer alternierenden Sequenz $s_0, t_0, s_1, t_1, \ldots, s_{k-1}, t_{k-1}, s_k$, von Zuständen und Transitionen aus tm entspricht, sodass

1. es sich bei s_0 um den Initialzustand handelt,
2. $(s_i, l_i, s_{i+1}) \in T$ für $0 \leq i \leq k-1$ gilt und
3. für jede ausgehende Transition t_i mit $0 \leq i \leq k-1$ aus dem durch die zuvor traversierten Transitionen erreichten Zustand im Zustandsraum gilt:
   - Das benötigte Ereignis ist erzeugbar.
   - Die Bedingung ist erfüllbar.
   - Die Aktion ist ausführbar.

Alle Testfälle, die für ein Testmodell tm die Relation *valid* erfüllen, werden im Folgenden durch die Menge $TC(tm) \subseteq T^*$ beschrieben. Im Folgenden gilt ein Testfall tc für eine Produktvariante $pc \in PC$ als *verwendbar* bzw. für ein produktspezifisches Testmodell tm als *valide*, genau dann wenn $valid(tc, tm)$ für einen Testfall tc und das zu pc gehörende Testmodell tm gilt.

Durch die Anwendung eines Überdeckungskriteriums C auf den Zustandsautomaten tm lassen sich *Testziele* $TG = C(tm)$ aus der Menge aller möglichen Testziele in tm, dargestellt durch $TG(tm)$, definieren. Für den im Folgenden vorgestellten Ansatz wird angenommen, dass in TG ausschließlich strukturelle Elemente bzw. Modellfragmente als Testziele enthalten sind. Damit die geforderte Überdeckung (definiert durch C) erreicht wird, muss für jedes Testziel $tg \in TG$ mindestens ein Testfall $tc \in TC(tm)$ erstellt werden, der tg abdeckt. Folgende Relation wird angenommen:

$covers \subseteq TC(tm) \times TG$, wobei gilt $covers(tc, tg) :\Leftrightarrow$ Ein für das Testmodell tm valider Testfall tc *deckt* das strukturelle Testziel tg auf dem Testmodell tm ab, wenn das durch tg vorgegebene Modellfragment bei der Ausführung des Testfalls auf tm traversiert wird.

6.3 MODELLBASIERTER SOFTWARE-PRODUKTLINIEN-TEST

Eine Software-Produktlinie besteht aus einer Menge von Produktvarianten, die alle aus derselben Plattform heraus entwickelt werden. Jede Produktvariante setzt auf derselben Basisimplementierung auf und besitzt damit dieselbe Kernfunktionalität. Jede Produktvariante erweitert diese Kernfunktionalität individuell durch eine Teilmenge von zur Konfiguration bereitgestellten Features $F = \{f_1, \ldots, f_n\}$. Welche Features aus F in einer Produktvariante enthalten sind, gibt deren *Produktkonfiguration* pc vor.

$$pc : F \to \mathbb{B}$$

Folglich weist eine Produktkonfiguration jedem Feature-Parameter einen booleschen Wert zu, welcher die Anwesenheit oder Abwesenheit des Features in der jeweiligen Produktvariante angibt. Im Allgemeinen ist aufgrund der zwischen den Features geltenden Abhängigkeiten nicht jede Featurekombination eine zulässige Produktkonfiguration. Die zulässigen Produktkonfigurationen und die Abhängigkeiten zwischen den Features lassen sich durch ein Featuremodell grafisch beschreiben, z.B. durch FODA-Featurediagramme [KCH+90]. Wird die grafische Darstellung eines Featuremodells abstrahiert, lassen sich die gültigen Produktkonfigurationen auch durch eine aussagenlogische Formel beschreiben [Bato5].

$$FM : (F \to \mathbb{B}) \to \mathbb{B}$$

Folglich handelt es sich bei einer Produktkonfiguration pc genau dann um eine *zulässige* Produktkonfiguration, wenn $FM(pc) = true$ gilt. Die Menge aller zulässigen Produktkonfigurationen einer SPL wird im Folgenden durch

$$PC = \{\ pc : F \to \mathbb{B} \mid FM(pc) = true\ \}$$

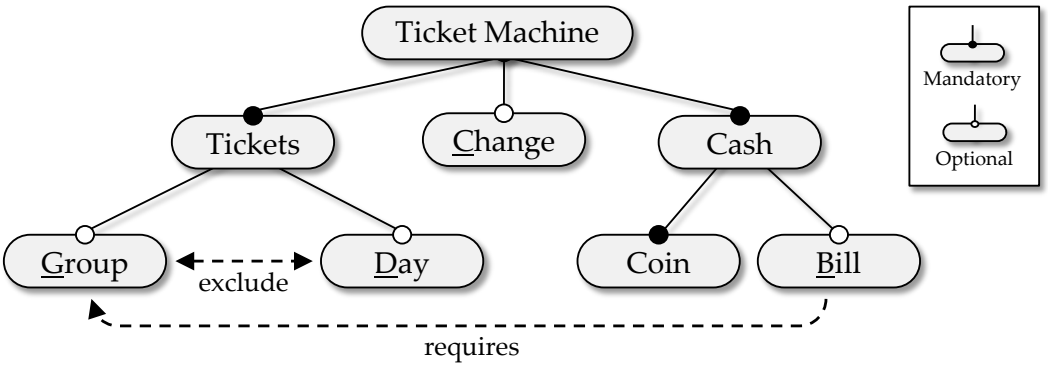


Abbildung 6.1: Featuremodell der FA-SPL
identisch zu Abbildung 5.2

Feature* \ PC	*fa1*	*fa2*	*fa3*	*fa4*	*fa5*	*fa6*	*fa7*	*fa8*
Group	0	1	0	0	1	0	1	1
Day	0	0	1	0	0	1	0	0
Change	0	0	0	1	1	1	0	1
Bill	0	0	0	0	0	0	1	1

* 0=nicht enthalten 1=enthalten

Abbildung 6.2: Produktkonfigurationen der FA-SPL
identisch zu Abbildung 5.3

beschrieben. Da jede Produktkonfiguration $pc \in PC$ eineindeutig einer Produktvariante zugeordnet werden kann, wird der Begriff *Produktkonfiguration* im Folgenden auch als Synonym für den Begriff *Produktvariante* verwendet.

Beispiel

In Abbildung 6.1 ist das Featuremodell der FA-SPL mit den Features $F_{FA} = \{G, D, C, B\}$ dargestellt. Unter Berücksichtigung der im Featuremodell dargestellten Beziehungen und Einschränkungen zwischen den Features lassen sich aus den 16 möglichen Featurekombinationen nur 8 Produktkonfigurationen für die FA-SPL herleiten. Diese 8 Produktkonfigurationen sind in Abbildung 6.2 dargestellt. Dass ein Feature in einer Produktvariante enthalten (anwesend) ist, wird durch eine 1 angezeigt, und dass es nicht enthalten (abwesend) ist, durch eine 0. Beispielsweise enthält Produktkonfigurationen *fa7* die Features *G* und *B*. Neben funktionalen Features wie *Group*, *Day*, *Change* und *Bill* existieren im Featuremodell der FA-SPL auch abstrakte Features (*Tickets*, *Cash* und *TicketMachine*), die nur zur grafischen Strukturierung genutzt werden. Da abstrakte Features in dieser Arbeit keine weitere Bedeutung besitzen, werden diese im Folgenden auch nicht in Produktkonfigurationen aufgeführt.

In den Abbildungen dieser Arbeit werden zwei Darstellungsarten für Mengen von Produktkonfigurationen verwendet. Neben der konkreten Bezeichnung (z.B.

fa8) wird auch die Darstellung als aussagenlogische Formel über Feature-Parameter (z.B. $Group \wedge \neg Day \wedge Change \wedge Bill$) verwendet. Durch eine aussagenlogische Formel ist es möglich, selbst sehr große Mengen von Produktkonfigurationen kompakt zu beschreiben. Um die aussagenlogischen Formeln kompakt darzustellen, wird im Folgenden eine Kurzform verwendet, bestehend aus den Zeichen „0“, „1“ oder „-“. Jedem Feature-Parameter wird jeweils eins dieser drei Zeichen zugewiesen. Dabei fordert „0“ die Abwesenheit und „1“ die Anwesenheit eines Features, wohingegen „-“ die Belegung offen lässt. Die durch diese Kurzform spezifizierte Produktmenge fällt umso kleiner aus, für je mehr Features die An- oder Abwesenheit gefordert wird.

Beispiel

Eine Menge bestehend aus den Produktkonfigurationen *fa2* und *fa7* (siehe Abbildung 6.2) lässt sich durch die aussagenlogische Formel „$Group \wedge \neg Day \wedge \neg Change$“ beschreiben. Da für den Parameter des Features *Bill* beide Belegungen zulässig sind, stellt „100-“ die Kurzform der aussagenlogischen Formel dar.

Die Kurzform lässt sich auch in eine aussagenlogische Formel zurück überführen, indem die Feature-Parameter, denen eine „0“ oder „1“ zugewiesen wurde, mit „$\wedge$“ logisch verknüpft werden, wobei die Feature-Parameter, denen eine „0“ zugewiesen wurde, in der aussagenlogischen Formel negiert werden.

Beispiel

Die Kurzform „01--“ lässt sich in die aussagenlogische Formel „$\neg Group \wedge Day$“ überführen. Sie beschreibt eine Menge von Produktkonfigurationen bestehend aus *fa3* und *fa6*.

6.4 150%-TESTMODELL

Um eine SPL bzw. all deren Produktvarianten modellbasiert Testen zu können, muss für jede Produktkonfiguration ein eigenes, produktspezifisches Testmodell existieren, aus dem dessen Testfälle generiert werden. Sollte nun jedes dieser produktspezifischen Testmodelle $tm_i \in TM = \{tm_1, \ldots, tm_k\} \subseteq TM(L)$ separat bzw. von Grund auf neu erstellt werden, würde das bei SPLs mit sehr vielen Produktkonfigurationen zu einem entsprechend hohen Aufwand führen.

Um den Aufwand bei der Modellerstellung sowie Modellwartung zu senken und Redundanz zu verhindern, können alle produktspezifischen Testmodelle einer SPL in einem einzigen mit Variabilität angereicherten Testmodell, im Folgenden 150%-Testmodell genannt, vereint werden. Der Begriff 150%-*Testmodell* bezieht sich auf die zusätzliche Variabilität im Modell, durch die alle produktspezifischen Testmodelle (jeweils 100%) einer SPL in diesem Modell kodiert werden. Die Variabilität im 150%-Testmodell vtm wird durch die Annotation eines jeden einzelnen Elements des zugrundeliegenden Zustandsautomaten $tm_{150} \in TM(L)$ mit einer Selektionsbedingung im Sinne einer aussagenlogischen Formel über Feature-Parameter für F erreicht. Durch die Angabe einer Produktkonfiguration $pc_i \in PC$ lassen sich die Werte der Feature-Parameter binden, was zur Folge

hat, dass sich aufgrund der Selektionsbedingungen die produktspezifischen Testmodelle tm_i durch Übernahme der entsprechenden Zustände und Transitionen herleiten lassen. Die Selektionsbedingung für ein einzelnes Element in tm_{150} ergibt sich dabei aus dem Funktional für Feature-Annotationen α.

Definition 6.3 (Funktional für Feature-Annotationen)

$$\alpha : (\mathcal{S} \cup \mathcal{T}) \to ((F \to \mathbb{B}) \to \mathbb{B})$$

Definition 6.4 (150%-Testmodell) *Ein* 150%-Testmodell *besteht aus einem Zustandsautomaten tm_{150} und einem Funktional für Feature-Annotationen α.*

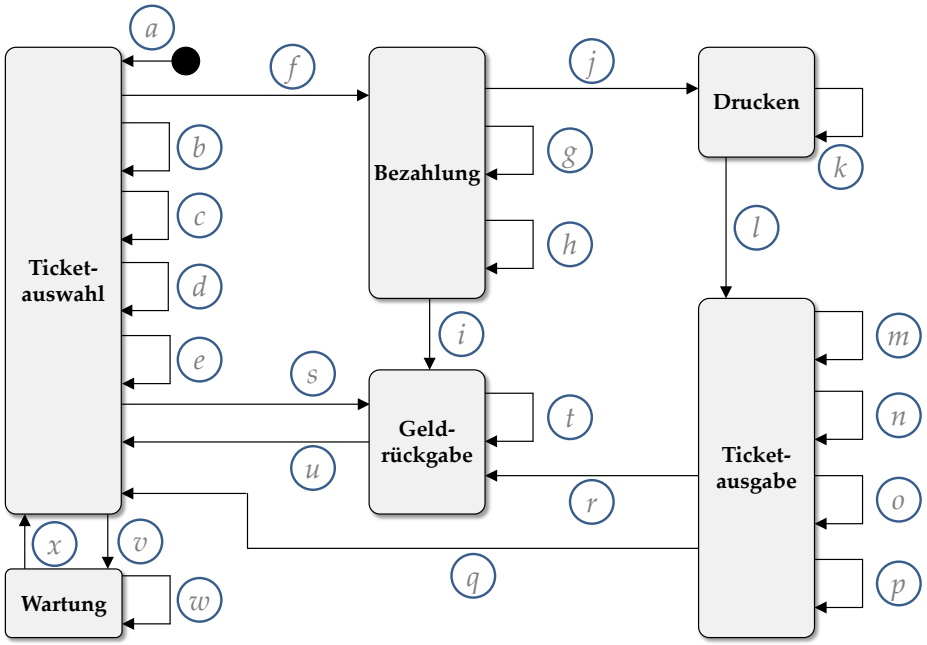
$$vtm = (tm_{150}, \alpha)$$

Beispiel

In Abbildung 6.3 ist das 150%-Testmodell dargestellt, welches für die FA-SPL verwendet wird. Dieses kann als Vereinigung der produktspezifischen Testmodelle aller 8 Produktvarianten der FA-SPL angesehen werden. In Abbildung 6.3a ist die Struktur des Automaten dargestellt, wohingegen in Abbildung 6.3b die zu den Transitionen passenden Transitionsbeschriftungen aufgelistet sind. Die Zuordnung einer Transitionsbeschriftung zu deren Transition geschieht über einen an eine Transition annotierten Bezeichner (Kleinbuchstaben). Die im 150%-Testmodell der FA-SPL dargestellte Kernfunktionalität und deren Erweiterungen lassen sich wie folgt beschreiben:

Zu Beginn befindet sich der Automat im Zustand *Ticketauswahl*. In diesem kann die benötigte Anzahl an Fahrkarten ausgewählt werden. Normale und reduzierte Fahrkarten lassen sich in jeder Produktvariante auswählen, Gruppenfahrkarten und Tagesfahrkarten hingegen nur wenn Feature *G* bzw. Feature *D* in der Produktkonfiguration enthalten ist. Anschließend folgt der Bezahlvorgang, bei dem Münzen und, wenn Feature *B* enthalten ist, auch Scheine als Zahlungsmittel akzeptiert werden. Daraufhin erfolgt im Zustand *Ticketausgabe* die Ausgabe der Fahrkarten. Ist dies geschehen findet anschließend, bei Anwesenheit des Features *C*, eine Ausgabe des Wechselgeldes statt oder, bei Abwesenheit, nicht.

Jedes produktspezifische Testmodell $tm \in TM$ besteht aus einer Teilmenge von Zuständen und Transitionen des Zustandsautomaten $tm_{150} = (\mathcal{S}, s_0, \mathcal{T}) \in TM(L)$ des 150%-Testmodells vtm, welches in der Regel selbst kein Element von TM ist.



(a) Struktur des Zustandsautomaten

Trans.	Selektions-bedingung	Beschriftung
a		/ costs=0; paid=0; print=0; tBlanks=10; tN=0; tR=0; tD=0; tG=0;
b		normal [tBlanks-tN-tR-tD-tG>0] / tN++; costs+=2;
c		reduced [tBlanks-tN-tR-tD-tG>0] / tR++; costs+=1;
d	D	day [tBlanks-tN-tR-tD-tG>0] / tD++; costs+=4;
e	G	group [tBlanks-tN-tR-tD-tG>0] / tG++; costs+=6;
f		next [costs>0]
g		coin [paid<costs] / paid++;
h	B	bill [paid<costs] / paid+=5;
i		cancel [paid<costs]
j		[paid>=costs] / print=10+tN+tR+tD+tG;
k		[print>0] / print--;
l		[print==0] / paid-=costs; costs=0; tBlanks=tBlanks-tN-tR-tD-tG;
m		[tN>0] / tN--;
n		[(tN==0) && (tR>0)] / tR--;
o	D	[(tN==0) && (tR==0) && (tD>0)] / tD--;
p	G	[(tN==0) && (tR==0) && (tD==0) && (tG>0)] / tG--;
q	!C	[(tN==0) && (tR==0) && (tD==0) && (tG==0)] / paid=0;
r	C	[(tN==0) && (tR==0) && (tD==0) && (tG==0)]
s		cancel [costs>0]
t		[paid>0] / paid--;
u		[paid==0] / tN=0; tR=0; tD=0; tG=0; costs =0;
v		maintenance
w		[tBlanks<20] / tBlanks--;
x		[tBlanks==20]

(b) Transitionsbeschriftung und Selektionsbedingung getrennt

Abbildung 6.3: 150%-Testmodell der FA-SPL

Definition 6.5 (Produktspezifisches Testmodell) *Ein* produktspezifisches Testmodell $tm_i = (S_i, s_0, T_i) \in TM$ *ergibt sich aus einem* 150%*-Testmodell* $vtm = (tm_{150}, \alpha)$ *mit* $tm_{150} = (\mathcal{S}, s_0, \mathcal{T}) \in TM(L)$ *wie folgt:*

- $S_i = \{s \in \mathcal{S} \mid (\alpha(s))(pc_i) = true\}$
- $T_i = \{t \in \mathcal{T} \mid (\alpha(t))(pc_i) = true\}$

Zur Herleitung des produktspezifischen Testmodells tm_i einer Produktkonfiguration pc_i aus einem 150%-Testmodells vtm wird die Funktion

$$bind(vtm, pc_i) = tm_i = (S_i, s_0, T_i)$$

verwendet. Um ein produktspezifisches Testmodell $tm_i = bind(vtm, pc_i)$ aus dem 150%-Testmodell $vtm = (tm_{150}, \alpha)$ herzuleiten, welches ausschließlich aus Elementen von vtm besteht, dürfen nur diejenigen Transitionen und Zustände aus tm_{150} in das produktspezifische Testmodell tm_i übernommen werden, deren Selektionsbedingung aufgrund der Produktkonfiguration pc_i erfüllt sind.

Für die Funktion *bind* wird gefordert, dass aus einem 150%-Testmodell nur produktspezifische Testmodelle aus $TM(L)$ hergeleitet werden können. Das bedeutet, dass produktspezifische Testmodell ist wohlgeformt und vollständig (siehe Abschnitt 6.1). Durch die Forderung, dass *bind* eine Bijektion ist, besitzt jede Produktkonfiguration $pc_i \in PC$ genau ein eigenes Testmodell tm_i.

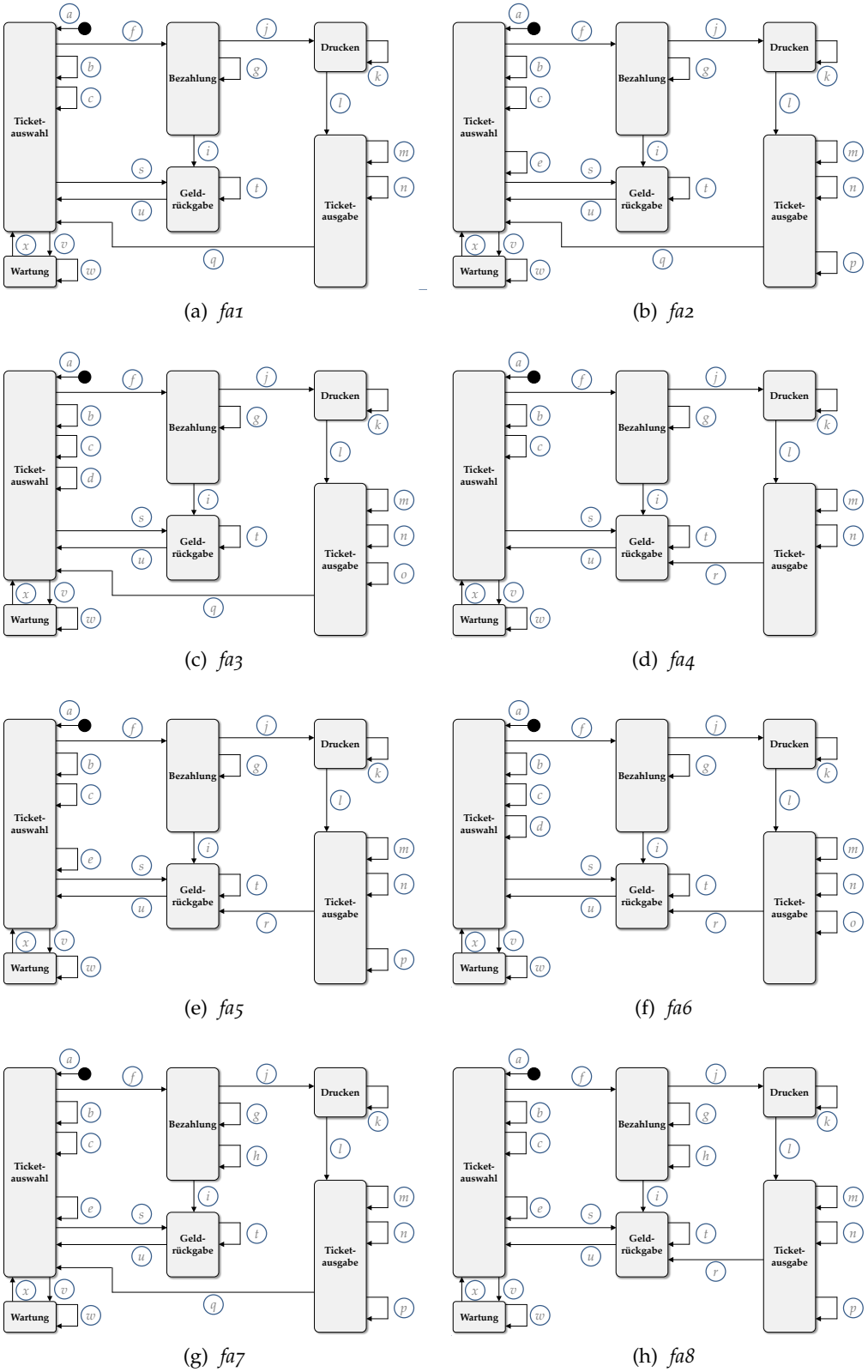
Beispiel

Die Selektionsbedingungen für die Transitionen des 150%-Testmodells der FA-SPL sind in der zweiten Spalte in Abbildung 6.3b dargestellt. In Abbildung 6.4 sind alle 8 produktspezifischen Testmodelle dargestellt, die sich aus dem 150%-Testmodell der FA-SPL herleiten lassen. Beispielsweise müssen, um das produktspezifische Testmodell der Produktvariante *fa7* (siehe Abbildung 6.4g) herzuleiten, alle Transitionen aus dem 150%-Testmodell bis auf die Transitionen *d*, *o* und *r* (da deren Selektionsbedingung für *fa7* nicht erfüllt ist) übernommen werden.

Da jedes produktspezifische Testmodell nur aus Zuständen und Transitionen des 150%-Testmodells besteht, lassen sich ebenfalls die zu überdeckenden Testziele eines jeden produktspezifischen Testmodells aus dem 150%-Testmodell herleiten. Durch die Anwendung eines Überdeckungskriteriums C, wie z.B. *all-transitions* oder *all-transition-pairs*, auf den Zustandsautomaten tm_{150} des 150%-Testmodells lassen sich Modellfragmente als Testziele $TG = \{tg_1, tg_2, \ldots, tg_l\}$ definieren. Die so gewählte Menge an Testzielen TG ist eine Obermenge bezogen auf die Testziele aller aus dem 150%-Testmodell herleitbaren produktspezifischen Testmodelle. Folglich besitzt jedes produktspezifische Testmodell $tm_i \in TM$ eine Teilmenge $TG_i \subseteq TG$ von Testzielen, die sich auf Modellfragmente beziehen, die aus dem 150%-Testmodell durch *bind* übernommen wurden.

Beispiel

Wird beispielsweise das Überdeckungskriterium *all-transitions* auf das 150%-Testmodell der FA-SPL als Auswahlkriterium für Testziele angewendet, stellt jede



(a) *fa1* (b) *fa2*

(c) *fa3* (d) *fa4*

(e) *fa5* (f) *fa6*

(g) *fa7* (h) *fa8*

Abbildung 6.4: Hergeleitete produktspezifische Testmodelle

PC	Features				Testziele (Transitionen)																					
	B	C	D	G	*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*	*l*	*m*	*n*	*o*	*p*	*q*	*r*	*s*	*t*	*u*	*v*	*w*	*x*
fa1	0	0	0	0	✓	✓	✓			✓	✓		✓	✓	✓	✓	✓	✓			✓		✓	✓	✓	✓	✓	✓
fa2	0	0	0	1	✓	✓	✓		✓	✓	✓		✓	✓	✓	✓	✓	✓		✓	✓		✓	✓	✓	✓	✓	✓
fa3	0	0	1	0	✓	✓	✓	✓		✓	✓		✓	✓	✓	✓	✓	✓	✓		✓		✓	✓	✓	✓	✓	✓
fa4	0	1	0	0	✓	✓	✓			✓	✓		✓	✓	✓	✓	✓	✓				✓	✓	✓	✓	✓	✓	✓
fa5	0	1	0	1	✓	✓	✓		✓	✓	✓		✓	✓	✓	✓	✓	✓		✓		✓	✓	✓	✓	✓	✓	✓
fa6	0	1	1	0	✓	✓	✓	✓		✓	✓		✓	✓	✓	✓	✓	✓	✓			✓	✓	✓	✓	✓	✓	✓
fa7	1	0	0	1	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓		✓	✓	✓	✓	✓	✓
fa8	1	1	0	1	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓		✓	✓	✓	✓	✓	✓	✓

Abbildung 6.5: Testziele der produktspezifischen Testmodelle

Testziel	Transitionspfad
r	*a c f g j k k k k k k k k k k k l n q c f g j k k k k k k k k k k k l n r*

Abbildung 6.6: Aus ungebundenem 150%-Testmodell erstellter Testfall

Transition ein Testziel dar, und es lassen sich 24 Testziele für TG herleiten (siehe Abbildung 6.5). Ebenfalls lässt sich erkennen, dass die Testziele TG_i eines jeden produktspezifischen Testmodells der FA-SPL eine Teilmenge von TG darstellen. In Abbildung A.1 sind die Testziele für das Überdeckungskriterium *all-transition-pairs* dargestellt.

Bevor die Testfallgenerierung basierend auf einem 150%-Testmodell beginnt, muss sichergestellt werden, dass die Variabilität des 150%-Testmodells entsprechend der Funktion *bind* für eine Produktkonfiguration $pc \in PC$ gebunden wird. Ansonsten kann ein Testfall $tc \in TC(tm_{150})$ generiert werden, der für kein produktspezifisches Testmodell $tm \in TM$ valide ist. In Abschnitt 9.2 werden zwei mögliche Varianten im Detail vorgestellt, wie sich das Binden der Variabilität in einem 150%-Testmodell realisieren lässt, um ausschließlich valide Testfälle zu generieren.

Beispiel

In Abbildung 6.6 ist ein Testfall dargestellt, der aus dem ungebundenen 150%-Testmodell der FA-SPL zum Abdecken der Transition r erstellt wurde. Da in einem ungebundenen 150%-Testmodell keine Belegung für die Feature-Parameter vorliegt, welche sich auf die Selektionsbedingungen der einzelnen Modellelemente auswirken und somit nur der Zustandsautomat tm_{150}, aber nicht das Funktional für Feature-Annotationen α des 150%-Testmodell $vtm = (tm_{150}, \alpha)$ verwendet wird, existiert Nicht-Determinismus im Zustand *Ticketausgabe*. Dadurch kann der Testfall sowohl die Transition q als auch die Transition r traversieren, obwohl sich diese eigentlich aufgrund ihrer Selektionsbedingungen ausschließen. So verlangt die Selektionsbedingung der Transition q die Abwesenheit des Features C, wohingegen die Transition r die Anwesenheit dieses Features benötigt. Folglich ist der erstellte Testfall auf keinem produktspezifischen Testmodell der FA-SPL valide.

Generiert für …	Features				Signal-Eingabesequenz	Transitionspfad
	B	C	D	G		
fa3	0	0	1	1	reduced, next, coin	*a c f g j k k k k k k k k k k k l n q*
fa6	0	1	1	0	reduced, next, coin	*a c f g j k k k k k k k k k k k l n r*

(a) Hergeleitet für konkrete Produktkonfiguration

Generiert für …	Relevante. Features				Signal-Eingabesequenz	Transitionspfad
	B	C	D	G		
fa3	-	0	-	-	reduced, next, coin	*a c f g j k k k k k k k k k k k l n q*
fa6	-	1	-	-	reduced, next, coin	*a c f g j k k k k k k k k k k k l n r*

(b) Verwendbar für mehrere Produktkonfigurationen

Abbildung 6.7: Zwei Testfälle aus dem 150%-Testmodell der FA-SPL

Da der Zustandsautomat tm_{150} eines 150%-Testmodells ohne Betrachtung des Funktionals α Nicht-Determinismus aufweisen kann, muss zwecks eindeutiger Überführung von einem konkreten Testfall in dessen Transitionspfad dasselbe gebundene 150%-Testmodell verwendet werden, wie bei der Erstellung des Testfalls.

Beispiel

In Abbildung 6.7a sind zwei aus dem 150%-Testmodell der FA-SPL hergeleitete Testfälle dargestellt. Beide Testfälle besitzen dieselbe Signal-Eingabesequenz, wurden aber aus einem unterschiedlich gebundenen 150%-Testmodell hergeleitet. Damit für beide Testfälle der korrekte Transitionspfad hergeleitet werden kann, muss die bei der Testfallerstellung verwendete Produktkonfiguration berücksichtigt werden, da ansonsten die Überführung nicht eindeutig ist (siehe Transitionspfade in Abbildung 6.7a). Wenn beispielsweise bei der Testfallerstellung die Produktkonfiguration *fa3* verwendet wurde, muss der Transitionspfad mit der Transition *q* enden.

6.5 VARIANTENÜBERGREIFENDE VERWENDUNG VON TESTFÄLLEN

Beim Testen eines einzelnen, separat entwickelten Softwareprodukts ist ein dafür erstellter Testfall auch nur für dieses Produkt vorgesehen. Im SPL-Test hingegen können Testfälle existieren, die variantenübergreifend verwendet werden können. Dadurch lassen sich die Kosten für die Erstellung, Ausführung und Wartung der Testfälle im SPL-Testprozess senken. Die Erstellung von solchen Testfällen, die häufig für mehr als eine Produktvariante verwendbar bzw. für mehr als ein produktspezifisches Testmodell valide sind, ist mittels eines 150%-Testmodells möglich. Dies ist auf den gemeinsamen Ursprung der produktspezifischen Testmodelle bei der Verwendung eines 150%-Testmodells zu deren Herleitung zurückzuführen. Wenn mehrere dieser produktspezifischen Testmodelle in Teilen identisch sind, ist es möglich, dass der Transitionspfad eines aus dem 150%-Testmodell erstellten Testfalls für all diese produktspezifischen Testmodelle valide ist. Um für einen aus einem gebundenen 150%-Testmodell generierten Testfall *tc* alle Produktkonfigurationen zu ermitteln, für welche dieser Testfall valide ist, ohne jedes produktspezifische Testmodell einzeln zu überprüfen, muss die Relevanz eines jeden Features (erforderliche An- oder Abwesenheit) für den Testfall *tc* er-

mittelt werden. Anschließend kann daraus die Menge an Produktkonfiguration $PC_{tc} \subseteq PC$ berechnet werden, für welche Testfall tc valide ist. Ein Feature ist für einen Testfall tc relevant, wenn die Belegung (An- oder Abwesend) des Feature-Parameters das Resultat einer einzigen Selektionsbedingung (zugewiesen durch α des 150%-Testmodells) der im Transitionspfad des Testfalls tc enthaltenen Transitionen $(t_1, t_2, \ldots, t_k) \in \mathcal{T}^*$ beeinflusst. Um für einen Testfall zu ermitteln, welche Features relevant sind, wird dessen Transitionspfad analysiert. Dafür werden die Selektionsbedingungen der vom Testfall traversierten Transitionen konjugiert, sodass eine aussagenlogische Formel entsteht. Um zu errechnen, für welche Produktvarianten $PC_{tc} \subseteq PC$ der Testfall tc verwendbar ist, kann das Funktional φ verwendet werden.

$$\varphi : \mathcal{T}^* \to ((F \to \mathbb{B}) \to \mathbb{B})$$

$$\varphi(tc) = \bigwedge_{\forall t \in tc} \alpha(t)$$

$$PC_{tc} := \{pc \in PC | (\varphi(tc))(pc) = true\}$$

Durch die Analyse eines neu erstellten Testfalls tc müssen für die Produktkonfigurationen PC_{tc} keine weiteren Testfälle erstellt werden, um die von tc abgedeckten Testziele auf den produktspezifischen Testmodellen diesen Produktkonfigurationen abzudecken.

Beispiel

Werden die beiden in Abbildung 6.7a dargestellten Testfälle analysiert, kann aus den Selektionsbedingungen der traversierten Transition entnommen werden, dass für beide Testfälle nur die An- bzw. Abwesenheit des Features *C* relevant ist. So lässt sich der obere Testfall, der ursprünglich aus dem produktspezifischen Testmodell der Produktkonfiguration *fa3* generiert wurde, für alle Produktkonfigurationen verwenden, in denen das Feature *C* abwesend ist (siehe Abbildung 6.7b). Die drei anderen Features sind für den Testfall nicht relevant. Der untere Testfall hingegen, der ursprünglich aus dem produktspezifischen Testmodell der Produktkonfiguration *fa6* generiert wurde, ist für alle Produktkonfigurationen verwendbar, in denen das Feature *C* anwesend ist.

7

GENERIERUNG VON VOLLSTÄNDIGEN SPL-TESTSUITEN

Bei einem einzigen zu testenden Softwareprodukt ist der Aufwand zur modellbasierten Generierung von Testfällen, die eine vollständige Überdeckung des produktspezifischen Testmodells erreichen, noch überschaubar. Sollen aber für alle Produktvarianten einer großen SPL produktspezifische Testsuiten, die eine vollständige Überdeckung auf ihrem produktspezifischen Testmodell erreichen, modellbasiert generiert werden, führt das bei einer ineffizienten Vorgehensweise schnell zu nicht mehr vertretbaren Kosten. Aus diesem Grund wird in diesem Kapitel ein neuer modellbasierter Testfallgenerierungsansatz präsentiert, mit dem sich eine Testsuite generieren lässt, welche für das produktspezifische Testmodell einer jeden Produktvariante der SPL eine Teilmenge an Testfällen enthält, die auf dem jeweiligen produktspezifischen Testmodell eine vollständige Abdeckung erreichen. Zu diesem Zweck wird ein 150%-Testmodell verwendet, mit dem es möglich ist variantenübergreifend verwendbare Testfälle zu generieren. Durch solche Testfälle ist es möglich die zuvor beschriebene Testsuite zu erstellen ohne für jede Produktvariante separat Testfälle (gegebenenfalls sogar identische) generieren zu müssen. Dadurch müssen weniger Testfälle generiert werden, was zur Folge hat, dass die generierte Testsuite in der Regel wesentlich kleiner ausfällt.

Darüber hinaus lässt sich die Information, auf welcher Produktvariante ein Testfall verwendbar ist, in einem anschließenden Schritt nutzen, um eine kleine Menge von Produktvarianten in der SPL zu identifizieren, welche repräsentativ dafür ist, dass jeder in der Testsuite enthaltene Testfall mindestens auf einer dieser Produktvarianten verwendbar ist.

7.1 VOLLSTÄNDIGE SPL-TESTSUITE

Im SPL-Test sollen meistens mehrere, wenn möglich sogar alle, Produktvarianten einer SPL getestet werden. Die dafür benötigten Testfälle lassen sich zu einer Menge zusammenfassen, die im Folgenden als SPL-Testsuite bezeichnet wird. Eine SPL-Testsuite TS_{SPL} gilt als *vollständig* bezogen auf das Überdeckungskriterium C, wenn für jede Produktkonfiguration $pc_i \in PC$ eine produktspezifische Testsuite $TS_{SPL,i} \subseteq TS_{SPL}$ herleitbar ist, die auf dem Zustandsautomaten tm_i des produktspezifischen Testmodells von pc_i eine vollständige Abdeckung aller der darauf durch C definierten Testziele $TG_i \subseteq TG$ erreicht.

Eine einfache aber ineffiziente Vorgehensweise eine *vollständige SPL-Testsuite* zu erstellen besteht darin, zuerst für jedes produktspezifische Testmodell einer SPL eine produktspezifische Testsuite zu erstellen, die eine vollständige Überdeckung aller Testziele des jeweiligen Testmodells erreicht. Anschließend wer-

den alle in den produktspezifischen Testsuiten enthaltenen Testfälle zu einer einzigen Testsuite zusammengefasst. Diese Testsuite erfüllt dann die Kriterien einer vollständigen SPL-Testsuite, da diese für jedes produktspezifische Testmodell Testfälle besitzt, die eine vollständige Abdeckung auf diesem Testmodell erreichen. Allerdings entspricht diese Vorgehensweise einem Product-By-Product-Ansatz [TTK04] und ist somit für Software-Produktlinien mit sehr vielen Produktvarianten wegen genannter Ineffizienz nicht geeignet.

Im Folgenden wird ein effizienter Algorithmus zur Generierung einer vollständigen SPL-Testsuite vorgestellt, der nur im schlechtesten Fall jede Produktkonfiguration einzeln betrachten muss. Zu diesem Zweck wird, statt vieler individuell erstellter produktspezifischer Testmodelle, ein einzelnes 150%-Testmodell verwendet. Durch dieses 150%-Testmodell können Testfälle erstellt werden, die im Sinne der in Abschnitt 6.2 definierten Relation *valid* variantenübergreifend verwendbar sind. Der Ansatz macht sich zunutze, dass kein weiterer Testfall für eine Produktvariante erstellt werden muss, wenn bereits zuvor (für andere Produktvarianten) alle zum Erreichen der produktspezifischen Überdeckung benötigten Testfälle, die auf der aktuellen Variante wiederverwendbar sind, generiert wurden. Eine aus einem 150%-Testmodell generierte *vollständige SPL-Testsuite* lässt sich wie folgt definieren:

Definition 7.1 (Vollständige SPL-Testsuite) *Eine SPL-Testsuite* $TS_{SPL} = \{tc_1, tc_2, \ldots, tc_k\}$ *ist* vollständig *für alle Produktkonfigurationen* PC *einer SPL bzgl. des Überdeckungskriteriums* C*, welches die Testziele* $TG \subseteq TG(tm_{150})$ *auf dem* 150%*-Testmodell* $vtm = (tm_{150}, \alpha)$ *definiert, genau dann wenn*

$$\forall tg \in TG, \forall pc \in PC :$$
$$(\exists tc' \in TC(tm_{150}) : valid(tc', bind(vtm, pc)) \wedge covers(tc', tg))$$
$$\Rightarrow$$
$$(\exists tc \in TS_{SPL} : valid(tc, bind(vtm, pc)) \wedge covers(tc, tg))$$

Folglich gilt eine aus einem 150%-Testmodell hergeleitete SPL-Testsuite als *vollständig*, wenn diese für jedes erfüllbare Testziel tg im produktspezifischen Testmodell einer jeden Produktkonfiguration $pc \in PC$ mindestens einen validen Testfall $tc \in TS_{SPL}$ enthält, der tg abdeckt.

7.2 ALGORITHMUS GENERATESPLTESTSUITE

Im Folgenden wird die Funktionsweise von Algorithmus 7.1 beschrieben und gezeigt, dass dieser bei einer vollständigen Ausführung unter Eingabe

- eines Featuremodells FM,
- eines 150%-Testmodells vtm und
- eines Überdeckungskriteriums C

Algorithmus 7.1 Generierung einer vollständigen SPL-Testsuite

```
1:  function generateSPLTestSuite(FM,vtm,C)
2:      PC ← { pc : F → 𝔹 | FM(pc) = true }
3:      TG ← C(tm_150)
4:      TS_SPL ← ∅
5:      for all tg ∈ TG do
6:          PC_left ← PC
7:          while PC_left ≠ ∅ do
8:              select PC_sel ⊆ PC_left
9:              tc ← generate(tg, PC_sel, vtm)
10:             if ∃pc_bound ∈ PC_sel : valid(tc, bind(vtm, pc_bound)) then
11:                 TS_SPL ← TS_SPL ∪ {tc}
12:                 PC_tc ← {pc ∈ PC | (φ(tc))(pc) = true}
13:                 PC_left ← PC_left \ PC_tc
14:             else
15:                 PC_left ← PC_left \ PC_sel
16:             end if
17:         end while
18:     end for
19:     return TS_SPL
20: end function
```

eine vollständige SPL-Testsuite TS_{SPL}, wie in Definition 7.1 beschrieben, generiert. Dafür muss sichergestellt werden, dass der Algorithmus für jedes Testziel tg im produktspezifischen Testmodell einer jeden $pc \in PC$ der SPL zumindest einen Testfall $tc \in TS_{SPL}$ generiert, sodass $valid(tc, pc)$ und $covers(tc, tg)$ gilt, wenn denn so ein Testfall erstellt werden kann.

Um externe Einflüsse auszuschließen, wird angenommen, dass die im Algorithmus verwendete Funktion *generate* (Zeile 9) basierend auf dem 150%-Testmodell immer einen Testfall tc für eine Produktkonfiguration $pc_i \in PC$ und ein erfüllbares Testziel $tg \in TG_i$ generiert.

Zu Beginn wird in Zeile 2 die endliche Menge an zulässigen Produktkonfigurationen PC aus dem Featuremodell der SPL ermittelt, für deren produktspezifischen Testmodelle eine vollständige Überdeckung erreicht werden soll. In Zeile 3 werden die Testziele TG aus der Struktur des 150%-Testmodells hergeleitet. Anschließend wird mit der Erstellung der Testfälle für die SPL-Testsuite TS_{SPL} begonnen.

Die äußere Schleife (Zeile 5 bis 18) iteriert über alle der durch C auf dem 150%-Testmodell definierten Testziele $tg \in TG$. Da die Testziele TG_i eines jeden produktspezifischen Testmodells tm_i, welches aus einem 150%-Testmodell hergeleitet wurde, immer eine Teilmenge der Testziele des 150%-Testmodells darstellen (siehe Abschnitt 6.4), berücksichtigt der Algorithmus alle Testziele in jedem produktspezifischen Testmodell. Da TG eine endliche Menge ist, ist sichergestellt, dass die äußere Schleife endet.

Innerhalb der äußeren Schleife beschreibt PC_{left} für das aktuelle Testziel tg alle Produktkonfigurationen, für welche in der aktuellen Iteration der äußeren Schlei-

fe noch nicht versucht wurde einen Testfall zu generieren, mit dem sich Testziel tg in deren produktspezifischen Testmodellen abdecken lässt. Diese Menge an zu bearbeitenden Produktkonfigurationen entspricht zu Beginn PC (Zeile 6), wird dann aber sukzessive durch die innere Schleife (Zeile 7 bis 17) verringert, bis alle Produktkonfigurationen bearbeitet wurden. In dieser Schleife versucht der von der Funktion *generate* aufgerufene Testfallgenerator für eine Produktkonfiguration $pc_{bound} \in PC_{sel} \subseteq PC_{left}$ einen Testfall tc für das Testziel tg zu generieren (Zeile 9). Die Menge $PC_{sel} \subseteq PC_{left}$ besteht aus Produktkonfigurationen, die eine besondere Relevanz für den Testprozess besitzen und deshalb den anderen Produktkonfigurationen $PC_{left} \setminus PC_{sel}$ vorgezogen werden sollen (siehe Abschnitt 7.5.5). Nach dem Aufruf der Funktion *generate* (Zeile 9) tritt einer der folgenden zwei Fälle ein:

- Wenn kein Testfall tc generiert werden kann, dann existiert für kein produktspezifisches Testmodell der in PC_{sel} enthaltenen Produktkonfigurationen ein Testfall, welcher das Testziel tg abdeckt. Somit wären alle die in PC_{sel} enthaltenen Produktkonfigurationen in nur einem einzigen Schritt bearbeitet worden und bräuchten nicht noch einmal betrachtet werden (Zeile 15).
- Wenn ein Testfall tc für eine beliebige Produktkonfiguration $pc_{bound} \in PC_{sel}$ generiert werden kann, wird tc in die Testsuite aufgenommen (Zeile 11). Anschließend wird Testfall tc mittels der in Abschnitt 6.5 vorgestellten Technik daraufhin analysiert, für welche weiteren produktspezifischen Testmodelle Testfall tc valide ist. Die zu diesen produktspezifischen Testmodellen gehörende Menge an Produktkonfigurationen wird durch PC_{tc} beschrieben (Zeile 12). Da der Testfall aus dem produktspezifischen Testmodell der Produktkonfiguration $pc_{bound} \in PC_{sel}$ erstellt wurde, muss $PC_{tc} \cap PC_{sel}$ mindestens pc_{bound} beinhalten. Und da $PC_{sel} \subseteq PC_{left}$ gilt, besteht auch $PC_{tc} \cap PC_{left}$ aus mindestens einer Produktkonfiguration, womit mindestens eine Produktkonfiguration aus PC_{left} in dieser Iteration der inneren Schleife bearbeitet wurde. Die bearbeiteten Produktkonfigurationen können anschließend aus PC_{left} entfernt werden (Zeile 13). Da PC_{tc} noch andere Produktkonfigurationen enthalten kann, als die in PC_{sel} enthalten sind, ist es möglich, dass in einer Iteration mehr als die durch PC_{sel} beschriebenen Produktkonfigurationen bearbeitet werden.

In beiden Fällen wird die noch zu bearbeitende Produktkonfigurationsmenge PC_{left} in jedem Durchlauf der inneren Schleife (Zeile 7 bis 17) weiter reduziert und endet im schlechtesten Fall nach spätestens so vielen Durchläufen, wie Produktkonfigurationen in der SPL existieren. Somit ist garantiert, dass die innere Schleife von Algorithmus 7.1 nach endlich vielen Durchläufen endet und dabei sicherstellt, dass in der SPL-Testsuite TS_{SPL} für jede Produktkonfiguration, ein valider Testfall, der Testziel tg abdeckt, enthalten ist, wenn so ein Testfall erstellt werden kann. Da die äußere Schleife die innere Schleife für jedes Testziel $tg \in TG$ aufruft, entsteht schließlich eine vollständige SPL-Testsuite TS_{SPL}, die für jedes erfüllbare Testziel eines jeden produktspezifischen Testmodells einen Testfall enthält, welcher dieses Testziel auf diesem Testmodell abdeckt (Zeile 19).

Proposition 7.1 *Eine mit Algorithmus 7.1 generierte SPL-Testsuite TS_{SPL} ist* vollständig *für alle PC einer SPL bezüglich des Überdeckungskriteriums C.*

7.3 BEISPIEL

Im Folgenden soll die Funktionsweise von Algorithmus 7.1 am Fallbeispiel *Fahrkartenautomat* verdeutlicht werden. So soll eine *vollständige SPL-Testsuite* für die FA-SPL zum Erreichen einer vollständigen *all-transitions*-Überdeckung generiert werden. Um variantenübergreifend verwendbare Testfälle generieren zu können, wird das 150%-Testmodell der FA-SPL verwendet (siehe Abbildung 6.3).

Noch bevor die eigentliche Testfallgenerierung beginnt, werden die Abhängigkeiten zwischen den Features *B*, *C*, *D* und *G* der FA-SPL ausgewertet (Zeile 2), um die Produktkonfigurationen *PC* zu berechnen, für welche Testfälle erstellt werden können (siehe Abbildung 6.1). Von den 16 aus dem Featuremodell der FA-SPL ableitbaren Featurekombinationen sind aufgrund von Abhängigkeiten nur 8 als Produktkonfigurationen für *PC* zulässig. Aus dem 150%-Testmodell der FA-SPL werden in Zeile 3 alle 24 Testziele $TG = \{a, b, ..., x\}$ für TG hergeleitet (siehe Abbildung 6.3). Da die äußere Schleife über all diese Testziele iteriert und immer dieselben Schritte wiederholt ohne Bezug auf vorherige Iterationen zu nehmen, ist es ausreichend, sich auf ein einzelnes Testziel aus dem 150%-Testmodell der FA-SPL zu fokussieren, um die Funktionsweise der äußeren Schleife zu beschreiben. Im Folgenden wird die Testfallgenerierung für das Testziel *m* detailliert beschrieben. Es wird angenommen, dass bereits die alphabetisch vor *m* liegenden Testziele $(a, b, ..., l)$ von der äußeren Schleife abgearbeitet und dafür insgesamt 12 Testfälle $(tc1, ..., tc12)$ erstellt wurden, die in der SPL-Testsuite TS_{SPL} enthalten sind.

Bevor die innere Schleife (Zeile 7 bis 17) betreten wird, wird in Zeile 6 die Menge der zu bearbeitenden Produktkonfigurationen PC_{left} der Menge *PC* gleichgesetzt. Nachdem in Zeile 7 festgestellt wurde, dass PC_{left} noch Produktkonfigurationen enthält und somit die Abbruchbedingung der inneren Schleife nicht erfüllt ist, beginnt die erste Iteration der inneren Schleife (vergleiche mit Abbildung 7.1a). Da alle 8 Produktvarianten in PC_{left} das Testziel *m* in ihrem produktspezifischen Testmodell enthalten (siehe Abbildung 6.5), kann die innere Schleife erst dann verlassen werden, wenn für jede der 8 Produktkonfigurationen ein Testfall in TS_{SPL} enthalten ist, welcher auf dieser verwendbar ist und Testziel *m* abdeckt. Bevor in Zeile 9 die Funktion *generate* aufgerufen wird, wird in Zeile 8 PC_{sel} gleichgesetzt zu der verbleibende Menge zu bearbeitender Produktkonfigurationen PC_{left} (vergleiche mit Abbildung 7.1b). Daraufhin sucht der Testfallgenerator für eine geeignete Produktkonfiguration $pc_{bound} \in PC_{sel}$ aus der übergebenen Menge PC_{sel} nach einem konkreten Testfall, der im entsprechenden produktspezifischen Testmodell $bind(vtm, pc_{bound})$ das Testziel *m* abdeckt. Für die Produktkonfiguration der Produktvariante *fa7* kann ein valider Testfall $tc13$ gefunden werden, welcher das Testziel *m* abdeckt (vergleiche mit Abbildung 7.1c). In Zeile 11 wird Testfall $tc13$ der SPL-Testsuite TS_{SPL} hinzugefügt (vergleiche mit Abbildung 7.1d). In Zeile 12 wird für Testfall $tc13$ die Menge der Produktkonfigurationen PC_{tc} bestimmt, für die sich dieser verwenden lässt (vergleiche mit Abbildung 7.1e). Diese Menge entspricht den Produktvarianten fa_7 und fa_8. In Zeile 13 werden schließlich die Produktkonfigurationen PC_{tc} aus der Menge der noch zu bearbeitenden Produktkonfigurationen PC_{left} entfernt, da nun ein Testfall in der TS_{SPL} enthalten ist, welcher Testziel *m* auf den zwei Produktkonfigurationen *fa7* und *fa8* abdeckt

(vergleiche mit Abbildung 7.1f). Anschließend ist die erste Iteration der inneren Schleife beendet.

Da erst für 2 von 8 Produktkonfigurationen ein Testfall generiert werden konnte, startet die zweite Iteration in Zeile 7 mit 6 verbleibenden Produktkonfigurationen PC_{left} (vergleiche mit Abbildung 7.1g). Für eine aus diesen 6 Produktkonfigurationen wird dann in Zeile 9 nach einem Testfall gesucht. Anschließend wird Testfall *tc*14 generiert, der in Zeile 11 der SPL-Testsuite TS_{SPL} hinzugefügt wird (vergleiche mit Abbildung 7.1i). Obwohl der konkrete Testfall *tc*14 ursprünglich für das produktspezifische Testmodell der Produktvariante *fa1* generiert wurde, welches aus dem 150%-Testmodell hergeleitet wurde, lässt sich *tc*14 für alle 8 Produktkonfigurationen verwenden (vergleiche mit Abbildung 7.1k). Nachdem nun Testfall *tc*14 in die SPL-Testsuite aufgenommen wurde, existiert für jede Produktkonfiguration der FA-SPL ein valider Testfall der Testziel *m* abdeckt. Da $PC_{tc13} \subset PC_{tc14}$ gilt, könnte der zuvor generierte Testfall *tc*13 ohne Auswirkung auf die produktspezifische Abdeckung aus der SPL-Testsuite entfernt werden. Da *tc*14 auf allen Produktkonfigurationen verwendbar ist, werden in Zeile 13 alle Produktkonfigurationen aus PC_{left} entfernt (vergleiche mit Abbildung 7.1l).

In der dritten Iteration der inneren Schleife ist die Bedingung in Zeile 7 nicht mehr erfüllt, weshalb die Schleife verlassen wird (vergleiche mit Abbildung 7.1m). Folglich wurden letztendlich für das Testziel *m* nicht mehr als 2 Testfälle generiert, um Testziel *m* auf den produktspezifischen Testmodellen der 8 Produktkonfigurationen der FA-SPL abzudecken, wohingegen ein Product-by-Product-Ansatz 8 Testfälle generiert hätte (für jede Produktkonfiguration einen).

Anschließend werden mit der äußeren Schleife die verbleibenden Testziele des 150%-Testmodells der FA-SPL bearbeitet, bis die in Abbildung 7.2a dargestellt vollständige SPL-Testsuite für die FA-SPL generiert ist. In Abbildung 7.2b sind für jeden enthaltenen Testfall die Produktkonfigurationen ersichtlich, auf welchen dieser verwendbar ist. Aus dieser vollständigen SPL-Testsuite lässt sich für jede aufgeführte Produktvariante der FA-SPL eine produktspezifische Testsuite bilden, die auf dem dazu passenden Testmodell eine vollständige Abdeckung erreicht.

7.4 EVALUATION

Der Algorithmus zur Generierung einer vollständigen SPL-Testsuite wurde sowohl auf die FA-SPL als auch auf die BCS-SPL angewendet.

Die FA-SPL besteht aus 4 Features, aus denen sich unter Berücksichtigung aller Einschränkungen 8 Produktkonfigurationen bilden lassen. Für diese Produktkonfigurationen konnte mit dem vorgestellten Algorithmus 7.1 eine vollständige SPL-Testsuite, bestehend aus 29 Testfällen, generiert werden. Diese vollständige SPL-Testsuite erreicht eine vollständige *all-transitions*-Abdeckung auf den 8 Produktkonfigurationen. Die geringe Anzahl an Testfällen wird durch die Verwendung eines 150%-Testmodells bei der Testfallgenerierung erreicht, da dieses die Erstellung von variantenübergreifend verwendbaren Testfällen ermöglicht.

Zu Vergleichszwecken kann der Algorithmus so modifiziert werden, dass auf die Möglichkeit „Testfälle variantenübergreifend zu verwenden“ verzichtet wird. In diesem Fall wird für jedes Testziel in einem produktspezifischen Testmodell

	Iter.	Zeile	PC_{left}	PC_{sel}	p_{Cbound}	Transitionspfad des Testfalls tc	PC_{tc}	TS_{SPL}
a)	1	7	*fa1,fa2, fa3, fa4, fa5, fa6, fa7, fa8*	-	-	-	-	...
b)		8	*fa1,fa2, fa3, fa4, fa5, fa6, fa7, fa8*	*fa1,fa2, fa3, fa4, fa5, fa6, fa7,fa8*	-	-	-	...
c)		10	*fa1,fa2, fa3, fa4, fa5, fa6, fa7, fa8*	*fa1,fa2, fa3, fa4, fa5, fa6, fa7,fa8*	*fa7*	*tc*13 = a c f i u b f h j k k k k k k k k l m	-	...
d)		11	*fa1,fa2, fa3, fa4, fa5, fa6, fa7, fa8*	*fa1,fa2, fa3, fa4, fa5, fa6, fa7,fa8*	*fa7*	*tc*13 = a c f i u b f h j k k k k k k k k k l m	-	..., *tc*13
e)		12	*fa1,fa2, fa3, fa4, fa5, fa6, fa7, fa8*	*fa1,fa2, fa3, fa4, fa5, fa6, fa7,fa8*	*fa7*	*tc*13 = a c f i u b f h j k k k k k k k k k l m	*fa7, fa8*	..., *tc*13
f)		13	*fa1,fa2, fa3, fa4, fa5, fa6*	*fa1,fa2, fa3, fa4, fa5, fa6, fa7,fa8*	*fa7*	*tc*13 = a c f i u b f h j k k k k k k k k k l m	*fa7, fa8*	..., *tc*13
g)	2	7	*fa1,fa2, fa3, fa4, fa5, fa6*	-	-	-	-	..., *tc*13
h)		8	*fa1,fa2, fa3, fa4, fa5, fa6*	*fa1,fa2, fa3, fa4, fa5, fa6*	-	-	-	..., *tc*13
i)		10	*fa1,fa2, fa3, fa4, fa5, fa6*	*fa1,fa2, fa3, fa4, fa5, fa6*	*fa1*	*tc*14 = a c f i u b f g g j k k k k k k k k k l m	-	..., *tc*13
j)		11	*fa1,fa2, fa3, fa4, fa5, fa6*	*fa1,fa2, fa3, fa4, fa5, fa6*	*fa1*	*tc*14 = a c f i u b f g g j k k k k k k k k k l m	-	..., *tc*13, *tc*14
k)		12	*fa1,fa2, fa3, fa4, fa5, fa6*	*fa1,fa2, fa3, fa4, fa5, fa6*	*fa1*	*tc*14 = a c f i u b f g g j k k k k k k k k k l m	*fa1, fa2, fa3, fa4, fa5, fa6, fa7, fa8*	..., *tc*13, *tc*14
l)		13	-	*fa1,fa2, fa3, fa4, fa5, fa6*	*fa1*	*tc*14 = a c f i u b f g g j k k k k k k k k k l m	*fa1, fa2, fa3, fa4, fa5, fa6, fa7, fa8*	..., *tc*13, *tc*14
m)	3	7	-	-	-	-	-	..., *tc*13, *tc*14

Abbildung 7.1: Teilschritte der Testfallgenerierung für Testziel *m*

tc#	Features				Testziel	Transitionspfad
	B	C	D	G		
1	-	-	-	-	*a*	*a*
2	-	-	-	-	*b*	*a b*
3	-	-	-	-	*c*	*a c*
4	-	-	1	0	*d*	*a d*
5	-	-	0	1	*e*	*a e*
6	-	-	-	-	*f*	*a b f*
7	-	-	-	-	*g*	*a b f g*
8	1	-	-	-	*h*	*a b f h*
9	-	-	-	-	*i*	*a b f i*
10	-	-	-	-	*j*	*a c f g j*
11	-	-	-	-	*k*	*a c f g j k*
12	-	-	-	-	*l*	*a c f g j k k k k k k k k k k l*
13	1	-	-	-	*m*	*a c f i u b f h j k k k k k k k k k k l m*
14	-	-	-	-	*m*	*a c f i u b f g g j k k k k k k k k k k l m*
15	-	-	-	-	*n*	*a c f g j k k k k k k k k k k k l n*
16	-	-	1	0	*o*	*a d f g g g g j k k k k k k k k k k k l o*
17	1	-	0	1	*p*	*a c f i u e f g g h j k k k k k k k k k k k l p*
18	-	-	0	1	*p*	*a c f i u e f g g g g g g j k k k k k k k k k k k l p*
19	-	0	-	-	*q*	*a c f g j k k k k k k k k k k k l n q*
20	-	1	-	-	*r*	*a c f g j k k k k k k k k k k k l n r*
21	-	-	-	-	*s*	*a b s*
22	-	-	1	0	*t*	*a d f g i t*
23	-	-	-	-	*t*	*a c b f g i t*
24	-	-	1	0	*u*	*a d f i u*
25	-	-	-	-	*u*	*a c f i u*
26	-	-	-	-	*v*	*a v*
27	-	-	-	-	*w*	*a v w*
28	-	-	1	0	*x*	*a d v w w w w w w w w w w x*
29	-	-	-	-	*x*	*a c v w w w w w w w w w w x*

(a) Transitionspfade

PC	Features				Testfälle (tc#)																									
	B	C	D	G	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
fa1	0	0	0	0	□	□	□			□	□		□	□	□	□		□	□				□		□		□		□	□	□		□
fa2	0	0	0	1	□	□	□		□	□	□		□	□	□	□		□	□			□	□		□		□		□	□	□		□
fa3	0	0	1	0	□	□	□	□		□	□		□	□	□	□		□	□	□			□		□	□	□	□	□	□	□	□	□
fa4	0	1	0	0	□	□	□			□	□		□	□	□	□		□	□					□	□		□		□	□	□		□
fa5	0	1	0	1	□	□	□		□	□	□		□	□	□	□		□	□			□		□	□		□		□	□	□		□
fa6	0	1	1	0	□	□	□	□		□	□		□	□	□	□		□	□	□				□	□	□	□	□	□	□	□	□	□
fa7	1	0	0	1	□	□	□		□	□	□	□	□	□	□	□	□	□	□		□	□	□		□		□		□	□	□		□
fa8	1	1	0	1	□	□	□		□	□	□	□	□	□	□	□	□	□	□		□	□		□	□		□		□	□	□		□

□ = verwendbar

(b) Variantenübergreifende Verwendbarkeit von Testfällen

Abbildung 7.2: Vollständige FA-SPL-Testsuite (*all-transitions*)

ein Testfall erstellt, ohne zu berücksichtigen, dass bereits Testfälle in anderen produktspezifischen Testsuiten existieren könnten, welche dieses Testziel bereits abdecken und sich variantenübergreifend verwenden lassen. Dafür muss im Algorithmus 7.1 die Zeile 12 durch „$PC_{tc} \leftarrow \{pc_{bound}\}$“ ersetzt werden. Mit diesem erzwungenen Product-by-Product-Vorgehen lässt sich immer noch eine vollständige SPL-Testsuite herleiten, jedoch müssen dann 168 statt 29 Testfälle erstellt werden (siehe Abbildung 7.2b). Folglich wird der große Unterschied in der Testfallanzahl mit einem Verhältnis von ungefähr 1 zu 5 durch die variantenübergreifende Verwendbarkeit von Testfällen erreicht.

Weitere Ergebnisse konnten durch die Anwendung des Ansatzes auf die BCS-SPL (siehe Abschnitt 1.2) gewonnen werden. Die BCS-SPL besitzt 9 signifikante Features (siehe Abbildung A.4), woraus sich unter Beachtung der Abhängigkeiten zwischen den Features 58 Produktkonfigurationen herleiten lassen. Das 150%-Testmodell der BCS-SPL besteht aus 44 Transitionen und 20 Zuständen (siehe Abbildung A.5). Für die Generierung einer vollständigen SPL-Testsuite für das Überdeckungskriterium *all-transitions* wurden 90 Testfälle erstellt. Bei einem wie zuvor beschriebenen Product-by-Product-Vorgehen hätten, wenn für jedes Testziel in den 58 produktspezifischen Testmodellen separat ein Testfall erstellt worden wäre, 1360 Testfälle erstellt werden müssen. Das entspricht einem Verhältnis von ungefähr 1 zu 15.

Diese Ergebnisse, sowohl für die FA-SPL als auch die BCS-SPL, zeigen im Vergleich zum Product-by-Product-Ansatz wie effizient eine vollständige SPL-Testsuite durch den neuen Ansatz erstellt werden kann. Ein Vergleich mit anderen SPL-Testsuite-Generierungsansätzen war nicht möglich, da bisher zu diesem Thema keine zu Vergleichszwecken verwendbaren Arbeiten existieren.

7.5 DISKUSSION

7.5.1 *Zuordnung des Beitrags*

Der in diesem Kapitel vorgestellte modellbasierte Testansatz zur Generierung einer vollständigen SPL-Testsuite lässt sich wie folgt in die in [UPL11] vorgestellte Taxonomie für modellbasierte Testansätze (siehe Abbildung 7.3) einordnen:

- Anwendungsmöglichkeit (Scope):
  In dieser Arbeit wird ein 150%-Testmodell zur Testfallgenerierung verwendet, welches sowohl Eingabedaten als auch die zu erwartenden Ausgabedaten spezifiziert. Jedoch ist der Ansatz auch für Modelle verwendbar, die nur Eingabedaten spezifizieren. Aufgrund von letzterem sind auch Systemmodelle geeignet.

- Eigenschaften (Characteristics):
  In dieser Arbeit wird davon ausgegangen, dass das Verhalten des verwendeten Testmodells und das Verhalten des SUT deterministisch ist, was dazu führt, dass derselbe Testfall immer dieselben Ausgaben erwartet und dieselben Ausgaben im SUT erzeugt.

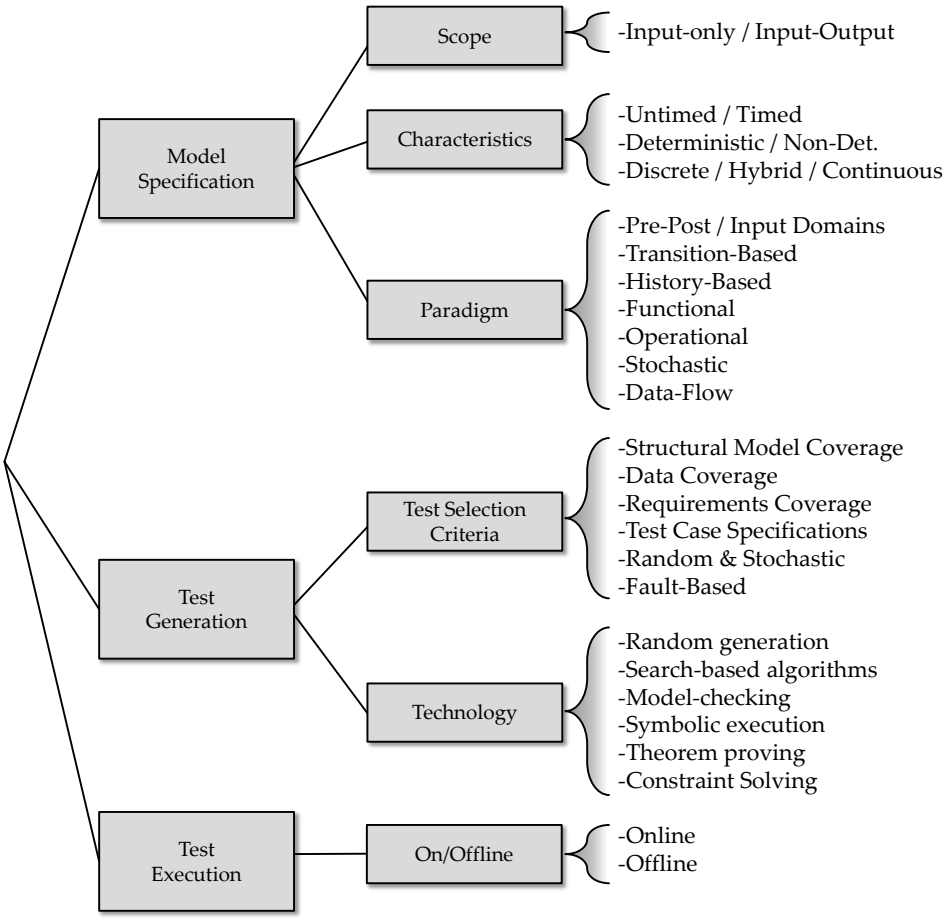


Abbildung 7.3: Taxonomy modellbasierter Testansätze
aus [UPL11]

- Bezugssystem (Paradigm):
  Zur Herleitung der Testfälle wird ein endlicher, ereignisgesteuerter Zustandsautomat als formale Spezifikation verwendet (siehe Abschnitt 6.1).

- Auswahlkriterium (Test Selection Criteria):
  Als Testauswahlkriterium wird von einem strukturellen Überdeckungskriterium ausgegangen. Bei entsprechender Modifikation sollten aber auch Überdeckungskriterien wie stochastische Überdeckung und Datenüberdeckung realisierbar sein. Letzteres lässt sich beispielsweise auf die im Automaten verwendeten Attribute übertragen.

- Technologie (Technology):
  Zur Generierung der Testfälle lassen sich prinzipiell alle Testfallgeneratoren einsetzen, welche die Testfallgenerierung basierend auf einem 150%-Testmodell unterstützen. Nähere Informationen dazu werden in Abschnitt 9.2 gegeben. In dieser Arbeit wurde zur Evaluation des Ansatzes das Testrahmenwerk Azmun (siehe Kapitel 9.1) eingesetzt, welches den Model-Checker SPIN [Holo3] als Testfallgenerator verwendet.

- On/Offline:
  Die Generierung der Testfälle erfolgt offline (ohne Interaktion mit dem SUT), da unter anderem im Domänentest (die Phase in der die SPL-Testsuite erstellt werden sollte) typischerweise noch keine Produktvarianten zum Test vorliegen.

7.5.2 *Skalierbarkeit des Ansatzes*

Der für den Ansatz angegebene Algorithmus 7.1 ist auch für große SPLs geeignet, da dieser in der Regel die Iteration über jede einzelne Produktkonfiguration vermeidet. Dies wird erreicht, indem die aus dem 150%-Testmodell erstellten Testfälle auf variantenübergreifende Verwendbarkeit hin analysiert werden. Ist dies der Fall, können all diese Produktkonfigurationen, für die der Testfall verwendbar ist, bei der Generierung eines Testfalls für das gewählte Testziel übersprungen werden. Aufgrund dieser Vorgehensweise kann eine SPL-Testsuite, die eine vollständige Überdeckung bzgl. eines Überdeckungskriteriums auf jedem produktspezifischen Testmodell erreicht, systematisch und effizient generiert werden.

Über wie viele Produktkonfigurationen explizit iteriert werden muss, ist davon abhängig wie viele der generierten Testfälle auf großen Mengen von Produktkonfigurationen verwendbar sind. Dies ist wiederum davon abhängig wie viele Gemeinsamkeiten bzw. identische Teile die produktspezifischen Testmodelle besitzen.

Die innerhalb der äußeren Schleife noch nicht bearbeiteten Produktkonfigurationen PC_{left} lassen sich, statt diese in einer langen Liste bzw. großen Tabelle aufzuführen, als logische Formel über Feature-Parameter darstellen. Eine kompakte Darstellung ist die Disjunktive Normalform (DNF). Um eine logische Formel in eine minimale DNF zu überführen, kann das Quine-McCluskey-Verfahren [JKM08] (eine Methode zur Minimierung von Booleschen Funktionen) angewendet werden. Beispielsweise lässt sich der in Zeile 8 (Algorithmus 7.1) beschriebene Vorgang wie in Abbildung 7.4 dargestellt umsetzen. Allerdings hat das Herleiten einer minimalen DNF exponentielles Laufzeitverhalten bezogen auf die Anzahl der Features. Bei zu großen Laufzeiten könnte daher ein Verzicht auf Minimalität eine Option darstellen, um die Laufzeit bei Verwendung einer DNF zu senken. Bei Verwendung von logischen Formeln lassen sich die in Zeile 13 und Zeile 15 abstrakt beschriebenen Mengenberechnungen auch konkret durch „Formel(PC_x) $\wedge(\neg$ Formel(PC_y))“ beschreiben.

Im vorgestellten Ansatz basiert die Testfallgenerierung auf einem 150%-Testmodell. Die damit einhergehenden Vor- und Nachteil sind teilweise abhängig von der gewählten Realisierung. In Abschnitt 9.2 werden zwei Realisierungen mit ihren Vor- und Nachteilen vorgestellt.

7.5.3 *Verhinderung von Redundanz während der Generierung*

Bei der Ausführung von Algorithmus 7.1 wird die dabei generierte vollständige SPL-Testsuite im Vergleich zu einem Product-By-Product-Ansatz bereits während ihrer Generierung in der Anzahl der enthaltenen Testfälle reduziert. Dies ist mög-

lich, da keine Testfälle für eine Produktvariante erstellt werden, wenn bereits ein für eine andere Produktvariante existierender Testfall für denselben Zweck verwendet werden kann. Obwohl dadurch potentiell vollkommen identische Testfälle verhindert werden, und damit Kosten bei der Generierung als auch Wartung eingespart werden, verhindert der vorgestellte Algorithmus nicht vollständig die Generierung redundanter Testfälle. In Kapitel 8 wird ausführlich auf dieses Thema eingegangen und es werden Algorithmen zur Reduktion einer SPL-Testsuite hinsichtlich der Anzahl enthaltener Testfälle, der Anzahl der benötigten Testfallausführungen und der Testfalllänge unter Erhalt der produktspezifischen Abdeckung vorgestellt.

Beispiel

Bei der Generierung einer SPL-Testsuite für die FA-SPL konnte aufgrund der Wiederverwendung von Testfällen die Anzahl der Testfälle von 168 auf 29 reduziert werden. Bei der Generierung einer SPL-Testsuite für die BCS-SPL sogar von 1360 auf 58. Dennoch enthält die für die FA-SPL generierte SPL-Testsuite noch Redundanz (siehe Abbildung 7.2a). Beispielsweise ist der gesamte Transitionspfad des Testfalls *tc*6 identisch zum Anfangsteil des Transitionspfad von Testfall *tc*7.

7.5.4 *SPL-Testsuite als Produkt-Auswahlkriterium*

Damit bei der Entwicklung einer neuen Produktvariante basierend auf der Softwareplattform einer SPL bereits ein Mindestmaß an Qualität für diese Produktvariante angenommen werden kann, sollte in der SPL-Entwicklung so früh wie möglich damit begonnen anhand des Tests einer kleinen Menge von Produktvarianten $PC_{rep} \subseteq PC$ Aussagen über die Qualität aller Produktvarianten PC der SPL treffen zu können. Welche Produktkonfigurationen dafür ausgewählt werden, hängt vom gewählten Produkt-Auswahlkriterium ab (siehe Abschnitt 5.5.2).

Eine SPL-Testsuite lässt sich auch als Produkt-Auswahlkriterium verwenden. So lässt sich mittels einer SPL-Testsuite eine Menge von Produktvarianten $PC_{rep} \subseteq PC$ auswählen, die repräsentativ bzgl. der Ausführbarkeit dieser SPL-Testsuite ist.

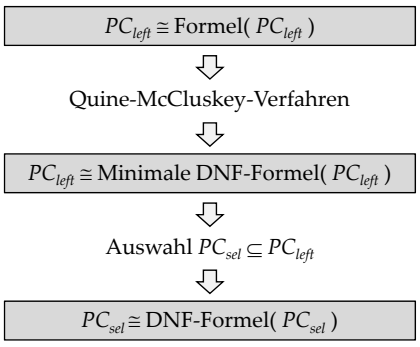


Abbildung 7.4: Auswahl der Produktkonfigurationen PC_{sel} aus PC_{left}

Definition 7.2 (Repräsentativ bzgl. der Ausführbarkeit einer SPL-Testsuite) *Eine Menge von Produktkonfigurationen* $PC_{rep} \subseteq PC$ *ist* repräsentativ *für die Produktkonfigurationen* PC *einer SPL* bezüglich der Ausführbarkeit einer SPL-Testsuite TS_{SPL}*, welche aus einem* 150%*-Testmodell* vtm *generiert wurde, genau dann wenn gilt*

$$\forall tc \in TS_{SPL} : (\exists pc \in PC_{rep} : valid(tc, bind(vtm, pc)))$$

Eine Menge von Produktkonfigurationen PC_{rep} ist folglich repräsentativ für PC bezogen auf die Ausführbarkeit einer SPL-Testsuite, wenn jeder in dieser SPL-Testsuite enthaltene Testfall auf mindestens einer der in PC_{rep} enthaltenen Produktkonfiguration verwendbar ist.

Eine solche repräsentative Produktmenge PC_{rep} lässt sich herleiten, indem für jeden Testfall $tc \in TS_{SPL}$ eine Produktvariante aus PC, auf welcher Testfall tc anwendbar ist, in die Produktmenge PC_{rep} aufgenommen wird. So wird auf einfache Weise sichergestellt, dass für jeden Testfall zumindest eine Produktvariante in dieser repräsentativen Menge enthalten ist, für die sich der Testfall verwenden lässt. Eine kleine, aber nicht zwingend minimale repräsentative Produktmenge lässt sich erreichen, wenn bevorzugt Produktvarianten in PC_{rep} aufgenommen werden, auf denen viele Testfälle, welche sich auf noch keiner bereits in PC_{rep} enthaltenen Produktkonfiguration verwenden lassen, verwendbar sind.

Beispiel

Für die 8 Produktvarianten der FA-SPL stellen die beiden Produktvarianten fa_6 und fa_7 eine repräsentative Produktmenge bzgl. der Ausführbarkeit der in Abbildung 7.2a dargestellten SPL-Testsuite dar, da sich jeder Testfall in dieser SPL-Testsuite für mindestens eine der beiden Produktvarianten verwenden lässt.

Die Anzahl der benötigten Produktvarianten für eine repräsentative Produktmenge hängt stark davon ab, mit wie vielen anderen Testfällen jeder Testfall aus der SPL-Testsuite eine gemeinsame Schnittmenge besitzt (bezüglich der Produktvarianten, auf denen der Testfall anwendbar ist). Im Folgenden wird diese Abhängigkeit an Abbildung 7.5 erklärt. Dafür wird eine SPL mit $PC = \{pc_1, pc_2, pc_3, pc_4\}$ und eine SPL-Testsuite $TS_{SPL} = \{tc', tc'', tc'''\}$ angenommen. Jeder dieser 3 Testfälle lässt sich auf einer Menge von Produktkonfigurationen verwenden, dargestellt durch $PC_{tc'}$, $PC_{tc''}$ oder $PC_{tc'''}$.

In Abbildung 7.5a besitzen alle 3 Testfälle eine Menge von Produktkonfigurationen (grafisch dargestellt durch Linien), die sich mit der Menge der anderen Testfälle überschneidet ($PC_{tc'} \cap PC_{tc''} \cap PC_{tc'''} \neq \emptyset$). Dadurch ist es möglich eine repräsentative Produktmenge mit nur einer einzigen Produktvariante, nämlich pc_3, zu bilden. Eine gemeinsame Schnittmenge ist für viele Testfälle zu erwarten, wenn wenige Einschränkungen zwischen den Features (z.B. xor) vorliegen. Dieser Fall ist beispielsweise dann erreichbar, wenn die SPL nur optionale Features aufweist und eine mit allen Features ausgestattete Produktvariante existiert, auf der alle Testfälle verwendbar sind.

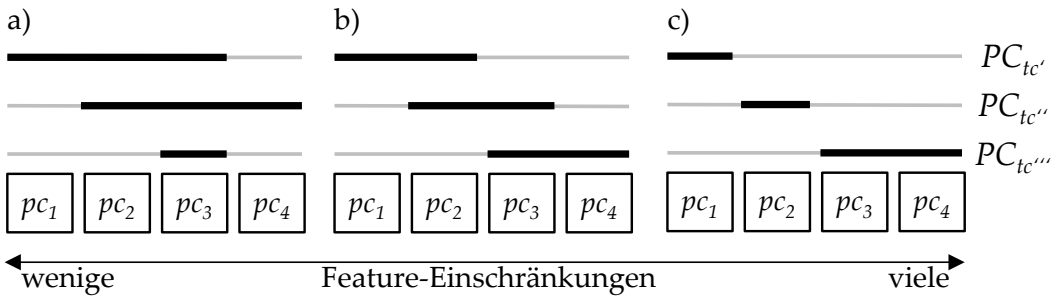


Abbildung 7.5: Beispielhafte Auswirkung der Feature-Einschränkungen auf Schnittmengen

In Abbildung 7.5c besitzt keiner der 3 Testfälle eine Menge von Produktkonfigurationen, die sich mit der Menge eines anderen Testfalls überschneidet. Folglich werden für eine repräsentative Produktmenge 3 Produktvarianten benötigt. Als Grund könnte beispielsweise aufgeführt werden, dass die Testfälle tc' und tc'' jeweils ein anderes Feature benötigen, welche sich aber gegenseitig durch eine xor-Einschränkung ausschließen, was zur Folge hat, dass keine Produktvariante existieren kann, auf der die beiden Testfälle verwendbar sind. Der schlechteste Fall, folglich $PC_{rep} = PC$, tritt ein, wenn keiner in der SPL-Testsuite enthaltenen Testfälle variantenübergreifend verwendbar ist. Dies kann den Grund haben, dass zwischen allen Features Einschränkungen (z.B. xor) existieren oder die produktspezifischen Testmodelle keine Gemeinsamkeiten besitzen.

Beispiel

Für die BCS-SPL, bestehend aus 58 Produktvarianten, wurde eine vollständige SPL-Testsuite für das Überdeckungskriterium *all-transitions* generiert. Wenn diese SPL-Testsuite als Produkt-Auswahlkriterium verwendet wird, um für die 58 Produktvarianten eine repräsentative Menge zu identifizieren, werden dafür nicht mehr als 2 Produktvarianten benötigt. Diese geringe Anzahl an Produktvarianten ist hauptsächlich auf die vielen optionalen Features und die wenigen Exclude-Abhängigkeiten im Featuremodell zurückführbar (siehe Abbildung A.4).

Ein Produkt-Auswahlkriterium sollte so gewählt werden, dass das Testergebnis der mit diesem Kriterium ausgewählten Produktvarianten $PC_{rep} \subseteq PC$ auch Rückschlüsse auf die Qualität der restlichen in PC enthaltenen Produktvarianten erlaubt. Rückschlüsse können aber nur in einem begrenzten Rahmen gezogen werden, da jede Produktvariante eine einzigartige Featurekombination aufweist, die einzigartige Seiteneffekte hervorrufen kann. Folglich sollte der Test einer repräsentativen Produktmenge bzgl. eines Produkt-Auswahlkriteriums nur dazu dienen die Qualität einer SPL einzuschätzen.

Damit die Testergebnisse einer Menge PC_{rep}, die repräsentativ bzgl. der Ausführbarkeit einer SPL-Testsuite ist, dazu genutzt werden können die Qualität einer SPL einzuschätzen, sollte sichergestellt werden, dass sich die Testergebnisse übertragen lassen. Dies ist möglich, indem die Implementierung sowie die Testmodelle aus einer gemeinsamen Softwareplattform entwickelt werden. Beispiels-

weise könnte auch für die Implementierung ein 150%-Ansatz benutzt werden. Da die zur Auswahl verwendeten Testfälle aus den Testmodellen hergeleitet werden, ist auch eine starke Korrelation zwischen der Ähnlichkeit der Produktimplementierungen (bzw. der produktspezifischen Systemmodelle) und der Ähnlichkeit der produktspezifischen Testmodelle hilfreich. So sollten Produktvarianten, deren produktspezifisches Testmodell Ähnlichkeit zueinander aufweist, auch eine große Ähnlichkeit bei deren Implementierung zueinander aufweisen, damit ein produktübergreifend verwendbarer Testfall auf diesen Produktvarianten dieselben Fehler aufdeckt bzw. dieselben Testergebnisse erzeugt. Sollte jedoch davon ausgegangen werden, dass so eine starke Korrelation in der Praxis nicht realisierbar ist, eignet sich kein Testmodell-basiertes Produkt-Auswahlkriterium für die Auswahl einer repräsentativen Produktmenge.

Unter der Annahme, dass die zuvor genannten Notwendigkeiten erfüllt sind und unter Berücksichtigung, dass produktindividuelle Seiteneffekte immer möglich sind, können alle Produktvarianten PC einer SPL als getestet angesehen werden bzgl. des Überdeckungskriteriums C, wenn jeder Testfall aus einer vollständigen SPL-Testsuite TS_{SPL} (vollständig bzgl. eines Überdeckungskriteriums C) auf einer Produktvariante $pc \in PC_{rep} \subseteq PC$ ausgeführt wird und PC_{rep} repräsentativ bzgl. der Ausführbarkeit dieser SPL-Testsuite TS_{SPL} ist.

7.5.5 *Vorgabe von zu bearbeitenden Produktkonfigurationen*

Algorithmus 7.1 bietet in Zeile 8 die Möglichkeit, eine Teilmenge von Produktkonfigurationen PC_{sel} aus der noch zu bearbeitenden Menge an Produktkonfigurationen PC_{left} auszuwählen. Diese Produktkonfigurationen PC_{sel} werden anschließend in Zeile 9 zusammen mit dem 150%-Testmodell vtm und dem abzudeckenden Testziel tg an den Testfallgenerator übergeben. Daraufhin versucht der Testfallgenerator für eine beliebige Produktkonfiguration $pc_{bound} \in PC_{sel}$ einen konkreten Testfall zu generieren, welcher das Testziel tg auf dem hergeleiteten produktspezifischen Testmodell $bind(vtm, pc_{bound})$ abdeckt. Wenn kein $pc_{bound} \in PC_{sel}$ existiert, für das ein konkreter Testfall generiert werden kann, sind alle $pc \in PC_{sel}$ in einem Schritt abgearbeitet worden, da sie verworfen werden können. Abhängig von der Zielsetzung des Testers kann PC_{sel} viele oder nur wenige Produktkonfigurationen von PC_{left} enthalten. Angenommen es wäre erwünscht, dass mit dem vorgestellten Algorithmus solange Testfälle für PC_{left} generiert werden, bis für die restlichen $pc \in PC_{left}$ kein Testfall mehr generiert werden kann, was zur Folge hätte, dass alle verbleibenden $pc \in PC_{left}$ in einem Schritt abgearbeitet wären, müsste $PC_{sel} = PC_{left}$ gelten.

Es gibt aber auch Gründe in denen die Auswahl einer Teilmenge $PC_{sel} \subset PC_{left}$ gerechtfertigt ist. So kann, statt dem Testfallgenerator die volle Freiheit bei der Auswahl eines geeigneten $pc \in PC_{left}$ zu lassen, bereits eine Priorisierung von außerhalb für die verbleibenden Produktkonfigurationen PC_{left} vorgegeben sein (z.B. vom Tester), welche durch $PC_{sel} \subset PC_{left}$ dargestellt wird.

Beispiel

Der SPL-Testansatz MoSo-PoLiTe [OMR10] identifiziert aus allen Produktvarianten einer SPL mittels einer Paarbildungsmethode über Features eine bezogen auf die Größe der SPL relativ kleine Teilmenge von Produktvarianten $PC_{prio} \subseteq PC$. Diese Menge wird zum Test von Interaktionen zwischen Features verwendet. Bei der Generierung einer vollständigen SPL-Testsuite für dieselbe SPL könnte nun gefordert werden, die Produktvarianten $PC_{sel} = (PC_{prio} \cap PC_{left})$ allen anderen Produktvarianten vorzuziehen bis $(PC_{prio} \cap PC_{left}) = \emptyset$ gilt.

Ein weiterer Grund, nur eine Teilmenge der zur Verarbeitung ausstehenden Produktkonfigurationen zu verwenden (folglich $PC_{sel} \subset PC_{left}$), können limitierte Ressourcen aber auch technische Einschränkungen sein.

Beispiel

Der zur Evaluation des Ansatzes verwendete Testfallgenerator, der Model-Checker SPIN, bekommt als Eingabe ein Modell *vtm* (in Promela-Code) und eine zu verifizierende Eigenschaft (als LTL-Formel) übergeben. Letztere entspricht einer Trap-Property (siehe Abschnitt 3.6), die bei der SPL-Testsuitegenerierung neben dem Testziel *tg* auch die Menge an Produktkonfigurationen PC_{sel} in Form einer aussagenlogischen Formel beschreibt. Werden alle zur Verarbeitung ausstehenden Produktkonfigurationen $PC_{sel} = PC_{left}$ in einer einzelnen LTL-Formel dargestellt, kann diese sehr lang werden. Jedoch unterstützt SPIN nur LTL-Formeln mit einer Länge von bis zu 1024 Zeichen. Deshalb können häufig nicht alle Produktkonfigurationen ausgewählt werden, sondern nur ein Teil $PC_{sel} \subseteq PC_{left}$, da dann die LTL-Formel weniger Zeichen benötigt.

8

REDUKTION VON SPL-TESTSUITEN

Im SPL-Test werden bereits bewährte Techniken aus unterschiedlichen Bereichen des Softwaretests (z.B. Modelbasierter Test und Regressionstest) angewendet, um den Test der Produktvarianten einer SPL effizienter zu gestalten und den Testaufwand zu verringern. Dafür werden beispielsweise Testfälle und Testmodelle bei geeigneten Produktvarianten wiederverwendet (siehe Kapitel 5). Eine weitere Technik zur Verringerung des Testaufwands ist die Testsuite-Reduktion.

Die Testsuite-Reduktion bezeichnet im Allgemeinen einen Vorgang, bei dem die Anzahl der in einer Testsuite enthaltenen Testfälle reduziert wird, um die Kosten bei der Testfallausführung zu senken (siehe Kapitel 4). Da im SPL-Test bei vielen zu testenden Produktvarianten auch mit vielen auszuführenden Testfällen zu rechnen ist, bietet sich die Testsuite-Reduktion im SPL-Kontext zur Verringerung der dabei entstehenden Testkosten an.

Beispiel

Die 29 Testfälle der in Abbildung 7.2a dargestellten vollständigen SPL-Testsuite erreichen eine *all-transitions*-Abdeckung auf den 8 Produktvarianten der FA-SPL. Wenn jeder dieser 29 Testfälle auf jeder Produktvariante zur Ausführung eingeplant wird, für welche dieser Testfall verwendbar ist, hätte das 168 Testfallausführungen zur Folge (siehe Abbildung 8.1). Die große Abweichung zwischen der Anzahl der in der SPL-Testsuite enthaltenen Testfälle und der Anzahl der zur Ausführung eingeplanten Testfälle ist darauf zurückzuführen, dass jeder Testfall für mehr als eine Produktvariante verwendbar ist. Für die folgenden Beispiele wird als Motivation angenommen, dass die Anzahl der enthaltenen Testfälle und die Anzahl der benötigten Testfallausführungen reduziert werden sollen, damit die Kosten für die Testfallwartung und Testfallausführung sinken.

PC	Features				Testfälle (tc#)																										
	B	C	D	G	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
fa1	0	0	0	0	☑	☑	☑			☑	☑		☑	☑	☑	☑		☑	☑				☑		☑		☑		☑	☑	☑		☑
fa2	0	0	0	1	☑	☑	☑		☑	☑	☑		☑	☑	☑	☑		☑	☑			☑	☑		☑		☑		☑	☑	☑		☑
fa3	0	0	1	0	☑	☑	☑	☑		☑	☑		☑	☑	☑	☑		☑	☑	☑			☑		☑	☑	☑	☑	☑	☑	☑	☑	☑
fa4	0	1	0	0	☑	☑	☑			☑	☑		☑	☑	☑	☑		☑	☑					☑	☑		☑		☑	☑	☑		☑
fa5	0	1	0	1	☑	☑	☑		☑	☑	☑		☑	☑	☑	☑		☑	☑			☑		☑	☑		☑		☑	☑	☑		☑
fa6	0	1	1	0	☑	☑	☑	☑		☑	☑		☑	☑	☑	☑		☑	☑	☑				☑	☑	☑	☑	☑	☑	☑	☑	☑	☑
fa7	1	0	0	1	☑	☑	☑		☑	☑	☑	☑	☑	☑	☑	☑	☑	☑	☑		☑	☑	☑		☑		☑		☑	☑	☑		☑
fa8	1	1	0	1	☑	☑	☑		☑	☑	☑	☑	☑	☑	☑	☑	☑	☑	☑		☑	☑		☑	☑		☑		☑	☑	☑		☑

☐ = verwendbar ☑ = eingeplant

Abbildung 8.1: Zur Ausführung eingeplante Testfälle der FA-SPL-Testsuite

Existierende Testsuite-Reduktionstechniken können nicht ohne weiteres auf den SPL-Kontext übertragen werden. Diese gehen zumeist implizit von der Annahme aus, dass die Testfälle einer Testsuite nur auf einem einzigen Produkt ausgeführt werden. Folglich gehen sie nicht von einer variantenübergreifenden Verwendung von Testfällen aus, was dazu führen kann, dass der Einsatz solcher Reduktionstechniken im SPL-Kontext die erreichte produktspezifische Testabdeckung bzgl. eines Überdeckungskriteriums verringert bzw. den Erhalt nicht garantiert [YH08].

Beispiel

Mit den bisherigen Testsuite-Reduktionsansätzen könnte aus der vollständigen SPL-Testsuite (siehe Abbildungen 7.2a und 8.1) fälschlicherweise der Testfall $tc25$ entfernt werden, in der Annahme, dass das von diesem Testfall abgedeckte Testziel u auch noch von Testfall $tc24$ abgedeckt wird. Dass Testfall $tc24$ aber nur auf fa_3 sowie fa_6 und nicht wie $tc25$ auf allen 8 Produktvarianten verwendbar ist, würde beim Entfernen nicht berücksichtigt werden. Somit ließe sich das Testziel u auf 6 von 8 Produktvarianten durch keinen Testfall mehr abdecken.

Bei Testsuite-Reduktionstechniken für Testsuiten von Einzel-Softwaresystemen entspricht die Anzahl der in einer Testsuite enthaltenen Testfälle der Anzahl der Testfallausführungen, wenn alle enthaltenen Testfälle zur Ausführung eingeplant sind. Im SPL-Test können jedoch Testfälle existieren, die sich variantenübergreifend verwenden lassen. Das hat zur Folge, dass die Anzahl der eingeplanten Testfallausführungen wesentlich höher ausfallen kann als die Anzahl der in einer SPL-Testsuite enthaltenen Testfälle. Da die Anzahl der enthaltenen Testfälle und die Anzahl der Testfallausführungen im SPL-Test voneinander entkoppelt sind, müssen beide Werte bei der Reduktion voneinander unabhängig betrachtet werden, um die Auswirkung der Reduktion auf die Kosten im Testprozess bewerten zu können. So kann eine geringfügige Reduktion der Testfallanzahl eine große Reduktion in der Anzahl der Testfallausführungen zur Folge haben.

8.1 SPL-TESTSUITE-REDUKTIONSPROBLEM

Die Beschreibung des in Kapitel 4 vorgestellten Testsuite-Reduktionsproblems (siehe Definition 4.1) lässt Raum für Interpretationen, wenn es auf den SPL-Kontext übertragen wird. So soll für eine Testsuite $TS = \{tc_1, tc_2, ..., tc_m\}$, die eine Menge von Testzielen $TG = \{tg_1, tg_2, ..., tg_n\}$ abdeckt, eine Teilmenge von Testfällen identifiziert werden, mit der sich jedes Testziel $tg \in TG$ abdecken lässt und deren Testfallanzahl minimal ist.

Allerdings ist in dieser Beschreibung nicht eindeutig festgelegt, wie vorgegangen werden soll, wenn mehrere Produktvarianten dasselbe Testziel aufweisen, wie es bei der Anwendung eines 150%-Testmodells im SPL-Kontext möglich ist. So lässt sich die Beschreibung dahingehend interpretieren, dass es ausreicht, einen Testfall zu erstellen, der das in mehreren Produktvarianten vorhandene Testziel abdeckt, aber der nicht auf allen Produktvarianten verwendbar ist, die das Testziel enthalten. In diesem Fall wäre das Testziel zwar wie gefordert abgedeckt, aber nicht auf allen Produktvarianten. Es könnte aber auch davon ausgegangen werden, dass selbst, wenn mehrere Produktvarianten dasselbe Testziel aufweisen,

für jede Produktvariante ein valider Testfall vorliegen muss, der das Testziel auf dieser abdeckt. Wenn letzteres der Fall ist und eine vollständige Überdeckung jeder einzelnen Produktvarianten einer SPL angestrebt wird, welche nach der Reduktion bestehen soll, muss die Definition wie folgt erweitert werden, um eine eindeutige Interpretation zu bieten:

Definition 8.1 (SPL-Testsuite-Reduktionsproblem) *Es existieren eine SPL-Testsuite $TS_{SPL} = \{tc_1, ..., tc_m\}$ und für jede Produktvariante $pc_i \in PC$ einer SPL eine Menge von Testzielen $TG_i = \{tg_{i,1}, ..., tg_{i,n}\}$, die auf dieser Variante pc_i durch valide Testfälle $tc \in TS_{SPL}$ abgedeckt sein müssen, um die geforderte Überdeckung zu erreichen.*

Finde eine Teilmenge von TS_{SPL}, sodass die Anzahl der enthaltenen Testfälle minimal ist und jedes Testziel $tg \in TG_i$ im produktspezifischen Testmodell einer jeden Produktvariante $pc_i \in PC$ durch mindestens einen validen Testfall abgedeckt wird.

Durch die erweiterte Definition ist nun selbst bei der Verwendung eines 150%-Testmodells für die SPL-Testsuite-Generierung klar definiert, dass es nicht ausreichend ist, einen einzigen validen Testfall für ein Testziel auf dem 150%-Testmodell zu erstellen, wenn sich dieser nicht für alle Produktvarianten verwenden lässt, die dasselbe Testziel aufweisen.

Beispiel

Die SPL-Testsuite der FA-SPL besitzt 29 Testfälle (siehe Abbildung 7.2a). Angenommen die SPL-Testsuite würde nach Definition 4.1 reduziert werden und die Testziele wären aus dem 150%-Testmodell hergeleitet, dann könnten bei einer Reduktion gegebenenfalls die Testfälle $tc14$, $tc18$, $tc23$, $tc25$ und $tc29$ aus dieser Testsuite entfernt werden, da auch ohne diese Testfälle die Abdeckung auf dem 150%-Testmodell nicht sinkt. Allerdings hätte diese Reduktion zur Folge, dass die produktspezifische Abdeckung einzelner Produktvarianten sinkt, da viele der verbleibenden Testfälle für diese nicht verwendbar sind, um die nicht mehr abgedeckten Testziele abzudecken.

Im Folgenden werden drei im SPL-Kontext einsetzbare Algorithmen zur Testsuite-Reduktion vorgestellt, welche, trotz Reduktion in der Anzahl der Testfälle und Testfallausführungen, die erreichte Abdeckung auf jeder Produktvariante erhalten. Den Erhalt der erreichten Fehlersensitivität (siehe Definition 4.3) einer SPL-Testsuite können diese Algorithmen allerdings nicht garantieren. Aus diesem Grund sollten die Algorithmen nur dann angewendet werden, wenn eine SPL-Testsuite aufgrund von Kostengründen reduziert werden muss und die geforderte Überdeckung bereits ausreichend sensitiv gegenüber Fehlern ist.

In Abschnitt 8.2 wird ein Algorithmus vorgestellt, der redundante Testfälle aus einer vollständige SPL-Testsuite *entfernt*, wohingegen der zweite Algorithmus in Abschnitt 8.3 eine Reduktion erreicht, indem dieser Paare von existierenden Testfällen durch jeweils einen einzelnen, neu erstellten Testfall *ersetzt*. Ein dritter in Abschnitt 8.4 vorgestellter Algorithmus stellt eine Erweiterung des zweiten Algorithmus dar und ergänzt diesen um ein weiteres Reduktionsziel.

8.2 ENTFERNEN REDUNDANTER TESTFÄLLE

Eine Möglichkeit, die Anzahl der Testfälle und die damit einhergehende Anzahl der möglichen Testfallausführungen in einer SPL-Testsuite TS_{SPL} zu reduzieren, besteht im gezielten Entfernen redundanter Testfälle. Ein Testfall einer SPL-Testsuite ist redundant, wenn sich auf keinem produktspezifischen Testmodell die erreichte Abdeckung der Testziele durch die SPL-Testsuite mit oder ohne diesen Testfall verändert. Dass ein Testfall tc redundant ist, lässt sich im einfachsten Fall daran erkennen, wenn ein Testfall $tc' \in TS_{SPL}$, $tc \neq tc'$ existiert, der tc *subsumiert*. Das heißt, tc' deckt mindestens dieselben Testziele wie tc auf mindestens denselben Produktkonfigurationen wie tc ab. Ausgehend von einem 150%-Testmodell vtm, einer Menge von Testzielen TG und einer Menge von zu testenden Produktkonfigurationen PC lässt sich eine Subsumptionsrelation $tc \preceq tc'$ für ein Testfallpaar $tc, tc' \in TS_{SPL}$ wie folgt definieren:

Definition 8.2 (SPL-Testfall-Subsumptionsrelation)

$$
\begin{gathered}
tc \preceq tc' :\Leftrightarrow \\
\forall tg \in TG, pc \in PC : \\
valid(tc, bind(vtm, pc)) \wedge covers(tc, tg)) \\
\Rightarrow \\
valid(tc', bind(vtm, pc)) \wedge covers(tc', tg))
\end{gathered}
$$

Beispiel

In der in Abbildung 7.2a dargestellten SPL-Testsuite wird der Testfall $tc6$ von Testfall $tc7$ subsumiert ($tc6 \preceq tc7$).

In der Regel sind redundante Testfälle in einer SPL-Testsuite nicht vollständig auf Subsumption zwischen Paaren von Testfällen zurückführbar. Vielmehr können Subsumptionsbeziehungen auch zwischen *Mengen* von Testfällen der Testsuite bestehen. Daher wird der Subsumptionsbegriff durch die folgende Definition weiter verallgemeinert.

Definition 8.3 (SPL-Testsuite-Subsumptionsrelation) *Eine SPL-Testsuite-Subsumptionsrelation $\preceq \subseteq \mathcal{P}(TS_{SPL}) \times \mathcal{P}(TS_{SPL})$ ist eine Ordnung auf Teilmengen von den Testfällen einer TS_{SPL}, sodass:*

$$
\begin{gathered}
TS'_{SPL} \preceq TS''_{SPL} :\Leftrightarrow \\
\forall tg \in TG, pc \in PC : \\
(\exists tc' \in TS'_{SPL} : valid(tc', bind(vtm, pc)) \wedge covers(tc', tg)) \\
\Rightarrow \\
(\exists tc'' \in TS''_{SPL} : valid(tc'', bind(vtm, pc)) \wedge covers(tc'', tg))
\end{gathered}
$$

Algorithmus 8.1 Entfernen von redundanten Testfällen

```
1: procedure removeTestCases(TS_SPL,vtm)
2:     PC ← { pc : F → 𝔹 | FM(pc) = true }
3:     for all tc ∈ TS_SPL do
4:         if TS_SPL ⪯ TS_SPL \ {tc} then
5:             TS_SPL ← TS_SPL \ {tc}
6:         end if
7:     end for
8: end procedure
```

Beispiel

In der in Abbildung 7.2a dargestellten SPL-Testsuite wird die Testfallmenge bestehend aus $tc7$ und $tc10$ von der Testfallmenge bestehend aus $tc6$, $tc19$ und $tc20$ subsumiert ($\{tc7, tc10\} \preceq \{tc6, tc19, tc20\}$).

Durch den nun definierten Subsumptionsbegriff ergibt sich folgender Satz, der sich direkt aus Definition 8.3 beweisen lässt:

Proposition 8.1 (SPL-Testsuite-Reduktion) *Wenn TS_{SPL} eine vollständige SPL-Testsuite ist, dann ist TS'_{SPL} auch eine vollständige SPL-Testsuite, genau dann wenn $TS_{SPL} \preceq TS'_{SPL}$ gilt.*

Folglich kann die Anzahl der Testfälle einer SPL-Testsuite mit Hilfe des eingeführten Subsumptionsbegriffes um redundante Testfälle reduziert werden, ohne die Abdeckung auf einem der produktspezifischen Testmodelle der SPL zu verringern.

8.2.1 *Algorithmus* removeTestCases

Im Folgenden wird der Algorithmus 8.1 zum Entfernen redundanter Testfälle aus einer SPL-Testsuite näher erläutert. Dieser Algorithmus überprüft jeden Testfall tc in einer SPL-Testsuite TS_{SPL} daraufhin (Zeile 3), ob jedes von tc abgedeckte Testziel auf jeder Produktkonfiguration $pc \in PC_{tc}$, auf der tc verwendbar ist, mindestens von noch einem anderen Testfall $tc' \in TS_{SPL}$ abgedeckt wird (Zeile 4). Sollte für ein Testziel tg einer Produktkonfiguration pc kein weiterer Testfall $tc' \in TS_{SPL}$ existieren, welcher tg abdeckt, handelt es sich bei tc um einen essentiellen Testfall (siehe Definition 4.2), der nicht aus der SPL-Testsuite entfernt werden kann ohne die SPL-Abdeckung zu verringern. Sollte es sich bei tc um keinen essentiellen Testfall handeln, ist dieser redundant und kann aus der TS_{SPL} entfernt werden (Zeile 5).

Damit für jedes von Testfall tc abgedeckte Testziel tg auf jeder Produktkonfiguration $pc \in PC_{tc}$, für die tc valide ist, effizient ermittelt werden kann, ob noch ein weiterer Testfall $tc' \in TS_{SPL}$ neben tc existiert der tg auf pc abdeckt, sollte für jedes Testziel in jeder Produktvariante protokolliert werden, von wie vielen Testfällen dieses Testziel abgedeckt wird (max. Speicherverbrauch: $|TG| \cdot |PC|$). Der Speicherverbrauch fällt beispielsweise wesentlich geringer aus, wenn die Produktkonfigurationen implizit als aussagenlogische Formeln über Feature-Parametern

tc#	Features				Transitionspfad
	B	C	D	G	
14	-	-	-	-	*a c f i u b f g g j k k k k k k k k k k l m*
16	-	-	1	0	*a d f g g g g j k k k k k k k k k k k l o*
17	1	-	0	1	*a c f i u e f g g h j k k k k k k k k k k k l p*
18	-	-	0	1	*a c f i u e f g g g g g g j k k k k k k k k k k k l p*
19	-	0	-	-	*a c f g j k k k k k k k k k k k k l n q*
20	-	1	-	-	*a c f g j k k k k k k k k k k k k l n r*
21	-	-	-	-	*a b s*
23	-	-	-	-	*a c b f g i t*
29	-	-	-	-	*a c v w w w w w w w w w w x*

(a) Transitionspfade

PC	Features				Testfälle (tc#)								
	B	C	D	G	14	16	17	18	19	20	21	23	29
fa1	0	0	0	0	☑				☑		☑	☑	☑
fa2	0	0	0	1	☑			☑	☑		☑	☑	☑
fa3	0	0	1	0	☑	☑			☑		☑	☑	☑
fa4	0	1	0	0	☑					☑	☑	☑	☑
fa5	0	1	0	1	☑			☑		☑	☑	☑	☑
fa6	0	1	1	0	☑	☑				☑	☑	☑	☑
fa7	1	0	0	1	☑		☑	☐	☑		☑	☑	☑
fa8	1	1	0	1	☑		☑	☐		☑	☑	☑	☑

☐ = verwendbar ☑ = eingeplant

(b) Produktspez. Testsuites (minimal)

Abbildung 8.2: FA-SPL-Testsuite (*all-transitions*) ohne redundante Testfälle

statt explizit als Menge dargestellt werden. Beispielsweise kann eine DNF verwendet werden, wie es bereits in Abschnitt 7.5.2 vorgeschlagen wurde.

Nach dem Entfernen eines Testfalls aus der TS_{SPL} muss die von den Testfällen auf den Produktkonfigurationen erreichte Abdeckung für jedes betreffende Testziel aktualisiert werden, damit in der nächsten Iteration (Zeile 3) aktuelle Daten bezüglich der SPL-Abdeckung vorliegen. Der Algorithmus kann nicht die Herleitung einer minimalen SPL-Testsuite garantieren. Dafür müsste die größtmögliche Menge an redundanten Testfällen aus einer SPL-Testsuite entfernt werden. Die Identifikation dieser Menge ist NP-schwer [HCL+03].

8.2.2 *Evaluation*

Bei der Anwendung des Algorithmus 8.1 auf die in Abbildung 7.2a dargestellte vollständige SPL-Testsuite der FA-SPL (bestehend aus 29 Testfällen) lassen sich 20 redundante Testfälle entfernen, sodass anschließend nur noch 9 Testfälle in der SPL-Testsuite verbleiben (siehe Abbildung 8.2a). Die Reduktion in der Anzahl der Testfälle wirkt sich auch auf die Anzahl der maximal möglichen einplanbaren Testfallausführungen aus. So sind in Abbildung 8.2b für jede Produktkonfiguration diejenigen Testfälle dargestellt, die sich auf dieser Produktkonfiguration verwenden lassen. Zusätzlich dazu ist für jede Produktkonfiguration die minima-

le Menge an Testfällen gekennzeichnet, die zur Ausführung eingeplant werden müssen, um die gewünschte Abdeckung *all-transitions* zu erreichen. Auf die gesamte SPL-Testsuite betrachtet sinkt durch die Reduktion der Testfallanzahl auch die Anzahl von 168 möglichen Testfallausführungen auf 48 minimal benötigte Testfallausführungen (vergleiche Abbildung 8.1 mit Abbildung 8.2b).

Die vollständige SPL-Testsuite der BCS-SPL für das Überdeckungskriterium *all-transitions* mit 90 Testfällen konnte durch das Entfernen redundanter Testfälle auf 44 Testfälle reduziert werden. Dadurch ließ sich die Anzahl der benötigten Testfallausführungen von 1360 auf 574 reduzieren.

8.2.3 *Diskussion*

Es ist zu beachten, dass ein auf einer Produktvariante verwendbarer Testfall nicht zwangsläufig auch zur Ausführung eingeplant sein muss, um die angestrebte Abdeckung der produktspezifischen Testmodelle zu gewährleisten.

Beispiel

In der reduzierten SPL-Testsuite (siehe Abbildung 8.2b) ist Testfall *tc*18 auf 4 Produktvarianten verwendbar, wird aber nur zur Ausführung auf 2 Produktvarianten eingeplant. Trotzdem ist das ausreichend, um mit den anderen in der SPL-Testsuite enthaltenen Testfällen eine *all-transitions*-Abdeckung aller Produktkonfigurationen zu erreichen.

Im Gegensatz zu bisher existierenden Testsuite-Reduktionstechniken berücksichtigt Algorithmus 8.1, dass im SPL-Kontext Testfälle existieren können, die zur Ausführung auf mehreren Produktvarianten eingeplant sind. Dadurch lässt sich die Anzahl der in einer SPL-Testsuite enthaltenen Testfälle reduzieren ohne dabei unbewusst die Abdeckung der einzelnen Produktvarianten zu verringern. Aufgrund der variantenübergreifenden Verwendung von Testfällen kann selbst eine geringe Reduktion in der Testfallanzahl eine starke Reduktion in der Anzahl der Testfallausführungen zur Folge haben. Der vorgestellte Algorithmus trägt dadurch zur Kostensenkung im Testprozess einer SPL sowohl bei der Wartung der enthaltenen als auch bei der Ausführung der eingeplanten Testfälle bei.

Sollte eine SPL-Testsuite bereits soweit reduziert worden sein, dass diese ausschließlich aus essentiellen Testfällen besteht und somit redundanzfrei ist, lässt sich eine weitere Reduktion in der Testfallanzahl unter Erhalt der produktspezifischen Abdeckung alle Produktkonfigurationen nur noch durch das Ersetzen von Testfällen erreichen.

8.3 ERSETZEN VON TESTFÄLLEN

Eine weitere Möglichkeit zur Reduktion der Testfallanzahl sowie der Anzahl der benötigten Ausführungen besteht im Ersetzen von Testfällen. Im Folgenden wird die Anzahl der in einer SPL-Testsuite TS_{SPL} enthaltenen Testfälle reduziert, indem eine Menge neu erstellter Testfälle TS^{in}_{SPL} mit $TS^{in}_{SPL} \cap TS_{SPL} = \emptyset$ eine größere Menge von Testfällen $TS^{out}_{SPL} \subseteq TS_{SPL}$ ersetzt. Dabei wird gefordert, dass TS^{in}_{SPL}

mindestens die Testziele auf den Produktvarianten abdeckt, die nicht abgedeckt wären, wenn TS^{out}_{SPL} ohne Ersatz aus TS_{SPL} entfernt werden würde. Dieses Vorgehen stellt keinen diekten Lösungsansatz für das Testsuite-Reduktionsproblem dar, da nicht ausschließlich existierende Testfälle entfernt sondern auch neue erstellt werden. Das Ersetzen von Testfällen sollte nur dann zwecks Kosteneinsparung durchgeführt werden, wenn Grund zur Annahme besteht, dass durch Erstellung und Ausführung der neu erstellten Testfälle weniger Kosten als durch die Ausführung der zu ersetzenden Testfälle entstehen. Damit beim Ersetzen von Testfällen eine Reduktion erreicht wird, müssen durch TS^{in}_{SPL} weniger Testfälle zur SPL-Testsuite hinzugefügt werden als durch TS^{out}_{SPL} entfernt werden. Folglich muss gelten:

$$| TS^{out}_{SPL} | > | TS^{in}_{SPL} |$$
$$\text{und somit auch}$$
$$| TS_{SPL} | > | (TS_{SPL} \setminus TS^{out}_{SPL}) \cup TS^{in}_{SPL} |$$

Damit beim Ersetzen die produktspezifische Abdeckung auf allen Produktkonfigurationen erhalten bleibt, müssen die Testfälle in TS^{in}_{SPL} alle Testziele auf den produktspezifischen Testmodellen abdecken, die durch das Entfernen von TS^{out}_{SPL} nicht mehr abgedeckt sind. Es muss gelten:

$$TS^{out}_{SPL} \preceq (TS_{SPL} \setminus TS^{out}_{SPL}) \cup TS^{in}_{SPL}$$
$$\text{und somit auch}$$
$$TS_{SPL} \preceq (TS_{SPL} \setminus TS^{out}_{SPL}) \cup TS^{in}_{SPL}$$

Das heißt, wenn ein Testziel tg einer Produktvariante $pc \in PC$ nur von Testfällen aus TS^{out}_{SPL} abgedeckt wird und von keinem verbleibenden Testfall in $TS_{SPL} \setminus TS^{out}_{SPL}$, dann muss mindestens ein Testfall in TS^{in}_{SPL} existieren, der tg auf pc abdeckt. Sollten jedoch noch verbleibende Testfälle in $TS_{SPL} \setminus TS^{out}_{SPL}$ existieren, die tg auf pc ebenfalls abdecken, sinkt die Anzahl der Testziele $tg \in TG_{uncovered}$, die von den neu zu erstellenden Testfällen zwingend abgedeckt werden müssen.

8.3.1 *Algorithmus* REPLACETESTCASES

Im Folgenden wird Algorithmus 8.2 erläutert, mit dem sich geeignete Testfallpaare in einer SPL-Testsuite durch jeweils einen neu erstellten Testfall ersetzen lassen.

Als Ausgangssituation wird angenommen, dass die zu reduzierende SPL-Testsuite keine redundanten Testfälle mehr enthält, da das Entfernen von redundanten Testfällen kostengünstiger ist als das Ersetzen, bei dem immer ein neuer Testfall erstellt werden muss, und somit bei effizienter Vorgehensweise bereits vorher erfolgt wäre.

Algorithmus 8.2 versucht jeweils ein Testfallpaar (tc', tc'') aus der SPL-Testsuite TS_{SPL} durch einen neu erstellten Testfall $tc_{substitute}$ zu ersetzen. Dabei werden nur solche Paare betrachtet, deren zwei Testfälle auf genau derselben Menge an Produktvarianten verwendbar sind (Zeile 5). Damit nach dem Ersetzungsschritt weiterhin jedes Testziel auf jeder Produktvariante abgedeckt wird, werden die Testziele $tg \in TG_{uncovered}$ bestimmt (Zeile 6), die nicht mehr abgedeckt werden, wenn

Algorithmus 8.2 Ersetzen von Testfallpaaren

1: **function** REPLACETESTCASES(TS_{SPL},vtm,TG)
2: **for all** $tc', tc'' \in TS_{SPL}, tc' \neq tc''$ **do**
3: $PC_{tc'} \leftarrow \{pc \in PC \mid (\varphi(tc'))(pc) = true\}$
4: $PC_{tc''} \leftarrow \{pc \in PC \mid (\varphi(tc''))(pc) = true\}$
5: **if** $PC_{tc'} = PC_{tc''}$ **then**
6: $TG_{uncovered} \leftarrow \{tg \in TG \mid \exists pc \in PC : \nexists\, tc' \in TS_{SPL} \setminus \{tc', tc''\}, valid(tc', bind(vtm, pc)), covers(tc', tg)\}$
7: $tc_{substitute} \leftarrow generate(TG_{uncovered}, PC_{tc'}, vtm)$
8: **if** $\exists pc_{bound} \in PC_{tc'} : valid(tc_{substitute}, bind(vtm, pc_{bound}))$ **then**
9: $PC_{tc_{substitute}} \leftarrow \{pc \in PC \mid (\varphi(tc_{substitute}))(pc) = true\}$
10: **if** $PC_{tc_{substitute}} \supseteq PC_{tc'}$ **then**
11: $TS_{SPL} \leftarrow TS_{SPL} \setminus \{tc', tc''\}$
12: $TS_{SPL} \leftarrow TS_{SPL} \cup \{tc_{substitute}\}$
13: **end if**
14: **end if**
15: **end if**
16: **end for**
17: **end function**

das Testfallpaar ersatzlos aus der Testsuite entfernt wird. Für diese ausgewählten Testziele wird anschließend ein neuer Testfall $tc_{substitute}$ gesucht, der auf den Produktkonfigurationen der beiden zu ersetzenden Testfälle verwendbar ist (Zeile 7). Wurde ein solcher Testfall $tc_{substitute}$ gefunden bzw. generiert (Zeile 8), muss anschließend die Menge an Produktkonfigurationen $PC_{tc_{substitute}}$ ermittelt werden, auf denen $tc_{substitute}$ verwendbar ist. Wenn der neue Testfall $tc_{substitute}$ mindestens auf denselben Produktkonfigurationen verwendbar ist wie das Testfallpaar (Zeile 10), kann Testfall $tc_{substitute}$ das Testfallpaar ersetzen ohne die SPL-Abdeckung zu senken, da gilt:

$$TS_{SPL} \preceq (TS_{SPL} \setminus \{tc', tc''\}) \cup \{tc_{substitute}\}$$

Durch jeden Ersetzungsschritt reduziert sich die Anzahl der in der SPL-Testsuite TS_{SPL} enthaltenen Testfälle um 1, wohingegen die Anzahl der minimal benötigten Testfallausführungen gegebenenfalls höher ausfallen kann.

Damit die Testziele für $TG_{uncovered}$ in Zeile 6 effizient ermittelt werden können, sollte für jedes Testziel in jeder Produktvariante protokolliert werden, von wie vielen Testfällen dieses abgedeckt wird (max. Speicherverbrauch: $|TG| \cdot |PC|$). Der Speicherverbrauch fällt beispielsweise wesentlich geringer aus, wenn die Produktkonfigurationen implizit als aussagenlogische Formeln über Feature-Parametern statt explizit als Menge dargestellt werden. Beispielsweise kann eine DNF verwendet werden, wie es bereits in Abschnitt 7.5.2 vorgeschlagen wurde. Nachdem die SPL-Testsuite durch den Ersetzungsvorgang verändert wurde, muss die SPL-Abdeckung für die nächste Iteration aktualisiert werden.

8.3.2 *Evaluation*

Wird Algorithmus 8.2 auf die redundanzfreie SPL-Testsuite der FA-SPL (siehe Abbildung 8.2) angewendet, lassen sich 6 Testfallpaare bilden, deren Testfälle auf derselben Menge von Produktkonfigurationen verwendbar sind und sich somit zum Ersetzen eignen. Da sich diese 6 Paare aus 4 Testfällen ($tc14$, $tc21$, $tc23$ und $tc29$) bilden lassen, können nur 2 Paare ersetzt werden. Für das Testfallpaar, bestehend aus $tc21$ und $tc23$, lässt sich der Testfall $tc21x23$ als Ersatz generieren (siehe Abbildung 8.3a). Für das verbleibende Testfallpaar $tc14$ und $tc19$ ließ sich jedoch mit dem in dieser Arbeit eingesetzten Testfallgenerator kein Testfall generieren, der auf derselben Menge von Produktkonfigurationen wie die zwei zu ersetzenden Testfälle verwendbar war. Die Gründe und eine mögliche Lösung dafür werden in Kapitel 9 vorgestellt.

Durch das ersetzte Testfallpaar ergibt sich eine Reduktion in der Anzahl der Testfälle von 9 auf 8 und eine Reduktion in der Anzahl der minimal benötigten Testfallausführungen von 46 auf 38 (siehe Abbildung 8.3b). Die stark reduzierte Anzahl der minimal benötigten Testfallausführungen zum Erreichen einer vollständigen SPL-Abdeckung lässt sich darauf zurückführen, dass die beiden Testfälle des Testfallpaares auf insgesamt 16 Produktvarianten der FA-SPL zur Ausführung eingeplant waren.

In der vollständigen, redundanzfreien SPL-Testsuite der BCS-SPL für das Überdeckungskriterium *all-transitions* mit 44 Testfällen konnten 24 Testfallpaare identifiziert werden, deren zwei Testfälle auf derselben Produktmenge verwendbar sind. Von diesen 24 Testfallpaaren konnte für 11 ein als Ersatz geeigneter Testfall generiert werden. Durch anschließendes Ersetzen konnte die Anzahl der Testfälle von 44 auf 33 gesenkt werden und die Anzahl der Testfallausführungen von minimal 574 auf minimal 414 gesenkt werden.

8.3.3 *Diskussion*

Im Gegensatz zu bisherigen Testsuite-Reduktionstechniken berücksichtigt dieser Algorithmus, dass im SPL-Kontext Testfälle existieren können, die zur Ausführung auf mehreren Produktvarianten eingeplant sind. Dadurch lässt sich mit diesem Algorithmus die Anzahl der in einer SPL-Testsuite enthaltenen Testfälle reduzieren, ohne dabei die zuvor erreichte Abdeckung der Testziele auf einer einzigen Produktvariante zu verringern. Aufgrund der variantenübergreifenden Verwendung von Testfällen kann selbst eine geringe Reduktion in der Testfallanzahl eine starke Reduktion in der Anzahl der Testfallausführungen zur Folge haben. Der Algorithmus trägt dadurch zur Kostensenkung im Testprozess einer SPL bei.

Die Forderung, dass nur Testfallpaare (tc', tc'') betrachtet werden, deren zwei Testfälle auf denselben Produktvarianten $(PC_{tc'} = PC_{tc''})$ verwendbar sind, basiert auf der Erfahrung, dass ein neu erstellter Testfall $tc_{substitute}$ häufig nur unter dieser Forderung auf mindestens denselben Produktenvarianten wie die zu ersetzenden Testfälle verwendbar ist ($PC_{tc_{subtitute}} \supseteq PC_{tc'} = PC_{tc''}$). Wenn ein solcher Testfall $tc_{substitute}$ erstellt werden kann, sinkt sowohl die Anzahl der enthaltenen Testfälle als auch die Anzahl der Testfallausführungen. Sollte der Fall eintreten, dass

tc#	Features				Transitionspfad
	B	C	D	G	
14	-	-	-	-	*a c f i u b f g g j k k k k k k k k k k l m*
16	-	-	1	0	*a d f g g g g j k k k k k k k k k k k l o*
17	1	-	0	1	*a c f i u e f g g h j k k k k k k k k k k l p*
18	-	-	0	1	*a c f i u e f g g g g g g j k k k k k k k k k k l p*
19	-	0	-	-	*a c f g j k k k k k k k k k k k l n q*
20	-	1	-	-	*a c f g j k k k k k k k k k k k l n r*
21	-	-	-	-	*a b s*
23	-	-	-	-	*a c b f g i t*
29	-	-	-	-	*a c v w w w w w w w w w w x*
21x23	-	-	-	-	*a c f i u b f g i t u b s*

(a) Transitionspfade

PC	Features				Testfälle (tc#)									
	B	C	D	G	14	16	17	18	19	20	21	23	29	21 23
fa1	0	0	0	0	☑				☑		☑	☑	☑	☑
fa2	0	0	0	1	☑			☑	☑		☑	☑	☑	☑
fa3	0	0	1	0	☑	☑			☑		☑	☑	☑	☑
fa4	0	1	0	0	☑					☑	☑	☑	☑	☑
fa5	0	1	0	1	☑			☑		☑	☑	☑	☑	☑
fa6	0	1	1	0	☑	☑				☑	☑	☑	☑	☑
fa7	1	0	0	1	☑		☑	☐	☑		☑	☑	☑	☑
fa8	1	1	0	1	☑		☑	☐		☑	☑	☑	☑	☑

□ = entfernt ☐ = verwendbar ☑ = eingeplant

(b) Produktspez. Testsuites (minimal)

Abbildung 8.3: Reduzierte FA-SPL-Testsuite (*all-transitions*) durch Ersetzen geeigneter Testfallpaare

Testfall $tc_{substitute}$ mit $PC_{tc_{subtitute}} \subset PC_{tc'} = PC_{tc''}$ generiert wird, ist es wahrscheinlich, dass dieser nicht zum Ersetzen des Testfallpaares verwendet werden kann. Bei der Hinzunahme des Testfalls $tc_{substitute}$ mit $PC_{tc_{subtitute}} \neq$ zur SPL-Testsuite steigt die Anzahl der Testfälle, da das Testfallpaar nicht entfernt werden kann ($TS_{SPL} \not\preceq (TS_{SPL} \setminus \{tc', tc''\}) \cup \{tc_{substitute}\}$). Dennoch schließt das nicht aus, dass die Anzahl der zwingend benötigten Ausführungen sinkt. In weiteren Schritten kann die Anzahl der enthaltenen Testfälle gegebenenfalls wieder gesenkt werden, wenn durch das Hinzufügen des Testfalls $tc_{subtitute}$ Redundanz entsteht.

Beispiel

In Abbildung 8.4a konnte für die zwei Testfälle $tc14$ und $tc29$, für die $PC_{tc14} = PC_{tc29}$ gilt, ein Testfall $tc14x29a$ mit $PC_{tc14x29a} \subset PC_{tc14}$ generiert werden. Dadurch stieg die Anzahl der in der SPL-Testsuite enthaltenen Testfälle an, die Anzahl der minimal benötigten Testfallausführung sank hingegen. Durch das Generieren eines weiteren Testfalls $tc14x29b$ (siehe Abbildung 8.4b) konnte die Anzahl der benötigten Testfallausführungen weiter gesenkt werden und die Anzahl der enthaltenen Testfälle wieder auf den ursprünglichen Wert zurückgesetzt werden, da die ursprünglichen Testfälle $tc14$ und $tc19$ entfernt werden konnten.

PC	Testfälle (tc#)									
	14	16	17	18	19	20	21	23	29	14 29 a
fa1	☐				☑		☑	☑	☐	☑
fa2	☐			☑	☑		☑	☑	☐	☑
fa3	☐	☑			☑		☑	☑	☐	☑
fa4	☐					☑	☑	☑	☐	☑
fa5	☑			☑		☑	☑	☑	☑	
fa6	☑	☑				☑	☑	☑	☑	
fa7	☑		☑	☐	☑		☑	☑	☑	
fa8	☑		☑	☐		☑	☑	☑	☑	

☐ = entfernt ☐ = verwendbar ☑ = eingeplant

(a) Anstiegt der Testfallanzahl

PC	Testfälle (tc#)										
	14	16	17	18	19	20	21	23	29	14 29 a	14 29 b
fa1	☑				☑		☑	☑	☑	☑	
fa2	☑			☑	☑		☑	☑	☑	☑	
fa3	☑	☑			☑		☑	☑	☑	☑	
fa4	☑					☑	☑	☑	☑	☑	
fa5	☑			☑		☑	☑	☑	☑		☑
fa6	☑	☑				☑	☑	☑	☑		☑
fa7	☑		☑	☐	☑		☑	☑	☑		☑
fa8	☑		☑	☐		☑	☑	☑	☑		☑

☐ = entfernt ☐ = verwendbar ☑ = eingeplant

(b) Beibehalt der Testfallanzahl

Abbildung 8.4: Reduktion der eingeplanten Testfallausführungen

Bei der Reduktion durch Ersetzen ist zu beachten, dass je mehr Testziele in $TG_{uncovered}$ enthalten sind und somit von $tc_{substitute}$ zwingend abgedeckt werden müssen, desto kleiner fällt die Produktmenge $PC_{tc_{substitute}}$, auf der sich der neu erstellte Testfall $tc_{substitute}$ anwenden lässt, in der Regel aus (abhängig vom 150%-Testmodell). Diese Annahme ist naheliegend, da zum Abdecken von vielen Testzielen auch meistens ein längerer Testfall benötigt wird. Das kann zur Folge haben, dass mehr Transitionen mit Feature-Annotation traversiert werden müssen, die sich wiederum bei der Analyse des Testfalls $tc_{substitute}$ auf die Größe von $PC_{tc_{substitute}}$ auswirken. Soll folglich ein Testfall mit $PC_{tc_{substitute}} \supseteq PC_{tc'} = PC_{tc''}$ generiert werden, sollte, ausgehend von obiger Annahme, bei der Erstellung eines Testfalls darauf geachtet werden, dass von diesem nur die Abdeckung der zwingend notwendigen Testziele gefordert wird. Des Weiteren ist im Allgemeinen davon auszugehen, dass ein Testfall schneller generiert werden kann, wenn die Menge der abzudeckenden Testziele $TG_{uncovered}$ klein ausfällt, da die Suche im Zustandsraum weniger aufwändig ist. Wenn obige Annahme getroffen wird, ist es außerdem naheliegend bei der Generierung eines neuen Testfalls als Ersatz für ein Testfallpaar, die Testfallpaare zu bevorzugen, bei denen $TG_{uncovered}$ im Vergleich zu den anderen Testfallpaaren kleiner ausfällt.

Die zu erwartende Reduktion in der Anzahl der Testfallausführungen ist davon abhängig, auf wie vielen Produktvarianten die beiden zu ersetzenden Testfälle zur Ausführung eingeplant sind. Folglich lässt sich Algorithmus 8.2 in seiner Effizienz steigern, wenn Paare beim Ersetzungsvorgang bevorzugt werden, welche auf vielen Produktvarianten zur Ausführung eingeplant sind.

Eine weitere Effizienzsteigerung könnte in Algorithmus 8.2 erreicht werden, indem in Zeile 8 nicht die Forderung „$PC_{tc_{substitute}} \supseteq PC_{tc'}$“ verwendet wird, sondern stattdessen „$TS_{SPL} \preceq (TS_{SPL} \setminus \{tc', tc''\}) \cup \{tc_{substitute}\}$“. Aufgrund der schwächeren Forderung wäre dann auch ein Testfall $tc_{substitute}$ als Ersatz zulässig, der nicht zwingend auf den gleichen oder noch weiteren Produktkonfigurationen wie das Testfallpaar verwendbar wäre.

PC	Testfälle (tc#)																
	14	16	17	18	19	20	21	23	29	14 29 c	14 29 d	14 29 e	14 29 f	14 29 g	14 29 h	14 29 i	14 29 j
fa1	☑				☑		☑	☑	☑	☑							
fa2	☑			☑	☑		☑	☑	☑		☑						
fa3	☑	☑			☑		☑	☑	☑			☑					
fa4	☑					☑	☑	☑	☑				☑				
fa5	☑			☑		☑	☑	☑	☑					☑			
fa6	☑	☑				☑	☑	☑	☑						☑		
fa7	☑		☑	☐	☑		☑	☑	☑							☑	
fa8	☑		☑	☐		☑	☑	☑	☑								☑

☐ = entfernt ☐ = verwendbar ☑ = eingeplant

Abbildung 8.5: Ersetzen von Testfällen ohne 150%-Testmodell

Neben dem Erhalt der produktspezifischen Abdeckung ist ein weiterer Vorteil von Algorithmus 8.2 gegenüber bisherigen Testsuite-Reduktionsansätzen, dass die als Ersatz verwendeten Testfälle aus einem 150%-Testmodell generiert werden. Würde kein 150%-Testmodell verwendet werden, wäre es möglich, dass für ein Testfallpaar mehrere produktspezifische Testfälle, die nicht variantenübergreifend verwendbar sind, erstellt werden müssen. Dadurch würde die Anzahl der zu generierenden Testfälle zunehmen, was zu steigenden Kosten bei der Testfallgenerierung und Testfallwartung führen würde.

Beispiel

Wird die in Abbildung 8.2a dargestellte redundanzfreie SPL-Testsuite der FA-SPL nicht mit Algorithmus 8.2 reduziert, sondern mit einem Algorithmus, der keine variantenübergreifend verwendbaren Testfälle generieren kann, müssten beim Ersetzen des Testfallpaares $tc14$ und $tc29$, die beide auf allen 8 Produktvarianten verwendet werden können, 8 produktspezifische Testfälle erstellt werden (siehe Abbildung 8.5), um den Erhalt der produktspezifischen Abdeckung zu garantieren. Dadurch würde die Anzahl der in der SPL-Testsuite enthaltenen Testfälle um 6 ansteigen, was sich negativ auf die Kosten für die Testfallgenerierung und Testfallwartung auswirken würde.

8.4 ERSETZEN UNTER BERÜCKSICHTIGUNG DER GESAMTLÄNGE

Das Ersetzen eines beliebigen Testfallpaares in der SPL-Testsuite durch einen neu erstellten Testfall führt immer zu einer Reduktion in der Anzahl der in dieser SPL-Testsuite enthaltenen Testfälle. Nicht zwingend ist jedoch, dass solch eine Ersetzung eine Reduktion der Gesamtlänge der SPL-Testsuite zur Folge hat. Die Gesamtlänge einer Testsuite lässt sich bestimmen, indem die Länge eines jeden in der Testsuite enthaltenen Testfalls aufsummiert wird. Die Länge eines Testfalls ergibt sich dabei aus der Länge seiner Transitionssequenz. In [HCL+03] wurde darauf hingewiesen, dass neben der Anzahl der in einer Testsuite enthaltenen Testfälle auch die Gesamtlänge einer Testsuite hohe Kosten im Testprozess verursachen kann, weshalb diese gegebenenfalls auch reduziert werden sollte.

Abhängig davon, welches Testfallpaar zum Ersetzen gewählt wird, kann die Gesamtlänge einer Testsuite nach dem Ersetzungsschritt sogar steigen statt sinken. Aus diesem Grund wird im Folgenden ein heuristischer Ansatz vorgestellt, mit dem sich Testfallpaare dahingehend bewerten lassen, wie hoch deren Beitrag zur Reduktion der Gesamtlänge der Testsuite wäre, wenn dieses Paar durch einen neu erstellten Testfall ersetzt werden würde. Durch diese Bewertung ist es möglich Testfallpaare gezielt statt zufällig auszuwählen, die eine hohe Reduktion der Gesamtlänge der Testsuite versprechen, was zur Folge hat, dass sich eine Reduktion der Gesamtlänge der Testsuite effizienter und kostengünstiger erreichen lässt. Mit dieser Heuristik ist es möglich, nur mit den zwei zu ersetzenden Testfällen als Informationsquelle eine Aussage über die zu erwartende Längenreduktion bei deren Ersetzung durch einen neuen Testfall zu tätigen, ohne diesen dafür generieren zu müssen.

Da es sich um eine Heuristik handelt, kann nicht garantiert werden, dass der durch die Heuristik ermittelte Wert immer im gleichen Verhältnis zum tatsächlich erreichten Reduktionswert steht.

Beispiel

Für die FA-SPL wurde mit Algorithmus 7.1 eine vollständige SPL-Testsuite generiert, die eine *all-transition-pairs* Abdeckung auf allen 8 Produktvarianten erreicht (siehe Abbildung A.2). Diese SPL-Testsuite, ursprünglich bestehend aus 100 Testfällen, konnte durch das Entfernen redundanter Testfälle auf 43 Testfälle reduziert werden (siehe Abbildung A.3). Im Folgenden soll diese SPL-Testsuite durch das Ersetzen von Testfällen weiter in der Anzahl der Testfälle reduziert werden, wobei die Länge eines als Ersatz generierten Testfalls kürzer ausfallen soll als die Gesamtlänge der zwei zu ersetzenden Testfälle.

8.4.1 *Bewertungsfunktion für Ähnlichkeit*

Der heuristische Ansatz basiert auf der Beobachtung, dass ein neu generierter Testfall $tc_{substitute}$, der ein Testfallpaar (tc',tc'') ersetzen soll, häufig in Teilen dieselben Transitionen im Testmodell traversiert wie die Testfälle tc' und tc''. Weisen die zwei zu ersetzenden Testfälle tc' und tc'' selbst in Teilen dieselben Transitionen auf, enthält der als Ersatz generierte Testfall $tc_{substitute}$ diese Transitionen häufig nur einmal statt zweimal. Folglich kann eine hohe Reduktion der Länge erwartet werden, wenn die beiden zu ersetzenden Testfälle tc' und tc'' viele identische Transitionsabschnitte im jeweiligen Pfad aufweisen.

In der verwendeten Heuristik wird nach langen identischen Transitionsabschnitten gesucht, da angenommen wird, dass diese eher von einem neu erstellten Testfall aufgenommen werden als Abschnitte die kurz sind und gegebenenfalls eine unterschiedliche Reihenfolge zueinander aufweisen. Da sich die identischen Transitionsabschnitte in den Testfällen an jeweils unterschiedlichen Positionen befinden können, ist deren Identifizierung schwierig. Zur effizienten Identifikation solch langer Abschnitte eignet sich der Smith-Waterman-Algorithmus [SW81], der ursprünglich dazu entwickelt wurde in der Bioinformatik DNA-Sequenzen zueinander auszurichten. Der Sequenz-Alignierungs-Algorithmus richtet durch

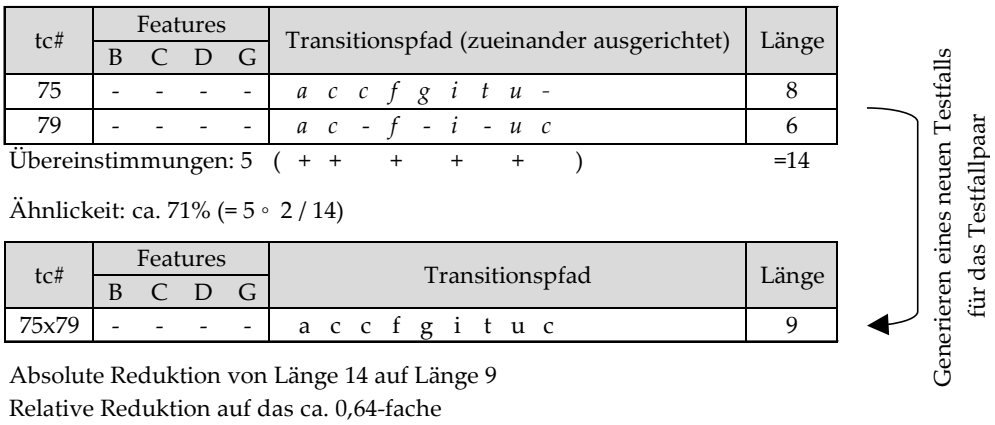


Abbildung 8.6: Beispiel für Ersetzungsschritt

Einfügen von Lücken die Transitionen zweier Testfälle so zueinander aus, dass lange identische Abschnitte entstehen. Dadurch, dass nur Lücken eingefügt werden, bleibt die Reihenfolge der Transitionen untereinander bestehen.

Beispiel

In Abbildung 8.6 sind zwei Testfälle *tc*75 und *tc*79 aus der redundanzfreien SPL-Testsuite (siehe Abbildung A.3) dargestellt. Diese beiden Testfälle sollen durch einen neu generierten Testfall ersetzt werden. Zu diesem Zweck wurden die beiden Testfälle zueinander ausgerichtet. Die eingefügten Lücken sind durch „-“ dargestellt. Ein möglicher Testfall, der sich als Ersatz für *tc*75 und *tc*79 erstellen lässt ist *tc*75*x*79. Es ist zu erkennen, dass dieser neu generierte Testfall viele der Transition aufweist, wie die zu ersetzenden Testfälle, aber nur einmal statt zweimal.

Zur Berechnung des Reduktionspotential eines Testfallpaares wird im Folgenden die Funktion *similarity* verwendet, welche die Ähnlichkeit zweier Testfälle zueinander berechnet. Dafür werden zuerst die beiden Testfälle tc' und tc'' zueinander ausgerichtet. Anschließend wird mit der Funktion *match* die Gesamtlänge der identischen Abschnitte bestimmt und in Beziehung zu der Gesamtlänge der beiden nicht ausgerichteten Testfälle tc' und tc'' gesetzt.

$$similarity(tc', tc'') = \frac{match(tc', tc'') \cdot 2}{length(tc') + length(tc'')}$$

Die durch den Ersetzungsschritt erreichte relative Reduktion der Länge lässt sich durch folgende Funktion berechnen:

$$reduction(tc', tc'', tc_{substitute}) = \frac{length(tc_{substitute})}{length(tc') + length(tc'')}$$

Beispiel

In Abbildung 8.6 besitzen die identischen Abschnitte von *tc*75 und *tc*79 die Gesamtlänge 5. Die Gesamtlänge der beiden nicht ausgerichteten Testfälle beträgt

Algorithmus 8.3 Ersetzen von Testfallpaaren mittels Sequenz-Alignierung

1: **procedure** REPLACETESTCASESWITHSEQALI(TS_{SPL},vtm,TG)
2: **for all** $\{(tc', tc'') \in TS_{SPL} \mid tc' \neq tc''\}$ **do**
3: $PC_{tc'} \leftarrow \{pc \in PC \mid (\varphi(tc'))(pc) = true\}$
4: $PC_{tc''} \leftarrow \{pc \in PC \mid (\varphi(tc''))(pc) = true\}$
5: **if** $PC_{tc'} = PC_{tc''}$ **then**
6: $pairs_{sorted} \leftarrow pairs_{sorted}$ sort $((tc', tc''), similarity(tc', tc''))$
7: **end if**
8: **end for**
9: **while** $pairs_{sorted} \neq \emptyset$ **do**
10: $pair = (tc', tc'') \leftarrow getAndRemovePairWithHighestSimilarity(pairs_{sorted})$
11: $TG_{uncovered} \leftarrow \{tg \in TG \mid \exists pc \in PC :$
$\nexists\, tc' \in TS_{SPL} \setminus \{tc', tc''\}, covers(tc', tg), valid(tc', bind(vtm, pc))\}$
12: $tc_{substitute} \leftarrow generate(TG_{uncovered}, PC_{tc'}, vtm)$
13: **if** $\exists pc_{bound} \in PC_{tc'} : valid(tc_{substitute}, bind(vtm, pc_{bound}))$ **then**
14: $PC_{tc_{substitute}} \leftarrow \{pc \in PC \mid (\varphi(tc_{substitute}))(pc) = true\}$
15: **if** $PC_{tc_{substitute}} \supseteq PC_{tc'}$ **and** $reduction(tc', tc'', tc_{substitute}) \leq 1$ **then**
16: $TS_{SPL} \leftarrow TS_{SPL} \setminus \{tc', tc''\}$
17: $TS_{SPL} \leftarrow TS_{SPL} \cup \{tc_{substitute}\}$
18: $pairs_{sorted} \leftarrow \{pair \in pairs_{sorted} \mid tc', tc'' \notin pair\}$
19: **end if**
20: **end if**
21: **end while**
22: **end procedure**

14. Wird mit der Funktion *similarity* die Ähnlichkeit dieses Testfallpaars ermittelt, beträgt diese ca. 71%. Wird dieses Testfallpaar durch den neu erstellten Testfall $tc75x79$ ersetzt, wird eine Reduktion in der Länge auf das ca. 0.64-fache erreicht.

Der ermittelte Wert für die Ähnlichkeit eines Testfallpaares ist auch vom verwendeten Modell abhängig, aus dem die Testfälle generiert wurden. So können die Transitionssequenz in einem semantisch äquivalenten Modell kürzer oder länger ausfallen, was zu unterschiedlichen Werten bei der Berechnung der Ähnlichkeit führt.

8.4.2 *Algorithmus* REPLACETESTCASESWITHSEQALI

Im Folgenden wird Algorithmus 8.3 beschrieben, welcher die vorgestellte Heuristik verwendet, um eine SPL-Testsuite sowohl in der Anzahl der enthaltenen Testfälle als auch in ihrer Gesamtlänge zu reduzieren.

Zu Beginn (Zeile 2 bis 8) wird für jedes Testfallpaar (tc', tc'') der SPL-Testsuite TS_{SPL}, bei dem die Testfälle tc' und tc'' auf derselben Menge von Produktkonfigurationen verwendbar sind, der Ähnlichkeitswert durch die Funktion *similarity* bestimmt (Zeile 6). Der ermittelte Ähnlichkeitswert wird verwendet, um dieses

Paar in die Menge $pairs_{sorted}$ dem Ähnlichkeitswert nach absteigend einzusortieren. Anschließend wird damit begonnen die in $pairs_{sorted}$ enthaltenen Testfallpaare durch einen neu erstellten Testfall zu ersetzen bis keine Testfallpaare mehr in $pairs_{sorted}$ enthalten sind (Zeile 9 bis 21). Dafür wird in Zeile 10 immer das Testfallpaar mit dem höchsten Ähnlichkeitswert aus $pairs_{sorted}$ gewählt. Lässt sich dann für dieses Paar ein Testfall $tc_{substitute}$ generieren (Zeile 12), welcher zum einen das Paar ersetzen kann ohne die produktspezifische Abdeckung einer beliebigen Produktvariante der SPL zu senken und zum anderen eine relative Reduktion in der Länge herbeiführt (Zeile 15), wird das Testfallpaar durch den Testfall $tc_{substitute}$ ersetzt. Nach dem Ersetzungsschritt werden noch alle Testfallpaare aus $pairs_{sorted}$ entfernt, die tc' oder tc'' beinhalten. Sollte hingegen kein zur Reduktion geeigneter Testfall $tc_{substitute}$ generiert werden können (Zeile 13 bzw. 15) wird in Zeile 9 mit dem nächsten Testfallpaar in $pairs_{sorted}$ fortgefahren.

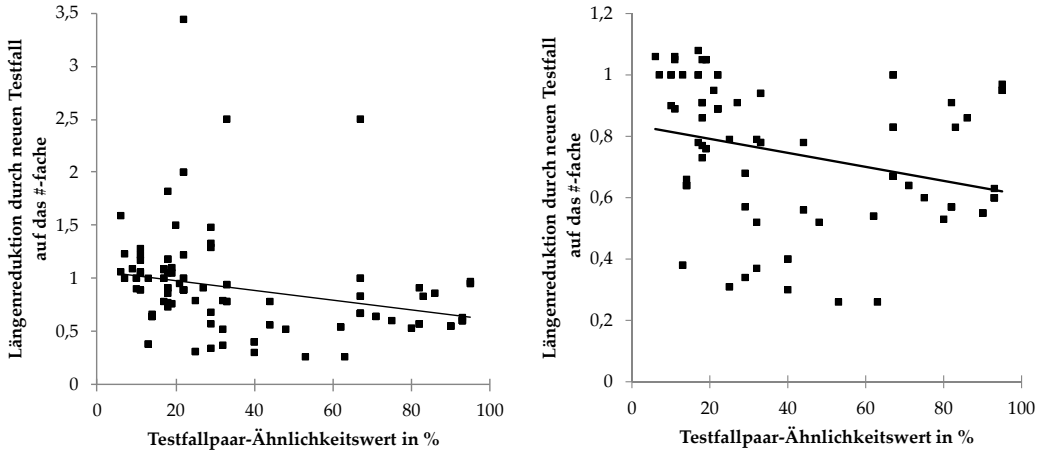
Algorithmus 8.3 kann der Klasse der Greedy-Algorithmen zugeordnet werden, da dieser immer das Testfallpaar mit dem höchsten Ähnlichkeitswert aus $pairs_{sorted}$ entnimmt (Zeile 10), dafür versucht einen Testfall zu generieren und mit diesem Testfall (Zeile 12), falls erstellbar, das Testfallpaar ersetzt (wenn die Bedingung in Zeile 15 erfüllt sind). Greedy-Algorithmen sind meistens schneller als Algorithmen, die das Problem optimal lösen. So besitzt auch Algorithmus 8.3 Verbesserungspotential. Beispielsweise könnten, statt ein Testfallpaar sofort zu ersetzen, erst ein paar Testfälle aus geeigneten Testfallpaaren generiert werden, von denen dann das Testfallpaar ersetzt wird, dessen neu generierter Testfall im Vergleich mit den anderen die höchste absolute Reduktion der Gesamtlänge der SPL-Testsuite aufweist. Des Weiteren könnte der Algorithmus verbessert werden, indem ein für Ersatzzwecke neu generierter Testfall gleich dazu verwendet wird, um mit den verbleibenden Testfällen in $pairs_{sorted}$ neue Paare zu bilden, die dann in weiteren Ersetzungsschritten berücksichtigt werden.

8.4.3 *Evaluation*

Im Folgenden soll die Güte der Heuristik bestimmt werden. Dafür werden die für Testfallpaare ermittelten Ähnlichkeitswerte mit der tatsächlichen erreichbaren Reduktion der Länge (relativ gesehen) bei Ersatz jener Testfallpaare durch einen neu erstellten Testfall in Beziehung gesetzt. Dabei wird zwecks besserer Vergleichbarkeit davon ausgegangen, dass immer derselbe Testfall als Ersatz für ein Testfallpaar generiert wird.

Zur Evaluation wurden die 43 Testfälle der redundanzfreien und vollständigen SPL-Testsuite benutzt, die eine *all-transition-pairs* Abdeckung auf allen 8 Produktvarianten der FA-SPL erreicht (siehe Abbildung A.3). Aus diesen 43 Testfällen lassen sich 143 Testfallpaare bilden, deren zwei Testfälle für dieselben Produktvarianten verwendbar sind. Von diesen 143 Testfallpaaren konnte aber nur für 117 ein neuer Testfall als Ersatz erstellt werden, welcher, wie in Zeile 15 des Algorithmus 8.3 gefordert, für dieselben Produktvarianten verwendbar ist wie die zwei zu ersetzenden Testfälle.

In Abbildung 8.7a ist für jedes dieser 117 Testfallpaare der errechnete Ähnlichkeitswert in Beziehung zu der erreichten Reduktion der Länge (relativ gesehen)



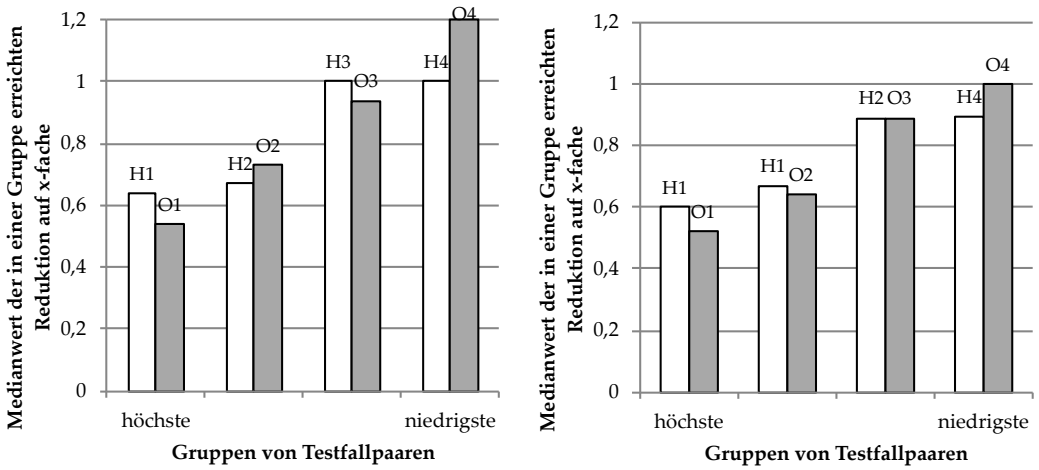
(a) Alle 117 Ersetzungen

(b) 94 Ersetzungen mit größter relativer Reduktion

Abbildung 8.7: Beziehung zwischen Ähnlichkeitswert und erreichter rel. Längenreduktion

gesetzt. Anhand der Trendlinie ist erkennbar, dass ein großer Ähnlichkeitswert häufig zu einer großen relativen Reduktion der Länge führt. So weisen Testfallpaare mit einem Ähnlichkeitswert größer als 60% häufig eine Reduktion auf, wohingegen Testfallpaare mit einem Ähnlichkeitswert unter 40% in Teilen sogar eine negative Reduktion aufweisen bzw. in der Länge zunehmen. Außerdem ist zu erkennen, dass für einige wenige Testfallpaare ein Testfall generiert wurde, dessen Länge nicht reduziert wurde, sondern stattdessen um mehr als 50% anstieg im Vergleich zu der Gesamtlänge der beiden im Testfallpaar enthaltenen Testfälle. Dies ist auf den Testfallgenerator zurückzuführen. Wenn der verwendete Testfallgenerator nicht in der Lage ist (beispielsweise aufgrund eines zu großen Zustandsraums) einen kurzen Testfall zu finden, kann dieser als Alternative einen längeren Testfall generieren, um die geforderten Testziele auf den geforderten Produktvarianten abzudecken. Da die Länge eines Testfalls nach oben nicht begrenzt sein muss, können sehr lange Testfälle entstehen, die sich nicht zur Reduktion der Gesamtlänge einer Testsuite eignen. Da dieser Effekt bei jedem beliebigen Testfallpaar (unabhängig von deren Ähnlichkeitswert) auftreten kann und dazu führen würde, dass der generierte Testfall nicht verwendet wird, werden in Abbildung 8.7b nur die Ähnlichkeitswerte derjenigen Testfallpaare in Beziehung zur erreichten Reduktion gesetzt, deren Reduktionswert zu den höchsten 80% der 117 Testfallpaare gehört. In dieser Abbildung, in der nur die Werte von 94 Testfallpaaren in Beziehung gesetzt wurden, ist auch ohne die sehr starken Ausreißer anhand der Trendlinie erkennbar, dass ein hoher Ähnlichkeitswert mit einer hohen relativen Reduktion korreliert.

Eine alternative Darstellungsform der Beziehung zwischen Ähnlichkeitswert eines Testfallpaares und der erreichten relativen Längenreduktion durch Ersetzen ist in Abbildung 8.8a für alle 117 Testfallpaare und in Abbildung 8.8b für die ausgewählten 97 Testfallpaare dargestellt. In beiden Abbildungen wurden die



(a) Alle 117 Ersetzungen

(b) 94 Ersetzungen mit größter relativer Reduktion

Abbildung 8.8: Erreichbare rel. Reduktion (Median) bei Auswahl der Testfälle aus Gruppen

Testfallpaare auf vier Gruppen, grafisch durch Balken dargestellt, verteilt. Jede Gruppe besteht dabei aus nahezu derselben Anzahl an Testfällen (jeweils 29 bzw. 24). Die Gruppierung erfolgt, indem die Testfallpaare zuvor sortiert wurden

- nach ihrer Ähnlichkeit (weiße Balken) oder
- nach der erreichten relativen Reduktion (graue Balken).

Für jede Gruppe ist der Medianwert für die erreichte relative Längenreduktion beim Ersetzen der enthaltenen Testfallpaare auf der y-Achse angegeben. Der Medianwert sagt aus, dass die über dem Median liegende Hälfte der in dieser Gruppe enthaltenen Testfallpaare eine höhere relative Längenreduktion als der Medianwert aufweist, wohingegen die andere Hälfte einen niedrigeren Wert aufweist.

Beispielsweise enthält in Abbildung 8.8a die Gruppe $H1$ genau die 29 Testfallpaare aus den 117 möglichen, die den höchsten Ähnlichkeitswert besitzen. Diese Gruppe besitzt als Medianwert ungefähr den Wert 0.65. Folglich konnte für 14 Testfallpaare in dieser Gruppe ein Testfall als Ersatz generiert werden, der kürzer als das 0.65-fache ist bezogen auf die Gesamtlänge der Testfälle des zu ersetzenden Testfallpaares.

Um die Güte der Heuristik zu bewerten, können die Gruppen $H1$, $H2$, $H3$ und $H4$ mit den Gruppen $O1$, $O2$, $O3$ und $O4$ verglichen werden. Dabei stellen die Gruppen $O1$, $O2$, $O3$ und $O4$ die optimale Zuweisung der Testfallpaare auf die Gruppen dar bzgl. real erreichbarer relativer Längenreduktion. So befinden sich in der Gruppe $O1$ die Testfallpaare für die ein Testfall generiert wurde, der im

Vergleich zu den anderen eine relative Längenreduktion aufweist, die zu den besten 25% gehört. Folglich ist der errechnete Medianwert dieser Gruppe (ungefähr 0.5) das bestmögliche Ergebnis.

Es kann festgestellt werden, dass die Medianwerte der Gruppen sowohl bei der Heuristik als auch bei der optimalen Zuweisung eine ähnliche Tendenz aufweisen, die auf eine Korrelation zwischen Ähnlichkeitswert und erreichbarer Längenreduktion schließen lässt. Wenn zum Ersetzen von Testfallpaaren nur Testfallpaare aus der Gruppe $H1$ verwendet werden, sollte die relative Längenreduktion in der Regel höher ausfallen, als wenn Testfallpaare zufällig aus einer der vier Gruppen gewählt werden. Folglich sollte die Testfallpaar-Auswahl mittels Heuristik der zufälligen Testfallpaar-Auswahl vorgezogen werden. Da bei der Verwendung der Heuristik keine Testfälle generiert werden müssen, um festzustellen, ob diese zur Längenreduktion geeignet sind, ist dieser heuristische Ansatz kostengünstig. So dauert die Testfallgenerierung mit Azmun auf einem PC mit *i*7-2600 CPU (3,7 Ghz) je nach Einstellung im Durchschnitt zwischen 30 und 120 Sekunden, wohingegen der Ähnlichkeitswert in weniger als einer Sekunde berechnet werden kann.

In Abbildung 8.9 wurden 10 Testfallpaare (tc',tc'') aus der in Abbildung A.3 dargestellten SPL-Testsuite jeweils durch einen neuen Testfall $tc_{substitute}$ ersetzt. Zu Vergleichszwecken wurden die Paare in Abbildung 8.9a zufällig ausgewählt, wohingegen die Paare in Abbildung 8.9b mittels der vorgestellten Heuristik bestimmt wurden und somit einen hohen Ähnlichkeitswert aufweisen. In den beiden Abbildung ist für jedes Testfallpaar die Gesamtlänge der beiden zu ersetzenden Testfälle und die Länge des neu erstellten Testfalls $tc_{substitute}$ dargestellt. Des Weiteren wird für jeden neu erstellten Testfall angegeben, auf wievielen Produktvarianten dieser verwendbar ist. Diese Anzahl hat Einfluss auf die Ausführungslänge einer SPL-Testsuite, die sich ergibt, wenn die Länge eines jeden enthaltenen Testfalls mit der Anzahl der Produktvarianten, auf der sich dieser Testfall anwenden lässt, multipliziert wird. Beispielsweise besitzt die SPL-Testsuite in Abbildung A.3 eine Ausführungslänge von 2629. Werden die 10 in Abbildung 8.9a dargestellten Testfallpaare durch den jeweiligen angegebenen Testfall ersetzt, steigt die Ausführungslänge von 2629 auf $2727 (2629 - 1202 + 1300)$, wohingegen sowohl die Anzahl der in dieser SPL-Testsuite enthaltenen Testfälle als auch die Anzahl der benötigten Testfallausführungen sinkt. Wenn jedoch die 10 in Abbildung 8.9b dargestellten Testfallpaare durch den jeweiligen angegebenen Testfall ersetzt werden, sinkt auch die Ausführungslänge von 2629 auf $2207 (2629 - 1206 + 784)$.

8.4.4 *Diskussion*

Anhand der Evaluation konnte gezeigt werden, dass sich durch Anwendung der Heuristik Testfallpaare in einer SPL-Testsuite identifizieren lassen, die im Durchschnitt eher zu einer Reduktion der Gesamtlänge einer Testsuite führen als zufällig ausgewählte Testfallpaare. Da die Heuristik als Informationsquelle ausschließlich existierende Testfallpaare benötigt, verursacht diese in ihrer An-

tc#	*ver-wend.*	*Länge*					
		tc′	*tc′′*	*gesamt*	*tc substitute*	*davor*	*danach*
1x32	8	3	6	9	8	72	64
2x37	8	3	15	18	16	144	128
3x7	4	3	3	6	4	24	16
4x5	2	3	3	6	5	12	10
6x18	4	3	31	34	54	136	216
11x36	8	27	16	43	47	344	376
14x22	4	24	20	44	25	176	100
29x33	2	13	6	19	7	38	14
31x35	8	8	3	11	20	88	160
34x38	8	6	15	21	27	168	216
				= 211	= 213	= 1202	= 1300

(a) Zufällig

tc#	*ver-wend.*	*Länge*					
		tc′	*tc′′*	*gesamt*	*tc substitute*	*davor*	*danach*
23x26	4	20	20	40	38	160	152
38x42	8	15	15	30	18	240	144
36x37	8	16	15	31	17	248	136
8x9	2	7	7	14	12	28	24
32x34	8	6	6	12	10	96	80
14x22	4	24	20	44	25	176	100
10x31	8	7	8	15	8	120	64
20x25	2	28	20	48	29	96	58
0x6	4	3	3	6	4	24	16
5x33	2	3	6	9	5	18	10
				= 249	= 166	= 1206	= 784

(b) Heuristik

Abbildung 8.9: Erreichte Längenreduktion je nach Auswahlstrategie

wendung kaum Kosten im Vergleich zur Testfallgenerierung, die benötigt wird, um den konkreten Reduktionswert zu ermitteln.

Um eine hohe Reduktion der Gesamtlänge der SPL-Testsuite zu erreichen sollten Testfallpaare ausgewählt werden, die eine hohe absolute Längenreduktion herbeiführen. So ist es offensichtlich, dass Testfallpaare mit einer niedrigen relativen, aber hohen absoluten Längenreduktion anderen Testfallpaaren mit einer hohen relativen, aber niedrigen absoluten Längenreduktion beim Ersetzen vorzuziehen sind, wenn die größtmögliche Reduktion der Gesamtlänge angestrebt wird.

Die Anwendbarkeit des heuristischen Ansatzes ist von der Länge der in einem Testfallpaar enthaltenen Testfälle abhängig. So muss beachtet werden, dass bei der Berechnung eines Ähnlichkeitswerts für ein Testfallpaar (tc', tc'') durch Sequenz-Alignierung eine Matrix der Größe $length(tc') \times length(tc'')$ berechnet werden muss.

Da Sequenz-Alignierung-Algorithmen für das Ausrichten von DNA-Sequenzen konzipiert wurden, die eindimensional sind, können auch nur Testfälle mit eindimensionalen Transitionssequenzen zueinander ausgerichtet werden. Dadurch dürfen, um den Ansatz verwenden zu können, keine Testmodelle zur Testfallgenerierung verwendet werden, die parallele Zustände enthalten, da diese zu Testfällen mit zweidimensionalen Transitionssequenzen führen können.

Auch wenn der Ansatz an einem Fallbeispiel aus dem SPL-Kontext vorgestellt wurde, ist dieser prinzipiell auch zur Reduktion von Testsuiten einzelner Softwaresysteme verwendbar.

9

REALISIERUNG

Die Herleitung eines neuen Testfalls aus einem 150%-Testmodell wird in den zuvor vorgestellten Algorithmen durch den Aufruf der Funktion *generate* ermöglicht. Diese Funktion lässt sich auf unterschiedliche Weisen realisieren. Im Folgenden wird auf die Realisierung dieser Funktion in dieser Arbeit eingegangen.

9.1 AZMUN

Zur Realisierung des in dieser Arbeit vorgestellten Ansatzes wurde Azmun [Has] verwendet. Azmun ist ein in der Programmiersprache Java entwickeltes Rahmenwerk für den modellbasierten Test. Dieses basiert auf dem Rahmenwerk der Open Service Gateway Initiative (OSGi) [OSG], welches den Austausch eingebundener Komponenten vereinfacht. Mittels der auf OSGi aufbauenden Modeling Workflow Engine (MWE) [SB] wird die Ausführungsreihenfolge der einzelnen Komponenten angegeben. Die in Azmun standardmäßig eingebundenen Komponenten unterstützen

- das Einlesen von in MagicDraw [NoM] erstellten UML-Zustandsautomaten,
- das Generieren von Testfällen aus Testmodellen für Einzel-Softwaresysteme durch den Model-Checker NuSMV [BCTT] und
- die Verwaltung der generierten Testfälle.

Zur Realisierung des vorgestellten Ansatzes mussten einige dieser im Rahmenwerk standardmäßig eingebundenen Komponenten durch neue, für diesen Ansatz ausgelegte Komponenten ersetzt werden. Übernommen wurden die Komponenten zum Einlesen von Daten eines UML-Zustandsautomaten aus einer XML-Datei in eine EMF-basierte Implementierung des UML-Metamodells der Model Development Tools (MDT) [Skrb]. Ebenfalls übernommen wurden die zur Verwaltung der generierten Testfälle eingesetzten Komponenten, die dafür das in Abbildung 9.1 dargestellte Metamodell verwenden. Ausgewechselt wurden hingegen die Komponenten für die Testfallgenerierung, da diese für einzelne Softwaresysteme und nicht für den SPL-Kontext ausgelegt waren. Außerdem war der standardmäßig eingebundene Model-Checker NuSMV [BCTT] nicht leistungsfähig genug, weshalb dieser gegen den geeigneteren Model-Checker SPIN [Holo3] ausgewechselt wurde. Um aus Azmun heraus auf SPIN zugreifen zu können, war die Entwicklung einer entsprechenden Schnittstelle nötig. Nähere Informationen dazu sind in Abschnitt 9.4 zu finden. Da SPIN nur in PROMELA (Process/Protocol Meta Language) [Com] beschriebene Modelle verarbeiten kann, musste eine

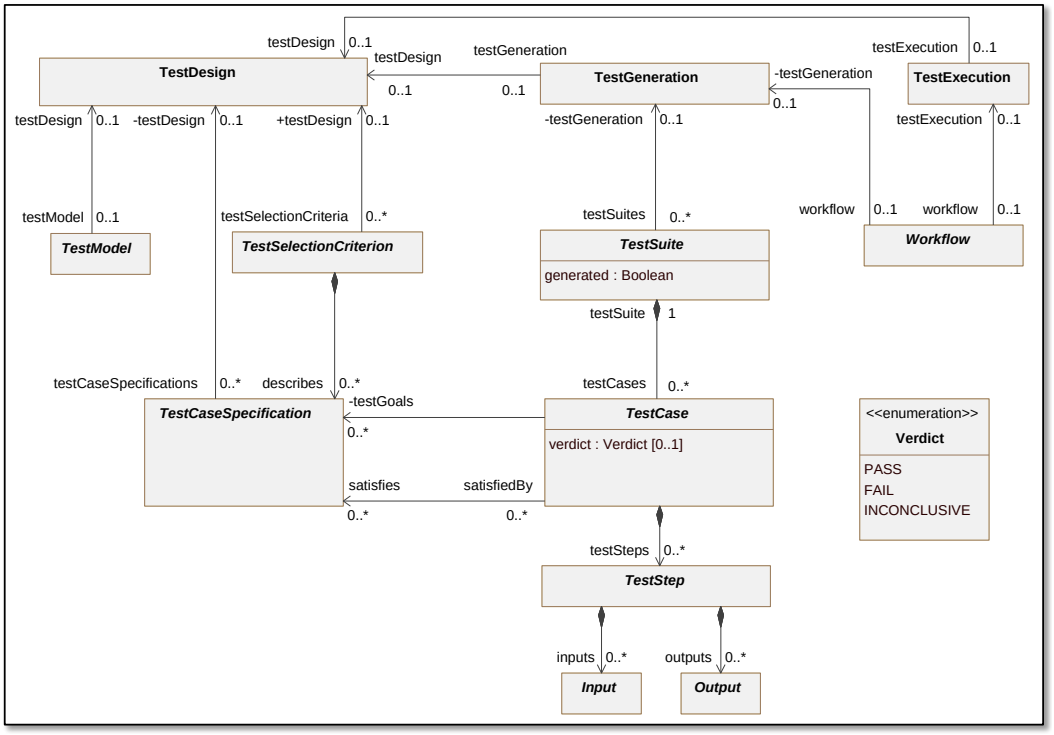


Abbildung 9.1: Metamodell zur Verwaltung der generierten Testfälle

Komponente entwickelt werden, welche die eingelesenen Daten des 150%-Testmodells in ein PROMELA-Programm transformiert. Zu diesem Zweck wurde XPand [Skra], eine Template-Sprache zum Generieren von Text aus beliebigen Modellen, eingesetzt. Nähere Informationen zum Aufbau des generierten PROMELA-Programms sind in Abschnitt 9.3 zu finden. Des Weiteren wurden Komponenten für den Quine-McCluskey-Algorithmus, die SPL-Testsuite-Reduktion und die Sequenz-Alignierung entwickelt, auf die im Folgenden aber nicht näher eingegangen wird.

Das Zusammenspiel der in Azmun für diese Arbeit verwendeten Komponenten ist in Abbildung 9.2 dargestellt. Zu Beginn wird das als XML-Datei vorliegende 150%-Testmodell von Komponente *A* in die auf EMF-basierende Implementierung des UML-Metamodells eingelesen. Komponente *B* generiert aus diesen Daten unter Verwendung von XPand ein PROMELA-Programm (siehe Abschnitt 9.3). Komponente *C* generiert eine LTL-Formel, welche die vom Testfall abzudeckenden Testziele beschreibt und einige Produktkonfigurationen der SPL vorgibt, von denen der Testfall zumindest auf einer verwendbar sein soll (siehe Abschnitt 9.4). Durch Übergabe des PROMELA-Programms und der LTL-Formel an die Komponente *D*, die als Schnittstelle zum Model-Checker SPIN dient, wird gegebenenfalls von SPIN ein Gegenbeispiel generiert. Dieses in Textform vorliegende Gegenbeispiel wird anschließend von Komponente *E* als Testfall interpretiert. Je nach Bedarf können anschließend weitere Testfälle für andere Testziele oder Produktkonfigurationen der SPL generiert werden.

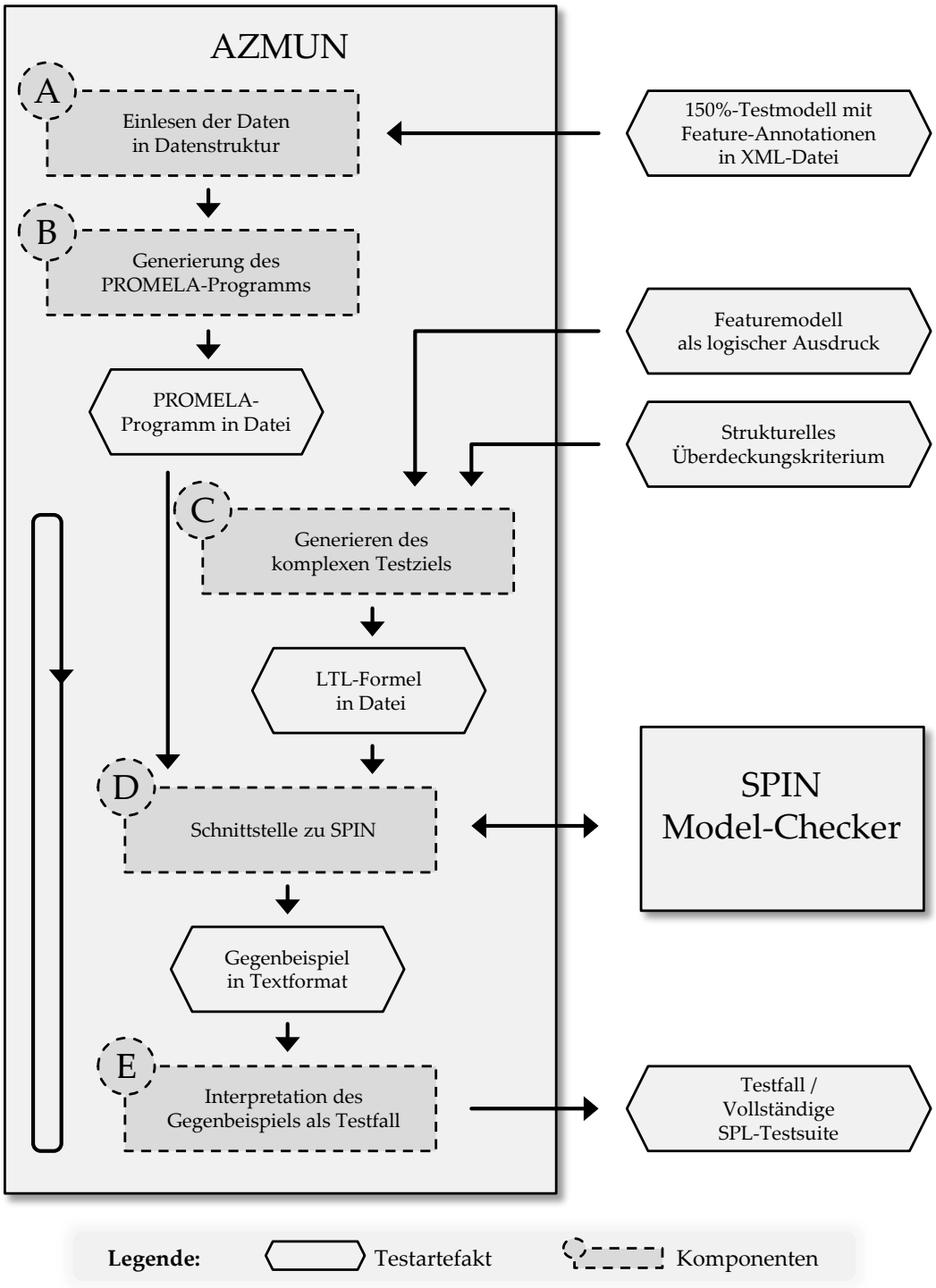


Abbildung 9.2: Workflow der AZMUN-Komponenten

9.2 TESTFALLGENERIERUNG BASIEREND AUF 150%-TESTMODELLEN

Die in dieser Arbeit verwendeten Algorithmen übergeben der Funktion *generate* als Parameter eine Menge von abzudeckenden Testzielen TG_{tc}, eine Menge von Produktkonfigurationen PC_{sel} und ein 150%-Testmodell vtm. Basierend auf dem 150%-Testmodell vtm soll für die Testziele TG_{tc} ein Testfall tc generiert werden, der für mindestens eine Produktkonfiguration $pc_{bound} \in PC_{sel}$ verwendbar ist. Damit der aus dem 150%-Testmodell hergeleitete Testfall tc zumindest für ein produktspezifisches Testmodell valide ist, muss die Variabilität des 150%-Testmodells für die Erstellung von Testfall tc gebunden werden.

Das Binden der Variabilität eines 150%-Testmodells lässt sich unterschiedlich realisieren. So kann dies bereits *vor* oder erst *während* der Testfallgenerierung durchgeführt werden. Da für dieser Arbeit die zweite Variante, Binden während der Testfallgenerierung, in einer Komponente realisiert wurde, wird auf diese ausführlich in Abschnitt 9.2.2 eingegangen.

9.2.1 *Binden vor der Testfallgenerierung*

Beim Binden der Variabilität eines 150%-Testmodells vor der Testfallgenerierung wird dem Testfallgenerator nicht das 150%-Testmodell übergeben, sondern ein daraus hergeleitetes, eigenständiges produktspezifisches Testmodell tm_{bound} ohne Variabilität (siehe Abbildung 9.3). Testmodell tm_{bound} enthält nur die Elemente aus dem Zustandsautomaten tm_{150} des 150%-Testmodells $vtm = (tm_{150}, \alpha)$, welche vom Feature-Annotations-Funktional α für $pc_{bound} \in PC_{sel}$ vorgegeben werden (siehe Abschnitt 6.4). Als Parameter wird dem Testfallgenerator das produktspezifische Testmodell tm_{bound} und das Testziel $tg_{complex}$ übergeben. Testziel $tg_{complex}$ setzt sich dabei aus all den Testzielen aus $TG_{tc} \subseteq TG$ zusammen, die tc abdecken soll. Das Binden vor der Testfallgenerierung hat folgende Vorteile:

- Durch das Entfernen von Modellelementen bei der Konstruktion des produktspezifischen Testmodells aus dem 150%-Testmodell verringert sich dessen Komplexität, was zu einem kleineren Zustandsraum im Vergleich zu dem des 150%-Testmodells führt. Dadurch vereinfacht sich für den Testfallgenerator die Suche nach einem Testfall.

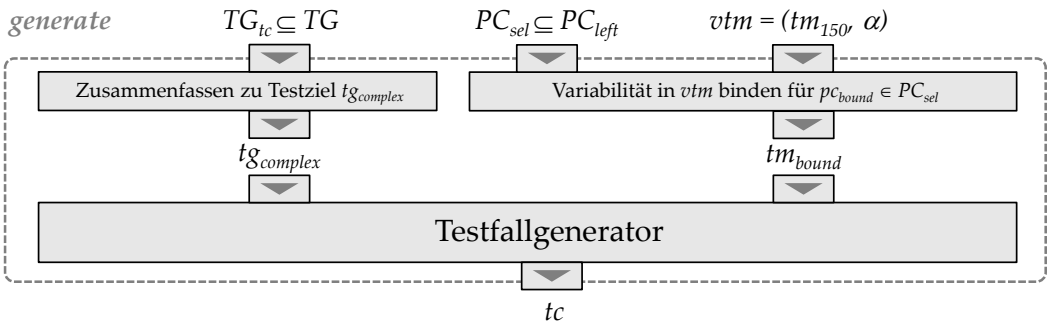


Abbildung 9.3: Binden der Variabilität vor der Testfallgenerierung

- Verglichen mit dem Zustandsautomat tm_{150} des 150%-Testmodells vtm enthält das eigenständige, produktspezifische Testmodell tm_{bound} nur die strukturellen Elemente, welche auch zur Produktkonfiguration pc_{bound} gehören. Dadurch können Testfall-Suchanfragen bezüglich der Abdeckung von strukturellen Testzielen, welche in tm_{150} aber nicht in tm_{bound} enthalten sind, noch bevor die Suche startet wegen Unerfüllbarkeit verworfen werden.

Es existieren aber auch Nachteile:

- Jedes benötigte produktspezifische Testmodell tm_{bound} muss aus dem 150%-Testmodell vtm konstruiert werden.
- Wenn vor der Testfallgenerierung nicht offensichtlich ist, ob eine Suchanfrage bzgl. eines Testziels (z.B. bestehend aus komplexen Bedingungen über Attributen/Variablen) auf einer Produktvariante erfüllbar ist, muss zwangsläufig nach einem Testfall für dieses Testziel gesucht werden. Wenn dieser Fall bei sehr vielen Produktvarianten auftritt, müssen alle einzeln bearbeitet werden.

9.2.2 *Binden während der Testfallgenerierung*

Das Binden der Variabilität kann auch an den Testfallgenerator delegiert werden. Dieser konfiguriert während der Testfallsuche das übergebene 150%-Testmodell $vtm = (tm_{150}, \alpha)$ zu einem geeigneten produktspezifischen Testmodell $bind(vtm, pc_{bound} \in PC_{sel})$ um, sodass ein Testfall tc erstellt werden kann, der alle Testziele $tg \in TG_{tc}$ abdeckt (siehe Abbildung 9.4). Damit die Variabilität des 150%-Testmodells vom Testfallgenerator während der Testfallsuche entsprechend der Produktkonfiguration pc_{bound} gebunden werden kann, muss in der Modellstruktur tm_{150} das Feature-Annotations-Funktional α realisiert werden. Dafür muss vor der Testfallgenerierung einmalig ein Testmodell $tm_{150enriched}$ basierend auf der Struktur von tm_{150} erstellt werden, welches um Featurevariablen, eingebettete Selektionsbedingungen und Konfigurationsaktionen ergänzt wird. Anschließend wird dem Testfallgenerator das Testmodell $tm_{150enriched}$ und das komplexe

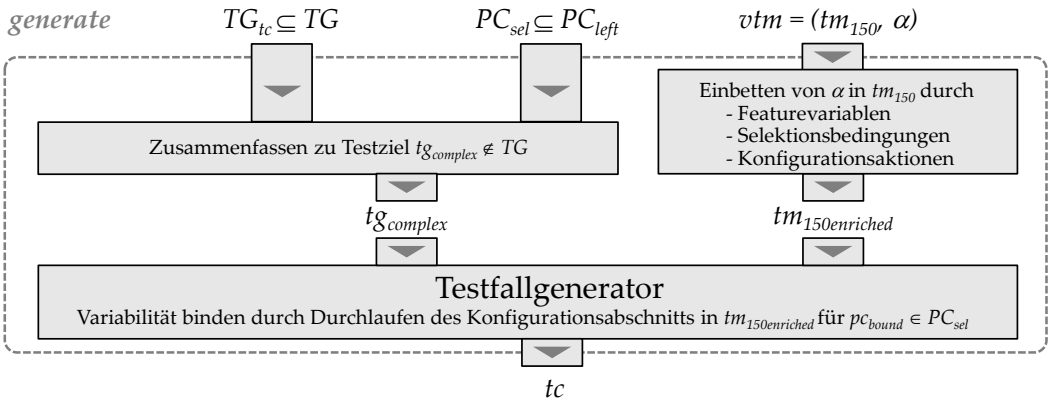


Abbildung 9.4: Binden der Variabilität während der Testfallgenerierung

Testziel $tg_{complex}$ zur Generierung eines Testfalls tc übergeben. Das komplexe Testziel $tg_{complex}$ setzt sich aus den Testzielen $TG_{tc} \subseteq TG$ und den zugelassenen Produktkonfigurationen $PC_{sel} \subseteq PC$ zusammen. Beispielsweise kann eine Testfallspezifikation (siehe Definition 2.7) das Abdecken eines komplexen Testziels $tg_{complex}$ fordern, welches sich aus $TG_{tc} = \{tg', tg''\}$ und $PC_{sel} = \{pc', pc''\}$ zusammensetzt und wie folgt beschrieben werden könnte: „Wähle zum Konfigurieren von $tm_{150enriched}$ entweder pc' oder pc'' und decke auf dem so entstandenen produktspezifischen Testmodell tg' und tg'' ab."

Durch die Konfigurationsaktionen konfiguriert der Testfallgenerator das Testmodell $tm_{150enriched}$ zum produktspezifischen Testmodell tm_{bound}. Die Konfigurationsaktionen stellen sicher, dass sich die Produktkonfiguration nicht während der Suche nach einem Testfall dynamisch ändert. Folglich kann garantiert werden, dass, wenn ein Testfall gefunden wird, dieser auch wirklich für zumindest ein produktspezifisches Testmodell pc_{bound} valide ist. Die Konfigurationsaktionen können als vorgeschalteter Konfigurationsabschnitt oder als eingebettete Konfigurationsaktionen modelliert werden.

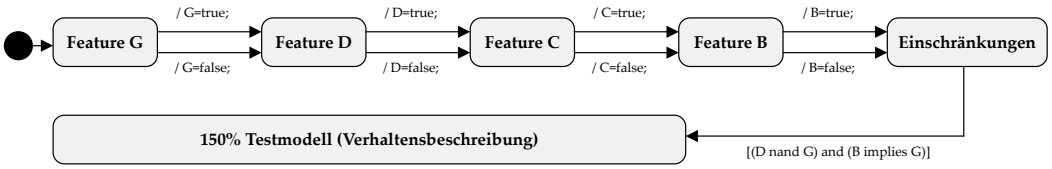
Der vorgeschaltete Konfigurationsabschnitt stellt eine Struktur in $tm_{150enriched}$ dar, die noch vor dem Eintritt in die Struktur des Zustandsautomaten tm_{150} durchlaufen werden muss. Der Konfigurationsabschnitt bietet alle Produktkonfigurationen PC zur Auswahl an, von denen eine nach dem Durchlauf ausgewählt ist. Um die beim Durchlauf gewählte Produktkonfiguration $pc_{bound} \in PC_{sel}$ zu speichern, existiert für jeden Feature-Parameter eine Featurevariable in $tm_{150enriched}$. Einer Featurevariable wird während des Durchlaufs des Konfigurationsabschnitts der Wert *true* zugewiesen, wenn das Feature in der gewählten Produktkonfiguration pc_{bound} enthalten ist, ansonsten *false*. Nachdem der Konfigurationsabschnitt durchlaufen wurde entsprechen alle Feature-Variablen zusammen folglich der gewählten Produktkonfiguration pc_{bound}.

Beispiel

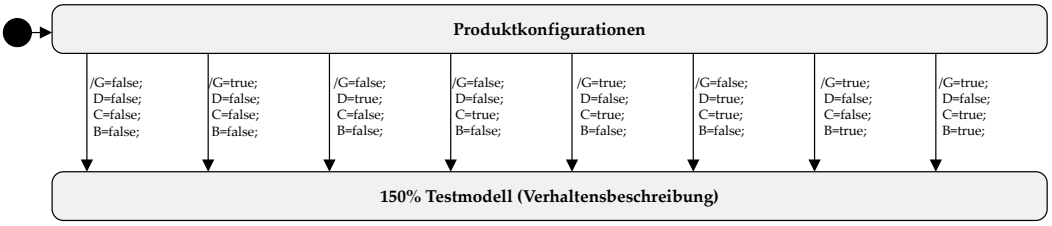
In Abbildung 9.5 sind zwei Möglichkeiten dargestellt, wie sich der Konfigurationsabschnitt im 150%-Testmodell der FA-SPL modellieren lässt. Als Bezeichner für die Featurevariablen werden G, D, C und B verwendet.

1. Bei der Realisierung des Konfigurationsabschnitts als Kette wird in jedem Glied der Kette eine Featurevariable entweder auf *anwesend* (*true*) oder *abwesend* (*false*) gesetzt (siehe Abbildung 9.5a). Erst am Ende der Kette wird durch eine Transitionsbedingung, welche die Abhängigkeiten zwischen den Features repräsentiert, überprüft, ob die Werte der Featurevariablen einer gültigen Produktkonfiguration entsprechen.

2. Bei der zweiten Möglichkeit kann auf die Überprüfung am Ende des Konfigurationsabschnitts verzichtet werden, da nur gültige Produktkonfigurationen zur Auswahl zugelassen werden. Zu diesem Zweck wird für jede zulässige Produktkonfiguration eine eigene Transition bereitgestellt, in deren Aktionsteil alle Featurevariablen atomar, entsprechend der gewählten Produktkonfiguration, auf anwesend oder abwesend gesetzt werden (siehe Abbildung 9.5b).



(a) Kette



(b) Einzeln

Abbildung 9.5: Konfigurationsabschnitt im 150%-Testmodell

Die erste Modellierungsweise ist kompakter als die zweite, da bei dieser für jedes Feature nur zwei weitere Transitionen hinzukommen, wohingegen bei der zweiten die Anzahl der Transitionen mit der Anzahl der Produktkonfigurationen linear steigt. Die zweite Modellierungsweise kann verwendet werden, wenn aufgrund von vielen Features sehr viele Featurekombinationen möglich sind, von denen aber nur sehr wenige eine zulässige Produktkonfiguration darstellen. Im Vergleich zum als Kette realisierten Konfigurationsabschnitt wird in diesem Fall nicht erst am Ende der Kette erkannt, ob die gewählte Featurekombination eine zulässige Produktkonfiguration darstellt oder nicht.

Damit bei der Testfallsuche die im Konfigurationsabschnitt ausgewählte Produktkonfiguration pc_{bound} berücksichtigt wird, muss jede Transition ihre Selektionsbedingung, die durch das Funktional α vorgegeben wird, mit in ihre Transitionsbedingung aufnehmen. Anschließend kann bei der Suche nach einem Testfall eine Transition nur dann traversiert werden, wenn die eingebettete Selektionsbedingung erfüllt ist. Dadurch lassen sich nur noch Testfälle aus dem 150%-Testmodell vtm generieren, die für das zu pc_{bound} gehörende, aus vtm hergeleitete produktspezifische Testmodell $bind(vtm, pc_{bound})$ valide sind. Sollte die Suche nach einem Pfad zum Testziel für die gewählte Produktkonfiguration $pc_{bound} \in PC_{sel}$ nicht erfolgreich sein, lässt sich eine weitere Produktkonfiguration $pc'_{bound} \in PC_{sel}$ nur durch das erneute Durchlaufen des Konfigurationsabschnitts auswählen.

Beispiel

In Abbildung 9.6 sind die Transitionsbeschriftungen des 150%-Testmodells der FA-SPL mit eingebetteten Selektionsbedingungen dargestellt.

Eine Alternative zu Realisierung der Konfigurationsaktionen als vorgeschalteten Konfigurationsabschnitt ist das Einbetten dieser in die mit Selektionsbedingungen annotierten Transitionen des Zustandsautomaten tm_{150}. Der Vorteil besteht darin, dass die Produktkonfiguration pc_{bound} nicht bereits vor dem Eintritt

Trans.	Beschriftung
a	/ costs=0; paid=0; print=0; tBlanks=10; tN=0; tR=0; tD=0; tG=0;
b	normal [tBlanks-tN-tR-tD-tG>0] / tN++; costs+=2;
c	reduced [tBlanks-tN-tR-tD-tG>0] / tR++; costs+=1;
d	day [(D) && (tBlanks-tN-tR-tD-tG>0)] / tD++; costs+=4;
e	group [(G) && (tBlanks-tN-tR-tD-tG>0)] / tG++; costs+=6;
f	next [costs>0]
g	coin [paid<costs] / paid++;
h	bill [(B) && (paid<costs)] / paid+=5;
i	cancel [paid<costs]
j	[paid>=costs] / print=10+tN+tR+tD+tG;
k	[print>0] / print--;
l	[print==0] / paid-=costs; costs=0; tBlanks=tBlanks-tN-tR-tD-tG;
m	[tN>0] / tN--;
n	[(tN==0) && (tR>0)] / tR--;
o	[**(D)** && (tN==0) && (tR==0) && (tD>0)] / tD--;
p	[**(G)** && (tN==0) && (tR==0) && (tD==0) && (tG>0)] / tG--;
q	[**(!C)** && (tN==0) && (tR==0) && (tD==0) && (tG==0)] / paid=0;
r	[**(C)** && (tN==0) && (tR==0) && (tD==0) && (tG==0)]
s	cancel [costs>0]
t	[paid>0] / paid--;
u	[paid==0] / tN=0; tR=0; tD=0; tG=0; costs =0;
v	maintenance
w	[tBlanks<20] / tBlanks--;
x	[tBlanks==20]

Abbildung 9.6: Eingebettete Selektionsbedingungen

in den Zustandsautomaten tm_{150} festgelegt werden muss, sondern dies erst während der Testfallsuche geschieht. Zu diesem Zweck besitzt jede Featurevariable zu Beginn den Wert *dontcare*. Diese Wertzuweisung ist gleichzusetzen mit „-“ und bedeutet, dass die An- oder Abwesendheit eines Features keine Auswirkung auf die Auswertung der Selektionsbedingungen der Transitionen im Transitionspfad des Testfalls hat. Sobald jedoch eine Transition traversiert wird, welche mit einer Selektionsbedingung annotiert ist, ändern sich der Werte einer jeden in der Selektionsbedingung verwendeten Featurevariable entsprechend der geforderten Belegung zu *true* oder *false*. Wird letztendlich ein Testfall gefunden, wurden somit nur die Featurevariablen von *dontcare* auf *true* oder *false* gesetzt, welche für diesen Testfall relevant sind. Dadurch beschreiben alle Featurevariablen zusammen nicht zwingend nur eine konkrete Produktkonfigurationen pc_{bound} sondern eine Menge von Produktkonfigurationen PC_{tc}, für die der Testfall valide ist. Da sich PC_{tc} bereits aus den Werten der Featurevariablen ableiten lässt, ist es nicht mehr notwendig im Anschluss die in Abschnitt 6.5 vorgestellte Analysetechnik auf den Testfall anzuwenden. Der Nachteil beim Einbetten der Konfigurationsaktionen besteht darin, dass durch die Hinzunahme des Wertes *dontcare* (ergänzend zu *true* und *false*) nun 2 Bits statt 1 Bit für eine Featurevariable verwendet werden müssen, was zur Folge hat, dass der Zustandsraum des Zustandsautomaten $tm_{150enriched}$ in Abhängigkeit von der Anzahl der Featurevariablen ansteigt.

Trans.	Transitionsbeschriftung
…	…
q	[**(C<=0.5)** && (tN==0) && (tR==0) && (tD==0) && (tG==0)] / paid=0; **C=0;**
r	[**(C>=0.5)** && (tN==0) && (tR==0) && (tD==0) && (tG==0)] / **C=1;**
…	…

Abbildung 9.7: Transitionsbeschriftung mit eingebetteten Konfigurationsaktionen

Beispiel

In Abbildung 9.7 sind die Transitionsbeschriftungen des 150%-Testmodells der FA-SPL mit eingebetteten Konfigurationsaktionen dargestellt. Jede Featurevariable bekommt als Initialwert den Wert 0.5 zugewiesen, der gleichbedeutend ist mit dem Wert *dontcare*. Wird dann während der Testfallsuche beispielsweise die Transition Q traversiert, bekommt die Featurevariable C den Wert 0 (gleichbedeutend mit $false$) zugewiesen. Anschließend können keine Transitionen mehr traversiert werden, welche die Anwesenheit des Features C verlangen.

Das Binden während der Testfallgenerierung hat folgende Vorteile:

- Die Prüfung, ob das Testziel $tg_{complex}$ auf irgendeiner $pc_{bound} \in PC_{sel}$ erfüllbar ist, wird an den Testfallgenerator delegiert. Sollte kein Testfall tc gefunden werden, sind alle Produktkonfigurationen $pc \in PC_{sel}$ mit nur einer einzigen Suchanfrage bearbeitet worden.
- Das Testmodell $tm_{150enriched}$ muss nur einmal konstruiert werden und kann danach für jedes beliebige Testziel und jede beliebige Produktkonfiguration verwendet werden.

Jedoch lassen sich auch folgende Nachteil feststellen:

- Durch das Hinzufügen des Konfigurationsabschnitts und der Featurevariablen vergrößert sich der Zustandsraum, was sich negativ auf die Testfallgenerierung hinsichtlich Dauer und Speicherverbrauch für das Testziel $tg_{complex}$ auswirken kann.
- Da jedes in einem produktspezifischen Testmodell enthaltene Testziel auch im 150%-Testmodell enthalten ist, kann es im Gegensatz zum Binden vor der Testfallgenerierung keine ungültige Anfragen bzgl. der Testziele $tg \in TG$ geben. Dadurch sucht der Testfallgenerator auch nach einem Testfall für ein Testziel, welches in keiner $pc \in PC_{sel}$ enthalten ist. Effizienter ist es, wenn nur Testfälle für Testziele gesucht werden, wenn das abzudeckende Testziel auch in einer Produktvariante $pc \in PC_{sel}$ enthalten ist. Für strukturelle Testziele wie Transitionen oder Zustände lässt sich relativ einfach feststellen, ob diese auch in einer Produktvariante $pc \in PC_{sel}$ enthalten sind, da aufgrund der Feature-Annotationen im 150%-Testmodell bekannt ist, welche produktspezifischen Testmodelle welche strukturellen Elemente enthalten. Schwieriger ist dies jedoch für Testziele, die sich aus Attributen/Variablen zusammensetzen.

9.3 AUFBAU EINES GENERIERTEN PROMELA-PROGRAMMS

In Azmun wird der Model-Checker SPIN als Testfallgenerator eingesetzt. Da SPIN nur PROMELA-Programme verarbeiten kann, muss das 150%-Testmodell (genauer: das Testmodell $tm_{150enriched}$ (siehe Abschnitt 9.2.2)) in ein PROMELA-Programm transformiert werden. PROMELA ist eine Sprache mit der sich nebenläufige Prozesse modellieren lassen. Ein PROMELA-Programm besteht aus Prozessen, Variablen und Nachrichtenkanälen. Letztere ermöglichen die synchrone oder asynchrone Kommunikation zwischen Prozessen. Detaillierte Informationen zur PROMELA-Syntax und den verwendbaren Programmkonstrukten sind in [Holo3, Com] aufgeführt. Im Folgenden wird auf den strukturelle Aufbau eines generierten PROMELA-Programms eingegangen. Dafür wird auf Abbildung 9.8 Bezug genommen.

Das durch einen Zustandsautomaten beschriebene Verhalten einer jeden im Testmodell enthaltenen Komponente wird im PROMELA-Programm durch einen eigenen Prozess modelliert (Zeile 3 bis Zeile 51). Der Programm-Aufbau eines jeden Prozesses lässt sich dabei in vier Abschnitte unterteilen:

- Deklaration und Initialisierung (Zeile 1 bis 9)
- Signale empfangen (Zeile 11 bis 18)
- Berechnung des nächsten Schritts (Zeile 19 bis 44)
- Signale senden (Zeile 45 bis 49)

Im Deklarations-Abschnitt (Zeile 1 bis 9) werden die vom Prozess benötigten Variablen deklariert sowie initialisiert. Zusätzlich dazu besitzt jeder Prozess einen globalen Kommunikationskanal (Zeile 1), der zum Empfang von Daten, die von anderen Prozessen versendet werden, verwendet wird. Da in PROMELA nur numerische Daten und keine Signale über einen Kommunikationskanal versendet werden können, müssen die im Testmodell verwendeten Signale auf numerische Werte abgebildet werden. Um die Komplexität bei der Bearbeitung des PROMELA-Programms zu verringern, erlaubt der deklarierte Kanal nur synchronen Datenaustausch (Rendezvous-Prinzip). Das bedeutet, dass der Kanal keine Daten zwischenspeichert. Folglich blockiert ein versendender Prozess so lange, bis der empfangende Prozess die Daten entgegen nimmt.

In Zeile 4 bis 9 werden die lokalen Variablen eines Prozesses deklariert und initialisiert. In Zeile 7 wird für jedes Attribut einer Komponente eine lokale Variable angelegt und dieser ein Initialwert zugewiesen. Die Anzahl der für die Variable zum Speichern von Werten reservierten Bits ergibt sich aus dem Wertebereich, der dem entsprechenden Attribut im Testmodell zugewiesen wurde. Ebenso wird in Zeile 6 für jede Featurevariable (siehe Abschnitt 9.2.2) eine lokale Variable vom Typ *bool* hinzugefügt. Damit für den verhaltensbeschreibenden Zustandsautomaten der Komponente und dessen Unterautomaten der aktuellen Zustand und die aktuelle Transition gespeichert werden kann, wird in Zeile 8 und 9 jeweils eine dafür vorgesehene Variable angelegt. Wie viele Bits einer Variable zugewiesen werden, hängt von der Anzahl der im jeweiligen Automat enthaltenen Zustände bzw. Transitionen ab. Die Zuweisung des Wertes 0 in Zeile 8 bedeutet, dass sich

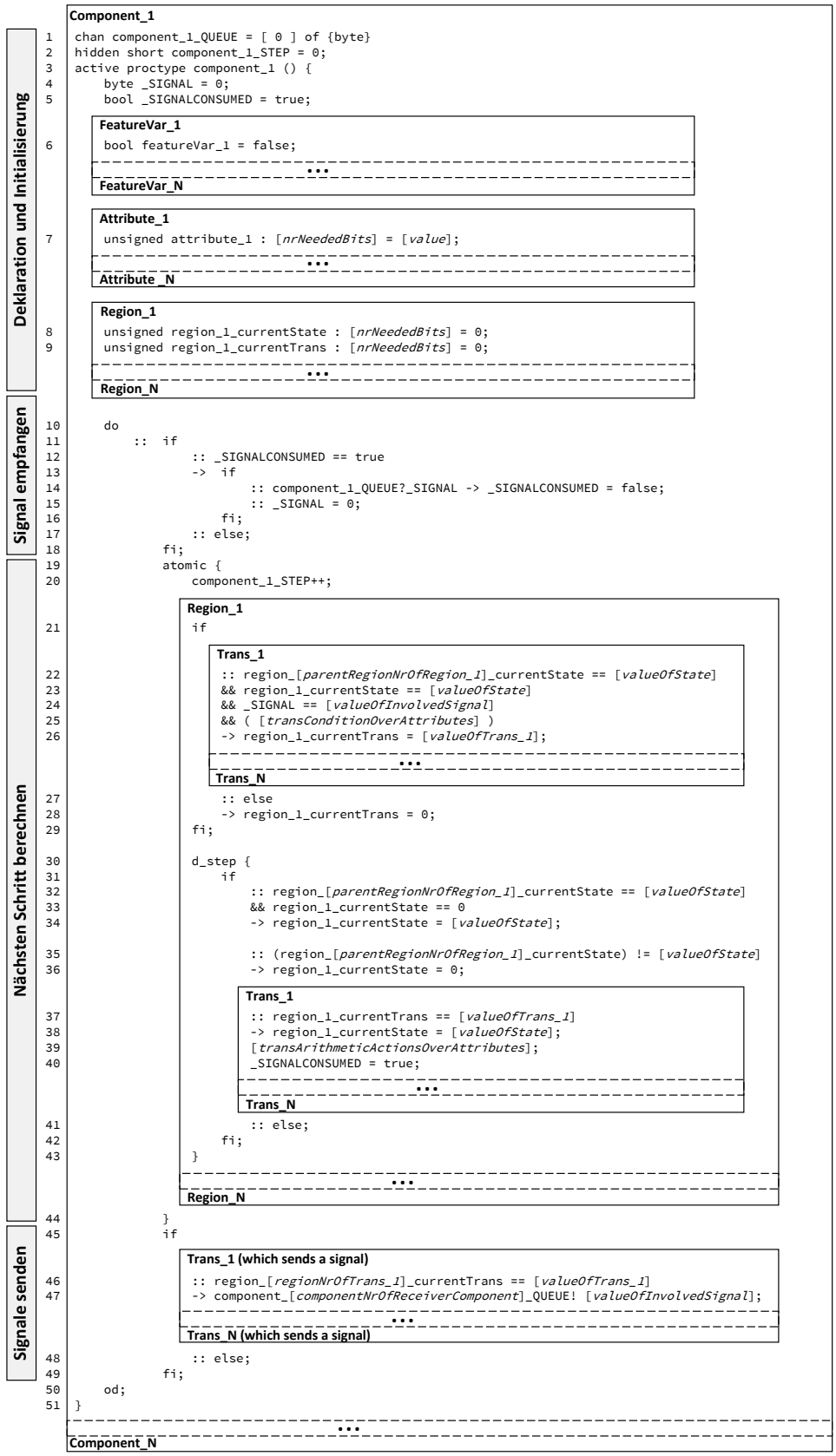


Abbildung 9.8: Struktur des generierten PROMELA-Programms

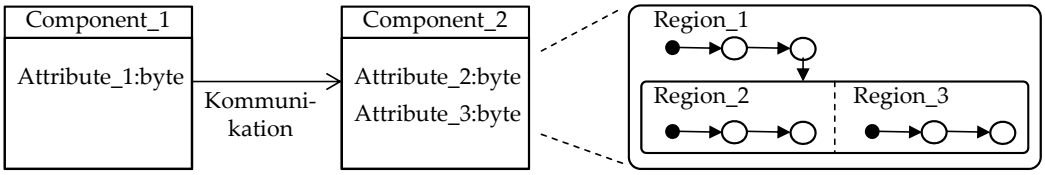


Abbildung 9.9: Schematische Darstellung eines Testmodells

der Automat im Initialzustand befindet. Die Zuweisung des Wertes 0 in Zeile 9 bedeutet hingegen, dass keine Transition ausgewählt wurde. In Zeile 10 beginnt eine nicht-endende Schleife (do-Anweisung). In jeder Iteration (Zeile 11 bis 49) wird für den Zustandsautomaten einer Komponenten und dessen Unterautomaten der nächste Zustand berechnet und gegebenenfalls Signale von anderen Prozessen empfangen oder an diese versendet.

In den Zeilen 11 bis 18 wird das Empfangen von Signalen behandelt. In Zeile 12 wird überprüft, ob das zuletzt aus dem Kanal entnommene Signal bereits verarbeitet wurde. Ist das der Fall, kann ein neues Signal, wenn ein anderer Prozess dieses sendet, gelesen und in die lokale Variable *_SIGNAL* geschrieben werden (Zeile 14). Sollte kein Signal von einem anderen Prozess versendet werden, wird der Variable *_SIGNAL* der Wert 0 (gleichbedeutend mit „kein Signal“) zugewiesen.

Zwischen Zeile 19 bis 44 findet die Berechnung des nächsten Schritts für den zur Komponente gehörenden Zustandsautomaten und dessen mögliche Unterautomaten bzw. orthogonal laufenden Automaten statt. Da sich die Zustände und Transitionen eines jeden Automaten einer Region zuordnen lassen, wird jede Region einzeln behandelt, beginnend mit der am höchsten liegenden Region (z.B. in Abbildung 9.9 die Region *Region_1*). Durch die Verwendung des Schlüsselworts *atomic* in Zeile 19 werden die Anweisungen von Zeile 19 bis 44 ohne Unterbrechung durch andere Prozesse ausgeführt, was eine Verringerung der Komplexität bei der Bearbeitung des PROMELA-Programms zur Folge hat. Die atomare Ausführung dieser Anweisung ist möglich, da keine dieser Anweisungen blockiert bzw. mit keinem anderen Prozess verzahnt ist.

In Zeile 22 bis 29 wird für eine Region die nächste aktive Transition berechnet. Welche Transition der entsprechenden Variable in Zeile 26 zugewiesen wird, ist davon abhängig,

- in welchem Zustand sich der Automat in der darüber liegenden Region befindet, dem die aktuelle Region angehört (Zeile 22),
- in welchem Zustand (Ausgangszustand der Transition) sich der Automat in der aktuellen Region befindet (Zeile 23),
- ob das für die Transition benötigte Signal eingelesen wurde (Zeile 24) und
- ob die Bedingung der Transition erfüllt ist (Zeile 25).

Erfüllen mehrere Transitionen die für sie benötigten Bedingungen kann eine dieser Transitionen nicht-deterministisch ausgewählt werden. Sollte keine Transition der Region die für sie benötigten Bedingungen erfüllen, wird in Zeile 28 der Transitionsvariable der Wert 0 (gleichbedeutend mit „keine Transition ausgewählt“) zugewiesen.

Abhängig von der gewählten Transition wird in Zeile 30 bis 43 der nächste aktive Zustand der Region bestimmt und die entsprechende arithmetische Transitionsaktion ausgeführt. Da sich der Zielzustand einer Transition eindeutig bzw. deterministisch ermitteln lässt, kann das Schlüsselwort *d_step* verwendet werden (Zeile 30), was zu einer Verringerung der Komplexität bei der Bearbeitung des PROMELA-Programms führt. Einen Spezialfall stellen die zwei Bedingungen in Zeile 32 und 35 dar.

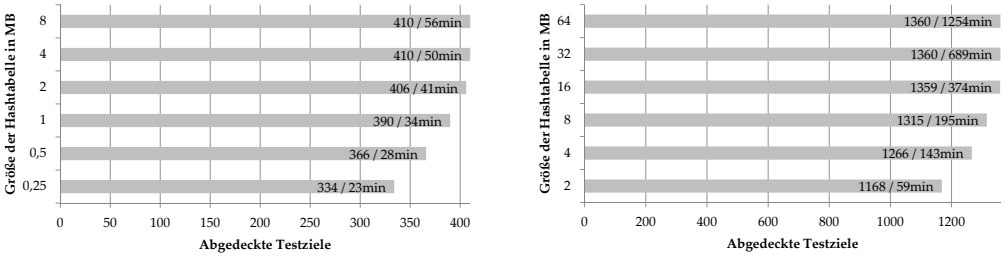
- Sollte die Zustandsvariable der Region den Wert des Initialszustands besitzen, wird dieser der Wert des Zustands zugewiesen, auf den der Initialzustand der Region zeigt (Zeile 32).
- Wenn der in der darüber liegenden Region befindliche Zustand, der die aktuelle Region enthält, nicht mehr aktiv ist (Zeile 35), wird die Zustandsvariable der aktuellen Region auf den Wert des Initialzustands zurückgesetzt.

Sollte keine dieser zwischen Zeile 30 und 43 aufgeführten Bedingungen erfüllt sein, findet keine Veränderung des Zustands statt (Zeile 41).

Da in einem *d_step*-Block oder *atomic*-Block keine blockierenden Anweisungen enthalten sein dürfen, wurde das Senden von Signalen als Folge von Transitionsaktionen in die Zeilen 45 bis 49 ausgelagert. Dort existiert für jede Transition, die als Aktion ein Signal versendet, eine Bedingung, die überprüft, ob die Transition in ihrer Region aktiv ist. Durch die Anweisung in Zeile 47 wird der Versand des Signals an den Kommunikationskanal der Empfänger-Komponente modelliert.

9.4 SPIN ALS TESTFALLGENERATOR

Der Model-Checker SPIN verifiziert die als PROMELA-Programm vorliegenden Modelle on-the-fly [Com]. Das bedeutet, dass der zu bearbeitende Zustandsgraph erst während der Verifikation nach Bedarf aufgebaut wird und nicht bereits vorher vollständig vorliegen muss, wie es beim Model-Checker NuSMV [BCTT] der Fall ist. Das on-the-fly-Vorgehen ist effizienter als den Zustandsgraphen zuvor zu erstellen, wenn die Wahrscheinlichkeit hoch ist, bereits in einem sehr kleinen Teil des vollständigen Zustandsgraphen ein Gegenbeispiel zu finden. Durch das on-the-fly-Vorgehen kann auch ein komplexes Programm, dessen Zustandsgraph zu groß ist, um diesen vollständig aufzubauen, in Teilen bearbeitet werden, was zur Folge hat, dass gegebenenfalls Gegenbeispiele generiert werden können. Sollte es jedoch nötig werden einen Großteil des Zustandsgraphen aufzubauen, kann das on-the-fly-Verfahren aufgrund schlechterer Komprimierung des bereits konstruierten Zustandsgraphen ineffizienter sein als diesen zuvor vollständig zu konstruieren [Com]. Da bei der Verwendung eines Model-Checkers als Testfallgenerator



(a) FA-SPL mit insg. abdeckbaren 410 Testzielen

(b) BCS-SPL mit insg. abdeckbaren 1360 Testzielen

Abbildung 9.10: Beziehung zwischen Anzahl abgedeckter Testziele und Größe der Hashtabelle

davon auszugehen ist, dass für die verifizierende Trap-Property (siehe Definition 3.6) ein Gegenbeispiel existiert und es somit nicht nötig ist den vollständigen Zustandsgraphen zu konstruieren, ist ein on-the-fly-Vorgehen in der Regel geeigneter als den Zustandsgraphen zuvor vollständig zu konstruieren.

9.4.1 *Verifikationsoption BITSTATE*

SPIN bietet eine Vielzahl von Optionen an, die sich auf die Bearbeitungsgeschwindigkeit oder den Speicherverbrauch auswirken können [Com]. Der maximal benötigte Speicherverbrauch bei der Verifikation eines PROMELA-Programms ergibt sich aus der Anzahl der im Zustandsraum enthaltenen Zustände und aus der Anzahl der benötigten Bits zum Darstellen eines Zustands. Jedoch übersteigt in der Praxis der benötigte Speicherverbrauch für komplexere Programme den vorhandenen Speicher bei weitem [Holo3]. Um im verfügbaren Speicher so viele besuchte Zustände wie möglich ablegen zu können, wird versucht die Anzahl der benötigten Bits zur Darstellung eines Zustands zu reduzieren. SPIN bietet dafür unter anderem folgende zwei Optionen an:

- *COLLAPSE* - Komprimiert den Speicherverbrauch eines Zustands um bis zu 80%, wobei sich jede komprimierte Darstellung ihrem ursprünglichen Zustand zuordnen lässt.
- *BITSTATE* - Komprimiert den Speicherverbrauch eines Zustands durch die Anwendung von Hashfunktionen auf 1 Bit in einer Hashtabelle, wobei aufgrund von Kollisionen keine eindeutige Zuordnung möglich ist. Die Häufigkeit von Kollisionen ist abhängig von der Hashfunktion und der Größe der Hashtabelle.

In dieser Arbeit wurde die Option *BITSTATE* bei der Evaluation verwendet, da bei der Nutzung von *COLLAPSE* für die definierten Testziele auf den 150%-Testmodellen der FA-SPL und BCS-SPL häufig kein Testfall (bzw. Gegenbeispiel) im aufgebauten Teil des vollständigen Zustandsgraphen von 2 Gigabyte Größe

gefunden werden konnte. Bei Verwendung der Option *BITSTATE* hingegen konnte bei einer entsprechenden Größe der Hashtabelle für jedes Testziel einer jeden Produktvariante ein Testfall gefunden werden (siehe Abbildung 9.10).

Für die FA-SPL mussten für eine vollständige *all-transition-pairs*-Abdeckung insgesamt 410 erfüllbare Testziele (die erfüllbaren Testziele aller produktspezifischen Testmodelle zusammen betrachtet) abgedeckt werden. Eine solche vollständige SPL-Testsuite konnte erst mit einer Hashtabelle von 4 Megabyte Größe generiert werden. Es existierten jedoch auch noch weitere Testziele (Transitionspaare) (weniger als 160), für die sich kein Testfall generieren liess, da diese Modellfragmente, wenn sie denn in einem produktspezifischen Testmodellen enthalten waren, aufgrund ihrer Transitionsbedingungen nicht erfüllbar waren. Außerdem konnten für mehr als 238 Testziele (Transitionspaare) kein Testfall generiert werden, da deren strukturelle Elemente nicht im entsprechenden produktspezifischen Testmodell enthalten waren.

Bei der BCS-SPL konnten erst ab einer Hashtabelle mit der Größe von 32 Megabyte alle durch das Überdeckungskriterium *all-transitions* definierten 1360 erfüllbaren Testziele durch Testfälle abgedeckt werden. Weitere 1192 Testziele waren nicht erfüllbar, da deren strukturelles Element nicht im entsprechenden produktspezifischen Testmodell enthalten war.

Bei der Verwendung von kleineren Hashtabellen konnte keine vollständige SPL-Abdeckung erreicht werden, da für einige Testziele auf einigen Produktkonfigurationen kein Testfall aus dem 150%-Testmodell hergeleitet werden konnte. Dieser Umstand ist darauf zurückzuführen, dass bei der Verwendung von Hashtabellen mit zu geringer Größe das Hashen der Zustände auf ein Bit in der Hashtabelle zu Kollisionen führen kann, was zur Folge hat, dass die vollständige Bearbeitung des vollständigen Zustandsgraphen unmöglich wird. Dennoch kann es möglich sein, dass mit einer größeren Hashtabelle ein Testfall gefunden wird. Somit ist die Wahl der richtigen Größe der Hashtabelle bei der Verwendung von *BITSTATE* zur Testfallgenerierung essentiell. Ansonsten wird erfolglos nach einem Testfall für ein Testziel gesucht, was unnötige Kosten verursacht. Zu groß gewählte Hashtabellen haben jedoch den Nachteil, dass sowohl die erfolgreiche als auch die erfolglose Suche nach einem Testfall mehr Zeit benötigt. Beispielsweise dauert auf einem PC mit *i*7-2600 CPU (3,7 Ghz) das Erstellen einer vollständigen SPL-Testsuite für die FA-SPL bei einer Hashtabelle von 4 Megabyte Größe 50 Minuten, wohingegen die Verwendung einer Hashtabelle mit 8 Megabyte Größe bereits 56 Minuten benötigt (vergleiche Abbildung 9.10). Ebenfalls lässt sich erkennen, dass bereits mit einer sehr kleinen Hashtabelle für den Großteil der Testziele der Fallbeispiele in kurzer Zeit ein Testfall generiert werden konnte im Vergleich zu den Ergebnissen bei Verwendung von großen Hashtabellen. Zwecks höherer Effizienz könnte in zukünftigen Arbeiten bei der Testfallerstellung mit *BITSTATE* zuerst unter Verwendung einer kleinen Hashtabelle für den Großteil der Testziele in kurzer Zeit ein Testfall generiert werden. Anschließend wird für all die Testziele, für die kein Testfall gefunden wurde, erneut nach einem Testfall gesucht, diesmal jedoch unter Verwendung einer wesentlich größeren Hashtabelle.

```
a) spin.exe -O -a -F [ltl-file-name] [pml-file-name]
b) gcc.exe -DSAFETY -DBITSTATE -DNOBOUNDCHECK -o pan pan.c
c) pan.exe -m[maxSpinSteps] -w[bitstateHashSize] -k4 [shortestPath]
d) spin.exe -p -t -l -g -B -r -s [pml-file-name]
```

Abbildung 9.11: SPIN-Aufruf

```
(<> [PC_sel]) && (<> [tg_1∈TG_tc]) && (<> [tg_2∈TG_tc]) && ... && (<> [tg_n∈TG_tc])
```

(a) Komplexes zusammengesetztes Testziel

```
[process-name]:[regionTransVar] == [valueOfTrans]
```

(b) Transition als Testziel $tg \in TG_{tc}$

```
[process-name]:[regionTransVar] == [valueOfIngoingTrans]
&&
(X      [process-name]:[regionTransVar] == [valueOfOutgoingTrans]
        ||
        (       [process-name]:[regionTransVar] == 0
                U
                [process-name]:[regionTransVar] == [valueOfOutgoingTrans]
        )
)
```

(c) Transitionspaar als Testziel $tg \in TG_{tc}$

Abbildung 9.12: Schemata für eine LTL-Formel in SPIN

9.4.2 *Schnittstelle zu SPIN*

In Abbildung 9.11 sind die Befehlszeilen dargestellt, die in dieser Arbeit verwendet wurden, um mit SPIN Testfälle zu generieren.

In Abbildung 9.11a wird durch das Aufrufen von SPIN eine „pan.c"-Datei (C-Quellcode) generiert. Dafür wird das PROMELA-Programm (siehe Abschnitt 9.3) und eine LTL-Formel, welche das Testziele $tg_{complex}$ (siehe Abschnitt 9.2.2) beschreibt, an SPIN übergeben. Der generelle Aufbau einer solchen LTL-Formel für SPIN ist in Abbildung 9.12a dargestellt. Diese LTL-Formel fordert, dass ein Testfall *tc* gefunden werden soll, der für eine Produktkonfiguration $pc_{bound} \in PC_{sel}$ verwendbar ist und all die Testziele $tg \in TG_{tc}$ im 150%-Testmodell abdecken soll. Da sich die LTL-Formel auf das PROMELA-Programm bezieht, müssen auch die Variablennamen des PROMELA-Programms verwendet werden (siehe Abbildung 9.9). Wenn beispielsweise ein Transitionspaar als Testziel vorliegt, welches vom Testfall *tc* abgedeckt werden soll, muss das in Abbildung 9.12c dargestellte Schema verwendet werden.

```
[spinStep]: process [nr] ([process-name]) [pml-file]:[stmt-line] (state [nr]) [stmt-code]
      [proc-name]([nr]):[manipulatedVariables] = [newValue]
```

(a) Schema

```
   ...
45:   proc  1 (Comp_TM) model.pml:219 (state 103)    [region_1_TRANS = 8]
   Comp_TM(1):region_1_TRANS = 8
46:   proc  1 (Comp_TM) model.pml:264 (state 210)    [IF]
46:   proc  1 (Comp_TM) model.pml:286 (state 148)    [region_1_STATE = 1]
   Comp_TM(1):region_1_STATE = 1
46:   proc  1 (Comp_TM) model.pml:347 (state 209)    [.(goto)]
   Comp_TM(1):region_1_STATE = 1
48:   proc  1 (Comp_TM) model.pml:101 (state 17)     [Comp_TM_STEP = (Comp_TM_STEP+1)]
   Comp_TM_STEP = 7
   ...
```

(b) Beispiel

Abbildung 9.13: SPIN-Ausgabe eines Gegenbeispiels

In Abbildung 9.11b wird durch Aufruf des GNU C Compilers (GCC) die zuvor generierte „pan.c“-Datei in eine „pan.exe“-Datei kompiliert. Dadurch wird bei der Bearbeitung eine bessere Performanz erzielt, als wenn das Programm nur interpretiert wird. Durch die Übergabe von zusätzlichen Parametern, z.B. *BITSTATE*, lässt sich die Struktur bzw. das Verhalten des Maschinencodes modifizieren. Anschließend wird in Abbildung 9.11c die „pan.exe“-Datei gestartet. Als Argumente lassen sich unter anderem die maximale Suchtiefe im Zustandsraum und die Größe der Hashtabelle angeben. Ebenso ist es möglich nach dem Gegenbeispiel mit der kürzesten Suchtiefe bezogen auf die verwendete Hashtabelle zu suchen.

Wurde ein Gegenbeispiel gefunden, kann dieses mit den Befehlen aus Abbildung 9.11d textuell ausgegeben werden. Die Menge der in der Ausgabe enthaltenen Informationen lässt sich durch Parameter steuern. In Abbildung 9.13a ist das zur Evaluation in dieser Arbeit verwendete Ausgabe-Schema des Gegenbeispiels dargestellt. Die dort abgebildeten Daten sind ausreichend, um das Gegenbeispiel in einen Testfall zu transformieren. Für jede im PROMELA-Programm ausgeführte Anweisung wird eine Ausgabe in entsprechender Reihenfolge des Aufrufs unter Angabe des aktiven Prozesses erzeugt (siehe Abbildung 9.13a). Kam es bei der Ausführung der Anweisung zu einer Veränderung eines Variablenwertes, wird der neue Wert eine Zeile tiefer aufgeführt. Ein Beispiel für einen Teilausschnitt einer Ausgabe ist in Abbildung 9.13b dargestellt.

Da in der Ausgabe des Gegenbeispiels jede Änderung einer Variable als ein interner Schritt von SPIN angesehen wird (siehe *spinStep* in Abbildung 9.13a), wurde eine Hilfsvariable für jeden Prozess eingeführt, durch die sich erkennen lässt, wann ein Berechnungsschritt für den Folgezustand eines Automaten abgeschlossen ist. Diese Hilfsvariable ist in Zeile 2 der Abbildung 9.8 deklariert und wird in Zeile 20 vor Betreten des Anweisungsblocks zur Berechnung des nächsten Schritts inkrementiert.

10

ZUSAMMENFASSUNG

In dieser Arbeit wurde ein modellbasierter Testansatz vorgestellt, mit dem sich für eine Software-Produktlinie (SPL) eine vollständige SPL-Testsuite generieren lässt. Diese vollständige SPL-Testsuite enthält für jede Produktvariante einer SPL eine Teilmenge von Testfällen, die auf dem produktspezifischen Testmodell dieser Variante eine vollständige Abdeckung bzgl. eines Modell-Überdeckungskriteriums erreicht. Der vorgestellte Ansatz ist effizienter als bisherige Product-by-Product-Ansätze, da variantenübergreifend verwendbare Testfälle aus einem 150%-Testmodell generiert werden. Durch diese Testfälle, die sich auf mehreren Produktvarianten anwenden lassen, fällt die Anzahl der zu generierenden Testfälle in der Regel wesentlich geringer aus, was sich positiv auf die Kosten bei der Generierung und der Wartung der Testfälle auswirkt.

Des Weiteren wurden in dieser Arbeit drei Algorithmen zur Testsuite-Reduktion vorgestellt, mit denen sich die Anzahl der in einer SPL-Testsuite enthaltenen Testfälle nachträglich reduzieren lässt. Der erste Algorithmus entfernt redundante Testfälle, wohingegen der zweite und dritte Algorithmus eine Reduktion durch Ersetzen erreicht. Die Neuerung der vorgestellten Algorithmen liegt darin, dass die mögliche Existenz von variantenübergreifend verwendbaren Testfällen in einer SPL-Testsuite berücksichtigt wird. Das hat zur Folge, dass die erreichte Überdeckung bzgl. des gewählten Überdeckungskriteriums auf jeder Produktvariante trotz Reduktion erhalten bleibt.

Aufgrund beschränkter Ressourcen im Testprozess ist es in der Regel nicht möglich jede Produktvariante einer SPL zu testen, auch wenn die dafür benötigten Testfälle bereits in der vollständigen SPL-Testsuite vorliegen. Als Kompromiss wurde in dieser Arbeit eine Methode vorgestellt, mit der sich mittels einer vollständigen SPL-Testsuite eine kleine Teilmenge von Produktvarianten identifizieren lässt, die repräsentativ dafür ist, dass jeder in der SPL-Testsuite enthaltene Testfall mindestens auf einer dieser Produktvarianten verwendbar ist. Werden anschließend auf dieser Teilmenge all die in der vollständigen SPL-Testsuite enthaltenen Testfälle ausgeführt, lassen die gewonnenen Testergebnisse Rückschlüsse auf die Qualität der restlichen Produktvarianten zu.

Der Ansatz wurde durch zwei an den Kontext angepasste Fallbeispiele evaluiert. Dafür wurde das Testrahmenwerk Azmun [Has] eingesetzt, welches um eigene an den SPL-Kontext angepasste Komponenten erweitert wurde.

Im Folgenden wird ein Ausblick auf Probleme gegeben, die auftreten können, wenn einige in dieser Arbeit getroffenen Annahmen nicht zutreffen.

- Weiterentwicklung der SPL:
  In dieser Arbeit wird davon ausgegangen, dass die Entwicklung einer Software-Produktlinien abgeschlossen ist. Dies ist in der Praxis jedoch nicht zwingend der Fall. So können im Laufe der Zeit weitere Domänenartefakte bzw. Features entwickelt werden, um neue Produktvarianten zu ermöglichen. Diese Veränderung kann dazu führen, dass Testartefakte aus dem Domänentest angepasst oder neu erstellt werden müssen. Beispielsweise kann sich die Struktur des 150%-Testmodells verändern, aus dem eine vollständige SPL-Testsuite generiert wurde, was dazu führen könnte, dass die SPL-Testsuite nicht mehr vollständig ist. Um veraltete Testfälle effizient aktualisieren und fehlende Testfälle effizient ergänzen zu können, müssen entsprechende Techniken [ME08, DS11, LSKL12] in den Ansatz integriert werden.

- Dynamische Software-Produktlinien:
  In dieser Arbeit wird davon ausgegangen, dass sich die Produktkonfiguration eines zu testenden Produkts zur Laufzeit nicht ändern kann. Beim Einsatz dynamischen Software-Produktlinien [CPTC08, HHPS08, DMFM10] können die Produkte jedoch zur Laufzeit rekonfiguriert werden. Dabei stehen nur zulässige Produktkonfigurationen aus einer vorgegebenen Teilmenge aller Produktkonfigurationen einer SPL zur Auswahl. Da das in dieser Arbeit verwendete 150%-Testmodell keine Rekonfiguration bzw. erneutes Binden der Variabilität während der Testfallgenerierung unterstützt, können mit dem bisherigen Ansatz keine Konfigurationsübergänge getestet werden.

- Weitere Automaten-Konstrukte:
  Als theoretische Grundlage zur Modellierung der Testmodelle dienten in dieser Arbeit Zustandsautomaten, bei denen Konzepte wie Parallelität oder Hierarchien außer Acht gelassen wurden. Da diese Konzepte mehr Ausdrucksmöglichkeiten bieten und in der Praxis verwendet werden, sollten diese Konzepte in Verbindung mit dem vorgestellten Ansatz näher untersucht werden.

- Nicht-strukturelle Überdeckungskriterien:
  In dieser Arbeit wurden die von Testfällen abzudeckenden Testziele durch strukturelle Überdeckungskriterien, die auf das 150%-Testmodell angewendet wurden, definiert. Es existieren aber noch andere Überdeckungskriterien, die zur Definition von Testzielen auf das 150%-Testmodell angewendet werden könnten. Da es bei komplexeren Testzielen keineswegs mehr trivial ist, die Erfüllbarkeit eines solchen Testziels durch einen Testfall zu erkennen, wäre die Entwicklung von einer geeigneten Heuristik sinnvoll, mit der sich die Erfüllbarkeit vorhersagen lässt, um unnötige Kosten bei der Suche nach einem Testfall für ein nicht-erfüllbares Testziel zu verhindern. Dieses Problem ist ein generelles Problem im modellbasierten Test, wirkt sich aber aufgrund der vielen produktspezifischen Testmodelle im modellbasierten SPL-Test wesentlich stärker aus.

- Herleitung des 150%-Testmodells:
  In dieser Arbeit wird bei der Generierung der Testfälle davon ausgegangen, dass das dafür notwendige 150%-Testmodell bereits existiert. Auf die Herleitung dieses Modells wird jedoch nicht näher eingegangen. Aus diesem Grund sollten in der Praxis verwendbare Techniken [LSKL12, LLSG12] zur effizienten Konstruktion und Manipulation eines 150%-Testmodells weiter erforscht werden.

- Gezieltes Suchen nach Testfällen für vorgegebene Produktmengen:
  Mit der in dieser Arbeit vorgestellten Technik zur Generierung von Testfällen kann für einen zu generierenden Testfall nicht konkret eine Produktmenge angegeben werden, auf der dieser Testfall variantenübergreifend verwendbar sein soll. So kann ein Testfall zwar mit der bisherigen Technik auf mehreren Produktvarianten verwendbar sein, konkret diese Produktmenge zu fordern ist bisher aber nicht möglich. Es wäre jedoch hilfreich, wenn für einen zu generierenden Testfall eine solche Menge vorgegeben werden könnte, da dann gezielt nach diesem gesucht werden könnte, um die Testziele auf mehreren noch zu bearbeitenden Produktkonfigurationen abzudecken. Dadurch ließe sich bereits bei der Testfallgenerierung die Anzahl der zum Erreichen einer vollständigen SPL-Testsuite benötigten Testfälle reduzieren. Außerdem wäre es hilfreich, wenn bei der Testfallgenerierung die Forderung gestellt werden könnte, dass der zu generierende Testfall auf so vielen Produktkonfigurationen wie möglich verwendbar sein soll.

- Model-Checker als Testfallgenerator:
  In dieser Arbeit wurde der vorgestellte Ansatz durch die Verwendung des Testrahmenwerks Azmun in Verbindung mit dem Model-Checker SPIN evaluiert. Die Verwendung von Model-Checkern zur Testfallgenerierung ist in der Test-Community umstritten, da diese dafür nicht entwickelt wurden und deshalb diesbezüglich Schwächen aufweisen. Für Forschungsvorhaben und prototypische Implementierungen haben Model-Checker jedoch den Vorteil, dass sie die Verarbeitung eines jeden Modells erlauben, wenn sich dieses in ein PROMELA-Programm übertragen lässt, und Testfälle selbst für komplexe bzw. untypische Testziele erstellen können. Nach der Erprobungsphase im theoretischen Kontext sollte dennoch ein speziell für den Test-Kontext konzipierter Testfallgenerator zum Einsatz kommen.

Der in dieser Arbeit vorgestellte Ansatz hat sich in ersten durchgeführten Evaluationen bewährt. In weiterführenden Arbeiten wird im Rahmen des von der DFG geförderten IMoTEP-Projekts (Integrated Model-based Testing of Continuously Evolving Software Product Lines) insbesondere das Testen von dynamischen sowie evolvierenden Software-Produktlinien am Fachgebiet Echtzeitsysteme der TU Darmstadt in Kooperation mit der TU Braunschweig untersucht. Zur Evaluation zukünftiger Ansätze befindet sich bereits das Testfallgenerierungswerkzeug IMoTEP in der Entwicklung. Dieses wird von Grund auf neu für den SPL-Kontext entwickelt und soll Azmun ersetzen.

LITERATURVERZEICHNIS

[ABM98] Ammann, P. E. ; Black, P. E. ; Majurski, W.: Using Model Checking to Generate Tests from Specifications. In: *(ICFEM'98) Proceedings of the Second IEEE International Conference on Formal Engineering Methods*, IEEE, 1998, S. 46–54 (Zitiert auf Seite 22.)

[Bal98] Balzert, H.: *Lehrbuch der Software-Technik*. Bd. 2. Spektrum, Akademischer Verlag, 1998 (Zitiert auf Seite 7, 10 und 17.)

[Bat05] Batory, D.: Feature Models, Grammars, and Propositional Formulas. In: *(SPLC'05) Proceedings of the 9th International Software Product Line Conferences*, Springer, 2005 (LNCS), S. 7–20 (Zitiert auf Seite 37 und 47.)

[BCTT] BK-irst ; CMU ; The University of Genova ; The University of Trento: *NuSMV home page*. `http://nusmv.fbk.eu/`, Abruf: 29.07.2012. Webseite (Zitiert auf Seite 22, 95 und 107.)

[Beu] Beuth Verlag GmbH: *DIN EN ISO 9000:2005-12, Norm DIN EN ISO 9000:2005-12, Norm , Beuth Verlag GmbH*. `http://www.beuth.de/de/norm/din-en-iso-9000/82009580`, Abruf: 29.07.2012. Webseite (Zitiert auf Seite 8.)

[BFGL06] Bertolino, A. ; Fantechi, A. ; Gnesi, S. ; Lami, G.: Product Line Use Cases: Scenario-Based Specification and Testing of Requirements. In: *Software Product Lines - Research Issues in Engineering and Management*. Springer, 2006, Kapitel 11, S. 425–445 (Zitiert auf Seite 41.)

[BG03] Bertolino, A. ; Gnesi, S.: Use Case-based Testing of Product Lines. In: *SIGSOFT Software Engineering Notes* 28 (2003), Nr. 5, S. 355–358 (Zitiert auf Seite 41.)

[BHP03] Bühne, S. ; Halmans, G. ; Pohl, K.: Modelling Dependencies between Variation Points in Use Case Diagrams. In: *(REFSQ'03) Proceedings of 9th International Workshop on Requirements Engineering-Foundations for Software Quality*, 2003, S. 59–70 (Zitiert auf Seite 34.)

[Bin00] Binder, R.: *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Professional, 2000 (Zitiert auf Seite 13.)

[Bos01] Bosch, J.: Software Product Lines: Organizational Alternatives. In: *(ICSE'01) Proceedings of the 23rd International Conference on Software Engineering*, IEEE Computer Society, 2001, S. 91–100 (Zitiert auf Seite 31.)

[BR01] Black, P. E. ; Ranville, S.: Winnowing Tests: Getting Quality Coverage from a Model Checker without Quantity. In: *Proceedings of the*

20th Conference in Digital Avionics Systems Bd. 2, 2001, S. 9B6/1–9B6/4 (Zitiert auf Seite 28.)

[Bro05] Broy, M.: *Model-Based Testing of Reactive Systems*. Bd. 3472. Springer, 2005 (Zitiert auf Seite 11 und 18.)

[BSRC10] Benavides, D. ; Segura, S. ; Ruiz-Cortés, A.: Automated Analysis of Feature Models 20 Years Later: A Literature Review. In: *Information Systems* 35 (2010), Nr. 6, S. 615–636 (Zitiert auf Seite 37.)

[BTC] BTC Embedded Systems AG: *Rhapsody ATG - - BTC AG*. `http://www.btc-ag.com/de/3006.htm`, Abruf: 29.07.2012. Webseite (Zitiert auf Seite 41.)

[CA05] Czarnecki, K. ; Antkiewicz, M.: Mapping Features to Models: A Template Approach Based on Superimposed Variants. In: *(GPCE'05) Generative Programming and Component Engineering* Bd. 3676 Springer, 2005, S. 422–437 (Zitiert auf Seite 34, 36 und 40.)

[CDS06] Cohen, M.B. ; Dwyer, M.B. ; Shi, J.: Coverage and Adequacy in Software Product Line Testing. In: *(ROSATEA'06) Proceedings of the Workshop on Role of Software Architecture for Testing and Analysis* ACM, 2006, S. 53–63 (Zitiert auf Seite 42.)

[CDS07] Cohen, M. B. ; Dwyer, M. B. ; Shi, J.: Interaction Testing of Highly-Configurable Systems in the Presence of Constraints. In: *(ISSTA '07) Proceedings of the International Symposium on Software Testing and Analysis*, ACM, 2007, S. 129–139 (Zitiert auf Seite 42.)

[CE08] Clarke, E.M. ; Emerson, E.A.: Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In: *(OTM'05) Proceedings of the On the Move to Meaningful Internet Systems Workshop* Bd. 5000, Springer, 2008 (LNCS), S. 196–215 (Zitiert auf Seite 23.)

[CG08] Calvagna, A. ; Gargantini, A.: A Logic-Based Approach to Combinatorial Testing with Constraints. In: *(TAP'08) 2nd International Conference on Tests and Proofs* Bd. 4966, Springer, 2008 (LNCS), S. 66–83 (Zitiert auf Seite 22.)

[CHE04] Czarnecki, K. ; Helsen, S. ; Eisenecker, U.: Staged Configuration Using Feature Models. In: *(SPLC'04) Proceedings of 3rd International Conference on Software Product Lines* Bd. 3154, Springer, 2004 (LNCS), S. 162–164 (Zitiert auf Seite 34.)

[CHE05a] Czarnecki, K. ; Helsen, S. ; Eisenecker, U.: Formalizing Cardinality-based Feature Models and their Specialization. In: *Software Process: Improvement and Practice* 10 (2005), Nr. 1, S. 7–29 (Zitiert auf Seite 37.)

[CHE05b] Czarnecki, K. ; Helsen, S. ; Eisenecker, U.: Staged Configuration through Specialization and Multilevel Configuration of Feature Models. In: *Software Process: Improvement and Practice* 10 (2005), Nr. 2, S. 143–169 (Zitiert auf Seite 34 und 36.)

[CHW98] COPLIEN, J. ; HOFFMAN, D. ; WEISS, D.: Commonality and Variability in Software Engineering. In: *Software, IEEE* 15 (1998), Nr. 6, S. 37–45 (Zitiert auf Seite 32.)

[CL95] CHEN, TY ; LAU, MF: Heuristics Towards the Optimization of the Size of a Test Suite. In: *(ICSQ'95) Proceedings of the 3rd International Conference on Software Quality Management* Bd. 2, 1995, S. 415–424 (Zitiert auf Seite 26 und 27.)

[Cla08] CLARKE, E.M.: The Birth of Model Checking. In: *25 Years of Model Checking* Bd. 5000, Springer, 2008 (LNCS), S. 1–26 (Zitiert auf Seite 22.)

[CN01] CLEMENTS, P. ; NORTHROP, L.: *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001 (Zitiert auf Seite 32 und 34.)

[Com] COMPUTING SCIENCES RESEARCH CENTER AT BELL LABS: *Spin - Formal Verification*. `http://spinroot.com/spin/whatispin.html`, Abruf: 29.07.2012. Webseite (Zitiert auf Seite 95, 104, 107 und 108.)

[CPTC08] CETINA, C. ; PELECHANO, V. ; TRINIDAD, P. ; CORTES, A.: *An Architectural Discussion on DSPL*. 2008 (Zitiert auf Seite 114.)

[CSE96] CALLAHAN, J. ; SCHNEIDER, F. ; EASTERBROOK, S.: Automated Software Testing Using Model-Checking. In: *Proceedings of the 2nd SPIN Workshop* Bd. 353, 1996 (Zitiert auf Seite 22.)

[CSW08] CZARNECKI, K. ; SHE, S. ; WASOWSKI, A.: Sample Spaces and Feature Models: There and Back Again. In: *(SPLC'08) Proceedings of the 12th International Conference on Software Product Lines* IEEE, 2008, S. 22–31 (Zitiert auf Seite 37.)

[CXZN08] CHEN, Z. ; XU, B. ; ZHANG, X. ; NIE, C.: A Novel Approach for Test Suite Reduction based on Requirement Relation Contraction. In: *(SAC'08) Proceedings of the ACM Symposium on Applied Computing*, ACM, 2008, S. 390–394 (Zitiert auf Seite 26.)

[DMFM10] DINKELAKER, T. ; MITSCHKE, R. ; FETZER, K. ; MEZINI, M.: A Dynamic Software Product Line Approach Using Aspect Models at Runtime. In: *Proceedings of Workshop on Composition and Variability*, 2010 (Zitiert auf Seite 114.)

[DS11] DAMIANI, F. ; SCHAEFER, I.: Dynamic Delta-oriented Programming. In: *(SPLC'11) Proceedings of Software Product Line Conferences*, ACM, 2011, S. 34:1–34:8 (Zitiert auf Seite 114.)

[ER10] ENGSTRÖM, E. ; RUNESON, P.: A Qualitative Survey of Regression Testing Practices. In: *Product-Focused Software Process Improvement* Bd. 6156, Springer, 2010 (LNCS), S. 3–16 (Zitiert auf Seite 41.)

[ER11] ENGSTRÖM, E. ; RUNESON, P.: Software Product Line Testing - A Systematic Mapping Study. In: *Information and Software Technology* 53 (2011), Nr. 1, S. 2–13 (Zitiert auf Seite 2, 38 und 39.)

[ERW10] Engström, E. ; Runeson, P. ; Wikstrand, G.: An Empirical Evaluation of Regression Testing Based on Fix-cache Recommendations. In: *(ICST'10) In Proceedings of the Third International Conference on Software Testing, Verification and Validation , 2010*, IEEE, 2010, S. 75–78 (Zitiert auf Seite 41.)

[ESR08] Engström, E. ; Skoglund, M. ; Runeson, P.: Empirical Evaluations of Regression Test Selection Techniques - A Systematic Review. In: *(ESEM'08) Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* ACM, 2008, S. 22–31 (Zitiert auf Seite 41.)

[FFB02] Fey, D. ; Fajta, R. ; Boros, A.: Feature Modeling: A Meta-Model to Enhance Usability and Usefulness. In: *Software Product Lines* Bd. 2379, Springer, 2002 (LNCS), S. 198–216 (Zitiert auf Seite 34.)

[FM12] Fink, G. ; Marwedel, P.: Rechnerstrukturen / TU Dortmund. 2012. – Forschungsbericht (Zitiert auf Seite 1.)

[FW07] Fraser, G. ; Wotawa, F.: Redundancy Based Test-Suite Reduction. In: *Fundamental Approaches to Software Engineering* Bd. 4422, Springer, 2007 (LNCS), S. 291–305 (Zitiert auf Seite 25, 26 und 28.)

[FWA09] Fraser, G. ; Wotawa, F. ; Ammann, P.: Issues in using Model Checkers for Test Case Generation. In: *Journal of Systems and Software* 82 (2009), Nr. 9, S. 1403–1418 (Zitiert auf Seite 23.)

[Ger] German Testing Board e.V.: *German Testing Board*. `http://www.german-testing-board.info`, Abruf: 29.07.2012. Webseite (Zitiert auf Seite 7, 8, 11 und 41.)

[GH99] Gargantini, A. ; Heitmeyer, C.: Using Model Checking to Generate Tests from Requirements Specifications. In: *SIGSOFT Software Engineering Notes* 24 (1999), Nr. 6, S. 146–162 (Zitiert auf Seite 22.)

[GKPR08] Grönniger, H. ; Krahn, H. ; Pinkernell, C. ; Rumpe, B.: Modeling Variants of Automotive Systems using Views, 2008, S. 76 (Zitiert auf Seite 36 und 40.)

[Gom04] Gomaa, H.: *Designing Software Product Lines with UML*. Addison-Wesley, 2004 (Zitiert auf Seite 34, 36 und 40.)

[Gom05] Gomaa, H.: Designing Software Product Lines with UML. In: *29th Annual IEEE/NASA Software Engineering Workshop-Tutorial Notes* IEEE, 2005, S. 160–216 (Zitiert auf Seite 36 und 40.)

[GS05] Gaston, C. ; Seifert, D.: Evaluating Coverage Based Testing. In: *Model-Based Testing of Reactive Systems* Bd. 3472, Springer, 2005 (LNCS), S. 293–322 (Zitiert auf Seite 11.)

[Har87] Harel, David: Statecharts: A Visual Formalism for Complex Systems. In: *Sci. Comput. Program.* 8 (1987), Nr. 3, S. 231–274 (Zitiert auf Seite 45.)

[Has] Haschemi, S.: *Azmun - The Model-Based Testing Framework*. `http://www.azmun.de`, Abruf: 29.07.2012. Webseite (Zitiert auf Seite 5, 22, 95 und 113.)

[HCL+03] Hong, H. S. ; Cha, S. D. ; Lee, I. ; Sokolsky, O. ; Ural, H.: Data Flow Testing as Model Checking. In: *(ICSE'03) Proceedings of the 25th International Conference on Software Engineering*, IEEE, 2003, S. 232–242 (Zitiert auf Seite 22, 27, 78 und 85.)

[HGS93] Harrold, M. J. ; Gupta, R. ; Soffa, M. L.: A Methodology for Controlling the Size of a Test Suite. In: *ACM Transactions Software Engineering Methodologies* 2 (1993), Nr. 3, S. 270–285 (Zitiert auf Seite 26.)

[HHPS08] Hallsteinsen, S. ; Hinchey, M. ; Park, S. ; Schmid, K.: Dynamic Software Product Lines. In: *IEEE* 41 (2008), S. 93–95 (Zitiert auf Seite 114.)

[HLSC01] Hong, H. S. ; Lee, I. ; Sokolsky, O. ; Cha, S. D.: Automatic Test Generation from Statecharts Using Model Checking. In: *In Proceedings of FATES'01, Workshop on Formal Approaches to Testing of Software, volume NS-01-4 of BRICS Notes Series*, 2001, S. 15–30 (Zitiert auf Seite 22.)

[HMR04] Hamon, G. ; Moura, L. ; Rushby, J.: Generating Efficient Test Sets with a Model Checker. In: *Proceedings of the Software Engineering and Formal Methods*, IEEE, 2004, S. 261–270 (Zitiert auf Seite 26 und 28.)

[Hol03] Holzmann, G.: *The SPIN Model Checker: Primer and Reference Manual*. 1. Addison-Wesley Professional, 2003 (Zitiert auf Seite 5, 17, 22, 45, 66, 95, 104 und 108.)

[HRV01] Heimdahl, M. ; Rayadurgam, S. ; Visser, W.: Specification Centered Testing. In: *Proceedings of the Second International Workshop on Automated Program Analysis, Testing and Verification*, 2001 (Zitiert auf Seite 22.)

[HST+08] Heymans, P. ; Schobbens, P.Y. ; Trigaux, J.C. ; Bontemps, Y. ; Matulevicius, R. ; Classen, A.: Evaluating Formal Properties of Feature Diagram Languages. In: *Software, IET* 2 (2008), Nr. 3, S. 281–302 (Zitiert auf Seite 36.)

[HU05] Hong, H. S. ; Ural, H.: Using Model Checking for Reducing the Cost of Test Generation. In: *Formal Approaches to Software Testing* 3395 (2005), 110–124. `http://www.springerlink.com/content/f20uyfa60880nc67` (Zitiert auf Seite 22.)

[HVR04] Hartmann, J. ; Vieira, M. ; Ruder, A.: A UML-based Approach for Validating Product Lines. In: *(SPLiT'04) Proceedings of the International Workshop on Software Product Line Testing*, 2004, S. 58–65 (Zitiert auf Seite 40.)

[IB] IT-Beauftragte der Bundesregierung: *V-Modell*. http://www.v-modell.iabg.de/, Abruf: 29.07.2012. Webseite (Zitiert auf Seite 10 und 34.)

[IBM] IBM: *IBM Software - Rational Rhapsody*. http://www-01.ibm.com/software/awdtools/rhapsody/, Abruf: 29.07.2012. Webseite (Zitiert auf Seite 17 und 41.)

[IEE] IEEE: *IEEE SA - 610.12-1990 - IEEE Standard Glossary of Software Engineering Terminology*. http://standards.ieee.org/findstds/standard/610.12-1990.html, Abruf: 29.07.2012. Webseite (Zitiert auf Seite 9.)

[ISOa] ISO: *ISO/IEC 25000:2005 - Software Engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE*. http://www.iso.org/iso/catalogue_detail.htm?csnumber=35683, Abruf: 29.07.2012. Webseite (Zitiert auf Seite 7.)

[ISOb] ISO: *ISO/IEC 9126-1:2001 - Software engineering – Product quality – Part 1: Quality model*. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22749, Abruf: 29.07.2012. Webseite (Zitiert auf Seite 7.)

[JH03] Jones, J. A. ; Harrold, M. J.: Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage. In: *IEEE Transactions on Software Engineering* 29 (2003), Nr. 3, S. 195–209 (Zitiert auf Seite 26.)

[JJ05] Jard, C. ; Jeron, T.: TGV: Theory, Principles and Algorithms: A Tool for the Automatic Synthesis of Conformance Test Cases for Non-Deterministic Reactive Systems. In: *(STTT'05) International Journal on Software Tools for Technology Transfer* 7 (2005), Nr. 4, S. 297–315 (Zitiert auf Seite 23.)

[JKM08] Jain, T. ; Kushwaha, D. ; Misra, A.: Optimization of the Quine-McCluskey Method for the Minimization of the Boolean Expressions. In: *(ICAS'08) Proceedings of the 4th International Conference on Autonomic and Autonomous Systems*, IEEE, 2008, S. 165–168 (Zitiert auf Seite 67.)

[KBK11] Kim, C.H.P. ; Batory, D.S. ; Khurshid, S.: Reducing Combinatorics in Testing Product Lines. In: *(AOSD'11) Proceedings of the 10th International Conference on Aspect-Oriented Software Development* ACM, 2011, S. 57–68 (Zitiert auf Seite 41.)

[KCH+90] Kang, K. C. ; Cohen, S. G. ; Hess, J. A. ; Novak, W. E. ; Peterson, A. S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study / Carnegie-Mellon University Software Engineering Institute. 1990. – Forschungsbericht (Zitiert auf Seite 34, 36 und 47.)

[KKT07] Korel, B. ; Koutsogiannakis, G. ; Tahat, L. H.: Model-Based Test Prioritization Heuristic Methods and their Evaluation. In: *(A-MOST*

'07) In Proceedings of the Third International Workshop on Advances in Model-Based Testing, ACM, 2007, S. 34–43 (Zitiert auf Seite 29.)

[KLD02] KANG, K.C. ; LEE, J. ; DONOHOE, P.: Feature-Oriented Product Line Engineering. In: *Software, IEEE* 19 (2002), Nr. 4, S. 58–65 (Zitiert auf Seite 34.)

[Kol03] KOLB, R.: A Risk-Driven Approach for Efficiently Testing Software Product Lines. In: *Net.ObjectDays* (2003), S. 409–414 (Zitiert auf Seite 43.)

[LG08] LUNA, C. ; GONZALEZ, A.: Behavior Specification of Product Lines via Feature Models and UML Statecharts with Variabilities. In: *(SCCC'08) International Conference of the Chilean Computer Science Society* IEEE, 2008, S. 32–41 (Zitiert auf Seite 40.)

[Lig02] LIGGESMEYER, P.: *Software-Qualität - Testen, Analysieren und Verifizieren von Software*. Spektrum, Akademischer Verlag, 2002 (Zitiert auf Seite 1, 9, 11 und 12.)

[LL07] LUDEWIG, J. ; LICHTER, H.: *Software Engineering - Grundlagen, Menschen, Prozesse, Techniken*. 1. dpunkt.verlag, 2007 (Zitiert auf Seite 9 und 10.)

[LLSG12] LITY, S. ; LOCHAU, M. ; SCHÄFER, I. ; GOLTZ, U.: Delta-oriented Model-based SPL Regression Testing. In: *(PLEASE'12) In Proceedings of the Third International Workshop on Product Line Approaches in Software Engineering* IEEE, 2012, S. 53–56 (Zitiert auf Seite 2, 40 und 115.)

[LSKL12] LOCHAU, M. ; SCHAEFER, I. ; KAMISCHKE, J. ; LITY, S.: Incremental Model-Based Testing of Delta-Oriented Software Product Lines. In: *Tests and Proofs* Bd. 7305, Springer, 2012 (LNCS), S. 67–82 (Zitiert auf Seite 2, 40, 114 und 115.)

[McG01] MCGREGOR, J. D.: Testing a Software Product Line / Carnegie Mellon, Software Engineering Institute. 2001. – Forschungsbericht (Zitiert auf Seite 2, 39 und 42.)

[ME08] MITSCHKE, R. ; EICHBERG, M.: Supporting the Evolution of Software Product Lines. In: *(ECMDA-TW) Proceedings of ECMDA-Traceability Workshop*, 2008, S. 87–96 (Zitiert auf Seite 114.)

[MG08] MÜLLER-GLASER, K.D.: Modellbasierter Test von Kfz-Steuergeräten - Chancen und Herausforderungen / Universität Karlsruhe. 2008. – Forschungsbericht (Zitiert auf Seite 1.)

[MGEW12] MLYNARSKI, M. ; GÜLDALI, B. ; ENGELS, G. ; WEISSLEDER, S.: Model-Based Testing: Achievements and Future Challenges. In: *Advances in Computers* 86 (2012), S. 2–35 (Zitiert auf Seite 15.)

[MHP+07] METZGER, A. ; HEYMANS, P. ; POHL, K. ; SCHOBBENS, P.Y. ; SAVAL, G.: Disambiguating the Documentation of Variability in Software Product

Lines: A Separation of Concerns, Formalization and Automated Analysis. In: *(RE'07) IEEE International Requirements Engineering Conference*, IEEE, 2007, S. 243–253 (Zitiert auf Seite 34.)

[ML02] Massen, T. von d. ; Lichter, H.: Modeling Variability by UML Use Case Diagrams. In: *Proceedings of the International Workshop on Requirements Engineering for Product Lines*, 2002, S. 19–25 (Zitiert auf Seite 34.)

[MLD+09] Müller, T. ; Lochau, M. ; Detering, S. ; Saust, F. ; Garbers, H. ; Märtin, L.: A Comprehensive Description of a Model-Based, Continuous Development Process for AUTOSAR Systems with Integrated Quality Assurance / TU Braunschweig. 2009 (06). – Forschungsbericht (Zitiert auf Seite 5.)

[NFLTJ04] Nebut, C. ; Fleurey, F. ; Le Traon, Y. ; Jézéquel, J.M.: A Requirement-Based Approach to Test Product Families. In: *Software Product-Family Engineering* Bd. 3014, Springer, 2004 (LNCS), S. 198–210 (Zitiert auf Seite 41.)

[NLTJ06] Nebut, C. ; Le Traon, Y. ; Jézéquel, J.M.: System Testing of Product Lines: From Requirements to Test Cases. In: *Software Product Lines Conference*. Springer, 2006, S. 447–477 (Zitiert auf Seite 41.)

[NoM] NoMagic: *MagicDraw*. `http://www.nomagic.com/products/magicdraw.html`, Abruf: 29.07.2012. Webseite (Zitiert auf Seite 17 und 95.)

[OG08] Olimpiew, E.M. ; Gomaa, H.: Model-based Test Design for Software Product Lines. In: *(SPLiT'08) In Proceedings of the Fifth International Workshop on Software Product Line Testing*, 2008, S. 40 (Zitiert auf Seite 40.)

[Oli08] Olimpiew, E. M.: *Model-Based Testing for Software Product Lines*, George Mason University, Diss., 2008 (Zitiert auf Seite 2 und 40.)

[OLZG11] Oster, S. ; Lochau, M. ; Zink, M. ; Grechanik, M.: Pairwise Feature-Interaction Testing for SPLs: Potentials and Limitations. In: *Proceeding of the International Software Product Line Conference Workshop FOSD*, 2011 (Zitiert auf Seite 5.)

[OMR10] Oster, S. ; Markert, F. ; Ritter, P.: Automated Incremental Pairwise Testing of Software Product Lines. In: *(SPLC'10) Proceedings of the 14th International Software Product Line Conference*, 2010, S. 196–210 (Zitiert auf Seite 2, 40, 41, 42 und 72.)

[OSG] OSGi Alliance: *OSGi Alliance | Main / OSGi Alliance*. `http://www.osgi.org/Main/HomePage`, Abruf: 29.07.2012. Webseite (Zitiert auf Seite 95.)

[Ost12] Oster, S.: *Feature Model-based Software Product Line Testing*. Darmstadt, TU Darmstadt, Diss., 2012 (Zitiert auf Seite 36, 37, 40, 41 und 42.)

[OWES11] Oster, S. ; Wübbeke, A. ; Engels, G. ; Schürr, A.: Model-Based Software Product Lines Testing Survey. In: *Model-based Testing for Embedded Systems*. CRC Press/Taylor&Francis, 2011 (Zitiert auf Seite 1 und 38.)

[PBL05] Pohl, K. ; Böckle, G. ; Linden, F. van d.: *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005 (Zitiert auf Seite 1, 2, 31, 32, 33, 34, 38 und 39.)

[PK10] Parsa, S. ; Khalilian, A.: On The Optimization Approach Towards Test Suite Minimization. In: *International Journal of Software Engineering and Its Applications* 4 (2010), Nr. 1, S. 15–27 (Zitiert auf Seite 26.)

[PM95] Parnas, D. L. ; Madey, J.: Functional Documents for Computer Systems. In: *Science of Computer Programming* 25 (1995), Nr. 1, S. 41–61 (Zitiert auf Seite 16.)

[Pnu77] Pnueli, A.: The temporal logic of programs. In: *(FOCS'77) 18th Annual Symposium on Foundations of Computer Science*, IEEE, 1977, S. 46–57 (Zitiert auf Seite 23.)

[PP05] Pretschner, A. ; Philipps, J.: Methodological Issues in Model-Based Testing. In: *Model-Based Testing of Reactive Systems* Bd. 3472, Springer, 2005 (LNCS), S. 281–292 (Zitiert auf Seite 14 und 15.)

[PS] Pure-Systems: *pure-systems GmbH*. `http://www.pure-systems.com`, Abruf: 29.07.2012. Webseite (Zitiert auf Seite 41.)

[PSK+10] Perrouin, G. ; Sen, S. ; Klein, J. ; Baudryand, B. ; Traon, Y.: Automated and Scalable T-wise Test Case Generation Strategies for Software Product Lines. In: *ICST'10 Proceedings of the International Conference on Software Testing*, 2010, S. 459–468 (Zitiert auf Seite 42.)

[RBGW10] Rossner, T. ; Brandes, C. ; Götz, H. ; Winter, M.: *Basiswissen Modellbasierter Test*. 1. dpunkt.verlag, 2010 (Zitiert auf Seite 9, 13, 14, 15, 16, 19, 20 und 22.)

[RBSP02] Riebisch, M. ; Böllert, K. ; Streitferdt, D. ; Philippow, I.: Extending Feature Diagrams with UML Multiplicities. In: *(IDPT'02) In Proceedings of the sixth Conference on Integrated Design and Process Technology*, 2002 (Zitiert auf Seite 37.)

[RHOH98] Rothermel, G. ; Harrold, M. J. ; Ostrin, J. ; Hong, C.: An Empirical Study of the Effects of Minimization on the Fault Detection Capabilities of Test Suites. In: *(ICSM '98) Proceedings of the International Conference on Software Maintenance*, IEEE, 1998, S. 34–43 (Zitiert auf Seite 26, 27 und 28.)

[RHRH02] Rothermel, G. ; Harrold, M. J. ; Ronne, J. von ; Hong, C.: Empirical Studies of Test-Suite Reduction. In: *Journal of Software Testing, Verification, and Reliability* 12 (2002), S. 219–249 (Zitiert auf Seite 25 und 26.)

[RKPR05] Reuys, A. ; Kamsties, E. ; Pohl, K. ; Reis, S.: Model-based System Testing of Software Product Families. In: *Advanced Information Systems Engineering* Bd. 3520, Springer, 2005 (LNCS), S. 379–380 (Zitiert auf Seite 2, 39 und 40.)

[SB] Skrypuch, N. ; Boldt, N.: *Eclipse Modeling - EMFT - Home*. `http://www.eclipse.org/modeling/emft/?project=mwe`, Abruf: 29.07.2012. Webseite (Zitiert auf Seite 95.)

[Sch07a] Scheidemann, Kathrin: *Verifying Families of System Configurations*, TU München, Diss., 2007 (Zitiert auf Seite 2 und 43.)

[Sch07b] Schieferdecker, I.: Modellbasiertes Testen. In: *OBJEKTspektrum* 3/07 (2007), S. 39–45 (Zitiert auf Seite 17.)

[Skra] Skrypuch, N.: *Eclipse Modeling - M2T - Home*. `http://www.eclipse.org/modeling/m2t/?project=xpand`, Abruf: 29.07.2012. Webseite (Zitiert auf Seite 96.)

[Skrb] Skrypuch, N.: *Eclipse Modeling - MDT - Home*. `http://www.eclipse.org/modeling/mdt/`, Abruf: 29.07.2012. Webseite (Zitiert auf Seite 95.)

[SL10] Spillner, A. ; Linz, T.: *Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester*. 4. dpunkt.verlag, 2010 (Zitiert auf Seite 1, 7, 8, 9 und 10.)

[SMFM00] Souza, S. ; Maldonado, J. ; Fabbri, S. ; Masiero, P.: Statecharts Specifications: A Family of Coverage Testing Criteria. In: *CLEI Electronic Journal*, 2000 (Zitiert auf Seite 21.)

[Sta73] Stachowiak, H.: *Allgemeine Modelltheorie*. Springer, 1973 (Zitiert auf Seite 13.)

[SW81] Smith, T. F. ; Waterman, M. S.: Identification of Common Molecular Subsequences. In: *Journal of Molecular Biology* 147 (1981), Nr. 1, S. 195–197 (Zitiert auf Seite 86.)

[TG05] Tallam, S. ; Gupta, N.: A Concept Analysis Inspired Greedy Algorithm for Test Suite Minimization. In: *(PASTE '05) Proceedings of the sixth ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ACM, 2005, S. 35–42 (Zitiert auf Seite 26.)

[TTK04] Tevanlinna, A. ; Taina, J. ; Kauppinen, R.: Product Family Testing: A Survey. In: *ACM SIGSOFT Software Engineering Notes* 29 (2004), Nr. 2, S. 12–12 (Zitiert auf Seite 1, 3 und 58.)

[UL07] Utting, M. ; Legeard, B.: *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., 2007 (Zitiert auf Seite 15, 17 und 20.)

[UPL11] UTTING, M. ; PRETSCHNER, A. ; LEGEARD, B.: A Taxonomy of Model-Based Testing Approaches. In: *Software Testing, Verification and Reliability* (2011) (Zitiert auf Seite 65 und 66.)

[VDLSR07] VAN DER LINDEN, F. ; SCHMID, K. ; ROMMES, E.: *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer, 2007 (Zitiert auf Seite 1, 32 und 33.)

[Ver00] VERSTEEGEN, G.: *Projektmanagement mit dem Rational Unified Process*. Springer, 2000 (Zitiert auf Seite 34.)

[VW] VW: *Volkswagen Deutschland*. `http://www.volkswagen.de`, Abruf: 29.07.2012. Webseite (Zitiert auf Seite 5.)

[Was04] WASOWSKI, A.: Automatic Generation of Program Families by Model Restrictions. In: *Software Product Lines* Bd. 3154, Springer, 2004, S. 196–198 (Zitiert auf Seite 2, 36 und 40.)

[WDS09] WHITE, J. ; DOUGHERTY, B. ; SCHMIDT, D.C.: Selecting Highly Optimal Architectural Feature Sets with Filtered Cartesian Flattening. In: *Journal of Systems and Software* 82 (2009), Nr. 8, S. 1268–1284 (Zitiert auf Seite 37.)

[WE09] WALTER, T. ; EBERT, J.: Combining DSLs and Ontologies using Metamodel Integration. In: *Domain-Specific Languages* Bd. 5658, Springer, 2009 (LNCS), S. 148–169 (Zitiert auf Seite 36.)

[Wey82] WEYUKER, E. J.: On Testing Non-testable Programs. In: *The Computer Journal* 25 (1982), Nr. 4, S. 465 (Zitiert auf Seite 11.)

[WHLM95] WONG, W. E. ; HORGAN, J. R. ; LONDON, S. ; MATHUR, A. P.: Effect of Test Set Minimization on Fault Detection Effectiveness . In: *(ICSE '95) Proceedings of the 17th International Conference on Software Engineering*, ACM, 1995, S. 41–50 (Zitiert auf Seite 26 und 27.)

[WL99] WEISS, D. M. ; LAI, C. T. R.: *Software Product-Line Engineering: A Family-Based Software Engineering Process*. Addison-Wesley, 1999 (Zitiert auf Seite 32.)

[WSS08] WEISSLEDER, S. ; SOKENOU, D. ; SCHLINGLOFF, H.: Reusing State Machines for Automatic Test Generation in Product Lines. In: *(ECMDA'08) European Conference on Modelling Foundations and Applications*, 2008 (Zitiert auf Seite 2 und 40.)

[Wüb10] WÜBBEKE, A.: *Variabilitätsmanagement in Anforderungs- und Testfallspezifikation für Software-Produktlinien*, University of Paderborn, Diss., 2010 (Zitiert auf Seite 41.)

[YH08] YOO, S. ; HARMAN, M.: Regression Testing Minimization, Selection and Prioritization: A Survey. In: *Software Testing, Verification and Reliability* (2008) (Zitiert auf Seite 25 und 74.)

[ZML07] Zeng, H. ; Miao, H. ; Liu, J.: Specification-based Test Generation and Optimization Using Model Checking. In: *TASE '07: Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering*. Washington, DC, USA : IEEE, 2007. – ISBN 0–7695–2856–2, S. 349–355 (Zitiert auf Seite 22.)

[ZZM06] Zhong, H. ; Zhang, L. ; Mei, H.: An Experimental Comparison of four Test Suite Reduction Techniques. In: *(ICSE '06) Proceedings of the 28th International Conference on Software Engineering*, ACM, 2006, S. 636–640 (Zitiert auf Seite 26.)

A

ANHANG

PC	Features				Testziele (Transitionspaare)																											
	B	C	D	G	*ab*	*ac*	*ad*	*ae*	*av*	*bc*	*bd*	*be*	*bf*	*bs*	*bv*	*cb*	*cd*	*ce*	*cf*	*cs*	*cv*	*db*	*dc*	*df*	*ds*	*dv*	*eb*	*ec*	*ef*	*es*	*ev*
fa1	0	0	0	0	✓	✓			✓	✓			✓	✓	✓	✓			✓	✓	✓										
fa2	0	0	0	1	✓	✓		✓	✓	✓		✓	✓	✓	✓	✓		✓	✓	✓	✓						✓	✓	✓	✓	✓
fa3	0	0	1	0	✓	✓	✓		✓	✓	✓		✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓					
fa4	0	1	0	0	✓	✓			✓	✓			✓	✓	✓	✓			✓	✓	✓										
fa5	0	1	0	1	✓	✓		✓	✓	✓		✓	✓	✓	✓	✓		✓	✓	✓	✓						✓	✓	✓	✓	✓
fa6	0	1	1	0	✓	✓	✓		✓	✓	✓		✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓					
fa7	1	0	0	1	✓	✓		✓	✓	✓		✓	✓	✓	✓	✓		✓	✓	✓	✓						✓	✓	✓	✓	✓
fa8	1	1	0	1	✓	✓		✓	✓	✓		✓	✓	✓	✓	✓		✓	✓	✓	✓						✓	✓	✓	✓	✓

PC	Features				Testziele (Transitionspaare)																											
	B	C	D	G	*fg*	*fh*	*fi*	*gh*	*gi*	*gj*	*hg*	*hi*	*hj*	*it*	*iu*	*jk*	*kl*	*lm*	*ln*	*lo*	*lp*	*mn*	*mo*	*mp*	*mq*	*mr*	*no*	*np*	*nq*	*nr*	*oq*
fa1	0	0	0	0	✓		✓		✓	✓				✓	✓	✓	✓	✓	✓			✓			✓				✓		
fa2	0	0	0	1	✓		✓		✓	✓				✓	✓	✓	✓	✓	✓		✓	✓		✓	✓			✓	✓		
fa3	0	0	1	0	✓		✓		✓	✓				✓	✓	✓	✓	✓	✓	✓		✓	✓		✓		✓		✓		✓
fa4	0	1	0	0	✓		✓		✓	✓				✓	✓	✓	✓	✓	✓			✓				✓				✓	
fa5	0	1	0	1	✓		✓		✓	✓				✓	✓	✓	✓	✓	✓		✓	✓		✓		✓		✓		✓	
fa6	0	1	1	0	✓		✓		✓	✓				✓	✓	✓	✓	✓	✓	✓		✓	✓			✓	✓			✓	
fa7	1	0	0	1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓		✓	✓			✓	✓		
fa8	1	1	0	1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓		✓		✓		✓		✓	

PC	Features				Testziele (Transitionspaare)																											
	B	C	D	G	*or*	*pq*	*pr*	*qb*	*qc*	*qd*	*qe*	*qv*	*rt*	*ru*	*su*	*tu*	*ub*	*uc*	*ud*	*ue*	*uv*	*vw*	*vx*	*wx*	*xb*	*xc*	*xd*	*xe*	*xf*	*xs*	*xv*
fa1	0	0	0	0				✓	✓			✓			✓	✓	✓	✓			✓	✓	✓	✓	✓	✓			✓	✓	✓
fa2	0	0	0	1		✓		✓	✓		✓	✓			✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓
fa3	0	0	1	0				✓	✓	✓		✓			✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓		✓	✓	✓
fa4	0	1	0	0										✓	✓	✓	✓	✓			✓	✓	✓	✓	✓	✓			✓	✓	✓
fa5	0	1	0	1			✓							✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓
fa6	0	1	1	0	✓									✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓		✓	✓	✓
fa7	1	0	0	1		✓		✓	✓		✓	✓			✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓
fa8	1	1	0	1			✓						✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓

Abbildung A.1: Vorhandene Testziele (*all-transition-pairs*) in den produktspezifischen Testmodellen

tc#	Features				Testziel	Transitionspfad
	B	C	D	G		
1	-	-	-	-	*ab*	*a b*
2	-	-	-	-	*ac*	*a c*
3	-	-	1	0	*ad*	*a d*
4	-	-	0	1	*ae*	*a e*
5	-	-	-	-	*av*	*a v*
6	-	-	-	-	*bc*	*a b c*
7	-	-	1	0	*bd*	*a b d*
8	-	-	0	1	*be*	*a b e*
9	-	-	-	-	*bf*	*a b f*
10	-	-	-	-	*bs*	*a b s*
11	-	-	-	-	*bv*	*a b v*
12	-	-	-	-	*cb*	*a c b*
13	-	-	1	0	*cd*	*a c d*
14	-	-	0	1	*ce*	*a c e*
15	-	-	-	-	*cf*	*a c f*
16	-	-	-	-	*cs*	*a c s*
17	-	-	-	-	*cv*	*a c v*
18	-	-	1	0	*db*	*a d b*
19	-	-	1	0	*dc*	*a d c*
20	-	-	1	0	*df*	*a d f*

tc#	Features				Testziel	Transitionspfad
	B	C	D	G		
21	-	-	1	0	*ds*	*a d s*
22	-	-	1	0	*dv*	*a d v*
23	-	-	0	1	*eb*	*a e b*
24	-	-	0	1	*ec*	*a e c*
25	-	-	0	1	*ef*	*a e f*
26	-	-	0	1	*es*	*a e s*
27	-	-	0	1	*ev*	*a e v*
28	-	-	-	-	*fg*	*a b f g*
29	1	-	-	-	*fh*	*a b f h*
30	-	-	-	-	*fi*	*a b f i*
31	1	-	-	-	*gh*	*a c b f g h*
32	-	-	-	-	*gi*	*a b f g i*
33	-	-	-	-	*gj*	*a c f g j*
34	1	-	0	1	*hg*	*a c c e f h g*
35	1	-	0	1	*hi*	*a c c e f h i*
36	1	-	-	-	*hj*	*a b f h j*
37	-	-	1	0	*it*	*a d f g i t*
38	-	-	-	-	*it*	*a c b f g i t*
39	-	-	1	0	*iu*	*a d f i u*
40	-	-	-	-	*iu*	*a c f i u*

tc#	Features				Testziel	Transitionspfad
	B	C	D	G		
41	-	-	-	-	*jk*	*a c f g j k*
42	-	-	-	-	*kl*	*a c f g j k k k k k k k k k k k l*
43	1	-	-	-	*lm*	*a c f i u b f h j k k k k k k k k k k k l m*
44	-	-	-	-	*lm*	*a c f i u b f g g j k k k k k k k k k k k l m*
45	-	-	-	-	*ln*	*a c f g j k k k k k k k k k k k l n*
46	-	-	1	0	*lo*	*a d f g g g g j k k k k k k k k k k k l o*
47	1	-	0	1	*lp*	*a c f i u e f g g h j k k k k k k k k k k k l p*
48	-	-	0	1	*lp*	*a c f i u e f g g g g g g g j k k k k k k k k k k k l p*
49	1	-	-	-	*mn*	*a c f i u b c f h j k k k k k k k k k k k l m n*
50	-	-	-	-	*mn*	*a c f i u b c f g g g g j k k k k k k k k k k k k l m n*
51	-	-	1	0	*mo*	*a d f g g g i t t t u b d f g g g g g g g j k k k k k k k k k k k k l m o*
52	1	-	0	1	*mp*	*a c f i u e b f g g g g g h j k k k k k k k k k k k k l m p*
53	-	-	0	1	*mp*	*a c f i u e b f g g g g g g g g g j k k k k k k k k k k k k l m p*
54	-	0	-	-	*mq*	*a c f i u b f g g g j k k k k k k k k k k k k l m q*
55	-	1	-	-	*mr*	*a c f i u b f g g g j k k k k k k k k k k k k l m r*
56	-	-	1	0	*no*	*a d f g g g i t t t u c d f g g g g g g j k k k k k k k k k k k k l n o*
57	1	-	0	1	*np*	*a c f i u e c f g g g g h j k k k k k k k k k k k k l n p*
58	-	-	0	1	*np*	*a c f i u e c f g g g g g g g g j k k k k k k k k k k k k l n p*
59	-	0	-	-	*nq*	*a c f g j k k k k k k k k k k k k l n q*
60	-	1	-	-	*nr*	*a c f g j k k k k k k k k k k k k l n r*
61	-	0	1	0	*oq*	*a d d d f g g g g g g g g g g g g g g j k k k k k k k k k k k k k l o o o o q*
62	-	1	1	0	*or*	*a d f g g g g j k k k k k k k k k k k k l o r*
63	-	0	0	1	*pq*	*a c f i u e f g g g g g g g j k k k k k k k k k k k k l p q*
64	-	1	0	1	*pr*	*a c f i u e f g g g g g g g j k k k k k k k k k k k k l p r*
65	-	0	-	-	*qb*	*a c f g j k k k k k k k k k k k k l n q b*
66	-	0	-	-	*qc*	*a c f g j k k k k k k k k k k k k l n q c*
67	-	0	1	0	*qd*	*a d d f g g g g g g g g g g j k k k k k k k k k k k k l o o q d*
68	-	0	0	1	*qe*	*a c f g j k k k k k k k k k k k k l n q e*
69	-	0	-	-	*qv*	*a c f g j k k k k k k k k k k k k l n q v*
70	1	1	-	-	*rt*	*a c c c f h j k k k k k k k k k k k k k k l n n n r t*
71	-	1	-	-	*ru*	*a c f g j k k k k k k k k k k k k l n r u*
72	-	-	1	0	*su*	*a d d d d d d d c c c s u*
73	-	-	-	-	*su*	*a c f i u b c c c c c s u*
74	-	-	1	0	*tu*	*a d d f g i t u*
75	-	-	-	-	*tu*	*a c c f g i t u*
76	-	-	1	0	*ub*	*a d f i u b*
77	-	-	-	-	*ub*	*a c f i u b*
78	-	-	1	0	*uc*	*a d f i u c*
79	-	-	-	-	*uc*	*a c f i u c*
80	-	-	1	0	*ud*	*a d f i u d*
81	-	-	0	1	*ue*	*a c f i u e*
82	-	-	1	0	*uv*	*a d f i u v*
83	-	-	-	-	*uv*	*a c f i u v*
84	-	-	-	-	*vw*	*a v w*
85	-	-	1	0	*vx*	*a d v w w w w w w w w w x v x*
86	-	-	-	-	*vx*	*a c v w w w w w w w w w w x v x*
87	-	-	1	0	*wx*	*a d v w w w w w w w w w w x*
88	-	-	-	-	*wx*	*a c v w w w w w w w w w w x*
89	-	-	1	0	*xb*	*a d v w w w w w w w w w w x b*
90	-	-	-	-	*xb*	*a c v w w w w w w w w w w x b*
91	-	-	1	0	*xc*	*a d v w w w w w w w w w w x c*
92	-	-	-	-	*xc*	*a c v w w w w w w w w w w x c*
93	-	-	1	0	*xd*	*a d d d d d v w w w w w w w w w w x d*
94	-	-	0	1	*xe*	*a c v w w w w w w w w w w x e*
95	-	-	1	0	*xf*	*a d v w w w w w w w w w w x f*
96	-	-	-	-	*xf*	*a c v w w w w w w w w w w x f*
97	-	-	1	0	*xs*	*a d v w w w w w w w w w w x s*
98	-	-	-	-	*xs*	*a c v w w w w w w w w w w x s*
99	-	-	1	0	*xv*	*a d v w w w w w w w w w w x v*
100	-	-	-	-	*xv*	*a c v w w w w w w w w w w x v*

Abbildung A.2: Vollständige FA-SPL-Testsuite (*all-transition-pairs*)

tc#	Features				Transitionspfad
	B	C	D	G	
8	-	-	0	1	*a b e*
10	-	-	-	-	*a b s*
11	-	-	-	-	*a b v*
14	-	-	0	1	*a c e*
18	-	-	1	0	*a d b*
21	-	-	1	0	*a d s*
26	-	-	0	1	*a e s*
27	-	-	0	1	*a e v*
34	1	-	0	1	*a c c e f h g*
35	1	-	0	1	*a c c e f h i*
38	-	-	-	-	*a c b f g i t*
50	-	-	-	-	*a c f i u b c f g g g j k k k k k k k k k k k k l m n*
51	-	-	1	0	*a d f g g g i t t t u b d f g g g g g g j k k k k k k k k k k k l m o*
53	-	-	0	1	*a c f i u e b f g g g g g g g g g j k k k k k k k k k k k k l m p*
54	-	0	-	-	*a c f i u b f g g j k k k k k k k k k k k l m q*
55	-	1	-	-	*a c f i u b f g g j k k k k k k k k k k k l m r*
56	-	-	1	0	*a d f g g g i t t t u c d f g g g g g g j k k k k k k k k k k k l n o*
57	1	-	0	1	*a c f i u e c f g g g h j k k k k k k k k k k k k l n p*
58	-	-	0	1	*a c f i u e c f g g g g g g g g j k k k k k k k k k k k k l n p*
62	-	1	1	0	*a d f g g g g g j k k k k k k k k k k k k l o r*
63	-	0	0	1	*a c f i u e f g g g g g g g j k k k k k k k k k k k k l p q*
64	-	1	0	1	*a c f i u e f g g g g g g g j k k k k k k k k k k k k l p r*
65	-	0	-	-	*a c f g j k k k k k k k k k k k l n q b*
66	-	0	-	-	*a c f g j k k k k k k k k k k k l n q c*
67	-	0	1	0	*a d d f g g g g g g g g g j k k k k k k k k k k k k l o o q d*
68	-	0	0	1	*a c f g j k k k k k k k k k k k l n q e*
69	-	0	-	-	*a c f g j k k k k k k k k k k k l n q v*
70	1	1	-	-	*a c c c f h j k k k k k k k k k k k k k k l n n n r t*
71	-	1	-	-	*a c f g j k k k k k k k k k k k l n r u*
72	-	-	1	0	*a d d d d d d d c c c s u*
73	-	-	-	-	*a c f i u b c c c c c s u*
75	-	-	-	-	*a c c f g i t u*
79	-	-	-	-	*a c f i u c*
80	-	-	1	0	*a d f i u d*
83	-	-	-	-	*a c f i u v*
84	-	-	-	-	*a v w*
86	-	-	-	-	*a c v w w w w w w w w w w x v x*
90	-	-	-	-	*a c v w w w w w w w w w w x b*
92	-	-	-	-	*a c v w w w w w w w w w w x c*
93	-	-	1	0	*a d d d d v w w w w w w w w w x d*
94	-	-	0	1	*a c v w w w w w w w w w x e*
96	-	-	-	-	*a c v w w w w w w w w w x f*
98	-	-	-	-	*a c v w w w w w w w w w x s*

(a) Transitionspfade

PC	Features				Testfälle (tc#)																				
	B	C	D	G	8	10	11	14	18	21	26	27	34	35	38	50	51	53	54	55	56	57	58	62	63	64
fa1	0	0	0	0		☑	☑								☑	☑			☑							
fa2	0	0	0	1	☑	☑	☑	☑			☑	☑			☑	☑		☑	☑				☑		☑	
fa3	0	0	1	0		☑	☑		☑	☑					☑	☑	☑		☑		☑					
fa4	0	1	0	0		☑	☑								☑	☑				☑						
fa5	0	1	0	1	☑	☑	☑	☑			☑	☑			☑	☑		☑		☑			☑			☑
fa6	0	1	1	0		☑	☑		☑	☑					☑	☑	☑			☑	☑			☑		
fa7	1	0	0	1	☑	☑	☑	☐			☑	☑	☑	☑	☑	☑		☑	☑			☑	☐		☑	
fa8	1	1	0	1	☑	☑	☑	☐			☑	☑	☑	☑	☑	☑		☑		☑		☑	☐			☑

PC	Features				Testfälle (tc#)																				
	B	C	D	G	65	66	67	68	69	70	71	72	73	75	79	80	83	84	86	90	92	93	94	96	98
fa1	0	0	0	0	☑	☑			☑				☑	☑	☑		☑	☑	☑	☑	☑			☑	☑
fa2	0	0	0	1	☑	☑		☑	☑				☑	☑	☑		☑	☑	☑	☑	☑		☑	☑	☑
fa3	0	0	1	0	☑	☑	☑		☑			☑	☐	☐	☐	☑	☑	☑	☑	☑	☑	☑		☑	☑
fa4	0	1	0	0							☑		☑	☑	☑		☑	☑	☑	☑	☑			☑	☑
fa5	0	1	0	1							☑		☑	☑	☑		☑	☑	☑	☑	☑		☑	☑	☑
fa6	0	1	1	0							☑	☑	☐	☐	☐	☑	☑	☑	☑	☑	☑	☑		☑	☑
fa7	1	0	0	1	☑	☑		☑	☑				☑	☑	☑		☑	☑	☑	☑	☑		☑	☑	☑
fa8	1	1	0	1						☑	☑		☑	☑	☑		☑	☑	☑	☑	☑		☑	☑	☑

☐ = verwendbar ☑ = eingeplant

(b) Produktspez. Testsuites (minimal)

Abbildung A.3: FA-SPL-Testsuite (*all-transition-pairs*) ohne Redundanz

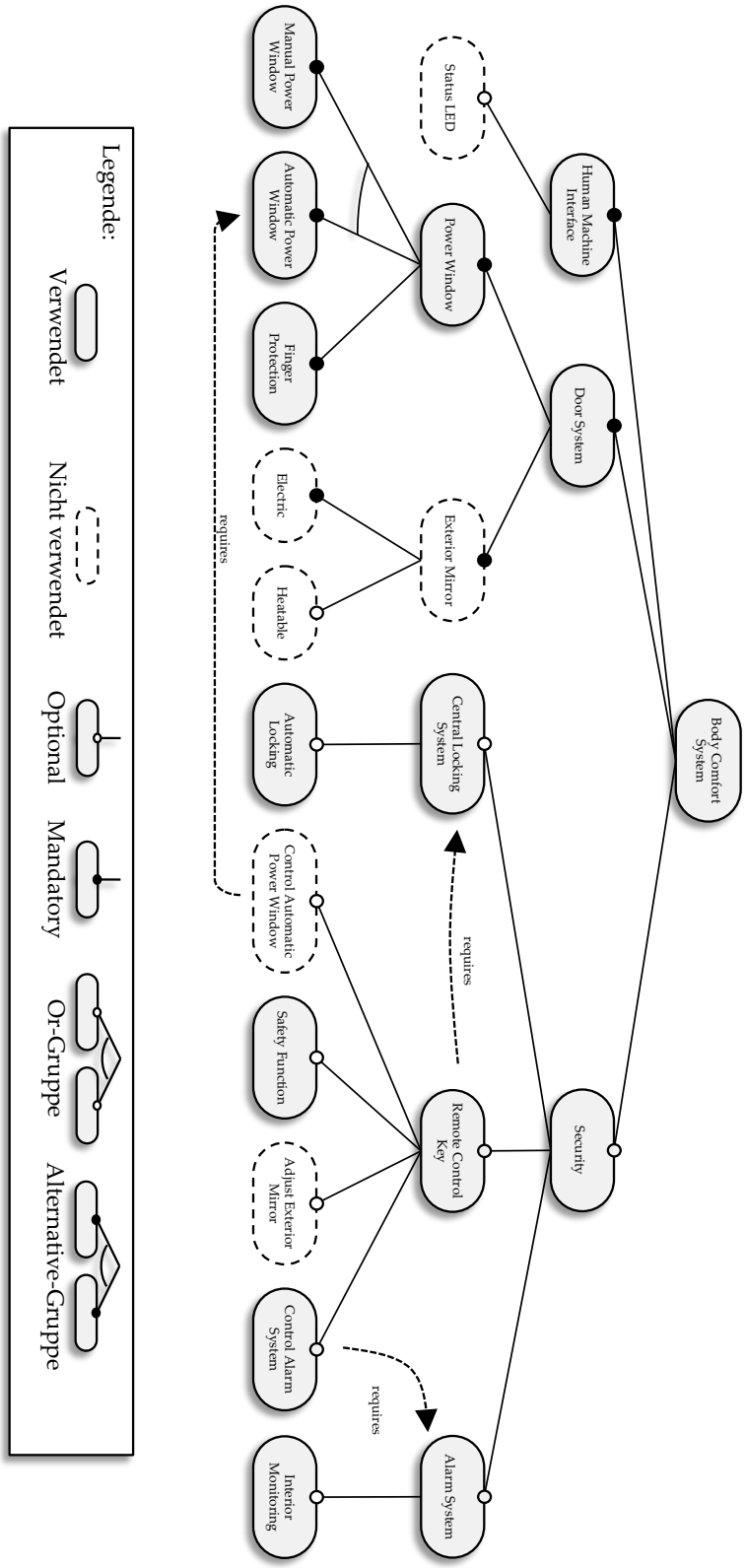


Abbildung A.4: Featuremodell der BCS-SPL

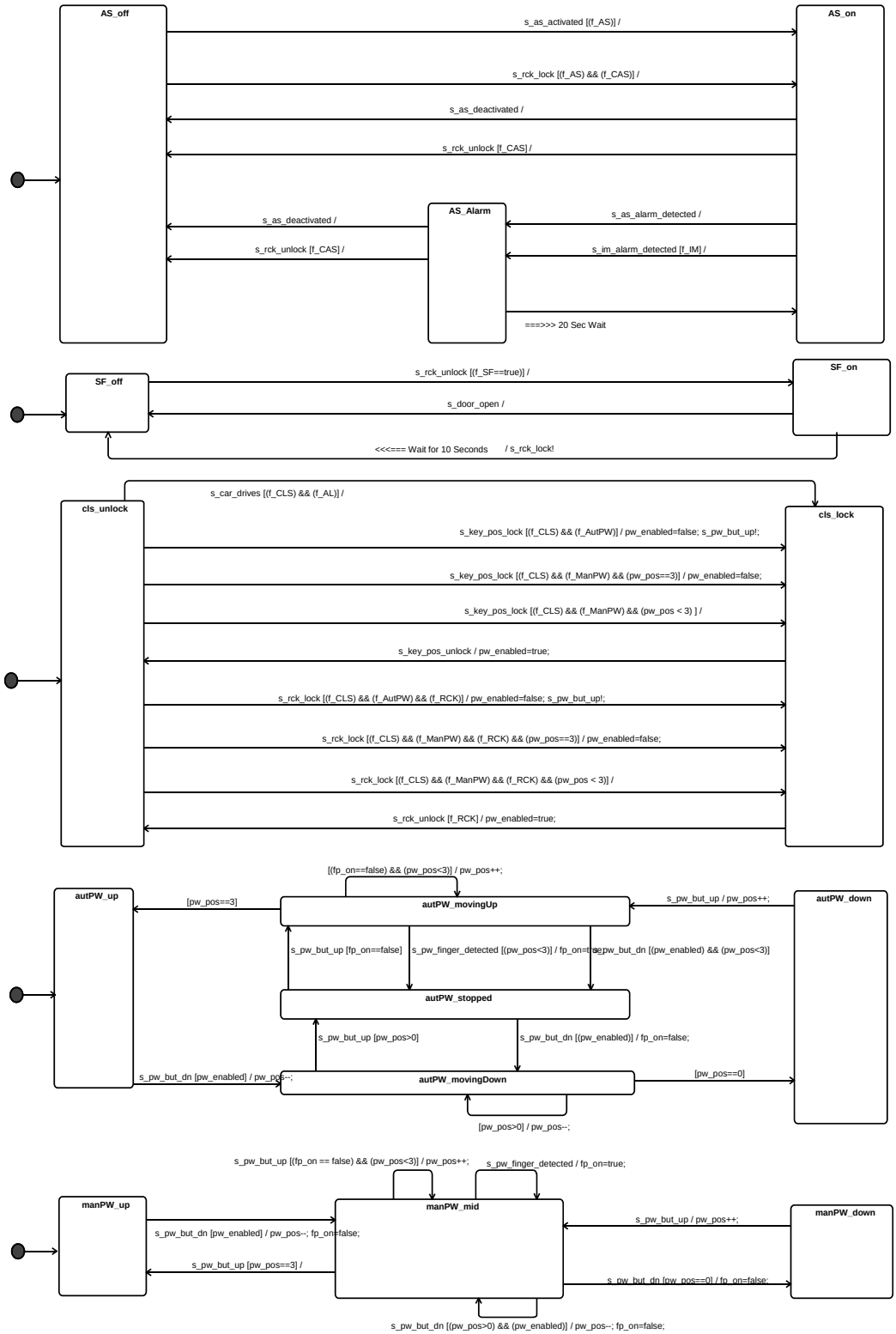


Abbildung A.5: Testmodell der BCS-SPL mit parallelen Unterautomaten

WISSENSCHAFTLICHER LEBENSLAUF

Name:	Harald Cichos
Geburtsdatum:	9. Januar 1981
Geburtsort:	Königs Wusterhausen
E-Mail:	harald.cichos@es.tu-darmstadt.de

1993 - 2000	Friedrich-Wilhelm-Gymnasium in Königs Wusterhausen
2001 - 2008	Informatikstudium an der Friedrich-Schiller-Universität Jena
2008 - 2013	Wissenschaftlicher Mitarbeiter an der TU Darmstadt
2011	Mitorganisator des MoDeVVa-Workshops auf der MoDELS'11