Anforderungsorientierter modellbasierter Softwaretest reaktiver Systeme

Christian Pfaller

Technische Universität München

Technische Universität München
Institut für Informatik

Anforderungsorientierter modellbasierter Softwaretest reaktiver Systeme

Christian Josef Pfaller

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Nassir Navab, Ph.D.
Prüfer der Dissertation:	
1.	Univ.-Prof. Dr. Dr. h.c. Manfred Broy
2.	Univ.-Prof. Dr. Peter O. A. Struss

Die Dissertation wurde am 17.05.2010 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 26.10.2010 angenommen.

Zusammenfassung

Testen ist eine der wesentlichen Qualitätssicherungsmaßnahmen in der Softwareentwicklung. Hauptziel des Testens ist es, Abweichungen vom erwarteten Verhalten des Systems aufzudecken. Da die manuelle Ermittlung von Testfällen einen hohen Anteil der Entwicklungskosten verursacht, wird eine starke Automatisierung angestrebt.

Zur automatischen Ermittlung von funktionalen Testfällen wurden bereits vielfach modellbasierte Techniken vorgeschlagen. Diese identifizieren eine Teilmenge aller möglichen Abläufe in einem (abstrakten) Verhaltensmodell des zu testenden Systems und weisen diese Teilmenge als Testfälle aus. Meist erfolgt die Auswahl anhand einer strukturelle Überdeckung, eine Kopplung mit explizit gegebenen einzelnen Anforderungen findet bisher im Allgemeinen nicht statt. Somit kann gegenüber Auftraggebern von Software oder zur Konformität mit Normen und Standards, der Nachweis, dass jede einzelne funktionale Anforderung ausreichend durch Tests abgedeckt ist, nicht unmittelbar erbracht werden. Ebenso wenig können Experten Ihre Erfahrung in die Testfallgenerierung bei einem gegebenen Modell einfließen lassen. Anforderungsorientierte Methoden betrachten einzelne Anforderungen dagegen nur isoliert um Testfälle abzuleiten. Die Vorteile aus der Integration einzelner Anforderungen zu einem Gesamtmodell, werden für den Test nicht genutzt.

Diese Arbeit diskutiert zunächst Aspekte zur Gütebewertung von Testfällen, wobei die Überdeckung von Anforderungen näher betrachtet wird. Im Hauptteil der Arbeit wird eine Methode zur automatischen Generierung von Testfällen aus einem Verhaltensmodell des zu testenden Systems hinsichtlich anfangs identifizierter Anforderungen beschrieben. Diese Methode erlaubt die Nutzung des Wissens von Domänen- und Testexperten zur Steuerung der Testfallgenerierung bei gleichzeitig hohem Automatisierungsgrad. Kernidee dieser Methode ist es dabei, zum Test jeder Einzelanforderung jeweils nur einen Ausschnitt des gesamten Modells zu betrachten. Damit werden einzelne Anforderungen nicht isoliert getestet, sondern Interaktionen und Abhängigkeiten mit anderen Anforderungen werden berücksichtigt.

Der Prozess zum Einsatz dieser Methode wird skizziert, ausgehend von informell beschriebenen einzelnen Anforderungen. Dieser orientiert sich an der Entwicklung von eingebetteten, reaktiven Softwaresystemen wie sie im Automobil Anwendung finden. Die Methode wird anhand eines typischen Fallbeispiels aus der Automobilindustrie dargestellt. Die ermittelte Testfallmenge wird mit Testfällen aus struktureller Überdeckung und Zufallstests verglichen. Es zeigt sich, dass diese Vergleichstestsuiten einzelne Anforderungen weniger adressieren.

Danksagung

Diese Arbeit entstand in ihrem wesentlichen Teil während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Lehrstuhl für Software & Systems Engineering der TU München. Ohne die Unterstützung und das Vertrauen zahlreicher Personen wäre ihre Anfertigung nicht möglich gewesen.

Allen voran sei Professor Dr. Dr. h.c. Manfred Broy für die Ermöglichung und Betreuung der Arbeit gedankt. Ohne das wissenschaftliche Umfeld und die Freiräume an seinem Lehrstuhl wäre diese Dissertation nicht möglich gewesen. Herzlicher Dank gilt ebenso Professor Dr. Peter Struss für die vielen hilfreichen Kommentare und die Bereitschaft, das Zweitgutachten zu übernehmen.

Die der Arbeit zu Grunde liegende praxisrelevante Problemstellung resultierte vor allem aus einem Projekt an den Ingolstadt Instituten der TU München (INI.TUM) in Zusammenarbeit mit der Audi AG. Professor Dr. Bernd Heißing, Dr. Peter-Felix Tropschuh sowie Markus Buhlman gilt der Dank für Initiierung und Betreuung davon.

Einen großen Beitrag zum Gelingen der Arbeit haben auch die vielen Diskussionen mit Kollegen am Lehrstuhl, am INI.TUM und bei fortiss geleistet. Für ihre kritischen, aber stets konstruktiven Kommentare gilt der Dank hier stellvertretend Dr. Peter Braun, Dr. Florian Deißenböck, Elmar Jürgens, Dagmar Koß, Dr. Markus Pister, Dr. Katharina Spies, Dr. Stefan Wagner und Dr. Sebastian Winter. Besonders hervorgehoben sei an dieser Stelle Dr. Bernhard Schätz für die hilfreichen Anmerkungen, besonders in der Endphase der Arbeit und die Möglichkeit, am An-Institut fortiss die Forschungstätigkeit weiter zu führen. Für die jederzeitige Hilfsbereitschaft in organisatorischen Belangen am Lehrstuhl sei auch Silke Müller und dem gesamten Sekretariat gedankt.

Abschließend danke ich meinen Eltern, meiner Schwester und besonders Dir, Wenke, die Ihr mir von privater Seite immer den Rücken gestärkt habt und mir so die Motivation und Zuversicht zur Vollendung der Arbeit gegeben habt.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1. Einleitung

Bei der Entwicklung von Softwaresystemen ist der Test ein entscheidendes Mittel zur Qualitätssicherung. Ziel ist es dabei, das Vertrauen in eine hinreichende Fehlerfreiheit des Systems zu erlangen. Da der Test eine exemplarische Überprüfung darstellt, kann damit die Abwesenheit von Fehlern im Allgemeinen nicht gezeigt werden. Deshalb soll das Testen derart erfolgen, dass dadurch möglichst viele Fehler aufgezeigt werden, welche im weiteren behoben werden. Die Testaktivitäten beanspruchen dabei einen nicht unerheblichen Teil des Gesamtaufwandes eines Software-Entwicklungsprojekts, sowohl in zeitlicher Hinsicht, als auch in Hinsicht auf die entstehenden Kosten. Daher besteht der Bedarf einer weitgehenden Automatisierung dieser Testaktivitäten. Eine Automatisierung ist hier in allen Bereichen des Testprozesses möglich: Bei der Testausführung, der Erstellung ausführbarer Testfälle sowie der Testfallermittlung. Die Automatisierung der Testausführung hat dabei zum Ziel, Testfälle möglichst ohne manuellen Eingriff auf dem Testobjekt ablaufen zu lassen und die Testergebnisse zu protokollieren. Voraussetzung dafür sind ausführbare Testfälle. Oftmals ist die ausführbare Notation von Testfällen unhandlich und wenig geeignet, um Testfälle zu spezifizieren und zu beschreiben. Testfälle werden daher oft zunächst in einer abstrakten Form angegeben. Im Rahmen der Testfallerstellung wird aus der abstrakten Spezifikation eines Testfalls eine konkrete Instanz, also ein ausführbarer Testfall gebildet. Dieser Schritt wird beispielsweise mithilfe von sogenannten Testtreibern automatisiert.

Die wohl größte Herausforderung zur Automatisierung des Softwaretests stellt die Automatisierung der Testfallermittlung[1] dar. Anders als bei der Instanziierung abstrakt spezifizierter Testfälle zu konkreten, ausführbaren Testfällen, bei welcher ein oder mehrere ansonsten beliebige Ausprägungen der abstrakten Testfallbeschreibung gefunden werden müssen, adressiert die Testfallermittlung die Frage, welche Abläufe des zu testenden Systems aus der Menge aller möglichen Systemabläufe als Testfall zu wählen sind. Der wesentliche Unterschied zur Transformation von abstrakten zu konkreten Testfällen liegt hier darin, dass die verwendeten Testfälle nicht beliebig gewählt werden. Ziel der Testfallermittlung ist es eine möglichst *sinnvoll* gewählte Teilmenge aller möglichen Systemabläufe als Testsuite zu bestimmen, da ein erschöpfender Test nur in trivialen Fällen möglich ist. Unter welchen Bedingungen eine Testsuite sinnvoll

[1] In dieser Arbeit wird mit dem Begriff der *Testfallermittlung* jedes Vorgehen zur Identifikation von Testfällen bezeichnet, unabhängig davon, ob dieses manuell oder automatisch erfolgt. Dagegen bezeichnet der Begriff der *Testfallgenerierung* ausschließlich automatisierte Vorgehensweisen.

gewählt ist, hängt dabei von dem geforderten Gütemaß ab. Hierzu sei an dieser Stelle auf Kapitel 3 verwiesen.

Die Ermittlung einer Menge von Testfällen, der Testsuite, hat zum Ziel, eine Testsuite von möglichst hoher Qualität zu bestimmen. Naheliegend ist dabei der Wunsch, Testfälle zu finden, welche geeignet sind, um möglichst viele Fehler und im Besonderen kritische Fehler im Testobjekt aufzudecken. Eine derartige Formulierung der Qualitätsanforderung an die Testsuite ist dabei kaum in einer Methode zur Testfallermittlung zu operationalisieren – schließlich kann die Überprüfung, ob tatsächlich ausreichend viele der bedeutenden Fehler im Test erkannt wurden, erst am Ende der Lebenszeit des Testobjekts erfolgen (sofern die Korrektheit des Systems nicht durch andere Methoden vollständig nachgewiesen wurde). Da für die meisten Softwaresysteme der vollständige Korrektheitsnachweis in der Praxis nicht möglich ist, kann während der gesamten Nutzungsdauer des Systems nie ausgeschlossen werden, dass das System noch weitere, im Test unentdeckte, Defekte beinhaltet, welche gegebenenfalls zu folgenschweren Fehlern führen. Dementsprechend ist es nach derzeitigem Stand der Forschung nicht möglich, für eine Testsuite zu bestimmen, ob diese zu einer ausreichenden Fehleraufdeckung geeignet ist. Dies ist allenfalls am Ende des Produktlebenszyklus möglich. Aus diesem Grund werden bei der Ermittlung von Testfällen häufig Gütekriterien angewendet, welche zumindest das Vertrauen erhöhen sollen, dass eine Testsuite zur Fehlererkennung, insbesondere von schwerwiegenden Fehlern, geeignet ist.

Weit verbreitet für eine derartige Gütebewertung von Testfällen und Testsuiten sind strukturelle Überdeckungskriterien, welche messen, wie hoch der Anteil des von Testfällen ausgeführten Programmcodes, beziehungsweise der Anteil der ausgeführten Pfade im Programm ist. Zur Bildung derartiger Testsuiten wurden bereits vielfach automatisierte Verfahren vorgeschlagen. Eine andere Art der Qualitätsbewertung von Testfällen stellt die Frage dar, ob die Testfälle die spezifizierten Anforderungen an das System oder die erwartete Nutzung des Systems wiedergeben.

Die Abdeckung von einzelnen Anforderungen durch Testfälle erlangt eine zusätzliche Bedeutung, da Auftraggeber eines Softwareprojekts oft vertragliche Garantien vom Ersteller der Software verlangen, dass die dokumentierten Anforderungen ausreichend getestet wurden. Ein Softwarelieferant muss also häufig gegenüber dem Auftraggeber den Nachweis erbringen, dass die formulierten Anforderungen ausreichend getestet wurden. Dieser Nachweis ist nur dann sinnvoll, falls dem Auftraggeber die Überprüfung möglich ist. Diese Überprüfbarkeit ist gegeben, falls die Zuordnung von Testfällen zu den einzelnen Anforderungen nachvollziehbar ist. Die initialen Anforderungen, welche einem Entwicklungsauftrag zu Grunde liegen, sind allerdings häufig nur informell gegeben, lassen Interpretationsspielraum und beschreiben die Funktionalität des Systems meist nicht vollständig. Vor diesem Hintergrund ergeben sich bei der Automatisierung der Testfallermittlung folgende Schwierigkeiten:

1. Die informell gegebenen Anforderungen müssen für eine automatisierte Testfallermittlung in eine formale Spezifikation, welche von einem Testfallgenerator

verarbeitet werden kann, überführt werden. Dies kann dazu führen, dass es für den Auftraggeber – oder andere Stakeholder – nicht mehr nachvollziehbar ist, ob die generierten Testfälle die initialen Anforderungen adressieren.

2. Eine alleinige Fokussierung auf das in den initialen Anforderungen beschriebene Verhalten kann die Testfallauswahl zu stark einschränken, sofern die Anforderungsdokumente das Verhalten des Systems nicht vollständig spezifizieren. Eine triviale Erfüllung der Anforderungsabdeckung, wobei zu jeder Anforderung ein Testfall beschrieben wird, lässt häufig zu viele Systemabläufe unberücksichtigt. Insbesondere sind auch Systemabläufe von Interesse, welche sich aus der Kombination mehrerer Anforderungen ergeben oder bei denen Anforderungen in leichten Varianten, welche nicht explizit spezifiziert sind, getestet werden.

Um den zweiten Sachverhalt zu entgegnen, wird bei einer manuellen Testfallermittlung der Testingenieur nicht allein die trivialen Systemabläufe, welche unmittelbar aus der Anforderungsbeschreibung ersichtlich sind, als Testfälle spezifizieren, sondern nach seinem Erfahrungswissen und seiner Intuition weitere Testfälle hinzunehmen, welche die Anforderung in Ausnahmefällen oder in der Kombination mit anderen Fällen testet. Der Testingenieur bildet sich somit ein mentales Modell der vollständigen Anforderung. Da dieses Wissen des Testingenieurs im Allgemeinen nicht explizit formalisierbar ist, sondern im Wesentlichen auf Intuition und Erfahrung basiert, kann es innerhalb einer vollständig automatisierten Testfallermittlung nicht genutzt werden.

Zur Automatisierung der Testfallermittlung wurde bereits vielfach die *Modellbasierte Testfallgenerierung* vorgeschlagen. Dabei wird das zu testende Systemverhalten durch ein (abstraktes) Verhaltensmodell vollständig spezifiziert und Pfade aus diesem Modell als Testfälle gewählt. Die Auswahl dieser Pfade kann beispielsweise zufällig erfolgen, oder aber es werden strukturelle Überdeckungskriterien, wie etwa Zustands- oder Transitionsüberdeckung, als Auswahlkriterien verwendet. Ein Testfallgenerator dient dazu, eine Testsuite mit Abläufen entsprechend dem Modell zu finden, welche diese Auswahlkriterien erfüllt. Diese Techniken zur Testfallgenerierung eigen sich insbesondere für reaktive Systeme, deren Verhalten sich durch Zustandsmaschinen beschreiben lässt. Reaktive Systeme verarbeiten fortwährend Eingaben, und die Ausgaben hängen nicht allein von der aktuellen Eingabe, sondern von der gesamten Eingabehistorie ab, diese Systeme verfügen also über einen internen Systemzustand. Derartige Systeme kommen vor allem als eingebettete Softwaresysteme zur Steuerung von umfassenderen technischen Systemen vor. Insbesondere in der Automobiltechnik dienen reaktive Systeme zur Realisierung unterschiedlicher Fahrzeugsysteme, beispielsweise Zentralverriegelung, Fensterheber, Blinkersteuerung oder einer adaptiven Geschwindigkeitsregelung (ACC).

Während die modellbasierte Testfallermittlung ein hohes Potential bei der Automatisierung des Testens bietet, stützen sich die meisten der heute vorgeschlagenen Verfahren vor allem auf die Erfüllung struktureller Kriterien bezüglich der Modelle. Diese Verfahren stellen im Allgemeinen nicht sicher, dass aus den generierten Testfälle

ein Bezug zu den dokumentierten Einzelanforderungen erkennbar ist. Aus diesem Umstand folgt die zentrale Fragestellung dieser Arbeit, welche im Folgenden erläutert wird.

1.1. Problemstellung

Die zentrale Problemstellung, zu deren Lösung diese Arbeit Beiträge liefert, lautet:

> *Wie können Techniken zur modellbasierten Testfallgenerierung im Entwicklungsprozess eingesetzt werden, um eine möglichst weitgehende Automatisierung der Testfallermittlung zu erreichen und um zugleich Testfälle zu erhalten, welche insbesondere die vorgegebenen Einzelanforderungen eines Anforderungskatalogs adressieren?*

Mit dieser Problemstellung sind folgende detaillierte Forderungen verbunden:

Kommunizierbarkeit und Nachweispflicht gegenüber Stakeholdern Durch einen *anforderungsorientierten* Test soll gegenüber den unterschiedlichen Stakeholdern des Systems, insbesondere gegenüber dem Auftraggeber, der *Nachweis* geführt werden, dass jede Einzelanforderung aus einem Anforderungskatalog getestet wurde. Dabei ist zu berücksichtigen, dass dieser Nachweis für die unterschiedlichen Stakeholder nachvollziehbar ist. Daraus folgt insbesondere, dass eine formale Beschreibung von Anforderungen und Testfällen für alle Stakeholder leicht *verständlich* sein muss. Hierzu ist zu berücksichtigen, dass vielen Stakeholdern tiefere mathematische Kenntnisse fehlen.

Keine isolierte Betrachtung einzelner Anforderungen Die Ableitung von Testfällen aus Anforderungen soll *nicht allein in trivialen Testfälle*n resultieren, welche unmittelbar aus der isolierten Betrachtung der Beschreibung einer einzelnen Anforderung folgen. Stattdessen sollen, analog zur manuellen Ermittlung von Testfällen, auch nicht explizit dokumentierte *Sonderfälle* oder *Kombinationen aus mehreren Anforderungen* in die identifizierte Menge von Testfällen einfließen. Damit sollen nicht nur die explizit dokumentierten Anforderungen im Test Berücksichtigung finden, sondern es sollen auch implizite beziehungsweise intendierte Anforderungen in den Test einfließen.

Robuste Interpretation von Anforderungen als Gütekriterium Der anforderungsorientierte Test soll darauf zielen, Anforderungen möglichst robust zu interpretieren. Die Testfälle sollen daher möglichst adäquat zur Nutzung des Systems gewählt werden. Einerseits soll der Test dazu auch Nutzungsszenarien einer Anforderung adressieren, welche nicht explizit angegeben sind. Andererseits soll der Test auch Abläufe umfassen, welche den Nutzungsszenarien einer Anforderung ähnlich sind, aber zu einem abweichenden Systemverhalten führen sollen.

Intention und Erfahrung von Testingenieuren bei Automatisierung nutzen Die *Automatisierung* hat vor allem das Ziel, unnötigen manuellen Spezifikationsaufwand für die Testingenieure zu reduzieren. Die Methode soll deshalb vordergründig als *Werkzeug* dienen, welche dem Testingenieur die Ermittlung und Spezifikation von Testfällen erleichtert. Allerdings soll die Methode die Möglichkeit schaffen, dass die *Testingenieure ihre Erfahrung und Intention nutzen* können. Insbesondere ist dies von Bedeutung, um die Testfallermittlung *flexibel* zu gestalten und sie den spezifischen Erfordernissen in unterschiedlichen Entwicklungsprojekten anzupassen.

Vor dem Hintergrund dieser Problemstellung ist der Titel der vorliegenden Arbeit zu verstehen. Die dort verwendeten Begriffe werden im Folgenden erläutert, um das Thema der Arbeit weiter zu präzisieren:

Anforderung Der Begriff der *Anforderung* bezeichnet in dieser Arbeit eine, durch einen Stakeholder explizit angegebene, Forderung an das Verhalten des Systems. Es werden hier allein *funktionale* Anforderungen betrachtet, deren Erfüllung durch Beobachtung des Ein-/Ausgabeverhalten an der Nutzungsschnittstelle des zu testenden Systems geprüft werden kann. Im Vordergrund steht somit die Ablauflogik der Funktion, welche durch Nutzungsszenarien beschrieben werden kann.

Die Menge dieser einzelnen Anforderungen bildet den (funktionalen) Anforderungskatalog an das zu testende System. Im Allgemeinen wird angenommen, dass die Anforderungen zunächst informell formuliert sind. Um die Validierung durch die Stakeholder zu ermöglichen, werden exemplarische Nutzungsszenarien als formale Repräsentanten dieser Anforderung verwendet. Eine weitere formale Spezifikation der einzelnen Anforderungen liegt dagegen nicht vor. Die Gesamtanforderung der zu testenden Funktionalität ist als formales Modell gegeben. Anforderungsszenarien und Modell der Gesamtfunktion dienen als Basis zur Bestimmung von Testfällen.

Anforderungsorientierung *„Anforderungsorientierter Test“* bezeichnet in dieser Arbeit einen Test, welcher zum Ziel hat, zu jeder einzelnen Anforderung aus dem Anforderungskatalog eine Menge von Testfällen zu bestimmen, welche sich an der Anforderung orientieren. *„Orientiert“* bedeutet in diesem Zusammenhang, dass von den Testfällen angenommen werden kann, dass diese aus einer Anforderung ableitbar oder dieser zuordenbar sind. Dies wird umso stärker angenommen, falls Testfälle Merkmale aufweisen, welche auch auf das jeweilige Anforderungsszenario zutreffen. Die Anforderungsorientierung ist dabei eine Eigenschaft, welche von der Menge der Testfälle, die zu einer Einzelanforderung identifiziert wurden, erfüllt wird. Sie ist dagegen nicht notwendigerweise für jeden einzelnen Testfall zu bestimmen. Die Forderung, dass Testfälle einer Anforderung zuordenbar sein sollen, ist dabei sehr allgemein. So kann auch ein Testfall, welcher zu einem anderen Systemverhalten als ein Anforderungsszenario führt, der Anforderung auch

zuordenbar sein. Dies wird hier angenommen, falls der Testfall eine ähnliche Nutzungssituation adressiert, wie jene, welche durch die Anforderung beschrieben ist.

Softwaretest Als *Test* wird in dieser Arbeit eine Maßnahme zur Qualitätssicherung bezeichnet, bei welcher Eingaben an das zu testende System übergeben und die vom System produzierten Ausgaben beobachtet werden. Die beobachteten Ist-Ausgaben werden mit den erwarteten Soll-Ausgaben verglichen. Ziel des Tests ist es, Abweichungen zwischen dem erwarteten Verhalten und dem tatsächlichen Verhalten aufzudecken. Ein Test stellt somit nur eine exemplarische Überprüfung der Funktionalität des Systems hinsichtlich der gewählten Ausgaben dar. Eine Verallgemeinerung der exemplarischen Testergebnisse auf andere Systemabläufe ist in der Regel nicht möglich. Durch eine sinnvolle Auswahl von Testfällen soll erreicht werden, dass die ausgeführten Testfälle geeignete Repräsentanten für die spätere Nutzung[2] des Systems sind.

Obwohl diese Arbeit sich auf reaktive Systeme beschränkt, welche insbesondere zur Steuerung in mechanische oder elektrische Systeme eingebettet sind, wird ausschließlich der Test der Software betrachtet. Der Test erfolgt aus einer Black-Box-Sicht, der Testingenieur hat demnach keine Kenntnisse über die Realisierung oder den internen Systemzustand.

Modellbasierung Ein Modell des zu testenden Systems dient in dieser Arbeit als Grundlage zur Ermittlung von Testfällen. Die Arbeit baut insofern vor allem auf den Arbeiten von Pretschner (2003) und Lötzbeyer (2003) auf. Dabei beschreibt das Modell abstrakt die zu testende Funktionalität des Systems. Elementar ist die Abstraktion – vordringliches Ziel ist es, Fehler zu identifizieren, welche während der Konkretisierung der abstrakten Verhaltensbeschreibung zur Realisierung entstehen. Dieses Testmodell ist als Zustandsmaschine gegeben und beschreibt dabei eine sowohl deterministische als auch totale Funktion hinsichtlich des (abstrakten) Eingabealphabets und des Zustandsraums.

Reaktive Systeme Analog zu den eben erwähnten Arbeiten zum modellbasierten Test zielt die vorgestellte Methode auf reaktive Systeme. Reaktive Systeme sind dabei als Systeme zu verstehen, welche fortwährend und potentiell unendlich lange mit ihrer Umwelt interagieren. Die korrekte Ausgabe des Systems hängt dabei nicht allein von den aktuell anliegenden Eingaben ab, sondern von der Historie der Eingaben. Diese Systeme sind in diesem Sinne zustandsbasiert, bei der Testausführung ist die Korrektheit einer Folge von Ausgaben zu einer Folge von Eingaben zu überprüfen. Betrachtet werden ferner nur Systeme, deren Verhalten sich durch diskrete Zustandsmodelle beschreiben lässt. Auch für die

[2] Bei der *Nutzung* ist nicht allein die Häufigkeit der Ausführung zu beachten, sondern auch die Auswirkungen einer Fehlfunktion durch die Nutzung des Systems.

zu testenden Systeme wird, wie schon für die Modelle, vorausgesetzt, dass diese sich hinsichtlich der Eingaben deterministisch verhalten.

1.2. Beitrag

Die zuvor ausführlich geschilderte Problemstellung ist vor allem deshalb relevant, da in der industriellen Softwareentwicklung dem Nachweis eines hinreichenden Tests jeder einzelnen funktionalen Anforderung eine hohe Bedeutung zugemessen wird. Der wesentliche Beitrag dieser Arbeit liegt darin, für eine derartige anforderungsorientierte Testfallermittlung Techniken der modellbasierten Testfallgenerierung zur Automatisierung nutzbar zu machen. Diese Arbeit liefert dazu sowohl methodische, pragmatische und technische Beiträge sowie Beiträge zur Metrisierung der Testfallgüte.

1.2.1. Methodik

Wesentlich für die praktische Anwendbarkeit von Techniken zur Testfallermittlung ist ihre Einbettung in den gesamten Entwicklungsprozess. Insbesondere die Schnittstellen zur Anforderungserhebung sind dabei von Bedeutung. Während häufig das Vorliegen eines geeigneten Testmodells zur modellbasierten Testfallgenerierung angenommen wird, geht diese Arbeit auch auf den Zusammenhang zwischen informell gegebenen Einzelanforderungen und einem vollständigen Verhaltensmodell ein. Es wird die Diskussion geführt, in wie weit Anforderungen formalisiert sein müssen, um zur Testfallgenerierung nutzbar zu sein.

1.2.2. Technik

Aufbauend auf Techniken der modellbasierten Testfallgenerierung wird ein Verfahren formal beschrieben, welches die Testfallgenerierung um den Aspekt der Anforderungsorientierung erweitert. Dieses basiert im Wesentlichen auf der Bildung von anforderungsspezifischen Teilmodellen aus dem gesamten Testmodell. In den Teilmodellen erfolgt der Einsatz bekannter Techniken zur Testfallgenerierung. Die Arbeit führt damit eine Technik zur modellbasierten Testfallgenerierung ein, welche es erlaubt, strukturelle oder stochastische Testauswahlkriterien mit funktionalen Testauswahlkriterien zu kombinieren.

Wesentlich ist dabei, dass eine naive Erfüllung einer Anforderungsabdeckung vermieden wird, indem das Verfahren sicherstellt, dass Anforderungen im Test nicht isoliert betrachtet werden, sondern das Testmodell insgesamt die Ermittlung der Testfälle beeinflusst.

1.2.3. Pragmatik

Der Softwaretest ist in der industriellen Praxis noch immer stark pragmatisch geprägt. Häufig wird auf die Intuition des Testingenieurs vertraut, welcher nach seiner Erfahrung „gute“ Testfälle hinsichtlich der einzelnen Anforderungen bestimmt. Dieses Vorgehen erscheint insofern gerechtfertigt, da bisweilen kein Gütemaß für Testfälle absehbar ist, welches zum Zeitpunkt der Ermittlung von Testfällen operationalisierbar ist und allgemein anerkannt wäre.

Die in der Arbeit beschriebene Methode zur Testfallermittlung greift diesen Pragmatismus auf und unterstützt ihn durch weitreichende Automatisierung. Es ist somit, im Gegensatz zu vergleichbaren Arbeiten, nicht das Ziel der Arbeit, die Testfallermittlung vollständig zu automatisieren. Dem Testingenieur wird das aufwändige Formulieren einzelner Testfälle abgenommen, dennoch kann er seine Intuition durch vergleichsweise einfache „Stellschrauben“ in die Testfallerstellung einfließen lassen.

1.2.4. Metrisierung

Wie im vorhergehenden Abschnitt bereits erwähnt, besteht weiter ein großer Forschungsbedarf hinsichtlich der Gütebewertung von Testfällen. Diese Arbeit leistet hierzu einen Beitrag, indem sie die Gütebewertung von Testfällen schematisiert und Bewertungskriterien sowie eine Metrik zur Beurteilung der Anforderungsabdeckung einführt.

1.2.5. Hypothese und Annahmen

Die zentrale Hypothese, welche durch diese Arbeit untersucht und bestätigt werden soll, lautet:

> *Techniken zur modellbasierten Testfallgenerierung sind geeignet, um die Ermittlung von spezifischen Testfällen zu einzelnen funktionalen Anforderungen, welche dem eingesetzten Modell zu Grunde liegen, zu unterstützen.*
>
> *Im Vergleich zur direkten Ableitung von Testfällen aus einzelnen Anforderungen lässt sich dabei ein höherer Automatisierungsgrad erreichen. Zudem wird vermieden, dass nur besonders nahe liegende Systemabläufe, welche unmittelbar aus den einzelnen Anforderungen ersichtlich sind, als Testfälle gewählt werden.*
>
> *Im Gegensatz zur Testfallgenerierung hinsichtlich struktureller oder stochastischer Testfallauswahl aus einem Testmodell ist in einem derartigen Verfahren der Nachweis inhärent, dass jede einzelne Anforderung explizit in den Testfällen adressiert wird, der Nachweis ist dabei für alle Stakeholder*

nachvollziehbar. Zudem erlaubt es der Ansatz, dass in die Testfallgenerierung die Intention des Testingenieurs zur Beurteilung geeigneter Testfälle einfließt.

Dazu wird in dieser Arbeit ein Verfahren zur Testfallgenerierung eingeführt, welches sich an Anforderungsszenarien orientiert. Die Anwendbarkeit dieses Verfahrens wird an einem vereinfachten Fallbeispiel aus der Automobilindustrie dargestellt. Es wird dabei angenommen, dass die formulierten Anforderungen die einzigen explizit dokumentierten Informationen zur Vervollständigung der Spezifikation und Erstellung der Testfälle sind. Insbesondere werden keine expliziten Nutzungs- oder Fehlermodelle verwendet. Für alle Systemteile und alle unterschiedlichen Systemabläufe wird zunächst angenommen, dass sie mit Fehlern behaftet sein können. Eine spezifische Erwartung zur Fehlerverteilung existiert nicht.

Weiter wird angenommen, dass – wie es meist in der Praxis der Softwareentwicklung der Fall sein dürfte – die initial formulierten Anforderungen nicht ausreichen, um eine vollständige totale und deterministische Beschreibung des Systemverhaltens zu liefern. Das hat insbesondere zur Folge, dass ein Test nur allein der formulierten einzelnen Anforderungen nicht ausreicht, um das System insgesamt zu beurteilen. Der Test soll auch weitere Varianten und Kombinationen der Anforderungen berücksichtigten.

1.3. Einordnung

Diese Arbeit ist vor allem in das Gebiet des Softwaretests einzuordnen. In einem nicht unerheblichen Teil betrifft der anforderungsorientierte Test dabei auch Fragen des Requirements Engineering. Insbesondere wird angegeben, wie testbare Anforderungen formuliert werden können und wie die Verfolgung (Tracing) von Anforderungen zu Testfällen bewerkstelligt werden kann.

1.4. Gliederung

Diese Arbeit erläutert zunächst Grundlagen des Softwaretests, insbesondere des modellbasierten Tests, und der Bewertung der Testfallgüte. Anschließend folgt eine Darstellung der erarbeiteten Methode zum anforderungsorientierten modellbasierten Softwaretest, wobei zunächst ein Überblick über den Gesamtprozess gegeben wird. Die einzelnen Kapitel behandeln dabei folgende Inhalte:

Kapitel 2, *Test reaktiver Systeme,* geht auf Grundlagen des Testens von Software, des modellbasiereten Tests und des Tests hinsichtlich Anforderungen ein. Es stellt den aktuellen Stand der Forschung und Anwendung dar und gibt Hinweise auf verwandte

Arbeiten. Fragen zur Güte von Testfällen werden in diesem Kapitel bewusst außer Acht gelassen, diese werden im folgenden Kapitel betrachtet.

Kapitel 3, *Qualitätsbewertung von Testsuiten,* behandelt die Bewertung der Testfallgüte. Es wird diskutiert, dass bisher kein zufriedenstellendes und während der Testerstellung operationalisierbares Gütemaß für Testfälle existiert und auch nicht absehbar ist. Es wird eine Systematisierung zur Klassifikation unterschiedlicher Qualitätsmaße angegeben und sich schließlich auf die Bewertung der Testfallgüte hinsichtlich der Überdeckung mit Anforderungen konzentriert. Abschließend werden in diesem Kapitel Bewertungskriterien für eine Anforderungsüberdeckung diskutiert. Diese Kriterien werden später in Kapitel 7 verwendet, um eine Metrik zur Anforderungsüberdeckung auf Basis der Notation von Anforderungen aus Kapitel 5 zu definieren.

Bevor die Methode zur anforderungsorientierten Testfallgenerierung im Detail dargestellt wird, folgt in **Kapitel 4**, *Überblick über den Entwicklungs- und Testprozess,* zunächst eine Beschreibung des zu Grunde liegenden Entwicklungs- und Testprozesses im Überblick. In den folgenden beiden Kapiteln werden die einzelnen Phasen des Prozesses genauer erörtert.

Kapitel 5, *Anforderungsspezifikation und Modellerstellung,* behandelt die Phasen zur Spezifikation von funktionalen Anforderungen. Es wird dabei von einer in informellen Beschreibung von Nutzungsszenarien ausgegangen und die Schnittstelle zwischen System und Umwelt beschrieben. Dies ist Voraussetzung für die anschließende Beschreibung von formalen Szenarien als exemplarische Abläufe. Die zweite Phase der Spezifikation stellt die Erstellung eines Verhaltensmodells dar, welches als Testmodell dient – mit dieser Vervollständigung der Spezifikation schließt dieses Kapitel. Die Diskussion des Verfahrens erfolgt anhand einer Fallstudie einer Fensterhebersteuerung aus der Automobiltechnik.

In **Kapitel 6**, *Anforderungsorientierte Testfallermittlung,* wird die Verwendung modellbasierter Techniken zur anforderungsorientierten Testfallermittlung im Detail erläutert. Es wird dabei zunächst diskutiert, warum strukturelle beziehungsweise statistische Testfallspezifikationen allein nicht ausreichend sind. Weiter wird erläutert, dass die vorgestellte Methode das Ziel verfolgt, die Vorteile beider Ansätze möglichst zu vereinen. Die weiteren Abschnitte adressieren die Klassifikation von Szenarien und die Übertragung dieser Klassifikation in das Modell sowie die Testfallgenerierung in Teilmodellen. Zusammen mit dem vorhergehenden Kapitel ist dies der technische Beitrag dieser Arbeit. Auch hier wird wiederum auf die Fallstudie des Fensterhebers zurückgegriffen.

Kapitel 7, *Bewertung der Methode,* definiert zunächst eine Metrik zur Anforderungsabdeckung entsprechend der in Kapitel 3 erklärten Bewertungskriterien. Die Testfälle der Fallstudie werden anhand dieser Metrik bewertet und mit Testfällen verglichen, welche allein durch strukturelle Testfallspezifikationen erzielt worden sind.

Die praktische Anwendbarkeit und die dadurch erzielten Vorzüge des vorgeschlagenen Verfahrens werden nochmals ausführlich diskutiert.

Kapitel 8, *Zusammenfassung und Ausblick,* fasst die Inhalte der Arbeit zusammen und zeigt weiterführende Arbeiten auf, welche nicht Bestandteil der vorliegenden Dissertation sind.

2. Test reaktiver Systeme

Zunächst sollen Grundlagen und bestehende Methoden zum Test reaktiver Systeme erläutert werden. Insbesondere wird auf die Generierung von Testfällen aus einem Verhaltensmodell des zu testenden Systems eingegangen. In diesem Kapitel wird die Bewertung der Qualität von Testfällen noch nicht behandelt, diese ist Gegenstand von Kapitel 3.

2.1. Begriff des Softwaretests

Myers (1979) definiert Testen als *die Tätigkeit der Programmausführung mit dem Zweck Fehler zu finden.* Testen ist somit eine Aktivität zur Qualitätssicherung von Software, in welcher das Programm, das Testobjekt, explizit ausgeführt wird. Dies unterscheidet Testen etwa von statischen Qualitätssicherungsmaßnahmen wie etwa Code-Reviews oder der statischen Überprüfung auf Typfehler. Auch die Definition in (IEEE 610, 1990; IEEE 829, 2008) definiert Testen als *eine Aktivität, in welcher ein System oder eine Komponente unter spezifizierten Bedingungen ausgeführt wird, die Ergebnisse beobachtet oder aufgezeichnet werden und eine Evaluierung eines Aspekts des System vorgenommen wird.* In der Literatur wird der Begriff des Tests häufig auch umfassender gebraucht. So unterscheiden Spillner und Linz (2007) zwischen dem *statischen Test* (Reviews und Quellcodeanalyse) und dem *dynamischen Test*, wobei es nur im dynamischen Test zur Programmausführung kommt.

Gegenstand dieser Arbeit ist der (dynamische) Test, bei welchen das zu testende System ausgeführt wird. Dabei werden Eingaben an das System übergeben und die Ausgaben beobachtet. Ein Fehlverhalten wird festgestellt, falls die beobachteten Ausgaben nicht mit den erwarteten Ausgaben übereinstimmen. Ferner beschränkt sich diese Arbeit auf den funktionalen Test, also der Überprüfung der Funktionalität. Andere Qualitätseigenschaften von Software, wie sie etwa in (ISO 9126, 2001) auch definiert sind, werden nicht näher betrachtet. Der Begriff des *funktionalen* Tests wird in dieser Arbeit in Hinsicht auf die zu überprüfende Qualitätseigenschaft, die Funktionalität, gebraucht. Damit lässt sich der funktionale Test etwas von Stress- oder Lasttests abgrenzen, welche die Zuverlässigkeit des Programms adressieren. Der funktionale Test ist von dem im folgenden Abschnitt diskutierten Begriff des funktionsorientierten Tests zu unterscheiden.

2.1.1. White-Box-Test / Black-Box-Test

In der Literatur (vgl. u. a. Myers (1979); Beizer (1990, 1995); Balzert (1998); Liggesmeyer (2002); Spillner und Linz (2007)) wird oft zwischen *White-Box-* und *Black-Box-*Testverfahren unterschieden. Primitive Definitionen beschreiben den White-Box-Test dabei als den Test, welcher mit Kenntnis des Programmcodes beziehungsweise der inneren Struktur des Testobjekts durchgeführt wird, der Black-Box-Test erfolgt dagegen ohne Kenntnisse der internen Programmstruktur und leitet sich allein aus der beabsichtigten Funktionalität ab. Hilfreicher ist dagegen eine Unterscheidung hinsichtlich des Kriteriums zur Auswahl der Testfälle: Soll sich die Auswahl der Testfälle vordringlich an den geforderten funktionalen Eigenschaften des Systems orientieren, stellt dies einen *funktionsorientierten Test* dar. Ist es dagegen das Ziel, dass die Menge der Testfälle, die *Testsuite*, so gewählt wird, dass die interne Struktur des Testobjekts zu einem bestimmten Grad überdeckt wird, handelt es sich um einen *strukturorientierten Test*.

Mit dem gewählten Auswahlkriterium geht gleichzeitig ein entsprechender Qualitätsbegriff für Testsuiten einher, sofern die Erfüllung des Auswahlkriteriums durch die Testsuite messbar ist. Im Fall des strukturorientierten Tests führt dies etwa zu den Überdeckungsmetriken von Programmcode, wie beispielsweise Anweisungs-, Zweig- oder Bedingungsüberdeckung. Es sei an dieser Stelle schon darauf hingewiesen, dass ein Zusammenhang zwischen derartigen Metriken und der Produktqualität bisher nicht nachgewiesen werden konnte. Auf die Qualitätsbewertung von Testfällen wird ausführlich in Kapitel 3 eingegangen. Dort wird auch dargestellt, dass neben Programmcode und Spezifikation weitere Dokumente als Grundlage für die Testfallauswahl dienen können.

Die vorliegende Arbeit konzentriert sich auf den funktionsorientierten Test. Es wird dabei davon ausgegangen, dass die innere Struktur der Implementierung des Testobjekts nicht bekannt ist. Im Hinblick auf die modellbasierte Testfallgenerierung ist anzumerken, dass die Struktur des Testmodells bekannt ist und diese Struktur auch in die Testfallauswahl einfließt. Dieses Modell ist jedoch eine abstrakte *Spezifikation* der geforderten Funktionalität des Testobjekts. Weiter wird kein Zusammenhang zwischen der Struktur des Testmodells und der inneren Struktur des Testobjekts angenommen. Somit handelt es sich aus der Perspektive des Testobjekts um einen funktionsorientierten (beziehungsweise Black-Box-) Test und nicht um einen strukturorientierten Test.

Der im Titel dieser Arbeit eingeführte Begriff der *Anforderungsorientierung* stellt eine noch engere Fassung der Funktionsorientierung dar, wobei vor allem die dokumentierten Einzelanforderungen im Test berücksichtigt werden sollen. Dies setzt natürlich voraus, dass einzelne Anforderungen an das System identifiziert wurden. Dazu sei an dieser Stelle auf Kapitel 5 verwiesen.

2.1.2. Teststufen

Üblicherweise (IEEE 829, 2008; Spillner und Linz, 2007) wird im Entwicklungsprozess zwischen Komponenten-, Integrations-, System- und Akzeptanz- oder Abnahmetest unterschieden. Im Komponententest werden dabei einzelne Komponenten der Implementierung, welche durch die Systemdekomposition im Rahmen des Architekturdesigns festgelegt wurden, getestet. Der Integrationstest behandelt das Vorgehen zur schrittweisen Integration und die damit verbundenen Tests der integrierten Komponenten bis hin zum Gesamtsystem. Der Systemtest testet das Gesamtsystem hinsichtlich der spezifizierten Funktionalität. Der Akzeptanz- oder Abnahmetest dient als Kriterium zur Vertragserfüllung gegenüber dem Auftraggeber der Software, im Abnahmetest steht nicht mehr die Sichtweise des Softwareerstellers sondern diejenige des Auftraggebers im Vordergrund.

Hinsichtlich der Teststufe fokussiert sich diese Arbeit auf Komponenten- und Systemtests, wobei zwischen diesen beiden Stufen nicht weiter unterschieden wird, da das zu testende System immer als Black-Box betrachtet wird. Ob diese Black-Box nun Teil eines größeren Systems ist oder aber das Gesamtsystem umfasst, ist für das eingeführte Verfahren unerheblich, es lässt sich sowohl auf der Ebene der Komponenten als auch auf der Ebene des Gesamtsystems anwenden. Voraussetzung ist in jedem Fall, dass Anforderungen für das jeweilige Testobjekt formuliert sind und ein Testmodell erstellt wird, welches das geforderte Verhalten der Komponente beziehungsweise des Systems spezifiziert. Auf die schrittweise Integration von (Teil-)Komponenten wird dagegen nicht explizit eingegangen. Der Abnahmetest wird insofern besonders berücksichtigt, da es Ziel der vorgestellten Methode ist, die durch die Stakeholder formulierten Anforderungen zur Selektion von Testfällen zu verwenden. Die hier eingeführte anforderungsorientierte Testfallgenerierung ist auch dadurch motiviert, gegenüber dem Auftraggeber den Test der einzelnen Anforderungen nachzuweisen.

2.2. Automatisierung des Softwaretests

Diese Arbeit behandelt die modellbasierte Testfallgenerierung als eine Maßnahme innerhalb des Testdesigns zur Automatisierung des Testprozesses. Im Rahmen des Softwaretests können noch weitere Aktivitäten automatisiert werden. Der Testprozess umfasst etwa nach Spillner und Linz (2007) die folgenden Aktivitäten:

Testplanung: Die Testplanung legt Aufgabe (Testobjekt) und Ziel des Tests fest, erstellt ein Testkonzept, welches die Teststrategie festlegt und definiert ein Testendekriterium.

Teststeuerung: Die Teststeuerung umfasst die fortwährende Überwachung der Testaktivitäten.

Testanalyse: Die als Testbasis zur Verfügung stehenden Dokumente werden analysiert und ggf. vervollständigt und präzisiert.

Testdesign: Entsprechend der Teststrategie werden logische (abstrakte) Testfälle ermittelt.

Testrealisierung: Aus den abstrakten, logischen Testfällen werden konkrete, ausführbare Testsequenzen gebildet.

Testdurchführung: Bereitstellung der Testinfrastruktur und Ausführung der konkreten Testfälle auf dem Testobjekt sowie Protokollierung der Testdurchführung und des Testergebnisses.

Testauswertung und Bericht: Auswertung der Testergebnisse, ob das in der Testplanung festgelegte Testziel (Testendekriterium) erreicht wurde, und Erstellung eines Berichts über die Testdurchführung und das Testergebnis.

In der Praxis findet eine Automatisierung der Testdurchführung bereits breite Anwendung. Ein Beispiel ist die automatisierte Ausführung von Unit-Tests für die Programmiersprache Java mithilfe der Testumgebung jUnit (Link, 2005). In der Automobilindustrie kommen bei der Steuergeräteentwicklung für Kraftfahrzeuge Software-in-the-Loop- (SiL) und Hardware-in-the-Loop-Tests (HiL) zum Einsatz (Hartmann, 2001). Gietelink u. a. (2006) sowie Bock (2008) beschreiben dazu weiter Simulationen der gesamten Umwelt zur automatisierten Durchführung von Fahrtests mit dem realen Fahrzeug. In *Model-in-the-Loop*-Simulationen (MiL) dient ein Modell des zu entwickelnden Systems als Testobjekt (Lamberg u. a., 2004), auf welchem automatisch Simulationsläufe des Modellverhaltens durchgeführt werden. Conrad (2004) klassifiziert Testmethoden, in welchen ein Modell des Systems das Testobjekt darstellt als eine Ausprägung des modellbasierten Tests. Es sei an dieser Stelle darauf hingewiesen, dass die modellbasierte Testfallgenerierung, wie sie auch Gegenstand der vorliegenden Arbeit ist, davon zu unterscheiden ist.

Meist in der Kombination mit einer automatischen Testdurchführung findet eine automatische Testrealisierung statt. Aus logischen Testfällen, welche etwa in Testbeschreibungssprachen wie TTCN-3 (European Telecommunications Standards Institute (ETSI), 2005) oder dem Testing Profile der UML 2 (OMG, 2005) beschrieben sind, werden ausführbare Instanzen von Testfällen gebildet.

Die nach Auffassung des Autors größte Herausforderung stellt die Automatisierung des Testdesigns dar. Zur automatischen Ermittlung von Testfällen (Testfallgenerierung) wurden bisher verschiedene modellbasierte Ansätze vorgeschlagen, welche im folgenden Abschnitt näher erläutert werden. In der Praxis findet die automatische Ermittlung (logischer) Testfälle bisher vergleichsweise wenig Verbreitung. Eine Ursache ist häufig des Fehlen von ausreichend dokumentierten Spezifikationen beziehungsweise von Modellen, aus welchen logische Testfälle automatisch abgeleitet werden können. Aber auch ein stärkeres Vertrauen in die Erfahrung und Intuition der Testingenieure, die

„richtigen“ Testfälle zu wählen, ist ein möglicher Grund, ein manuelles Testdesign zu bevorzugen.

2.3. Modellbasierte Testfallgenerierung

Die modellbasierte Testfallgenerierung dient der Automatisierung des Testdesigns, genauer der Testfallermittlung. Eine Überblick über bestehende Techniken geben (Broy u. a., 2005), (Utting u. a., 2006) und (Utting und Legeard, 2006).

2.3.1. Ausprägungen

Der Begriff der *modellbasierten Testfallgenerierung* findet für unterschiedliche Vorgehensweisen Verwendung. Laut Utting und Legeard (2006, S. 7) werden damit folgende Vorgehensweisen bezeichnet:

- die Generierung von Eingabedaten aus einem Domänenmodell,
- die Generierung von Testeingaben aus einem Umgebungsmodell,
- die Generierung von Testfällen mit erwarteten Ausgaben aus einem Verhaltensmodell des Testobjekts sowie
- die Generierung von Testskripten aus abstrakten Testfällen.

Sollen nur Eingabedaten ermittelt werden, kann dies mithilfe eines Datenmodells der Domäne erfolgen. Aus diesem können unterschiedliche Kombination von Eingaben generiert werden, etwa durch paarweise Kombinationen von Werten, wie etwa von Tai und Lei (2002) beschrieben. Allerdings können damit keine vollständigen Testfälle ermittelt werden, welche auch die erwarteten Ausgaben des Systems umfassen. Es werden dabei auch nur Eingaben zu einem Zeitpunkt an das System betrachtet. Zur Erzeugung von Testsequenzen, welche Eingaben in mehreren aufeinander folgenden Testschritten umfassen, sind diese Methoden nur eingeschränkt geeignet, da die kombinatorische Komplexität auch für einfache Datentypen über mehrere Testschritte sehr schnell zu einer zu einer praktisch nicht mehr handhabbaren Menge an Testfällen führt.

Eine weitere Möglichkeit stellt die Gewinnung von Testeingaben aus einem Umgebungsmodell dar. Das Modell beschreibt in diesem Fall die Umgebung, in welcher das zu testende System eingesetzt wird. Verwendung finden hier vor allem Nutzungsprofile (Musa und Ackerman, 1989), welche das Verhalten der Benutzer des Systems beschreiben. Da ein Umgebungsmodell nur die Umgebung beschreibt, nicht aber das Verhalten des zu testenden Systems, können auch hier nur Eingabedaten gewonnen werden.

Liegt dagegen ein Modell des zu testenden Systems vor, welches dessen Verhalten (abstrakt) beschreibt, können sowohl Testeingaben als auch die zu erwarteten Soll-Ausgaben aus dem Modell generiert werden. Das Modell fungiert in diesem Fall auch als Orakel der erwarteten Ausgaben. Diese Ausprägung der modellbasierten Testfallgenerierung wurde unter anderem von Lötzbeyer (2003) und Pretschner (2003) beschrieben. Sie ist auch der Gegenstand der vorliegenden Arbeit. Die hier beschriebene Methode baut auf den beiden genannten Arbeiten auf. Die Generierung von Testfällen aus einem Verhaltensmodell wird in dem nachfolgenden Abschnitt detaillierter betrachtet.

Die zuletzt genannte Anwendungsmöglichkeit, Testskripte aus abstrakten Testfällen zu generieren, bedeutet die Instanziierung von abstrakten Testfällen zu ausführbaren und ist damit vor allem eine Möglichkeit der Testrealisierung. Bisher gibt die Literatur keine klare Abgrenzung zwischen Testfallinstanziierung und Testfallermittlung (Testdesign) an. Intuitiv wird von Testfallinstanziierung gesprochen, falls das Modell bereits einen logischen Ablauf eines Testfalls vorgibt und für die Wahl des konkreten Testfalls keine weiteren Präferenzen zu berücksichtigen sind. Andererseits kann der abstrakte Testfall auch als Testfallspezifikation verstanden werden und das Modell Informationen über die Struktur und die Schnittstelle des Testobjekts beschreiben (vgl. Utting und Legeard (2006)).

2.3.2. Verhaltensmodellierung

Modelle sind allgemein ein vereinfachte Abbildung der Realität. In der Software-Entwicklung sind Modelle dabei meist ein Abbild des zu entwickelnden beziehungsweise des entwickelten Systems. Daneben werden auch die bereits erwähnten Umgebungsmodelle verwendet. Zur Beschreibung des Entwicklungsprozesses kommen weiter Prozessmodelle zum Einsatz.

Die modellbasierte Testfallgenerierung, wie sie Gegenstand dieser Arbeit ist, verwendet ein Verhaltensmodell des zu testenden Systems, um daraus Testfälle zu gewinnen. Voraussetzung ist dabei, dass das Modell das Verhalten *abstrakt* beschreibt, also logische Systemabläufe aus Eingaben und erwarteten Ausgaben wiedergibt und nicht deren konkrete technische Umsetzung. Idealerweise beschreibt das Modell das Verhalten des Systems nur in der Weise, wie es für die Nutzer des Systems von Bedeutung ist. Verhaltensanteile, welche nur im Zusammenhang mit der gewählten technischen Realisierung gültig sind, sind dagegen nicht Bestandteil des Testmodells. Ein Testmodell kann das Verhalten auch nur ausschnittsweise wiedergeben, falls zum Beispiel nur einzelne (Teil-)Funktionen des Systems Gegenstand des (modellbasierten) Tests sein sollen oder mehrere unterschiedliche Modelle für einzelne Funktionalitäten erstellt werden.

Abstraktion

Die Forderung nach einer Abstraktion von dem zu testenden System ist essentiell für das Testmodell, sofern die Testfälle auch die erwarteten Ausgaben umfassen sollen. Würde etwa der Programmcode als Modell zur Testfallgenerierung verwendet, würden sich Fehler im Code und ebenso in den Testfällen wiederfinden. Da Testfälle und Programm in diesem Fall keine Abweichungen hinsichtlich der Ausgaben aufweisen können, könnten durch die Tests keine Fehler aufgedeckt werden. Allenfalls Fehler bei der Übersetzung des Programmcodes können zu einem abweichenden Verhalten führen. Daraus folgt, dass in einer solchen Situation der Compiler das Testobjekt ist, nicht das entwickelte Programm. Gleiches gilt im Rahmen der modellbasierten Softwareentwicklung für Modelle, welche zur Generierung von Programmcode dienen. Sie sind nicht als Testmodelle geeignet, da damit im Wesentlichen der Codegenerator und nicht das entwickelte System getestet würde.[1] Allenfalls zur Ermittlung von Testeingaben können diese Modelle in der Testfallgenerierung verwendet werden.

Prenninger und Pretschner (2005) beschreiben *funktionale Abstraktion*, *Datenabstraktion*, *Abstraktion der Kommunikation* und *zeitliche Abstraktion* als Prinzipien, welche bei der Erstellung von Testmodellen Verwendung finden. Die Abstraktion bedeutet demnach einen bewussten Verzicht auf Information, wobei die verbliebene Information als korrekt angesehen wird. Potentielle Fehler können auch zur Bestimmung einer geeigneten Abstraktion verwendet werden, wie von Struss (1994) beschrieben. Wie dort auch dargestellt ist, ist die Datenabstraktion insbesondere bei Eingabedaten über kontinuierlichen Wertebereichen essentiell.

Spezifikations- und Testmodelle

In der modellbasierten Softwareentwicklung werden Modelle dazu verwendet, eine Spezifikation der Funktionalität des zu entwickelnden Systems anzugeben. Diese Modelle sind somit nicht eigens für den Test erstellt worden, sondern sind Teil oder Ergebnis der Erhebung von Anforderungen.

Die im vorhergehenden Abschnitt erläuterte elementare Forderung, dass Testmodelle vom Testobjekt abstrahieren müssen, erfüllen zumeist auch Modelle der Systemspezifikation, welche (auch) zur Realisierung des Systems verwendet werden. Im Rahmen der Realisierung werden die Modelle konkretisiert, das bedeutet die Informationen, welche auf Grund der Abstraktion in dem Modell nicht vorhanden sind, werden hinzugefügt. Diese Ergänzung von Informationen beim Übergang von Spezifikation zu Realisierung ist eine Quelle von möglichen Fehlern; in der Realisierung ist allgemein nicht mehr sichergestellt, dass das abstrakte Verhalten, welches durch das Spezifikationsmodell

[1] Neben Fehlern des Codegenerators können ggf. auch Fehler in anderen Komponenten, welche zur Programmerzeugug (z. B. Compiler) oder Programmausführung (Betriebssystem, Middleware) verwendet wurden, aufgezeigt werden

angegeben ist, erhalten bleibt. Wird das Modell der Systemspezifikation als Testmodell verwendet, können derartige Fehler, die durch den Übergang von der abstrakten Spezifikation zu konkreten Realisierung entstehen, durch die Testfälle aufgedeckt werden.

Der in dieser Arbeit vorgestellte Ansatz geht von einem Verhaltensmodell aus, welches das Systemverhalten insgesamt spezifiziert, es ist dabei unerheblich, ob das Modell im Rahmen der Anforderungserhebung bereits vor der Entwicklung des Systems oder erst später eigens zum Zwecke der modellbasierten Testfallgenerierung erstellt wurde.

Validierung

Essentiell für die modellbasierte Testfallgenerierung ist die Validierung des Testmodells. Dies kann einerseits durch einen Korrektheitsnachweis von geforderten Eigenschaften im Modell erfolgen, beispielsweise mit Techniken des *model checking* (Clarke Jr. u. a., 1999). Allerdings verlagert sich somit das Problem der Validierung auf die Validierung der spezifizierten Eigenschaften. Eine weitere Möglichkeit stellen Simulationen dar, in welchen die Stakeholder des Systems das durch das Modell beschriebene Verhalten beurteilen. Hier können wiederum aus dem Modell gewonnene Testfälle zum Einsatz kommen, wobei die Stakeholder durch die Simulation die Testfälle analysieren und prüfen, ob die Ausgaben ihren Erwartungen entsprechen. Die modellbasierte Testfallgenerierung kann somit auch zur Validierung des Verhaltensmodells eingesetzt werden. Allerdings sollten dann für die Ausführung auf dem Testobjekt zusätzliche Testfälle aus dem Modell gewonnen werden.

Die Validierung des Modells beziehungsweise die Modellerstellung trägt aber auch zur Validierung und Präzisierung der Anforderungen selbst bei und erhöht somit die Qualität der Anforderungen, wie es auch von Pretschner u. a. (2005) beobachtet wurde.

2.3.3. Methodik der modellbasierten Testfallgenerierung

In Abbildung 2.1 ist der grundlegende Prozess der modellbasierten Testfallgenerierung angegeben. Aus den Anforderungen wird ein Testmodell abgeleitet. Die Anforderungen können in Anforderungsdokumenten explizit formuliert sein oder aber Gegenstand eines mentalen Modells (Binder, 1999) sein. Das Testmodell repräsentiert die Erwartungen der Nutzer des Systems an das Systemverhalten, im Rahmen des funktionalen Tests sind hierbei nur funktionale Anforderungen von Bedeutung. Wie im vorhergehenden Abschnitt diskutiert, ist eine zentrale Forderung an das Testmodell die Abstraktion von dem zu testenden System. Das Testmodell soll nur das logische Verhalten wiedergeben ohne auf (vor allem technische) Details der gewählten Realisierung

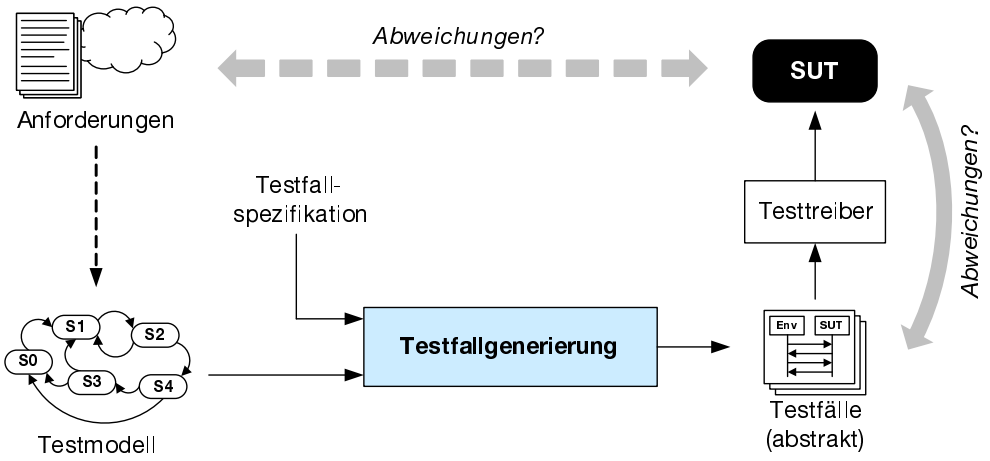

Abbildung 2.1.: Grundlegender Prozess der modellbasierten Testfallgenerierung

einzugehen. Es handelt sich also um ein Modell der funktionalen Anforderungen, sofern die entsprechenden Anforderungen Gegenstand des Tests sein sollen.

Das Testmodell ist eine Eingabe für die Testfallgenerierung, eine weitere ist die Testfallspezifikation. Sie stellt das Auswahlkriterium dar, nach dem aus der (potentiell unendlichen) Menge aller möglichen Abläufe, welche das Testmodell beschreibt, eine endliche Teilmenge von Abläufen als Testsuite (Menge von Testfällen) selektiert wird. Die Testfallspezifikation definiert also Eigenschaften, welche eine durch die Testfallgenerierung ermittelte Testsuite erfüllen muss. Auf unterschiedliche Arten von Testfallspezifikationen wird im folgenden Abschnitt näher eingegangen.

Ausgabe der Testfallgenerierung ist somit eine Menge abstrakter Testfälle. Die Testfälle sind abstrakt, da sie Abläufe des abstrakten Verhaltensmodells sind. Aus diesem Grund können sie nicht unmittelbar auf dem Testobjekt ausgeführt werden. Die Übersetzung der generierten abstrakten Testfälle in konkrete ausführbare Testfälle wird durch sogenannte Testtreiber vorgenommen. Der Testtreiber muss somit die gewählte Abstraktion zwischen logischer Verhaltensbeschreibung und technischer Realisierung überbrücken. Die Eingaben der konkreten Testfälle werden an das zu testende System (engl. *system under test*, SUT), das Testobjekt, übergeben und die vom Testobjekt produzierten Ausgaben erfasst. Für die Entscheidung, ob ein Testfall erfolgreich war oder fehl geschlagen ist, muss entweder eine Rückübersetzung der konkreten Ausgaben in die abstrakten Ausgaben vorgenommen werden, dann findet der Vergleich der Soll- mit den Ist-Ausgaben auf der abstrakten Ebene statt. Oder aber die Konkretisierung der Testfälle umfasst auch die Soll-Ausgaben des abstrakten Testfalls und der Vergleich findet auf der konkreten Ebene statt. Da im Allgemeinen ein abstrakter Testfall auf eine Menge von konkreten Testfällen abgebildet werden kann, ist die Übersetzung, welche der Testtreiber vorzunehmen hat, nicht trivial.

Durch die Ausführung der Testfälle und Vergleich der vom Testobjekt produzierten Ist-Ausgaben mit den in den Testfällen erwarteten Soll-Ausgaben können Abweichungen zwischen den Testfällen und dem tatsächlichen Verhalten des Testobjekts festgestellt werden. Da die Testfälle aus einem Modell gewonnen wurden, welches die Anforderungen an das System wiedergibt, deckt eine beobachtete Abweichung bei der Ausführung der Testfälle auch eine Abweichung zwischen dem realisierten System und den Anforderungen auf. Im Umkehrschluss wird das Vertrauen in die Korrektheit der Realisierung erhöht, falls keine Abweichungen festgestellt werden.

Diese Schlussfolgerung gilt allerdings nur unter der Annahme, dass auch der Übergang von Anforderung zu Testfällen korrekt erfolgt ist. Insgesamt ergeben sich folgende mögliche Ursachen für eine im Test identifizierte Abweichung:

- die Realisierung des Systems ist fehlerhaft
- die Wiedergabe der Anforderungen durch das Modell ist fehlerhaft, die Realisierung aber korrekt
- die Übersetzung abstrakter in konkreter Testfälle durch den Testtreiber ist fehlerhaft (dies schließt die fehlerhafte Übersetzung der Ausgaben mit ein)
- die Interpretation des Modells durch den Testfallgenerator ist fehlerhaft

Insbesondere die Ableitung eines Verhaltensmodells aus den Anforderungen (welche häufig nur unvollständig und gegebenenfalls auch widersprüchlich dokumentiert sind) kommt hier eine hohe Bedeutung und ein hohes Fehlerpotenzial zu. Aus diesem Grund ist eine Validierung des Verhaltensmodells unerlässlich.

Aus der Abbildung ist ersichtlich, dass in diesem Szenario kein Zusammenhang zwischen (Test-)Modell und dem zu testenden System angenommen wird (außer jener, nachdem beide Artefakte den gleichen Anforderungen im nach außen sichtbaren Verhalten genügen sollen). Das zu testende System (SUT) wird als „Black-Box“ angenommen, dessen innere Struktur unbekannt ist, insbesondere ist unbekannt, ob die interne Struktur des SUT der Struktur des Testmodells entspricht. Gleichwohl schließt dies nicht aus, dass das Modell als Verhaltensspezifikation auch bei der Realisierung des Systems Verwendung gefunden hat.

Neben diesem Einsatzszenario beschreibt Pretschner (2003) auch andere Szenarien der modellbasierten Testfallgenerierung. Dies sind die Verwendung eines gemeinsamen Modells zur Testfall- und Programmcodegenerierung sowie die Extraktion des Testmodells aus dem Programmcode.

2.4. Testfallspezifikation

Im vorhergehenden Abschnitt wurde bereits dargestellt, dass das Testmodell allein noch keine hinreichende Eingabe für eine Testfallgenerierung ist. Eine weitere notwendige Eingabe ist die *Testfallspezifikation*. Eine Testfallspezifikation ist eine formale Repräsentation einer Testsuite bezogen auf einen Testfallgenerator (Pretschner und Leucker, 2005). Im Gegensatz zum Testziel, welches nicht notwendigerweise operationalisierbar sein muss, wird von einer Testfallspezifikation gefordert, dass sie derart präzise Eigenschaften definiert, welche eine Testsuite zu erfüllen hat, dass mithilfe eins Testfallgenerators ein solche Testsuite ermittelt werden kann. Die Testfallspezifikation legt somit fest, welche Abläufe des Modells als Testfälle auszuwählen sind.

Im Rahmen des modellbasierten Test existieren unterschiedliche Klassifikationen von Arten von Testfallspezifikationen. Pretschner (2003); Pretschner und Leucker (2005) unterscheiden funktionale, strukturelle und stochastische Testfallspezifikationen während Utting und Legeard (2006) feiner zwischen struktureller Überdeckung, Datenüberdeckung, fehlerbasierten Kriterien, anforderungsbasierten Kriterien, expliziten Testfallspezifikationen und statistischen Methoden unterscheiden. Im Folgenden werden die verschiedenen Arten charakterisiert, eine Diskussion von Vor- und Nachteilen vor allem von strukturellen und funktionalen Testfallspezifikationen wird später in Abschnitt 6.1 geführt.

2.4.1. Randomisierte und statistische Testfallspezifikationen

Die zufällige Auswahl von Testfällen stellt eine erste Möglichkeit einer Testfallspezifikation dar. Zu unterscheiden ist einerseits die zufällige Auswahl von Testeingaben und andererseits die zufällige Auswahl von Modellelementen des Testmodells bei dessen Ausführung. Beispielsweise wird dazu in jedem Testschritt eine der möglichen Transitionen einer Zustandsmaschine zufällig gewählt. Die Eingabe ist dann entsprechend zu belegen, so dass dieser Zustandsübergang ausgelöst wird. Der Testfall wird durch Berechnung der erwarteten Soll-Ausgaben zu den Testeingaben im Modell vervollständigt.

Statistische (stochastische) Testfallspezifikationen sind randomisierte Testfallspezifikationen, bei denen die zufällige Auswahl nicht gleichverteilt vorgenommen wird, sondern die Wahrscheinlichkeiten für die Wahl von Eingaben oder Modellelementen unterschiedlich gewählt werden. Hierzu werden vor allem Nutzungsprofile (Musa, 1993) verwendet. Nutzungsprofile beschreiben die erwartete Nutzung des Systems. Damit werden häufiger ausgeführte Abläufe beim Einsatz des Systems von weniger häufig ausgeführten unterschieden. Die Motivation, dies als Testauswahlkriterium zu verwenden rührt daher, die Zahl der Fehlsituationen im praktischen Einsatz verringern zu wollen; eine Fehler, welcher nur in einem Szenario auftritt, welches nie zur Ausführung kommt, wäre demnach zu vernachlässigen. Für die Testfallermittlung wurden

Nutzungsprofile im Rahmen von Cleanroom (Prowell u. a., 1999) verwendet. Whittaker und Thomason (1994) verwenden Nutzungsmodelle basierend auf Markov-Ketten zur Testfallgenerierung.

2.4.2. Strukturorientierte Testfallspezifikationen

Eine strukturorientierte Testfallspezifikation definiert Eigenschaften anhand der Elemente des Modells, welche eine Testsuite erfüllen muss. Einfachstes Beispiel ist die Überdeckung von Kontrollzuständen oder Transitionen einer Zustandsmaschine. Dabei wird von der Testsuite gefordert, dass mit den darin enthaltenen Testfällen jeder Kontrollzustand beziehungsweise jede Transition des Modells mindestens einmal erreicht beziehungsweise ausgeführt wurde. Diese Überdeckungskriterien entsprechen im Modell jenen der Anweisungs- oder Zweigüberdeckung von Programmcode. Da im Fall von erweiterten Zustandsmaschinen (*extended finite state machines*, EFSM) die Wächter der Transitionen auch komplexe Bedingungen umfassen können, kommen oft auch Bedingungsüberdeckungen zum Einsatz. Ein Beispiel dafür ist die modifizierte Bedingungs-/Entscheidungsüberdeckung (*modified condition/decision coverage*, MC/DC) (Hayhurst u. a., 2001) welche für Steuergerätesoftware der zivilen Luftfahrt gefordert wird (RTCA, 1992; EUROCAE, 1992).

Datenüberdeckungen sind geforderte Eigenschaften auf Variablen des Modells. Da sich der Zustand eines Modells aus dem Kontroll- und Datenzustand zusammensetzt, sind sie eine Form der Zustandsüberdeckung auf dem Datenzustand. Eine vertiefende Übersicht über unterschiedliche strukturelle Überdeckungskriterien findet sich in Utting und Legeard (2006, Kapitel 4).

2.4.3. Funktionsorientierte Testfallspezifikationen

Funktionsorientierte Testfallspezifikationen werden aus den Anforderungen an die Funktionalität des zu testenden Systems abgeleitet. Beispielsweise können (unvollständige) Szenarien in Anforderungsdokumenten als Testfallspezifikation verwendet werden. In der Testfallgenerierung wird dann nach Testfällen gesucht, welche Instanzen dieser Szenarienspezifikationen sind, und diese werden zu vollständigen Abläufen durch den Testfallgenerator ergänzt. Funktionsorientierte Testfallspezifikationen lassen sich in *anforderungsbasierte* und *fehlerbasierte* Testfallspezifikationen unterscheiden.

Anforderungsbasiert

Anforderungsbasierte Testfallspezifikationen fordern eine Abdeckung der betrachteten Anforderungen durch die Testfälle. Meist werden dafür vergleichsweise einfache Kriterien verwendet, wie etwa dass zu jeder Anforderung ein Testfall ermittelt werden

soll. Whalen u. a. (2006) schlagen dazu eine Abedeckungsmetrik für Anforderungen vor, welche in linearer temporaler Logik spezifiziert sind. Abgesehen davon wird eine Anforderungsüberdeckung entweder durch explizite Testfallspezifikationen für jede einzelne Anforderung erreicht, oder im Modell werden Modellelemente mit Annotationen versehen, welche auf einzelne Anforderungen verweisen.

Beide Vorgehensweisen erfordern, dass eine funktionale Anforderung in Aussagen über die Elemente und Struktur des Testmodells überführt wird. Im Falle der Annotation von Modellelementen besteht zudem die Schwierigkeit, dass eine eineindeutige Zuordnung von Modellelementen zu Anforderungen meist nicht möglich ist, in der Regel wird eine Anforderung durch eine Menge von Modellelementen realisiert, andererseits trägt ein Modellelement möglicherweise zur Realisierung mehrerer Anforderungen bei.

Eine weitere Schwierigkeit stellt die Validierung einer expliziten Testfallspezifikation dar, letztlich müssten die Stakeholder beurteilen können, ob die Testfallspezifikation, welche Aussagen über Modellelemente umfasst, eine ausreichende Wiedergabe der Anforderung ist. Die explizite Formulierung jeder einzelnen Anforderung stellt zudem einen hohen Aufwand dar. Zu beachten ist auch, dass ein ausreichender Test nur dann erfolgt, falls alle geforderten Anforderungen sich in einer Testfallspezifikation wiederfinden. Insbesondere vor dem Hintergrund, dass oftmals viele implizite Annahmen an das Verhalten des Systems getroffen sind, welche nicht explizit als Anforderung formuliert sind, stellt dies ein Problem dar.

Fehlerbasiert

Ebenfalls den funktionsorientierten Testfallspezifikationen können fehlerbasierte Testfallspezifikation zugeordnet werden. Diese beschreiben Situationen, in welchen das System die geforderten Eigenschaften nicht erfüllt. Beispielsweise kann es sich dabei um mit Fehlern behaftete Szenarien oder auch Zustandsmaschinen handeln. Esser und Struss (2006, 2007) verwenden dazu explizite Modelle des Fehlverhaltens zusätzlich zum Modell des korrekten Verhaltens. Zur Testfallgenerierung wird nun versucht, Eingaben zu identifizieren, in welchen das Fehlermodell und das korrekte Modell ein unterschiedliches Verhalten aufweisen. Diese Eingaben liefern Testfälle für hinsichtlich des Fehlers, der durch das Fehlermodell beschrieben wurde.

2.5. Zusammenfassung

Wird das erwartete Soll-Verhalten eines reaktiven Systems durch ein Testmodell beschrieben, kann damit die Testfallermittlung automatisiert werden. Dieses Modell ist die wesentliche Eingabe für eine modellbasierte Testfallgenerierung. Eine weitere notwendige Information zur Testfallgenerierung ist die Testfallspezifikation. Diese

stellt das Auswahlkriterium dar, nach welchen bestimmte Ausführungspfade im Modell als Testfälle ausgewiesen werden. Neben der zufälligen Auswahl von Testfällen, werden dazu strukturelle und funktionale Testfallspezifikationen verwendet. Bei der Verwendung von strukturellen Kriterien besteht in der Regel kein direkter Bezug zwischen einzelnen Anforderungen und Testfällen, außerdem kann der Testingenieur seine Erfahrung und Intuition nicht in die Testfallermittlung einfließen lassen. Funktionale Testfallspezifikationen erfordern dagegen, dass zu jeder einzelnen Anforderung explizit eine Testfallspezifikation erstellt wird oder die betroffenen Modellteile manuell im Modell annotiert werden. Aus diesen Beobachtungen leitet sich das wesentliche Ziel der vorliegenden Arbeit ab: Wie kann eine modellbasierte Testfallgenerierung eingesetzt werden, um weitgehend automatisiert zu einer Menge von einzelnen Anforderungen Testfälle erzeugen zu können. Dabei soll die Methode es auch erlauben, dass Testingenieure ihre Erfahrung und Intuition in die Testfallgenerierung einfließen lassen können, um der jeweiligen Projektsituation gerecht zu werden.

Entscheidend für die Auswahl von Testfällen sind die Qualitätskriterien, welche an eine zu ermittelnde Testsuite gestellt werden. Im folgende Kapitel wird deshalb zunächst die Qualitätsbewertung von Testsuiten eingehend diskutiert.

3. Qualitätsbewertung von Testsuiten

Die wesentliche Grundlage für die Auswahl einer Testmethode ist die Qualität der Testfälle, welche mit dieser Testmethode ermittelt werden. Dadurch stellt sich unmittelbar die Frage nach Kriterien zur Bewertung der Qualität von Testfällen. Es ist offensichtlich, dass hier im Allgemeinen nicht die Qualität eines einzelnen Testfalls von Bedeutung ist, sondern die Qualität der Menge von Testfällen (Testsuite), welche eine Methode erzeugt.

Die Gütebewertung von Testfällen beziehungsweise Testsuiten hängt von dem zu optimierenden Ziel innerhalb des Software-Lebenszyklus ab: Sind die Kosten für die *Testausführung* hoch, und sollen diese reduziert werden, sind „gute“ Testfälle vor allem möglichst kurze Testfälle (welche wenig Ausführungszeit benötigen) und kleine Mengen von Testfällen vorzuziehen. Wird dagegen die Erhöhung der *Produktqualität* als vorrangiges Ziel verfolgt, sollen Testfälle mit möglichst hoher Wahrscheinlichkeit mögliche Fehler aufdecken. Dabei ist nicht allein die Zahl der Fehler von Bedeutung sondern auch ihre Eintrittswahrscheinlichkeit und die Folgekosten der Fehler. Präziser formuliert: Durch die Eliminierung der aufgedeckten Fehler lassen sich die Folgekosten, welche im Falle ihrer Nicht-Entdeckung über den gesamten Lebenszyklus des Systems entstanden wären, reduzieren.

Wie in (Pfaller u. a., 2008) dargestellt, ist eine Generalisierung von Qualitätsaussagen zu einer Testmethode hinsichtlich der von ihr erzeugten Testsuiten nur selten möglich. Hierzu ist es erforderlich, den Nachweis zu führen, dass eine Testmethode in jeder Projektsituation, in welcher sie eingesetzt werden soll, eine Testsuite von vergleichbarer Qualität erzielt. Die vielfältigen Einflussfaktoren unterschiedlicher Projekte machen diesen Nachweis häufig unmöglich. Besonders problematisch stellt sich die Situation im Falle der Betrachtung gefundener Fehler dar: Die Anzahl und Art der vorhandenen Fehler ist in Projekten in der Regel unbekannt – findet eine Methode nun viele Fehler in einem Projekt, wäre unter anderem zu zeigen, dass in anderen Projekten eine vergleichbare Verteilung der Fehler zu erwarten ist. Eine Aussage zur Testqualität bezogen auf gefundene Fehler in früheren Projekten lässt sich auf künftig zu testende Systeme nur übertragen, wenn sich auch eine vergleichbare Fehlerverteilung in den künftigen Projekten garantieren lässt.

In diesem Kapitel wird zunächst ein Überblick über bekannte und häufig verwendete Arten der Qualitätsbewertung von Testfällen gegeben, danach werden unterschiedliche Arten von Qualitätskriterien systematisiert. Abschließend wird das dieser Arbeit zu

Grunde liegende Kriterium, die Überdeckung von Anforderungen, in diese Systematisierung eingeordnet und beschrieben. Dieses Kapitel ist auf den Softwaretest im Allgemeinen anwendbar, nicht nur auf die im Rahmen dieser Arbeit betrachteten reaktiven Systeme. Gleichwohl sind die getroffen Aussagen und Schlussfolgerungen für reaktive Systeme gültig.

3.1. Bekannte Qualitätsbewertungen

Werden Testmethoden bezüglich ihrer Qualität bewertet, wird meist auf eine von zwei Bewertungen zurückgegriffen: Strukturelle Abdeckung im Testobjekt oder in Modellen des Testobjekts einerseits und die Anzahl entdeckter Fehler andererseits. Die häufige Anwendung struktureller Überdeckungskriterien ist vor allem auf ihre vergleichsweise einfache Anwendung zurückzuführen, obwohl, wie im folgenden dargestellt wird, ihre Aussagekraft hinsichtlich der Produktqualität beschränkt ist. Das heißt, dass auch bei erfüllter struktureller Überdeckung nicht garantiert ist, dass die Anzahl der vorhanden Restfehler ausreichend gering ist. Dagegen gilt die Aufdeckungsrate von Fehlern, insbesondere von schwerwiegenden Fehlern, als das Ideal der Gütebewertung von Testfällen, da hiermit die Produktqualität unmittelbar einhergeht. Allerdings lässt sich eine solche Bewertung zum Zeitpunkt der Testerstellung in der Praxis nicht durchführen, da Art und Anzahl der vorhanden Fehler im zu untersuchenden System unbekannt sind. Um dieses Problem zu umgehen, werden oft Aussagen über Tests von Systemen getroffen, bei denen Fehler bereits bekannt sind – in diesen Fällen besteht die Schwierigkeit einer Verallgemeinerung: In wie weit lassen sich die Beobachtungen auf den Test anderer Systeme übertragen? Außerdem wird meist nur die Zahl der Fehler betrachtet, nicht deren Schwere.

Häufig setzen sich Testmethoden das Ziel, zumindest eines dieser beiden Kriterien zu erfüllen. Allerdings muss ihre Eignung zur Bewertung von Testmethoden kritisch hinterfragt werden. Dazu wird zunächst näher auf strukturelle Überdeckungen eingegangen und im Anschluss die Bewertung der Fehleraufdeckung betrachtet.

3.1.1. Strukturelle Überdeckung

Strukturelle Überdeckungskriterien messen die Güte einer Menge von Testfällen anhand der Programmkonstrukte im Testobjekt, welche durch die Tests erreicht werden. Bekannte Vertreter sind beispielsweise Anweisungsüberdeckung, Zweigüberdeckung oder Bedingungsüberdeckung. Diese werden seit geraumer Zeit zur Ableitung und Qualitätsbewertung von Testfällen verwendet, vgl. hierzu zum Beispiel (Myers, 1979). Die Vorteile struktureller Überdeckungen sind vor allem durch zwei Eigenschaften bestimmt:

- Überdeckungen, sofern sie sinnvoll gewählt sind, stellen sicher, dass alle Teile des implementierten Programms in Testfällen berücksichtigt wurden.

- Die Bestimmung der erreichten Überdeckung ist im Allgemeinen sehr einfach anhand des Programmcodes durchzuführen. Diese lässt sich auch automatisiert ermitteln.

Die Erreichung einer gewissen strukturellen Abdeckung von Programmcode hat zur Qualitätsbeurteilung von Testfällen bereits vielfach Einfluss in Normen und Standards zur Softwareentwicklung gefunden. Im Bereich der zivilen Luftfahrtindustrie wird etwa im Standard DO-178B / ED-12B (RTCA, 1992; EUROCAE, 1992) die *modifizierte Bedingungs-/Entscheidungsüberdeckung* (modified condition/decission coverage, MC/DC), vgl. Hayhurst u. a. (2001), gefordert.

Ursprünglich wurden diese Kriterien für die Überdeckung von Programmcode definiert. Im Zusammenhang mit einer modellbasierten Entwicklung können die Überdeckungskriterien auf Modelle übertragen werden, siehe dazu auch Abschnitt 2.3. Die einfache Messbarkeit sowie die Forderungen von Normen begründen die Existenz vielfacher Techniken und Werkzeuge zur automatisierten Generierung von Testsuiten, welche eine Überdeckung im Modell garantieren: Von Pretschner und Lötzbeyer (2001); Pretschner (2003) wird die Generierung von MC/DC-Testsuiten aus erweiterten Zustandsmaschinen[1] mit Hilfe von Constraint Logik Programmierung (Frühwirth und Abdennadher, 1997; Apt und Wallace, 2007) beschrieben. Testfallgenerierung mittels Modellprüfung *(model checking)* zielt meist ebenso auf die Erfüllung struktureller Überdeckungen ab. Dies wurde unter anderem von Rayadurgam und Heimdahl (2001, 2003) sowie Hamon u. a. (2004, 2005) vorgeschlagen.

Bezogen auf eine Verbesserung der Produktqualität müssen strukturelle Überdeckungskriterien kritisch betrachtet werden: Heimdahl u. a. (2004) zeigen auf, dass die strukturelle Überdeckung eines Modells nur unzureichend auch einer Überdeckung im Programmcode gerecht wird. Von Rajan u. a. (2008) wird weiter dargestellt, dass der Grad der MC/DC Überdeckung, welcher mit einer Testsuite erreicht wird, in semantisch äquivalenten Programmen sehr unterschiedlich sein kann – die Anzahl der Testfälle, welche zur Erfüllung einer Überdeckung benötigt werden hängt also sehr stark von der internen Struktur des Testobjekts ab. Cai und Lyu (2005) argumentieren, dass höhere strukturelle Überdeckung nur in bestimmten Testsituationen eine gute Korrelation zu höherer Fehleraufdeckung aufweist, etwa falls sich die Tests besonders auf den Test von Ausnahmebedingungen statt der gewöhnlichen Nutzung fokussieren.

[1] Analog zur oben erwähnten MC/DC-Überdeckung von Programmcode werden hier die Bedingungen an den Transitionen der Zustandsmaschine betrachtet: Jedes Literal in der Bedingung soll einmal zu `true` und einmal zu `false` evaluiert werden, außerdem soll jedes Literal so belegt werden, dass die Bedingung einmal zu `true` und einmal zu `false` evaluiert. Vollständige MC/DC-Testsuiten sind dabei nicht immer existent.

Zusammenfassend ist festzustellen, dass eine Qualitätsbewertung von Testfällen nach strukturellen Überdeckungskriterien vor allem dann ihre Berechtigung hat, wenn das entsprechende Kriterium gerade nicht konstruktiv eingesetzt wurde, sondern nur zur Qualitätsbeurteilung einer vorhanden Testsuite herangezogen wurde. Werden allerdings gezielt Testfälle zur Erfüllung einer Überdeckung erzeugt, kann die Qualitätsaussage dieser Testfälle bezogen auf die Produktqualität zumindest angezweifelt werden. Die wesentlichen Kritikpunkte sind dabei:

- Geforderte Funktionalität, welche nicht implementiert ist, bleibt unberücksichtigt.
- Modelle beziehungsweise Code können semantisch äquivalent transformiert werden, so dass das Überdeckungskriterium mit weniger Aufwand, also mit einer reduzierten Testsuite, erfüllbar ist. Durch die äquivalente Umformung bleiben aber alle Fehler im Code erhalten. Die reduzierte Testsuite verfügt im Allgemeinen über geringeres Potential der Fehlerentdeckung, das Überdeckungskriterium ist jedoch gleichermaßen erfüllt.
- Die Testfälle betrachten im Allgemeinen nicht, worin der übliche Nutzungskontext besteht.
- Eine Unterscheidung hinsichtlich der möglichen Fehlerwirkungen wird nicht getroffen: (Teil-)Funktionalitäten, bei denen falsche Ausgaben zu folgenschwereren Auswirkungen führen, werden im gleichen Maße berücksichtigt wie weniger kritische Funktionen.

Bisher wurde nicht abschließend geklärt, in welchem Maße und unter welchen Umständen strukturelle Überdeckungen mit einer hohen Produktqualität korrelieren. Der Softwaretest trägt vor allem zur Steigerung der Produktqualität bei, wenn er eine hohe Zahl von Fehler, insbesondere folgenschwere Fehler mit hoher Eintrittswahrscheinlichkeit, möglichst verlässlich identifiziert. Die Anzahl der nicht entdeckten Restfehler soll nach Abschluss einer Testphase somit möglichst gering sein, und insbesondere sollen häufig ausgeführte Interaktionen sowie kritische Systemabläufe weitgehend fehlerfrei sein. Es bleibt unklar, ob die Erfüllung einer strukturellen Überdeckung diese Forderungen in ausreichendem Maße sicherstellen kann.

Durch Aussagen über die Fehleraufdeckung soll die Qualität einer Testsuite oder einer Testmethode hinsichtlich der der Verbesserung der Produktqualität erreicht werden. Allerdings ist sowohl die Anwendung wie auch Verallgemeinerung solcher Aussagen nur unzureichend möglich, wie der nächste Abschnitt zeigt.

3.1.2. Fehleraufdeckung

Wie bereits erwähnt wurde, ist die Eignung einer Testsuite, unbekannte Fehler aufzudecken, das wesentliche Qualitätsmaß hinsichtlich der Produktqualität. Dieses Kriterium

ist bei der Definition von Testfällen nicht unmittelbar überprüfbar und somit bei der Qualitätsbewertung von Testfällen und Testsuiten zunächst nicht hilfreich. Die Messung der Fehleraufdeckung eignet sich jedoch, um im Nachhinein die Güte der Testfälle zu bewerten, etwa als Beitrag, um den Testprozess zu verbessern. Weiter werden vielfach Aussagen über die Zuverlässigkeit von Testmethoden angegeben, indem die erreichte Fehleraufdeckung in Systemen mit bekannten Fehlern gemessen werden.

Bewertung der Testgüte nach Ende des Produktlebenszyklus

Am Ende des Lebenszyklus eines Systems können Testsuiten dahingehend bewertet werden, wie viele tatsächlich aufgetretene Fehler durch die Testfälle in der Testsuite nicht erkannt wurden – dies sind die Fehler, welche bei der Verwendung des Systems aufgetreten sind. Dies reicht alleine allerdings noch nicht aus, um auch die Qualität der Testmethode, mit welcher die Testsuite ermittelt wurde, zu beurteilen. Hierzu muss auch die Übertragbarkeit auf andere Systeme gezeigt werden. Schließlich ist nicht gewährleistet, dass sich die Fehler in anderen Systemen in vergleichbarer Weise verteilen.

Die genannte Aussage, dass es ausreicht, die aufgetretenen Fehler bis zum Ende des Produktlebenszyklus zur Bewertung der Tests heranzuziehen, impliziert, dass eine optimale Testsuite nicht notwendigerweise *alle* im System vorhandenen Defekte aufgedeckt haben muss. Fehler, welche während der gesamten Einsatzzeit eines Systems nie zur Ausführung gelangen, beeinflussen die nachträgliche Qualitätsbewertung einer Testsuite nicht. Diese Sichtweise begründet sich darin, dass sich die Produktqualität nur dann erhöht, wenn die erhöhte Produktqualität auch wahrgenommen wird. Ein Kostenbetrachtung legt diese Sichtweise ebenfalls nahe: Fehlsituationen, welche nie ausgeführt werden, ziehen keine Folgekosten nach sich. Die Bedeutung, welche das Nutzerverhalten auf die tatsächlich aufgetretenen Fehler hat, wurde von Adams (1984) sowie Fenton und Neil (1999) unterstrichen.

Fehleraufdeckung zur Bewertung von Testmethoden

Zur Bewertung von Testmethoden wird häufig ihre Eignung zur Aufdeckung von Fehlern analysiert. Die Fehleraufdeckung kann einerseits an exemplarischen, realen Systemen mit bekannten Fehlern gemessen werden, andererseits können auch Varianten von Programmen, so genannte *Mutanten*, generiert werden, welche künstlich mit Defekten angereichert wurden. Eine solche *Mutationsanalyse* beziehungsweise *mutantenbasiertes Testen* wurde bereits von Hamlet (1977) sowie DeMillo u. a. (1978) vorgeschlagen um Aussagen über die Testqualität zu präzisieren. Um Mutanten von Programmen zu erstellen, stehen mittlerweile Werkzeuge zur Verfügung, etwa *MuJava* (Ma u. a., 2006) für die Programmiersprache Java (Sun, 1996). Während die Analyse

von realen Systemen und realen Fehlern nur sehr exemplarisch ist und sich wiederum die Frage stellt, in weit sich diese Ergebnisse auf andere oder künftige Systeme beziehungsweise Entwicklungsprojekte übertragen lassen, erlaubt die Mutationsanalyse eine allgemeinere Aussage, da eine große Zahl unterschiedlicher Mutanten des Referenzsystems erzeugt werden. Für die Aussagekraft einer Mutationsanalyse ist es entscheidend, welche Fehler und an welcher Stelle diese Fehler in die erzeugten Mutanten injiziert werden. Diese Verteilung und Art der Fehler muss der zu erwartenden Fehlerverteilung im zu testenden System möglichst gut entsprechen, um eine verlässliche Aussage über die Testfallgüte hinsichtlich des zu testenden Systems zu erhalten. Hierzu werden oft Fehlerklassifikationen wie die *orthogonale Defektklassifikation* (orthogonal defect classification, ODC) (Chillarege u. a., 1992; Chillarege, 1996) verwendet. ODC klassifiziert Fehler dabei hinsichtlich ihres Typs nach *Zuweisungsfehlern* (assignment), *Überprüfungsfehlern* (checking), *Schnittstellenfehlern* (interface), *Zeit-/Sequentialisierungsfehler* (timing/serialization), *Berechnungsfehlern* (algorithm) und *Funktionalitätsfehlern* (function). Duraes und Madeira (2006) haben in einer Analyse von zwölf realen Softwareprodukten gezeigt, dass diese ODC Klassen jenen realer Fehlern entsprechen. Diese Studie zeigt aber auch, dass die einzelnen Fehlerklassen sich in verschiedenen Softwareprojekten sehr unterschiedlich verteilen:

- *Zuweisungsfehler* machten in zwei Programmen nur 10 % oder weniger aller Fehler aus, in anderen Projekten dagegen über 50 % aller Fehler
- *Überprüfungsfehler* kamen in zwei Programmen nicht vor, in den übrigen lag ihr Anteil an allen Fehlern zwischen 5 % und 52,5 %
- Der Anteil von *Schnittstellenfehlern* betrug zwischen 4,1 % und 54,5 %, sofern Schnittstellen überhaupt fehlerhaft waren. In vier Programmen traten keine Fehler dieses Typs auf.
- *Berechnungsfehler* waren in einem Fall 52,8 % aller Fehler, in einem anderen Fall nur 9,1 %. In zwei Projekten traten keine algorithmischen Fehler auf.
- Sieben Programm wiesen keine *funktionellen Fehler* auf, bei den übrigen machten diese Fehler bis zu 15,1 % aller Fehler aus.

Tabelle 3.1 fasst die klassifizierten Fehler dieser Studie nochmals zusammen, dabei werden nur jeweils die drei Projekte aufgeführt, welche einen besonders hohen beziehungsweise besonders niedrigen Anteil des jeweiligen Fehlertyps aufwiesen. Um eine statistisch höhere Verlässlichkeit zu erlangen, blieben Programme mit weniger als 20 Fehlern unberücksichtigt, in Tabelle 3.1 sind somit die Programme *GAIM*[2], *MingW*[3],

[2]GAIM: Multi-Protokoll-Client für Instant Messaging, `http://sourceforge.net/projects/gaim/`

[3]MingW: GNU Entwicklungswerkzeuge für Microsoft Windows, `http://www.mingw.org/`

Tabelle 3.1.: Verteilung von Fehlern nach ODC Typen in den in (Duraes und Madeira, 2006, Table 3) analysierten Programmen. Angeben sind jeweils die drei Programm mit dem geringsten und höchsten Fehleranteil je Fehlerklasse; Programme mit weniger als 20 Fehlern wurden nicht berücksichtigt. Zudem ist jeweils der Mittelwert der drei Programme in der Spalte ∅ angegeben.

ODC Typ	Programme mit **geringen** Anteil von Fehlern des Typs			∅	Programme mit **hohen** Anteil von Fehlern des Typs			∅
Zuweisung	GAIM 4,3 %	MingW 10,0 %	FCiV 11,3 %	8,5 %	Pdf2h 55,0 %	Joe 25,6 %	ScummVM 24,3 %	35,0 %
Überprüfung	Pdf2h 5,0 %	ScummVM 8,1 %	FCiv 13,2 %	8,8 %	GAIM 52,5 %	Joe 44,9 %	MingW 38,3 %	45,2 %
Schnittstelle	Pdf2h 0,0 %	ScummVM 4,1 %	GAIM 4,3 %	2,8 %	Joe 14,1 %	FCiv 7,5 %	Vim 6,4 %	9,3 %
Berechnung	Joe 15,4 %	GAIM 26,1 %	LKernel 33,3 %	24,9 %	ScummVM 56,8 %	FCiv 52,8 %	MingW 46,6 %	52,1 %
Funktionalität	GAIM 0,0 %	MingW 0,0 %	FCiV 0,0 %	0,0 %	Pdf2h 15,1 %	Joe 13,0 %	ScummVM 12,9 %	13,7 %

FCiv[4], *Pdf2h*[5], *Joe*[6], *ScummVM*[7], *Vim*[8] und *LKernel*[9] der Studie berücksichtigt. Auch in dieser Darstellung wird deutlich, dass die Verteilung der unterschiedlichen Fehlertypen in den verschiedenen Programmen stark schwankend ist. Besonders deutlich wird die Abweichung bei Zuweisungs- oder Überprüfungsfehlern.

Im Allgemeinen lässt sich also keine regelmäßige Verteilung von Fehlern feststellen; es erscheint derzeit daher kaum möglich, verlässlich Fehlerverteilungen in realen Projekten zu prognostizieren. Eine Fehlerprognose wird durch die vielfältige Zahl von Einflussfaktoren erschwert. Hierunter fallen nicht nur technische Aspekte, wie beispielsweise die verwendete Programmiersprache, die Anwendungsdomäne des Programms oder der Umfang und die Komplexität des Programms, sondern vor allem auch Aspekte der Projektorganisation, wie etwa die Größe des Entwicklungsteams, der Ausbildungsstand der Mitarbeiter, der Reifegrad der Anforderungen. Nagappan u. a. (2008) haben dazu bei einer Auswertung von Fehlern im Betriebssystem *Windows Vista*[10] festgestellt, dass die Organisationsstruktur von Entwicklungsteams in Teilprojekten in einem besonderen Maße die Qualität von Software beeinflusst.

Ein wesentliche Schwierigkeit bei der Bewertung der Testfallgüte hinsichtlich der Fehleraufdeckung liegt darin, dass nicht allein die Menge von Testfällen – oder die Me-

[4] FCiv: FreeCiv, Mehrspieler Strategiespiel, `http://www.freeciv.org/`

[5] Pdf2h: pdf2html, Übersetzer für PDF-Dokumente nach HTML, `http://sourceforge.net/projects/pdftohtml/`

[6] Joe: Texteditor, `http://sourceforge.net/projects/joe-editor/`

[7] ScummVM: Interpreter für grafische Adventure-Spiele, `http://sourceforge.net/projects/scummvm/`

[8] Vim: Erweiterung des UNIX-Texteditors vi, `http://www.vim.org/`

[9] LKernel: Kernel des Betriebssystems Linux (Version 2.0.39 und 2.2.22), `www.kernel.org`

[10] Microsoft Corporation, Redmond, USA. `http://www.microsoft.com/`

thode, nach der diese Testfälle ermittelt wurden – die Testfallgüte beeinflusst sondern diese auch immer von der Verteilung der Fehler im jeweiligen Programm abhängt. Da die Verteilung von Fehlern in verschiedenen Projekten stark unterschiedlich ist und da sich diese Verteilung nicht vorhersagen lässt, sind aussagekräftige Aussagen von den vorhin angesprochenen Mutationsanalysen oft kritisch zu betrachten. Um verlässliche Aussagen zu liefern, müssten die künstlich erzeugten Fehler in den Mutanten jener Verteilung von Fehlern folgen, die für die betrachtete Klasse von Programmen zu erwarten ist. Es ist nach Meinung des Verfassers nicht zu erwarten, dass derartige Vorhersagen jemals verlässlich möglich sind. In einer früheren Studie, in welcher die Vorhersagekraft von Metriken in Bezug auf aufgetretene Fehler im Feld untersucht wurde, kommen Nagappan u. a. (2006) zu einem vergleichbaren Schluss: Innerhalb einzelner Projekte ist die Vorhersagbarkeit für die Fehlerverteilung mit geeigneten Metriken gegeben, eine Verallgemeinerung über unterschiedliche Projekte war dagegen nicht erkennbar. Zu einer kritischen Einschätzung von Defektvorhersagen kommen auch Fenton und Neil (1999) in einer Auswertung von Veröffentlichungen zu Vorhersagemodellen.

3.2. Systematisierung von Qualitätskriterien

Im vorhergehenden Abschnitt wurde dargestellt, dass zwei sehr häufig genannte Qualitätskriterien von Tests und Testmethoden, strukturelle Überdeckung und Grad der Fehleraufdeckung, keine letztendlich befriedigende Lösung zur Bewertung der Qualität von Testfällen und insbesondere Testmethoden darstellen. Zum einen liegt dies an der mangelnden Aussagekraft, zum anderen an der kaum möglichen Operationalisierung während der Testerstellung. Wie schon von Pretschner (2003, S. 92) angegeben, muss es als unwahrscheinlich angesehen werden, dass jemals allgemein gültige, operationalisierbare Kriterien zur Testbewertung gefunden werden. Sinnvoller erscheint es daher, unterschiedliche Qualitätskriterien für Testfälle stärker zu systematisieren.

Vom Verfasser wurde dazu bereits vorgeschlagen, eine Klassifizierung von Qualitätsbewertung für Testfälle vorzunehmen (Pfaller u. a., 2008). Testgüte wird meist nicht allein anhand der Testfälle bewertet sondern ergibt sich aus der Relation zu weiteren Entwicklungsdokumenten. Anschaulich wird dies zum Beispiel bei den bereits erwähnten strukturellen Überdeckungskriterien von Programmcode: Die Testgüte bemisst sich aus dem Verhältnis von im Test ausgeführten Programmcode zum gesamten Programmcode. Auch die bereits genannte Fehleraufdeckungsrate lässt sich nicht allein an den Testfällen messen, hierzu sind ebenfalls Fehlerprotokolle bekannter Fehler notwendig.

Natürlich können in beschränktem Maße auch Qualitätsaussagen allein an der Menge von Testfällen postuliert werden, vor allem wenn die Effizienz der Testausführung betrachtet werden soll, kürzere Testfälle sind dann etwa längeren vorzuziehen. Aber nicht nur die Effizienz der Ausführung, sondern auch Hinweise auf die Sicherstellung

der Produktqualität lassen sich allein aus Testfällen ablesen, indem beispielsweise die Verteilung der Eingabewerte bewertet wird. Derart einfache Maße weisen nur eine sehr begrenzte Aussagekraft auf.

3.2.1. Dokumente in Relation zu Testfällen

Um die Bewertung von Testfällen differenzierter vorzunehmen, kommen unterschiedliche Dokumente in Betracht, in deren Relation die Güte von Testfällen bewertet wird. Teilweise stehen diese Dokumente erst in späteren Entwicklungsphasen oder nach Abschluss der Entwicklung zur Verfügung, in diesem Fall ist die Verwendung zur Auswahl von Testfällen kaum möglich, die Bewertung der Testfallgüte kann aber zu Prozessverbesserungen dienen. Mögliche Dokumente sind etwa:

Anforderungsspezifikation Dokumente, welche einzelne Anforderungen an das System auflisten, sind häufig Grundlage für die Bewertung von Testfällen. Dabei wird untersucht, ob jede Anforderung geeignet durch Tests abgedeckt ist. Da meist nicht weiter definiert wird, unter welchen Umständen eine Anforderung „geeignet“ durch Testfälle abgedeckt ist, werden meist triviale Kriterien verwendet, etwa dass jeder definierten Einzelanforderung mindestens ein Testfall zuzuordnen sein muss.

Programmcode Quellcode des Programms wird häufig zur Bewertung der Testgüte herangezogen. Dies führt zu den bereits erwähnten strukturellen Überdeckungskriterien, siehe hierzu Abschnitt 3.1.1.

Testergebnisse Ergebnisse der Testausführung können verwendet werden, um die Zahl der erkannten Fehler zu quantifizieren. Dies eignet sich etwa, um zwei Testmethoden zu vergleichen, für die Auswahl von Testfällen sind diese Bewertungen kaum hilfreich. Allerdings können sie dazu verwendet werden, Testendekriterien festzulegen, etwa dass weitere Tests notwendig sind, falls weiterhin eine hohe Zahl von Fehlern entdeckt wird.

Nutzungsprofile Analysen über das zu erwartende Verhalten der Anwender des Systems dienen dazu, besonders häufig ausgeführte Funktionen zu identifizieren. In den häufig ausgeführten Funktionen ist die Wahrscheinlichkeit, dass vorhandene Fehler entdeckt werden, besonders hoch, so dass gute Tests in Relation zu Nutzungsprofilen diese Funktionen besonders ausführlich berücksichtigen.

Risikoanalysen Methoden der Fehlermöglichkeits- und Fehlereinflussanalyse (*failure mode and effects analysis*, FMEA) und Fehlerbaumanalyse (*fault tree analysis*, FTA) (Pister, 2009) dienen dazu, besonderes kritische Funktionsteile zu identifizieren. Treten hier Fehler auf, sind die Auswirkungen besonders schwerwiegend. Tests können dahingehend bewertet werden, ob besonders riskante Abläufe in Testfällen berücksichtigt sind.

Fehlerberichte, welche Fehler angeben, die nach Ende des Tests, etwa im produktiven Betrieb oder in späteren Testphasen, aufgetreten sind, erlauben es, Tests dahingehend zu bewerten, welche Fehler nicht durch die Tests gefunden wurden.

Diese Auflistung erfolgt ohne Anspruch auf Vollständigkeit. So können zum Beispiel auch noch Architekturdokumente verwendetet werden, um Testfälle hinsichtlich der betroffenen Komponenten zu beurteilen.

In der obigen Auflistung wurde kurz skizziert, wie die Testgüte in Relation zu einzelnen Dokumentarten bestimmt werden kann. Interessant ist daneben auch die Kombination von mehreren Dokumentarten zur Bewertung der Testgüte. Dies ist insbesondere erforderlich, wenn einzelne Dokumente zu widersprüchlichen Bewertungen führen können, wie dies etwa bei Nutzungsprofilen und Risikoanalysen der Fall ist. Häufig sind gerade selten genutzte Funktionen mit schwerwiegenden Folgen im Fehlerfall behaftet. Im folgenden Abschnitt wird darum eine Klassifikation von Gütekriterien angegeben, welche die Kombination von Dokumenten berücksichtigt.

3.2.2. Klassifizierung von Maßen der Testgüte

Vom Verfasser wurde bereits vorgeschlagen, zur Klassifizierung von Qualitätsaussagen zu Testfällen jeweils die betrachteten Dokumente aus dem Entwicklungsprozess zu berücksichtigen, hinsichtlich derer die Beurteilung erfolgt (Pfaller u. a., 2008). Im vorigen Abschnitt wurden sechs unterschiedliche Arten von Dokumenten angegeben, anhand welcher sich die Güte von Testfällen bewerten lässt. Teilweise können dabei Qualitätsbewertungen zu sehr unterschiedlichen Aussagen kommen, oder sogar gegensätzlich sein. Als Beispiel sei hier ein System angenommen, in welchem die üblichen Abläufe, welche Nutzer häufig ausführen, wenig kritische Auswirkungen im Fehlerfall verursachen. Folgenschwere Fehlsituationen sind dagegen vor allem Ausnahmefälle der Normalabläufe. In einem solchen System würde etwa eine Bewertung der Testgüte hinsichtlich von Nutzungsprofilen zu einem anderen Schluss kommen als die Bewertung hinsichtlich Risikoanalysen. Aus diesem Grund ist es auch sinnvoll, die Testgüte in Relation zu Kombinationen von unterschiedlichen Dokumenten zu bewerten. Mit den sechs Dokumentarten ergeben sich insgesamt $2^6 = 64$ mögliche unterschiedliche Klassen von Gütemaßen. In Tabelle 3.2 sind davon zwölf Beispielklassen angegeben, welche

Tabelle 3.2.: Klassifikation von Maßen zur Testgütebewertung gegenüber unterschiedlichen Dokumenten (Auswahl).

Klasse		1	2	3	4	5	6	7	8	9	10	11	12
Testfälle		X	X	X	X	X	X	X	X	X	X	X	X
vs.	Anforderungsspezifikation		X		X						X		
	Programmcode			X	X							X	
	Testergebnisse					X						X	X
	Nutzungsprofile						X			X	X		
	Risikoanalysen, FMEA							X		X			
	Fehlerberichte								X				X

Tabelle 3.3.: Beschreibung der Klassen von Gütemaßen zur Testbewertung (aus Tabelle 3.2).

Klasse	**Beschreibung**
1	Bewertung von Testfällen ohne Einbeziehung weiter Dokumente
2	Überdeckung von Anforderungen durch Testfälle
3	Überdeckung von Programmcode durch Testfälle
4	Überdeckung von Anforderungen und Programmcode
5	Bewertung der Testergebnisse, z. B. identifizierte Defekte
6	Entsprechung der Tests bezüglich des erwarteten Nutzerverhaltens
7	Berücksichtigung folgenschwerer Abläufe
8	Bewertung hinsichtlich nicht entdeckter Fehler
9	Kritische und häufig ausgeführte Funktionen, Gesamtfolgekosten von Fehlern
10	Überdeckung von gewichteten Anforderungen hinsichtlich Nutzungsprofil
11	Überdeckung von Programmcode und Fehlererkennung
12	Verhältnis entdeckte Defekte zu nicht entdeckten Defekten

jeweils eine sinnvoll erscheinende Kombination von maximal zwei Dokumentarten sowie den Testfällen sind. Tabelle 3.3 beschreibt die angegebenen Klassen kurz.

Eine derartige Klassifikation ist notwendig, sollen Aussagen über die Güte von Testfällen oder die Güte von Methoden, welche Testfälle erzeugen, miteinander verglichen werden. Es ist offensichtlich, dass Qualitätsaussagen aus unterschiedlichen Klassen nur schwer miteinander zu vergleichen sind. Deutlich wird dies am Beispiel der Überdeckung von Programmcode: wie bereits in Abschnitt 3.1.1 erläutert, ist der Zusammenhang zwischen struktureller Überdeckung und einer Erhöhung der Produktqualität umstritten. Zu Relationen zwischen weiteren Güteklassen untereinander fehlen Untersuchungen, ob hier gegebenenfalls Abhängigkeiten bestehen kann gegenwärtig nur vermutet werden.

Die Klassifizierung der Gütemaße anhand der notwendigen Dokumente zeigt zudem auf, ob ein gewisses Gütemaß überhaupt anwendbar ist, schließlich müssen die entsprechenden Dokumente vorhanden sein. So wird deutlich, dass das meist gewünschte Gütemaß, die Fähigkeit von Testfällen, Fehler zu entdecken, nur bewertet werden

kann, falls Fehlerberichte vorliegen. Berichte über tatsächlich im produktiven Betrieb aufgetretene Fehler stehen während der Entwicklung im Allgemeinen nicht zur Verfügung. Somit ist dies ein ungeeignetes Qualitätsmaß wenn die Testfallgüte während der Entwicklung, etwa zum Zeitpunkt der Testfallerstellung, dementsprechend bewertet werden soll.

3.3. Qualitätsbeurteilung anhand Überdeckung von Anforderungen

Im vorhergegangenen Abschnitt wurde dargestellt, dass die Qualitätsbewertung des Softwaretest nach sehr unterschiedlichen Kriterien erfolgen kann. Ob eine entsprechende Qualitätsbeurteilung möglich ist, hängt im Wesentlichen von der Verfügbarkeit der entsprechenden produktspezifischen Dokumente ab, gegen welche eine Qualitätsaussage getroffen werden soll. Da heute über Zusammenhänge zwischen den einzelnen Klassen zur Qualitätsbeurteilung (vergleiche Tabelle 3.2, Seite 37) im Allgemeinen nur Vermutungen angestellt werden können, müssen zum Zeitpunkt der Testerstellung Gütemaße Verwendung finden, für welche die notwendigen Dokumente zu diesem Zeitpunkt zur Verfügung stehen.

Da in dieser Arbeit der funktionale Test von reaktiven System betrachtet wird, ist die Frage nach der geeigneten Wiedergabe von funktionalen Anforderungen durch die Testfälle ein sinnvolles Kriterium zur Bewertung der Testgüte. Die entsprechenden Dokumente – also die Anforderungsspezifikation – stehen im Regelfall zum Zeitpunkt der Testerstellung zur Verfügung. Somit ist ein Gütemaß notwendig, welches die Eignung einer Menge von Testfällen zur Überprüfung von explizit spezifizierten Anforderungen bewertet. Obwohl Verträge, Normen und Standards häufig für ein Softwareprojekt fordern, dass alle spezifizierten Anforderungen geeignet durch Testfälle abgedeckt sein müssen, fehlt eine präzise Definition, welche Kriterien erfüllt sein müssen, damit eine derartige Überdeckung vorliegt. In der Literatur wird ein anforderungsbasierter Test beziehungsweise eine Anforderungsüberdeckung dann als erfüllt angesehen, wenn ein Testfall aus der Anforderung *herleitbar* ist oder ein Testfall einer Anforderung *zugeordnet* werden kann: Spillner und Linz (2007, S. 70) fordern hierzu etwa, dass *zu jeder Anforderung mindestens ein Testfall abgeleitet* wird. Sneed u. a. (2009, S. 178) beschreiben die Anforderungsüberdeckung als Quotient aus der Anzahl getesteter Anforderungen und der Anzahl aller Anforderungen. Eine Anforderung wird dann als getestet angesehen, wenn *mindestens ein Testfall bezogen auf diese Anforderung erstellt und ausgeführt* wurde. Fournier (2009, S. 190,191) setzt den Begriff der Anforderungsüberdeckung mit dem der Rückverfolgbarkeit *(traceability)* von Testfällen zu Anforderungen gleich. Maße und Metriken, um die Abdeckung von Anforderungen durch Testfälle zu quantifizieren, sind dagegen kaum verbreitet. Dies führt in der

Praxis dazu, dass meist nur triviale Maße – *„zu jeder Anforderung muss mindestens auf einen Testfall verwiesen werden“* – zur Anwendung kommen.

Auch existieren nur wenige Vorschläge für eine präziseren Bewertung der Anforderungsüberdeckung. Hammer u. a. (1997) beschreiben dazu verschiedene Maße, welche auf der Zuordnung von Anforderungen zu Testfällen beruhen: Die Anzahl der Anforderungen, welche in Testfällen berücksichtigt sind; die Anzahl der Anforderungen, die in einzelnen Testfällen überprüft werden; die Anzahl der Testfälle, die einzelnen Anforderungen zugeordnet sind; die Gesamtzahl der Zuordnungen zwischen Testfällen und Anforderungen. Allerdings bleibt hierbei die Qualität der Zuordnung von Testfällen zu Anforderungen unberücksichtigt. Wie *passend* ein einzelner Testfall zu einer Anforderung ist, wird nicht bewertet.

Whalen u. a. (2006) sowie Rajan (2006) definieren Überdeckungsmetriken auf Basis von Anforderungen welche in Form temporaler Logik, genauer *Linear Time Temporal Logic (LTL)* (Clarke Jr. u. a., 1999), spezifiziert sind. Einen ähnlichen Ansatz verfolgen Pecheur u. a. (2009) zur Generierung von Testfällen aus LTL-Formeln. Da nur in vergleichsweise wenigen Entwicklungsprojekten Anforderungen in Form von LTL-Formeln spezifiziert werden, ist die Anwendung dieser Metriken begrenzt. Ein weiteres Problem besteht darin, dass die Validierung der Anforderungen, wenn sie als LTL-Formeln gegeben sind, nur schwer durch die beteiligten Stakeholder, wie beispielsweise Auftraggeber oder Anwender, möglich ist.

Da bisher in Praxis und Literatur kaum geeignete anforderungsorientierte Überdeckungsmaße Verwendung finden, sollen zunächst Eigenschaften analysiert werden, welche ein geeignetes Überdeckungsmaß hierzu erfüllen soll. Für die später in Kapitel 5 eingeführte Beschreibungstechnik für Anforderungen wird in Kapitel 7 eine Metrik eingeführt, welche auf diesen Eigenschaften basiert.

3.3.1. Aussagekraft einer Anforderungsüberdeckung und Fehlerhypothese

Das Ziel der Bewertung einer Anforderungsüberdeckung soll es sein, Aussagen zu erhalten, ob eine Menge von Testfällen eine Menge von Anforderungen „besser“ überdeckt als eine andere Menge von Testfällen. Das Interesse soll sich hier also auf ein relatives Maß beschränkten. Eine Bewertung der Anforderungsüberdeckung soll also eine Antwort auf Fragen folgender Art liefern:

> *„Ist Testfallmenge A oder Testfallmenge B besser geeignet, die Erfüllung der Anforderungen in R zu überprüfen?“*

$R = \{\rho_1, \rho_2, \ldots \rho_n\}$ bezeichnet dabei eine Menge von einzelnen Anforderungen.

Ob und in wie weit derartige Aussagen sinnvoll sind, hängt in starkem Maß davon ab, in welcher Form die Anforderungen vorliegen und welcher Art die Anforderungen

sind. Die Messung der Anforderungsüberdeckung kann sich letztlich nur auf die in Spezifikationsdokumenten formulierten Anforderungen erstrecken. Zu bedenken ist dabei, dass die Intention der Stakeholder häufig weitergehend oder sogar teilweise abweichend von den dokumentierten Anforderungen sein kann.

Die Überdeckung der Anforderungen stellt hier ein Gütemaß nach Klasse 2 aus Tabelle 3.3 dar. Bewertet werden soll allein die Güte der Testfälle in Relation zu Anforderungsdokumenten, weitere Dokumente fließen nicht in die Bewertung mit ein. Insbesondere liegt keine dedizierte Fehlerhypothese vor, welche fehlerträchtige Teile des Testobjekts identifiziert. Zur Bewertung der Anforderungsüberdeckung wird somit angenommen:

Alle Programmteile gleichwertig Alle Teile und Teilfunktionen des Programms werden als gleichwertig für den Test betrachtet. Es wird nicht zwischen besonders fehlerträchtigen oder besonders kritischen Programmteilen differenziert. Es existieren keine Kritikalitäts- oder Fehlerauswirkungsanalysen. Auch liegen keine expliziten Nutzungsprofile vor, welche die erwartete Häufigkeit der Nutzung von Teilfunktionen angeben. Die Fehlerverteilung ist unbekannt.

Keine Annahmen über Fehlerwahrscheinlichkeit in Eingaben Es existieren keine Annahmen über die Fehlerwahrscheinlichkeit von Eingabesequenzen oder zu abhängigen Wahrscheinlichkeiten. Jede Eingabesequenz ist potentiell gleichermaßen dazu geeignet, Fehler aufzudecken.

Keine Informationen über die Realisierung Es liegen keine Informationen über die Realisierung vor, die Betrachtung der Testgüte erfolgt aus einer Black-Box-Sicht.

Diese Einschränkungen tragen dem Umstand Rechnung, dass explizit dokumentierte Informationen zu Fehlerannahmen, Nutzungsszenarien die Realisierung oder ähnlichen häufig nicht vorliegen, welche für genaue Bewertung notwendig werden.

Für nicht-funktionale Anforderungen wie die Wartbarkeit des Systems ist der Test im Allgemeinen kein geeignetes Mittel, um die Erfüllung der Anforderung zu überprüfen. Auch für weitere Qualitätsanforderungen von Software, wie sie beispielsweise in der Norm ISO 9126 (2001) definiert sind, ist (mit Ausnahme der Anforderungen an die Funktionalität) der funktionale Test ein wenig geeignetes Mittel, derartige nichtfunktionale Anforderungen zu überprüfen.

Die in dieser Arbeit eingeführte Methode zur Ermittlung von anforderungsorientierten Testfällen basiert auf der Repräsentation von Anforderungen durch Szenarien. Dementsprechend erfolgt die Gütebewertung der ermittelten Testfälle hinsichtlich der gegebenen Szenarien. Bevor ab Abschnitt 3.3.4 auf die Überdeckung von Szenarien durch Testfälle eingegangen wird, werden erst allgemeine Überlegungen zum Begriff der Anforderungsüberdeckung dargestellt.

3.3.2. Prüfung von Anforderungen durch Testfälle

Wie zuvor erläutert wurde, existiert bisher keine allgemein akzeptierte präzise Definition unter welchen Bedingungen eine Überdeckung einer Anforderung durch einen Testfall gegeben ist. Ein präziserer, allerdings auch strikter, Begriff zur Relation zwischen Testfällen und Anforderungen wird erreicht, wenn betrachtet wird, ob ein Testfall zur *Prüfung* einer Anforderung geeignet ist. Die Prüfung ist dabei so zu verstehen, dass mithilfe des Testfalls ein etwaiges, von der Anforderung abweichendes, Fehlverhalten des Systems hinsichtlich dieser Anforderung festgestellt werden kann.

In den folgenden Abschnitten wird dieser Begriff der Prüfung einer Anforderung durch einen Testfall präzise definiert. Dabei wird das System als eine Funktion repräsentiert welche einen Folge von Eingaben verarbeitet und eine Folge von Ausgaben produziert. Das zugrunde legende Systemmodell folgt somit jenem der stromverarbeitenden Funktionen von Broy u. a. (1992); Broy und Stølen (2001). Aus letzterer Publikation wurde die im Folgenden verwendete Notation übernommen; eine Zusammenfassung der verwendeten Symbole und Operatoren findet sich in Anhang H.

Anforderungen an ein System

Ein- und Ausgaben reaktiver Systeme, welche fortwährend mit ihrer Umwelt interagieren, können als Sequenzen (Ströme) von Aktionen dargestellt werden. Durch die Angabe der Menge I von Eingabeaktionen und der Menge O von Ausgabeaktionen ist die *syntaktische Schnittstelle* $(I \triangleright O)$ festgelegt:

Definition 1 (Syntaktische Schnittstelle) *Die Syntaktische Schnittstelle $(I \triangleright O)$ bezeichnet die Menge der möglichen Eingabeaktionen I und die Menge der möglichen Ausgabeaktionen O eines Systems.*

Über die syntaktische Schnittstelle lässt sich die Menge aller möglichen Funktionen hinsichtlich dieser Schnittstelle bestimmen.

Definition 2 (Menge der Funktionen) *Die Menge der monotonen Funktionen über eine syntaktische Schnittstelle $(I \triangleright O)$ wird im Folgenden durch die Menge*

$$FUN_{(I \triangleright O)} \stackrel{\text{def}}{=} I^{\omega} \to O^{\omega}$$

angegeben.

$M^\omega = M^* \cup M^\infty$ bezeichnet dabei die Menge der endlichen (M^*) und unendlichen (M^∞) Sequenzen von Elementen einer Menge M.

Das damit definierte Systemmodell umfasst zwar keine expliziten Konstrukte zur Formulierung zeitlicher Informationen. Dennoch können auch mit diesem Systemmodell zeitliche Anforderungen, welche für reaktive Systeme oft von großer Bedeutung sind, ausgedrückt werden. Zum einen kann mit der Ein-/Ausgabesequenz ein konstanter Verarbeitungstakt interpretiert werden. Die Verarbeitung einer Aktion in der Eingabe geschieht dann in jeweils einer (logischen) Zeiteinheit. Zeitliche Anforderungen lassen sich dann über die Anzahl der Takte formulieren. Zum anderen kann ein Uhr auch über die Schnittstelle des Systems zeitliche Informationen liefern, welche in die Verarbeitung einfließen.

Im Rahmen der vorliegenden Arbeit werden weiter nur deterministische Systeme betrachtet, so dass zu jeder Folge von Eingabeaktionen I^ω eindeutig eine Folge O^ω von Ausgabeaktionen bestimmt werden kann. Durch eine Menge $R = \{\rho_1, \rho_2, \ldots, \rho_n\}$ von Anforderungen soll eine echte Teilmenge

$$FUN_R \subsetneq FUN_{(I \triangleright O)}$$

von Funktionen spezifiziert werden, welche diesen Anforderungen genügen. Eine Anforderung ist dabei wie folgt als Prädikat definiert:

Definition 3 (Anforderung) *Eine Anforderung ρ ist ein Prädikat über die Menge der Funktionen:*

$$\rho \in FUN_{(I \triangleright O)} \to \mathbb{B}\mathrm{ool}$$

Beispiel 1 (Einelementiger Puffer – Anforderungen)

Als einfaches Beispiel sollen Anforderungen für eine Funktion

$$\mathsf{buffer} \in I^\omega \to O^\omega$$

angegeben werden, wobei die syntaktische Schnittstelle durch

$$I \stackrel{\text{def}}{=} P \times G \text{ mit } P = \{\bot\} \cup \{\mathsf{put}(x) | x \in \mathbb{N}\},\ G = \{\bot, \mathsf{get}\}$$
$$O \stackrel{\text{def}}{=} \{\bot\} \cup \mathbb{N}$$

gegeben sei. Die Eingabe besteht demnach aus Tupeln (p, g) wobei p entweder eine die leere Nachricht $\bot$ oder ein *Put*-Kommando $\mathsf{put}(x)$ mit $x \in \mathbb{N}$ ist und g eine leere Nachricht $\bot$ oder das *Get*-Kommando get ist. Durch ein *Put*-Kommando soll ein Wert x in den Puffer geschrieben werden, mit einem *Get*-Kommando soll der Wert des Puffers ausgegeben werden. Die Ausgabe ist einweder ein Wert oder die leere Nachricht.

Der Puffer soll die folgenden informell angegebenen Anforderungen erfüllen:

1. Eine leere Eingabe hat eine leere Ausgabe zur Folge.
2. Ist das Get-Kommando nicht gesetzt, wird die leere Nachricht ausgegeben.
3. Wurde noch kein Put-Kommando ausgeführt, liefert ein Get-Kommando die leere Nachricht.
4. Wurde mit dem Put-Kommando ein Wert gesetzt, liefert ein Get-Kommando den zuletzt gesetzten Wert.
5. Wurde mit einem Get-Kommando ein Wert gelesen und kein neuer Wert mit einem Put-Kommando gesetzt, liefert ein erneuter Aufruf des Get-Kommandos die leere Nachricht.

Diese Anforderungen seien formal durch Prädikate $\rho_i \in (I^\omega \to O^\omega) \to \mathbb{B}\text{ool}$ gegeben, $ins \in I^\omega$ bezeichnet jeweils die Eingabesequenz:

$$
\begin{aligned}
\rho_1(\text{buffer}) &\overset{\text{def}}{\Leftrightarrow} ins = \langle\rangle \Rightarrow \text{buffer}(ins) = \langle\rangle \\
\rho_2(\text{buffer}) &\overset{\text{def}}{\Leftrightarrow} \forall s \in I^*, p \in P : \\
&\quad ins = s \frown \langle(p, \bot)\rangle \Rightarrow \text{buffer}(ins) = \text{buffer}(s) \frown \langle\bot\rangle \\
\rho_3(\text{buffer}) &\overset{\text{def}}{\Leftrightarrow} \forall n \in \mathbb{N}, j \in [1 \dots n], g_j \in G, p \in P : \\
&\quad ins = \langle(\bot, g_j)\rangle^n \frown \langle(p, \text{get})\rangle \Rightarrow \text{buffer}(ins) = \langle\bot\rangle^{n+1} \\
\rho_4(\text{buffer}) &\overset{\text{def}}{\Leftrightarrow} \forall s \in I^*, x \in \mathbb{N}, n \in \mathbb{N}, g \in G, p \in P : \\
&\quad ins = s \frown \langle(\text{put}(x), g)\rangle \frown \langle(\bot, \bot)\rangle^n \frown \langle(p, \text{get})\rangle \\
&\quad \Rightarrow \text{buffer}(ins) = \text{buffer}(s \frown \langle(\text{put}(x), g)\rangle) \frown \langle\bot\rangle^n \frown \langle x\rangle \\
\rho_5(\text{buffer}) &\overset{\text{def}}{\Leftrightarrow} \forall s \in I^*, n \in \mathbb{N}, j \in [1 \dots n], g_j \in G, p \in P : \\
&\quad ins = s \frown \langle(\bot, \text{get})\rangle \frown \langle(\bot, g)\rangle^n \frown \langle(p, \text{get})\rangle \\
&\quad \Rightarrow \text{buffer}(ins) = \text{buffer}(s \frown \langle(\bot, \text{get})\rangle) \frown \langle\bot\rangle^{n+1}
\end{aligned}
$$

In Beispiel 1 sind Einzelanforderungen an eine Puffer-Funktion als Prädikate angegeben. Hier wird mit der Menge $\{\rho_1, \rho_2, \rho_3, \rho_4, \rho_5\}$ eine vollständige und deterministische Spezifikation des Verhaltens von buffer angeben. Dabei wurden die Prädikate der Anforderungen jeweils als Implikation $\Rightarrow$ definiert. Demzufolge definiert jede Anforderung die gewünschte Funktion nur partiell. Nur falls die Forderungen an die Eingabe erfüllt sind, ergeben sich auch Vorgaben an die erwartete Ausgabe, ansonsten ist das Prädikat in jedem Fall erfüllt.

Die Menge der Funktionen, welche eine Anforderung ρ erfüllen, ist somit durch die Menge

$$FUN_\rho = \{f \in FUN_{(I \triangleright O)} \mid \rho(f)\}.$$

definiert.

Für eine vollständige und widerspruchsfreie Menge von Einzelanforderungen $R = \{\rho_1 \dots \rho_n\}$ gibt die Konjunktion der Einzelanforderungen die Gesamtanforderung an

die Funktion FUN_R an:

$$FUN_R \stackrel{\text{def}}{=} \left\{ f \in FUN_{(I \triangleright O)} \;\middle|\; \bigwedge_{\rho \in R} \rho(f) \right\}$$

Testfälle zu einer Anforderung

Testfälle werden in dieser Arbeit als ein Paar von Eingabe und erwarteter Ausgabe betrachtet. Ein- sowie Ausgabe sind dabei Sequenzen von Aktionen entsprechend der gewählten syntaktischen Schnittstelle $(I \triangleright O)$ und somit hinsichtlich dieser Schnittstelle voll instanziierte Abläufe.

Häufig werden Testfälle auch nur partiell spezifiziert, wenn nur Teile der Eingabe beziehungsweise Ausgabe für die Beobachtung des gewünschten Verhaltens von Bedeutung sind. Dies ist vor allem für die effiziente Ausführung von Testfällen von großer Bedeutung, falls mit dem Test gezielt bestimmte Hypothesen überprüft werden sollen. Durch eine entsprechende partielle Spezifikation von Testfällen können hinsichtlich dieser Hypothese redundante Abläufe identifiziert werden. Eine Möglichkeit, diesem Problem der Redundanz bei vollständig angegeben Testabläufen zu begegnen, ist die automatische Reduzierung von Testsuites, wie sie etwa in Struss (2007) vorgeschlagen wird.

Da im Rahmen dieser Arbeit jedoch keine dedizierten Fehlerhypothese angenommen werden (alle Programmteile sind gleichwertig für den Test) und auch die interne Struktur des Testobjekts unbekannt ist (siehe Abschnitt 3.3.1), ist potentiell jede Eingabesequenz gleichermaßen dazu geeignet, Fehler aufzudecken. Daher wird ein Testfall wie folgt definiert:

Definition 4 (Testfall) *Ein Testfall $t_f = (ins_{t_f}, outs_{t_f}) \in I^* \times O^*$ für eine Funktion $f \in FUN_R$ ist ein Paar einer Folge ins_{t_f} von Eingaben und einer dazu erwarteten Folge $outs_{t_f}$ von Ausgaben, wobei $f(ins_{t_f}) = outs_{t_f}$ gültig ist.*

Es ist zu beachten, dass an Testfälle die Forderung ihrer *Gültigkeit* gestellt wird. Die im Testfall angegeben Ausgabe muss also die zu erwartende Reaktion des Systems auf die Testeingabe angeben.[11]

Im anforderungsorientierten Test ist für einen Testfall zu entscheiden, ob dieser geeignet ist, eine (exemplarische) Überprüfung hinsichtlich der Erfüllung einer Anforderung vorzunehmen. Es ist dazu ausreichend, die Eingaben des Testfalls zu betrachten, da

[11] *Ungültige Testfälle* sind hier von Testfällen über „ungültigen“ Eingaben (Eingaben, welche zwar syntaktisch zulässig sind, aber nicht erwartet werden) zu unterscheiden. Für deterministische Systeme ist auch für derartige fehlerhaften Eingaben die erwartete Systemreaktion definiert. Siehe hierzu auch Abschnitt 3.3.2 zu nichtdeterministischen Anforderungen.

einerseits die Gültigkeit des Testfalls angenommen wird (die Ausgaben des Testfalls somit auch einer Anforderung ρ unter der Testeingabe genügen müssen). Andererseits wurde die Widerspruchsfreiheit der Anforderungen vorausgesetzt, so dass zu einer Eingabe keine konträren Ausgaben spezifiziert sind.

Für eine Eingabesequenz $ins \in I^*$ können die gültigen und ungültigen Ausgaben bezüglich einer Anforderung ρ ermittelt werden. Die Menge aller syntaktisch möglichen Ausgaben O^ω lässt sich in die Menge der gültigen und ungültigen Ausgaben partitionieren:

Definition 5 (Gültige Ausgaben) *Für eine Anforderung $\rho \in FUN_{(I \triangleright O)} \to \mathbb{B}\text{ool}$ und eine Eingabe $ins \in I^*$ gibt die Menge*

$$O^\omega_{\rho,ins} \stackrel{\text{def}}{=} \left\{ outs \in O^\omega \;\middle|\; \exists f \in FUN_{(I \triangleright O)} : f(ins) = outs \wedge \rho(f) \right\}$$

die Menge der gültigen Ausgaben zu ins bezüglich ρ an.

Entsprechend wird mit

$$\begin{aligned} \overline{O^\omega_{\rho,ins}} &\stackrel{\text{def}}{=} \left\{ outs \in O^\omega \;\middle|\; \nexists f \in FUN_{(I \triangleright O)} : f(ins) = outs \wedge \rho(f) \right\} \\ &= O^\omega \setminus O^\omega_{\rho,ins} \end{aligned}$$

die Menge der ungültigen Ausgaben angeben.

Die Menge $O^\omega_{\rho,ins}$ gibt also an, welche Ausgaben für eine bestimmte Eingabe nach Maßgabe der Anforderung ρ gültig sind. Eine Anforderung beschreibt im Allgemeinen nur für eine Teilmenge I^ω_ρ aller möglichen Eingaben Vorgaben an die Ausgabe. Es lässt sich somit die Menge der Eingaben festlegen, welche von einer Anforderung betroffen sind:

Definition 6 (Eingaben einer Anforderung) *Für eine Anforderung $\rho \in FUN_{(I \triangleright O)} \to \mathbb{B}\text{ool}$ gibt die Menge*

$$\begin{aligned} I^\omega_\rho &\stackrel{\text{def}}{=} \left\{ ins \in I^\omega \;\middle|\; O^\omega_{\rho,ins} \subsetneq O^\omega \right\} \\ &= \left\{ ins \in I^\omega \;\middle|\; \exists f \in FUN_{(I \triangleright O)}, outs \in O^\omega : f(ins) = outs \wedge \neg\rho(f) \right\} \end{aligned}$$

die Menge der von der Anforderung betroffenen Eingaben an.

Diese Definition schließt triviale beziehungsweise nicht definierte Eingaben aus. Triviale Eingaben sind hier solche, für die das Prädikat der Anforderung mit jeder beliebigen Ausgabe erfüllt ist, also $O^\omega_{\rho,ins} = O^\omega$ gilt. Damit kann angegeben werden, unter welcher Voraussetzung ein Testfall zur Prüfung einer Anforderung geeignet ist:

Definition 7 (Prüfung einer Anforderung durch einen Testfall) *Ein Testfall t* prüft *eine Anforderung ρ, falls die Menge der gültigen Ausgaben zur Testeingabe ins_t bezüglich der Anforderung ρ eine echte Teilmenge aller syntaktisch möglichen Ausgaben ist. Formal wird dies durch das Prädikat*

$$\text{checks} \in (FUN_{(I \triangleright O)} \to \mathbb{B}\text{ool}) \times (I^* \times O^*) \to \mathbb{B}\text{ool}$$

beschrieben, welches für eine Anforderung $\rho \in FUN_{(I \triangleright O)} \to \mathbb{B}\text{ool}$ und einen Testfall $t = (ins_t, outs_t) \in I^* \times O^*$ definiert ist als

$$\begin{aligned} \text{checks}(\rho, t) &\overset{\text{def}}{\Leftrightarrow} outs_t \in O^\omega_{\rho, ins_t} \wedge O^\omega_{\rho, ins_t} \subsetneq O^\omega \\ &\Leftrightarrow outs_t \in O^\omega_{\rho, ins_t} \wedge ins_t \in I^\omega_\rho. \end{aligned}$$

Jeder Testfall, welcher das Prädikat checks hinsichtlich einer Anforderung ρ erfüllt, eignet sich zur Prüfung einer Anforderung. Dies findet seine Berechtigung dadurch, dass hinsichtlich der Eingabe *inst* sowohl ein Erfüllen *(pass)* als auch das Nicht-Erfüllen *(fail)* der Anforderung festgestellt werden kann. Beispiel 2 verdeutlicht diesen Zusammenhang.

Beispiel 2 (Prüfung einer Anforderung durch einen Testfall)

Anforderung ρ_1 aus Beispiel 1 (Seite 42) wird nur durch einen Testfall $t = (\langle\rangle, \langle\rangle)$ mit Eingabe $ins_t = \langle\rangle$ geprüft: In diesem Fall ist die Menge der gültigen Ausgaben $O^\omega_{\rho, ins_t} = \{\langle\rangle\}$ und die Forderung $O^\omega_{\rho, ins_t} \subsetneq O^\omega$ ist erfüllt. Somit gilt das Prädikat checks(ρ_1, t).

Für alle anderen Eingaben $ins \in I^* \setminus \{\langle\rangle\}$ gilt dagegen $O^\omega_{\rho_1, ins} = O^\omega$, das Prädikat checks ist damit nicht erfüllt.

Grad der Prüfung einer Anforderung

Die Definition der Prüfung einer Anforderung mittels des Prädikats checks erlaubt eine absolute Aussage, ob ein Testfall geeignet ist, eine Anforderung zu prüfen. Zusätzlich ist aber auch der Vergleich von mehreren Testfällen miteinander von Interesse, ob also mit einem Testfall t_1 eine stärkere oder schwächere Überprüfung einer Anforderung vorgenommen wird als mit einem anderen Testfall t_2. Hierzu ist eine Ordnung über den Testfällen notwendig. Wie in Beispiel 3 gezeigt, führen unterschiedlich gewählte Testeingaben dazu, dass die Menge $O^\omega_{\rho, ins}$ der gültigen Ausgaben im Allgemeinen von unterschiedlicher Mächtigkeit sind.

Beispiel 3 (Stärkere und schwächere Überprüfung einer Anforderung)

Zu einer Funktion $f \in \{A, B\}^\omega \to \{O, P\}^\omega$ sei

$$\rho(f) \overset{\text{def}}{\Leftrightarrow} \forall s \in \{A, B\}^* : ins = s \frown A \Rightarrow f(ins) = f(s) \frown O$$

eine Anforderung. Es ist also gefordert, dass f zu jedem A in der Eingabesequenz ein O in der Ausgabesequenz produzieren soll. Es werden weiter zwei Testfälle $t_1 = (ins_1, outs_1), t_2 = (ins_2, outs_2)$ betrachtet, wobei

$$ins_1 = \langle B, B, A \rangle$$
$$ins_2 = \langle A, B, A \rangle .$$

Dann ergeben sich für die Mengen der gültigen Ausgaben von f bezüglich ρ

$$O^\omega_{\rho,ins_1} = \{\langle O, O, O \rangle, \langle O, P, O \rangle, \langle P, O, O \rangle, \langle P, P, O \rangle\} \text{ beziehungsweise}$$
$$O^\omega_{\rho,ins_2} = \{\langle O, O, O \rangle, \langle O, P, O \rangle\}$$

(wobei hier angenommen wurde, dass auch eine Eingabe von B die Ausgabe immer um genau ein Element verlängert, also $\#ins = \#outs$ gilt).

Damit kann eine Ordnung über die Testfälle definiert werden:

Definition 8 (Stärke der Prüfung einer Anforderung) *Mit*

$$t_1 \succ_\rho t_2$$

wird angegeben, dass ein Testfall t_1 im Vergleich zu einem Testfall t_2 eine stärkere Prüfung einer Anforderung ρ vornimmt.

Die Relation

$$\succ_\rho \in (I^* \times O^*) \times (I^* \times O^*)$$

sei dabei für Testfälle $t_1 = (ins_{t_1}, outs_{t_1}), t_2 = (ins_{t_2}, outs_{t_2})$ definiert durch

$$t_1 \succ_\rho t_2 \Leftrightarrow \left|O^\omega_{\rho,ins_{t_1}}\right| < \left|O^\omega_{\rho,ins_{t_2}}\right| \vee O^\omega_{\rho,ins_{t_1}} \subsetneq O^\omega_{\rho,ins_{t_2}}.$$

Es wird somit von einer *stärkeren Prüfung* einer Anforderung gesprochen, falls die Menge der gültigen Ausgaben zu einer Testeingabe hinsichtlich einer Anforderung kleiner oder eine echte Teilmenge der Menge der gültigen Ausgaben zu einer anderen

Testeingabe ist.[12] Diese Definition ist dadurch zu rechtfertigen, dass eine kleinere Menge von gültigen Ausgaben weniger Freiheitsgrade bei der Erfüllung der Anforderung zulässt und somit die Prüfung der Anforderung stärker ist. Im Beispiel 3 kommt etwa in der Eingabe $\langle B, B, A\rangle$ die Anforderung ρ nur an einer Stelle in der Eingabesequenz zum Tragen, während $\langle A, B, A\rangle$ eine wiederholte Anwendung der Anforderung ρ prüft.

Es sei an dieser Stelle daran erinnert, dass in die hier eingeführte Bewertung der Testgüte hinsichtlich Anforderungen keine Annahmen zu Fehlerwahrscheinlichkeiten (vgl. Abschnitt 3.3.1) einfließen. Daher bleiben Annahmen wie beispielsweise *„je später A in der Eingabe erstmals auftritt, desto wahrscheinlicher ist ein Verstoß gegen die Anforderung“* oder ähnliches bei der Bewertung unberücksichtigt. Allein das Verhältnis von Testfall zu Anforderung ist Grundlage für die Bewertung.

Prüfung mehrerer Anforderungen und Granularität von Anforderungen

Wie in Abschnitt 3.3.2 dargestellt, lässt sich durch die Konjunktion von einzelnen Anforderung die Gesamtanforderung an ein System angeben:

Definition 9 (Gesamtanforderung) *Für eine nicht leere Menge $R = \{\rho_1, \rho_2, \ldots, \rho_n\}$ von Anforderungen an ein Funktion $f \in FUN$ wird die Gesamtanforderung an f mit*

$$\rho_R(f) \overset{\text{def}}{\Leftrightarrow} \rho_1(f) \wedge \rho_2(f) \wedge \ldots \wedge \rho_n(f)$$

angegeben.

Aufgrund der Verstärkung der Anforderung durch die Konjunktion gilt, unter Annahme der Widerspruchsfreiheit der Anforderungen, für jeden Testfall t mit Testeingabe ins_t

$$\forall \rho \in R : O^{\omega}_{\rho_R, ins_t} \subseteq O^{\omega}_{\rho, ins_t}$$

und damit

$$\text{checks}(\rho, t) \Rightarrow \text{checks}(\rho_R, t).$$

Dies entspricht also der intuitiven Annahme, dass eine Testeingabe, welche sich für den Test einer Anforderung ρ eignet, auch – zu einem gewissen Teil – einen Test hinsichtlich der Gesamtanforderung durchführt. Analog dazu können Aussagen zur Prüfung hinsichtlich beliebiger Teilmengen $R' \subseteq R$ von Anforderungen gemacht werden.

[12] Durch die Betrachtung der Teilmengeneigenschaft ist die Relation $\succ_\rho$ analog auch auf unendliche Mengen anwendbar.

Umgekehrt folgt daraus auch, dass die Zerlegung einer Anforderung ρ in einzelne (Teil-)Anforderungen ρ', ρ'' mit $\rho \Leftrightarrow \rho' \wedge \rho''$ zu einer stärkeren Forderung an geeignete Testfälle zur Prüfung der Anforderung ρ führt, falls dann für jede einzelne Anforderung ρ', ρ'' ein Testfall gefordert wird. Ein äquivalente Umformung von Anforderungen in mehrere einzelne Anforderungen führt also im Allgemeinen zu einer stärkeren Forderung an die Testfälle. Somit ist die Qualität der Testsuite stark abhängig von der Granularität der Anforderungen. Beispiel 4 verdeutlicht dies.

Beispiel 4 (Anforderungen unterschiedlicher Granularität)

Für eine Funktion $f \in \{A, B, C\}^\omega \to \{P, Q, R\}^\omega$ sei

$$\rho(f) \Leftrightarrow (ins = s \frown \langle A \rangle \vee ins = s \frown \langle B \rangle) \Rightarrow f(ins) = f(s) \frown \langle P \rangle$$

eine Anforderung, wobei $s \in \{A, B, C\}^*$.

Eine Prüfung von ρ ist beispielsweise mit einem Testfall

$$t = (\langle A \rangle, \langle P \rangle)$$

gegeben.

ρ lässt sich in zwei Anforderungen ρ', ρ'' zerlegen, welche eine zu ρ äquivalente Gesamtanforderung beschreiben:

$$\rho'(f) \Leftrightarrow ins = s \frown \langle A \rangle \Rightarrow f(ins) = f(s) \frown \langle P \rangle$$
$$\rho''(f) \Leftrightarrow ins = s \frown \langle B \rangle \Rightarrow f(ins) = f(s) \frown \langle P \rangle$$

Nun ist für eine Testsuite, welche die Gesamtanforderung prüft, neben t ein weiterer Testfall notwendig, zum Beispiel $t' = (\langle b \rangle, \langle p \rangle)$.

Eine weitere Zerlegung einer Anforderung ist oft möglich, wenn zusätzlich andere Anforderungen in Betracht gezogen werden, die Anforderung also vor dem Hintergrund (Kontext) anderer Anforderungen zerlegt wird. Beispiel 5 zeigt dies.

Beispiel 5 (Zerlegung von Anforderungen im Kontext anderer Anforderungen)

An eine Funktion $f \in \{A, B, C\}^\omega \to \{O, P, Q\}^\omega$ werden die folgende Anforderungen ρ_0, ρ_1 gestellt:

$$\rho_0(f) \Leftrightarrow ins = s \frown \langle x \rangle \Rightarrow f(ins) = f(s) \frown \langle y \rangle$$
$$\rho_1(f) \Leftrightarrow ins = s \frown \langle A, B, C \rangle \Rightarrow f(ins) = f(s) \frown \langle O, P, Q \rangle$$

wobei $s \in \{A, B, C\}^*$, $x \in \{A, B, C\}$ und $y \in \{O, P, Q\}$. Zusammen mit ρ_0 kann ρ_1 zerlegt werden in:

$$\rho_1'(f) \Leftrightarrow ins = s \frown \langle A \rangle \Rightarrow f(ins) = f(s) \frown \langle O \rangle$$
$$\rho_1''(f) \Leftrightarrow ins = s \frown \langle A, B \rangle \Rightarrow f(ins) = f(s \frown \langle A \rangle) \frown \langle P \rangle$$
$$\rho_1'''(f) \Leftrightarrow ins = s \frown \langle A, B, C \rangle \Rightarrow f(ins) = f(s \frown \langle A, B \rangle) \frown \langle Q \rangle$$

Wenn die Zerlegung einer Anforderung im Kontext einer anderen Anforderung auch intuitiv sinnvoll erscheint, besteht dennoch das Problem, dass damit die Frage, ob eine Anforderung durch einen Testfall geprüft wird, nicht mehr eindeutig zu beantworten ist. Zu Beispiel 5 wird etwa mit einem Testfall $t = (\langle A, B \rangle, \langle O, P \rangle)$, welcher sich aus Anforderung ρ_1'' direkt ergibt, nur ρ_0, nicht aber ρ_1 geprüft. Da die Anforderungen in diesem Beispiel insgesamt äquivalent sind, also

$$p_0(f) \wedge p_1(f) \Leftrightarrow p_0(f) \wedge p_1'(f) \wedge p_1''(f) \wedge p_1'''(f)$$

gilt, wird wiederum deutlich, wie stark die Aussage, dass ein Testfall geeignet ist, eine Anforderung zu prüfen, von der Formulierung der Anforderung abhängt. Dies gilt umso mehr, wenn Abhängigkeiten zwischen einzelnen Anforderungen bestehen. Um auch in diesen Fällen Aussagen treffen zu können, dass ein Testfall eine Anforderung prüft, wird das Prädikat checks erweitert:

Definition 10 (Prüfung einer Anforderung im Kontext) *Sei R eine Menge von Anforderungen und $\sigma \notin R$ eine weitere Anforderung. Ein Testfall t mit Eingabe ins_t prüft σ im Kontext von R, falls die Hinzunahme von σ zu einer Menge $R' \subseteq R$ dazu führt, dass sich die Menge der gültigen Ausgaben zu ins_t hinsichtlich der aus R' gebildeten Gesamtanforderung verringert. Dies drückt das Prädikat*

$$\text{checks} \in (FUN_{(I \triangleright O)} \to \mathbb{B}\text{ool}) \times \mathbb{P}\,(FUN_{(I \triangleright O)} \to \mathbb{B}\text{ool}) \times (I^* \times O^*) \to \mathbb{B}\text{ool}$$

aus. Für eine Anforderung σ, eine Menge weiterer Anforderungen R mit $\sigma \notin R$ und einen Testfall $t = (ins_t, outs_t)$ ist checks *dabei definiert als:*

$$\text{checks}(\sigma, R, t) \overset{\text{def}}{\Leftrightarrow} \exists R' \subseteq R : O^{\omega}_{\sigma_{R' \cup \{\sigma\}}, ins_t} \subsetneq O^{\omega}_{\sigma_{R'}, ins_t} \vee \left| O^{\omega}_{\sigma_{R' \cup \{\sigma\}}, ins_t} \right| < \left| O^{\omega}_{\sigma_{R'}, ins_t} \right|.$$

Vorstehende Definition entspricht der intuitiven Vorstellung, dass ein Testfall auch dann zur Überprüfung einer Anforderung geeignet ist, wenn durch diese Anforderung die Menge gültiger Ausgaben für die Testeingabe weiter eingeschränkt wird, als dies ohne diese Anforderung der Fall wäre. Der Testfall muss somit Aspekte der Anforderung betreffen, ansonsten würde die Hinzunahme der Anforderung nicht zu einer Verringerung der Menge gültiger Ausgaben führen.

Der eingeführte Begriff der *Prüfung* von Anforderungen durch Testfälle kann unmittelbar als Definiton einer Anforderungsüberdeckung übernommen werden. Für die praktische Anwendbarkeit ist es jedoch sinnvoll, den Begriff der Anforderungsüberdeckung weiter zu fassen. Dies ist Gegenstand des nächsten Abschnitts.

Prüfung nichtdeterministischer Anforderungen

Soll der Begriff der Prüfung einer Anforderung durch einen Testfall auch auf nichtdeterministische Spezifikationen von Anforderungen angewendet werden, ist zunächst notwendig, die Beschreibung von Testfällen zu verallgemeinern, da zu einer Testeingabe dann im Allgemeinen eine Menge gültiger Ausgaben den Test erfüllt, ein Testfall t weist dann allgemein die Form

$$t = (ins, \{outs_1, outs_2, \ldots, outs_n\}) \in I^* \times \mathbb{P}(O^\omega)$$

auf, wobei mitunter die Menge $\{outs_1, outs_2, \ldots, outs_n\}$ der gültigen Ausgaben unendlich sein kann.

Bei einer deterministischen Anforderungsspezifikation ist durch eine Eingabe auch die zu erwartende Ausgabe festgelegt. Damit sind alle anderen Ausgaben zu dieser Eingabe ungültige Ausgaben. Eine deterministische Spezifikation des gültigen Verhaltens impliziert also ebenso alle ungültigen (fehlerhaften) Systemabläufe. Dies ist im nichtdeterministischen Fall nicht mehr gegeben. Häufig werden dann auch *negative Anforderungen*, also Anforderungen welche ein fehlerhaftes Verhalten des Systems beschreiben, formuliert um explizit den Ausschluss dieses nicht gewünschte Verhaltens zu spezifizieren. Dies ist, je nach Art der Anforderung, im Einzelfall praktikabler als die korrekten Abläufe zu spezifizieren.

Hinsichtlich des Begriffs der Prüfung einer Anforderung ist dann zwischen *positiven* und *negativen Testfällen* zu unterscheiden: Ein Testfall

$$t^{neg} = (ins, Outs^{neg}) \in I^* \times \mathbb{P}(O^\omega)$$

gibt dann die Menge $Outs^{neg}$ der Ausgaben an, welche das zu testende Systems für die Eingabe *ins* nicht produzieren darf. Dementsprechend kann unterschieden werden, ob eine Testfall eine Anforderung positiv, negativ oder nicht prüft.

Im Gegensatz zur Testfallgenerierung aus deterministischen Spezifikationen, wie diese unter anderem von Pretschner (2003) beschrieben ist, sind Techniken zur Testfallgenerierung aus nichtdeterministischen Spezifikationen noch wenig etabliert. Aus diesem Grund wird in dieser Arbeit im Weiteren vom deterministischen Fall ausgegangen und die modellbasierte Testfallgenerierung um den Aspekt der Orientierung an einzelnen Anforderungen erweitert. In Kapitel 5, Abschnitt 5.3.4, wird in Zusammenhang mit der Modellerstellung nochmal kurz auf nichtdeterministische Modelle eingegangen.

3.3.3. Anforderungsüberdeckung

Wenn ein Testfall eine Anforderung prüft, kann für diese Anforderung die Menge der gültigen und ungültigen Ausgaben unterschieden werden. Mit Hilfe des Prädikats checks kann eine Äquivalenzrelation über Testfälle angegeben werden:

Definition 11 (Äquivalenz hinsichtlich Prüfung einer Anforderung) *Für Testfälle $t, t' \in I^* \times O^*$ und eine Anforderung $\rho \in FUN_{(I \triangleright O)} \to \mathbb{B}\text{ool}$ ist eine Äquivalenzrelation*

$$\sim_\rho \in (I^* \times O^*) \times (I^* \times O^*)$$

hinsichtlich der Prüfung von ρ durch

$$t \sim_\rho t' \overset{\text{def}}{\Leftrightarrow} \text{checks}(\rho, t) \wedge \text{checks}(\rho, t')$$

definiert. Soll die Prüfung im Kontext einer Menge R von Anforderungen betrachtet werden, wird die Äquivalenzrelation erweitert zu

$$t \sim_{\rho,R} t' \overset{\text{def}}{\Leftrightarrow} \text{checks}(\rho, R, t) \wedge \text{checks}(\rho, R, t').$$

Mit dem Prädikat checks sind somit zu jeder Anforderung ρ die Äquivalenzklassen der prüfenden und nicht-prüfenden Testfälle festgelegt: Jeder Testfall t, für den $\text{checks}(\rho, t)$ beziehungsweise $\text{checks}(\rho, R, t)$ gilt, liegt in der Äquivalenzklasse der ρ prüfenden Testfälle. Beispielsweise definieren die Anforderungen ρ_1 bis ρ_5 in Beispiel 1 entsprechende Äquivalenzrelationen und damit Klassen, in welchen Testfälle liegen können. Damit kann bereits eine Form der Überdeckung einer Anforderung durch einen Testfall definiert werden: Ein Testfall überdeckt eine Anforderung dann, wenn er diese prüft. Wie in Abschnitt 3.3.2 bereits dargestellt wurde, wird in der Literatur die Anforderungsüberdeckung nur informell angegeben, indem etwa gefordert wird, dass der Testfall aus einer Anforderung abgeleitet wird, oder der Testfall einer Anforderung zugeordnet wird. Dieser informelle Begriff der Überdeckung erlaubt, dass auch Testfälle, welche eine Anforderung nicht prüfen, diese überdecken können. Berechtigung erlangt diese Frage vor allem durch folgende Beobachtungen:

1. Anforderungen sind häufig nicht als Prädikat im Sinne der Definition 3 oder vergleichbar präzise formal spezifiziert. Die Notation einer Anforderung ist dagegen meist informell und / oder exemplarisch.
2. Analogieschlüsse verleiten häufig dazu, in ähnlichen Nutzungssituationen ein ähnliches Verhalten des Systems anzunehmen. Derartige falsche Annahmen über ein analoges Systemverhalten sind häufig Ursache einer fehlerhaften Implementierung.
3. Testfälle in Grenzwerten oder Ausnahmesituationen werden in der Literatur vielfach empfohlen.

Diese Aspekte werden in den nachfolgenden Abschnitten ausführlich diskutiert. Zusätzlich ist zu berücksichtigen, dass in der Praxis häufig nicht alle Anforderungen explizit formuliert sind, sondern auch eine Reihe impliziter Anforderungen existieren.

Informelle und / oder exemplarische Anforderungen

Die etwa in Beispiel 1 (Seite 42) verwendete explizite Angabe von Prädikaten über die Menge der Funktionen als Anforderungen ist in der Praxis nur selten gegeben. Bei komplexeren Funktionen wird die präzise Anforderungsspezifikation mittels Prädikaten äußerst umfangreich und zunehmend nicht mehr beherrschbar. Ursache ist hier vor allem, dass eine derartige Anforderungsformulierung für viele Stakeholder ohne vertiefte mathematische Kenntnisse nur schwer verständlich ist. Damit ist die Validierung der Anforderungen kaum möglich. Verbreiteter sind dagegen informelle Anforderungsbeschreibungen, etwa als unstrukturierter Text oder in Form von strukturierten Vorlagen, wie zum Beispiel von Fleischmann (2008) vorgeschlagen. Um den Nachteil der unterschiedlichen Interpretationen von informellen Beschreibungen zu entgegnen, werden häufig einfache formale und semi-formale Notationen zur Formulierung exemplarischer Abläufe eingesetzt. Hierunter fallen beispielsweise Message Sequence Charts (International Telecommunication Union (ITU), 2004) oder Sequenzdiagramme der Unified Modelling Language (Object Managment Group, Inc. (OMG), 2005). Im Hauptteil der vorliegenden Arbeit wird ebenfalls von der Angabe exemplarischer Abläufe, als Szenarien bezeichnet, ausgegangen. In Kapitel 5 wird auf die Formulierung von Anforderung als Szenarien detailliert eingegangen.

Unabhängig davon, ob Anforderungen informell oder als exemplarische Abläufe (Szenarien) beschrieben sind, besteht das Problem, dass jeweils die präzise und vollständige Anforderung im Allgemeinen unbekannt bleibt, eine eindeutige und zweifelsfreie Bestimmung der Anforderungen als Prädikate $\rho \in FUN_{(I \triangleright O)} \to \mathbb{B}\mathrm{ool}$ also nicht möglich ist. Im Falle informeller Beschreibungen können unterschiedliche Interpretationen der natürlichen Sprache zu unterschiedlichen Vorstellungen über eine Anforderung ρ führen. Sind Anforderungen zwar formal gegeben, dabei aber nur exemplarische Abläufe beschrieben, bleibt das Problem der Verallgemeinerung dieser Abläufe. Um dieser Problematik zu begegnen können auch für die Anforderung als relevant vermutete Testfälle verwendet werden. Mit deren Hilfe können Anforderungen präzisiert und vervollständigt werden sowie durch die Stakeholder validiert werden. Hierzu kann auch die in Kapitel 6 eingeführte Methode der Testfallgenerierung verwendet werden, um damit die Verhaltensspezifikation zu validieren. Die Testfälle stellen schließlich beispielhafte Abläufe dar, mit der Analyse der ermittelten Testausgaben für die jeweiligen Eingaben kann beurteilt werden, ob das spezifizierte Verhalten dem beabsichtigten entspricht.

Ohne eine präzise Anforderungsspezifikation können für eine Testeingabe die Mengen der gültigen und ungültigen Ausgaben nicht eindeutig bestimmt werden. Damit ist es

im Allgemeinen nicht möglich, zu entscheiden, ob ein Testfall eine Anforderung prüft, da das Prädikat checks nicht anwendbar ist. Dies gilt umso mehr, da grundsätzlich nicht davon ausgegangen werden kann, dass die Menge aller beschriebenen Anforderungen das beabsichtigte Verhalten vollständig spezifiziert. Möglicherweise ist es also zweifelhaft, ob eine Ausgabe zu einer Testeingabe von einer Anforderung beeinflusst werden soll. Unabhängig von verschiedenen Interpretationen einer Anforderung kann es auch beabsichtigt sein, einen Testfall einer Anforderung zuzuordnen, welche diesen tatsächlich nicht prüft. Dies ist, wie im Folgenden erläutert wird, unter anderem aufgrund falscher Analogieschlüsse oder durch die Bedeutung von Grenzwerttestfällen gerechtfertigt.

Falsche Analogieschlüsse

Aus den formulierten Anforderungen sind oftmals einzelne Abläufe offensichtlich und vergleichsweise einfach zu erfassen, während in anderen Ablaufpfaden das gewünschte Verhalten nicht unmittelbar ersichtlich ist und nur nach intensiver Durchdringung der Anforderungen deutlich wird. Dies gilt insbesondere in dem Extremfall, wenn Anforderungen nur als exemplarische Szenarien gegeben sind. Diese Problematik besteht grundsätzlich auch bei einer vollständigen formalen Anforderungsspezifikation, da sich bei der menschlichen Interpretation unterschiedliche Teile der Spezifikation unterschiedlich intuitiv erschließen.

Daraus, dass Teile der Anforderungen offensichtlicher sind als andere, besteht bei der Entwicklung eines Systems die Gefahr falscher Analogieschlüsse: Ein bestimmtes Verhalten des Systems wird analog auch für ähnliche Nutzungssituationen angenommen, beabsichtigt ist in manchen Situationen aber ein deutlich abweichendes Verhalten. Ein falscher Analogieschluss bedeutet somit die Verletzung der Äquivalenzklasse der Eingaben, welche durch die intendierte Anforderung aufgespannt wird. Dieses Problem besteht sowohl bei der Spezifikation von Anforderungen als auch bei der Realisierung des Systems. Testfälle, welche auf das Erkennen falscher Analogieschlüsse zielen, können damit zur Validierung der Anforderungen als auch zur exemplarischen Verifikation der Realisierung eingesetzt werden:

- Die Anforderungsspezifikation kann dahingehend validiert werden, ob eine Anforderung auch in ähnlichen Nutzungssituationen zutrifft.

- Die Realisierung kann dahingehend überprüft werden, ob fälschlicherweise das Verhalten einer Anforderung auch für Nutzungssituationen implementiert wurde, in denen diese Anforderung nicht zutrifft.

Falsche Analogieschlüsse können in Bezug auf eine intendierte Anforderung ρ und deren explizite Repräsentation r zu zwei Fehlersituation führen führen:

1. In einer Nutzungssituation, in welcher ρ gelten soll, wird fälschlich ein zu einer anderen Anforderung r' analoges Verhalten angenommen.

2. Ein zu r analoges Verhalten wird in einer Nutzungssituation angenommen, in welcher eine andere Anforderung ρ' gelten soll.

Im ersten Fall wird die Anforderung ρ verletzt, in letzteren ist die Repräsentation r der Anforderung ρ und ein daraus resultierender Analogieschluss Ursache für eine Verletzung einer anderen Anforderung ρ'. Beispiel 6 verdeutlicht dies.

Beispiel 6 (Falsche Analogieschlüsse)

Als Beispiel werden die Anforderungen ρ_4 und ρ_5 der Pufferfunktion aus Beispiel 1 (Seite 42) betrachtet.

Diesen (intendierten) Anforderungen seien explizit nur durch die folgende textuelle Beschreibung angegeben:

r_4: *„Wurde mit dem Put-Kommando ein Wert gesetzt, liefert ein folgendes Get-Kommando den zuletzt gesetzten Wert.'"*

r_5: *„Wurde kein neuer Wert mit dem Put-Kommando gesetzt, liefert der Aufruf des Get-Kommandos die leere Nachricht."*

Die Anforderungsspezifikation ist hier weniger präzise gefasst, ebenso wie dies häufig in der Praxis der Fall ist, insbesondere bei der Formulierung initialer Anforderungen.

Eine fehlerhafte Implementierung buffer_{impl} der Pufferfunktion zeigt nun folgendes Verhalten:

$$\text{buffer}_{impl}(\langle(\text{put}(5), \bot), (\bot, \bot), (\bot, \text{get})\rangle) = \langle \bot, \bot, \bot \rangle \quad \text{(a)}$$

$$\text{buffer}_{impl}(\langle(\text{put}(5), \bot), (\bot, \text{get}), (\bot, \text{get})\rangle) = \langle \bot, 5, 5 \rangle \quad \text{(b)}$$

Im Fall a führt ein falscher Analogieschluss von r_5 zu einem Verstoß gegen ρ_4: Irrtümlich wurde angenommen, dass ρ_5 fordere, dass der Wert unmittelbar vor Aufruf des Get-Kommandos gesetzt werden müsste. (Korrekt wäre die Ausgabe $\langle \bot, \bot, 5 \rangle$.)

Im Fall b ist eine falsche Analogieannahme von r_4 dagegen Ursache für einen Verstoß gegen ρ_5: Es wurde angenommen, dass ρ_4 fordere, jeder Aufruf eines Get-Kommandos sollte den zuletzt gesetzten Wert liefern. (Korrekt wäre die Ausgabe $\langle \bot, 5, \bot \rangle$.)

Die Zuordnung von Testfällen zu einer Anforderung ist mit jedem der beiden Zusammenhängen gerechtfertigt. Für die Bestimmung der erwarteten Ausgaben in den Testfällen ist allerdings ein zusätzliches Orakel erforderlich. In der vorliegenden Arbeit wird hierzu das Testmodell verwendet, welches die Gesamtanforderung an das System beschreibt (Kapitel 5, Abschnitt 5.3).

Grenzwerttestfälle

Unter anderen aus den zuvor erläuterten Analogieschlüssen sind Grenzwertanalysen zur Bestimmung von Testfällen motiviert. Diese wurden bereits vielfach zur Ermittlung von Testfällen empfohlen, unter anderem von Myers (1979, 2001) oder Spillner und Linz (2007), da Fehler in Programmen häufig in Grenzbereichen auftreten. Reid (1997) zeigte in einer Studie, dass Testfälle auf Basis von Grenzwertanalysen im Vergleich zur Partitionierung in Äquivalenzklassen oder randomisierten Testfällen am besten geeignet waren, um Fehler in einem System zu entdecken. Dabei sind vor allem Testfälle mit Eingabewerten an den Grenzen von Äquivalenzklassen von Interesse, wobei sowohl Werte innerhalb wie auch außerhalb der Klasse gewählt werden. Beispielsweise sollen Funktionen, welche etwa eine positive Zahl als Eingabe erwarten, auch mit 0 oder negativen Werten getestet werden. Bei einer erwarteten Zeichenkette ist ebenso ein Test mit der leeren Zeichenkette zu empfehlen.

Wie zuvor erläutert wurde, definiert das Prädikat checks eine Äquivalenzklasse von prüfenden Testeingaben hinsichtlich einer Anforderung. Jede dieser Eingaben ist dazu geeignet, die Anforderung zu prüfen. Vor dem Hintergrund des Testens von Grenzwerten sind jedoch auch Testfälle außerhalb dieser Äquivalenzklasse für eine Anforderung von Interesse. Die Zuordnung einer Anforderung zu Testfällen, welche außerhalb der prüfenden Eingaben für diese Anforderung liegen, jedoch „nahe“ an den Grenzen der Klasse der prüfenden Eingaben liegt, ist damit ebenso gerechtfertigt. Werte die „nahe“ an diesen Grenzen liegen, weisen nur geringe Abweichungen zu den prüfenden Eingaben auf. Sie sind von besonderen Interesse, da intuitiv meist ein linearer Abschluss dieser Äquivalenzklassen angenommen wird. Es wird also unterstellt, dass geringfügige Änderungen der Eingaben zu nur geringfügigen Änderungen in der Ausgabe führen. Dies ist in der Regel dann der Fall, wenn Eingaben in der durch das Prädikat checks für eine Anforderung ρ definierten Äquivalenzklasse liegen. Führen marginale Änderungen der Eingabe in einem Testfall t dazu, dass diese nicht $\text{checks}(\rho, t)$ nicht mehr gilt, ändert sich das Ausgabeverhalten dagegen deutlich. Besonders zur Validierung von Anforderungen sind solche Testfälle außerhalb der Äquivalenzklasse hilfreich. Dies gilt umso mehr, da in der Praxis meist davon ausgegangen werden muss, dass die Menge der explizit angegebenen Anforderungen unvollständig ist.

Beim Test von reaktiven Systemen bilden Sequenzen von Eingabeaktionen die Eingaben. Diese Aktionen umfassen zwar auch Werte aus kontinuierlichen Datenbereichen (zum Beispiel Sensorwerte), aber häufig auch diskrete Werte, wie Kommandos von Nutzern oder Systemereignisse. Die Bestimmung von Grenzfällen ist hier im Allgemeinen nicht ohne Weiteres möglich. Im Gegensatz zu einfacheren Daten, über deren Typen eine Ordnung definiert ist, unterliegen Sequenzen von Aktionen keiner eindeutigen Ordnung. Es kann somit zwar unterschieden werden, ob eine Eingabe eine Anforderung prüft oder nicht, allerdings kann nicht präzise festgelegt werden, ob eine Eingabesequenz nahe an dieser Grenze liegt.

Bedeutung der Spezifikationstechnik

Zusammenfassend lässt sich festhalten, dass die Zuordnung von Anforderungen und Testfällen – also die Überdeckung einer Anforderung durch einen Testfall – in der Regel nicht präzise und eindeutig vorgenommen werden kann. Inwieweit eine Zuordnung sinnvoll und eindeutig möglich ist, hängt insbesondere von den folgenden Faktoren ab:

- Der Granularität der Anforderungen,
- der Vollständigkeit der Anforderungen,
- dem Grad der Formalisierung der Anforderungen sowie
- der Existenz und der Ausprägung einer Ordnung über den Eingaben.

Diese Faktoren werden zu einem großen Teil von der Spezifikationstechnik bestimmt, in welcher die Anforderungen angegeben werden. Im Hauptteil der vorliegenden Arbeit werden Anforderungen durch exemplarische Ein-/Ausgabe-Szenarien formalisiert. Diese Szenarien bilden die Ausgangsbasis für die anforderungsorientierte Testfallermittlung. Im nächsten Abschnitt wird deshalb näher auf die Überdeckung von Anforderungsszenarien eingegangen.

3.3.4. Überdeckung von Anforderungsszenarien

Die Bewertung der Anforderungsüberdeckung ist sowohl hinsichtlich informell beschriebenen Anforderungen, als auch hinsichtlich einer präzisen formalen Spezifikation von Anforderungen, wie in Definition 3 (Seite 42) angegeben, nicht zufriedenstellend:

- Informell angegeben Anforderungen erlauben nur sehr primitive Bewertungen der Überdeckung von Anforderungen Testfälle, wie etwa die Angabe, ob zu jeder Anforderung (mindestens) ein Testfall identifiziert wurde.
- Die Angabe von Anforderungen als Prädikate erschwert die Validierung der Anforderungsspezifikation durch alle Stakeholder. Diese Stakeholder können damit mitunter nicht beurteilen, ob eine gegebene Überdeckung sich auf ihre Intention der Anforderungen bezieht.

Während informelle Anforderungen also nur eine sehr grobe Zuordnung von Testfällen zu Anforderungen ermöglichen, wird bei formalen Prädikaten möglicherweise das Ziel der Bewertung der Anforderungsüberdeckung verfehlt: Gerade gegenüber Auftraggebern oder den späteren Nutzern des Systems soll das Überdeckungsmaß Aussagekraft besitzen, ob die von diesen Stakeholdern geforderten Anforderungen durch Tests adressiert werden.

Einerseits ist also eine formale Anforderungsspezifikation erforderlich, um eine nicht nur intuitive Zuordnung von Testfällen zu Anforderungen zu ermöglichen, andererseits muss dazu eine Spezifikationstechnik verwendet werden, welche auch von den Stakeholdern, insbesondere den Auftraggebern, möglichst einfach zu validieren ist. Aus diesem Grund werden in dieser Arbeit *Szenarien* zur formalen Spezifikation von Anforderungen verwendet.

Die Grundidee der verwendeten Szenarien zur formalen Anforderungsspezifikation liegt darin, dass eine Anforderung ρ, welche sich auf eine Menge von Eingaben $I^{\omega}_{\rho} \subsetneq I^{\omega}$ bezieht (Definition 6, Seite 45), nur durch eine charakteristische endliche Sequenz von Eingabeaktionen $\langle i_1, i_2, \ldots i_n \rangle \in I^{\omega}_{\rho}$ und den zugehörigen Soll-Ausgaben beschrieben wird.[13]

Abbildung 3.1 zeigt den Zusammenhang zwischen der Gesamtanforderung an die Funktion, einer einzelnen Anforderung ρ und der Repräsentation von ρ durch ein Szenario anhand der jeweils beschriebenen Pfade in einem Kontrollflussgraphen.

Anstelle einer vollständigen Formalisierung der einzelnen Anforderungen werden als deren formale Repräsentation somit nur besonders charakteristische Abläufe angegeben. In der Regel werden diese durch informelle Beschreibungen ergänzt. Das vollständige Verhalten des Systems ist somit allein aus den Szenarien nicht ersichtlich, dies kann dagegen zum Beispiel durch eine Zustandsmaschine angegeben werden. Für die Anforderungen der Puffer-Funktion buffer sind in Beispiel 7 typische Szenarien angegeben.

Beispiel 7 (Anforderungen als Szenarien)

Szenerien für die Anforderungen des einelementigen Puffers (Beispiel 1, Seite 42) sind in Tabelle 3.4 dargestellt.

Tabelle 3.4.: Szenarien zu Anforderungen der Funktion buffer

Anforderung	Eingabe	Soll-Ausgabe
ρ_1	$\langle\rangle$	$\langle\rangle$
ρ_2	$\langle(\bot, \bot), (\mathsf{put}(1), \bot)\rangle$	$\langle\bot, \bot\rangle$
ρ_3	$\langle(\bot, \bot), (\bot, \mathsf{get})\rangle$	$\langle\bot, \bot\rangle$
ρ_4	$\langle(\mathsf{put}(1), \bot), (\mathsf{put}(2), \bot), (\bot, \bot), (\bot, \mathsf{get})\rangle$	$\langle\bot, \bot, \bot, 2\rangle$
ρ_5	$\langle(\mathsf{put}(1), \bot), (\bot, \mathsf{get}), (\bot, \mathsf{get})\rangle$	$\langle\bot, 1, \bot\rangle$

Auf die Formulierung von Anforderungen von Szenarien wird in Kapitel 5 im Detail eingegangen. Diese Szenarien sind dann (neben einem Verhaltensmodell in Form einer Zustandsmaschine) die Ausgangsbasis für die in dieser Arbeit dargestellte Methode der anforderungsorientierten Testfallermittlung. Die zuvor grundlegend diskutierten

[13] Zur Präzisierung der Anforderung kann es gegebenenfalls notwendig sein, zu einer intendierten Anforderung ρ mehrere Szenarien anzugeben. An dieser Stelle sei zur Vereinfachung eine 1 : 1-Zuordnung angenommen.

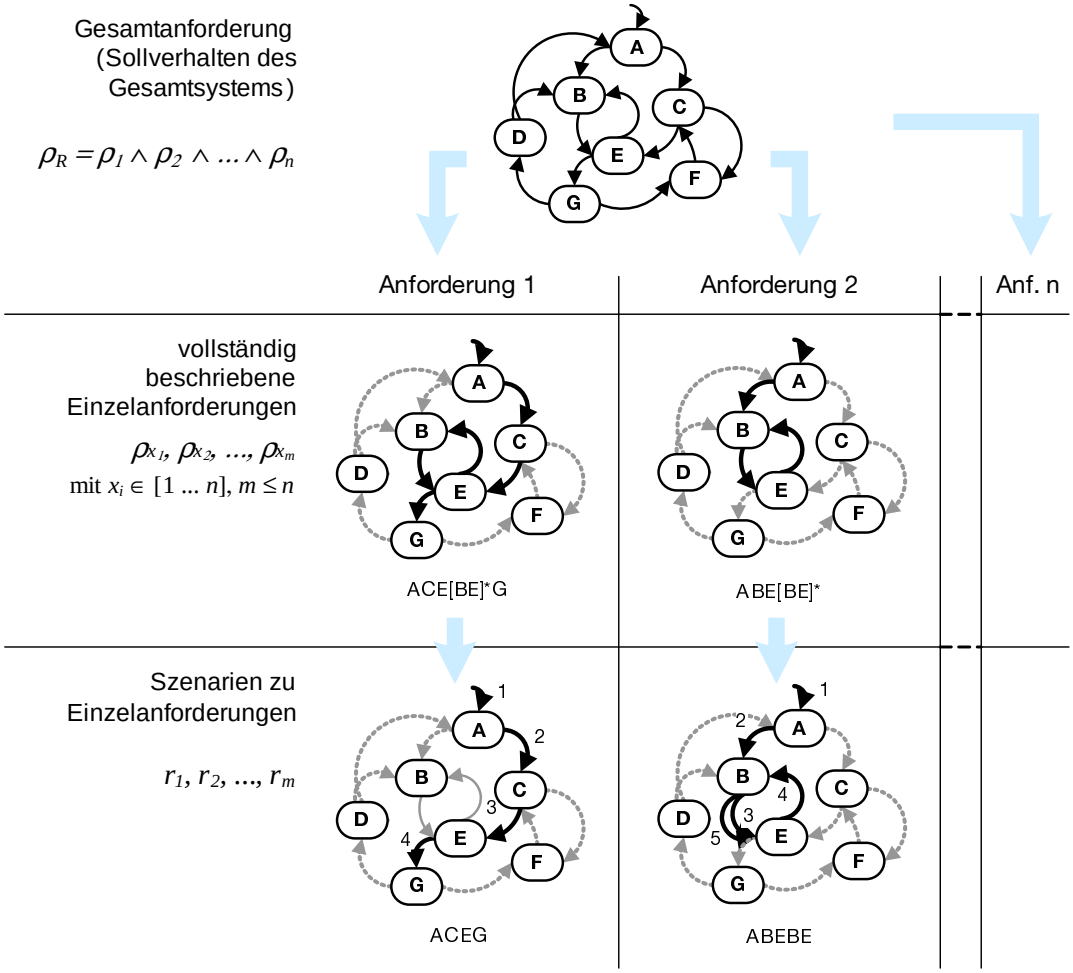

Abbildung 3.1.: Gesamtanforderung, einzelne Anforderung und Szenario, illustriert durch Kontrollflussgraphen: Zu einem System (oben) mit Gesamtanforderung ρ_R werden unterschiedliche Einzelanforderungen identifiziert. Eine Anforderung ρ_i betrifft nach ihrer Intention eine Teilmenge aller Systemabläufe (Mitte). Szenarien r_i repräsentieren davon einen charakteristischen Ablauf (unten). Zusätzlich zu den Kontrollflussgraphen sind die jeweils möglichen Folgen im Kontrollfluss als reguläre Ausdrücke angegeben.

Überlegungen zur Qualitätsbewertung von Testfällen hinsichtlich Anforderungen, insbesondere das Konzept der *Prüfung einer Anforderung durch einen Testfall* (Abschnitt 3.3.2), werden später in Kapitel 7 zur Bewertung dieser Methode herangezogen. Dazu werden Annahmen über die intendierten Anforderungen getroffen, welche durch Szenarien nur exemplarisch repräsentiert sind. Zuvor gibt das folgende Kapitel einen Überblick über die Methode und den Prozess zur anforderungsorientierten Testfallermittlung.

3.4. Zusammenfassung

Methoden zur Testfallermittlung werden heute zumeist entweder anhand struktureller Überdeckungskriterien oder ihrer Eignung zur Fehleraufdeckung bewertet. In beiden Fällen ist die Aussagekraft der Bewertung fraglich: Ob eine strukturelle Überdeckung des Programmcodes durch Testfälle in einer Relation zur Fähigkeit, Fehler aufzudecken, steht ist nach wie vor umstritten. Experimentelle Ergebnisse zur Fehleraufdeckung sind mit dem Problem behaftet, dass unklar ist, inwieweit sich diese Ergebnisse auf andere Testobjekte übertragen lassen. Insbesondere werden meist keine Aussagen getroffen, in welcher Klasse von Systemen eine vergleichbare Fehlerverteilung zu erwarten ist. Vermutlich ist es auch nicht möglich, hierzu einen Nachweis zu erbringen. Bewertungskriterien, welche auch bei der Testfallerstellung operationalisierbar sind – und somit konstruktiv eingesetzt werden können – müssen sich daher auf die Beurteilung der Testfälle in Relation zu anderen Dokumenten begnügen. Abhängig von den betrachteten Dokumenten lassen sich unterschiedliche Qualitätsbewertungen klassifizieren. Die Qualität der Testfälle in Relation zu Programmcode, also die Messung struktureller Überdeckung, ist hier eine Klasse der möglichen Bewertungen. Der vorliegenden Arbeit liegt aber die Beurteilung der Testfälle in Relation zu einzelnen Anforderungen zu Grunde, also der Überdeckung von Anforderungen durch Testfälle.

Außer trivialen Kriterien („mindestens ein Testfall pro Anforderung“) finden bisher kaum Metriken zur Bewertung der Anforderungsüberdeckung Verwendung. Falls einzelne Anforderungen vollständig in Form von Prädikaten formalisiert sind, kann angegeben werden, ob ein Testfall eine Anforderung prüft. Mithilfe dieser Eigenschaft der Prüfung kann eine Anforderungsüberdeckung definiert werden. In der Praxis liegen Anforderungen jedoch nur selten in einer derartigen formalen Form vor. Außerdem ist damit die Validierung der Anforderung durch die Stakeholder oft nicht möglich. Aus diesem Grund werden im Hauptteil dieser Arbeit Anforderungen durch exemplarische Szenarien formal repräsentiert. Die Qualitätsbewertung von Testfällen hinsichtlich solcher Anforderungsszenarien wird weiter in Kapitel 7 detaillierter beschrieben, die vorhergehende Diskussion bildet dafür die Grundlage.

4. Überblick über den Entwicklungs- und Testprozess

Bevor in den nachfolgenden Kapiteln 5 und 6 die erarbeitete Methode zur anforderungsorientierten modellbasierten Testfallgenerierung im Detail beschrieben wird, soll zunächst ein Überblick über den mit der Methode verbunden Entwicklungsprozess gegeben werden. Dieser Prozess wurde vom Autor bereits teilweise in (Pfaller und Pister, 2008; Pfaller, 2008) veröffentlicht, in Abbildung 4.1 ist er schematisch dargestellt.

Dieses Kapitel dient vornehmlich dazu, die einzelnen Schritte im Gesamtbild des Prozesses darzustellen. Die methodischen Vorgehensweisen, welche in den einzelnen Schritten angewendet werden, sind im Detail Gegenstand der nachfolgenden Kapitel.

4.1. Ausgangslage

Es wird zunächst angenommen, dass Anforderungsdokumente vorliegen, welche informell Anforderungen an das zu testende beziehungsweise an das zu entwickelnde System wiedergeben. In diesen Dokumenten soll eine Menge einzelner funktionaler Anforderungen identifizierbar sein, welche das gewünschte Soll-Verhalten des Systems zumindest teilweise beschreiben. Der Test des Systems soll sich im Besonderen auf diese explizit beschriebenen Anforderungen fokussieren.

Natürlicherweise sind möglichst vollständige, detaillierte und widerspruchsfreie Spezifikationen von Anforderungen bereits zu Beginn eines Entwicklungsprojekts wünschenswert. Gleichwohl ist es in den frühen Phasen der Softwareentwicklung meist nicht ohne weiteres möglich, derart präzise Dokumente zu erstellen – sei es aus Mangel an Zeit oder aufgrund von Unsicherheiten über die genaue Ausgestaltung des zu entwickelnden Systems. In der vorliegenden Arbeit wird deshalb allgemein auf die Forderung verzichtet, dass die ursprünglichen Anforderungsdokumente vollständig oder widerspruchsfrei sein müssen. Die detaillierte funktionale Anforderungsspezifikation des Systems wird erst durch die spätere Modellerstellung erreicht, siehe später in Abschnitt 4.2.3.

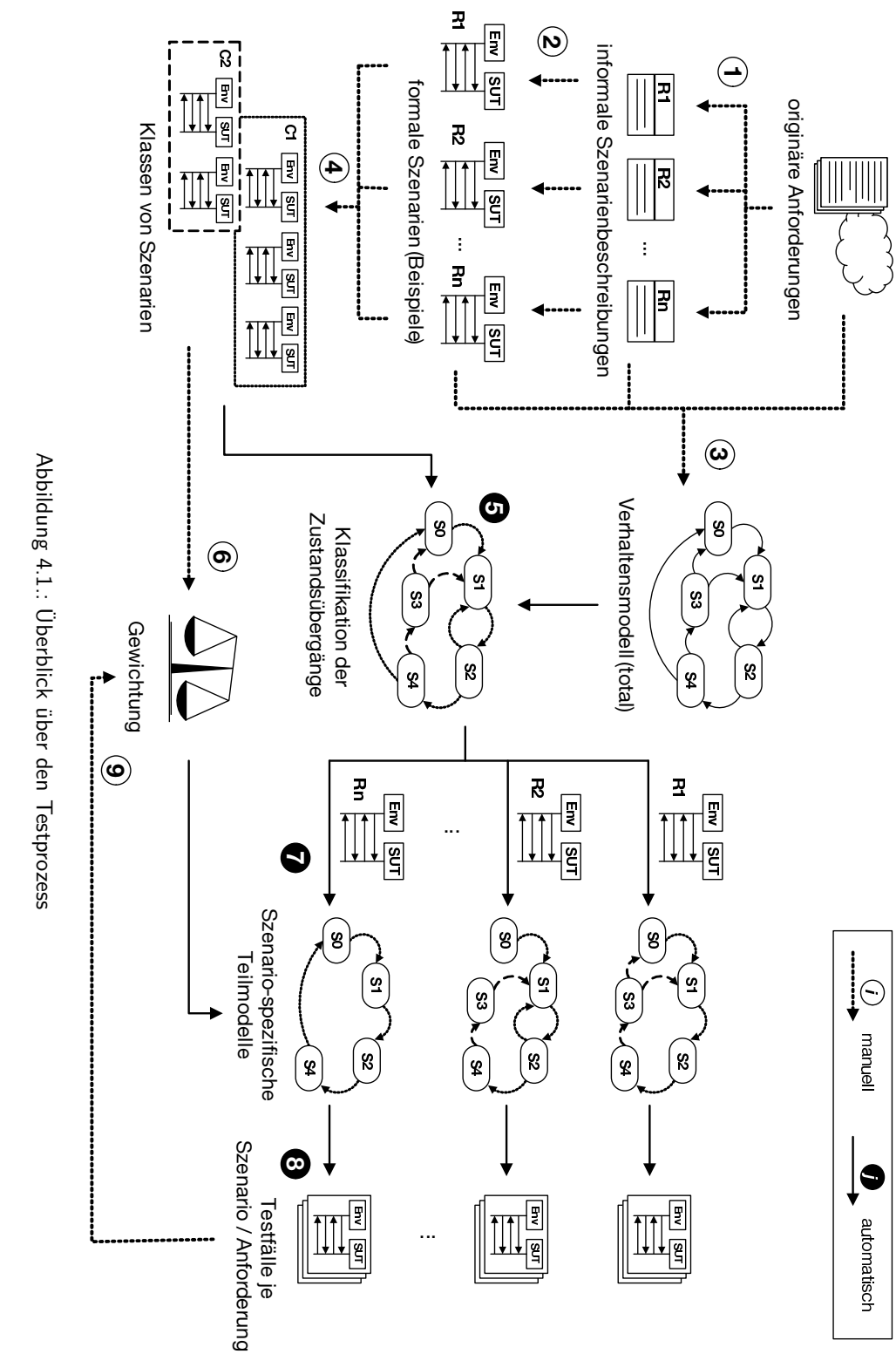

Abbildung 4.1.: Überblick über den Testprozess

4.2. Anforderungsspezifikation

Ein Methode zur automatisierten Testfallermittlung, welche auf Anforderungen basiert, setzt voraus, dass diese Anforderungen in einer Art gegeben sind, so dass sie automatisiert verarbeitet werden können. In den folgenden Abschnitten werden die nötigen Schritte während der Spezifikation[1] des Systemverhaltens erläutert, welche die spätere Anwendung der Testmethodik ermöglichen.

4.2.1. Identifikation einzelner funktionaler Anforderungen

Im ersten Schritt ist es notwendig, in den Anforderungsdokumenten einzelne funktionale Anforderungen zu identifizieren und dazu informelle Beschreibungen von Szenarien anzugeben **(1)**[2]. Im nächsten Schritt soll zu jeder dieser Anforderungen ein exemplarischer Ablauf formal angeben werden. Daher ist es notwendig, dass die Einzelanforderungen derart feingranular sind, dass daraus ein typisches, konkretes Ausführungsszenario (Folge von Ein- und Ausgaben) ersichtlich ist. Gegebenenfalls ist es notwendig, dass die Anforderungen hier zunächst verfeinert werden müssen.

4.2.2. Angabe exemplarischer Szenarien zu Anforderungen

Die einzelnen funktionalen Anforderungen, welche im vorhergehenden Schritt identifiziert wurden, liegen bisher nur als informale Szenarienbeschreibungen vor. Um Anforderungen bei der Testfallgenerierung verwenden zu können, ist dagegen eine formale Repräsentation der Anforderung notwendig. Die Erstellung einer vollständigen und präzisen formalen Spezifikation einzelner Anforderungen, beispielsweise in Formeln temporaler Logik oder als Prädikate, ist dabei oftmals kein trivialer Vorgang, zudem sind die daraus resultierenden Beschreibungen meist nur für Spezialisten verständlich. Eine Validierung der formalen Repräsentation einer Anforderung durch die verschiedenen Stakeholder (Auftraggeber, Anwender) hinsichtlich der Frage, ob die Anforderung ihre Bedürfnisse und Erwartungen korrekt wiedergibt, ist deshalb nur schwer möglich.

Aus diesem Grund erfordert die dargestellte Methode nicht die vollständige formale Spezifikation der einzelnen Anforderungen, stattdessen wird nur die Angabe exemplarischer, formaler Szenarien gefordert, welche ein Beispiel für Ein-/Ausgabefolgen eines

[1] Im Rahmen der vorliegenden Arbeit wird als *Spezifikation* stets die Spezifikation der Anforderungen an das System verstanden, welche lediglich das für die Systemumgebung wahrnehmbare Verhalten des Systems beschreibt. Zu unterschieden ist davon die, meist technische, *Spezifikation der Implementierung*. Letztere ist in der vorgestellten Methode ohne große Bedeutung.

[2] Mit **(.)** ist die entsprechende Ziffer aus Abbildung 4.1 für den jeweiligen Verfahrensschritt angegeben.

Systemablaufs angeben, der typisch für die einzelne Anforderung ist **(2)**. Die Formalisierung der Anforderungen beschränkt sich somit auf das Festlegen der syntaktischen Ein- und Ausgabe Schnittstelle des Systems, also der Bestimmung der Mengen von Ein- und Ausgabeaktionen, und die Definition von exemplarischen Folgen über diese Aktionen, welche Anwendungsfälle des Systems an seiner Schnittstelle darstellen.

Die Ein- und Ausgabeaktionen werden dabei an dieser Stelle der Spezifikation bewusst abstrakt gehalten, sie dienen der Repräsentation der Interaktion zwischen den Nutzern des Systems, oder allgemeiner der Systemumgebung, mit dem System. Deshalb sollen diese der Sicht des Nutzers entsprechen und nicht in erster Linie der (späteren) tatsächlichen Realisierung. Für ein Steuergerät, welches beispielsweise über ein Bussystem wie CAN[3] oder FlexRay[4] kommuniziert, soll die Aktion nicht als konkrete Bus-Nachricht dargestellt werden, sondern davon zur Bedeutung für den Nutzer beziehungsweise die Systemumgebung abstrahieren - also etwa „Schalter gedrückt“.

4.2.3. Erstellung eines vollständigen Modells zur Verhaltensbeschreibung

Die Anforderungsdokumente sowie die exemplarischen Szenarien spezifizieren das geforderte Verhalten im Allgemeinen nur in den typischen Situation. Es ist im Normalfall nicht möglich, aus den Anforderungsdokumenten und/oder den Szenarien für jede mögliche Folge von Eingaben eindeutig auf die entsprechenden korrekten Ausgaben des Systems zu schließen. Die Anforderungsdokumente lassen hier aufgrund ihrer Natürlichsprachlichkeit Interpretationsspielräume und lassen oft, teilweise auch bewusst, Freiräume. Die formalen Anforderungsszenarien stellen dagegen nur exemplarische, typische Abläufe zu den einzelnen Anforderungen dar. Aus der tatsächlichen Nutzung des Systems ergeben sich durch das Aufeinanderfolgen unterschiedlicher Szenarien potentiell beliebig lange Abläufe. Daneben bleibt der interne Zustand, in welchem sich das System nach jedem Verarbeitungsschritt befindet, unbekannt. Aufgrund der allgemein unbeschränkten Länge der Systemabläufe ergibt sich somit eine potentiell unendliche Menge möglicher Abläufe; diese sind nicht durch eine endliche, vergleichsweise kleine, Zahl von Szenarien darstellbar. Ziel der hier vorgestellten automatisierten Ermittlung von Testfällen ist es letztendlich, eine Menge von Testfällen zu erhalten, welche mehr Aspekte des Verhaltens berücksichtigt, als dies alleine durch die exemplarischen Szenarien beschrieben ist.

Dazu wird ein Verhaltensmodell, gegeben als Zustandsmaschine, als Basis für die Testfallgenerierung erstellt **(3)**. Dieses Modell stellt eine Verfeinerung der ursprünglichen Anforderungsspezifikation dar, da es das wahrnehmbare Systemverhalten nun total beschreibt, also zu jeder möglichen Eingabe eine entsprechende Ausgabe liefert. Die Modellierung durch Zustandsmaschinen bietet hier den Vorteil, dass damit grundsätzlich auch unendliche Abläufe beschrieben sind. Da die Szenarien diese Totalität nicht

[3] Controller area network (ISO 11898-1, 2003)

[4] `http://www.flexray.com/`

besitzen (können), ist zusätzliches Wissen zu Erstellung des Systems notwendig. Diese weiteren Informationen können einerseits etwa aus informal beschriebenen Bedingungen in den Anforderungsdokumenten stammen, die nicht durch Szenarien erfasst wurden, oder es kann sich um implizite Rahmenbedingungen der Anwendungsdomäne handeln, welche nicht explizit als Anforderungen erfasst sind. Außerdem können bewusst Freiheiten für diese spätere Spezifikationsphase offen gelassen werden, zu denen eine Entscheidung erst bei der Modellbildung vorgesehen ist. Aus diesem Modell ist auch das Verhalten der Kombination unterschiedlicher einzelner Anforderungen ersichtlich, es gibt auch das entsprechende Sollverhalten vor, wenn nacheinander verschiedene einzelne Anforderungen ausgeführt werden.

Für dieses Verhaltensmodell wird weiterhin gefordert, dass es von der zu testenden Implementierung abstrahiert, ansonsten würde das System quasi gegen sich selbst getestet. Das Modell soll lediglich das Black-Box-Verhalten des Systems beschreiben, wie es von der Systemumgebung wahrgenommen wird. Details zur technischen Realisierung, die aus Sicht der Systemumgebung unbedeutend sind, soll das Modell nicht beschreiben. Für die vorgestellte Methode ist es essentiell, dass die Ein-/Ausgabeschnittstelle des Modells der Schnittstelle der Szenarien entspricht, bzw. die Aktionen in den Szenarien auf Aktionen im Modell abbildbar sind. Zudem ist es erforderlich, dass alle Szenarien, sofern sie als Anforderung Bestand haben sollen, vom Modell akzeptiert werden.

Die so erstellte Verhaltensbeschreibung kann als Spezifikation für die weitere Entwicklung des Systems dienen. Für die in dieser Arbeit vorgestellte Methode zur Testfallermittlung ist es von untergeordneter Bedeutung, ob das System auf Basis dieses Modells erstellt wurde oder unabhängig davon entwickelt wurde. Wesentlich für die Methode ist allerdings, dass das Modell das Sollverhalten spezifizieren muss, welches das zu testende System zu erfüllen hat.

4.3. Anforderungsorienterter Test

Die bisher vorgestellten Schritte dienten vornehmlich zur Erstellung einer vollständigen abstrakten Verhaltensspezifikation für das System. Diese könnte auch allein zu Zwecken der Spezifikation verwendet werden. Diese Art der Spezifikation ermöglicht die in dieser Arbeit vorgestellte automatisierte Ermittlung von Testfällen. Die folgende Abschnitte beschreiben die Verfahrensschritte, welche für die weitere Ermittlung der Testfälle notwendig sind.

Grundidee der Methode ist es, dass zu jeder Anforderung – genauer zu jedem der definierten Szenarien – zunächst ein Ausschnitt aus dem Verhaltensmodell des Systems bestimmt wird und die Testfallgenerierung für diese Anforderung nur dieses Teilmodell betrachtet. Die Menge aller Testfälle ergibt sich damit aus der Vereinigung der Mengen von Testfällen, die zu den einzelnen Szenarien aus den jeweiligen Teilmodellen erzeugt wurden. Um diese Teilmodelle bestimmen zu können, werden die Szenarien zunächst

in verschiedene Anforderungsklassen eingeteilt und diese Klassifikation auf das Modell übertragen. Eine Gewichtung der Anforderungsklassen definiert die konkrete Wahl des Modellausschnitts.

4.3.1. Klassifikation von Szenarien

Um auf Anforderungen basierte Testfälle zu erzeugen wird zunächst eine Klassifikation der Szenarien in Anforderungsklassen vorgenommen **(4)**, jedes zu einer Anforderung gehörende Szenario wird dabei einer oder mehreren Klassen zugeordnet. Die Klassifizierung von Anforderungsszenarien dient dazu, zwischen für die Testfallermittlung besonders bedeutenden Anforderungen und weniger relevanten Anforderungen zu unterscheiden. Es wird also angenommen, dass nicht alle, mittels Szenarien repräsentierten, Anforderungen von vergleichbarer „Wichtigkeit“ für den Test des Systems sind.

Aus der Klassifikation der Szenarien ist direkt eine Klassifikation der Zustandsübergänge im Modell ableitbar **(5)**: Schaltet eine Transition bei Ausführung eines Szenarios der Anforderungsklasse C so wird diese Transition auch der Klasse C zugeordnet. Schaltet eine Transition in Szenarien verschiedener Anforderungsklassen so werden der Transition alle entsprechenden Klassen zugeordnet. Es ist zu beachten, dass nicht alle Transitionen einer Klasse zugeordnet werden: Transitionen die während der Modellerstellung zur Totalisierung ergänzt wurden, werden in keinem der Szenarien ausgeführt. Zusammen mit der im folgenden Abschnitt definierten Gewichtung bestimmt die Klassifikation der Zustandsübergänge die spätere Wahl des Modellausschnitts für die einzelnen Anforderungen.

4.3.2. Gewichtung der Anforderungsklassen

Im vorhergehenden Schritt wurden die Anforderungen, beziehungsweise die den Anforderungen zugeordneten Szenarien, in unterschiedliche Klassen eingeteilt, und diese Klassifikation wurde auch auf die Zustandsübergangsrelation des Modell übertragen. Zur Steuerung der Testfallgenerierung wird für jede einzelne Anforderungsklasse im nächsten Schritt eine Gewichtung festgelegt **(6)**. Eine höhere Gewichtung bedeutet hier einen höheren Einfluss der Anforderungen einer Klasse auf die Testfallgenerierung insgesamt. Insbesondere bedeutet dies, dass die mit dieser Anforderungsklasse assoziierten Modellteile einen höheren Einfluss auf die Erzeugung der Testfälle haben. Die zugrunde liegenden Anforderungen wirken sich somit stärker auf die Testfallermittlung insgesamt aus.

Die Gewichtung stellt zusammen mit der Klassifikation der Anforderungen die Parametrierung der anforderungsspezifischen Testfallermittlung dar. Für die Wahl einer angemessenen Belegung dieser Parameter kann dabei keine absolute Empfehlung

gegeben werden, da sie sehr stark von der spezifischen Situation in einem Entwicklungsprojekt als auch von der Zielsetzung der Testaktivitäten abhängt. Es wird davon ausgegangen, dass der Testingenieur ein intuitives Verständnis entwickelt hat, welche Art von Testfällen für ein bestimmtest System viel versprechend und lohnend – im Sinne des Erkennens von Fehlern – ist. Durch die Gewichtung der Anforderungsklassen hat der Testingenieur die Möglichkeit, die ansonsten weitgehend automatisierte Testfallermittlung derart zu beeinflussen, dass das Ergebnis diesem intuitiven Verständnis möglichst nahe kommt.

4.3.3. Bestimmung anforderungsspezifischer Teilmodelle

Das weitere Vorgehen erfolgt nun separat für jedes einzelne Anforderungsszenario. Ausgehend von einem dieser Szenarien, der Klassifikation der Zustandsübergangsrelation und der Gewichtung der Anforderungsklassen wird aus dem ursprünglichen Modell ein szenario- und damit anforderungsspezifisches Teilmodell bestimmt **(7)**. Welche Modellteile dabei in einem Teilmodell ausgewählt werden, bestimmt sich im Wesentlichen durch zwei Faktoren: Zunächst wird das Szenario betrachtet und welche Zustandsübergänge bei Ausführung dieses Szenarios erfolgen. Ausgehend von diesen Transitionen wird das Modell um weitere Teile der Zustandsübergangsrelation erweitert. Der Umfang dieser Erweiterung hängt im Weiteren von der Gewichtung der Anforderungsklassen ab. Transitionen, welche einer niedrig gewichteten Anforderungsklasse zugeordnet sind, werden dabei nur gewählt, falls diese (abhängig von der konkreten Gewichtung) in wenig zusätzlichen Schritten, ausgehend von dem Pfad des Szenarios, erreichbar sind. Transitionen höherer Gewichtung werden auch gewählt, falls sie „weiter entfernt“ vom betrachteten Szenario liegen (d. h. eine höhere Anzahl von Ausführungsschritten ist notwendig um sie zu erreichen). Auf diese Art wird für jedes Szenario ein anforderungsspezifisches Teilmodell ermittelt, welches weiter die Basis für die modellbasierte Testfallgenerierung je Szenario ist. Da ein Teilmodell auch Übergänge, welche aus anderen Anforderungen oder aus der Vervollständigung des Modells folgen, umfassen kann, wird vermieden, dass die Anforderungen im Test jeweils nur isoliert betrachtet werden.

4.3.4. Generierung von Testfällen in den Teilmodellen

Das Teilmodell zu einem Szenario dient nun als Testmodell, um zu der mit dem Szenario verbundenen Anforderung Testfälle zu ermitteln. Das Teilmodell stellt somit einen Ausschnitt aus dem Gesamtmodell dar, welcher sich insbesondere auf die betreffende Anforderung fokussiert. Sofern die Gewichtung der Anforderungsklassen sinnvoll gewählt wurde, beschreibt das Teilmodell zusätzliches Verhalten, das über das im Szenario spezifizierte Verhalten hinausgeht, aber dabei Verhaltensanteile ausblendet, welche kaum in Bezug zu der Anforderung stehen.

Zur Testfallgenerierung werden (bekannte) Techniken eingesetzt, welche Testfälle nach strukturellen Kriterien aus dem (Teil-)Modell ermitteln. Beispielsweise Testfallgeneratoren, welche eine Menge von Testfällen erzeugen, die eine Überdeckung der Transitionen oder Zustände im Modell sicherstellt. Somit wird zu jedem Szenario, und damit zu jeder Anforderung, eine Menge von Testfällen ermittelt. Eine Verfolgung von Anforderungen zu Testfällen *(Traceability)* wird auf diese Weise ohne weiteren Aufwand erreicht.

Die Menge aller Testfälle für das System bestimmt sich aus der Vereinigung der Mengen von Testfällen, welche aus den Teilmodellen zu den einzelnen Szenarien ermittelt wurden. Zusätzlich können aus der Gesamtmenge alle diejenigen Testfälle eliminiert werden, welche Präfix eines anderen (längeren) Testfalls sind. Daraus ergibt sich, dass die Kardinalität der Relation zwischen Anforderungen (bzw. Szenarien) und Testfällen letztendlich nicht $1 : n$ sondern allgemein $m : n$ lautet. Einer Anforderung ist allgemein also eine Menge von Testfällen zugeordnet – aber ein Testfall kann auch dem Test mehrerer Anforderungen dienen.

4.3.5. Review von Testfällen und Anpassung der Gewichtung

Wie bereits in Abschnitt 4.3.2 erwähnt kann für die Gewichtung der einzelnen Anforderungsklassen keine absolute allgemeine Empfehlung vorgegeben werden. Daher ist es sinnvoll, dass die erzeugten Testfälle zunächst vom Testingenieur und gegebenenfalls anderen Domänenexperten einem Review unterzogen werden. Dieses Review dient zur Feststellung, inwieweit die Testfälle dem intuitiven Verständnis von *lohnenden* Testfällen entsprechen. Stellt der Testingenieur etwa fest, dass eine bestimmte Klasse von Anforderungen zu wenig, andere Anforderungen aber zu dominant in den Testfällen wiedergegeben werden, ist die Gewichtung entsprechend anzupassen. Wird etwa festgestellt, dass der Bezug von Anforderungen zu Testfällen kaum ersichtlich ist, kann die Gewichtung insgesamt reduziert werden, um spezifischere Testfälle zu erzeugen.

4.4. Zusammenfassung

Ausgangspunkt für den Prozess der anforderungsorientierten modellbasierten Testfallgenerierung sind Anforderungsdokumente, aus welchen sich einzelne funktionale Anforderungen extrahieren lassen, zu welchen exemplarische Szenarien angegeben werden können. Es sind zwei Phasen zu unterscheiden: Die formale Anforderungsspezifikation einerseits und die modellbasierte Testfallgenerierung andererseits. In der Phase der Anforderungsspezifikation werden einzelne funktionale Anforderungen identifiziert und dazu jeweils charakteristische Szenarien angegeben. Zur vollständigen Beschreibung des Verhaltens wird ein abstraktes Modell in Form einer Zustandsmaschine erstellt.

In der Phase der Testfallgenerierung werden die Szenarien in Klassen eingeteilt und jede Klasse entsprechend ihrer Priorität für den Test gewichtet. Die weiteren Schritte zur Testfallermittlung erfolgen nun automatisch: Die Klassifikation und Gewichtung der Szenarien wird in das Verhaltensmodell übertragen, und zu jedem Szenario wird ein anforderungsspezifisches Teilmodell extrahiert. Die Testfallgenerierung erfolgt schließlich für jedes Szenario aus dem jeweiligen Teilmodell. Die Phase der Testfallermittlung ist iterativ vorgesehen, nach der Generierung von Testfällen erfolgt ein manuelles Review der Testfälle und gegebenenfalls eine Anpassung der Gewichtung.

5. Anforderungsspezifikation und Modellerstellung

Eine automatisierte Ermittlung von Testfällen setzt eine formale Spezifikation des zu testenden Systems voraus, welches das wesentliche Eingabedokument der Testfallgenerierung ist. Da in der vorliegenden Arbeit der funktionale Test aus einer reinen Black-Box-Sicht erfolgt, ist eine Spezifikation erforderlich, welche allein die funktionalen Anforderungen an das System beschreibt, also das durch die Nutzer beziehungsweise die Systemumgebung wahrnehmbare Verhalten. Diese Anforderungsspezifikation unterscheidet sich damit deutlich von einer Spezifikation der (Software-)Implementierung, welche auch Vorgaben zur internen Struktur oder zur technischen Realisierung des zu entwickelnden Systems angibt. Da diese Vorgaben in dem funktionalen Black-Box-Test nicht geprüft werden, sollen diese auch nicht Bestandteil der für die Erstellung der funktionalen Testfälle verwendeten Anforderungsspezifikation sein.

Sofern es allein die Absicht ist, Eingabedaten für Testfälle zu erzeugen, über welche das Testurteil später manuell oder durch ein separates Orakel getroffen wird, reichen Angaben über die Eingabeschnittstelle oder über die Umwelt (etwa das erwartete Verhalten der Nutzer) aus. Sollen dagegen, wie es auch das Ziel in dieser Arbeit ist, vollständige Testfälle – also Folgen von Ein- und Ausgaben – erzeugt werden, ist eine formale Spezifikation des Systemverhaltens notwendig. Dieses *Modell* muss hinsichtlich des für die Testfallspezifikation gewählten Abstraktionsgrades *vollständig* sein.

Im Idealfall ist das Endprodukt der Anforderungsspezifikation bereits eine vollständige Beschreibung des Systems. *Vollständig* ist hier im Sinne einer *totalen* Funktion zu verstehen, wobei zu jeder möglichen Folge von (abstrakten) Eingaben aus der Spezifikation eine eindeutige Folge von Ausgaben bestimmt werden kann.[1] Sind solche vollständigen Systembeschreibungen nicht bereits das Ergebnis der Anforderungsspezifikation, sind sie zum Zweck der Testfallgenerierung eigens zu erstellen. Für die in dieser Arbeit vorgestellte Methodik ist es von nachrangiger Bedeutung, in welcher Prozessphase das Modell erstellt wird. Entscheidend für die Anwendbarkeit ist einerseits die Spezifikation von Anforderungsszenarien aus Einzelanforderungen und andererseits die Erstellung eines abstrakten Modells zur vollständigen Verhaltensspezifikation. Die Szenarien müssen dabei Abläufe des Modells sein.

Für diese Verhaltensspezifikation muss gelten, dass sie eine *abstrakte* Beschreibung des zu testenden (beziehungsweise des zu erstellenden) Systems darstellt. Der Ter-

[1] Wie in Kapitel 3 erwähnt, ist es unter anderem mit Szenarien nicht möglich, eine vollständige Anforderungsspezifikation anzugeben. Die Funktion kann aber zum Beispiel durch eine Zustandsmaschine, vgl. Abschnitt 5.3, vollständig angegeben werden.

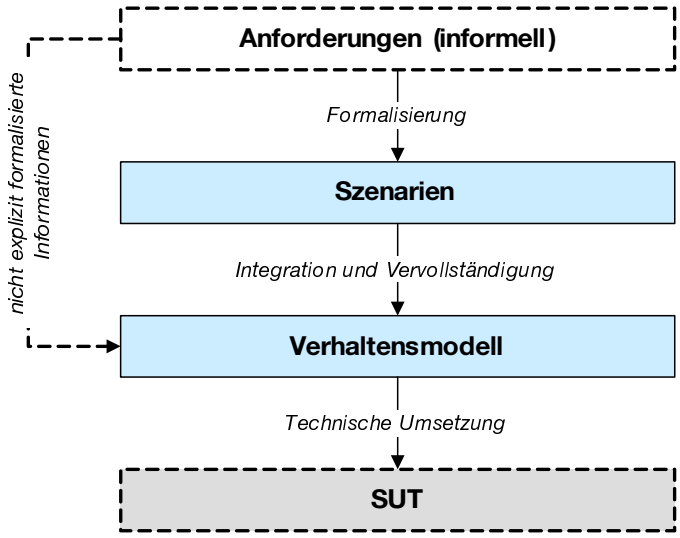

Abbildung 5.1.: Abstraktionsebenen der Spezifikation und Realisierung des zu testenden Systems *(SUT)*.

minus „abstrakt“ bedeutet hierbei, dass ausgehend von der Spezifikation zusätzliche Entwurfsentscheidungen bei der Realisierung des Systems zu treffen sind. Die konkrete Ausprägung des Systems wird also weiter von der Kreativität des Entwicklers beeinflusst und nicht etwa automatisch aus der Spezifikation erzeugt. Wie von Pretschner und Philipps (2005) dargestellt, würde ein gemeinsames Modell zur Code- und Testfallgenerierung die für den Test notwendige Redundanz vernachlässigen, allenfalls die Funktionsweise des Codegenerators oder Annahmen über die Umwelt können auf diese Weise überprüft werden. Das Ziel der vorliegenden Arbeit ist es jedoch, Fehler während der Entwicklung des betrachteten Systems, welche durch die zusätzlich notwendigen Entwurfsentscheidungen verursacht werden, zu identifizieren. Eine Abstraktion zwischen Spezifikation und Implementierung ist somit unabdingbar.

Das Vorgehen zur Spezifikation, wie es in diesem Kapitel beschrieben wird, entspricht in seinen wesentlichen Teilen jener von Wild u. a. (2006) vorgeschlagenen Vorgehensweise zur Spezifikation von Automotive Software über mehrere Abstraktionsebenen, wobei in dieser Arbeit der Übergang von der abstrakten Verhaltensbeschreibung zur Realisierung nicht in weitere Schritte zergliedert wird, da dies für die vorgestellte Methodik unerheblich ist. Eine Möglichkeit zur Unterstützung mehrerer Abstraktionsebenen wird von Pfaller u. a. (2006) aufzeigt. Anstelle des komplexeren Dienstbegriffs (Meisinger und Rittmann (2008) haben hierzu verschieden Ansätze der dienstbasierten Entwicklung verglichen) werden zudem allein Szenarien zur Spezifikation von einzelnen Anforderungen verwendet. Die in dieser Arbeit betrachteten Abstraktionsebenen sind in Abbildung 5.1 schematisch dargestellt. Die formale Spezifikation des erwarteten Systemverhaltens erfolgt damit in zwei Stufen:

1. Angabe von *charakteristischen Abläufen* mithilfe von *Szenarien*
2. Angabe des Verhaltens als *totale Funktion* mithilfe eines *Verhaltensmodells*

Im folgenden Abschnitt wird zunächst auf die Beschreibung von einzelnen funktionalen Anforderungen eingegangen. Abschnitt 5.2 gibt dann die formale Definition des Szenarienbegriffs an, wie er für die anforderungsorientierte Testfallgenerierung benötigt wird. In Abschnitt 5.3 wird schließlich der Übergang von Szenarien zu einem Verhaltensmodell dargestellt.

5.1. Spezifikation von Anforderungen

Viele Ansätze des Requirements Engineering, wie zum Beispiel von van Lamsweerde (2001) beschrieben, sehen zunächst die Identifikation von *Zielen* vor. Ein Ziel ist dabei die intentionale Beschreibung eines Merkmals des zu entwickelnden Systems (Pohl, 2007, S. 91). Bei Zielen steht die Intention der Stakeholder im Vordergrund, nicht die Spezifikation der konkreten Ausprägung eines Merkmals, wie es im System realisiert sein soll. Eine Ableitung von Testfällen für den funktionalen Test ist nicht unmittelbar möglich, da Ziele häufig sowohl funktionale als auch nicht-funktionale Anteile beschreiben.

Diese Arbeit behandelt allein den funktionalen Test reaktiver Systeme, wie er etwa auch von Pretschner (2003) beschrieben wurde. Somit sind ausschließlich funktionale Anforderungen Gegenstand der Überprüfung. Funktionale Anforderungen sind dabei Anforderungen, deren Erfüllung durch Stimulation des SUT mit einer Folge von Eingaben und Beobachtung der Ausgaben des SUT überprüft werden kann. Nicht-funktionale Anforderungen weiterer Qualitätsmerkmale – nach ISO 9126 (2001) Zuverlässigkeit, Benutzbarkeit, Effizienz, Änderbarkeit oder Übertragbarkeit – können durch den funktionalen Test im Allgemeinen nicht überprüft werden.

5.1.1. Informale Beschreibung funktionaler Anforderungen

Wie oben erwähnt, sind Ziele zu allgemeine Artefakte der Anforderungserhebung, als dass daraus unmittelbar funktionale Eigenschaften des zu entwickelnden Systems, welche durch Tests überprüft werden können, abgeleitet werden können. Für den anforderungsorientierten Test sind daher weiter detaillierte Anforderungsbeschreibungen eine notwendige Grundlage. Aus der Menge aller Ziele muss somit eine Menge von einzelnen funktionalen Anforderungen ermittelt werden. Jede einzelne Anforderung soll dabei zur Erreichung der Ziele beitragen. Für das Requirements Engineering wird vielfach vorgeschlagen, diese Konkretisierung von Zielen mittels sogenannter *Szenarien* oder *Use Cases* vorzunehmen. Unabhängig davon werden Anforderungen vielfach auch basierend auf Systemzuständen beschrieben.

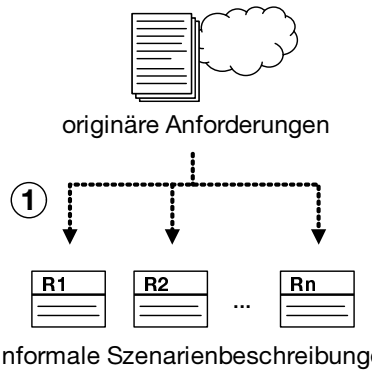

Abbildung 5.2.: Schritt 1 des vorgeschlagenen Entwicklungs- und Testprozesses: Identifikation einzelner Anforderungen (Ausschnitt aus Abbildung 4.1)

Dieser Abschnitt adressiert den ersten Schritt des vorgeschlagenen Entwicklungsprozesses, wie er in Abbildung 5.2 im Ausschnitt nochmals dargestellt ist. Ziel ist es, aus den Anforderungen der Stakeholder an das System, welche in Anforderungsdokumenten niedergeschrieben sein können, wie sie aber häufig auch nur als mentale Vorstellung vorliegen, zu informell beschriebenen Szenarien zu gelangen.

Zustandsbasierte Anforderungsbeschreibung

Eine zustandsbasierte Beschreibung von Anforderungen gibt Eigenschaften an das System nach folgendem Muster an:

> *„Falls sich das System in Zustand S befindet und die Eingabe i erfolgt, wechselt das System in Zustand S' und gibt die Ausgabe o aus.“*

Damit wird im Wesentlichen eine Zustandsübergangsrelation angegeben, welche das zu erstellende System erfüllen soll. Da in der Praxis einzelne Anforderungen häufig nicht vollständig dem obigen Muster genügen, werden meist implizite Annahmen zur Interpretation derartiger Formulierungen hinzugenommen:

- Ist der Ausgangszustand S nicht angegeben, gilt der Zustandsübergang in jedem Zustand

  > *„Falls die Eingabe i erfolgt, wechselt das System in Zustand S' und gibt die Ausgabe o aus.“*

- Falls der Folgezustand S' nicht angegeben ist, verharrt das System im Ausgangszustand S

  > *„Falls sich das System in Zustand S befindet und die Eingabe i erfolgt, gibt das System die Ausgabe o aus.“*

- Wird keine Ausgabe o angegeben, wird keine Ausgabe produziert

  „Falls sich das System in Zustand S befindet und die Eingabe i erfolgt, wechselt das System in Zustand S'."

Diese Art der Beschreibung ist allerdings mit einigen Nachteilen behaftet:

- Aus Sicht der Umwelt beziehungsweise der Nutzer des Systems ist der interne Zustand, in welchem sich das System befindet, in der Regel nicht von Bedeutung – meist ist der interne Zustand auch nicht sichtbar.
- Es bleibt damit unklar, ob die Annahmen über Zustände tatsächlich auf den allgemeinen Erwartungen der Nutzer beruht oder ob die Zustände nur im Rahmen der speziellen Ausprägung der Spezifikation eingeführt wurden.
- Werden die Zustandsübergänge nicht vollständig spezifiziert, bleibt unklar, ob eine Unterspezifikation vorliegt oder ob die Interpretation wie zuvor genannt zu vervollständigen ist.
- Da der interne Zustand meist von außen nicht sichtbar ist, ist eine Überprüfung dieser Anforderungen durch den Test kaum möglich – bei der Testausführung kann der geforderte Folgezustand im Allgemeinen nicht überprüft werden.

Besonders der zuletzt genannte Einwand zeigt, dass eine derartige Formulierung von Anforderungen oft ungeeignet ist, um gezielt Testfälle hinsichtlich der Anforderungen zu erstellen. Auch die Problematik, dass unklar ist, ob der interne Systemzustand tatsächlich eine Forderung der Nutzer oder aber nur Hilfsmittel zur Spezifikation ist, erschwert die Ableitung von Testfällen.

Aus diesen Gründen ist hinsichtlich des Systemtests eine Spezifikation von Anforderungen vorzuziehen, welche allein auf der beobachtbaren Kommunikation an der Schnittstelle zwischen System und Umwelt beruht. Dies führt auf die schon häufiger erwähnte Beschreibung von Szenarien. Im folgenden Abschnitt wird darauf näher eingegangen.

5.1.2. Kommunikation zwischen System und Umwelt

In der Regel interagiert ein System mit unterschiedlichen Kommunikationspartnern wie dies in Abbildung 5.3 dargestellt ist. Das System kann beispielsweise Eingaben von menschlichen Nutzern (1) und anderen Steuerungskomponenten (2) entgegennehmen, sowie den Kontext der Umgebung durch passive Sensoren (3), etwa durch einen Lichtsensor, erfassen. Die Ausgaben des Systems werden wiederum von menschlichen Nutzern (4) wahrgenommen, wobei diese nicht notwendigerweise auch Eingaben an das System liefern (5) weiterhin können die Ausgaben auch von anderen Komponenten als Eingaben weiterverarbeitet werden. Alle diese Kommunikationspartner werden im

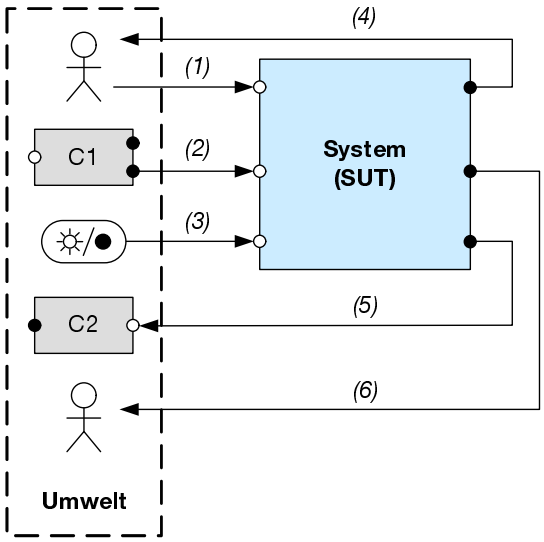

Abbildung 5.3.: Unterschiedliche Kommunikationspartner des zu testenden Systems (SUT)

Rahmen der vorliegenden Arbeit als *Nutzer* des (zu testenden) Systems bezeichnet.[2] Die Gesamtheit der Nutzer wird als *Umwelt* des SUT bezeichnet.

Die Nutzer nehmen das System nur als *Black-Box* wahr, die internen Abläufe und Zustände innerhalb des SUT bleiben ihnen, wie bereits oben erläutert, verborgen.[3] Daraus folgt, dass eine auf internen Zuständen basierende Beschreibung von Anforderungen, ebenso wie derartige Testfälle, die Sicht der Nutzer auf das System nicht unmittelbar widerspiegeln. Aus diesem Grund wird im Rahmen dieser Arbeit von einer strikten Black-Box-Sicht auf das System ausgegangen. Allein das durch Beobachtung von Ein- und Ausgaben sichtbare Verhalten ist Grundlage für die eingeführte Methode.

Wie eben erläutert, interagiert das System üblicherweise mit unterschiedlichen Nutzern. Im Rahmen dieser Arbeit wird im Weiteren nicht näher auf diese Unterscheidung eingegangen, sondern die Umwelt in ihrer Gesamtheit als einziger Kommunikationspartner des zu testenden Systems angesehen. Diese Vereinfachung ist zulässig, da von synchronen reaktiven System mit einem einheitlichen Takt ausgegangen wird: In jedem Takt werden alle Eingabeports gelesen und auf alle Ausgabeports geschrieben. Zudem erfordert die vorgestellte Methode nur die Spezifikation beziehungsweise Modellierung

[2] Es ist zu beachten, dass nach dieser Definition einem Kommunikationspartner diese Eigenschaft als *Nutzer* des Systems nicht bewusst sein muss.

[3] Oft ist es den Nutzern zwar möglich, Annahmen über den Zustand des Systems zu treffen. Beispielsweise kann nach Betätigung des Einschalters das Systems angenommen werden, dass es sich im Zustand „eingeschaltet“ befindet. Der Nutzer hat dabei jedoch keine Garantie, dass das System tatsächlich in diesen Zustand gewechselt ist (es sei denn, dies wäre wiederum an der Systemschnittstelle sichtbar).

des zu testenden Systems und setzt keine weitere Angaben über die Umwelt voraus, wie es beispielsweise beim Testen auf Basis von Nutzungsmodellen nötig wäre.

5.1.3. Szenarien als charakteristische Abläufe

Da sich der anforderungsorientierte Funktionstest auf die Beobachtung von Ein- und Ausgaben beschränkt, stellen Aussagen über Folgen von Ein- und Ausgaben ein geeignetes Mittel sowohl zur Beschreibung von Anforderungen als auch von Testfällen dar. Derartige *Nutzungsszenarien* werden für das Requirements Engineering vielfach zur Verfeinerung von Zielen vorgeschlagen. Im Gegensatz zur zustandsbasierten Beschreibung stehen hier nicht interne Zustände des Systems im Vordergrund, sondern allein das beobachtbare Verhalten an der Schnittstelle des Systems. In den Anforderungsdokumenten werden dementsprechend Szenarienbeschreibungen identifiziert, aus welchen sich Szenarien nach folgendem allgemeinen Muster ableiten lassen:

> *„Falls an das System die Eingaben $i_1, i_2, \ldots, i_n$ übergeben werden erfolgen die Ausgaben $o_1, o_2, \ldots, o_n$“*

In Beispiel 8 sind typische Szenarienbeschreibungen angegeben. Anhand dieses Fensterheber-Beispiels wird die Methode auch im Weiteren erläutert.

Beispiel 8 (Szenarienartige Anforderungsbeschreibung)

Von Houdek und Paech (2002) wurde ein beispielhaftes Lastenheft eines Türsteuergerätes im Automobil veröffentlicht, welches unter Anderem die Funktionalität eines elektrischen Fensterhebers beschreibt. Für die Basisfunktion eines Fensterhebers in einem Automobil sind darin folgende Anforderungen gegeben:

- *Ist die relevante Schalterstellung gleich* Fenster runter man. *oder ist die relevante CAN-Botschaft WIN_x_OP=01, so wird die Scheibe nach unten bewegt. Die Bewegung endet, wenn*
  - *das entsprechende Signal nicht mehr anliegt (bzw. nicht mehr gesendet wird),*
  - *oder sich die Scheibe in der unteren Position befindet (d.h. F_UNTEN bzw. FF_UNTEN),*
  - *oder ein anderer Befehl zum Bewegen dieser Scheibe zu einem späteren Zeitpunkt eingeht; in diesem Fall wird der neue Bewegungsbefehl bearbeitet,*
  - *oder der Scheibenbewegungssensor (F_BEWEG bzw. FF_BEWEG) keine Signale sendet, obwohl der Scheibenmotor angesteuert wird und sich die Scheibe noch nicht in der unteren Position befindet; in diesem Fall wird die Botschaft ERROR_WIN = 1 gesendet und der Fehlercode 0x35 in den Fehlerspeicher eingetragen*

  - *oder die Ansteuerung länger als 3 sec. dauert, ohne dass erkannt wird, dass sich die Scheibe in der unteren Position befindet; in diesem Fall wird die Botschaft ERROR_WIN = 1 gesendet und der Fehlercode 0x35 in den Fehlerspeicher eingetragen.*

- *Ist die relevante Schalterstellung gleich* Fenster hoch man. *oder ist die relevante CAN-Botschaft WIN_x_CL=01, so wird die Scheibe nach oben bewegt. Die Bewegung endet, wenn*
  - *das entsprechende Signal nicht mehr anliegt (bzw. nicht mehr gesendet wird),*
  - *oder sich die Scheibe in der oberen Position befindet (d.h. F_OBEN bzw. FF_OBEN),*
  - *oder ein anderer Befehl zum Bewegen dieser Scheibe zu einem späteren Zeitpunkt eingeht; in diesem Fall wird der neue Bewegungsbefehl bearbeitet,*
  - *oder der Scheibenbewegungssensor (F_BEWEG bzw. FF_BEWEG) keine Signale sendet, obwohl der Scheibenmotor angesteuert wird und sich die Scheibe noch nicht in der oberen Position befindet; in diesem Fall wird der Einklemmschutz aktiviert,*
  - *oder die Ansteuerung länger als 3 sec. dauert, ohne dass erkannt wird, dass sich die Scheibe in der oberen Position befindet; in diesem Fall wird Botschaft ERROR_WIN = 1 gesendet und der Fehlercode 0x35 in den Fehlerspeicher eingetragen*

Anstelle einer Angabe von konkreten beispielhaften Abläufen, sogenannten *Instanz-Szenarien*, können szenarienartige Beschreibungen allgemeiner gefasst werden, indem *Annahmen* über Eigenschaften der Eingabe i getroffen werden und Eigenschaften über die Ausgabe o angegeben werden, welche, falls die Eingabe i diese Eigenschaft erfüllt, von der Ausgabe o *garantiert* werden müssen. Eine allgemeine Formulierung derartiger *Assumption/Guarantee* Spezifikationen, siehe dazu auch (Broy und Stølen, 2001), ist etwa

> *„Falls an das System eine Eingabe i übergeben wird und* $\mathsf{asm}(i)$ *gilt muss die vom System produzierte Ausgabe o die Eigenschaft* $\mathsf{gar}(o)$ *erfüllen“*

Damit wird nicht eine konkrete Folge von Ein- und Ausgaben beschrieben sondern eine Menge von Ein- und Ausgaben, mit bestimmten, durch die Annahme und die Garantie spezifizierten, Charakteristika. Allgemein führt dies dazu, eine Anforderung als ein Prädikat über der Menge von Funktionen zu beschreiben, wie dies bereits in Kapitel 3, Abschnitt 3.3 dargestellt wurde. Eine weitere Möglichkeit, jeweils Mengen von Abläufen zu charakterisieren, ist die Verwendung von temporalen Logiken (Clarke Jr. u. a., 1999). Eine derartige Spezifikation von *Typ-Szenarien* bietet den großen Vorteil, eine wesentlich umfassendere Beschreibung des gewünschten Gesamt-Systemverhaltens spezifizieren zu können, als dies mit Instanz-Szenarien möglich ist. Hinsichtlich der in dieser Arbeit beschriebenen Methode zum anforderungsorientierten Softwaretest zeigen sich allerdings Nachteile:

Handhabung Die Formulierung von Typ-Szenarien ist nicht in jedem Fall intuitiv zu bewerkstelligen. Mitunter ist etwa die präzise Formulierung einer A/G-Spezifikation zu einer gegebenen informellen Anforderungsbeschreibung nur durch komplexe formale Ausdrücke möglich.

Validierung durch Stakeholder Aus dem zuvor genannten Aspekt der wenig intuitiven Handhabung folgt, dass eine Validierung des formalen Anforderungsszenarios durch die betroffenen Stakeholder in vielen Fällen nicht möglich ist, da Auftraggeber beziehungsweise Anwender eines Systems häufig über keine Erfahrung im Umgang mit formalen Spezifikationen verfügen.

Erkennen von Inkonsistenzen Möglicherweise enthalten die verschiedenen Szenarienspezifikationen Widersprüche. Dies ist auch bei Instanzszenarien nicht auszuschließen. Allerdings ist das Erkennen von Inkonsistenzen auf der Ebene von voll instanziierten Abläufen ungleich einfacher zu erreichen: Die einzige Form einer möglichen Inkonsistenz ist, dass gleiche Sequenzen von Eingaben zu unterschiedlichen Ausgaben führen. Da das Ziel eine vollständige Beschreibung des Systemverhaltens im Sinne einer totalen Funktion ist, also deterministisch ist, liegt ein Widerspruch vor, der aufgelöst werden muss.

Verhaltensmodell zur vollständigen Spezifikation Im Kontext der vorliegenden Arbeit ist zu beachten, dass die Szenarienspezifikation noch keine vollständige Spezifikation des Systemverhaltens darstellt. Die vollständige Verhaltensspezifikation wird durch das Systemmodell (siehe Abschnitt 5.3) erzielt.

Aus diesen Gründen, insbesondere der Notwendigkeit, dass Szenarien durch die unterschiedlichen Stakeholder validierbar sein müssen, wurden für die in dieser Arbeit vorgestellte Testmethodik voll instanziierte Szenarien, also konkrete Beispielabläufe gewählt.

Für die Wahl der Szenarien ist es unabdingbar, dass diese *charakteristische* Repräsentanten für die einzelnen funktionalen Anforderungen an das System sind. Das bedeutet, dass Anforderungen derart atomar sein müssen, dass ihr wesentlicher Inhalt durch konkrete Beispielabläufe vermittelt werden kann. Oftmals werden zu einer funktionalen Anforderung unterschiedliche Ausprägungen bestehen, dann lässt sich gegebenenfalls ein einziges charakteristisches Szenario nur schwer bestimmen. In diesen Situationen ist die Anforderung weiter in atomare Anforderungen zu verfeinern. Diese Vorgabe trägt zur Qualitätssicherung der Anforderungen bei: Kann für eine Anforderung kein charakteristisches Szenario angeben werden, ist sie weiter zu verfeinern und/oder zu präzisieren.

Allerdings ergeben sich durch die Angabe von Instanzszenarien Einschränkungen bezüglich der Ausdrucksmächtigkeit von Anforderungen. Insbesondere können universelle Eigenschaften und Negativabläufe nur eingeschränkt spezifiziert werden. Die folgenden beiden Abschnitte erläutern, weshalb diese Einschränkungen für den anforderungsorientierten Test nicht von großer Bedeutung sind.

Universelle Eigenschaften / Safety Properties

Häufig werden an das Verhalten eines Systems Anforderungen der Art

> *„Immer wenn Ereignis A eintritt, hat das System mit Ausgabe B zu reagieren.“*

gestellt. Solche Eigenschaften betreffen im Allgemeinen eine unendliche Teilmenge aller potentiell möglichen Systemabläufe. Ein erschöpfendes Testen ist daher, unabhängig von weiteren Einschränkungen der praktischen Testdurchführung, nicht möglich. Insofern stellen aus Sicht des realisierten Systems viele der Einzelanforderungen derartige „universelle Eigenschaften“ dar. Vor dem Hintergrund des anforderungsorientierten Softwaretest ist die Unterscheidung, ob es sich um eine universelle Eigenschaft handelt oder nicht, aus Sicht der Stakeholder zu treffen: Steht für die Stakeholder (insbesondere für Anwender oder Auftraggeber) die Umsetzung der Funktionalität („auf Ereignis A folgt Ausgabe B“) oder die Beschränkung in der Allgemeinheit („immer“) im Vordergrund? Letzteres ist hier aus Stakeholder-Sicht als *universelle Eigenschaft* zu verstehen. Eine pragmatische Methode, diese Klassifizierung vorzunehmen, ist die Formulierung von Szenarien: Für eine im Vordergrund funktionale Anforderung ist es im Allgemeinen möglich, ein charakteristisches Ein-/Ausgabe-Szenario anzugeben. Universelle Eigenschaften können dagegen nicht durch einen einzelnen oder wenige Beispielabläufe charakterisiert werden – die im Fokus stehende Forderung nach der Allgemeingültigkeit lässt sich nicht durch Szenarien zufriedenstellend ausdrücken.

Diese universellen Eigenschaften, welche häufig Bestandteil von Anforderungen sind, sind durch Tests nur unzureichend überprüfbar. Wesentliches Ziel eines anforderungsorientierten Tests ist es, zu einer Anforderung eine Menge von Testfällen anzugeben, welche das Vertrauen erhöhen, dass diese Anforderung von dem System korrekt erfüllt wird. Obwohl meist direkt Testfälle angeben werden können, welche die geforderten funktionalen Bestandteile einer universellen Eigenschaft überprüfen ist, damit die im Vordergrund stehende Forderung der Universalität nur unbefriedigend gezeigt.

Für die Überprüfung von universellen Eigenschaften sind daher Techniken der formalen Verifikation vorzuziehen. Da der funktionale Anteil in universellen Eigenschaften meist eine geringe Komplexität aufweist, lassen sie sich mit vertretbarem Aufwand etwa als temporallogische Formeln darstellen. Das später in Abschnitt 5.3 eingeführte

Verhaltensmodell – gegebenenfalls auch ein detailliertes Modell der Implementierung – erlaubt die formale Verifikation dieser Eigenschaften, zum Beispiel durch Modellprüfung (Clarke Jr. u. a., 1999).

Negative Anforderungen

Häufig werden in der Anforderungsanalyse auch *negative* Eigenschaften formuliert. Die Bezeichnungen „Negativanforderung“, „Negativeigenschaft“, „Negativszenario“ oder dergleichen werden dabei in unterschiedlicher Weise gebraucht, welche sich hinsichtlich der Relevanz für die Testaktivitäten stark unterscheidet.

Negierte universelle Eigenschaften Zum einen können universelle Eigenschaften, welche beim Erfüllen einer Vorbedingung immer, also in allen Ausführungspfaden nach Eintritt der Vorbedingung, eine bestimmtes Verhalten des Systems ausschließen, als „Negativszenarien“ verstanden werden. Dies sind Anforderungen beispielsweise der Form

*„wenn Ereignis A eintritt, darf das System **nie** mit Ausgabe C reagieren.“*

Für derartige Anforderungen gilt gleiches wie im vorhergehenden Abschnitt beschrieben.

Nicht vorgesehene Eingaben Als „Negativszenarien“ werden oft auch Anwendungsfälle beschreiben, deren Eingaben nicht vorgesehen sind (obwohl sie syntaktisch zulässig sind). Im Regelfall liegt aber eine Spezifikation vor, welche den Umgang des Systems mit derartigen ungültigen Eingaben beschreibt, beispielsweise dass das System in solchen Fällen in einen Fehlerzustand wechselt oder die Eingabe ignoriert. Für den Softwaretest unterscheiden sich diese Anforderungen nicht von anderen funktionalen Anforderungen, da in diesen Fällen die korrekte Reaktion des Systems ebenso bestimmbar ist. Einen Sonderfall stellen Systeme dar, bei welchen im Falle ungültiger Eingaben beliebige Systemreaktionen zulässig sind. In dieser Situation ist ein Test derartiger Anforderungen nicht notwendig, da jede Ausgabe des Systems korrekt ist.[4]

Zuletzt existieren auch Beschreibungen von Funktionalitäten, welche vom System nicht umgesetzt werden müssen, also dementsprechend keine Anforderung sind. In der Phase des Requirements Engineering ist dies hilfreich, um die umzusetzende Funktion von nicht benötigten Funktionalitäten abzugrenzen. Solche Anforderungsbeschreibungen können auch als „negative Anfoferderungen“ verstanden werden.

[4] Allerdings ist zu beachten, dass dies zu einer nicht-deterministischen Spezifikation führt.

5.1.4. Ausgangszustand und Vorbedingungen

Werden Anforderungen an zustandsbehaftete Systeme durch Ein-/Ausgabeszenarien beschrieben, muss der Zustand, welcher vor Ausführung dieses Szenario im System vorliegt, bekannt sein. Ein Zustand des Systems ist eindeutig bestimmbar, falls ein Initialzustand des Systems bekannt ist und eine Folge von Eingaben, welche ausgehend von diesem Initialzustand in den aktuellen Zustand geführt hat.

Um Szenarien eindeutig zu formulieren, würde es ausreichen, nur den Initialzustand zu definieren und jedes Szenario ausgehend von diesem Zustand als Folge von Ein- und Ausgaben zu beschreiben. Für die Formulierung von Anforderungen, genauer für die Angabe eines charakteristischen Szenarios zu einer Anforderung, ist es nicht immer praktikabel, die vollständige Folge von Ein- und Ausgaben zu beschreiben. Häufig existieren Teilmengen der Anforderungen an ein System, wobei allen Anforderungen in einer Teilmenge gemeinsam ist, dass sie bis zu einem bestimmten Schritt dieselbe Ausführungsfolge beschreiben und die Einzelanforderungen erst ab diesem Punkt unterscheiden. Die gemeinsame Ausführungsfolge, beginnend im Initialzustand, wird im Folgenden als *Vorbedingung* bezeichnet.

Zum Beispiel können für die Steuerung eines Fensterhebers unterschiedliche Anforderungen zum Schließen des Fensters vorliegen, wobei aber allen Anforderung gemeinsam ist, dass das Fenster zunächst geöffnet worden ist. Eine Folge von Eingaben, welche das Fenster in den offenen Zustand versetzt, wäre eine derartige Vorbedingung.

In diesen Fällen ist die vollständige Spezifikation eines Ausführungspfades, beginnend im Initialzustand, aus folgenden Gründen unzureichend:

Redundante Information Die unterschiedlichen Anforderungsszenarien würden jeweils die gleiche Vorbedingung beschreiben, im Sinne der Vermeidung von Redundanzen ist dies nicht wünschenswert

Fehlende Fokussierung auf charakteristisches Merkmal Szenarien sollen charakteristisch für die jeweilige informell beschriebenen Anforderung sein. Sie müssen geeignet sein, die Motivation für diese Anforderung und die Intention des Stakeholders prägnant wiederzugeben. Diese Prägnanz geht verloren, falls das Anforderungsszenario in der Vorbedingung einen großen Teil von Aktionen beschreibt, welche nicht im Vordergrund für diese Anforderung stehen.

Unterschiedliche Gewichtung Dadurch, dass die Vorbedingungen Bestandteil vieler weiterer Anforderungen sind, würden sie im Vergleich zu Anforderungen welche nicht als Vorbedingung vorkommen, eine nicht zu rechtfertigende höhere Gewichtung erhalten.

Aus diesem Grund ist es sinnvoll, Anforderungsszenarien in zwei Teile zu zerlegen:

- Eine *Vorbedingung*, welche eine Ein-/Ausgabefolge beginnend im Initialzustand des Systems darstellt. Diese Folge ist dabei nicht motivierend, um die jeweilige Anforderung einzuführen.
- Einen *bestimmenden Teil*, welcher durch eine Folge von Ein- und Ausgaben die Intention der Anforderung charakteristisch repräsentiert. Der bestimmende Teil adressiert die Motivation der Stakeholder, diese Anforderung einzuführen.

Vorbedingungen – wie im zuvor genannten Beispiel das Öffnen des Fensters – sind im Regelfall selbst wieder eigenständige Einzelanforderungen. Anstelle der erneuten vollständigen Spezifikation eines Szenarios beginnend im Initialzustand reicht es aus, andere Anforderungen als Vorbedingung zu referenzieren. Damit wird einerseits das Szenario auf den charakteristischen Anteil der Anforderung reduziert, andererseits kann das Szenario zu einem vollständigen Pfad, beginnend im Initialzustand expandiert werden und ist so präzise definiert.

Oft existieren mehrere Anforderungen, welche als Vorbedingung für eine weitere Anforderung gelten können. Eine Vorbedingung für eine Anforderung r_{neu} ist somit allgemein eine Menge von Anforderungen $P = \{r_0 \dots r_n\}$ wobei die Anforderung r_{neu} unter jeder Vorbedingung $r_i \in P$ gültig sein muss.

Wird der Menge P von Anforderungen, welche eine Vorbedingung darstellt, ein Name zugewiesen führt dies zu einem Zustandsbegriff. Aus Sicht des Nutzers befindet sich das System im Zustand P, falls ein Szenario $r_i \in P$ ausgehend vom Initialzustand ausgeführt wurde. Dieser Zustandsbegriff wird fundamental anders verwendet als jene Zustände, wie sie in Abschnitt 5.1.1 erwähnt wurden. Dort handelt es sich um interne Systemzustände, hier um Zustände aus Sicht des Nutzers.

In diesem Abschnitt wurde die Verwendung von Szenarien als charakteristische Folgen von Ein- und Ausgaben zur formalen Anforderungsspezifikation motiviert und der Szenariobegriff informell eingeführt. Im Folgenden wird eine formale Definition dazu angegeben.

5.2. Formale Darstellung von Anforderungen

Um Anforderungen in der modellbasierten Testfallgenerierung verwenden zu können, ist eine formale Darstellung dieser Anforderungen an das zu testende System notwendig. Informell gegebene Anforderungen können kaum maschinell weiterverarbeitet werden und reichen zur Unterstützung einer automatisierten Testfallermittlung nicht aus. Gleichwohl ist eine einfache, möglichst intuitiv verständliche Sprache zur Formalisierung von Anforderung hilfreich, um die Validierung der formulierten Anforderungen durch möglichst viele der beteiligten Stakeholder zu ermöglichen, insbesondere durch Anwender oder Auftraggeber.

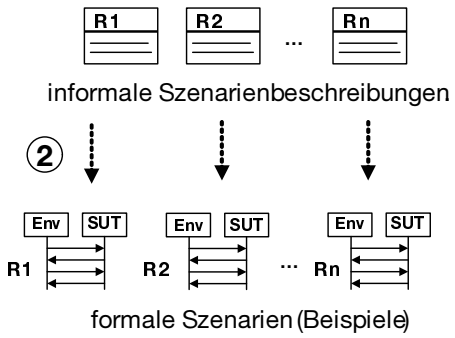

Abbildung 5.4.: Schritt 2 des vorgeschlagenen Entwicklungs- und Testprozesses: Formale Spezifikation von Szenarien (Ausschnitt aus Abbildung 4.1)

Wurden im vorhergehenden Abschnitt informelle Szenarien eingeführt, befasst sich die Arbeit nun mit der Ermittlung von formalen Repräsentanten der Szenarien. Dies ist der zweite Schritt des Entwicklungsprozesses aus Kapitel 4. Siehe dazu auch Abbildung 5.4. Die formale Spezifikation von Szenarien erfolgt manuell, da die Auswahl geeigneter Repräsentanten im Allgemeinen ein kreativer Prozess ist. Gleichwohl wird von Kof (2005, 2007a,b, 2008) eine Unterstützung dieses Vorgehens durch automatisierte Textanalyse mithilfe von Techniken der Computerlinguistik vorgeschlagen. In dieser Arbeit werden Szenarien als Paare von Ein- und Ausgabe an das zu testende System verwendet. Diese können aus komplexeren Notationen zur Spezifikation von Szenarien wie Message Sequence Charts abgeleitet werden. Dazu sei an dieser Stelle auf Abschnitt 5.2.5 verwiesen. Wie auch von Broy (2005) angeführt, eignen sich Szenarien nicht zur vollständigen Spezifikation eines Systems, dementsprechend sind Szenarien nur als Vorstufe zu einer vollständigen Spezifikation des Verhaltens zu verstehen. Die vollständige Verhaltensspezifikation wird durch einen Zustandsautomaten angegeben. Dies wird in Abschnitt 5.3 erläutert.

5.2.1. Szenarien als Paare von Eingabe und Ausgabe

Im Folgenden wird das Konzept der *Szenarien* eingeführt, wie es im Rahmen dieser Arbeit als Grundlage für die Formalisierung von Anforderungen dient. Glinz (2000) definiert ein Szenario wie folgt:

Definition 12 ((Kommunikations-)Szenario, allgemein) *Ein Szenario ist eine geordnete Menge von Interaktionen zwischen Kommunikationspartnern, üblicherweise zwischen einem System und einer Menge von externen Aktoren. Es kann eine konkrete Abfolge von Interaktionsschritten (Instanz-Szenario) oder eine Menge von möglichen Interaktionsschritten (Typ-Szenario) umfassen.*

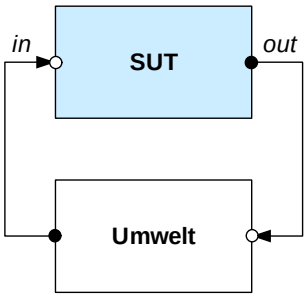

Abbildung 5.5.: Kommunikationspartner für Szenarien: Das zu testende System *(SUT)* kommuniziert über einen Eingabekanal und einen Ausgabekanal mit der Umwelt.

Entsprechend dieser Definition finden Szenarien im Rahmen dieser Arbeit in ihrer Ausprägung als Instanz-Szenarien Verwendung, wobei nur zwei Kommunikationspartner betrachtet werden: Das zu testende System *(system under test, SUT)* und die Umwelt, mit der dieses interagiert. Dies entspricht einer Black-Box-Sichtweise, welche auch die Sichtweise der Gesamtheit der Nutzer des Systems ist. Allein das beobachtbare Verhalten des Systems an seiner Schnittstelle ist von Bedeutung ,wie es in Abbildung 5.5 dargestellt ist. Ein Interaktionsschritt in einem Szenario soll als *Aktion* bezeichnet werden, wobei zwischen *Eingabeaktionen* und *Ausgabeaktionen* unterschieden wird.

In diesem Modell verfügt das SUT nur über einen Eingabe- und einen Ausgabekanal. Allgemein ist jedoch zu erwarten, dass ein System mehrere Verbindungen zu externen Kommunikationspartnern besitzt, wie es in Abbildung 5.6 oben skizziert ist. Dabei beschreiben die Aktionen $in_i \in I_i$ für $0 < i \leq m$ eine Eingabe des i-ten Eingabekanals und die Aktionen $out_j \in O_j$ für $0 < j \leq n$ eine Ausgabe des j-ten Ausgabekanals. Mit I_i und O_j wird das Alphabet aller möglichen Eingabe- beziehungsweise Ausgabeaktionen an einem Kanal bezeichnet.

Werden, wie in Abbildung 5.6 unten dargestellt, Tupel von Ein- beziehungsweise Ausgaben gebildet, lässt sich diese Situation auf einen Ein- und einen Ausgabekanal zurückführen. Diese Abbildung ist zulässig, da im Rahmen dieser Arbeit nur Systeme betrachtet werden, welche über einen diskreten und synchronen Takt gesteuert werden: Alle Eingabekanäle werden in jedem Taktschritt gelesen. Analog wird in jedem Schritt auf alle Ausgabekanäle geschrieben.

Allgemein sind damit die Alphabete der Ein- und Ausgaben als

$$I \stackrel{\text{def}}{=} I_1 \times I_2 \times \ldots \times I_m$$
$$O \stackrel{\text{def}}{=} O_1 \times O_2 \times \ldots \times O_n$$

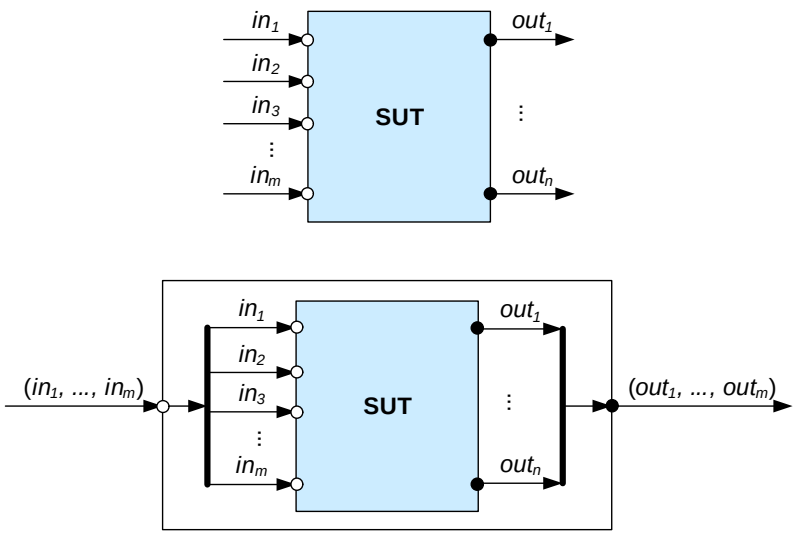

Abbildung 5.6.: Verallgemeinerung der Schnittstelle des SUT: Mehre Ein- beziehungsweise Ausgabekanäle lassen sich zu einem Ein- und einem Ausgabekanal abstrahieren, in dem Aktionen der Eingaben und Ausgaben zu Tupeln zusammengefasst werden.

definiert, falls das SUT über m Eingabekanäle und n Ausgabekanäle verfügt. Entsprechend gilt für eine Eingabeaktion $in \in I$ und eine Ausgabeaktion $out \in O$

$$in = (in_1, \ldots, in_m)$$
$$out = (out_1, \ldots, out_n)$$

mit $in_i \in I_i$ und $out_j \in O_j$ für $0 < i \leq m, 0 < j \leq n$.

Es ist somit auch für Systeme, welche über mehrere Ein- oder Ausgabekanäle zu Kommunikationspartnern verbunden sind, möglich, ihre Ein- und Ausgaben über ein einziges Alphabet von Eingabeaktionen und ein einziges Alphabet von Ausgabeaktionen zu beschreiben. Die Beschreibung von Szenarien erfolgt deshalb im Folgenden allein auf Basis der Mengen I und O von Ein- beziehungsweise Ausgabeaktionen. Die Mengen der Eingabeaktionen I und Ausgabeaktionen O geben die syntaktische Schnittstelle $(I \triangleright O)$ entsprechend Definition 1 (Seite 41) des Systems an.

Es wird hier im Allgemeinen eine Spezifikation von vollständigen Szenarien angenommen, auf die häufig praktikablere Verwendung partieller Szenarien wird in Abschnitt 5.2.6 eingegangen.

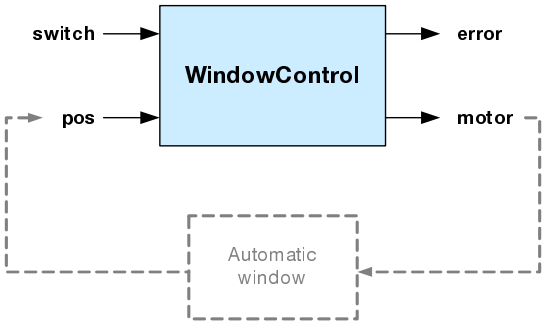

Abbildung 5.7.: Beispiel: Syntaktische Schnittstelle eines Fensterhebers.

Beispiel 9 (Syntaktische Schnittstelle)

Abbildung 5.7 zeigt schematisch die syntaktische Schnittstelle einer Steuerung für einen Fensterheber. Die Datentypen für die einzelnen Kanäle *switch, pos, motor* und *err* seien dabei mit

$$\begin{aligned} SWITCH &\stackrel{\text{def}}{=} \{open, close, \bot\} \\ POS &\stackrel{\text{def}}{=} \{top, bottom, middle, moving\} \\ MOTOR &\stackrel{\text{def}}{=} \{down, up, stop\} \\ ERROR &\stackrel{\text{def}}{=} \{err, \bot\} \end{aligned}$$

definiert.

- *SWITCH* gibt die Aktionen an, welche der Benutzer am Bedienelement des Fensterhebers durchführen kann, $\bot$ steht dabei für die Nullstellung des Schalters.
- *POS* ist eine Rückmeldung vom Fenster, welches sich in den Extrempositionen oben (*top*) und unten (*bottom*) oder dazwischen (*middle*) befinden, sowie in Bewegung (*moving*) sein kann.[5]
- *MOTOR* dient zur Steuerung des Fensterhebermotors über die Aktionen *down*, *up* und *stop* durch die Fensterhebersteuerung.
- *ERROR* gibt an, ob eine eine Fehlsituation vorliegt. Dies wird über die Nachricht *err* signalisiert. Im Normalfall liegt keine Nachricht ($\bot$) an.

SWITCH und *POS* sind Eingaben, *MOTOR* und *ERROR* die Ausgaben des Systems. Damit lautet die syntaktische Schnittstelle der Fensterhebersteuerung

$$((SWITCH \times POS) \triangleright (MOTOR \times ERROR)).$$

[5]Insbesondere *POS* verdeutlicht hier auch die gewählte Abstraktion der technischen Schnittstelle (z. B. Sensorwert) auf eine für die Szenarienbeschreibung und Testspezifikation relevante Menge von Aktionen.

Analog zu Broy u. a. (1992) sei für eine Menge Act von Aktionen die Menge der endlichen Ströme über diese Aktionen mit Act^* angegeben. Die in diesem Kapitel verwendete Notation von Strömen entspricht jener in (Broy und Stølen, 2001), siehe dazu auch Anhang H. Ein Szenario über die Eingaben I und Ausgaben O kann damit formal als ein Paar von endlichen Strömen über diese Aktionsmengen angegeben werden:

Definition 13 (Szenario, formal) *Ein Szenario r auf Basis der syntaktischen Schnittstelle $(I \triangleright O)$ ist ein Paar $(ins, outs)$ von Eingaben $ins \in I^*$ und dazu korrespondierenden Ausgaben $outs \in O^*$ mit $\#ins = \#outs$.*

Ein Szenario ist somit ein Folge von Eingaben an das System mit dazu korrespondierenden Ausgaben, wobei durch $\#ins = \#outs$ gefordert wird, dass beide Folgen dieselbe Anzahl Aktionen umfassen. Ein Szenario ist derart zu interpretieren, dass nach der i-ten Eingabeaktion aus ins die zu erwartende Ausgabe des System durch die i-te Ausgabeaktion in $outs$ beschrieben wird.

Für den Zugriff auf die Ein- und Ausgaben eines Szenarios werden entsprechende Hilfsfunktionen

$$\mathsf{in} \in I^* \times O^* \to I^*$$
$$\mathsf{out} \in I^* \times O^* \to O^*$$

eingeführt, wobei für ein Szenario $r = (ins, outs) \in I^* \times O^*$

$$\mathsf{in}.r = ins$$
$$\mathsf{out}.r = outs$$

gilt.

Durch diese formale Definition lassen sich Szenarien, wie sie zuvor informal in Abschnitt 5.1.3 beschrieben wurden, darstellen. Pro informal festgelegter Einzelanforderung ist dabei zumindest ein charakteristisches Szenario zu wählen. Da die informale Beschreibung einer Anforderung oft mehrere, teilweise stark unterschiedliche, Abläufe zulässt, kann es sinnvoll sein, mehrere Szenarien zu einer Anforderung zu wählen – allerdings ist anzunehmen, dass dann auch die informale Beschreibung de facto mehrere unterschiedliche Anforderungen umfasst.[6] In dieser Arbeit wird deshalb zur Vereinfachung im Weiteren eine 1 : 1-Beziehung zwischen Anforderung und Szenario angenommen. Zur klaren Terminologie wird nun zwischen der *Anforderung*, welche formal oder informal gegeben sein kann, und dem *Szenario*, welches formal entsprechend vorstehender Definition spezifiziert ist, unterschieden.

[6] Dies hängt vor allem von der Präzision ab, mit welcher die Einzelanforderungen erhoben werden. In einem idealen Prozess zur Anforderungserhebung sind die Einzelanforderungen entsprechend dem gewählten Abstraktionsgrad atomar und eine weitere Verfeinerung nicht sinnvoll.

5.2.2. Szenarien zur Anforderungsspezifikation

Für ein System SUT mit der Eingabeschnittstelle I und der Ausgabeschnittstelle O werden durch eine Menge

$$REQ \subsetneq \{(ins, outs) \in I^* \times O^* | \#ins = \#outs > 0\}$$

Szenarien definiert, welche Anforderung an das System beschreiben. Wird das beabsichtigte Verhalten von SUT dabei als *stromverarbeitende Funktion* (Broy und Stølen, 2001)

$$f_{SUT} \in S \to I^\omega \to O^\omega$$

verstanden, muss jedes Szenario $r \in REQ$ die Bedingung

$$f_{SUT}[s_0](\mathsf{in}.r) = \mathsf{out}.r$$

erfüllen. Die Menge S sind dabei die möglichen Zustände des System SUT und $s_0 \in S$ ist ein definierter Initialzustand des Systems[7].

Beispiel 10 (Szenarien)

In Tabelle 5.1 sind Szenarien zu den Anforderungen der Fensterhebersteuerung aus Beispiel 8 (Seite 77), entsprechend der in Beispiel 9 (Seite 87) definierten syntaktischen Schnittstelle, angegeben.

Alle Szenarien sind jeweils ausgehend von einem Initialzustand beschrieben, in welchem sich die Fensterscheibe nicht bewegt und in einer mittleren Position befindet.[8]

Tabelle 5.1.: Szenarien für die Fensterhebersteuerung des Beispiels

Id	Beschreibung	Ports		Ausführungsschritte 1	2	3	4
r_1	Ist die Schalterstellung *open*, wird die Scheibe nach unten bewegt.	In	*switch pos*	*open middle*	*open moving*		
		Out	*motor*	*down*	*down*		
			error	$\perp$[9]	$\perp$		
r_2	Die Bewegung endet, wenn das Schaltersignal nicht mehr anliegt	In	*switch pos*	*open middle*	*open moving*	$\perp$ *moving*[10]	
		Out	*motor*	*down*	*down*	*stop*[10]	
			error	$\perp$	$\perp$	$\perp$	
r_3	...oder sich die Scheibe in der unteren Position befindet	In	*switch pos*	*open middle*	*open moving*	*open bottom*	
		Out	*motor*	*down*	*down*	*stop*	
			error	$\perp$	$\perp$	$\perp$	

[7] Durch I^ω und O^ω werden auch unendliche Ströme als Ein- und Ausgabe zugelassen – im Gegensatz zu den einzelnen Szenarien, welche hier als endliche Abläufe eingeführt wurden, kann das System SUT potentiell auch unendliche Ströme verarbeiten.

[8] Die mittlere Fensterstellung wurde hier als Initialzustand gewählt, um für die Darstellung im Text möglichst kurze Szenarien zu erhalten.

[9] Das Symbol $\perp$ gibt die Nullstellung des Schalters bzw. die Abwesenheit einer Fehlermeldung an, vgl. Beispiel 9 (Seite 87)

[10] Das scheinbar gleichzeitige Auftreten von *moving* und *stop* erklärt sich dadurch, dass *moving* die Eingabe ist, die am Beginn des Verarbeitungsschrittes anliegt, *stop* dagegen die Ausgabe, welche am Ende des Verarbeitungsschrittes produziert wird.

r_4	...oder ein anderer Befehl zum Bewegen dieser Scheibe zu einem späteren Zeitpunkt eingeht; in diesem Fall wird der neue Bewegungsbefehl bearbeitet	In	*switch pos*	*open middle*	*open moving*	*close moving*	*close middle*
		Out	*motor error*	*down* $\perp$	*down* $\perp$	*stop* $\perp$	*up* $\perp$
r_5	...oder der Sensor *pos* kein *moving* Signal sendet, obwohl der Motor angesteuert wird und sich die Scheibe noch nicht in der unteren Position befindet; in diesem Fall wird ein Fehler ausgegeben	In	*switch pos*	*open middle*	*open moving*	*open middle*	
		Out	*motor error*	*down* $\perp$	*down* $\perp$	*stop err*	
r_6	Ist die Schalterstellung *close*, wird die Scheibe nach oben bewegt.	In	*switch pos*	*close middle*	*close moving*		
		Out	*motor error*	*up* $\perp$	*up* $\perp$		
r_7	Die Bewegung endet, wenn das Schaltersignal nicht mehr anliegt	In	*switch pos*	*close middle*	*close moving*	$\perp$ *moving*	
		Out	*motor error*	*up* $\perp$	*up* $\perp$	*stop* $\perp$	
r_8	...oder sich die Scheibe in der oberen Position befindet	In	*switch pos*	*close middle*	*close moving*	*close top*	
		Out	*motor error*	*up* $\perp$	*up* $\perp$	*stop* $\perp$	
r_9	...oder ein anderer Befehl zum Bewegen dieser Scheibe zu einem späteren Zeitpunkt eingeht; in diesem Fall wird der neue Bewegungsbefehl bearbeitet	In	*switch pos*	*close middle*	*close moving*	*open moving*	*open middle*
		Out	*motor error*	*up* $\perp$	*up* $\perp$	*stop* $\perp$	*down* $\perp$
r_{10}	...oder der Sensor *pos* kein *moving* Signal sendet, obwohl der Motor angesteuert wird und sich die Scheibe noch nicht in der oberen Position befindet; in diesem Fall wird ein Fehler ausgegeben	In	*switch pos*	*close middle*	*close moving*	*close middle*	
		Out	*motor error*	*up* $\perp$	*up* $\perp$	*stop err*	

Die Menge REQ spezifiziert das beabsichtigte Systemverhalten f_{SUT} nur partiell. Dies ist schon deshalb gegeben, da ein Szenario $r \in REQ$ nur ein charakteristischer Ablauf einer Anforderung ist, die Anforderung aber allgemein eine Menge von Abläufen betrifft; siehe hierzu Abschnitt 5.1.3.

Die Partialität folgt weiter daraus, dass in der Praxis Anforderungsdokumente das Verhalten des Systems meist nicht vollständig spezifizieren, es liegt also eine Unterspezifikation vor. Häufig beschränken sich Anforderungsdokumente auf das Beschreiben wesentlicher Funktionen und lassen weniger bedeutende Ausnahmefälle unberücksichtigt.

Daher wird zur vollständigen Verhaltensspezifikation die Totalisierung der Funktion f_{SUT} mithilfe von Zustandsmaschinen angeben. Dazu sei an dieser Stelle auf Abschnitt 5.3 verwiesen.

5.2.3. Initialzustand und Vorbedingungen

Wie bereits in Abschnitt 5.1.4 informell beschrieben, ist es notwendig, einen Initialzustand zu definieren, auf welchem die Szenarien basieren. Ein Zustand beschreibt dabei die Belegung der internen Variablen, inklusive der Variablen zur Steuerung des Kontrollflusses (diese entsprechen dem *Kontrollzustand*). Falls Ausgabewerte des SUT zu Eingaben rückkoppeln, umfasst der Zustand auch die Belegung der Ausgabevariablen. Der *Initialzustand* s_0 ist eine Belegung dieser Variablen, auf deren Basis die Anforderungen und Szenarien beschrieben sind. Dieser Initialzustand gilt ebenso für die Testausführung: Vor jeder Ausführung eines Testfalls ist das zu testende System in den Initialzustand, oder in einem für den jeweiligen Testfall äquivalenten Ausgangszustand, zu bringen.

Alle Szenarien $r \in REQ$ sind, entsprechend der zuvor genannten Forderung

$$f_{SUT}[s_0](\mathsf{in}.r) = \mathsf{out}.r,$$

ausgehend vom Initialzustand s_0 definiert. Wie ebenfalls zuvor schon erläutert wurde, ist es für die praktischen Handhabung von Szenarien oft nahe liegend, nicht jede einzelne Anforderung ausgehend vom Initialzustand zu beschreiben und sich stattdessen auf Vorbedingungen zu stützen, welche durch andere Anforderungen bereits gegeben sind. Dazu wird ein Szenario mithilfe folgender Funktionen in die *Vorbedingung* und den *bestimmenden Teil* unterteilt:

$$\begin{aligned} \mathrm{pre} &\in REQ \to REQ \cup (\langle\rangle, \langle\rangle) \\ \det &\in REQ \to I^* \times O^* \end{aligned}$$

wobei für jedes Szenario $r \in REQ$

$$r = ((\mathsf{in}.(\mathsf{pre}.r) \frown \mathsf{in}.(\mathsf{det}.r)), (\mathsf{out}.(\mathsf{pre}.r) \frown \mathsf{out}.(\mathsf{det}.r)))$$

gilt. Damit wird zum einen die Beschreibung der Szenarien kompakter, andererseits geht damit aus dem Szenario hervor, welcher Teil des Szenarios sich direkt aus der jeweilige Anforderung ableitet. Die Vorbedingung pre.r ist damit wiederum ein Szenario einer Anforderung oder das leere Szenario $(\langle\rangle, \langle\rangle)$. Der bestimmende Teil der Anforderung wird durch ein Paar von Eingaben I^* und Ausgaben O^* dargestellt. Damit lässt sich ein Szenario r auch auf Basis einer Vorbedingung $r_{\text{pre}} \in REQ$ angeben:

$$\begin{aligned} \text{pre}.r &= r_{\text{pre}} \\ \text{det}.r &= (\langle i_j, \ldots, i_n \rangle, \langle o_j, \ldots, o_n \rangle) \end{aligned}$$

Beispiel 11 (Vorbedingungen)

Die Szenarien der Fensterhebersteuerung aus Beispiel 10 (Seite 89) lassen sich mithilfe von Vorbedingungen kompakter formulieren. Die Vorbedingung für die Anforderungen, in welchen die Scheibenbewegung endet, sind die Anforderungen, welche das Öffnen beziehungsweise das Schließen des Fensters veranlassen. In Tabelle 5.2 sind diese Vorbedingungen angegeben.

Tabelle 5.2.: Szenarien mit Vorbedingungen für die Fensterhebersteuerung des Beispiels

r	pre.r	**Ports**		det.r	
r_1	–	In	*switch pos*	*open middle*	*open moving*
		Out	*motor error*	*down* $\perp$	*down* $\perp$
r_2	r_1	In	*switch pos*	$\perp$ *moving*	
		Out	*motor error*	*stop* $\perp$	
r_3	r_1	In	*switch pos*	*open bottom*	
		Out	*motor error*	*stop* $\perp$	
r_4	r_1	In	*switch pos*	*close moving*	*close middle*
		Out	*motor error*	*stop* $\perp$	*up* $\perp$
r_5	r_1	In	*switch pos*	*open middle*	
		Out	*motor error*	*stop err*	
r_6	–	In	*switch pos*	*close middle*	*close moving*
		Out	*motor error*	*up* $\perp$	*up* $\perp$

r_7	r_6	In	*switch pos*	$\perp$ *moving*	
		Out	*motor error*	*stop* $\perp$	
r_8	r_6	In	*switch pos*	*close top*	
		Out	*motor error*	*stop* $\perp$	
r_9	r_6	In	*switch pos*	*open moving*	*open middle*
		Out	*motor error*	*stop* $\perp$	*down* $\perp$
r_{10}	r_6	In	*switch pos*	*close middle*	
		Out	*motor error*	*stop err*	

Mithilfe von Vorbedingungen lassen sich ebenfalls Zustände aus Sicht des Nutzers[11] bilden. Ein Zustand $uState \subsetneq REQ$ aus Nutzersicht ist eine Menge von Szenarien. Ein solcher Zustand kann verwendet werden, um deutlich zu machen, dass die Vorbedingung für eine Anforderung auf unterschiedliche Weise erfüllt werden kann. Dies führt zu einer Menge

$$\{r | \mathsf{pre}.r \in uState \wedge \mathsf{det}.r = (\langle i_j, \ldots, i_n \rangle, \langle o_j, \ldots, o_n \rangle)\}$$

von Szenarien. Jedes dieser Szenarien muss ein gültiger Ablauf von SUT sein. Zustände aus Nutzersicht als Vorbedingung entfalten sich somit zu einer Menge von Szenarien mit gleichem bestimmenden Teil.

5.2.4. Testfälle und Szenarien

Ein Szenario stellt bereits einen *charakteristischen Beispieltestfall* zu einer Anforderung dar. Damit trägt die Definition von Szenaren auch zur Qualität der Anforderungsbeschreibung bei, indem

- die *Verständlichkeit* von Anforderungen durch Angabe eines konkreten Beispielablaufs erhöht wird,
- die *Konsistenz* verbessert wird, da alle Anforderungen über eine einheitliche Systemschnittstelle beschrieben sind und keine unterschiedliche Abstraktionsgrade verwendet werden,

[11] Zustände aus Sicht des Nutzers sind nicht mit den internen Zuständen des SUT zu verwechseln. Siehe dazu auch Abschnitt 5.1.4.

- die *Atomarität* von Anforderungen gewährleistet wird – lässt sich eine Anforderung nicht geeignet durch ein Beispiel charakterisieren, ist sie weiter zu zerlegen – und
- die *Überprüfbarkeit* von funktionalen Anforderungen sichergestellt wird.

Allein durch die Angabe von Szenarien ist die triviale Anforderungsüberdeckung, wobei zu jeder funktionalen Anforderung mindestens ein Testfall vorliegen soll, erreicht. Wie aber in Abschnitt 3.3 dargestellt, dient die in dieser Arbeit vorgestellte Methode dazu, eine umfassendere Überdeckung von Anforderungen durch Testfälle zu erreichen. Ein Test, welcher allein auf den als Szenarien angebenden Beispielen beruht, weist folgende Unzulänglichkeiten auf:

Unvollständigkeit der Anforderungen Im Allgemeinen sind die erstellten Listen von Anforderungen nicht vollständig, so dass nicht für jede mögliche Folge von Eingaben eindeutig die zu erwartenden Ausgaben ermittelt werden können. Für den Test ist dagegen sinnvoll, Testfälle aus einem Verhaltensmodell abzuleiten, welches das Verhalten vollständig beschreibt.

Priorisierung von Anforderungen Da Szenarien nur charakteristische Abläufe von informell beschriebenen Anforderungen sind, bleibt die Priorisierung von Anforderungen nur unzureichend erfasst. Oft bleiben Situationen bestehen, in welcher mehrere Anforderungen zutreffen. Die notwendige Priorisierung ist in den Szenarien, da sie auf den einzelnen Anforderungen beruhen, meist nicht enthalten.

Abweichender Nutzungskontext Mitunter beeinflusst die Historie der Nutzung eines Systems die Ausprägung von einzelnen Anforderungen. Es entstehen Wechselwirkungen zwischen Anforderungen, indem sich der Kontext für eine Anforderungen durch die vorangegangene Nutzung ändert. Da sich das System nun in einem anderen Zustand befindet, trifft das gewählte Szenario nicht mehr zu. Der Ausgangszustand entspricht dann nicht dem Initialzustand, beziehungsweise nicht dem Zustand, der nach der Vorbedingung gilt.

Widersprüche in Anforderungen Schließlich enthalten Anforderungsdokumente in frühen Phasen in der Praxis oft Widersprüche, welche erst in späteren Phasen der Spezifikation, etwa durch eine Modellbildung, aufgelöst werden. Somit sind möglicherweise Szenarien festgelegt, welche keinen gültigen Ablauf beschreiben.

Diese Unzulänglichkeiten von Szenarien als alleinige Testfälle werden durch die Erstellung eines Verhaltensmodells und die Generierung von anforderungsorientierten Testfällen aus diesem Modell beseitigt. Durch die Modellerstellung wird die Anforderungsspezifikation vervollständigt. Mithilfe der anforderungsorientierten Generierung von Testfällen aus diesem Modell werden Testfälle gewonnen, welche auch Prioritäten und abweichende Nutzungskontexte von Anforderungen berücksichtigen.

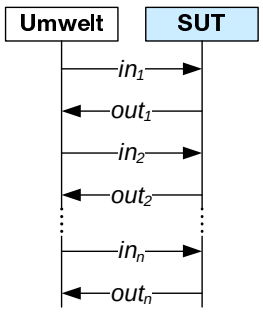

Abbildung 5.8.: Szenario als Message Sequence Chart

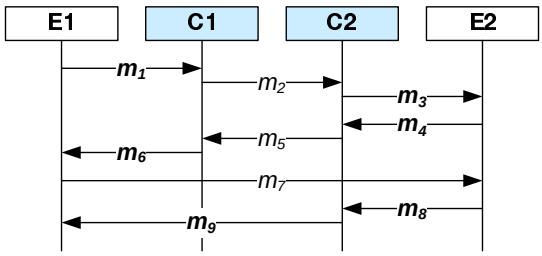

Abbildung 5.9.: MSC mit mehreren Kommunikationspartnern

5.2.5. Abbildung anderer Beschreibungstechniken auf Szenarien

Szenarien, wie sie hier als Folge von Ein-/Ausgabepaaren eingeführt wurden, sind für die Notation von Beispielabläufen häufig nicht besonders praktikabel. Hierzu werden graphische Notationen, wie etwa *Message Sequence Charts (MSC)* (International Telecommunication Union (ITU), 2004) oder *Sequenzdiagramme* als deren Entsprechung in der UML (Object Managment Group, Inc. (OMG), 2005), bevorzugt.

Ein Szenario $r = (\langle in_1, in_2, \ldots, in_n \rangle, \langle out_1, out_2, \ldots, out_n \rangle)$ kann wie in Abbildung 5.8 als MSC dargestellt werden. Dabei umfasst das MSC nur zwei Kommunikationspartner (Akteure), die Umwelt und das zu testende System. MSCs werden allerdings dazu genutzt, die Kommunikation zwischen mehreren Akteuren, in der Regel Komponenten des Gesamtsystems, zu beschreiben. Ein strukturelles Beispiel dazu ist in Abbildung 5.9 angegeben. In diesem Fall lassen sich MSCs nicht als Paare von Ein- und Ausgaben eines Systems angeben.

Vor dem Hintergrund, dass Szenarien beispielhafte Testfälle darstellen, lassen sich Szenarien aus MSCs extrahieren. Für den Test reicht es aus, zwischen zu testendem System *SUT* und seiner Umwelt zu unterscheiden, vergleiche dazu Abschnitt 5.1.2. In einem MSC sind somit die dargestellten Akteure dahingehend zu unterscheiden, ob sie Bestandteil des *SUT* sind, oder außerhalb diesem in dessen Umwelt liegen. Für das Szenario sind dann allein Nachrichten von Bedeutung welche zwischen Komponenten

des SUT und der Umwelt ausgetauscht werden. Nachrichten innerhalb des zu testenden Systems oder zwischen Umweltkomponenten sind für den Test unerheblich, da sie an der Testschnittstelle nicht sichtbar sind. Stellen im Beispiel aus Abbildung 5.9 etwa die Akteure C1 und C2 zusammen das SUT dar, und die Akteure E1 und E2 Komponenten der Systemumgebung, sind Nachrichten m_2, m_5 und m_7 keine Nachrichten an der Schnittstelle des zu testenden Systems. Es ergibt sich somit das Szenario $(\langle m_1, m_4, m_8 \rangle, \langle m_3, m_6, m_9 \rangle)$.

5.2.6. Verwendung partieller Szenarien

Die zuvor eingeführten Szenarien dienen hier als eine Ausgangsbasis für die Generierung anforderungsorientierter Testfälle. Sie geben exemplarische Abläufe, also beispielhafte Testfälle, an. Hierzu wird jeweils von einer vollständige Spezifikation der Ein- und Ausgabekanäle ausgegangen. Die Szenarien dienen im Rahmen der vorliegenden Arbeit weder unmittelbar zur Spezifikation (einer Menge) von Testfällen noch zur vollständigen Spezifikation der zu testenden Eigenschaften. Sie stellen vor allem ein Mittel zur Steuerung der Testfallgenerierung dar. Die vollständige Systemspezifikation wird dagegen durch das Testmodell in Form einer Zustandsmaschine angegeben (Abschnitt 5.3).

Die vollständige Beschreibung eines Szenarios ist zur Formulierung einer zu testenden Eigenschaft in der Regel eine Überspezifikation. Meist sind nur Teile der Ein- und Ausgabe für die Prüfung einer Eigenschaft von Interesse. Für die Beschreibung von Testfällen ist daher oft eine partielle Spezifikation von Szenarien ausreichend und meist praktikabler.

Werden Szenarien nur partiell spezifiziert, können die nicht spezifizierten Ein- und Ausgaben ergänzt werden, um einzelne vollständige Szenarien zu erzielen, welche im Weiteren verwendet werden. Dies kann manuell aber auch (teil-)automatisiert erfolgen. Hierzu können entweder Eingaben zufällig gewählt werden und die Ausgaben aus dem Testmodell berechnet werden oder mit bekannten Techniken der Testfallgenerierung werden einzelne Beispieltestfälle mit den geforderten Eigenschaften ermittelt (etwa mittels Modellprüfung oder Constraint-Logik-Programmierung, siehe Kapitel 2, Abschnit 2.3).

Allerdings lässt sich nicht vermeiden, dass vollständige Szenarien auch Teile in Ein- und Ausgabe umfassen, welche für die zugehörige Anforderung unerheblich sind. Um dennoch möglichst charakteristische Szenarien für eine Anforderung zu erhalten, ist daher auf Folgendes zu achten:

- Für die syntaktische Schnittstelle ist der Abstraktionsgrad so zu wählen, das diese nur auf das zu testende Verhalten fokussiert ist.
- Vorbedingungen sind umfangreich einzusetzen.

- Einzelne Szenarien sind auf ein Mindestmaß an Schritten zu beschränken.

In der vorliegenden Arbeit werden vollständige Szenarien vor allem deshalb verwendet, da in der Praxis bisher keine einheitliche und formal präzise Spezifikationstechnik zur partiellen Formulierung von Szenarien Verbreitung gefunden hat. Dagegen lassen sich andere Beschreibungstechniken, wie beispielsweise MSCs, auf vollständige Szenarien abbilden, beziehungsweise diese können daraus abgeleitet werden. Schließlich trägt die vollständige Angabe von Abläufen zur Validierung der Spezifikation bei.

5.3. Vervollständigung der Spezifikation

Szenarien, wie sie in den vorhergehenden Abschnitten beschrieben wurden, stellen eine formale Repräsentation von Anforderungen dar. Allerdings ist damit das Verhalten des Systems noch nicht vollständig – bezogen auf den gewählten Abstraktionsgrad – beschreiben. Diese Unvollständigkeit fußt in zwei Ursachen:

- Szenarien sind lediglich *charakteristische* Abläufe von Anforderungen. Eine einzelne Anforderung spezifiziert aber in der Regel eine Menge von verschiedenen Abläufen mit gleichen Merkmalen. (Siehe hierzu Abschnitt 5.1.3.)
- Durch die informellen Szenarienbeschreibungen zu den einzelnen Anforderungen ist nicht sichergestellt, dass für jede mögliche Folge von Eingaben eine eindeutige Folge von Ausgaben festgelegt ist. Im Allgemeinen sind dazu weitere Präzisierungen und Ergänzungen notwendig. Ursache dafür ist, dass informelle Szenarienbeschreibungen meist in der Annahme eines nicht präzise definierten „Normalverhaltens“ der übrigen Systemumgebung formuliert sind.

Die Gesamtanforderung an das System, also das für die Umgebung wahrnehmbare Verhalten des Systems, geht somit über das durch die Szenarien angegebene Verhalten hinaus. Gegenstand des Tests soll aber die Gesamtanforderung sein. Aus diesem Grund ist zur Generierung von Testfällen eine vollständige Beschreibung des für die Umgebung wahrnehmbaren Verhaltens notwendig. Die Angabe eines Verhaltensmodells, welches das erwartete Verhalten des Systems als totale Funktion beschreibt, ist die zweite Stufe der Anforderungsspezifikation.[12] Im vorgestellten Entwicklungsprozess ist dies der dritte Schritt, wie in Abbildung 5.10 nochmals gezeigt. Dieser Schritt beruht nicht allein auf den formalen Szenarien sondern auch auf informal gegebenen Anforderungen, ebenso wie auf Anforderungen welche (noch) nicht explizit formuliert sind, da zusätzliche Informationen zur Totalisierung des Modells notwendig sind. Dementsprechend ist auch dies ein kreativer Prozess, welcher nicht vollständig automatisierbar ist. Wesentliche Voraussetzung für die Praktikabilität dieser totalen Funktionsspezifikation ist die geeignete Wahl des Abstraktionsgrades, welcher durch

[12] Vgl. Abbildung 5.1, Seite 72.

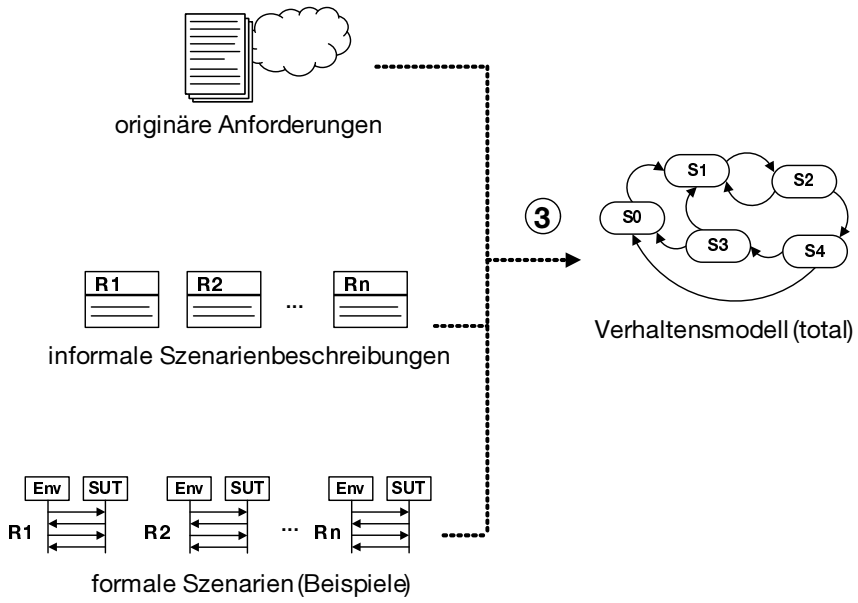

Abbildung 5.10.: Schritt 3 des vorgeschlagenen Entwicklungs- und Testprozesses: Erstellung eines vollständigen Verhaltensmodells (Ausschnitt aus Abbildung 4.1)

die syntaktische Schnittstelle, wie sie in Abschnitt 5.2.1 bereits verwendet wurde, festgelegt wird. Die totale Funktion beschreibt hier die Anforderung an das System, wie sie die Nutzer (oder allgemein die Systemumgebung) wahrnehmen, nicht jedoch die detaillierte Kommunikation an der technischen Schnittstelle des Systems.

5.3.1. Verhaltensmodell als zweite Stufe der Spezifikation

Ausgehend von den Szenarien wird zur Spezifikation des Verhaltens des zu testenden Systems ein *Verhaltensmodell* erstellt. Im Gegensatz zu der Menge von Szenarien – und im Allgemeinen auch im Gegensatz zu den informell beschriebenen Anforderungen – wird durch das Verhaltensmodell das beabsichtigte Verhalten des *SUT* vollständig im gewählten Abstraktionsgrad beschrieben. Das Verhaltensmodell spezifiziert damit das Verhalten als *totale* Funktion.

Dieses Verhaltensmodell dient für die Generierung von Testfällen als *Testmodell*. Es ist möglich, dieses Modell auch als Grundlage für die Erstellung des Systems einzusetzen. Für die in dieser Arbeit vorgestellte Methode ist es unerheblich, ob das Verhaltensmodell eigens für Zwecke des Tests erstellt wurde oder ob ein bereits vorliegendes Modell aus der Entwurfsphase wiederverwendet wird, sofern die im Folgenden genannten Forderungen an den Abstraktionsgrad, die Akzeptanz von Szenarien und die Totalität erfüllt sind.

Diese Arbeit stützt sich bei der Beschreibung von Modellen auf *Mealy-Maschinen*. Für den Einsatz in der Praxis geeigneter sind *erweiterte endliche Zustandsmaschinen (EFSM),* welche auf Mealy-Maschinen abgebildet werden können. Beide Ansätze werden nun eingeführt.

Mealy-Maschinen

Zur Spezifikation reaktiver System wurden von Mealy (1955) endliche Zustandsmaschinen vorgeschlagen, welche Ausgaben beim Zustandsübergang produzieren. Mealy-Maschinen lassen sich als 5-Tupel

$$M = (I, O, S, \delta, \lambda)$$

beschreiben. Definition und Notation folgen dabei jener von Lee und Yannakakis (1994, 1996) sowie von Jonsson (2004) verwendeten. Dabei bezeichnet

- I eine endliche, nicht leere Menge von Eingabesymbolen – das *Eingabealphabet*
- O eine endliche, nicht leere Menge von Ausgabesymbolen – das *Ausgabealphabet*
- S eine endliche, nicht leere Menge von *Zuständen*
- $\delta \in S \times I \to S$ die *Zustandsübergangsfunktion*
- $\lambda \in S \times I \to O$ die *Ausgabefunktion*

Wird eine Eingabesequenz $ins = \langle in_1, in_2, \ldots, in_k \rangle \in I^*$ von einer Mealy-Maschine verarbeitet, durchläuft die Maschine ausgehend von einem initialen Zustand s_0 nacheinander eine Folge von Zustanden $s_1, s_2, \ldots, s_k$ und erzeugt eine Ausgabesequenz $outs = \langle out_1, out_2, \ldots, out_k \rangle \in O^*$. Dabei gilt $s_j = \delta(s_{j-1}, in_j)$ und $out_j = \lambda(s_{j-1}, in_j)$ für $j = 1, \ldots, k$. Die Zustandsübergangsfunktion δ und die Ausgabefunktion λ lassen sich zu Funktionen

$$\Delta \in S \times I^* \to S$$
$$\Lambda \in S \times I^* \to O^*$$

über Eingabesequenzen erweiterten. Diese sind wie folgt rekursiv definiert:

$$\begin{aligned} \Delta(s, \langle\rangle) &= s \\ \Delta(s, \langle x \rangle \frown ins) &= \Delta(\delta(s, x), ins) \\ \Lambda(s, \langle\rangle) &= \langle\rangle \\ \Lambda(s, \langle x \rangle \frown ins) &= \langle \lambda(s, x) \rangle \frown \Lambda(\delta(s, x), ins) \end{aligned}$$

Damit kann das Verhalten eines Systems SUT für alle Eingaben $ins \in I^*$, ausgehend von einem Initialzustand s_0, durch die Ausgabefunktion der Mealy-Maschine angegeben werden:

$$f_{SUT}[s_0](ins) \equiv \Lambda(s_0, ins)$$

Beispiel 12 (Verhaltensmodell als Mealy-Maschine)

Zur Vervollständigung der Spezifikation des Fensterhebers aus den Beispielen 8 – 11 wurde die Mealy-Maschine

$$M = ((SWITCH \times POS), (MOTOR \times ERR), S, \delta, \lambda)$$

mit

$$S \stackrel{\text{def}}{=} \{Idle, Down, Up\}$$

und δ, λ entsprechend Tabelle 5.3 spezifiziert.

Tabelle 5.3.: Zustandsübergangsfunktion δ und Ausgabefunktion λ einer Mealy-Maschine für die Basisfunktion der Steuerung eines Fensterhebers (ursprüngliches Modell)

	s	in	$\delta(s, in)$	$\lambda(s, in)$
t_1	*Idle*	(*open*, *middle*)	*Down*	(*down*, $\perp$)
t_2	*Idle*	(*open*, *top*)	*Down*	(*down*, $\perp$)
t_3	*Idle*	(*open*, *bottom*)	*Idle*	(*stop*, $\perp$)
t_4	*Idle*	(*open*, *moving*)	*Idle*	(*stop*, *err*)
t_5	*Idle*	(*close*, *middle*)	*Up*	(*up*, $\perp$)
t_6	*Idle*	(*close*, *top*)	*Idle*	(*stop*, $\perp$)
t_7	*Idle*	(*close*, *bottom*)	*Up*	(*up*, $\perp$)
t_8	*Idle*	(*close*, *moving*)	*Idle*	(*stop*, *err*)
t_9	*Idle*	($\perp$, *middle*)	*Idle*	(*stop*, $\perp$)
t_{10}	*Idle*	($\perp$, *top*)	*Idle*	(*stop*, $\perp$)
t_{11}	*Idle*	($\perp$, *bottom*)	*Idle*	(*stop*, $\perp$)
t_{12}	*Idle*	($\perp$, *moving*)	*Idle*	(*stop*, *err*)
t_{13}	*Down*	(*open*, *middle*)	*Idle*	(*stop*, *err*)
t_{14}	*Down*	(*open*, *top*)	*Down*	(*down*, $\perp$)
t_{15}	*Down*	(*open*, *bottom*)	*Idle*	(*stop*, $\perp$)
t_{16}	*Down*	(*open*, *moving*)	*Down*	(*down*, $\perp$)
t_{17}	*Down*	(*close*, *middle*)	*Up*	(*up*, $\perp$)
t_{18}	*Down*	(*close*, *top*)	*Idle*	(*stop*, $\perp$)
t_{19}	*Down*	(*close*, *bottom*)	*Up*	(*up*, $\perp$)
t_{20}	*Down*	(*close*, *moving*)	*Up*	(*up*, $\perp$)
t_{21}	*Down*	($\perp$, *middle*)	*Idle*	(*stop*, *err*)
t_{22}	*Down*	($\perp$, *top*)	*Idle*	(*stop*, $\perp$)
t_{23}	*Down*	($\perp$, *bottom*)	*Idle*	(*stop*, $\perp$)
t_{24}	*Down*	($\perp$, *moving*)	*Idle*	(*stop*, $\perp$)

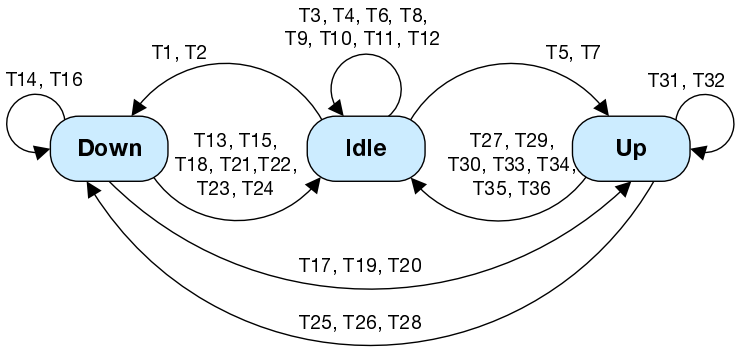

Abbildung 5.11.: Beispiel: Graphische Darstellung der Mealy-Maschine zur Basisfunktion des Fensterhebers. Zur Vereinfachung wurden alle Zustandsübergänge zwischen zwei Zuständen jeweils zu einer Kante zusammengefasst. An den Kanten sind jeweils die damit repräsentierten Transitionen annotiert, vgl. Tabelle 5.3.

t_{25}	*Up*	(*open*,	*middle*)	*Down*	(*down*, $\perp$)
t_{26}	*Up*	(*open*,	*top*)	*Down*	(*down*, $\perp$)
t_{27}	*Up*	(*open*,	*bottom*)	*Idle*	(*stop*, $\perp$)
t_{28}	*Up*	(*open*,	*moving*)	*Down*	(*down*, $\perp$)
t_{29}	*Up*	(*close*,	*middle*)	*Idle*	(*stop*, *err*)
t_{30}	*Up*	(*close*,	*top*)	*Idle*	(*stop*, $\perp$)
t_{31}	*Up*	(*close*,	*bottom*)	*Up*	(*up*, $\perp$)
t_{32}	*Up*	(*close*,	*moving*)	*Up*	(*up*, $\perp$)
t_{33}	*Up*	($\perp$,	*middle*)	*Idle*	(*stop*, *err*)
t_{34}	*Up*	($\perp$,	*top*)	*Idle*	(*stop*, $\perp$)
t_{35}	*Up*	($\perp$,	*bottom*)	*Idle*	(*stop*, $\perp$)
t_{36}	*Up*	($\perp$,	*moving*)	*Idle*	(*stop*, $\perp$)

In Abbildung 5.11 ist die Mealy-Maschine des Fensterhebers als gerichteter Graph dargestellt.

Mealy-Maschinen sind somit eine Möglichkeit, das Verhalten des Systems vollständig zu spezifizieren. Die Vollständigkeit ist gegeben, da die Funktion $\Lambda \in S \times I^* \rightarrow O^*$ für Mealy-Maschinen total ist, sofern die Zustandsübergangsfunktion δ und Ausgabefunktion λ total sind. Da diese beiden Funktionen über endliche Mengen von Zuständen und Eingabesymbolen definiert sind, ist die Sicherstellung der Totalität trivial.

Erweiterte endliche Zustandsmaschinen

Die Angabe von Mealy-Maschinen ist für die Spezifikation umfangreicher Systeme meist nicht praktikabel. In der Industrie werden zunehmend Modellierungswerkzeuge wie Statemate (IBM Corporation), Ascet (ETAS) oder Stateflow (The MathWorks,

Inc.) zur Spezifikation reaktiver Systeme eingesetzt. Diese Werkzeuge nutzen Sprachen auf Basis der von Harel (1987) vorgeschlagenen *Statecharts* und den entsprechenden *state machine diagrams* in der Unified Modeling Language (UML) (Object Managment Group, Inc. (OMG), 2005). Statecharts sind Erweiterungen von Mealy-Maschinen (Harel, 1988). Eine vergleichbare Notation nutzt das Werkzeug AutoFOCUS 2 (AF2), welches im Rahmen dieser Arbeit zur Spezifikation verwendet wird. Damit finden in dieser Arbeit Testmodelle Verwendung, wie sie zuvor auch von Lötzbeyer und Pretschner (2000); Pretschner und Lötzbeyer (2001); Pretschner (2003); Lötzbeyer (2003) zur Testfallgenerierung verwendet wurden, allerdings auf Basis der Werkzeugs AutoFOCUS (Huber u. a., 1997; Schätz u. a., 2002; Schätz und Huber, 1999), dem Vorgänger von AutoFOCUS 2.

In AutoFOCUS beziehungsweise AutoFOCUS 2 werden Systeme in mehrere *Komponenten* gegliedert. Diese Komponenten werden mit *Kanälen*, über welche Komponenten miteinander kommunizieren, in *Systemstrukturdiagrammen (system structure diagram, SSD)* modelliert. Zur Verhaltensbeschreibung werden in den atomaren Komponenten *Erweiterte Endliche Zustandsmaschinen (extended finite state machine, EFSM)* genutzt. EFSMs sind Mealy-Maschinen, erweitert um lokale Variablen. Die Transitionen sind dabei um Wächter erweitert. Ein Wächter ist ein Ausdruck über die Eingabeports und die lokalen Variablen, eine Transition schaltet nur, falls der Wächter zu `true` ausgewertet wird. In AutoFOCUS oder AutoFOCUS 2 werden diese Zustandsmaschinen als *Zustandstransitionsdiagramme (state transition diagram, STD)* angegeben.

Damit das Verhaltensmodell für die in dieser Arbeit beschriebene Methode der anforderungsorientierten Testfallgenerierung verwendet werden kann, müssen diese Voraussetzungen erfüllt sein:

- Der Abstraktionsgrad muss dem Abstraktionsgrad der Szenarien entsprechen
- Die Szenarien zu den einzelnen Anforderungen müssen von dem Modell akzeptiert werden, also gültige Abläufe des Modells sein
- Das Modell muss total hinsichtlich aller möglichen Folgen von Eingaben sein

Diese Voraussetzungen werden in den folgenden Abschnitten genauer beschrieben.

5.3.2. Abstraktionsgrad entsprechend jenem der Szenarien

Wie bereits in Abschnitt 2.3 dargestellt, ist es für den modellbasierten Test essentiell, dass das Modell, aus welchem Testfälle generiert werden, vom zu testenden System abstrahiert. Der Umfang der Abstraktion bestimmt die potentielle Mächtigkeit zur Entdeckung von Fehlern mit den generierten Testfällen. Abweichungen zwischen der Spezifikation, also dem Testmodell, und der Implementierung des zu testenden

Systems, können nur entstehen, wenn zur Erstellung der Implementierung weitere Details berücksichtigt werden müssen und dadurch die Vorgaben des Testmodells verletzt werden können. Methodisch wird hierzu empfohlen, im Testmodell nur die Funktion des Systems aus Sicht der Nutzer zu beschreiben und vor allem technische Details, welche zur Realisierung des Systems nötig sind, im Modell unberücksichtigt zu lassen.

Dieser Vorgabe wird die hier vorgestellte Methodik insofern gerecht, als sie fordert, dass das Verhaltensmodell und die zuvor identifizierten Szenarien derselben Systemschnittstelle genügen müssen. Szenarien und Modell sollen somit über dasselbe Eingabealphabet I und dasselbe Ausgabealphabet O beschrieben sein. Diese Vorgabe für das Verhaltensmodell erfüllt zwei Funktionen: Zunächst wird damit sichergestellt, dass das Verhaltensmodell ebenfalls aus der Sicht des Nutzers beschrieben ist. Da wiederum die Systemschnittstelle der Szenarien verwendet wird, werden nur Ein-/Ausgaben betrachtet, welche an der Schnittstelle zur Umwelt auftreten. Die Schnittstelle umfasst damit nur Ein- beziehungsweise Ausgaben, welche in den Anforderungen zur Spezifikation von Szenarien verwendet wurde. Da Szenarien nur Aktionen beschreiben, über welche System und Umwelt kommunizieren, sind weitere Aktionen innerhalb des Systems nicht im Umfang der verwendeten Schnittstelle.

Weiter ist es für den anforderungsorientierten Test notwendig, die Szenarien wiederum im Testmodell identifizieren zu können, auch dies erfordert eine gemeinsame Schnittstelle. Dies wird im folgenden Abschnitt und später in Kapitel 6 genauer beschrieben.

5.3.3. Akzeptanz von Szenarien

Die Szenarien, welche zuvor als charakteristische Beispiele für die Anforderungen an das System identifiziert wurden, müssen von dem Verhaltensmodell akzeptiert werden. Jedes Szenario aus der Menge der Szenarien muss damit die Ausgabefunktion des Verhaltensmodells erfüllen, also für alle $r = (ins, outs) \in REQ$ muss

$$\Lambda(ins) = outs$$

gelten. Diese Bedingung ist nur unter der idealen Voraussetzung zu erfüllen, dass alle Anforderungen und alle gewählten Beispielabläufe auch im weiteren Spezifikationsprozess ihre Gültigkeit behalten. In der praktischen Anwendung wird allerdings nur in seltenen Fällen die initial definierte Menge von Anforderungen unverändert Bestand haben. Durch die Präzisierung, welche mit der Erstellung eines vollständigen Verhaltensmodells einhergeht, werden oft auch Widersprüche und falsche Annahmen in den ursprünglichen Anforderungen aufgezeigt. In diesem Sinne dient die Erstellung eines Verhaltensmodells auch der Klärung von zweifelhaften Anforderungen. Aus

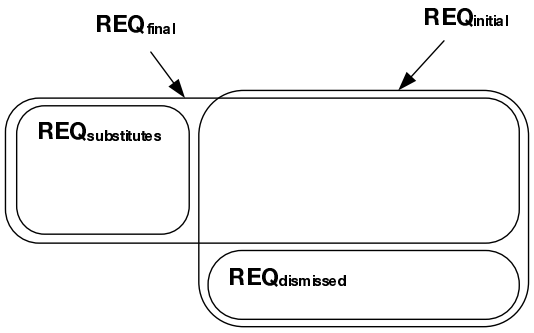

Abbildung 5.12.: Anforderungsmengen im Spezifikationsprozess

diesem Grund ist es sinnvoll, während des Spezifikationsprozesses unterschiedliche Mengen von Anforderungen zu unterscheiden, wie dies vom Autor auch schon in (Pfaller, 2008) dargestellt wurde.

Menge der initialen Szenarien Mit $REQ_{initial}$ wird die Menge der initial definierten Szenarien bezeichnet, welche potentiell auch Elemente enthält, die im weiteren Spezifikationsprozess verworfen werden.

Menge der verworfenen Szenarien Die Menge $REQ_{dismissed} \subseteq REQ_{initial}$ bezeichnet die Menge der Szenarien, welche zwar initial spezifiziert wurden, im weiteren Verlauf der Spezifikation aber verworfen wurden, da nicht als Anforderung bestand hielten.

Menge der hinzugefügten Szenarien Die Menge $REQ_{substitutes}$, mit $REQ_{substitutes} \cap REQ_{initial} = \{\}$, bezeichnet die Menge der Anforderungen beziehungsweise Szenarien, welche im Verlauf der Spezifikation zu den ursprünglich definierten Szenarien hinzugefügt wurde, insbesondere als Ersatz beziehungsweise Präzisierung für die Szenarien aus $REQ_{dismissed}$.

Menge der finalen Szenarien Mit REQ_{final} wird die Menge der Szenarien bezeichnet, welche im Laufe der Spezifikation explizit angegeben wurden und von dem zu erstellenden und zu testenden System tatsächlich erfüllt werden müssen. Diese Menge ergibt sich aus den zuvor genannten Mengen: $REQ_{final} = (REQ_{initial} \backslash REQ_{dismissed}) \cup REQ_{substitutes}$.

Abbildung 5.12 stellt diese Mengen in einem Mengendiagramm schematisch dar. Sofern nicht explizit etwas anders angegeben ist, bezeichnet die Menge REQ die Menge der Szenarien, welche tatsächlich vom System ausgeführt werden, ist also mit der Menge REQ_{final} identisch. Die finale Menge der Szenarien steht in der Praxis somit meist erst am Ende des gesamten Prozesses der Anforderungsspezifikation fest. Beispiel 13 verdeutlicht diese unterschiedlichen Mengen von Anforderungsszenarien anhand der Spezifikation der Fensterheberfunktion.

Beispiel 13 (Geänderte Anforderungen)

Zu den Anforderungen der Fensterhebersteuerung aus Beispiel 10 (Seite 89) sei angenommen, dass sich im Laufe der Detailspezifikation herausstellte, dass der Bedienschalter des Fensterhebers im Zusammenspiel mit dem Fensterhebermotor es bauartbedingt erfordert, in die Neutralstellung gebracht zu werden, bevor ein Richtungswechsel möglich ist. Die initialen Szenarien r_4 und r_9 (Tabelle 5.4) widersprechen dem aber und wurden verworfen.

Tabelle 5.4.: Verworfene Szenarien der Fensterheberspezifikation

r	pre.r	**Ports**		det.r	
r_4	r_1	In	switch	close	close
			pos	moving	middle
		Out	motor	stop	up
			error	$\perp$	$\perp$
r_9	r_6	In	switch	open	open
			pos	moving	middle
		Out	motor	stop	down
			error	$\perp$	$\perp$

Die verworfenen Anforderungen r_4 und r_9 wurden durch neue Anforderungen r_4', r_4'', r_9' und r_9'' ersetzt, wie sie in Tabelle 5.5 angeben sind. Die Szenarien r_4' und r_9' geben dabei den Fall an, dass ein unmittelbarer Richtungswechsel zum Stillstand des Motors führt, aber keine Bewegung in entgegengesetzter Richtung auslöst. Nur falls, wie in den Szenarien r_4'' und r_9'', der Schalter in die Neutralstellung ($\perp$) gebracht wurde, wird eine umgekehrte Bewegung am Motor ausgelöst.

Tabelle 5.5.: Hinzugefügte Szenarien der Fensterheberspezifikation

r	pre.r	**Ports**		det.r		
r_4'	r_1	In	switch	close	close	close
			pos	moving	middle	middle
		Out	motor	stop	stop	stop
			error	$\perp$	$\perp$	$\perp$
r_4''	r_1	In	switch	close	$\perp$	close
			pos	moving	middle	middle
		Out	motor	stop	stop	up
			error	$\perp$	$\perp$	$\perp$
r_9'	r_6	In	switch	open	open	open
			pos	moving	middle	middle
		Out	motor	stop	stop	stop
			error	$\perp$	$\perp$	$\perp$

r_9''	r_6	In	switch	open	$\perp$	open
			pos	moving	middle	middle
		Out	motor	stop	stop	down
			error	$\perp$	$\perp$	$\perp$

In diesem Beispiel lassen sich folgende Anforderungsmengen unterscheiden:

$$
\begin{aligned}
REQ_{initial} &= \{r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8, r_9, r_{10}\} \\
REQ_{dismissed} &= \{r_4, r_9\} \\
REQ_{substitutes} &= \{r_4', r_4'', r_9', r_9''\} \\
REQ_{final} &= \{r_1, r_2, r_3, r_4', r_4'', r_5, r_6, r_7, r_8, r_9', r_9'', r_{10}\}
\end{aligned}
$$

5.3.4. Totalität und Nichtdeterminismus

Das Verhaltensmodell wird vor allem deshalb erstellt, um die Anforderungsspezifikation zu vervollständigen. Der Begriff der „Vervollständigung“ wird in Bezug auf die Spezifikation eines Systems unterschiedlich verwendet, einerseits für die vollständige Erfassung der Wünsche und Bedürfnisse der Stakeholder, andererseits wird damit die vollständige Beschreibung des Ein-/Ausgabe-Verhaltens bezeichnet. Das Verhaltensmodell verfolgt dabei das primäre Ziel, das Ein-/Ausgabeverhalten vollständig zu beschreiben.

Vollständige Erfassung der Erwartungen der Stakeholder

Damit ein System die Stakeholder und insbesondere die Nutzer zufrieden stellt, müssen deren Erwartungen möglichst vollständig erfasst worden sein. In einem idealen Entwicklungsprozess ist diese Aktivität bereits durch die Spezifikation von Einzelanforderungen abgeschlossen, etwa indem eine vollständige Liste von informell beschriebenen Anforderungen erstellt wird. Meist treten im weiteren Entwicklungsprozess allerdings zusätzliche Fragestellungen nach den Erwartungen der Nutzer in bestimmten Situationen auf. Den Nutzern sind viele Fragestellungen und Situationen oft nicht bewusst. Die Erstellung einer vollständigen Verhaltensbeschreibung macht derartigen Klärungsbedarf explizit. Werden in diesem Sinne die Anforderungen erweitert, sind zusätzliche Szenarien für die neu eingeführten Anforderungen zu erstellen.

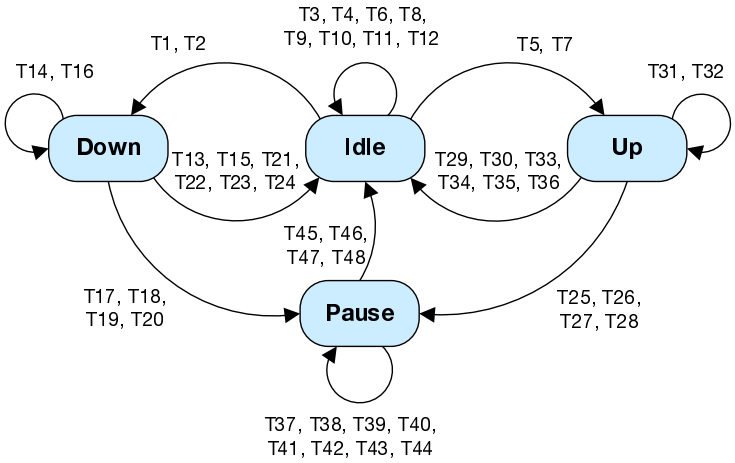

Abbildung 5.13.: Beispiel: Graphische Darstellung der endgültigen Mealy-Maschine zur Basisfunktion des Fensterhebers. Neu eingeführt wurde der Zustand *Pause*. Für die Beschreibung der Transitionen siehe Tabelle 5.6.

Beispiel 14 (Endgültiges Modell nach Anpassung der Anforderungen)

Abbildung 5.13 und Tabelle 5.6 geben das finale Modell der Fensterheberbasisfunktion an, nachdem die geänderten Anforderungen, siehe Beispiel 13 (Seite 105), in das Modell eingearbeitet wurde.

Die Zustandsmenge S wurde dabei um den Zustand *Pause* erweitert, so dass nun gilt:

$$M = ((SWITCH \times POS), (MOTOR \times ERR), S, \delta, \lambda)$$

mit

$$S \stackrel{\text{def}}{=} \{Idle, Down, Up, Pause\}.$$

Tabelle 5.6.: Zustandsübergangsfunktion δ und Ausgabefunktion λ einer Mealy-Maschine für die Basisfunktion der Steuerung eines Fensterhebers (mit geänderten Anforderungen)

	s	in	$\delta(s, in)$	$\lambda(s, in)$
t_1	*Idle*	(*open*, *middle*)	*Down*	(*down*, $\perp$)
t_2	*Idle*	(*open*, *top*)	*Down*	(*down*, $\perp$)
t_3	*Idle*	(*open*, *bottom*)	*Idle*	(*stop*, $\perp$)
t_4	*Idle*	(*open*, *moving*)	*Idle*	(*stop*, *err*)
t_5	*Idle*	(*close*, *middle*)	*Up*	(*up*, $\perp$)
t_6	*Idle*	(*close*, *top*)	*Idle*	(*stop*, $\perp$)
t_7	*Idle*	(*close*, *bottom*)	*Up*	(*up*, $\perp$)
t_8	*Idle*	(*close*, *moving*)	*Idle*	(*stop*, *err*)

t_9	*Idle*	($\perp$,	*middle*)	*Idle*	(*stop*,	$\perp$)
t_{10}	*Idle*	($\perp$,	*top*)	*Idle*	(*stop*,	$\perp$)
t_{11}	*Idle*	($\perp$,	*bottom*)	*Idle*	(*stop*,	$\perp$)
t_{12}	*Idle*	($\perp$,	*moving*)	*Idle*	(*stop*,	*err*)
t_{13}	*Down*	(*open*,	*middle*)	*Idle*	(*stop*,	*err*)
t_{14}	*Down*	(*open*,	*top*)	*Down*	(*down*,	$\perp$)
t_{15}	*Down*	(*open*,	*bottom*)	*Idle*	(*stop*,	$\perp$)
t_{16}	*Down*	(*open*,	*moving*)	*Down*	(*down*,	$\perp$)
t_{17}	*Down*	(*close*,	*middle*)	*Pause*	(*stop*,	$\perp$)
t_{18}	*Down*	(*close*,	*top*)	*Pause*	(*stop*,	$\perp$)
t_{19}	*Down*	(*close*,	*bottom*)	*Pause*	(*stop*,	$\perp$)
t_{20}	*Down*	(*close*,	*moving*)	*Pause*	(*stop*,	$\perp$)
t_{21}	*Down*	($\perp$,	*middle*)	*Idle*	(*stop*,	*err*)
t_{22}	*Down*	($\perp$,	*top*)	*Idle*	(*stop*,	$\perp$)
t_{23}	*Down*	($\perp$,	*bottom*)	*Idle*	(*stop*,	$\perp$)
t_{24}	*Down*	($\perp$,	*moving*)	*Idle*	(*stop*,	$\perp$)
t_{25}	*Up*	(*open*,	*middle*)	*Pause*	(*stop*,	$\perp$)
t_{26}	*Up*	(*open*,	*top*)	*Pause*	(*stop*,	$\perp$)
t_{27}	*Up*	(*open*,	*bottom*)	*Pause*	(*stop*,	$\perp$)
t_{28}	*Up*	(*open*,	*moving*)	*Pause*	(*stop*,	$\perp$)
t_{29}	*Up*	(*close*,	*middle*)	*Idle*	(*stop*,	*err*)
t_{30}	*Up*	(*close*,	*top*)	*Idle*	(*stop*,	$\perp$)
t_{31}	*Up*	(*close*,	*bottom*)	*Up*	(*up*,	$\perp$)
t_{32}	*Up*	(*close*,	*moving*)	*Up*	(*up*,	$\perp$)
t_{33}	*Up*	($\perp$,	*middle*)	*Idle*	(*stop*,	*err*)
t_{34}	*Up*	($\perp$,	*top*)	*Idle*	(*stop*,	$\perp$)
t_{35}	*Up*	($\perp$,	*bottom*)	*Idle*	(*stop*,	$\perp$)
t_{36}	*Up*	($\perp$,	*moving*)	*Idle*	(*stop*,	$\perp$)
t_{37}	*Pause*	(*open*,	*middle*)	*Pause*	(*stop*,	$\perp$)
t_{38}	*Pause*	(*open*,	*top*)	*Pause*	(*stop*,	$\perp$)
t_{39}	*Pause*	(*open*,	*bottom*)	*Pause*	(*stop*,	$\perp$)
t_{40}	*Pause*	(*open*,	*moving*)	*Pause*	(*stop*,	*err*)
t_{41}	*Pause*	(*close*,	*middle*)	*Pause*	(*stop*,	$\perp$)
t_{42}	*Pause*	(*close*,	*top*)	*Pause*	(*stop*,	$\perp$)
t_{43}	*Pause*	(*close*,	*bottom*)	*Pause*	(*stop*,	$\perp$)
t_{44}	*Pause*	(*close*,	*moving*)	*Pause*	(*stop*,	*err*)
t_{45}	*Pause*	($\perp$,	*middle*)	*Idle*	(*stop*,	$\perp$)
t_{46}	*Pause*	($\perp$,	*top*)	*Idle*	(*stop*,	$\perp$)
t_{47}	*Pause*	($\perp$,	*bottom*)	*Idle*	(*stop*,	$\perp$)
t_{48}	*Pause*	($\perp$,	*moving*)	*Idle*	(*stop*,	*err*)

Totalisierung des Ein-/Ausgabe-Verhaltens

Die Erstellung des Verhaltensmodells dient vor allem dazu, die Beschreibung des Systemverhaltens als totale Funktion anzugeben. Damit lässt sich das Verhalten als Mealy-Maschine beschreiben, für welche die Zustandsübergangs- als auch die Ausgabefunktion total sind. Eine entsprechende Mealy-Maschine $M = (I, O, S, \delta, \lambda)$ gibt so zu jedem Eingabewort $ins \in I^*$ ein Ausgabewort $outs \in O^*$ an, welches eine korrekte Ausgabe des beschriebenen Systems ist. Da sowohl δ als auch λ in Mealy-Maschinen Funktionen sind, wird das System durch ein deterministisches Systemmodell beschrieben. Ziel des Verhaltensmodells ist somit die Totalisierung der Funktion $f_{SUT}[s](i) = o$, welche das Verhalten des Systems beschreibt.

Durch die Menge der einzelnen Anforderungen und Szenarien ist das Verhalten eines Systems meist nicht total und deterministisch beschrieben, auch falls angenommen wird, dass Erwartungen der Stakeholder vollständig erfasst wurden. Gründe hierfür sind:

- Oft wird für manche Situationen die erwartete Reaktion des Systems nur generisch angegeben. Beispielsweise indem bei nicht vorgesehenen Eingaben an das System dieses in einen definierten Fehlerzustand übergeht. Diese generischen Anforderungen werden nicht für jede mögliche Situation, in welcher sie anwendbar sind, als Einzelanforderung erfasst, müssen aber in der Modellierung des Systemverhaltens berücksichtigt werden.
- Viele erwartete Reaktionen des Systems sind aus den explizit formulierten Anforderungen bereits implizit gegeben, da diese unter Berücksichtigung grundlegender Kenntnisse der Anwendungsdomäne offensichtlich sind. Derartige Annahmen werden aufgrund ihrer Trivialität nicht explizit als Anforderung erfasst.
- Häufig existieren in bestimmten Situationen mehrere, aus Sicht der Stakeholder gleichwertige, mögliche Reaktionen des Systems. Aus Sicht der Stakeholder gibt es keine Priorisierung der möglichen Varianten einer einzelnen Anforderung. Die Entscheidung, welche Systemreaktion korrekt ist, wird dem Ersteller der Spezifikation überlassen. Häufig wird bei der Modellierung des Gesamtsystemverhaltens deutlich, welche Variante unter Berücksichtigung des übrigen spezifizierten Systemverhaltens vorzuziehen ist.

Zumindest theoretisch ist es möglich, eine hinreichend große Menge an Szenarien anzugeben, welche bezüglich der syntaktischen Schnittstelle $(I \triangleright O)$ des Systems eindeutig zu einer vollständigen Zustandsmaschine führt. Hierfür wurden vielfach automatisierbare Verfahren vorgeschlagen: Krüger u. a. (1999) beschreiben dazu eine Methode zur Integration von *Message Sequence Charts (MSCs)* in Statecharts. Die Synthese von Verhaltensmodellen in Form von *Labelled Transition Systems (LTS)* aus MSCs unter zusätzlicher Verwendung von *High-Level-MSCs (hMSCs)* wurde dazu von Uchitel u. a. (2003) vorgeschlagen. Damas u. a. (2005) geben die Synthese eines

LTS aus positiven und negativen MSCs an. Bollig u. a. (2007) beschreiben dazu ein interaktives Lernverfahren vor um aus MSCs *Message-Passing-Automaten (MPA)* zu erstellen.

Um ein totale und deterministische Zustandsmaschine zu erhalten, setzen diese Ansätze entweder voraus, dass eine ausreichend große Menge an Szenarien definiert ist, welche eine Zustandsmaschine eindeutig bestimmt, oder es sind zusätzliche Informationen zur Erstellung der Zustandsmaschine notwendig. Im Falle der Methode von Uchitel u. a. (2003) ist dies die Angabe eines High-Level-MSCs, für den Ansatz von Bollig u. a. (2007) ist die Beantwortung der Lernfragen notwendig.

Da die Stakeholder nur selten derart umfangreiche und detaillierte Mengen von Szenarien angeben, beziehungsweise, wie oben erläutert, bewusst Entscheidungen dem Ersteller der Systemspezifikation überlassen, kann in der Praxis nur in wenigen Fällen allein auf Basis der durch die Stakeholder formulierten Anforderungen eine totale Verhaltensbeschreibung angegeben werden. In das Verhaltensmodell fließen somit im Regelfall auch zur Beschreibung des Verhaltens aus Nutzersicht zusätzliche Informationen ein, welche durch die von den Stakeholdern angegebenen Anforderungen und Szenarien nicht bestimmt waren. Für die in dieser Arbeit vorgeschlagene Methode zur Testfallgenerierung ist es dabei unerheblich, ob diese zusätzlichen Informationen direkt manuell zu dem Verhaltensmodell hinzugefügt wurden, oder ob das Verhaltensmodell aus Szenarien und gegebenenfalls weiteren Informationen synthetisiert wurde.

Aufgrund der zusätzlichen Informationen im Verhaltensmodell bietet eine Testfallermittlung aus diesem Modell einen wesentlichen Mehrwert gegenüber der ausschließlichen Verwendung der Szenarien. Da das Modell die Menge aller möglichen Szenarien beschreibt, ist auch die kombinierte Ausführung von Szenarien der Stakeholder enthalten. Gleichzeitig ist das Modell über die Schnittstelle $(I \triangleright O)$ beschrieben, welche nur abstrakte Ein- und Ausgaben aus Sicht der Systemumgebung umfasst (siehe Abschnitt 5.2.1), damit ist das Modell abstrakt genug um zur Testfallgenerierung geeignet zu sein.

5.4. Zusammenfassung

Anforderungen, welche Gegenstand des Tests sein sollen, werden in folgenden Schritten spezifiziert: Zunächst sind in Anforderungsdokumenten oder durch die Stakeholder einzelne funktionale Anforderungen zu identifizieren. Damit diese für eine anforderungsorientierte Testfallgenerierung verwendet werden können, ist eine formale Repräsentation der Anforderungen notwendig. Gleichzeitig muss diese formale Darstellung von Anforderungen aber auch durch alle Stakeholder validierbar sein. Aus diesem Grund scheiden für viele Projekte komplexe mathematische Formalismen aus. Stattdessen werden in der vorliegenden Arbeit exemplarische Szenarien verwendet, um Anforderungen zu formalisieren. Dies stellt die erste Phase der Spezifikation dar.

Ein vollständige Beschreibung der Gesamtanforderung an das System – im Sinne einer totalen Funktion – wird erst durch die Angabe eines Verhaltensmodells erreicht. Dies ist die zweite Phase der Anforderungsspezifikation. Für dieses Modell wird gefordert, dass die zuvor identifizierten Szenarien Abläufe im Modell sind. Das Modell muss dabei von dem zu testenden System abstrahieren. Zur Abstraktion ist die Schnittstelle des Systems entsprechend zu beschreiben, so dass diese ausschließlich für die Umgebung relevante Informationen umfasst. Details der technischen Realisierung bleiben in der Spezifikation unberücksichtigt. Da die Realisierung mit Fehlern behaftet sein kann, ist das Modell auch als Testmodell geeignet.

6. Anforderungsorientierte Testfallermittlung

Dieses Kapitel beschreibt die Technik einer anforderungsorientierten, modellbasierten Testfallgenerierung. Wie in Kapitel 5 dargestellt wurde, basiert diese Methode auf einer zweistufigen Spezifikation:

- Einer Menge von einzelnen Anforderungen, zu welcher jeweils (mindestens) ein Szenario angegeben ist.
- Einem abstrakten Verhaltensmodell, welches das Systemverhalten vollständig beschreibt.

Anforderungsorientiert ist hier so zu verstehen, dass die Testfallgenerierung gezielt jedes der definierten einzelnen Anforderungsszenarien adressiert. Dabei soll aber nicht allein das jeweilige Szenario Grundlage für die Testfallgenerierung sein, sondern die *modellbasierte Testfallgenerierung* soll auf Basis der Informationen im Verhaltensmodell des Gesamtsystems erfolgen. Die Methode in dieser Arbeit kombiniert dazu die funktionale Auswahl von Testfällen hinsichtlich einzelner Anforderungen mit einer strukturellen oder statistischen Auswahl von Testfällen aus einem Verhaltensmodell.

6.1. Kombination funktionaler und struktureller Testfallauswahl

In den Abschnitten 2.3 und 2.4 wurde bereits dargestellt, dass zur modellbasierten Testfallgenerierung strukturelle und statistische sowie funktionale Testfallspezifikationen verwendet werden können. Die Anwendung von strukturellen beziehungsweise statistischen Testfallspezifikationen einerseits und funktionalen Testfallspezifikationen andererseits ist jeweils in unterschiedlichen Situationen geeignet.

6.1.1. Strukturelle und statistische Testfallspezifikationen

Bei einer strukturellen Testfallspezifikation werden Eigenschaften über die Modellelemente des Testmodells angegeben. Ziel der Testfallgenerierung ist es, eine Testsuite zu finden, welche diese Eigenschaften erfüllt. Strukturelle Testfallspezifikationen weisen gegenüber funktionalen Testfallspezifikationen folgende Vorteile auf:

Automatisierbarkeit Strukturelle Testfallspezifikationen erlauben einen höheren Grad an Automatisierung des Testprozesses, da die Auswahlkriterien, zum Beispiel Zustands- oder Transitionsüberdeckung, allein über die Elemente der Modellierungssprache definiert sind. Damit sind strukturelle Testfallspezifikationen unmittelbar auf alle Systeme anwendbar, zu welchen ein Testmodell in einer geeigneten Sprache vorliegt. Eine system- oder projektspezifische Definition der Testfallspezifikation ist nicht erforderlich. Aus dem Testmodell können somit Testfälle sofort ohne zusätzliche projektspezifische Informationen automatisch gewonnen werden.

Globale Betrachtung des Testmodells Strukturelle Testfallspezifikationen sind Eigenschaften der Menge der Modellelemente, wobei die Eigenschaften allgemein über die Typen (etwa Transitionen oder Zustände) der Modellelemente ausgedrückt werden und nicht über die spezifischen Instanzen dieser Typen in einem Testmodell. Damit betrachten strukturelle Testfallspezifikationen alle Teile des Testmodells und die damit beschriebenen Ausführungspfade gleichermaßen. Es ist somit sichergestellt, dass nicht allein Teilfunktionalitäten im Test berücksichtigt werden. Insbesondere sind auch Ausführungspfade, welche in den einzelnen Anforderungen nicht explizit adressiert sind, im potentiellen Testumfang enthalten.

Für statistische und stochastische Testfallspezifikationen gelten diese Vorzüge gleichermaßen. Alle diese, vor allem auf der Struktur des Testmodells, nicht aber auf den durch die Stakeholder identifizierten einzelnen Anforderungen basierenden Testfallspezifikationen gehen mit diesen Nachteilen einher:

Fehlender Bezug zu Anforderungen Zunächst fehlt den mittels strukturellen oder statistischen Testfallspezifikationen gewonnen Testfällen der Bezug zu den durch die Stakeholder geforderten Einzelanforderungen. Das ist besonders dann von Bedeutung, wenn die Stakeholder den Nachweis des Tests in Form einer expliziten Angabe einer Menge von Testfällen zu jeder einzelnen festgelegten Anforderung erwarten. Aus der Menge von Testfällen ist ein *Tracing* zu den Einzelanforderungen nicht unmittelbar möglich.

Mangelnde Flexibilität Weiter ist es bei strukturellen und meist auch bei stochastischen Testfallspezifikationen dem Testingenieur nicht oder nur eingeschränkt möglich, die Menge der generierten Testfälle zu beeinflussen. System- und projektspezifische Besonderheiten können so in der Testfallauswahl nur unzureichend berücksichtigt werden. So sollen beispielsweise Anforderungen und Teilfunktionen, bei welchen eine erhöhte Fehlerwahrscheinlichkeit angenommen wird, verstärkte Berücksichtigung in den Testfällen finden.

Diesen Nachteilen kann zum Teil dadurch begegnet werden, dass mehrere Testmodelle erstellt werden, welche unterschiedliche Teilfunktionalitäten des Systems beschreiben.

Neben dem Erstellungs- und Pflegeaufwand dieser Modelle ist hierbei zu berücksichtigen, dass durch die Wahl der Grenzen der Funktionsumfänge in den einzelnen Modellen der Teilfunktionen bereits eine funktionale Beschränkung der möglichen Testfälle vorgenommen wird. Somit wird implizit eine Mischform aus strukturellen und funktionaler Testfallspezifikation angewendet. Analog gilt dies, falls die Testfallgenerierung durch Priorisierungen oder Gewichtungen auf den Instanzen der Modellelemente im Testmodell gesteuert wird.

6.1.2. Funktionale Testfallspezifikation

Bei einer funktionalen Testfallspezifikation sind dagegen Eigenschaften von Ausführungspfaden des Systems, vor allem Eigenschaften der Eingaben an das System, angegeben. Das Modell dient dann zur Vervollständigung der Eingaben mit Ausgaben – oder zur Vervollständigung mit zusätzlichen Eingaben, damit vollständige Testläufe erzielt werden. Die Motivation funktionaler Testfallspezifikation rührt im Wesentlichen daher, die formulierten Anforderungen im Test gezielt zu adressieren sowie die Intuition des Testingenieurs nutzen zu können und Tests flexibel an die Projektsituation anpassen zu können

Fokussierung auf Anforderungen Funktionale Testfallspezifikationen leiten sich unmittelbar aus den von den Stakeholdern geforderten Einzelanforderungen ab. Die damit ermittelten Testfälle liefern dann unmittelbar den Nachweis eines Tests der entsprechenden Anforderungen.

Anpassbar an Projekterfordernisse Funktionale Testfallspezifikationen sind individuell für jedes Projekt beziehungsweise System festzulegen. Damit können die spezifischen Erfordernisse in einem Projekt gezielt berücksichtigt werden, etwa indem kritischere Systemteile verstärkt in den Testfallspezifikationen berücksichtigt werden.

Intention des Testingenieurs Es obliegt dem Testingenieur, ausgehend von der funktionalen Beschreibung einer Anforderung eine Testfallspezifikation zu erstellen. Der Testingenieur kann seine Intention und Erfahrung nutzen, um möglichst aussagekräftige Testfälle zu den Anforderungen zu erzielen.

Offensichtlich können funktionale Testfallspezifikationen die Vorteile struktureller Testfallspezifikationen nicht nutzen. Es zeigen sich folgende Nachteile:

Manueller Aufwand Im Gegensatz zu strukturellen Kriterien kann hier keine generische Testfallspezifikation für eine Vielzahl von Systemen verwendet werden. Für jedes System müssen manuell aus den Anforderungen entsprechende funktionale Testfallspezifikationen abgeleitet werden.

Isolierte Betrachtung von Anforderungen Die funktionale Testfallspezifikation birgt weiterhin die Gefahr in sich, dass Anforderungen im Test nur isoliert betrachtet werden. Wie in Abschnitt 5.3 dargestellt wurde, beschreiben Einzelanforderungen nur Teile des Gesamtverhaltens des Systems. Insbesondere ist oftmals das Verhalten des Systems bei der Kombination mehrerer Anforderungen innerhalb einer Ausführung nur unzureichend beschrieben. Funktionale Testfallspezifikationen können nicht sicherstellen, dass auch derartige nicht explizit formulierte Szenarien gleichermaßen im Test berücksichtigt werden. In extremen Fällen können damit Teile des Modells – und damit Teile des Gesamtverhaltens – vollständig ohne Einfluss auf die Testfallermittlung bleiben.

6.1.3. Vorteile aus der Kombination struktureller und funktionaler Testfallspezifikation

Da, wie eben erläutert, sowohl funktionale als auch strukturelle Testfallspezifikationen gleichermaßen mit Nachteilen behaftet sind, kombiniert die hier vorgeschlagen Methode zur Testfallgenerierung beide Arten. Damit werden die Vorteile der verschiedenen Testfallspezifikationen vereint. Somit wird

- eine weitgehende Automatisierung,
- ein inhärentes Tracing zu den einzelnen Anforderungen,
- eine einheitliche Berücksichtigung aller Modellteile, insbesondere auch die Kombination von Einzelanforderungen, sowie
- eine flexible Anpassbarkeit an die Projektsituation und Intention des Testingenieurs

der Testfallermittlung erreicht. Wie bereits in Kapitel 4 skizziert, werden dazu eine Klassifikation von Szenarien und deren Gewichtung vorgenommen, Teilmodelle anhand dieser Gewichtung gebildet und die Testfälle aus diesen Teilmodellen generiert.

6.2. Klassifikation von Szenarien

Ein wesentliches Element der hier vorgestellten Methode zur Testfallermittlung ist die Klassifikation von Anforderungen, beziehungsweise der Szenarien, welche diese Anforderungen repräsentieren, hinsichtlich ihrer Bedeutung für den Test. Dem Testingenieur soll damit die Möglichkeit gegeben werden, die Anforderungen unterschiedlich zu priorisieren. Dazu wird im Folgenden zunächst dargestellt, welche Faktoren für eine unterschiedliche Priorisierung von Anforderungen sprechen, um weiter die Klassifikation von Anforderungen innerhalb der anforderungsorientierten modellbasierten Testfallgenerierung zu erläutern.

6.2.1. Unterschiedliche Priorisierung von Anforderungen für den Test

Eine undifferenzierte Betrachtung der Anforderungen an ein System, wobei jede einzelne Anforderung als gleichermaßen bedeutend für den beabsichtigten Nutzen, welchen das System erbringen soll, angesehen wird, ist nicht angebracht. Boehm (2003) argumentiert etwa für eine am Nutzen orientierte Softwareentwicklung *(Value-Based Software Engineering)*, wobei unter anderem Anforderungen der Stakeholder nach Ihren Nutzen beziehungsweise Wert für die Stakeholder zu priorisieren sind. In ähnlicher Weise treten auch Karlsson (1996); Karlsson und Ryan (1997) für eine kosten- und nutzenorientierte Bewertung von Anforderungen ein. Diese Ansätze sind vor allem vor dem Hintergrund der Auswahl von Anforderungen, etwa zur Releaseplanung, entworfen worden. Im Bezug auf einen anforderungsorientierten Test ist aber ebenso eine vergleichbare Priorisierung von Anforderungen angebracht. So wurde etwa im Sinne einer nutzen-orientierten Softwareentwicklung bereits von Srikanth und Williams (2005); Srikanth (2005) eine Vorgehensweise zur Priorisierung von Anforderungen für den Softwaretest vorgeschlagen. Dabei werden die *Volatilität*, die *vom Auftraggeber vorgenommene Priorisierung*, die *vom Entwickler eingeschätzte Komplexität der Implementierung* und die Fehlerträchtigkeit (basierend auf gemeldeten Fehlern in früheren Versionen des Systems) einer Anforderung als Faktoren zur Priorisierung von Anforderungen verwendet. Allgemeiner lassen sich Faktoren für die Priorisierung von Anforderungen im Test in drei Kategorien einteilen:

- das *Fehlerpotenzial* einer Anforderung, also Annahmen über die Wahrscheinlichkeit, dass die Implementierung einer Anforderung fehlerhaft vorgenommen wurde
- der *Anwendernutzen* einer Anforderung, also die Bedeutung der Anforderungen für den Anwender oder Auftraggeber
- das *Entwicklungsstadium* des Systems, also Phasen im Projektverlauf, in welchen eine Anforderung von besonderer Bedeutung für den Test ist

Tabelle 6.1 ordnet die von Srikanth (2005) genannten Faktoren in diese Kategorien ein und gibt beispielhaft weitere Faktoren an, welche für eine Priorisierung von Anforderungen im Test angewendet werden können.

Hinsichtlich des Fehlerpotenzials von Anforderungen können etwa die Volatilität einer Anforderung, die Komplexität der Implementierung einer Anforderung oder der Ausbildungsstand und die Erfahrung der Entwickler, welche eine Anforderung umgesetzt haben, zur Priorisierung verwendet werden. Die *Volatilität* gibt an, wie häufig eine Anforderung nach ihrer initialen Formulierung abgeändert wurde. Wie von Malaiya und Denton (1999) gezeigt wurde, erhöhen insbesondere späte Änderungen an den Anforderungen die Fehlerdichte in den betroffenen Komponenten. Einschätzungen von Entwicklern hinsichtlich der *Komplexität der Implementierung* der unterschiedlichen Anforderungen sind ein weiteres Kriterium, um Anforderungen hinsichtlich ihres

Tabelle 6.1.: Beispiele für die Klassifikation von Anforderungen

Klassifikationskategorie	Bewertungsfaktoren (Beispiele)
Fehlerpotenzial	• Volatilität • Komplexität der Implementierung • Erfahrung der Entwickler
Anwendernutzen	• Häufigkeit der Nutzung • Auswirkungen von Fehlern
Entwicklungsstadium	• Releaseumfang • Ausprägung in einer Produktlinie

Fehlerpotenzial zu bewerten. Schließlich können organisatorische Aspekte herangezogen werden, indem etwa nach dem *Kenntnisstand und der Erfahrung* der Entwickler unterschieden wird, welche die einzelnen Anforderungen realisiert haben.

Auswirkungen auf den Anwendernutzen ergeben sich einerseits aus der Häufigkeit der Nutzung, andererseits auch durch die Schwere der negativen Auswirkungen von Fehlern. Entsprechend der in Kapitel 3 diskutierten Qualität von Testfällen gilt dies bezogen auf Anforderungen entsprechend: Fehlerhaft realisierte Anforderungen, welche sehr *häufig ausgeführt* werden, werden von dem Anwender besonders stark wahrgenommen und mindern die Benutzbarkeit des Systems. Andererseits mindert die fehlerhafte Umsetzung von Anforderungen, auch wenn diese selten ausgeführt werden, den Gesamtnutzen des Systems erheblich, wenn die *Fehlfunktion schwerwiegende Folgen* in sich birgt.

Auch durch das jeweilige Entwicklungsstadium im Projektverlauf können bestimmte Anforderungen von höherer Bedeutung für den Test sein als andere. Beispielsweise ist dies der Fall, wenn in einer Freigabeversion (Release) bereits mehr Funktionen implementiert sind, als zu diesem Zeitpunkt gefordert. Dann sollen vor allem die zu diesem Zeitpunkt geforderten Anforderungen verstärkt getestet werden. Ähnlich trifft dies auf verschiedene Varianten eines Systems einer Software-Produktlinie zu – hier sind für eine Variante vor allem die variantenspezifischen Anforderungen von Interesse.

6.2.2. Bildung von Anforderungsklassen

Wie im vorhergehenden Abschnitt erläutert, sind nicht alle Anforderungen von gleicher Bedeutung für den Test. Nun wird das Vorgehen erläutert, wie zur anforderungsorientierten Testfallgenerierung unterschiedliche Anforderungsklassen gebildet werden.

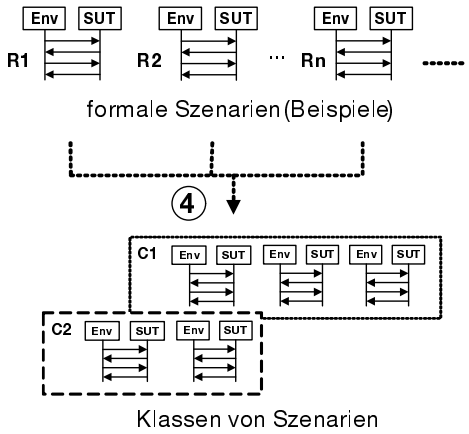

Abbildung 6.1.: Schritt 4 des vorgeschlagenen Entwicklungs- und Testprozesses: Klassifikation von Szenarien (Ausschnitt aus Abbildung 4.1)

Es entspricht dem in Abbildung 6.1 dargestellten vierten Schritt des vorgeschlagenen Testprozesses.

Ziel dieses Schritts ist es, die Klassen unterschiedlicher Anforderungen zu definieren und jedes Szenario $r \in REQ$ diesen Klassen zuzuordnen. Die Menge REQ beschreibt wiederum die Menge aller Szenarien, wie sie in Kapitel 5 eingeführt wurde.

Die Definition der Anforderungsklassen erfolgt hinsichtlich eines oder mehrerer der im vorhergehenden Abschnitt aufgezählten Kriterien. Die Wahl das Kriteriums ist nach der Intention des Software-Tests in der jeweiligen Projektsituation zu treffen. Die Menge aller Klassen, die Klassifikation, wird im Folgenden mit

$$CLAS = \{c_1, c_2, \ldots, c_n\}$$

bezeichnet, wobei $c_1, c_2, \ldots, c_n$ die Bezeichner der n gewählten Klassen sind. In Tabelle 6.2 sind beispielhaft Klassen für die bereits zuvor genannten Faktoren dargestellt. Die Menge $CLAS$ ist so zu wählen, dass jedes Szenario von dieser Klassifikation erfasst wird. Meist ist es möglich, die Klassifikation so zu wählen, dass jedes Szenario eindeutig einer Klasse zugeordnet werden kann. Dies ist jedoch nicht grundsätzlich nötig. In manchen Fällen ist die Klassifikation einfacher zu vollziehen, wenn einer Anforderung eine Menge von Klassen zugeordnet wird. Dies ist beispielsweise eine angebrachte Vorgehensweise, um in einer Produktlinien-Entwicklung eine Anforderung mehreren Varianten zuzuordnen. Um auch solche Klassifikationen zu erlauben, wird hierzu jedem Anforderungsszenario eine Teilmenge von $CLAS$ zugeordnet.

Die Klassifikation der Szenarien $r \in REQ$ gemäß der Menge an Klassen $CLAS$ wird mit der Funktion

$$\text{classReq} \in REQ \to \mathbb{P}(CLAS)$$

Tabelle 6.2.: Klassen von Anforderungen nach unterschiedlichen Klassifikationen. (Hier sind Beispiele angegeben, je nach Situation werden die Klassen geeignet festgelegt. Wie etwa in Beispiel 15)

Klassifikation	**Klassen**
Volatilität	$CLAS = \{initial, verändert, hinzugenommen\}$
Komplexität der Implementierung	$CLAS = \{trivial, normal, hoch\}$
Erfahrung der Entwickler	$CLAS = \{wenig, groß\}$
Häufigkeit der Nutzung	$CLAS = \{Grundfunktion, regelmäßig, Sonderfall\}$
Auswirkungen von Fehlern	$CLAS = \{normal, kritisch\}$
Releaseumfang	$CLAS = \{früheresRel, aktuellsRel, künftigesRel\}$
Ausprägung in einer Produktlinie	$CLAS = \{Domäne, VarianteA, VarianteB\}$

bezeichnet, wobei für alle Szenarien $r \in REQ$

$$\text{classReq}(r) \neq \{\}$$

gelten muss.

Beispiel 15 (Anforderungsklassen)

Im Fallbeispiel des Fensterhebers wurden mit Beispiel 13 (Seite 105) neue Szenarien $REQ_{substitutes} = \{r_4', r_4'', r_9', r_9''\}$ erst in einer späteren Entwurfsphase hinzugefügt. Der Test soll nun insbesondere diese geänderten Anforderungen fokussieren. Daher werden als Klassen

$$CLAS = \{initial, added\}$$

gewählt. Mit *initial* werden dabei die initialen und beibehaltenen Szenarien $REQ_{initial} \setminus REQ_{dismissed}$ klassifiziert, mit *added* die neu hinzugefügten (geänderten) Szenarien $REQ_{substitutes}$. Für die Funktion classReq ergibt sich somit:

$$\text{classReq}(r) = \begin{cases} \{initial\} & \text{falls } r \in \{r_1, r_2, r_3, r_5, r_6, r_7, r_8, r_{10}\} \\ \{added\} & \text{falls } r \in \{r_4', r_4'', r_9', r_9''\} \end{cases}$$

In diesem Fall ist die Klassifikation eindeutig.

Je nach Art der Klassifikation kann diese schon während der Erhebung von Anforderungen (beispielsweise aufgrund Kritikalität oder Nutzungshäufigkeit) festgelegt werden oder aber auch erst im Rahmen der Testerstellung, wenn etwa das Fehlerpotenzial beachtet werden soll. Zur Definition der Funktion classReq(r) ist eine tabellarische Darstellung geeignet.

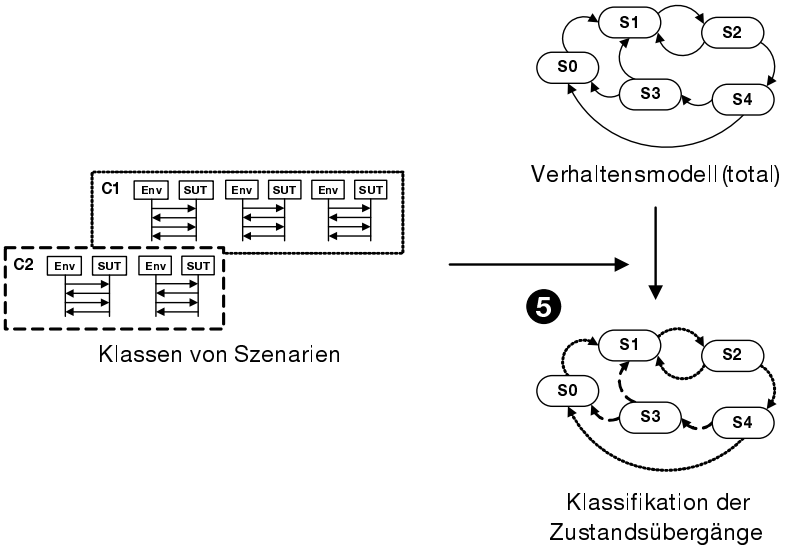

Abbildung 6.2.: Schritt 5 des vorgeschlagenen Entwicklungs- und Testprozesses: Übertragung der Klassifikation der Szenarien in das Verhaltensmodell (Ausschnitt aus Abbildung 4.1)

6.3. Übertragung der Klassifikation in das Verhaltensmodell

Die Klassifikation der Szenarien lässt sich in das in Verhaltensmodell übertragen. Dies ist der sechste Schritt des gesamten Entwicklungs- und Testprozesses, siehe hierzu auch Abbildung 6.2. Dieser Schritt benötigt keine weiteren Eingaben und lässt sich somit vollständig automatisieren.

Wie in Abschnitt 5.3.3 beschrieben wurde, muss jedes Szenario von dem erstellten Verhaltensmodell, welches als Testmodell dient, akzeptiert werden. Das Szenario beschreibt somit einen Ausführungspfad, also eine Folge von Zustandsübergängen, in diesem Modell. Zustandsübergänge (Transitionen) können somit Szenarien, in welchen sie ausgeführt werden, zugeordnet werden. Damit können die Klassen der Szenarien, in welchen eine Transition schaltet, auch an der Transition annotiert werden.

Da das erstellte Verhaltensmodell deterministisch ist, ist jede Transition in einem Modell $M = (I, O, S, \delta, \lambda)$ eindeutig durch Angabe ihres Ausgangszustands $s \in S$ und der Eingabe $in \in I$ identifizierbar. Für eine Mealy-Maschinen ist damit die Menge ihrer Transitionen durch das Kreuzprodukt

$$S \times I$$

gegeben, das alle Paare von Zuständen und möglichen Eingabesymbolen beschreibt. Um die Menge aller Transitionen zu ermitteln, welche bei Ausführung eines Szenarios

$r \in REQ$ schalten, reicht es somit aus, die Eingaben $\mathsf{in}.r \in I^*$ des Szenarios zu betrachten. Hierzu wird für eine Mealy-Maschine $M = (I, O, S, \delta, \lambda)$ die die Funktion

$$\text{trans} \in S \to I^* \to \mathbb{P}(S \times I)$$

eingeführt. Diese ist wie folgt definiert:

$$\text{trans}[s](ins) \stackrel{\text{def}}{=} \begin{cases} \{\} & \text{falls } ins = \langle\rangle \\ \{(s, in)\} \cup \text{trans}[s'](ins') & \text{falls } ins = \langle in \rangle \frown ins' \wedge \delta(s, in) = s' \end{cases}$$

Mithilfe dieser Funktion wird die Menge aller Transitionen angegeben, welche ausgehend von einem Initialzustand bei der Ausführung des Szenarios r schalten. Da für das weitere Verfahren die Reihenfolge der Ausführung der Transitionen unerheblich ist, werden die Transitionen als Menge und nicht als Sequenz angegeben. Wie in Abschnitt 5.2.3 erläutert wurde, ist im Allgemeinen nicht das vollständige Szenario für die zu Grunde liegende Anforderung charakteristisch: Ein Teil des Szenarios dient als Vorbedingung, um das System ausgehend vom Initialzustand s_0 in einen Zustand zu bringen, in welchem die Anforderung relevant wird. Für die Übertragung der Klassifikation der Anforderungen in das Modell soll allein der bestimmende Teil der Szenarien von Bedeutung sein. Dazu wird eine spezifischere Funktion

$$\text{transDet} \in S \to REQ \to \mathbb{P}(S \times I)$$

verwendet, welche auf die Funktion trans zurückgreift. Für ein Szenario $r \in REQ$ mit Vorbedingung $\mathsf{pre}.r = (ins_{\text{pre}}, outs_{\text{pre}})$ und bestimmenden Teil $\mathsf{det}.r = (ins_{\text{det}}, outs_{\text{det}})$ ist diese Funktion gegeben durch

$$\text{transDet}[s_0](r) = \text{trans}[\Delta(s_0, ins_{\text{pre}})](ins_{\text{det}}).$$

Mit $\Delta(s_0, ins)$ wird dabei wiederum der Zustand angegeben, welcher nach Ausführung der Folge von Eingaben ins ausgehend von s_0 erreicht wird.

Beispiel 16 (Transitionen eines Szenarios)

Im Modell des Fensterhebers aus Beispiel 14 (Seite 107) schalten für Anforderung r_4'' (vgl. Beispiel 13, Seite 105) folgende Transitionen:

$$\mathsf{trans}[\mathit{Idle}](\mathsf{in}.r_4') = \{t_1, t_{16}, t_{20}, t_{41}\}$$
$$= \left\{ \left(\mathit{Idle}, \begin{pmatrix}\mathit{open}\\ \mathit{middle}\end{pmatrix}\right), \left(\mathit{Down}, \begin{pmatrix}\mathit{open}\\ \mathit{moving}\end{pmatrix}\right), \left(\mathit{Down}, \begin{pmatrix}\mathit{close}\\ \mathit{moving}\end{pmatrix}\right), \left(\mathit{Pause}, \begin{pmatrix}\mathit{close}\\ \mathit{middle}\end{pmatrix}\right) \right\}$$

$$\mathsf{transDet}[\mathit{Idle}](\mathsf{in}.r_4') = \{t_{20}, t_{41}\} = \left\{ \left(\mathit{Down}, \begin{pmatrix}\mathit{close}\\ \mathit{moving}\end{pmatrix}\right), \left(\mathit{Pause}, \begin{pmatrix}\mathit{close}\\ \mathit{middle}\end{pmatrix}\right) \right\}$$

Werden auf diese Weise zu allen einzelnen Szenarien aus REQ die schaltenden Transitionen ermittelt, ergibt sich in der Umkehrung ebenfalls für jede Transition die Menge der Szenarien, zu deren Umsetzung im Modell die Transition beiträgt (dabei ist lediglich der bestimmende Teil des Szenarios von Interesse). Dazu sei die Funktion

$$\text{contrib} \in S \to S \times I \to \mathbb{P}(REQ)$$

mit

$$\text{contrib}[s_0](s, in) = \{r \in REQ | (s, in) \in \text{transDet}[s_0](r)\}$$

eingeführt, welche zu einer Transition $(s, in) \in S \times I$ die Menge der Szenarien $r \in REQ$ angibt, bei deren Ausführung die Transition (s, in) im bestimmenden Teil der Anforderung schaltet. In diesem Sinn trägt die Transition (s, in) zur Erfüllung von r bei.

Da nun die Szenarien, zu deren Erfüllung eine Transition beiträgt, ermittelt werden können, kann die Klassifikation der Szenarien für die Transition übernommen werden. Die Funktion

$$\text{classTrans} \in S \to (S \times I) \times (REQ \to CLAS) \to \mathbb{P}(CLAS)$$

mit

$$\text{classTrans}[s_0]((s, in), \text{classReq}) = \bigcup_{r \in \text{contrib}[s_0](s,in)} \text{classReq}(r)$$

gibt dazu die Menge der Anforderungsklassen $\subseteq CLAS$ jener Szenarien, zu deren Erfüllung eine Transition beiträgt. Dies ist die Vereinigung der Anforderungsklassen der entsprechenden Szenarien. Auf diese Weise ist die Klassifikation der Szenarien auf das Verhaltensmodell übertragen. Somit werden die Transitionen mit den Klassen der Szenarien assoziiert, in welchen die Transitionen ausgeführt werden. Dies wird im Weiteren zur Bildung von Teilmodellen verwendet. Durch diese Assoziation werden die einzelnen Anforderungsklassen im Modell identifiziert, um bei der Bildung von Teilmodellen ihre Gewichtung berücksichtigen zu können. Dies wird im Anschluss genauer erläutert.

Beispiel 17 (Klassifikation der Transitionen)

Tabelle 6.3 gibt zu allen Transitionen im Modell der Fensterheberfunktion (siehe Tabelle 5.6 auf Seite 107) die Szenarien an, zu deren Erfüllung eine Transition jeweils beiträgt ($\text{contrib}[s_0](tr)$) sowie die daraus resultierende Menge an Anforderungsklassen, welcher eine Transition zugeordnet ist ($\text{classTrans}[s_0](tr, \text{classReq})$). Die Klassifikation classReq wurde dabei aus Beispiel 15 übernommen.

Wie aus der Tabelle ersichtlich ist, tragen viele Transitionen zu keinem Szenario bei. Dies ist nicht verwunderlich, da die Szenarien nur eine partielle Beschreibung der Funktion hinsichtlich ihrer vollständigen Ein-/Ausgabeschnittstelle darstellen. Erst durch die Vervollständigung der Spezifikation

durch das Verhaltensmodell (vgl. Abschnitt 5.3) wurde die Funktion total beschrieben. Die nicht klassifizierten Transitionen, sind diejenigen, welche zur Totalisierung hinzugefügt wurden.

Tabelle 6.3.: Klassifikation der Transitionen im Modell der Fensterheberfunktion

tr	contrib$[s_0](tr)$	classTrans$[s_0]$ $(tr, \text{classReq})$	tr	contrib$[s_0](tr)$	classTrans$[s_0]$ $(tr, \text{classReq})$
t_1	$\{r_1, r_9''\}$	$\{initial, added\}$	t_{25}	$\{\}$	$\{\}$
t_2	$\{\}$	$\{\}$	t_{26}	$\{\}$	$\{\}$
t_3	$\{\}$	$\{\}$	t_{27}	$\{\}$	$\{\}$
t_4	$\{\}$	$\{\}$	t_{28}	$\{r_9', r_9''\}$	$\{added\}$
t_5	$\{r_6, r_4''\}$	$\{initial, added\}$	t_{29}	$\{\}$	$\{\}$
t_6	$\{\}$	$\{\}$	t_{30}	$\{r_8\}$	$\{initial\}$
t_7	$\{\}$	$\{\}$	t_{31}	$\{\}$	$\{\}$
t_8	$\{\}$	$\{\}$	t_{32}	$\{r_6\}$	$\{initial\}$
t_9	$\{\}$	$\{\}$	t_{33}	$\{\}$	$\{\}$
t_{10}	$\{\}$	$\{\}$	t_{34}	$\{\}$	$\{\}$
t_{11}	$\{\}$	$\{\}$	t_{35}	$\{\}$	$\{\}$
t_{12}	$\{\}$	$\{\}$	t_{36}	$\{r_7\}$	$\{initial\}$
t_{13}	$\{r_5\}$	$\{initial\}$	t_{37}	$\{r_9'\}$	$\{added\}$
t_{14}	$\{\}$	$\{\}$	t_{38}	$\{\}$	$\{\}$
t_{15}	$\{r_3\}$	$\{initial\}$	t_{39}	$\{\}$	$\{\}$
t_{16}	$\{r_1\}$	$\{initial\}$	t_{40}	$\{\}$	$\{\}$
t_{17}	$\{\}$	$\{\}$	t_{41}	$\{r_4'\}$	$\{added\}$
t_{18}	$\{\}$	$\{\}$	t_{42}	$\{\}$	$\{\}$
t_{19}	$\{\}$	$\{\}$	t_{43}	$\{\}$	$\{\}$
t_{20}	$\{r_4', r_4''\}$	$\{added\}$	t_{44}	$\{\}$	$\{\}$
t_{21}	$\{\}$	$\{\}$	t_{45}	$\{r_4'', r_9''\}$	$\{added\}$
t_{22}	$\{\}$	$\{\}$	t_{46}	$\{\}$	$\{\}$
t_{23}	$\{\}$	$\{\}$	t_{47}	$\{\}$	$\{\}$
t_{24}	$\{r_2\}$	$\{initial\}$	t_{48}	$\{\}$	$\{\}$

In Abbildung 6.3 ist die Klassifikation der Transitionen im Zustandsübergangsgraph der Fensterheberfunktion dargestellt.

Abschließend sei noch eine Anmerkung zu dem ersten Argument s_0 der oben eingeführten Funktionen gegeben: Dieser bezeichnet den initialen Zustand des System, welcher für die Beschreibung der Anforderungen gewählt wurde, vergleiche hierzu Abschnitt 5.2.3. Der Initialzustand s_0 ist somit eine Konstante – diese Notation wurde gewählt um die Konsistenz mit der angegeben Notation von Mealy-Maschinen zu wahren. Zudem vereinfacht sie die Angabe der Funktion trans.

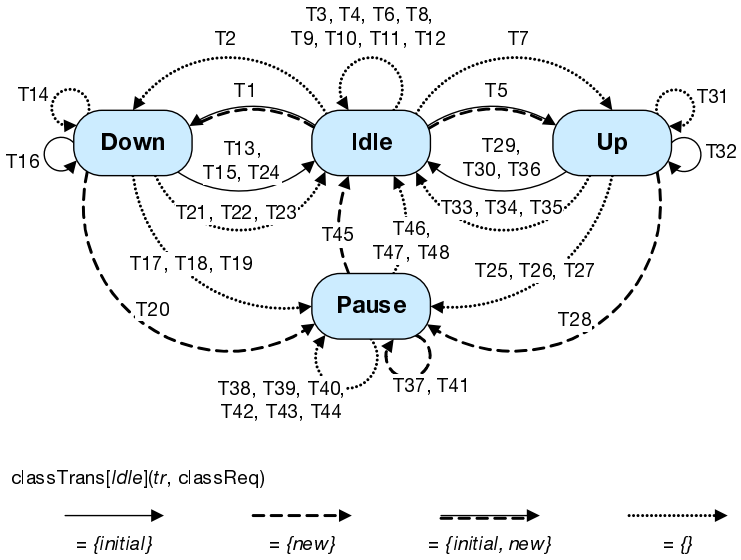

Abbildung 6.3.: Beispiel: Übertragung der Klassifikation der Anforderungsszenarien auf die Transitionen

6.4. Gewichtung der Anforderungsklassen

Die Klassifikation von Anforderungen, beziehungsweise von Szenarien, dient dazu, verschiedene Mengen von Anforderungen zu identifizieren, welche den Test unterschiedlich stark beeinflussen sollen. Somit ist allein die Klassifikation noch nicht ausreichend, um den Test entsprechend zu steuern. Weiter ist eine Gewichtung notwendig, welche diesen Einfluss auf die Testfälle quantifiziert. In dem in Kapitel 4 vorgeschlagenen Prozess ist dies der sechste Schritt, siehe auch Abbildung 6.4. Für die Gewichtung einer Klassifikation $CLAS$ ist eine Funktion

$$\text{weight} \in CLAS \rightarrow \mathbb{N}$$

anzugeben. Zu jeder Klasse $c_i \in CLAS$ ist also ein ganzzahliger Wert ≥ 0 anzugeben, welcher das Gewicht dieser Klasse repräsentiert.

6.4.1. Bedeutung der Gewichtung

Anforderungen und die dazu angegebenen Szenarien einer höher gewichteten Klasse haben – hinsichtlich der gewählten Klassifikation – eine größere Bedeutung für das zu testende System insgesamt. Daher ist es angebracht, Anforderungsszenarien aus Klassen mit hoher Gewichtung verstärkt im Zusammenspiel mit anderen Anforderungen zu testen. Die Wahl der Gewichtung wird somit durch die Frage

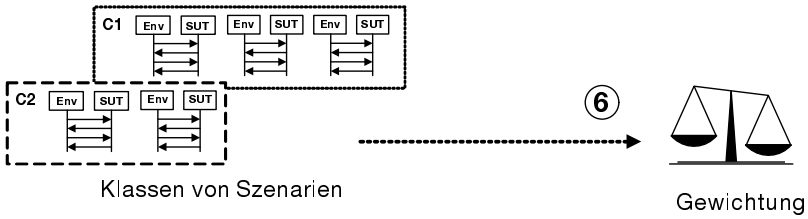

Abbildung 6.4.: Schritt 6 des vorgeschlagenen Entwicklungs- und Testprozesses: Gewichtung der Anforderungsklassen (Ausschnitt aus Abbildung 4.1)

> „Wie stark sollen die Anforderungen in dieser Klasse auch die Testfälle zu allen anderen Anforderungen beeinflussen?“

geleitet. Ziel der in dieser Arbeit beschriebenen Methode ist es ja, zu jedem einzelnen Szenario $r \in REQ$ jeweils eine eigene Menge von Testfällen zu ermitteln. Dabei soll aber das zu testende Szenario r nicht isoliert betrachtet werden, sondern es soll im Verbund mit den übrigen Anforderungen getestet werden. Die einzelne Anforderung, repräsentiert durch das Szenario r, stellt somit lediglich den zentralen Fokus der Testfallgenerierung dar. Wie stark die übrigen Anforderungen die Testfallgenerierung hinsichtlich r beeinflussen, ist abhängig von deren Gewichtung, welche über deren Anforderungsklasse bestimmt ist.

Die anforderungsorientierte Testfallgenerierung basiert darauf, dass nur ein auf die einzelne Anforderung bezogener Ausschnitt des gesamten Verhaltensmodell zur Testfallgenerierung für die einzelnen Anforderungen verwendet wird. Wie später in Abschnitt 6.5 dargestellt wird, beeinflusst die Gewichtung unmittelbar die Wahl dieses Ausschnitts aus dem Verhaltensmodell. Bestimmte Teile des Modells sind dabei eindeutig dem jeweiligen Szenario, welches die zu testende Anforderung repräsentiert, zuzuordnen. Eine Beschränkung allein auf diese Modellteile führt allerdings wiederum nur zu einem isolierten Test der Anforderung. Um die Anforderung im Verbund mit weiteren Anforderungen zu testen, müssen zusätzliche Modellteile in das Teilmodell übernommen werden. Die Gewichtung dient somit dazu, zu bestimmen, welche Teile des Modells neben den Modellteilen, welche eindeutig der Anforderung zuzuordnen sind, zusätzlich in das Teilmodell zur Testfallgenerierung übernommen werden.

6.4.2. Wahl der Gewichtung

Die geeignete Wahl der Gewichtungen hängt von projektspezifischen Faktoren ab, so dass keine generische Empfehlung für eine optimale Belegung der absoluten Werte der Gewichtung gegeben werden kann. Insbesondere ist entscheidend, welche Zielsetzung für die Testfallauswahl angestrebt wird:

- Steht eine stärker *funktionale Testfallauswahl* im Vordergrund, bei welcher Testfälle gesucht sind, welche nur geringe Abweichungen zu den ursprünglichen Szenarien aufweisen, oder

- soll sich die Testfallauswahl stärker an der *Struktur des gesamten Verhaltensmodell* orientieren, bei welcher die Szenarien nur einen vagen Ausgangspunkt für weitere Testfälle darstellen?

Im ersten Fall sind die Werte für die Gewichtungen niedrig zu setzen. Niedrige Gewichtungen bewirken, dass die Szenarien der niedrig gewichteten Anforderungsklassen kaum Einfluss auf die Testfallgenerierung zu einem einzelnen Szenarien $r \in REQ$ ausüben. Im zweiten Fall würden dagegen hohe Gewichtungen gewählt werden, welche dazu führen, dass weite Teile des übrigen Modells zur Testfallgenerierung für jedes einzelne Szenario verwendet würden. Die Gewichtungen beeinflussen unmittelbar die Wahl der Modellausschnitte, welche zur Testfallgenerierung für die einzelnen Szenarien verwendet werden. Die Wahl der Gewichtungen ist somit auch entscheidend von der Struktur des Modells abhängig. Um dies zu erläutern, soll jedoch zunächst die Bildung der Teilmodelle dargestellt werden. Anschließend wird in Abschnitt 6.7 nochmals detailliert auf die Wahl der Gewichtung eingegangen.

Meist wird hier ein iteratives Vorgehen notwendig sein: Nach einer initialen Belegung der Gewichte der Anforderungsklassen werden Testfälle generiert. Durch ein anschließendes Review der Testfälle (vgl. Abschnitt 6.7) wird gegebenenfalls ein Optimierungsbedarf der Gewichtung ermittelt. Der Testingenieur hat somit die Möglichkeit, die Testfallmenge derart zu beeinflussen, dass sie seiner Intention einer sinnvollen Auswahl von Testfällen entspricht.

6.4.3. Gewichtung nicht klassifizierter Transitionen

Wie im folgenden Abschnitt 6.5 ausführlich erläutert wird, dient die Gewichtung vor allem dazu, die Transitionen zu bestimmen, welche in ein anforderungsspezifisches Teilmodell übernommen werden.

Da einer Transition nur dann eine Anforderungsklasse zugeordnet wird, wenn die Transition im bestimmenden Teil eines Anforderungsszenarios schaltet, sind im Allgemeinem im Modell auch unklassifizierte Transitionen vorhanden. Um auch diese Transitionen zu einem gewissen Grad in die Teilmodelle einfließen zu lassen, wird hierfür ein zusätzlicher Wert zur Gewichtung dieser Transitionen verwendet. Näheres dazu wird in Abschnitt 6.5.1 erläutert.

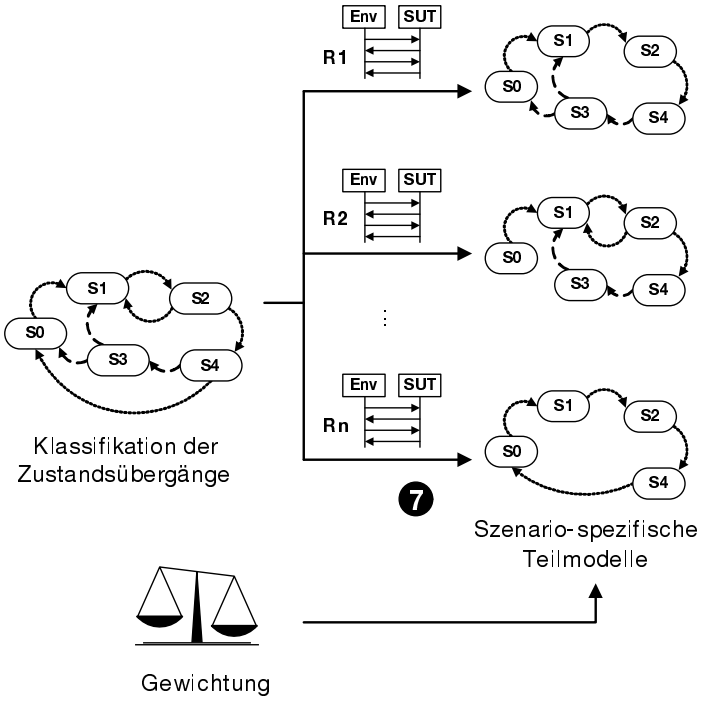

Abbildung 6.5.: Schritt 7 des vorgeschlagenen Entwicklungs- und Testprozesses: Ermittlung anforderungsspezifischer Teilmodelle (Ausschnitt aus Abbildung 4.1)

Für ein Szenario **r** *und die Gewichtung* **(l_0, l_1, … l_k)**
Teilmodell = ∅
Wähle die Menge **Trans(r)**, *welche im bestimmenden Teil von* **r** *feuern;*
füge **Trans(r)** *zum Teilmodel hinzu;*
zu jedem Zustand **s,** *welcher durch eine Transition in* **Trans(r)** *erreicht wird*
suche alle Pfade ***p*** *welche in* ***s*** *beginnen wobei*
p *maximal* $\mathbf{l_i}$ *Transitionen der Klasse* **i** *enthält;*
füge die Transitionen in **p** *dem Teilmodell hinzu;*
füge alle Zustände hinzu, welche zu einer Transition im Teilmodel verbunden sind;

Abbildung 6.6.: Algorithmus zur Bestimmung anforderungsspezifischer Teilmodelle für ein Szenario (Skizze).

6.5. Bestimmung anforderungsspezifischer Teilmodelle

Die Generierung von Testfällen erfolgt für jedes Szenario, und damit für jede Anforderung, aus einem eigenen anforderungsspezifischen Teilmodell. Es sind somit zunächst diese Teilmodelle zu bilden. Dies ist der siebte Schritt des vorgeschlagenen Entwicklungs- und Testprozesses, siehe dazu auch Abbildung 6.5.

Wesentliche Eingabe für diesen Schritt, welcher sich vollständig automatisieren lässt, ist das Verhaltensmodell des zu testenden Systems, in welchem die Anforderungsklassen an die Zustandsübergänge übertragen wurden. Dies wurde im vorhergehenden Abschnitt erläutert. Für jedes Szenario $r \in REQ$ wird ein Teilmodell gebildet. Ein Szenario ist also eine weitere Eingabe für die Bildung eines Teilmodells. Der Umfang des Teilmodells bestimmt sich durch die gewählte Gewichtung. Während das Szenario sicherstellt, dass der Pfad des Szenarios immer in dem jeweiligen Teilmodell enthalten ist, bestimmt die Gewichtung, wie stark die übrigen Modellteile bei der Bildung des Teilmodells berücksichtigt werden. Der Algorithmus zur Bildung des Teilmodells ist in Abbildung 6.6 skizziert. Die wesentlichen Schritte sind dabei:

1. Bestimmen der Transitionen, welche im bestimmenden Teil[1] des Szenarios schalten

2. Bestimmen der Zustände, welche durch diese Transitionen erreicht werden

3. Ermitteln der weiteren anforderungsspezifischen Transitionen. Diese gehen von den erreichten Zuständen aus und berücksichtigen die Klassifizierung der Transitionen und die gewählte Gewichtung der Klassen.

[1] vgl. Abschnitt 5.2.3

Die Transitionen im Szenario und die Transitionen der erweiterten Pfade bilden die Transitionen, welche das Teilmodell aufspannen. Die einzelnen Schritte zur Bildung der anforderungsspezifischen Teilmodelle werden im Folgenden formal definiert.

Abbildung 6.7 illustriert den Ablauf der Teilmodellbildung anhand des Modells der Fensterheberfunktion.

6.5.1. Formale Spezifikation des Teilmodells

Der im vorhergehenden Abschnitt skizzierte Algorithmus zur Bestimmung der Teilmodelle soll nun formalisiert werden. Ziel ist es, bei einem gegebenen Verhaltensmodell $M = (I, O, S, \delta, \lambda)$ und einer gegebenen Menge von Anforderungsszenarien REQ eine Funktion

$$\begin{aligned}\text{submod}_{M,REQ} \in (S \times (REQ \to CLAS) \times (CLAS \to \mathbb{N}) \times \mathbb{N}) \to REQ \\ \to \mathbb{P}(I) \times \mathbb{P}(O) \times \mathbb{P}(S) \times (S' \times I' \to S') \times (S' \times I' \to O')\end{aligned}$$

zu bestimmen, mit $S' \subseteq S$, $I' \subseteq I$ und $O' \subseteq O$.

Anmerkung: Zur kompakten Darstellung wird im Weiteren auf die Angabe der Subskripte M, REQ in der Funktionsbezeichnung $\text{submod}_{M,REQ}$ *verzichtet. Dies gilt ebenso für die in den folgenden Abschnitten eingeführten Funktionen.*

Bei einer Anwendung

$$\text{submod}[s_0, \text{classReq}, \text{weight}, n](r) = (I', O', S', \delta', \lambda')$$

der Funktion bezeichnet

s_0	den definierten Initialzustand für das Modell M,
classReq	die Klassifikation der Anforderungen,
weight	die Gewichtung der Anforderungsklassen,
n	die Gewichtung nicht klassifizierter Transitionen sowie
r	das Anforderungsszenario, für welches das Teilmodell bestimmt werden soll.

Die Mealy-Maschine für das Teilmodell M' ist durch das Tupel $(I', O', S', \delta', \lambda')$ angegeben.

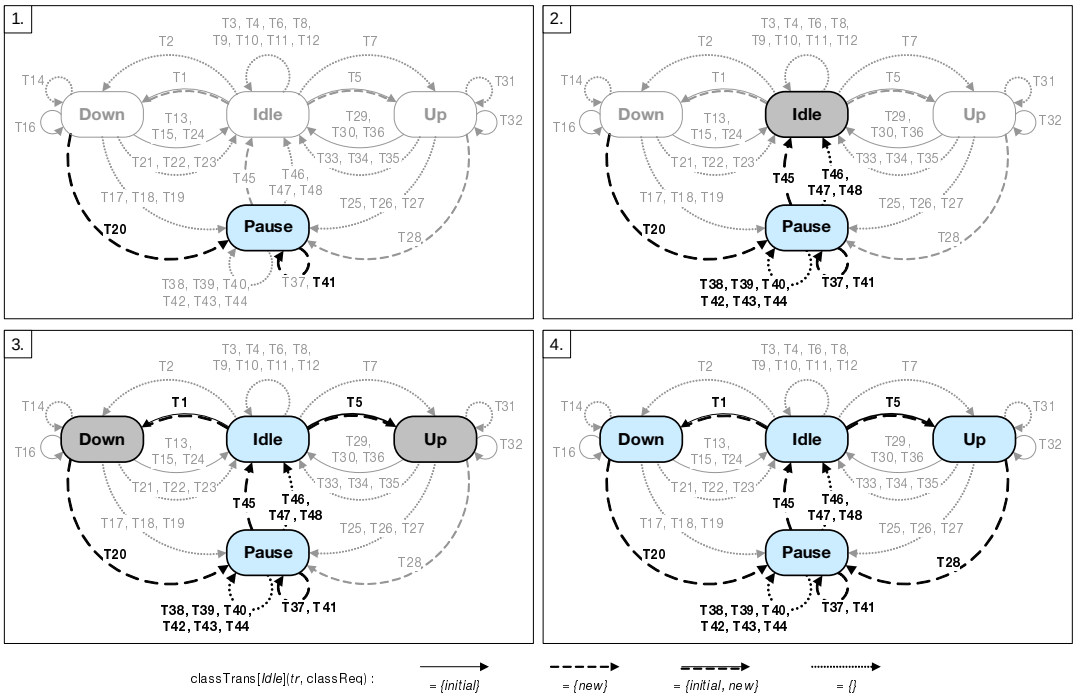

Abbildung 6.7.: Beispiel zum Ablauf der Teilmodellbildung (die hervorgehobenen Modellelemente werden dem Teilmodell hinzugefügt). Für Anforderungsszenario r_4' der Fensterheberfunktion und Gewichtung $(\text{weight}(\textit{initial}), \text{weight}(\textit{added}), n) = (1, 2, 1)$, n gibt dabei die Gewichtung für nicht-klassifizierte Transitionen an. **1.** Im bestimmenden Teil von r_4' schalten Transitionen t_{20} und t_{41}, diese sind initial Bestandteil des Teilmodells. Als einziger Zustand wird *Pause* erreicht. **2.** Da alle Gewichte > 0 sind, werden alle von *Pause* ausgehenden Transitionen gewählt. Damit wird auch der Zustand *Idle* erreicht. Abhängig von der gewählten Transition dorthin, werden die Gewichtungsparameter dekrementiert: Wird *Idle* über t_{45} der Klasse *added* erreicht, verbleibt eine Gewichtung von $(1, 1, 0)$ (der Gewichtungswert n wird in jedem Schritt dekrementiert). Über den Weg einer nicht klassifizierten Transition (t_{46}, t_{47} oder t_{48}) verbleibt dagegen eine Gewichtung von $(1, 2, 0)$. **3.** Mit dieser Gewichtung können ausgehend von *Idle* weiter die Transitionen t_1 und t_5 gewählt werden, es verbleiben jeweils die Gewichtungswerte $(0, 1, 0)$. **4.** Ausgehend von *Up* kann nun zuletzt Transition t_{28} zum Teilmodell hinzugenommen werden. Da nun alle Gewichtungswerte zu 0 dekrementiert sind, ist das Teilmodell vollständig gebildet.

Bestimmen der Transitionen des Szenarios

Um die Transitionen in einem als Mealy-Maschine angegebenen Modell M für ein Szenario zu bestimmen, wurde bereits in Abschnitt 6.3 die Funktion

$$\text{transDet} \in S \to REQ \to \mathbb{P}(S \times I)$$

eingeführt; für die Bildung eines Teilmodells wird diese Funktion erneut angewendet. Ein Aufruf $\text{transDet}[s_0](r)$ liefert für ein Szenario r ausgehend vom Initialzustand s_0 die Menge der Transitionen, welche im bestimmenden Teil $\mathsf{det}.r$ des Szenarios ausgeführt werden. Für die Bildung der Teilmodelle ist die Vorbedingung der Anforderung somit ohne Bedeutung. Die Teilmodelle werden nur auf Basis des bestimmenden Teils der Anforderung erstellt. (Siehe hierzu auch Abschnitte 5.1.4 und 5.2.3).

Bestimmen der erreichen Zustände

Die Zustände, welche im Modell $M = (I, O, S, \delta, \lambda)$ durch den bestimmenden Teil eines Szenarios r erreicht werden, ergeben sich unmittelbar aus der Menge der (bestimmenden) Transitionen des Szenarios $\text{transDet}[s_0](r)$ und der Zustandsübergangsfunktion δ. Die erreichten Zustände werden somit durch die Funktion

$$\text{states} \in S \to REQ \to \mathbb{P}(S)$$

für den Initialzustand $s_0 \in S$ und eine Anforderung $r \in REQ$ definiert als

$$\text{states}[s_0](r) \stackrel{\text{def}}{=} \{s' \in S | s' = \delta(s, in) \quad \forall (s, in) \in \text{transDet}[s_0](r)\}.$$

Beispiel 18 (Durch Anforderung erreichte Zustände)

Durch Anforderung r_4', welche bereits im Beispiel 16 auf Seite 122 behandelt wurde, wird nur der Zustand *Pause* erreicht:

$$\mathsf{states}[s_0](r_4'') = \{\mathit{Pause}\}$$

Für Szenario r_4'' gilt dagegen:

$$\mathsf{states}[s_0](r_4'') = \{\mathit{Pause}, \mathit{Idle}, \mathit{Up}\}.$$

Die im bestimmenden Teil des Szenarios erreichten Zustände sind Ausgangspunkt für die Wahl von weiteren Transitionen.

Ermitteln der weiteren anforderungsspezifischen Pfade

Die Selektion der weiteren Pfade für das anforderungsspezifische Teilmodell nimmt die durch eine Anforderung r erreichten Zustande $\text{states}[s_0](r)$ als Ausgangspunkt. Als weitere Pfade werden zunächst für jeden Zustand $s \in \text{states}[s_0](r)$ alle von s ausgehenden Transitionen betrachtet. Eine Transition (s, in) wird dabei nur gewählt, falls eine Klasse $c \in CLAS$ existiert, zu welcher die Transition zugeordnet ist und deren Gewichtung > 0 ist, die notwendige Bedingung zur Auswahl lautet somit:

$$\exists c \in CLAS : c \in \text{classTrans}[s_0]((s, in), \text{classReq}) \wedge \text{weight}(c) > 0.$$

Die Klassifikation der Transitionen wurde bereits in Abschnitt 6.3 beschrieben, wie dort in der Definition der Funktion classTrans angegeben ist, ist die Klassifikation der Transitionen aus den Klassen der Anforderungsszenarien abgeleitet, in welchen eine Transition schaltet.

Dieses Vorgehen wird iterativ fortgeführt: Ausgehend von den Zuständen, welche durch die neu hinzugenommen Transitionen erreicht werden, werden nach demselben Verfahren wiederum Transitionen hinzugenommen – dabei wird in jedem Schritt der Wert der entsprechenden Gewichtung um 1 dekrementiert.

Transitionen, denen keine Klasse zugeordnet ist, werden ebenso berücksichtigt. Hierfür wird ein eigener Gewichtungswert verwendet. Um den Algorithmus der Teilmodellbildung präzise zu beschreiben, wird zunächst die Funktion

$$\text{addPath} \in S \to S \times (REQ \to CLAS) \times (CLAS \to \mathbb{N}) \times \mathbb{N} \to \mathbb{P}(S \times I)$$

eingeführt, welche zu einem Zustand $s \in S$, einer Gewichtung $\text{weight} \in (CLAS \to \mathbb{N})$ und einem Wert $n \in \mathbb{N}$ für die Gewichtung nicht klassifizierter Transitionen eine Menge von Transitionen (s, in) ermittelt, welche in das Teilmodell ausgehend von Zustand s übernommen werden. Das erste Argument bezeichnet wiederum den für das Modell gewählten Initialzustand s_0.

Zur Definition der Funktion addPath sollen zunächst die Hilfsfunktionen

$$\text{selTransC} \in S \to S \times (REQ \to CLAS) \times (CLAS \to \mathbb{N}) \to \mathbb{P}(S \times I)$$

sowie

$$\text{selTransUC} \in S \to S \times (REQ \to CLAS) \times \mathbb{N} \to \mathbb{P}(S \times I)$$

eingeführt werden, welche zu einem Zustand s die Menge der zu selektierenden Transitionen angeben. Die Funktion $\text{selTransC}[s_0](s, \text{weight})$ gibt dabei für einen Zustand s und eine Gewichtung $\text{weight} \in CLAS \to \mathbb{N}$ die Menge der selektierten Transitionen an, sofern den Transitionen eine Anforderungsklasse zugeordnet ist. Die Funktion ist wie folgt definiert:

$$\text{selTransC}[s_0](s, \text{classReq}, \text{weight}) \stackrel{\text{def}}{=} \{(s, in) | \exists c \in \text{classTrans}[s_0]((s, in), \text{classReq}) : \text{weight}(c) > 0\}$$

Die Funktion $\text{selTransUC}[s_0](s, n)$ ermittelt dagegen für den Zustand s und die Gewichtung n die selektierten Transition aus der Menge der Transitionen, welchen keine Anforderungsklasse zugeordnet ist:

$$\text{selTransUC}[s_0](s, \text{classReq}, n) \stackrel{\text{def}}{=} \begin{cases} \{\} & \text{falls } n = 0 \\ \{(s, in) | \, \text{classTrans}[s_0]((s, in), \text{classReq}) = \{\}\} & \text{falls } n > 0 \end{cases}$$

Beide Funktionen wählen somit jeweils nur Transitionen aus, falls die entsprechende Gewichtung > 0 ist.

Beispiel 19 (Gewählte Transitionen)

Für die Fensterheberfunktion, bei welcher die Transitionen entsprechend Beispiel 17 (Seite 123 und Abbildung 6.3) klassifiziert wurden, werden für Zustand *Down* folgende Transitionen gewählt:

Aus einer Gewichtung

$$\text{weight}(\mathit{initial}) = 1$$
$$\text{weight}(\mathit{added}) = 0$$

ergibt sich

$$\text{selTransC}[\mathit{Idle}](\mathit{Down}, \text{classReq}, \text{weight}) = \{t_{13}, t_{15}, t_{16}, t_{24}\}$$

Analog gilt etwa:

$$\text{selTransUC}[\mathit{Idle}](\mathit{Down}, \text{classReq}, 2) = \{t_{14}, t_{17}, t_{18}, t_{19}, t_{21}, t_{22}, t_{23}\}$$

Damit kann nun die Funktion addPath rekursiv definiert werden:[2]

[2] Anders als in Abbildung 6.6 skizziert, wird hier eine Breitensuche verwendet.

$$\begin{aligned}\text{addPath}[s_0](s, \text{classReq}, \text{weight}, n) \\ &\stackrel{\text{def}}{=} T \cup T' \cup \bigcup_{(s,in)\in T} \text{addPath}[s_0](\delta(s, in), \text{weight}'[s_0, (s, in)], n-1) \\ &\qquad \cup \bigcup_{(s,in)\in T'} \text{addPath}[s_0](\delta(s, in), \text{weight}, n-1)\end{aligned}$$

wobei

$$T = \text{selTransC}[s_0](s, \text{classReq}, \text{weight})$$

und

$$T' = \text{selTransUC}[s_0](s, \text{classReq}, n)$$

.

Die Pfade, welche für einem Zustand s gewählt werden umfassen damit die Transitionen aus diesem Zustand, die einer Anforderungsklasse mit Gewichtung > 0 zugeordnet sind (Menge T), die Transitionen aus diesem Zustand, welche unklassifiziert sind, sofern der entsprechende Gewichtungswert $n > 0$ ist (Menge T') sowie für jeden Zustand s' in T beziehungsweise T' wieder die so selektierten Zustände, wobei die Gewichtung erniedrigt wird. Die Anpassung der Gewichtungsfunktion weight erfolgt mithilfe der Funktion

$$\text{weight}' \in S \times (S \times I) \to \mathit{CLAS} \to \mathbb{N}$$

welche für einen Initialzustand s_0 und eine Transition (s, in) durch

$$\text{weight}'[s_0, (s, in)](c) \stackrel{\text{def}}{=} \begin{cases} \text{weight}(c) - 1 & \text{falls } c \in \text{classTrans}[s_0](s, in) \\ \text{weight}(c) & \text{sonst} \end{cases}$$

definiert ist.

Die Gewichtung der nicht klassifizierten Transitionen wird dagegen für jede gewählte Transition verringert. Dies vermeidet eine Überbewertung der nicht klassifizierten Transitionen.

Beispiel 20 (Gewählte Pfade aus einem Zustand)

Im Modell der Fensterheberfunktion ergibt die Funktion addPath bei einer Gewichtung von weight(*initial*) = 1, weight(*added*) = 2 und einer Gewichtung der nicht klassifizierten Transitionen

$n = 1$ die folgende Menge an Transitionen, welche die gewählten Pfade aufspannen:

$$
\begin{aligned}
&\mathsf{addPath}[\mathit{Idle}](\mathit{Pause}, \mathsf{classReq}, (\mathit{initial} \mapsto 1, \mathit{added} \mapsto 2), 1) \\
&\quad = \{t_{37}, t_{41}, t_{45}\} \\
&\qquad \cup \{t_{38}, t_{39}, t_{40}, t_{42}, t_{43}, t_{44}, t_{46}, t_{47}, t_{48}\} \\
&\qquad \cup \mathsf{addPath}[\mathit{Idle}](\mathit{Pause}, \mathsf{classReq}, (\mathit{initial} \mapsto 1, \mathit{added} \mapsto 1), 0) \\
&\qquad \cup \mathsf{addPath}[\mathit{Idle}](\mathit{Idle}, \mathsf{classReq}, (\mathit{initial} \mapsto 1, \mathit{added} \mapsto 1), 0) \\
&\qquad \cup \mathsf{addPath}[\mathit{Idle}](\mathit{Pause}, \mathsf{classReq}, (\mathit{initial} \mapsto 1, \mathit{added} \mapsto 2), 0) \\
&\qquad \cup \mathsf{addPath}[\mathit{Idle}](\mathit{Idle}, \mathsf{classReq}, (\mathit{initial} \mapsto 1, \mathit{added} \mapsto 2), 0) \\
&\quad = \{t_{37}, t_{41}, t_{45}, t_{38}, t_{39}, t_{40}, t_{42}, t_{43}, t_{44}, t_{46}, t_{47}, t_{48}\} \\
&\qquad \cup \mathsf{addPath}[\mathit{Idle}](\mathit{Idle}, \mathsf{classReq}, (\mathit{initial} \mapsto 1, \mathit{added} \mapsto 2), 0) \\
&\quad = \{t_{37}, t_{41}, t_{45}, t_{38}, t_{39}, t_{40}, t_{42}, t_{43}, t_{44}, t_{46}, t_{47}, t_{48}\} \\
&\qquad \cup \{t_1, t_5\} \\
&\qquad \cup \mathsf{addPath}[\mathit{Idle}](\mathit{Down}, \mathsf{classReq}, (\mathit{initial} \mapsto 0, \mathit{added} \mapsto 1), 0) \\
&\qquad \cup \mathsf{addPath}[\mathit{Idle}](\mathit{Up}, \mathsf{classReq}, (\mathit{initial} \mapsto 0, \mathit{added} \mapsto 1), 0) \\
&\quad = \{t_{37}, t_{41}, t_{45}, t_{38}, t_{39}, t_{40}, t_{42}, t_{43}, t_{44}, t_{46}, t_{47}, t_{48}, t_1, t_5\} \\
&\qquad \cup \{t_{20}\} \cup \mathsf{addPath}[\mathit{Idle}](\mathit{Pause}, \mathsf{classReq}, (\mathit{initial} \mapsto 0, \mathit{added} \mapsto 0), 0) \\
&\qquad \cup \{t_{28}\} \cup \mathsf{addPath}[\mathit{Idle}](\mathit{Pause}, \mathsf{classReq}, (\mathit{initial} \mapsto 0, \mathit{added} \mapsto 0), 0) \\
&\quad = \{t_{37}, t_{41}, t_{45}, t_{38}, t_{39}, t_{40}, t_{42}, t_{43}, t_{44}, t_{46}, t_{47}, t_{48}, t_1, t_5\} \\
&\qquad \cup \{t_{20}\} \cup \{\} \\
&\qquad \cup \{t_{28}\} \cup \{\} \\
&\quad = \{t_1, t_5, t_{20}, t_{28}, t_{37}, t_{38}, t_{39}, t_{40}, t_{41}, t_{42}, t_{43}, t_{44}, t_{45}, t_{46}, t_{47}, t_{48}\}
\end{aligned}
$$

Damit lassen sich für ein Szenario r, ausgehend vom Initialzustand s_0, alle im Teilmodell enthalten Transitionen ermitteln. Hierzu werden die Transitionen des Szenarios, gegeben durch $\mathrm{transDet}[s_0](r)$ gewählt sowie ausgehend von den durch das Szenario erreichten Zustände, gegeben durch $\mathrm{states}[s_0](r)$, alle weiteren anforderungsspezifischen Pfade gemäß $\mathrm{addPath}[s_0](s, \mathrm{weight}, n)$ gewählt. Die entsprechende Funktion

$$\mathrm{submodTrans} \in S \to r \times (REQ \to CLAS) \times (CLAS \to \mathbb{N}) \times \mathbb{N} \to \mathbb{P}\, S \times I$$

ist dabei definiert als

$$
\begin{aligned}
&\mathrm{submodTrans}[s_0](r, \mathrm{classReq}, \mathrm{weight}, n) \\
&\quad \stackrel{\mathrm{def}}{=} \mathrm{transDet}[s_0](r) \cup \bigcup_{s \in \mathrm{states}[s_0](r)} \mathrm{addPath}[s_0](s, \mathrm{classReq}, \mathrm{weight}, n).
\end{aligned}
$$

Beispiel 21 (Transitionen in anforderungsspezifischen Teilmodellen)

In Anhang C sind für verschiedene Gewichtungen jeweils die gewählten Transitionen für die einzelnen anforderungsspezifischen Teilmodelle angegeben.

Die für das Teilmodell selektierten Transitionen hängen dabei auch von der Wahl des Zustandsraums ab: Eine höhere Anzahl von Zuständen führt in der Regel zu längeren kreisfreien Pfaden im Modell. Damit unterscheiden sich die Teilmodelle für verschiedene Anforderungsszenarien stärker von einander. Im anderen Extremfall, wenn das Modell nur einen Zustand umfasst, von welchem alle Transitionen ausgehen und in welchen alle Transitionen münden, ist eine sinnvolle Bildung von Teilmodellen nicht möglich und der vorgestellte Ansatz nicht anwendbar. In der Praxis sind derartige Modelle aber unüblich und sollen auch vermieden werden, da diese meist unverständlich sind, vor allem wenn damit eine große Anzahl von einzelnen Anforderungen spezifiziert wird. Der Zustandsraum des Modells ist vor allem auch bei der Wahl der Gewichtungen zu berücksichtigen, darauf wird in Abschnitt 6.7 noch eingegangen.

Sei

$$T' = \text{submodTrans}[s_0](r, \text{classReq}, \text{weight}, n)$$

die Menge der anforderungsspezifischen Transitionen. Dann ist das anforderungsspezifische Teilmodell $M' = (I', O', S', \delta', \lambda')$ gegeben durch:

$$\begin{aligned}
I' &\stackrel{\text{def}}{=} \{in \in I | \exists (s, in) \in T'\} \\
O' &\stackrel{\text{def}}{=} \{out \in O | \exists (s, in) \in T' : \lambda(s, in) = out\} \cup \{\lightning\} \\
S' &\stackrel{\text{def}}{=} \{s \in S | \exists (s, in) \in T'\} \cup \{s' \in S | \exists (s, in) \in T' : \delta(s, in) = s'\} \cup \{s_\lightning\} \\
\delta' \in S' \times I' \to S' \quad &\text{mit} \quad \delta'(s, in) \stackrel{\text{def}}{=} \begin{cases} \delta(s, in) & \text{falls } (s, in) \in T' \\ s_\lightning & \text{sonst} \end{cases} \\
\lambda' \in S' \times I' \to O' \quad &\text{mit} \quad \lambda'(s, in) \stackrel{\text{def}}{=} \begin{cases} \lambda(s, in) & \text{falls } (s, in) \in T' \\ \lightning & \text{sonst} \end{cases}
\end{aligned}$$

Die Zustandsmenge S' besteht damit aus allen Zuständen, welche als Ausgangszustand einer Transition $(s, in) \in T'$ vorkommen oder als ein Folgezustand einer dieser Transitionen im ursprünglichen Modell vorkommen. Zusätzlich wird ein Zustand $s_\lightning \notin S$ hinzugenommen. Dieser Zustand ist der Folgezustand für die Kombination von Eingaben $in' \in I'$ und Zustände $s' \in S'$, welche nicht als Transition in T' vorkommen, also für $(s', in') \notin T'$. Die Eingaben $I' \subseteq I$ bestimmen sich dabei aus den Eingaben, welche in den gewählten Transitionen T' vorkommen. Analog zu der Menge der Zustände ist das Ausgabealphabet O', die Menge der Ausgaben $\subseteq O$ welche von der Ausgabefunktion λ des ursprünglichen Modells für die Transitionen in T' produziert werden.

Auch hier wurde das Ausgabealphabet um ein Symbol $\lrcorner \notin O$ erweitert, welches eine undefinierte Ausgabe für den Fall $(s', in') \notin T'$ mit $s' \in S', in' \in I'$ angibt.

Für die Transitionen $(s, in) \in T'$, welche aus dem Gesamtmodell in das Teilmodell übernommen wurden, entsprechen die Zustandsübergangsfunktion δ' und die Ausgabefunktion λ' den Funktionen δ beziehungsweise λ des Modells M. Für Kombinationen aus Eingabe und Zustand des Teilmodells, welche nicht in der gewählten Menge T' der Transitionen vorkommen, führt δ' in den Zustand $s_\lrcorner$ und λ' produziert die Ausgabe $\lrcorner$. Das Ausgabesymbol $\lrcorner$ und der Zustand $s_\lrcorner$ wurden eingeführt, um auch für das Teilmodell wieder eine Mealy-Maschine mit totalen Zustandsübergangs- und Ausgabefunktionen zu erhalten. Der besondere Zustand $s_\lrcorner$ beziehungsweise die besondere Ausgabe $\lrcorner$ zeigen an, dass das entsprechende Verhalten nicht mehr durch das Teilmodell beschrieben wird.

Damit ist die Funktion

$$\text{submod}_{M,REQ}[s_0, \text{classReq}, \text{weight}, n](r) = (I', O', S', \delta', \lambda')$$

vollständig definiert und das Teilmodell M' durch das Tupel $(I', O', S', \delta', \lambda')$ angegeben.

Für jede Anforderung $r \in REQ$ gibt die Funktion $\text{submod}_{M,REQ}$ ein Teilmodell abhängig von der Klassifizierung der Anforderungen und der Gewichtung an. Diese Teilmodelle sind die Testmodelle für die einzelnen Anforderungen.

Der für dieses Modell zu wählende **Initialzustand** ist der Zustand, welcher nach Ausführung der Vorbedingung $\text{pre}.r = (ins_{\text{pre}}, outs_{\text{pre}})$ des Anforderungsszenarios r erreicht wurde. Für den Initialzustand s'_0 des Teilmodells zu Anforderung r gilt somit:

$$s'_0 = \Delta(s_0, ins_{\text{pre}})$$

Ausgehend von diesem Zustand akzeptiert das Teilmodell den bestimmenden Teil des Szenarios r durch die mit $\text{transDet}[s_0](r)$ gegebene Menge von Transitionen, welche aufgrund der Definition der Funktion submodTrans immer Bestandteil des Teilmodells ist.

Beispiel 22 (Anforderungsspezifisches Teilmodell)

Tabelle 6.4 gibt die Transitionen des Teilmodells der Fensterheberfunktion für die Anforderung r'_4 bei einer gewählten Gewichtung $\text{weight}(\textit{initial}) = 1, \text{weight}(\textit{added} = 2), n = 2$. Da gilt, dass $\text{states}[s_0](r'_4) = \{\textit{Pause}\}$ (vgl. Beispiel 18), entsprechen die in das Teilmodell übernommenen Transitionen denjenigen aus Beispiel 20 auf Seite 135. Als Initialzustand s'_0 wird für das Teilmodell entsprechend der Vorbedingung $\text{pre}.r'_4 = r_1$ der Zustand *Down* gewählt.

Tabelle 6.4.: Zustandsübergangsfunktion δ und Ausgabefunktion λ des anforderungsspezifischen Teilmodells der Fensterheberfunktion für Anforderung $r4'$.

	s	in	$\delta(s, in)$	$\lambda(s, in)$
t_1	*Idle*	(*open*, *middle*)	*Down*	(*down*, $\perp$)
t_2	*Idle*	(*open*, *top*)	$s_\lrcorner$	$\dashv$
t_3	*Idle*	(*open*, *bottom*)	$s_\lrcorner$	$\dashv$
t_4	*Idle*	(*open*, *moving*)	$s_\lrcorner$	$\dashv$
t_5	*Idle*	(*close*, *middle*)	*Up*	(*up*, $\perp$)
t_6	*Idle*	(*close*, *top*)	$s_\lrcorner$	$\dashv$
t_7	*Idle*	(*close*, *bottom*)	$s_\lrcorner$	$\dashv$
t_8	*Idle*	(*close*, *moving*)	$s_\lrcorner$	$\dashv$
t_9	*Idle*	($\perp$, *middle*)	$s_\lrcorner$	$\dashv$
t_{10}	*Idle*	($\perp$, *top*)	$s_\lrcorner$	$\dashv$
t_{11}	*Idle*	($\perp$, *bottom*)	$s_\lrcorner$	$\dashv$
t_{12}	*Idle*	($\perp$, *moving*)	$s_\lrcorner$	$\dashv$
t_{13}	*Down*	(*open*, *middle*)	$s_\lrcorner$	$\dashv$
t_{14}	*Down*	(*open*, *top*)	$s_\lrcorner$	$\dashv$
t_{15}	*Down*	(*open*, *bottom*)	$s_\lrcorner$	$\dashv$
t_{16}	*Down*	(*open*, *moving*)	$s_\lrcorner$	$\dashv$
t_{17}	*Down*	(*close*, *middle*)	$s_\lrcorner$	$\dashv$
t_{18}	*Down*	(*close*, *top*)	$s_\lrcorner$	$\dashv$
t_{19}	*Down*	(*close*, *bottom*)	$s_\lrcorner$	$\dashv$
t_{20}	*Down*	(*close*, *moving*)	*Pause*	(*stop*, $\perp$)
t_{21}	*Down*	($\perp$, *middle*)	$s_\lrcorner$	$\dashv$
t_{22}	*Down*	($\perp$, *top*)	$s_\lrcorner$	$\dashv$
t_{23}	*Down*	($\perp$, *bottom*)	$s_\lrcorner$	$\dashv$
t_{24}	*Down*	($\perp$, *moving*)	$s_\lrcorner$	$\dashv$
t_{25}	*Up*	(*open*, *middle*)	$s_\lrcorner$	$\dashv$
t_{26}	*Up*	(*open*, *top*)	$s_\lrcorner$	$\dashv$
t_{27}	*Up*	(*open*, *bottom*)	$s_\lrcorner$	$\dashv$
t_{28}	*Up*	(*open*, *moving*)	*Pause*	(*stop*, $\perp$)
t_{29}	*Up*	(*close*, *middle*)	$s_\lrcorner$	$\dashv$
t_{30}	*Up*	(*close*, *top*)	$s_\lrcorner$	$\dashv$
t_{31}	*Up*	(*close*, *bottom*)	$s_\lrcorner$	$\dashv$
t_{32}	*Up*	(*close*, *moving*)	$s_\lrcorner$	$\dashv$
t_{33}	*Up*	($\perp$, *middle*)	$s_\lrcorner$	$\dashv$
t_{34}	*Up*	($\perp$, *top*)	$s_\lrcorner$	$\dashv$
t_{35}	*Up*	($\perp$, *bottom*)	$s_\lrcorner$	$\dashv$
t_{36}	*Up*	($\perp$, *moving*)	$s_\lrcorner$	$\dashv$
t_{37}	*Pause*	(*open*, *middle*)	*Pause*	(*stop*, $\perp$)
t_{38}	*Pause*	(*open*, *top*)	*Pause*	(*stop*, $\perp$)
t_{39}	*Pause*	(*open*, *bottom*)	*Pause*	(*stop*, $\perp$)
t_{40}	*Pause*	(*open*, *moving*)	*Pause*	(*stop*, *err*)
t_{41}	*Pause*	(*close*, *middle*)	*Pause*	(*stop*, $\perp$)
t_{42}	*Pause*	(*close*, *top*)	*Pause*	(*stop*, $\perp$)
t_{43}	*Pause*	(*close*, *bottom*)	*Pause*	(*stop*, $\perp$)

t_{44}	*Pause*	(*close*,	*moving*)	*Pause*	(*stop*,	*err*)
t_{45}	*Pause*	(⊥,	*middle*)	*Idle*	(*stop*,	⊥)
t_{46}	*Pause*	(⊥,	*top*)	*Idle*	(*stop*,	⊥)
t_{47}	*Pause*	(⊥,	*bottom*)	*Idle*	(*stop*,	⊥)
t_{48}	*Pause*	(⊥,	*moving*)	*Idle*	(*stop*,	*err*)
t_{49}	$s_{\lrcorner}$	(*open*,	*middle*)	$s_{\lrcorner}$	⊣	
t_{50}	$s_{\lrcorner}$	(*open*,	*top*)	$s_{\lrcorner}$	⊣	
t_{51}	$s_{\lrcorner}$	(*open*,	*bottom*)	$s_{\lrcorner}$	⊣	
t_{52}	$s_{\lrcorner}$	(*open*,	*moving*)	$s_{\lrcorner}$	⊣	
t_{53}	$s_{\lrcorner}$	(*close*,	*middle*)	$s_{\lrcorner}$	⊣	
t_{54}	$s_{\lrcorner}$	(*close*,	*top*)	$s_{\lrcorner}$	⊣	
t_{55}	$s_{\lrcorner}$	(*close*,	*bottom*)	$s_{\lrcorner}$	⊣	
t_{56}	$s_{\lrcorner}$	(*close*,	*moving*)	$s_{\lrcorner}$	⊣	
t_{57}	$s_{\lrcorner}$	(⊥,	*middle*)	$s_{\lrcorner}$	⊣	
t_{58}	$s_{\lrcorner}$	(⊥,	*top*)	$s_{\lrcorner}$	⊣	
t_{59}	$s_{\lrcorner}$	(⊥,	*bottom*)	$s_{\lrcorner}$	⊣	
t_{60}	$s_{\lrcorner}$	(⊥,	*moving*)	$s_{\lrcorner}$	⊣	

Formal umfasst die Mealy-Maschine für dieses Teilmodell eine höhere Anzahl von Transitionen als das Ausgangsmodell. Ursache hierfür ist, dass auch die gewählten Transitionen das vollständige Eingabealphabet verwenden, zusätzlich aber der Pseudo-Zustand $s_{\lrcorner}$ eingeführt wurde. Da aber alle Transitionen aus diesem Zustand sowie die nicht gewählten Transitionen, welche nun die Pseudo-Ausgabe ⊣ produzieren und in diesen Pseudo-Zustand führen, in der Testfallermittlung unberücksichtigt bleiben, ist die Bezeichnung „Teilmodell“ dennoch gerechtfertigt.

Abbildung 6.8 illustriert dieses Teilmodell sowie einige weitere Beispiele als Zustandsübergangsgraph. Aus Anhang C sind weiter die Teilmodelle zu allen Anforderungen des Fensterhebers für verschiedene Gewichtungen ersichtlich.

6.5.2. Anwendung auf erweiterte Zustandsmaschinen

Bevor näher auf die Generierung der Testfälle aus den Teilmodellen eingegangen wird, soll diskutiert werden, wie sich der zuvor eingeführte Algorithmus zur Teilmodellbildung auf erweiterte Zustandsmaschinen übertragen lässt.

Überführung von erweiterten Zustandsmaschinen in Mealy-Maschinen

Zum einen kann eine erweiterte Zustandsmaschine, sofern sie nur endliche Datentypen verwendet, in eine semantisch äquivalente Mealy-Maschine transformiert werden. Der beschriebene Algorithmus zur Bildung von Teilmodellen lässt sich dann auf der Mealy-Maschinen-Repräsentation des Modells durchführen. Für Modelle, welche in der Praxis verwendet werden, wird eine derartige Transformation häufig zu rechenintensiv sein und insbesondere zu einem zu großen Zustandsraum führen, welcher nicht mehr

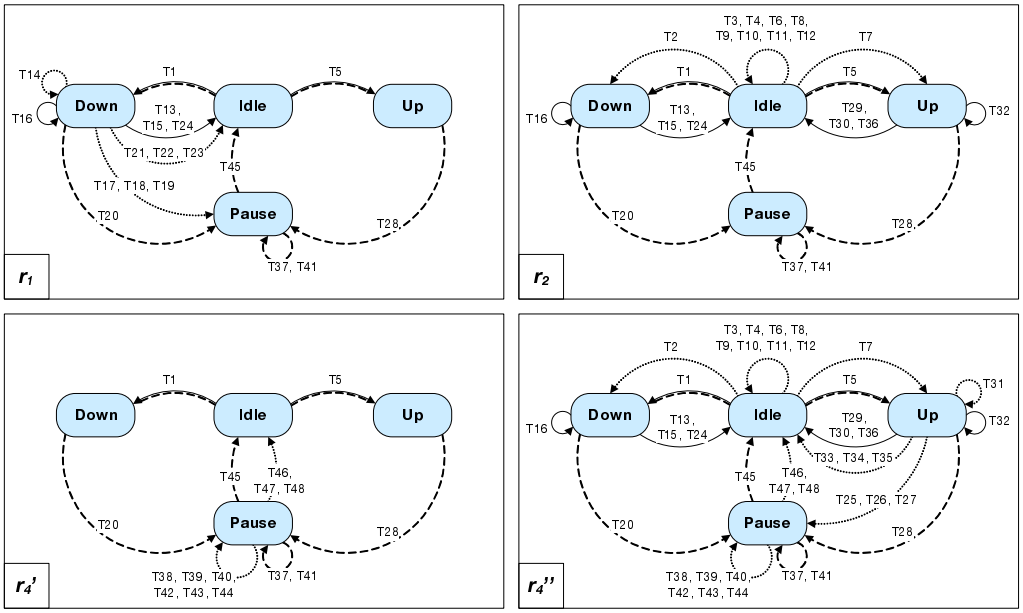

Abbildung 6.8.: Zustandsübergangsgraph der anforderungsspezifischen Teilmodelle der Fensterheberfunktion, für die Anforderungen r_1, r_2, r_4', r_4'' bei einer Gewichtung $\text{weight}(\textit{initial}) = 1, \text{weight}(\textit{added} = 2), n = 2$. Transitionen in den und aus dem Pseudo-Zustand $s_\lrcorner$ sind in den Abbildungen nicht dargestellt.

effizient dargestellt werden kann. Diesem Problem kann begegnet werden, indem der Wertebereich der verwendeten Datentypen beschränkt werden.

Strukturelle Abstraktion der erweiterten Zustandsmaschine

Eine weitere Möglichkeit ist, den oben beschriebenen Algorithmus allein auf die Struktur der erweiterten Zustandsmaschinen anzuwenden. Für die Teilmodellbildung werden in diesem Fall nur noch explizit angegebene Kontrollzustände unterschieden, eine weitere Unterscheidung des Datenzustands findet nicht statt. Als Transitionen werden ebenso nur die explizit im Modell angebenden Kanten, welche Übergänge zwischen Kontrollzustände darstellen, betrachtet. Ist ein Kante gewählt, gelten damit alle mit ihr beschriebenen Transitionen (im Sinne der äquivalenten Mealy-Maschine) als gewählt.

6.5.3. Operationalisierung

Für die Durchführung der Fallstudie wurde der entworfene Algorithmus zur Bildung von szenarienspezifischen Teilmodellen als Constraint-Logik-Programm implementiert.

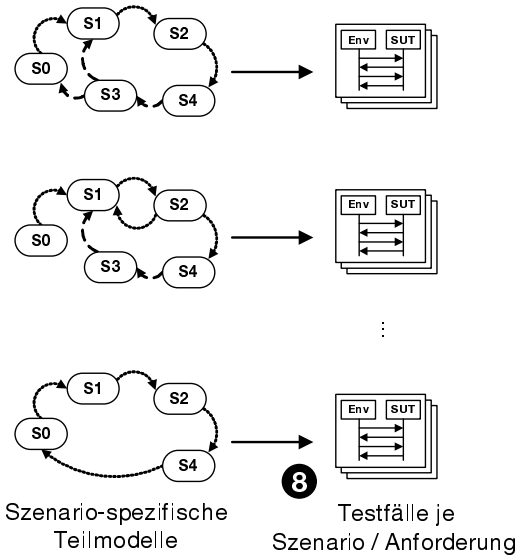

Abbildung 6.9.: Schritt 8 des vorgeschlagenen Entwicklungs- und Testprozesses: Testfallgenerierung aus anforderungsspezifischen Teilmodellen (Ausschnitt aus Abbildung 4.1)

Das entsprechende CLP-Programm ist in Anhang A angegeben. Als Constraint-Löser wurde *ECLiPSe CLP*[3] verwendet. In diesem Programm sind ebenfalls die Mealy-Maschine des Gesamtmodells sowie die Szenarien als CLP-Prädikate codiert.

Das Programm liefert die Menge der Transitionen, welche in den Teilmodellen erhalten bleiben – das jeweilige Teilmodell ist dann unmittelbar gegeben, wenn die übrigen Transitionen entfernt werden.

6.6. Testfallgenerierung

Zur anforderungsorientierten Generierung von Testfällen aus der totalen Verhaltensspezifikation des zu testenden Systems, welche durch das Modell M angegeben ist, wird das Modell, wie zuvor beschrieben, in Teilmodelle zerlegt. Jedes Teilmodell ist dabei spezifisch für jeweils ein einzelnes Anforderungsszenario. Diese Teilmodelle dienen als Testmodelle, welche zur eigentlichen modellbasierten Testfallgenerierung genutzt werden.

Die weitere Bestimmung der Testfälle aus den Teilmodellen ist dabei nicht an eine spezifische Technik zur Testfallgenerierung gebunden, sofern die Technik auf Mealy-Maschinen (beziehungsweise erweiterte Zustandsmaschinen, EFSMs) anwendbar ist. In Abschnitt 2.3 wurden die bekannten Verfahren hierzu vorgestellt.

[3] www.eclipse-clp.org

In den folgenden beiden Abschnitten wird zunächst auf die Testfallgenerierung in einem einzelnen anforderungsspezifischen Teilmodell eingegangen. Anschließend wird die Generierung von Testfällen für das Gesamtmodell betrachtet.

6.6.1. Testfallauswahl in einem Teilmodell

Für die Generierung von Testfällen hinsichtlich einer Anforderung $r \in REQ$ wird das anforderungsspezifische Teilmodell

$$M_r = \text{submod}_{M,REQ}[s_0, \text{classReq}, \text{weight}, n](r) = (I', O', S', \delta', \lambda')$$

als Testmodell verwendet. Die Testfälle werden mithilfe eines Testfallgenerators, welcher einer Funktion

$$\text{testcases}_{struct} \in \mathcal{M}_{I',O'} \times S' \times TCS_{struct} \to \mathbb{P}(I'^* \times O'^*)$$

entspricht, bestimmt. $\mathcal{M}_{I',O'}$ beschreibt dabei die Menge der als Mealy-Maschinen (vgl. Abschnitt 5.3.1) angegebenen Modelle mit Eingabealphabet I' und Ausgabealphabet O', TCS_{struct} die Menge struktureller Testfallspezifikationen. Zur Testfallgenerierung in dem Teilmodell kommen somit strukturelle Testfallspezifikationen zum Einsatz.

Dabei gilt

$$\text{testcases}_{struct}(M_r, s'_0, tcs) \subseteq \{(ins, outs) \in I'^* \times O'^* | outs = \Lambda'(s'_0, ins)\}.$$

Als Initialzustand s'_0 wird, wie bereits zuvor erläutert, der Zustand gewählt, welche sich nach Ausführung der Vorbedingung $\mathsf{pre}.r = (ins_{\text{pre}}, outs_{\text{pre}})$ ergibt. Ist s_0 der Initialzustand des Gesamtmodells, so gilt $s'_0 = \Delta(s_0, ins_{\text{pre}})$. Ein aus dem Teilmodell generierter anforderungsspezifischer Testfall $t_r = (ins_{t_r}, outs_{t_r})$ wird um die Vorbedingung erweitert, um einen Testfall t zu erhalten, welcher vom Initialzustand s_0 des Gesamtmodells ausgeht. Es gilt somit $t = (ins_{\text{pre}} \frown ins_{t_r}, outs_{\text{pre}} \frown outs_{t_r})$. Weiter ist zu beachten, dass der Testfall in den Ausgaben das Symbol $\lrcorner$ enthalten kann, für Ausgaben, welche aus Kombinationen von Eingabeaktion und Zustand folgen, welche im Teilmodell nicht berücksichtigt sein sollten. Der Testfall umfasst damit nur die Ein-/Ausgabeaktionen bis vor dem ersten Auftreten der Ausgaben $\lrcorner$. Grund dafür ist, dass ein Testfall aus dem Teilmodell ab dieser Stelle nur noch $\lrcorner$-Ausgaben enthält, da dann jede Transition konstruktionsbedingt wiederum in den Pseudo-Zustand $s_\lrcorner$ führt.

Zur Elimination der Schritte mit Pseudo-Ausgaben $\lrcorner$ wird die Funktion

$$\text{recast} \in REQ \times (I^* \times O^*) \to I^* \times O^*,$$

definiert durch

$$\text{recast}(r, t) \stackrel{\text{def}}{=} \left(\left(\mathsf{in}.\,(\mathsf{pre}.r) \frown (\mathsf{in}.t)|_{\text{find}(\mathsf{out}.t, \lrcorner)}\right), \left(\mathsf{out}.\,(\mathsf{pre}.r) \frown (\mathsf{out}.t)|_{\text{find}(\mathsf{out}.t, \lrcorner)}\right)\right),$$

eingeführt, wobei

$$\text{find} \in A^* \times A \to \mathbb{N}$$

die Position des ersten Auftretens einer Aktion $a \in A$ in einer Folge $s \in A^*$ angibt und wie folgt definiert ist:

$$\text{find}(s, a) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{falls } \mathsf{ft}(s) = a \vee s = \langle\rangle \\ \text{find}(\mathsf{rt}(s), a) + 1 & \text{sonst} \end{cases}$$

Mit der Funktion

$$\text{testcases}_r \in \mathcal{M}_{I',O'} \times S' \times TCS_{struct} \to \mathbb{P}(I'^* \times O'^*),$$

definiert durch

$$\begin{aligned} &\text{testcases}_r(M_r, s'_0, tcs) \\ &\qquad = \{t \in I^* \times O^* | t' = \text{testcases}_{struct}\,(M_r, s'_0, tcs) \wedge t = \text{recast}\,(r, t')\} \end{aligned}$$

wird die anforderungsorientierte Testfallgenerierung beschrieben, welche die Testfälle um die Vorbedingung der Anforderung erweitert und die Pseudo-Ausgaben $\dashv$ in den Testfällen eliminiert.

Beispiel 23 (Testfälle aus Teilmodellen)

In Anhang D sind jeweils die Mengen von Testfällen angegeben, welche aus den anforderungsspezifischen Teilmodellen der Fensterheberfunktion durch die Testfallgenerierung

$$\text{testcases}_r(M_r, s'_0, \mathit{transitionCoverage}) = TC_r$$

ermittelt wurden.[4] Das anforderungsspezifische Teilmodell M_r zur Anforderung r wurde dabei analog zu Beispiel 22 (Seite 138) gebildet. (Gewichtung: weight *initial* $= 1$, weight *added* $= 2$, $n = 1$.)

transitionCoverage $\in TCS_{struct}$ bezeichnet dabei die Überdeckung aller Transitionen, welche als strukturelle Testfallspezifikation verwendet wurde.

Zur Generierung von Testfällen in den einzelnen Teilmodellen wird somit eine strukturelle Testfallspezifikation verwendet. Wie am Anfang dieses Kapitels in Abschnitt 6.1 dargestellt wurde, stellt der anforderungsorientierte modellbasierte Test eine Kombination aus struktureller und funktionaler Testfallspezifikation dar. Die strukturellen Anteile, beziehen sich auf eine Überdeckung der Struktur der Teilmodelle. Es ist das Ziel der Methode, dass sich Testfälle an den Einzelanforderungen orientieren, aber nicht ausschließlich auf das formulierte Anforderungsszenario beschränken.

[4] Da die anforderungsspezifischen Teilmodelle für Anforderungen r_2, r_3 und r_5 sowie für Anforderungen r_7, r_8 und r_{10} äquivalent sind, wurden die Testfallmengen hier jewils nur einmal angeführt.

Durch die Verwendung einer strukturellen Überdeckung im Teilmodell ist gewährleistet, dass eine Anforderung nicht isoliert getestet wird, sondern im Kontext des Teilmodells berücksichtigt wird. Das führt dazu, dass die ermittelte Testsuite auch Testfälle umfasst, welche nicht durch die explizit formulierten Einzelanforderungen erfasst sind, sondern auch Systemabläufe als Testfälle gewählt werden, welche erst durch die Vervollständigung im Verhaltensmodell beschrieben werden. Der Test beschränkt sich somit nicht nur auf Systemabläufe, welche unmittelbar aus der Anforderung folgen. Dies ist ein entscheidender Vorteil gegenüber einer Ableitung von Testfällen aus den Einzelanforderungen.

Das Szenario der Anforderung r, für welche das Teilmodell M_r gebildet wurde, ist zentraler Bestandteil des Teilmodells. Die weiteren Modellteile des Teilmodells bestimmen sich aus der Gewichtung der Anforderungsklassen. Eine höhere Gewichtung führt dazu, dass vermehrt Transitionen, die dieser Anforderungsklasse zugeordnet sind, in das Teilmodell übernommen wurden. Somit beeinflussen Anforderungen, deren Anforderungsklasse höher gewichtet ist, das Testmodell und damit die Testfallgenerierung für *jede* Anforderung in einem stärkeren Maße, als dies bei niedriger gewichteten Anforderungsklassen der Fall ist. Auf diese Weise spiegelt sich die Priorisierung von Anforderungen für den Test in den einzelnen Testmodellen und den daraus generierten Testfällen wieder.

6.6.2. Generierung von Testfällen für alle Anforderungen

Die Testfälle für den Test des gesamten zu testenden System ergeben sich aus der Vereinigung aller Testfallmengen, welche für die einzelnen Anforderungen generiert wurden. Die Bildung von Teilmodellen und die Generierung von Testfällen aus dem Teilmodell hat also für jede Anforderung zu erfolgen.

Die Testfallspezifikation für die Generierung von Testfällen aus dem Gesamtmodell $M \in \mathcal{M}_{I,O}$ mit syntaktischer Schnittstelle $(I \triangleright O)$ setzt sich somit zusammen aus

- der Menge der Anforderungsszenarien $REQ \subsetneq I^* \times O^*$,
- der Klassifikation der Anforderungen $\text{classReq} \in REQ \to CLAS$,
- der Gewichtung der Anforderungsklassen $\text{weight} \in CLAS \to \mathbb{N}$,
- der Gewichtung der unklassifizierten Transitionen $n \in \mathbb{N}$ sowie,
- für die Generierung aus den Teilmodellen die gewählte strukturelle Testfallspezifikation $tcs_{struct} \in TCS_{struct}$.

Die Testfallgenerierung wird somit durch die Funktion

$$\text{testcases}_{REQ} \in \mathcal{M}_{I,O} \times S \times TCS_{REQ} \to \mathbb{P}(I^* \times O^*)$$

mit

$$TCS_{REQ} = \mathbb{P}(I^* \times O^*) \times (REQ \to CLAS) \times (CLAS \to \mathbb{N}) \times \mathbb{N} \times TCS_{struct}$$

angegeben. Sie ist für ein Modell $M = (I, O, S, \delta, \lambda)$ und den Initialzustand s_0 definiert als

$$\text{testcases}_{REQ}(M, s_0, (REQ, \text{classReq}, \text{weight}, n, tcs_{struct})) \stackrel{\text{def}}{=} \bigcup_{r \in REQ} \text{testcases}_r(M_r, s_{0r}, tcs_{struct})$$

mit

$$M_r = \text{submod}_{M,REQ}[s_0, \text{classReq}, \text{weight}, n](r)$$
$$s_{0r} = \Delta(\mathsf{in}.(\mathsf{pre}.r)).$$

Die auf diese Weise ermittelte Menge an Testfällen weist allerdings Redundanzen auf: Testfälle welche Präfix eines anderen Testfalls sind, können eliminiert werden. Da in der vorliegenden Arbeit nur deterministische Systeme von Interesse sind, reicht dazu die Betrachtung der Testeingaben. Falls also für Testfälle t_1, t_2 gilt, dass $\mathsf{in}.t_2 \sqsubseteq \mathsf{in}.t_1$, so kann t_2 aus der Testfallmenge entfernt werden, da durch t_2 keine zusätzliche Überprüfung stattfindet, welche nicht auch durch t_1 erfolgen würde. Die Funktion testcases_{REQ} kann also alternativ auch durch

$$\text{testcases}_{REQ}(M, s_0, (REQ, \text{classReq}, \text{weight}, n, tcs_{struct})) \stackrel{\text{def}}{=} \text{compact}\left(\bigcup_{r \in REQ} \text{testcases}_{struct}(M_r, s_{0r}, tcs_{struct})\right)$$

definiert werden, wobei Präfixe entfernt werden. Für die Hilfsfunktion

$$\text{compact} \in \mathbb{P}(I^* \times O^*) \to \mathbb{P}(I^* \times O^*)$$

gilt dabei für eine Testfallmenge T

$$\text{compact}(T) \stackrel{\text{def}}{=} \{t \in T | \nexists t' \in t, t' \neq t : \mathsf{in}.t \sqsubseteq \mathsf{in}.t'\}.$$

Im Regelfall ist die zweite, kompaktere Definition der Testfallmenge vorzuziehen, da damit unnötige Redundanzen bei der Testausführung vermieden werden. Zur Dokumentation der Überdeckung von Anforderungen ist gegebenenfalls die erste Variante geeigneter, da hier jeder Testfall, welcher aus den einzelnen anforderungsspezifischen Teilmodellen generiert wurde, explizit enthalten ist. Dies erleichtert das Tracing von Testfällen zu Anforderungen.

Abbildung 6.10.: Schritt 9 des vorgeschlagenen Entwicklungs- und Testprozesses: Review der generierten Testfälle und Anpassung der Gewichtung (Ausschnitt aus Abbildung 4.1)

Beispiel 24 (Menge von Testfällen aus allen Teilmodellen)

In Anhang E ist die zusammengefasste Menge der Testfälle aus den anforderungsspezifischen Teilmodellen der Fensterheberfunktion angegeben.

Die konkrete Ausprägung der Testfallmenge hängt unter anderem von der Klassifikation der Anforderungen und deren Gewichtung ab. Wie schon in Abschnitt 6.4.2 diskutiert wurde, ist es nicht möglich, eine allgemeingültige Empfehlung zur Wahl der Gewichtung anzugeben. Um diesem Problem zu begegnen, folgt der Testfallgenerierung ein Review der erzeugten Testfälle.

6.7. Review der Testfälle

Nach der Testfallgenerierung erfolgt ein Review der Testfälle durch den Testingenieur. Ziel des Reviews ist es, die Gewichtungen und gegebenenfalls die Klassifizierung der Anforderungen derart anzupassen, dass die ermittelte Testsuite der Intention des Testingenieurs einer geeigneten Testsuite entspricht. Der Testingenieur hat somit die Testfälle zu validieren und auf ihre Plausibilität zu analysieren. Ein Testfall ist plausibel, falls er als geeignet erscheint, bedeutende Fehler im Testobjekt aufdecken zu können. Wie schon in Abschnitt 6.4 dargestellt, ist dabei abzuwägen, ob die Testfallgenerierung stärker auf eine

- *funktionale Testfallauswahl* oder
- *strukturelle Testfallauswahl*

zielen soll. Zudem ist die Relation der Anforderungsklassen zueinander in den Testfällen zu prüfen. Im Review sind die Testfälle daher vor allem hinsichtlich dieser beiden Fragestellungen zu untersuchen:

- Entspricht das Verhältnis zwischen dem Einfluss der funktionalen und strukturellen Testfallauswahl den Erwartungen?
- Entspricht der Einfluss, welche die einzelnen Klassen von Anforderungen auf die Menge der Testfälle hatten, den Erwartungen?

Wird bei dieser Prüfung eine unbefriedigende Testfallmenge festgestellt, ist die Funktion *weight* für die Gewichtungen oder gegebenenfalls auch die Klassifikation *classReq* der Anforderungen anzupassen. Da sowohl die Ermittlung der Teilmodelle als auch die Generierung der Testfälle vollständig automatisierbar ist, sind die Auswirkungen einer Anpassung ohne großen Aufwand überprüfbar. Eventuell nötige weitere Anpassungen können iterativ durchgeführt werden.

Im Extremfall, falls für alle Klassen $c \in CLAS$ eine Gewichtung $\text{weight}(c) = 0$ gilt und auch die nicht klassifizierten Transitionen mit 0 gewichtet sind, erfolgt die Testfallgenerierung in den einzelnen Teilmodellen aus einem Modell, welches nur das ursprüngliche Szenario akzeptiert. Grund hierfür ist, dass in das Teilmodell nur die Transitionen übernommen werden, welche auch bei der Ausführung des Szenarios $r \in REQ$ schalten. Die Testfallgenerierung liefert dann nur das ursprüngliche Szenario r (gegebenenfalls iteriert) als Testfall. Im anderen Extremfall, in welchen alle Gewichte unendlich hoch gewählt werden, würde jedes Teilmodell wiederum das ursprüngliche Modell beschreiben, und für alle Anforderungen würde dieselbe Menge an Testfällen ermittelt werden (sofern alle Zustände im Modell erreichbar sind).

Allgemein ist die Gewichtung so zu wählen, dass die Teilmodelle nicht zu isoliert nur die ursprünglichen Szenarien beschreiben, aber auch eine ausreichende Verschiedenartigkeit zueinander besteht. Die Teilmodelle sollen somit möglichst kleine Schnittmengen von Zuständen und/oder Transition aufweisen. Für die Wahl der Gewichtung sind dabei folgende Faktoren zu berücksichtigen:

Modellstruktur Hinsichtlich der Struktur des Gesamtmodells ist zu berücksichtigen, wie nahe das Modell einem vollständigen Graphen kommt. Im Falle eines nahezu vollständigen Graphen (in welchen von jedem Zustand aus nahezu jeder andere Zustand direkt durch eine Transition erreichbar ist) sind die Gewichtungen vergleichsweise niedrig zu wählen.

Anzahl der Anforderungsklassen Je mehr Anforderungsklassen unterschieden werden, desto geringer sind die einzelnen Gewichte zu wählen.

Disjunktheit der Klassen Sind die Mengen von Anforderungen oder die Mengen der Transitionen einer Anforderungsklasse weitgehend disjunkt zu den übrigen

Anforderungen beziehungsweise Transitionen, können die Gewichte höher gewählt werden.

Granularität und Vollständigkeit der Anforderungen Sind die Anforderungen sehr detailliert beschreiben und geben die Szenarien somit das Verhalten des Systems nahezu vollständig an, sind die Gewichte niedriger zu wählen. Würden alle zulässigen Systemabläufe als Szenarien angegeben, würde eine Gewichtung $weight(c) = 0$ für alle Anforderungsklassen ausreichen, um eine vollständige Testsuite zu erhalten, da sie alle Szenarien enthalten würde, welche als Systemablauf auftreten können.[5]

Nach der Anpassung der Gewichtung erfolgt eine erneute Bildung der Teilmodelle und eine erneute Generierung der Testfälle. Möglicherweise führt das Review auch zu einer Anpassung der Klassifikation der Anforderungen (insbesondere können zusätzliche Klassen eingefügt werden), um die Gewichte differenzierter angeben zu können.

Die Gewichtungen und die Klassifikation der Anforderungen sind in diesem Verfahren die „Stellschrauben“, mit welchen der Testingenieur seine Intention und Erfahrung in die Testfallgenerierung auf einfache Weise einfließen lasssen kann. Die ansonsten aufwändige Definition einzelner Testfälle wird dagegen durch den Automatismus der Teilmodellbildung und Testfallgenerierung übernommen.

Das Verfahren zur anforderungsorientierten modellbasierten Testfallgenerierung wurde nun vollständig beschrieben. Im folgenden Kapitel erfolgt die Analyse der so ermittelten Testfälle und die abschließende Bewertung der Methode.

6.8. Zusammenfassung

Die anforderungsorientierte Testfallgenerierung vereinigt die Vorteile von struktureller und funktionaler Testfallspezifikationen. Insbesondere wird ein hoher Grad der Automatisierung erreicht, und es ist gleichzeitig möglich, zu jeder identifizierten Einzelanforderung gezielt Testfälle zu generieren. Zudem erhält der Testingenieur über die „Stellschrauben“ Anforderungsklassifikation und Gewichtungen die Möglichkeit, die generierte Testsuite zu beeinflussen. Anforderungen, welche stärkeren Einfluss auf den Test insgesamt haben sollen, können somit priorisiert werden. Es ist zu beachten, dass sich eine höhere Gewichtung auf den Test hinsichtlich aller Anforderungen auswirkt. Anforderungen in priorisierten Klassen werden somit stärker im Zusammenspiel mit allen andern Anforderungen getestet.

Der Algorithmus zur Bildung der anforderungsspezifischen Teilmodelle, aus welchen die jeweiligen Testfälle generiert werden, basiert zunächst darauf, dass die Klassifikation der Anforderungen in das Modell übertragen wird. Dazu werden im Gesamtmodell

[5] Die vollständige Angabe aller möglichen Szenarien ist praktisch nur für triviale Systeme möglich.

die Transitionen, welche bei Ausführung des Szenarios schalten, mit der jeweiligen Anforderungsklasse assoziiert. Der zentrale Bestandteil jedes einzelnen Teilmodells ist der Pfad, welcher von dem entsprechenden Szenario durchlaufen wird. Ausgehend von den auf diesen Pfad erreichten Zuständen, werden weitere Transitionen hinzugenommen. Die Auswahl wird durch die Gewichte beschränkt, welche für die Anforderungsklassen vergeben wurden, sie gegeben die maximale Anzahl von Transitionen einer Klasse vor, welche zusätzlich gewählt werden kann. In den szenarienspezifischen Teilmodellen erfolgt die Testfallgenerierung unter Verwendung von strukturellen Testfallspezifikationen wiederum automatisch. Es folgt ein manuelles Review der Testfälle und gegebenfalls eine Anpassung der Gewichte sowie eine erneute Generierung der Teilmodelle und Testfälle.

7. Bewertung der Methode

In den vorhergehenden beiden Kapitel wurde die Methode zur anforderungsorientierten Testfallgenerierung mithilfe von Szenarien und eines Verhaltensmodells eingeführt. Die Validierung der Methode hinsichtlich ihres praktischen Nutzens ist nun Gegenstand dieses Kapitels.

Die Validierung eines methodischen Ansatzes hinsichtlich eines möglichen Vorteils in der Praxis der Software-Entwicklung müsste letztlich anhand einer ökonomischen Betrachtung geführt werden. Dazu wäre der Nachweis zu führen, dass der zusätzliche Nutzen der Methode die nötigen Mehraufwände im Vergleich zu alternativen Ansätzen übersteigt. Ein derartige Evaluierung ist für Testmethoden im Allgemeinen kaum durchführbar. Dies hat seine Ursache zum einen darin, dass in einer Kosten-/Nutzenanalyse vor allem auch die (Folge-)Kosten nicht entdeckter Fehler bewertet werden müssen. Wie in Kapitel 3 bereits dargestellt, lässt sich die Frage nach der Eignung einer Testsuite zur Fehleraufdeckung nicht zufriedenstellend beantworten; zumindest nicht, bevor das System nicht mehr genutzt wird. Die Frage der Wirtschaftlichkeit ist daher heute auch für den modellbasierten Test im Allgemeinen noch nicht abschließend beantwortet.

Insofern wird in der vorliegenden Arbeit eine derartige Evaluierung nicht durchgeführt. Das Fehlen eines vom wirtschaftlichen Nutzen abgeleiteten Gütemaßes für Testsuiten ist ein Grund dafür, andererseits erfordert eine empirische Absicherung das Vorliegen umfangreicher Experimente. Repräsentative Fallstudien, welche hierzu ausreichend Daten (Anforderungen, Vergleichstestfälle, Angaben zu nicht entdeckten Fehlern, Erhebungen zu Fehlerfolgekosten) umfassen, sind kaum verfügbar, und in der Durchführung stellen sich prinzipielle Schwierigkeiten, insbesondere hinsichtlich der Gewinnung allgemeingültiger Aussagen.

Es ist auch nicht Anspruch der vorgeschlagenen Methode, dass durch ihre Anwendung Testfälle ermittelt werden können, welche eine höhere Fehlerentdeckung garantieren. Das betrachtete Gütekriterium ist dagegen die Eignung der Testfälle und Testsuiten zum Test der spezifizierten Einzelanforderungen. „Eignung“ ist hier vor allem derart zu verstehen, dass die Menge der Testfälle geeignet ist, um den Test der einzelnen Anforderungen gegenüber den Stakeholdern nachweisen zu können, und insbesondere, dass das vorgeschlagene Verfahren hierzu vorteilhaft im Vergleich zu anderen Testmethoden ist. Im Sinne der in Abschnitt 3.2 eingeführten Systematisierung von Qualitätskriterien wird somit die Relation von Testfällen zu Anforderungen betrachtet.

Im Folgenden wird das bereits eingeführte Fallbeispiel der Fensterheberfunktion verwendet, um diese Anforderungsorientierung der Testfälle zu diskutieren. Zunächst folgt dazu eine informelle Analyse, zusätzlich wird weiter ein Maß zur Bewertung der Anforderungsüberdeckung eingeführt, das sich aus den Überlegungen aus Abschnitt 3.3 ableitet und die Ergebnisse des Fallbeispiels bewertet.

7.1. Informelle Analyse hinsichtlich der Anforderungsorientierung

In diesem Abschnitt werden zunächst die ermittelten anforderungsspezifischen Teilmodelle und anschließend die daraus gewonnen Testfälle analysiert. Zur Analyse der Testfälle werden diese mit Testfällen in alternativen Testsuiten verglichen.

Wesentlicher Gegenstand der Analyse ist die Frage, ob sich die gewählte Gewichtung der Anforderungen in der beabsichtigten Weise auf die Teilmodelle beziehungsweise Testfälle auswirkt.

7.1.1. Auswahl der Teilmodelle

Die Gewichtung der Anforderungsklasse hat zum Ziel, dass Modellteile, welche aufgrund von Anforderungen in einer höher bewerteten Anforderungsklasse eingeführt wurden, sich verstärkt in den Teilmodellen zu allen Anforderungen wiederfinden. Die Anforderungen der höher gewichteten Klassen sollen den Test insgesamt stärker beeinflussen.

In dem Fallbeispiel führt die Hinzunahme der neuen Anforderungen der Klasse *added* $\in$ classReq(r) dazu, dass ein neuer Zustand *Pause* eingeführt wird (vgl. Beispiel 14, Seite 107). Somit ist beabsichtigt, dass die eingehenden und ausgehenden Transitionen dieses Zustands bei einer relativ höheren Gewichtung der Klasse *added* in einer höheren Anzahl von Teilmodellen zu finden sind als bei einer gleichwertigen Gewichtung.

In den Tabellen in Anhang C sind für unterschiedliche Gewichtungen die Transitionen angegeben, welche im Fallbeispiel jeweils für die anforderungsspezifischen Teilmodelle gewählt wurden. Die Transitionen t_{20}, t_{28}, t_{37}, t_{41} sowie t_{45} sind diejenigen Transitionen, welche in den Szenarien der Klasse *added* vorkommen und in den neuen Zustand *Pause* führen beziehungsweise von diesem ausgehen. Tabelle 7.1 gibt einen Überblick über die Häufigkeit des Vorkommens dieser Transitionen in den Teilmodellen.

Wie aus der Tabelle ersichtlich ist, sind die entsprechenden Transitionen bei höherer Gewichtung der Klasse (weight(*added*) = 2) in einer größeren Anzahl von Teilmodellen vertreten. Insbesondere sind bei der Gewichtung $(1, 2, 1)$ alle Transitionen dieser Klasse in allen anforderungsspezifischen Teilmodellen enthalten. Auf die Wahl der übrigen

Tabelle 7.1.: Vorkommen von Transitionen der Klasse *added* in Teilmodellen unterschiedlicher Gewichtung:

Gewichtung				
weight(*initial*)	1	1	1	1
weight(*added*)	1	**2**	1	**2**
n	0	0	1	1
Anzahl der Teilmodelle für welche Transition t_i gewählt wurde:				
t_{20}	4	**11**	10	**12**
t_{28}	4	**11**	10	**12**
t_{37}	4	**6**	6	**12**
t_{41}	4	**6**	6	**12**
t_{45}	4	**6**	6	**12**

Transitionen hat die alleinige Veränderung der Gewichtung weight(*added*) dagegen keinen Einfluss (vgl. Tabellen in Anhang C). Daraus folgt, dass die Transitionen aus und in den Zustand *Pause* die Testfallgenerierung insgesamt stärker beeinflussen. Da andere Transitionen dagegen in mehreren einzelnen Teilmodellen fehlen, wird die Testfallgenerierung nach einer strukturellen Überdeckung in wechselnde Pfade geführt, um die höher gewichteten Transitionen dennoch zu erreichen. Da Transitionen t mit $added \in \text{classTrans}(t)$ unmittelbar den Szenarien zugeordnet werden können, führt dies zu einem höheren Einfluss der Anforderungsklasse *added* auf die Gesamtmenge der Testfälle. Dies bestätigt auch die folgende Analyse der erzeugten Testsuite.

7.1.2. Vergleich von Testsuiten

Im Anhang D sind die Testfälle der Fensterheberfunktion angeführt, welche mit der beschriebenen Methode aus den anforderungsspezifischen Teilmodellen ermittelt wurden. Dabei wurde zur Teilmodellbildung die Gewichtung $\text{weight}(initial) = 1, \text{weight}(added) = 2, n = 1$ angewendet, zur Testfallgenerierung in den Teilmodellen wurde die Überdeckung der Transitionen als strukturelle Testfallspezifikation verwendet. In Anhang E findet sich die daraus zusammengefasste Testsuite, bei welcher Testfälle, welche Präfixe eines anderen Testfalls sind, eliminiert wurden.

Zur weiteren Validierung der Methode wird nun diese Testsuite hinsichtlich ihrer Anforderungsorientierung mit zwei alternativen Testsuites verglichen:

1. Mit einer Testsuite, welche nach *struktureller Überdeckung im Gesamtmodell*[1] ermittelt wurde, sowie

[1] Auch im Gesamtmodell wurde wiederum die Überdeckung von Transitionen als Testfallspezifikation verwendet

Tabelle 7.2.: Vergleich der Testsuiten

Testsuite	TS_{REQ}	TS_{struct}	TS_{random}
Testfall-generierung	anforderungs-orientiert	Transitions-überdeckung im Gesamtmodell	randomisierte Eingaben
Anzahl von Testfällen	119	45	119
Testschritte insgesamt	460	92	470
mittlere Testfalllänge (Schritte)	3,9	2,0	3,9

Tabelle 7.3.: Anzahl der Testfälle der verschiedenen Testsuites, in welchen Transition t_i schaltet

	TS_{REQ}		TS_{struct}		TS_{random}	
	Anzahl Testfälle	rel. Anteil	Anzahl Testfälle	rel. Anteil	Anzahl Testfälle	rel. Anteil
t_{20}	18	15,1 %	12	26,7 %	8	6,7 %
t_{28}	17	14,3 %	1	2,2 %	4	3,4 %
t_{37}	4	3,4 %	1	2,2 %	6	5,0 %
t_{41}	4	3,4 %	1	2,2 %	3	2,5 %
t_{45}	4	3,4 %	1	2,2 %	4	3,4 %

2. mit einer Testsuite, welche auf randomisierten Testeingaben basiert – dabei wurde der Umfang so gewählt, dass er mit jener der anforderungsorientierten Testsuite vergleichbar ist.

Tabelle 7.2 gibt einen Überblick über die verschiedenen Testsuiten. Daraus ist bereits ersichtlich, dass die in dieser Arbeit vorgestellte Methode zur anforderungsorientierten Testfallgenerierung eine höhere Anzahl und auch längere (Anzahl der Testschritte pro Testfall) Testfälle ermittelt, als eine Testfallgenerierung nach struktureller Überdeckung aus dem Gesamtmodell der Fensterheberfunktion.

Um die beabsichtigte Eignung der erzielten Testfälle hinsichtlich der Orientierung an den Einzelanforderungen zu überprüfen, wird wiederum untersucht, ob sich die höher gewichteten Anforderungen verstärkt in den Testfällen widerspiegeln.

Wie im vorhergehenden Abschnitt werden dazu wieder die Transitionen t_{20}, t_{28}, t_{37}, t_{41} sowie t_{45} näher betrachtet; diese Transitionen wurden nur aufgrund der Anforderungen der Klasse *added* eingeführt. Für diese Analyse wird nun betrachtet, wie häufig diese Transitionen in den Testfällen ausgeführt werden. Tabelle 7.3 zeigt die entsprechenden Häufigkeiten.

Im Vergleich der Testsuite TS_{REQ}, welche nach der in dieser Arbeit beschriebenen Methode erstellt wurde, mit der Testsuite TS_{struct} (strukturelle Überdeckung im Gesamtmodell) zeigt sich, dass alle betreffenden Transitionen der Anforderungsklasse *added* in einer höheren absoluten Anzahl von Testfällen der Testsuite TS_{REQ} zur

Ausführung kommen, als in der Testsuite TS_{struct}. Mit Ausnahme von Transition t_{20} ist auch der relative Anteil von Testfällen, welche diese Transitionen erreichen, in TS_{REQ} höher. Besonders auffällig ist die Abweichung bei Transition t_{28}. Ursache hierfür ist, dass die Testfallgenerierung der strukturellen Überdeckung im Gesamtmodell zu Testfällen führte, welche nur die Transition t_{20} ($Down \rightarrow Pause$) ausführen um den Zustand *Pause* zu erreichen und nie die Transition t_{28} ($Up \rightarrow Pause$). Dies bedeutet, dass die Transitionen aus diesem Zustand ausschließlich im Zusammenhang mit der Teilfunktion „Öffnen“ der Fensterheberfunktion getestet werden. Im Gegensatz dazu umfasst die Testsuite TS_{REQ} eine ausgewogenere Verteilung von Testfällen, und insbesondere sind die höher gewichteten Anforderungen stärker berücksichtigt.

Im Vergleich mit der Testsuite TS_{random} aus randomisierten Eingaben führen die Testfälle der anforderungsspezifischen Testsuite TS_{REQ} die betreffenden Transitionen, im Mittel häufiger aus; einzige Ausnahme bildet die Transition t_{37}. Weiter ist bei den zufällig erzeugten Testfällen kein Zusammenhang zu den Anforderungsszenarien und deren Gewichtung zu erkennen.

7.2. Bewertung der Anforderungsüberdeckung

Nach der informellen Analyse im vorhergehenden Abschnitt soll nun eine Bewertung der Anforderungsüberdeckung durchgeführt werden. Grundlage hierfür sind die Darstellungen aus Kapitel 3 zur Qualitätsbewertung von Testfällen. In den Tabellen 3.2 und 3.3 (Seite 37) fällt die Bewertung der *Anforderungsüberdeckung* in die Klasse 2 der dort klassifizierten Gütemaße, da Testfälle in Relation zu dokumentierten Anforderungen gesetzt werden. Es wird also allein betrachtet wie „passend“ (im Sinne der Darstellung in den folgenden Abschnitten) die Testfälle zu den dokumentierten Anforderungen sind. Insbesondere ist damit aber keine Bewertung möglich, ob die Testfälle besonders fehlerträchtige Teile des zu testenden Systems adressieren.[2]

Zunächst soll diskutiert werden, wie sich insbesondere das, in Kapitel 3, Abschnitt 3.3 eingeführte, Bewertungskriterium der *Prüfung* einer Anforderung auf Szenarien übertragen lässt. Da ein Szenario nur einen exemplarischen Ablauf einer Anforderung beschreibt, ist es notwendig, Annahmen über die Verallgemeinerung eines Szenarios r zur intendierten Anforderung ρ zu treffen. Diese Annahmen und deren Rechtfertigung sind Gegenstand der folgenden Abschnitte. Damit werden Merkmale begründet, welche zur Bewertung der Überdeckung von Testfällen und Anforderungsszenarien herangezogen werden können.

Die Abläufe, welche durch die Szenarien beschrieben werden, sind im Allgemeinen von hoher Priorität. Dies begründet sich dadurch, dass die Stakeholder versuchen,

[2] siehe hierzu auch Kapitel 3, Abschnitt 3.3.1.

vor allem solche Szenarien anzugeben, welche dem aus Sicht der Stakeholder beabsichtigten Zweck des Systems nahe kommen. Die triviale Anforderungsüberdeckung, dass zu jeder Anforderung mindestens ein Testfall existiert, ist bereits durch das Szenario gegeben, wenn es als Testfall verwendet wird. Ein Szenario ist hier aber vor allem ein exemplarischer Repräsentant einer Menge von Systemabläufen, welche einer intendierten Anforderung ρ entsprechen. Es sind Variationen in den Systemabläufen möglich. Trotzdem betreffen diese Varianten aus Sicht der Stakeholder jeweils dieselben Anforderungen. Eine Menge von Testfällen, welche nun verstärkt Varianten des Szenarios umfasst, bedeutet eine stärkere Überdeckung der Anforderung. Szenarien stellen auch exemplarische Testfälle dar. Dadurch kann die Variantenbeziehung zwischen zwei Szenarien formal dahingehend definiert werden, ob beide Szenarien dieselbe Anforderung prüfen:

Definition 14 (Varianten von Szenarien) *Eine Szenario ist eine Variante eines anderen Szenarios, falls beide Szenarien dieselbe Anforderung ρ prüfen. Dies wird durch das Prädikat*

$$\text{variant} \in (I^* \times O^*) \times (I^* \times O^*) \times \mathbb{P}(FUN_{(I \triangleright O)} \to \mathbb{B}\text{ool}) \to \mathbb{B}\text{ool}$$

angegeben. variant *ist für zwei Szenarien $r, t \in I^* \times O^*$ und eine Menge von Anforderungen $R = \{\rho_1, \rho_2, \ldots, \rho_n\} \subseteq FUN_{(I \triangleright O)} \to \mathbb{B}\text{ool}$ definiert als:*

$$\text{variant}(r, t, R) \overset{\text{def}}{\Leftrightarrow} \exists\ \rho \in R : \text{checks}(r, \rho) \wedge \text{checks}(t, \rho)$$

Ob ein Szenario eine Variante eines anderen Szenarios ist, hängt somit stark davon ab, welcher Funktionsumfang von den Anforderungen $\rho_1, \rho_2 \ldots \rho_n$, aus denen sich die Menge der Anforderungen R zusammensetzt, jeweils betroffen ist. Mangels einer expliziten vollständigen Spezifikation der Prädikate $\rho_i \in R$ ist die eindeutige Bestimmung der Varianten ohne zusätzliche Informationen nicht möglich. Im Folgenden werden aber die Eigenschaften, welche bei der Prüfung einer als Prädikat gegeben Anforderung gelten, verwendet, um zumindest näherungsweise wahrscheinliche Varianten eines Szenarios zu bestimmen.

7.2.1. Kriterien zur Auswahl und Formulierung von Szenarien

Um zu beurteilen, unter welchen Bedingungen Systemabläufe Varianten eines Szenarios sind und damit dieselbe Anforderung ρ adressieren, sollen zunächst Kriterien untersucht werden, welche zur Auswahl der Szenarien führen. Entscheidend ist dabei, dass die Eingaben so gewählt werden, dass diese, unter Berücksichtigung der einzelnen Anforderungen, zu einem beobachtbaren Verhalten in der Ausgabe führen. Die Beschränkung auf die Betrachtung der Eingaben an das System ist ausreichend, da Gegenstand dieser Arbeit ausschließlich deterministische Systeme sind. Wie auch aus

Beispiel 7 (Kapitel 3, Seite 58) zur Funktion buffer oder aus Beispiel 10 (Kapitel 5, Seite 89) zur Fensterhebersteuerung ersichtlich ist, finden etwa folgende Auswahlkriterien bei der Wahl der Eingaben Verwendung:

1. Eingabeaktion beeinflusst unmittelbar Ausgabe
2. Temporale Ordnung der Eingabeaktionen beeinflusst die Ausgabe
3. Eingabesequenz beeinflusst die Ausgabe

Eine Eingabeaktion kann also unmittelbar auf die Ausgabe wirken, bei der Puffer-Funktion aus Beispiel 1 (Kapitel 3, Seite 42) ist dies etwa für Anforderung ρ_4 der Fall, wobei das get-Kommando zur Ausgabe des gespeicherten Wertes führt. Die temporale Ordnung der Eingabeaktionen ist in den Szenarien zu ρ_2 bis ρ_5 von Bedeutung: Bei anderer Reihenfolge der Eingabeaktionen kommen die entsprechenden Anforderungen nicht zum Tragen. Sequenzen von Anforderungen bestimmen beispielsweise die Szenarien zu ρ_3 – hier ist entscheidend für die Ausgabe, dass nur leere Eingabeaktionen dem get-Kommando vorausgehen.

7.2.2. Annahmen über Varianten

Mithilfe dieser Kriterien zur Formulierung von Szenarien sowie den durch das Prädikat checks definierten Eigenschaften für die Prüfung einer Anforderung sollen nun Annahmen abgeleitet werden, welche anzeigen, ob ein Szenario t eine Variante eines gegebenen Szenarios r ist. Da es im Allgemeinen nicht möglich ist, aus einem gegebenen Szenario r auf die zugrunde liegende Anforderung ρ zu schließen, können ohne Angabe von ρ keine absoluten Regeln genannt werden, wann die Variantenbeziehung zweier Szenarien erfüllt ist. Eine solche absolute Aussage ist aber nicht nötig, da bei der Bewertung der Anforderungsüberdeckung lediglich eine relative Bewertung vorgenommen werden soll. Es soll also die Frage beantwortet werden, von welchen Szenario t oder t' es häufiger anzunehmen ist, dass es sich dabei um eine Variante von r hinsichtlich einer (nicht näher spezifizierten) Menge von Anforderung R handelt. Dazu werden im Folgenden Annahmen über die Anforderungen $\rho \in R$ getroffen, welche der Formulierung des Szenarios r zu Grunde liegen.

Für die Beurteilung der möglichen Varianteneigenschaft ist die in Kapitel 3 eingeführte Relation $\succ_\rho$ hilfreich, welche angibt, ob ein Testfall eine Anforderung ρ stärker prüft als ein anderer Testfall (Definition 8, Seite 47). Aus den oben angegebenen Auswahlkriterien für Eingaben von Szenarien lassen sich Muster der zu Grunde liegenden Anforderungen ableiten:

Direkter Einfluss Ein $X \in I$ in der Eingabe impliziert eine geforderte Eigenschaft q_{dir} der Ausgabe:

$$\rho_{dir}(f) \overset{\text{def}}{\Leftrightarrow} \forall u, v \in I^* : q_{dir}\left(f\left(u \frown \langle X \rangle \frown v\right)\right)$$

Temporale Ordnung Ein $Y \in I$, welches in der Eingabe auf ein $X \in I$ folgt, impliziert eine geforderte Eigenschaft q_{ord} der Ausgabe:

$$\rho_{ord}(f) \overset{\text{def}}{\Leftrightarrow} \forall u, v, w \in I^* : q_{ord}\left(f\left(u \frown \langle X \rangle \frown v \frown \langle Y \rangle \frown w\right)\right)$$

Sequenz Eine Eingabesequenz $x \in I^* : p_x(x)$, die eine Eigenschaft $p_x \in I^\omega \to \mathbb{B}\text{ool}$ erfüllt und auf die unmittelbar ein $Y \in I$ folgt, impliziert eine geforderte Eigenschaft q_{seq} der Ausgabe:

$$\rho_{seq}(f) \overset{\text{def}}{\Leftrightarrow} \forall u, v, x \in I^* mit p_x(x) : q_{seq}\left(f\left(u \frown x \frown \langle Y \rangle \frown v\right)\right)$$

Dabei bezeichnen $q_{dir}, q_{ord}, q_{seq} \in O^\omega \to \mathbb{B}\text{ool}$ Prädikate über die erwartete Ausgabe, wobei jedes Prädikat $q \in \{q_{dir}, q_{ord}, q_{seq}\}$ jeweils nur für eine echte Teilmenge aller möglichen Ausgaben erfüllt ist, also $\{outs \in O^\omega \mid q(outs)\} \subsetneq O^\omega$ gilt.

Diese drei Muster von formalen Anforderungen ergeben bei der Anwendung auf ein gegebenes Szenario wiederum komplexere Anforderungen. Für die Eingabe

$$\mathsf{in}.r = \langle a_1, a_2, \ldots, a_n \rangle \in I^*$$

eines Szenarios r ergeben sich bei jeweils konsequenter Anwendung der Anforderungsmuster die folgenden Anforderungen (q_i, $q_{i,j}$ und $q'_{i,j}$ bezeichnen wiederum beliebige Prädikate, welche die Forderung an die Ausgabe spezifizieren):

$$\rho_{dir}[r](f) \Leftrightarrow \bigwedge_{i=1}^{n} \left(q_i\left(f\left(u \frown \langle a_i \rangle \frown v\right)\right) \ \forall u, v \in I^*\right)$$

$$\rho_{ord}[r](f) \Leftrightarrow \bigwedge_{i=1}^{n-1} \bigwedge_{j=i+1}^{n} \left(q_{i,j}\left(f\left(u \frown \langle a_i \rangle \frown v \frown \langle a_j \rangle \frown w\right)\right) \ \forall u, v, w \in I^*\right)$$

$$\rho_{seq}[r](f) \Leftrightarrow \bigwedge_{i=1}^{n-1} \bigwedge_{j=i+1}^{n} \left(q'_{i,j}\left(f\left(u \frown \langle a_i, \ldots, a_{j-1}, a_j \rangle \frown v\right)\right) \ \forall u, v \in I^*\right)$$

Dabei wird für jedes der Prädikate $q, q' \in \left\{q_1, \ldots, q_n, q_{1,2} \ldots q_{n-1,n}, q'_{1,2} \ldots q'_{n-1,n}\right\}$ mit $q' \neq q$ angenommen, dass es nicht für die vollständige Menge der Ausgaben gültig ist, und dass zwei unterschiedliche q, q' jeweils von unterschiedlichen Mengen von Ausgaben erfüllt werden:

$$\{outs \in O^\omega \mid q(outs)\} \subsetneq O^\omega \tag{7.1}$$

$$\wedge \{outs \in O^\omega \mid q(outs)\} \neq \{outs \in O^\omega \mid q'(outs)\} \tag{7.2}$$

Damit stellt jedes q, q' eine individuelle Forderung an die Ausgabe dar.

Aus der konjunktiven Verknüpfung resultieren sehr restriktive Forderungen, welche unterstellen, dass jedes, in einem Anforderungsszenario auftretende, Merkmal eigenständig zu einer bestimmten Erwartung an die Ausgabe führt. Letzteres trifft im Allgemeinen nicht zu. Diese potentiellen Anforderungen, welche aus einem Szenario ableiten lassen, sind aber hilfreich, um Eigenschaften von Testfällen zu bestimmen, welche zur Beurteilung der Anforderungsüberdeckung verwendet werden können. Diese Eigenschaften umfassen gemeinsame Eingaben, die Reihenfolge der Eingaben, ununterbrochene Sequenzen und Wiederholungen von Eingaben.

Gemeinsame Eingaben

Ein erstes Merkmal, nach dem die Überdeckung einer als Szenario gegebenen Anforderung durch einen Testfall beurteilt werden kann, sind die gemeinsamen Eingaben in Anforderung und Testfall. Mit

$$CI_{r,t} \stackrel{\text{def}}{=} \mathsf{rng.in}.r \cap \mathsf{rng.in}.t$$

wird dazu die Menge der Aktionen $\{in_1, \ldots, in_n\} \subseteq I$ bezeichnet, welche in den Eingaben beider Szenarien r, t auftreten. Mit t wird hier ein Testfall angegeben, die Menge $CI_{r,t}$ ist dabei aber allgemein für zwei Ein-/Ausgabe Szenarien $r, t \in I^* \times O^*$ definiert. $\mathsf{rng}.x$ bezeichnet dabei die Menge der Aktionen, welche in einem Strom x vorkommen[3]. Ein Testfall t, welcher ausschließlich Eingabeaktionen umfasst, die nicht in der Eingabe $\mathsf{in}.r = \langle a_1, a_2, \ldots, a_n \rangle$ einer Anforderung r vorkommen, prüft diese nicht. In diesem Fall ist

$$CI_{r,t} = \{\},$$

damit gilt

$$\forall i \in [1 \ldots n] : \nexists u, v \in I^* : \mathsf{in}.t = u \frown \langle a_i \rangle \frown v$$

und weiter

$$\begin{aligned}
& \rho_{dir}[r](f) \Leftrightarrow \bigwedge_{i=1}^{n} (\mathsf{false} \Rightarrow (f(\mathsf{in}.t) = outs \wedge q_i(outs))) \Leftrightarrow \mathsf{true} \\
\Rightarrow\ & \forall outs \in O^\omega : \rho_{dir}[r](\mathsf{in}.t, outs) = \mathsf{true} \\
\Rightarrow\ & O^\omega_{\rho_{dir}[r],\mathsf{in}.t} = O^\omega \\
\Rightarrow\ & \text{checks}(\rho_{dir}[r], t) = \mathsf{false}.
\end{aligned}$$

Somit ist gezeigt, dass derartige Testfälle die Anforderung $\rho_{dir}[r]$ nicht prüfen. Da dann auch die Bedingungen an die Eingabe der Anforderungen $\rho_{ord}[r]$ und $\rho_{seq}[r]$ nicht erfüllt werden, existiert zu dem Paar r, t keine Anforderung, für welche das Prädikat checks erfüllt ist.

[3] Vgl. Anhang H.

Eine Prüfung der Anforderung ρ_{dir} ist bereits gegeben, falls Szenario und Testfall nur eine gemeinsame Eingabeaktion aufweisen, $CI_{r,t}$ somit nicht leer ist. Zu einem $a_i \in CI_{r,t}$, welches Bestandteil der Testeingabe $\mathsf{in}.t = u \frown \langle a_i \rangle \frown v$ ist, wird, entsprechend ρ_{dir}, für die Ausgabe die Eigenschaft $q_i(\mathsf{out}.t)$ gefordert. Da für q_i gefordert wurde, dass $\{outs \in O^{\omega} | q_i(outs)\} \subsetneq O^{\omega}$ ist, ist das Prädikat $\text{checks}(\rho_{dir}[r], t)$ und damit auch die Varianteneigenschaft $\text{variant}(r, t)$ erfüllt.

Aufgrund der Formulierung der angenommenen Anforderungen $\rho_{dir}[r], \rho_{ord}[r]$ und $\rho_{seq}[r]$ zu einem Anforderungsszenario r ist die Varianteneigenschaft nur von sehr geringer Aussagekraft. Daher stehen, wie bereits eingangs erläutert, relative Aussagen im Vordergrund. Hinsichtlich der Anforderung $\rho_{dir}[r]$ bedeutet dies, dass eine größere Anzahl von Aktionen, welche sowohl in der Eingabe des Szenarios r als auch in der Eingabe des Testfalls t vorkommen, zu einer höheren Bewertung der Überdeckung führen sollen. Für zwei Testfälle t, t' mit

$$\mathsf{in}.t = u \frown v$$
$$\mathsf{in}.t' = u \frown \langle a_i \rangle \frown v$$

soll somit gelten:

$$t' \succ_{\rho_{dir}[r]} t$$

Je nachdem, ob die Aktionen in der Eingabe des Testfalls auftreten oder nicht, oder ob sich diese wiederholen, sind drei Fälle zu unterscheiden:

Im **ersten Fall** gilt $\forall j \in [1 \ldots n] : a_j \notin \mathsf{rng}.\mathsf{in}.t$, also dass die Eingabe von t ausschließlich Aktionen umfasst, welche nicht in der Eingabe des Szenarios auftraten wurde zuvor beschrieben. Die Relation $t' \succ_{\rho_{dir}[r]} t$ ist dann trivial erfüllt.

Im **zweiten Fall** gilt $\exists j \in [1 \ldots n] : a_j \in \mathsf{rng}.\mathsf{in}.t \wedge a_j \neq a_i$; es gibt also eine Aktion a_j welche in dem Szenario r und in den Testfällen t und t' vorkommt, aber ungleich der Aktion a_i ist, welche zusätzlich in t' vorhanden ist. Hinsichtlich $\rho_{dir}[r]$ ergeben sich damit die folgenden Forderungen an die Menge der gültigen Ausgaben zu t und t':

$$O^{\omega}_{\rho_{dir}[r],\mathsf{in}.t} = \{outs | q_j(outs)\}$$
$$O^{\omega}_{\rho_{dir}[r],\mathsf{in}.t'} = \{outs | q_i(outs) \wedge q_j(outs)\}$$

Da für alle Prädikate q_k die Bedingungen (7.1) und (7.2) angenommen werden, gilt

$$O^{\omega}_{\rho_{dir}[r],\mathsf{in}.t'} \subsetneq O^{\omega}_{\rho_{dir}[r],\mathsf{in}.t}$$

und somit wiederum

$$t' \succ_{\rho_{dir}[r]} t.$$

Der **dritte Fall**, $a_i \in \mathsf{rng}.\mathsf{in}.t$, a_i kommt bereits in der Eingabe zu Testfall t und demnach zweimal in der Eingabe von t' vor – wird später im Abschnitt zur Wiederholung von Eingaben betrachtet.

Wahrung der Ordnung von Eingaben

Falls Anforderung und Testfall gemeinsame Eingaben aufweisen, und somit eine (teilweise) Überdeckung zwischen Anforderung und Test gegeben ist, ist von weiterer Bedeutung, ob die Eingaben im Testfall in gleicher oder unterschiedlicher Reihenfolge wie in dem Anforderungsszenario auftreten. Die Überdeckung durch einen Testfall t', in welchen Eingaben in derselben Reihenfolge wie in der Anforderung auftreten, wird höher bewertet als die Überdeckung durch einem Testfall t, in welchen die Eingaben in vertauschter Reihenfolge vorkommen. Für zwei Testfälle t, t' mit den Eigenschaften

$$\begin{aligned}\mathsf{in}.t &= u \frown \langle a_j \rangle \frown v \frown \langle a_i \rangle \frown w \\ \mathsf{in}.t' &= u \frown \langle a_i \rangle \frown v \frown \langle a_j \rangle \frown w\end{aligned}$$

sowie $i < j$ und $a_i \neq a_j$ gilt somit wiederum, dass

$$t' \succ_{\rho_{ord}[r]} t.$$

Dies ergibt sich unmittelbar aus der allgemeinen Anforderung ρ_{ord}. Die Anforderung $\rho_{ord}[r]$ ist im Allgemeinen zu restriktiv: Im Regelfall wird die Reihenfolge nicht aller möglichen Aktionen a_i, a_j in einem Szenario von Bedeutung sein. Da eine präzisere formale Spezifikation der Anforderung jedoch nicht vorliegt, soll unterstellt werden, dass jede paarweise Anordnung von Aktionen potentiell von Interesse ist. Da wiederum nur eine relative Bewertung von Testfällen vorgenommen wird, ist dies aber vernachlässigbar, da nur von Interesse ist, ob ein Testfall diese fiktive Anforderung $\rho_{ord}[r]$ stärker prüft als eine anderer Testfall. Aus $\rho_{ord}[r]$ folgt im Fall der Eingabe $\mathsf{in}.t'$, dass

$$\forall outs \in O^{\omega}_{\rho_{ord}[r],\mathsf{in}.t'} : q_{i,j}(outs)$$

An die Eingabe $\mathsf{in}.t$ wird diese Forderung dagegen nicht gestellt. Zusammen mit den Annahmen (7.1) und (7.2) gilt damit wieder

$$O^{\omega}_{\rho_{ord}[r],\mathsf{in}.t'} \subsetneq O^{\omega}_{\rho_{ord}[r],\mathsf{in}.t}.$$

Wiedergabe von Eingabesequenzen

Wenn Eingaben der Anforderung im Testfall in der selben Reihenfolge auftreten, ist weiterhin zu unterscheiden, ob im Testfall die Eingabesequenz der Sequenz in der Anforderung entspricht. Zunächst ist es möglich, dass eine Sequenz des Szenarios unterbrochen ist. Dies ist bei einem Testfall t in Vergleich zu einem Testfall t' der Fall, wenn diese die Eigenschaften

$$\begin{aligned}\mathsf{in}.t &= u \frown \langle a_i \rangle \frown v \frown \langle a_{i+1} \rangle \frown w \\ \mathsf{in}.t' &= u \frown \langle a_i, a_{i+1} \rangle \frown w\end{aligned}$$

erfüllen. Da $\rho_{seq}[r]$ für die Eingabe $\mathsf{in}.t'$ fordert, dass

$$\forall outs \in O^{\omega}_{\rho_{seq}[r],\mathsf{in}.t'} : q'_{i,i+1}(outs)$$

für die Eingabe zu t dies jedoch nicht gefordert wird, gilt aufgrund der Annahmen (7.1) und (7.2)

$$O^{\omega}_{\rho_{seq}[r],\mathsf{in}.t'} \subsetneq O^{\omega}_{\rho_{seq}[r],\mathsf{in}.t}$$

und somit

$$t' \succ_{\rho_{dir}[r]} t.$$

Außer einer Unterbrechung einer Sequenz führt auch deren verkürzte Wiedergabe zu einer schwächeren Prüfung der Anforderung ρ_{seq}. Hierzu seien zwei Testfälle t, t' mit den Eigenschaften

$$\begin{aligned}\mathsf{in}.t &= u \frown \langle a_i, \ldots, a_j \rangle \frown v \\ \mathsf{in}.t' &= u \frown \langle a_i, \ldots, a_j, a_{j+1} \rangle \frown v\end{aligned}$$

sowie $i \leq j < n-1$ betrachtet. Aus $\rho_{seq}[r]$ folgt für die Mengen der gültigen Ausgaben zu t und t':

$$\begin{aligned}O^{\omega}_{\rho_{seq}[r],\mathsf{in}.t} &= \left\{ outs \in O^{\omega} \middle| q'_{i,i+1} \wedge \ldots \wedge q'_{j-1,j} \right\} \\ O^{\omega}_{\rho_{seq}[r],\mathsf{in}.t'} &= \left\{ outs \in O^{\omega} \middle| q'_{i,i+1} \wedge \ldots \wedge q'_{j-1,j} \wedge q'_{j,j+1} \right\}\end{aligned}$$

auch hieraus ergibt sich wieder, zusammen mit der Annahme (7.2),

$$O^{\omega}_{\rho_{seq}[r],\mathsf{in}.t'} \subsetneq O^{\omega}_{\rho_{seq}[r],\mathsf{in}.t}$$

und es gilt

$$t' \succ_{\rho_{seq}[r]} t.$$

Wiederholungen von Eingaben

Bereits bei der Betrachtung der gemeinsamen Eingaben in Szenario und Testfall stellte sich zuvor die Frage, wie eine Wiederholung von Aktionen im Testfall zu bewerten ist. Ob also für zwei Testfälle t, t' mit

$$\begin{aligned}\mathsf{in}.t &= u \frown \langle a_i \rangle \frown v \frown w \\ \mathsf{in}.t' &= u \frown \langle a_i \rangle \frown v \frown \langle a_i \rangle \frown w\end{aligned}$$

mit $a_i \notin \mathsf{rng}.u \frown v \frown w$ ebenfalls

$$t' \succ_{\rho_{dir}[r]} t.$$

gelten soll.

Aus den auf Seite 157 angegebenen Annahmen zu Varianten von Szenarien lässt sich dies nicht unmittelbar ableiten. Die Annahme zum direkten Einfluss einer Aktion in der Eingabe auf die Ausgabe ist dort so formuliert, dass allein ein einmaliges Auftreten einer Aktion an einer beliebigen Stelle im Eingabestrom zu einer bestimmten Forderung q_{dir} an die Ausgabe führt. Dagegen kann der direkte Einfluss auch so interpretiert werden, dass jedes Auftreten einer Aktion eine neue Forderung an die Ausgabe stellt, es also eine Menge von unterschiedlichen Prädikaten q gibt, abhängig davon an welcher Stelle eine Aktion auftritt. Diese Annahme kann wie folgt formuliert werden:

$$\rho_{dirAll}[r](ins \mapsto outs) \Leftrightarrow$$
$$\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{\#ins} \Big(ins = \langle x_1, \ldots, x_{j-1} \rangle \frown \langle a_i \rangle \frown \langle x_{j+1}, \ldots, x_{\#ins} \rangle \Rightarrow q''_{i,j}(outs)\ \forall x_k \in I \Big)$$

Daraus folgt für die Ausgaben von t und t':

$$O^{\omega}_{\rho_{dirAll}[r],\mathsf{in}.t} = \Big\{ outs \in O^{\omega} \Big| q''_{i,j} \Big\}$$
$$O^{\omega}_{\rho_{dirAll}[r],\mathsf{in}.t'} = \Big\{ outs \in O^{\omega} \Big| q''_{i,j} \wedge q''_{i,j'} \Big\}$$

wobei $j = \#u + 1$ und $j' = \#u + \#v + 2$.

$$O^{\omega}_{\rho_{dirAll}[r],\mathsf{in}.t'} \subsetneq O^{\omega}_{\rho_{dirAll}[r],\mathsf{in}.t}$$

und somit auch

$$t' \succ_{\rho_{dirAll}[r]} t$$

sind dadurch und aufgrund der Annahmen (7.1) und (7.2) erfüllt.

Das wiederholte Auftreten von Eingabeaktionen derart zu bewerten, findet seine Berechtigung auch darin, dass im Allgemeinen nur Systeme über einem endlichen Zustandsraum von Interesse sind. Damit kann angenommen werden, dass nach einer ausreichend langen Eingabesequenz sich das System wieder in einem Zustand befindet, welcher bereits früher erreicht wurde. Ausgehend von einem bestimmten Zustand gelten die Anforderungen dann entsprechend erneut. Bezogen auf die gesamte Ausgabesequenz resultiert aus jeder Wiederholung aber eine neue Forderung an die Ausgabe, da sich auch die Ausgabesequenz wiederholt.

Diese Betrachtungen führen dazu, dass in diesem Sinne das jeweilige Anforderungsszenario selbst der „stärkste“ einzelne Testfall ist. Die identifizierten Merkmale sollen aber dazu dienen, die Anforderungsüberdeckung von Testfällen zu bewerten, die nicht mit dem Anforderungsszenario identisch sind. Insbesondere ist die Bewertung von unterschiedlichen Testfallmengen von Interesse. Bevor näher auf die Überdeckung einer Menge von Anforderungen durch eine Menge von Testfällen eingegangen wird, sollen zuvor getroffene Annahmen zunächst noch einmal gerechtfertigt werden.

7.2.3. Berechtigung der Annahmen

In den vorhergehenden Abschnitten wurden die Anforderungen $\rho_{dir}[r]$, $\rho_{ord}[r]$, $\rho_{seq}[r]$ sowie $\rho_{dirAll}[r]$, welche sich aus einem gegebenen Szenario r mit Eingabe $\mathsf{in}.r = \langle a_1, a_2, \ldots, a_n \rangle$ ableiten lassen, eingeführt. Hinsichtlich dieser Prädikate lässt sich somit bestimmen, ob ein Testfall t diese Anforderungen prüft, beziehungsweise, ob ein Testfall t' eine stärkere Prüfung der Anforderung vornimmt als der Testfall t. Mit diesen Prädikaten wurde eine Verallgemeinerung der Anforderungen angegeben, welche aus einem exemplarischen Anforderungsszenario r abgeleitet werden können.[4] Es wird somit versucht, aus einem Szenario die Intention der Anforderung zu extrapolieren.

Diese Extrapolation unterstellt, dass die einzelnen Zusammenhänge (direkter Einfluss, temporale Ordnung, Sequenzen) für jeden Teil des Szenarios eine eigene Anforderung begründen. In der Regel wird dies nicht vollständig der intendierten Anforderung ρ entsprechen, welche durch ein Szenario r exemplarisch angegeben ist. Beispielsweise können in dem Szenario Aktionen enthalten sein, welche für die Anforderung unerheblich sind und daher willkürlich gewählt wurden. Aus einer solchen Aktion soll alleine keine Forderung an die Ausgabe folgen. Damit ist für einen Testfall t, welcher zwar eine der Anforderungen $\rho_{dir}[r]$, $\rho_{ord}[r]$, $\rho_{seq}[r]$ oder $\rho_{dirAll}[r]$ prüft, nicht garantiert, dass t auch die intendierte Anforderung ρ prüft. Trotz dessen kann aus den Merkmalen der gemeinsamen Eingaben, der Wahrung der Ordnung, der Wiedergabe von Sequenzen und Wiederholungen von Eingaben auf eine Überdeckung der Anforderung geschlossen werden. Dies begründet sich aufgrund folgender Aspekte:

Prüfung als hinreichendes Merkmal Die Beobachtung, dass ein Testfall t eine Anforderung ρ prüft, also dass Prädikat $\mathrm{checks}(\rho, t)$ erfüllt ist, ist ein hinreichendes, nicht aber ein notwendiges Merkmal zur Feststellung einer Überdeckung einer Anforderung ρ durch den Testfall t. Wie bereits anfangs von Abschnitt 3.3 in Kapitel 3 dargestellt wurde, wird für die Erfüllung der Anforderungsüberdeckung meist nur gefordert, dass der Testfall aus einer Anforderung ableitbar ist beziehungsweise einer Anforderung zuordenbar ist. Diese Forderungen sind erfüllt, falls ein Testfall die Anforderung prüft, andererseits können auch Testfälle aus einer Ableitung abgeleitet oder dieser zugeordnet werden, welche diese nicht prüfen. Sinnvoll kann dies zum Beispiel sein, wenn (nicht eigens dokumentierte) Ausnahmesituationen zu Anforderungen betrachtet werden.

Charakteristische Szenarien Werden Szenarien zur Beschreibung von Anforderungen verwendet, ist es sinnvoll, die Szenarien so zu wählen, dass diese für die Anforderung charakteristische Repräsentanten der entsprechenden Systemabläufe sind. Charakteristische Szenarien sind dabei solche, bei denen die Aktionen so gewählt werden, dass diese sich unmittelbar aus der Anforderung ergeben. Dagegen soll

[4] Wobei hierzu nur der Eingabeteil der Anforderungen betrachtet wurde. Zu den Ausgaben wurden nur die nicht näher bestimmten Prädikate $q_i, q_{i,j}, q'_{i,j}, q''_{i,j}$ angegeben. Dies ist ausreichend, da die Ausgaben für die Bestimmung der Überdeckung nicht berücksichtigt wurden.

das Szenario möglichst keine Teile umfassen, welche für die Anforderung ohne Belang sind, und weitgehend beliebig gewählt werden können. Deshalb kann angenommen werden, dass eine Vielzahl der in diesen Szenarien vorkommenden Aktionen, Ordnungen und Sequenzen für die jeweilige Anforderung von Bedeutung sind. Da hier bezüglich der syntaktischen Schnittstelle vollständige Szenarien angegeben werden, ist es zwar oft nicht möglich, ein Szenario anzugeben, welches ausschließlich Aktionen umfasst, die für die Anforderung von Bedeutung sind. Diese zusätzlichen Aktionen sollen dann aber minimal sein. Durch eine geeignet abstrakte Wahl der syntaktischen Schnittstelle lässt sich diese Problematik verringern. [5]

Bedeutung von Grenzwerttests In Kapitel 3 (Abschnitt 3.3.3, Seite 56) wurde zuvor schon erläutert, dass sich Testfälle an den Grenzen von Äquivalenzklassen besonders eignen, um Fehler zu entdecken. Für eine Anforderung, welche nur durch exemplarische Szenarien formal beschrieben ist, lässt sich hinsichtlich der Anforderung keine klare Grenze der Äquivalenzklasse definieren. Da ein Szenario r aber möglichst charakteristisch für eine Anforderung ρ gewählt wird, stellen Abläufe mit ähnlichen Eingaben zu r, welche aber an sich nicht von ρ erfasst werden, solche Testfälle an den Grenzen dar. Die Bedeutung von Grenzwerttests rechtfertigt somit wiederum das Vorgehen, einen Testfall einer Anforderung zuzuordnen, auch wenn dieser die Anforderung nicht prüft.

Die Ungenauigkeiten bei der Interpretation eines Anforderungsszenarios, mit welchen die Prädikate $\rho_{dir}[r]$, $\rho_{ord}[r]$, $\rho_{seq}[r]$ oder $\rho_{dirAll}[r]$ behaftet sind, werden aus diesen Gründen zumindest teilweise aufgehoben. Damit lässt sich anhand der Merkmale der *gemeinsamen Aktionen*, der *gleichen temporalen Ordnung von Aktionen*, der *Wiedergabe von Sequenzen* sowie der *Wiederholungen von Teilen eines Szenarien* die Überdeckung dieses Szenarios durch einen Testfall bewerten.

Eine entsprechende Metrikfunktion zur Bewertung der Überdeckung (Überdeckungsfunktion) eines Anforderungsszenarios durch einen Testfall

$$\mathrm{cov}_{1:1} \in REQ \times TC \to \mathbb{R}_0^+,$$

welche diese Kriterien hinsichtlich Anforderungsszenarien entsprechend Kapitel 5 operationalisiert, ist in Anhang B angegeben. Dazu wurden die notwendigen Eigenschaften einer Überdeckungsfunktion, welche sich aus den diesen Bewertungskriterien ergeben, zunächst präzise formalisiert (Abschnitt B.1.1) und dann die Metrikfunktion definiert (Abschnitt B.2.1).

[5] Siehe dazu auch Kapitel 5, Abschnitt 5.2.

7.2.4. Überdeckung einer Menge von Anforderungen durch eine Menge von Testfällen

Mit der Funktion $\text{cov}_{1:1}$ kann nur die Überdeckung *einer* Anforderung durch *einen* Testfall bewertet werden. In der Praxis ist aber die Überdeckung einer Menge von Anforderungen durch eine Menge von Testfällen (Testsuite) von größerem Interesse. Diese Überdeckung einer Menge von Anforderungen durch eine Menge von Testfällen kann einerseits anhand der Überdeckung von einer Anforderung durch eine Menge von Testfällen und andererseits anhand der Überdeckung von einer Menge von Anforderungen durch einen Testfall beurteilt werden. Kriterien dazu werden in den folgenden Abschnitten diskutiert.

Überdeckung einer Anforderung durch eine Menge von Testfällen

Wird die Überdeckung einer Anforderung durch eine Menge von Testfällen betrachtet, kann diese induktiv bewertet werden. Eine leere Menge von Testfällen überdeckt die Anforderung dabei nicht, für die einelementige Menge kann die Überdeckung nach den Kriterien im vorhergehenden Abschnitt bewertet werden. Werden weitere Testfälle zu einer Menge von Testfällen hinzugefügt, sind folgende Kriterien zu prüfen:

Vorhandenes Präfix Falls ein Testfall Präfix eines anderen Testfalls ist, trägt er nicht weiter zur Überdeckung bei, da der längere Testfall die selben Überprüfungen vornimmt.

Einzelne Überdeckung Ein zur Testfallmenge hinzu genommener Testfall kann nur dann die Überdeckung positiv beeinflussen, sofern dieser für sich genommen eine Überdeckung der Anforderung erreicht. Hier wird angenommen, dass eine höhere Anzahl von Testfällen die Überdeckung verbessert. Die Bewertung kann anhand der Kriterien im vorhergehenden Abschnitt erfolgen.

Im Anhang B.1.2 sind diese Eigenschaften für eine Metrikfunktion

$$\text{cov}_{1:\text{n}} \in REQ \times \mathbb{P}(TC) \to \mathbb{R}_0^+,$$

formalisiert, und in Anhang B.2.2 wird die Funktion $\text{cov}_{1:\text{n}}$ definiert.

Analog zur Überdeckung von einer Anforderung durch mehrere Testfälle stellt sich die Frage nach der Bewertung der Überdeckung von mehreren Anforderungen durch nur einen Testfall.

Überdeckung einer Menge von Anforderungen durch einen Testfall

Soll bewertet werden, wie gut ein Testfall geeignet ist, eine Menge von verschiedenen Anforderungen zu überprüfen, hängt dies zunächst auch von der Bedeutung der einzelnen Anforderungen ab. Es wird versucht werden, den Testfall vor allem hinsichtlich von wichtigen Anforderungen zu bewerten, weniger wichtige Details werden nur nachrangig beachtet werden. Da die Bedeutung beziehungsweise Priorität von einzelnen Anforderungen letztlich immer anwendungsspezifisch ist, ist es nur schwer möglich, in einer allgemeinen Bewertung unterschiedliche Prioritäten von Anforderungen zu berücksichtigen. Deshalb soll hier davon ausgegangen werden, dass alle Anforderungen von vergleichbarer Bedeutung für die Stakeholder des Systems sind. Um dennoch unterschiedlich priorisierte Anforderungen zu berücksichtigen, kann die Menge der Anforderungen in disjunkte Teilmengen zerlegt werden. Jede der Teilmengen umfasst dabei die Anforderungen einer bestimmten Priorität.

Unter der eben erläuterten Annahme, dass alle Anforderungen von gleicher Bedeutung für die Bewertung der Überdeckung eines Testfalls sind, kann die Überdeckung einer Menge von Anforderungen durch einen Testfall nach folgenden Kriterien untersucht werden:

Singuläre Überdeckung Einerseits ist natürlich von Bedeutung, wie die singuläre Überdeckung, also die Überdeckung jeder einzelnen Anforderung durch den Testfall, bewertet wird. Falls zu der Menge der Anforderungen eine weitere Anforderung hinzugenommen wird, kann dies die Überdeckung dann positiv beeinflussen, falls die neue Anforderung durch den Testfall ausreichend stark (überdurchschnittlich) überdeckt wird.

Anzahl der Anforderungen Falls die singuläre Überdeckung der einzelnen Anforderungen durch den Testfall jeweils gleich bewertet wird, soll die Mächtigkeit der Anforderungsmenge keinen Einfluss auf die Bewertung der Überdeckung der Anforderungsmenge haben. Dieses Kriterium ist notwendig, damit die Qualität der Überdeckung nicht allein durch Veränderung der Anzahl betrachteter Anforderungen beeinflusst werden kann. Insbesondere ist zu verhindern, dass die Hinzunahme von Anforderungen unmittelbar zu einer Erhöhung der Überdeckung führt. Die Überdeckung soll somit hinsichtlich der Anzahl der Anforderungen normiert sein.

Auch hier sind diese Eigenschaften wiederum im Anhang formal angegeben, siehe dazu Abschnitt B.1.3. Zusammen mit den zuvor genannten Eigenschaften für die Überdeckung einer Anforderung beziehungsweise einer Menge von Anforderungen durch einen Testfall kann eine Überdeckungsfunktion

$$\mathrm{cov}_{\mathrm{m:n}} \in \mathbb{P}(REQ) \times \mathbb{P}(TC) \to \mathbb{R}_0^+,$$

Tabelle 7.4.: Anforderungsüberdeckung in den ermittelten Testsuiten

T	$\text{cov}_{1:n}(r_i, T)$												$\text{cov}_{m:n}(R, T)$
	r_1	r_2	r_3	r_4'	r_4''	r_5	r_6	r_7	r_8	r_9'	r_9''	r_{10}	
TS_{REQ}	81,7	50,6	56,4	31,5	29,6	62,2	69,0	43,0	43,0	27,7	25,9	53,8	47,9
TS_{struct}	16,7	10,4	10,4	6,4	6,2	11,1	14,7	9,3	9,3	5,2	5,1	9,9	9,5
TS_{random}	37,2	24,7	26,8	13,8	14,1	25,1	33,9	23,0	23,4	13,8	13,7	22,9	22,7

definiert werden. Ein mögliche derartige Funktion ist im Anhang in Abschnitt B.2.3 angegeben, die zusammengefassten notwendigen Eigenschaften dieser Funktion sind in AbschnittB.1.4 formalisiert. Diese Metrikfunktion wird im Folgenden zur Bewertung der Überdeckung von Anforderungen durch Testfälle für die Fallstudie verwendet.

7.2.5. Anwendung auf die Fallstudie

Die Anforderungsüberdeckung der drei Testsuiten TS_{REQ}, TS_{struct} und TS_{random} wurde mit Hilfe der im Anhang B definierten Metrikfunktionen, welche die zuvor diskutierten Eigenschaften erfüllt, bewertet.

In Anhang G sind die ermittelten 1:1-Überdeckungen $\text{cov}_{1:1}(r, t)$ für alle Paare von Testfällen t und Anforderungen r aufgeführt. In Tabelle 7.4 sind die daraus resultierenden $1 : n$-Überdeckungen $\text{cov}_{1:n}(r, TS_x)$ angeführt, die jeweils die Überdeckung einer Anforderung durch die entsprechende Testsuite angeben. Wie zu sehen ist, erzielt die Testsuite TS_{REQ}, welche durch die anforderungsorientierte Testfallgenerierung gewonnen wurde, einen deutlich höhere Wert in der Überdeckung als die Testsuite TS_{struct}, gewonnen mittels struktureller Überdeckung im Gesamtmodell.

Die höhere Abdeckung durch TS_{REQ} ist auch auf den größeren Umfang der Testsuite zurückzuführen. Sie weist aber auch gegenüber der Testsuite TS_{random} einen höheren Überdeckungsgrad auf. Somit kann der bessere Überdeckungsgrad nicht allein durch die Größe der Testsuite begründet sein.

Eine weitere Analyse der Überdeckung ist in Tabelle 7.5 angeben. Hier ist die Anzahl der Testfälle in den einzelnen Testsuites angegeben, welche eine überdurchschnittliche 1:1-Überdeckung einer Anforderung im Vergleich zu den übrigen Testfällen erzielen. Hier sind Auswirkungen vergleichbar zu jenen in der Analyse aus Abschnitt 7.1.2 festzustellen. Für die höher gewichteten Anforderungen r_4', r_4'', r_9', r_9'' erzielt die anforderungsorientierte Testfallgenerierung eine größere Anzahl von Testfällen, welche eine überdurchschnittliche Überdeckung dieser Anforderungen aufweisen. Die Auswirkungen der Gewichtung von Anforderungsklassen zeigt sich somit auch in der Analyse der Anforderungsüberdeckung.

Wie schon die Analyse der Teilmodelle (Abschnitt 7.1.1) und der Vergleich der Testsuiten (Abschnitt 7.1.2), weist somit auch die Bewertung der Anforderungsüberdeckung

Tabelle 7.5.: Anzahl der Testfälle $tc \in TS_x$, bei welchen eine Anforderung r_i eine überdurchschnittliche 1 : 1-Anforderungsüberdeckung $\text{cov}_{1:1}(r, t)$ aufweist. Angeben ist auch der relative Anteil bezogen auf die Menge aller Testfälle der jeweiligen Testsuite.

	TS_{REQ}		TS_{struct}		TS_{random}	
Anzahl Testfälle	119		45		119	
mittl. $\text{cov}_{1:1}$	0,40		0,21		0,19	
	Testfälle t mit $\text{cov}_{1:1}(r_x, t) >$ mittl. $\text{cov}_{1:1}$					
	Anzahl	rel. Anteil	Anzahl	rel. Anteil	Anzahl	rel. Anteil
r_1	72	60,5 %	26	57,8 %	60	50,4 %
r_2	54	45,4 %	23	51,1 %	75	63,0 %
r_3	54	45,4 %	23	51,1 %	78	65,5 %
r_4'	18	15,1 %	4	8,9 %	16	13,4 %
r_4''	17	14,3 %	5	11,1 %	7	5,9 %
r_5	58	48,7 %	26	57,8 %	60	50,4 %
r_6	55	46,2 %	26	57,8 %	57	47,9 %
r_7	39	32,7 %	14	31,1 %	69	58,0 %
r_8	39	32,7 %	14	31,1 %	74	62,2 %
r_9'	17	14,3 %	1	2,2 %	15	12,6 %
r_9''	17	14,3 %	1	2,2 %	9	7,6 %
r_{10}	42	35,3 %	26	57,8 %	57	47,9 %

für das in dieser Arbeit beschriebene Verfahren eine stärkere Orientierung der Testfälle an den Anforderungen aus, als dies bei der Verwendung von strukturellen oder randomisierten der Fall ist. Während bei beiden letzteren Verfahren einige Anforderungen (insbesondere r_4', r_4'', r_9'' sowie r_9'') nur wenig im Test berücksichtigt wurden, wird durch die anforderungsorientierte Testfallgenerierung erreicht, dass auch diese Anforderungen sich verstärkt in der Testsuite widerspiegeln.

7.3. Zusammenfassende Bewertung

Zweck der in dieser Arbeit vorgestellten anforderungsorientierten modellbasierten Testfallgenerierung ist es, implizit intendierte, aber nicht explizit formalisierte Anforderungen in den generierten Testfällen stärker zu adressieren, als dies bisher mit Techniken zur modellbasierten Testfallgenerierung möglich ist. Insbesondere soll auch eine Gewichtung der Anforderungen möglich sein, damit eine Steuerung der Testfallgenerierung hinsichtlich der Anforderungen möglich ist.

Ziel der Methode ist es somit, die Qualität der Testsuiten in Relation zu Inhalten von Anforderungsdokumenten positiv zu beeinflussen. Es sollen somit nicht beliebige Abläufe des Testmodells als Tests gewählt werden, stattdessen sollen die Testfälle vor allem die durch Szenarien exemplarisch formalisierten einzelnen Anforderungen

adressieren. Wie auch für alle anderen konstruktiv einsetzbaren Gütemaße von Testfällen kann damit keine direkte Aussage getroffen werden, ob besonders fehlerbehaftete Teile des zu testenden Systems vorrangig im Test berücksichtigt werden. Bei den spezifizierten Szenarien handelt es sich um exemplarische Repräsentanten von Äquivalenzklassen, welche durch die intendierten Anforderungen jeweils gebildet werden. Intuitiv wird damit oftmals ein linearer interpolierter Abschluss verbunden: Geringfügige Änderungen an den Eingaben eines Szenarios führen auch nur zu geringfügigen Änderungen der Ausgabe. Eine derartige Variante eines Szenarios bleibt dann in der Äquivalenzklasse der Anforderung. Testfälle, welche in diesem Sinn nur Variationen in den Eingaben aufweisen, sind somit besser geeignet, die Adressierung der explizit formulierten Anforderungen durch den Test zu belegen. Auch in Fällen, in welchen diese lineare Extrapolation nicht haltbar ist, werden sinnvolle Testfälle erzielt: Eine geringfügige Variation von Eingaben, welche zu einem vollkommen veränderten Ausgabeverhalten führt, stellt einerseits Grenzfälle zu einer Anforderung dar, andererseits sind diese Szenarien hilfreich, um Anforderungen zu validieren.

Es ist auch davon auszugehen, dass die explizit dokumentierten Szenarien und Variationen davon der intendierten (typischen) Nutzung des Systems stärker entsprechen, als beliebige Abläufe. Mit der größeren Anzahl von tatsächlichen Ausführungen solcher Abläufe geht eine höhere Auftrittswahrscheinlichkeit von potentiellem Fehlverhalten einher. Auch aus diesem Sinn ist es gerechtfertigt, dass Testfälle sich an den explizit formulierten Anforderungsszenarien orientieren.

Dass die vorgestellte Methode der Testfallgenerierung dazu führt, dass sich Testfälle stärker an den Anforderungsszenarien orientieren, wird durch mehrere Beobachtungen belegt: Zunächst zeigt bereits die Analyse der gewonnenen Teilmodelle, dass die höher gewichteten Anforderungen stärkere Berücksichtigung finden. Die Transitionen, welche von diesen Szenarien ausgeführt werden, kommen überproportional häufig in den Teilmodellen vor. Weiter wird durch die Analyse der Testfälle deutlich, dass diese Transitionen auch vermehrt in den Testfällen ausgeführt werden, welche aus den Teilmodellen gewonnen wurden. Im Gegensatz dazu werden diese Transitionen, welche für die Ausführung der Anforderungen charakteristisch sind, bei Testsuiten auf Basis von rein strukturellen oder stochastischen Testfallspezifikationen weniger häufig durchlaufen. Schließlich bestätigt dies auch die quantifizierte Bewertung der Anforderungsüberdeckung; auch hier zeigt sich dass die Testfälle, welche durch das vorgestellte anforderungsorientierte Verfahren ermittelt wurden, stärker an die Anforderungsszenarien angelehnt sind. Dagegen ist bei den mit Hilfe struktureller und stochastischer Testfallspezifikationen gewonnen Testsuiten der Bezug zu den Anforderungsszenarien weit weniger ausgeprägt und fehlt vor allem weitgehend hinsichtlich einiger Anforderungen, für welche im Test eine starke Gewichtung vorgesehen war.

Die Bewertung der Anforderungsüberdeckung auf Basis von Szenarien nimmt dabei an, dass die Szenarien exemplarische Repräsentanten von Anforderungen sind und damit Anforderungen hinsichtlich des Auftretens von Eingabeaktionen, von der

Reihenfolge dieser Aktionen und der Sequenzen von Aktionen beschreiben. Um die Anforderungsüberdeckung auf Basis von Szenarien zu bewerten, ist es notwendig, die exemplarische Darstellung der Anforderung zu verallgemeinern. Dazu wurde angenommen, dass ein Szenario Anforderungen hinsichtlich des Auftretens, der Reihenfolge sowie der Sequenzen von Eingabeaktionen beschreibt. Damit können Bedingungen identifiziert werden, unter denen ein Testfall diese verallgemeinerten Anforderungen prüft und somit zu einer Überdeckung des Szenarios beiträgt. Dies ist Grundlage für eine Metrik zur Bewertung der Überdeckung von Szenarien durch Testfälle, wie sie etwa in Anhang B definiert ist. Die angewendete Anforderungsüberdeckung bewertet somit die Überdeckung der verallgemeinerten Anforderungen $\rho_{dir}[r]$, $\rho_{ord}[r]$, $\rho_{seq}[r]$ sowie $\rho_{dirAll}[r]$ für ein Szenario r.

Natürlich kann durch dedizierte funktionale Testfallspezifikationen, welche für jede einzelne Anforderung definiert werden, eine intuitiv stärkere Überdeckung der Anforderungen durch die Testfälle erreicht werden. Dies hat jedoch zum einen den Nachteil, dass ein wesentlich höherer manueller Aufwand zur Formulierung dieser funktionalen Testfallspezifikationen notwendig ist. Im Gegensatz dazu ist im vorgeschlagenen Verfahren neben der Formulierung der Anforderungsszenarien nur deren Klassifikation und Gewichtung notwendig. Zum anderen können manuell erstellte Testfallspezifikationen mitunter den Freiheitsgrad bei der Wahl der Testfälle zu stark einschränken. Die generierten Testfällen fokussieren dann zu spezifisch auf die explizit formulierten einzelnen Anforderungen – implizite Anforderungen, welche nur in der Vervollständigung des Modells berücksichtigt wurden, werden gegebenenfalls weniger berücksichtigt, wenn nicht alle Modellteile bei der Testfallgenerierung mit einbezogen werden.

8. Zusammenfassung und Ausblick

8.1. Ergebnisse

Diese Arbeit stellt einen Methode vor, welche Techniken der modellbasierten Testfallgenerierung nutzt, um anforderungsspezifische Testfälle zu ermitteln. Der wesentliche Zweck dieser anforderungsspezifischen Testfälle ist dabei, gegenüber allen Stakeholdern des zu testenden Systems – insbesondere Auftraggeber und Anwender – den Test der formulierten funktionalen Einzelanforderungen nachzuweisen.

Die wesentlichen Beiträge dieser Arbeit sind dabei:

- Automatisierbare, funktionale Testfallauswahl
- Integration von Anforderungsbeschreibung und Testfallerstellung
- Nutzung von Intention und Erfahrung des Testingenieurs

Zunächst ermöglicht die beschriebene Methode eine weitgehende **Automatisierung der Testfallermittlung** mittels Testfallgenerierung aus abstrakten Verhaltensmodellen. Wurden bisher bei der modellbasierten Testfallgenerierung vor allem strukturelle Überdeckungen als Auswahlkriterium (Testfallspezifikation) verwendet, orientiert sich die in dieser Arbeit vorgestellte Technik zur Testfallgenerierung vor allem an den zu testenden einzelnen funktionalen Anforderungen. In diesem Zusammenhang kommt der Verwendung von spezifizierten Einzelanforderungen *und* einem Testmodell, welches das abstrakte Soll-Verhalten total beschreibt, eine entscheidende Bedeutung zu. Während einerseits die einzelne Anforderung dazu dient, dass die ermittelten Testfälle nicht allein willkürliche Abfolgen von Eingaben adressieren, welche häufig nicht der erwarteten Verwendung des Systems entsprechen, wird andererseits durch die Testfallgenerierung im Verhaltensmodell sichergestellt, dass sich die Testfälle nicht allein auf die als einzelne Anforderungen spezifizierten Abläufe beschränken.

In der Praxis ist häufig davon auszugehen, dass die Menge der dokumentierten Einzelanforderungen nicht so vollständig und detailliert ist, dass damit bereits das gesamte Verhalten des Systems für alle möglichen Folgen von Eingaben im Sinne einer totalen Funktion spezifiziert ist. Daher führen bisher bekannte Methoden zur direkten Ableitung von Testfällen aus einzelnen Anforderungen im Allgemeinen dazu, dass große Teile des Systemverhaltens im Test unberücksichtigt bleiben. Mithilfe eines Verhaltensmodells, welches hinsichtlich seiner abstrakten Schnittstelle das Systemverhalten total beschreibt, wird die Spezifikation vervollständigt. Aus dem Modell können

somit auch Testfälle gewonnen werden, welche durch die als Anforderungen gegebenen Abläufe nicht dokumentiert sind. Da aber allgemein angenommen werden kann, dass die dokumentierten Anforderungen für die Stakeholder des Systems, insbesondere für Nutzer und Auftraggeber, von besonderer Bedeutung sind, ist eine Orientierung der Testfälle an den ursprünglichen Einzelanforderungen meist gewünscht. Insbesondere fordern Auftraggeber von Softwareprodukten häufig den Nachweis, dass jede der formulierten einzelnen Anforderungen durch eigene Testfälle überprüft wurde.

Um diesen beiden, sich im Grunde widersprechenden, Forderungen nach Adressierung der einzelnen Anforderungen einerseits und der Berücksichtigung von nicht explizit dokumentierten Systemabläufen andererseits, im Test zu genügen, werden in der vorgestellten Methode anforderungsspezifische Teilmodelle gebildet aus welchen mittels modellbasierter Testfallgenerierung Testfälle ermittelt werden. In den Teilmodellen kommen dabei strukturelle Überdeckungskriterien zur Testfallspezifikation zum Einsatz. Das beschriebene Verfahren stellt somit eine Testfallgenerierung dar, bei welcher die Testfallspezifikation eine Kombination aus funktionalen und strukturellen Elementen ist.

Die Ermittlung von Testfällen steht in einem engen Zusammenhang zur Spezifikation von Anforderungen. Daher ist ein weiterer wesentlicher Beitrag dieser Arbeit, den **Testprozess in Zusammenhang mit der Beschreibung von Anforderungen** darzustellen. Diese Dissertation schlägt dazu die Spezifikation von exemplarischen Szenarien zu jeder (informell) geforderten funktionalen Anforderung vor. Szenarien bieten den Vorteil, dass diese verhältnismäßig einfach durch die Stakeholder hinsichtlich ihrer Gültigkeit validiert werden können. Szenarien können dabei auch als Beispiel für Testfälle verstanden werden. Somit ist es notwendig, dass ein Testmodell diese Szenarien als Abläufe akzeptieren muss. Dies führt zu der Notwendigkeit, die abstrakte Schnittstelle so zu beschreiben, dass diese sowohl für die Formulierung von Anforderungen als auch von Testfällen geeignet ist. In dem in der Arbeit geschilderten Vorgehen reicht dafür die Spezifikation der abstrakten syntaktischen Schnittstelle mit ihren Ein-/Ausgabekanäle und deren Typisierung aus. Damit wird eine klare Definition der Systemgrenzen erreicht, welche für den Test essentiell ist, damit das Testobjekt klar festgelegt ist. Nach Auffassung des Verfassers trägt die explizite Berücksichtigung des Systemtests bereits in der Anforderungserhebungsphase nicht nur zur Qualität der Testfälle sondern auch zur Qualität der Anforderungen bei. Dies gilt sowohl in inhaltlicher Hinsicht (die Korrektheit der Anforderung wird validiert und der Grad der Vollständigkeit der Anforderungen deutlich gemacht) als auch in organisatorischer Hinsicht (die Überprüfbarkeit der Anforderung ist gewährleistet). Im Bezug auf die modellbasierte Testfallgenerierung ist es wichtig, die Beziehung zwischen Testmodell und geforderten Einzelanforderungen zu berücksichtigten, wie es auch im Rahmen des vorgestellten Verfahrens der Fall ist.

Schließlich resultiert die beschriebene Methode in einer weitgehenden Automatisierung der Testfallermittlung, bietet aber gleichzeitig die Möglichkeit, dass die **Testinge-**

nieure ihre Erfahrung und Intention in die Generierung von Testfällen einfließen lassen können. Anders als eine randomisierte Testfallermittlung oder bei der Verwendung rein struktureller Testfallspezifikationen ist die Testfallgenerierung steuerbar. Dazu werden Anforderungen klassifiziert und die Anforderungsklassen unterschiedlich gewichtet. Der Testingenieur kann somit im Test einen Schwerpunkt auf Anforderungen legen, welche nach seiner Ansicht besondere Berücksichtigung finden sollen. Gleichwohl hat der Testingenieur keine vollständige Kontrolle über die generierte Testsuite. Durch die Verwendung struktureller Testfallspezifikationen in den Teilmodellen wird sichergestellt, dass alle Teile des Modells in die Bildung der Testfälle einfließen. Das Nutzen der Intention der Testingenieure erscheint vor dem Hintergrund sinnvoll, als dass kein, für die Testfallermittlung operationalisierbares, Qualitätskriterium existiert, welches sicherstellt, dass Testfälle ausreichend dazu geeignet sind, Fehler aufzudecken. Dies gilt insbesondere, da dabei auch Auftrittswahrscheinlichkeit und Folgekosten von Fehlern berücksichtigt werden müssen.

Weitere Beiträge leistet diese Arbeit durch die **Strukturierung von Qualitätsbewertungen** für Testfälle und die Diskussion zur **Bewertung der Überdeckung von Anforderungen**.

Die vorgestellte Methode wurde anhand eines vereinfachten Beispiels aus dem Automotive Software Engineering erarbeitet, der Ansatz ist dabei aber auf reale Systeme übertragbar. Zur **Validierung** des Ansatzes wurden einerseits die ermittelten anforderungsspezifischen Teilmodelle analysiert, sowie die resultierenden Testfälle mit Testfällen aus rein struktureller Überdeckung und zufälliger Testfallauswahl verglichen. Dabei zeigte sich, dass Modell- und Verhaltensanteile, welche durch höher gewichtete Anforderungen motiviert waren, verstärkt in den Teilmodellen und Testfällen vertreten waren. Für die Bewertung der Anforderungsüberdeckung wurde ein Überdeckungsmaß definiert. Die Bewertung hinsichtlich dieses Überdeckungsmaßes bestätigt die Analyse der Teilmodelle und Testfälle und zeigt eine höhere Überdeckung der Anforderungen.

8.2. Zukünftige Arbeiten

Die vorliegende Arbeit beschreibt eine mögliche Methode zur modellbasierten Testfallgenerierung hinsichtlich einzelner Anforderung. Gleichwohl existieren eine Reihe weiterführender Themen, ich welchen auch in Zukunft noch Forschungsbedarf besteht. Im Besonderen sind hierbei zu nennen:

Qualitätsbewertung von Testfällen Wie in Kapitel 3 dargestellt, sind die heute verwendeten Gütemaße für Testfälle noch unbefriedigend. Eine Aussagekraft hinsichtlich der Fähigkeit, Fehler zu entdecken, ist vor allem deswegen fraglich, da oftmals die Verteilung von Fehlern im Testobjekt vernachlässigt wird. Insofern erscheint es zwar schwierig, hier eine abschließende Lösung zu finden, gleichwohl sind differenziertere

Bewertungsmodelle als die aktuell verwendeten Überdeckungskriterien denkbar. Es existieren zwar Qualitätsmodelle (Wagner und Deissenboeck, 2007; Deissenboeck u. a., 2007) für Software, vergleichbare Ansätze zur Bewertung von Testfällen sind dagegen kaum vorhanden.

Anforderungsüberdeckung In der Arbeit wurden bereits Kriterien zur Messung der Überdeckung von Anforderungen durch Testfälle genannt und eine Metrik zu einer derartigen Bewertung hinsichtlich Anforderungsszenarien angewendet. Werden Anforderungen dagegen nicht als Szenarien, sondern als partielle Funktionen beziehungsweise Dienste (Broy und Stølen, 2001; Meisinger und Rittmann, 2008; Rittmann, 2008) formuliert, können differenzierte Überdeckungen betrachtet werden. Schwierigkeiten stellen dabei die Beziehung der formalen Anforderungen zu informalen Anforderungen, sowie die Validierung der Anforderungen durch die Stakeholder dar. Ohne letzteres hat eine Anforderungsüberdeckung für die Stakholder oft nur wenig Aussagekraft.

Mächtigere Notationen zur Anforderungsspezifikation Die vorgestellte Methode basiert auf der Formulierung von Anforderungen durch Szenarien. Obwohl die Methode auch bei anderen Formen der Anforderungsbeschreibung verwendbar ist (indem Szenarien als Instanzen zu den Anforderungen angegeben werden), können mächtigere Notationen zur Anforderungsspezifikation gegebenenfalls dazu genutzt werden, die Teilmodellbildung und Testfallgenerierung gezielter durchzuführen. Beispielsweise indem Anforderungen oder Teile davon um Informationen zu Kritikalität, Fehlerwahrscheinlichkeit oder Nutzungshäufigkeit ergänzt werden. Auch durch die im vorhergehenden Absatz erwähnte Spezifikation von Diensten beziehungsweise partiellen Funktionen könnte die Teilmodellbildung gezielter gesteuert werden.

Testtreiber Eine im Rahmen der modellbasierten Testfallgenerierung weitgehend vernachlässigte Fragestellung ist bisher die Formulierung von Testtreibern. Sofern die Abstraktion zwischen Testmodell und Testobjekt nicht trivial ist, genügen einfache Datentransformationen zur Übersetzung der abstrakten Testfälle in konkrete Testfälle nicht. Heute wird davon ausgegangen, dass Treiberkomponenten spezifisch für jedes Testobjekt, aber mindestens spezifisch für jede Domäne, erstellt werden. Ansätze zur Spezifikation von Systemen auf verschiedenen Abstraktionsebenen wie sie zum Beispiel in (Wild u. a., 2006; Broy u. a., 2008) beschrieben sind, können möglicherweise dazu verwendet werden, Treiberkomponenten automatisch zu erstellen. Voraussetzung dafür ist, dass die Relation zwischen abstrakten und konkreten Ebenen präzise definiert ist. Gleichzeitig muss die Konkretisierung aber zusätzliche Informationen erfordern, welche nicht in den abstrakten Ebenen enthalten sind. Ein Test ist in diesem Zusammenhang interessant, wenn durch diese Hinzunahme von Informationen das Verhalten auf der konkreten Ebene von demjenigen abweichen kann, welches auf der abstrakten Ebene

spezifiziert ist. Dieses wären Fehler, welche durch einen Testfall, der aus dem Modell der abstrakten Ebene generiert wurde, entdeckt werden könnten.

Literaturverzeichnis

[Adams 1984] Adams, Edward N.: Optimizing Preventive Service of Software Products. In: *IBM Journal of Research and Development* 28 (1984), Januar, Nr. 1

[AF2] Technische Universität München: AutoFOCUS 2. `http://www4.in.tum.de/~af2/`

[Apt und Wallace 2007] Apt, Krzysztof R. ; Wallace, Mark G.: *Constraint Logic Programming using ECLiPSe*. Cambridge, UK : Cambridge University Press, 2007

[Balzert 1998] Balzert, Helmut: *Lehrbuch der Softwaretechnik, Teil 2: Softwaremanagement, Software-Qualitaetssicherung, Unternehmensmodellierung*. Heidelberg, Berlin : Spektrum Akademischer Verlag, 1998. – 503–530 S

[Beizer 1990] Beizer, Boris: *Software testing techniques (2nd ed.)*. New York, NY, USA : Van Nostrand Reinhold Co., 1990

[Beizer 1995] Beizer, Boris: *Black-box testing: techniques for functional testing of software and systems*. New York, NY, USA : John Wiley & Sons, Inc., 1995

[Binder 1999] Binder, Robert V.: *Testing Object-Oriented Systems. Models, Paterns and Tools*. Addisoin Wesley, 1999

[Bock 2008] Bock, Thomas: *Vehicle in the Loop – Test- und Simulationsumgebung für Fahrerassistenzsysteme*, Technische Universität München, Dissertation, November 2008

[Boehm 2003] Boehm, Barry: Value-based software engineering: reinventing. In: *SIGSOFT Softw. Eng. Notes* 28 (2003), Nr. 2, S. 3

[Bollig u. a. 2007] Bollig, Benedikt ; Katoen, Joost-Pieter ; Kern, Carsten ; Leucker, Martin: Replaying Play In and Play Out: Synthesis of Design Models from Scenarios by Learning. In: Grumberg, Orna (Hrsg.) ; Huth, Michael (Hrsg.): *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg : Springer-Verlag, 2007 (Lecture Notes in Computer Science 4424), S. 435–450

[Broy 2005] Broy, Manfred: A semantic and methodological essence of message sequence charts. In: *Science of Computer Programming* 54 (2005), Februar, Nr. 2-3, S. 213–256

[Broy u. a. 1992] Broy, Manfred ; Dederich, Frank ; Dendorfer, Claus ; Fuchs, Max ; Gritzner, Thomas ; Weber, Rainer: The Design of Distributed Systems – An Introduction to FOCUS. Technische Universität München, Januar 1992 (TUM-I9202). – Technischer Bericht

[Broy u. a. 2008] Broy, Manfred ; Feilkas, Martin ; Grünbauer, Johannes ; Gruler, Alexander ; Harhurin, Alexander ; Hartmann, Judith ; Penzenstadler, Birgit ; Schätz, Bernhard ; Wild, Doris: Umfassendes Architekturmodell für das Engineering eingebetteter Software-intensiver Systeme / Technische Universität München. jun 2008 (TUM-I0816). – Technischer Bericht

[Broy u. a. 2005] Broy, Manfred (Hrsg.) ; Jonsson, Bengt (Hrsg.) ; Katoen, Joost-Pieter (Hrsg.) ; Leucker, Martin (Hrsg.) ; Pretschner, Alexander (Hrsg.): *Model-Based Testing of Reactive Systems, Advanced Lectures*. Bd. *3472*. Springer, 2005. (Lecture Notes in Computer Science)

[Broy und Stølen 2001] Broy, Manfred ; Stølen, Ketil ; Gries, David (Hrsg.) ; Schneider, Fred B. (Hrsg.): *Specification and Development of Interactive Systems. FOCUS on streams, interfaces and refinement*. New York, Berlin, Heidelberg : Springer, 2001 (Monographs in Computer Science)

[Cai und Lyu 2005] Cai, Xia ; Lyu, Michael R.: The effect of code coverage on fault detection under different testing profiles. In: *A-MOST '05: Proceedings of the 1st international workshop on Advances in model-based testing*. New York, NY, USA : ACM, 2005, S. 1–7

[Chillarege 1996] Chillarege, R.: *Orthogonal Defect Classification*. Kap. 9, S. 359–400. In: Lyu, Michael R. (Hrsg.): *Handbook of Software Reliability Engineering*. Los Alamitos, Calif. : McGraw-Hill Computing and IEEE Computer Society Press, 1996

[Chillarege u. a. 1992] Chillarege, R. ; Bhandari, I.S. ; Chaar, J.K. ; Halliday, M.J. ; Moebus, D.S. ; Ray, B.K. ; Wong, M.-Y.: Orthogonal defect classification-a concept for in-process measurements. In: *Software Engineering, IEEE Transactions on* 18 (1992), Nov, Nr. 11, S. 943–956

[Clarke Jr. u. a. 1999] Clarke Jr., Edmund M. ; Grumberg, Orna ; Peled, Doron A.: *Model Checking*. Cambridge, Massachusetts and London, England : The MIT Press, 1999

[Conrad 2004] Conrad, Mirko: *Auswahl und Beschreibung von Testszenarien für den Modell-basierten Test eingebetteter Software im Automobil*, Technische Universität Berlin, Dissertation, 2004

[Damas u. a. 2005] Damas, Christophe ; Lambeau, Bernard ; Dupont, Pierre ; Lamsweerde, Axel van: Generating Annotated Behavior Models from End-User Scenarios. In: *IEEE Transactions on Software Engineering* 31 (2005), Nr. 12, S. 1056–1073

[Deissenboeck u. a. 2007] Deissenboeck, Florian ; Wagner, Stefan ; Pizka, Markus ; Teuchert, Stefan ; Girard, Jean-Francois: An Activity-Based Quality Model for Maintainability. In: *Proceedings of the 23rd International Conference on Software Maintenance (ICSM 2007)*, IEEE CS Press, 2007

[DeMillo u. a. 1978] DeMillo, R.A. ; Lipton, R.J. ; Sayward, F.G.: Hints on Test Data Selection: Help for the Practicing Programmer. In: *Computer* 11 (1978), April, Nr. 4, S. 34–41

[Duraes und Madeira 2006] Duraes, J.A. ; Madeira, H.S.: Emulation of Software Faults: A Field Data Study and a Practical Approach. In: *Software Engineering, IEEE Transactions on* 32 (2006), Nov., Nr. 11, S. 849–867

[Esser und Struss 2006] Esser, Michael ; Struss, Peter: Model-based Test Generation for Embedded Software. In: *DX'06, 17th International Workshop on Principles of Diagnosis*. Penaranda de Duero, Burgos (Spain), Juni 2006

[Esser und Struss 2007] Esser, Michael ; Struss, Peter: Fault-Model-Based Test Generation for Embedded Software. In: (Veloso, 2007), S. 342–347

[ETAS] ETAS Inc: *Ascet*. Ann Arbor, MI (USA): . – URL `http://www.etas.com/de/products/ascet_software_products.php`

[EUROCAE 1992] European Organisation for Civil Aviation Equipment (EUROCAE): *ED-12B, Software Considerations in Airborne Systems and Equipment Certification*. Luzern: , 1992

[European Telecommunications Standards Institute (ETSI) 2005] European Telecommunications Standards Institute (ETSI): *ETSI ES 201 873: TTCN-3, Edition 3.1.1*. http://www.ttcn-3.org/Specifications.htm. Juni 2005. – URL `http://www.ttcn-3.org/Specifications.htm`

[Fenton und Neil 1999] Fenton, N.E. ; Neil, M.: A critique of software defect prediction models. In: *Software Engineering, IEEE Transactions on* 25 (1999), Sep/Oct, Nr. 5, S. 675–689

[Fleischmann 2008] Fleischmann, Andreas: *Modellbasierte Formalisierung von Anforderungen für eingebettete Systeme im Automotive-Bereich*, Technische Universität München, Dissertation, Mai 2008

[Fournier 2009] Fournier, Greg: *Essential Software Testing. A Use-Case Approach*. Boca Raton : Auerbach Publications, 2009 (CRC Press). – 280 S

[Frühwirth und Abdennadher 1997] Frühwirth, Thom ; Abdennadher, Slim: *Constraint-Programmierung*. Berlin, Heidelberg : Springer, 1997

[Gietelink u. a. 2006] Gietelink, Olaf ; Ploeg, Jeroen ; Schutter, Bart D. ; Verhaegen, Michel: Development of advanced driver assistance systems with vehicle hardware-in-the-loop simulations. In: *Vehicle System Dynamics* 44 (2006), Juli, Nr. 7, S. 569–590

[Glinz 2000] Glinz, Martin: Improving the Quality of Requirements with Scenarios. In: *Proceedings of the Second World Congress for Software Quality.* Yokohama, Japan, September 2000, S. 55–60

[Hamlet 1977] Hamlet, R.G.: Testing Programs with the Aid of a Compiler. In: *Software Engineering, IEEE Transactions on* SE-3 (1977), July, Nr. 4, S. 279–290

[Hammer u. a. 1997] Hammer, Theodore ; Rosenberg, Linda ; Huffman, Lenore ; Hyatt, Lawrence: Measuring requirements testing:: experience report. In: *ICSE '97: Proceedings of the 19th international conference on Software engineering.* New York, NY, USA : ACM Press, 1997, S. 372–379

[Hamon u. a. 2004] Hamon, G. ; Moura, L. de ; Rushby, J.: Generating efficient test sets with a model checker. In: *Software Engineering and Formal Methods, 2004. SEFM 2004. Proceedings of the Second International Conference on*, 2004, S. 261–270

[Hamon u. a. 2005] Hamon, Grégoire ; Moura, Leonardo de ; Rushby, John: Automated Test Generation with SAL / Computer Science Laboratory, SRI International. Menlo Park CA 94025 USA, Januar 2005. – CSL Technical Note

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

[Harel 1988] Harel, David: On visual formalisms. In: *Commun. ACM* 31 (1988), Nr. 5, S. 514–530

[Hartmann 2001] Hartmann, Nico: *Automation des Tests eingebetteter Systeme am Beispiel der Kraftfahrzeugelektronik*, Universität Karlsruhe, Dissertation, Januar 2001. – URL `http://digbib.ubka.uni-karlsruhe.de/volltexte/1642001`

[Hayhurst u. a. 2001] Hayhurst, Kelly J. ; Veerhusen, Dan S. ; Chilenski, John J. ; Rierson, Leanna K.: A Practical Tutorial on Modified Condition/Decision Coverage / National Aeronautics and Space Administration, Langley Research Center. Hampton, Mai 2001 (NASA/TM-2001-210876). – Technical Memorandum

[Heimdahl u. a. 2004] Heimdahl, M.P.E. ; Devaraj, G. ; Weber, R.: Specification test coverage adequacy criteria = specification test generation inadequacy criteria? In: *High Assurance Systems Engineering, 2004. Proceedings. Eighth IEEE International Symposium on*, 2004, S. 178–186

[Houdek und Paech 2002] Houdek, Frank ; Paech, Barbara: Das Türsteuergerät – eine Beispielspezifikation / Fraunhofer Institut Experimenteles Software Engineering (IESE). Kaiserslautern, Germany, Januar 2002 (002.02/D). – IESE-Report

[Huber u. a. 1997] Huber, Franz ; Schätz, Bernhard ; Einert, Geralf: Consistent Graphical Specification of Distributed Systems. In: Fitzgerald, John (Hrsg.) ; Jones, Cliff B. (Hrsg.) ; Lucas, Peter (Hrsg.): *FME '97: 4th International Symposium of Formal Methods Europe, Lecture Notes in Computer Science 1313*, Springer, 1997, S. 122 – 141

[IBM Corporation] IBM Corporation: *Telelogic Statemate*. Armonk, NY (USA): . – URL `http://modeling.telelogic.com/products/statemate/index.cfm`

[IEEE 610 1990] Institute of Electrical and Electronics Engineers, Inc. (IEEE): *IEEE Std 610.12-1990, IEEE standard glossary of software engineering terminology*. New York: , Dezember 1990. – – S. – URL `http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=159342&isnumber=4148`

[IEEE 829 2008] Institute of Electrical and Electronics Engineers, Inc. (IEEE): *IEEE Std 829-2008, IEEE Standard for Software and System Test Documentation*. New York: , Juli 2008. – 1–118 S. – URL `http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4578383&isnumber=4578382`

[International Telecommunication Union (ITU) 2004] International Telecommunication Union (ITU): *ITU-T Recommendation Z.120, Messege Sequence Chart (MSC)*. Geneva, Switzerland: , April 2004. – URL `http://www.itu.int/ITU-T/studygroups/com17/languages/`

[ISO 11898-1 2003] International Organization for Standardization (ISO): *ISO 11898-1:2003 Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling*, 2003

[ISO 9126 2001] International Organization for Standardization (ISO): *ISO/IEC 9126: Software engineering – Product quality*, 2001

[Jonsson 2004] Jonsson, Bengt: Finite State Machines. In: (Broy u. a., 2005), S. 611–614

[Karlsson 1996] Karlsson, J.: Software requirements prioritizing. In: *Requirements Engineering, 1996., Proceedings of the Second International Conference on* (1996), Apr, S. 110–116

[Karlsson und Ryan 1997] Karlsson, J. ; Ryan, K.: A cost-value approach for prioritizing requirements. In: *Software, IEEE* 14 (1997), Sep/Oct, Nr. 5, S. 67–74

[Kof 2005] Kof, Leonid: *Text Analysis for Requirements Engineering*. München, Technische Universität München, Dissertation, November 2005. – URL `http://www4.in.tum.de/publ/papers/Leonid_Kof_Text_Analysis_Diss.pdf`

[Kof 2007a] Kof, Leonid: Scenarios: Identifying Missing Objects and Actions by Means of Computational Linguistics. In: *15th IEEE International Requirements Engineering Conference*. New Delhi, India : IEEE Computer Society Conference Publishing Services, October 15–19 2007, S. 121 – 130

[Kof 2007b] Kof, Leonid: *Text Analysis for Requirements Engineering, Application of Computational Linguistics*. Saarbrücken, Germany : VDM Verlag Dr. Müller, 2007

[Kof 2008] Kof, Leonid: From Textual Scenarios to Message Sequence Charts: Inclusion of Condition Generation and Actor Extraction. In: *16th IEEE International Requirements Engineering Conference*. Barcelona, Spain : IEEE Computer Society Conference Publishing Services, September 10-12 2008, S. 331–332

[Krüger u. a. 1999] Krüger, Ingolf ; Grosu, Radu ; Scholz, Peter ; Broy, Manfred: From MSCs to statecharts. In: *DIPES '98: Proceedings of the IFIP WG10.3/WG10.5 international workshop on Distributed and parallel embedded systems*. Norwell, MA, USA : Kluwer Academic Publishers, 1999, S. 61–71

[Lamberg u. a. 2004] Lamberg, Klaus ; Beine, Michael ; Eschmann, Mario ; Otterbach, Rainer ; Conrad, Mirko ; Fey, Ines: Model-Based Testing Of Embedded Automotive Software Using Mtest. In: *Transaction of 2004 SAE World Congress*. Detroit, MI, USA., März 2004 (SAE Technical Paper 2004-01-1593)

[van Lamsweerde 2001] Lamsweerde, A. van: Goal-Oriented Requirements Engineering: A Guided Tour. In: *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on* (2001), S. 249–262

[Lee und Yannakakis 1994] Lee, D. ; Yannakakis, M.: Testing finite-state machines: state identification and verification. In: *Computers, IEEE Transactions on* 43 (1994), Nr. 3, S. 306–320

[Lee und Yannakakis 1996] Lee, D. ; Yannakakis, M.: Principles and methods of testing finite state machines-a survey. In: *Proceedings of the IEEE* 84 (1996), Nr. 8, S. 1090–1123

[Liggesmeyer 2002] Liggesmeyer, Peter: *Software-Qualität. Testen, Analysieren und Verifizieren von Software*. Heidelberg, Berlin : Spektrum Akademischer Verlag, 2002

[Link 2005] Link, Johannes: *Softwaretests mit JUnit*. 2. Heidelberg : dpunkt.verlag, Januar 2005. – 416 S

[Lötzbeyer und Pretschner 2000] Lötzbeyer, H. ; Pretschner, A.: AutoFocus on Constraint Logic Programming. In: *Proc. (Constraint) Logic Programming and Software Engineering (LPSE'2000), London*, 2000

[Lötzbeyer 2003] Lötzbeyer, Heiko: *Modellbasierte Testfallermittlung für eingebettete Systeme in sicherheitskritischen Anwendungen*, Technische Universität München, Fakultät für Informatik, Dissertation, 2003

[Ma u. a. 2006] Ma, Yu-Seung ; Offutt, Jeff ; Kwon, Yong-Rae: MuJava: a mutation system for java. In: *ICSE '06: Proceedings of the 28th international conference on Software engineering*. New York, NY, USA : ACM, 2006, S. 827–830

[Malaiya und Denton 1999] Malaiya, Y.K. ; Denton, J.: Requirements volatility and defect density. In: *Software Reliability Engineering, 1999. Proceedings. 10th International Symposium on* (1999), S. 285–294

[Mealy 1955] Mealy, G. H.: A method for Synthesizing Sequential Circuits. In: *Bell System Technical Journal* 34 (1955), September, Nr. 5, S. 1045 – 1079

[Meisinger und Rittmann 2008] Meisinger, Michael ; Rittmann, Sabine: A comparison of service-oriented development approaches / Technische Universität München. August 2008 (TUM-I0825). – Technischer Bericht

[Musa 1993] Musa, J.D.: Operational profiles in software-reliability engineering. In: *Software, IEEE* 10 (1993), Mar, Nr. 2, S. 14–32

[Musa und Ackerman 1989] Musa, John D. ; Ackerman, A. F.: Quantifying Software Validation: When to Stop Testing? In: *IEEE Softw.* 6 (1989), Nr. 3, S. 19–27

[Myers 1979] Myers, Glenford J.: *The Art of Software Testing*. John Wiley & Sons, 1979

[Myers 2001] Myers, Glenford J.: *Methodisches Testen von Programmen*. 7. Auflage. München : Oldenbourg, 2001

[Nagappan u. a. 2006] Nagappan, Nachiappan ; Ball, Thomas ; Zeller, Andreas: Mining metrics to predict component failures. In: *ICSE '06: Proceedings of the 28th international conference on Software engineering*. New York, NY, USA : ACM, 2006, S. 452–461

[Nagappan u. a. 2008] Nagappan, Nachiappan ; Murphy, Brendan ; Basili, Victor: The influence of organizational structure on software quality: an empirical case study. In: *ICSE '08: Proceedings of the 30th international conference on Software engineering*. New York, NY, USA : ACM, 2008, S. 521–530

[Object Managment Group, Inc. (OMG) 2005] Object Managment Group, Inc. (OMG): *UML Superstructure Specification, v2.0*. formal/05-07-04. Needham: , Juli 2005. – URL `http://www.omg.org/cgi-bin/doc?formal/05-07-04`

[OMG 2005] Object Managment Group, Inc. (OMG): *UML Testing Profile, Version 1.0*. http://www.omg.org/technology/documents/formal/test_profile.htm. Juli 2005. – URL `http://www.omg.org/technology/documents/formal/test_profile.htm`

[Pecheur u. a. 2009] Pecheur, Charles ; Raimondi, Franco ; Brat, Guillaume: A Formal Analysis of Requirements-Based Testing. In: *ISSTA '09: Proceedings of the 2009 ACM SIGSOFT International symposium on Software teting and analysis*. Chigaco, Illinois, USA, Juli 2009

[Pfaller 2008] Pfaller, Christian: Requirements-based test case specification by using information from model construction. In: *AST '08: Proceedings of the 3rd international workshop on Automation of software test*. New York, NY, USA : ACM, 2008, S. 7–16

[Pfaller u. a. 2006] Pfaller, Christian ; Fleischmann, Andreas ; Hartmann, Judith ; Rappl, Martin ; Rittmann, Sabine ; Wild, Doris: On the integration of design and test: a model-based approach for embedded systems. In: *AST '06: Proceedings of the 2006 international workshop on Automation of software test*. New York, NY, USA : ACM Press, 2006, S. 15–21

[Pfaller und Pister 2008] Pfaller, Christian ; Pister, Markus: Combining Structural and Functional Test Case Generation. In: Herrman, Korbinian (Hrsg.) ; Brueg-ge, Bernd (Hrsg.): *Software Engineering 2008. Fachtagung des GI-Fachbereichs Softwaretechnik 18.–22.02.2008 in München*. Bonn : Gesselschaft für Informatik e.V., Februar 2008 (Lecture Notes in Informatics 121), S. 229–241

[Pfaller u. a. 2008] Pfaller, Christian ; Wagner, Stefan ; Gericke, Jörg ; ; Wiemann, Matthias: Multi-Dimensional Measures for Test Case Quality. In: *Software Testing Verification and Validation Workshop, 2008. ICSTW '08. IEEE International Conference on* (2008), April, S. 364–368

[Pister 2009] Pister, Markus: *Integration formaler Fehlereinflussanalyse in die Funktionsentwicklung bei der Automobilindustrie*. München, Technische Universität München, Dissertation, 2009. – URL `http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20080611-653131-1-3`

[Pohl 2007] Pohl, Klaus: *Requirements Engineering. Grundlagen, Prinzipien, Techniken*. 1. Auflage. Heidelberg : dpunkt-Verl., 2007. – 741 S

[Prenninger und Pretschner 2005] Prenninger, Wolfgang ; Pretschner, Alexander: Abstractions for Model-Based Testing. In: *Electronic Notes in Theoretical Computer Science* 116 (2005), S. 59 – 71. – Proceedings of the International Workshop on Test and Analysis of Component Based Systems (TACoS 2004)

[Pretschner 2003] Pretschner, Alexander: *Zum modellbasierten funktionalen Test reaktiver Systeme*, Technische Universität München, Fakultät für Informatik, Dissertation, 2003. – URL `http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2003/pretschner.pdf`

[Pretschner und Leucker 2005] Pretschner, Alexander ; Leucker, Martin: Model-Based Testing – A Glossary. In: (Broy u. a., 2005), Kap. 20, S. 607–609

[Pretschner und Lötzbeyer 2001] Pretschner, Alexander ; Lötzbeyer, Heiko: Model Based Testing with Constraint Logic Programming: First Results and Challenges. In: *Proc. 2nd ICSE Intl. Workshop on Automated Program Analysis, Testing and Verification (WAPATV'01), Toronto*, 2001

[Pretschner und Philipps 2005] Pretschner, Alexander ; Philipps, Jan: Methodological Issues in Model-Based Testing. In: (Broy u. a., 2005), Kap. 10, S. 281–291

[Pretschner u. a. 2005] Pretschner, Alexander ; Prenninger, Wolfgang ; Wagner, Stefan ; Kühnel, Christian ; Baumgartner, Martin ; Sostawa, B. ; Zölch, R. ; Stauner, T.: One evaluation of model-based testing and its automation. In: *ICSE '05: Proceedings of the 27th international conference on Software engineering*, 2005, S. 392–401

[Prowell u. a. 1999] Prowell, Stacy J. ; Trammell, Carmen J. ; Linger, Richard C. ; Poore, Jesse H.: *Cleanroom Software Engineering. Technology and Process*. Amsterdam : Addison-Wesley Longman, April 1999 (The SEI Series in Software Engineering). – 416 S

[Rajan 2006] Rajan, Ajitha: Coverage Metrics to Measure Adequacy of Black-Box Test Suites. In: *Automated Software Engineering, 2006. ASE '06. 21st IEEE/ACM International Conference on*, 2006, S. 335–338

[Rajan u. a. 2008] Rajan, Ajitha ; Whalen, Michael W. ; Heimdahl, Mats P.: The effect of program and model structure on mc/dc test adequacy coverage. In: *ICSE '08: Proceedings of the 30th international conference on Software engineering*. New York, NY, USA : ACM, 2008, S. 161–170

[Rayadurgam und Heimdahl 2001] Rayadurgam, S. ; Heimdahl, M.P.E.: Coverage based test-case generation using model checkers. In: *Engineering of Computer Based Systems, 2001. ECBS 2001. Proceedings. Eighth Annual IEEE International Conference and Workshop on the*, 2001, S. 83–91

[Rayadurgam und Heimdahl 2003] Rayadurgam, S. ; Heimdahl, M.P.E.: Generating MC/DC adequate test sequences through model checking. In: *Software Engineering Workshop, 2003. Proceedings. 28th Annual NASA Goddard*, 2003, S. 91–96

[Reid 1997] Reid, S.C.: An empirical analysis of equivalence partitioning, boundary value analysis and random testing. In: *Software Metrics Symposium, 1997. Proceedings., Fourth International*, Nov 1997, S. 64–73

[Rittmann 2008] Rittmann, Sabine: *A methodology for modeling usage behavior of multi-functional systems.* München, Technische Universität München, Dissertation, 2008. – URL `http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20071211-635723-1-7`

[RTCA 1992] RTCA: *DO-178B, Software Considerations in Airborne Systems and Equipment Certification.* Washington D. C.: RTCA, Inc. (Veranst.), 1992

[Schätz und Huber 1999] Schätz, Bernhard ; Huber, Franz: Integrating Formal Description Techniques. In: Wing, Jeannette M. (Hrsg.) ; Woodcock, Jim (Hrsg.) ; Davies, Jim (Hrsg.): *FM'99 – Formal Methods, Proceedings of the World Congress on Formal Methods in the Development of Computing Systems* Bd. 1709, Springer Verlag, sep 1999, S. 1206 – 1225

[Schätz u. a. 2002] Schätz, Bernhard ; Pretschner, Alexander ; Huber, Franz ; Philipps, Jan: Model-Based Development of Embedded Systems / Technische Universität München, 2002 (TUM-I0204). – Technischer Bericht

[Sneed u. a. 2009] Sneed, Harry M. ; Baumgartner, Manfred ; Seidl, Richard: *Der Systemtest. Von den Anforderungen zum Qualitätsnachweis.* 2., aktualisierte und erweiterete Auflage. München : Carl Hanser Verlag, 2009

[Spillner und Linz 2007] Spillner, Andreas ; Linz, Tilo: *Basiswissen Softwaretest.* 3. Heidelberg : dpunkt, 2007. – 276 S

[Srikanth 2005] Srikanth, Hema: *Value-Driven System Level Test Case Prioritization.* Raleigh, NC, North Carolina State University, PhD Thesis, Dezember 2005. – URL `http://www.lib.ncsu.edu/theses/available/etd-12052005-000331/`

[Srikanth und Williams 2005] Srikanth, Hema ; Williams, Laurie: On the economics of requirements-based test case prioritization. In: *EDSER '05: Proceedings of the seventh international workshop on Economics-driven software engineering research.* New York, NY, USA : ACM, 2005, S. 1–3

[Struss 1994] Struss, Peter: Testing Physical Systems. In: *AAAI 1994: Proceedings of the 12th National Conference on Artificial Intelligence* Bd. 1. Seattle, USA : AAAI Press, 1994, S. 251–256

[Struss 2007] Struss, Peter: Model-Based Optimization of Testing through Reduction of Stimuli. In: (Veloso, 2007), S. 593–598

[Sun 1996] Sun Microsystems, Inc.: *Java.* `http://java.sun.com`. 1996

[Tai und Lei 2002] Tai, Kuo-Chung ; Lei, Yu: A test generation strategy for pairwise testing. In: *Software Engineering, IEEE Transactions on* 28 (2002), Jan, Nr. 1, S. 109–111

[The MathWorks, Inc.] The MathWorks, Inc.: *Stateflow*. Natick, MA (USA): . – URL `http://www.mathworks.de/products/stateflow/`

[Uchitel u. a. 2003] Uchitel, S. ; Kramer, J. ; Magee, J.: Synthesis of behavioral models from scenarios. In: *Software Engineering, IEEE Transactions on* 29 (2003), Nr. 2, S. 99–115

[Utting u. a. 2006] Utting, M. ; Pretschner, A. ; Legeard, B.: A taxonomy of model-based testing / Department of Computer Science, The Universiy of Waikato (New Zealand), 2006 (04/2006). – Technischer Bericht

[Utting und Legeard 2006] Utting, Mark ; Legeard, Bruno: *Practical Model-Based Testing: A Tools Approach*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2006

[Veloso 2007] Veloso, Manuela M. (Hrsg.): *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*. 2007

[Wagner und Deissenboeck 2007] Wagner, Stefan ; Deissenboeck, Florian: An Integrated Approach to Quality Modelling. In: *Proc. 5th Workshop on Software Quality (5-WoSQ)*, IEEE Computer Society Press, 2007

[Whalen u. a. 2006] Whalen, Michael W. ; Rajan, Ajitha ; Heimdahl, Mats P. ; Miller, Steven P.: Coverage metrics for requirements-based testing. In: *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*. New York, NY, USA : ACM, 2006, S. 25–36

[Whittaker und Thomason 1994] Whittaker, J.A. ; Thomason, M.G.: A Markov chain model for statistical software testing. In: *Software Engineering, IEEE Transactions on* 20 (1994), Oct, Nr. 10, S. 812–824

[Wild u. a. 2006] Wild, Doris ; Fleischmann, Andreas ; Hartmann, Judith ; Pfaller, Christian ; Rappl, Martin ; Rittmann, Sabine: An Architecture-Centric Approach towards the Construction of Dependable Automotive Software. In: *Proceedings of the SAE 2006 World Congress*, 2006

A. CLP-Programm zur Bestimmung anforderungsspezifischer Teilmodelle

Im Folgenden ist der Quellcode des CLP-Programms angegeben, mit welchen die anforderungsspezifischen Teilmodelle berechnet wurden. Verwendet wurde dazu ECL^iPS^e CLP[1], Version 6.0. Eine Einführung zu ECL^iPS^e findet sich in (Apt und Wallace, 2007).

Die Mealy-Maschine des Gesamtmodells – hier das Fallbeispiel des Fensterhebers – wird durch das Prädikat `transition/5`, die Szenarien durch das Prädikat `req/4` formuliert. Ein Teilmodell wird mithilfe des Prädikats `subMod/4` berechnet. Ein Aufruf lautet beispielsweise wie folgt:

```
:- submod(sIdle, r4a, [(init,1),(new,2),(null,2)], TL).
```

Die Variable `TL` enthält dann die Transitionen, welche im Teilmodell zur Anforderung r_4' bei einer Gewichtung von

$$\begin{aligned} \text{weight}(\textit{initial}) &= 1 \\ \text{weight}(\textit{added}) &= 2 \\ n &= 2 \end{aligned}$$

enthalten sind.[2]

Im übrigen sei an dieser Stelle auf die als Kommentare im Programmcode angegebene technische Dokumentation verwiesen.

Listing A.1: CLP-Quellcode zur Bestimmung der Teilmodelle

```
%% ********************************************************************
%% PREDICATES FOR ENCODING WINDOWCONTROL EXAMPLE
%% ********************************************************************

%% --- transiton/5 ------------------------------------------------------
%%
%% transition(?Id, ?SrcState, ?Input, ?Output, ?DestState)
%%
```

[1] `http://www.eclipse-clp.org/`

[2] Aufgrund der CLP Syntax wurden die Bezeichner für die Anforderungen r_4', r_4'', r_9', r_9'' im Programmcode durch `r4a`, `r4b`, `r9a` und `r9b` ersetzt.

```
%% Codes the model of the system as Mealy machine. Every clause
%% specifies a transition in the model
%%
%% Id        - an identifier (name) of the transition
%% SrcState  - required state of system before execution
%% Input     - required input to fire
%% Output    - produced output
%% DestState - state reached after execution
%%
transition(t01, sIdle,  (open, middle),   (down, no_msg), sDown).
transition(t02, sIdle,  (open, top),      (down, no_msg), sDown).
transition(t03, sIdle,  (open, bottom),   (stop, no_msg), sIdle).
transition(t04, sIdle,  (open, moving),   (stop, err),    sIdle).
transition(t05, sIdle,  (close, middle),  (up, no_msg),   sUp).
transition(t06, sIdle,  (close, top),     (stop, no_msg), sIdle).
transition(t07, sIdle,  (close, bottom),  (up, no_msg),   sUp).
transition(t08, sIdle,  (close, moving),  (stop, err),    sIdle).
transition(t09, sIdle,  (no_msg, middle), (stop, no_msg), sIdle).
transition(t10, sIdle,  (no_msg, top),    (stop, no_msg), sIdle).
transition(t11, sIdle,  (no_msg, bottom), (stop, no_msg), sIdle).
transition(t12, sIdle,  (no_msg, moving), (stop, err),    sIdle).
transition(t13, sDown,  (open, middle),   (stop, err),    sIdle).
transition(t14, sDown,  (open, top),      (down, no_msg), sDown).
transition(t15, sDown,  (open, bottom),   (stop, no_msg), sIdle).
transition(t16, sDown,  (open, moving),   (down, no_msg), sDown).
transition(t17, sDown,  (close, middle),  (stop, no_msg), sPause).
transition(t18, sDown,  (close, top),     (stop, no_msg), sPause).
transition(t19, sDown,  (close, bottom),  (stop, no_msg), sPause).
transition(t20, sDown,  (close, moving),  (stop, no_msg), sPause).
transition(t21, sDown,  (no_msg, middle), (stop, err),    sIdle).
transition(t22, sDown,  (no_msg, top),    (stop, no_msg), sIdle).
transition(t23, sDown,  (no_msg, bottom), (stop, no_msg), sIdle).
transition(t24, sDown,  (no_msg, moving), (stop, no_msg), sIdle).
transition(t25, sUp,    (open, middle),   (stop, no_msg), sPause).
transition(t26, sUp,    (open, top),      (stop, no_msg), sPause).
transition(t27, sUp,    (open, bottom),   (stop, no_msg), sPause).
transition(t28, sUp,    (open, moving),   (stop, no_msg), sPause).
transition(t29, sUp,    (close, middle),  (stop, err),    sIdle).
transition(t30, sUp,    (close, top),     (stop, no_msg), sIdle).
transition(t31, sUp,    (close, bottom),  (up, no_msg),   sUp).
transition(t32, sUp,    (close, moving),  (up, no_msg),   sUp).
transition(t33, sUp,    (no_msg, middle), (stop, err),    sIdle).
transition(t34, sUp,    (no_msg, top),    (stop, no_msg), sIdle).
transition(t35, sUp,    (no_msg, bottom), (stop, no_msg), sIdle).
transition(t36, sUp,    (no_msg, moving), (stop, no_msg), sIdle).
transition(t37, sPause, (open, middle),   (stop, no_msg), sPause).
transition(t38, sPause, (open, top),      (stop, no_msg), sPause).
transition(t39, sPause, (open, bottom),   (stop, no_msg), sPause).
transition(t40, sPause, (open, moving),   (stop, err),    sPause).
transition(t41, sPause, (close, middle),  (stop, no_msg), sPause).
transition(t42, sPause, (close, top),     (stop, no_msg), sPause).
transition(t43, sPause, (close, bottom),  (stop, no_msg), sPause).
```

```
transition(t44, sPause, (close, moving), (stop, err),    sPause).
transition(t45, sPause, (no_msg, middle), (stop, no_msg), sIdle).
transition(t46, sPause, (no_msg, top),    (stop, no_msg), sIdle).
transition(t47, sPause, (no_msg, bottom), (stop, no_msg), sIdle).
transition(t48, sPause, (no_msg, moving), (stop, err),    sIdle).
%% ------------------------------------------------------------------

%% --- req/4 ------------------------------------------------------------
%%
%% req(?Id, ?PreCond, ?Inputs, ?Class)
%%
%% Codes the scenarios (requiremens). Every clause indicates
%% an scenario.
%%
%% Id        - the identifier (name) of the scenario
%% PreCond   - the pre-condition of the scenario, refers to an other
%%             scenario Id or 'no' if the scenario has no pre-conditon
%% Inputs    - list of input actions in the scenario
%% Class     - the requirements class of this scneario
%%
req(r1,  no, [(open,middle),(open,moving)],                    init).
req(r2,  r1, [(no_msg,moving)],                                init).
req(r3,  r1, [(open,bottom)],                                  init).
req(r4a, r1, [(close,moving),(close,middle),(close,middle)],  new).
req(r4b, r1, [(close,moving),(no_msg,middle),(close,middle)], new).
req(r5,  r1, [(open,middle)],                                  init).
req(r6,  no, [(close,middle),(close,moving)],                  init).
req(r7,  r6, [(no_msg,moving)],                                init).
req(r8,  r6, [(close,top)],                                    init).
req(r9a, r6, [(open,moving),(open,middle),(open,middle)],      new).
req(r9b, r6, [(open,moving),(no_msg,middle),(open,middle)],    new).
req(r10, r6, [(close,middle)],                                 init).
%% ------------------------------------------------------------------


%% ******************************************************************
%% ENTRY POINT FOR SUBMODEL CALCULATION
%% ******************************************************************

%% --- subMod/4 ---------------------------------------------------------
%%
%% subMod(+S0, +Req, +Weight, -Transitions)
%%
%% Calculates the transitions in a specific submodel. The full model
%% must be given by transition/5 predicate, requirements must be
%% stated by req/4 predicate.
%%
%% S0          - the initial state
%% Req         - the identifier of the requiremnts scenario for which
%%               the submodel shall be calculated
%% Weight      - a list of weights for the requirements classes. for
```

```
%%                each class a pair (ClassId, Weight) should be part in
%%                the list as well as the pair (null, Weight) for
%%                unclassified transitions
%% Transitions - list holding the transitons Ids of the transitions
%%                in the submodel
%%
%% Example:
%%   :- submod(sIdle, r4a, [(init,1),(new,2),(null,2)], TL)
%%
subMod(S0,R,W,TL) :-
        transR(S0,R,RT,_),
        statesReached(RT,SL),
        allTransFromStateList(S0,SL,W,AllT),
        append(RT,AllT,AllTrans),
        removeDups(AllTrans,TL).
%% ------------------------------------------------------------------------



%% ************************************************************************
%% FURTHER PREDICATES USED BY subMod/4 (IN ALPHABETICAL ORDER)
%% ************************************************************************

%% --- allTrans/1 -------------------------------------------------------------
%%
%% allTrans(-Transitons)
%%
%% Transitons - list of transition ids
%%
allTrans(Trans) :- findall(T,transition(T,_,_,_,_),Trans).
%% ------------------------------------------------------------------------

%% --- allTransFromState/4 ----------------------------------------------------
%%
%% allTransFromState(+S0,+State,+Weight,-Transitions)
%%
%% Transitions is the list of ids of all transitions which are on
%% pathes from state State and within the limits of Weight.
%%
%% S0           - the overall initial state
%% State        - the respective state
%% Weight       - a list of weights for the requirements classes. for
%%                each class a pair (ClassId, Weight) should be part in
%%                the list as well as the pair (null, Weight) for
%%                unclassified transitions
%% Transitions - list of transiton identifieres
%%
allTransFromState(S0,S,W,AllT) :-
        findall(TL,pathFromState(S0,S,W,TL),AllPath),
        makeOneList(AllPath,AllTrans),
        removeDups(AllTrans,AllT).
%% ------------------------------------------------------------------------
```

```
%% --- allTransFromStateList/4 ----------------------------------------
%%
%% allTransFromState(+S0,+States,+Weight,-Transitions)
%%
%% Transitions is the list of ids of all transitions which are on
%% pathes from every state in States and within the limits of Weight.
%%
%% S0           - the overall initial state
%% States       - list of states
%% Weight       - a list of weights for the requirements classes. for
%%                each class a pair (ClassId, Weight) should be part in
%%                the list as well as the pair (null, Weight) for
%%                unclassified transitions
%% Transitions - list of transiton identifieres
%%
allTransFromStateList(_,[],_,[]).
allTransFromStateList(S0,[S|Ss],W,AllT) :-
        allTransFromState(S0,S,W,STrans),
        allTransFromStateList(S0,Ss,W,SsTrans),
        append(STrans,SsTrans,AllTrans),
        removeDups(AllTrans,AllT).
%% --------------------------------------------------------------------

%% --- checkTrans/4 ---------------------------------------------------
%%
%% checkTrans(+S0,+Transitions,+Weight,?TransWithinWeight)
%%
%% TransWithinWeight are the transitons out of Transitions which
%% are in the limits of the given Weight
%%
%% S0                 - overall initial state
%% Transitions        - list of transition identifieres
%% Weight             - a list of weights for the requirements classes.
%%                      for each class a pair (ClassId, Weight) should
%%                      be part in the list as well as the pair
%%                      (null, Weight) for unclassified transitions
%% TransWithinWeight - list of transitions withn limits of Weight
%%
checkTrans(_,[],_,[]).
checkTrans(S0,[T|Ts],W,[T|Tns]) :-
        clTrans(S0,T,Cs),
        checkWeight(Cs,W),
        checkTrans(S0, Ts, W, Tns),!.
checkTrans(S0,[_|Ts],W,Tns) :-
        checkTrans(S0, Ts, W, Tns).
%% --------------------------------------------------------------------

%% --- checkWeight/2 --------------------------------------------------
%%
%% checkWeight(Classes, Weight)
%%
```

```
%% All classes in Classes have a positiv weight in Weight
%%
%% Classes            - list of requirements classes
%% Weight             - a list of weights for the requirements classes.
%%                      for each class a pair (ClassId, Weight) should
%%                      be part in the list as well as the pair
%%                      (null, Weight) for unclassified transitions
%%
checkWeight([C|_],Ws) :-
        member((C,W),Ws),
        W > 0, !.
checkWeight([_|Cs],Ws) :-
        checkWeight(Cs,Ws).
%% ------------------------------------------------------------------------

%% --- clTrans/3 ----------------------------------------------------------
%%
%% clTrans(+S0,+Trans,?Classes)
%%
%% Classes lists all requirements classes of the requirements
%% scenario which execute the given transitoin. If the transition
%% is not executetd by any scenario Classes = [null]
%%
%% S0         - overall inital state
%% Transition - the transition identifier
%% Classes    - list of requirements classes
%%
clTrans(S0,T,[null]) :- clTrans2(S0,T,[]).
clTrans(S0,T,[C|Cs]) :- clTrans2(S0,T,[C|Cs]).
%% ------------------------------------------------------------------------

%% --- clTrans2/3 ---------------------------------------------------------
%%
%% clTrans2(+S0,+Transition,?Classes)
%%
%% Classes lists all requirements classes of the requirements
%% scenario which execute the given transitoin. If the transition
%% is not executetd by any scenario Classes = []
%%
%% S0         - overall inital state
%% Transition - the transition identifier
%% Classes    - list of requirements classes
%%
clTrans2(S0,T,Cs) :-
        findall(C,(rTrans(S0,T,Rs),member((_,C),Rs)),Cs).
%% ------------------------------------------------------------------------

%% --- decWeight/3 --------------------------------------------------------
%%
%% decWeight(+Class, +Weight, ?WeightNew)
%%
%% In WeightNew the weight of Class is decreased by one from
```

```
%% the weight in Weight
%%
%% Class      -
%% Weight     -
%% WeightNew -
%%
decWeight(C,W,Wnew) :-
        member(null,C), decWeight2(C,W,Wnew), !.
decWeight(C,W,Wnew) :-
        decWeight2([null|C],W,Wnew).
%% ------------------------------------------------------------------------

%% --- decWeight2/3 -------------------------------------------------------
%%
%% decWeight2(+Class, +Weight, ?WeightNew)
%%
%% In WeightNew the weight of Class is decreased by one from
%% the weight in Weight
%%
%% Class      -
%% Weight     -
%% WeightNew -
%%
decWeight2(_,[],[]).
decWeight2(Cs,[(C,W)|Ws],[(C,Wn)|Wns]) :-
        member(C,Cs),
        Wn is W - 1,
        decWeight(Cs,Ws,Wns), !.
decWeight2(Cs,[(C,W)|Ws],[(C,W)|Wns]) :-
        decWeight(Cs,Ws,Wns).
%% ------------------------------------------------------------------------

%% --- execute/4 ----------------------------------------------------------
%%
%% execute(+StartState, +Inputs, -EndState, -Transitions)
%%
%% Executes a list of inputs
%%
%% StartState - the state before execution
%% Inputs     - list of input actions
%% EndState   - the state which is reached after execution
%% Transitons - list of transition identifiers which have been fired
%%              during execution
%%
execute(S, [], S, []).
execute(S, [In|Ins], D, [T|Trans]) :-
        transition(T,S,In,_,Snext),
        execute(Snext,Ins,D,Trans).
%% ------------------------------------------------------------------------

%% --- makeOneList/2 ------------------------------------------------------
%%
```

```
%% makeOneList(+ListOfLists,?SingleList)
%%
%% Helper predicate for concatenation of lists to one single list.
%%
%% ListOfLists - a list which elements are lists
%% SingleList  - list whic is a concatenation of all lists in
%%               ListOfLists
%%
makeOneList([],[]).
makeOneList([LE|SubListList], List) :-
        makeOneList(SubListList, NewList),
        append(LE, NewList, List).
%% ------------------------------------------------------------------

%% --- outTrans/2 ---------------------------------------------------
%%
%% outTrans(+State, -Transitions)
%%
%% Indicates Transitions going out from State
%%
%% State       - the respective state
%% Transitions - list of outgoing transition ids from State
%%
outTrans(S,TL) :- findall(T,transition(T,S,_,_,_),TL).
%% ------------------------------------------------------------------

%% --- outTrans/4 ---------------------------------------------------
%%
%% outTrans(+S0,+State,+Weights,-Transitions)
%%
%% Calculates all outgoing transtions from a state which are within
%% the limits of the given weight.
%%
%% S0          - the overall initial state
%% State       - the state of the outgoing transitions
%% Weight      - a list of weights for the requirements classes. for
%%               each class a pair (ClassId, Weight) should be part in
%%               the list as well as the pair (null, Weight) for
%%               unclassified transitions
%% Transitions - list of transitons
%%
outTrans(S0,S,W,TL) :-
        outTrans(S,AllTrans),
        checkTrans(S0,AllTrans,W,TL).
%% ------------------------------------------------------------------

%% --- pathFromState/4 ----------------------------------------------
%%
%% pathFromState(+S0,+State,+Weight,-Transitions)
%%
%% Indicates, if Transitions is a list of ids of consecutive
%% transistions which are a path from the given state wehre the whole
```

```
%% path is within the limitis of the given weights
%%
%% S0          - the overall initial state
%% State       - the state of the outgoing transitions
%% Weight      - a list of weights for the requirements classes. for
%%               each class a pair (ClassId, Weight) should be part in
%%               the list as well as the pair (null, Weight) for
%%               unclassified transitions
%% Transitions - list of transitons which are a consecutive path from
%%               State and within the limits of Weight
%%
pathFromState(S0,S,W,[T|Ts]) :-
        outTrans(S0,S,W,OutTrans),
        OutTrans = [_|_],
        !, member(T,OutTrans),
        clTrans(S0,T,Cs),
        decWeight(Cs,W,Wnew),
        transition(T,_,_,_,Snew),
        pathFromState(S0,Snew,Wnew,Ts).
pathFromState(S0,S,W,[]) :-
        outTrans(S0,S,W,[]).
%% ------------------------------------------------------------------------

%% --- removeDups/2 -------------------------------------------------------
%%
%% removeDups(+List,?ListWithoutDuplicates)
%%
%% Helper predicate, removes duplicate elements from alist
%%
%% List                - List which may hold duplicates of
%%                       elements
%% ListWithoutDuplicates - List with no duplicate elements
%%
removeDups([],[]).
removeDups([X|Xs],Y) :-
        member(X,Xs),
        removeDups(Xs,Y), !.
removeDups([X|Xs],[X|Y]) :-
        removeDups(Xs,Y).
%% ------------------------------------------------------------------------

%% --- rTrans/3 -----------------------------------------------------------
%%
%% rTrans(+S0, ?Trans, ?ReqsAndClasses)
%%
%% States all requirments together with its classes which fire
%% the transiton Trans (excluding pre-condition)
%%
%% S0             - the overall initial state
%% Trans          - id of respective transiton
%% ReqsAndClasses - list of pairs (R,C) of requirements ids R and
%%                  requirements class C
```

```
%%
rTrans(SInit, T, Rs) :-
        findall((R,C),(transR(SInit, R, Trans, C),member(T,Trans)),Rs).
%% ------------------------------------------------------------------------

%% --- stateAfterPre/3 ----------------------------------------------------
%%
%% stateAfterPre(+S0, +Req, -S)
%%
%% Gives the state which is reached after executing the pre-conditon
%% of an requirement scenario
%%
%% S0  - the overall initial state
%% Req - id of requirement scenario
%% S   - the state which is reachead after execution the pre-conditon
%%       of Req
%%
stateAfterPre(InitS,R,InitS) :-
        req(R, no, _, _).
stateAfterPre(InitS,R,S) :-
        req(R, RPre, _, _),
        stateAfterPre(InitS,RPre,Snext),
        req(RPre, _, Ins, _),
        execute(Snext, Ins,S, _).
%% ------------------------------------------------------------------------

%% --- statesReached/2 ----------------------------------------------------
%%
%% statesReached(+Transitions,-States)
%%
%% Indicates states which are reached by given transitions
%%
%% Transitons - list of transiton ids
%% States     - list of reached states by these transitions
%%
statesReached([],[]).
statesReached([T|Ts],[D|Ds]) :-
        transition(T,_,_,_,D),
        statesReached(Ts,Ds).
%% ------------------------------------------------------------------------

%% --- transR/4 -----------------------------------------------------------
%%
%% transR(+S0, +Req, -Transitons, ?Class)
%%
%% Gives the transitons which are executed by an requirement scenario,
%% excluding pre-condition
%%
%% S0         - the overall initial state
%% Req        - id of the requirment scenario
%% Transitons - list of transitons which are executedf by Req
%% Class      - class of requirement
```

```
%%
transR(SInit, R, Trans, C) :-
        req(R,_,Ins,C),
        stateAfterPre(SInit, R, S),
        execute(S, Ins, _, Trans).
%% ------------------------------------------------------------------------
```

B. Bewertungsmetrik der Anforderungsüberdeckung

In diesem Anhang wird einen Metrik zur relativen Bewertung der Anforderungsüberdeckung definiert, welche auf den in Kapitel 7, Abschnitt 7.2 diskutierten Merkmale

- gemeinsame Eingaben,
- Reihenfolge der Eingaben,
- ununterbrochene (Teil-)Sequenzen und
- Wiederholung von (Teil-)Sequenzen

zur Bewertung der Überdeckung von Anforderungsszenarien durch Testfälle basiert. In Abschnitt B.1 sind zunächst diese Merkmale als Eigenschaften einer Metrik formalisiert, in Abschnitt B.2 wird die in Kapitel 7 verwendete Metrik angegeben.

Wie bereits im Hauptteil der vorliegenden Arbeit bezeichnet

$$REQ \subsetneq I^* \times O^*$$

die Menge der Szenarien, welche exemplarisch Anforderungen repräsentieren. I^* bezeichnet dabei die Menge der endlichen Wörter über das Eingabealphabet I, O^* die endlichen Wörter über das Ausgabealphabet. Analog dazu bezeichnet

$$TC \subsetneq I^* \times O^*$$

die Menge der Testfälle.

Die im Folgenden verwendeten Operatoren entsprechen wiederum jenen in (Broy und Stølen, 2001, Kapitel 4) definierten Operatoren über Ströme, sie sind ebenso im Anhang H angeführt.

B.1. Notwendige Eigenschaften einer Metrikfunktion

Hilfsprädikate

Zur Vereinfachung der Definition der Überdeckungen werden folgende Hilfsprädikate verwendet:

$$\text{partOf} \in M^* \times M^* \to \mathbb{B}\text{ool}$$

mit

$$\text{partOf}(s, x) \overset{\text{def}}{\Leftrightarrow} \exists u, v \in M^* : x = u \frown s \frown v$$

welches angibt, dass eine Sequenz $s \in M^*$ eine Teilsequenz der Sequenz $x \in M^*$ ist, sowie

$$\text{disjAct} \in M^* \times M^* \to \mathbb{B}\text{ool}$$

mit

$$\text{disjAct}(x, y) \overset{\text{def}}{\Leftrightarrow} \mathsf{rng}.x \cap \mathsf{rng}.y = \{\}$$

welches ausgedrückt, dass die Mengen der Aktionen, welche in den Sequenzen $x, y \in M^*$ auftreten, disjunkt sind. Es gilt dementsprechend

$$\text{disjAct}(x, y) \Leftrightarrow \nexists m \in M : \text{partOf}(\langle m \rangle, x) \wedge \text{partOf}(\langle m \rangle, y).$$

B.1.1. 1:1-Anforderungsüberdeckung $\text{cov}_{1:1}$

Eine Funktion

$$\text{cov}_{1:1} \in REQ \times TC \to \mathbb{R}_0^+$$

eignet sich zur Bewertung der Überdeckung *eines* Anforderungsszenarios durch *einen* Testfall, falls sie die folgenden Bedingungen erfüllt:

Den Forderungen, dass **gemeinsame Eingaben** von Anforderungsszenario und Testfall sowie **Wiederholungen** von Eingaben des Anforderungsszenarios zu einer höheren Überdeckung führen, wird durch folgende Eigenschaften von $\text{cov}_{1:1}$ genügt:

Für ein Anforderungsszenario $r \in REQ$ und Testfälle $t, t' \in TC$ gilt

$$\forall s \in I^* \setminus \{\langle\rangle\} : \text{partOf}(s, \mathsf{in}.r) \wedge \mathsf{in}.t' = \mathsf{in}.t \frown s \implies \text{cov}_{1:1}(r, t) < \text{cov}_{1:1}(r, t') \quad \text{(B.1)}$$

die Überdeckung wird somit höher bewertet, falls ein Testfall um eine Eingabe s erweitert wird, welche Bestandteil der Eingabe von r ist. Im Gegensatz dazu gibt folgende Eigenschaft an, dass Aktionen, welche nicht in der Eingabe des Anforderungsszenarios auftreten, keinen Einfluss auf die Bewertung der Überdeckung haben:

$$\forall s \in I^* : \text{disjAct}(s, \mathsf{in}.r) \wedge \mathsf{in}.t' = \mathsf{in}.t \frown s \implies \text{cov}_{1:1}(r, t) = \text{cov}_{1:1}(r, t'). \quad \text{(B.2)}$$

Um möglichst **ununterbrochenen Teilsequenzen** von Eingaben des Anforderungsszenarios, welche auch im Testfall auftreten, höher zu bewerten wird weiter gefordert:

$$\begin{aligned} \forall s_1 \in I^*, s_2, x \in I^* \setminus \{\langle\rangle\} : \text{partOf}(s_1 \frown s_2, \mathsf{in}.r) \wedge \text{disjAct}(x, \mathsf{in}.r) \\ \wedge\ \mathsf{in}.t = s_1 \frown s_2 \wedge \mathsf{in}.t' = s_1 \frown x \frown s_2 \implies \text{cov}_{1:1}(r, t) > \text{cov}_{1:1}(r, t') \quad \text{(B.3)} \end{aligned}$$

Tabelle B.1.: Beispiele für die Eigenschaften zur Überdeckungsfunktion $\mathsf{cov}_{1:1} \in \mathit{REQ} \times \mathit{TC} \to \mathbb{R}_0^+$ für eine Anforderung r mit Eingabesequenzen $\mathsf{in}.r = \langle a, b, c, d \rangle \in \mathit{REQ}$ und Testfälle $t, t' \in I^*$. Für das Eingabealphabet I gilt $\{a, b, c, d, x, y, z\} \subseteq I$.

$\mathsf{in}.t$	$\mathsf{in}.t'$	**Eigenschaft**	
$\langle x, a, b \rangle$	$\langle x, a, b, c \rangle$	$\mathsf{cov}_{1:1}(r, t) < \mathsf{cov}_{1:1}(r, t')$	(B.1)
$\langle x, a, b \rangle$	$\langle x, a, b, y, z \rangle$	$\mathsf{cov}_{1:1}(r, t) = \mathsf{cov}_{1:1}(r, t')$	(B.2)
$\langle a, b, c \rangle$	$\langle a, b, x, y, c \rangle$	$\mathsf{cov}_{1:1}(r, t) > \mathsf{cov}_{1:1}(r, t')$	(B.3)
$\langle a, b, x, c, d \rangle$	$\langle c, d, x, a, b \rangle$	$\mathsf{cov}_{1:1}(r, t) > \mathsf{cov}_{1:1}(r, t')$	(B.4)
$\langle a, b, c, d \rangle$	–	$\mathsf{cov}_{1:1}(r, t) = 1$	(B.5)
$\langle x, y, z \rangle$	–	$\mathsf{cov}_{1:1}(r, t) = 0$	(B.6)

Weiter soll die **Reihenfolge der Eingaben** zu einer höheren Überdeckung frühen, falls diese in Anforderungsszenario und Testfall gleich ist. Dies wird durch die folgende Eigenschaft erreicht.

$$\begin{aligned} &\forall s_1, s_2 \in I^* \setminus \{\langle\rangle\} : \mathrm{partOf}(s_1 \frown s_2, \mathsf{in}.r) \wedge \neg\, \mathrm{partOf}(s_2 \frown s_1, \mathsf{in}.r) \\ &\quad \wedge \mathsf{in}.t = s_1 \frown x \frown s_2 \wedge \mathsf{in}.t' = s_2 \frown x \frown s_1 \implies \mathrm{cov}_{1:1}(r, t) > \mathrm{cov}_{1:1}(r, t') \end{aligned} \tag{B.4}$$

Abschließend sei die Überdeckungsfunktion $\mathrm{cov}_{1:1}$ wie folgt normiert:

$$\mathsf{in}.r = \mathsf{in}.t \implies \mathrm{cov}_{1:1}(r, t) \stackrel{\mathsf{def}}{=} 1 \tag{B.5}$$

$$\mathrm{disjAct}(\mathsf{in}.r, \mathsf{in}.t) \implies \mathrm{cov}_{1:1}(r, t) \stackrel{\mathsf{def}}{=} 0 \tag{B.6}$$

Ein Testfall, dessen Eingaben identisch zu der Eingabe des Anforderungsszenarios sind, wird mit dem einer Überdeckung von 1 bewertet, ein Testfall, welcher ausschließlich andere Eingaben als das Anforderungsszenario beschreibt, wird mit 0 bewertet. Zu beachten ist, dass 1 nicht der Maximalwert der Überdeckungsfunktion ist.

In Tabelle B.1 sind Beispieltestfälle dargestellt, welche die zuvor definierten Eigenschaften einer Überdeckungsfunktionen für Anforderungen verdeutlichen.

B.1.2. 1:n-Anforderungsüberdeckung $\mathrm{cov}_{1:n}$

Für die Bewertung der Überdeckung einer Anforderung durch eine Menge von Testfällen wird zusätzlich

- die Anzahl der Testfälle und
- die Redundanz durch Präfixe in den Testfällen

berücksichtigt. Eine Funktion

$$\mathrm{cov}_{1:n} \in \mathit{REQ} \times \mathbb{P}(\mathit{TC}) \to \mathbb{R}_0^+$$

eignet sich zur Bewertung der Überdeckung *eines* einzelnen Anforderungsszenarios durch eine *Menge* von Testfällen, falls für eine Anforderung $r \in REQ$, eine Menge von Testfällen $T \subsetneq I^*$ und Testfälle $t, t' \in I^*$ gilt:

$$T = \{\} \implies \mathrm{cov}_{1:\mathrm{n}}(r, T) = 0 \tag{B.7}$$

$$\mathrm{cov}_{1:1}(r, t) = \mathrm{cov}_{1:1}(r, t') \implies \mathrm{cov}_{1:\mathrm{n}}(r, \{t\}) = \mathrm{cov}_{1:\mathrm{n}}(r, \{t'\}) \tag{B.8}$$

$$\mathrm{cov}_{1:1}(r, t) < \mathrm{cov}_{1:1}(r, t') \implies \mathrm{cov}_{1:\mathrm{n}}(r, \{t\}) < \mathrm{cov}_{1:\mathrm{n}}(r, \{t'\}) \tag{B.9}$$

$$\mathrm{cov}_{1:1}(r, t') = 0 \implies \mathrm{cov}_{1:\mathrm{n}}(r, T) = \mathrm{cov}_{1:\mathrm{n}}(r, T \cup \{t'\}) \tag{B.10}$$

$$\exists t \in T : t' \sqsubseteq t \implies \mathrm{cov}_{1:\mathrm{n}}(r, T) = \mathrm{cov}_{1:\mathrm{n}}(r, T \cup \{t'\}) \tag{B.11}$$

$$\nexists t \in T : t' \sqsubseteq t \wedge \mathrm{cov}_{1:1}(r, t') > 0 \implies \mathrm{cov}_{1:\mathrm{n}}(r, T) < \mathrm{cov}_{1:\mathrm{n}}(r, T \cup \{t'\}) \tag{B.12}$$

Bedingung (B.7) definiert $\mathrm{cov}_{1:\mathrm{n}}(r, T) = 0$ falls die Menge T der Testfälle keine Elemente enthält. Für einelementige Mengen von Testfällen muss die Ordnung erhalten bleiben, welche durch 1:1-Anforderungsüberdeckung $\mathrm{cov}_{1:1}$ für die Testfälle gegeben ist (Eigenschaften (B.8) und (B.9)).

Eigenschaft (B.10) bringt zum Ausdruck, dass die Hinzunahme eines Testfalls t' zur Testfallmenge T den Wert Überdeckung $\mathrm{cov}_{1:\mathrm{n}}(r, T)$ nicht verändert, falls t' nichts zur Überdeckung von r beiträgt, also $\mathrm{cov}_{1:1}(r, t') = 0$ gilt.

Ebenso unerheblich für die Überdeckung durch eine Testfallmenge ist es, wenn zu der Menge T ein Testfall t' hinzugenommen wird und in T bereits ein Testfall t existiert, dessen Präfix t' ist, da dann t' einen vollkommen redundanten Testfall zu t darstellt, dazu wird Eigenschaft (B.11) hinzugenommen. Eigenschaft (B.11) wird bewusst nicht zu

$$\exists t \in T : (\mathsf{rng}.r)Ⓢt' \sqsubseteq (\mathsf{rng}.r)Ⓢt \implies \mathrm{cov}_{1:\mathrm{n}}(r, T) = \mathrm{cov}_{1:\mathrm{n}}(r, T \cup \{t'\})$$

verstärkt, da auch die Differenz des Testfalls in Eingaben, welche nicht im Anforderungsszenario vorkommen, für einen unterschiedlichen Testfall sorgt. Das Anforderungsszenario (beziehungsweise Teilsequenzen davon) wird dadurch in unterschiedlichen Kontexten betrachtet.

Eigenschaft (B.12) fordert schließlich, dass ein Testfall t', welcher nicht als Präfix eines anderen Testfalls bereits in T enthalten ist, dazu führt, die Anforderungsüberdeckung höher bewerten, falls t' zu T hinzugenommen wird und $\mathrm{cov}_{1:1}(r, t') > 0$ ist.

Für diese Eigenschaften, welche eine Funktion $\mathrm{cov}_{1:\mathrm{n}} \in REQ \times \mathbb{P}(I^*) \to \mathbb{R}_0^+$ erfüllen soll, finden sich Beispiele in Tabelle B.2.

Tabelle B.2.: Beispiele für die Eigenschaften zur Überdeckungsfunktion $\mathsf{cov}_{1:n} \in \mathit{REQ} \times \mathbb{P}(\mathit{I}^*) \to \mathbb{R}_0^+$ für eine Anforderung $r = \langle a, b, c, d\rangle \in \mathit{REQ}$ und Testfallmengen $T, T' \subsetneq \mathit{I}^*$. Für das Eingabealphabet I gilt wiederum $\{a, b, c, d, x, y, z\} \subseteq \mathit{I}$.

T	T′	Eigenschaft	
$\{\}$	–	$\mathsf{cov}_{1:n}(r, T) = 0$	(B.7)
$\{\langle x, a, b\rangle\}$	$\{\langle x, a, b, y, z\rangle\}$	$\mathsf{cov}_{1:n}(r, T) = \mathsf{cov}_{1:n}(r, T')$	(B.8)
$\{\langle x, a, b\rangle\}$	$\{\langle x, a, b, c\rangle\}$	$\mathsf{cov}_{1:n}(r, T) < \mathsf{cov}_{1:n}(r, T')$	(B.9)
$\{t_1, \ldots, t_n\}$	$\{t_1, \ldots, t_n, \langle x, y, z\rangle\}$	$\mathsf{cov}_{1:n}(r, T) = \mathsf{cov}_{1:n}(r, T')$	(B.10)
$\{\langle x, a, b, c\rangle, \langle c, d, a\rangle\}$	$\{\langle x, a, b, c\rangle, \langle c, d, a\rangle, \langle x, a, b\rangle\}$	$\mathsf{cov}_{1:n}(r, T) = \mathsf{cov}_{1:n}(r, T')$	(B.11)
$\{\langle x, a, b, c\rangle, \langle c, d, a\rangle\}$	$\{\langle x, a, b, c\rangle, \langle c, d, a\rangle, \langle a, b, c\rangle\}$	$\mathsf{cov}_{1:n}(r, T) < \mathsf{cov}_{1:n}(r, T')$	(B.12)

B.1.3. m:1-Anforderungsüberdeckung $\mathrm{cov}_{\mathrm{m:1}}$

Für die Bewertung der Überdeckung einer Menge von Anforderungen durch einen Testfall wird weiter die

- Anzahl der betrachtetern Anforderungen

berücksichtigt. Eine Funktion

$$\mathrm{cov}_{\mathrm{m:1}} \in \mathbb{P}(\mathit{REQ})\backslash\{\} \times \mathit{TC} \to \mathbb{R}_0^+$$

eignet sich zur Bewertung der Überdeckung einer *Menge* von Anforderung durch *einen* Testfall, falls für eine Menge $R \subseteq \mathit{REQ}, R \neq \{\}$ von Anforderungsszenarien mit $r, r' \in \mathit{REQ}$ und einem Testfall $t \in \mathit{TC}$ gilt:

$$\mathrm{cov}_{1:1}(r,t) = \mathrm{cov}_{1:1}(r',t) \implies \mathrm{cov}_{\mathrm{m:1}}(\{r\},t) = \mathrm{cov}_{\mathrm{m:1}}(\{r'\},t) \tag{B.13}$$

$$\mathrm{cov}_{1:1}(r,t) < \mathrm{cov}_{1:1}(r',t) \implies \mathrm{cov}_{\mathrm{m:1}}(\{r\},t) < \mathrm{cov}_{\mathrm{m:1}}(\{r'\},t) \tag{B.14}$$

$$\mathrm{cov}_{1:1}(r',t) = 0 \wedge \mathrm{cov}_{\mathrm{m:1}}(R,t) = 0 \implies \mathrm{cov}_{\mathrm{m:1}}(R \cup \{r'\},t) = 0 \tag{B.15}$$

$$\frac{\sum_{r\in R} \mathrm{cov}_{1:1}(r,t)}{|R|} < \mathrm{cov}_{1:1}(r',t) \implies \mathrm{cov}_{\mathrm{m:1}}(R,t) < \mathrm{cov}_{\mathrm{m:1}}(R \cup \{r'\},t) \tag{B.16}$$

$$\frac{\sum_{r\in R} \mathrm{cov}_{1:1}(r,t)}{|R|} > \mathrm{cov}_{1:1}(r',t) \implies \mathrm{cov}_{\mathrm{m:1}}(R,t) > \mathrm{cov}_{\mathrm{m:1}}(R \cup \{r'\},t) \tag{B.17}$$

Die Eigenschaften (B.13) und (B.14) stellen sicher, dass für einelementige Mengen von Anforderungen die durch $\mathrm{cov}_{1:1}(r,t)$ definierte Ordnung erhalten bleibt. Eigenschaft (B.15) definiert den trivialen Fall für $\mathrm{cov}_{\mathrm{m:1}}(R \cup \{r'\},t)$, falls $\mathrm{cov}_{\mathrm{m:1}}(R,t) = 0$ und $\mathrm{cov}_{1:1}(r',t) = 0$.

Für den allgemeinen Fall, dass $\mathrm{cov}_{1:1}(r',t) \geq 0$ gilt, wird mit (B.16) und (B.17) gefordert, dass die Hinzunahme einer weiteren Anforderung r' den Überdeckungsgrad $\mathrm{cov}_{\mathrm{m:1}}(R,t)$ der Anforderungsmenge erhöht oder verringert, abhängig davon, ob die Überdeckung $\mathrm{cov}_{1:1}(r',t)$ über oder unter dem Mittelwert der Überdeckungen der einzelnen Anforderungen in R liegt.

B.1.4. m:n-Anforderungsüberdeckung $\mathrm{cov}_{\mathrm{m:n}}$

Mit Hilfe der zuvor geforderten Eigenschaften für eine $1:n$ und $m:1$ Anforderungsüberdeckung lassen sich unmittelbar die Eigenschaften herleiten.

Eine Funktion

$$\mathrm{cov}_{\mathrm{m:n}} \in \mathbb{P}(REQ)\backslash\{\} \times \mathbb{P}(TC) \to \mathbb{R}_0^+$$

eignet sich zur Bewertung der Überdeckung einer *Menge von Anforderungen* durch eine *Menge von Testfällen*, falls für $T \subsetneq I^*, R \subsetneq REQ, t, t' \in I^*, r, r' \in REQ$ gilt:

$$T = \{\} \implies \mathrm{cov}_{\mathrm{m:n}}(R, T) = 0 \tag{B.18}$$

$$\mathrm{cov}_{\mathrm{m:1}}(R, t) = \mathrm{cov}_{\mathrm{m:1}}(R, t') \implies \mathrm{cov}_{\mathrm{m:n}}(R, \{t\}) = \mathrm{cov}_{\mathrm{m:n}}(R, \{t'\}) \tag{B.19}$$

$$\mathrm{cov}_{\mathrm{m:1}}(R, t) < \mathrm{cov}_{\mathrm{m:1}}(R, t') \implies \mathrm{cov}_{\mathrm{m:n}}(R, \{t\}) < \mathrm{cov}_{\mathrm{m:n}}(R, \{t'\}) \tag{B.20}$$

$$\mathrm{cov}_{\mathrm{m:1}}(R, t') = 0 \implies \mathrm{cov}_{\mathrm{m:n}}(R, T) = \mathrm{cov}_{\mathrm{m:n}}(R, T \cup \{t'\}) \tag{B.21}$$

$$\exists t \in T : t' \sqsubseteq t \implies \mathrm{cov}_{\mathrm{m:n}}(R, T) = \mathrm{cov}_{\mathrm{m:n}}(R, T \cup \{t'\}) \tag{B.22}$$

$$\nexists t \in T : t' \sqsubseteq t \wedge \mathrm{cov}_{\mathrm{m:1}}(R, t') > 0 \implies \mathrm{cov}_{\mathrm{m:n}}(R, T) < \mathrm{cov}_{\mathrm{m:n}}(R, T \cup \{t'\}) \tag{B.23}$$

$$\mathrm{cov}_{\mathrm{1:n}}(r, T) = \mathrm{cov}_{\mathrm{1:n}}(r', T) \implies \mathrm{cov}_{\mathrm{m:n}}(\{r\}, T) = \mathrm{cov}_{\mathrm{m:n}}(\{r'\}, T) \tag{B.24}$$

$$\mathrm{cov}_{\mathrm{1:n}}(r, T) < \mathrm{cov}_{\mathrm{1:n}}(r', T) \implies \mathrm{cov}_{\mathrm{m:n}}(\{r\}, T) < \mathrm{cov}_{\mathrm{m:n}}(\{r'\}, T) \tag{B.25}$$

$$\mathrm{cov}_{\mathrm{1:n}}(r', T) = 0 \ \wedge \mathrm{cov}_{\mathrm{m:n}}(R, T) = 0 \implies \mathrm{cov}_{\mathrm{m:n}}(R \cup \{r'\}, T) = 0 \tag{B.26}$$

$$\frac{\sum_{r \in R} \mathrm{cov}_{\mathrm{1:n}}(r, T)}{|R|} < \mathrm{cov}_{\mathrm{1:n}}(r', T) \implies \mathrm{cov}_{\mathrm{m:n}}(R, T) < \mathrm{cov}_{\mathrm{m:n}}(R \cup \{r'\}, T) \tag{B.27}$$

$$\frac{\sum_{r \in R} \mathrm{cov}_{\mathrm{1:n}}(r, T)}{|R|} > \mathrm{cov}_{\mathrm{1:n}}(r', T) \implies \mathrm{cov}_{\mathrm{m:n}}(R, T) > \mathrm{cov}_{\mathrm{m:n}}(R \cup \{r'\}, T) \tag{B.28}$$

Die Bedingungen (B.18) – (B.23) folgen hier aus den Eigenschaften (B.7) – (B.12) für die Funktion $\mathrm{cov}_{\mathrm{1:n}}$, die Bedingungen (B.24) – (B.28) aus den Eigenschaften (B.13) – (B.17) für $\mathrm{cov}_{\mathrm{m:1}}$.

B.2. Definition der Bewertungsmetriken

Auf Basis der in Abschnitt B.1 definierten Eigenschaften, welche eine Metrikfunktion für Testfälle zur Bewertung der Anforderungsüberdeckung erfüllen soll, werden im

Menge der Teilsequenzen von $\langle a, b, c, a\rangle$:

$$\text{pseq}_{\text{all}}(\langle a, b, c, a\rangle) = \{\langle a\rangle, \langle b\rangle, \langle c\rangle, \langle a, b\rangle, \langle b, c\rangle, \langle c, a\rangle, \langle a, b, c\rangle, \langle b, c, a\rangle, \langle a, b, c, a\rangle\}$$

Menge der Teilsequenzen mit Länge 2 von $\langle a, b, c, a\rangle$:

$$\text{pseq}_{\text{n}}(\langle a, b, c, a\rangle, 2) = \{\langle a, b\rangle, \langle b, c\rangle, \langle c, a\rangle\}$$

Menge der gemeinsamen, nicht-überlappenden Teile von $\langle a, b, c, a\rangle$ und $\langle a, e, f, a, b\rangle$:

$$\text{parts}(\langle a, b, c, a\rangle, \langle a, e, f, a, b\rangle) = \{\langle a, b\rangle, \langle a\rangle\}$$

Anzahl des Vorkommens von $\langle a, a\rangle$ in $\langle a, a, a, a\rangle$:

$$\text{seq\#}(\langle a, a, a, a\rangle, \langle a, a\rangle) = 2$$

Tritt $\langle a\rangle$ bevor $\langle a, b\rangle$ innerhalb von $\langle a, b, a\rangle$ auf?

$$\text{bf}(\langle a\rangle, \langle a, b\rangle, \langle a, b, a\rangle) = \mathsf{false}$$

Abbildung B.1.: Beispielhafte Anwendung der Hilfsfunktionen, welche für die Definition der Anforderungsüberdeckung verwendet werden.

Folgenden entsprechende Metrikfunktionen für eine 1:1-, 1:n- und m:n-Überdeckung definiert. Es wird darauf verzichtet, auch die m:1-Anforderungsüberdeckung anzugeben, da diese nur einen Spezialfall der m:n-Überdeckung bildet.

Als Basis für die Bewertung der Überdeckung werden die gemeinsamen Teilsequenzen über I^* betrachtet, welche sowohl in den Testfällen als auch den Anforderungsszenarien vorkommen. Insbesondere werden folgende Fragestellungen berücksichtigt:

- Wie hoch ist die Anzahl der Teilsequenzen, die sowohl in Testfall als auch Anforderungsszenario vorkommen?
- Entspricht die Reihenfolge der Teilsequenzen im Testfall der Reihenfolge in der Anforderungsszenario?

Hilfsfunktionen

Zur Definition der Metrikfunktionen $\text{cov}_{1:1}$ und $\text{cov}_{\text{m:n}}$ werden die folgende Hilfsfunktionen verwendet. Zur Vereinfachung sind diese nur über den Eingabeteil von Anforderungsszenarien beziehungsweise Testfällen definiert.

Da sich die Metrikfunktion zur Überdeckungsbewertung auf die gemeinsamen Teilsequenzen und deren Anordnung stützt, werden Hilfsfunktionen benötigt, um Teil-

sequenzen und gemeinsame, nicht überlappende Teile in einem Testfall und eines Anforderungsszenarios zu ermitteln. Dazu beschreiben zunächst die Funktionen

$$\text{pseq}_{\text{all}} \in I^* \to \mathbb{P}(I^*)$$

und

$$\text{pseq}_{\text{n}} \in I^* \times \mathbb{N}_+ \to \mathbb{P}(I^*)$$

die Ermittlung aller nicht-leeren Teilsequenzen, welche in einer Sequenz $x \in I^*$ vorkommen. Die zweite Funktion pseq_{n} liefert dabei nur die Teilsequenzen der angegebenen Länge. Sie sind definiert als

$$\text{pseq}_{\text{all}}(x) \stackrel{\text{def}}{=} \{s \neq \langle\rangle \,|\exists u, v \in I^* : u \frown s \frown v = x\}$$

beziehungsweise

$$\text{pseq}_{\text{n}}(x, n) \stackrel{\text{def}}{=} \{s | s \in \text{pseq}_{\text{all}}(x) \wedge \#s = n\}\,.$$

Die Hilfsfunktion

$$\text{parts} \in \mathit{REQ} \times I^* \to \mathbb{P}(I^*)$$

gibt die gemeinsamen Teilsequenzen in einem Anforderungsszenario $r \in \mathit{REQ}$ und einem Testfall $t \in I^*$ an, wobei die Teilsequenzen maximaler Länge zuerst ausgewählt werden. Diese Funktion ist definiert wie folgt:

$$\text{parts}(r, t) \stackrel{\text{def}}{=} \text{parts}'(r, t, \#t)$$

wobei wiederum

$$\text{parts}' \in I^* \times I^* \times \mathbb{N} \to \mathbb{P}(I^*)$$

definiert durch

$$\text{parts}'(r, t, n) \stackrel{\text{def}}{=} \begin{cases} \{\} & \text{falls } n = 0 \\ \text{parts}'(r, t, n-1) & \text{falls } n > 0 \wedge \text{pseq}_{\text{n}}(r, n) \cap \text{pseq}_{\text{n}}(t, n) = \{\} \\ \{s\} \cup \text{parts}'(u \frown v, t, n) & \text{sonst, mit } r = u \frown s \frown v, u, v \in I^* \end{cases}$$

die gemeinsamen nicht-überlappenden Teilsequenzen in r und t mit maximaler Länge n angibt.

Die Funktion

$$\text{seq}\# \in I^* \backslash \langle\rangle \times I^* \to \mathbb{N}$$

gibt die Anzahl der Wiederholungen einer Sequenz s als Teilsequenz innerhalb einer Sequenz x an. Für diese gilt:

$$\text{seq}\#(s, x) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{falls } s \notin \text{pseq}_{\text{all}}(x) \\ \text{seq}\#(u \frown v) + 1 & \text{falls } \exists u, v \in I^* : x = u \frown s \frown v \end{cases}$$

Das Prädikat

$$\mathrm{bf} \in I^* \times I^* \times I^* \to \mathbb{B}\mathrm{ool},$$

definiert durch

$$\mathrm{bf}(s_1, s_2, s) \stackrel{\mathsf{def}}{=} \exists u, v \in I^* : u \frown s_1 \frown v \frown s_2 \sqsubseteq s \wedge s_2 \notin \mathrm{pseq}_{\mathrm{all}}(u \frown s_1 \frown v)$$

gibt an, ob die Sequenz s_1 vor der Sequenz s_2 als Teilsequenz innerhalb von s vorkommt.

In Abbildung B.1 sind zur Verdeutlichung einige beispielhafte Anwendungen der eingeführten Hilfsfunktionen angegeben.

B.2.1. Definition der 1:1-Anforderungsüberdeckung $\mathrm{cov}_{1:1}$

Die Überdeckung einer Anforderung durch einen Testfall wird definiert als

$$\mathrm{cov}_{1:1}(r, t) \stackrel{\mathsf{def}}{=} \mathrm{rSeq}(\langle \triangleright \rangle \frown \mathsf{in}.r, \langle \triangleright \rangle \frown \mathsf{in}.t) * \mathrm{rOrd}(\langle \triangleright \rangle \frown \mathsf{in}.r, \langle \triangleright \rangle \frown \mathsf{in}.t).$$

Die Metrik setzt sich aus zwei Faktoren zusammen:

- $\mathrm{rSeq}(\langle \triangleright \rangle \frown \mathsf{in}.r, \langle \triangleright \rangle \frown \mathsf{in}.t)$, dem Verhältnis von kumulierten Vorkommen der Teilsequenzen der Anforderung im Testfall zu der Anzahl der Teilsequenzen der Anforderung
- $\mathrm{rOrd}(\langle \triangleright \rangle \frown \mathsf{in}.r, \langle \triangleright \rangle \frown \mathsf{in}.t)$, dem Verhältnis der Anzahl von Paaren gemeinsamer Teile in Anforderung und Testfall, welche in Anforderung und Testfall in der gleichen Reihenfolge vorkommen, zur Anzahl aller möglichen Paare von gemeinsamen Teilen

Wie dargestellt wird hier den Eingaben eines Testfall t beziehungsweise einer Anforderung r ein Zeichen $\triangleright$ vorangestellt. Das ursprüngliche Eingabealphabet I wird also zur Messung der Überdeckung um $\triangleright \notin I$ erweitert. Dies ist erforderlich, um den Anfang einer Anforderung auch bei der Betrachtung von Teilsequenzen zu berücksichtigen. Eine Teilsequenz, welche etwa am Beginn der Anforderung auftritt, ist nur dann vollständig durch den Testfall abgedeckt, falls auch dieser mit ihr beginnt.

Der erste der beiden Faktoren ist dabei wie folgt definiert:

$$\mathrm{rSeq}(r, t) \stackrel{\mathsf{def}}{=} \frac{1}{|\mathrm{pseq}_{\mathrm{all}}(r)|} * \sum_{s \in \mathrm{pseq}_{\mathrm{all}}(r)} \mathrm{seq}\#(s, t)$$

Hier wird die Anzahl der Vorkommen von Teilsequenzen der Anforderung im Testfall aufsummiert und durch die Anzahl aller Teilsequenzen der Anforderung dividiert. Es ist

zu beachten, dass längere Teilsequenzen diesen Faktor überproportional beeinflussen, da sich diese wiederum in mehrere Teilsequenzen zerlegen lassen. Findet sich beispielsweise in Testfall und Anforderung die Teilsequenz $\langle a, b, c\rangle$, so resultieren daraus insgesamt 6 einzelne Teilsequenzen ($\langle\langle a\rangle, \langle b\rangle, \langle c\rangle, \langle a, b\rangle, \langle b, c\rangle, \langle a, b, c\rangle\rangle$), in welchen sich Testfall und Anforderung überdecken. Somit wird eine Überdeckung langer Teilsequenzen stärker gewichtet. (Allgemein ist das Gewicht einer Teilsequenz s mit $\frac{\#s*(\#s+1)}{2}$ bewertet.) Wiederholungen fließen durch die Verwendung von $\text{seq}\#(s,t)$ in die Bewertung mit ein, wodurch sich auch $\text{rSeq}(r,t) > 1$ ergeben kann. Da der Anforderung und dem Testfall allerdings das Startsymbol $\triangleright$ vorangestellt ist, ist eine vollständige Wiederholung nicht möglich und die Überdeckung wird nur in geringerem Maße beeinflusst. Beispielsweise gilt $\text{rSeq}(\langle\triangleright, a, b\rangle, \langle\triangleright, a, b\rangle = \frac{6}{6}$, wogegen bei direktem zweimaligen Aneinanderreihen der Anforderungsszenarios $\text{rSeq}(\langle\triangleright, a, b\rangle, \langle\triangleright, a, b, a, b\rangle = \frac{9}{6}$ gilt.

Der zweite Faktor zielt dagegen nicht auf die Anzahl der gemeinsamen Teilsequenzen ab, sondern darauf, ob nicht-überlappende Teile in Anforderung und Testfall paarweise in gleicher Reihenfolge auftreten. Der Faktor ist definiert als

$$\text{rOrd}(r,t) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{falls } |\,\text{parts}(r,t)| \leq 1 \\ \frac{|P'|}{|P|} & \text{sonst} \end{cases}$$

mit

$$P = \{\{p_1, p_2\} | p_1, p_2 \in \text{parts}(r,t)\}$$
$$P' = \{\{p_1, p_2\} \in P |\, \text{bf}(p_1, p_2, r) = \text{bf}(p_1, p_2, t)\}$$

Dabei werden zunächst die gemeinsamen, nicht-überlappenden Teile maximaler Länge in Testfall und Anforderung ermittelt (Funktion parts) und daraus alle möglichen Paarungen von Teilen ermittelt (Menge P). Ein Teilmenge davon ($P' \subseteq P$) gibt alle Paarungen an, welche in Testfall und Anforderung in gleicher Reihenfolge auftreten. Falls es keine zwei gleichen Teile in Anforderung und Testfall gibt, ist der Faktor 1. Wiederholungen bleiben hier unberücksichtigt, es wird nur überprüft, ob die Ordnung in einem Fall erhalten ist. Somit ist dieser Faktor immer im Bereich $[0 \ldots 1]$.

B.2.2. Definition der 1:n-Anforderungsüberdeckung $\text{cov}_{1:n}$

Für die Bewertung der Überdeckung einer Anforderung durch eine Menge von Testfällen müssen zusätzlich die folgenden Eigenschaften berücksichtigt werden:

- Zusätzliche Testfälle erhöhen die Überdeckung, falls diese selbst zur Überdeckung beitragen (Eigenschaft B.12)
- Testfälle, welche Präfixe von anderen Testfällen sind, bleiben unberücksichtigt (Eigenschaft B.11)

Diese Eigenschaften sind erfüllt, indem die Werte der 1:1-Überdeckungen für alle Testfälle, welche nicht Präfix eines anderen Testfalls sind, summiert werden. $\text{cov}_{1:n}$ wird demnach definiert als

$$\text{cov}_{1:n}(r, T) \stackrel{\text{def}}{=} \sum_{t \in T'} \text{cov}_{1:1}(r, t)$$

mit

$$T' = \{t \in T | \nexists t' \in T : t \sqsubseteq t' \wedge t \neq t'\}$$

B.2.3. Definition der m:n-Anforderungsüberdeckung $\text{cov}_{m:n}$

Die Überdeckung einer Menge von Anforderungen durch eine Menge von Testfällen beruht auf folgenden Überlegungen, welche die zusätzliche Bedingung für die m:1-Überdeckung $\text{cov}_{m:1}$ widerspiegelt:

- Zusätzliche Anforderungen verringern die Überdeckung (Eigenschaften B.16 und B.17).

Dieser Eigenschaft wird genügt, wenn der summierte Wert der 1:n-Überdeckungen $\text{cov}_{1:n}$ (Überdeckung einer Anforderung durch eine Menge von Testfällen) hinsichtlich der Anzahl der Anforderungen normiert wird. $\text{cov}_{m:n}$ kann somit durch

$$\text{cov}_{m:n}(R, T) \stackrel{\text{def}}{=} \frac{1}{|R|} * \sum_{r \in R} \text{cov}_{1:n}(r, T)$$

definiert werden. Die m:n-Anforderungsüberdeckung resultiert somit aus der Summe aller paarweisen 1:1-Anforderungsüberdeckung für Anforderungsszenarien und Testfälle im Verhältnis zur Anzahl der Anforderungen. Für eine Testfallmenge, welche von gleicher Mächtigkeit wie die Menge der Szenarien ist, ergibt sich damit (abgesehen von Wiederholungen in den Testfällen), dass die Menge der Szenarien die optimale Testsuite darstellt. Die vorgestellte Metrik zielt aber vor allem darauf ab, dass eine Testfallmenge bewertet wird, welche zusätzliche Testfälle (zu jeder Anforderung) umfasst.

C. Anforderungsspezifische Teilmodelle

Aus den Tabellen auf den folgenden Seiten sind Beispiele zu Teilmodellen des Modells der Fensterheberfunktion ersichtlich, welche sich für verschiedene Gewichtungen ergeben. Jede Spalte repräsentiert ein Teilmodell zu einer Anforderung $r_x \in REQ$. In den Zeilen sind die Transitionen des Gesamtmodells angegeben. $\times$ kennzeichnet, dass die Transition im Teilmodell für die entsprechende Anforderung gewählt wurde.

Auf die Angabe der Transitionen, welche in den Zustand $s_{\lrcorner}$ führen und die Pseudo-Ausgabe $\lrcorner$ produzieren wurde verzichtetet, da diese nur zur Vervollständigung der Mealy-Maschine ergänzt wurden und für die Testfallgenerierung ohne Belang sind.

Die Tabellen geben somit für alle $r \in REQ$ die Menge der Transitionen

$$T_r = \text{submodTrans}[s_0](r, \text{classReq}, \text{weight}, n)$$

mit classReq entsprechend Beispiel 15 (Seite 120) und Gewichtung

$$\text{weight}(c) = \begin{cases} w_{initial} & \text{für } c = initial \\ w_{added} & \text{für } c = added \end{cases}$$

ermittelt wurden. Die Belegungen für $w_{initial}$, w_{added} und n sind in den Kopfzeilen der Tabellen angegeben.

$w_{initial} = 1, w_{added} = 1, n = 0$												
	r_1	r_2	r_3	r_4'	r_4''	r_5	r_6	r_7	r_8	r_9'	r_9''	r_{10}
t_1	×	×	×	×	×	×	×	×	×	×	×	×
t_2												
t_3												
t_4												
t_5	×	×	×	×	×	×	×	×	×	×	×	×
t_6												
t_7												
t_8												
t_9												
t_{10}												
t_{11}												
t_{12}												
t_{13}	×					×					×	
t_{14}												
t_{15}	×		×								×	
t_{16}	×										×	
t_{17}												
t_{18}												
t_{19}												
t_{20}	×			×	×						×	
t_{21}												
t_{22}												
t_{23}												
t_{24}	×	×									×	
t_{25}												
t_{26}												
t_{27}												
t_{28}					×		×			×	×	
t_{29}					×		×					×
t_{30}					×		×		×			
t_{31}												
t_{32}					×		×					
t_{33}												
t_{34}												
t_{35}												
t_{36}					×		×	×				
t_{37}				×	×					×	×	
t_{38}												
t_{39}												
t_{40}												
t_{41}				×	×					×	×	
t_{42}												
t_{43}												
t_{44}												
t_{45}				×	×					×	×	
t_{46}												
t_{47}												
t_{48}												

$w_{initial} = 1, w_{added} = 2, n = 0$												
	r_1	r_2	r_3	r_4'	r_4''	r_5	r_6	r_7	r_8	r_9'	r_9''	r_{10}
t_1	×	×	×	×	×	×	×	×	×	×	×	×
t_2												
t_3												
t_4												
t_5	×	×	×	×	×	×	×	×	×	×	×	×
t_6												
t_7												
t_8												
t_9												
t_{10}												
t_{11}												
t_{12}												
t_{13}	×					×					×	
t_{14}												
t_{15}	×		×								×	
t_{16}	×										×	
t_{17}												
t_{18}												
t_{19}												
t_{20}	×	×	×	×	×	×	×	×	×		×	×
t_{21}												
t_{22}												
t_{23}												
t_{24}	×	×									×	
t_{25}												
t_{26}												
t_{27}												
t_{28}	×	×	×		×	×	×	×	×	×	×	×
t_{29}					×		×					×
t_{30}					×		×		×			
t_{31}												
t_{32}					×		×					
t_{33}												
t_{34}												
t_{35}												
t_{36}					×		×	×				
t_{37}	×			×	×		×			×	×	
t_{38}												
t_{39}												
t_{40}												
t_{41}	×			×	×		×			×	×	
t_{42}												
t_{43}												
t_{44}												
t_{45}	×			×	×		×			×	×	
t_{46}												
t_{47}												
t_{48}												

$w_{initial}=1, w_{added}=1, n=1$													
	r_1	r_2	r_3	r_4'	r_4''	r_5	r_6	r_7	r_8	r_9'	r_9''	r_{10}
t_1	×	×	×	×	×	×	×	×	×	×	×	×
t_2		×	×		×	×		×	×		×	×
t_3		×	×		×	×		×	×		×	×
t_4		×	×		×	×		×	×		×	×
t_5	×	×	×	×	×	×	×	×	×	×	×	×
t_6		×	×		×	×		×	×		×	×
t_7		×	×		×	×		×	×		×	×
t_8		×	×		×	×		×	×		×	×
t_9		×	×		×	×		×	×		×	×
t_{10}		×	×		×	×		×	×		×	×
t_{11}		×	×		×	×		×	×		×	×
t_{12}		×	×		×	×		×	×		×	×
t_{13}	×	×	×		×	×		×	×		×	×
t_{14}	×										×	
t_{15}	×	×	×		×	×		×	×		×	×
t_{16}	×	×	×		×	×		×	×		×	×
t_{17}	×										×	
t_{18}	×										×	
t_{19}	×										×	
t_{20}	×	×	×	×	×	×		×	×		×	×
t_{21}	×										×	
t_{22}	×										×	
t_{23}	×										×	
t_{24}	×	×	×		×	×		×	×		×	×
t_{25}					×		×					
t_{26}					×		×					
t_{27}					×		×					
t_{28}		×	×		×	×	×	×	×	×	×	×
t_{29}		×	×		×	×	×	×	×		×	×
t_{30}		×	×		×	×	×	×	×		×	×
t_{31}					×		×					
t_{32}		×	×		×	×	×	×	×		×	×
t_{33}					×		×					
t_{34}					×		×					
t_{35}					×		×					
t_{36}		×	×		×	×	×	×	×		×	×
t_{37}	×			×	×		×			×	×	
t_{38}				×	×					×	×	
t_{39}				×	×					×	×	
t_{40}				×	×					×	×	
t_{41}	×			×	×		×			×	×	
t_{42}				×	×					×	×	
t_{43}				×	×					×	×	
t_{44}				×	×					×	×	
t_{45}	×			×	×		×			×	×	
t_{46}				×	×					×	×	
t_{47}				×	×					×	×	
t_{48}				×	×					×	×	

$w_{initial}=1, w_{added}=2, n=1$												
	r_1	r_2	r_3	r_4'	r_4''	r_5	r_6	r_7	r_8	r_9'	r_9''	r_{10}
t_1	×	×	×	×	×	×	×	×	×	×	×	×
t_2		×	×		×	×		×	×		×	×
t_3		×	×		×	×		×	×		×	×
t_4		×	×		×	×		×	×		×	×
t_5	×	×	×	×	×	×	×	×	×	×	×	×
t_6		×	×		×	×		×	×		×	×
t_7		×	×		×	×		×	×		×	×
t_8		×	×		×	×		×	×		×	×
t_9		×	×		×	×		×	×		×	×
t_{10}		×	×		×	×		×	×		×	×
t_{11}		×	×		×	×		×	×		×	×
t_{12}		×	×		×	×		×	×		×	×
t_{13}	×	×	×		×	×		×	×		×	×
t_{14}	×										×	
t_{15}	×	×	×		×	×		×	×		×	×
t_{16}	×	×	×		×	×		×	×		×	×
t_{17}	×										×	
t_{18}	×										×	
t_{19}	×										×	
t_{20}	×	×	×	×	×	×	×	×	×	×	×	×
t_{21}	×										×	
t_{22}	×										×	
t_{23}	×										×	
t_{24}	×	×	×		×	×		×	×		×	×
t_{25}					×		×					
t_{26}					×		×					
t_{27}					×		×					
t_{28}	×	×	×	×	×	×	×	×	×	×	×	×
t_{29}		×	×		×	×	×	×	×		×	×
t_{30}		×	×		×	×	×	×	×		×	×
t_{31}					×		×					
t_{32}		×	×		×	×	×	×	×		×	×
t_{33}					×		×					
t_{34}					×		×					
t_{35}					×		×					
t_{36}		×	×		×	×	×	×	×		×	×
t_{37}	×	×	×	×	×	×	×	×	×	×	×	×
t_{38}				×	×					×	×	
t_{39}				×	×					×	×	
t_{40}				×	×					×	×	
t_{41}	×	×	×	×	×	×	×	×	×	×	×	×
t_{42}				×	×					×	×	
t_{43}				×	×					×	×	
t_{44}				×	×					×	×	
t_{45}	×	×	×	×	×	×	×	×	×	×	×	×
t_{46}				×	×					×	×	
t_{47}				×	×					×	×	
t_{48}				×	×					×	×	

D. Testfälle aus den einzelnen anforderungsspezifischen Teilmodellen

D.1. Testfälle aus Teilmodell zu Anforderung r_1

tc	Schritt 1	Schritt 2	Schritt 3	Schritt 4	Schritt 5	Schritt 6
1	in.*tc* = ⟨(*open*, *middle*), (*open*, *bottom*)⟩					
	out.*tc* = ⟨(*down*, ⊥),	(*stop*, ⊥)⟩				
2	in.*tc* = ⟨(*open*, *middle*),	(⊥, *middle*)⟩				
	out.*tc* = ⟨(*down*, ⊥),	(*stop*, *err*)⟩				
3	in.*tc* = ⟨(*open*, *middle*),	(⊥, *top*)⟩				
	out.*tc* = ⟨(*down*, ⊥),	(*stop*, ⊥)⟩				
4	in.*tc* = ⟨(*open*, *middle*),	(⊥, *bottom*)⟩				
	out.*tc* = ⟨(*down*, ⊥),	(*stop*, ⊥)⟩				
5	in.*tc* = ⟨(*open*, *middle*), (*open*, *middle*)⟩					
	out.*tc* = ⟨(*down*, ⊥),	(*stop*, *err*)⟩				
6	in.*tc* = ⟨(*open*, *middle*),	(⊥, *moving*)⟩				
	out.*tc* = ⟨(*down*, ⊥),	(*stop*, ⊥)⟩				
7	in.*tc* = ⟨(*open*, *middle*),	(*open*, *top*)⟩				
	out.*tc* = ⟨(*down*, ⊥),	(*down*, ⊥)⟩				
8	in.*tc* = ⟨(*open*, *middle*), (*open*, *moving*)⟩					
	out.*tc* = ⟨(*down*, ⊥),	(*down*, ⊥)⟩				
9	in.*tc* = ⟨(*open*, *middle*), (*close*, *middle*)⟩					
	out.*tc* = ⟨(*down*, ⊥),	(*stop*, ⊥)⟩				
10	in.*tc* = ⟨(*open*, *middle*), (*close*, *bottom*)⟩					
	out.*tc* = ⟨(*down*, ⊥),	(*stop*, ⊥)⟩				
11	in.*tc* = ⟨(*open*, *middle*), (*close*, *moving*)⟩					
	out.*tc* = ⟨(*down*, ⊥),	(*stop*, ⊥)⟩				
12	in.*tc* = ⟨(*open*, *middle*),	(*close*, *top*),	(*close*, *middle*)⟩			
	out.*tc* = ⟨(*down*, ⊥),	(*stop*, ⊥),	(*stop*, ⊥)⟩			
13	in.*tc* = ⟨(*open*, *middle*),	(*close*, *top*),	(*open*, *middle*)⟩			
	out.*tc* = ⟨(*down*, ⊥),	(*stop*, ⊥),	(*stop*, ⊥)⟩			
14	in.*tc* = ⟨(*open*, *middle*),	(*close*, *top*),	(⊥, *middle*)⟩			
	out.*tc* = ⟨(*down*, ⊥),	(*stop*, ⊥),	(*stop*, ⊥)⟩			
15	in.*tc* = ⟨(*close*, *middle*), (*open*, *moving*)⟩					
	out.*tc* = ⟨(*up*, ⊥),	(*stop*, ⊥)⟩				

D.2. Testfälle aus Teilmodell zu Anforderungen r_2, r_3 und r_5

tc	Schritt 1	Schritt 2	Schritt 3	Schritt 4	Schritt 5	Schritt 6
1	in.*tc* = ⟨(*open*, *middle*), (*open*, *moving*), (*open*, *middle*)⟩					
	out.*tc* = ⟨(*down*, ⊥),	(*down*, ⊥),	(*stop*, *err*)⟩			

2	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (\bot, moving)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot)\rangle$
3	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, moving)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (down, \bot)\rangle$
4	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, bottom), (\bot, middle)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, \bot)\rangle$
5	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, bottom), (\bot, top)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, \bot)\rangle$
6	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, bottom), (\bot, bottom)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, \bot)\rangle$
7	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, bottom), (\bot, moving)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, err)\rangle$
8	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, bottom), (open, middle)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (down, \bot)\rangle$
9	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, bottom), (open, bottom)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, \bot)\rangle$
10	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, bottom), (open, top)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (down, \bot)\rangle$
11	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, bottom), (open, moving)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, err)\rangle$
12	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, bottom), (close, top)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, \bot)\rangle$
13	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, bottom), (close, bottom)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (up, \bot)\rangle$
14	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, bottom), (close, moving)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, err)\rangle$
15	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (close, moving), (open, middle)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, \bot)\rangle$
16	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (close, moving), (close, middle)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, \bot)\rangle$
17	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (close, moving), (\bot, middle)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, \bot)\rangle$
18	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, bottom), (close, middle), (\bot, moving)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (up, \bot), (stop, \bot)\rangle$
19	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, bottom), (close, middle), (open, moving)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (up, \bot), (stop, \bot)\rangle$
20	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, bottom), (close, middle), (close, middle)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (up, \bot), (stop, err)\rangle$
21	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, bottom), (close, middle), (close, top)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (up, \bot), (stop, \bot)\rangle$
22	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, bottom), (close, middle), (close, moving)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (up, \bot), (up, \bot)\rangle$

D.3. Testfälle aus Teilmodell zu Anforderung r_4'

tc	Schritt 1	Schritt 2	Schritt 3	Schritt 4	Schritt 5	Schritt 6
1	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (close, moving), (\bot, bottom), (open, middle)\rangle$					
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, \bot), (down, \bot)\rangle$					
2	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (close, moving), (open, top)\rangle$					
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, \bot)\rangle$					

3	in.tc = ⟨(*open*, *middle*), (*open*, *moving*), (*close*, *moving*), (⊥, *middle*)⟩
	out.tc = ⟨(*down*, ⊥), (*down*, ⊥), (*stop*, ⊥), (*stop*, ⊥)⟩
4	in.tc = ⟨(*open*, *middle*), (*open*, *moving*), (*close*, *moving*), (⊥, *top*)⟩
	out.tc = ⟨(*down*, ⊥), (*down*, ⊥), (*stop*, ⊥), (*stop*, ⊥)⟩
5	in.tc = ⟨(*open*, *middle*), (*open*, *moving*), (*close*, *moving*), (⊥, *moving*)⟩
	out.tc = ⟨(*down*, ⊥), (*down*, ⊥), (*stop*, ⊥), (*stop*, *err*)⟩
6	in.tc = ⟨(*open*, *middle*), (*open*, *moving*), (*close*, *moving*), (*open*, *bottom*)⟩
	out.tc = ⟨(*down*, ⊥), (*down*, ⊥), (*stop*, ⊥), (*stop*, ⊥)⟩
7	in.tc = ⟨(*open*, *middle*), (*open*, *moving*), (*close*, *moving*), (*open*, *middle*)⟩
	out.tc = ⟨(*down*, ⊥), (*down*, ⊥), (*stop*, ⊥), (*stop*, ⊥)⟩
8	in.tc = ⟨(*open*, *middle*), (*open*, *moving*), (*close*, *moving*), (*open*, *moving*)⟩
	out.tc = ⟨(*down*, ⊥), (*down*, ⊥), (*stop*, ⊥), (*stop*, *err*)⟩
9	in.tc = ⟨(*open*, *middle*), (*open*, *moving*), (*close*, *moving*), (*close*, *middle*)⟩
	out.tc = ⟨(*down*, ⊥), (*down*, ⊥), (*stop*, ⊥), (*stop*, ⊥)⟩
10	in.tc = ⟨(*open*, *middle*), (*open*, *moving*), (*close*, *moving*), (*close*, *top*)⟩
	out.tc = ⟨(*down*, ⊥), (*down*, ⊥), (*stop*, ⊥), (*stop*, ⊥)⟩
11	in.tc = ⟨(*open*, *middle*), (*open*, *moving*), (*close*, *moving*), (*close*, *bottom*)⟩
	out.tc = ⟨(*down*, ⊥), (*down*, ⊥), (*stop*, ⊥), (*stop*, ⊥)⟩
12	in.tc = ⟨(*open*, *middle*), (*open*, *moving*), (*close*, *moving*), (*close*, *moving*)⟩
	out.tc = ⟨(*down*, ⊥), (*down*, ⊥), (*stop*, ⊥), (*stop*, *err*)⟩
13	in.tc = ⟨(*open*, *middle*), (*open*, *moving*), (*close*, *moving*), (⊥, *bottom*), (*close*, *middle*), (*open*, *moving*)⟩
	out.tc = ⟨(*down*, ⊥), (*down*, ⊥), (*stop*, ⊥), (*stop*, ⊥), (*up*, ⊥), (*stop*, ⊥)⟩

D.4. Testfälle aus Teilmodell zu Anforderung r_4''

tc	Schritt 1	Schritt 2	Schritt 3	Schritt 4	Schritt 5	Schritt 6
1	in.tc = ⟨(*open*, *middle*), (*open*, *moving*), (*open*, *bottom*)⟩					
	out.tc = ⟨(*down*, ⊥), (*down*, ⊥), (*stop*, ⊥)⟩					
2	in.tc = ⟨(*open*, *middle*), (*open*, *moving*), (⊥, *moving*)⟩					
	out.tc = ⟨(*down*, ⊥), (*down*, ⊥), (*stop*, ⊥)⟩					
3	in.tc = ⟨(*open*, *middle*), (*open*, *moving*), (*open*, *moving*)⟩					
	out.tc = ⟨(*down*, ⊥), (*down*, ⊥), (*down*, ⊥)⟩					
4	in.tc = ⟨(*open*, *middle*), (*open*, *moving*), (*open*, *middle*), (⊥, *bottom*)⟩					
	out.tc = ⟨(*down*, ⊥), (*down*, ⊥), (*stop*, *err*), (*stop*, ⊥)⟩					
5	in.tc = ⟨(*open*, *middle*), (*open*, *moving*), (*open*, *middle*), (⊥, *middle*)⟩					
	out.tc = ⟨(*down*, ⊥), (*down*, ⊥), (*stop*, *err*), (*stop*, ⊥)⟩					
6	in.tc = ⟨(*open*, *middle*), (*open*, *moving*), (*open*, *middle*), (⊥, *top*)⟩					
	out.tc = ⟨(*down*, ⊥), (*down*, ⊥), (*stop*, *err*), (*stop*, ⊥)⟩					
7	in.tc = ⟨(*open*, *middle*), (*open*, *moving*), (*open*, *middle*), (⊥, *moving*)⟩					
	out.tc = ⟨(*down*, ⊥), (*down*, ⊥), (*stop*, *err*), (*stop*, *err*)⟩					
8	in.tc = ⟨(*open*, *middle*), (*open*, *moving*), (*open*, *middle*), (*open*, *bottom*)⟩					
	out.tc = ⟨(*down*, ⊥), (*down*, ⊥), (*stop*, *err*), (*stop*, ⊥)⟩					
9	in.tc = ⟨(*open*, *middle*), (*open*, *moving*), (*open*, *middle*), (*open*, *middle*)⟩					
	out.tc = ⟨(*down*, ⊥), (*down*, ⊥), (*stop*, *err*), (*down*, ⊥)⟩					
10	in.tc = ⟨(*open*, *middle*), (*open*, *moving*), (*open*, *middle*), (*open*, *top*)⟩					
	out.tc = ⟨(*down*, ⊥), (*down*, ⊥), (*stop*, *err*), (*down*, ⊥)⟩					
11	in.tc = ⟨(*open*, *middle*), (*open*, *moving*), (*open*, *middle*), (*open*, *moving*)⟩					
	out.tc = ⟨(*down*, ⊥), (*down*, ⊥), (*stop*, *err*), (*stop*, *err*)⟩					
12	in.tc = ⟨(*open*, *middle*), (*open*, *moving*), (*open*, *middle*), (*close*, *top*)⟩					
	out.tc = ⟨(*down*, ⊥), (*down*, ⊥), (*stop*, *err*), (*stop*, ⊥)⟩					

13	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, middle), (close, bottom)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, err), (up, \bot)\rangle$
14	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, middle), (close, moving)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, err), (stop, err)\rangle$
15	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (close, moving), (close, top)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, \bot)\rangle$
16	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (close, moving), (\bot, middle)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, \bot)\rangle$
17	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (close, moving), (\bot, top)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, \bot)\rangle$
18	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (close, moving), (\bot, moving)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, err)\rangle$
19	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (close, moving), (open, middle)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, \bot)\rangle$
20	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (close, moving), (open, top)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, \bot)\rangle$
21	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (close, moving), (open, bottom)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, \bot)\rangle$
22	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (close, moving), (open, moving)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, err)\rangle$
23	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (close, moving), (close, middle)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, \bot)\rangle$
24	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (close, moving), (close, bottom)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, \bot)\rangle$
25	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (close, moving), (close, moving)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, err)\rangle$
26	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (close, moving), (\bot, bottom)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, \bot)\rangle$
27	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, middle), (close, middle), (open, middle)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, err), (up, \bot), (stop, \bot)\rangle$
28	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, middle), (close, middle), (\bot, top)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, err), (up, \bot), (stop, \bot)\rangle$
29	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, middle), (close, middle), (\bot, bottom)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, err), (up, \bot), (stop, \bot)\rangle$
30	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, middle), (close, middle), (\bot, moving)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, err), (up, \bot), (stop, \bot)\rangle$
31	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, middle), (close, middle), (open, bottom)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, err), (up, \bot), (stop, \bot)\rangle$
32	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, middle), (close, middle), (open, top)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, err), (up, \bot), (stop, \bot)\rangle$
33	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, middle), (close, middle), (open, moving)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, err), (up, \bot), (stop, \bot)\rangle$
34	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, middle), (close, middle), (close, middle)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, err), (up, \bot), (stop, err)\rangle$
35	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, middle), (close, middle), (close, top)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, err), (up, \bot), (stop, \bot)\rangle$
36	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, middle), (close, middle), (close, bottom)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, err), (up, \bot), (up, \bot)\rangle$
37	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, middle), (close, middle), (close, moving)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, err), (up, \bot), (up, \bot)\rangle$
38	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, middle), (close, middle), (\bot, middle)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, err), (up, \bot), (stop, err)\rangle$

D.5. Testfälle aus Teilmodell zu Anforderung r_6

tc	Schritt 1	Schritt 2	Schritt 3	Schritt 4	Schritt 5	Schritt 6
1	in.*tc* = ⟨(*open*, *middle*), (*close*, *moving*), (*open*, *middle*)⟩					
	out.*tc* = ⟨(*down*, ⊥), (*stop*, ⊥), (*stop*, ⊥)⟩					
2	in.*tc* = ⟨(*open*, *middle*), (*close*, *moving*), (*close*, *middle*)⟩					
	out.*tc* = ⟨(*down*, ⊥), (*stop*, ⊥), (*stop*, ⊥)⟩					
3	in.*tc* = ⟨(*open*, *middle*), (*close*, *moving*), (⊥, *middle*)⟩					
	out.*tc* = ⟨(*down*, ⊥), (*stop*, ⊥), (*stop*, ⊥)⟩					
4	in.*tc* = ⟨(*close*, *middle*), (*close*, *top*)⟩					
	out.*tc* = ⟨(*up*, ⊥), (*stop*, ⊥)⟩					
5	in.*tc* = ⟨(*close*, *middle*), (⊥, *top*)⟩					
	out.*tc* = ⟨(*up*, ⊥), (*stop*, ⊥)⟩					
6	in.*tc* = ⟨(*close*, *middle*), (⊥, *bottom*)⟩					
	out.*tc* = ⟨(*up*, ⊥), (*stop*, ⊥)⟩					
7	in.*tc* = ⟨(*close*, *middle*), (⊥, *moving*)⟩					
	out.*tc* = ⟨(*up*, ⊥), (*stop*, ⊥)⟩					
8	in.*tc* = ⟨(*close*, *middle*), (*open*, *middle*)⟩					
	out.*tc* = ⟨(*up*, ⊥), (*stop*, ⊥)⟩					
9	in.*tc* = ⟨(*close*, *middle*), (*open*, *top*)⟩					
	out.*tc* = ⟨(*up*, ⊥), (*stop*, ⊥)⟩					
10	in.*tc* = ⟨(*close*, *middle*), (*open*, *bottom*)⟩					
	out.*tc* = ⟨(*up*, ⊥), (*stop*, ⊥)⟩					
11	in.*tc* = ⟨(*close*, *middle*), (*open*, *moving*)⟩					
	out.*tc* = ⟨(*up*, ⊥), (*stop*, ⊥)⟩					
12	in.*tc* = ⟨(*close*, *middle*), (*close*, *middle*)⟩					
	out.*tc* = ⟨(*up*, ⊥), (*stop*, *err*)⟩					
13	in.*tc* = ⟨(*close*, *middle*), (*close*, *bottom*)⟩					
	out.*tc* = ⟨(*up*, ⊥), (*up*, ⊥)⟩					
14	in.*tc* = ⟨(*close*, *middle*), (*close*, *moving*)⟩					
	out.*tc* = ⟨(*up*, ⊥), (*up*, ⊥)⟩					
15	in.*tc* = ⟨(*close*, *middle*), (⊥, *middle*)⟩					
	out.*tc* = ⟨(*up*, ⊥), (*stop*, *err*)⟩					

D.6. Testfälle aus Teilmodell zu Anforderung r_7, r_8 und r_{10}

tc	Schritt 1	Schritt 2	Schritt 3	Schritt 4	Schritt 5	Schritt 6
1	in.*tc* = ⟨(*close*, *middle*), (*close*, *moving*), (*close*, *middle*), (*open*, *middle*), (*close*, *moving*)⟩					
	out.*tc* = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, *err*), (*down*, ⊥), (*stop*, ⊥)⟩					
2	in.*tc* = ⟨(*close*, *middle*), (*close*, *moving*), (*close*, *middle*), (*open*, *middle*), (*open*, *bottom*)⟩					
	out.*tc* = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, *err*), (*down*, ⊥), (*stop*, ⊥)⟩					
3	in.*tc* = ⟨(*close*, *middle*), (*close*, *moving*), (*close*, *middle*), (*open*, *middle*), (*open*, *middle*)⟩					
	out.*tc* = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, *err*), (*down*, ⊥), (*stop*, *err*)⟩					
4	in.*tc* = ⟨(*close*, *middle*), (*close*, *moving*), (*close*, *middle*), (*open*, *middle*), (⊥, *moving*)⟩					
	out.*tc* = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, *err*), (*down*, ⊥), (*stop*, ⊥)⟩					
5	in.*tc* = ⟨(*close*, *middle*), (*close*, *moving*), (*close*, *middle*), (*open*, *middle*), (*open*, *moving*)⟩					
	out.*tc* = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, *err*), (*down*, ⊥), (*down*, ⊥)⟩					
6	in.*tc* = ⟨(*close*, *middle*), (*close*, *moving*), (*close*, *middle*), (⊥, *top*)⟩					
	out.*tc* = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, *err*), (*stop*, ⊥)⟩					

7	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*close*, *middle*), (⊥, *bottom*)⟩
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, *err*), (*stop*, ⊥)⟩
8	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*close*, *middle*), (⊥, *moving*)⟩
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, *err*), (*stop*, *err*)⟩
9	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*close*, *middle*), (*open*, *bottom*)⟩
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, *err*), (*stop*, ⊥)⟩
10	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*close*, *middle*), (*open*, *top*)⟩
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, *err*), (*down*, ⊥)⟩
11	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*close*, *middle*), (*open*, *moving*)⟩
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, *err*), (*stop*, *err*)⟩
12	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*close*, *middle*), (*close*, *middle*)⟩
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, *err*), (*up*, ⊥)⟩
13	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*close*, *middle*), (*close*, *top*)⟩
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, *err*), (*stop*, ⊥)⟩
14	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*close*, *middle*), (*close*, *bottom*)⟩
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, *err*), (*up*, ⊥)⟩
15	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*close*, *middle*), (*close*, *moving*)⟩
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, *err*), (*stop*, *err*)⟩
16	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*close*, *middle*), (⊥, *middle*)⟩
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, *err*), (*stop*, ⊥)⟩
17	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*open*, *moving*), (⊥, *middle*)⟩
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, ⊥), (*stop*, ⊥)⟩
18	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*open*, *moving*), (*open*, *middle*)⟩
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, ⊥), (*stop*, ⊥)⟩
19	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*open*, *moving*), (*close*, *middle*)⟩
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, ⊥), (*stop*, ⊥)⟩
20	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*close*, *top*)⟩
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, ⊥)⟩
21	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*close*, *moving*)⟩
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*up*, ⊥)⟩
22	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (⊥, *moving*)⟩
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, ⊥)⟩

D.7. Testfälle aus Teilmodell zu Anforderung r'_9

tc	Schritt 1	Schritt 2	Schritt 3	Schritt 4	Schritt 5	Schritt 6
1	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*open*, *moving*), (⊥, *bottom*), (*open*, *middle*), (*close*, *moving*)⟩					
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, ⊥), (*stop*, ⊥), (*down*, ⊥), (*stop*, ⊥)⟩					
2	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*open*, *moving*), (⊥, *bottom*), (*close*, *middle*)⟩					
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, ⊥), (*stop*, ⊥), (*up*, ⊥)⟩					
3	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*open*, *moving*), (*open*, *top*)⟩					
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, ⊥), (*stop*, ⊥)⟩					
4	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*open*, *moving*), (⊥, *middle*)⟩					
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, ⊥), (*stop*, ⊥)⟩					
5	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*open*, *moving*), (⊥, *top*)⟩					
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, ⊥), (*stop*, ⊥)⟩					
6	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*open*, *moving*), (⊥, *moving*)⟩					
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, ⊥), (*stop*, *err*)⟩					
7	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*open*, *moving*), (*close*, *moving*)⟩					
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, ⊥), (*stop*, *err*)⟩					

8	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*open*, *moving*), (*close*, *middle*)⟩
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, ⊥), (*stop*, ⊥)⟩
9	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*open*, *moving*), (*open*, *middle*)⟩
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, ⊥), (*stop*, ⊥)⟩
10	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*open*, *moving*), (*open*, *bottom*)⟩
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, ⊥), (*stop*, ⊥)⟩
11	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*open*, *moving*), (*open*, *moving*)⟩
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, ⊥), (*stop*, *err*)⟩
12	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*open*, *moving*), (*close*, *top*)⟩
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, ⊥), (*stop*, ⊥)⟩
13	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*open*, *moving*), (*close*, *bottom*)⟩
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, ⊥), (*stop*, ⊥)⟩

D.8. Testfälle aus Teilmodell zu Anforderung r_9''

tc	Schritt 1	Schritt 2	Schritt 3	Schritt 4	Schritt 5	Schritt 6
1	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*close*, *middle*), (*open*, *middle*), (*close*, *top*)⟩					
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, *err*), (*down*, ⊥), (*stop*, ⊥)⟩					
2	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*close*, *middle*), (*open*, *middle*), (⊥, *middle*)⟩					
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, *err*), (*down*, ⊥), (*stop*, *err*)⟩					
3	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*close*, *middle*), (*open*, *middle*), (⊥, *top*)⟩					
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, *err*), (*down*, ⊥), (*stop*, ⊥)⟩					
4	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*close*, *middle*), (*open*, *middle*), (⊥, *bottom*)⟩					
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, *err*), (*down*, ⊥), (*stop*, ⊥)⟩					
5	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*close*, *middle*), (*open*, *middle*), (*open*, *middle*)⟩					
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, *err*), (*down*, ⊥), (*stop*, *err*)⟩					
6	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*close*, *middle*), (*open*, *middle*), (*open*, *bottom*)⟩					
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, *err*), (*down*, ⊥), (*stop*, ⊥)⟩					
7	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*close*, *middle*), (*open*, *middle*), (⊥, *moving*)⟩					
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, *err*), (*down*, ⊥), (*stop*, ⊥)⟩					
8	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*close*, *middle*), (*open*, *middle*), (*open*, *top*)⟩					
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, *err*), (*down*, ⊥), (*down*, ⊥)⟩					
9	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*close*, *middle*), (*open*, *middle*), (*open*, *moving*)⟩					
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, *err*), (*down*, ⊥), (*down*, ⊥)⟩					
10	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*close*, *middle*), (*open*, *middle*), (*close*, *middle*)⟩					
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, *err*), (*down*, ⊥), (*stop*, ⊥)⟩					
11	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*close*, *middle*), (*open*, *middle*), (*close*, *bottom*)⟩					
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, *err*), (*down*, ⊥), (*stop*, ⊥)⟩					
12	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*close*, *middle*), (*open*, *middle*), (*close*, *moving*)⟩					
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, *err*), (*down*, ⊥), (*stop*, ⊥)⟩					
13	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*close*, *middle*), (*open*, *bottom*)⟩					
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, *err*), (*stop*, ⊥)⟩					
14	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*close*, *middle*), (⊥, *top*)⟩					
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, *err*), (*stop*, ⊥)⟩					
15	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*close*, *middle*), (⊥, *bottom*)⟩					
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, *err*), (*stop*, ⊥)⟩					
16	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*close*, *middle*), (⊥, *moving*)⟩					
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, *err*), (*stop*, *err*)⟩					
17	in.tc = ⟨(*close*, *middle*), (*close*, *moving*), (*close*, *middle*), (*open*, *top*)⟩					
	out.tc = ⟨(*up*, ⊥), (*up*, ⊥), (*stop*, *err*), (*down*, ⊥)⟩					

18	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (close, middle), (open, moving)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, err), (stop, err)\rangle$
19	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (close, middle), (close, middle)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, err), (up, \bot)\rangle$
20	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (close, middle), (close, top)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, err), (stop, \bot)\rangle$
21	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (close, middle), (close, bottom)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, err), (up, \bot)\rangle$
22	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (close, middle), (close, moving)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, err), (stop, err)\rangle$
23	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (close, middle), (\bot, middle)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, err), (stop, \bot)\rangle$
24	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (open, moving), (\bot, moving)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, \bot), (stop, err)\rangle$
25	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (open, moving), (close, top)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, \bot), (stop, \bot)\rangle$
26	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (open, moving), (close, bottom)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, \bot), (stop, \bot)\rangle$
27	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (open, moving), (close, moving)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, \bot), (stop, err)\rangle$
28	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (open, moving), (\bot, middle)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, \bot), (stop, \bot)\rangle$
29	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (open, moving), (\bot, top)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, \bot), (stop, \bot)\rangle$
30	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (open, moving), (\bot, bottom)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, \bot), (stop, \bot)\rangle$
31	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (open, moving), (open, middle)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, \bot), (stop, \bot)\rangle$
32	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (open, moving), (open, top)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, \bot), (stop, \bot)\rangle$
33	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (open, moving), (open, bottom)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, \bot), (stop, \bot)\rangle$
34	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (open, moving), (open, moving)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, \bot), (stop, err)\rangle$
35	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (open, moving), (close, middle)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, \bot), (stop, \bot)\rangle$
36	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (close, top)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, \bot)\rangle$
37	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (close, moving)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (up, \bot)\rangle$
38	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (\bot, moving)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, \bot)\rangle$

E. Zusammenfassung der anforderungsorientierten Testfälle

tc	Schritt 1	Schritt 2	Schritt 3	Schritt 4	Schritt 5	Schritt 6
1	in.*tc* = ⟨(*open*, *middle*), (*open*, *bottom*)⟩					
	out.*tc* = ⟨(*down*, ⊥), (*stop*, ⊥)⟩					
2	in.*tc* = ⟨(*open*, *middle*), (⊥, *middle*)⟩					
	out.*tc* = ⟨(*down*, ⊥), (*stop*, *err*)⟩					
3	in.*tc* = ⟨(*open*, *middle*), (⊥, *top*)⟩					
	out.*tc* = ⟨(*down*, ⊥), (*stop*, ⊥)⟩					
4	in.*tc* = ⟨(*open*, *middle*), (⊥, *bottom*)⟩					
	out.*tc* = ⟨(*down*, ⊥), (*stop*, ⊥)⟩					
5	in.*tc* = ⟨(*open*, *middle*), (*open*, *middle*)⟩					
	out.*tc* = ⟨(*down*, ⊥), (*stop*, *err*)⟩					
6	in.*tc* = ⟨(*open*, *middle*), (⊥, *moving*)⟩					
	out.*tc* = ⟨(*down*, ⊥), (*stop*, ⊥)⟩					
7	in.*tc* = ⟨(*open*, *middle*), (*open*, *top*)⟩					
	out.*tc* = ⟨(*down*, ⊥), (*down*, ⊥)⟩					
8	in.*tc* = ⟨(*open*, *middle*), (*open*, *moving*), (*open*, *middle*), (⊥, *bottom*)⟩					
	out.*tc* = ⟨(*down*, ⊥), (*down*, ⊥), (*stop*, *err*), (*stop*, ⊥)⟩					
9	in.*tc* = ⟨(*open*, *middle*), (*close*, *middle*)⟩					
	out.*tc* = ⟨(*down*, ⊥), (*stop*, ⊥)⟩					
10	in.*tc* = ⟨(*open*, *middle*), (*close*, *bottom*)⟩					
	out.*tc* = ⟨(*down*, ⊥), (*stop*, ⊥)⟩					
11	in.*tc* = ⟨(*open*, *middle*), (*close*, *moving*), (*open*, *middle*)⟩					
	out.*tc* = ⟨(*down*, ⊥), (*stop*, ⊥), (*stop*, ⊥)⟩					
12	in.*tc* = ⟨(*open*, *middle*), (*close*, *top*), (*close*, *middle*)⟩					
	out.*tc* = ⟨(*down*, ⊥), (*stop*, ⊥), (*stop*, ⊥)⟩					
13	in.*tc* = ⟨(*open*, *middle*), (*close*, *top*), (*open*, *middle*)⟩					
	out.*tc* = ⟨(*down*, ⊥), (*stop*, ⊥), (*stop*, ⊥)⟩					
14	in.*tc* = ⟨(*open*, *middle*), (*close*, *top*), (⊥, *middle*)⟩					
	out.*tc* = ⟨(*down*, ⊥), (*stop*, ⊥), (*stop*, ⊥)⟩					
15	in.*tc* = ⟨(*close*, *middle*), (*open*, *moving*)⟩					
	out.*tc* = ⟨(*up*, ⊥), (*stop*, ⊥)⟩					
16	in.*tc* = ⟨(*open*, *middle*), (*open*, *moving*), (⊥, *moving*)⟩					
	out.*tc* = ⟨(*down*, ⊥), (*down*, ⊥), (*stop*, ⊥)⟩					
17	in.*tc* = ⟨(*open*, *middle*), (*open*, *moving*), (*open*, *moving*)⟩					
	out.*tc* = ⟨(*down*, ⊥), (*down*, ⊥), (*down*, ⊥)⟩					
18	in.*tc* = ⟨(*open*, *middle*), (*open*, *moving*), (*open*, *bottom*), (⊥, *middle*)⟩					
	out.*tc* = ⟨(*down*, ⊥), (*down*, ⊥), (*stop*, ⊥), (*stop*, ⊥)⟩					
19	in.*tc* = ⟨(*open*, *middle*), (*open*, *moving*), (*open*, *bottom*), (⊥, *top*)⟩					
	out.*tc* = ⟨(*down*, ⊥), (*down*, ⊥), (*stop*, ⊥), (*stop*, ⊥)⟩					
20	in.*tc* = ⟨(*open*, *middle*), (*open*, *moving*), (*open*, *bottom*), (⊥, *bottom*)⟩					
	out.*tc* = ⟨(*down*, ⊥), (*down*, ⊥), (*stop*, ⊥), (*stop*, ⊥)⟩					
21	in.*tc* = ⟨(*open*, *middle*), (*open*, *moving*), (*open*, *bottom*), (⊥, *moving*)⟩					

	$\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, err)\rangle$
22	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, bottom), (open, middle)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (down, \bot)\rangle$
23	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, bottom), (open, bottom)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, \bot)\rangle$
24	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, bottom), (open, top)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (down, \bot)\rangle$
25	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, bottom), (open, moving)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, err)\rangle$
26	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, bottom), (close, top)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, \bot)\rangle$
27	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, bottom), (close, bottom)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (up, \bot)\rangle$
28	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, bottom), (close, moving)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, err)\rangle$
29	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (close, moving), (open, middle)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, \bot)\rangle$
30	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (close, moving), (close, middle)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, \bot)\rangle$
31	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (close, moving), (\bot, middle)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, \bot)\rangle$
32	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, bottom), (close, middle), (\bot, moving)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (up, \bot), (stop, \bot)\rangle$
33	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, bottom), (close, middle), (open, moving)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (up, \bot), (stop, \bot)\rangle$
34	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, bottom), (close, middle), (close, middle)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (up, \bot), (stop, err)\rangle$
35	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, bottom), (close, middle), (close, top)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (up, \bot), (stop, \bot)\rangle$
36	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, bottom), (close, middle), (close, moving)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (up, \bot), (up, \bot)\rangle$
37	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (close, moving), (\bot, bottom), (open, middle)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, \bot), (down, \bot)\rangle$
38	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (close, moving), (open, top)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, \bot)\rangle$
39	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (close, moving), (\bot, top)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, \bot)\rangle$
40	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (close, moving), (\bot, moving)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, err)\rangle$
41	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (close, moving), (open, bottom)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, \bot)\rangle$
42	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (close, moving), (open, moving)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, err)\rangle$
43	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (close, moving), (close, top)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, \bot)\rangle$
44	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (close, moving), (close, bottom)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, \bot)\rangle$
45	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (close, moving), (close, moving)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, err)\rangle$
46	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (close, moving), (\bot, bottom), (close, middle), (open, moving)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, \bot), (stop, \bot), (up, \bot), (stop, \bot)\rangle$
47	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, middle), (\bot, middle)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, err), (stop, \bot)\rangle$

48	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, middle), (\bot, top)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, err), (stop, \bot)\rangle$
49	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, middle), (\bot, moving)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, err), (stop, err)\rangle$
50	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, middle), (open, bottom)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, err), (stop, \bot)\rangle$
51	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, middle), (open, middle)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, err), (down, \bot)\rangle$
52	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, middle), (open, top)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, err), (down, \bot)\rangle$
53	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, middle), (open, moving)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, err), (stop, err)\rangle$
54	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, middle), (close, top)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, err), (stop, \bot)\rangle$
55	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, middle), (close, bottom)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, err), (up, \bot)\rangle$
56	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, middle), (close, moving)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, err), (stop, err)\rangle$
57	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, middle), (close, middle), (open, middle)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, err), (up, \bot), (stop, \bot)\rangle$
58	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, middle), (close, middle), (\bot, top)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, err), (up, \bot), (stop, \bot)\rangle$
59	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, middle), (close, middle), (\bot, bottom)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, err), (up, \bot), (stop, \bot)\rangle$
60	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, middle), (close, middle), (\bot, moving)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, err), (up, \bot), (stop, \bot)\rangle$
61	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, middle), (close, middle), (open, bottom)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, err), (up, \bot), (stop, \bot)\rangle$
62	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, middle), (close, middle), (open, top)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, err), (up, \bot), (stop, \bot)\rangle$
63	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, middle), (close, middle), (open, moving)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, err), (up, \bot), (stop, \bot)\rangle$
64	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, middle), (close, middle), (close, middle)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, err), (up, \bot), (stop, err)\rangle$
65	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, middle), (close, middle), (close, top)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, err), (up, \bot), (stop, \bot)\rangle$
66	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, middle), (close, middle), (close, bottom)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, err), (up, \bot), (up, \bot)\rangle$
67	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, middle), (close, middle), (close, moving)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, err), (up, \bot), (up, \bot)\rangle$
68	$\mathsf{in}.tc = \langle(open, middle), (open, moving), (open, middle), (close, middle), (\bot, middle)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (down, \bot), (stop, err), (up, \bot), (stop, err)\rangle$
69	$\mathsf{in}.tc = \langle(open, middle), (close, moving), (close, middle)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (stop, \bot), (stop, \bot)\rangle$
70	$\mathsf{in}.tc = \langle(open, middle), (close, moving), (\bot, middle)\rangle$ $\mathsf{out}.tc = \langle(down, \bot), (stop, \bot), (stop, \bot)\rangle$
71	$\mathsf{in}.tc = \langle(close, middle), (close, top)\rangle$ $\mathsf{out}.tc = \langle(up, \bot), (stop, \bot)\rangle$
72	$\mathsf{in}.tc = \langle(close, middle), (\bot, top)\rangle$ $\mathsf{out}.tc = \langle(up, \bot), (stop, \bot)\rangle$
73	$\mathsf{in}.tc = \langle(close, middle), (\bot, bottom)\rangle$ $\mathsf{out}.tc = \langle(up, \bot), (stop, \bot)\rangle$

74	$\mathsf{in}.tc = \langle(close, middle), \quad (\bot, moving)\rangle$
	$\mathsf{out}.tc = \quad \langle(up, \bot), \quad (stop, \bot)\rangle$
75	$\mathsf{in}.tc = \langle(close, middle), (open, middle)\rangle$
	$\mathsf{out}.tc = \quad \langle(up, \bot), \quad (stop, \bot)\rangle$
76	$\mathsf{in}.tc = \langle(close, middle), \quad (open, top)\rangle$
	$\mathsf{out}.tc = \quad \langle(up, \bot), \quad (stop, \bot)\rangle$
77	$\mathsf{in}.tc = \langle(close, middle), (open, bottom)\rangle$
	$\mathsf{out}.tc = \quad \langle(up, \bot), \quad (stop, \bot)\rangle$
78	$\mathsf{in}.tc = \langle(close, middle), (close, middle)\rangle$
	$\mathsf{out}.tc = \quad \langle(up, \bot), \quad (stop, err)\rangle$
79	$\mathsf{in}.tc = \langle(close, middle), (close, bottom)\rangle$
	$\mathsf{out}.tc = \quad \langle(up, \bot), \quad (up, \bot)\rangle$
80	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (close, middle), (open, middle), (close, moving)\rangle$
	$\mathsf{out}.tc = \quad \langle(up, \bot), \quad (up, \bot), \quad (stop, err), \quad (down, \bot), \quad (stop, \bot)\rangle$
81	$\mathsf{in}.tc = \langle(close, middle), \quad (\bot, middle)\rangle$
	$\mathsf{out}.tc = \quad \langle(up, \bot), \quad (stop, err)\rangle$
82	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (close, middle), (open, middle), (open, bottom)\rangle$
	$\mathsf{out}.tc = \quad \langle(up, \bot), \quad (up, \bot), \quad (stop, err), \quad (down, \bot), \quad (stop, \bot)\rangle$
83	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (close, middle), (open, middle), (open, middle)\rangle$
	$\mathsf{out}.tc = \quad \langle(up, \bot), \quad (up, \bot), \quad (stop, err), \quad (down, \bot), \quad (stop, err)\rangle$
84	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (close, middle), (open, middle), \quad (\bot, moving)\rangle$
	$\mathsf{out}.tc = \quad \langle(up, \bot), \quad (up, \bot), \quad (stop, err), \quad (down, \bot), \quad (stop, \bot)\rangle$
85	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (close, middle), (open, middle), (open, moving)\rangle$
	$\mathsf{out}.tc = \quad \langle(up, \bot), \quad (up, \bot), \quad (stop, err), \quad (down, \bot), \quad (down, \bot)\rangle$
86	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (close, middle), \quad (\bot, top)\rangle$
	$\mathsf{out}.tc = \quad \langle(up, \bot), \quad (up, \bot), \quad (stop, err), \quad (stop, \bot)\rangle$
87	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (close, middle), \quad (\bot, bottom)\rangle$
	$\mathsf{out}.tc = \quad \langle(up, \bot), \quad (up, \bot), \quad (stop, err), \quad (stop, \bot)\rangle$
88	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (close, middle), \quad (\bot, moving)\rangle$
	$\mathsf{out}.tc = \quad \langle(up, \bot), \quad (up, \bot), \quad (stop, err), \quad (stop, err)\rangle$
89	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (close, middle), (open, bottom)\rangle$
	$\mathsf{out}.tc = \quad \langle(up, \bot), \quad (up, \bot), \quad (stop, err), \quad (stop, \bot)\rangle$
90	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (close, middle), \quad (open, top)\rangle$
	$\mathsf{out}.tc = \quad \langle(up, \bot), \quad (up, \bot), \quad (stop, err), \quad (down, \bot)\rangle$
91	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (close, middle), (open, moving)\rangle$
	$\mathsf{out}.tc = \quad \langle(up, \bot), \quad (up, \bot), \quad (stop, err), \quad (stop, err)\rangle$
92	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (close, middle), (close, middle)\rangle$
	$\mathsf{out}.tc = \quad \langle(up, \bot), \quad (up, \bot), \quad (stop, err), \quad (up, \bot)\rangle$
93	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (close, middle), \quad (close, top)\rangle$
	$\mathsf{out}.tc = \quad \langle(up, \bot), \quad (up, \bot), \quad (stop, err), \quad (stop, \bot)\rangle$
94	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (close, middle), (close, bottom)\rangle$
	$\mathsf{out}.tc = \quad \langle(up, \bot), \quad (up, \bot), \quad (stop, err), \quad (up, \bot)\rangle$
95	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (close, middle), (close, moving)\rangle$
	$\mathsf{out}.tc = \quad \langle(up, \bot), \quad (up, \bot), \quad (stop, err), \quad (stop, err)\rangle$
96	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (close, middle), \quad (\bot, middle)\rangle$
	$\mathsf{out}.tc = \quad \langle(up, \bot), \quad (up, \bot), \quad (stop, err), \quad (stop, \bot)\rangle$
97	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (open, moving), \quad (\bot, middle)\rangle$
	$\mathsf{out}.tc = \quad \langle(up, \bot), \quad (up, \bot), \quad (stop, \bot), \quad (stop, \bot)\rangle$
98	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (open, moving), (open, middle)\rangle$
	$\mathsf{out}.tc = \quad \langle(up, \bot), \quad (up, \bot), \quad (stop, \bot), \quad (stop, \bot)\rangle$
99	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (open, moving), (close, middle)\rangle$
	$\mathsf{out}.tc = \quad \langle(up, \bot), \quad (up, \bot), \quad (stop, \bot), \quad (stop, \bot)\rangle$

100	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (close, top)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, \bot)\rangle$
101	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (close, moving)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (up, \bot)\rangle$
102	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (\bot, moving)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, \bot)\rangle$
103	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (open, moving), (\bot, bottom), (open, middle), (close, moving)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, \bot), (stop, \bot), (down, \bot), (stop, \bot)\rangle$
104	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (open, moving), (\bot, bottom), (close, middle)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, \bot), (stop, \bot), (up, \bot)\rangle$
105	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (open, moving), (open, top)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, \bot), (stop, \bot)\rangle$
106	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (open, moving), (\bot, top)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, \bot), (stop, \bot)\rangle$
107	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (open, moving), (\bot, moving)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, \bot), (stop, err)\rangle$
108	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (open, moving), (close, moving)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, \bot), (stop, err)\rangle$
109	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (open, moving), (open, bottom)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, \bot), (stop, \bot)\rangle$
110	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (open, moving), (open, moving)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, \bot), (stop, err)\rangle$
111	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (open, moving), (close, top)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, \bot), (stop, \bot)\rangle$
112	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (open, moving), (close, bottom)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, \bot), (stop, \bot)\rangle$
113	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (close, middle), (open, middle), (close, top)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, err), (down, \bot), (stop, \bot)\rangle$
114	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (close, middle), (open, middle), (\bot, middle)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, err), (down, \bot), (stop, err)\rangle$
115	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (close, middle), (open, middle), (\bot, top)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, err), (down, \bot), (stop, \bot)\rangle$
116	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (close, middle), (open, middle), (\bot, bottom)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, err), (down, \bot), (stop, \bot)\rangle$
117	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (close, middle), (open, middle), (open, top)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, err), (down, \bot), (down, \bot)\rangle$
118	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (close, middle), (open, middle), (close, middle)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, err), (down, \bot), (stop, \bot)\rangle$
119	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (close, middle), (open, middle), (close, bottom)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, err), (down, \bot), (stop, \bot)\rangle$

F. Vergleichs-Testsuiten

F.1. Testfälle aus Transitionsüberdeckung im Gesamtmodell

tc	Schritt 1	Schritt 2	Schritt 3	Schritt 4	Schritt 5	Schritt 6
1	in.tc = $\langle$(*open*, *middle*),	(⊥, *middle*)$\rangle$				
	out.tc = $\langle$(*down*, ⊥),	(*stop*, *err*)$\rangle$				
2	in.tc = $\langle$(*open*, *middle*),	(⊥, *top*)$\rangle$				
	out.tc = $\langle$(*down*, ⊥),	(*stop*, ⊥)$\rangle$				
3	in.tc = $\langle$(*open*, *middle*),	(⊥, *bottom*)$\rangle$				
	out.tc = $\langle$(*down*, ⊥),	(*stop*, ⊥)$\rangle$				
4	in.tc = $\langle$(*open*, *middle*),	(*close*, *top*)$\rangle$				
	out.tc = $\langle$(*down*, ⊥),	(*stop*, ⊥)$\rangle$				
5	in.tc = $\langle$(*open*, *middle*),	(*open*, *middle*)$\rangle$				
	out.tc = $\langle$(*down*, ⊥),	(*stop*, *err*)$\rangle$				
6	in.tc = $\langle$(*open*, *middle*),	(*open*, *bottom*)$\rangle$				
	out.tc = $\langle$(*down*, ⊥),	(*stop*, ⊥)$\rangle$				
7	in.tc = $\langle$(*open*, *middle*),	(⊥, *moving*)$\rangle$				
	out.tc = $\langle$(*down*, ⊥),	(*stop*, ⊥)$\rangle$				
8	in.tc = $\langle$(*open*, *middle*),	(*open*, *top*)$\rangle$				
	out.tc = $\langle$(*down*, ⊥),	(*down*, ⊥)$\rangle$				
9	in.tc = $\langle$(*open*, *middle*),	(*open*, *moving*)$\rangle$				
	out.tc = $\langle$(*down*, ⊥),	(*down*, ⊥)$\rangle$				
10	in.tc = $\langle$(*open*, *middle*),	(*close*, *middle*)$\rangle$				
	out.tc = $\langle$(*down*, ⊥),	(*stop*, ⊥)$\rangle$				
11	in.tc = $\langle$(*open*, *middle*),	(*close*, *bottom*)$\rangle$				
	out.tc = $\langle$(*down*, ⊥),	(*stop*, ⊥)$\rangle$				
12	in.tc = $\langle$(*open*, *bottom*)$\rangle$					
	out.tc = $\langle$(*stop*, ⊥)$\rangle$					
13	in.tc = $\langle$(⊥, *top*)$\rangle$					
	out.tc = $\langle$(*stop*, ⊥)$\rangle$					
14	in.tc = $\langle$(⊥, *bottom*)$\rangle$					
	out.tc = $\langle$(*stop*, ⊥)$\rangle$					
15	in.tc = $\langle$(⊥, *moving*)$\rangle$					
	out.tc = $\langle$(*stop*, *err*)$\rangle$					
16	in.tc = $\langle$(*open*, *top*)$\rangle$					
	out.tc = $\langle$(*down*, ⊥)$\rangle$					
17	in.tc = $\langle$(*open*, *moving*)$\rangle$					
	out.tc = $\langle$(*stop*, *err*)$\rangle$					
18	in.tc = $\langle$(*close*, *top*)$\rangle$					
	out.tc = $\langle$(*stop*, ⊥)$\rangle$					
19	in.tc = $\langle$(*close*, *bottom*)$\rangle$					
	out.tc = $\langle$(*up*, ⊥)$\rangle$					
20	in.tc = $\langle$(*close*, *moving*)$\rangle$					
	out.tc = $\langle$(*stop*, *err*)$\rangle$					

21	in.tc =	$\langle(\bot, middle)\rangle$
	out.tc =	$\langle(stop, \bot)\rangle$
22	in.tc =	$\langle(open, middle), (close, moving), (open, middle)\rangle$
	out.tc =	$\langle(down, \bot), (stop, \bot), (stop, \bot)\rangle$
23	in.tc =	$\langle(open, middle), (close, moving), (\bot, middle)\rangle$
	out.tc =	$\langle(down, \bot), (stop, \bot), (stop, \bot)\rangle$
24	in.tc =	$\langle(open, middle), (close, moving), (\bot, top)\rangle$
	out.tc =	$\langle(down, \bot), (stop, \bot), (stop, \bot)\rangle$
25	in.tc =	$\langle(open, middle), (close, moving), (\bot, moving)\rangle$
	out.tc =	$\langle(down, \bot), (stop, \bot), (stop, err)\rangle$
26	in.tc =	$\langle(open, middle), (close, moving), (open, top)\rangle$
	out.tc =	$\langle(down, \bot), (stop, \bot), (stop, \bot)\rangle$
27	in.tc =	$\langle(open, middle), (close, moving), (open, bottom)\rangle$
	out.tc =	$\langle(down, \bot), (stop, \bot), (stop, \bot)\rangle$
28	in.tc =	$\langle(open, middle), (close, moving), (open, moving)\rangle$
	out.tc =	$\langle(down, \bot), (stop, \bot), (stop, err)\rangle$
29	in.tc =	$\langle(open, middle), (close, moving), (close, middle)\rangle$
	out.tc =	$\langle(down, \bot), (stop, \bot), (stop, \bot)\rangle$
30	in.tc =	$\langle(open, middle), (close, moving), (close, top)\rangle$
	out.tc =	$\langle(down, \bot), (stop, \bot), (stop, \bot)\rangle$
31	in.tc =	$\langle(open, middle), (close, moving), (close, bottom)\rangle$
	out.tc =	$\langle(down, \bot), (stop, \bot), (stop, \bot)\rangle$
32	in.tc =	$\langle(open, middle), (close, moving), (close, moving)\rangle$
	out.tc =	$\langle(down, \bot), (stop, \bot), (stop, err)\rangle$
33	in.tc =	$\langle(open, middle), (close, moving), (\bot, bottom)\rangle$
	out.tc =	$\langle(down, \bot), (stop, \bot), (stop, \bot)\rangle$
34	in.tc =	$\langle(close, middle), (close, moving)\rangle$
	out.tc =	$\langle(up, \bot), (up, \bot)\rangle$
35	in.tc =	$\langle(close, middle), (\bot, top)\rangle$
	out.tc =	$\langle(up, \bot), (stop, \bot)\rangle$
36	in.tc =	$\langle(close, middle), (\bot, bottom)\rangle$
	out.tc =	$\langle(up, \bot), (stop, \bot)\rangle$
37	in.tc =	$\langle(close, middle), (\bot, moving)\rangle$
	out.tc =	$\langle(up, \bot), (stop, \bot)\rangle$
38	in.tc =	$\langle(close, middle), (open, middle)\rangle$
	out.tc =	$\langle(up, \bot), (stop, \bot)\rangle$
39	in.tc =	$\langle(close, middle), (open, top)\rangle$
	out.tc =	$\langle(up, \bot), (stop, \bot)\rangle$
40	in.tc =	$\langle(close, middle), (open, bottom)\rangle$
	out.tc =	$\langle(up, \bot), (stop, \bot)\rangle$
41	in.tc =	$\langle(close, middle), (open, moving)\rangle$
	out.tc =	$\langle(up, \bot), (stop, \bot)\rangle$
42	in.tc =	$\langle(close, middle), (close, middle)\rangle$
	out.tc =	$\langle(up, \bot), (stop, err)\rangle$
43	in.tc =	$\langle(close, middle), (close, top)\rangle$
	out.tc =	$\langle(up, \bot), (stop, \bot)\rangle$
44	in.tc =	$\langle(close, middle), (close, bottom)\rangle$
	out.tc =	$\langle(up, \bot), (up, \bot)\rangle$
45	in.tc =	$\langle(close, middle), (\bot, middle)\rangle$
	out.tc =	$\langle(up, \bot), (stop, err)\rangle$

F.2. Randomisierte Testeingaben

tc	Schritt 1	Schritt 2	Schritt 3	Schritt 4	Schritt 5	Schritt 6
1	$in.tc = \langle(close, middle),$	$(close, top),$	$(close, bottom),$	$(close, top),$	$(open, middle),$	$(close, moving)\rangle$
	$out.tc = \langle(up, \bot),$	$(stop, \bot),$	$(up, \bot),$	$(stop, \bot),$	$(down, \bot),$	$(stop, \bot)\rangle$
2	$in.tc = \langle(open, middle),$	$(\bot, bottom),$	$(\bot, top),$	$(close, moving),$	$(open, moving)\rangle$	
	$out.tc = \langle(down, \bot),$	$(stop, \bot),$	$(stop, \bot),$	$(stop, err),$	$(stop, err)\rangle$	
3	$in.tc = \langle(\bot, top),$	$(close, top),$	$(close, middle),$	$(open, moving)\rangle$		
	$out.tc = \langle(stop, \bot),$	$(stop, \bot),$	$(up, \bot),$	$(stop, \bot)\rangle$		
4	$in.tc = \langle(close, middle),$	$(open, middle),$	$(open, middle),$	$(open, middle),$	$(close, middle)\rangle$	
	$out.tc = \langle(up, \bot),$	$(stop, \bot),$	$(stop, \bot),$	$(stop, \bot),$	$(stop, \bot)\rangle$	
5	$in.tc = \langle(close, top),$	$(\bot, top),$	$(close, bottom),$	$(open, bottom)\rangle$		
	$out.tc = \langle(stop, \bot),$	$(stop, \bot),$	$(up, \bot),$	$(stop, \bot)\rangle$		
6	$in.tc = \langle(close, middle),$	$(close, top)\rangle$				
	$out.tc = \langle(up, \bot),$	$(stop, \bot)\rangle$				
7	$in.tc = \langle(\bot, bottom),$	$(\bot, top),$	$(\bot, moving),$	$(\bot, bottom),$	$(open, bottom)\rangle$	
	$out.tc = \langle(stop, \bot),$	$(stop, \bot),$	$(stop, err),$	$(stop, \bot),$	$(stop, \bot)\rangle$	
8	$in.tc = \langle(close, middle),$	$(close, middle),$	$(close, middle),$	$(open, middle)\rangle$		
	$out.tc = \langle(up, \bot),$	$(stop, err),$	$(up, \bot),$	$(stop, \bot)\rangle$		
9	$in.tc = \langle(open, bottom),$	$(\bot, bottom),$	$(\bot, moving),$	$(\bot, bottom)\rangle$		
	$out.tc = \langle(stop, \bot),$	$(stop, \bot),$	$(stop, err),$	$(stop, \bot)\rangle$		
10	$in.tc = \langle(close, middle),$	$(\bot, bottom),$	$(\bot, middle)\rangle$			
	$out.tc = \langle(up, \bot),$	$(stop, \bot),$	$(stop, \bot)\rangle$			
11	$in.tc = \langle(open, middle),$	$(\bot, moving),$	$(close, top),$	$(close, top)\rangle$		
	$out.tc = \langle(down, \bot),$	$(stop, \bot),$	$(stop, \bot),$	$(stop, \bot)\rangle$		
12	$in.tc = \langle(close, bottom),$	$(open, middle)\rangle$				
	$out.tc = \langle(up, \bot),$	$(stop, \bot)\rangle$				
13	$in.tc = \langle(close, moving),$	$(\bot, top),$	$(\bot, moving),$	$(\bot, top),$	$(open, top)\rangle$	
	$out.tc = \langle(stop, err),$	$(stop, \bot),$	$(stop, err),$	$(stop, \bot),$	$(down, \bot)\rangle$	
14	$in.tc = \langle(close, top),$	$(open, top)\rangle$				
	$out.tc = \langle(stop, \bot),$	$(down, \bot)\rangle$				
15	$in.tc = \langle(\bot, top),$	$(\bot, top)\rangle$				
	$out.tc = \langle(stop, \bot),$	$(stop, \bot)\rangle$				
16	$in.tc = \langle(\bot, bottom),$	$(close, moving),$	$(open, moving),$	$(close, top),$	$(\bot, top)\rangle$	
	$out.tc = \langle(stop, \bot),$	$(stop, err),$	$(stop, err),$	$(stop, \bot),$	$(stop, \bot)\rangle$	
17	$in.tc = \langle(open, top),$	$(close, middle),$	$(close, bottom),$	$(open, top)\rangle$		
	$out.tc = \langle(down, \bot),$	$(stop, \bot),$	$(stop, \bot),$	$(stop, \bot)\rangle$		
18	$in.tc = \langle(open, top),$	$(\bot, top),$	$(open, top),$	$(\bot, middle),$	$(\bot, top)\rangle$	
	$out.tc = \langle(down, \bot),$	$(stop, \bot),$	$(down, \bot),$	$(stop, err),$	$(stop, \bot)\rangle$	
19	$in.tc = \langle(close, moving),$	$(\bot, moving),$	$(close, moving),$	$(close, middle)\rangle$		
	$out.tc = \langle(stop, err),$	$(stop, err),$	$(stop, err),$	$(up, \bot)\rangle$		
20	$in.tc = \langle(\bot, bottom),$	$(close, bottom),$	$(\bot, top),$	$(close, bottom)\rangle$		
	$out.tc = \langle(stop, \bot),$	$(up, \bot),$	$(stop, \bot),$	$(up, \bot)\rangle$		
21	$in.tc = \langle(close, top),$	$(open, top)\rangle$				
	$out.tc = \langle(stop, \bot),$	$(down, \bot)\rangle$				
22	$in.tc = \langle(open, moving),$	$(open, moving),$	$(close, bottom),$	$(open, bottom)\rangle$		
	$out.tc = \langle(stop, err),$	$(stop, err),$	$(up, \bot),$	$(stop, \bot)\rangle$		
23	$in.tc = \langle(close, moving),$	$(\bot, bottom),$	$(open, bottom)\rangle$			
	$out.tc = \langle(stop, err),$	$(stop, \bot),$	$(stop, \bot)\rangle$			
24	$in.tc = \langle(open, moving),$	$(close, middle),$	$(\bot, bottom)\rangle$			
	$out.tc = \langle(stop, err),$	$(up, \bot),$	$(stop, \bot)\rangle$			
25	$in.tc = \langle(open, top),$	$(open, moving),$	$(\bot, top),$	$(close, top)\rangle$		

	out.tc =	$\langle$(*down*, $\perp$),	(*down*, $\perp$),	(*stop*, $\perp$),	(*stop*, $\perp$)$\rangle$	
26	in.tc =	$\langle$($\perp$, *top*),	(*open*, *moving*),	($\perp$, *top*)$\rangle$		
	out.tc =	$\langle$(*stop*, $\perp$),	(*stop*, *err*),	(*stop*, $\perp$)$\rangle$		
27	in.tc = $\langle$(*close*, *moving*),	(*open*, *top*)$\rangle$				
	out.tc =	$\langle$(*stop*, *err*),	(*down*, $\perp$)$\rangle$			
28	in.tc = $\langle$(*close*, *moving*),	($\perp$, *moving*),	($\perp$, *middle*)$\rangle$			
	out.tc =	$\langle$(*stop*, *err*),	(*stop*, *err*),	(*stop*, $\perp$)$\rangle$		
29	in.tc = $\langle$(*open*, *moving*),	(*open*, *bottom*),	($\perp$, *bottom*),	(*close*, *bottom*),	(*open*, *bottom*)$\rangle$	
	out.tc =	$\langle$(*stop*, *err*),	(*stop*, $\perp$),	(*stop*, $\perp$),	(*up*, $\perp$),	(*stop*, $\perp$)$\rangle$
30	in.tc =	$\langle$($\perp$, *middle*),	(*close*, *middle*),	(*close*, *moving*)$\rangle$		
	out.tc =	$\langle$(*stop*, $\perp$),	(*up*, $\perp$),	(*up*, $\perp$)$\rangle$		
31	in.tc = $\langle$(*open*, *middle*),	(*close*, *moving*),	(*close*, *middle*)$\rangle$			
	out.tc =	$\langle$(*down*, $\perp$),	(*stop*, $\perp$),	(*stop*, $\perp$)$\rangle$		
32	in.tc = $\langle$(*close*, *moving*),	($\perp$, *moving*),	($\perp$, *moving*),	($\perp$, *top*),	(*close*, *top*)$\rangle$	
	out.tc =	$\langle$(*stop*, *err*),	(*stop*, *err*),	(*stop*, *err*),	(*stop*, $\perp$),	(*stop*, $\perp$)$\rangle$
33	in.tc = $\langle$(*open*, *middle*),	(*close*, *bottom*),	($\perp$, *middle*),	($\perp$, *moving*),	(*open*, *top*)$\rangle$	
	out.tc =	$\langle$(*down*, $\perp$),	(*stop*, $\perp$),	(*stop*, $\perp$),	(*stop*, *err*),	(*down*, $\perp$)$\rangle$
34	in.tc = $\langle$(*open*, *bottom*),	($\perp$, *top*),	(*open*, *top*)$\rangle$			
	out.tc =	$\langle$(*stop*, $\perp$),	(*stop*, $\perp$),	(*down*, $\perp$)$\rangle$		
35	in.tc =	$\langle$(*open*, *top*),	($\perp$, *middle*),	(*close*, *top*),	(*open*, *top*),	(*open*, *middle*)$\rangle$
	out.tc =	$\langle$(*down*, $\perp$),	(*stop*, *err*),	(*stop*, $\perp$),	(*down*, $\perp$),	(*stop*, *err*)$\rangle$
36	in.tc =	$\langle$($\perp$, *top*),	(*close*, *moving*),	($\perp$, *middle*),	(*open*, *top*),	(*close*, *middle*),	(*open*, *top*)$\rangle$
	out.tc =	$\langle$(*stop*, $\perp$),	(*stop*, *err*),	(*stop*, $\perp$),	(*down*, $\perp$),	(*stop*, $\perp$),	(*stop*, $\perp$)$\rangle$
37	in.tc = $\langle$(*close*, *bottom*),	(*open*, *middle*)$\rangle$				
	out.tc =	$\langle$(*up*, $\perp$),	(*stop*, $\perp$)$\rangle$			
38	in.tc =	$\langle$($\perp$, *moving*),	(*close*, *moving*),	(*open*, *bottom*),	($\perp$, *top*),	(*close*, *middle*)$\rangle$
	out.tc =	$\langle$(*stop*, *err*),	(*stop*, *err*),	(*stop*, $\perp$),	(*stop*, $\perp$),	(*up*, $\perp$)$\rangle$
39	in.tc = $\langle$(*close*, *middle*),	(*open*, *top*)$\rangle$				
	out.tc =	$\langle$(*up*, $\perp$),	(*stop*, $\perp$)$\rangle$			
40	in.tc = $\langle$(*close*, *middle*),	(*close*, *moving*),	($\perp$, *bottom*),	(*close*, *top*)$\rangle$		
	out.tc =	$\langle$(*up*, $\perp$),	(*up*, $\perp$),	(*stop*, $\perp$),	(*stop*, $\perp$)$\rangle$	
41	in.tc = $\langle$(*open*, *middle*),	(*open*, *moving*),	(*open*, *bottom*),	(*close*, *middle*),	(*close*, *bottom*)$\rangle$	
	out.tc =	$\langle$(*down*, $\perp$),	(*down*, $\perp$),	(*stop*, $\perp$),	(*up*, $\perp$),	(*up*, $\perp$)$\rangle$
42	in.tc = $\langle$(*open*, *moving*),	(*open*, *middle*)$\rangle$				
	out.tc =	$\langle$(*stop*, *err*),	(*down*, $\perp$)$\rangle$			
43	in.tc = $\langle$(*close*, *middle*),	(*close*, *bottom*),	(*open*, *middle*),	(*close*, *moving*),	(*open*, *middle*),	(*open*, *middle*)$\rangle$
	out.tc =	$\langle$(*up*, $\perp$),	(*up*, $\perp$),	(*stop*, $\perp$),	(*stop*, *err*),	(*stop*, $\perp$),	(*stop*, $\perp$)$\rangle$
44	in.tc = $\langle$(*close*, *bottom*),	($\perp$, *top*),	(*close*, *middle*),	(*open*, *top*)$\rangle$		
	out.tc =	$\langle$(*up*, $\perp$),	(*stop*, $\perp$),	(*up*, $\perp$),	(*stop*, $\perp$)$\rangle$	
45	in.tc = $\langle$(*close*, *bottom*),	($\perp$, *moving*),	($\perp$, *middle*),	($\perp$, *top*)$\rangle$		
	out.tc =	$\langle$(*up*, $\perp$),	(*stop*, $\perp$),	(*stop*, $\perp$),	(*stop*, $\perp$)$\rangle$	
46	in.tc =	$\langle$(*close*, *top*),	(*close*, *bottom*),	($\perp$, *top*)$\rangle$		
	out.tc =	$\langle$(*stop*, $\perp$),	(*up*, $\perp$),	(*stop*, $\perp$)$\rangle$		
47	in.tc = $\langle$(*open*, *bottom*),	(*open*, *bottom*),	(*close*, *top*),	(*close*, *top*),	(*close*, *middle*)$\rangle$	
	out.tc =	$\langle$(*stop*, $\perp$),	(*stop*, $\perp$),	(*stop*, $\perp$),	(*stop*, $\perp$),	(*up*, $\perp$)$\rangle$
48	in.tc =	$\langle$($\perp$, *middle*),	(*close*, *top*),	(*close*, *top*),	(*close*, *bottom*),	($\perp$, *moving*),	($\perp$, *moving*)$\rangle$
	out.tc =	$\langle$(*stop*, $\perp$),	(*stop*, $\perp$),	(*stop*, $\perp$),	(*up*, $\perp$),	(*stop*, $\perp$),	(*stop*, *err*)$\rangle$
49	in.tc =	$\langle$($\perp$, *bottom*),	(*open*, *middle*),	($\perp$, *bottom*),	($\perp$, *bottom*)$\rangle$	
	out.tc =	$\langle$(*stop*, $\perp$),	(*down*, $\perp$),	(*stop*, $\perp$),	(*stop*, $\perp$)$\rangle$	
50	in.tc =	$\langle$($\perp$, *top*),	(*open*, *top*)$\rangle$			
	out.tc =	$\langle$(*stop*, $\perp$),	(*down*, $\perp$)$\rangle$			
51	in.tc = $\langle$(*open*, *moving*),	(*close*, *bottom*),	($\perp$, *middle*),	($\perp$, *moving*),	($\perp$, *middle*),	($\perp$, *moving*)$\rangle$
	out.tc =	$\langle$(*stop*, *err*),	(*up*, $\perp$),	(*stop*, *err*),	(*stop*, *err*),	(*stop*, $\perp$),	(*stop*, *err*)$\rangle$

52	in.tc =	$\langle(open, top), (close, moving)\rangle$
	out.tc =	$\langle(down, \bot), (stop, \bot)\rangle$
53	in.tc =	$\langle(open, bottom), (open, middle), (open, bottom), (close, bottom), (open, bottom), (close, bottom)\rangle$
	out.tc =	$\langle(stop, \bot), (down, \bot), (stop, \bot), (up, \bot), (stop, \bot), (stop, \bot)\rangle$
54	in.tc =	$\langle(close, top), (\bot, bottom), (\bot, top), (\bot, bottom)\rangle$
	out.tc =	$\langle(stop, \bot), (stop, \bot), (stop, \bot), (stop, \bot)\rangle$
55	in.tc =	$\langle(open, top), (close, bottom), (\bot, bottom), (open, middle)\rangle$
	out.tc =	$\langle(down, \bot), (stop, \bot), (stop, \bot), (down, \bot)\rangle$
56	in.tc =	$\langle(open, middle), (open, bottom), (open, moving), (open, bottom), (close, bottom)\rangle$
	out.tc =	$\langle(down, \bot), (stop, \bot), (stop, err), (stop, \bot), (up, \bot)\rangle$
57	in.tc =	$\langle(close, bottom), (open, moving), (open, bottom), (close, moving), (open, middle), (open, moving)\rangle$
	out.tc =	$\langle(up, \bot), (stop, \bot), (stop, \bot), (stop, err), (stop, \bot), (stop, err)\rangle$
58	in.tc =	$\langle(\bot, middle), (close, middle), (\bot, moving)\rangle$
	out.tc =	$\langle(stop, \bot), (up, \bot), (stop, \bot)\rangle$
59	in.tc =	$\langle(\bot, middle), (close, top), (close, top), (open, moving), (close, top), (open, moving)\rangle$
	out.tc =	$\langle(stop, \bot), (stop, \bot), (stop, \bot), (stop, err), (stop, \bot), (stop, err)\rangle$
60	in.tc =	$\langle(close, moving), (open, moving)\rangle$
	out.tc =	$\langle(stop, err), (stop, err)\rangle$
61	in.tc =	$\langle(open, middle), (open, bottom), (open, bottom), (close, bottom), (\bot, bottom), (open, middle)\rangle$
	out.tc =	$\langle(down, \bot), (stop, \bot), (stop, \bot), (up, \bot), (stop, \bot), (down, \bot)\rangle$
62	in.tc =	$\langle(\bot, middle), (\bot, bottom), (close, middle), (close, moving), (open, middle)\rangle$
	out.tc =	$\langle(stop, \bot), (stop, \bot), (up, \bot), (up, \bot), (stop, \bot)\rangle$
63	in.tc =	$\langle(open, bottom), (\bot, moving), (open, top), (close, middle), (open, middle), (open, top)\rangle$
	out.tc =	$\langle(stop, \bot), (stop, err), (down, \bot), (stop, \bot), (stop, \bot), (stop, \bot)\rangle$
64	in.tc =	$\langle(close, moving), (\bot, moving), (\bot, middle)\rangle$
	out.tc =	$\langle(stop, err), (stop, err), (stop, \bot)\rangle$
65	in.tc =	$\langle(open, bottom), (\bot, top), (close, middle), (open, bottom)\rangle$
	out.tc =	$\langle(stop, \bot), (stop, \bot), (up, \bot), (stop, \bot)\rangle$
66	in.tc =	$\langle(\bot, top), (\bot, top), (close, bottom)\rangle$
	out.tc =	$\langle(stop, \bot), (stop, \bot), (up, \bot)\rangle$
67	in.tc =	$\langle(\bot, bottom), (open, moving), (open, moving), (close, middle), (open, bottom)\rangle$
	out.tc =	$\langle(stop, \bot), (stop, err), (stop, err), (up, \bot), (stop, \bot)\rangle$
68	in.tc =	$\langle(open, middle), (close, moving), (\bot, middle), (close, bottom), (open, moving), (close, middle)\rangle$
	out.tc =	$\langle(down, \bot), (stop, \bot), (stop, \bot), (up, \bot), (stop, \bot), (stop, \bot)\rangle$
69	in.tc =	$\langle(\bot, moving), (\bot, moving), (close, top)\rangle$
	out.tc =	$\langle(stop, err), (stop, err), (stop, \bot)\rangle$
70	in.tc =	$\langle(\bot, middle), (open, top), (\bot, middle)\rangle$
	out.tc =	$\langle(stop, \bot), (down, \bot), (stop, err)\rangle$
71	in.tc =	$\langle(open, moving), (open, top), (\bot, bottom)\rangle$
	out.tc =	$\langle(stop, err), (down, \bot), (stop, \bot)\rangle$
72	in.tc =	$\langle(\bot, top), (close, moving)\rangle$
	out.tc =	$\langle(stop, \bot), (stop, err)\rangle$
73	in.tc =	$\langle(\bot, top), (open, middle)\rangle$
	out.tc =	$\langle(stop, \bot), (down, \bot)\rangle$
74	in.tc =	$\langle(\bot, top), (\bot, bottom), (close, moving), (open, top)\rangle$
	out.tc =	$\langle(stop, \bot), (stop, \bot), (stop, err), (down, \bot)\rangle$
75	in.tc =	$\langle(open, bottom), (\bot, top)\rangle$
	out.tc =	$\langle(stop, \bot), (stop, \bot)\rangle$
76	in.tc =	$\langle(open, top), (open, middle), (close, bottom), (open, bottom), (open, middle)\rangle$
	out.tc =	$\langle(down, \bot), (stop, err), (up, \bot), (stop, \bot), (stop, \bot)\rangle$
77	in.tc =	$\langle(close, middle), (open, moving), (\bot, bottom)\rangle$
	out.tc =	$\langle(up, \bot), (stop, \bot), (stop, \bot)\rangle$

78	$\mathsf{in}.tc = \langle(close, bottom), (open, bottom)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (stop, \bot)\rangle$
79	$\mathsf{in}.tc = \langle(\bot, top), (close, moving)\rangle$
	$\mathsf{out}.tc = \langle(stop, \bot), (stop, err)\rangle$
80	$\mathsf{in}.tc = \langle(open, moving), (\bot, bottom), (\bot, top)\rangle$
	$\mathsf{out}.tc = \langle(stop, err), (stop, \bot), (stop, \bot)\rangle$
81	$\mathsf{in}.tc = \langle(close, middle), (close, moving), (\bot, top), (open, top), (open, top), (open, top)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (up, \bot), (stop, \bot), (down, \bot), (down, \bot), (down, \bot)\rangle$
82	$\mathsf{in}.tc = \langle(open, moving), (open, bottom), (open, moving), (\bot, middle)\rangle$
	$\mathsf{out}.tc = \langle(stop, err), (stop, \bot), (stop, err), (stop, \bot)\rangle$
83	$\mathsf{in}.tc = \langle(\bot, top), (open, moving), (close, bottom), (close, top), (\bot, middle), (open, moving)\rangle$
	$\mathsf{out}.tc = \langle(stop, \bot), (stop, err), (up, \bot), (stop, \bot), (stop, \bot), (stop, err)\rangle$
84	$\mathsf{in}.tc = \langle(\bot, middle), (open, top), (\bot, bottom), (open, bottom)\rangle$
	$\mathsf{out}.tc = \langle(stop, \bot), (down, \bot), (stop, \bot), (stop, \bot)\rangle$
85	$\mathsf{in}.tc = \langle(open, moving), (\bot, top), (open, moving)\rangle$
	$\mathsf{out}.tc = \langle(stop, err), (stop, \bot), (stop, err)\rangle$
86	$\mathsf{in}.tc = \langle(open, bottom), (\bot, middle), (close, top)\rangle$
	$\mathsf{out}.tc = \langle(stop, \bot), (stop, \bot), (stop, \bot)\rangle$
87	$\mathsf{in}.tc = \langle(close, bottom), (open, bottom)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (stop, \bot)\rangle$
88	$\mathsf{in}.tc = \langle(close, middle), (\bot, bottom), (\bot, top), (open, bottom), (open, moving)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (stop, \bot), (stop, \bot), (stop, \bot), (stop, err)\rangle$
89	$\mathsf{in}.tc = \langle(open, middle), (close, bottom), (\bot, middle), (open, moving), (close, bottom)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (stop, \bot), (stop, \bot), (stop, err), (up, \bot)\rangle$
90	$\mathsf{in}.tc = \langle(open, bottom), (close, moving), (close, middle), (open, bottom), (open, top)\rangle$
	$\mathsf{out}.tc = \langle(stop, \bot), (stop, err), (up, \bot), (stop, \bot), (stop, \bot)\rangle$
91	$\mathsf{in}.tc = \langle(close, top), (open, bottom), (\bot, moving), (open, top), (open, middle), (close, middle)\rangle$
	$\mathsf{out}.tc = \langle(stop, \bot), (stop, \bot), (stop, err), (down, \bot), (stop, err), (up, \bot)\rangle$
92	$\mathsf{in}.tc = \langle(close, bottom), (open, bottom), (close, moving)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (stop, \bot), (stop, err)\rangle$
93	$\mathsf{in}.tc = \langle(close, bottom), (open, bottom), (open, moving), (\bot, top), (close, middle), (\bot, bottom)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (stop, \bot), (stop, err), (stop, \bot), (up, \bot), (stop, \bot)\rangle$
94	$\mathsf{in}.tc = \langle(open, bottom), (open, top), (\bot, bottom), (open, top), (\bot, bottom), (\bot, middle)\rangle$
	$\mathsf{out}.tc = \langle(stop, \bot), (down, \bot), (stop, \bot), (down, \bot), (stop, \bot), (stop, \bot)\rangle$
95	$\mathsf{in}.tc = \langle(\bot, top), (\bot, bottom), (\bot, bottom), (\bot, middle)\rangle$
	$\mathsf{out}.tc = \langle(stop, \bot), (stop, \bot), (stop, \bot), (stop, \bot)\rangle$
96	$\mathsf{in}.tc = \langle(close, moving), (open, top), (close, bottom), (\bot, top)\rangle$
	$\mathsf{out}.tc = \langle(stop, err), (down, \bot), (stop, \bot), (stop, \bot)\rangle$
97	$\mathsf{in}.tc = \langle(close, top), (open, middle), (open, bottom), (close, moving), (close, top)\rangle$
	$\mathsf{out}.tc = \langle(stop, \bot), (down, \bot), (stop, \bot), (stop, err), (stop, \bot)\rangle$
98	$\mathsf{in}.tc = \langle(open, middle), (\bot, bottom), (close, top), (open, bottom), (close, moving), (close, middle)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (stop, \bot), (stop, \bot), (stop, \bot), (stop, err), (up, \bot)\rangle$
99	$\mathsf{in}.tc = \langle(open, top), (open, bottom)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (stop, \bot)\rangle$
100	$\mathsf{in}.tc = \langle(\bot, bottom), (\bot, top)\rangle$
	$\mathsf{out}.tc = \langle(stop, \bot), (stop, \bot)\rangle$
101	$\mathsf{in}.tc = \langle(close, bottom), (close, top)\rangle$
	$\mathsf{out}.tc = \langle(up, \bot), (stop, \bot)\rangle$
102	$\mathsf{in}.tc = \langle(open, moving), (open, middle)\rangle$
	$\mathsf{out}.tc = \langle(stop, err), (down, \bot)\rangle$
103	$\mathsf{in}.tc = \langle(open, top), (\bot, middle), (open, moving)\rangle$
	$\mathsf{out}.tc = \langle(down, \bot), (stop, err), (stop, err)\rangle$

104	in.tc =	$\langle$(*close*, *top*), (*open*, *middle*), (⊥, *moving*)$\rangle$
	out.tc =	$\langle$(*stop*, ⊥), (*down*, ⊥), (*stop*, ⊥)$\rangle$
105	in.tc =	$\langle$(*close*, *moving*), (⊥, *top*), (*open*, *top*), (*close*, *moving*)$\rangle$
	out.tc =	$\langle$(*stop*, *err*), (*stop*, ⊥), (*down*, ⊥), (*stop*, ⊥)$\rangle$
106	in.tc =	$\langle$(*open*, *top*), (*close*, *moving*), (⊥, *middle*), (*open*, *moving*), (*open*, *moving*)$\rangle$
	out.tc =	$\langle$(*down*, ⊥), (*stop*, ⊥), (*stop*, ⊥), (*stop*, *err*), (*stop*, *err*)$\rangle$
107	in.tc =	$\langle$(⊥, *top*), (⊥, *moving*)$\rangle$
	out.tc =	$\langle$(*stop*, ⊥), (*stop*, *err*)$\rangle$
108	in.tc =	$\langle$(⊥, *bottom*), (*close*, *top*), (*open*, *middle*), (*open*, *bottom*), (*open*, *moving*), (*close*, *middle*)$\rangle$
	out.tc =	$\langle$(*stop*, ⊥), (*stop*, ⊥), (*down*, ⊥), (*stop*, ⊥), (*stop*, *err*), (*up*, ⊥)$\rangle$
109	in.tc =	$\langle$(⊥, *top*), (*close*, *middle*), (*open*, *middle*), (⊥, *moving*)$\rangle$
	out.tc =	$\langle$(*stop*, ⊥), (*up*, ⊥), (*stop*, ⊥), (*stop*, *err*)$\rangle$
110	in.tc =	$\langle$(⊥, *bottom*), (⊥, *top*), (*close*, *top*)$\rangle$
	out.tc =	$\langle$(*stop*, ⊥), (*stop*, ⊥), (*stop*, ⊥)$\rangle$
111	in.tc =	$\langle$(*open*, *middle*), (*close*, *moving*), (*open*, *moving*), (*open*, *moving*), (*open*, *middle*)$\rangle$
	out.tc =	$\langle$(*down*, ⊥), (*stop*, ⊥), (*stop*, *err*), (*stop*, *err*), (*stop*, ⊥)$\rangle$
112	in.tc =	$\langle$(⊥, *moving*), (⊥, *middle*)$\rangle$
	out.tc =	$\langle$(*stop*, *err*), (*stop*, ⊥)$\rangle$
113	in.tc =	$\langle$(*open*, *moving*), (*open*, *middle*), (*close*, *moving*)$\rangle$
	out.tc =	$\langle$(*stop*, *err*), (*down*, ⊥), (*stop*, ⊥)$\rangle$
114	in.tc =	$\langle$(*open*, *middle*), (*open*, *top*), (*open*, *top*), (⊥, *bottom*), (*open*, *top*)$\rangle$
	out.tc =	$\langle$(*down*, ⊥), (*down*, ⊥), (*down*, ⊥), (*stop*, ⊥), (*down*, ⊥)$\rangle$
115	in.tc =	$\langle$(*close*, *top*), (⊥, *middle*), (*open*, *middle*), (⊥, *bottom*), (⊥, *middle*), (*open*, *moving*)$\rangle$
	out.tc =	$\langle$(*stop*, ⊥), (*stop*, ⊥), (*down*, ⊥), (*stop*, ⊥), (*stop*, ⊥), (*stop*, *err*)$\rangle$
116	in.tc =	$\langle$(*open*, *middle*), (*open*, *moving*), (*open*, *bottom*), (⊥, *middle*)$\rangle$
	out.tc =	$\langle$(*down*, ⊥), (*down*, ⊥), (*stop*, ⊥), (*stop*, ⊥)$\rangle$
117	in.tc =	$\langle$(*open*, *moving*), (*close*, *moving*), (⊥, *bottom*), (*close*, *top*), (*close*, *top*), (*open*, *middle*)$\rangle$
	out.tc =	$\langle$(*stop*, *err*), (*stop*, *err*), (*stop*, ⊥), (*stop*, ⊥), (*stop*, ⊥), (*down*, ⊥)$\rangle$
118	in.tc =	$\langle$(*open*, *bottom*), (*open*, *bottom*), (⊥, *moving*), (*close*, *bottom*)$\rangle$
	out.tc =	$\langle$(*stop*, ⊥), (*stop*, ⊥), (*stop*, *err*), (*up*, ⊥)$\rangle$
119	in.tc =	$\langle$(⊥, *moving*), (*close*, *top*), (*open*, *middle*), (*open*, *top*), (*close*, *middle*), (*close*, *bottom*)$\rangle$
	out.tc =	$\langle$(*stop*, *err*), (*stop*, ⊥), (*down*, ⊥), (*down*, ⊥), (*stop*, ⊥), (*stop*, ⊥)$\rangle$

G. 1:1-Anforderungsüberdeckung für die Testsuiten

G.1. Testfälle aus anforderungsorientierten Teilmodellen

t	r_1	r_2	r_3	r_4'	r_4''	r_5	r_6	r_7	r_8	r_9'	r_9''	r_{10}	$\sum$
tc_2	0,50	0,30	0,30	0,15	0,19	0,33	0,17	0,10	0,10	0,10	0,10	0,11	2,45
tc_3	0,50	0,30	0,30	0,15	0,14	0,33	0,17	0,10	0,10	0,10	0,10	0,11	2,40
tc_4	0,50	0,30	0,30	0,15	0,14	0,33	0,17	0,10	0,10	0,10	0,10	0,11	2,40
tc_5	0,67	0,40	0,40	0,20	0,19	0,44	0,17	0,10	0,10	0,20	0,14	0,11	3,12
tc_6	0,50	0,40	0,30	0,15	0,14	0,33	0,17	0,20	0,10	0,10	0,10	0,11	2,60
tc_7	0,50	0,30	0,30	0,15	0,14	0,33	0,17	0,10	0,10	0,10	0,10	0,11	2,40
tc_8	1,17	0,70	0,70	0,35	0,33	1,11	0,17	0,10	0,10	0,17	0,13	0,11	5,13
tc_9	0,50	0,30	0,30	0,20	0,19	0,33	0,33	0,20	0,20	0,10	0,10	0,22	2,97
tc_{10}	0,50	0,30	0,30	0,15	0,14	0,33	0,17	0,10	0,10	0,10	0,10	0,11	2,40
tc_{11}	0,67	0,40	0,40	0,25	0,24	0,44	0,33	0,20	0,20	0,13	0,13	0,22	3,62
tc_{12}	0,50	0,30	0,30	0,20	0,19	0,33	0,33	0,20	0,20	0,10	0,10	0,22	2,97
tc_{13}	0,67	0,40	0,40	0,20	0,19	0,44	0,17	0,10	0,20	0,15	0,14	0,11	3,17
tc_{14}	0,50	0,30	0,30	0,15	0,19	0,33	0,17	0,10	0,20	0,10	0,10	0,11	2,55
tc_{15}	0,33	0,20	0,20	0,10	0,10	0,22	0,50	0,30	0,30	0,20	0,19	0,33	2,97
tc_{16}	1,00	1,00	0,60	0,30	0,29	0,67	0,17	0,20	0,10	0,10	0,10	0,11	4,63
tc_{17}	1,17	0,70	0,70	0,35	0,33	0,78	0,17	0,10	0,10	0,13	0,13	0,11	4,77
tc_{18}	1,00	0,60	1,00	0,30	0,33	0,67	0,17	0,10	0,10	0,10	0,13	0,11	4,60
tc_{19}	1,00	0,60	1,00	0,30	0,29	0,67	0,17	0,10	0,10	0,10	0,10	0,11	4,53
tc_{20}	1,00	0,60	1,00	0,30	0,29	0,67	0,17	0,10	0,10	0,10	0,10	0,11	4,53
tc_{21}	1,00	0,70	1,00	0,30	0,29	0,67	0,17	0,20	0,10	0,10	0,10	0,11	4,73
tc_{22}	1,17	0,70	1,10	0,35	0,33	0,78	0,17	0,10	0,10	0,13	0,13	0,11	5,17
tc_{23}	1,00	0,60	1,10	0,30	0,29	0,67	0,17	0,10	0,10	0,10	0,10	0,11	4,63
tc_{24}	1,00	0,60	1,00	0,30	0,29	0,67	0,17	0,10	0,10	0,10	0,10	0,11	4,53
tc_{25}	1,17	0,70	1,10	0,35	0,33	0,78	0,17	0,10	0,10	0,13	0,13	0,11	5,17
tc_{26}	1,00	0,60	1,00	0,30	0,29	0,67	0,17	0,10	0,20	0,10	0,10	0,11	4,63
tc_{27}	1,00	0,60	1,00	0,30	0,29	0,67	0,17	0,10	0,10	0,10	0,10	0,11	4,53
tc_{28}	1,00	0,60	1,00	0,35	0,33	0,67	0,33	0,20	0,20	0,10	0,10	0,22	5,10
tc_{29}	1,17	0,70	0,70	0,55	0,52	0,78	0,33	0,20	0,20	0,13	0,12	0,22	5,62
tc_{30}	1,00	0,60	0,60	0,75	0,52	0,67	0,33	0,20	0,20	0,10	0,10	0,30	5,37
tc_{31}	1,00	0,60	0,60	0,50	0,71	0,67	0,33	0,20	0,20	0,10	0,14	0,22	5,28
tc_{32}	1,00	0,70	1,00	0,35	0,33	0,67	0,33	0,30	0,20	0,10	0,10	0,22	5,30
tc_{33}	1,17	0,70	1,10	0,40	0,38	0,78	0,33	0,20	0,20	0,13	0,12	0,22	5,73
tc_{34}	1,00	0,60	1,00	0,45	0,38	0,67	0,50	0,30	0,30	0,13	0,12	0,33	5,78
tc_{35}	1,00	0,60	1,00	0,35	0,33	0,67	0,33	0,20	0,30	0,10	0,10	0,22	5,20
tc_{36}	1,00	0,60	1,00	0,27	0,25	0,67	0,67	0,40	0,40	0,15	0,14	0,44	5,99
tc_{37}	1,17	0,70	0,70	0,55	0,52	0,78	0,33	0,20	0,20	0,13	0,12	0,22	5,62

t	r_1	r_2	r_3	r_4'	r_4''	r_5	r_6	r_7	r_8	r_9'	r_9''	r_{10}	$\sum$
tc_{38}	1,00	0,60	0,60	0,50	0,48	0,67	0,33	0,20	0,20	0,10	0,10	0,22	4,99
tc_{39}	1,00	0,60	0,60	0,50	0,48	0,67	0,33	0,20	0,20	0,10	0,10	0,22	4,99
tc_{40}	1,00	0,70	0,60	0,50	0,48	0,67	0,33	0,40	0,20	0,10	0,10	0,22	5,29
tc_{41}	1,00	0,60	0,70	0,50	0,48	0,67	0,33	0,20	0,20	0,10	0,10	0,22	5,09
tc_{42}	1,17	0,70	0,70	0,55	0,52	0,78	0,33	0,20	0,20	0,20	0,19	0,22	5,76
tc_{43}	1,00	0,60	0,60	0,50	0,48	0,67	0,33	0,20	0,40	0,10	0,10	0,22	5,19
tc_{44}	1,00	0,60	0,60	0,50	0,48	0,67	0,33	0,20	0,20	0,10	0,10	0,22	4,99
tc_{45}	1,00	0,60	0,60	0,55	0,52	0,67	0,50	0,30	0,30	0,13	0,12	0,33	5,62
tc_{46}	1,17	0,70	0,70	0,60	0,57	0,78	0,33	0,20	0,20	0,12	0,11	0,22	5,71
tc_{47}	1,17	0,70	0,70	0,35	0,38	1,11	0,17	0,10	0,10	0,17	0,16	0,11	5,21
tc_{48}	1,17	0,70	0,70	0,35	0,33	1,11	0,17	0,10	0,10	0,17	0,13	0,11	5,13
tc_{49}	1,17	0,80	0,70	0,35	0,33	1,11	0,17	0,20	0,10	0,17	0,13	0,11	5,33
tc_{50}	1,17	0,70	0,80	0,35	0,33	1,11	0,17	0,10	0,10	0,17	0,13	0,11	5,23
tc_{51}	1,33	0,80	0,80	0,40	0,38	1,22	0,17	0,10	0,10	0,40	0,16	0,11	5,97
tc_{52}	1,17	0,70	0,70	0,35	0,33	1,11	0,17	0,10	0,10	0,17	0,13	0,11	5,13
tc_{53}	1,50	0,90	0,90	0,45	0,43	1,33	0,17	0,10	0,10	0,20	0,16	0,11	6,35
tc_{54}	1,17	0,70	0,70	0,35	0,33	1,11	0,17	0,10	0,20	0,17	0,13	0,11	5,23
tc_{55}	1,17	0,70	0,70	0,35	0,33	1,11	0,17	0,10	0,10	0,17	0,13	0,11	5,13
tc_{56}	1,17	0,70	0,70	0,40	0,38	1,11	0,33	0,20	0,20	0,15	0,12	0,22	5,68
tc_{57}	1,33	0,80	0,80	0,45	0,43	1,22	0,33	0,20	0,20	0,18	0,14	0,22	6,31
tc_{58}	1,17	0,70	0,70	0,40	0,38	1,11	0,33	0,20	0,20	0,15	0,12	0,22	5,68
tc_{59}	1,17	0,70	0,70	0,40	0,38	1,11	0,33	0,20	0,20	0,15	0,12	0,22	5,68
tc_{60}	1,17	0,80	0,70	0,40	0,38	1,11	0,33	0,30	0,20	0,15	0,12	0,22	5,88
tc_{61}	1,17	0,70	0,80	0,40	0,38	1,11	0,33	0,20	0,20	0,15	0,12	0,22	5,78
tc_{62}	1,17	0,70	0,70	0,40	0,38	1,11	0,33	0,20	0,20	0,15	0,12	0,22	5,68
tc_{63}	1,33	0,80	0,80	0,45	0,43	1,22	0,33	0,20	0,20	0,18	0,14	0,22	6,31
tc_{64}	1,17	0,70	0,70	0,50	0,43	1,11	0,50	0,30	0,30	0,18	0,14	0,33	6,36
tc_{65}	1,17	0,70	0,70	0,40	0,38	1,11	0,33	0,20	0,30	0,15	0,12	0,22	5,78
tc_{66}	1,17	0,70	0,70	0,40	0,38	1,11	0,33	0,20	0,20	0,15	0,12	0,22	5,68
tc_{67}	1,17	0,70	0,70	0,30	0,29	1,11	0,67	0,40	0,40	0,00	0,17	0,44	6,34
tc_{68}	1,17	0,70	0,70	0,40	0,29	1,11	0,33	0,20	0,20	0,15	0,17	0,22	5,64
tc_{69}	0,50	0,30	0,30	0,30	0,24	0,33	0,33	0,20	0,20	0,10	0,10	0,30	3,20
tc_{70}	0,50	0,30	0,30	0,20	0,29	0,33	0,33	0,20	0,20	0,10	0,13	0,22	3,10
tc_{71}	0,17	0,10	0,10	0,10	0,10	0,11	0,50	0,30	0,40	0,15	0,14	0,33	2,50
tc_{72}	0,17	0,10	0,10	0,10	0,10	0,11	0,50	0,30	0,30	0,15	0,14	0,33	2,40
tc_{73}	0,17	0,10	0,10	0,10	0,10	0,11	0,50	0,30	0,30	0,15	0,14	0,33	2,40
tc_{74}	0,17	0,20	0,10	0,10	0,10	0,11	0,50	0,40	0,30	0,15	0,14	0,33	2,60
tc_{75}	0,33	0,20	0,20	0,10	0,10	0,22	0,50	0,30	0,30	0,20	0,19	0,33	2,97
tc_{76}	0,17	0,10	0,10	0,10	0,10	0,11	0,50	0,30	0,30	0,15	0,14	0,33	2,40
tc_{77}	0,17	0,10	0,20	0,10	0,10	0,11	0,50	0,30	0,30	0,15	0,14	0,33	2,50
tc_{78}	0,17	0,10	0,10	0,20	0,14	0,11	0,67	0,40	0,40	0,20	0,19	0,44	3,12
tc_{79}	0,17	0,10	0,10	0,10	0,10	0,11	0,50	0,30	0,30	0,15	0,14	0,33	2,40
tc_{80}	0,33	0,20	0,20	0,18	0,14	0,22	1,33	0,80	0,80	0,45	0,43	1,22	6,31
tc_{81}	0,17	0,10	0,10	0,10	0,10	0,11	0,50	0,30	0,30	0,15	0,19	0,33	2,45
tc_{82}	0,33	0,20	0,30	0,15	0,12	0,22	1,17	0,70	0,70	0,40	0,38	1,11	5,78
tc_{83}	0,50	0,30	0,30	0,18	0,14	0,33	1,17	0,70	0,70	0,50	0,43	1,11	6,36
tc_{84}	0,33	0,30	0,20	0,15	0,12	0,22	1,17	0,80	0,70	0,40	0,38	1,11	5,88
tc_{85}	0,67	0,40	0,40	0,00	0,17	0,44	1,17	0,70	0,70	0,30	0,29	1,11	6,34
tc_{86}	0,17	0,10	0,10	0,17	0,13	0,11	1,17	0,70	0,70	0,35	0,33	1,11	5,13

t	r_1	r_2	r_3	r_4'	r_4''	r_5	r_6	r_7	r_8	r_9'	r_9''	r_{10}	$\sum$
tc_{87}	0,17	0,10	0,10	0,17	0,13	0,11	1,17	0,70	0,70	0,35	0,33	1,11	5,13
tc_{88}	0,17	0,20	0,10	0,17	0,13	0,11	1,17	0,80	0,70	0,35	0,33	1,11	5,33
tc_{89}	0,17	0,10	0,20	0,17	0,13	0,11	1,17	0,70	0,70	0,35	0,33	1,11	5,23
tc_{90}	0,17	0,10	0,10	0,17	0,13	0,11	1,17	0,70	0,70	0,35	0,33	1,11	5,13
tc_{91}	0,33	0,20	0,20	0,15	0,12	0,22	1,17	0,70	0,70	0,40	0,38	1,11	5,68
tc_{92}	0,17	0,10	0,10	0,40	0,16	0,11	1,33	0,80	0,80	0,40	0,38	1,22	5,97
tc_{93}	0,17	0,10	0,10	0,17	0,13	0,11	1,17	0,70	0,80	0,35	0,33	1,11	5,23
tc_{94}	0,17	0,10	0,10	0,17	0,13	0,11	1,17	0,70	0,70	0,35	0,33	1,11	5,13
tc_{95}	0,17	0,10	0,10	0,20	0,16	0,11	1,50	0,90	0,90	0,45	0,43	1,33	6,35
tc_{96}	0,17	0,10	0,10	0,17	0,16	0,11	1,17	0,70	0,70	0,35	0,38	1,11	5,21
tc_{97}	0,33	0,20	0,20	0,10	0,14	0,22	1,00	0,60	0,60	0,50	0,71	0,67	5,28
tc_{98}	0,33	0,20	0,20	0,10	0,10	0,30	1,00	0,60	0,60	0,75	0,52	0,67	5,37
tc_{99}	0,33	0,20	0,20	0,13	0,12	0,22	1,17	0,70	0,70	0,55	0,52	0,78	5,62
tc_{100}	0,17	0,10	0,10	0,10	0,10	0,11	1,00	0,60	1,00	0,30	0,29	0,67	4,53
tc_{101}	0,17	0,10	0,10	0,13	0,13	0,11	1,17	0,70	0,70	0,35	0,33	0,78	4,77
tc_{102}	0,17	0,20	0,10	0,10	0,10	0,11	1,00	1,00	0,60	0,30	0,29	0,67	4,63
tc_{103}	0,33	0,20	0,20	0,12	0,11	0,22	1,17	0,70	0,70	0,60	0,57	0,78	5,71
tc_{104}	0,33	0,20	0,20	0,13	0,12	0,22	1,17	0,70	0,70	0,55	0,52	0,78	5,62
tc_{105}	0,33	0,20	0,20	0,10	0,10	0,22	1,00	0,60	0,60	0,50	0,48	0,67	4,99
tc_{106}	0,33	0,20	0,20	0,10	0,10	0,22	1,00	0,60	0,60	0,50	0,48	0,67	4,99
tc_{107}	0,33	0,40	0,20	0,10	0,10	0,22	1,00	0,70	0,60	0,50	0,48	0,67	5,29
tc_{108}	0,33	0,20	0,20	0,20	0,19	0,22	1,17	0,70	0,70	0,55	0,52	0,78	5,76
tc_{109}	0,33	0,20	0,40	0,10	0,10	0,22	1,00	0,60	0,60	0,50	0,48	0,67	5,19
tc_{110}	0,50	0,30	0,30	0,13	0,12	0,33	1,00	0,60	0,60	0,55	0,52	0,67	5,62
tc_{111}	0,33	0,20	0,20	0,10	0,10	0,22	1,00	0,60	0,70	0,50	0,48	0,67	5,09
tc_{112}	0,33	0,20	0,20	0,10	0,10	0,22	1,00	0,60	0,60	0,50	0,48	0,67	4,99
tc_{113}	0,33	0,20	0,20	0,15	0,12	0,22	1,17	0,70	0,80	0,40	0,38	1,11	5,78
tc_{114}	0,33	0,20	0,20	0,15	0,17	0,22	1,17	0,70	0,70	0,40	0,29	1,11	5,64
tc_{115}	0,33	0,20	0,20	0,15	0,12	0,22	1,17	0,70	0,70	0,40	0,38	1,11	5,68
tc_{116}	0,33	0,20	0,20	0,15	0,12	0,22	1,17	0,70	0,70	0,40	0,38	1,11	5,68
tc_{117}	0,33	0,20	0,20	0,15	0,12	0,22	1,17	0,70	0,70	0,40	0,38	1,11	5,68
tc_{118}	0,33	0,20	0,20	0,18	0,14	0,22	1,33	0,80	0,80	0,45	0,43	1,22	6,31
tc_{119}	0,33	0,20	0,20	0,15	0,12	0,22	1,17	0,70	0,70	0,40	0,38	1,11	5,68
$\sum$	81,67	50,60	56,40	31,50	29,56	62,19	69,00	43,00	43,00	27,74	25,94	53,81	574,40

G.2. Testfälle aus Transtionsüberdeckung im Gesamtmodell

t	r_1	r_2	r_3	r_4'	r_4''	r_5	r_6	r_7	r_8	r_9'	r_9''	r_{10}	$\sum$
tc_1	0,50	0,30	0,30	0,15	0,19	0,33	0,17	0,10	0,10	0,10	0,10	0,11	2,45
tc_2	0,50	0,30	0,30	0,15	0,14	0,33	0,17	0,10	0,10	0,10	0,10	0,11	2,40
tc_3	0,50	0,30	0,30	0,15	0,14	0,33	0,17	0,10	0,10	0,10	0,10	0,11	2,40
tc_4	0,50	0,30	0,30	0,15	0,14	0,33	0,17	0,10	0,20	0,10	0,10	0,11	2,50
tc_5	0,67	0,40	0,40	0,20	0,19	0,44	0,17	0,10	0,10	0,20	0,14	0,11	3,12
tc_6	0,50	0,30	0,40	0,15	0,14	0,33	0,17	0,10	0,10	0,10	0,10	0,11	2,50
tc_7	0,50	0,40	0,30	0,15	0,14	0,33	0,17	0,20	0,10	0,10	0,10	0,11	2,60
tc_8	0,50	0,30	0,30	0,15	0,14	0,33	0,17	0,10	0,10	0,10	0,10	0,11	2,40
tc_9	1,00	0,60	0,60	0,30	0,29	0,67	0,17	0,10	0,10	0,10	0,10	0,11	4,13

t	r_1	r_2	r_3	r_4'	r_4''	r_5	r_6	r_7	r_8	r_9'	r_9''	r_{10}	$\sum$
tc_{10}	0,50	0,30	0,30	0,20	0,19	0,33	0,33	0,20	0,20	0,10	0,10	0,22	2,97
tc_{11}	0,50	0,30	0,30	0,15	0,14	0,33	0,17	0,10	0,10	0,10	0,10	0,11	2,40
tc_{12}	0,17	0,10	0,20	0,05	0,05	0,11	0,17	0,10	0,10	0,05	0,05	0,11	1,25
tc_{13}	0,17	0,10	0,10	0,05	0,05	0,11	0,17	0,10	0,10	0,05	0,05	0,11	1,15
tc_{14}	0,17	0,10	0,10	0,05	0,05	0,11	0,17	0,10	0,10	0,05	0,05	0,11	1,15
tc_{15}	0,17	0,20	0,10	0,05	0,05	0,11	0,17	0,20	0,10	0,05	0,05	0,11	1,35
tc_{16}	0,17	0,10	0,10	0,05	0,05	0,11	0,17	0,10	0,10	0,05	0,05	0,11	1,15
tc_{17}	0,33	0,20	0,20	0,10	0,10	0,22	0,17	0,10	0,10	0,10	0,10	0,11	1,82
tc_{18}	0,17	0,10	0,10	0,05	0,05	0,11	0,17	0,10	0,20	0,05	0,05	0,11	1,25
tc_{19}	0,17	0,10	0,10	0,05	0,05	0,11	0,17	0,10	0,10	0,05	0,05	0,11	1,15
tc_{20}	0,17	0,10	0,10	0,10	0,10	0,11	0,33	0,20	0,20	0,10	0,10	0,22	1,82
tc_{21}	0,17	0,10	0,10	0,05	0,10	0,11	0,17	0,10	0,10	0,05	0,10	0,11	1,25
tc_{22}	0,67	0,40	0,40	0,25	0,24	0,44	0,33	0,20	0,20	0,13	0,13	0,22	3,62
tc_{23}	0,50	0,30	0,30	0,20	0,29	0,33	0,33	0,20	0,20	0,10	0,13	0,22	3,10
tc_{24}	0,50	0,30	0,30	0,20	0,19	0,33	0,33	0,20	0,20	0,10	0,10	0,22	2,97
tc_{25}	0,50	0,40	0,30	0,20	0,19	0,33	0,33	0,40	0,20	0,10	0,10	0,22	3,27
tc_{26}	0,50	0,30	0,30	0,20	0,19	0,33	0,33	0,20	0,20	0,10	0,10	0,22	2,97
tc_{27}	0,50	0,30	0,40	0,20	0,19	0,33	0,33	0,20	0,20	0,10	0,10	0,22	3,07
tc_{28}	0,67	0,40	0,40	0,17	0,16	0,44	0,33	0,20	0,20	0,17	0,16	0,22	3,52
tc_{29}	0,50	0,30	0,30	0,30	0,24	0,33	0,33	0,20	0,20	0,10	0,10	0,30	3,20
tc_{30}	0,50	0,30	0,30	0,20	0,19	0,33	0,33	0,20	0,40	0,10	0,10	0,22	3,17
tc_{31}	0,50	0,30	0,30	0,20	0,19	0,33	0,33	0,20	0,20	0,10	0,10	0,22	2,97
tc_{32}	0,50	0,30	0,30	0,25	0,24	0,33	0,50	0,30	0,30	0,13	0,13	0,33	3,62
tc_{33}	0,50	0,30	0,30	0,20	0,19	0,33	0,33	0,20	0,20	0,10	0,10	0,22	2,97
tc_{34}	0,17	0,10	0,10	0,10	0,10	0,11	1,00	0,60	0,60	0,30	0,29	0,67	4,13
tc_{35}	0,17	0,10	0,10	0,10	0,10	0,11	0,50	0,30	0,30	0,15	0,14	0,33	2,40
tc_{36}	0,17	0,10	0,10	0,10	0,10	0,11	0,50	0,30	0,30	0,15	0,14	0,33	2,40
tc_{37}	0,17	0,20	0,10	0,10	0,10	0,11	0,50	0,40	0,30	0,15	0,14	0,33	2,60
tc_{38}	0,33	0,20	0,20	0,10	0,10	0,22	0,50	0,30	0,30	0,20	0,19	0,33	2,97
tc_{39}	0,17	0,10	0,10	0,10	0,10	0,11	0,50	0,30	0,30	0,15	0,14	0,33	2,40
tc_{40}	0,17	0,10	0,20	0,10	0,10	0,11	0,50	0,30	0,30	0,15	0,14	0,33	2,50
tc_{41}	0,33	0,20	0,20	0,10	0,10	0,22	0,50	0,30	0,30	0,20	0,19	0,33	2,97
tc_{42}	0,17	0,10	0,10	0,20	0,14	0,11	0,67	0,40	0,40	0,20	0,19	0,44	3,12
tc_{43}	0,17	0,10	0,10	0,10	0,10	0,11	0,50	0,30	0,40	0,15	0,14	0,33	2,50
tc_{44}	0,17	0,10	0,10	0,10	0,10	0,11	0,50	0,30	0,30	0,15	0,14	0,33	2,40
tc_{45}	0,17	0,10	0,10	0,10	0,10	0,11	0,50	0,30	0,30	0,15	0,19	0,33	2,45
$\sum$	16,67	10,40	10,40	6,37	6,16	11,11	14,67	9,30	9,30	5,23	5,06	9,85	114,52

G.3. Testfälle mit randomisierten Testeingaben

t	r_1	r_2	r_3	r_4'	r_4''	r_5	r_6	r_7	r_8	r_9'	r_9''	r_{10}	$\sum$
tc_1	0,33	0,20	0,20	0,13	0,13	0,22	0,67	0,40	0,40	0,17	0,16	0,44	3,45
tc_2	0,67	0,40	0,40	0,17	0,16	0,44	0,33	0,20	0,20	0,17	0,16	0,22	3,52
tc_3	0,33	0,20	0,20	0,10	0,10	0,22	0,33	0,20	0,20	0,15	0,14	0,22	2,40
tc_4	0,67	0,40	0,40	0,20	0,19	0,44	0,67	0,40	0,40	0,40	0,33	0,44	4,95
tc_5	0,17	0,10	0,20	0,05	0,05	0,11	0,17	0,10	0,20	0,05	0,05	0,11	1,35
tc_6	0,17	0,10	0,10	0,10	0,10	0,11	0,50	0,30	0,40	0,15	0,14	0,33	2,50

t	r_1	r_2	r_3	r_4'	r_4''	r_5	r_6	r_7	r_8	r_9'	r_9''	r_{10}	$\sum$
tc_7	0,17	0,20	0,20	0,05	0,05	0,11	0,17	0,20	0,10	0,05	0,05	0,11	1,45
tc_8	0,33	0,20	0,20	0,20	0,16	0,22	0,83	0,50	0,50	0,30	0,29	0,56	4,29
tc_9	0,17	0,20	0,20	0,05	0,05	0,11	0,17	0,20	0,10	0,05	0,05	0,11	1,45
tc_{10}	0,17	0,10	0,10	0,10	0,10	0,11	0,50	0,30	0,30	0,15	0,19	0,33	2,45
tc_{11}	0,50	0,40	0,30	0,15	0,14	0,33	0,17	0,20	0,30	0,10	0,10	0,11	2,80
tc_{12}	0,33	0,20	0,20	0,10	0,10	0,22	0,17	0,10	0,10	0,10	0,10	0,11	1,82
tc_{13}	0,17	0,20	0,10	0,10	0,10	0,11	0,33	0,30	0,20	0,10	0,10	0,22	2,02
tc_{14}	0,17	0,10	0,10	0,05	0,05	0,11	0,17	0,10	0,20	0,05	0,05	0,11	1,25
tc_{15}	0,17	0,10	0,10	0,05	0,05	0,11	0,17	0,10	0,10	0,05	0,05	0,11	1,15
tc_{16}	0,33	0,20	0,20	0,10	0,10	0,22	0,33	0,20	0,30	0,20	0,19	0,22	2,60
tc_{17}	0,17	0,10	0,10	0,10	0,10	0,11	0,33	0,20	0,20	0,10	0,10	0,22	1,82
tc_{18}	0,17	0,10	0,10	0,05	0,10	0,11	0,17	0,10	0,10	0,05	0,10	0,11	1,25
tc_{19}	0,17	0,20	0,10	0,25	0,19	0,11	0,44	0,40	0,27	0,13	0,13	0,37	2,76
tc_{20}	0,17	0,10	0,10	0,05	0,05	0,11	0,17	0,10	0,10	0,05	0,05	0,11	1,15
tc_{21}	0,17	0,10	0,10	0,05	0,05	0,11	0,17	0,10	0,20	0,05	0,05	0,11	1,25
tc_{22}	0,50	0,30	0,40	0,15	0,14	0,33	0,17	0,10	0,10	0,15	0,14	0,11	2,60
tc_{23}	0,17	0,10	0,20	0,10	0,10	0,11	0,33	0,20	0,20	0,10	0,10	0,22	1,92
tc_{24}	0,33	0,20	0,20	0,15	0,14	0,22	0,33	0,20	0,20	0,10	0,10	0,22	2,40
tc_{25}	0,33	0,20	0,20	0,10	0,10	0,22	0,17	0,10	0,20	0,10	0,10	0,11	1,92
tc_{26}	0,33	0,20	0,20	0,10	0,10	0,22	0,17	0,10	0,10	0,10	0,10	0,11	1,82
tc_{27}	0,17	0,10	0,10	0,10	0,10	0,11	0,33	0,20	0,20	0,10	0,10	0,22	1,82
tc_{28}	0,17	0,20	0,10	0,10	0,14	0,11	0,33	0,40	0,20	0,10	0,14	0,22	2,22
tc_{29}	0,33	0,20	0,50	0,10	0,10	0,22	0,17	0,10	0,10	0,10	0,10	0,11	2,12
tc_{30}	0,17	0,10	0,10	0,10	0,16	0,11	0,67	0,40	0,40	0,20	0,16	0,44	3,01
tc_{31}	0,50	0,30	0,30	0,30	0,24	0,33	0,33	0,20	0,20	0,10	0,10	0,30	3,20
tc_{32}	0,17	0,30	0,10	0,10	0,10	0,11	0,33	0,50	0,30	0,10	0,10	0,22	2,42
tc_{33}	0,50	0,40	0,30	0,15	0,19	0,33	0,17	0,20	0,10	0,10	0,10	0,11	2,65
tc_{34}	0,17	0,10	0,20	0,05	0,05	0,11	0,17	0,10	0,10	0,05	0,05	0,11	1,25
tc_{35}	0,33	0,20	0,20	0,10	0,10	0,22	0,17	0,10	0,20	0,10	0,14	0,11	1,97
tc_{36}	0,17	0,10	0,10	0,15	0,24	0,11	0,33	0,20	0,20	0,10	0,13	0,22	2,05
tc_{37}	0,33	0,20	0,20	0,10	0,10	0,22	0,17	0,10	0,10	0,10	0,10	0,11	1,82
tc_{38}	0,17	0,20	0,20	0,15	0,14	0,11	0,33	0,20	0,20	0,10	0,10	0,22	2,12
tc_{39}	0,17	0,10	0,10	0,10	0,10	0,11	0,50	0,30	0,30	0,15	0,14	0,33	2,40
tc_{40}	0,17	0,10	0,10	0,10	0,10	0,11	1,00	0,60	0,70	0,30	0,29	0,67	4,23
tc_{41}	1,00	0,60	1,00	0,35	0,33	0,67	0,33	0,20	0,20	0,10	0,10	0,22	5,10
tc_{42}	0,33	0,20	0,20	0,10	0,10	0,30	0,17	0,10	0,10	0,20	0,14	0,11	2,05
tc_{43}	0,67	0,40	0,40	0,15	0,14	0,44	0,67	0,40	0,40	0,40	0,22	0,44	4,74
tc_{44}	0,17	0,10	0,10	0,10	0,10	0,11	0,33	0,20	0,20	0,10	0,10	0,22	1,82
tc_{45}	0,17	0,20	0,10	0,05	0,10	0,11	0,17	0,20	0,10	0,05	0,10	0,11	1,45
tc_{46}	0,17	0,10	0,10	0,05	0,05	0,11	0,17	0,10	0,20	0,05	0,05	0,11	1,25
tc_{47}	0,17	0,10	0,30	0,10	0,10	0,11	0,33	0,20	0,27	0,10	0,10	0,22	2,09
tc_{48}	0,17	0,30	0,10	0,05	0,10	0,11	0,17	0,30	0,30	0,05	0,10	0,11	1,85
tc_{49}	0,33	0,20	0,20	0,10	0,10	0,22	0,17	0,10	0,10	0,10	0,10	0,11	1,82
tc_{50}	0,17	0,10	0,10	0,05	0,05	0,11	0,17	0,10	0,10	0,05	0,05	0,11	1,15
tc_{51}	0,33	0,40	0,20	0,10	0,19	0,22	0,17	0,30	0,10	0,10	0,19	0,11	2,41
tc_{52}	0,17	0,10	0,10	0,10	0,10	0,11	0,33	0,20	0,20	0,10	0,10	0,22	1,82
tc_{53}	0,33	0,20	0,33	0,10	0,10	0,22	0,17	0,10	0,10	0,10	0,10	0,11	1,96
tc_{54}	0,17	0,10	0,10	0,05	0,05	0,11	0,17	0,10	0,20	0,05	0,05	0,11	1,25
tc_{55}	0,33	0,20	0,20	0,10	0,10	0,22	0,17	0,10	0,10	0,10	0,10	0,11	1,82

t	r_1	r_2	r_3	r_4'	r_4''	r_5	r_6	r_7	r_8	r_9'	r_9''	r_{10}	$\sum$
tc_{56}	0,67	0,40	0,70	0,20	0,19	0,44	0,17	0,10	0,10	0,10	0,10	0,11	3,27
tc_{57}	0,83	0,50	0,47	0,20	0,19	0,56	0,33	0,20	0,20	0,17	0,16	0,22	4,03
tc_{58}	0,17	0,20	0,10	0,10	0,19	0,11	0,33	0,30	0,20	0,10	0,10	0,22	2,12
tc_{59}	0,50	0,30	0,30	0,15	0,13	0,33	0,17	0,10	0,40	0,15	0,13	0,11	2,77
tc_{60}	0,33	0,20	0,20	0,10	0,10	0,22	0,33	0,20	0,20	0,20	0,19	0,22	2,50
tc_{61}	0,67	0,40	0,60	0,20	0,19	0,44	0,17	0,10	0,10	0,15	0,14	0,11	3,27
tc_{62}	0,33	0,20	0,20	0,10	0,12	0,22	0,67	0,40	0,40	0,25	0,24	0,44	3,57
tc_{63}	0,33	0,20	0,20	0,10	0,10	0,22	0,33	0,20	0,20	0,15	0,14	0,22	2,40
tc_{64}	0,17	0,20	0,10	0,10	0,14	0,11	0,33	0,40	0,20	0,10	0,14	0,22	2,22
tc_{65}	0,17	0,10	0,30	0,10	0,10	0,11	0,33	0,20	0,20	0,10	0,10	0,22	2,02
tc_{66}	0,17	0,10	0,10	0,05	0,05	0,11	0,17	0,10	0,10	0,05	0,05	0,11	1,15
tc_{67}	0,50	0,30	0,40	0,20	0,19	0,33	0,33	0,20	0,20	0,13	0,13	0,22	3,14
tc_{68}	0,67	0,40	0,40	0,25	0,38	0,44	0,33	0,20	0,20	0,13	0,13	0,22	3,76
tc_{69}	0,17	0,30	0,10	0,05	0,05	0,11	0,17	0,30	0,20	0,05	0,05	0,11	1,65
tc_{70}	0,17	0,10	0,10	0,05	0,14	0,11	0,17	0,10	0,10	0,05	0,14	0,11	1,34
tc_{71}	0,33	0,20	0,20	0,10	0,10	0,22	0,17	0,10	0,10	0,10	0,10	0,11	1,82
tc_{72}	0,17	0,10	0,10	0,10	0,10	0,11	0,33	0,20	0,20	0,10	0,10	0,22	1,82
tc_{73}	0,33	0,20	0,20	0,10	0,10	0,22	0,17	0,10	0,10	0,10	0,10	0,11	1,82
tc_{74}	0,17	0,10	0,10	0,10	0,10	0,11	0,33	0,20	0,20	0,10	0,10	0,22	1,82
tc_{75}	0,17	0,10	0,20	0,05	0,05	0,11	0,17	0,10	0,10	0,05	0,05	0,11	1,25
tc_{76}	0,50	0,30	0,27	0,15	0,14	0,33	0,17	0,10	0,10	0,15	0,14	0,11	2,46
tc_{77}	0,33	0,20	0,20	0,10	0,10	0,22	0,50	0,30	0,30	0,20	0,19	0,33	2,97
tc_{78}	0,17	0,10	0,20	0,05	0,05	0,11	0,17	0,10	0,10	0,05	0,05	0,11	1,25
tc_{79}	0,17	0,10	0,10	0,10	0,10	0,11	0,33	0,20	0,20	0,10	0,10	0,22	1,82
tc_{80}	0,33	0,20	0,20	0,10	0,10	0,22	0,17	0,10	0,10	0,10	0,10	0,11	1,82
tc_{81}	0,17	0,10	0,10	0,10	0,10	0,11	1,00	0,60	0,60	0,30	0,29	0,67	4,13
tc_{82}	0,50	0,30	0,50	0,15	0,19	0,33	0,17	0,10	0,10	0,15	0,24	0,11	2,84
tc_{83}	0,50	0,30	0,30	0,15	0,13	0,33	0,17	0,10	0,20	0,15	0,13	0,11	2,57
tc_{84}	0,17	0,10	0,20	0,05	0,10	0,11	0,17	0,10	0,10	0,05	0,10	0,11	1,35
tc_{85}	0,50	0,30	0,30	0,15	0,14	0,33	0,17	0,10	0,10	0,15	0,14	0,11	2,50
tc_{86}	0,17	0,10	0,20	0,05	0,10	0,11	0,17	0,10	0,20	0,05	0,10	0,11	1,45
tc_{87}	0,17	0,10	0,20	0,05	0,05	0,11	0,17	0,10	0,10	0,05	0,05	0,11	1,25
tc_{88}	0,33	0,20	0,20	0,10	0,10	0,22	0,50	0,30	0,30	0,20	0,19	0,33	2,97
tc_{89}	0,67	0,40	0,40	0,20	0,16	0,44	0,17	0,10	0,10	0,10	0,10	0,11	2,94
tc_{90}	0,17	0,10	0,30	0,20	0,14	0,11	0,33	0,20	0,20	0,10	0,10	0,30	2,25
tc_{91}	0,33	0,20	0,20	0,15	0,14	0,22	0,33	0,20	0,20	0,10	0,10	0,22	2,40
tc_{92}	0,17	0,10	0,20	0,10	0,10	0,11	0,33	0,20	0,20	0,10	0,10	0,22	1,92
tc_{93}	0,33	0,20	0,20	0,15	0,14	0,22	0,33	0,20	0,20	0,10	0,10	0,22	2,40
tc_{94}	0,17	0,10	0,20	0,05	0,10	0,11	0,17	0,10	0,10	0,05	0,10	0,11	1,35
tc_{95}	0,17	0,10	0,10	0,05	0,10	0,11	0,17	0,10	0,10	0,05	0,10	0,11	1,25
tc_{96}	0,17	0,10	0,10	0,10	0,10	0,11	0,33	0,20	0,20	0,10	0,10	0,22	1,82
tc_{97}	0,33	0,20	0,30	0,15	0,14	0,22	0,33	0,20	0,50	0,10	0,10	0,22	2,80
tc_{98}	0,50	0,30	0,40	0,30	0,24	0,33	0,33	0,20	0,20	0,10	0,10	0,30	3,30
tc_{99}	0,17	0,10	0,20	0,05	0,05	0,11	0,17	0,10	0,10	0,05	0,05	0,11	1,25
tc_{100}	0,17	0,10	0,10	0,05	0,05	0,11	0,17	0,10	0,10	0,05	0,05	0,11	1,15
tc_{101}	0,17	0,10	0,10	0,05	0,05	0,11	0,17	0,10	0,20	0,05	0,05	0,11	1,25
tc_{102}	0,33	0,20	0,20	0,10	0,10	0,30	0,17	0,10	0,10	0,20	0,14	0,11	2,05
tc_{103}	0,33	0,20	0,20	0,10	0,10	0,22	0,17	0,10	0,10	0,10	0,10	0,11	1,82
tc_{104}	0,33	0,30	0,20	0,10	0,10	0,22	0,17	0,20	0,20	0,10	0,10	0,11	2,12

t	r_1	r_2	r_3	r_4'	r_4''	r_5	r_6	r_7	r_8	r_9'	r_9''	r_{10}	$\sum$
tc_{105}	0,17	0,10	0,10	0,15	0,14	0,11	0,50	0,30	0,30	0,15	0,14	0,33	2,50
tc_{106}	0,50	0,30	0,30	0,13	0,19	0,33	0,33	0,20	0,20	0,20	0,20	0,22	3,11
tc_{107}	0,17	0,20	0,10	0,05	0,05	0,11	0,17	0,20	0,10	0,05	0,05	0,11	1,35
tc_{108}	0,50	0,30	0,33	0,20	0,19	0,33	0,33	0,20	0,20	0,10	0,10	0,22	3,01
tc_{109}	0,33	0,30	0,20	0,10	0,10	0,22	0,33	0,30	0,20	0,15	0,14	0,22	2,60
tc_{110}	0,17	0,10	0,10	0,05	0,05	0,11	0,17	0,10	0,20	0,05	0,05	0,11	1,25
tc_{111}	1,00	0,60	0,60	0,23	0,22	0,78	0,33	0,20	0,20	0,27	0,22	0,22	4,88
tc_{112}	0,17	0,20	0,10	0,05	0,10	0,11	0,17	0,20	0,10	0,05	0,10	0,11	1,45
tc_{113}	0,33	0,20	0,20	0,17	0,16	0,30	0,33	0,20	0,20	0,17	0,13	0,22	2,60
tc_{114}	0,50	0,30	0,30	0,15	0,14	0,33	0,17	0,10	0,10	0,10	0,10	0,11	2,40
tc_{115}	0,50	0,30	0,30	0,15	0,16	0,33	0,17	0,10	0,20	0,10	0,19	0,11	2,61
tc_{116}	1,00	0,60	1,00	0,30	0,33	0,67	0,17	0,10	0,10	0,10	0,13	0,11	4,60
tc_{117}	0,33	0,20	0,20	0,17	0,16	0,22	0,33	0,20	0,40	0,17	0,16	0,22	2,76
tc_{118}	0,17	0,20	0,30	0,05	0,05	0,11	0,17	0,20	0,10	0,05	0,05	0,11	1,55
tc_{119}	0,33	0,20	0,20	0,15	0,14	0,22	0,33	0,20	0,20	0,10	0,10	0,22	2,40
$\sum$	37,17	24,70	26,80	13,75	14,12	25,11	33,94	23,00	23,43	13,79	13,74	22,93	272,49

H. Mathematische Symbole und Operatoren

M^∞	Menge der unendlichen Ströme über M
M^*	Menge der endlichen Ströme über M
M^ω	$M^\infty \cup M^*$
$\langle\rangle$	leerer Strom
$\langle m_1, m_2, \ldots, m_n\rangle$	Strom aus m_1 gefolgt von $m_2, \ldots, m_n$
$\#s$	Länge eines Stroms
$s \sqsubseteq r$	s ist Präfix von r
$A \circledS s$	filtert aus einem Strom s nur die Aktionen in A
$s \frown r$	Konkatenation von s und r
$\mathsf{ft}(s)$	erstes Element von s
$\mathsf{rt}(s)$	s ohne sein erstes Element
$s\vert_j$	Präfix von s der Länge j
$s.j$	j-tes Element in s
$\mathsf{dom}.s$	$[1 \ldots \#s]$
$\mathsf{rng}.s$	$\{s.j \vert j \in \mathsf{dom}.s]$
$A_1 \subsetneq A_2$	echte Teilmenge: $A_1 \subseteq A_2 \wedge A_1 \neq A_2$