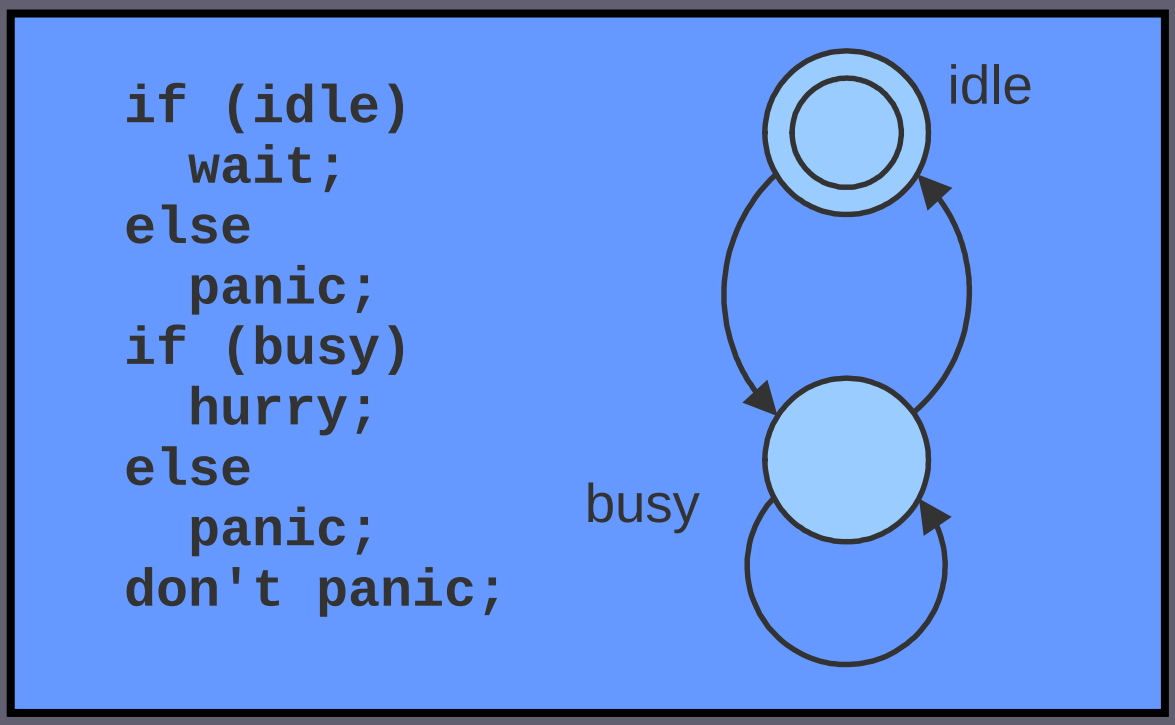
Modellbasierter Modultest technischer Softwaresysteme



Christopher Robinson-Mallett

Hasso-Plattner-Institut für Software-Systemtechnik
Softwaretechnik und Qualitätsmanagement

Modellbasierte Modulprüfung für die Entwicklung technischer, softwareintensiver Systeme

mit Real-Time Object-Oriented Modeling

Dissertation
zur Erlangung des akademischen Grades
"Doktor der Ingenieurwissenschaften"
(Dr.-Ing.)
in der Wissenschaftsdisziplin "Softwaretechnik"

von der
Mathematisch-Naturwissenschaftlichen Fakultät
der Universität Potsdam
genehmigte Dissertation

von
Christopher Robinson-Mallett

Tag der Aussprache: 28. Oktober 2005

Potsdam, im Oktober 2005

Danksagung

Die vorliegende Arbeit entstand im Rahmen meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Fachgebiet für Softwaretechnik und Qualitätsmanagement des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam. Sie bildet einen Baustein in dem Teilthema *Implementierung und Modulprüfung sicherer Software* der Kooperation *Risikoanalysetechniken für hybride Systeme* zwischen dem Lehrstuhl von Prof. Dr. Peter Liggesmeyer und der Siemens AG, München, Abteilung Corporate Technology (CT PP 2).

An erster Stelle möchte ich Prof. Dr. Peter Liggesmeyer für die Möglichkeit zur Anfertigung dieser Arbeit danken. In der Rolle des "Doktorvaters" war mir Prof. Dr. Liggesmeyer stets ein kompetenter und kritischer Ratgeber und Diskussionspartner. Insbesondere möchte ich mich bei Prof. Dr. Liggesmeyer für den großen Vertrauensvorschub und den von ihm gebotenen Freiraum bedanken. Beides trug wesentlich zur Ausbildung meiner Fähigkeiten zum selbstständigen Forschen und Lehren bei.

Ich möchte mich auch bei allen Mitgliedern der Prüfungskommission für ihre Anstrengungen und ihr kompetentes Urteil bedanken. Bei Prof. Dr. Dieter Rombach und Prof. Dr. Helmut Balzert bedanke ich mich für die Begutachtung meiner Dissertationsschrift. Vielen Dank an Prof. Andreas Polze für die Übernahme des Vorsitzes und die souveräne Leitung und Moderierung der Disputation. Mein Dank gilt auch Prof. Dr. Bettina Schnor, Prof. Dr. Andreas Schwill, Prof. Dr. Torsten Schaub, Prof. Dr. Mathias Weske, Prof. Dr. Jürgen Döllner und Prof. Dr. Dr. hc. Klaus Denecke für die freundliche Unterstützung als Mitglieder der Prüfungskommission.

Bei Dr. Volker Berkhahn und Prof. Dr. Rudolf Damrath (†18.10.2003) möchte ich mich für die Zeit am Institut für Bauinformatik und für die vielen guten Ratschläge, die wichtigen Impulse und die bis heute nachwirkenden Anregungen bedanken. Diese Jahre und die freundschaftliche Unterstützung durch Volker Berkhahn haben wesentlich zu meinem Werdegang und dem Erfolg dieser Arbeit beigetragen.

Mein Dank gilt auch meinen ehemaligen Kollegen am Hasso-Plattner-Institut und insbesondere Thomas Bauer und Ralf Merettig, die mir in ihrer damaligen Funktion als studentische Mitarbeiter wertvolle Hilfe bei der Realisierung dieser Arbeit boten. Besonders möchte ich mich auch bei Isabel Peuker für ihre Unterstützung bei der Erstellung von Veröffentlichungen und Lehrmaterial bedanken.

Mein Freundeskreis bot mir in den Jahren meines Studiums und während der Jahre meiner Promotion stets einen wichtigen Rückhalt, für den ich mich an dieser Stelle nicht genug bedanken kann. Besonders herzlich danke ich Britta Kothe, die mir über Jahre eine kritische Wegbegleiterin und meine wichtigste Ratgeberin war. Mein Dank gilt an dieser Stelle auch Jens Scheffermann und Jens Hündling, die mir als Diskussionspartner und als Freunde bei fachlichen Problemstellungen kompetente Hilfestellung und in Stresssituationen die notwendige Ablenkung boten.

Die Unterstützung durch meine Familie kann nicht durch diese Worte aufgewogen werden. Trotzdem möchte ich meinen Eltern Monika und Wolfgang für ihr unerschütterliches Vertrauen in meine Fähigkeiten und ihre verlässliche Begleitung durch alle Höhen und Tiefen meines Lebens danken. Ich möchte auch Susanne danken, als ältere Schwester vorangegangenen zu sein und in wichtigen Momenten immer zu mir gehalten zu haben.

Abschließend und von ganzem Herzen danke ich Dir, Tani, für Deine endlose Geduld und Deine liebevolle Unterstützung, die mir in den vergangenen Monaten, während der Endphase der Erstellung dieser Arbeit, unermesslich wichtig waren.

Christopher Robinson-Mallett
Berlin, den 28. Oktober 2005

Kurzfassung

Mit zunehmender Komplexität technischer Softwaresysteme ist die Nachfrage an produktiveren Methoden und Werkzeugen auch im sicherheitskritischen Umfeld gewachsen. Da insbesondere objektorientierte und modellbasierte Ansätze und Methoden ausgezeichnete Eigenschaften zur Entwicklung großer und komplexer Systeme besitzen, ist zu erwarten, dass diese in naher Zukunft selbst bis in sicherheitskritische Bereiche der Softwareentwicklung vordringen.
Mit der *Unified Modeling Language Real-Time* (UML-RT) wird eine Softwareentwicklungsmethode für technische Systeme durch die *Object Management Group* (OMG) propagiert. Für den praktischen Einsatz im technischen und sicherheitskritischen Umfeld muss diese Methode nicht nur bestimmte technische Eigenschaften, beispielsweise temporale Analysierbarkeit, besitzen, sondern auch in einen bestehenden Qualitätssicherungsprozess integrierbar sein. Ein wichtiger Aspekt der Integration der UML-RT in ein qualitätsorientiertes Prozessmodell, beispielsweise in das V-Modell, ist die Verfügbarkeit von ausgereiften Konzepten und Methoden für einen systematischen Modultest.
Der Modultest dient als erste Qualititätssicherungsphase nach der Implementierung der Fehlerfindung und dem Qualitätsnachweis für jede separat prüfbare Softwarekomponente eines Systems. Während dieser Phase stellt die Durchführung von systematischen Tests die wichtigste Qualitätssicherungsmaßnahme dar. Während zum jetzigen Zeitpunkt zwar ausgereifte Methoden und Werkzeuge für die modellbasierte Softwareentwicklung zur Verfügung stehen, existieren nur wenig überzeugende Lösungen für eine systematische modellbasierte Modulprüfung.

Die durchgängige Verwendung ausführbarer Modelle und Codegenerierung stellen wesentliche Konzepte der modellbasierten Softwareentwicklung dar. Sie dienen der konstruktiven Fehlerreduktion durch Automatisierung ansonsten fehlerträchtiger, manueller Vorgänge. Im Rahmen einer modellbasierten Qualitätssicherung sollten diese Konzepte konsequenterweise in die späteren Qualitätssicherungsphasen transportiert werden. Daher ist eine wesentliche Forderung an ein Verfahren zur modellbasierten Modulprüfung ein möglichst hoher Grad an Automatisierung.

In aktuellen Entwicklungen hat sich für die Generierung von Testfällen auf Basis von Zustandsautomaten die Verwendung von *Model Checking* als effiziente und an die vielfältigsten Testprobleme anpassbare Methode bewährt. Der Ansatz des *Model Checking* stammt ursprünglich aus dem Entwurf von Kommunikationsprotokollen und wurde bereits erfolgreich auf verschiedene Probleme der Modellierung technischer Software angewendet. Insbesondere in der Gegenwart ausführbarer, automatenbasierter Modelle erscheint die Verwendung von *Model Checking* sinnvoll, das die Existenz einer formalen, zustandsbasierten Spezifikation voraussetzt. Ein ausführbares, zustandsbasiertes Modell erfüllt diese Anforderungen in der Regel. Aus diesen Gründen ist die Wahl eines *Model Checking* Ansatzes für die Generierung von Testfällen im Rahmen eines modellbasierten Modultestverfahrens eine logische Konsequenz.

Obwohl in der aktuellen Spezifikation der UML-RT keine eindeutigen Aussagen über den zur Verhaltensbeschreibung zu verwendenden Formalismus gemacht werden, ist es wahrscheinlich, dass es sich bei der UML-RT um eine zu *Real-Time Object-Oriented Modeling* (ROOM) kompatible Methode handelt. Alle in dieser Arbeit präsentierten Methoden und Ergebnisse sind somit auf die kommende UML-RT übertragbar und von sehr aktueller Bedeutung.

Aus den genannten Gründen verfolgt diese Arbeit das Ziel, die analytische Qualitätssicherung in der modellbasierten Softwareentwicklung mittels einer modellbasierten Methode für den Modultest zu verbessern. Zu diesem Zweck wird eine neuartige Testmethode präsentiert, die auf automatenbasierten Verhaltensmodellen und *CTL Model Checking* basiert. Die Testfallgenerierung kann weitgehend automatisch erfolgen, um Fehler durch menschlichen Einfluss auszuschließen. Das entwickelte Modultestverfahren ist in die technischen Konzepte *Model Driven Architecture* und ROOM, beziehungsweise *UML-RT*, sowie in die organisatorischen Konzepte eines qualitätsorientierten Prozessmodells, beispielsweise das V-Modell, integrierbar.

Abstract

In consequence to the increasing complexity of technical software-systems the demand on highly productive methods and tools is increasing even in the field of safety-critical systems. In particular, object-oriented and model-based approaches to software-development provide excellent abilities to develop large and highly complex systems. Therefore, it can be expected that in the near future these methods will find application even in the safety-critical area.
The *Unified Modeling Language Real-Time* (UML-RT) is a software-development methods for technical systems, which is propagated by the *Object Management Group* (OMG). For the practical application of this method in the field of technical and safety-critical systems it has to provide certain technical qualities, e.g. applicability of temporal analyses. Furthermore, it needs to be integrated into the existing quality assurance process. An important aspect of the integration of UML-RT in an quality-oriented process model, e.g. the V-Model, represents the availability of sophisticated concepts and methods for systematic unit-testing.
Unit-testing is the first quality assurance phase after implementation to reveal faults and to approve the quality of each independently testable software component. During this phase the systematic execution of test-cases is the most important quality assurance task. Despite the fact, that today many sophisticated, commercial methods and tools for model-based software-development are available, no convincing solutions exist for systematic model-based unit-testing.

The use of executable models and automatic code generation are important concepts of model-based software development, which enable the constructive reduction of faults through automation of error-prone tasks. Consequently, these concepts should be transferred into the testing phases by a model-based quality assurance approach. Therefore, a major requirement of a model-based unit-testing method is a high degree of automation. In the best case, this should result in fully automatic test-case generation.

Model checking already has been approved an efficient and flexible method for the automated generation of test-cases from specifications in the form of finite state-machines. The *model checking* approach has been developed for the verification of communication protocols and it was applied successfully to a wide range of problems in the field of technical software modelling. The application of model checking demands a formal, state-based representation of the system. Therefore, the use of *model checking* for the generation of test-cases is a beneficial approach to improve the quality in a model-based software development with executable, state-based models.

Although, in its current state the specification of UML-RT provides only little information on the semantics of the formalism that has to be used to specify a component's behaviour, it can be assumed that it will be compatible to *Real-Time Object-Oriented Modeling*. Therefore, all presented methods and results in this dissertation are transferable to UML-RT.

For these reasons, this dissertations aims at the improvement of the analytical quality assurance in a model-based software development process. To achieve this goal, a new model-based approach to automated unit-testing on the basis of state-based behavioural models and *CTL Model Checking* is presented. The presented method for test-case generation can be automated to avoid faults due to error-prone human activities. Furthermore it can be integrated into the technical concepts of the *Model Driven Architecture* and ROOM, respectively UML-RT, and into a quality-oriented process model, like the *V-Model*.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Mit zunehmender Komplexität technischer Softwaresysteme ist die Nachfrage an produktiveren Methoden und Werkzeugen in diesem Sektor stetig gewachsen. Da insbesondere objektorientierte und modellbasierte Ansätze und Methoden [58, 64, 113, 143, 61, 78] ausgezeichnete Eigenschaften zur Entwicklung großer und komplexer Systeme besitzen, ist zu erwarten, dass diese in naher Zukunft auch bis in sicherheitskritische Bereiche der Softwareentwicklung vordringen. Obwohl zum jetzigen Zeitpunkt ausgereifte Methoden und Werkzeuge für die modellbasierte Softwareentwicklung zur Verfügung stehen, finden sich nur wenig überzeugende Lösungen für die modellbasierte Qualitätssicherung. Der Modultest dient als erste Qualititätssicherungsphase nach der Implementierung der umfassenden Fehlersuche und dem Qualitätsnachweis jeder separat prüfbaren Softwarekomponente eines Systems. Während dieser Phase stellt die Durchführung von systematischen Tests die wichtigste Qualitätssicherungsmaßnahme dar.

Diese Arbeit hat das Ziel die Qualität komplexer, technischer Softwaresysteme durch die Entwicklung leistungsfähiger, modellbasierter Methoden für Implementierung und Modultest zu verbessern. Zu diesem Zweck wird in dieser Arbeit eine automatisierbare Methode für den Modultest mit *Real-Time Object-Oriented Modeling* präsentiert.

Motivation

Mit der *Unified Modeling Language Real-Time* wird eine Softwareentwicklungsmethode für technische Systeme durch die *Object Management Group* propagiert [113]. Für den praktischen Einsatz im technischen und sicherheitskritischen Umfeld muss diese Methode nicht nur bestimmte technische Eigenschaften, beispielsweise temporale Analysierbarkeit, besitzen, sondern auch in einen bestehenden Qualitätssicherungsprozess integrierbar sein. Ein wichtiger Aspekt der Integration der *Unified Modeling Language Real-Time* in ein qualitätsorientiertes Prozessmodell, beispielsweise das V-Modell [27, 26], stellt die Verfügbarkeit von ausgereiften Konzepten und Methoden für einen systematischen Modultest dar. Leider existieren bis zum jetzigen Zeitpunkt nur wenig überzeugende Konzepte und Methoden einer systematischen modellbasierten Modulprüfung.

Obwohl in der aktuellen Spezifikation der *Unified Modeling Language Real-Time* keine eindeutigen Aussagen über den zur Verhaltensbeschreibung zu verwendenden Formalismus gemacht werden, ist es wahrscheinlich, dass es sich bei der *Unified Modeling Language Real-Time* [113] um eine zu *Real-Time Object-Oriented Modeling* [143] kompatible Methode handelt. Alle in dieser Arbeit präsentierten Methoden und Ergebnisse sind somit auf die kommende *Unified Modeling Language Real-Time* übertragbar und von sehr aktueller Bedeutung. Die Wahl von *Real-Time Object-Oriented Modeling*, anstelle der *Unified Modeling Language Real-Time*, als Grundlage für diese Arbeit hat weiterhin den Vorteil, dass eine formale und detaillierte Spezifikation von *Real-Time Object-Oriented Modeling* [143] vorliegt.

Die durchgängige Verwendung ausführbarer Modelle und Codegenerierung stellen wesentliche Konzepte der modellbasierten Softwareentwicklung dar. Sie dienen der konstruktiven Fehlerreduktion durch Automatisierung ansonsten fehlerträchtiger Vorgänge und sollten im Rahmen einer modellbasierten Qualitätssicherung konsequenterweise in die späteren Qualitätssicherungsphasen transportiert werden. Daher ist eine wesentliche Forderung an ein Verfahren zur modellbasierten

Modulprüfung ein möglichst hoher Grad an Automatisierung, was im besten Fall in vollautomatischer Testfallgenerierung resultiert. Diese Verfahren basieren auf der Struktur der ausführbaren Modelle, in der Regel in Form von Zustandsautomaten. In aktuellen Entwicklungen hat sich für die Generierung von Testfällen auf Basis von Zustandsautomaten die Verwendung von *Model Checking* als effiziente und an die vielfältigsten Testprobleme anpassbare Methode bewährt. Der Ansatz des *Model Checking* stammt ursprünglich aus dem Entwurf von Kommunikationsprotokollen und wurde bereits erfolgreich auf verschiedene Probleme der Modellierung technischer Software angewendet. In den vergangenen Jahren hat sich *Model Checking* als effektive Methode für die Sicherheits- und Strukturanalyse von technischer Software etabliert. Insbesondere in der Gegenwart ausführbarer, automatenbasierter Modelle erscheint die Verwendung von *Model Checking* sinnvoll, das die Existenz einer formalen, zustandsbasierten Spezifikation voraussetzt. Ein ausführbares, zustandsbasiertes Modell erfüllt diese Anforderungen in der Regel. Aus diesen Gründen erscheint die Wahl eines *Model Checking* Ansatzes für die Generierung von Testfällen im Rahmen eines modellbasierten Modultestverfahrens als logische Konsequenz.

Die Problematik der Anwendung objektorientierter Methoden im Bereich zeitkritischer Systeme ist hinreichend bekannt. Neuere Forschungsergebnisse und Erfahrungsberichte lassen allerdings hoffen, dass auch hier noch unausgeschöpftes Potential vorhanden ist [9, 62, 119, 128, 108, 76, 23, 147]. Beispielsweise kann auf Basis einer Kritikalitätsanalyse eine sicherheitsrelevante Softwareentwicklung unter abgestufter Einbeziehung objektorientierter Konzepte erfolgen. Je nach Höhe des Kritikalitätsgrades einer Systemkomponente können unzweckmäßige objektorientierte Konzepte identifiziert und von der Entwicklung ausgeschlossen werden. In den meisten Fällen ist beispielsweise die Verwendung von Polymorphie oder dynamischer Speicherverwaltung in Komponenten mit höchster Kritikalität nicht ratsam [136, 42]. Da hoch kritische Komponenten oftmals der Umsetzung von Kernfunktionalität dienen und eine vergleichsweise einfache Struktur besitzen, ist ein Verzicht auf die Verwendung objektorientierter Konzepte zur Komplexitätsbeherrschung in der Regel unproblematisch. In weniger kritischen Komponenten mit komplexerer Struktur können diese Konzepte dagegen sehr hilfreich sein und sollten verwendet werden. Unter diesen Voraussetzungen können objektorientierte Methoden auch in Entwicklungen mit sicherheitskritischen Bestandteilen erfolgreich eingesetzt werden. Diese Annahmen werden durch verschiedene Arbeiten bezüglich der Anwendbarkeit von *Real-Time Object-Oriented Modeling* und *Unified Modeling Language Real-Time* unterstützt. Die Analysierbarkeit des Zeitverhaltens dieser Methoden für die Anwendung in der Entwicklung sicherheitskritischer Echtzeitsysteme wurde in verschiedenen Arbeiten präsentiert [86, 139, 141, 53]. Die strukturellen Eigenschaften wurden ebenfalls in verschiedenen Veröffentlichungen analysiert [86, 13, 107].

Aus den genannten Gründen verfolgt diese Arbeit das Ziel, die analytische Qualitätssicherung in der modellbasierten Softwareentwicklung mittels einer modellbasierten Methode für den Modultest zu verbessern. Zu diesem Zweck soll eine neuartige Testmethode entwickelt werden, die auf automatenbasierten Verhaltensmodellen und *CTL Model Checking* basiert. Die Testfallgenerierung soll möglichst automatisch erfolgen, um Fehler durch menschlichen Einfluss weitestgehend auszuschließen. Das entwickelte Modultestverfahren soll in die technischen Konzepte *Model Driven Architecture* und *Real-Time Object-Oriented Modeling*, beziehungsweise *Unified Modeling Language Real-Time*, sowie in die organisatorischen Konzepte eines qualitätsorientierten Prozessmodells, beispielsweise das V-Modell, integrierbar sein.

Wissenschaftliches Umfeld

Der Test basierend auf *Statecharts* und ähnlichen zustandsbasierten Spezifikationssprachen wurde bereits in verschiedenen Arbeiten untersucht [22, 21, 20, 25, 71, 148, 111, 110, 127, 56]. Keine dieser Arbeiten adressiert den Test basierend auf der Semantik von ROOMcharts. Die Ergebnisse der genannten Arbeiten lassen sich auf ROOMcharts nur teilweise übertragen und anwenden. Weiterhin hat keines dieser Verfahren die Behandlung aller Entwurfselemente von *Statecharts* vollständig beschrieben. Die detaillierte Behandlung von Auswahlpunkten und Gedächtniszuständen wird in keiner der genannten Arbeiten hinreichend beschrieben. In [20] empfiehlt Bogdanov eine detailliertere Untersuchung dieser Entwurfselemente.

Auch die Generierung von Testfällen aus zustandsbasierten Spezifikationen mittels *Model* -

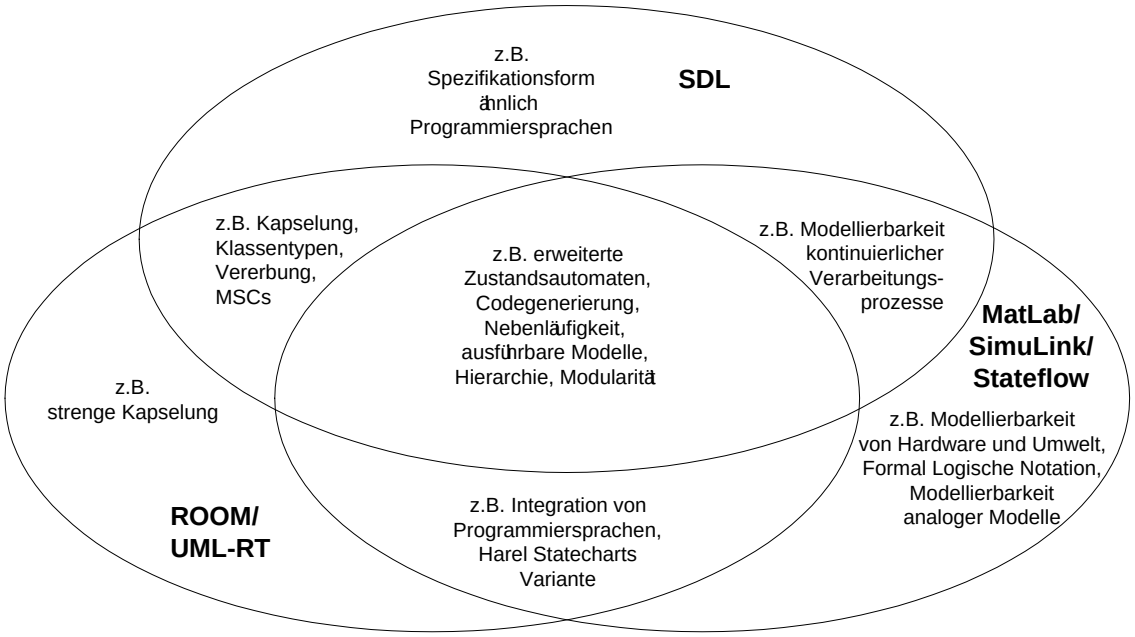


Abbildung 1.1: Modellbasierte Softwareentwicklungsmethoden im Vergleich

Checking wurde bereits in verschiedenen Arbeiten veröffentlicht [41, 75, 103, 47, 52, 93, 118, 130, 126, 47]. Es wurden vergleichbare Ansätze in [49, 80] und [48, 153, 152] vorgestellt, die konzeptionell und algorithmisch ähnlich wie *LTL Model Checking* verfahren. Diese wurden zunächst als reine *Black-Box*-Verfahren für LOTOS entwickelt und erst kürzlich an *UMLStatecharts* angepasst.

Keiner dieser Ansätze ist ähnlich anpassungsfähig an neue Testanforderungen wie der hier vorgestellte Ansatz mittels *Model Checking.* Weiterhin beinhaltet keine der vorangehend genannten Arbeiten vergleichbare Untersuchung hinsichtlich der konditionalen und hierarchischen Konzepte von *ROOMcharts*, bzw. *Statecharts.* Insbesondere fehlen Aussagen bezüglich der Komplexität der Testfallgenerierung auf Basis von ROOMcharts mittels *CTL Model Checking.*

Technisches Umfeld

In der modellbasierten Softwareentwicklung technischer Systeme können drei Strömungsrichtungen identifiziert werden, die sich aufgrund ihrer Werkzeugunterstützung und ihrer speziellen Eigenschaften in unterschiedlichen Anwendungsgebieten von technischer Software etabliert haben.

Die in der Softwareentwicklungsbranche vermutlich populärste dieser Entwicklungsrichtungen stellt die *Unified Modeling Language* dar, die mit ihrem Ableger *Unified Modeling Language Real-Time* die Methode *Real-Time Object-Oriented Modeling* aufgegriffen hat. Da die *Unified Modeling Language* Defizite in der Durchgängigkeit und Stringenz in den Analyse- und Entwurfsmethoden aufweist und zudem stark durch objektorientierte Belange geprägt ist, ist die Akzeptanz im technischen Umfeld noch eher verhalten. Mit der Veröffentlichung der *Unified Modeling Language Real-Time* und der vermutlich daraus resultierenden Entwicklung verbesserter Softwareentwicklungswerkzeuge ist eine weitere Verbreitung dieser Methoden allerdings wahrscheinlich.

In der Telekommunikationsbranche hat sich mit der *Specification and Description Language* ein Standard durchgesetzt, der in der aktuellen Fassung SDL-2000 [78] einen Schwerpunkt auf objektorientierte Konzepte legt. Besonders auffällig ist die semantische Ähnlichkeit zwischen *Real-Time Object-Oriented Modeling* und SDL-2000, die eine hohe Übertragbarkeit von Erkenntnissen und Konzepten bezüglich dieser Methoden bewirkt. Die SDL-2000 basiert auf der Verhaltensbeschreibung mit erweiterten Zustandsautomaten und eignet sich für die Spezifikation reaktiver, verteilter Echtzeitsysteme. Obwohl ursprünglich als Spezifikationssprache für Telekommunikationssysteme entwickelt, beginnt die *Specification and Description Language* auch in anderen Anwendungsge-

bieten technischer Softwaresysteme Verbreitung zu finden. Einige Gründe für diese Entwicklung sind neben dem hohen Reifegrad dieser Methode vor allem in der großen Anzahl verfügbarer, kommerzieller Entwicklungswerkzeuge zu finden. Ein guter Überblick über den Entwicklungsstand und die wichtigsten Entwicklungswerkzeuge kann auf den Internetseiten des SDL-Forums (*www.sdl-forum.org*) gewonnen werden.

Mit MATLAB, *Simulink* und *Stateflow* hat der Hersteller *The MathWorks* eine sehr umfangreiche und ausgereifte Entwicklungsumgebung für die Modellierung mathematisch, technischer Lösungen präsentiert. MATLAB bietet neben grundlegenden mathematischen Funktionen auch grafische Werkzeuge für die mathematische, statistische und ingenieurmäßige Auswertung großer Datenmengen. Basierend auf MATLAB bietet *Simulink* die Möglichkeit zur Simulation und zur prototypischen Entwicklung von dynamischen Systemen. Die Erweiterung *Stateflow* erlaubt es, ereignisgesteuerte Systeme in der Simulink-Umgebung zu modellieren und zu simulieren. Die auf MATLAB basierende modellbasierte Softwareentwicklung nimmt gegenüber den beiden anderen genannten Ansätzen eine Sonderstellung ein, da sie aus einem von mathematischen Belangen geprägten Umfeld entstanden ist. Hier liegt die besondere Stärke dieses Ansatzes, der die Verknüpfung analoger und diskreter Modelle erlaubt. Weiterhin existiert mit *Stateflow* eine Modellierungssprache, die auf *Harel Statecharts* basiert und daher viele Parallelen zu den anderen modellbasierten Entwicklungsmethoden erkennen lässt. Dokumentationen und Spezifikationen zu MATLAB*, Simulink* und *Stateflow* können auf den Internetseiten des Herstellers *The MathWorks* gefunden werden (*www.mathworks.com*). Eine Einführung in die Modellierung mit diesen Werkzeugen wird in [5] präsentiert.

In Abbildung 1.1 sind einige Merkmale von ROOM/UML-RT, SDL und MATLAB/*Simulink*/*Stateflow* schematisch dargestellt. Es fällt auf, dass diese drei Methoden für die Verhaltensspezifikation erweiterte Zustandsautomaten nutzen. Dies erhöht die Übertragbarkeit einer Vielzahl der in dieser Arbeit präsentierten Ergebnisse und Methoden auf andere modellbasierte Softwareentwicklungsmethoden.

Da diese Arbeit im Umfeld der objektorientierten Softwareentwicklung angesiedelt ist, scheidet MATLAB/*Simulink*/*Stateflow* bereits im Vorfeld für eine nähere Betrachtung aus. Die Entscheidung zugunsten *Real-Time Object-Oriented Modeling* fiel unter anderem aufgrund baldigen Veröffentlichung der *Unified Modeling Language Real-Time*.

Konzept

Bei *Real-Time Object-Oriented Modeling* handelt es sich um eine ausgereifte Methode der modellbasierten Softwareentwicklung [61, 143], die wesentlichen Einfluss auf die Entwicklung der *Model Driven Architecture* [61] hatte und deren Konzepte in der weitgehend kompatiblen *Unified Modeling Language Real-Time* [113] weitere Verbreitung finden. Einen wesentlichen Aspekt bei der Verwendung von modellbasierten Entwicklungsmethoden, insbesondere *Real-Time Object-Oriented Modeling*, stellt die durchgängige Verwendung von ausführbaren Modellen mit abschließender Codegenerierung dar. Auf Basis dieser Modelle können Tests bereits zu einem sehr frühen Zeitpunkt durchgeführt werden, was zu einer erheblichen Reduzierung der Fehlerbeseitigungskosten führen kann. Zu diesem Zweck werden Entwurfshilfen präsentiert, die es erlauben, ROOMcharts mit testbarer Struktur zu erstellen oder ROOMcharts zu transformieren, die nicht den in dieser Arbeit vorgestellten Testbarkeitsanforderungen genügen.

Mit *Model Checking* steht eine formale Qualitätssicherungsmethode zur Verfügung, die sich gut zur automatisierten Testfallgenerierung eignet. Da die Komplexität ein zentrales Problem des *Model Checking* darstellt, muss nahezu jede praktische Anwendung des *Model Checking* durch komplexitätsreduzierende Maßnahmen unterstützt werden. Ein *Model Checker* kann in der Regel nur dann effektiv zur Testfallgenerierung eingesetzt werden, wenn bereits während der frühen Entwicklungsphasen konstruktive Anforderungen berücksichtigt werden. Daher wird die Komplexität der vorgestellten Methoden zur Testfallgenerierung ausführlich untersucht und es werden Maßnahmen zur Komplexitätsreduktion vorgestellt und diskutiert.

In Abbildung 1.2 ist das Konzept der Kombination von *Model Checking* mit dem Ansatz der modellbasierten Softwareentwicklung mit *Real-Time Object-Oriented Modeling* dargestellt. Die während der Analyse verwendeten formalen Modelle können in abstrakte, unvollständige Reali-

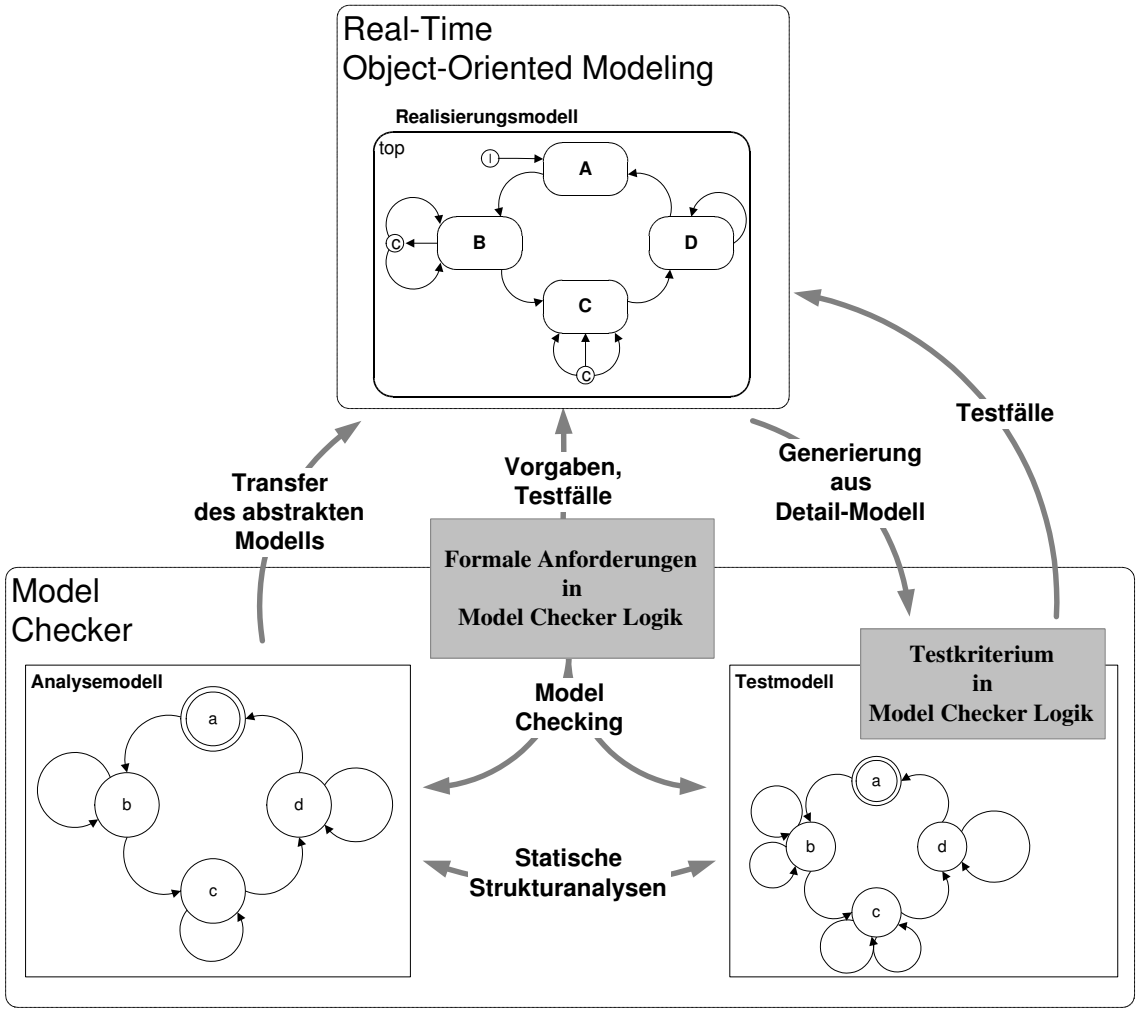


Abbildung 1.2: Durchführung der Qualitätssicherungphasen

sierungsmodelle transformiert werden. Während der modellbasierten Entwicklung können die formalen Anforderungen als Vorgaben für den Entwurf und als Testfälle für den funktionsorientierten, modellbasierten Test dienen. Der strukturorientierte, modellbasierte Test erfolgt auf Basis eines Testmodells, das aus dem ausführbaren, detaillierten Realisierungsmodell abgeleitet wurde. Die strukturorientierten Testfälle werden mittels generierter Testanforderungen erzeugt. Für die Erzeugung von Testanforderungen stehen verschiedene Kriterien zur Auswahl, die hinsichtlich Eignung und Komplexität in dieser Arbeit untersucht werden.

Die Verwendung von *Model Checking* zur Testfallgenerierung passt sich gut in einen sicherheitsrelevanten, modellbasierten Softwareentwicklungsprozess ein. Neben der Testfallgenerierung ermöglicht *Model Checking* das Konzept der Fehlerreduktion durch Automatisierung in die Qualitätssicherungsphase zu transportieren. In Abbildung 1.2 sind die verschiedenen Anwendungsmöglichkeiten des *Model Checking* eines abgeleiteten Testmodells dargestellt. Für die Entwicklung sicherheitskritischer Module mit strengen Zeitanforderungen kann sich das Konzept der Codegenerierung als problematisch erweisen. Trotzdem ist die vorgestellte Methode von Nutzen, wenn der Softwareentwicklungsprozess auf Basis ausführbarer Modelle erfolgt und nur die Implementierung manuell durchgeführt wird. Die Testfallgenerierung mittels *Model Checking* dient in diesem Fall als funktionsorientiertes Testverfahren, indem ROOMcharts als Basis für die Testfallermittlung und als Prüfreferenz dienen.

Die Konzepte dieser Arbeit wurden auf verschiedenen internationalen Workshops und Konferenzen präsentiert [132, 131, 133, 134, 135]. Weitere Veröffentlichungen der in dieser Arbeit

präsentierten Ergebnisse sind in Vorbereitung.

Gliederung

Diese Arbeit ist in elf Kapitel gegliedert. In den Kapiteln 2 bis 5 werden die wesentlichen Grundlagen der modellbasierten Softwareentwicklung, von *Real-Time Object-Oriented Modeling*, des *Model Checking* und verschiedener Testverfahren dargestellt und ihr Bezug zu dieser Arbeit aufgezeigt. In Kapitel 6 werden Testbarkeitsanforderungen an ROOMcharts formuliert und detailliert erläutert. Mit *Hierarchischen Pseudozuständen* wird ein neues Entwurfselement für die integrierte Modellierung komplexer Verarbeitungsabläufe in ROOMcharts vorgestellt. Die Generierung von Testmodellen aus ROOMcharts wird in Kapitel 8 dargestellt und in Beispielen erklärt. Neben dem allgemeinen Verfahren zur Generierung von endlichen Zustandsautomaten aus ROOMcharts wird auch die Generierung von Testmodellen für den *Model Checker Uppaal* ausgeführt. Die Testfallgenerierung auf Basis endlicher Zustandsautomaten und mittels Uppaal wird in Kapitel 9 präsentiert. Neben den verschiedenen Testkriterien werden auch problematische Eigenschaften von Testmodellen hinsichtlich der automatischen Testfallgenerierung erläutert. Die Komplexität des Ansatzes der Generierung von Testfällen mit Uppaal wird diskutiert und es werden verschiedene Techniken zur Komplexitätsreduktion vorgestellt und in Beispielen erläutert. In Kapitel 10 werden Realisierungskonzepte des Testwerkzeugs ROOMtest für *Rational Rose Real-Time* vorgestellt, das auf Basis der hier vorgestellten Testmethode entwickelt wurde. Abschließend werden in Kapitel 11 die Ergebnisse dieser Arbeit zusammenfassend dargestellt, kritisch bewertet und Vorschläge für nachfolgende Forschungsarbeiten präsentiert.

Kapitel 2

Modellbasierte Softwareentwicklung

Das Modell eines Systems ist eine Beschreibung oder Spezifikation dieses Systems und seiner Umwelt mit einem bestimmten Zweck [61]. Ein Modell wird häufig durch eine Kombination visueller und textueller, formaler und informaler Mittel dargestellt. Ein System wird nach Birolini [15] als eine Zusammenfassung technischer und organisatorischer Mittel zur autonomen Erfüllung eines Aufgabenkomplexes definiert. Es besteht im Allgemeinen aus den verschiedensten Dingen, beispielsweise Hardware, Software, Menschen und logistischer Unterstützung. In einem technischen System können die Einflüsse durch Menschen und Logistik bei der Betrachtung vernachlässigt werden.

Die modellbasierte Softwareentwicklung vereinigt eine Vielzahl von Ansätzen und Methoden für die Entwicklung von Softwaresystemen unter einem einheitlichen Konzept der Transformation von zumeist ausführbaren Modellen von der abstrakten Anforderungsspezifikation bis zur Implementation. Sie umfasst ebenfalls den Ansatz der Codegenerierung und die damit verbundene hohe Durchgängigkeit der verschiedenen Systemmodelle. Mit *Real-Time Object-Oriented Modeling* existiert eine der konsequentesten Ausprägungen des modellbasierten Entwicklungsansatzes, obwohl die von der *Object Management Group* als Standard der modellbasierten Softwareentwicklung propagierte *Model Driven Architecture* deutlich jünger ist. In diesem Kapitel werden verschiedene Konzepte der *Model Driven Architecture* erläutert und an das vorliegende Testproblem angepasst.

2.1 Model Driven Architecture

Mit der *Model Driven Architecture* wurde durch die *Object Management Group* ein einheitliches Konzept für die Durchführung modellbasierter Softwareentwicklungen präsentiert [61]. Die zugrunde liegende Idee ist die Separation der Spezifikation eines Systems von den Implementierungsdetails der Zielplattform. Ein System wird unabhängig von der Zielplattform spezifiziert. Eine Zielplattform ist eine Zusammenfassung einer Menge von Subsystemen, die unabhängig von deren Implementierung von Anwendungen genutzt werden können. Die Funktionalitäten oder Dienste einer Plattform werden weitgehend unabhängig von möglichen Anwendungen spezifiziert. Eine Anwendung stellt die Realisierung einer Anzahl von Funktionalitäten oder Diensten auf der Plattform dar. Mit Plattformunabhängigkeit wird in diesem Kontext die Unabhängigkeit eines Systems von den Funktionalitäten einer beliebigen Plattform bezeichnet. Die *Model Driven Architecture* zielt auf die Steigerung der Portabilität und Interoperabilität sowie auf die Erhöhung des Grads der Wiederverwendbarkeit von Software und Spezifikation. Die Rolle der Modelle in der Entwicklung wird gegenüber der Implementierung hervorgehoben. Es werden Mittel für die Nutzung von Modellen unterstützt, um den gesamten Softwareentwicklungsprozess kontrollierbarer zu gestalten. Dies betrifft die Analyse, den Entwurf, die Konstruktion, die Installation, den Einsatz, die Wartung und die Änderung von Systemen.

Um den unterschiedlichen Anforderungen während der Softwareentwicklungsphasen gerecht zu werden, wird zwischen verschiedenen Sichtweisen auf das System unterschieden.

Berechnungsunabhängiges Modell Die von der Berechnung unabhängige Sichtweise fokussiert auf die Anforderungen und die Umgebung des Systems. Detaillierte Strukturbeschreibungen

oder Berechnungen werden vernachlässigt oder sind noch nicht vorhanden. Ein abstraktes Modell, das vordergründig Systemumgebung und Anforderungen darstellt, wird als unabhängig von der Berechnung (*engl. Computation Independent Model, CIM*) bezeichnet. Für die Erstellung eines berechnungsunabhängigen Modells ist kein Wissen über Realisierungsdetails des Systems notwendig.

Plattformunabhängiges Modell Die von der Plattform unabhängige Sichtweise fokussiert auf die Arbeitsweise des Systems. Von der Plattform abhängige Verarbeitungsdetails werden vernachlässigt oder sie sind noch unbestimmt. Ein von der Plattform unabhängiges Modell (*engl. Platform Independent Model, PIM*) spezifiziert jene Teile des Systems, für deren Realisierung keine detaillierten Kenntnisse der Zielplattform notwendig sind. Es besitzt den notwendigen Grad an Plattformunabhängigkeit, um das Systemmodell auf verschiedenen Plattformtypen umzusetzen. Für die Darstellung kann vorzugsweise eine universelle oder eine auf das Problemgebiet angepasste Modellierungssprache verwendet werden. Eine virtuelle Maschine ist eine Zusammenfassung von Funktionalitäten und Diensten, die plattformunabhängig für eine Vielzahl von Plattformen realisiert werden können. Durch den Entwurf eines Systems für eine virtuelle Maschine kann ein hoher Grad an Plattformunabhängigkeit erreicht werden.

Plattformspezifisches Modell Ein plattformspezifisches Modell (*engl. Platform Specific Model, PSM*) kombiniert die Spezifikation des plattformunabhängigen Modells mit den Details über die Nutzung von Diensten und Funktionalitäten einer bestimmten Plattform.

Plattformmodell Ein Modell einer Plattform (*engl. Platform Model, PM*) umfasst eine Menge von technischen Konzepten zur Beschreibung der Bestandteile und Dienste der Plattform. Es wird weiterhin spezifiziert, wie die genutzten Bestandteile und Dienste einer Plattform in einem plattformunabhängigen Modell dargestellt werden.

Implementation Die Implementation eines Systems ist eine Spezifikation, die alle Informationen zur Konstruktion und Inbetriebnahme des Systems besitzt [61].

Die Transformationen zwischen den verschiedenen Modellen erfolgen nach einem Muster ähnlich jenem in Abbildung 2.1. Jede Transformation zwischen zwei Modellen wird durch eine Abbildungsfunktion (*engl. Mapping Function*) [97, 61] spezifiziert, die eine Zusammenfassung von detaillierten Regeln und Algorithmen für eine bestimmte Art von Abbildungen darstellt. Die in dieser Arbeit vorgestellte Ableitungsvorschrift für Testmodelle aus ROOMcharts stellt eine solche Abbildungsfunktion dar. Diese Abbildungsfunktion ist allerdings auf keine bestimmten Modelltypen beschränkt, insofern es sich um ausführbare ROOMcharts handelt. Weiterhin kann ein Testmodell für unterschiedlichste Zwecke dienen, es wird in der Regel aber keine Transformation zu einem Modell mit erhöhtem Detaillierungsgrad vorgenommen.

2.2 Transformationen zwischen Modellen

Eine Transformation zwischen Modellen ist ein Prozess, in dem ein Modell eines Systems in ein anderes Modell des gleichen Systems umgewandelt wird.

In Abbildung 2.1 ist schematisch die beispielhafte Transformation eines plattformunabhängigen Modells in ein plattformabhängiges Modell dargestellt. Diese Darstellung der Transformation von Modellen ist an ein Entwurfsmuster der *Model Driven Architecture* angelehnt [61]. Die Modifikation gegenüber dem MDA-Entwurfsmuster betrifft die Darstellung des menschlichen Einflusses. Das plattformunabhängige Modell PIM und weitere für das System relevante Beschreibungen, also auch Beschreibungen der Plattform, werden kombiniert und zu einem plattformabhängigen Modell PSM transformiert. Der menschliche Einfluss ist durch eine Person schematisch dargestellt, die eine eigene Interpretation des Systems in den Prozess der Transformation einbringt. Der menschliche Einfluss kann durch die Verwendung von formalen Modellen auf ein Minimum reduziert werden. In den meisten Fällen wird der menschliche Einfluss so groß sein, dass eine Umkehr des Prozesses problematisch ist.

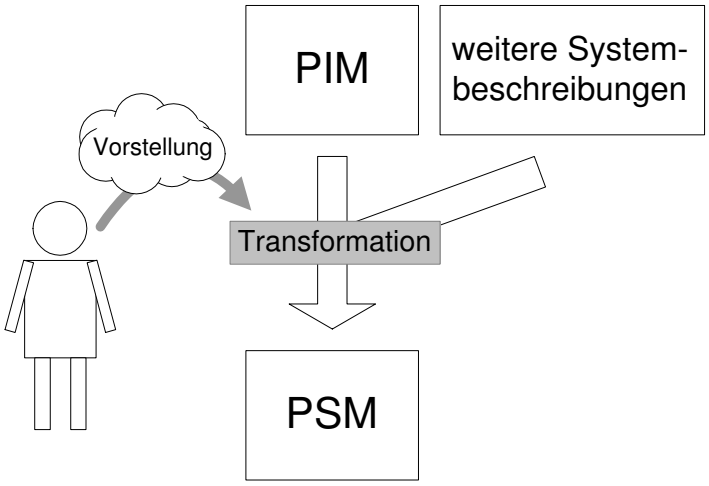


Abbildung 2.1: Beispielhafte Transformation von Modellen

2.3 Modellbasierter Softwareentwicklungsprozess

Ein modellbasierter und phasengesteuerter Softwareentwicklungsprozess kann beschrieben werden, ohne auf Transformationsdetails von Modellen einzugehen. Die modellbasierten Softwareentwicklungsansätze behandeln ausschließlich technische Belange der Transformation eines Systems vom abstrakten Modell bis zur Implementierung. Der organisatorische Ablauf des Softwareentwicklungsprozesses und der Einsatz von Modellen während einzelner Softwareentwicklungsphasen werden nicht behandelt.

Ein möglicher organisatorischer Rahmen für die Softwareentwicklung stellen Prozessmodelle dar. Ein typischer Vertreter ist das V-Modell [27, 26], das für die Softwareentwicklung für den öffentlichen Sektor in Deutschland vorgeschrieben ist. Während der Analysephase werden die Anforderungen informal oder formal in Lasten- und Pflichtenheften spezifiziert und zum berechnungsunabhängigen Modell CIM transformiert. Weitere übliche Darstellungsformen für Anforderungen sind beispielsweise *Use-Case-Maps*, Sequenzdiagramme, *Statecharts* oder temporaler Logik [78, 7, 113, 118]. Die Analysephase wird abgeschlossen, wenn das berechnungsunabhängige Modell CIM fertig gestellt ist. Während des Entwurfs wird das berechnungsunabhängige Modell zu einem plattformunabhängigen Modell PIM verfeinert und ergänzt. Die Implementierung verfeinert und erweitert dieses Modell zu einem plattformspezifischen Modell PSM. Jede Transformation ist mit menschlichem Einfluss und daraus resultierenden Konsequenzen verbunden. Dagegen sind Codegenerierung und Kompilierung frei von menschlichem Einfluss, insofern der generierte Code nicht manuell verändert wird. In Abbildung 2.2 ist das Modell eines phasengesteuerten, modellbasierten Softwareentwicklungsprozesses dargestellt. Am Ende der Entwicklungsphasen steht die Codegenerierung mit anschließender Kompilierung und Installation auf der Zielplattform.

Modellbasiertes Testen Im Rahmen der modellbasierten Softwareentwicklung mit ausführbaren Modellen können bereits zu einem sehr frühen Zeitpunkt dynamische Prüfungen durchgeführt werden. Diese Tests werden begleitend während der Softwareentwicklungsphasen und als Ergänzung zu den üblichen statischen Prüfungen durchgeführt. Die modellbasierten Tests besitzen die gleiche Aussagekraft wie Tests in einer künstlichen Entwicklungs- oder Testumgebung, wenn die getesteten, korrigierten Modelle unverändert zur Codegenerierung genutzt werden. Wenn keine Codegenerierung möglich sein sollte, können die ausführbaren Modelle als Referenz für funktionsorientierte Tests dienen.

Diese vergleichsweise frühe dynamische Prüfung kann zu signifikanten Einsparungen bei den Fehlerbeseitigungskosten beitragen [109]. Während der späten analytischen Qualitätssicherungsphasen der Modul-, Integrations- und Systemprüfung können ergänzend zu den modellbasierten Tests weitere struktur- oder funktionsorientierte Verfahren angewendet werden.

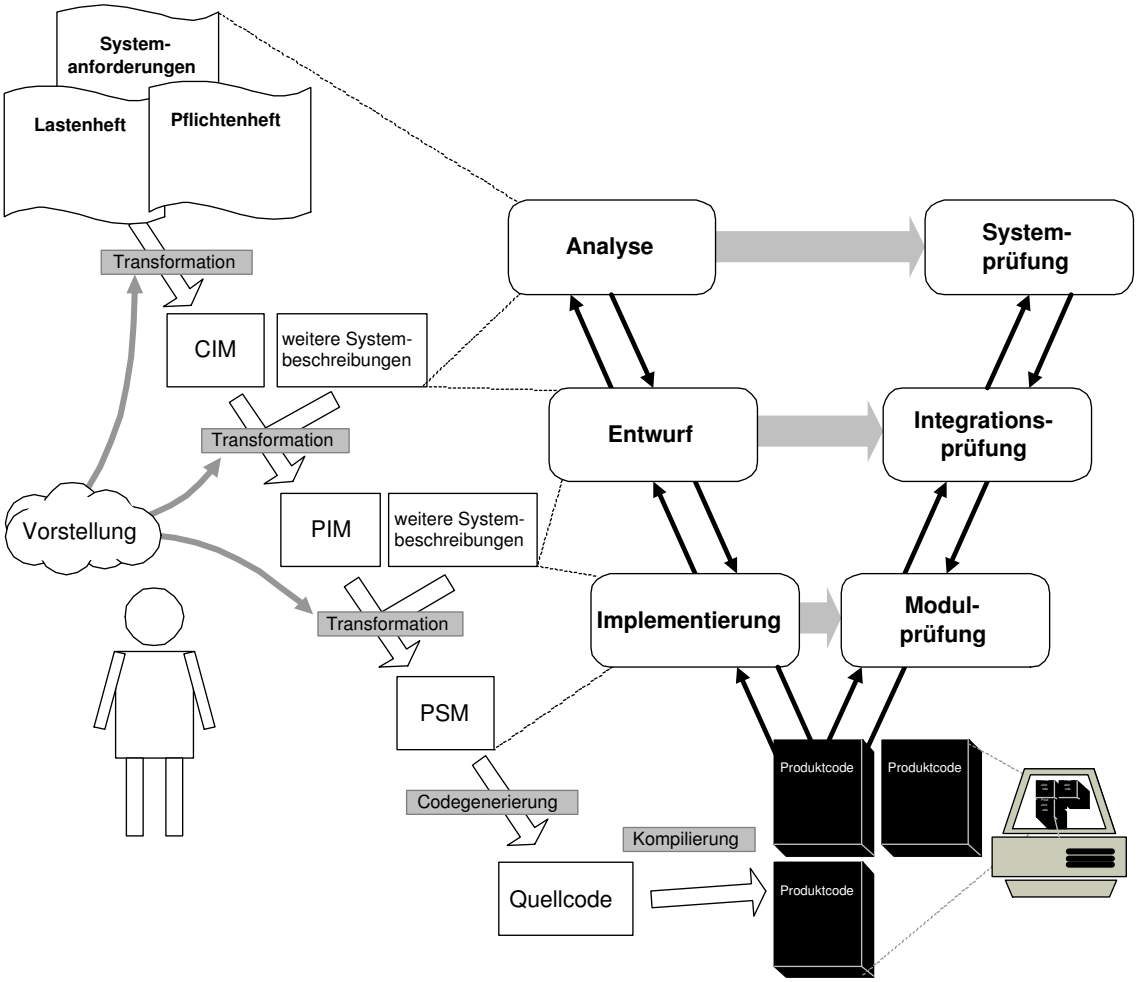


Abbildung 2.2: Beispielhafte Transformation von Modellen

Modellbasierter Modultest für Real-Time Object-Oriented Modeling Für die Modularisierung eines Systems in *Real-Time Object-Oriented Modeling* während des Entwurfs bieten sich Aktoren und Protokolle an. Diese Entwurfskomponenten repräsentieren kompakte und eigenständige Einheiten, die sich ebenfalls gut für einen isolierten Modultest eignen. Das plattformunabhängige Modell wird während der Entwurfsphase erstellt und enthält Aktoren, die das erwünschte, abstrakte Verhalten spezifizieren. Neben abstrakten Aktoren werden während des Entwurfs weitere Modulspezifikationen erstellt, die als Implementierungsvorlagen dienen.
Die Entwurfsphase ist beendet, wenn alle Anforderungen an die Aktoren vollständig spezifiziert wurden.
Während der Implementierung wird das plattformspezifische Modell aus dem plattformunabhängigen Modell erstellt. Das abstrakte Verhalten der Aktoren wird verfeinert, bis alle für die Codegenerierung notwendigen Informationen enthalten sind.
Die Implementierungsphase ist beendet, wenn das Verhalten jedes Aktors vollständig und detailliert realisiert wurde.

Im Gegensatz zum modellbasierten Testen während der frühen Softwareentwicklungsphasen erfolgt die Prüfung im modellbasierten Modultest auf Basis von detaillierten und vollständigen ROOMcharts, den plattformspezifischen Modellen. Dieser hohe Detaillierungsgrad führt in der Regel zu einer höheren Komplexität, da beispielsweise komplexe Aktionen, erweiterte Zustandsvariablen und Wächterbedingungen zu berücksichtigen sind.

Eine Testmethode, die erfolgreich im modellbasierten Test anwendbar ist kann daher nur dann im modellbasierten Modultest verwendet werden, wenn sie auf ROOMcharts mit höchstem De-

taillierungsgrad anwendbar ist. Dagegen kann die in dieser Arbeit präsentierte Methode für einen modellbasierten Modultest auch auf abstrakte ROOMcharts im modellbasierten Test während früher Entwicklungsphasen verwendet werden.

2.3.1 Portierung und Installation

Der modellbasierte Ansatz mit ausführbaren Modellen benötigt insbesondere für die Entwicklung von technischer Software eine komplexe Entwicklungs- und Simulationsumgebung. Das Zielsystem wird durch ein Plattformmodell repräsentiert, in dem alle zur Entwicklung notwendigen Annahmen und Informationen enthalten sind. Spätestens mit dem Ende der Modulprüfung wird das entwickelte Softwaresystem auf der Zielplattform installiert.

Während der Installation wird das Plattformmodell gegen die reale Zielplattform ausgetauscht und die Software wird von der Entwicklungsumgebung auf die Zielumgebung portiert. Häufig wird ein System für die Installation auf einer Vielzahl unterschiedlicher Zielplattformen entwickelt. Es ist ebenfalls möglich, dass lange nach Beendigung der Entwicklung Anforderungen für die Installation des Systems auf eine neue Zielplattform erhoben werden. Obwohl während Installation und Portierung nicht zwangsläufig weitere Entwicklungsschritte erfolgen, muss das System auf mögliche Inkonsistenzen zwischen Plattformmodell und Zielplattform geprüft werden. Diese Prüfung kann mittels automatischer Regressionstests erfolgen. Zu diesem Zweck werden Testfälle wiederholt, die während Entwicklung und Prüfung erstellt wurden. Dies setzt eine hohe Portabilität der Testfälle voraus. In der Regel ist dies nur für Testfälle gewährleistet, die während der Analysephase entwickelt werden. Während Entwurf, Implementierung und Modultest werden dagegen oftmals strukturorientierte Testverfahren eingesetzt, die eine Instrumentierung der Software voraussetzen. Diese Instrumentierung dient in der Regel der Verbesserung der Beobachtbarkeit und Steuerbarkeit der Software im Test. Da die Instrumentierung oftmals gar nicht oder nur eingeschränkt auf der Zielplattform eingesetzt werden kann, verlieren strukturorientierte Testfälle an Aussagekraft. Dieses Problem verschärft sich im modellbasierten Test mit Codegenerierung, da ein ausführbares Modell sowohl Implementation, als auch Dokumentation der Software darstellt. Die aus einem ausführbaren, zur Codegenerierung genutzten Modell erzeugten Testfälle sind somit als strukturorientiert anzusehen. Die automatenbasierten Testverfahren [18] kommen ohne Instrumentierung aus und sind für einen Portierungstest daher besonders vorteilhaft.

In Abbildung 2.3 ist die beispielhafte Portierung und Installation auf verschiedene Plattformen schematisch dargestellt. Für die Portierung werden aus einem plattformunabhängigen Modell PIM und den jeweiligen Plattformmodellen PM0 bis PM5 verschiedene plattformabhängige Modelle entwickelt. Die Installation der lauffähigen Software auf die Zielplattform wird durch Portierungstests unterstützt.

2.3.2 Testphasen

Die Durchführung von Tests kann in einem modellbasierten Prozess bereits zu einem frühen Zeitpunkt entwicklungsbegleitend erfolgen. Obwohl diese Tests einen hohen Nutzen versprechen, ist die Durchführung der Softwareprüfungsphasen des Modul-, Integrations- und Systemtests nicht obsolet. Während dieser Qualitätssicherungsphasen wird die Erfüllung von Qualitätsanforderungen angestrebt, die in Normen, Standards und vertraglichen Vereinbarung festgelegt sind.

In Abbildung 2.4 ist schematisch die Durchführung der drei Qualitätssicherungsphasen Modul-, Integrations- und Systemtest präsentiert. Auf der linken Seite sind die während der Entwicklung erstellten Dokumententypen aufgeführt. Auf der rechten Seite sind die Sichtweisen auf das lauffähige Softwaresystem während der Qualitätssicherungssphasen dargestellt.
In der ersten Qualitätssicherungsphase des *Modultest* können Testfälle aus allen zur Verfügung stehenden Dokumenten abgeleitet werden. Außerdem kann die Struktur der Module für die Ableitung von Testfällen und die Definition von Testvollständigkeitskriterien genutzt werden. Während der Durchführung von Testläufen sind in der Regel Beobachtbarkeit und Steuerbarkeit der Module gegeben. Die Durchführung der Testfälle liefert das isolierte, möglichst vollständige Verhalten jedes Moduls. Die Prüfung des Modulverhalten erfolgt gegen die Modulspezifikation. Das Ziel des Modultest ist das Finden von Fehlern, die möglichst vollständige Prüfung von Anforderungen und Struktur sowie der Nachweis bestimmter Qualitätseigenschaften.

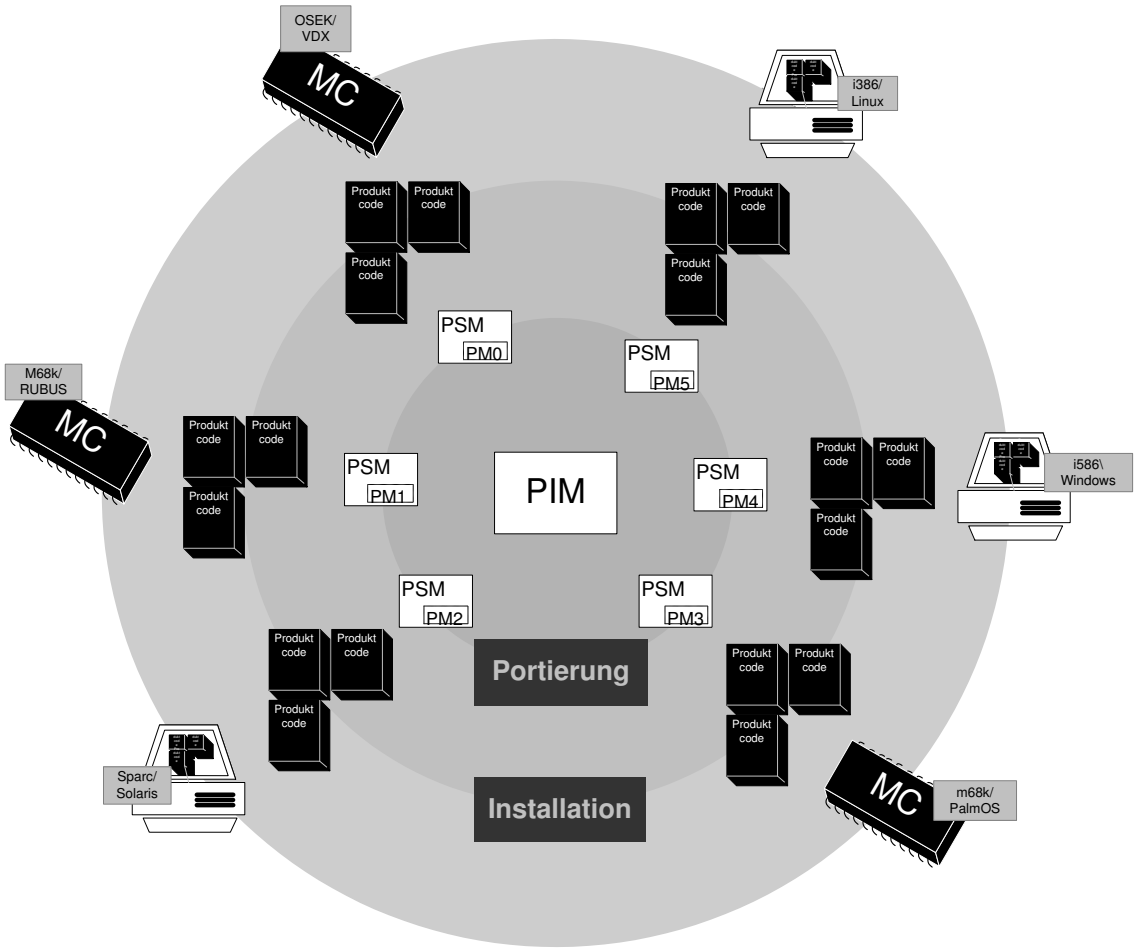


Abbildung 2.3: Portierung auf verschiedene Plattformen

Während der Qualitätssicherungsphase des *Integrationstest* werden Testfälle in der Regel schrittweise bis zur vollständigen Systemintegration aus den Entwurfsdokumenten [8] abgeleitet. Außerdem kann die Struktur des Systems für die Ableitung von Testfällen und die Definition von Testvollständigkeitskriterien genutzt werden. Während der Durchführung von Testläufen sind die interne Struktur und das interne Verhalten der einzelnen Module in der Regel nicht sichtbar. Während der Durchführung der Testfälle werden Interaktion und Kommunikation zwischen den Modulen beobachtet und protokolliert. Dieses interaktive Verhalten kann gegen alle bis zur Entwurfsphase erstellten Dokumente geprüft werden. Das Ziel des Integrationstests ist das Aufdecken von Fehlern in der Kommunikation und der Interaktion der Systemmodule sowie die fehlerfreie Integration des Systems.

Während der Qualitätssicherungsphase des *Systemtest* werden die Analysedokumente für die Ableitung von Testfällen und die Definition von Testvollständigkeitskriterien genutzt. Die Ausführung der Testfälle liefert das reale Systemverhalten, welches gegen die Analysedokumente evaluiert wird. Die interne Struktur und das interne Verhalten des Systems sind in der Regel nicht sichtbar. Das Ziel des Systemtests ist das Aufdecken fehlerhaften Systemverhaltens.

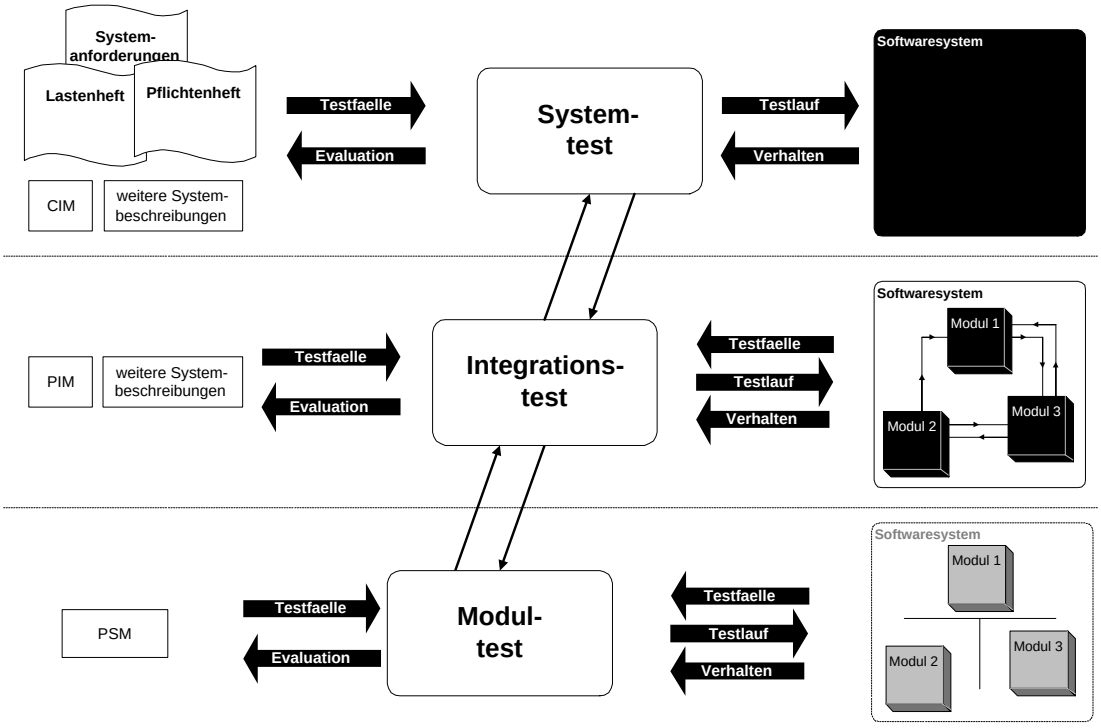


Abbildung 2.4: Durchführung der Qualitätssicherungphasen

Kapitel 3

Real-Time Object-Oriented Modeling

Die Softwareentwicklungsmethode *Real-Time Object Oriented Modeling* [143] erlaubt die Modellierung verteilter Echtzeitsysteme basierend auf dem Objekt-Paradigma [99]. Die Struktur eines Systems wird in nebenläufig arbeitende Komponenten mit sequentiellem Verhalten aufgeteilt. Für die Verhaltensbeschreibung einer Komponente wird eine Variation von *Harel-Statecharts* [64] verwendet. In diesem Kapitel folgt eine kurze Einführung in die wesentlichen Konzepte für die Struktur- und Verhaltensbeschreibung.

3.1 Strukturelle Konzepte

Die fundamentalen Entwurfskonzepte von *Real-Time Object-Oriented Modeling* stellen Aktoren und Protokolle dar. Die Kommunikation zwischen Aktoren wird durch Protokolle definiert. Ein System kann mittels Assoziation und Komposition von Aktoren modular modelliert werden. Die Organisation in hierarchischen Vererbungsstrukturen erlaubt die Abstraktion oder Verfeinerung von Aktor- und Protokollspezifikationen.

Aktoren Ein Aktor repräsentiert ein aktives Objekt, das eine klar definierte Aufgabe hat und das nebenläufig zu anderen Objekten arbeiten kann. Mittels Aktoren werden signifikante, funktionale Komponenten eines reaktiven Systems modelliert, deren Aufgaben eine Abstraktion ihrer verschiedenen funktionalen Eigenschaften darstellen. Jeder Aktor kann zur Implementierung komplexer Funktionalitäten dienen, die in Subkomponenten zerlegt werden können. Die funktionalen Eigenschaften und die Aufgabe eines Aktors werden durch seine Kapselung bestimmt. Diese Kapselung stellt die verschiedenen funktionalen Eigenschaften als eine Einheit dar, die der Aufgabe des Aktors dienen. Ein Aktor schützt mittels Kapselung seine internen Strukturen gegen externe Zugriffe und verhindert den Zugriff auf externe Strukturen durch interne Komponenten. Da mehrere Aktoren eines Systems häufig sehr ähnliche Verantwortlichkeiten besitzen, beispielsweise eine Anlagensteuerung mit mehreren gleichartigen Motoren, ist jeder Aktor das Exemplar einer Aktorklasse.

Ports Eine Kapselung kann nur an einem Port durchbrochen werden. An diesen Öffnungen der Kapsel kann ein Aktor mit seiner Umwelt über Datenflüsse in Form diskreter Nachrichtenpakete kommunizieren. Durch die Kapselung kann das interne Verhalten eines Aktors ausschließlich durch das auf seinen Ports sichtbare Ein- und Ausgabeverhalten abgeleitet werden. Ebenso nimmt ein Aktor seine Umwelt nur durch seine Ports wahr. In einem konjugierten Port werden die gültigen Ein- und Ausgabeströme spiegelsymmetrisch gegeneinander vertauscht.

Protokolle Eines der wichtigsten Attribute eines Ports ist das ihm zugeordnete Protokoll. Dieses besteht aus einer Menge gültiger Nachrichtentypen, die den Port passieren dürfen, und gültigen Sequenzen für den Nachrichtenaustausch. Diese Definition einer Objektschnittstelle geht über die üblichen Konzepte objektorientierter Programmiersprachen hinaus, die in der Regel ausschließlich auf Signaturen zulässiger Datentypen basieren. Da Protokolle häufig von mehreren Ports genutzt

werden, gibt es das Konzept der Protokollklasse. Eine Protokollklasse ist eine allgemeine Spezifikation für alle Port-Exemplare, die das gleiche Protokoll besitzen.

Bindungen Eine Bindung ist eine Abstraktion eines zugrunde liegenden Kommunikationskanals, der Nachrichten zwischen zwei Aktoren transportiert. Bindungen können nur zwischen zwei Ports existieren, die das gleiche Protokoll besitzen. Jede Bindung kann sowohl bidirektional als auch unidirektional definiert werden. Dies wird ausschließlich durch den zugrunde liegenden Kommunikationsdienst bestimmt. Das Konzept der Bindung regelt und beschränkt die gültigen Kommunikationsbeziehungen zwischen Aktoren. Ein Port kann in konjugierter Form verwendet werden, um Bindungen mit asymmetrischen Protokollen zu ermöglichen. Um die Kommunikation eines Aktors mit einer Vielzahl von Aktoren zu ermöglichen, können Ports multipliziert werden.

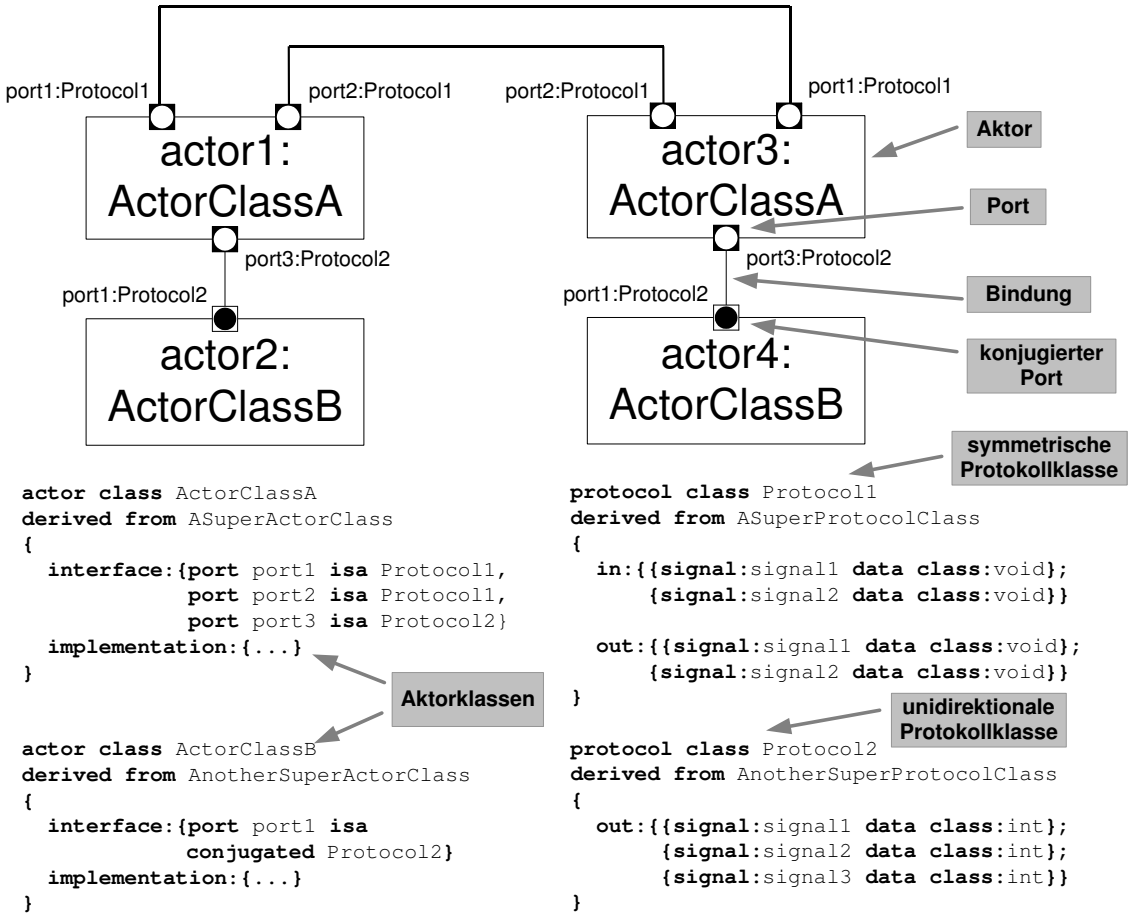


Abbildung 3.1: Strukturdiagramm mit Aktor- und Protokollklassen

Das beispielhafte Strukturdiagramm in Abbildung 3.1 stellt vier Aktoren dar, die über Ports und Bindungen kommunizieren. Die Aktoren *actor1* und *actor3* der Klasse *ActorClassA* besitzen drei Ports. Die Ports *port1* und *port2* der Aktorklasse *ActorClassA* sind Exemplare der Protokollklasse *Protocol1*, die im unteren Teil textuell spezifiziert ist. Sie können die Signale *signal1* und *signal2* vom Typ *void* empfangen. Ein Signal ist äquivalent mit einer Nachricht zu betrachten. Das Eintreffen eines Signals bzw. einer Nachricht auf einem Port stellt ein Ereignis dar. Der Port *port3* ist von der Protokollklasse *Protocol1* und kann die ganzzahligen Signale *signal1*, *signal2* und *signal3* vom Typ *int* senden. Die Aktoren *actor2* und *actor4* besitzen je einen konjugierten Port. Um eine Bindung der Aktorklassen *ActorClassA* und *ActorClassB* über das Protokoll *Protocol2* zu

erlauben, ist der Port *port1* der Aktorklasse *ActorClassB* konjugiert. Daher ist diese Kommunikation zwischen den Aktorklassen *ActorClassA* und *ActorClassB* unidirektional. Die Aktoren *actor1* und *actor3* sind Exemplare der gleichen Aktorklasse und kommunizieren bidirektional über die Ports mit dem symmetrischen Protokoll *Protocol1*.

3.1.1 Nachrichten und Ereignisse

Eine Nachricht ist in *Real-Time Object-Oriented Modeling* ein Objekt eines beliebigen Datentyps der zugrunde liegenden objektorientierten Programmiersprache. Damit können Nachrichten nahezu beliebig komplexe und umfangreiche Daten enthalten. Ein Ereignis wird generiert, wenn eine Nachricht von einem Aktor durch dessen Schnittstellenkomponenten gesendet wird. Ein Ereignis tritt auf, wenn eine Nachricht durch eine Schnittstellenkomponente eines Aktors empfangen wird. Ein Ereignis ist im Gegensatz zu einer Nachricht ein temporales Konzept. Das Senden und das Empfangen von Nachrichten sind die einzigen Möglichkeiten ein Ereignis zu erzeugen.

Auf das Empfangen einer Nachricht durch einen Aktor wird oftmals eine Antwort erwartet. Dies beinhaltet aktorinterne Berechnungen, um eine passende Antwort zu formulieren, und das Senden einer oder mehrerer Nachrichten. Ein Aktor befindet sich entweder im Zustand der Erwartung eines Ereignisses, oder er verarbeitet das zuletzt aufgetretene Ereignis. Wenn ein Ereignis einen Aktor erreicht, während bereits ein anderes verarbeitet wird, so wird das gerade aufgetretene Ereignis in eine Warteschlange aufgenommen und nach Beendigung der derzeitigen Verarbeitung behandelt (*engl. Run-To-Completion* [143]). Der größte Nachteil dieses Verarbeitungsmodells ist, dass keine Unterbrechung der derzeitigen Verarbeitung möglich ist. Daher dürfen in zeitkritischen Modellen einzelne Verarbeitungen nur geringe Zeit beanspruchen, um eine zeitgerechte Beantwortung vorrangiger Ereignisse zu ermöglichen. Auf Systemebene erfolgt die Ereignisverarbeitung dagegen *preemptive* (dt. bevorrechtigt, präventiv) [144, 143]. Ein Aktor kann nicht den Prozessor monopolisieren, da andere Aktoren mit hochrangigeren Ereignissen die aktuelle Verarbeitung eines anderen Aktors unterbrechen können. Daher kann ein hoch priorisiertes Ereignis auf einem Aktor die Ereignisverarbeitung anderer Aktoren unterbrechen.

Die Prioritäten von Ereignissen in *Real-Time Object-Oriented Modeling* bestimmen die Bearbeitungsreihenfolge beim gleichzeitigen Auftreten mehrerer Ereignisse auf den Ports eines Aktors. Der Rang von Ereignissen ist nur bei der Kommunikation zwischen Aktoren von Bedeutung. Die Kommunikation zwischen Aktoren und die Behandlung von Ereignisprioritäten hängt maßgeblich vom zugrunde liegenden System ab. Die vorrangige Abfertigung eines hochrangigen Ereignisses kann daher nur dann garantiert werden, wenn dies vom zugrunde liegenden Kommunikationssystem garantiert wird.

Die Ablaufkoordination von Ereignissen auf Prozessen wird durch Terminierung innerhalb der virtuellen Umgebung von *Real-Time Object-Oriented Modeling* vorgenommen. Ein Entwurfsmodell und dessen System können sich über mehrere physikalische Verarbeitungsplattformen erstrecken. Jede Plattform stellt den koordinierten Ablauf und die Verarbeitung von Ereignissen her. Jede einzelne Verhaltenskomponente existiert auf genau einer physikalischen Verarbeitungsplattform, obwohl sich der umgebende Aktor über mehrere Plattformen erstrecken kann. Jede Schnittstellenkomponente eines Aktors besitzt eine eigene Warteschlange, die je nach Definition bei Überschreiten der Kapazität überläuft oder überzählige Ereignisse verwirft.

3.1.2 Verhaltensschnittstellenobjekte

Für die Implementierung von Aktionen sollten vorzugsweise objektorientierte Programmiersprachen genutzt werden. Dies stellt sicher, das kein Paradigmenwechsel während des gesamten Entwicklungsprozesses vorgenommen werden muss.

Die Konzepte von Aktoren, hierarchischen Zustandsautomaten, Ports und Protokollen sind unabhängig von der verwendeten Programmiersprache definierbar. Ebenso sind die von der Programmiersprache unterstützten Konzepte unabhängig von *Real-Time Object-Oriented Modeling*. Die Verbindung dieser beiden Ebenen geschieht durch spezielle Objektbibliotheken. Diese stehen in Aktionen, Bedingungen, Transitionstriggern oder in Aktoroperationen zur Verfügung. Die Semantik dieser Objekte wird durch *Real-Time Object-Oriented Modeling* definiert, wogegen sie in der Syntax der gewählten Programmiersprache ausgedrückt werden. Dies beinhaltet Objekte, die

sowohl auf der strukturellen Ebene als auch in der Verhaltensbeschreibung sichtbar sind. Diese Verhaltensschnittstellenobjekte dienen der Kommunikation zwischen Aktoren und zur Kommunikation mit der Systemumgebung.

Nachrichten werden durch die Verhaltensschnittstelle eines Aktors empfangen und verschickt. Eine Verhaltensschnittstelle setzt sich aus *Endports*, *Service Access Points (SAP)* und *Service Provision Points (SPP)* zusammen. Ein *SAP* dient zum Empfangen von Nachrichten aus der Systemumgebung, und ein *SPP* dient zum Senden von Nachrichten an die Systemumgebung.

Diese Objekte repräsentieren die Verbindung zwischen dem Verhalten und der Struktur eines Aktors. In der Verhaltensspezifikation erscheinen diese Objekte als passive Datenobjekte, während sie in der Strukturspezifikation definiert werden. Das Verschicken einer Nachricht wird durch den Aufruf einer entsprechenden Operation auf einem solchen Objekt initiiert. Das Empfangen einer Nachricht wird durch die Referenz eines solchen Objekts in einem Transitionstrigger verarbeitet. Jedes Verhaltensschnittstellenobjekt definiert eine Warteschlange für eintreffende Nachrichten.

3.1.3 Kommunikationsdienste

Die genaue Semantik der wesentlichen Kommunikationselemente wird durch die Qualität des zugrunde liegenden Kommunikationsdienstes bestimmt. In *Real-Time Object-Oriented Modeling* ist mit dem *Basic Communication Service* ein primitiver Kommunikationsdienst vorhanden, der eine geringe, aber bestimmte Wahrscheinlichkeit für den Verlust, die Umordnung oder die Vervielfachung von Nachrichten und Nachrichtensequenzen besitzt.

Die Kommunikation zwischen den Aktoren eines Systems findet durch den Austausch von Nachrichten statt. Diese Art der Kommunikation kann synchron oder asynchron geschehen. In technischen Systemen sind beide Kommunikationsformen anzutreffen und anwendbar. Viele Programmiersprachen unterstützen beide Ansätze. Daher ist die Wahl der Kommunikationsform für ein verteiltes System eher im Entwurf anzusiedeln, auch wenn sie auf Ebene der Programmiersprache ihre Realisierung findet.

Asynchrone Kommunikation Die asynchrone Kommunikation zwischen Aktoren erlaubt dem Sender mit der eigenen Verarbeitung fortzufahren, sobald die Nachricht an den Kommunikationsdienst übermittelt wurde. Auf diese Weise kann die Zeitspanne, die ein Aktor auf keine Nachrichten reagieren kann, kurz gehalten werden. Weiterhin können die für die Kommunikation benötigten Mechanismen einfach realisiert werden. Daher ist diese Art der Kommunikation vorzuziehen, wenn möglichst kurze Reaktionszeiten gefordert werden.

Eine asynchrone Kommunikation erfolgt durch die Übergabe einer Nachricht durch den Sender an ein Verhaltensschnittstellenobjekt. Auf Codeebene wird das asynchrone Senden einer Nachricht durch eine Anweisung mit der Syntax

Portname.**send(***Signalname*,*Prioritätszahl*,*Datenobjekt***)**

dargestellt. Es wird kein Rückgabeobjekt erwartet. An anderer Stelle in dieser Arbeit sind auch vereinfachte Darstellungen dieser Kommunikation vorhanden, die sich in der Regel auf vereinfachte Parameterlisten beschränken.

Die Nachricht wird vom Kommunikationsdienst befördert und an die Verhaltensschnittstelle des Empfängers übergeben. Wenn der Empfänger bereit ist, wird die Nachricht aus der Verhaltenschnittstelle konsumiert und kann eine Transition auslösen. Wenn die Nachricht keine Transition auslöst, wird sie ohne Benachrichtigung des Senders verworfen.

Synchrone Kommunikation Bei synchroner Kommunikation zwischen Aktoren wird die Verarbeitung des Senders blockiert bis er eine Antwort auf die Nachricht erhalten hat. Während dieses Intervalls kann der Sender nicht auf eintreffende Nachrichten reagieren. Der Vorteil der synchronen Kommunikation ist der hohe Grad an Kontrolle über den zeitlichen Ablauf der Kommunikation.

Eine synchrone Kommunikation besteht aus dem kontrollierten Austausch von Nachrichten. Auf Codeebene wird das synchrone Senden einer Nachricht durch eine Anweisung mit der folgenden Syntax dargestellt:

rückgabeobjektname = **invoke(***Signalname*,*Datenobjekt***)**

In dieser Arbeit sind auch vereinfachte Darstellungen dieser Kommunikation vorhanden, die sich in der Regel auf vereinfachte Parameterlisten beschränken.

Die Beförderung der Nachricht durch den Kommunikationsdienst erfolgt auf gleiche Weise wie bei asynchroner Kommunikation. Wenn die Nachricht aus der Verhaltensschnittstelle des Empfängers konsumiert wird, muss die Aktion der ausgelösten Transition eine Antwort an den Sender durch die gleiche Schnittstellenkomponente beinhalten. Diese Antwortnachricht hat Vorrang vor allen anderen Nachrichten, die auf der Verhaltenschnittstelle des Senders eingereiht sein können. Nach Konsumierung der Antwortnachricht wird der Sender für weitere Verarbeitungen freigegeben.

Auf Codeebene wird das synchrone Senden einer Antwort durch eine Anweisung mit der folgenden Syntax dargestellt:

nachricht.**reply**(*Signalname*,*Antwortobjekt*)

In dieser Arbeit sind auch vereinfachte Darstellung dieser Kommunikation vorhanden, die sich in der Regel auf die Parameterliste beziehen.

Ein wesentlicher Nachteil der synchronen Kommunikation ist die Möglichkeit von *Deadlocks*, wenn die Verarbeitung des Systems zum Stillstand kommt. Dies kann beispielsweise durch den quasi gleichzeitigen, gegenseitigen Aufruf zweier Aktoren kommen. Eine andere Möglichkeit ist die Entstehung eines Deadlocks durch zyklische synchrone Kommunikation, die verhältnismäßig schwer zu analysieren ist.

Ein weiterer Nachteil sind erhöhte Latenzzeiten infolge von der erzwungenen Wartezeit beim Versenden einer Nachricht. Hier gehen die Transportzeiten der Nachricht und der Antwort, sowie die Latenz- und Verarbeitungszeit des Empfängers ein.

Um diese Effekte zu reduzieren, können bestehende Kommunikationen nach einer definierten Zeitspanne abgebrochen werden, wenn keine Antwort mehr erwartet werden kann. Dies hat den Verlust der Nachricht zur Folge und kann eine Störung des Systemablaufs bedeuten.

Empfang von Nachrichten In Zustandsautomaten und ähnlichen Formalismen ist keine explizite Definition des *Empfangens* von Nachrichten notwendig. Da in der Verhaltensspezifikation eines Aktors häufig auf die in der empfangenen Nachricht eingebetteten Informationen Zugriff erfolgen muss, ist eine syntaktische Definition für die Verarbeitung von Nachrichten notwendig.

Der Zugriff auf die in einer Nachricht eingebetteten Daten wird folgendermaßen dargestellt:

nachricht.**data**

Um den Zugriff auf das Signal der Nachricht zu ermöglichen, wird folgender Syntax genutzt:

nachricht.**signal**

Verschieben von Ereignissen Eine Nachricht mit höchster Priorität muss auch dann in der Verarbeitung vorangestellt werden, wenn diese unerwartet während der Verarbeitung einer komplexen Ereignissequenz eintrifft. Diese unerwünschte Unterbrechung der komplexen Ereignisverarbeitung sollte bei unerwarteten Nachrichten untergeordneter Priorität nicht erfolgen. Zu diesem Zweck soll die Verarbeitung des unerwarteten Ereignisses bis zur vollständigen Verarbeitung der komplexen Ereignissequenz verschoben werden. Zu diesem Zweck stellt *Real-Time Object-Oriented Modeling* die Möglichkeit des Verschiebens der Ereignisverarbeitung bereit. Dies gibt der Anwendung die Kontrolle über die Reihenfolge, in der Nachrichten verarbeitet werden. Eine verschobenes Ereignis wird in der Warteschlange der Schnittstelle vorgehalten, bis es im Code explizit freigegeben wird.

Um ein Ereignis zu verschieben wird folgende Syntax genutzt:

nachricht.**defer**()

Dies stellt das Ereignis in eine Warteschlange verschobener Ereignisse.

Um das erste verschobene Ereignis aus der Warteschlange freizugeben, wird die folgende Syntax verwendet:

port.**recall**(*frontFlag*)

Der boolesche Wert *frontFlag* entscheidet, ob das nächste Ereignis direkt verarbeitet wird oder ob es an die Ereigniswarteschlange der Schnittstellenkomponente *port* angehängt wird. Für den Fall *Wahr* wird das Ereignis sofort verarbeitet, im Fall *Falsch* wird das Ereignis in die Liste der nicht verschobenen Ereignisse eingestellt, die zwischen Verschieben und Freigeben des Ereignisses aufgetreten sein können.

Um alle Ereignisse aus der Warteschlangen freizugeben, wird die folgende Syntax verwendet.

port.**recallAll**(*frontFlag*)

Mit der Freigabe aller Ereignisse werden diese sofort in der vorhandenen Reihenfolge verarbeitet oder in die Warteschlange der Schnittstellenkomponente eingestellt.

3.1.4 Ausnahmebehandlung

Um die Realisierung von fehlertoleranten Softwaresystemen zu ermöglichen, bietet *Real-Time Object-Oriented Modeling* die Möglichkeit nachrichtenbasierte Ausnahmen zu generieren. Die Generierung von Ausnahmen kann grundsätzlich auf Ebene der zugrunde liegenden objektorientierten Programmiersprache oder als echter Bestandteil des Entwurfsmodells geschehen.

Programmiersprachliche Ausnahmebehandlung Die Behandlung von Ausnahmen variiert zwischen verschiedenen Programmiersprachen. Häufig besteht sie aus einer expliziten Identifizierung von Anweisungen, die gegen bestimmte Fehler durch anwendungsspezifische Wiederherstellungsmechanismen geschützt sind. Die programmiersprachliche Ausnahmebehandlung hat immer Vorrang vor der in *Real-Time Object-Oriented Modeling* spezifizierten Ausnahmebehandlung.

Ausnahmebehandlung in Real-Time Object-Oriented Modeling In *Real-Time Object-Oriented Modeling* werden Ausnahmen, die nicht von der programmiersprachlichen Ausnahmebehandlung erfasst wurden, durch einen unterliegenden Dienst abgefangen. Die Ausnahme wird dort in eine Nachricht verhüllt und an die Verhaltenskomponente geschickt, welche die Ausnahme produziert hat. Um die Nachricht empfangen zu können, benötigt die entsprechende Komponente eine Bindung mit dem Dienst der Ausnahmebehandlung. Die Ausnahmebehandlung erfolgt in der Komponente durch Transitionen, die auf Signale des entsprechenden Anschlusspunkts auslösen.

3.1.5 Zeitbedingungen

In jedem Echtzeitmodell können Zeitintervalle und Zeitpunkte als Bedingungen im Verhalten von Bedeutung sein. Für die Auswertung von Zeitbedingungen ist mindestens ein Zeitdienst notwendig. Um einen festgelegten Zeitpunkt definieren zu können wird ein globaler Zeitgeber benötigt, der einen gemeinsamen Bezug zwischen verteilten Systemkomponenten herstellt. Für die Auswertung von Zeitintervallen genügt ein lokaler Zeitdienst pro Komponente; ein globaler Zeitdienst ist nicht notwendig. Ein Zeitdienst kann systemextern oder systemintern vorhanden sein. Ein externer Zeitdienst wird über einen Endport angeschlossen, während ein interner Zeitdienst über einen Systemport angeschlossen wird.

Eine Anfrage an das Schnittstellenobjekt eines Zeitdienstes kann einen Zeitintervall oder einen Zeitpunkt spezifizieren. Nach Annahme der Anfrage wird ein Zeitgeber initialisiert. Jede Anfrage bewirkt die Initialisierung eines neuen Zeitgebers. Nach Ablauf des Zeitintervalls oder mit Erreichen des Zeitpunkts wird eine Nachricht der Zeitüberschreitung (*engl. Timeout*) vom Zeitgeber an die anfragende Komponente geschickt. Diese Nachricht wird in die Warteschlange des entsprechenden Anschlusspunktes gestellt, so dass eine Verzögerung der Bearbeitung der Nachricht möglich ist. Jeder Zeitgeber kann vor oder nach Zeitüberschreitung abgebrochen und zerstört werden. Die Spezifizierung von Prioritäten bei der Initialisierung eines Zeitgebers kann zur Realisierung harter periodischer Zeitanwendungen genutzt werden. Dies geschieht durch die Anfrage an den Verhaltensschnittstellenobjekt eines Zeitdienstes zur Initialisierung eines Zeitgebers mit höchster Priorität. Die Nachricht der Zeitüberschreitung dieses Zeitgebers erhält Vorrang vor allen Nachrichten untergeordneter Priorität bei der Nachrichtenübermittlung und in der Warteschlange.

Nach Senden einer Nachricht kann eine maximale Antwortzeit spezifiziert werden. Nach Ablauf dieser Zeitspanne kann angenommen werden, dass der Empfänger fehlerhaft oder gar nicht arbeitet. Die Definition von absoluten Zeitpunkten ist beispielsweise für sich täglich wiederholende Aktionen notwendig. Um Zeitbedingungen in *Real-Time Object-Oriented Modeling* darzustellen, müssen diese auf Nachrichten abgebildet werden. Daher besitzt *Real-Time Object-Oriented Modeling* einen *Timing Service*, der zu absoluten Zeitpunkten oder nach Ablauf von Zeitintervallen regelmäßige *Timeout*-Nachrichten generieren kann.

In einfacher Form wird ein Zeitintervall *Intervall* von einem Schnittstellenobjekt *timerport* in folgender Syntax definiert:

timerid = *timerport*.**informIn**(*Intervall*)
Ein Zeitgeber auf einem Schnittstellenobjekt *timerport* mit dem Identifikator *timerid* wird in folgender Syntax abgebrochen:
timerport.**cancelTimer**(*timerid*)
Um eine Benachrichtigung zu einem absoluten Zeitpunkt *time* zu erhalten wird folgende Syntax verwendet:
timerid = *timerport*.**informAt**(*time*)
Diese Dienste können um Prioritäten und Senderidentifikation erweitert werden, so dass in verteilten Systemen harte, periodische Zeitbedingungen realisiert werden können.

3.2 ROOMcharts

Die Aktoren eines Modells sind strukturierte Entitäten, die einen logischen Behälter für Verhaltensbeschreibungen bieten. Das Verhalten eines Aktors ist ein Bestandteil seiner Spezifikation. Jeder Aktor enthält einen obersten, hierarchisch strukturierten Zustand *top*, der eine Verhaltensbeschreibung in Form eines ROOMcharts enthält. Dieses ROOMchart, ein erweiterter Zustandsautomat [117], beschreibt das Antwortverhalten des Aktors auf Ereignisse aus der Umwelt und von anderen Aktoren.

Wie auch *Statecharts* [113, 64] bieten ROOMcharts hierarchische Strukturelemente und Steuerungsmechanismen. Im Gegensatz zu Statecharts bieten ROOMcharts aber eine eindeutige Semantik und lassen die hierarchische Definition von Namensräumen und erweiterten Zustandsvariablen zu. Da Nebenläufigkeit in *Real-Time Object-Oriented Modeling* nur auf der Systemebene definiert wird, besitzen ROOMcharts keine nebenläufigen Strukturelemente. Das mit ROOMcharts dargestellte Verhalten ist sequentiell und auf einen Aktor begrenzt. Die Ereignisverarbeitung wird auf ROOMcharts abgebildet, indem die Zustände den Modus der Ereigniserwartung abbilden und die Transitionen die Ereignisverarbeitung darstellen. Ein Ereignis löst eine Transition aus, die eine Ereignisverarbeitung bewirkt. Die auf die Transitionen abgebildete Ereignisverarbeitung wird durch eine beliebige, vorzugsweise eine objektorientierte Programmiersprache spezifiziert. In den folgenden Abschnitten werden die für eine Modulprüfung wesentlichen Entwurfselemente von ROOMcharts erläutert und in Beispielen dargestellt.

3.2.1 Zustände und Transitionen

Die Definition eines Zustands ermöglicht die Modellierung von hierarchischen Strukturen und erweiterter Funktionalität. Jeder Zustand kann eine beliebige Anzahl Unterzustände, Transitionen und zusätzlicher erweiterter Zustandsvariablen enthalten. Dies erlaubt die detaillierte Verfeinerung eines Verhaltens zu einem beliebigen Zeitpunkt im Entwicklungsprozess. Ein Zustand bildet einen Namensraum für lokale, erweiterte Zustandsvariablen. Ein Zustand, der keine Unterzustände enthält, wird elementarer Zustand genannt.

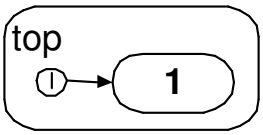


Abbildung 3.2: Zustände in ROOMcharts

Ein Zustand wird durch ein Rechteck mit gerundeten Ecken dargestellt und durch eine eindeutige Bezeichnung identifiziert. Der oberste hierarchische Zustand *top* wird von jedem Aktor separat definiert und besitzt einen Initialpunkt. Eine initiale Transition beginnt in einem Initialpunkt und endet in einem Unterzustand von *top*. In Abbildung 3.2 ist beispielhaft ein ROOMchart mit einem Zustand *1*, einem Initialpunkt und einer initialen Transition dargestellt.

Eine Transition bewirkt einen Zustandswechsel im Verhalten eines Aktors und führt eine Menge von Aktionsschritten über erweiterte Zustandsvariablen, temporäre Variablen und Nachrichtenobjekte aus. Jede Transition kann durch eine beliebige Anzahl Ereignisse ausgelöst werden. Die Spe-

zifikation der auslösenden Ereignisse einer Transition wird als Trigger bezeichnet. Eine Transition kann eine beliebige Bezeichnung besitzen.

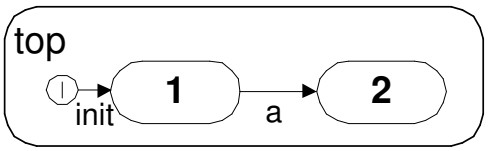


Abbildung 3.3: ROOMchart mit Transitionsbezeichnungen

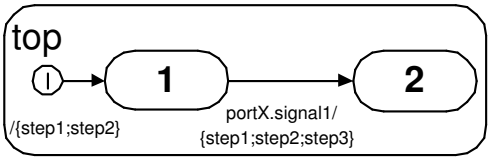


Abbildung 3.4: ROOMchart mit Transitionsspezifikationen

Die grafische Darstellung einer Transition erfolgt durch einen Pfeil vom Ausgangszustand zum Endzustand und kann optional mit einer Bezeichnung oder mit Trigger und Aktion beschriftet werden. Die Beispiele in den Abbildungen 3.3 und 3.4 stellen die möglichen grafischen Beschriftungsarten von Transitionen mit Bezeichnung oder Spezifikation dar. Die Beispiele können sich auf die gleichen Transitionen beziehen, da Bezeichnung und Spezifikation voneinander unabhängig sind.

Transitionstrigger Der Trigger (*engl. to trigger,* dt. auslösen, schalten) einer Transition besteht aus dem Namen des auslösenden Ereignisses, dem Namen des Schnittstellenobjekts und einer optionalen Wächterbedingung. Ein auslösendes Ereignis muss ein gültiges Eingangssignal auf einem Schnittstellenobjekt des Aktors sein. Die Wächterbedingung ist ein boolescher Ausdruck der dynamisch evaluiert wird, wenn das Ereignis zur Verarbeitung vorgesehen ist. Eine Transition wird ausgelöst, wenn das passende Ereignis eintritt und die Wächterbedingung erfüllt ist. Ein Trigger kann mehrere auslösende Ereignisse enthalten, die disjunktiv miteinander verknüpft sind. Es ist möglich, in einem Zustand mehrere Transitionen mit gleichen auslösenden Ereignissen oder nicht disjunkten Wächterbedingungen zu spezifizieren. In diesem Fall wird theoretisch eine beliebige Transition nicht-determistisch ausgeführt. In der Praxis hängt dieses Verhalten vom Codegenerator und dessen Umformungsregeln für Transitionen mit Wächterbedingung ab, da auf Codeebene kein Nichtdeterminismus existiert. Häufig ergibt sich eine deterministische Reihenfolge aus den Zeitpunkten der Erzeugung der einzelnen Transitionen, deren Auslöser beispielsweise in *switch*-Konstrukten [150, 100] evaluiert werden.

In einer Schachtelung von hierarchischen Zuständen können mehrere Transitionen für die Auslösung durch ein Ereignis zum gleichen Zeitpunkt in Frage kommen. Wenn ein ROOMchart sich während der Ausführung in einem elementaren Unterzustand befindet, werden auch alle übergeordneten hierarchischen Zustände eingenommen. Von einem elementaren Unterzustand bis zum obersten hierarchischen Zustand *top* können Transitionen mit gleichen Auslösern definiert sein. In einem ROOMchart wird im beschriebenen Fall nur die innerste Transition ausgelöst und das Ereignis konsumiert (*engl. Deepest First Triggering Rule*). Wenn in einer Hierarchieebene keine Transition ausgelöst werden kann, muss auf der nächsthöheren Hierarchieebene mit der Suche fortgefahren werden. Wenn auf der höchsten Hierarchieebene noch keine Transition ausgelöst werden kann, wird das Ereignis verworfen.

Detaillierter textueller Aktionscode Die Ereignisverarbeitung wird in Form von Aktionen an Transitionen und Zuständen definiert. Prinzipiell kann jede Programmiersprache zur Spezifi-

kation von Aktionscode genutzt werden. Die objektorientierten Programmiersprachen, beispielsweise JAVA oder C++ [100, 150], gelten als besonders geeignet, da sie das Objektparadigma in die nächsttiefere Implementierungsebene transportieren. Die wesentlichen Entwurfskonzepte sind unabhängig von einer spezifischen Programmiersprache. Um eine Verbindung zu der programmiersprachlichen Ebene herzustellen, werden spezielle Objektbibliotheken benutzt. Die Semantik dieser Objekte wird in *Real-Time Object-Oriented Modeling* definiert, während sie durch die Syntax der gewählten Programmiersprache repräsentiert werden. Ein Teil dieser Objekte ist in ROOMcharts und auf programmiersprachlicher Ebene zugänglich, beispielsweise Nachrichten und verschiedene Formen von Ports. Diese Objekte werden als *Detail Level Linkage Objects* bezeichnet. Mit detailliertem Programmtext können Aktionscode an Transitionen und Zuständen, Wächterbedingungen, mit dem ROOMchart assoziierte, interne Funktionen und Bedingungen an Auswahlpunkten definiert werden. Charakteristische Merkmale von Aktionscode sind die Definition der Kommunikation mit anderen Aktoren über die Verhaltensschnittstelle und der Zugriff auf Dienste von *Real-Time Object-Oriented Modeling*, wie beispielsweise Zeitgeber.

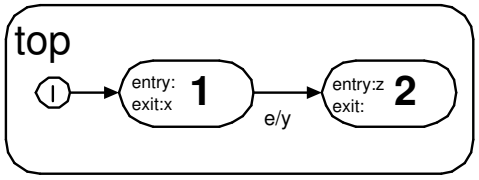


Abbildung 3.5: ROOMchart mit Eingangs- und Ausgangsaktionen

Eingangs- und Ausgangsaktionen von Zuständen Ein Zustand kann optionale Eingangs- und Ausgangsaktionen besitzen, die beim Betreten bzw. Verlassen des Zustands ausgeführt werden. Eine Eingangsaktion wird durch jede eingehende Transition ausgelöst und nach der Aktion dieser Transition ausgeführt. Eine Ausgangsaktion wird von jeder ausgehenden Transition ausgelöst und vor der Aktion dieser Transition ausgeführt.

Das Beispiel in Abbildung 3.5 zeigt ein ROOMchart mit zwei Zuständen und verschiedenen Eingangs- und Ausgangsaktionen. Der Zustand *1* besitzt die Ausgangsaktion x, und der Zustand *2* besitzt die Eingangsaktion z. Durch das Ereignis e wird die Aktion y ausgeführt und vom Zustand *1* in den Zustand *2* gewechselt. Dieses Beispiel führt die Aktion xyz aus.

3.2.2 Hierarchische Konzepte

Ein hierarchischer Zustand enthält einen Zustandsautomaten einer tieferen hierarchischen Ebene und abstrahiert ein detailliertes Verhalten. Der interne Automat eines hierarchischen Zustands kann als dessen Implementation betrachtet werden. Wenn in einem ROOMchart ein Unterzustand eingenommen wird, soll sich das System gleichzeitig in dessen Oberzuständen befinden.

Das Beispiel in 3.6 zeigt ein ROOMchart mit dem hierarchischen Zustand *B*. Der oberste Zustand *top* wird automatisch für jeden Aktor erzeugt und stellt ebenfalls einen hierarchischen Zustand dar. Der Oberzustand *top* enthält neben *B* auch zwei elementare Zustände *1* und *2*. Der hierarchische Zustand *B* enthält die elementaren Unterzustände *1* und *2*. Es kommt nicht zu einem Namenskonflikt in diesem ROOMchart, da jeder elementare Zustand durch seinen Namen und die Namen seiner Oberzustände identifiziert werden kann. Die Zustände einer Hierarchieebene innerhalb eines kompositionellen Zustands müssen eindeutig identifizierbar sein.

Gültigkeitsgrenzen erweiterter Zustandsvariablen In ROOMcharts ist die Einbettung von detailliertem Verhalten in einem abstrakten hierarchischen Zustand erlaubt. Diese Detaillierung ist syntaktisch vergleichbar mit der Bildung von Blöcken in strukturierten Programmiersprachen, beispielsweise Schleifenrümpfe oder *Wahr*- und *Falsch*-Block einer Bedingungsanweisung. Jeder Zustand definiert in ROOMcharts einen Namensraum für lokale, erweiterte Zustandsvariablen. Von einem Punkt in einem hierarchischen Zustand kann auf alle erweiterten Zustandsvariablen der

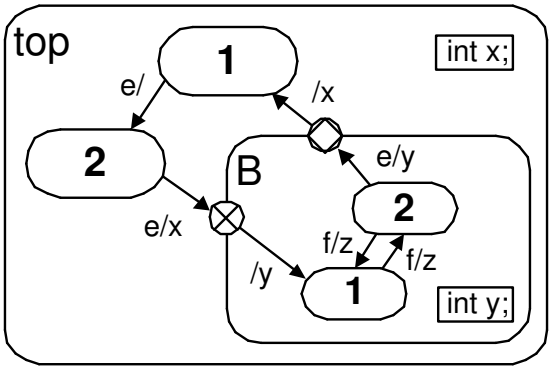


Abbildung 3.6: Hierarchisches ROOMchart

gleichen und aller höheren Hierarchieebenen zugegriffen werden. Von einer höheren Hierarchieebene kann aber nicht auf die erweiterten Zustandsvariablen tieferer hierarchischer Zustände zugegriffen werden. In dem beispielhaften ROOMchart in Abbildung 3.6 ist eine erweiterte Zustandsvariable x in *top* definiert, und eine erweiterte Zustandsvariable y ist in B definiert. Aus dem hierarchischen Zustand B ist der Zugriff auf die Variablen x und y möglich. Aus dem Zustand *top*, nicht innerhalb von B, ist nur der Zugriff auf die Variable x möglich.

Segmentierte Transitionen und Anschlusspunkte Ein hierarchischer Zustand s_c beinhaltet eine hierarchisch untergeordnete Ebene des Kontrollflusses. Um diese Ebene zu erreichen, muss die Zustandsgrenze von s_c durchbrochen werden. Dies geschieht mittels Anschlusspunkten, die auf der Zustandsgrenze definiert werden und beliebig viele eingehende mit einem ausgehenden Transitionssegment verbinden. Ein absteigender Anschlusspunkt wird durch einen Kreis mit einem diagonalen Kreuz dargestellt. Ein aufsteigender Anschlusspunkt wird durch einen Kreis mit einer Raute dargestellt. Die Segmente einer Transition, innerhalb und außerhalb von s_c, können nur auf die erweiterten Zustandsvariablen ihrer jeweiligen hierarchischen Ebene zugreifen.

In dem beispielhaften ROOMchart in Abbildung 3.6 werden die Zustandsgrenzen von B an Anschlusspunkten durchbrochen. Die Transition vom Zustand *2* in *top* zum Zustand *1* in B muss über einen Anschlusspunkt die Zustandsgrenze von *top* durchbrechen. Die Transition vom Zustand *2* in B zu Zustand *1* in *top* ist durch einen Anschlusspunkt segmentiert.

Gruppentransitionen Für die Subzustände eines kompositionellen Zustands dürfen gemeinsame Transitionen definiert werden, die aus jedem Subzustand in einen beliebigen internen oder externen Folgezustand führen dürfen. Diese Transitionen werden für die Gruppe der Subzustände definiert und werden daher Gruppentransitionen genannt. Eine Gruppentransition kann für einen gewöhnlichen kompositionellen Zustand intern oder extern verwendet werden. Für den obersten Zustand *top* können nur interne Gruppentransitionen definiert werden. Im Unterschied zu extern definierten Gruppentransitionen, lösen interne Gruppentransitionen nicht die Eingangs- und Ausgangsaktionen des kompositionellen Zustands aus.

Das Beispiel in Abbildung 3.7 zeigt ein ROOMchart mit einem hierarchischen Zustand B und verschiedenen Gruppentransitionen. Der hierarchische Zustand B besitzt eine externe Gruppentransition, die aus jedem Subzustand von B durch das Ereignis e ausgelöst werden kann, im Zustand *1* von *top* endet und die Aktion x ausführt. Weiterhin enthält das Beispiel eine interne, reflexive Gruppentransition im obersten Zustand *top*, die von jedem Unterzustand von *top* durch das Ereignis f ausgelöst werden kann und die Aktion y ausführt.

Überschreiben von Gruppentransitionen Eine Gruppentransition kann durch Transitionen der gleichen oder einer tieferen Ebene überschrieben werden. Das Überschreiben einer Gruppentransition erfolgt durch die Definition einer Transition niedrigerer Hierarchiestufe und mit gleichem Trigger. In dem Beispiel in Abbildung 3.7 überschreiben die Transitionen zwischen den Zuständen

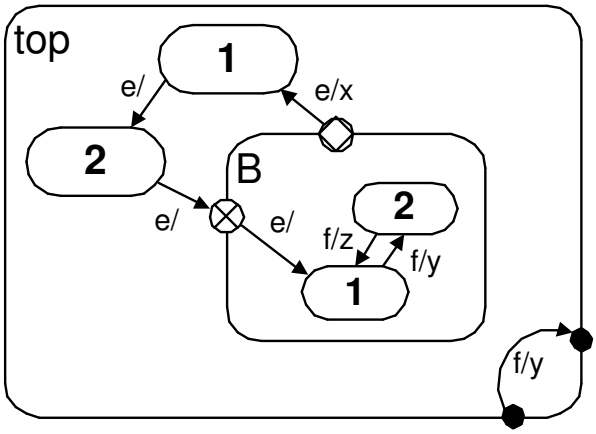


Abbildung 3.7: ROOMchart mit Gruppentransitionen

1 und *2* im hierarchischen Zustand *B* die interne Gruppentransition mit dem Trigger *f* in Zustand *top*.

Gedächtnispunkte Eine Gedächtnisfunktion dient dem temporären Speichern eines Unterzustands, um interne Abläufe eines kompositionellen Zustands zu schützen. Beim aufsteigenden Verlassen einer Hierarchieebene wird der derzeitige Zustand bis zum nächsten absteigenden Betreten durch eine Gedächtnisfunktion gespeichert. Eine Gedächtnisfunktion wird durch eine Transition zu einem hierarchischen Zustand definiert, die keinen expliziten Endzustand besitzt und daher in einem Anschlusspunkt, dem *Gedächtnispunkt* endet. Eine Transition, die zur Definition einer Gedächtnisfunktion führt, wird als Gedächtnistransition bezeichnet. Eine Gedächtnistransition kann in jeden Unterzustand führen, der ein Verlassen des hierarchischen Zustands ermöglicht.

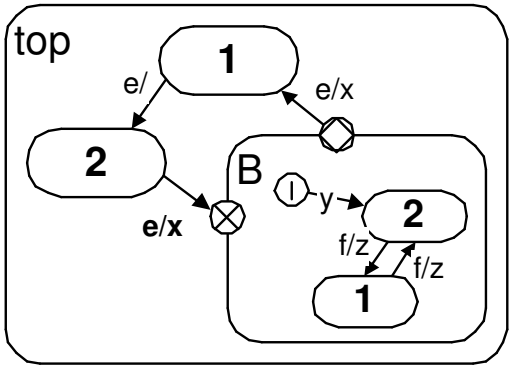


Abbildung 3.8: ROOMchart mit Gedächtnisfunktion

Das Beispiel in Abbildung 3.8 stellt ein ROOMchart mit einem hierarchischen Zustand *B* dar, auf dem eine Gedächtnisfunktion definiert ist. Die Transition von Zustand *2* in *top* zu *B* stellt eine Gedächtnistransition dar. Diese kann in jedem Unterzustand von *B* enden, vorausgesetzt der Unterzustand erlaubt ein direktes Verlassen von *B* oder ist Zielzustand der initialen Transition von B. Die Gruppentransition vom hierarchischen Zustand *B* zum Zustand *1* in *top* ermöglicht ein Verlassen von *B* aus allen Unterzuständen. Die Gedächtnistransition kann daher im Zustand *1* oder Zustand *2* von *B* enden. Die Gedächtnistransition endet im Unterzustand *1*, wenn B zuletzt aus diesem Zustand verlassen wurde. Dies gilt analog für den Subzustand *2* von *B*. Wenn der hierarchische Zustand *B* zum ersten Mal durch die Gedächtnistransition betreten wird, ist der Initialpunkt der Gedächtniszustand und die Initialtransition wird ausgeführt.

3.2.3 Konditionale Konzepte

Für die Konstruktion von bedingten Kontrollstrukturen stehen in ROOMcharts Auswahlpunkte und Wächterbedingungen in Transitionstriggern zur Verfügung. Die Verwendung von bedingten Transitionstriggern wurde bereits an anderer Stelle erläutert. Jeder mit bedingten Transition modellierte Kontrollfluss kann in eine Struktur mit Auswahlpunkten transformiert werden. In einem binären Auswahlpunkt wird eine Bedingung formuliert, welche die Werte *Wahr* und *Falsch* liefert und die dynamisch zur Laufzeit evaluiert wird. Für *Wahr* und für *Falsch* besitzt ein Auswahlpunkt jeweils maximal eine Transition, die auslöst, sobald der entsprechende Fall eintritt. Ein Auswahlpunkt stellt nicht eine Ereigniserwartung dar und wird als Pseudozustand bezeichnet.

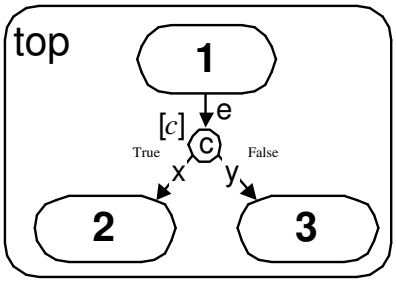


Abbildung 3.9: ROOMchart mit Auswahlpunkt

Das Beispiel in Abbildung 3.9 stellt ein ROOMchart mit Auswahlpunkt dar. Von Zustand *1* führt eine Transition mit dem Trigger e in einen Auswahlpunkt, in dem die Bedingung c evaluiert wird. Wenn c wahr wird, ist der Zustand *2* der Endzustand und es wird eine Aktion x produziert. Wenn c falsch wird, ist der Zustand *3* der Endzustand und es wird eine Aktion y produziert.

Eine detaillierte Beschreibung von Auswahlpunkten kann in [143, 113, 64] gefunden werden. Die dort dargestellten Auswahlpunkte lassen die Definition komplexer Entscheidungen mit beliebig vielen ausgehenden Transitionen zu. Die mit den Transitionen korrespondierenden Bedingungen müssen nicht disjunkt sein. Dies führt theoretisch zu unerwünschtem, nichtdeterministischem Verhalten. Als Anwendungsvoraussetzung für das in dieser Arbeit vorgestellte Verfahren der Testfallgenerierung aus ROOMcharts wird die ausschließliche Verwendung von binären Auswahlpunkten gefordert.

3.3 Echtzeitanwendungen

Der Ansatz der Codegenerierung ist nicht in jedem Fall praktisch durchführbar. Dies kann beispielsweise der Fall sein, wenn eine besonders dichte Systemintegration benötigt wird - was aufgrund der Verwendung virtueller Maschinen problematisch ist. In diesem Fall wird von Selic die manuelle Transformation eines Systems auf die Zielplattform vorgeschlagen. Dies soll in den meisten Fällen zweckmäßig sein, da *Real-Time Object-Oriented Modeling* auf einer Abstraktion bekannter Echtzeit-Ansätze [44, 144] basiert.

3.3.1 Echtzeitumgebungen

Eine Implementationsplattform besteht aus einer Programmierumgebung, dies beinhaltet eine Programmiersprache und eine Laufzeitumgebung. Eine Echtzeitumgebung bietet weiterhin nebenläufige Programmierung (engl. *Concurrent Programming Paradigm*), die entweder direkt oder in Form einer Bibliothek von der Programmiersprache unterstützt wird.

Echtzeit-Programmiersprachen Für die Programmierung von Echtzeitsystemen werden bevorzugt blockorientierte, imperative Programmiersprachen eingesetzt [144, 1, 74, 12]. Objektorientierte Programmiersprachen finden in eingeschränkter, oftmals experimenteller Form Anwendung [79].

Multitasking Eine Echtzeitumgebung sollte eine Form leichtgewichtiger Nebenläufigkeit oder *Threading* (engl. *thread* - Faden, Erzählstrang, Gedankengang)[144] unterstützen. Jeder *Thread* benötigt einen eigenen Kontext in dem Zustandsinformationen während Ruhephasen gespeichert werden. Ein System mit dynamischer Struktur muss die dynamische Erzeugung und Zerstörung von *Threads* erlauben [105].

Thread Synchronisation Die Überlappung mehrerer Threads auf einem System mit einem Prozessor kann zu gegenseitigen Störungen der Thread-Abläufe führen. Dies wird häufig durch den Einsatz fehleranfälliger und aufwändiger Ausschlusstechniken vermieden. Ein in *Real-Time Object-Oriented Modeling* realisiertes System sollte keine explizite Synchronisation von Threads benötigen, da es theoretisch keinen gemeinsamen Speicher gibt und die Verarbeitungen nicht unterbrochen werden können (*engl. Run-To-Completion*). Wenn die eingesetzte Programmiersprache gemeinsame Speicherverwaltung unterstützt, ist allerdings eine gegenseitige Beeinflussung verschiedener Aktoren nicht ausgeschlossen. Die gemeinsame Nutzung von Speicher kann wegen geringer Ressourcen oder zur Leistungsverbesserung notwendig sein. In diesen Fällen ist die Anwendung von Synchronisationsmaßnahmen, wie beispielsweise Semaphoren, die Definition kritischer Speicherregionen oder Wächterfunktionen, empfehlenswert.

Interprozedurale Kommunikation Die Kommunikation erfolgt durch diskrete Nachrichtenpakete, die im Kontext eines Threads erzeugt und in den Kontext eines anderen Threads transferiert werden. Im Idealfall sollte das Nachrichtenpaket nach dem Verschicken nicht mehr vom Sender zugreifbar sein. Viele Echtzeit-Rechnerarchitekturen unterstützen das Nachrichtenkonzept. Die einfachste Form der Kommunikation stellen asynchrone Nachrichten dar. Da mehrere Nachrichten quasi gleichzeitig einen Empfänger erreichen können, muss eine Art von Warteschlange vom zugrunde liegenden Rechnerkern unterstützt werden. Dies ist häufig in Form von *Mailbox-*, *Socket-* oder *Pipe*-Konzepten gegeben. Wenn ein Entwurf synchrone Kommunikation zwischen nebenläufigen Komponenten verlangt, muss der Rechnerkern entweder *Rendezvous* oder *Remote Procedure Call* unterstützen [144, 143, 105].

Scheduling Über die Semantik von Nachrichten- oder Ereignisprioritäten werden in *Real-Time Object-Oriented Modeling* nur schwache Annahmen gemacht. Für die meisten Entwürfe sind die *Scheduling*-Konzepte (engl. *schedule* - Zeitplan) der Laufzeitumgebung zweckmäßig [144, 143, 105]. Allerdings kann die Forderung der strikten Verarbeitungsreihenfolge nach Ereignisprioritäten problematisch sein, da die meisten Rechnerkerne nach *Thread*-Prioritäten terminieren. In solchen Fällen kann eine Erweiterung oder ein Ersatz der *Scheduling*-Vorrichtung der Zielplattform notwendig sein. Eine mögliche Modifikation könnte der dynamische Abgleich von *Thread*-Prioritäten auf die jeweils höchsten Ereignisprioritäten sein. In verschiedenen Arbeiten haben Freedman et al. die temporale Analysierbarkeit von *Real-Time Object-Oriented Modeling* nachgewiesen [141, 140].

Speicherverwaltung Alle Speicherreferenzen eines nebenläufigen Systems sind lokal auf Aktoren beschränkt. Hieraus leitet sich die Forderung nach individuell geschützten Speicherräumen für individuelle Aktoren ab. Wenn eine Programmiersprache ausschließlich gemeinsame Speicherverwaltung unterstützt, würde ein Schema für den Speicherschutz die Fehleranfälligkeit der Implementierung signifikant reduzieren [105, 144].

Implementierung Es gibt zwei Wege einen Entwurf in *Real-Time Object-Oriented Modeling* auf einem Echtzeitsystem zu implementieren. Der Entwurf wird gemeinsam mit der virtuellen Laufzeitumgebung gebunden und für die Zielplattform kompiliert. Dieser Ansatz bietet eine hohe Zuverlässigkeit und ist gegenüber einem manuellen Ansatz deutlich kosteneffizienter. Die traditionelle Alternative sieht die Anwendung von diversen Abbildungsvorschriften vor, die den ursprünglichen Entwurf in eine dem Zielsystem angepasstere Implementierung transformieren. Dieser Ansatz findet immer Anwendung, wenn keine auf das Zielsystem passende virtuelle Maschine verfügbar ist, oder aus Leistungsgründen eine an das Zielsystem besonders angepasste Implementierung notwendig ist. Die Eigenentwicklung einer virtuellen Maschine kann in manchen Fällen eine zweckmäßige Kombination der Vorteile beider Ansätze darstellen [143, 112, 97].

3.4 Testkonzept

Die qualitätsorientierte Softwareentwicklung mit Modul-, Integrations- und Systemtest für *Real-Time Object-Oriented Modeling* erfordert die Entwicklung eines technischen Konzepts für die Durchführung und die Ziele jeder einzelnen Qualitätssicherungsphase. Zu diesem Zweck werden im Folgenden die in den Phasen des Modul-, Integrations- und Systemtest wichtigen Merkmale untersucht und ein Konzept für den Modultest vorgestellt.

3.4.1 Modultest

Während der Modultestphase in einem modellbasierten Softwareentwicklungsprozess und der Verwendung von *Real-Time Object-Oriented Modeling* wird das Verhalten jedes Aktors eines Systems separat geprüft. Die Schnittstelle eines Aktors wird durch seine Ports und deren Protokolle definiert. Da Protokolle unabhängig von Aktoren verändert werden können, sollten die im Protokoll definierten Ereignissequenzen während des Modultests eines Aktors nicht berücksichtigt werden. Diese Sequenzen können alternativ in Form funktionsorientierter Tests geprüft werden. Erst durch den rigiden Einsatz von Anforderungsverfolgungsmethoden [54] können Aktoren mit Protokollen getestet werden, da die Auswirkungen der Änderung eine Protokolls sichtbar werden und zusätzliche Tests in Konsequenz durchgeführt werden können.

Die Testfälle können aus allen im Entwicklungsprozess entstandenen Dokumenten abgeleitet werden, die einen Bezug zum Aktor besitzen. Dies kann auch Analyse- und Entwurfsdokumente umfassen, die beispielsweise in Form von Lastenheften oder Sequenzdiagrammen vorliegen. Zusätzlich kann die Struktur des Aktors für die Testfallgenerierung oder für die Definition von Testvollständigkeitskriterien dienen.

Jeder Aktor wird in einem speziellen Testrahmen geprüft, der alle durch den Aktor genutzten Dienste simulieren und alle vom Aktor angebotenen Dienste abfragen kann. Zu diesem Zweck besitzt der Testrahmen eine aktive Komponente - den Testtreiber - und mehrere passive Komponenten - die *Stubs*.

Ein Aktor stellt eine sequentielle Komponente in einem nebenläufigen System dar und kann eine komplexe Funktionalität realisieren. Daher ist der separate Test eines Aktors auf der Zielplattform bereits im Modultest sinnvoll und möglich. Im Gegensatz zur Entwicklungsplattform ist die vollständige Beobachtbarkeit der internen Strukturen und des internen Verhaltens eines Aktors auf der Zielplattform nicht gesichert. Für diese Art von Tests bieten sich daher funktionsorientierte und automatenbasierte Tests mit hoher Portabilität der Testfälle an. Die Generierung solcher Tests aus der Struktur von ROOMcharts wird in dieser Arbeit vorgestellt.

Da es sich bei einem Aktor um eine Komponente mit ausschließlich sequentiellem Verhalten handelt, sind im Modultest von den Echtzeitkonzepten in *Real-Time Object-Oriented Modeling* nur das Antwortzeitverhalten und der Speicherverbrauch entscheidend. Beides kann bei einem Test auf der Zielplattform geprüft werden.

In Abbildung 3.10 ist das Schema eines beispielhaften Testrahmens für den Modultest in *Real-Time Object-Oriented Modeling* dargestellt. Ein Aktor ist über seine drei Ports mit dem Testtreiber und zwei *Stubs* verbunden. Der Aktor ist auf der Zielplattform installiert. Der Testrahmen ist auf einer Testplattform installiert, die eine automatische Protokollierung des Tests erlaubt und während der Testausführung mit dem Aktor auf der Zielplattform kommuniziert.

3.4.2 Integrationstest

Während der Integrationstestphase wird die Kommunikation zwischen den Aktoren des Systems geprüft. Jeder Aktor stellt eine nebenläufige Komponente dar. Die Integration des Systems erfolgt schrittweise und erfordert für jeden dieser Schritte die Erstellung eines Testrahmens. In dieser Prüfungsphase sind die Zeitkonzepte von *Real-Time Object-Oriented Modeling* und des Systems für die Prüfung entscheidend. Da diese Arbeit ausschließlich der Betrachtung von Implementation und Modultest dient, wird auf eine detailliertere Darstellung verzichtet. Allerdings eignet sich der präsentierte Ansatz der Testfallgenerierung mit einem *Model Checker* auch hervorragend für den Test nebenläufiger Systeme. Daher ist eine Ausweitung dieser Methode auf den Integrationstest

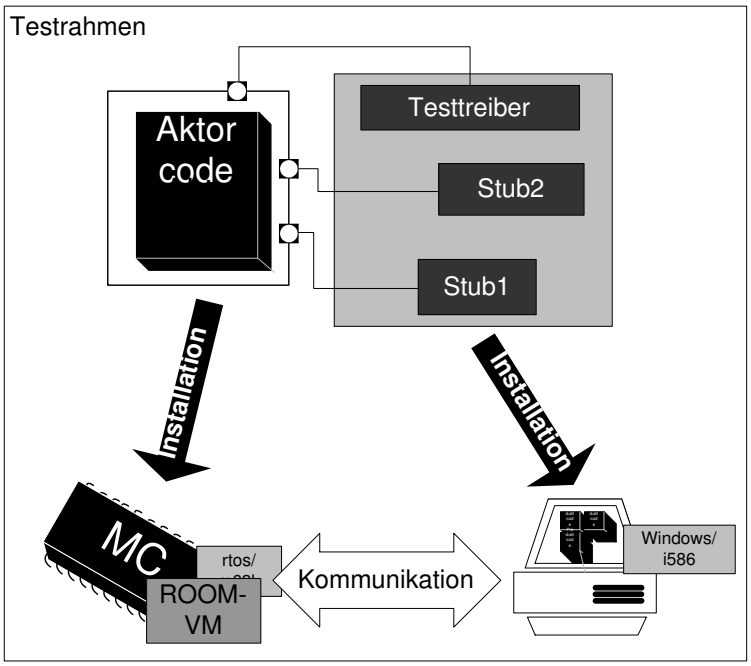


Abbildung 3.10: Aktor im Testrahmen

für *Real-Time Object-Oriented Modeling* vielversprechend und für nachfolgende Forschung empfehlenswert.

3.4.3 Systemtest

Der Systemtest wird unabhängig von der verwendeten Entwicklungsmethode durchgeführt. Eine Anpassung bekannter Ansätze und Methoden des Systemtest an *Real-Time Object-Oriented Modeling* erscheint daher nicht notwendig.

Kapitel 4

Model Checking

Eine der ersten Anwendungen von endlichen Zustandsautomaten und formalen Methoden in der Softwareentwicklung war die Modellierung und die Analyse von Kommunikationsprotokollen und Schaltkreisen [17, 102]. Zur gleichen Zeit wurde auch der Ruf nach effektiven Qualitätssicherungsmethoden für die neuen Spezifikationsmittel laut. Ein früher Algorithmus zur Verifikation von zustandsbasierten Kommunikationsprotokollen basierte auf der Suche innerhalb des von der Spezifikation aufgespannten Zustandsraums [163]. Neben der Entwicklung automatenbasierter Testmethoden [66, 55, 59], die an anderer Stelle ausführlich erläutert werden, wurde mit *Model Checking* ein erfolgreicher, formaler Ansatz zur Prüfung komplexer und nebenläufiger Kommunikationssysteme entwickelt.Mit *Model Checking* wird die algorithmische Verifikation von formalen Modellen gegen temporal logische Spezifikationen bezeichnet. Diese Technik wurde in zwei Ausprägungen unabhängig voneinander und nahezu gleichzeitig durch Clarke und Emerson [38, 45] sowie Quielle und Sifakis [129] entwickelt.

Wie bei nahezu allen formalen Methoden stellt auch bei der praktischen Nutzung des *Model Checking* die Komplexität ein gravierendes Problem dar. Bereits bei der ausschließlichen Betrachtung endlicher Zustandsautomaten ist die Anzahl zu untersuchender Zustände im Lösungsraum oftmals sehr hoch. Bei der Betrachtung erweiterter Zustandsautomaten verschärft sich dieses Problem drastisch. Zudem multiplizieren sich die Komplexitäten sequentieller Automaten in einem nebenläufigen System. Daher benutzen *Model Checker* typischerweise eine Vielzahl von Heuristiken, um diese *Explosion des Zustandsraums* (*engl. State Space Explosion*) auf ein berechenbares Maß zu reduzieren.

Die theoretischen Berechenbarkeitsgrenzen des *Model Checking* [116, 118] müssen nicht die praktische Nutzung dieses Ansatzes zur Lösung von Testproblemen verhindern. Die Komplexität wird von der zugrunde liegenden Logik, von der Möglichkeit zur Abstraktion und von dem Modell selbst beeinflusst [90, 118]. Um Komplexitäten zu bewerten, muss entschieden werden, wie diese Eingangsgrößen gemessen werden. Dies geschieht in der Regel mittels verhältnismäßig unpräziser Maße, beispielsweise der Landau-Notation [82]. Eine Komplexitätsbetrachtung kann für die Beurteilung und die Verbesserung von Transformationen eines Entwurfs in ein Testmodell genutzt werden.

Die Testfallgenerierung mittels *Model Checking* wurde bereits in verschiedenen Arbeiten vorgestellt, beispielsweise in [130, 80, 32, 106, 63]. Die Eignung der *Model Checking* Werkzeuge SAL, SMV, SPIN und Uppaal wurde bereits untersucht. Der *Model Checker* SMV basiert auf *Computational Tree Logic (CTL)* und hat sich zur Testfallgenerierung bewährt [75]. Der *Model Checker* SPIN basiert auf *Linear Temporal Logic (LTL)* und Tiefensuche. Da SPIN aufgrund der Tiefensuche tendenziell längere Testfälle als andere *Model Checker* mit Breitensuche erzeugt, hat er sich für Testzwecke als vergleichsweise ungeeignet erwiesen [52]. Der im Vergleich zu SMV und SPIN relativ neue und auf eingeschränkter *CTL* basierende *Model Checker* Uppaal hat sich bereits zur Generierung von Testfällen mit Zeitbedingung bewährt [106, 69, 68, 101, 32, 103]. Ebenfalls eine neuere Entwicklung liegt mit SAL vor, wobei es sich um eine ganze Bibliothek von Methoden und Werkzeugen für die symbolische Analyse handelt. Dieses Paket beinhaltet verschiedene *Model Checker* die CTL und LTL beherrschen.

Es ist fraglich, ob der Einsatz anderer *Model Checker* einen Leistungszuwachs gegenüber Up-

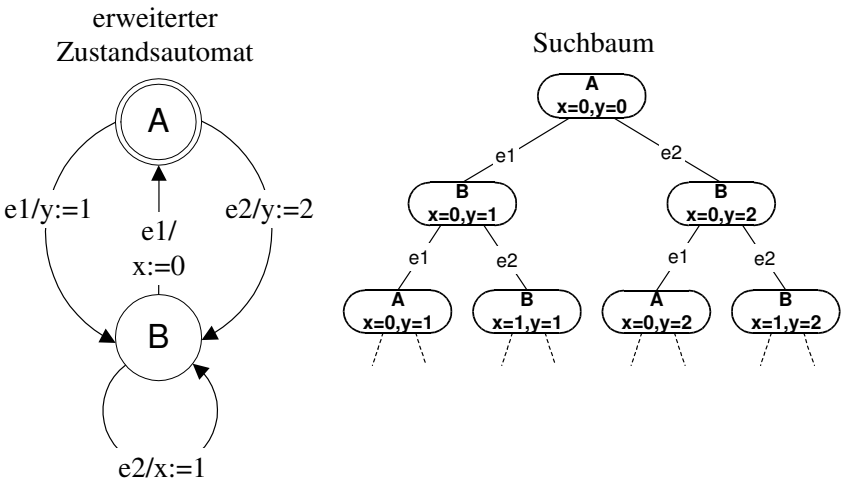


Abbildung 4.1: Erweiterter Automat mit Suchbaum

paal bewirken würden. Eine vergleichende Untersuchung könnte in einer anschließenden Arbeit durchgeführt werden. Neben der Leistung spielen in diesem Zusammenhang auch die System-Definitionssprache und das Eingabeformat eine wesentliche Rolle bei der Bewertung eines *Model Checkers*. Mit Uppaal bietet sich hier ein klares Konzept mit XML als Austauschformat.

4.1 Suche im Zustandsraum

Während der Analyse erstellt ein *CTL Model Checker* den Lösungsraum in Form eines Zustands-Transitions-Graphen (*engl. state transition graph*), der auch als Suchbaum bezeichnet wird [118]. Jeder Zustand im endlichen Zustandsautomaten, im Folgenden als *Location* (dt. Ort, Stelle, Lage) bezeichnet, wird mit jedem möglichen Wert der Systemvariablen zu einem Zustand im Suchbaum kombiniert. Die Zweige im Suchbaum korrespondieren mit der Kontrollstruktur im System. Der Zustand eines erweiterten Zustandsautomaten ist ein Tupel (var_1,...,var_n,L_i) der *Location* L_i und der aktuellen Werte der n erweiterten Zustandsvariablen var_1,...,var_n.

Das Beispiel in Abbildung 4.1 stellt einen erweiterten Zustandsautomaten mit Suchbaum dar. Der initiale Zustand A des Automaten ist durch einen Doppelkreis gekennzeichnet. Der Automat besitzt die *Locations* A und B und reagiert auf die Ereignisse *e1* und *e2*. Der Automat ist um die ganzzahligen Variablen x und y erweitert, die in den Transitionen verarbeitet werden. Der Suchbaum enthält alle Zustände dieses Systems, die aus den Kombinationen der Werte der Variablen x und y sowie den *Locations* A und B gebildet werden.

Ein endliches Zustandssystem mit der Menge initialer Zustände I kann mit dem Algorithmus in 4.2 durchsucht werden. Nach diesem Algorithmus wird jeder Zustand mindestens einmal besucht, der ausgehend von einem initialen Zustand erreichbar ist.

Dieser Algorithmus spezifiziert nicht, wie der Zustand s aus *New* in Zeile 3 gewählt wird. Es wird auch nicht spezifiziert, wie *s'* in *New* gespeichert wird. Eine Möglichkeit s zu wählen und *s'* zu speichern kann die Implementierung von *New* als Warteschlange mit Ausgabe nach *First-In-First-Out (FIFO)* [43] sein. Diese Suchstrategie wird *Breitensuche* (engl. *Breadth First Search, BFS*) genannt und liefert eine Menge kürzester Pfade [118, 117]. Eine weitere Strategie ist die Implementierung von *New* als Stapel mit Ausgabe nach *Last-In-First-Out (LIFO)* [43]. Diese Strategie wird *Tiefensuche* genannt (engl. *Depth First Search, DFS*) und liefert tendenziell längere Pfade als die Breitensuche. Es ist unentschieden, welche Strategie für den allgemeinen Fall die effizientere ist. Für den Fall der Prüfung von Anforderungen, die nur eine vergleichsweise kleine Teilmenge des Lösungsraums betreffen, hat sich *CTL Model Checking* mit Breitensuche bewährt. Eine Form solcher Anforderungen stellen die Erreichbarkeit bestimmter Zustände dar [118] und dienen zur Abbildung von Testpfaden.

```
1   New enthält die Menge initialer Zustände I, Old ist leer
2   While New nicht leer do
3       Wähle einen Zustand s aus New
4       Entferne s aus New
5       Addiere s zu Old
6       For each Transition t ist auslösbar in s do
7           Erzeuge s', durch Anwendung von t in s
8           If s' ist nicht in Old oder New, dann
9               Addiere s' zu New
```

Abbildung 4.2: Suchalgorithmus für Zustandsräume

4.2 Problem der Explosion des Zustandsraums

Das wichtigste Problem bei der praktischen Verwendung des *Model Checking* stellt die kombinatorische Explosion des Zustandsraums dar. Der Zustandsraum eines Systems ergibt sich aus dem Produkt aller Zustände, Variablen der nebenläufigen Systemkomponenten. Eine nebenläufige, sequentielle Systemkomponente wird in der Regel durch einen endlichen, oftmals erweiterten Zustandsautomaten beschrieben. Die Größe des Zustandsraums l von j Systemkomponenten ist das Produkt der Wertebereiche der Variablen und den Zuständen aller Automaten, ausgedrückt durch

$l = \prod_i vardim_i * \prod_j locations_j$

Mit $vardim_i$ wird der Wertebereich jeder Variablen var_i bezeichnet. Mit $locations_j$ wird die Anzahl der Entwurfszustände im Automaten m_j bezeichnet. Um diese Komplexität zu beherrschen oder zu reduzieren werden verschiedene Strategien verwendet, die in diesem Kapitel erläutert werden [118, 117, 116]. Weiterhin werden in dieser Arbeit Methoden für die Komplexitätsreduktion für die Verwendung eines *CTL Model Checkers* mit Breitensuche behandelt.

4.3 Temporale Logik

Für die Spezifizierung nebenläufiger Systeme hat sich temporale Logik bewährt. Die zeitliche Ordnung von Ereignissen wird beschrieben, ohne Zeit explizit einführen zu müssen. Die Klassifikation von temporal-logischen Kalkülen erfolgt entsprechend ihrer jeweiligen zugrunde liegenden Annahme, ob Zeit eine lineare oder eine verzweigende Struktur besitzt. In verschiedenen Arbeiten wurden Ansätze entwickelt, temporale Logik für die Analyse von Programmen zu nutzen [125, 83, 29]. Der Ansatz von Pnueli [125] befasste sich als erster mit der Analyse nebenläufiger Systeme und wurde von Bochmann [16] sowie von Malachi und Owicki [95] für die Verifikation von Steuerkreisen weiterentwickelt. Diese manuellen Verfahren waren in der Praxis aufgrund der Komplexität nur schwer anzuwenden. Die Einführung von *Model Checking* basierend auf temporaler Logik [38] erlaubt die Automatisierung dieser Ansätze.

Der Algorithmus von Clarke und Emerson für *Branching-Time Logic* CTL ist polynomial bezüglich der Größe des Modells und linear bezüglich der Länge der Spezifikation.

Für die lineare temporale Logik LTL zeigten Sistla und Clarke [149], dass dieses Problem *PSPACE*-vollständig [116] ist. Die Untersuchungen von Pnueli und Lichtenstein [91] zeigten jedoch, dass die Komplexität von *LTL Model Checking* mit sich linear zu der Größe des globalen Zustands-Transitions-Graphen verhält. Dies führte zu der Annahme, das LTL für kurze Formeln eine akzeptable Komplexität besitzt.

Mit CTL* wurde von Clarke, Emerson und Sistla eine ausdrucksstarke Logik vorgestellt, die Merkmale von CTL und LTL miteinander kombiniert und von der gleichen Komplexitätsklasse wie LTL ist [39].

Die symbolische Darstellung von Zustands-Transitions-Graphen [118, 28, 96] basiert auf geordneten binären Entscheidungsdiagrammen (engl. *ordered binary decision diagram,* OBDD). Ein OBDD bietet eine kanonische Darstellung boolescher Formeln, die deutlich kompakter als übliche Normalformen ist. Für OBDDs existieren darüber hinaus effiziente Analyse- und Berechnungsalgo-

rithmen. Durch die Verwendung des CTL *Model Checking* [38] mit OBDDs wurden bereits Systeme mit großen Zustands-Transitions-Graphen von mehr als 10^{120} Zuständen verifiziert [28].

4.3.1 Komplexitätsreduktion

Die Verifikation von Software mit *Model Checkern* verursacht im Vergleich mit Hardware oftmals schwerwiegende Komplexitätsprobleme. Dies liegt an der tendenziell geringeren Strukturierung und an der Verschärfung des Problems der Explosion des Zustandsraums bei der Verifikation nebenläufiger, asynchroner Softwaresysteme.

Partial Order Reduction Für nebenläufige Systeme ist *Partial Order Reduction* [158, 57] eine der erfolgreichsten Techniken, die Größe des Zustandsraums zu reduzieren. Diese Techniken nutzen die Unabhängigkeit von nebenläufig auftretenden Ereignissen. Nebenläufig auftretende Ereignisse sind unabhängig voneinander, wenn sie im gleichen globalen Zustand in beliebiger Reihenfolge auftreten dürfen. Das *Interleaving Model* eignet sich besonders für die Darstellung nebenläufiger Software. In dieser Darstellung werden Ereignisse einer Berechnung in einer *Interleaving Sequence* angeordnet. Nebenläufig auftretende Ereignisse erscheinen in dieser Folge in beliebiger Ordnung zueinander. Die meisten logischen Kalküle für die Spezifikation von Systemeigenschaften unterscheiden zwischen *Interleaving* Sequenzen, in denen zwei nebenläufige Ereignisse in unterschiedlicher Reihenfolge ausgeführt werden. Dies führt zu einem unnötigen, und sehr starken Anstieg der Komplexität. Mit *Partial Order Reduction* wird die Anzahl von *Interleaving* Sequenzen reduziert.

Modularität Durch die Ausnutzung der *Modularität* in vielen Softwaresystemen kann die Komplexität ebenfalls deutlich reduziert werden [118]. Die Spezifikation solcher Systeme kann häufig in beherrschbare Komponenten unterteilt werden. Eine typische Anwendungsform ist die Verifikation genau jenes Teils eines Modells, den die zu prüfenden Anforderungen betreffen.

Datenabstraktion Eine der wichtigsten Techniken zur Komplexitätsreduktion ist *Datenabstraktion* [40, 118]. Diese Technik ist besonders für die Prüfung von reaktiven System mit Daten von Bedeutung. Durch symbolisches *Model Checking* wird die Behandlung von Systemen mit nichttrivialen Operationen auf Daten ermöglicht. Trotzdem ist die Komplexität vieler reaktiver Systeme mit Daten erheblich. Die Abstraktion von Daten basiert auf der Beobachtung, dass Systemspezifikationen mit Daten tendenziell eher einfache Operationen beschreiben. Eine Datenabstraktion wird kann durch eine Abbildung von realen Systemdaten auf eine kleinere Menge abstrakter Daten vorgenommen werden. Durch die Erweiterung dieses Ansatzes auf Zustände und Transitionen können Eigenschaften auf einem deutlich kleineren und einfacheren System nachgewiesen werden.

Symmetrie Durch Ausnutzung von *Symmetrie* kann das Problem der Explosion des Zustandsraums weiter reduziert werden [118]. In vielen Systemen sind replizierte Komponenten enthalten, die mittels einer Äquivalenzrelation für eine Systemreduktion genutzt werden können.

Induktion Mittels *Induktion* kann eine automatische Analyse für ganze Familien von endlichen Zustandssystemen erfolgen [118]. Mit der Induktion soll gezeigt werden, dass jedes Mitglied einer Familie eine bestimmte Anzahl temporal-logischer Anforderungen erfüllt.

Für die Testfallgenerierung basierend auf sequentiellen, erweiterten Automaten bietet sich zur Komplexitätsreduktion vor allem die Verwendung von Datenabstraktion an. Auf Systemebene kann auch im Rahmen des Modultest mittels Induktion, Symmetrie und Modularität die Komplexität deutlich reduziert werden, da diese Methoden gut auf Aktor- und Protokollklassen und deren Assoziationen anwendbar sind. Der Ansatz der *Partial Order Reduction* kann im Integrationstest angewendet werden, wenn die Interaktion der Module geprüft wird. Dagegen wird im Modultest die Reihenfolge von Ereignissen von der Testmethode vorgegeben, und das beschriebene Problem der *Interleaving* Sequenzen tritt nicht auf.

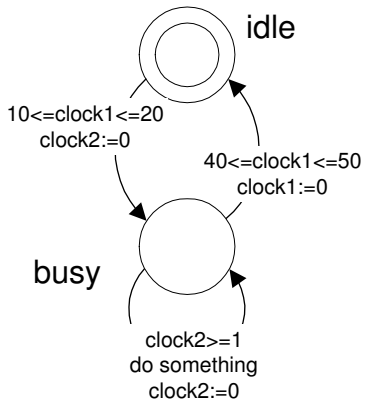


Abbildung 4.3: Beispielhafter zeitattributierter Automat

4.3.2 Zeitattributierte Automaten

Ein zeitattributierter Automat (*engl. timed automaton*) ist eine Gebilde aus einer Menge von Zuständen, einer Menge von Transitionen und einer Menge von Uhren [4, 3]. Eine Uhr ist ein reale Variable, die mit 0 initialisiert wird, sobald das System startet. Mit dem Systemablauf erhöhen die Uhren ihre Werte synchron und mit der gleichen Rate. Üblicherweise verstreicht die Zeit nur in Zuständen, während sie in Transitionen nicht vergeht. Eine Bedingung über eine Uhr wird benutzt, um das Verhalten des Automaten einzuschränken. Eine Uhr kann durch eine Transition zurückgestellt werden. Ein Zustand und eine Transition können Zeitbedingungen definieren. Die Zeitbedingung eines Zustands ist eine Invariante. Die Zeitbedingung in einer Transition beschreibt absolut oder relativ einen Zeitpunkt, zu dem die Transition ausgelöst werden kann. Jede Transition kann eine Beschriftung besitzen, welche für die Synchronisation mit Transitionen anderer zeitattributierter Automaten dient. Jeder zeitattributierte Automat muss einen initialen Zustand besitzen.

Das Beispiel in Abbildung 4.3 stellt einen zeitattributierten Automaten mit den Uhren *clock1* und *clock2* dar. Die Transition vom Zustand *idle* zum Zustand *busy* wird ausgelöst, sobald die Uhr *clock1* einen Wert zwischen 10 und 20 einnimmt. Im Zustand *busy* wird die reflexive Transition ausgeführt, solange der Wert der Uhr *clock1* nicht im Bereich [40,50] liegt. Die reflexive Transition kann ausgelöst werden, wenn der Wert der Uhr *clock2* größer oder gleich 1 ist. In der Transition wird die Aktion *do something* ausgeführt und anschließend wird die Uhr *clock2* auf 0 zurückgesetzt. In der Zeitspanne von 0 bis 1 von *clock2* ist ein Auslösen der Transition von Zustand *busy* zu Zustand *idle* möglich, wenn *clock1* einen Wert im Intervall [40,50] einnimmt.

Es wurden eine Anzahl von Werkzeugen und Methoden für das *Model Checking* auf Basis zeitattributierter Automaten entwickelt, die der Modellierung und Verifikation von Echtzeitsystemen dienen [87, 162, 31].

4.3.3 Zeitattributierte Uppaal-Automaten

Die Uppaal-Modellierungssprache erlaubt die Beschreibung von Netzen aus zeitattributierten Automaten. Es wird eine erweiterte Form von zeitattributierten Zustandsautomaten unterstützt. Eine detaillierte Beschreibung von zeitattributierten Uppaal-Automaten kann in [87, 11, 68] gefunden werden.

Netzwerke zeitattributierter Automaten Eine parallele Komposition $A_1|...|A_n$ ist ein Netzwerk aus zeitattributierten Automaten $A_1...A_n$, die Prozesse genannt werden. Die zeitattributierten Automaten werden über parallele Kompositionsoperatoren kombiniert. Ähnlich *Calculus for Communicating Systems* (Abk. CCS) [117, 118] werden die Operatoren *!* und *?* benutzt, um die Synchronisation von zwei Transitionen verschiedener Prozesse zu definieren. Das Ausgabealphabet

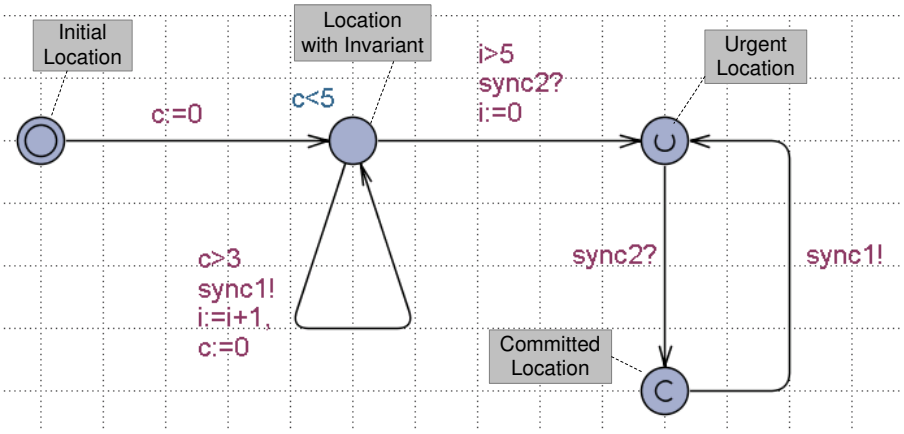


Abbildung 4.4: Beispielhafter UPPAAL-Automat

eines Automaten wird mit dem Operator *!* assoziiert, und das Eingabealphabet wird mit dem Operator *?* assoziiert. Der Zustand eines Netzwerks ist ein Paar $<l,v,u>$. Mit l wird der Vektor der aktuellen *Locations* bezeichnet. Die Werte der Variablen des Systems werden mit v bezeichnet. Mit u werden die aktuellen Werte der Uhren des Systems bezeichnet.

Variablen In Uppaal werden ganzzahlige Variablen mit begrenzbaren Wertebereichen und Konstanten unterstützt, deren Wertebelegungen in der Art einer imperativen Programmiersprache definiert werden können.

Urgent Channels Um ein Modell mit dringlicher (*engl. urgent*) Synchronisation von Transitionen zu entwerfen, unterstützt Uppaal das Konzept dringlicher Kommunikationskanäle (*engl. urgent channels*) [162]. Eine dringliche Synchronisation wird bevorzugt behandelt und kann nicht verschoben werden. Transitionen, die über dringliche Kommunikationskanäle synchronisieren, dürfen keine Wächterbedingung besitzen.

Urgent Locations In einer *Urgent Location* darf keine Zeit vergehen. Daher kann eine Transition des Systems unverzüglich nach Erreichen einer *Urgent Location* ausgelöst werden. Eine *Urgent Location* wird durch einen Kreis mit einem **U** dargestellt.

Committed Locations Um die Modellierung von sequentiellen, atomaren Aktionen zu ermöglichen, bietet Uppaal die Definition von *Committed Locations*. In einer *Committed Location* darf keine Zeit vergehen und es muss unverzüglich eine ausgehende Transition ausgelöst werden. Dies ist vergleichbar mit einem Pseudozustand, da der Automat nicht auf die nächste Eingabe wartet, sondern unverzüglich mit der Bearbeitung fortfährt. In die Abschätzung der Komplexität gehen *Committed Locations* nicht ein. Eine *Committed Location* wird durch einen Kreis mit einem **C** dargestellt.

Das Beispiel in Abbildung 4.4 stellt einen zeitattributierten Uppaal-Automaten dar. Der Automat besitzt eine Uhr c und eine ganzzahlige Variable i. Beginnend im initialen Zustand wird die Transition zum Zustand mit Invariante ausgelöst und c auf 0 zurückgesetzt. Solange c kleiner 5 ist verbleibt der Automat in diesem Zustand. Wenn $c>3$ gilt wird die reflexive Transition ausgelöst und *sync1* ausgegeben, inkrementiert i und setzt c auf 0 zurück. Wenn $i>5$ und eine Eingabe *sync2* erfolgt, wird die Transition in die *Urgent Location* ausgelöst und i auf 0 zurückgesetzt. Im nächsten Schritt nach Eingabe von *sync2* wird die Transition zur *Committed Location* ausgelöst. Der Automat verbleibt nicht in der *Committed Location*, sondern fährt sofort mit dem Auslösen der Transition zurück zur *Urgent Location* und der Ausgabe von *sync1* fort.

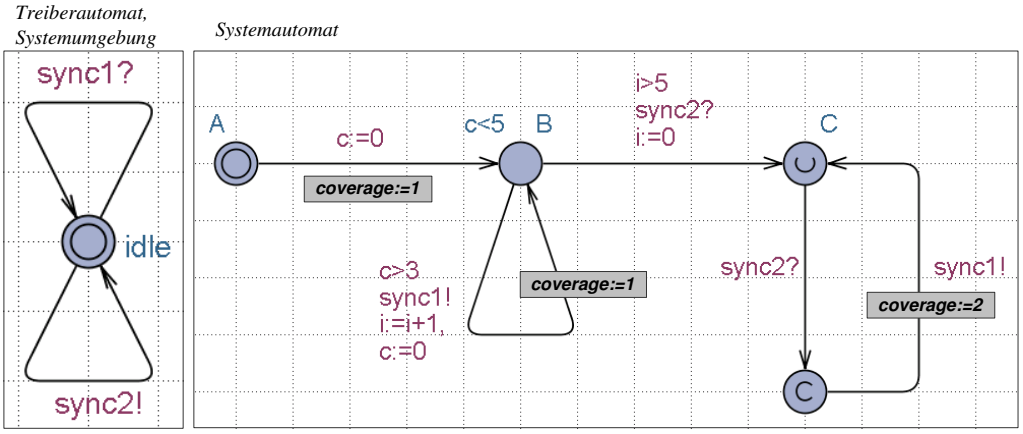


Abbildung 4.5: Beispielhaftes Uppaal-Testmodell

4.3.4 Uppaal-Logik

Die Uppaal Anforderungsbeschreibungssprache unterstützt fünf Typen von temporallogischen Ausdrücken. Mit p wird eine logische Bedingung spezifiziert. Mit s_i werden Zustände bezeichnet.

Possibly $E\Diamond\, p$

Es existiert ein Pfad $< s_0, s_1, ..., s_n >$, mit s_0 als Anfangszustand, in dem p in s_n gilt.

Invariantly $A\Box\, p$

Auf jedem Pfad $< s_0, s_1, ..., s_n >$ gilt in jedem Zustand s_i die Bedingung p.

Potentially Always $E\Box\, p$

Es existiert ein Pfad $< s_0, s_1, ..., s_n >$ auf dem p in jedem Zustand gilt, und entweder $n = \infty$ oder für $s_n = (var_n, L_n)$ folgendes gilt:

* für jedes $d : (var_n + d)$ der erweiterten Zustandsvariablen gilt p

* es existieren keine Ausgangstransitionen in s_n

Eventually $A\Diamond\, p$

Für alle Pfade $< s_0, s_1, ..., s_n >$, mit einem Anfangszustand s_0, gilt die Bedingung p in einem Zustand s_n.

Implikation $p \rightarrow q$

Die Bedingung q gilt, wenn die Bedingung p gilt.

Für die Generierung von Testfällen wird im Folgenden der Operator *Possibly* $E\Diamond\, t$ benutzt, da in einem Testmodell ein Pfad gesucht wird, der eine bestimmte Testbedingung t erfüllt. Andere Beispiele für die Verwendung der Uppaal-Logik können in [11, 68, 87] gefunden werden.

4.4 Testfallgenerierung

Die Generierung von Testfällen mit Uppaal erfolgt mithilfe eines Treiberautomaten, verschiedenen Instrumentierungen und Testanforderungen. Ein System wird für *Model Checking* immer mit einer Systemumgebung entworfen. Im Falle der Testfallgenerierung stellt die Systemumgebung einen Testrahmen dar, der aus Treibern und anderen Hilfsobjekten bestehen kann. Für die Testfallgenerierung auf Basis eines einzelnen sequentiellen Automaten ist ein zusätzlicher Testtreiber ausreichend, der als Empfänger und Sender von Nachrichten des Systemautomaten dient. Die Instrumentierung markiert Elemente des Systemautomaten und dient der Definition von Anforderungen, welche die generierten Testpfade erfüllen müssen.

Das Beispiel in Abbildung 4.5 stellt ein Testmodell des Automaten in Abbildung 4.4 dar. Das Modell ist um die ganzzahlige Variable *coverage* erweitert, die der Definition von Testüberdeckungskriterien dient. Der Treiberautomat besitzt einen Zustand *idle* und zwei reflexive Transitionen, die mit einigen Transitionen im Systemautomaten synchronisieren. Der Treiberautomat ist nichtdeterministisch und gibt dem Modell den notwendigen Freiheitsgrad für die Suche im Lösungsraum. Die grau hinterlegten zusätzlichen Codezeilen stellen die Instrumentierung dar und charakterisieren die *Location B* mit *coverage=1* und die Transition von der *Committed Location* zur *Location C* mit *coverage=2*.

Mittels der Anforderung in Uppaal-Logik

$E\Diamond\, coverage = 1$

wird ein Testpfad ermittelt, der *Location B* erreicht.

Mittels der Anforderung in Uppaal-Logik

$E\Diamond\, coverage = 2$

wird ein Testpfad ermittelt, der die Transition von der *Committed Location* zu *Location C* ausführt.

Verschiedene Testüberdeckungskriterien für die Testfallgenerierung mittels Uppaal wurden von Mücke und Huhn in [103] präsentiert. Diese und weitere Kriterien werden basierend auf einer Komplexitätsbetrachtung an das vorliegende Testproblem angepasst und teilweise verbessert.

Kapitel 5

Testverfahren

Die große Anzahl von Testverfahren in Theorie und Praxis macht eine Verfahrensklassifikation notwendig. Eine übliche Klassifikation unterscheidet funktions- und strukturorientierte Testverfahren [92, 6, 10], die im Folgenden kurz erläutert werden soll.
Die funktionsorientierten Verfahren basieren auf der Funktionsbeschreibung der Software im Test, die zur Definition der Testvollständigkeit, zur Herleitung von Testfällen und zur Beurteilung des Verhaltens der Software im Test dient. Die Testfälle dienen der Überprüfung der vollständigen und korrekten Umsetzung aller geforderten Funktionen. Der Programmcode muss zur Testdurchführung nicht instrumentiert werden, daher sind funktionsorientierte Testfälle in der Regel portabel und lassen den Test des realen Laufzeitverhaltens zu. Selbst ein vollständiger funktionsorientierter Test kann nicht die vollständige Abdeckung des Programmcodes gewährleisten. Daher gelten diese Verfahren in isolierter Anwendung als nicht hinreichend.

Die strukturorientierten Verfahren nutzen den inneren Aufbau der Implementation zur Definition von Testvollständigkeitskriterien. Im selteneren Fall dienen strukturorientierte Überdeckungskriterien auch zur Generierung von Testfällen. Die Beurteilung der Testvollständigkeit wird anhand des Abdeckungsgrades der Implementation mit Testfällen durchgeführt. Für die Messung der Strukturüberdeckung mit Testfällen ist eine Instrumentierung der Software mit zusätzlichem Code notwendig, die das Laufzeitverhalten der Software verändert.

Eine ausführliche Beschreibung der funktions- und strukturorientierten Testverfahren kann in [92, 6, 10] gefunden werden.

Im Modultest dient die Modulimplementation und -spezifikation als Basis für den Test. In der Regel können Module zweckmäßig als die kleinsten, isoliert testbaren Einheiten identifiziert werden. Die Modulspezifikation besteht aus allen Dokumenten und Informationen, die für eine Implementation des Moduls notwendig sind. Die Struktur eines Moduls hängt maßgeblich von dem verwendeten Softwareentwicklungsverfahren und von der verwendeten Programmiersprache ab [7]. Ein Modul stellt in *Real-Time Object-Oriented Modeling* ein Aktor dar, der eine eigenständige Entität mit einer definierten Aufgabe ist.

Im modellbasierten Test mit ausführbaren Spezifikationen ist die Unterscheidung zwischen struktur- und funktionsorientierten Verfahren nicht zweckmäßig, da die zum Test dienenden Modelle sowohl Dokumentation als auch Implementation des zu testenden Moduls darstellen. Aus diesem Grund werden Testverfahren zusätzlich nach der zugrunde liegenden Spezifikationsstruktur in graphen- und automatenbasierte Verfahren unterteilt. Da die Bedingungsüberdeckungsverfahren nicht von der Darstellungsart des Kontrollflusses abhängen, werden sie separat aufgeführt.

5.1 Graphenbasierte Testverfahren

Die im Folgenden vorgestellten Verfahren basieren auf imperativem, strukturiertem Programmcode und deren Darstellung in Form von Kontrollflussgraphen. Allen in diesem Abschnitt beschriebenen Verfahren ist die Definition von Überdeckungskriterien auf den Graphen gemein, die zur Definition der Testvollständigkeit oder zur Testfallgenerierung dienen. Die Anwendung dieser Kriterien ist nicht auf Kontrollflussgraphen beschränkt, sondern kann auf beliebige Graphen übertragen werden.

```
boolean ALL_POSITIVE(int[] array,int len) {
  boolean result;
  int i,tmp;
  i=0;
  result=true;
  while (i<len&&result) {
    tmp=array[i];
    if (tmp<=0)
      result=false;
    i++;
  }
  return result;
}
```

Abbildung 5.1: Beispielprogramm

An anderer Stelle dieser Arbeit dienen die graphenbasierten Testverfahren zur Testfallgenerierung auf Basis von Zustandsautomaten.

Das Beispiel in Abbildung 5.1 zeigt eine einfache Operation in Java [60, 100].
Die Funktion ALL_POSITIVE überprüft, ob alle Elemente eines ganzzahligen Feldes positiv sind. Die Operation besitzt das ganzzahlige Feld *array* und die ganzzahlige Variable *len* als formale Parameter. Der Parameter *len* gibt die Anzahl der Elemente des Parameters *array* an. Die Schleife im Rumpf der Operation iteriert aufsteigend die Elemente von *array* und bricht ab, wenn ein Element gefunden wird, das kleiner oder gleich 0 ist.

5.1.1 Kontrollflussgraphen

Zur Darstellung der Programmstruktur [6, 7, 92, 10] während Entwurf und Prüfung von funktionalen Modulen dienen häufig graphische Mittel, beispielsweise Nassy-Shneidermann-Diagramme, Programmablaufpläne und Kontrollflussgraphen. Während der Modulprüfung stellen Kontrollflussgraphen die verbreiteste graphische Darstellungsform von Programmcode dar, auf deren Knoten die Anweisungen abgebildet werden und deren Kanten den Kontrollfluss darstellen. Der Kontrollfluss beginnt im Knoten n_{start} und endet im Knoten n_{final}. Ein Knoten n_i, mit $i \geq 0$, $i \in \mathbb{Z}$, stellt eine beliebige Anzahl Anweisungen mit ausschließlich sequentieller Bearbeitungsfolge dar. Eine Kante zwischen zwei Knoten n_i und n_j, mit $i, j \geq 0$, $i, j \in \mathbb{Z}$, stellt eine Abfolge der Bearbeitung von n_i nach n_j dar.

In Abbildung 5.2 ist für die Funktion ALL_POSITIVE ein beispielhafter Kontrollflussgraph dargestellt. Der Kontrollflussgraph besitzt einen Startknoten n_{start} und einen Endknoten n_{final}. Im Knoten n_4 ist die Schleife definiert, die den Rumpf bis Knoten n_8 wiederholt oder abbricht und nach Knoten n_9 verzweigt. Im Schleifenrumpf ist in Knoten n_6 eine Bedingungsanweisung dargestellt.

5.1.2 Anweisungsüberdeckung

Das einfachste kontrollflussorientierte Testvollständigkeitskriteriums stellt die Anweisungsüberdeckung dar. Es wird gefordert, dass jede Anweisung der Funktion ausgeführt bzw. jeder Knoten im Kontrollflussgraphen durch Testfälle abgedeckt werden soll. Für die Funktion ALL_POSITIVE bzw. deren Kontrollflussgraph in Abbildung 5.2 wird die Überdeckung aller Knoten bzw. die Ausführung aller mit den Knoten korrespondierenden Anweisungen gefordert. Dies wird durch die Ausführung des Pfades *abcdefhik* erreicht. Ein möglicher Testfall zur Ausführung dieses Pfades könnte lauten:

Testfall Anweisungsüberdeckung ALL_POSITIVE

ALL_POSITIVE({-1},1) → false

Im Beispiel in Abbildung 5.2 wird die Kante (n_6, n_8) durch diesen Testfall nicht abgedeckt. Diese Kante entspricht dem *Else*-Zweig der Bedingungsanweisung, der keine Anweisung enthält und

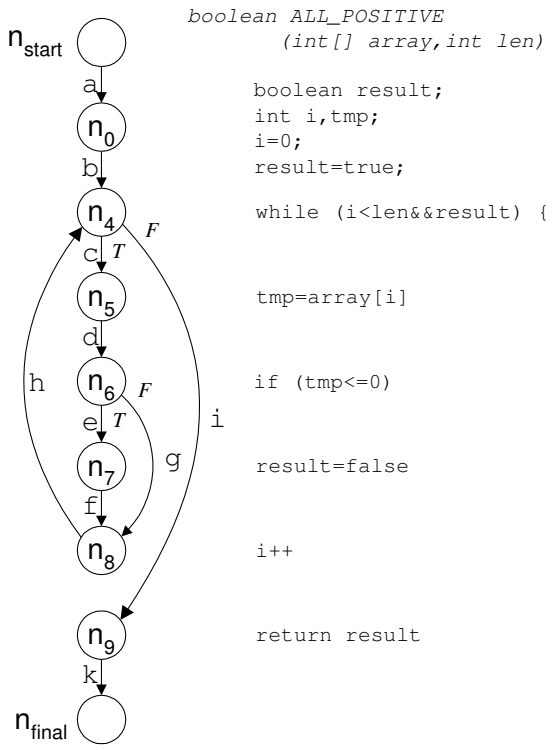


Abbildung 5.2: Kontrollflussgraph

daher nicht von dem Anweisungsüberdeckungskriterium erfasst wird. Aus diesem Grund gilt das Anweisungsüberdeckungskriterium im Allgemeinen als nicht hinreichend für den Test von sicherheitskritischer Software.

5.1.3 Zweigüberdeckung

Das Zweigüberdeckungskriterium fordert die mindestens einmalige Ausführung jedes Zweiges im Kontrollflussgraphen durch Testfälle. Diese Forderung korrespondiert mit der mindestens einmaligen Ausführung jeder Bedingungsanweisung, bzw. Schleifenbedingung, mit *Wahr* und mit *Falsch*. Eine Mehrfach-Bedingungsanweisung wird mit der mindestens einmaligen Ausführung jedes Falles, einschließlich dem Standardfall, ausgeführt. Für die Funktion ALL_POSITIVE bzw. deren Kontrollflussgraph in Abbildung 5.2 wird die Überdeckung aller Zweige gefordert. Dies wird durch die Ausführung des Pfades *abcdefhcdghik* erreicht. Ein möglicher Testfall zur Ausführung dieses Pfades könnte lauten:

Testfall Zweigüberdeckung ALL_POSITIVE

ALL_POSITIVE({1,-1},2) → false

Das Zweigüberdeckungskriterium gilt als allgemeines Minimalkriterium für den strukturorientierten Test von Software.

5.1.4 Strukturierte Pfadüberdeckung und *Boundary-Interior* Test

Die Forderung nach der Überdeckung aller Pfade eines Programms durch Testfälle ist problematisch, wenn die Anzahl der Pfade sehr groß ist oder ihre Ausführung zuviel Zeit beanspruchen würde. Oftmals besitzen Programme theoretisch unendlich viele Pfade, wenn eine Schleifenanweisung mit *a priori* nicht bestimmbarer Anzahl von Durchläufen vorliegt. Aus diesem Grund ist die

Pfadüberdeckung in der Praxis nur von untergeordneter Bedeutung. Um dennoch dieses Verfahren für die Praxis nutzbar zu machen, muss die Anzahl der Schleifendurchläufe zweckmäßig eingegrenzt werden. Die *strukturierte Pfadüberdeckung* reduziert die Anzahl auszuführender Schleifendurchläufe eines Pfadüberdeckungstests auf ein praktikables Maß.

Zur Erfüllung der strukturierten Pfadüberdeckung wird für ein Programm P die mindestens einmalige Ausführung aller Pfade verlangt, die weniger oder gleich k Ausführungen von Schleifenrümpfen, also (0,1,...,k), beinhalten. Der *Boundary Interior* Test ist äquivalent zur strukturierten Pfadüberdeckung mit dem Parameter *k=2*.

Pfadausdruck
für das Beispiel in Abbildung 5.2

$ab\,(cd[e]fh)^{\mathbf{k}}\,ik$

In dem Pfadausdruck für das Beispiel in Abbildung 5.2 stellen hochgestellte Zahlen die Anzahl Wiederholungen dar und eckige Klammern kennzeichnen optionale Ausführungen.

Durch Einsetzen der zulässigen Werte von k in den Pfadausdruck ergeben sich die in Tabelle 5.1 dargestellten Pfade und Testfälle.

Testfall	k	Pfad	Ein-/Ausgabeverhalten
1	0	$abik$	ALL_POSITIVE({},0)→false
2	1	$abcdefhik$	ALL_POSITIVE({-1},1)→false
3	1	$abcdghik$	ALL_POSITIVE({1},1) →true
4	2	$abcdefhcdefhik$	nicht ausführbar
5	2	$abcdghcdghik$	ALL_POSITIVE({1,1},2) →false
6	2	$abcdefhcdghik$	nicht ausführbar
7	2	$abcdghcdefhik$	ALL_POSITIVE({1,-1},2)→false

Tabelle 5.1: Testfälle *Boundary Interior* Kriterium

Zwei Pfade sind programmtechnisch nicht ausführbar, da die Schleife nach Ausführen des *Wahr*-Rumpfes und der inneren Bedingung zwangsläufig keine weitere Wiederholung zulässt. Alle ausführbaren Pfade nach dem *Boundary-Interior* Test bzw. dem strukturierten Pfadtest mit k=2 sind daher die verbliebenen fünf Pfade.

5.1.5 Datenflussorientierte Verfahren

In vielen imperativen Programmiersprachen [1, 98, 150, 100] werden Daten definiert und für Berechnungen oder Entscheidungen während des Programmablaufs benutzt. Die datenflussorientierten Verfahren fordern die Überdeckung von definitionsfreien Pfaden zwischen Definitionen und Benutzungen von Variablen mit Testfällen. Die Überdeckung dieser Datenflüsse dient in der Regel zur Bewertung der Testvollständigkeit.

Datenflussdarstellung von Kontrollflussgraphen Zur Definition von Testvollständigkeitskriterien verwenden die datenflussbasierten Testverfahren einen modifizierten Kontrollflussgraphen in Datenflussdarstellung. Der im Vorangegangenen vorgestellte Kontrollflussgraph wird um Knoten zum Im- und Export globaler Variablen und Attribute zur Darstellung von Datenzugriffen erweitert. Das Beispiel des Kontrollflussgraphen der Operation ALL_POSITIVE ist in Abbildung 5.3 in Datenflussdarstellung gegeben. Die Feldvariable *array* wird in *Java* als Referenz übergeben und gilt damit global. Der Parameter *len* gilt dagegen ausschließlich lokal.

Es wird ein zusätzlicher Knoten n_{in} hinter n_{start} eingefügt, der dem Import von globalen Variablen und Parametern dient. Weiterhin wird ein zusätzlicher Knoten n_{out} vor n_{final} eingefügt, der dem Export von globalen Variablen dient. Jeder Knoten des Kontrollflussgraphen wird für jede Variable x zusätzlich mit Datenflussattributen der Art *def(x)*, *c-use(x)* und *p-use(x)* attributiert. Eine Definition von x im Knoten n wird mit *def(x)* angegeben und an n angetragen. Eine Benutzung von x in einer Berechnung im Knoten n wird mit *c-use(x)* angegeben und an n angetragen. Eine

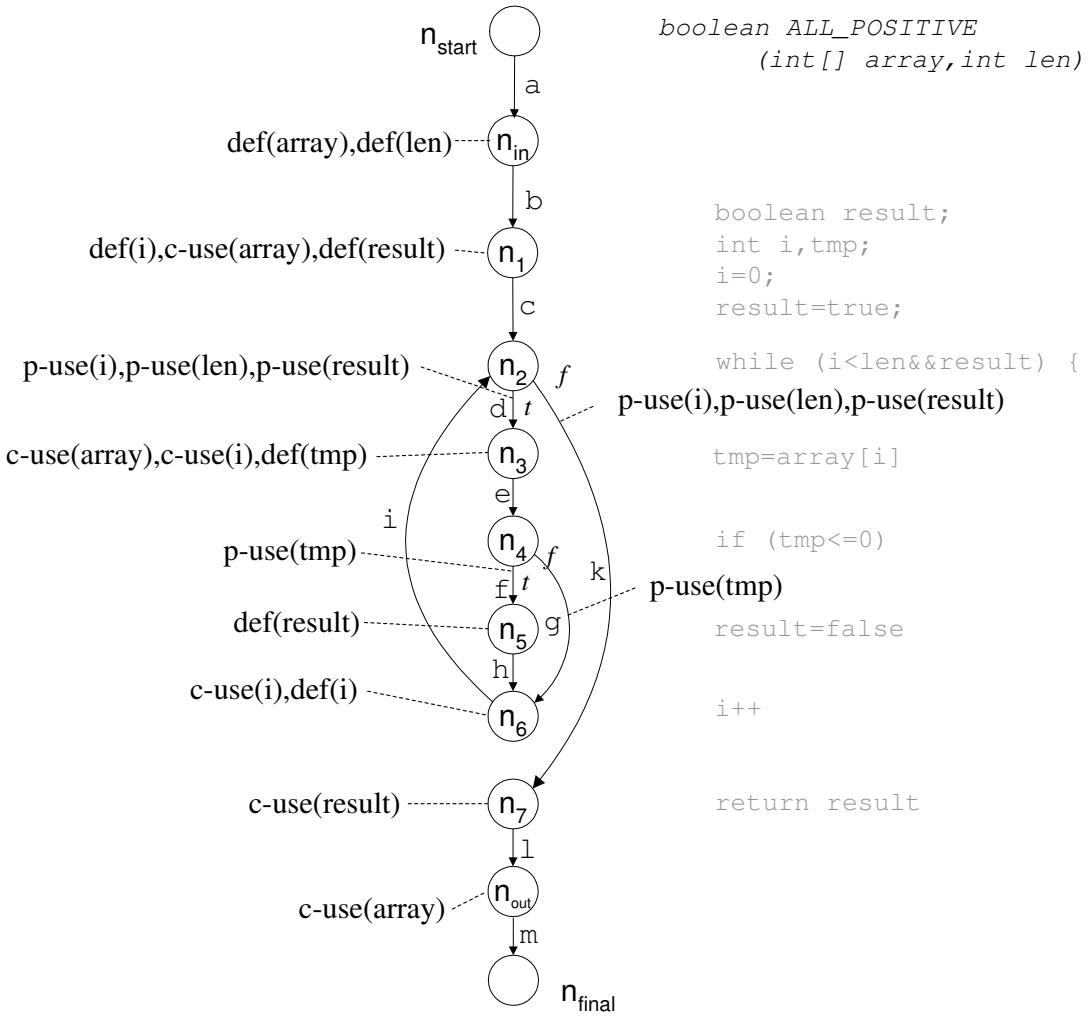


Abbildung 5.3: Kontrollflussgraph in Datenflussdarstellung

Benutzung von x in einer Bedingung am Knoten n wird mit *p-use(x)* angegeben und an jede ausgehende Kante von n angetragen. Die Parameter der Operation werden am Importknoten definiert. Eine globale Variable wird nur dann am Importknoten definiert, wenn sie in der Operation ohne vorherige Definition benutzt wird. Am Exportknoten werden alle globalen Variablen mit einem *c-use(x)* notiert, um stellvertretend alle nachfolgenden Zugriffe auf diese Variablen darzustellen.

definitionsfreie Pfade Ein definitionsfreier Pfad beginnt in einem Knoten mit der Definition einer Variablen und endet in einem Knoten mit einer Benutzung derselben Variablen. Für die Darstellung definitionsfreier Pfade wird in dieser Arbeit folgende Notation verwendet:

(Knoten der Definition, Kantenzug zur Variablenbenutzung, Variablenname)

Zur Angabe eines Kantenzuges werden Kleinbuchstaben benutzt, welche die Kanten im Kontrollflussgraphen identifizieren. Im Falle eines *c-uses* führt der Kantenzug von der Definition zum Knoten mit einem berechnenden Datenzugriff. Im Falle eines *p-uses* entspricht die letzte Kante dem Ort des Datenzugriffs. Die folgenden Definitionen werden im Rahmen dieser Arbeit vorgenommen: Die Menge aller in einem Knoten n definierten Variablen wird mit D_n bezeichnet. Die Menge $C_{x,n}$ enthält für jede Variable $x \in D_n$ alle Knoten, in denen auf x berechnend zugegriffen wird und zu denen ein definitionsfreier Pfad von n bezüglich x existiert. Die Menge $P_{x,n}$ enthält für jede Variable $x \in D_n$ alle Kanten, an denen x infolge einer Bedingung am Ausgangsknoten notiert ist und zu denen ein definitionsfreier Pfad bezüglich x ausgehend von n existiert.

all defs Das *all defs*-Kriterium verlangt für ein Programm S und jede Variable x in S die mindestens einmalige Ausführung jeder Definition von x mit mindestens einem definitionsfreien Pfad zu einer Variablenbenutzung von x. Im Beispiel in Abbildung 5.3 fordert das *all defs*-Kriterium die Ausführung der Pfade, die durch folgende *def-use* Paare spezifiziert werden:

$$
\begin{array}{ll}
array: & (n_{in}, b, array) \\
len: & (n_{in}, bcd, len) \\
tmp: & (n_3, ef, tmp) \\
result: & (n_1, cd, result), (n_5, id, result) \\
i: & (n_1, cd, i), (n_6, id, i)
\end{array}
$$

Für jeden Knoten n und jede Variable $x \in D_n$ muss mindestens ein definitionsfreier Pfad zu einem Element von $C_{x,n}$ oder $P_{x,n}$ in der Menge ausgeführter Programmpfade enthalten sein. Das *all defs*-Kriterium subsumiert nicht das Zweigüberdeckungskriterium und kann als isoliertes Verfahren daher nicht als hinreichend für einen strukturorientierten Modultest gelten.

all c-uses Das *all c-uses*-Kriterium verlangt für ein Programm S und jede Variable x in S die mindestens einmalige Ausführung jedes definitionsfreien Pfades bezüglich x, von jeder Definition von x zu jeder berechnenden Benutzung von x. Im Beispiel fordert das *all c-uses*-Kriterium die Ausführung aller Pfade, die durch die Menge der *def-use*-Paare spezifiziert wird:

$$
\begin{array}{ll}
array: & \{(n_{in}, b, array), (n_{in}, bcd, array), (n_{in}, abcklm, array)\} \\
len: & \{\} \\
tmp: & \{\} \\
result: & \{(n_1, ck, result), (n_5, hik, result)\} \\
i: & \{(n_1, cd, i), (n_1, cdefh, i), (n_6, idefh, i)\}
\end{array}
$$

Für jeden Knoten n und jede Variable $x \in D_n$ muss mindestens ein definitionsfreier Pfad zu jedem Element von $C_{x,n}$ in der Menge ausgeführter Programmpfade enthalten sein. Das *all c-uses*-Kriterium subsumiert nicht das Zweigüberdeckungskriterium und kann als isoliertes Verfahren daher nicht als hinreichend für einen strukturorientierten Modultest gelten.

all p-uses Das *all p-uses*-Kriterium verlangt für ein Programm S und jede Variable x in S die mindestens einmalige Ausführung jedes definitionsfreien Pfades bezüglich x, von jeder Definition von x zu jeder Benutzung von x innerhalb einer Bedingung. Im Beispiel fordert das *all p-uses*-Kriterium die Ausführung aller Pfade, die durch die Menge der *def-use*-Paare spezifiziert wird.

$$
\begin{array}{ll}
array: & \{\} \\
len: & \{(n_{in}, bcd, len), (n_{in}, bck, len)\} \\
tmp: & \{(n_3, ef, tmp), (n_3, eg, tmp)\} \\
result: & \{(n_1, cd, result), (n_1, ck, result), \\
 & (n_5, hid, result), (n_5, hik, result)\} \\
i: & \{(n_1, cd, i), (n_1, ck, i), (n_6, id, i), (n_6, ik, i)\}
\end{array}
$$

Für jeden Knoten n und jede Variable $x \in D_n$ muss mindestens ein definitionsfreier Pfad zu jedem Element von $P_{x,n}$ in der Menge ausgeführter Programmpfade enthalten sein. Da das *all p-uses*-Kriterium das Zweigüberdeckungskriterium subsumiert, kann es als hinreichend bezüglich dem Minimalkriterium für den strukturorientierten Test sicherheitskritischer Software gelten.

all uses Das *all uses*-Kriterium verlangt für ein Programm S und jede Variable x in S die mindestens einmalige Ausführung jedes definitionsfreien Pfades bezüglich x, von jeder Definition von x zu jeder Benutzung von x innerhalb einer Bedingung oder innerhalb einer Berechnung. Im Beispiel fordert das *all uses*-Kriterium die Ausführung aller Pfade, die durch die Menge der *def-use*-Paare spezifiziert werden.

Testfall	(a>=4)	(b<3)	(c==0)	(d<0)
1	F $a = 3$	F $b = 3$	$-$ $c = *$	$-$ $d = *$
2	F $a = 3$	T $b = 2$	F $c = 1$	T $d = 1$
3	F $a = 3$	T $b = 2$	T $c = 0$	$-$ $d = *$
4	T $a = 4$	$-$ $b = *$	F $c = 1$	F $d = 0$

Tabelle 5.2: Testfälle einfache Bedingungsüberdeckung

$$
\begin{array}{ll}
array: & \{(n_{in}, b, array), (n_{in}, bcd, array), (n_{in}, abcklm, array)\} \\
len: & \{(n_{in}, bcd, len), (n_{in}, bck, len)\} \\
tmp: & \{(n_3, ef, tmp), (n_3, eg, tmp)\} \\
result: & \{(n_1, cd, result), (n_1, ck, result), (n_5, hid, result), \\
 & (n_5, hik, result), (n_1, ck, result), (n_5, hik, result)\} \\
i: & \{(n_1, cd, i), (n_1, ck, i), (n_6, id, i), (n_6, ik, i) \\
 & (n_1, cd, i), (n_1, cdefh, i), (n_6, idefh, i)\}
\end{array}
$$

Für jeden Knoten n und jede Variable $x \in D_n$ muss mindestens ein definitionsfreier Pfad zu jedem Element von $P_{x,n}$ in der Menge ausgeführter Programmpfade enthalten sein. Da das *all uses*-Kriterium das Zweigüberdeckungskriterium subsumiert, kann es als hinreichendes Minimalkriterium des datenflussbasierten Modultests gelten.

5.2 Bedingungsüberdeckungsverfahren

Der Kontrollfluss eines Moduls wird von den Prädikaten in Bedingungsanweisungen und Schleifenbedingungen maßgeblich bestimmt. Diese Bedingungen können von hoher Komplexität sein, daher sollte ihr innerer Aufbau bei der Bildung von Testfällen berücksichtigt werden. Bereits der Zweigüberdeckungstest verlangt die Ausführung jeder Bedingungsanweisung mit Testfällen, welche diese mindestens einmal *Wahr* und einmal *Falsch* werden lassen. Für die aus mehreren Teilbedingungen zusammengesetzten Entscheidungen gilt dieses Verfahren als nicht hinreichend. Eine Bedingung, die nicht in Teilbedingungen zerlegt werden kann, wird *atomar* genannt. Besitzt eine Bedingungsanweisung eine zusammengesetzte Bedingung, so hängt es von der Auswertungsmethode des *Compilers* ab, ob jede atomare Bedingung ausgewertet wird. Bei einer vollständigen Evaluation wird jede atomare Bedingung vom Übersetzungsprogramm ausgewertet. Aus Effizienzgründen wird aber üblicherweise bereits mit der Auswertung abgebrochen, wenn die Gesamtentscheidung von den verbleibenden Bedingungen nicht mehr verändert werden kann. Die folgenden Beispiele basieren auf der Annahme unvollständiger Auswertung von links nach rechts. Eine Bedingung, die nicht vom *Compiler* evaluiert wird, ist in den Beispielen durch einen Spiegelstrich gekennzeichnet. Der Wert der Entscheidungsvariablen kann in diesem Fall beliebig gewählt werden und ist durch * dargestellt.

```
wenn ((a>=4)||b<3)&&((c==0)||d<0)) dann ...
```

Abbildung 5.4: Bedingungsanweisung in Pseudocode

Eine ausführliche Beschreibung dieser Bedingungsüberdeckungstestverfahren kann in [92] gefunden werden. Im Folgenden werden die einfache Bedingungsüberdeckung und die modifizierte Bedingungs-/Entscheidungsüberdeckung erläutert.

Einfache Bedingungsüberdeckung Die einfache Bedingungsüberdeckung fordert die Ausführung jeder Bedingungsanweisung, so dass für jede der atomaren Teilbedingungen mindestens einmal *Wahr* und einmal *Falsch* gilt.

In Tabelle 5.2 sind beispielhafte Testfälle für das Beispiel in Abbildung 5.4 dargestellt. Die Testfälle lassen jede atomare Teilentscheidung mindestens einmal *Wahr* und mindestens einmal *Falsch* werden. Da die einfache Bedingungsüberdeckung nicht die Gesamtentscheidung berücksichtigt, subsumiert sie im allgemeinen Fall nicht den Zweigüberdeckungstest. Durch die Verwendung eines *Compilers* mit unvollständiger Evaluation der Teilbedingungen von links nach rechts ist die Subsumption der Zweigüberdeckung allerdings sichergestellt.

Modifizierte Bedingungs-/Entscheidungsüberdeckung

Die modifizierte Bedingungs-/Entscheidungsüberdeckung fordert die Ausführung jeder Bedingungsentscheidung, zur Demonstration, dass jede Teilentscheidung separat die Gesamtentscheidung beeinflusst. Zu diesem Zweck wird jede Teilentscheidung jeweils mit den Werten *Wahr* und *Falsch* ausgeführt, während alle sonstigen Teilentscheidungen unverändert bleiben und die Gesamtentscheidung ihren Wert wechselt.
Die Art der Auswertung der Teilbedingungen durch den *Compiler* hat auf die Durchführung dieses Verfahrens keinen Einfluss, da nicht evaluierte Teilentscheidungen die Gesamtentscheidung offensichtlich nicht beeinflussen können.

Testfall	Paar	A $(a \geq 4)$	B $(b < 3)$	C $(c = 0)$	D $(d < 0)$	*gesamt*
1	D	F $a = 3$	F $b = 2$	$-$ $c = 1$	$-$ $d = 0$	F
2	A,B,C,D	F $a = 3$	T $b = 2$	F $c = 1$	T $d = -1$	W
3	B	F $a = 3$	T $b = 3$	T $c = 1$	$-$ $d = -1$	F
4	A	T $a = 4$	$-$ $b = *$	F $c = 1$	F $d = -1$	F
5	C	T $a = 3$	T $a = 2$	T $a = 0$	T $a = *$	F

Tabelle 5.3: Testfälle modifizierte Bedingungs-/Entscheidungsüberdeckung

In Tabelle 5.3 ist beispielhaft eine Anzahl von Testfällen für die Erfüllung des modifizierten Bedingungs-/Entscheidungsüberdeckungstests dargestellt. In der zweiten Spalte sind die atomaren Bedingungen eingetragen, welche durch die jeweiligen Testfälle geprüft werden.

5.3 Automatenbasierte Verfahren

Die Entwicklung von automatenbasierten Testmethoden wurde ursprünglich durch Probleme beim funktionsorientierten Test von Steuerkreisen inspiriert. Diese Methoden zur Ableitung vollständiger Testsuiten aus Zustandsautomaten basieren alle auf den Ansätzen des *Transition Checking* von Hennie [66] und den *Distinguishing Sequences* von Moore [102] und später von Gönenc [59]. Basierend auf diesen Ansätzen wurden diverse Anwendungen und Entwicklungen beschrieben, die teilweise in verschiedenen Veröffentlichungen zusammenfassend dargestellt werden [18, 146, 155]. Jede dieser Methoden benötigt ein besonderes *Characterization Set* [159], das zur Zustandsidentifikation benutzt wird. Ein *Characterization Set* ist eine Testeingabe, die es erlaubt, jeden Zustand eines minimalen Automaten anhand seines charakteristischen Ein-/Ausgabeverhaltens zu identifizieren. Eine Menge von Testsequenzen für vollständige Transitionsüberdeckung in Verbindung mit einem *Characterization Set* führt zu einer Eingabesequenz, die alle Transitionen am Ein-/Ausgabeverhalten identifiziert und als *Checking Sequence* bezeichnet wird. Die produzierten Testfälle sind äußerst portabel bei verhältnismäßig hohem Aufwand.

In dieser Arbeit werden *Checking Sequences* aus den in *Real-Time Object-Oriented Modeling* zur Verhaltensbeschreibung verwendeten ROOMcharts mittels dem *Model Checker Uppaal* generiert. In Abbildung 5.5 ist dieser Prozess schematisch dargestellt.

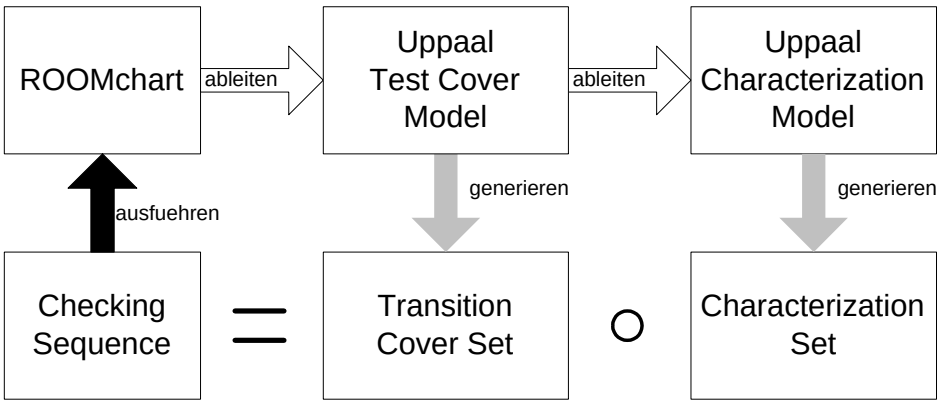


Abbildung 5.5: Generierung und Ausführung von *Checking Sequences*

Weiterführende Ansätze des *Transition Checking* betreffen die Verbesserung von deren Leistungsfähigkeit [2, 72] und Anwendbarkeit [46, 94, 161, 73]. Die Anwendbarkeit dieser Methoden für den Test von objektorientierter Software wurde u.a. in [18, 154, 85, 84] gezeigt. In [84] demonstrieren Kung et al. wie zustandsbasiertes Testen und Fehlerbaumanalyse kombiniert auf objektorientierte Software angewendet werden können. Die Anwendung der zustandsautomatenbasierten Verfahren auf zeitkritische Systeme wurde beispielsweise in [46, 32, 157] dargestellt.

Mit [18] lieferten v. Bochmann und Petrenko eine detaillierte Zusammenfassung der grundlegenden automatenbasierten Testmethoden mit vollständiger Fehleraufdeckung. Andere Überblicksarbeiten sind [146, 155], in denen auch einige informale Verfahren erläutert werden. Eine umfassende, annotierte Literatursammlung hat ebenfalls Petrenko vorgestellt [121].

Ein allgemeiner Ansatz und eine Methode für den Einsatz dieser Verfahren in der Softwareentwicklung wurde von Chow vorgestellt und als W-Methode bekannt [37]. Die Methoden nach Chow und nach Yevtushenko et al. [161] bieten den theoretischen Vorteil, auch fehlerhaft implementierte, zusätzliche Zustände identifizieren zu können. Die Komplexität dieser Methoden ist exponentiell über die Anzahl zusätzlicher Zustände, die zudem *a priori* bekannt sein muss. Da dies in der Praxis selten der Fall ist, muss eine Schätzung vorgenommen werden, was wiederum einen Nachweis der Abwesenheit zusätzlicher Zustände unmöglich macht. Diese Vorgehensweise ist kaum praktikabel und nur von theoretischer Bedeutung. Daher wird im Rahmen dieser Arbeit vorgeschlagen, die schwache Konformität beider Automaten nachzuweisen und die Prüfung zusätzlicher Zustände nach wirtschaftlichen Kriterien durchzuführen. Die *partial W* oder *Wp*-Methode wurde in [51] vorgestellt und bietet keine Identifikation zusätzlicher Zustände, produziert aber kürzere Testsequenzen.

Die in [160, 33] vorgestellte *UIOv* Methode basiert auf *Unique Input Output* Sequenzen (UIO) [137, 138] und produziert vergleichsweise kurze Testfälle. Ebenso wie für *Distinguishing Sequences* gilt auch für die Konstruktion von *UIOs*, dass nicht für jeden minimalen Automaten eine Lösung gefunden werden kann. Jede automatenbasierte Testmethode mit voller Fehlerfindungsrate (engl. *full fault coverage* [18, 37, 145]) verlangt die Existenz einer Spezifikation in Form eines minimalen Zustandsautomaten.

Eine allgemeine Methode für die Generierung effizienter *Checking Sequences* haben Hierons und Ural et al. [72, 156] vorgestellt. Es wurden verschiedene Ansätze präsentiert, diese Methoden auf erweiterte Zustandsautomaten anzuwenden, beispielsweise [77, 20, 110, 111, 19, 22, 148, 30].
Die Arbeit von Bogdanov befasst sich mit dem automatenbasierten Test von *Statecharts* [20]. Diese und die vorliegende Arbeit weisen zwangsläufig in einigen Punkten Parallelen auf, allerdings ist der Ansatz von Bogdanov nicht vollständig auf *Real-Time Object-Oriented Modeling* anwendbar. Weiterhin ist keiner der aufgeführten Ansätze in ähnlicher Form an verschiedenste Testprobleme anpassbar, was erst infolge der durchgängigen Nutzung eines *Model Checkers* zur Generierung von *Characterization Sets* ermöglicht wird.

In [151] präsentieren Sun et al. eine effiziente Methode, um *Unique Input Output Sequences* zu generieren, die jedoch keine optimalen Ergebnisse liefert. In [89] untersuchen Lee und Yannakakis

die Komplexität der Konstruktion von *Distinguishing Sequences* [102] und *Unique Input Output Sequences* [137, 138, 33] mit dem negativen Ergebnis, dass diese PSPACE-vollständig ist. Der von Lee und Yannakakis vorgestellte Ansatz zur Konstruktion adaptiver *Distinguishing Sequences* ist von polynomialer Komplexität bei voller Fehlerfindungsrate. Die Umsetzung dieses Ansatzes ist erst nach erheblichen Modifikationen des *Uppaal Model Checkers* möglich. Für nachfolgende Forschungsarbeiten wird eine Umsetzung dieses Ansatzes mittels einem modifizierten *Uppaal* oder einem anderen *Model Checker* vorgeschlagen. Weitere Untersuchungen zur Komplexitätsreduktion der Generierung von *Characterization Sets* können in [35, 36, 34] gefunden werden.

Im Rahmen dieser Arbeit wurde erstmalig eine Methode für die Generierung optimaler *Distinguishing Sequences* mittels *Model Checking* entwickelt [135]. Diese Ergebnisse sind insbesondere von Bedeutung, da bisher nur wenige Algorithmen für die Generierung von *Characterization Sets* präsentiert wurden. Die Komplexität der Generierung von *Distinguishing Sequences* entspricht annähernd der Komplexität der Testfallgenerierung auf Basis von Überdeckungskriterien, vergleichend können die Ergebnisse in [135, 103] herangezogen werden, daher ist die präsentierte Methode optimal in jeden Ansatz der Testfallgenerierung mittels *Model Checking* integrierbar. Die Generierung von *Distinguishing Sequences* und UIOs mittels *Model Checking* weist prinzipiell die von Lee und Yannakakis dargestellten Komplexitätsprobleme auf. Generell kann allerdings für die Generierung von *Characterization Sets* mittels *Model Checking* erwartet werden, dass die Komplexität im Vergleich zu manuell implementierten Lösungen nicht zwangsläufig höher ist. Vielmehr hängt diese Komplexität weitgehend von den verwendeten Methoden und Algorithmen des *Model Checkers* ab, die in vielen Fällen auf Standardverfahren wie beispielsweise Breiten- und Tiefensuche basieren. Durch die Verwendung verbesserter *Model Checker* kann daher auch die Komplexität der Generierung von *Characterization Sets* reduziert werden.

Einen weiteren wichtigen Vorteil stellt die hohe Anpassungsfähigkeit der präsentierten Methode an neue Testprobleme dar. Durch die Verwendung des *Model Checkers* ist die Testfallgenerierung auf die Formulierung des Testproblems vereinfacht. Eine fehlerintensive und aufwendige Implementierung des Generierungsalgorithmus entfällt und der präsentierte Ansatz wird automatisch verbessert, wenn der *Model Checking* Algorithmus verbessert wird.

Obwohl der Ansatz der Generierung von *Characterization Sets* mittels *Model Checking* ausschließlich anhand von *Distinguishing Sequences* demonstriert wird, ist er auch für die Generierung von *W-Sets* und *UIOs* geeignet. Geeignete Regeln für die Konstruktion von *W-Sets* und *UIOs* werden in dieser Arbeit vorgestellt.

Grundsätzlich werden in automatenbasierten Testverfahren zwei Zustandsautomaten gegeneinander auf Konformität geprüft. Einer dieser Automaten liegt vor, während der andere Automat sich im Verhalten der Software manifestiert und ansonsten unbekannt ist. Die in dieser Arbeit behandelten ROOMcharts bieten eine geeignete Grundlage für die Generierung von *Checking Sequences*. Die genaue Vorgehensweise ist in anderen Kapiteln dieser Arbeit ausführlich beschrieben worden und soll daher nicht weiter erläutert werden. Die ermittelten Testfälle werden auf dem generierten und kompilierten Code ausgeführt und gegen eine abstrakte, nicht notwendigerweise formale Spezifikation evaluiert.

Im Folgenden werden die Verwendung von *W-Sets* (W, Wp) und *Distinguishing Sequences* (DS) zur Konstruktion von *Checking Sequences* anhand einfacher Beispiele demonstriert.

W-Methode

Im Folgenden wird eine vereinfachte Form der W-Methode ohne Identifikation zusätzlicher Zustände vorgestellt. Die Äquivalenz zweier Automaten wird unter der Annahme der gleichen Anzahl Zustände in Spezifikation und Implementation durchgeführt. Unter dieser Annahme kann die W-Methode von Chow vereinfacht als ein Verfahren zur Transitionsüberdeckung mit zusätzlicher Zustandsidentifikation beschrieben werden. Die Transitionsüberdeckung soll die Konformität von Spezifikation und Implementation prüfen. Die Zustandsidentifikation besteht aus Ein- und Ausgaben, die jeden Zustand eindeutig identifizieren. Die vereinfachte W-Methode besteht aus zwei Schritten:

- Generierung der Testsequenzen basierend auf dem vorliegenden Zustandsautomaten
- Evaluierung der Ausgaben der ermittelten Eingabesequenzen

Die Generierung der Testsequenzen beginnt mit der Konstruktion einer Ein-/Ausgabenmenge P, die alle Transitionen überdeckt. Zu diesem Zweck wird ein Baum gebildet, dessen partielle Teilpfade die transitionsüberdeckende Testpfadmenge bildet. Die korrespondierenden Ein-/Ausgaben dieser Testpfadmenge bilden P.

Konstruktion der Transitionsüberdeckung nach Chow [37]

1. Zur Überdeckung aller Transitionen eines endlichen Zustandsautomaten wird ein Wurzelbaum gebildet.
2. Die Wurzel des Baums entspricht dem Initialzustand des Automaten. Jeder Knoten im Baum entspricht einem Zustand des Automaten. Jede Kante im Baum entspricht einer Transition im Automaten und ist mit deren Eingabe beschriftet. Jeder Knoten im Baum besitzt ein Attribut seiner Entfernung vom Wurzelknoten, die als Ebene des Knoten bezeichnet wird.
3. Jede ausgehende Transition aus einem Zustand des Automaten bildet einen ausgehenden Ast im Baum.
4. Ein Knoten ist terminierend, wenn der vom Knoten A_i der Ebene i abgebildete Zustand schon durch einen Vorgängerknoten A_h im Baum abgebildet wird und $h \leq i$ gilt. Der Knoten A_i bildet ein Blatt des Baums.

Ein Pfad im Baum beginnt im Wurzelknoten und endet in einem beliebigen nicht terminalen oder terminalen Knoten. Die Pfadmenge P zur Konstruktion einer transitionüberdeckenden Eingabemenge des Automaten besteht aus allen partiellen Pfaden des Baums. Jeder Pfad im Baum, und damit jedes Element von P, bildet eine Eingabesequenz auf dem Alphabet des Automaten. Für jeden Zustand s_i besitzt P eine Eingabesequenz, die im Initialzustand beginnt und in s_i endet.

Im nächsten Schritt der Generierung der Testsequenzen wird die Eingabemenge W ermittelt, die zur Zustandsidentifikation dient. Die Menge W ist eine Teilmenge des Eingabealphabets X und führt für jedes Zustandstupel (s_i,s_j), mit $i \neq j$, eine unterscheidbare Ausgabe herbei. Die Testsequenzen für den vorliegenden Automaten ergeben sich durch die Kettung $P \circ W$. Für jeden Zustand s_i enthält $P \circ W$ jene Eingabesequenzen, die zu s_i führen und eine von allen anderen Zustände unterscheidbare Aktion herbeiführen. Mit der Menge $P \circ W$ kann jeder fehlerhafte Folgezustand und jede fehlerhafte Aktion eines endlichen Zustandsautomaten mit bekannter Anzahl Zustände erkannt werden.

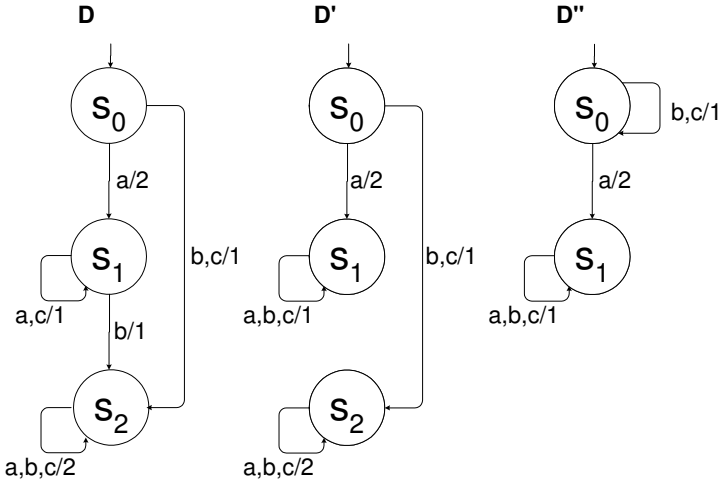


Abbildung 5.6: Zustandsautomaten D, D' und D"

Am Beispiel der drei Zustandsautomaten *D*, *D'* und *D"* werden im Folgenden verschiedene automatenbasierte Testverfahren demonstriert. Die grafische Repräsentation der Automaten in

Abbildung 5.6 ist dahingehend vereinfacht, dass Transitionen mit gleichen Anfangs- und Endzuständen, aber unterschiedlichen Eingaben auf einer Kante, zusammengefasst dargestellt werden. Der Automat D stellt die Spezifikation dar, gegen die D' und D'' geprüft werden. Der Automat D' stellt einen zu D nicht äquivalenten Automaten mit gleicher Anzahl Zustände dar. Der Automat D'' ist nicht äquivalent zu D und besitzt einen Zustand weniger als D. Die Automaten D' und D'' besitzen bezüglich D ausschließlich fehlerhafte Folgezustände und keine fehlerhaften Ausgaben. Es ist leicht, für D' und D'' Transitionsüberdeckung, beispielsweise *{aacb,babc,cabc}* zu erstellen, ohne die beiden Automaten von D anhand ihres Schnittstellenverhaltens unterscheiden zu können.

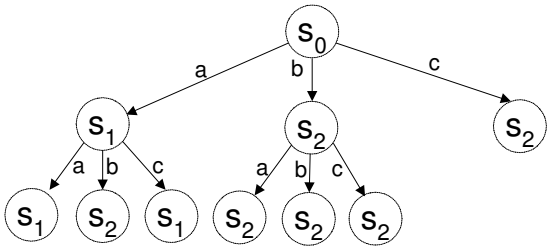


Abbildung 5.7: Pfadbaum D

Konstruktion P

P ist die Menge aller partiellen Pfade des Pfadbaums D.

$P = \{a, b, c, aa, ab, ac, ba, bb, bc\}$

Konstruktion von W

W ist die Menge der charakteristischen Eingaben von D.

$W = \{a, b\}$

	a	b	c
s_0	2	1	1
s_1	1	1	1
s_2	2	2	2

Die obige Tabelle enthält in der ersten Spalte alle Zustände und in der ersten Zeile beliebige Eingaben. Die Ausgaben auf die Eingaben in den jeweiligen Zustände sind Elemente der Tabelle. Im Beispiel kann mit der Eingabemenge *{a,b}* jeder Zustand im Automaten D identifiziert werden, da in jedem Zustand eine andere Ausgabe erzeugt wird.

Konstruktion des Testfallmenge

Die Testfallmenge ergibt sich aus der Kettung $P \circ W$.

$$P \circ W = \{aa, ab, ba, bb, ca, cb, aaa, aab, aba, abb, aca, acb, \\ , baa, bab, bba, bbb, bca, bcb\}$$

Die Testmenge $P \circ W$ bietet Unterscheidbarkeit zwischen D und D' sowie D und D''. Dies wird durch die Betrachtung der Ausgaben zu den jeweiligen Testfällen deutlich. Um Unterscheidbarkeit zu zeigen reicht es, für beide Fälle jeweils eine Testsequenz in der Testfallmenge zu zeigen, die bei D' bzw. D'' eine unterschiedliche Ausgabe bewirkt. Im folgenden Beispiel wird eine Eingabe x in einen Automaten M im Initialzustand mit einer Ausgabe y durch $M(x) \rightarrow y$ beschrieben.

Unterscheidbarkeit von D, D' und D'' durch die Testfallmenge

$D(abb) \rightarrow 212$

$D'(abb) \rightarrow 211$

$D''(abb) \rightarrow 211$

Die Eingabesequenz *abb* zeigt, dass die Automaten *D'* und *D"* nicht $P \circ W$-äquivalent zu D sind. Die Testsequenz T_w wird durch die Kettung aller Elemente von $P \circ W$ mit der Eingabe für die *Reset*-Funktion r konstruiert.

Testsequenz W-Methode

$$T_w = \langle aarabrbarbbrcarcbraaaraabrabarabbracaracb \; rbaarbabrbbarbbbrbcarbcb\rangle$$

Partielle W-Methode

Die partielle W-Methode, oder W_p-Methode, von Fujiwara et al. [51] reduziert die Länge des Tests gegenüber der W-Methode. Die W_p-Methode bestimmt für jeden Zustand eine partielle, charakteristische Eingabemenge. Dadurch ergeben sich in der Kettung der Mengen P und W bzw. W_p kürzere Testsequenzen. An dem vorangegangenen Beispiel soll das Verfahren demonstriert werden. Die Bestimmung der Menge P erfolgt wie gehabt. Die Konstruktion von W erfolgt zunächst analog zur W-Methode und ebenfalls unter der Annahme der gleichen Anzahl von Zuständen.

Das W_p-Verfahren erfolgt in zwei Phasen. Zunächst werden alle Knoten A_i mit W identifiziert. Dann werden alle verbliebenen Sequenzen in P mit den partiellen W gekettet. Die Kettung der partiellen W erfolgt, indem entsprechend dem Endzustand s_i der Eingabesequenz p das entsprechende W_i zu $p \circ W_i$ gekettet wird.

Ein-/Ausgabentabelle zur Konstruktion von W_p

W_p	a	b	c
W_0	2	1	-
W_1	1	-	-
W_2	2	2	-

Die Konstruktion von *Wp* erfolgt ähnlich der Konstruktion von *W*, nur dass die Eingaben für jeden Zustand unterschiedlich sein können.

Eingabealphabet X

$X = \{a, b, c\}$

Charakteristische Eingabemenge W

$W = \{a, b\}$

Partielle charakteristische Eingabemenge W_p

$W_p = \{\{a, b\}_0, \{a\}_1, \{a, b\}_2\}$

Transitionsüberdeckende Pfadmenge P

$P = \{a, b, c, aa, ab, ac, ba, bb, bc\}$

Zustandsüberdeckende Pfadmenge Q

$Q = \{a, b\}$

Phase 1: Zustandsidentifikation

$Q \circ W = \{aa, ab, ba, bb\}$

Phase 2: Transitionsidentifikation

$R = P - Q = \{c, aa, ab, ac, ba, bb, bc, ca, cb, cc\}$

$$R \otimes W_p = \{ca, cb, aaa, aba, abb, aca, baa, bab, bba, \; bbb, bca, bcb, caa, cab, cba, cbc, cca, ccb\}$$

Konstruktion des Testfallmenge nach W_p-Methode

Die Menge der Testeingaben ergibt sich aus der Vereinigung von $Q \circ W$ und $R \otimes W_p$

$$Q \circ W \cup R \otimes W_p = \{aa, ab, ba, bb, ca, cb, aaa, aba, abb, aca, baa, bab, \; bba, bbb, bca, bcb, caa, cab, cba, cbc, cca, ccb\}$$

$$T_{wp} = \{raarabrbarbbrcarcbraaarabarabbracarbaarbab \; rbbarbbbrbcarbcbrcaarcabrcbarcbcrccarccb\}$$

DS-Methode

Die DS-Methode unterscheidet sich im Wesentlichen nur durch die Art der Zustandscharakterisierung von der W-Methode. Eine *Distinguishing Sequence* stellt den speziellen Fall dar, dass ein Element in W ausreicht, um alle Zustände zu identifizieren.

Konstruktion DS

Distinguishing Sequence={aa->21,aa->11,aa->22}

	aa
s_0	21
s_1	11
s_2	22

Transitionsüberdeckende Pfadmenge P

$P = \{a, b, c, aa, ab, ac, ba, bb, bc\}$

Konstruktion der Testfallmenge

$P \circ DS = \{aaa, baa, caa, aaaa, abaa, acaa, baaa, bbaa, bcaa\}$

Konstruktion der Testsequenz

$T_{wp} = \{aaarbaarcaaraaaarabaaracaarbaaarbbaarbcaa\}$

5.4 Fehlersensitivität

Die in diesem Kapitel vorgestellten Testverfahren können unter Beachtung von Testbarkeitsanforderungen und möglicherweise notwendigen Modifikationen, für einen Modultest basierend auf ROOMcharts, verwendet werden. Für die Untersuchung der Effektivität der vorgestellten Verfahrensklassen ist die Definition eines Fehlermodells notwendig. Ein Fehlermodell für den Funktionsteil von ROOMcharts unterscheidet sich nicht von bekannten Fehlermodellen für imperativen Programmcode [81, 10, 14, 92]. Daher wird diese Kategorie von Fehlern an dieser Stelle nicht näher betrachtet. Als Nebeneffekt der in dieser Arbeit empfohlenen einfachen Struktur des Funktionsteils bietet sich für sicherheitskritische Software zusätzlich zu Testverfahren der separate formale Nachweis für jede Transition an. Für eine Vielzahl von Anwendungen wird die durch den Test des Kontrollteils implizierte Überdeckung ausreichend sein. Bei ROOMcharts mit komplexerem Aktionscode sollte eine separate Prüfung jeder Transition durchgeführt werden.

Der Kontrollteil eines minimalen ROOMcharts wird mit der in dieser Arbeit vorgestellten Methode in einen flachen, endlichen Minimalautomaten mit Bedingungen transformiert. Für Spezifikationen in Form endlicher Zustandsautomaten wurden bereits verschiedene Fehlermodelle vorgestellt. Für automatenbasierte Verfahren wurde der Nachweis für bestimmte Fehlerklassen geführt [37, 159, 66, 55, 51, 160]. Diese Verfahren basieren auf dem einfachen Fehlermodell, dass jeder nicht äquivalente Automat fehlerhaft ist.

Ein Beispiel für ein nicht-formales Fehlermodell liefert Binder in [14]. Die Fehlerklassen nach Binder sind stark durch praktische Belange geprägt und infolgedessen nicht disjunkt. In [123, 124] haben Petrenko und Bochmann auf Basis eines Fehlermodells die Ableitung von unvollständigen Testsuiten demonstriert. Das im Folgenden vorgestellte Fehlermodell ist mit jenem von Petrenko und Bochmann kompatibel. Die hier demonstrierten automatenbasierten Verfahren besitzen dagegen vollständige Fehlerfindungsraten für Minimalautomaten und liefern dementsprechend vollständige Testsuiten.

In Tabelle 5.4 sind die Testmodelle von Chow [37] und Binder [14] zusammen mit einem Fehlermodell für die in dieser Arbeit vorgestellten Verfahren (*ROOMtest*) vergleichend dargestellt.

An dieser Stelle wird das disjunkte Fehlermodell in Tabelle 5.4 vorgestellt. Dieses Fehlermodell enthält keine fehlerhaften Ausgaben als separate Fehlerklasse, da diese als eine fehlende und eine zusätzliche Transition interpretiert werden können. Ein Schnittstellenfehler, oder nach Binder *Trap Door* [14], stellt eine Erweiterung des Eingabealphabets dar. Dies kann nur über die Erweiterung der Schnittstelle erfolgen. Die in dieser Arbeit dargestellten automatenbasierten Verfahren finden sicher auf Basis des Ein- und Ausgabeverhaltens die Fehlerklassen II bis V. Das Verfahren nach Chow [37] findet, zumindest theoretisch, auch die Fehlerklasse I sicher. Bei hinreichender

ROOMtest	Binder[14]	Chow [37]
I zusätzlicher Zustand	*Extra, missing* oder *corrupt state*	*Extra State* *Missing State*
II fehlender Zustand		
III zusätzliche Transition	*Missing* oder *Incorrect Transition*, *Missing* oder *Incorrect Event*, *Sneak Path*, *Illegal Message Failure*, *Missing* oder *Incorrect Action*	*Transfer Error*
IV fehlende Transition		
V Schnittstellenfehler	*Trap Door*	–

Tabelle 5.4: Vergleich Fehlermodelle

Instrumentierung können auch mit den graphenbasierten Verfahren diese Fehlerklassen gefunden werden. Bei fehlender Beobachtbarkeit der Modulinterna gibt ein Testfall nur sehr eingeschränkt Aufschluss über die Fehlerklasse, wenn ein Fehlverhalten aufgedeckt wurde. Da Verhalten in einem Testmodell ausschließlich in Transitionen definiert wird, kann die Ursache auf die Fehlerklassen III und IV eingegrenzt werden. Eine zusätzliche Transition kann allerdings eine falsche Ausgabe produzieren oder in einem fehlerhaften, möglicherweise zusätzlichen Zustand enden. Daher können implizit auch die Fehlerklassen I und II in die Ursache eines Fehlverhaltens involviert sein. Während sich eine fehlerhafte Ausgabe unmittelbar in einem Fehlverhalten manifestiert, kann ein fehlerhafter Endzustand erst nach weiteren Eingaben aufgedeckt werden. Keines der präsentierten Testverfahren adressiert die Fehlerklasse V, da diese in der Regel nur durch zusätzliche statische Analysen gefunden werden. Diese Zusammenhänge sind in Tabelle 5.5 dargestellt.

In [110] wurden verschiedene Überdeckungskriterien für *Statecharts* untersucht und empirisch validiert. Die hier demonstrierte Anwendung dieser Verfahren geht insbesondere in Komplexitäts- und Anwendungsproblemen über die in [110] präsentierten Ergebnisse hinaus. In Tabelle 5.6 sind die Klassen der graphenbasierten, bedingungsorientierten und automatenbasierten Testverfahren vergleichend dargestellt. Die graphenbasierten und bedingungsorientierten Verfahren können zur Definition der Testvollständigkeit oder zur Testfallgenerierung verwendet werden. Die automatenbasierten Testverfahren dienen ausschließlich der Testfallgenerierung.

Ursache	Wirkung	Maßnahme	Verfahren
I	nachfolgend unerwartete Ausgabe	Ausführen aller charakteristischen Eingaben	automatenbasiert
II	nachfolgend fehlende Ausgabe	Ausführen aller charakteristischen Eingaben	automatenbasiert
III	unerwartete Ausgabe oder fehlerhafter Folgezustand	Ausführen der Transition	automatenbasiert
IV	fehlende Ausgabe	Ausführen aller Transitionen	automatenbasiert oder Transitionsüberdeckung

Tabelle 5.5: Fehlersensitivität

	Anwendbarkeit	Portabilität	Effektivität	Effizienz
Graphen-basierte und bedingungs-orientierte Testüber-deckung	gut, keine Test-barkeitsan-forder-ungen	schlecht, In-strument-ierung notwendig	mittel, von mensch-licher Expertise abhängig, Ins-trument-ierung notwendig	von mensch-licher Expertise abhängig
Graphen-basierte und bedingungs-orientierte Testfall-generierung	mittel, Pfad-beding-ungen sind problema-tisch	schlecht, Instrument-ierung notwendig	mittel, von Generier-ungs-algorith-mus abhän-gig, Instrument-ierung notwendig	von Generier-ungsalgo-rithmus abhängig
Automaten-basierte Testfall-generierung	schlecht, minimaler Automat, Pfad-beding-ungen sind pro-blematisch	gut, keine Instrumen-tierung notwendig	gut, Nachweis der Fehler sensitivität	von Generier-ungs-algorith-mus abhän-gig

Tabelle 5.6: Eigenschaften von Testverfahren

Kapitel 6

Testbarkeitsanforderungen

Für eine erfolgreiche Durchführung von Softwareentwicklungsprojekten mit den vorgestellten Methoden ist nicht die analytische Qualitätssicherung allein von Bedeutung. Bereits während Analyse, Entwurf und Implementierung sollten konstruktive Maßnahmen ergriffen und Testbarkeitsanforderungen berücksichtigt werden. Dies dient einerseits der konstruktiven Vermeidung von Fehlern und andererseits erleichtert es das Aufdecken und Lokalisieren von Fehlern. Darüber hinaus verlangen einige der in dieser Arbeit dargestellten Testverfahren die Einhaltung strikter, struktureller Anforderungen. Daher werden in diesem Abschnitt die Testbarkeit und die daraus abgeleiteten Anforderungen untersucht und erläuternd dargestellt.

Die in *Real-Time Object-Oriented Modeling* verwendeten ROOMcharts sind eine Form erweiterter Zustandsautomaten. Die Erweiterung gegenüber der Grundform endlicher Zustandsautomaten [55] umfasst erweiterte Zustandsvariablen, konditionale und hierarchische Entwurfselemente, parametrisierte Ereignisse und beliebig komplexe Berechnungen mittels imperativem Aktionscode. An anderer Stelle wird in dieser Arbeit die Abbildung der konditionalen und hierarchischen Entwurfselemente auf einen endlichen Zustandsautomaten dargestellt. Die Kontrollstruktur eines ROOMcharts, das auch konditionale und hierarchische Entwurfselemente umfassen darf, wird im Folgenden als *Kontrollteil* bezeichnet. Die Verwendung von Variablen, Bedingungen und Aktionscode weicht von der Semantik endlicher Zustandsautomaten ab und wird im Folgenden als *Funktionsteil* eines ROOMchart bezeichnet. Für die Entwicklung eines Testverfahrens für ROOMcharts im modellbasierten Softwareentwicklungsprozess ist die separate Untersuchung der Testbarkeit von Kontroll- und Funktionsteil der ROOMcharts erforderlich.

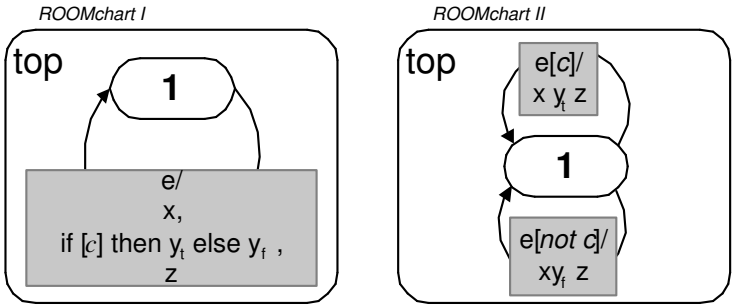


Abbildung 6.1: Auswahlpunkt Bedingung

In Abbildung 6.1 sind zwei beispielhafte ROOMcharts dargestellt. Die Funktionsteile der ROOMcharts sind grau hinterlegt. In ROOMchart *I* im Zustand *1* wird durch das Ereignis e eine Transition ausgelöst, die in Abhängigkeit von der Bedingung c entweder die Aktion $xy_t z$ oder die Aktion $xy_f z$ ausführt. In ROOMchart *II* wird in Abhängigkeit von c durch e eine von zwei Transitionen ausgeführt. Dieses Beispiel stellt zwei semantisch äquivalente Modellierungen eines Problems dar.

Während in *ROOMchart I* sich die Komplexität vor allem im Funktionsteil befindet, ist sie in *ROOMchart II* in den Kontrollteil verlagert. Die Auswirkungen dieser verschiedenen Modellierungen werden im Folgenden vergleichend dargestellt.

6.1 Anforderungen an den Kontrollteil

Für den Kontrollteil eines ROOMchart leiten sich die Testbarkeitsanforderungen aus den automatenbasierten Testverfahren ab [18]. Diese fordern eine Spezifikation in Form eines minimalen, vollständigen und endlichen Zustandsautomaten mit strengem Zusammenhang. Diese Anforderungen können in der Praxis häufig abgeschwächt werden. Aus einem ROOMchart, das nicht den Anforderungen vollständig genügt, kann ein testbares Modell abgeleitet werden. Im Folgenden werden diese Anforderungen erläuternd dargestellt.

Minimalität In einem minimalen, endlichen Automaten kann jeder Zustand durch mindestens ein Wort über dem Eingabealphabet, anhand der produzierten Ausgabe über dem Ausgabealphabet von allen anderen Zuständen des Automaten unterschieden werden. Obwohl jeder nicht-minimale Automat in einen minimalen Automaten transformiert werden kann, wird im Rahmen dieser Arbeit die Existenz eines minimalen Automaten gefordert. Die Existenz eines nicht-minimalen Automaten kann in der Praxis mehrere Ursachen haben, wenn beispielsweise optionale oder alternative Funktionen spezifiziert werden. In einer sicherheitskritischen Software sind nicht ausführbare Funktionen nicht zu tolerieren [42, 136, 67]. Daher wird im Rahmen dieser Arbeit die Existenz eines minimalen Automaten für die Anwendung automatenbasierter Testverfahren vorausgesetzt.

Vollständigkeit In einem vollständig spezifizierten endlichen Zustandsautomaten existiert in jedem Zustand für jedes Symbol im Eingabealphabet eine ausgehende Transition, die durch dieses Symbol ausgelöst wird. Nach der *Completeness Assumption*[18] wird ein unvollständig spezifizierter Automat unter der Annahme als vollständig angenommen, dass die Software bei Auftreten eines nicht spezifizierten Ereignisses keine Ausgabe produziert und im selben Zustand verbleibt. In dieser Arbeit wird ein unvollständiger Automat auf Basis der *Completeness Assumption* vervollständigt, indem für jedes nicht spezifiziertes Ereignis an einem Zustand eine reflexive Transition mit leerer Aktion ergänzt wird.

Strenger Zusammenhang Um alle Zustände eines Automaten prüfen zu können, muss jeder Zustand vom initialen Zustand erreichbar sein. Dies setzt nicht voraus, dass der initiale Zustand von jedem Zustand aus erreichbar ist. Diese Eigenschaft der Struktur eines Automaten wird als *schwacher Zusammenhang* [115] bezeichnet. Ein ROOMchart mit sicherheitskritischem Bezug wird diese Eigenschaft in der Regel besitzen [67, 136, 42]. Durch Ergänzung einer Resetfunktion kann in einem schwach zusammenhängenden Automaten der initiale Zustand aus jedem Zustand erreicht werden. Die daraus resultierende Erreichbarkeit jedes Zustands, von jedem Zustand aus, wird als *strenger Zusammenhang* bezeichnet.

Existenz einer zuverlässigen Resetfunktion Eine Resetfunktion versetzt ein ROOMchart in einen initialen Zustand und definiert die erweiterten Zustandsvariablen mit initialen Werten. Die Spezifikation einer Resetfunktion kann beispielsweise durch eine interne Gruppentransition erfolgen, dies wird an anderer Stelle dieser Arbeit demonstriert. Um einen Test auf eine Resetfunktion zu basieren, muss diese in der lauffähigen Software nachgewiesen werden. Dies kann aufgrund der oftmals sehr einfachen Struktur durch Testen oder durch verifizierende Methoden geschehen.

6.2 Anforderungen an den Funktionsteil

Um die Testfallgenerierung für einen erweiterten Zustandsautomaten zu ermöglichen muss in der Regel das, an anderer Stelle in dieser Arbeit erläuterte, Problem der Ausführbarkeit von Pfaden gelöst werden. Bei der Verwendung eines *Model Checkers* bestimmt die Semantik der Systembeschreibungssprache die erlaubte Semantik der in Aktionen und Bedingungen verwendeten programmier-

sprachlichen Konstrukte. Die Systembeschreibungssprache von Uppaal erlaubt die Definition von ganzzahligen Variablen, Konstanten und Feldern in einer der Programmiersprache *Java* [60, 100] ähnlichen Ausdruckssprache.

Die Definition von Bedingungen und Berechnungen wird mittels einer durch Kommata getrennten Liste von Ausdrücken vorgenommen. Eine Bedingung wird als eine Konjunktion dieser Ausdrücke interpretiert. Alternativ können die Kommata auch durch den UND-Operator && ersetzt werden. Die Benutzung des ODER-Operators || und des NOT-Operators *not* ist auf Ausdrücke über Ganzzahlen beschränkt.

Im Rahmen dieser Arbeit werden Bedingungen in konjunktiver Normalform gefordert.

Die Backus-Naur-Form der Listen für Bedingungen und arithmetische Operationen stellen den eingeschränkten Sprachumfang für Aktionen und Bedingungen in ROOMcharts dar:

List ::= [Expression (',' Expression)*]

Expression ::= ID | NAT | Expression '[' Expression ']' | '(' Expression ')' | Expression '++' | '++' Expression | Expression '–' | '–' Expression | Expression Assign Expression | Unary Expression | Expression Binary Expression | Expression '?' Expression ':' Expression | Expression '.' ID | 'deadlock' | 'true' | 'false'

Assign ::= ':=' | '+=' | '-=' | '*=' | '/=' | '%=' | '|=' | '&=' | '^=' | '< <=' | '> >='

Unary ::= '-' | '!' | 'not'

Binary ::= '<' | '<=' | '==' | '!=' | '>=' | '>' | '+' | '-' | '*' | '/' | '%' | '&' | '|' | '^' | '< <' | '> >' | '&&' | '||' | '<?' | '>?' | 'or' | 'and' | 'imply'

Declarations ::= (VariableDecl | ConstantDecls)*

ConstantDecls ::= 'const' ConstantDecl (',' ConstantDecl)* ';'

ConstantDecl ::= ID ArrayDecl* [Initialiser]

VariableDecls ::= Type VariableDecl (',' VariableDecl)* ';'

VariableDecl ::= ID ArrayDecl* [':=' Initialiser]

Initialiser ::= Expression | '{' Initialiser (',' Initialiser)* '}'

ArrayDecl ::= '[' Expression ']'

Type ::= Prefix TypeId [Range]

Prefix ::= 'urgent' | 'broadcast'

TypeId ::= 'int' | 'clock' | 'chan' | 'bool'

Range ::= '[' Expression ',' Expression ']'

In dieser Arbeit wird der Sprachumfang auf Operationen über ganzzahlige Variablen vom Typ *int* beschränkt. Die Semantik von Uppaal erlaubt darüber hinaus die Definition von Uhren, Feldern und booleschen Werten, die für die Anwendung in ROOMcharts geeignet erscheinen. Eine Ausweitung des hier vorgestellten Ansatzes auf diese Typen erscheint zweckmäßig und kann in einer nachfolgenden Arbeit vorgenommen werden. Die Ausdrucksmächtigkeit der Systembeschreibungssprache beeinflusst die Semantik der zu testenden ROOMcharts. Andere *Model Checker* als Uppaal besitzen ausdrucksmächtigere Spezifikationssprachen [118], deren Nutzung die Komplexität des *Model Checking*-Problems erhöht. Die Komplexität verschiedener Ansätze des *Model Checking* wird an anderer Stelle in dieser Arbeit ausführlich erläutert.

6.3 Anforderungen an Funktions- und Kontrollteil

Obwohl die beiden ROOMcharts in Abbildung 6.1 verhaltensäquivalent sind, ist ROOMchart *II* bezüglich strukturorientierter Verfahren testbarer als ROOMchart *I*. Ein minimaler transitionsüberdeckender Test würde in ROOMchart *I* entweder den Fall, dass c gilt, oder den Fall, dass c nicht gilt, prüfen. In ROOMchart *II* würde dagegen jeder transitionsüberdeckende Test auch zur

Überdeckung beider Fälle führen. Die Verwendung von ROOMcharts mit linearen Strukturen im Aktionscode eröffnet die Möglichkeit vollständige Zweigüberdeckung im Quellcode mittels Tests herzustellen, die auf ROOMcharts basieren. Linearer Code enthält keine verzweigenden Kontrollstrukturen. Wenn alle Bedingungen eines erweiterten Zustandsautomaten in konjunktiver Normalform vorliegen und jede Aktion nur einen Pfad besitzt, wird der Automat als *Normalform* [142] bezeichnet.

Für ein ROOMchart in Normalform ist die Relation zwischen Transitionsüberdeckung und Zweigüberdeckung des generierten Quellcodes nur vom verwendeten Codegenerator abhängig. Daher wird im Rahmen dieser Arbeit die Existenz eines ROOMcharts in Normalform gefordert.

Für zukünftige Forschungsarbeiten ist die detaillierte Untersuchung der Abbildung verschiedener Überdeckungskriterien eines ROOMchart in Normalform auf den generierten Quellcode empfehlenswert. Ein Nachweis der wichtigsten Überdeckungskriterien im Quellcode aus einem Test basierend auf dem ROOMchart kann die Kosteneffizienz im sicherheitskritischen Umfeld steigern und den Nachweis der Einhaltung von maßgeblichen Standards und Normen [67, 42, 136] deutlich vereinfachen.

Kapitel 7

Hierarchische Pseudozustände

In dieser Arbeit wurden Testbarkeitsbedingungen bezüglich der Struktur des Kontroll- und des Funktionsteils von ROOMcharts aufgestellt. Eine wichtige Bedingung betrifft die Struktur des Aktionscodes und resultiert in der Forderung nach ROOMcharts in Normalform. Um den Entwurf von ROOMcharts in Normalform zu vereinfachen, wird in diesem Kapitel das Entwurfselement des *hierarchischen Pseudozustands* vorgestellt. Ein Pseudozustand ist ein konstruktives Entwurfselement, um kontinuierliche Kontrollflüsse zu modellieren. Im Gegensatz zu regulären Zuständen wird die Ereignisverarbeitung beim Erreichen eines Pseudozustands nicht angehalten, sondern mit einer ausgehenden Transition, bis zum Erreichen eines regulären Zustands, unverzüglich fortgesetzt. Ein Auswahlpunkt ist eine typische Form eines Pseudozustands, der zur Modellierung bedingter Kontrollflüsse dient [143, 64]. Andere Pseudozustände sind Anschlusspunkte und initiale Punkte. Die Verwendung von Auswahlpunkten für die Modellierung kontinuierlicher, bedingter Kontrollflüsse eignet sich besonders gut zur Erstellung von ROOMcharts in Normalform.

7.1 Transformation kontinuierlicher Kontrollflüsse

Ein Auswahlpunkt teilt eine Transition in mehrere, nicht notwendigerweise disjunkte Zweige [143, 113, 64], deren Endzustände und Aktionen unterschiedlich sein können. Die Bedingung unter der ein Zweig ausgeführt wird, kann von Werten abhängen, die im vorangehenden Transitionsteil berechnet werden. Jeder Zweig eines Auswahlpunkts ist mit einer Wächterbedingung ausgestattet, die dynamisch evaluiert wird, wenn der Auswahlpunkt erreicht wird. Da die Verwendung von nicht disjunkten Bedingungen theoretisch zu nichtdeterministischem Verhalten führen kann, wird für die Entwicklung von technischer Software die ausschließliche Verwendung von binären Auswahlpunkten empfohlen. Ein binärer Auswahlpunkt verknüpft beliebig viele eingehende Transitionssegmente mit maximal zwei ausgehenden Transitionssegmenten. Das mit *T (engl. true)* beschriftete Transitionssegment wird ausgelöst, wenn die Bedingung gilt. Das mit *F (engl. false)* beschriftete Transitionssegment wird ausgelöst, wenn die Bedingung nicht gilt. Um *Deadlocks* [117] zu vermeiden, wird für jede Verwendung von binären Auswahlpunkten die Definition beider ausgehenden Transitionssegmente empfohlen. Eine Ausnahme dieser Empfehlung stellen Sequenzpunkte dar, die zur Definition sequentieller, kontinuierlicher Kontrollflüsse dienen und nicht blockieren können.

Sequenzpunkte Ein Sequenzpunkt ist äquivalent mit einem Auswahlpunkt mit der konstanten Bedingung *Wahr*. In diesem Fall wird immer das mit T beschriftete Transitionssegment ausgelöst, während das mit F beschriftete Transitionssegment vernachlässigt werden kann. In den folgenden grafischen Darstellungen werden Sequenzpunkte als unbeschriftete, leere Punkte dargestellt.

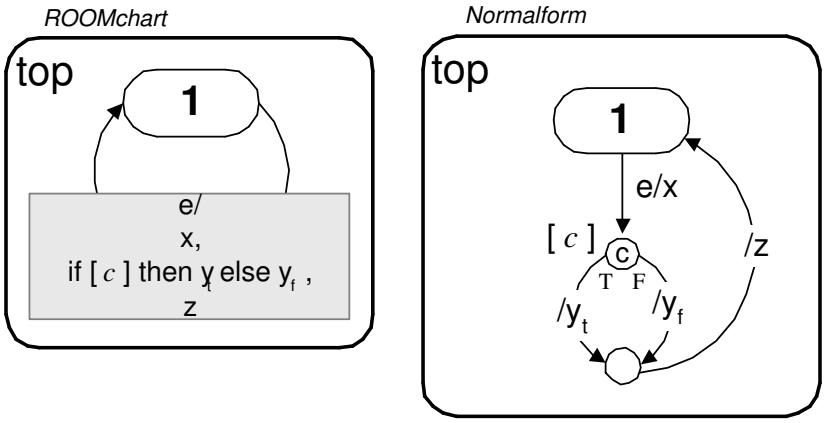


Abbildung 7.1: Auswahlpunkt Bedingung

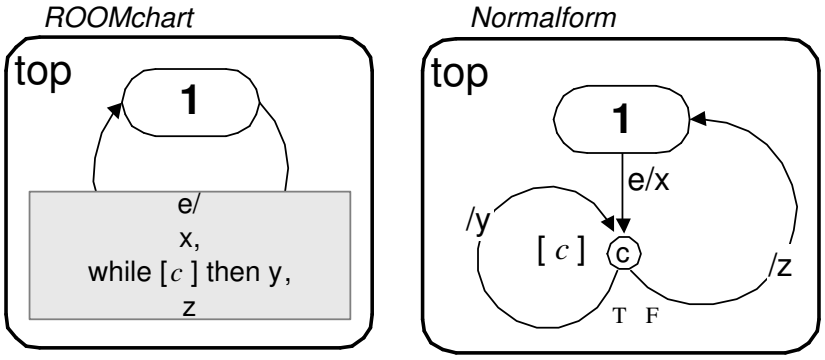


Abbildung 7.2: Auswahlpunkt abweisende Schleife

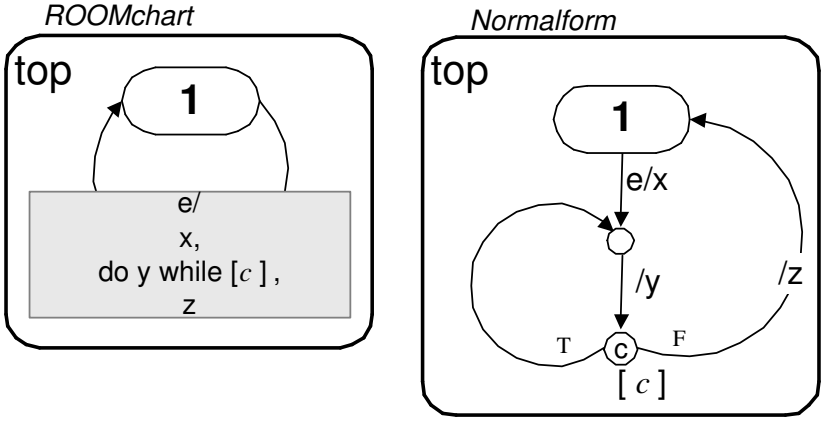


Abbildung 7.3: Auswahlpunkt nicht abweisende Schleife

Die folgenden Beispiele in den Abbildungen 7.1, 7.2, 7.4 und 7.5 stellen die Verwendung von Auswahlpunkten zur Modellierung bedingter, kontinuierlicher Kontrollstrukturen dar. Auf der linken Seite der Abbildungen werden ROOMcharts mit Bedingungen und Schleifen im Aktionscode dargestellt, während auf der rechten Seite semantisch äquivalente ROOMcharts in Normalform, unter Verwendung von Auswahlpunkten, präsentiert werden.

Eine Bedingungsanweisung stellt eine binäre Verzweigung des Kontrollflusses dar, wobei die Verzweigung in Abhängigkeit von einer Bedingung erfolgt. Eine Bedingungsanweisung kann verhaltensäquivalent durch einen binären Auswahlpunkt ersetzt werden.

Das ROOMchart auf der linken Seite in Abbildung 7.1 enthält den Zustand *1* mit einer reflexiven Transition, die den Trigger e besitzt und die Aktionen xy_tz oder xy_fz in Abhängigkeit von der Bedingung c ausführt. Das ROOMchart, in Normalform auf der rechten Seite, besitzt anstelle der reflexiven Transition mit komplexem Code eine Struktur aus einem Auswahl- und einem Sequenzpunkt. In Zustand *1* wird ebenfalls durch e eine Transition ausgelöst und die Aktionen xy_tz oder xy_fz werden in Abhängigkeit von der Bedingung c ausgeführt.

Eine Schleife kontrolliert die Ausführung der in der Schleife definierten Aktionen, so dass diese nicht oder beliebig oft ausgeführt werden. Eine Schleifenanweisung kann durch einen eine zyklische Struktur aus Auswahlpunkten verhaltensäquivalent ersetzt werden.

Das Beispiel in Abbildung 7.2 stellt die Abbildung einer abweisenden Schleife auf ein ROOMchart in Normalform mit Auswahlpunkten dar. Das ROOMchart auf der linken Seite enthält den Zustand *1* mit einer reflexiven Transition, die durch e ausgelöst wird und die folgenden Aktionen in Abhängigkeit von c ausführt:

$$\textit{Pfadausdruck Abb. 7.2}: \bigvee_{i=0}^{n} xy^iz, \text{ mit } n, i \in \mathbb{Z}$$

Auf der rechten Seite der Abbildung 7.2 ist ein semantisch äquivalentes ROOMchart in Normalform gegeben. Dieses enthält einen Auswahlpunkt, der durch einen reflexiven *Wahr*-Zweig wiederholt erreicht wird. Die Aktion im *Wahr*-Zweig y wird daher solange wiederholt ausgeführt, wie die Bedingung c nach jeder Ausführung von y zu *Wahr* evaluiert wird. Die gesamte Struktur wird daher durch e ausgelöst und führt in Abhängigkeit von c die gleichen Aktionen aus.

Das Beispiel der nicht abweisenden Schleife in Abbildung 7.3 kann auf gleiche Art und Weise auf ein ROOMchart in Normalform abgebildet werden. Der Pfadausdruck dieses Beispiels ist:

$$\textit{Pfadausdruck Abb. 7.3}: \bigvee_{i=1}^{n} xy^iz, \text{ mit } n, i \in \mathbb{Z}$$

Durch Schachtelung werden Bedingungen in einen kausalen Zusammenhang gesetzt. Die innere Bedingung wird nur dann evaluiert, wenn die äußere Bedingung gilt. Analog kann eine innere Bedingung gerade dann evaluiert werden, wenn die äußere Bedingung nicht gilt. Eine Schachtelung von Bedingungsanweisungen kann verhaltensäquivalent durch gekettete Auswahlpunkte ersetzt werden.

Das ROOMchart auf der linken Seite in Abbildung 7.4 enthält die Schachtelung von zwei Bedingungsanweisungen. Die reflexive Transition am Zustand *1* wird durch e ausgelöst und führt die Aktionen xz oder xyz in Abhängigkeit von den Bedingungen c_0 und c_1 aus. Wenn die Bedingung c_0 erfüllt ist, wird die Bedingung c_1 evaluiert. Für den Fall, dass c_1 wahr ist, wird die Aktion xyz ausgeführt, und sonst wird die Aktion xz ausgeführt. Auf der rechten Seite der Abbildung 7.4 ist ein ROOMchart in Normalform mit einer zum linken ROOMchart äquivalenten Semantik dargestellt. Im ROOMchart in Normalform sind zwei Auswahlpunkte enthalten, die durch die Transition mit dem Ereignis e und der Aktion x erreicht werden. Die zweite, innere Bedingung c_1 wird evaluiert, wenn die erste, äußere Bedingung erfüllt ist. Wenn die Bedingung c_1 erfüllt ist, wird die Aktion xyz ausgeführt, und sonst wird die Bedingung xz ausgeführt. Dies ist äquivalent zu der Aktion der reflexiven Transition im linken ROOMchart.

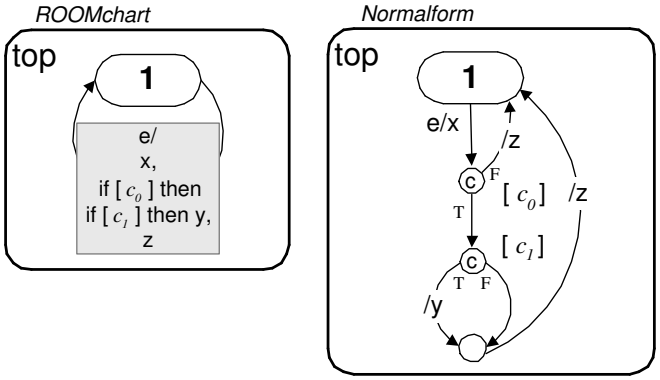


Abbildung 7.4: Auswahlpunkt geschachtelte Bedingungen

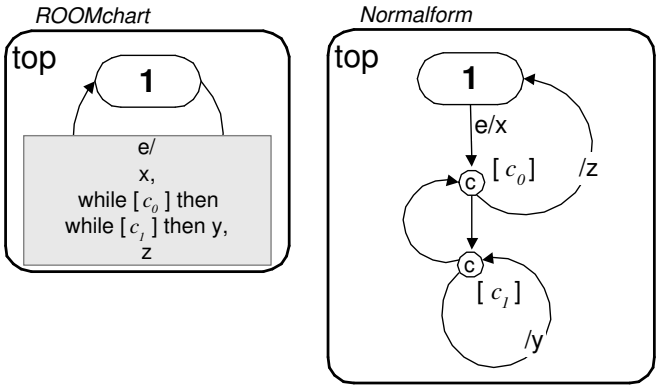


Abbildung 7.5: Auswahlpunkt geschachtelte abweisende Schleifen

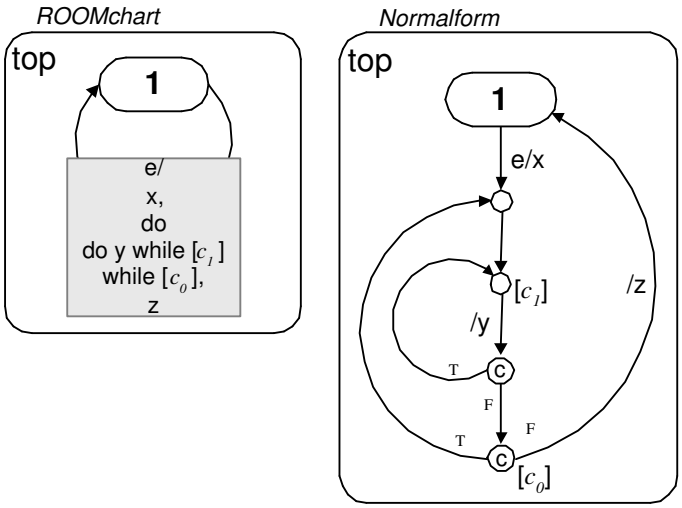


Abbildung 7.6: Auswahlpunkt geschachtelte nicht abweisende Schleifen

Durch Schachtelung werden Schleifen in einen kausalen Zusammenhang gesetzt, so dass die innere Schleife nur dann evaluiert wird, solange die äußere Schleifenbedingung erfüllt ist. Eine Schachtelung von Schleifenanweisungen kann verhaltensäquivalent durch gekettete Auswahlpunkte ersetzt werden.

Das ROOMchart auf der linken Seite in Abbildung 7.5 besitzt einen Zustand mit einer reflexiven Transition, die durch e ausgelöst wird und die Aktionen

$$\textit{Pfadausdruck Abb. 7.5:} \bigvee_{i=0}^{n} \bigvee_{j=0}^{m} xy^{i*j}z, \text{ mit } m, n, i, j \in \mathbb{Z},$$

in Abhängigkeit von den Bedingungen c_0 und c_1 ausführt. Das Normalform ROOMchart auf der rechten Seite besitzt zwei Auswahlpunkte, so dass die Bedingung c_1 nur dann evaluiert wird, wenn c_0 gültig ist. Damit ist die Struktur der Auswahlpunkte auf der rechten Seite semantisch äquivalent zu der Schachtelung der Schleifenanweisungen auf der linken Seite.

Abbildung 7.6 stellt eine geschachtelte nicht abweisende Schleifen dar. Der Pfadausdruck für diese Konstrukte lautet

$$\textit{Pfadausdruck Abb. 7.6:} \bigvee_{i=1}^{n} \bigvee_{j=1}^{m} xy^{i*j}z, \text{ mit } m, n, i, j \in \mathbb{Z}.$$

Die Ausführungen für abweisende Schleifen gelten für diesen Pfadausdruck unverändert. Es ist leicht nachzuvollziehen, dass die Struktur auf der rechten Seite in Abbildung 7.6 semantisch äquivalent zu der linken Seite ist.

7.2 Kapselung kontinuierlicher Kontrollflüsse

Die Verwendung von Auswahlpunkten ist intuitiv und anschaulich, kann aber zu komplexen und unübersichtlich strukturierten ROOMcharts führen. Ein wichtiges Mittel der Komplexitätsbeherrschung stellt die Hierarchisierung dar, welche in ROOMcharts auf dem Entwurfselement des hierarchischen Zustands beruht. Ein hierarchischer Zustand definiert eine Anzahl erweiterter Zustandsvariablen, Eingangs- und Ausgangsaktionen, sowie die untergeordneten Zustände und Transitionen. Jeder hierarchische Zustand enthält einen vollständigen erweiterten Zustandsautomaten und abstrahiert dadurch ein detailliertes Verhalten. Weiterhin werden Gedächtnispunkte, Gruppentransitionen, segmentierte Transitionen und Anschlusspunkte für die Verbindung angrenzender hierarchischer Ebenen eines ROOMcharts genutzt [143]. Ein hierarchischer Zustand definiert einen eigenen Namensraum, ähnlich zu Programmblöcken in Programmiersprachen [100, 98, 150, 1]. In dieser Arbeit wird die Ableitung von flachen Testmodellen aus hierarchischen ROOMcharts ausführlich behandelt, daher wird an dieser Stelle nicht auf die Problematik von Hierarchien bei der Entwicklung strukturorientierter Tests eingegangen, sondern auf die Verbesserung der Testbarkeit durch Einsatz von hierarchischen Elementen.

Hierarchische Pseudozustände Ein hierarchischer Pseudozustand ist ein zusammengesetzter Zustand, der ausschließlich Pseudozustände enthält. Er wird durch ein Ereignis betreten und automatisch wieder verlassen, wenn eine im hierarchischen Zustand definierte Verarbeitung abgeschlossen ist.

Die Darstellung von komplexen Kontrollstrukturen innerhalb eines hierarchischen Pseudozustands ist semantisch äquivalent mit der Definition einer Prozedur ohne Rückgabewert in einer strukturierten Programmiersprache. Da kein expliziter Rückgabewert definiert ist, müssen Daten über globale Variablen übergeben werden. Bei der Kapselung von kontinuierlichen Abläufen in einem hierarchischen Pseudozustand s_h , muss die auslösende Transition außerhalb von s_h liegen, während alle Aktionen vorzugsweise innerhalb von s_h ausgeführt werden. Durch Kapselung der Verzweigungsstruktur und der Aktionen in s_h wird die separate Prüfung dieser komplexen Verarbeitungslogik begünstigt und die abstrakte Kontrollstruktur des ROOMcharts bleibt erhalten. Durch eine kompakte und vollständige Darstellung wird die separate Prüfung der Verarbeitungslogik eines hierarchischen Pseudozustands unterstützt. Für die Definition einer vollständigen, kompakten

und intuitiven Darstellungsform ist die Beschränkung auf einen Ein- und einen Ausgangspunkt empfehlenswert.

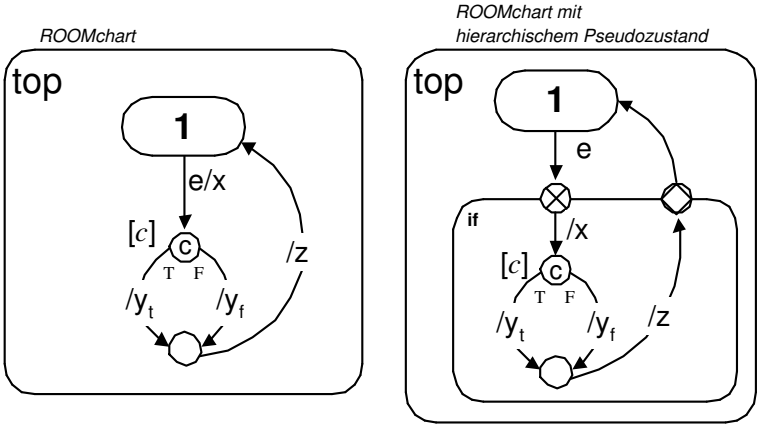


Abbildung 7.7: Hierarchischer Pseudozustand *if*

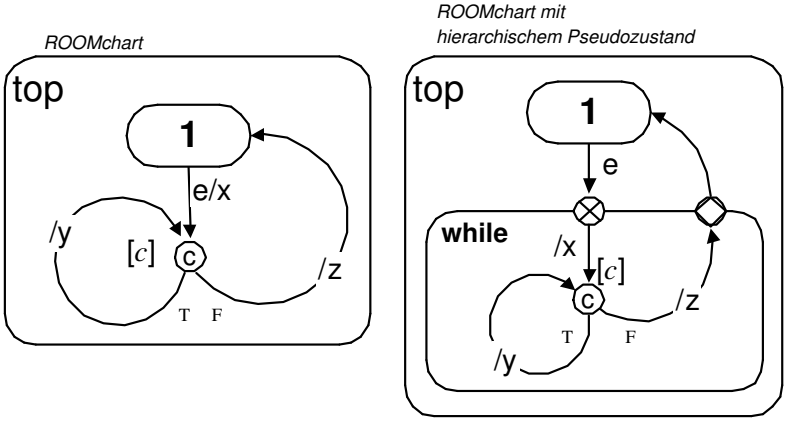


Abbildung 7.8: Hierarchischer Pseudozustand *while*

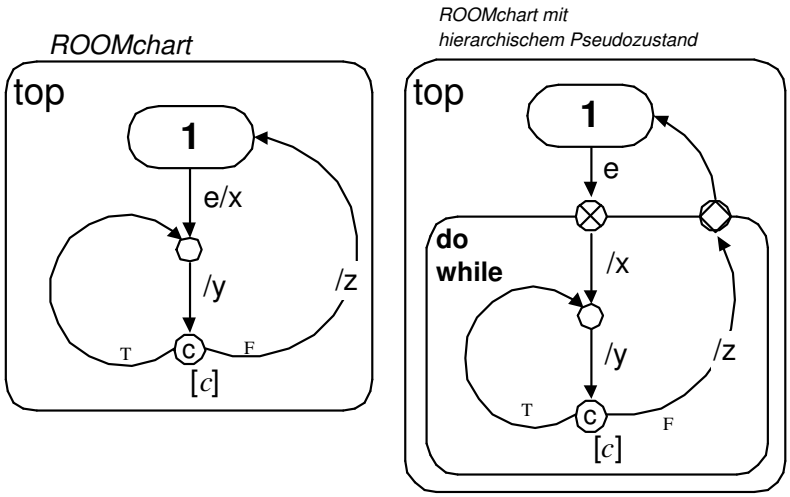


Abbildung 7.9: Hierarchischer Pseudozustand *do while*

Ein Auswahlpunkt einer Bedingung oder Schleife kann zweckmäßig in einem hierarchischen Pseudozustand gekapselt werden. Das auslösende Ereignis muss außerhalb der Kapselung verbleiben, um den Kontrollfluss auf abstrakter Ebene zu bewahren. Alle mit dem auslösenden Ereignis assoziierten Aktionen müssen im hierarchischen Pseudozustand gekapselt werden. Die Beispiele in den Abbildungen 7.7, 7.8, 7.10 und 7.11 stellen die Definition hierarchischer Pseudozustände mit den im Vorangegangenen beispielhaft demonstrierten Entwurfselementen dar.

In Abbildung 7.7 wird die Kapselung eines Auswahlpunkts zu einem hierarchischen Pseudozustand dargestellt. Auf der linken Seite wird bei Auftreten des Ereignisses e und in Abhängigkeit von der Bedingung c entweder die Aktion xy_tz oder die Aktion xy_fz produziert. Durch Definition eines hierarchischen Zustands *if* um den Auswahlpunkt und den Durchgangspunkt entsteht ein hierarchischer Pseudozustand. Der Eingang und der Ausgang sind durch Anschlusspunkte definiert. Ein hierarchischer *if*-Pseudozustand kann generisch zur Definition bedingter kontinuierlicher Kontrollflüsse dienen.

In Abbildung 7.8 ist ein Auswahlpunkt mit der Semantik einer abweisenden Schleife dargestellt, der in einem hierarchischen Pseudozustand *while* gekapselt wird. Durch Auftreten des Ereignisses e werden unter der Bedingung c die folgenden Aktionen ausgeführt:

$$\textit{Pfadausdruck Abb. 7.8:} \bigvee_{i=0}^{n} xy^iz, \text{ mit } n, i \in \mathbb{Z}$$

Durch die Kapselung in einem hierarchischen Zustand wird das Verhalten bewahrt. Der hierarchische Pseudozustand *while* der abweisenden Schleife besitzt einen Eingangspunkt und einen Ausgangspunkt.

Die Semantik einer nicht abweisenden Schleife kann analog durch einen hierarchischen Pseudozustand *do while* dargestellt werden. In Abbildung 7.9 ist das Beispiel eines ROOMcharts mit *do while*-Pseudozustand dargestellt, das die folgenden Aktionen ausführt:

$$\textit{Pfadausdruck Abb. 7.9:} \bigvee_{i=1}^{n} xy^iz, \text{ mit } n, i \in \mathbb{Z}$$

In Abbildung 7.10 sind zwei Auswahlpunkte mit der Semantik geschachtelter Bedingungsanweisungen dargestellt, die in einem hierarchischen Pseudozustand gekapselt werden. Nach Auftreten des Ereignisses e werden in Abhängigkeit von den Bedingungen c_0 und c_1 die Aktionen xz oder xyz ausgeführt. Wenn c_0 und c_1 beide erfüllt sind, wird xyz ausgeführt. In jedem anderen Fall wird xz ausgeführt. Durch die Schachtelung in einem hierarchischen Zustand bleibt dieses Verhalten erhalten.

In Abbildung 7.11 sind zwei Auswahlpunkte mit der Semantik geschachtelter, abweisender Schleifen dargestellt, die in einem hierarchischen Pseudozustand gekapselt werden. Nach dem Auftreten von e führt das ROOMchart auf der linken Seite in Abhängigkeit von c_0 und c_1 die folgenden Aktionen aus:

Pfadausdruck Abb. 7.11: $\bigvee_{i=0}^{m} \bigvee_{j=0}^{n} xy^{i*j}z$, mit $m, n, i, j \in \mathbb{Z}$

Durch die Kapselung in einem hierarchischen Zustand wird dieses Verhalten nicht verändert. Das Beispiel in Abbildung 7.12 stellt ein ROOMchart mit geschachtelten, nicht abweisenden Schleifen dar, die analog folgende Aktionen ausführen:

Pfadausdruck Abb. 7.12: $\bigvee_{i=1}^{m} \bigvee_{j=1}^{n} xy^{i*j}z$, mit $m, n, i, j \in \mathbb{Z}$

Das Konzept der hierarchischen Pseudozustände ermöglicht den gekapselten Entwurf und separaten Test komplexer kontinuierlicher Kontrollflüsse in einem ROOMchart. Ein hierarchischer Pseudozustand kann separat mit prozeduralen strukturorientierten Modultestverfahren geprüft werden [10, 92]. Weiterhin kann die Abbildung von Aktionscode auf hierarchische Pseudozustände dermaßen formalisiert werden, dass eine automatische Umformung von ROOMcharts in die Normalform möglich ist.

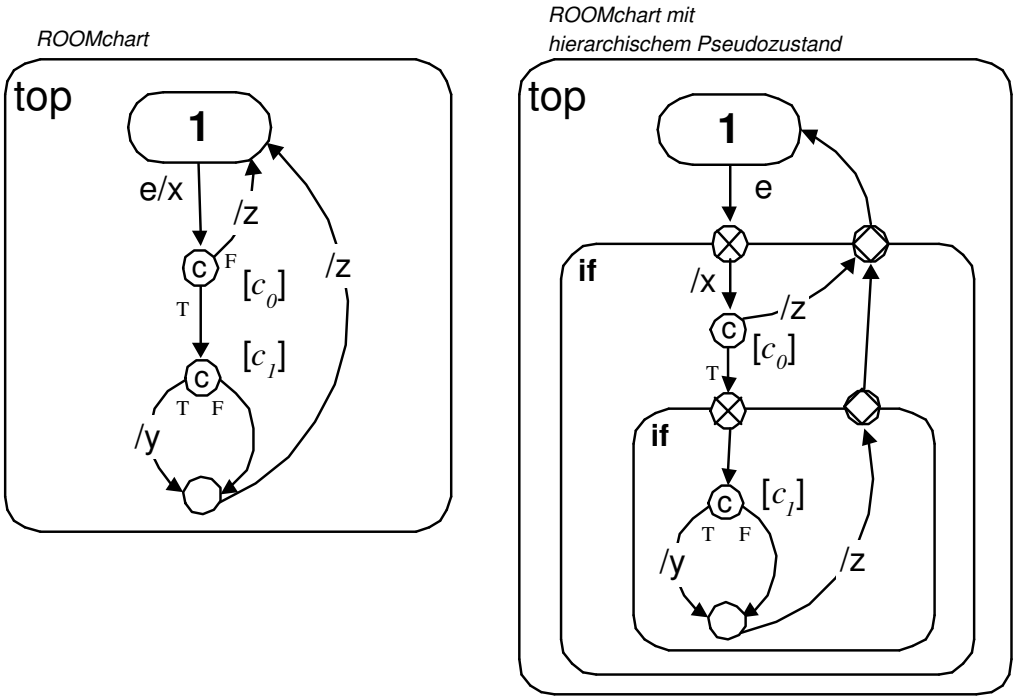


Abbildung 7.10: Geschachtelte *if*-Pseudozustände

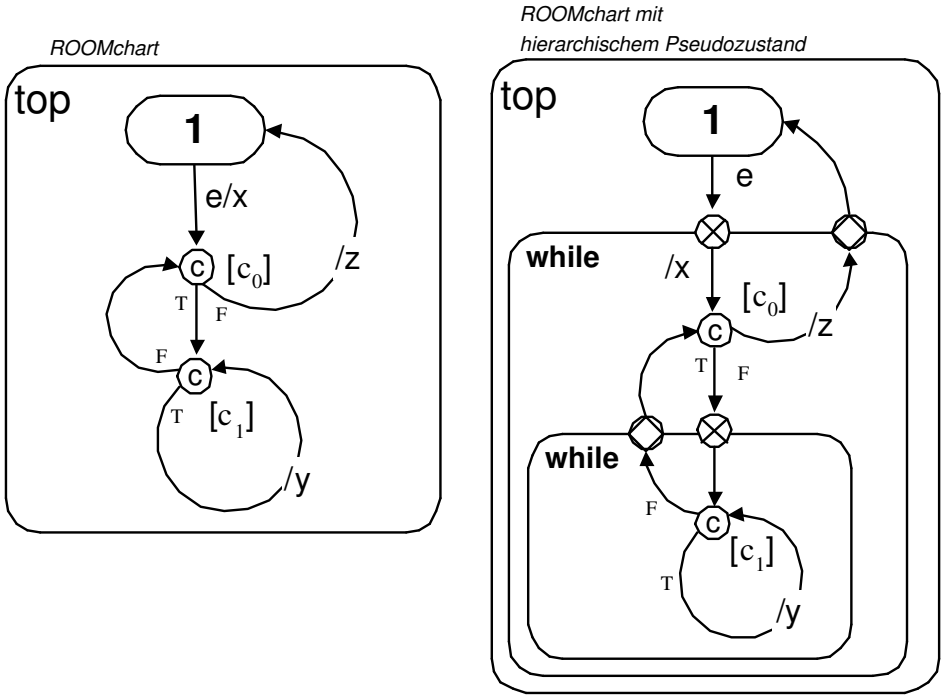


Abbildung 7.11: Geschachtelte *while*-Pseudozustände

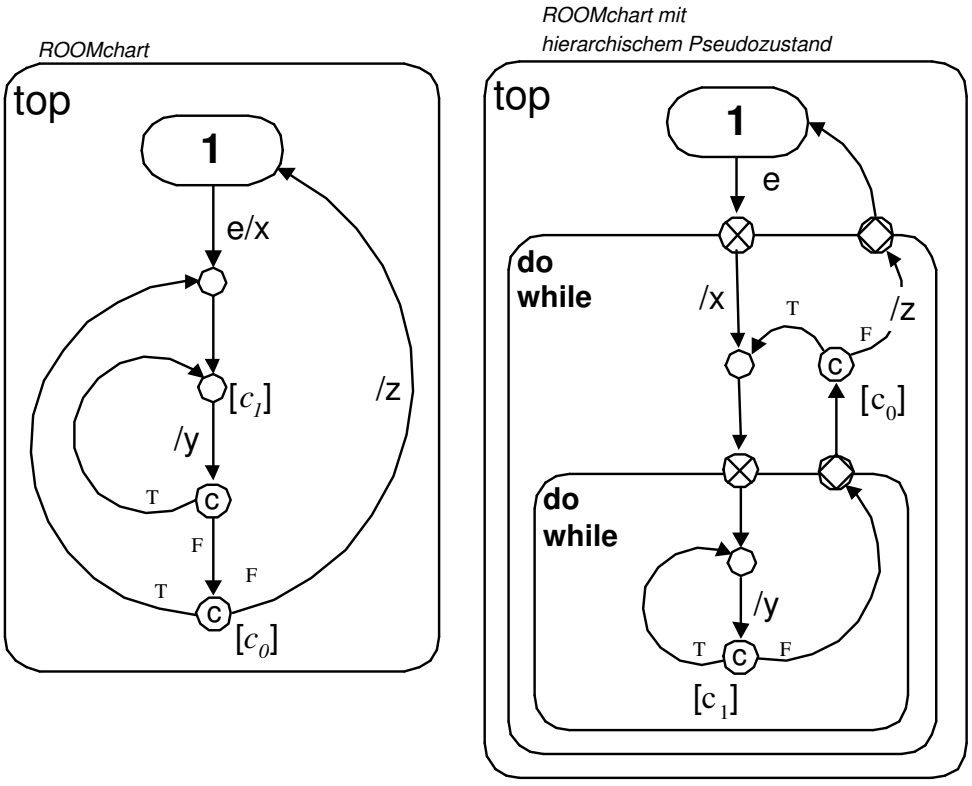


Abbildung 7.12: Geschachtelte *do while*-Pseudozustände

Kapitel 8

Testmodellgenerierung

Der in *Real-Time Object-Oriented Modeling* zur Verhaltensmodellierung verwendete Formalismus *ROOMcharts* ist nicht ohne Modifikationen für die Generierung von Testfällen mit den meisten automatenbasierten Testverfahren geeignet. Viele dieser Verfahren basieren auf dem semantisch einfacheren Formalismus endlicher Zustandsautomaten [18]. Aus den zu testenden ROOMcharts ist daher ein semantisch äquivalentes Testmodell in Form eines endlichen Zustandsautomaten abzuleiten.
Die Erweiterungen der ROOMcharts gegenüber endlichen Zustandsautomaten dienen der Komplexitätsbeherrschung und erweitern die Semantik um Elemente zur Daten- und Funktionsmodellierung.

In diesem Abschnitt werden die notwendigen Schritte zur Ableitung eines endlichen Zustandsautomaten aus ROOMcharts hergeleitet und in Beispielen demonstriert. Jeder Ableitungsschritt stellt eine Vorschrift dar, eine bestimmte Art von Erweiterung zu behandeln. Am Ende des Abschnitts werden die einzelnen Schritte zusammengefasst.

8.1 Ein- und Ausgabealphabet

Das Ein- bzw. Ausgabealphabet eines ROOMcharts wird durch die Ein- bzw. Ausgangssignale auf den Ports des betreffenden Aktors definiert. Die Ports eines Aktors, und damit die Schnittstelle des Aktors, werden im Strukturdiagramm des Systems oder im Strukturdiagramm eines Aktors definiert. Das Protokoll eines Ports definiert die Eingangs- bzw. Ausgangssignale, die durch den Port transportiert werden dürfen.

Ein eintreffendes Signal auf einem Port kann in einem Aktor nur verarbeitet werden, wenn es sich um einen Endport handelt. Signale auf Relaisports dienen dem Transport der Signale zu Subaktoren, und werden daher auf Aktorebene ignoriert.

Durch Konjugieren eines Ports werden die Eingangssignale des Ports zu Ausgangssignalen der Kapsel, zu welcher der Port gehört. Ebenso werden durch Konjugieren die Ausgangssignale des Ports zu Eingangssignalen der Kapsel. Dies muss bei der Konstruktion des Eingabe- und des Ausgabealphabets des Testmodells berücksichtigt werden.

Ein Port kann multipel definiert werden, um die Breitbandkommunikation eines Aktors mit mehreren Empfängern bzw. Sendern zu ermöglichen. Im Modultest muss die Breitbandkommunikation zwischen mehreren Modulen aus folgenden Gründen nicht berücksichtigt werden: Wenn ein Signal auf einem multiplen Port eintrifft, wurde es von einem Aktor gesendet - dies ist äquivalent zum Empfang eines Signals auf einem singulären Port. Wenn ein Signal über einen multiplen Port gesendet wird, wird die Multiplikation erst im Port vorgenommen - dies ist modulintern äquivalent zum Senden eines Signals über einen singulären Port.

In Abbildung 8.1 ist beispielhaft das Strukturdiagramm eines Aktors mit einem Subaktor dargestellt, in dem verschiedene Arten singulärer Ports definiert sind. Das Protokoll P enthält das Eingangssignal *s1* vom leeren Datentyp *void* und ein Ausgangssignal *s2* vom Datentyp *int*. Das symmetrische Protokoll Q enthält am Ein- und Ausgang das ganzzahlige Signal *s3* vom Typ *int*. Der Aktor besitzt die Endports *p3* und *p4*, vom Typ P, sowie *p5* vom Typ Q, die das Alphabet des Testmodells definieren. Der interne Endport *p5* ist an den Subaktor gebunden. Der Endport

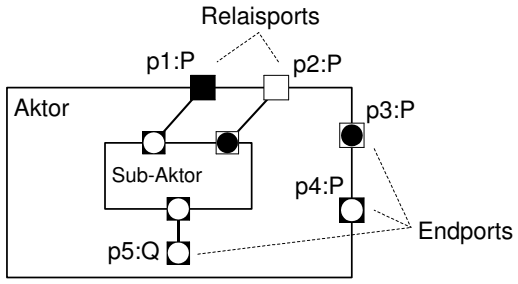


Abbildung 8.1: Strukturdiagramm

p3 ist konjugiert. Das Eingangsalphabet X bildet sich aus den Eingangssignalen von *p4* und *p5* sowie dem Ausgangssignal von *p3*. Das Ausgangsalphabet Y bildet sich aus den Ausgangssignalen von *p4* und *p5* sowie dem Eingangssignal von *p3*.

Eingaben $X = \{p3.s2, p4.s1, p5.s3\}$

Ausgaben $Y = \{p3.s1, p4.s2, p5.s3\}$

8.2 Unvollständig spezifizierte ROOMcharts

Wenn in einem Zustand s ein Eingangsereignis e nicht zur Auslösung einer ausgehenden Transition dient, wird e als nicht spezifiziert in s bezeichnet. Ein Zustandsautomat wird als unvollständig spezifiziert bezeichnet, wenn nicht in jedem Zustand alle Eingabeereignisse spezifiziert sind.

Häufig sind ROOMcharts nur unvollständig spezifiziert. Dies geschieht aus Gründen der Lesbarkeit und unter der Annahme, dass diese Ereignisse ignoriert werden sollen, wenn das System sich im betreffenden Zustand befindet. Daher wird erwartet, dass bei Auftreten eines unspezifizierten Ereignisses der Automat in seinem Zustand ohne eine sichtbare Reaktion verharrt. Diese Annahme entspricht der *Completeness Assumption* von Bochmann und Petrenko [18]. Bei der Ableitung eines Testmodells werden unspezifizierte Ereignisse auf reflexive Transitionen abgebildet, die eine leere Ausgabe besitzen.

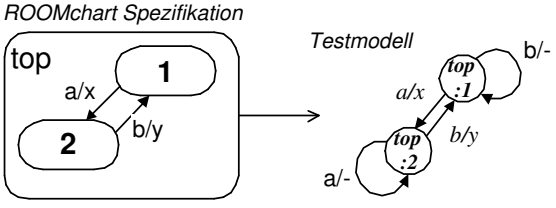


Abbildung 8.2: *Completeness Assumption*

Das Beispiel in Abbildung 8.2 stellt die Anwendung der *Completeness Assumption* anhand eines einfachen ROOMcharts mit zwei Zuständen dar. Von Zustand *top:1* führt in Zustand *top:2* eine Transition mit dem auslösenden Ereignis a und der Ausgabe x. Von *top:2* nach *top:1* führt eine Transition mit dem auslösenden Ereignis b und der Ausgabe y. Der Automat ist unvollständig spezifiziert, da in *top:1* die Spezifikation des Ereignisses b und in *top:2* die Spezifikation des Ereignisses a fehlt. In der Ableitung des Testmodells werden daher reflexive Transitionen an *top:1* und *top:2* ergänzt, welche keine Ausgabe produzieren.

8.3 Substitutionstabelle

Die auslösenden Ereignisse und die Ausgabeereignisse von Transitionen können in ROOMcharts von jedem Typ sein, den die zugrunde liegende objektorientierte Programmiersprache zulässt. Das semantisch einfachere Modell endlicher Zustandsautomaten erlaubt dagegen ausschließlich einfache Symbole als Eingangs- und Ausgangsereignisse. Auslösende Ereignisse bestehen in *Real-Time Object-Oriented Modeling* aus einem Signal, einer optionalen Parameterliste und einer optionalen Wächterbedingung. Die komplexen Ereignisse der ROOMcharts müssen auf eine endliche Menge von Symbolen abgebildet werden, um eine Anwendung automatenbasierter Testverfahren zu ermöglichen. Ein Tripel aus Signal, Parameterliste und Wächterbedingung wird im Testmodell auf ein eindeutiges Symbol abgebildet. Mehrere Tripel mit gleichen auslösenden Ereignissen und semantisch äquivalenten Wächterbedingungen werden auf gleiche Symbole abgebildet.

Ebenso wie auslösende Ereignisse werden Aktionen durch Symbole ersetzt. Die betrachteten Systeme besitzen sehr einfach strukturierte Aktionen mit nur einem Pfad. Die Aktionen werden in die extern sichtbaren und die internen Bestandteile separiert, da für die automatenbasierten Testverfahren nur das extern sichtbare Verhalten von Bedeutung ist. Die extern sichtbaren Aktionen werden im Programmtext durch Anweisungen zum Verschicken von Signalen auf die Ports des Aktors spezifiziert. Diese stellen die Ausgaben des Automaten dar und dienen der Prüfung der Kontrollstruktur des ROOMcharts. Die externen Ausgabeereignisse eines ROOMcharts können eindeutig anhand von Port und Signal im Programmtext identifiziert werden.

Die internen Aktionen bestehen aus beliebigen Anweisungen in der zugrunde liegenden objektorientierten Programmiersprache und bestimmen die Ausführbarkeit von Pfaden im ROOMchart.

Das Beispiel in Abbildung 8.3 demonstriert die Abbildung der Transitionsbeschriftungen eines ROOMcharts auf einen endlichen Zustandsautomaten des Testmodells. Die zugehörige Substitutionstabelle 8.1 enthält alle Abbildungen der komplexen Transitionsbeschriftungen im ROOMchart auf einfache Symbole.

Die Differenzierung der Eingabeereignisse auf *portA.signal1* erfolgt mittels der Bedingungen über *x*. Die Differenzierung der Ausgabeereignisse auf *portA.signal2* erfolgt über die Bedingungen über *x* und die Werte der Datenobjekte *x*, 0 und -*x*.

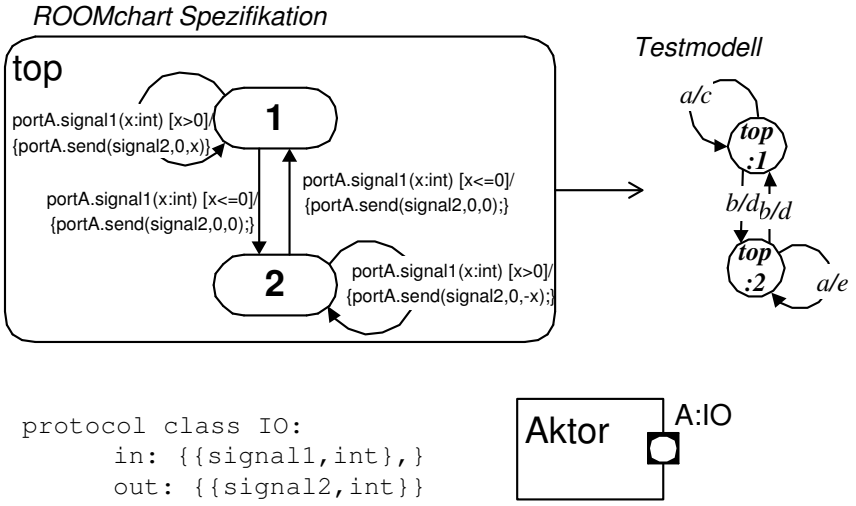


Abbildung 8.3: Transitionsbeschriftungen

8.4 Reset von ROOMcharts

Die Existenz einer zuverlässigen Resetfunktion ist ein notwendiges Kriterium für die Anwendbarkeit der meisten automatenbasierten Testverfahren [18]. Die Resetfunktion sollte möglichst früh in die

Entwurfsmodell	Testmodell
$portA.signal1(x : int)[x > 0]$	a
$portA.signal1(x : int)[x \leq 0]$	b
$portA.send(signal2, 0, x)[x > 0]$	c
$portA.send(signal2, 0, 0)[x \leq 0]$	d
$portA.send(signal2, 0, -x)[x > 0]$	e

Tabelle 8.1: Substitutionstabelle zu Abbildung 8.3

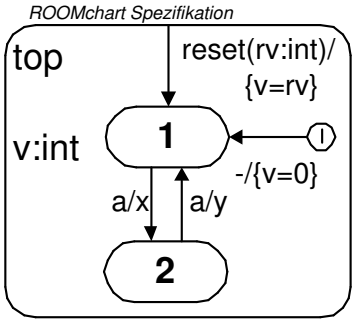


Abbildung 8.4: Resetfunktion

Modulanforderungen aufgenommen und bereits während der Implementierung realisiert werden. Es ist ebenso möglich, eine Resetfunktion zu Testzwecken automatisch zu generieren. Einige modellbasierte Softwareentwicklungsumgebungen weisen eine Resetfunktion während der simulierten Ausführung auf. Die Generierung einer Resetfunktion oder die Nutzung vorhandener Resetfunktionen ermöglichen die Funktionalität des Moduls minimal zu halten. Weiterhin können Fehler in solchen Funktionen nahezu ausgeschlossen werden. Eine Resetfunktion wird durch eine Gruppentransition *reset(P)* spezifiziert, wobei P für eine beliebige Parameterliste steht, die es ermöglicht, den erweiterten Zustand zu definieren. Somit kann jede erweiterte Zustandsvariable nach Bedarf definiert werden.

Das beispielhafte ROOMchart in Abbildung 8.4 ist um eine Resetfunktion erweitert. Diese wird als Gruppentransition spezifiziert und bei der Ableitung eines Testmodells ignoriert.

8.5 Abstrakte Ereignistypen

Ein Ereignis korrespondiert in *Real-Time Object-Oriented Modeling* mit dem Auftreten eines Signals an einem Endport. Der Typ eines Signals ist auf die Typen der genutzten objektorientierten Programmiersprache beschränkt. Daher können Signale beinahe beliebig abstrakte Informationen transportieren. Dies stellt einen erheblichen Unterschied zum Modell des endlichen Zustandsautomaten dar. Dort enthalten Ereignisse ausschließlich die Information ihres Auftretens; dies korrespondiert mit dem Ereignistypen *void*. Daher werden Ereignisse im Testmodell als Trigger und als Ausgabeereignisse von Transitionen parametrisch dargestellt. Beispielsweise wird ein Signal *time* vom Typ *DATE(h:int;min:int;s:int)* als Trigger *time(h,min,s)* dargestellt. Das Auftreten eines Signals mit dem Wert *12:00:00* wird dementsprechend mit *time(12,00,00)* dargestellt.

Die Parameter eines Ereignisses bestimmen häufig direkt die Werte der erweiterten Zustandsvariablen. Wächterbedingungen und Auswahlpunkte sind häufig an erweiterte Zustandsvariablen und die Werte abstrakter Signale geknüpft. Daher kann mittels Ereignisparametern der Kontrollfluss von ROOMcharts gesteuert werden.

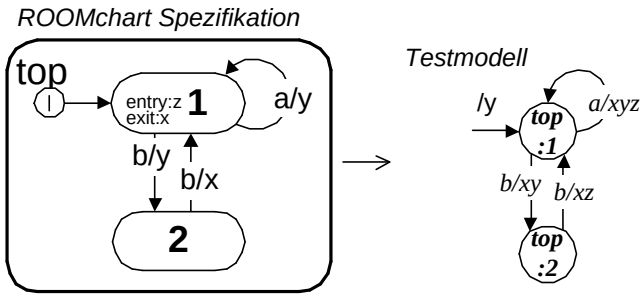


Abbildung 8.5: Eingangs- und Ausgangsaktionen

8.6 Eingangs- und Ausgangsaktionen

Der Formalismus von ROOMcharts erlaubt die Definition von Eingangs- und Ausgangsaktionen, die beim Betreten und Verlassen eines Zustands ausgeführt werden. Zur Ableitung eines Testmodells wird der Aktion jeder ausgehenden Transition eines Zustands die Ausgangsaktion vorangestellt. Der Aktion jeder eingehenden Transition eines Zustands wird die Eingangsaktion angehängt.

In Abbildung 8.5 ist beispielhaft die Behandlung von Eingangs- und Ausgangsaktionen dargestellt. Die initiale Transition führt im ROOMchart keine Aktion aus, implizit löst sie allerdings die Eingangsaktion von Zustand *1* aus. Die reflexive Transition an Zustand *1* führt die Ausgangsaktion x, die Aktion y und die Eingangsaktion z aus. Die Transition von Zustand *1* zu Zustand *2* führt die Ausgangsaktion x und die Aktion y aus. Die Transition von Zustand *2* zu Zustand *1* führt die Aktion x und die Eingangsaktion z aus.

8.7 Hierarchische Elemente von ROOMcharts

Im Gegensatz zu Zustandsdiagrammen in der *Unified Modeling Language* [114] besitzen ROOMcharts eine eindeutige Semantik für die Definition hierarchischer Automaten [148, 143, 64, 113]. Die Verwendung von hierarchischen Automaten während der Entwurfsphase ist ein wertvolles Hilfsmittel der Komplexitätsbeherrschung. Die Definition eines Testmodells, das mit einer Vielzahl automatenbasierter Verfahren kompatibel sein soll [18, 138, 37, 33], verlangt dagegen alle hierarchischen Elemente zu entfernen. In [14] wird der Prozess der Entfernung von Hierarchien eines Automaten, bzw. die Transformation eines hierarchischen Automaten zu einem flachen Automaten, als *Flattening* (dt. Plätten, Abflachen) bezeichnet.
Die Ableitung eines flachen Zustandsautomaten aus einem hierarchischen ROOMchart geschieht vereinfacht in folgenden drei Schritten:

1. Um Namenskonflikte zu vermeiden, müssen während des Entfernens der Hierarchieebenen eines Automaten eine Umbenennung der Zustände des Automaten erfolgen. Um die Umkehrbarkeit dieses Schrittes zu gewährleisten, werden alle Umbenennungen in einer Tabelle notiert.

2. Es werden alle geteilten Transitionen vereinigt, wobei die Aktionen der Teiltransitionen und optionale Eingangs- und Ausgangsaktionen in der gegebenen Reihenfolge gekettet werden.

3. Die hierarchischen Zustandsgrenzen und deren Anschlusspunkte werden entfernt.

Nach Ausführung der drei Schritte auf einem ROOMchart mit hierarchischen Zuständen entsteht ein flacher Zustandsautomat, der ausschließlich elementare Transitionen und Zustände enthält. Es ist zu beachten, dass die beschriebenen drei Schritte nur auf ein hierarchisches ROOMchart ohne Gedächtniszustände und Gruppentransitionen anwendbar ist. Die Behandlung dieser Entwurfselemente wird an anderer Stelle in dieser Arbeit ausführlich erläutert.

Das Beispiel in Abbildung 8.6 demonstriert die Ableitung eines flachen Zustandsautomaten aus einem hierarchischen ROOMchart. Das vorliegende ROOMchart hat zwei Hierarchieebenen, die

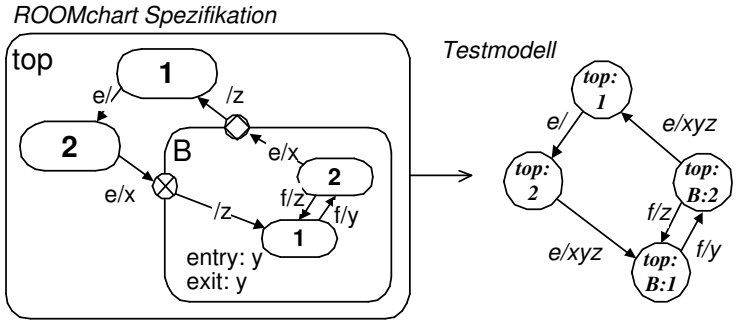


Abbildung 8.6: Hierarchische Zustände

durch die Zustände *top* und B definiert werden, wobei B in *top* enthalten ist. Die Bezeichnungen der vier nicht hierarchischen, elementaren Zustände würden ohne Umbenennung nach Entfernen des hierarchischen Zustands B Namenskonflikte verursachen. Daher werden die Zustände *1* und *2* in *top* zu *top:1* und *top:2* und die Zustände *1* und *2* in B zu *top:B:1* und *top:B:2* umbenannt. Im zweiten Schritt entstehen zwei Transitionen von *top:2* zu *top:B:1* und von *top:B:2* zu *top:1*.
Die Transition von *top:2* zu *top:B:1* hat den Trigger der Teiltransition von *2* in *top* zu dem Eingangs-Anschlusspunkt von B. Die Aktion der Transition von *top:2* zu *top:B:1* ergibt sich aus der Kettung der Aktion des ersten Transitionsteils, der Eingangsaktion und des zweiten Transitionsteils der im Zustand *1* in B endet.
Die Transition von *top:B:2* zu *top:1* hat den Trigger der Teiltransition von *2* in B zu dem Ausgangs-Anschlusspunkt von B. Die Aktion der Transition von *top:2* zu *top:B:1* ergibt sich aus der Kettung der Aktion des ersten Transitionsteils, der Ausgangsaktion und des zweiten Transitionsteils der im Zustand *1* in *top* endet.
Nachdem alle Namenskonflikte aufgelöst und alle geteilten Transitionen vereinigt worden sind, können die Grenzen der hierarchischen Zustände *top* und B in Schritt 3 entfernt werden. Als Resultat der drei durchgeführten Schritte ergibt sich das Testmodell auf der rechten Seite in Abbildung 8.6.

8.7.1 Gruppentransitionen

Eine Gruppentransition eines hierarchischen Zustand s_c kann von jedem Unterzustand von s_c ausgelöst werden und endet entweder in einer höheren oder in derselben Hierarchieebene. Eine Gruppentransition, die in s_c beginnt und endet, wird als reflexive Gruppentransition bezeichnet. Eine Gruppentransition, die s_c verlässt, wird als extern bezeichnet. Eine reflexive Gruppentransition, die s_c nicht verlässt, wird als intern bezeichnet.

Wenn ein hierarchischer Zustand aufgelöst wird, muss jede Gruppentransition durch jeweils eine Transition aus jedem Unterzustand ersetzt werden. Die Ersatztransitionen besitzen den gleichen Eingangszustand, den gleichen Trigger und die gleichen Aktionen. Als Ausnahmen werden überschriebene Gruppentransitionen sowie die unterschiedliche Behandlung von internen und externen Gruppentransitionen betrachtet.

Eine externe Gruppentransition t_g von s_c zu einem externen Zustand s_{extern} wird durch jeweils eine Transition t_{ersatz} aus jedem Unterzustand s_{intern} von s_c zu s_{extern} ersetzt. Diese Ersatztransitionen besitzen den Trigger von t_g und die gekettete Aktion aus der Ausgangsaktion von s_c und der Aktion von t_g. Wenn die Ausgangsaktion mit a_{exit} und die Aktion der Gruppentransition mit $a_{transition}$ bezeichnet wird, ergibt sich die Aktion der Ersatztransition zu $a_{exit} \bullet a_{transition}$.

In dem Beispiel in Abbildung 8.7 wird die Behandlung einer Gruppentransition bei der Entfernung der hierarchischen Zustände aus einem ROOMchart demonstriert. Von dem hierarchischen Zustand B ist eine externe Gruppentransition zu Zustand *1* in B definiert. In dem abgeleiteten Testmodell wird die externe Gruppentransition durch zwei Transitionen ersetzt. Eine Ersatztransition beginnt in Zustand *top:B:1*, endet in Zustand *top:1*, besitzt den Trigger e der Gruppentransition und führt erst die Ausgangsaktion von B und dann die Aktion der Gruppentransition aus. Die zweite Ersatztransition ist analog aus dem Zustand *top:B:2* definiert.

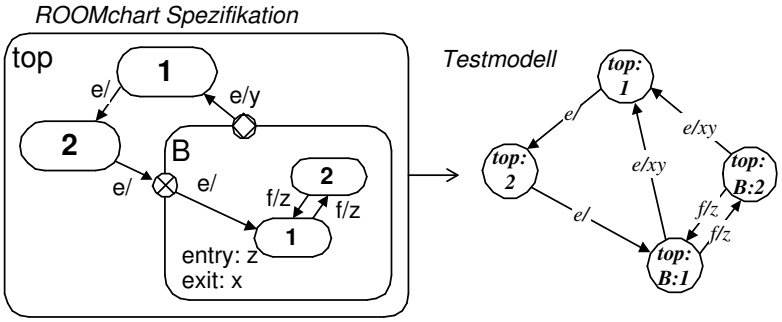


Abbildung 8.7: Externe Gruppentransition

Die Behandlung von Gruppentransitionen muss danach differenziert durchgeführt werden, ob es sich um eine interne oder externe Gruppentransition handelt. Im Gegensatz zu externen Gruppentransitionen lösen interne Gruppentransitionen die Eingangs- bzw. Ausgangsaktionen des hierarchischen Zustands nicht aus. Eine interne Gruppentransition in s_c wird durch jeweils eine Transition von jedem Unterzustand von s_c zu dem Eingangszustand der Gruppentransition ersetzt, die den gleichen Trigger und die gleiche Aktion besitzen.

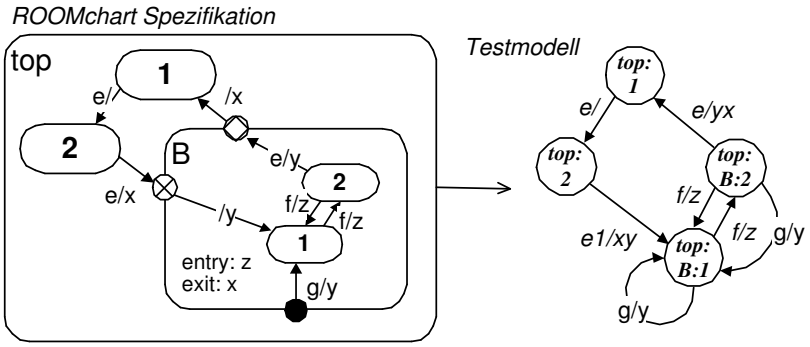


Abbildung 8.8: Interne Gruppentransition

Das Beispiel in Abbildung 8.8 zeigt die Behandlung einer internen Gruppentransition bei der Entfernung hierarchischer Zustände aus einem ROOMchart. Der hierarchische Zustand B enthält eine interne Gruppentransition von B in den Zustand *1* mit dem Trigger g und der Aktion y. Die interne Gruppentransition wird durch eine reflexive Transition an *top:B:1* und eine Transition von *top:B:2* zu *top:B:1* ersetzt, die beide durch g ausgelöst werden und die Aktion y ausführen.

Die Behandlung reflexiver, externer Gruppentransitionen verlangt die Beachtung von Ein- und Ausgangsaktionen des hierarchischen Zustands s_c. Jede externe, reflexive Gruppentransition t eines hierarchischen Zustands s_c wird durch jeweils eine reflexive Transition an jedem Unterzustand von s_c ersetzt, die den gleichen Eingangszustand, den gleichen Trigger und die Ausgangsaktion a_{exit} von s_c, die Aktion $a_{transition}$ von t und die Eingangsaktion a_{entry} von s_c, also die Gesamtaktion $a_{exit} \bullet a_{transition} \bullet a_{entry}$ besitzt .

Das Beispiel in Abbildung 8.9 stellt die Behandlung externer, reflexiver Gruppentransitionen dar. Die externe, reflexive Gruppentransition am hierarchischen Zustand B mit dem Trigger g und der Aktion y wird durch reflexive Transitionen an den Zuständen *top:B:1* und *top:B:2* ersetzt. Die Aktionen der Ersatztransitionen beinhalten die Eingangs- und Ausgangsaktionen von B. Die Eingangsaktion von B ist z, die Ausgangsaktion von B ist x, wodurch die Ersatztransitionen die Aktion xyz erhalten.

Die Behandlung von reflexiven Gruppentransitionen muss ebenfalls nach der betreffenden Hierarchieebene differenziert durchgeführt werden. Im Gegensatz zu externen, reflexiven Gruppentransitionen lösen interne, reflexive Gruppentransitionen die Eingangs- und Ausgangsaktionen des

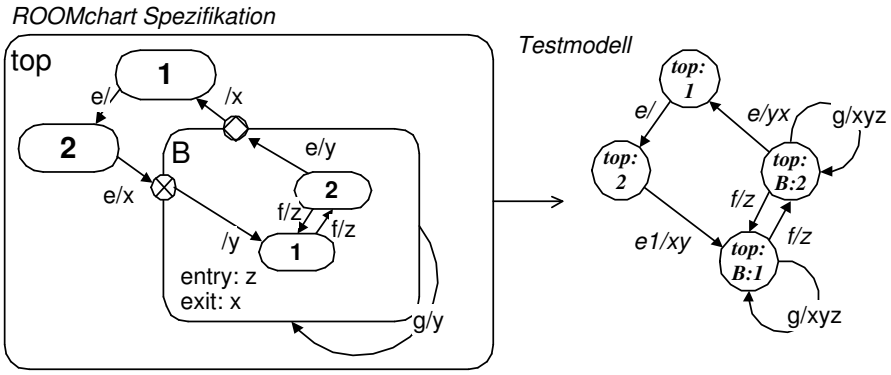


Abbildung 8.9: Externe, reflexive Gruppentransition

hierarchischen Zustands nicht aus und werden daher durch reflexive Transitionen, mit Trigger und Aktion der Gruppentransition, an jedem Unterzustand ersetzt.

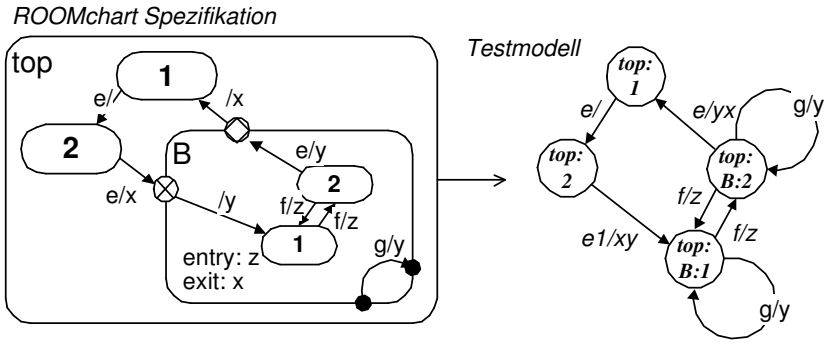


Abbildung 8.10: Interne, reflexive Gruppentransition

Das Beispiel in Abbildung 8.10 zeigt die Behandlung von internen reflexiven Gruppentransitionen bei der Entfernung hierarchischer Zustände aus einem ROOMchart. Die interne, reflexive Gruppentransition im Zustand B wird durch g ausgelöst und produziert y. In dem flachen Zustandsautomaten des Testmodells wird die interne, reflexive Gruppentransition durch jeweils eine reflexive Transition an jedem Unterzustand von B ersetzt. Die Ersatztransitionen besitzen ebenfalls den Trigger g und produzieren y.

Die Regeln zur Auslösung von Transitionen in *Real-Time Object-Oriented Modeling* bevorzugen innere vor äußeren Transitionen. Daher werden nur dann Ersatztransitionen in das Testmodell eingefügt, wenn noch keine ausgehende Transition mit dem auslösenden Ereignis der ersetzten Gruppentransition am betreffenden Zustand vorhanden ist. Da Gruppentransitionen eines hierarchischen Zustands s_c immer auf einer höheren Ebene definiert sind, als die in s_c enthaltenen Transitionen, werden letztere immer bevorzugt ausgelöst. Wenn eine Transition t_u einer anderen Transition t_o vorgezogen wird, so wird t_o durch t_u überschrieben.

In dem Beispiel in Abbildung 8.11 wird die Behandlung einer internen Gruppentransition mit einer überschreibenden Transition dargestellt. Die Transition vom Zustand *1* zum Zustand *2* besitzt den gleichen Trigger wie die Gruppentransition. Im abgeleiteten Testmodell findet sich daher nur eine reflexive Ersatztransition am Zustand *top:1*.

8.7.2 Gedächtniszustände

Das Gedächtnis eines hierarchischen Zustands s_c speichert temporär einen Zustand, um die internen Prozesse von s_c während der Ausführung externer Verarbeitung zu schützen. Das Gedächtnis

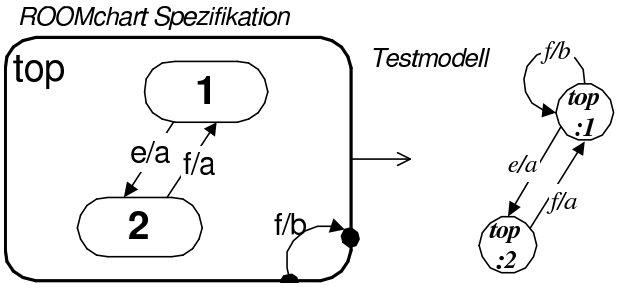


Abbildung 8.11: Überschriebene Gruppentransition

eines Zustands ist für jedes Zeitintervall zwischen dem Verlassen und dem nächsten Betreten von s_c definiert. Während eines solchen Zeitintervalls wird der direkt vor dem Verlassen von s_c eingenommene Unterzustand gespeichert, um beim nächsten Betreten von s_c durch einen Gedächtnispunkt (*engl. History-Connector*) wieder eingenommen zu werden. Ein Gedächtnis-Anschlusspunkt von s_c besitzt eine eingehende Transition, die Gedächtnistransition (*engl. History-Transition*), und keine ausgehenden Transitionen. Als Gedächtniszustand (*engl. History-State*) wird ein Register von s_c bezeichnet, das den Zustand direkt vor dem Verlassen von s_c speichert. Synonym kann auch der gespeicherte Zustand als Gedächtniszustand bezeichnet werden. Wenn s_c noch nicht betreten wurde, ist der initiale Zustand im Gedächtniszustand gespeichert.

In den folgenden Beispielen wird vereinfachend auf die Darstellung von Eingangs- und Ausgangsaktionen der hierarchischen Zustände verzichtet, da deren Behandlung bereits mehrfach demonstriert wurde.

Eine Transition t von einem Zustand s_{extern} zu einem hierarchischen Zustand s_c, wobei s_{extern} kein Unterzustand von s_c ist, kann zu einem beliebigen Unterzustand s_{intern} von s_c führen. Wenn explizit kein Unterzustand von s_c als Folgezustand von t definiert ist und t also in einem Anschlusspunkt von s_c endet, wird der Gedächtnis-Zustand zum Folgezustand von t. Wenn s_c noch nicht betreten wurde, ist der initiale Zustand der Folgezustand.

Die Ableitung eines Testmodells aus einem ROOMchart mit Gedächtniszuständen führt zu einem theoretisch nichtdeterministischen, endlichen Zustandsautomaten. Das Testmodell muss um Informationen über das Verhalten infolge der Gedächtnisfunktionen erweitert werden. Der Nichtdeterminismus manifestiert sich in dem Testmodell durch Ersatztransitionen mit gleichem Trigger.

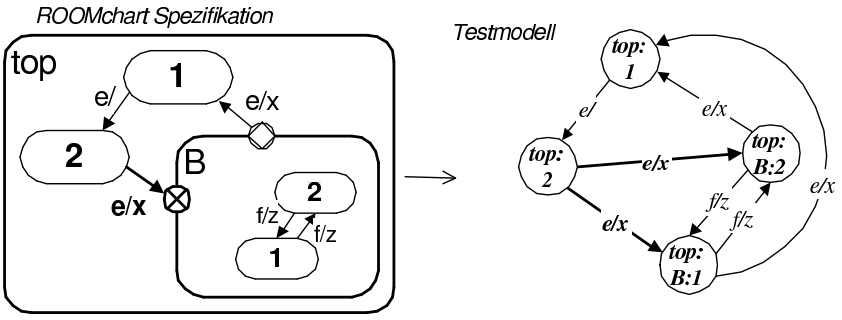


Abbildung 8.12: Endlicher nichtdeterministischer Zustandsautomat

In Abbildung 8.12 ist ein beispielhaftes ROOMchart mit einem Gedächtniszustand B gegeben. Die Transition von Zustand *2* in *top* zu Zustand B ist eine Gedächtnistransition.

Es ist möglich, dass nicht von jedem Unterzustand ausgehend ein hierarchischer Zustand s_c verlassen werden kann. Dies ist auch bei Verwendung von externen Gruppentransitionen der Fall, wenn von dem betreffenden Unterzustand eine ausgehende Transition definiert ist, welche die Gruppentransition überschreibt. In diesem Fall darf keine Ersatztransition in den Unterzustand im Testmodell eingefügt werden.

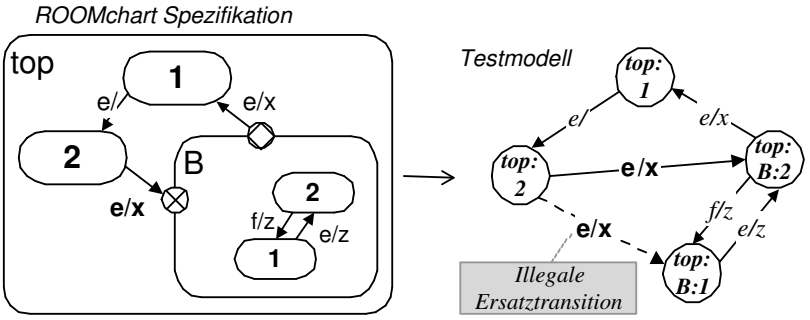


Abbildung 8.13: Überschriebene Gedächtnistransition

Das Beispiel eines ROOMcharts in Abbildung 8.13 demonstriert die Behandlung eines hierarchischen Zustands mit Gedächtnisfunktion und einer überschriebenen Gruppentransition. Die Transition von Zustand *1* in B zu Zustand *2* in B überschreibt die Gruppentransition von B zum Zustand *1* in *top*. Da vom Zustand *1* in B der hierarchische Zustand B nicht direkt verlassen werden kann, wird keine Ersatztransition für die Gedächtnistransition von Zustand *top:2* zu Zustand *top:B:1* definiert. In dem Beispiel ist die illegale Ersatztransition gestrichelt dargestellt.

Die Ableitung eines Testmodells aus einem ROOMchart mit Gedächtnisfunktion liefert einen nichtdeterministischen, endlichen Zustandsautomaten. Da der Großteil automatenbasierter Testverfahren nur auf deterministischen Automaten anwendbar ist, beispielsweise die Verfahren präsentiert in [18, 138, 33, 37], muss ein weiterer Schritt zur Ableitung eines deterministischen Testmodells ausgeführt werden.

Ein deterministisches Testmodell kann aus einem ROOMchart mit Gedächtniszuständen abgeleitet werden, indem jede Gedächtnistransition t_H durch Transitionen mit Wächterbedingung ersetzt wird. Jede Ersatztransition t_{ersatz} führt vom Ausgangszustand von t_H zu jeweils einem Unterzustand des betreffenden hierarchischen Zustands s_c. Jede Ersatztransition besitzt den Trigger von t_H und führt die Aktion von t_H und die Eingangsaktion von s_c aus. Eine Anzahl Pfade, die in dem Testmodell eines ROOMcharts mit Gedächtnisfunktionen vorhanden sind, ist nicht ausführbar. Ein Pfad der eine Gedächtnisfunktion beinhaltet, muss daher bestimmte Bedingungen erfüllen.

Gedächtniszyklen Ein ausführbarer Pfad, der eine Gedächtnistransition t_H beinhaltet, muss in dem gleichen Unterzustand s_{intern} von s_c beginnen und enden und wird daher Gedächtniszyklus genannt.

Der Zyklus darf zwischen Verlassen von s_c, direkt aus s_{intern}, und dem Erreichen von t_H nicht durch s_c führen. Im Falle des ersten Betretens von s_c beginnt der Pfad zu t_H im Initialzustand. Dieser Fall bildet eine Ausnahme von der beschriebenen Bedingung und wird durch eine Disjunktion berücksichtigt. Jede Ersatztransition t_{ersatz} muss um eine Wächterbedingung erweitert werden, die den Gedächtniszyklus bezüglich dem Endzustand von t_{ersatz} beschreibt.

Im folgenden Absatz wird eine aussagelogische Bedingung aus der Beschreibung von Gedächtniszyklen hergeleitet.

Ein Gedächtniszyklus ist ein Pfad, der in einem ROOMchart eine Gedächtnistransition t_H eines hierarchischen Zustands s_c beinhaltet. Mit Ausnahme der ersten Aktivierung schließt t_H einen Zyklus im ROOMchart ab. Die Wächterbedingung einer Ersatztransitionen t_{ersatz} für t_H mit dem Ausgangszustand s_{extern} muss wahr werden, wenn (a) der Zielzustand s_{intern} ein Vorgängerzustand im Pfad nach s_{extern} ist, (b) s_c direkt aus s_{intern} verlassen wurde und (c) s_c erst durch t_H wieder betreten wird.

Damit kann für ein ROOMchart bzw. das abgeleitete Testmodell eine Funktion der Wächterbedingungen $W^{ersatz} : T \times P \rightarrow \{W, F\}$, mit den Mengen der Transition T und Pfade P, der Ersatztransitionen für das Betreten von hierarchischen Zuständen durch Gedächtnistransitionen definiert werden:

Allgemeine Wächterbedingung Ersatztransition

$$
\begin{aligned}
W^{ersatz}(t_{ersatz}, p) \quad = \quad & p \in P : \langle t_0, t_1, \ldots t_n, t_{ersatz} \rangle \wedge \\
& \bigwedge_{i=1}^{n} \; t_i \in (T_{super} - T_{sub}) \wedge \\
& p \in P : \langle t_{ersatz}, t_0 \rangle \\
P \quad = \quad & \textit{Menge aller Pfade} \\
& \textit{eines ROOMtest Testmodells} \\
T_{super} \quad = \quad & \textit{Menge der Transitionen, enthalten im Oberzustand} \\
& \textit{eines hierarchischen Zustands } s_c \\
T_{sub} \quad = \quad & \textit{Menge der Transitionen, enthalten in} \\
& \textit{einem hierarchischen Zustand } s_c
\end{aligned}
$$

Falls ein hierarchischer Zustand mit Gedächtnisfunktion s_c einen oder mehrere hierarchische Zustände enthält, kann eine Gedächtnistransition mehrere Unterzustände von s_c als betreffen. Dieser Konflikt wird aufgelöst, indem immer der hierarchisch tiefste der möglichen Folgezustände gewählt wird. Diese Art der Behandlung von hierarchisch geschachtelten Zuständen mit Gedächtnistransition wird tiefe Gedächtnisfunktion (*engl. Deep History*) genannt. Die Ableitung von Testmodellen mit Gedächtnisfunktion sollte aus diesem Grund Hierarchien von innen nach außen auflösen.

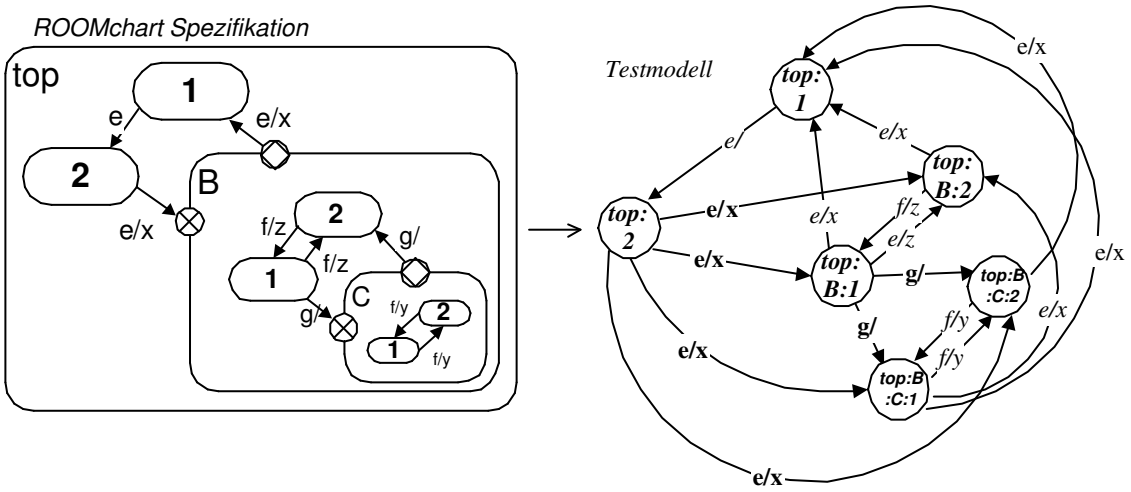


Abbildung 8.14: Tiefe Gedächtnisfunktion

Das Beispiel eines ROOMcharts in Abbildung 8.14 demonstriert die Ableitung eines endlichen Zustandsautomaten mit hierarchisch geschachtelten Zuständen mit Gedächtnisfunktion. Vor Behandlung der Gedächtnistransition vom Zustand *2* in *top* zum Zustand *B* wird der hierarchische Zustand *C* in *B* entfernt. Dadurch ergeben sich die vier elementaren Zustände *top:B:1*, *top:B:2*, *top:B:C:*1 und *top:B:C:2* in *B*, die als mögliche Folgezustände der Gedächtnistransition dienen. Diese Vorgehensweise wird in dieser Arbeit für die Auflösung von Hierarchien in ROOMcharts festgelegt und entspricht einer *Inside-Out*-Strategie.

Nach der Definition in [143] kann optional ein initialer Zustand für jeden hierarchischen Zustand s_c definiert werden. Für den Fall des ersten Betretens bildet der initiale Zustand den Folgezustand der Gedächtnistransition. Wenn kein initialer Zustand in s_c definiert ist, soll die interne Hierarchieebene von s_c nicht betreten werden. Dieses Verhalten erscheint wenig intuitiv und würde in der Anwendung eine mögliche Fehlerquelle bedeuten, wenn beispielsweise irrtümlich die Definition eines initialen Zustands unterlassen wird. Daher wird hier für die Anwendung von ROOMcharts die Definition eines initialen Zustands für jeden hierarchischen Zustand gefordert.

Die Behandlung des initialen Zustands bei der Verwendung der Gedächtnisfunktion eines hierarchischen Zustands wird nicht durch die Definition von Gedächtniszyklen abgedeckt. Daher muss bei der Testfallgenerierung das erstmalige Betreten des hierarchischen Zustands gesondert behandelt werden.

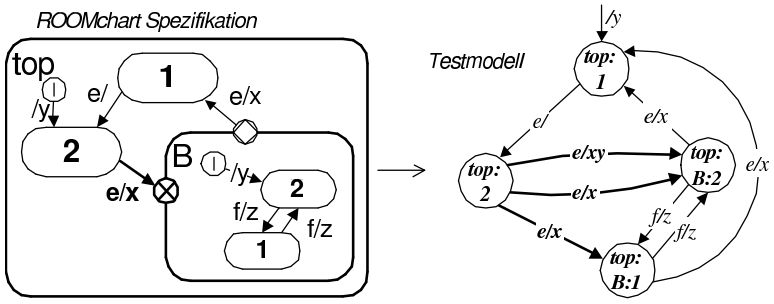


Abbildung 8.15: Gedächtniszustand mit Initialpunkt

Das Beispiel eines ROOMcharts mit Gedächtniszustand in Abbildung 8.15 demonstriert die Ableitung eines nichtdeterministischen Zustandsautomaten. Neben den Ersatztransitionen nach Definition der Gedächtniszyklen muss eine weitere Ersatztransition dem Modell hinzugefügt werden, welche die Aktion der initialen Transition beinhaltet. Im Beispiel wird eine weitere Transition von *top:2* zu *top:B:2* eingefügt, welche die Aktion *xy* ausführt, die sich durch Kettung der Aktion der Gedächtnistransition und der Aktion der initialen Transition ergibt.

Damit kann für ein ROOMchart bzw. das abgeleitete Testmodell eine Funktion der Wächterbedingungen $W^{initial} : T \times P \rightarrow \{W, F\}$ der Ersatztransitionen für das erstmalige Betreten von hierarchischen Zuständen durch Gedächtnistransitionen definiert werden:

Wächterbedingung initialer Ersatztransition

$$W^{initial}(t_{ersatz}, p) \quad = \quad p \in P : \langle t^{top}_{initial}, t_1, \ldots t_n, t_{ersatz}, t^{s_c}_{initial} \rangle \wedge \bigwedge_{i=1}^{n} (t_i \in (T_{super} - T_{sub}))$$

P		*Menge der ausführbaren Pfade eines ROOMtest-Testmodells*
T_{super}	=	*Menge der Transitionen des Superzustands eines hierarchischen Zustands s_c*
T_{sub}	=	*Menge der Transitionen, enthalten in einem hierarchischen Zustand s_c*
T	=	Menge der Transitionen *eines ROOMtest-Testmodells*
$t^{initial}_{top}$	=	*Initiale Transition des obersten Zustands top*
$t^{initial}_{s_c}$	=	*Initiale Transition des hierarchischen Zustands s_c*

Jede Ersatztransition für das erste Betreten eines hierarchischen Zustands durch eine Gedächtnistransition wird mit der beschriebenen Wächterbedingung ausgestattet. Die Wächterbedingung prüft, ob die Ersatztransition in einem Pfad von der initialen Transition des obersten Zustands *top* zu der initialen Transition des hierarchischen Zustands benutzt wird.

Die in der Wächterbedingung geprüften Eigenschaften beziehen sich nicht ausschließlich auf die Struktur des Testmodells, können aber aus diesem extrahiert werden, da die Namen der Zustände auf die Hierarchien des ursprünglichen ROOMcharts verweisen.

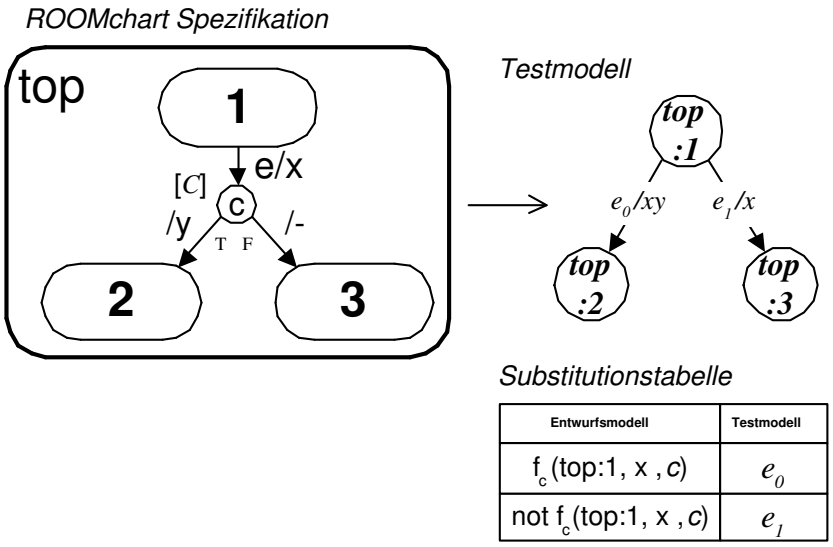


Abbildung 8.16: Binärer Auswahlpunkt

8.8 Auswahlpunkte

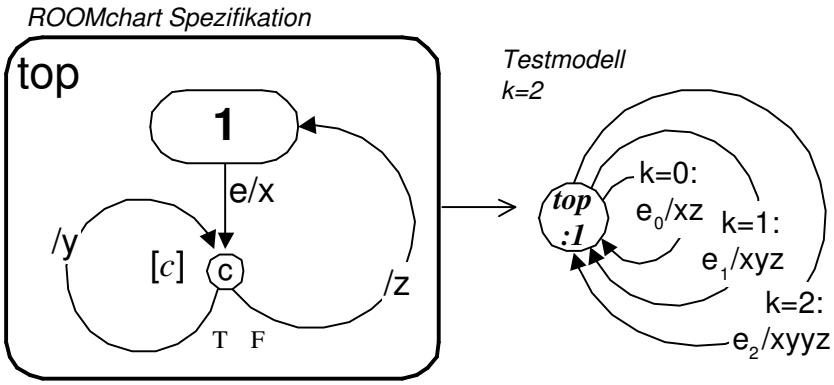
Neben der Verwendung von Wächterbedingungen stehen in ROOMcharts auch Auswahlpunkte für die Modellierung bedingter Kontrollstrukturen zur Verfügung. An anderer Stelle in dieser Arbeit wurden bereits verschiedene Verwendungsmöglichkeiten von Auswahlpunkten vorgestellt. In diesem Abschnitt werden diese Vorschläge erneut aufgegriffen und deren Behandlung während der Ableitung eines endlichen, flachen Zustandsautomaten aus ROOMcharts erläutert.

Für die Auswertung von Bedingungen in Auswahlpunkten wird eine boolesche Funktion $f_c : S \times A \times C \rightarrow \{T, F\}$, mit der Menge Zustände S, der Menge Aktionen A und der Menge Bedingungen C eingeführt. Die Funktion $f_c(s, a, c) \rightarrow \{T, F\}$ evaluiert eine Bedingung c nach Ausführung einer Aktion a ausgehend vom Zustand s. Vor dem Verlassen des Zustands und vor dem Erreichen des Auswahlpunktes kann eine Aktion ausgeführt werden. Dabei können die Werte der Entscheidungsvariablen verändert werden. Daher wird im Testmodell die Bedingung eines Auswahlpunktes mittels der Funktion f_c angegeben.

Die Ableitung eines Testmodells mit der Semantik eines endlichen Zustandsautomaten aus einem ROOMchart mit Auswahlpunkten verlangt die Entfernung des Auswahlpunktes und der Bedingungen. Die Behandlung bedingter Transitionen wurde bereits in einem vorangegangenen Kapitel erläutert. Ein binärer Auswahlpunkt verzweigt den Kontrollfluss auf zwei Pfade für die Evaluation der Bedingung zu *Wahr* und *Falsch*. In einem Testmodell wird jeder binäre Auswahlpunkt durch zwei Transitionen ersetzt.

Das Beispiel in Abbildung 8.16 stellt die Behandlung eines Auswahlpunktes bei der Ableitung eines Testmodells von einem ROOMchart dar. Eine Transition mit dem Trigger e und der Aktion x führt zu einem Auswahlpunkt mit der Bedingung c. Wenn c gilt, wird die Aktion xy produziert, und sonst wird die Aktion x produziert. In dem Testmodell wird jeder Pfad, der den Auswahlpunkt durchläuft durch eine Transition ersetzt. Da jede der Ersatztransitionen einem Fall der Bedingung c entspricht, sind unterschiedliche Trigger für die Ersatztransition notwendig. In der Substitutionstabelle in Abbildung 8.16 ist das Ereignis e durch e_0 ersetzt, unter der Bedingung, dass c nach Verlassen von *top:1* und nach Ausführen von x, ausgedrückt durch $f_c(top : 1, x, c)$, gilt. Das Ereignis e_1 ersetzt den Fall, dass die Bedingung c nach Verlassen von *top:1* und nach Ausführen von x gilt.

Durch Verwendung reflexiver Transitionen kann ein Auswahlpunkt auch zur Konstruktion von Schleifen genutzt werden. Ein Auswahlpunkt zur Darstellung einer Schleife stellt, ebenso wie eine Schleife, potentiell unendlich viele Pfade dar. Für eine vollständige Abbildung im Testmodell müss-



Substitutionstabelle

Entwurfsmodell	Testmodell
e[fc(top:1 , x ,c)]	e_0
e[f_c(top:1 , x ,c) ^ not f_c(top:1 , xy ,c)]	e_1
e[f_c(top:1 , x ,c) ^ f_c(top:1 , xy ,c) ^ not f_c(top:1 , xyy ,c)]	e_2

Abbildung 8.17: Auswahlpunkt mit Schleifensemantik

ten ebenso viele, und damit auch potentiell unendlich viele, Ersatztransitionen für einen solchen Auswahlpunkt vorhanden sein. Da dies offensichtlich nicht praktikabel ist, wird ein Testmodell nach dem Prinzip des strukturierten Pfadtest mit k=2 erzeugt. Es ist allerdings möglich die dargestellte Vorgehensweise auf beliebige k auszuweiten. Im Gegensatz zu der üblichen Anwendung der strukturierten Pfadtestverfahren auf Programmtext, kann hier das Verfahren nicht direkt auf die Implementation angewendet werden, sondern es wird ein Testmodell mit den geforderten Pfaden erzeugt.

Das Beispiel in Abbildung 8.17 zeigt ein ROOMchart mit einem Auswahlpunkt, der einer abweisenden Schleife entspricht. Solange die Bedingung c hält, wird die reflexive Transition am Auswahlpunkt durchlaufen. Im Testmodell mit dem Pfadparameter k=2 ersetzen drei Transition für k=0, k=1 und k=2 den Auswahlpunkt und seine Transitionen. Die Transition für den Fall k=0 stellt die Abweisung an der Schleife dar, ohne dass die reflexive Transition durchlaufen wird. Es wird daher auf das Ereignis e die Aktion xz ausgeführt. Die Bedingung für den Fall k=0 wird mit $not\ f_c(top:1, x, c)$ evaluiert, so dass c nach Verlassen von *top:1* und nach Ausführen von x nicht gilt. Die Transition für den Fall k=1 stellt die einmalige Ausführung des Schleifenrumpfes, hier die reflexive Transition, dar. Nach dem Auslösen durch e wird daher die Aktion xyz ausgeführt und die Bedingung zu $f_c(top:1, x, c) \wedge not\ f(top:1, xy, c)$ evaluiert, so dass also nach Verlassen von *top:1* die Schleifenbedingung vor Ausführung der reflexiven Transition gilt, aber nicht danach. Die Transition für den Fall *k=2* stellt die zweimalige Ausführung des Schleifenrumpfes dar. Nach dem Auslösen durch e wird die Aktion $xyyz$ ausgeführt und die Bedingung c zu $f_c(top:1, x, c) \wedge f(top:1, xy, c) \wedge not\ f(top:1, xyy, c)$ evaluiert, so dass die Bedingung c nach einem Schleifendurchlauf noch gilt, aber nicht nach dem Zweiten.

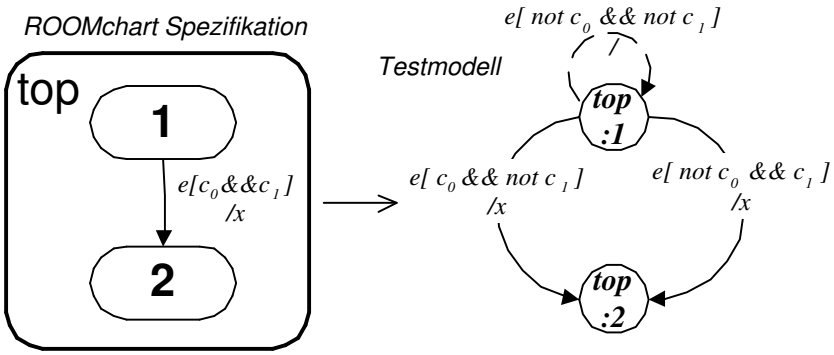


Abbildung 8.18: Wächterbedingung Bedingungsüberdeckung

8.9 Bedingungsüberdeckung

Während der Ableitung eines Testmodells aus einem ROOMchart muss bereits die mögliche Anwendung eines Bedingungsüberdeckungstestverfahrens berücksichtigt werden. Jedes dieser Verfahren fordert die explizite Evaluation der Teilentscheidungen einer Bedingung zu *Wahr* und *Falsch*. Dies führt im Testmodell zu zusätzlichen Konfigurationen des Testmodells. Jede Konfiguration enthält einen Teil der notwendigen Wertebelegungen für die Bedingungen. Alle Konfigurationen gemeinsam erfüllen das Bedingungsüberdeckungskriterium.

Ein Trigger kann eine beliebig geformte Wächterbedingung über Ereignisparameter und erweiterte Zustandsvariablen besitzen. Nach Huhn und Mücke wird für diesen Fall ein Testmodell abgeleitet, in dem reflexive Transition die Fälle abdecken, in denen die Wächterbedingung nicht erfüllt ist. In Abbildung 8.18 wird ein Beispiel für ein Testmodell für modifizierte Bedingungs--/Entscheidungsüberdeckung aufgeführt. Die gestrichelte reflexive Transition ist im Testmodell ausführbar, wenn keine Teilbedingung im Wächter gilt. Die Ersatztransition von *top:1* zu *top:2* sind ausführbar, wenn jeweils eine der Teilbedingungen gilt. Analog kann auch ein Modell für einfache Bedingungsüberdeckung konstruiert werden, dargestellt in [103].

Die in dieser Arbeit bevorzugten Normalform ROOMcharts können mithilfe von Auswahlpunkten konstruiert werden. Die Ableitung von Testmodellen für Bedingungsüberdeckung auf der Basis von ROOMcharts mit Auswahlpunkten und den in dieser Arbeit vorgestellten hierarchischen Pseudozustände, muss gesondert betrachtet werden. Ein binärer Auswahlpunkt besitzt im Gegensatz zu einer Wächterbedingung einen Zweig, der ausgelöst wird, wenn die Bedingung nicht erfüllt ist. Die Fälle, in denen die Bedingung nicht erfüllt ist, können auf das Testmodell daher ohne Erweiterung abgebildet werden, indem mehrere Konfigurationen des Testmodells gebildet werden.

Das Beispiel in Abbildung 8.19 präsentiert die Ableitung eines ROOMcharts mit einem Auswahlpunkt. Der Auswahlpunkt enthält die Bedingung *(a||b&&c||d)* mit dem ODER-Operator ||, dem UND-Operator && und den atomaren Teilentscheidungen *a*, *b*, *c* und *d*. Die Bedingung ist semantisch äquivalent mit der Bedingung im Abschnitt *Bedingungsüberdeckungsverfahren* des Kapitels *Testverfahren*. Dort kann eine ausführlichere Darstellung der Bedingungsüberdeckungsverfahren anhand dieser Bedingung gefunden werden. Das Komplement einer Entscheidung c wird im Folgenden durch *not c* dargestellt. Die einfache Bedingungsüberdeckung fordert die Evaluation jeder Teilentscheidung zu *Wahr* und *Falsch*, unabhängig von der Gesamtentscheidung. Ein grundsätzliches Kriterium ist allerdings die Transitionsüberdeckung. Daher muss auch bei der einfachen Bedingungsüberdeckung die Gesamtentscheidung eines binären Auswahlpunktes zu *Wahr* und zu *Falsch* evaluiert werden. In dem Beispiel in Abbildung 8.19 wird die Ableitung eines Testmodell nach der im Vorangegangenen beschriebenen Methode dargestellt. Der endliche Zustandsautomat besitzt zwei Transitionen, die durch die substituierten Ereignisse e_0 und e_1 ausgelöst werden. Die Ereignisse e_0 und e_1 werden je nach Bedingungsüberdeckungskriterium gegen Bedingungen ausgetauscht, die in der Substitutionstabelle aufgeführt sind. Für die einfache Bedingungsüberdeckung reicht eine Konfiguration des Testmodells aus, um das Kriterium zu erfüllen. Die Substitutionsta-

belle für einfache Bedingungsüberdeckung enthält die booleschen Funktionen für die Trigger der beiden Transitionen.

Für die Erfüllung der modifizierten Bedingungs-/Entscheidungsüberdeckung sind am Beispiel in Abbildung 8.19 vier Konfigurationen notwendig. Jede Konfiguration testet eine Teilbedingung, die in Klammern notiert ist. Im Fall der modifizierten Bedingungs-/Entscheidungsüberdeckung entstehen vier endliche Zustandsautomaten bzw. Testmodelle.

Für die Anzahl der notwendigen Testmodelle kann eine Obergrenze N_{max} angegeben werden.

Für die einfache Bedingungsüberdeckung liegt diese Obergrenze bei

$$N_{max,C_2} = \sum_{i=1}^{m} n_i,$$

mit der Anzahl Auswahlpunkte m und der Anzahl atomarer Teilentscheidungen n_i des Auswahlpunktes i.

Die Zeitkomplexität dieses Ansatzes liegt damit für einfache Bedingungsüberdeckung bei $O(n)$, in Abhängigkeit von der Gesamtzahl atomarer Teilentscheidungen n eines ROOMcharts.

Für die modifizierte Bedingungs-/Entscheidungsüberdeckung kann eine Obergrenze bei

$$N_{max,MCDC} = \sum_{i=1}^{m}(n_i + 1) \text{ angegeben werden.}$$

Die Zeitkomplexität dieses Ansatzes liegt für modifizierte Bedingungs-/Entscheidungsüberdeckung ebenfalls bei $O(n)$.

Der vorgestellte Ansatz für die Bildung von Testmodellen für Bedingungsüberdeckungsverfahren ist linear bezüglich der Anzahl von Teilentscheidungen einer Bedingung. Die Raumkomplexität wird von diesem Ansatz nicht beeinflusst.

8.10 Pfadüberdeckung

Die strukturierten Pfadüberdeckungsverfahren wurden bereits bei der Ableitung eines Testmodells mit zyklischen Auswahlpunkten genutzt. Eine weitergehende Anwendung dieser Verfahren auf Zustandsautomaten erscheint aus Komplexitätsgründen nicht zweckmäßig. Eine Schätzung der Komplexität kann über die Erreichbarkeit [115] des mit dem Automaten assoziierten schlichten Graphen $G := (V; R)$, mit $R \subseteq V \times V$ und mit der Knotenmenge V, vorgenommen werden. Die elementaren Zyklen eines Graphen mit n Knoten können maximal von der Länge n sein. Ein Zyklus der Länge 1 ist ein Element der Hauptdiagonalen der Relation R. Ein Zyklus der Länge i ist ein Element der Hauptdiagonalen R^i. Daher ist folgende Abschätzung der Anzahl Zyklen eines schlichten Graphen möglich:

$$Z_{max} = \sum_{i=1}^{n} i$$

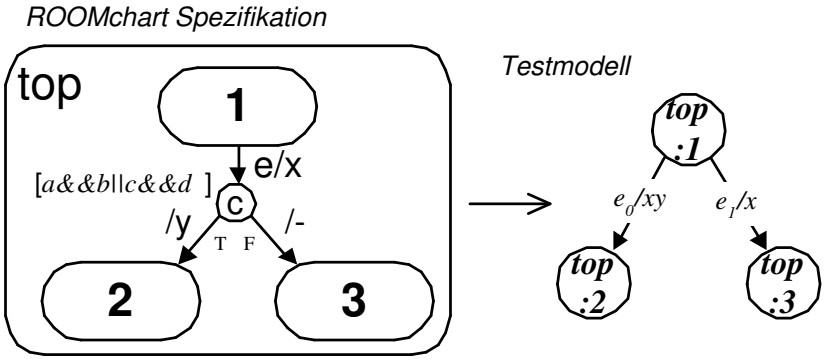
Die Schätzung ist bezüglich schlichten Graphen konservativ, da auch äquivalente Zyklen gezählt werden. Ein endlicher Zustandsautomat kann allerdings beliebig viele Kanten zwischen zwei Knoten besitzen und ist daher mit einem Multigraphen assoziiert. Unter dieser Annahme ist die maximale Anzahl Zyklen unendlich. Eine Anwendung von strukturierten Pfadüberdeckungsverfahren auf die Semantik von endlichen Zustandsautomaten erscheint angesichts dieser Abschätzung nicht zweckmäßig.

8.11 Vererbung

In *Real-Time Object-Oriented Modeling* sind System und Komponenten Klassen. Das Verhalten eines *Real-Time Object-Oriented Modeling-Systems* ist in Aktorklassen gekapselt. Um Änderungen an der Struktur oder dem Verhalten vorzunehmen, können Aktorklassen abgeleitet werden. Hier soll nur die Vererbung von Verhalten und dessen Testbarkeit behandelt werden.

Das Verhalten einer Aktorklasse wird durch folgende Attribute bestimmt:

- Funktionen
- Erweiterte Zustandsvariablen



Substitutionstabelle für einfache Bedingungsüberdeckung

Konfiguration	Entwurfsmodell	Testmodell
I	f_c(top:1,x, *a&&b||c&&d*)	e_0
	f_c(top:1,x, *not a&&not b||not c&&not d*)	e_1

Substitutionstabelle für modifizierte Bedingungs-/Entscheidungsüberdeckung

Konfiguration	Entwurfsmodell	Testmodell
I (b)	f_c(top:1,x,*a&&b||c&&not d*)	e_0
	f_c(top:1,x,*a&&not b||c&&not d*)	e_1
II (d)	f_c(top:1,x,*a&&not b||c&&d*)	e_0
	f_c(top:1,x,*a&&not b||c&&not d*)	e_1
III (a)	f_c(top:1,x,*a&&b||c&&not d*)	e_0
	f_c(top:1,x,*not a&&b||c&&not d*)	e_1
IV (c)	f_c(top:1,x,*a&&not b||c&&d*)	e_0
	f_c(top:1,x,*a&&not b||not c &&not d*)	e_1

Abbildung 8.19: Auswahlpunkt Bedingungsüberdeckung

- Zustände
- Transitionssegmente
- Zustandsdekompositionen
- Eingangs- und Ausgangsaktionen
- Transitionstrigger
- Transitionsaktionen
- Zweigprädikate von Auswahlpunkten

Jedes dieser Attribute kann in einer abgeleiteten Aktorklasse verändert, ergänzt oder verdeckt werden. Oftmals wird Vererbung angewendet, wenn das Verhalten zu einem frühen Zeitpunkt noch nicht vollständig bekannt ist. Durch die Verwendung von abstrakten Aktorklassen kann das Verhalten von Aktoren unvollständig spezifiziert werden. Es entsteht ein endlicher Zustandsautomat mit abstrakten Zuständen und abstrakten Transitionen. Abstrakte Zustände sind nicht hierarchisch und lassen eine spätere Verfeinerung zu, die durch Vererbung erfolgt. Abstrakte Transitionen enthalten

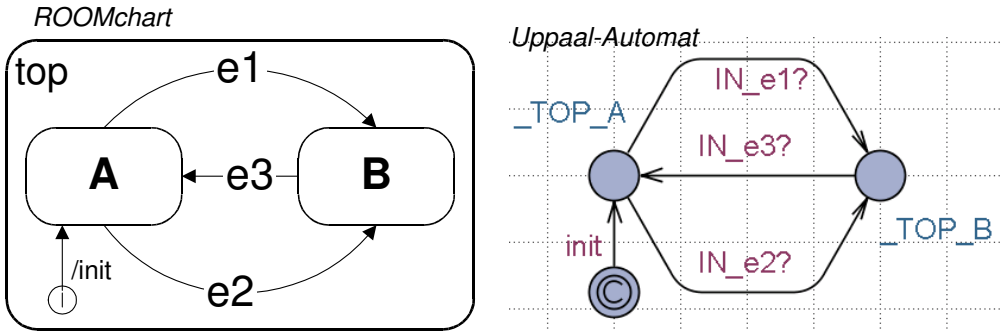


Abbildung 8.20: Einfacher beispielhafter Uppaal-Automat

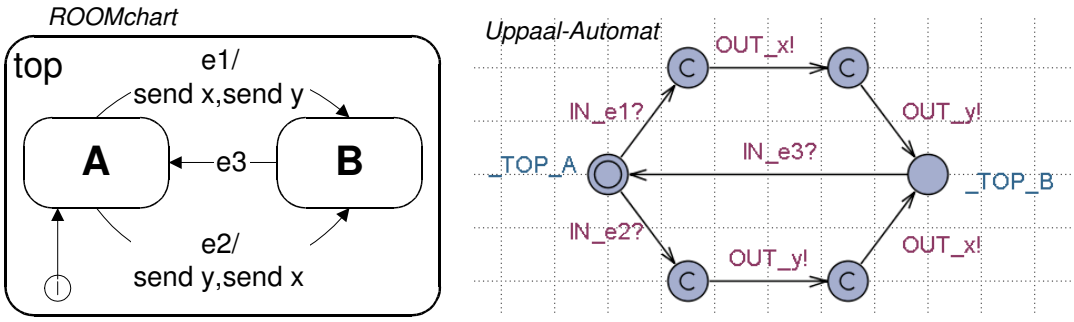


Abbildung 8.21: Komplexe Transitionen in Uppaal

abstrakte Spezifikationen der auslösenden Ereignisse, keinen Trigger und keine oder unvollständige Aktionen. Eine abstrakte Spezifikation des Triggers kann in späteren Verfeinerungsschritten konkretisiert werden.

Um auch in frühen Phasen den Test von abstrakten Aktorklassen zu ermöglichen, sollten diese immer lauffähig realisiert werden. Sollte keine lauffähige Aktorklasse vorhanden sein, können erst nach weiteren Entwicklungsschritten Tests durchgeführt werden. Lauffähige, abstrakte Aktorklassen können früh getestet werden und bieten daher eine erhöhte Sicherheit in Bezug auf das Kernverhalten der abgeleiteten Aktorklassen. Im modellbasierten Test kann bei einem plattformunabhängigen Modell auf einen detaillierten Entwurf der Aktoren verzichtet werden. Die Verwendung von Bibliotheken mit abstrakten Aktorklassen erscheint hier vorteilhaft. Während der Transformation in ein plattformspezifisches Modell werden die abstrakten Aktorklassen spezialisiert.

Analog zur Vererbung in objektorientierten Programmiersprachen sind auch in *Real-Time Object-Oriented Modeling* nur die veränderten Merkmale einem erweiterten Test zu unterziehen [120]. Ein Teil der vorhandenen Testfälle einer Aktorklasse kann wahrscheinlich unverändert für den Test abgeleiteter Subaktoren dienen.

In einem modellbasierten Softwareentwicklungsprozess mit lauffähigen abstrakten Aktorklassen in frühen Entwicklungsphasen kann für jede Entwicklungsstufe ein Testmodell abgeleitet werden. Nach einem Verfeinerungsschritt müssen diese Testfälle auch auf dem neuen Aktor in einem Regressionstest ausgeführt werden. Zusätzliche Testfälle prüfen das verfeinerte Verhalten der abgeleiteten Aktorklasse. Die Generierung von Testfällen auf Basis von Aktorklassen mit geerbtem Verhalten unterscheidet sich nicht von der Vorgehensweise bei flacher Vererbungshierarchie. Durch diese Vorgehensweise ist bereits zum frühestmöglichen Zeitpunkt ein Test, in der Simulation, des Systems möglich. Daher können Fehler bereits vor der Vollendung der Implementation gefunden werden. Dies kann zu Kostenersparnissen gegenüber herkömmlichen Entwicklungsmethoden führen, da die Fehlerbeseitigungskosten in den frühen Phasen deutlich niedriger ausfallen als in späteren Phasen.

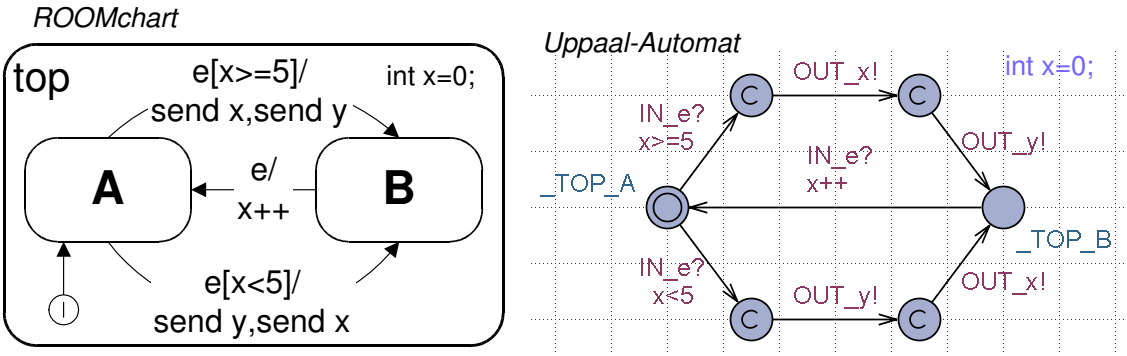


Abbildung 8.22: Komplexe Transitionen in Uppaal

8.12 Testmodellgenerierung für Uppaal

Die Konstruktion eines Modells für die Generierung von Tests mittels Uppaal ist der bereits beschriebenen Ableitung endlicher Zustandsautomaten aus ROOMcharts ähnlich. In Uppaal werden zeitattributierte Automaten zur Systembeschreibung verwendet, die um binäre Synchronisation, ähnlich dem *Calculus of Communicating Systems* (CCS) [65], Daten, Wächterbedingungen und eine Ausdruckssprache zur Datenmanipulation erweitert sind. Da Zustandsautomaten in Uppaal die Definition von Wächterbedingungen an Transitionen erlauben, können diese von ROOMcharts nahezu unverändert übernommen werden. Im Folgenden werden die wesentlichen Schritte zur Ableitung von Uppaal-Automaten aus einem Testmodell erläutert.

8.12.1 Zustände und Transitionen

Für die Abbildung der Zustände eines hierarchischen ROOMcharts auf ein Testmodell für Uppaal wird eine Namenskonvention, ähnlich der Konvention für die Ableitung eines flachen, endlichen Zustandsautomaten, benötigt. Da Uppaal keine Doppelpunkte in Namen erlaubt, muss eine andere umkehrbare Substitution gefunden werden. Im Folgenden werden die Doppelpunkte lediglich durch Unterstriche ersetzt, wodurch die Verwendung von Unterstrichen in Namen in *Real-Time Object-Oriented Modeling* ausgeschlossen wird. Diese Einschränkung kann leicht aufgehoben werden, indem eine komplexere Namenskonvention eingeführt wird. Zugunsten der besseren Lesbarkeit, soll hier die einfache Darstellung mit Unterstrichen bevorzugt werden.

Der Initialpunkt eines ROOMcharts wird auf eine *Committed Location* abgebildet. Die initiale Transition wird auf eine Transition abgebildet, welche diese *Committed Location* verlässt. In dem Beispiel in Abbildung 8.20 ist die Abbildung eines einfachen ROOMcharts mit initialem Punkt und einer symbolischen Aktion *init* an der initialen Transition dargestellt.

8.12.2 Trigger und Aktionen

In Uppaal können das Empfangen und Senden einer Nachricht nicht explizit spezifiziert werden. Stattdessen besitzt Uppaal die Operatoren *!* und *?* für die Synchronisation zweier Transitionen in verschiedenen Prozessen über im System spezifizierte Kanäle. Das Empfangen und Senden von Nachrichten in *Real-Time Object-Oriented Modeling* kann nicht direkt auf Uppaal-Automaten übertragen werden, da diese keine unidirektionale Kommunikation erlauben. Es wird jedes Eingangssignal e_i eines Aktors auf einen Kanal mit dem Namen IN_e_i abgebildet. Jedes Ausgangssignal e_o wird entsprechend auf einen Kanal OUT_e_o abgebildet. Der Empfang einer Nachricht durch ein Signal e_i, und das Auslösen einer Transition durch diese Nachricht, wird durch $IN\_e_i?$ dargestellt, während das Schicken einer Nachricht durch eine Transition durch $OUT\_e_o!$ dargestellt wird.

Diese einfache Abbildung der Kommunikationsmechanismen aus *Real-Time Object-Oriented Modeling* auf die Uppaal Semantik ist nur möglich, da es sich bei dem Modultest um die separate Prüfung einer Komponente handelt. Daher ist der Testtreiber die einzige weitere Komponente im

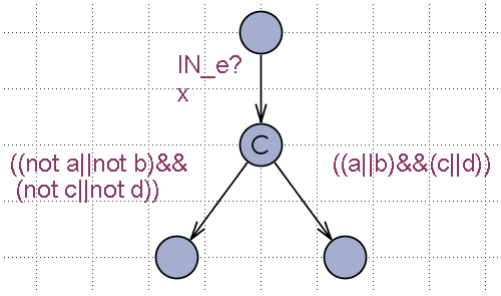


Abbildung 8.23: Uppaal-Automat einfache Bedingungsüberdeckung

System und als quasi synchron anzunehmen. Daher können Reihenfolgekonflikte nicht auftreten. Jedes Uppaal-Modell besteht aus einer Komponente für das betrachtete System und einer oder mehrerer Komponenten für die Umwelt mit der das System interagiert.

Für die Testfallgenerierung wird eine Treiberkomponente benötigt, welche die Synchronisation mit allen Transitionen in der Systemkomponente erlaubt. Die Treiberkomponente besitzt je nach Eingabealphabet einen hohen Grad an Nichtdeterminismus, der den notwendigen Spielraum für den Suchalgorithmus des *Model Checkers* bietet.

Das Beispiel in Abbildung 8.20 stellt die Abbildung eines einfachen beispielhaften ROOMcharts auf ein Uppaal-System dar. Auf der linken Seite ist ein einfaches ROOMchart mit den zwei Zuständen *A* und *B* und drei Transitionen ohne Aktion dargestellt. Dieses ROOMchart ist auf den Uppaal-Automaten auf der rechten Seite abgebildet. Die Zustände sind auf *Locations* im Uppaal-Automaten abgebildet. Eine *Location* ist semantisch zu einem Zustand in *Real-Time Object-Oriented Modeling* äquivalent und wird daher im Folgenden als semantischer Zustand bezeichnet. Der initiale Zustand *A* im ROOMchart ist im Uppaal-Automaten durch einen inneren Kreis gekennzeichnet. Die Namenskonvention folgt gleichen Regeln wie im Vorangegangenen für endliche Zustandsautomaten bereits ausführlich erläutert wurde.

Die Abbildung von Transitionen mit auslösendem Ereignis und mindestens einem ausgehenden Signal in der Aktion verlangt nach einer komplexeren Konstruktion in Uppaal. Eine Transition in Uppaal kann genau eine Synchronisation definieren. Durch die Verwendung von *Committed Locations*, die semantisch äquivalent zu Pseudozuständen in ROOMcharts sind, kann dieses Problem gelöst werden. Eine *Committed Location* ist äquivalent mit einem Pseudozustand und wird im gleichen Zeitschritt wieder verlassen. Durch eine Kettung von Transitionen mittels *Committed Locations* können Verarbeitungen und Versenden mehrerer Nachrichten modelliert werden.

In Abbildung 8.21 ist die Ableitung eines Uppaal-Automaten aus einem beispielhaften ROOMchart mit komplexen Transitionen dargestellt. Das Versenden von Nachricht *e* wird vereinfacht durch *send e* dargestellt. Die Transitionen mit den Trigger *e1* und *e2* schicken in unterschiedlicher Reihenfolge die Signale *x* und *y*. Im Uppaal-Automaten ist dies durch gekettete Transitionen dargestellt. Die erste Transition einer Kette bildet den Trigger ab, hier *IN_ e1?* oder *IN_ e2*?, während die folgenden Transition das Senden von Signalen abbilden, hier *OUT_ x!* und *OUT_ y!*. Die *Committed Locations* dienen als Bindeglieder zwischen den einzelnen Transitionen einer Kette und unterbrechen den Kontrollfluss nicht. Sie stellen daher keine semantischen Zustände in Uppaal dar und erhöhen daher nicht die Komplexität, was für die spätere Testfallgenerierung von besonderer Wichtigkeit ist.

8.12.3 Wächterbedingungen und Auswahlpunkte

Die Ableitung eines Testmodells für Uppaal aus einem ROOMchart mit Wächterbedingungen oder Auswahlpunkten kann durch Abbildung der ROOMchart Bedingungen auf Uppaal Wächterbedingungen geschehen. Im Gegensatz zu endlichen Zustandsautomaten sind Uppaal-Automaten neben anderen Erweiterungen auch um Wächterbedingungen an Transitionen erweitert. Unter der Voraussetzung, dass die in *Real-Time Object-Oriented Modeling* verwendeten Bedingungen auf Uppaal-Semantik abbildbar sind, können Wächterbedingungen übernommen und Auswahlpunkte

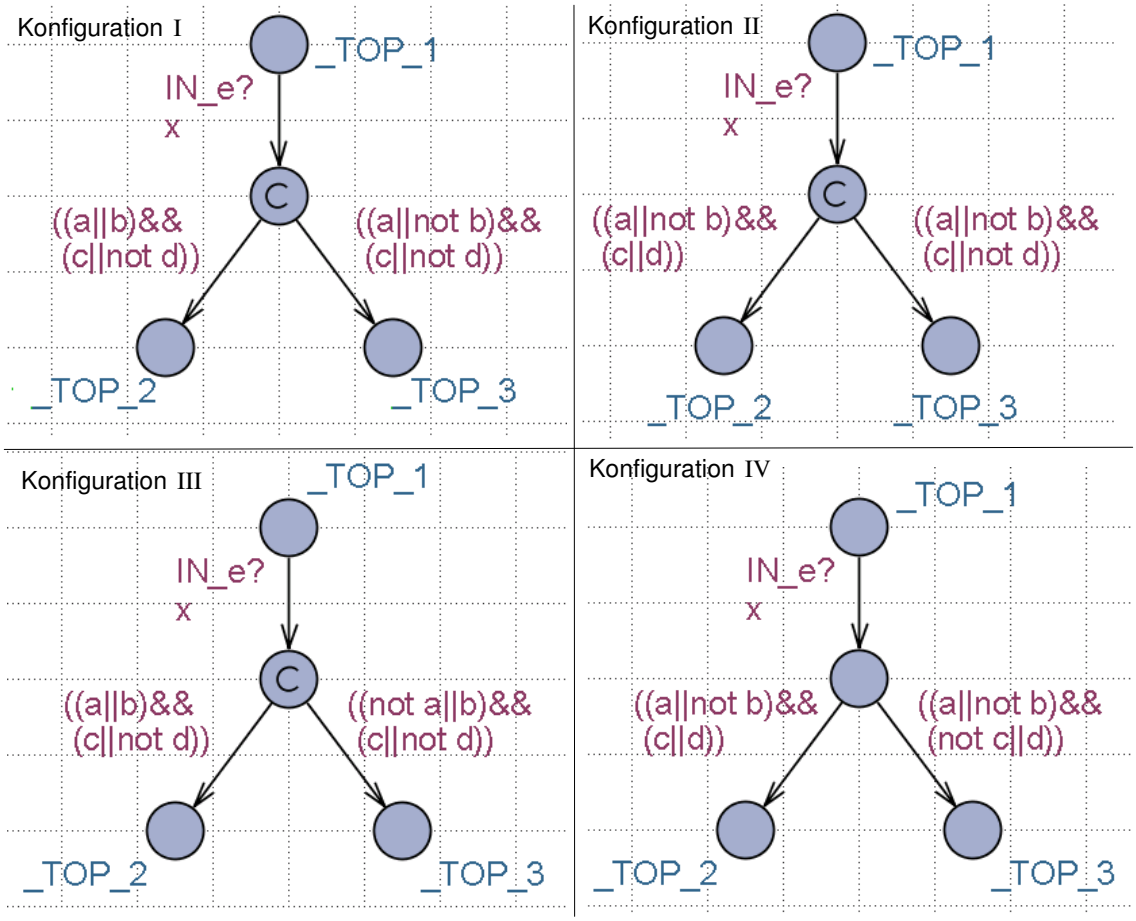


Abbildung 8.24: Uppaal-Automat MCDC

zu bedingten Transitionen umgeformt werden. Die Umformung von Auswahlpunkten zu Transitionen mit Wächterbedingungen ist dabei identisch zu der im Vorangegangenen beschriebenen Methode. Da in Uppaal Wächterbedingungen an Transitionen möglich sind, ist keine Abbildung bedingter Ereignisse auf terminale Symbole nötig, und somit werden auch keine Substitutionstabellen benötigt.

Das Beispiel in Abbildung 8.22 stellt die beispielhafte Ableitung eines ROOMcharts mit erweiterter Zustandsvariable x, Wächterbedingungen und Aktionscode dar. Die beiden Transitionen von Zustand A nach Zustand B werden durch das Ereignis e ausgelöst und besitzen die disjunkten Wächterbedingungen $x>=0$ und $x<0$. Die Transition von Zustand B nach Zustand A wird ebenfalls durch das Ereignis e ausgelöst und inkrementiert die ganzzahlige Variable x. Die ersten fünf Ereignisse e im Zustand A lösen die Transition mit der Ausgabe yx aus. Für alle weiteren Ereignisse e im Zustand A wird die Transition mit der Ausgabe xy ausgelöst.

8.12.4 Bedingungsüberdeckung

Die Anwendung von Bedingungsüberdeckungsverfahren erfolgt auf Basis der Transitionsüberdeckung verschiedener Konfigurationen des Testmodells. An anderer Stelle in dieser Arbeit wurde die Anwendung der einfachen Bedingungsüberdeckung und der modifizierten Bedingungs-/Entscheidungsüberdeckung gezeigt. An gleicher Stelle wurde der lineare Komplexitätszuwachs bei der Anwendung dieser Kriterien auf das vorliegende Testproblem dargestellt. Für das Beispiel in Abbildung 8.19 werden eine Konfiguration für einfache Bedingungsüberdeckung und vier Konfigurationen für modifizierte Bedingungsüberdeckung erzeugt. Eine *Committed Location* bildet die Kontrollflussverzweigung anstelle des Auswahlpunktes dar. Dies ermöglicht die Modellierung von Aktionen direkt

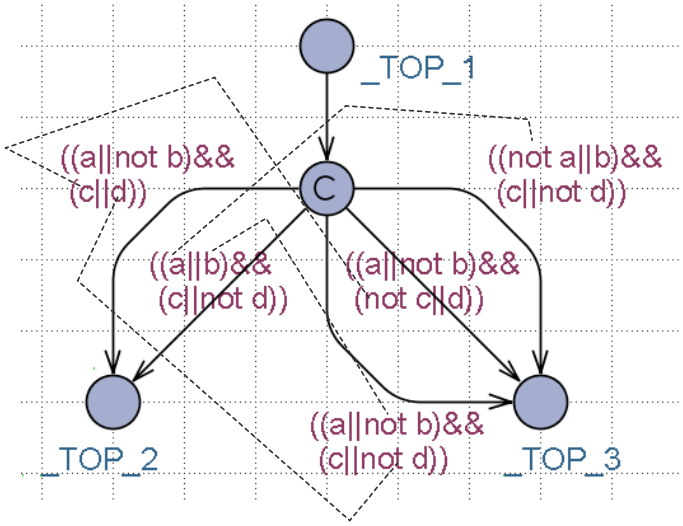


Abbildung 8.25: Kombinierter Uppaal-Automat MCDC

vor Evaluierung der Bedingungen. In dem Beispiel in Abbildung 8.19 wird die Aktion x ausgeführt, bevor die Bedingungen evaluiert werden. Die Bedingungen der Ersatztransitionen können in Uppaal unverändert als Wächterbedingungen dargestellt werden.

Das Beispiel in Abbildung 8.23 stellt das Uppaal-Testmodell für das Beispiel in Abbildung 8.19 bei einfacher Bedingungsüberdeckung dar.

Das Beispiel in Abbildung 8.25 stellt die vier Uppaal-Testmodelle für das Beispiel in Abbildung 8.19 bei modifizierter Bedingungs-/Entscheidungsüberdeckung dar. Jede Konfiguration des Beispiels in Abbildung 8.19 wird auf ein Testmodell in Uppaal abgebildet. Da bei dieser einfachen Art der Abbildung der modifizierten Bedingungs-/Entscheidungsüberdeckung verschiedene Teilkonfigurationen der Bedingungen, beispielsweise *(a&&b)||(c&&not d)*, mehrfach berechnet werden, wird ein kombiniertes Testmodell konstruiert. Jede Teilkonfiguration ist disjunkt zu allen anderen Teilkonfigurationen, daher können alle Konfigurationen in einem Testmodell berechnet werden. In Abbildung 8.25 ist ein aus den vier Konfigurationen I, II, III und IV kombiniertes Testmodell dargestellt. Da sich die Wächterbedingungen gegenseitig ausschließen, besitzt dieses Testmodell die gleiche Komplexität wie jedes der vier einzelnen Modelle. An anderer Stelle in dieser Arbeit erfolgt eine detailliertere Komplexitätsbetrachtung.

8.12.5 Gedächtniszustände

Die Ableitung eines Uppaal Testmodells von einem ROOMchart mit Gedächtnisfunktion wird nach der im Vorangegangenen beschriebenen Methode für Testmodelle mit der Semantik endlicher Zustandsautomaten durchgeführt. Da Uppaal-Automaten im Gegensatz zu endlichen Zustandsautomaten die Definition von erweiterten Zustandsvariablen und Wächterbedingungen erlauben, kann auf die Abbildung der bedingten Ereignisse auf atomare Symbole mittels Substitutionstabelle verzichtet werden.

Für jeden hierarchischen Zustand mit Gedächtnisfunktion wird eine zusätzliche Gedächtnisvariable im abgeleiteten Uppaal-Automaten eingeführt, die mit einem eindeutigen Schlüssel für den derzeitigen Gedächtniszustand, vor dem ersten Betreten der initiale Zustand des hierarchischen Zustands, definiert ist. Jede Ersatztransition der Gedächtnisfunktion wird durch eine Wächterbedingung ausgestattet, die überprüft, ob der Schlüssel des jeweiligen Zielzustands gleich dem Wert der Gedächtnisvariablen ist.

Das Beispiel in Abbildung 8.26 präsentiert ein ROOMchart mit Gedächtnisfunktion am hierarchischen Zustand *D*. Das ROOMchart besitzt drei elementare Zustände, von denen zwei in *D* enthalten sind. Die Transition von Zustand *A* nach Zustand *D* ist eine Gedächtnistransition und wird im Uppaal-Modell durch drei Transitionen ersetzt. Da die Ersatztransitionen einen Trigger

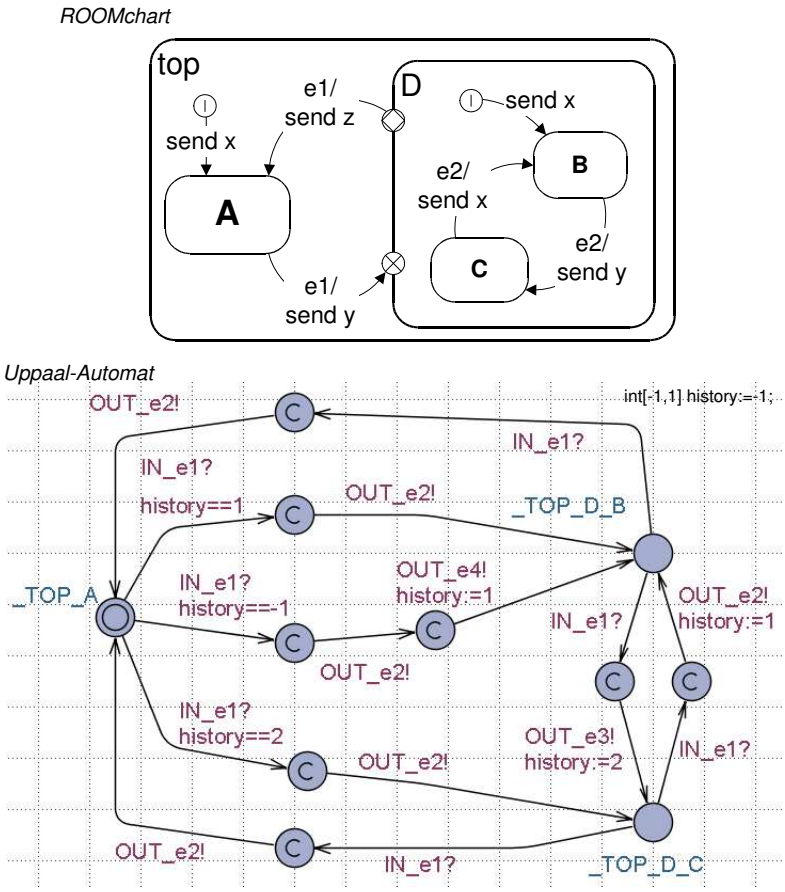


Abbildung 8.26: Gedächtnisfunktion in Uppaal

und mehrere Aktionen enthalten, muss die im vorangegangenen Abschnitt vorgestellte Vorgehensweise angewandt werden. Für das erstmalige Betreten von D wird eine Ersatztransition aus drei Teilen eingeführt, die den Trigger und die Aktion der Gedächtnistransition und die Aktion der initialen Transition von D ausführt. Die beiden Ersatztransitionen in die Zustände C und B, bzw. $\_TOP\_D\_B$ und $\_TOP\_D\_C$, enthalten den Trigger und die Aktion der Gedächtnistransition. Der UPPAAL-Automat besitzt für die Gedächtnisfunktion des Zustands D die ganzzahlige Variable *history*, welche mit dem Wert *0* für den Zustand B, den Wert *1* für den Zustand C und dem Wert *-1* für das erstmalige Betreten des hierarchischen Zustands D definiert wird. Jede Ersatztransition ist mit einer Wächterbedingung ausgestattet, in der die Äquivalenz des Werts von *history* mit dem Schlüsselwert des jeweiligen Zielzustands, also der Wert *0* für B und der Wert *1* für C, oder mit dem Initialwert *-1* geprüft wird.

8.12.6 DS Modelle

Die Anwendung automatenbasierter Testverfahren benötigt neben einer strukturüberdeckenden Testfallmenge eine charakteristische Eingabemenge, die jeden Zustand identifiziert. Im Rahmen dieser Arbeit erfolgt die Zustandsidentifikation mittels *Distinguishing Sequences* (DS), da diese vergleichsweise kurze Testsequenzen bewirken. Die Generierung von *Distinguishing Sequences* mittels Uppaal erfolgt durch die Ableitung eines zusätzlichen DS-Modells aus einem Testmodell. Da *Distinguishing Sequences* das beobachtbare Verhalten darstellen und in *Model Checkern* dagegen Daten und Zustände zur Formulierung von Bedingungen dienen, werden die Ein- und Ausgaben der Transitionen des Testmodells auf Daten im DS-Modell abgebildet.

Eine *Distinguishing Sequence* eines Automaten besteht aus einer Eingabesequenz und jeweils einer charakteristischen Ausgabe für jeden der Zustände des Automaten. Eine Eingabesequenz

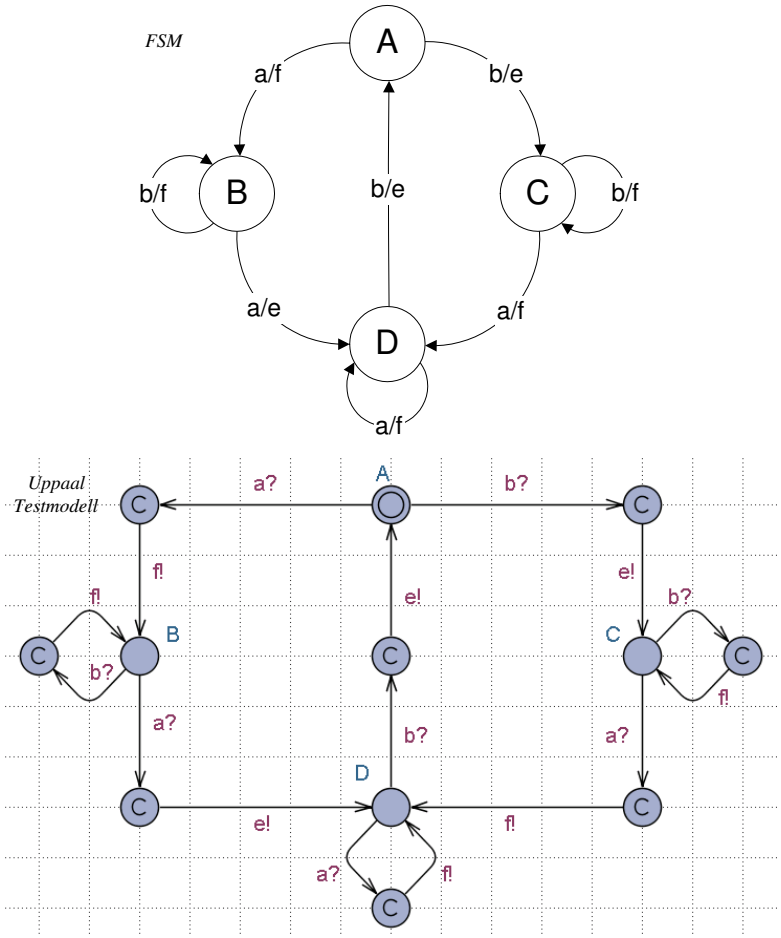


Abbildung 8.27: FSM Testmodell

ds_M eines Automaten M, die in einem Zustand s_i ausgeführt wird, produziert eine für s_i charakteristische Ausgabe o_i. Die Ausgabe o_i gleicht keiner der Sequenzen o_j, die aus den Zuständen s_j, mit $i \neq j$, von M produziert wird. Für die Generierung einer *Distinguishing Sequence* wird im DS-Modell für jeden Zustand s_i ein Automat M_i benötigt, der den Startzustand s_i besitzt. Jeder Automat M_i erhält eine Instrumentierungsvariable out_i, welche die kodierten, aktuellen Ausgaben der M_i über einen Suchpfad darstellen. Jede Ausgabe wird eindeutig mit einem ganzzahligen Wert kodiert. In diesem Kapitel wurde bereits die Abbildung von komplexen Ereignissen auf einfache Symbole mittels einer Substitutionstabelle dargestellt, die im Folgenden auch für DS-Modelle geeignet ist. Da in einem DS-Modell mehrere Systemautomaten der M_i mit einem Treiberautomaten synchronisieren, müssen Breitbandkanäle (*engl. broadcast channel*) definiert werden. Im Treiberautomaten des DS-Modells erfolgt die Auswertung der Ausgaben out_i der einzelnen Automaten M_i nach jeder Eingabe. Im Treiberautomaten werden nach jeder Eingabe und den resultierenden Antworten der M_i die folgenden Bedingungen evaluiert:

$$\prod_{i=0}^{n-1} \prod_{j=i+1}^{n} out_i \neq out_j$$

Für jede dieser Bedingungen enthält das DS-Modell eine boolesche Variable $distinct_{i,j}$, die mit *False* initialisiert wird. Eine Variable $distinct_{i,j}$ wird mit *Wahr* definiert, wenn eine Bedingung $out_i \neq out_j$ nach einer Eingabe gilt. Für jede *Distinguishing Sequence* eines Automaten M müssen alle Variablen $distinct_{ij}$ mit *Wahr* definiert worden sein. Daher kann für die Anforderung einer *Distinguishing Sequence,* basierend auf einem DS-Modell eines Automaten *M,* die folgende Konstruktionsvorschrift angegeben werden, wobei das Produkt in Uppaal durch den Operator *&&* notiert wird:

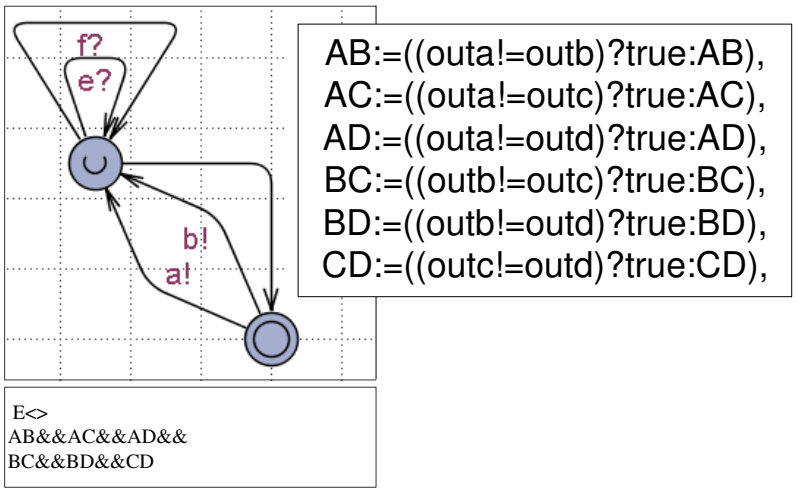


Abbildung 8.28: FSM DS Modell Treiber

	baa
A	*eff*
B	*eef*
C	*fff*
D	*efe*

Tabelle 8.2: Kürzeste *Distinguishing Sequence* für Abbildung 8.27

$$E\Diamond \prod_{i=0}^{n-2} \prod_{j=i+1}^{n-1} distinct_{i,j}$$

Der *Model Checker* Uppaal besitzt die Option den kürzesten Pfad für die Erfüllung einer Anforderung auszugeben. Daher ist mit der hier beschriebenen Methode die Generierung kürzester *Distinguishing Sequence*s möglich.

In den Abbildungen 8.27, 8.28 und 8.29 ist das Beispiel eines DS-Modells für einen endlichen Zustandsautomaten mit vier Zuständen dargestellt. Die Ausgabeereignisse e und f sind auf die Werte 1 und 2 abgebildet. Die leere Ausgabe wird durch den Wert 0 dargestellt, in diesem Beispiel jedoch nicht benötigt. Jede Ausgabevariable wird mit -1 initialisiert. Der Treiberautomat besitzt einen Initialzustand, der mit Auftreten der Eingangssignale a oder b verlassen wird. Der dringliche Zustand besitzt reflexive Transitionen für die Antworten der Automaten M_A, M_B, M_C und M_D. Nach Empfang aller Antworten wird die Transition zurück in den Initialzustand ausgelöst, in welcher die Auswertung der Ausgaben erfolgt. Für vier Zustände werden die sechs Ausgabebedingungen $out_A \neq out_B$, $out_A \neq out_C$, $out_A \neq out_D$, $out_B \neq out_C$, $out_B \neq out_D$, $out_C \neq out_D$ ausgewertet und den booleschen Vergleichsvariablen AB, AC, AD, BC, BD und CD zugewiesen, wenn die Bedingungen *Wahr* sind. Eine *Distinguishing Sequence* wird gefunden, wenn ein Pfad existiert, auf dem alle Vergleichsvariablen den Wert *Wahr* annehmen.

Für das FSM-Beispiel in den Abbildungen 8.27, 8.28 und 8.29 lautet die entsprechende Anforderung in Uppaal:

$$E\Diamond\, AB\&\&AC\&\&AD\&\&BC\&\&BD\&\&CD$$

Die Generierung der kürzesten *Distinguishing Sequence* mit Uppaal liefert die Eingabesequenz *baa*, dargestellt in Tabelle 8.2.

In den Abbildungen 8.30, 8.31 und 8.32 wird das Beispiel eines DS-Modells für einen erweiterten, endlichen Zustandsautomaten veranschaulicht. Der erweiterte Automat besitzt die Zustände A, B und C. An Zustand B sind zwei reflexive, bedingte Transition definiert, die in Abhängigkeit von der erweiterten Zustandsvariablen x ausgelöst werden, wenn das Ereignis b eintritt. Eine transitionsüberdeckende Eingabemenge mit den zugehörigen, erweiterten Endzuständen ist in den

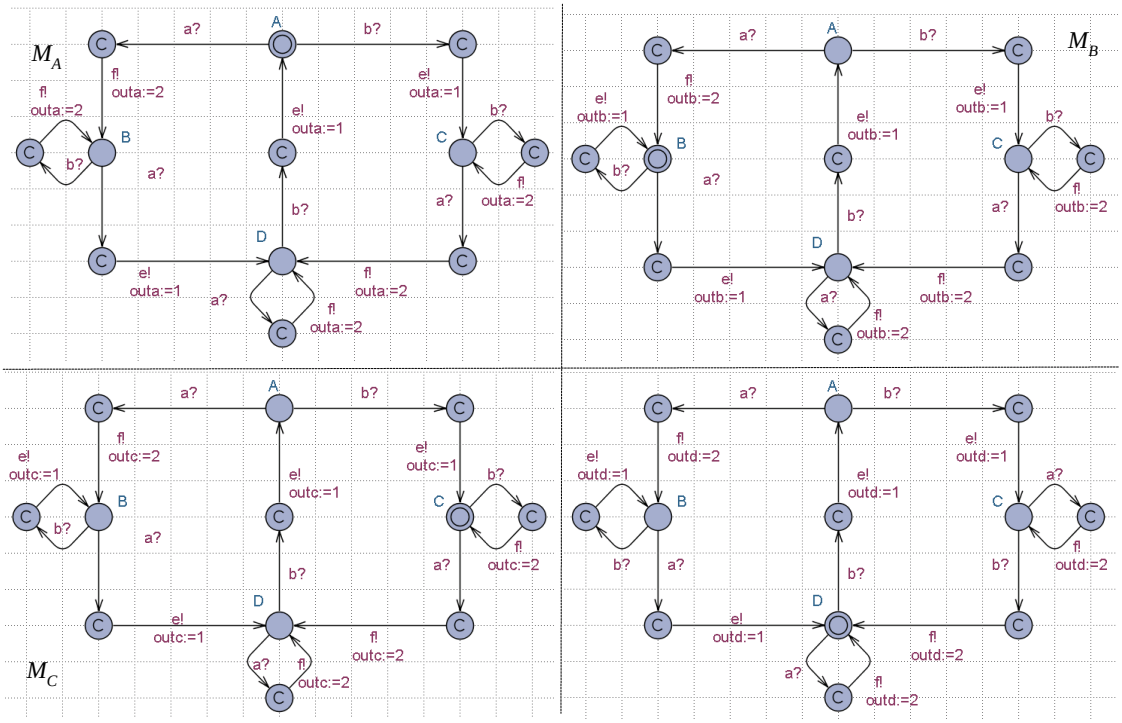


Abbildung 8.29: FSM DS Modell

Tabellen 8.3 und 8.4 dargestellt. Die Generierung einer *Distinguishing Sequence* für einen erweiterten Zustandsautomaten macht die Berücksichtigung erweiterter Zustände notwendig [122]. Daher besitzt das DS-Modell in Abbildung 8.32 die Automaten M_A, M_{B0}, M_{B1}, M_{B2} und M_C, welche die erweiterten Zustände

$$(A, x = 0), (B, x = 0), (B, x = 1), (B, x = 2), (C, x = 0)$$

repräsentieren.

Der entsprechende Treiberautomat in 8.31 evaluiert die Ausgaben der Teilautomaten und bestimmt die Werte der Variablen AB0, AB1, AB2, AC, B0C, B1C und B2C. Die Bedingungen

$$outb0 \neq outb1, outb0 \neq outb2, outb1 \neq outb2$$

werden an dieser Stelle nicht evaluiert, da die Existenz einer *Distinguishing Sequence* für die betrachteten erweiterten Zustände nicht garantiert werden kann. Die Berücksichtigung der erweiterten Zustände dient der Verifikation, so dass die ermittelte *Distinguishing Sequence* auch aus den erweiterten Zuständen jeden nicht erweiterten Zustand identifiziert.

Auf Basis der temporallogischen Anforderung

$$E\Diamond\ AB0\&\&AB1\&\&AB2\&\&AC\&\&B0C\&\&B1C\&\&B2C$$

produziert der *Model Checker* DS:*ab*, dargestellt in Tabelle 8.3.

Die Anforderung nach einem minimalen Automaten bezieht sich nicht auf die Erweiterung der betrachteten Automaten. Daher kann nicht garantiert werden, dass der durch die Erweiterung aufgespannte Zustandsautomat ebenfalls minimal ist und somit jeder Zustand anhand seines charakteristischen Ein-/Ausgabeverhaltens identifiziert werden kann. Trotzdem besteht die Möglichkeit, dass für einen eingeschränkten Teil des erweiterten Zustandsraums eine *Distinguishing Sequence* existiert. In dem Fall des Beispiels der Abbildungen 8.30, 8.31 und 8.32 kann eine *Distinguishing Sequence* für die betrachteten erweiterten Zustände generiert werden, indem die Bedingungsvariablen

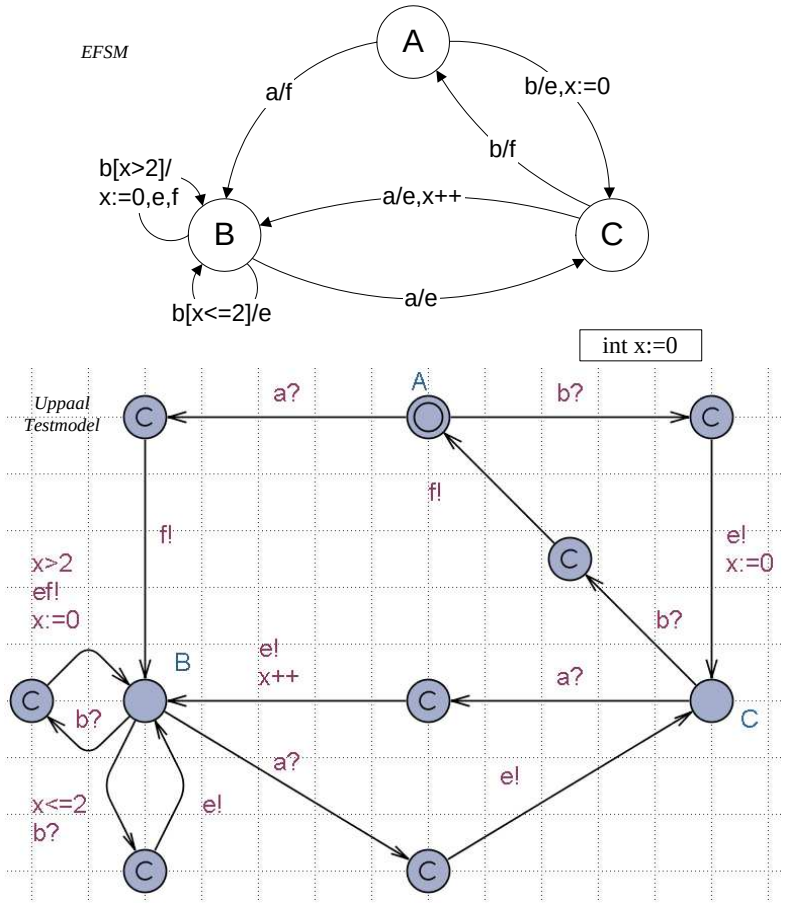


Abbildung 8.30: EFSM Modell

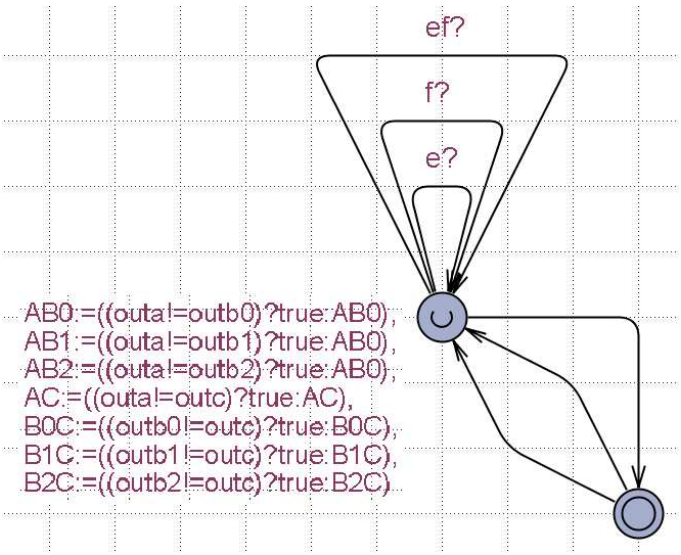


Abbildung 8.31: EFSM DS Modell Treiber

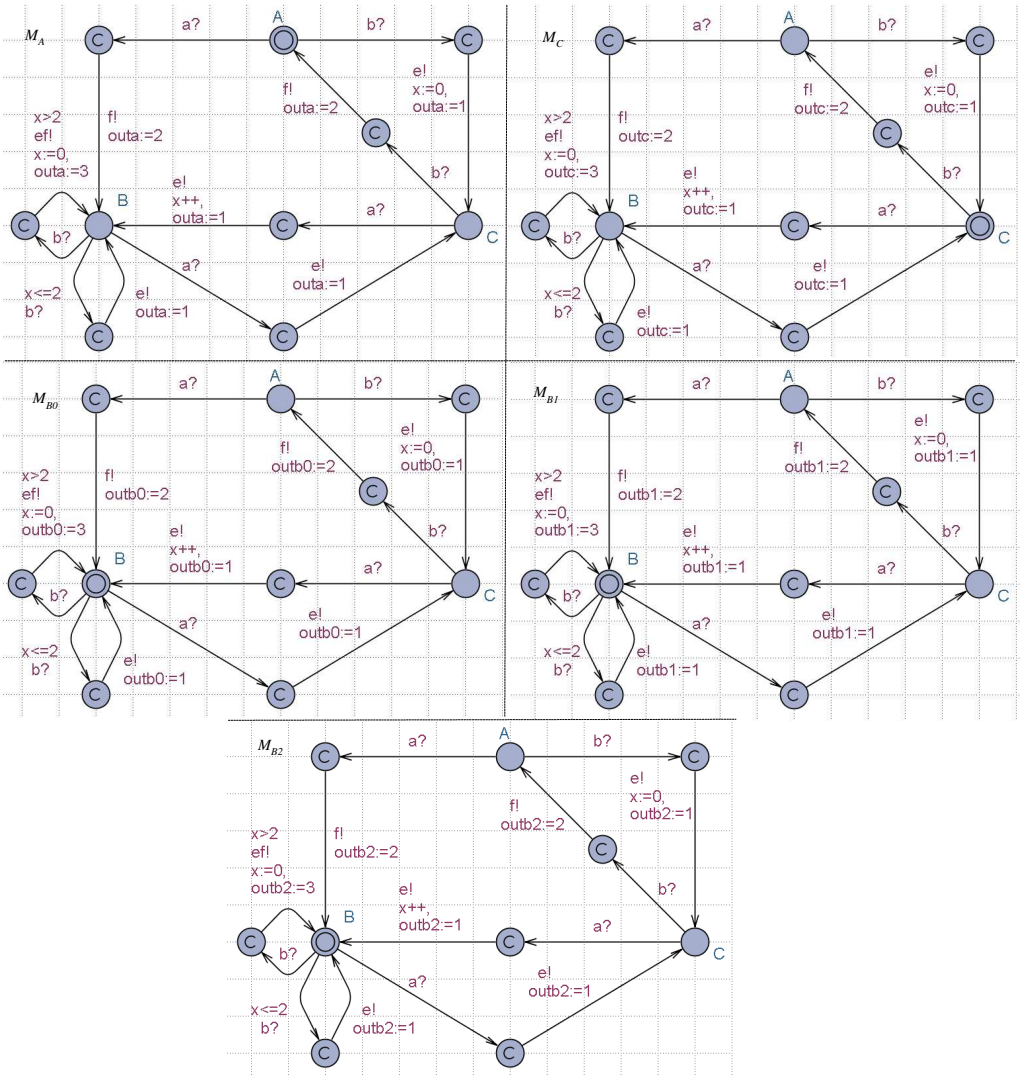


Abbildung 8.32: EFSM DS Modell

B0B1,B0B2,B1B2
zu dem Treiberautomaten und der temporallogischen Anforderung hinzugefügt werden. Für diesen Fall produziert der *Model Checker* auf die folgende Anforderung das Ergebnis DS:*aabaab*, dargestellt in Tabelle 8.4:

$$E\Diamond \quad AB0\&\&AB1\&\&AB2\&\&AC\&\&B0B1\&\& \\ B0B2\&\&B1B2\&\&B0C\&\&B1C\&\&B2C$$

Komplexitätsbetrachtung

Eine detaillierte Komplexitätsbetrachtung der Testfallgenerierung mit Uppaal und Breitensuche wird an anderer Stelle in dieser Arbeit präsentiert. An dieser Stelle soll die Komplexität des vorgestellten Ansatzes zur Generierung von *Distinguishing Sequences* erläutert werden.

Die Komplexitäten des vorgeschlagenen Ansatzes zur Generierung von *Distinguishing Sequence*s mittels *CTL Model Checking* wird durch das Eingangsmodell und durch den *Model Checking* Algorithmus beeinflusst. Ein DS-Modell impliziert keine höhere zeitliche Komplexität als das Testmodell, aus dem es abgeleitet wurde. Daher ist der vorgestellte Ansatz in den meisten Fällen anwendbar, wenn Testfallgenerierung mittels *CTL Model Checking* anwendbar ist.

Eingabe	Endzustand	Erweiterter Endzustand	DS: *ab*
a	*B*	x=0	*ef*
aa,b	*C*	x=0	*ee*
bb	*A*	x=0	*fe*
ba	*B*	x=1	*ef*
baaab	*B*	x=2	*ef*

Tabelle 8.3: Kürzeste *Distinguishing Sequence* unter erweiterten Zuständenin Abbildung 8.30

Obwohl ein betrachteter Automat M k-fach, mit der Anzahl Zustände k, vervielfältigt wird und zusätzliche Instrumentierungsvariablen notwendig sind, haben diese Modifikationen keinen Einfluss auf die effektive zeitliche Komplexität des *Model Checking* Problems.

In [103] haben Huhn und Mücke demonstriert, dass für Breitensuche und Anforderungen, die auf einem vergleichsweise kurzen Pfad erfüllt sind, die Zeitkomplexität des *Model Checking* Problems $O(n^l)$ ist, mit n als dem nichtdeterministischen Freiheitsgrad des Systems und der Pfadlänge l. Die Pfadlänge l einer kürzesten *Distinguishing Sequence* hängt ausschließlich von der Struktur des betrachteten Automaten ab. Der nichtdeterministische Freiheitsgrad n ist maximal gleich der Größe des Eingabealphabets des Automaten. Die zusätzlich benötigten Variablen erhöhen die Zeitkomplexität nicht, da sie weder n noch l beeinflussen. Dagegen erhöhen diese Variablen den zusätzlichen Speicherbedarf für die Darstellung eines Zustands linear. Die Vervielfältigung der Automaten, und folglich des Zustandsraums, beeinflusst die zeitliche Komplexität nicht, da die Automaten synchron mit dem Treiberautomaten ausgeführt werden.

Aus den genannten Gründen ist der hier präsentierte Ansatz der Generierung von *Distinguishing Sequence*s mittels *CTL Model Checking* zeitlich unabhängig von der Anzahl Zustände, polynomial von der Größe des Eingabealphabets und exponential von der Länge der *Distinguishing Sequence*. Die Experimente im Rahmen dieser Arbeit haben gezeigt, dass in den meisten Fällen eine *Distinguishing Sequence* von akzeptabler Länge existiert und der Ansatz in der Praxis einsetzbar ist. Weiterhin kann mittels Breitensuche eine kürzeste *Distinguishing Sequence* gefunden werden, wodurch der Zeitverbrauch gegenüber Tiefensuche deutlich reduziert werden kann.

Die Raumkomplexität dieses Ansatzes kann mittels alternativer DS-Anforderungen reduziert werden. Beispielsweise kann eine *Distinguishing Sequence* gefunden werden, wenn in einem Zustand alle Ausgaben der Teilautomaten gleichzeitig verschieden sind. Für das Beispiel in Abbildung 8.31 würde eine alternative Bedingung folgendermaßen lauten:

$$E\Diamond \quad outa \neq outb \&\& outa \neq outc \&\& outa \neq outd \&\& \\ outb \neq outc \&\& outb \neq outd \&\& outc \neq outd$$

Diese Bedingung erlaubt in der Regel nicht die Konstruktion einer optimalen *Distinguishing Sequence*, reduziert allerdings die Anzahl notwendiger Instrumentierungsvariablen von n^2 auf n.

8.12.7 Wp- und UIO-Generierung

Auf Basis des im Vorangegangenen vorgestellten DS-Modells können auch auf einfache Weise UIOs und Wp-Sets generiert werden.

Eingabe	Endzustand	Erweiterter Endzustand	DS: *aabaab*
a	*B*	x=0	*eeeeee*
aa,b	*C*	x=0	*eeffef*
bb	*A*	x=0	*feffef*
ba	*B*	x=1	*eeeeeef*
baaab	*B*	x=2	*eeefeee*

Tabelle 8.4: Kürzeste *Distinguishing Sequence* für erweiterte Zuständein Abbildung 8.30

Eine *Unique Input Output Sequence* (UIO) ist eine Sequence, die genau einen Zustand eines endlichen Zustandsautomatens identifiziert. Für einen Automaten mit n Zuständen sind folglich n UIOs notwendig, um alle Zustände eindeutig identifizieren zu können.

Für die Generierung aller UIOs eines endlichen Zustandsautomaten mit n Zuständen sind ebenfalls n Anforderungen in Uppaal notwendig. Jede Anforderung verlangt die Unterscheidung des betreffenden Zustands s_i von allen anderen Zuständen s_j, mit $i \neq j$.

Für das FSM-Beispiel in den Abbildungen 8.27, 8.28 und 8.29 lautet die Konstruktionsvorschrift für n UIOs eines endlichen Zustandsautomaten mit n Zuständen:

$$UIO_M = E\Diamond \bigcup_{i=0}^{n-1} \prod_{j=0}^{n-1} (j = j) \vee distinct_{i,j}$$

Die Generierung eines *Wp-Sets* kann ebenfalls auf Basis des DS-Modells erfolgen. Ein *Wp-Set* besteht aus einer Menge von Eingabesequenzen, die einen Zustand eindeutig anhand der produzierten Ausgaben identifizieren, wenn sie in diesem Zustand ausgeführt werden.

Ein *Wp-Set* wird aus verschiedenen Eingabesequenzen generiert, von denen jede mindestens einen Zustand von einem anderen Zustand anhand der produzierten Ausgaben unterscheidet.

Im ersten Schritt wird ein *Wp-Set* mittels folgender Vorschrift generiert:

$$W_p = \bigcup_{i=0}^{n-2} \bigcup_{j=i+1}^{n-1} E\Diamond\, distinct_{i,j}$$

In einem zweiten Schritt werden aus dem *Wp-Set* alle Sequenzen entfernt, die Startsequenzen einer anderen Sequenz im *Wp-Set* sind.

$$W_p = \{s, t \in W_p |\, \neg\exists_t st \in W_p\}$$

Die Komplexitäten der Generierung von *Wp-Sets* und UIOs entsprechen denen der Generierung von *Distinguishing Sequences*. Da die Bedingungen für die Existenz von *Wp-Set* und UIOs schwächer als die für *Distinguishing Sequences* sind, können allerdings vergleichsweise kurze Sequenzen erwartet werden. Eine Aussage, welche der drei Methoden im allgemeinen Fall die günstigste ist, kann daraus allerdings nicht abgeleitet werden.

Kapitel 9

Testfallermittlung

Die Testfallermittlung für ROOMcharts soll auf den im Vorangegangenen vorgestellten Testmodellen erfolgen. In diesem Kapitel wird eine allgemeine Methode der Testfallgenerierung basierend auf dem Testmodell mit der Semantik endlicher Zustandsautomaten vorgestellt. Weiterhin wird die Testfallgenerierung mit dem *Model Checker* Uppaal ausführlich dargestellt. Da die Ausführbarkeit von Pfaden und Transitionen im ROOMchart ein grundsätzliches Problem beider Ansätze ist, wird es allgemein erläutert.

9.1 Ausführbarkeit von Pfaden

Durch die Verwendung von Wächterbedingungen und Auswahlpunkten werden Pfade und Transitionen im Zustandsautomaten möglicherweise unausführbar. Die Frage, ob und unter welchen Bedingungen ein Pfad ausgeführt werden kann, ist als *Path Feasibility Problem* [70] bekannt und NP-vollständig [117, 116].

Ein Pfad p in einem ROOMchart ist eine Sequenz von Transitionen $<t_0,...,t_m>$, wobei t_0 die erste Transition des Pfades bezeichnet. Die Sequenz korrespondierender Ereignisse $<e_0,...,e_m>$ - ein Ereignis e_i löst eine Transition t_i aus - führt in einem ROOMchart zu der Ausführung von p. Die Wächterbedingung einer Transition enthält eine aussagenlogische Formel über erweiterte Zustandsvariablen und Ereignisparameter. Ein direkter, externer Zugriff auf erweiterte Zustandsvariablen ist nicht möglich, da diese durch die Kapselung des Aktors geschützt werden. Ein Ereignisparameter ist innerhalb der Aktion einer Transition gültig und kann für Wertzuweisungen an erweiterte Zustandsvariablen oder Berechnungen dienen. Der Wert einer erweiterten Zustandsvariablen kann in der Aktion einer Transition durch Ereignisparameter definiert werden. Daher bestimmen Ereignisparameter, neben dem Auftreten des Ereignisses im Ausgangszustand, sowohl mittelbar als auch unmittelbar das Auslösen einer Transition. Die Frage, ob eine Transition unter einer bestimmten Wertebelegung der erweiterten Zustandsvariablen und der Ereignisparameter ausgelöst werden kann, wird als Ausführbarkeit der Transition bezeichnet. Dementsprechend entscheidet sich die Ausführbarkeit eines Pfades an der Ausführbarkeit der in ihm enthaltenen Transitionen.

Eine nicht ausführbare Transition wird als fehlerhaft angenommen. Dies ist eine Anforderung, die der Abwesenheit von nicht ausführbarem Programmcode entspricht. Wenn Wächterbedingungen oder Auswahlpunkte im Entwurf genutzt werden, ist die Existenz nicht ausführbarer Pfade kaum zu vermeiden. Für die automatische Testfallgenerierung ist dies allerdings äußerst problematisch.

Die folgenden Beispiele demonstrieren das Problem der Ausführbarkeit von Transitionen und Pfaden in ROOMcharts, in Abhängigkeit von erweiterten Zustandsvariablen und Ereignisparametern.

In Abbildung 9.1 ist beispielhaft das Problem der Ausführbarkeit einer Transition in Abhängigkeit von einem Ereignisparameter dargestellt. Das ROOMchart enthält die elementaren Zustände *1* und *2* im Oberzustand *top*, eine ganzzahlige erweiterte Zustandsvariable *a* und drei Transitionen. Die initiale Transition führt zum Zustand *1* und definiert in der Aktion die erweiterte Zustandsvariable *a* mit *0*. Die reflexive Transition an Zustand *1* wird durch das Ereignis *e1* ausgelöst, das

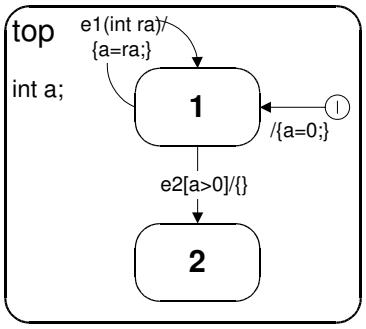


Abbildung 9.1: Ausführbarkeit in Abhängigkeit von Ereignisparametern

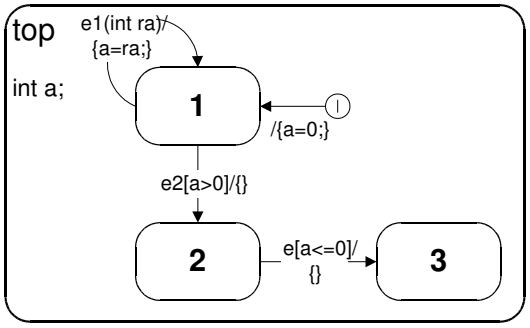


Abbildung 9.2: Widersprüchliche Wächterbedingungen

den Parameter *ra* besitzt und in der Aktion *a* mit *ra* definiert. Die Transition von Zustand *1* zu Zustand *2* wird durch das Ereignis *e2* ausgelöst, unter der Bedingung, dass *a* größer *0* ist. Die Ausführung der Transition von Zustand *1* zu Zustand *2* hängt von dem Ereignisparameter *ra* ab.

In Abbildung 9.2 ist ein Beispiel eines nicht ausführbaren Pfades gegeben. Die Wächterbedingungen der Transitionen von Zustand *1* nach Zustand *2* und von Zustand *2* nach Zustand *3* sind widersprüchlich. Die Transition von Zustand *2* nach Zustand *3* ist nicht ausführbar.

Eine nicht ausführbare Transition kann durch zusätzliche Gruppentransitionen zur Definition der erweiterten Zustandsvariablen ausgeführt werden. Diese Transitionen können zu Testzwecken automatisch erzeugt und entfernt werden.

In Abbildung 9.3 ist ein beispielhaftes ROOMchart mit einer zu Testzwecken erzeugten internen Gruppentransition *setA* dargestellt. Diese Gruppentransition dient zum Definieren der erweiterten Zustandsvariablen *a*. Die im vorherigen Beispiel nicht ausführbare Transition von Zustand *2* zu Zustand *3* kann jetzt ausgeführt werden, indem im Zustand *2* die interne Gruppentransition mit einem Ereignis *setA(ra)*, mit $ra \leq 0$, ausgeführt wird.

Die Generierung von internen Gruppentransitionen zur Definition erweiterter Zustandsvariablen kann die Anzahl benötigter Testeingaben zur Erfüllung von Überdeckungskriterien erheblich reduzieren.

Das Beispiel in Abbildung 9.4 stellt die mögliche Verkürzung eines Transitionsüberdeckungstests dar. Das beispielhafte ROOMchart enthält die elementaren Zustände *1* und *2*, sowie vier Transitionen und eine automatisch generierte, interne Gruppentransition zur Definition der erweiterten Zustandsvariablen *a*. Ohne die interne Gruppentransition würde ein Transitionsüberdeckungstest mindestens die Ereignisfolge

<e1,e1,e1,e1,e1,e1,e1,e1,e1,e1,e1,e1,e1,e1,e1,e1,e1,e1,e1,e1,e1,e1,e1,e1,e2>

notwendig machen. Mit der internen Gruppentransition kann Transitionsüberdeckung durch die Ereignisfolge

<e1,setA(11),e2,e1>

erreicht werden.

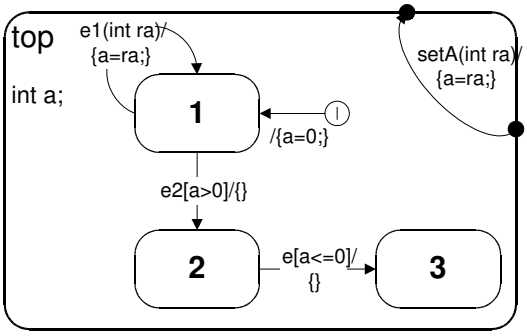


Abbildung 9.3: Ausführbarbeit durch Instrumentierung

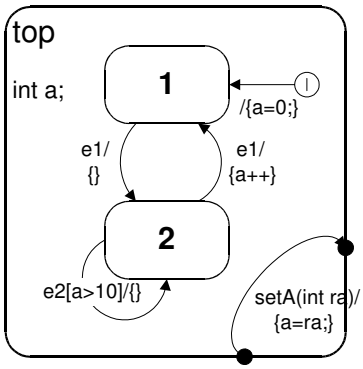


Abbildung 9.4: Verkürzung von Testpfaden

Die Beispiele in den Abbildungen 9.3 und 9.4 haben zwei mögliche Konsequenzen der Generierung von internen Gruppentransitionen zur Definition von erweiterten Zustandsvariablen demonstriert. Einerseits kann eine erhebliche Verkürzung der notwendigen Testsequenzen erreicht werden, andererseits können unsinnige Testfolgen entstehen, wenn ursprünglich nicht ausführbare Transitionen getestet werden.

9.2 Testfallermittlung mittels Graphentheorie

Die Testfallermittlung auf Basis von erweiterten Zustandsautomaten kann mittels Suchalgorithmen zur Testpfadermittlung und arithmetischen Algorithmen zur Analyse der Ausführbarkeit des Pfades erfolgen. Ein wesentlicher Nachteil vieler in der Literatur beschriebener Verfahren ist die mangelhafte Berücksichtigung nicht ausführbarer Pfade und Transitionen.

Für die Generierung von Testpfaden wird häufig die Breitensuche zugunsten der ineffizienteren Tiefensuche bevorzugt [37]. Die Berechnung der notwendigen Wertebelegung für die erweiterten Zustandsvariablen und Ereignisparameter können im zweiten Schritt mittels *Constraint Solving*, beispielsweise [132, 133] erfolgen. Die Berechnung der notwendigen Wertebelegung der Ereignisparameter wird als Testdatenermittlung bezeichnet.

Am Beispiel in Abbildung 9.4 soll die Testfallermittlung mittels dem Suchalgorithmus von Chow zur Transitionüberdeckung und die Testdatenermittlung mittels *Constraint Solving* basierend auf Aussagelogik demonstriert werden. Der Suchalgorithmus von Chow ist eine Breitensuche beginnend im initialen Zustand, die einen Suchpfad im Endzustand oder nach Erreichen eines elementaren Zyklus terminiert. Eine mögliche Testpfadmenge nach Chow, für das Beispiel in Abbildung 9.4 unter *Completeness Assumption* und ohne Zuhilfenahme der internen Gruppentransition, ist beispielhaft durch den Baum in Abbildung 9.5 dargestellt. Die unterstrichenen Bezeichner markieren die Endzustände der Pfade, die jeweils einen elementaren Zyklus abschließen. Diese Methode berück-

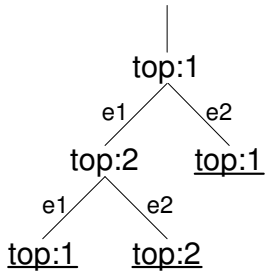


Abbildung 9.5: Pfadbaum für Abbildung 9.4

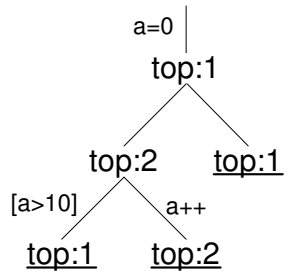


Abbildung 9.6: Bedingter Pfadbaum für Abbildung 9.4

sichtigt nicht, dass der Pfad *<top:1,top:2,top:1>* nicht ausführbar ist. Der Pfadbaum in Abbildung 9.6 ist mit den Aktionen und Wächterbedingungen attributiert und verdeutlicht den Widerspruch im beschriebenen Testpfad. Die Auswertung des Pfadbaums mit Zusicherungsmethoden und symbolischer Ausführung während der Testdatengenerierung kann auch zu dem Ergebnis führen, dass ein Pfad nicht ausführbar ist. In diesem Fall muss eine erneute Testpfadgenerierung durchgeführt werden.

Die Vernachlässigung der Pfadbedingungen während der graphentheoretischen Testfallgenerierung ist als wesentlicher Nachteil anzusehen. Die im folgenden Abschnitt beschriebene Methode der Testfallgenerierung mittels *Model Checking* betrachtet die Struktur und die Bedingungen der Pfade und gewinnt sogar durch den Einsatz von Bedingungen an Effizienz.

9.3 Testfallermittlung mit UPPAAL

Die im Vorangegangenen dargestellte Ableitung eines flachen, endlichen Zustandsautomaten aus einem ROOMchart eignet sich ebenfalls zur Generierung eines Systemmodells für einen *Model Checker* [117]. Der *Model Checker Uppaal* [11] basiert auf zeitattributierten, endlichen Zustandsautomaten [4] und erlaubt die statische Analyse mittels einer einfachen Pfadprädikatenlogik. Um die Analyse und Testpfadgenerierung mit Uppaal zu ermöglichen, muss das Systemmodell um eine Systemumgebung und verschiedene Instrumentierungen erweitert werden.

9.3.1 Systemumgebung und Instrumentierung des Systemautomaten

Die Systemumgebung zur Testpfadgenerierung mit Uppaal ist vergleichbar mit einem Testtreiber in der dynamischen Softwareprüfung. Das Systemmodell in Uppaal wird um einen Automaten erweitert, der die notwendigen Transitionen besitzt, um mit allen Transitionen im Systemautomaten zu synchronisieren. Weiterhin kann die Systemumgebung optionale Transitionen zur Wertzuweisung der Systemvariablen enthalten. Für die Generierung der Testpfade ist eine Instrumentierung des Systemautomaten notwendig. Die Instrumentierung ist charakteristisch für das verwendete Überdeckungskriterium und dient als Basis für die Formulierung der Pfadbedingungen in Uppaal-

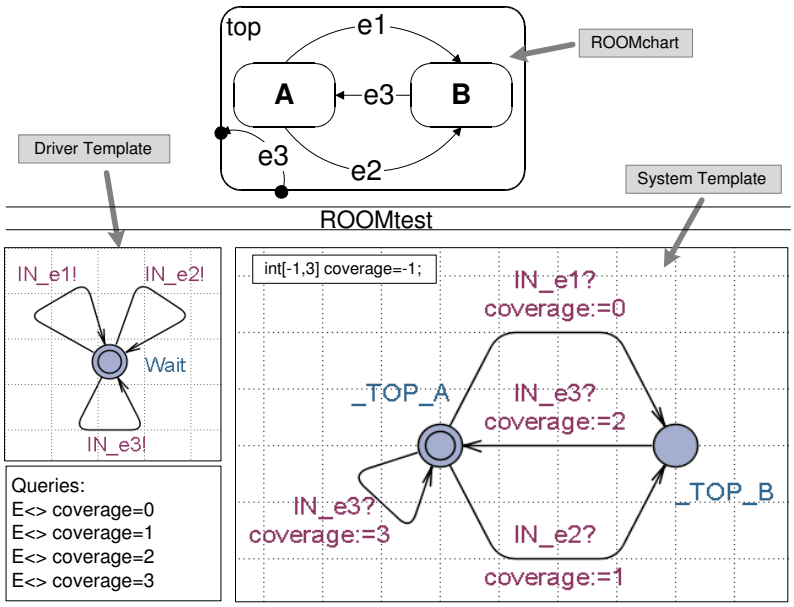


Abbildung 9.7: Testfallgenerierung Transitionsüberdeckung mit Uppaal

Logik. Für den Test von ROOMcharts werden in dieser Arbeit Transitionsüberdeckung, Bedingungsüberdeckung und Datenflussüberdeckung verwendet. Bedingungsüberdeckung wird mittels Transitionsüberdeckung verschiedener Konfigurationen des Systemautomaten erzeugt. Daher wird im Folgenden ausschließlich die Instrumentierung für Transitionsüberdeckung und die Verwendung von Datenflussüberdeckungskriterien dargestellt.

Transitionsüberdeckung

Das Beispiel in Abbildung 9.7 demonstriert die Ableitung eines Testmodells für Uppaal aus einem ROOMchart. Aus dem ROOMchart oberhalb des Trennbalkens wird das untere Modell in Uppaal abgeleitet. Das Modell besteht aus dem Systemautomaten (*System Template*), dem Treiberautomaten (*Driver Template*) und einer Anzahl Anforderungen in Uppaal-Logik. Der Systemautomat ist um die Variable *coverage* erweitert, die zur Generierung der Transitionsüberdeckung dient. Der Treiberautomat besitzt den Zustand *Wait* und eine Anzahl reflexiver Transitionen, die mit den Transitionen des Systemautomaten über die Kanäle *IN_e1*, *IN_e2* und *IN_e3* synchronisieren. Die Anforderungsdefinition enthält für jede Transition im Systemmodell eine Bedingung, welche die mögliche Existenz eines Pfades formuliert, auf dem die Variable *coverage* die Werte 0, 1, 2 oder 3 annimmt. Da die Transitionen im Systemautomaten mit diesen Werten eindeutig kodiert sind, führen die Anfragen zur Generierung von drei Pfaden, die zusammen Transitionsüberdeckung bieten. Die Transitionsüberdeckung kann auch auf verschiedene andere Weisen formuliert werden. In [103] werden verschiedene Überdeckungskriterien für die Testfallgenerierung mit Uppaal präsentiert. Das in dieser Arbeit präsentierte Verfahren zur Generierung der Transitionsüberdeckung ist effizienter als die Methode von Huhn und Mücke, da sie die Anzahl zusätzlicher Instrumentierungsvariablen minimiert. Dies wird an anderer Stelle in dieser Arbeit ausführlich erläutert und daher hier nicht weiter ausgeführt.

Parametrisierte Ereignisse

Ein parametrisiertes Ereignis ist in *Real-Time Object-Oriented Modeling* mit einer Datenmenge beliebigen Typs assoziiert. In Uppaal können Synchronisationen mittels einer Erweiterung mit globalen Daten assoziiert werden.

In Abbildung 9.8 ist ein beispielhaftes ROOMchart mit parametrisierten Ereignissen dargestellt. Durch die interne Gruppentransition ist die Definition der Variablen x in jedem Zustand möglich.

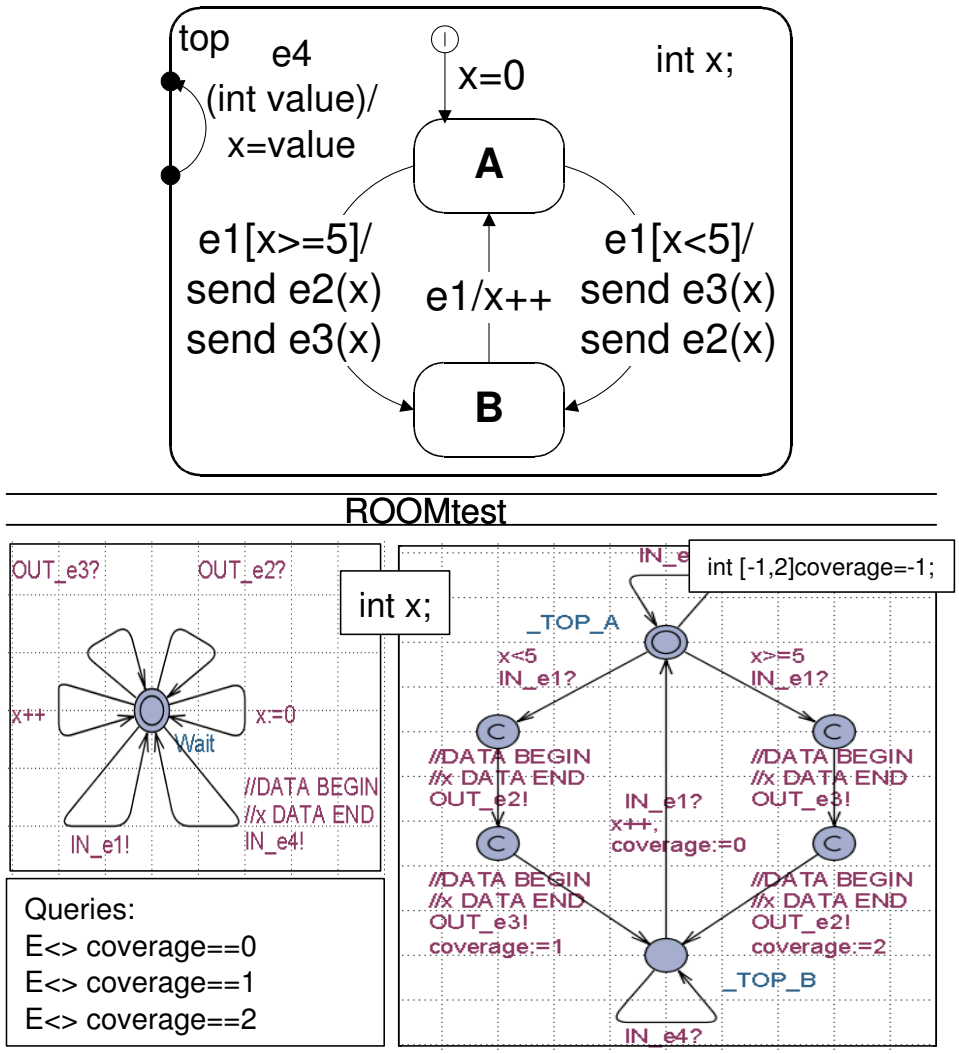


Abbildung 9.8: Parametrisierte Ereignisse

Die beiden reflexiven Transitionen an den Zuständen $\_TOP\_A$ und $\_TOP\_B$ im Testmodell sind von der reflexiven Gruppentransition abgeleitet. Um die Generierung parametrisierter Nachrichten im Test zu ermöglichen, muss der Treiberautomat um zwei reflexive Transitionen zur Wertzuweisung von x erweitert werden. Eine Transition setzt x gleich *0*, und eine Transition inkrementiert x. Dadurch kann x außerhalb des Systemautomatens beliebig definiert werden. Diese Konstruktion ermöglicht dem Suchalgorithmus eine Wertebelegung zu finden, unter der die Transitionen mit Wächterbedingungen ausgelöst werden können.

Gedächtnisfunktion Die Gedächtnisfunktion eines hierarchischen Zustands schützt dessen interne Verarbeitung während externer Verarbeitung [143, 64]. Die Ableitung eines Uppaal Testmodells wurde bereits an anderer Stelle in dieser Arbeit demonstriert. Die Testfallgenerierung erfolgt auf Basis beliebiger Überdeckungskriterien - hier der Transitionsüberdeckung. Das Beispiel in Abbildung 9.9 stellt die Ableitung eines Uppaal-Systems zur Testfallgenerierung aus einem ROOMchart mit Gedächtnisfunktion dar, dass für Transitionsüberdeckung instrumentiert ist.

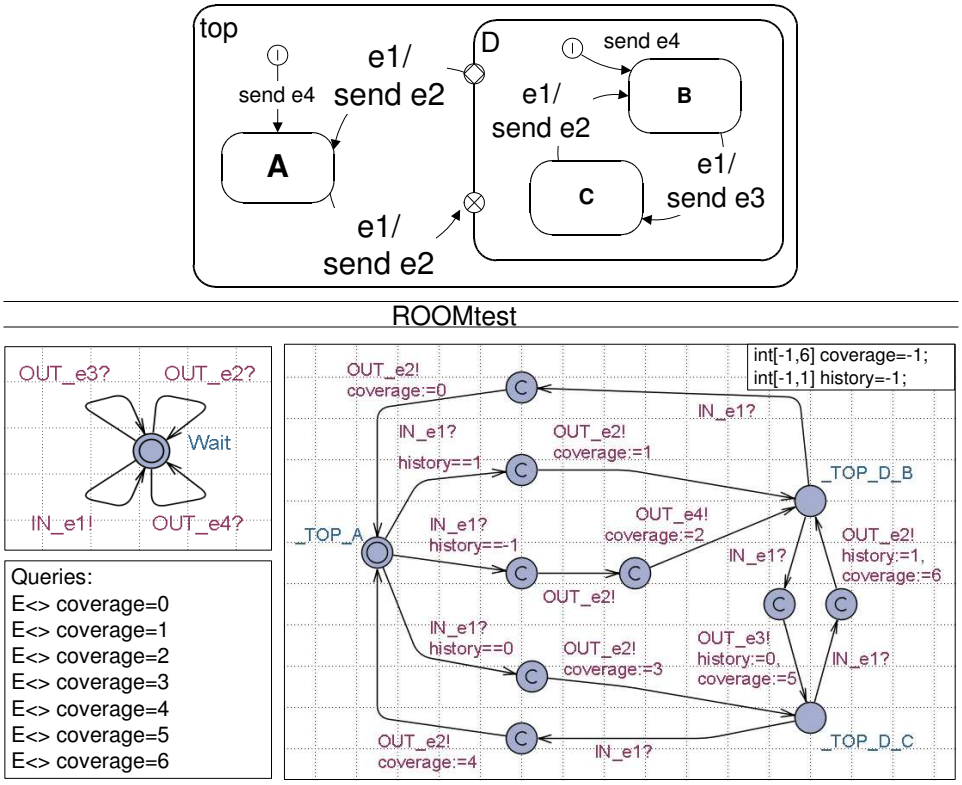


Abbildung 9.9: Gedächtnisfunktionen

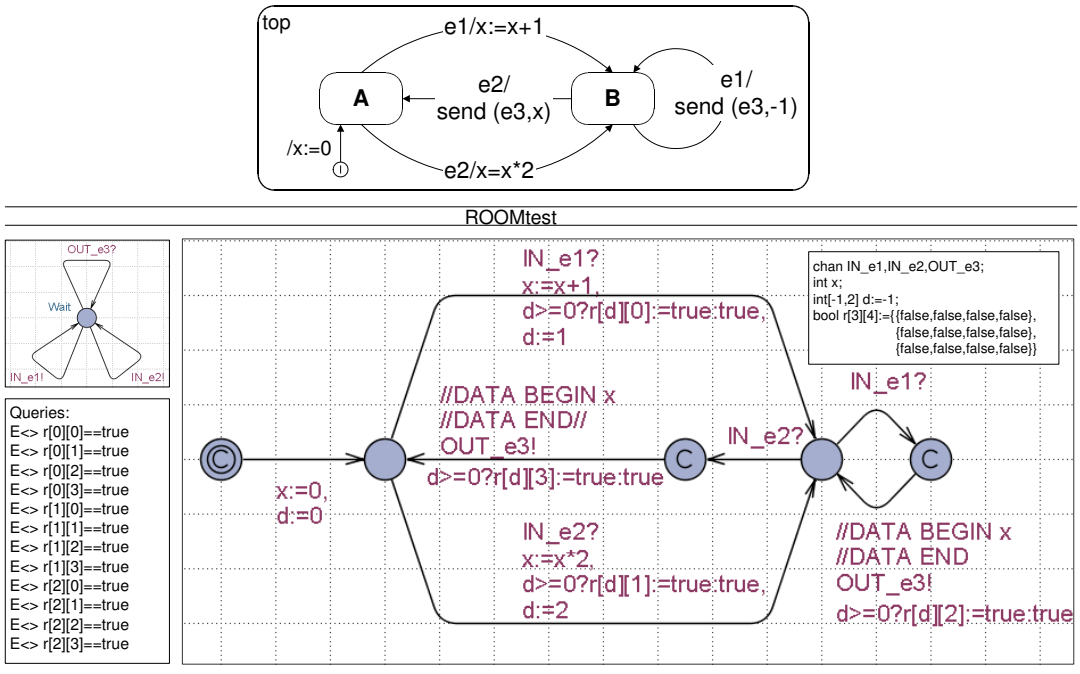


Abbildung 9.10: *all uses* mit Uppaal

Datenflussbasierte Verfahren

Die Testfallgenerierung für datenflussbasierte Verfahren erfolgt auf einem Uppaal-Testmodell nach einer Methode, die in [103] vorgestellt wurde und an ROOMcharts in Normalform angepasst wurde. Die Komplexität dieses Verfahrens wird analysiert und ein alternatives Verfahren wird vorgeschlagen. Im Folgenden wird die Instrumentierung eines Systemautomaten für *all uses* erläutert. Die Methode ist unverändert auf *all c-uses* und *all p-uses* übertragbar.

Die Menge aller Definitionen einer Variablen x in einem Automaten wird mit D_x bezeichnet. Die Menge aller Benutzungen einer Variablen x in einem Automaten wird mit R_x bezeichnet. Für die Definitionen $def_i \in D_x$ wird eine ganzzahlige Variable $d \in [-1, (|D_x| - 1)]$ definiert. Die Variable d wird mit -1 initialisiert und jeder Wert $i>=0$ von d ist für genau ein def_i reserviert. Bei jeder Ausführung von def_i wird $d = i$ definiert.

Für jede Definition einer Variablen x wird ein boolesches Feld mit den von den Definitionen def_i erreichbaren Benutzungen $r_{i,j} \in R_{x,i} \subseteq R_x$

$$\mathbf{r_x} = \begin{vmatrix} r_{0,0} & r_{1,0} & \cdots & r_{|D_x|-1,0} \\ r_{0,1} & r_{1,1} & \cdots & r_{|D_x|-1,1} \\ \cdots & \cdots & \cdots & \cdots \\ r_{0,|R_{x,0}|} & r_{1,|R_{x,1}|} & \cdots & r_{|D_x|-1,|R_{x,|D_x|-1}|} \end{vmatrix}$$

definiert, dessen Elemente mit *Falsch* initialisiert werden. Bei jeder Ausführung einer Variablenbenutzung $r_j \in R_x$ wird das Element $r_{i,j} \in \mathbf{r_x}$ mit *Wahr* definiert. Das *all uses* Kriterium ist erfüllt, wenn alle Elemente aller Felder $\mathbf{r}$ mit Wahr definiert sind. Die Komplexität dieser Methode wird durch die Anzahl zusätzlich benötigter Instrumentierungsvariablen bestimmt. Es werden eine ganzzahlige Variable d mit dem Wertebereich $dom(d) = |D_x|$ und ein boolesches Feld der maximalen Größe $l_v = |D_x| * |R_x|$ benötigt. Jedes Element des Feldes geht in die Komplexität wie eine boolesche Variable ein, daher ist $l_v + 1$ die maximale Anzahl zusätzlicher Instrumentierungsvariablen. An anderer Stelle in dieser Arbeit wird die Komplexität der Testfallgenerierung mit Uppaal detaillierter betrachtet.

In dem Beispiel in Abbildung 9.10 ist die Testfallgenerierung für *all uses* mit Uppaal dargestellt. Das ROOMchart im oberen Teil der Abbildung enthält die drei Definitionen und vier Benutzungen der Variablen *x*. Somit werden für die Instrumentierung für *all uses* eine Variable d mit dem Wertebereich [-1,2] und ein zweidimensionales, boolesches Feld r[4][3] benötigt. Nach jeder Definition wird der Variablen d ein Index i zugewiesen. Nach jeder Variablenbenutzung mit dem Index j wird für den Fall, dass bereits eine Definition der Variablen vorgenommen wurde, das Feld an der Stelle *(i,j)* mit *true* definiert. Die Anfragen fordern die Existenz jeweils eines Pfades für jedes Element $r_{i,j} \in \mathbf{r}$, für das $r_{i,j} = true$ gilt.

9.3.2 Betrachtung der Zeitkomplexität

Die Explosion des Zustandsraums ist in allen Anwendungsbereichen des *Model Checking* ein zentrales Problem. Um *Model Checking* für die Praxis nutzbar zu machen, ist eine Komplexitätsbetrachtung unumgänglich. Jedes System besitzt eine Basiskomplexität, die unabhängig von der Analysemethode nicht unterschritten werden kann. Die Basiskomplexität eines Testmodells zur Erfüllung eines bestimmten Überdeckungskriteriums wird auf Basis des instrumentierten Systemautomaten mit Treiberautomaten festgelegt.

Für die Diskussion der Zeitkomplexität werden im Folgenden verschiedene Maße in Landau-Notation verwendet, deren Verläufe in Abbildung 9.11 graphisch dargestellt sind.

Eine besonders häufig verwendete Komplexitätsabschätzung für *Model Checking* Probleme legt die Anzahl der Zustände des zu prüfenden Systems zugrunde, die sich aus dem Produkt der Dimensionen der Wertebereiche aller Variablen und der Anzahl Zustände jedes Automaten im System ermittelt. Demnach kann ein *Model Checking* Problem innerhalb folgender Zeit entschieden werden:

$$O_s(\textstyle\prod_i vardim_i * \prod_j locations_j)$$

Mit $vardim_j$ wird die Dimension des Wertebereichs der Variablen var_i bezeichnet. Mit $locations_j$ wird die Anzahl der semantischen Zustände im Automaten m_j bezeichnet.

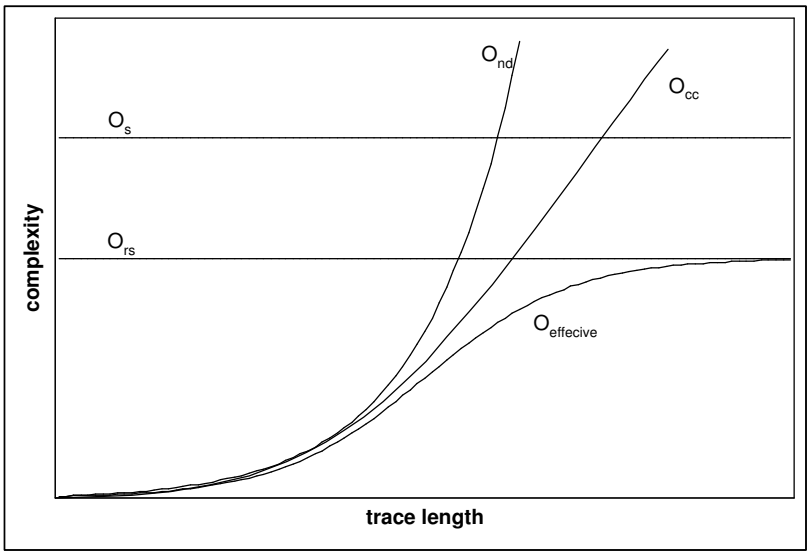


Abbildung 9.11: Komplexität Wächterbedingungen

Da nicht alle Zustände im Lösungsraum erreichbar sind, ist die Abschätzung der Zeitkomplexität über die Anzahl erreichbarer Zustände O_{rs} ein theoretisch genauerer Ansatz. Die Komplexität O_{rs} ist eine rein theoretische Größe, für die aber $O_{rs} \subseteq O_s$ bekannt ist. Die Komplexität O_{rs} dient im Folgenden ausschließlich als Argumentationshilfe.

Für spezielle Probleme, die innerhalb eines verhältnismäßig kurzen Pfades mittels Breitensuche entschieden werden, haben Huhn und Mücke [103] das theoretische Komplexitätsmaß O_{nd} vorgestellt. Dieses Maß ist für die Testfallgenerierung besonders zweckmäßig, da hier oftmals nur ein kleiner Teil des Zustandsraums durchsucht werden muss. Demnach kann ein Testproblem mittels Breitensuche und CTL *Model Checking* innerhalb folgender Zeit entschieden werden:

$$O_{nd}(non\_determinism^{trace\_length})$$

Diese Zeitschranke stellt für die Testfallgenerierung einen deutlich genaueren Ansatz als die Abschätzung über die Gesamtzahl der Zustände dar. Die Länge des ermittelten Testpfades (*engl. trace length*) ist gleich der Anzahl benötigter Suchschritte n. Der nichtdeterministische Freiheitsgrad ist konstant über alle Suchschritte und entspricht der Größe des Eingabealphabets.

Eine ebenfalls theoretische Zeitschranke kann angegeben werden, wenn berücksichtigt wird, dass der Freiheitsgrad über die Suchschritte variiert. Ein Testproblem lässt sich mittels CTL *Model Checking* und Breitensuche innerhalb folgender Zeit entscheiden:

$$O_{succ}(\sum_{i=0}^{trace\_length} succeding\_states_i)$$

Die Zeitkomplexität O_{succ} berücksichtigt, dass nicht in jedem Suchschritt der maximale nichtdeterministische Freiheitsgrad vorhanden ist. Vielmehr bestehen strukturelle Einschränkungen, beispielsweise Wächterbedingungen und nicht spezifizierte Ereignisse, welche die Auswahlmöglichkeiten reduzieren. Die Einschränkung des Freiheitsgrads während eines Suchschrittes vermindert die Anzahl der aus einem Zustand erreichbaren Folgezustände (*engl.* ***succ****eding states*) und es gilt $O_{cc} \subseteq O_{nd}$.

Die Zeitschranken O_s und O_{rs} sind genau dann relevant, wenn der gesuchte Testpfad nicht existiert. In diesem Fall wird der gesamte erreichbare Zustandsraum durchsucht und es gilt $O_{eff} = O_{rs}$. Im schlechtesten Fall sind alle Zustände im Lösungsraum erreichbar und es gilt $O_{rs} = O_s$. Wenn der gesuchte Pfad existiert, wird er innerhalb von O_{cc} bzw. O_{nd} gefunden.

Für das System in Abbildung 9.7 ist in Abbildung 9.12 ein beispielhafter Suchbaum nach Breitensuche dargestellt. In Abbildung 9.12 sind nicht erreichbare Zustände enthalten, die grau dargestellt sind.

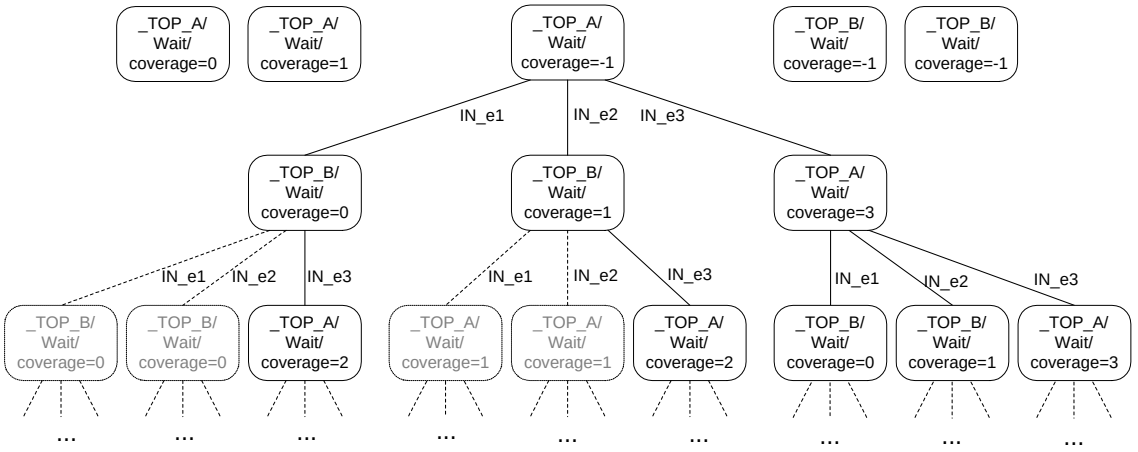


Abbildung 9.12: Suchbaum

Die folgenden Beispiele zeigen die Auswirkungen der Verwendung von Wächterbedingungen, parametrisierten Ereignissen und Gedächtnisfunktionen auf die Komplexität der Testfallgenerierung mit Uppaal und Breitensuche.

Wächterbedingungen Die Verwendung von Wächterbedingungen schränkt die Erreichbarkeit von Zuständen ein. Der Suchbaum wird durch diese Maßnahme verkleinert und die Anzahl nicht erreichbarer Zustände nimmt zu. Daher verringert sich die nach O_{rs} und O_{cc} benötigte Rechenzeit bei Verwendung von Breitensuche und CTL *Model Checking*. Da möglicherweise der maximale nichtdeterministische Freiheitsgrad des Systems verringert wird, nimmt nach O_{nd} auch die benötigte Rechenzeit ab. Die Gesamtzahl der Zustände und die damit verbundene Komplexität O_s bleiben unverändert.

Das Beispiel in Abbildung 9.13 stellt die Ableitung eines Uppaal-Systems zur Testfallgenerierung aus einem ROOMchart mit Wächterbedingungen dar. Der nichtdeterministische Freiheitsgrad des Systems ist 1, da in beiden Zuständen nur eine Transition ausgelöst werden kann. Im Gegensatz zu einem analytischen Lösungsverfahren, das die Lösung der Pfadbedingungen notwendig machen würde und demnach PSPACE-vollständig ist [116], wirken sich die Pfadbedingungen bei Verwendung von Breitensuche und CTL *Model Checking* zur Lösung des Testproblems positiv aus.

Gedächtnisfunktion Die Gedächtnisfunktion eines hierarchischen Zustands s_c in *Realtime Object-Oriented Modeling* schützt die interne Verarbeitung von s_c während externer Verarbeitung [143, 64].Die zusätzliche Variable *history* erhöht nach O_s den Zeitbedarf, da zusätzliche Variablen linear die Anzahl Zustände im Lösungsraum erhöhen. Die Anzahl der erreichbaren Zustände wird durch die zusätzliche Variable ebenfalls linear erhöht. Da allerdings eine zusätzliche Variable nicht den nichtdeterministischen Freiheitsgrad oder die Pfadlänge beeinflusst, bleibt der Zeitbedarf nach O_{nd} und O_{cc} unverändert.

Parametrisierte Ereignisse Die notwendigen Modifikationen zur Ermittlung von Ereignisparametern beeinflussen nach O_s die Rechenzeit nicht, da die Größe des Zustandsraumes unverändert bleibt. Dagegen wird nach O_{rs} die benötigte Rechenzeit erhöht, da die Anzahl erreichbarer Zustände zunimmt. Weiterhin wird durch die zusätzlichen Transitionen im System- und Treiberautomaten der nichtdeterministische Freiheitsgrad erhöht. Die Testpfadlänge kann allerdings durch die zusätzlichen Transitionen verringert werden. Im Beispiel in Abbildung 9.10 wird der Freiheitsgrad von 1 auf 4 erhöht, die Pfadlänge wird für *coverage*=2 um 5 verringert. Da die Pfadlänge größeren Einfluss auf die Zeitkomplexität hat, wird sich O_{nd} bei großen oder strukturell komplexen Systemen tendentiell eher verringern. Eine Erhöhung der Komplexität oder die Kompensation beider Einflüsse kann nicht ausgeschlossen werden. Durch Datenabstraktion [118] kann der erhöhte Zeitbedarf

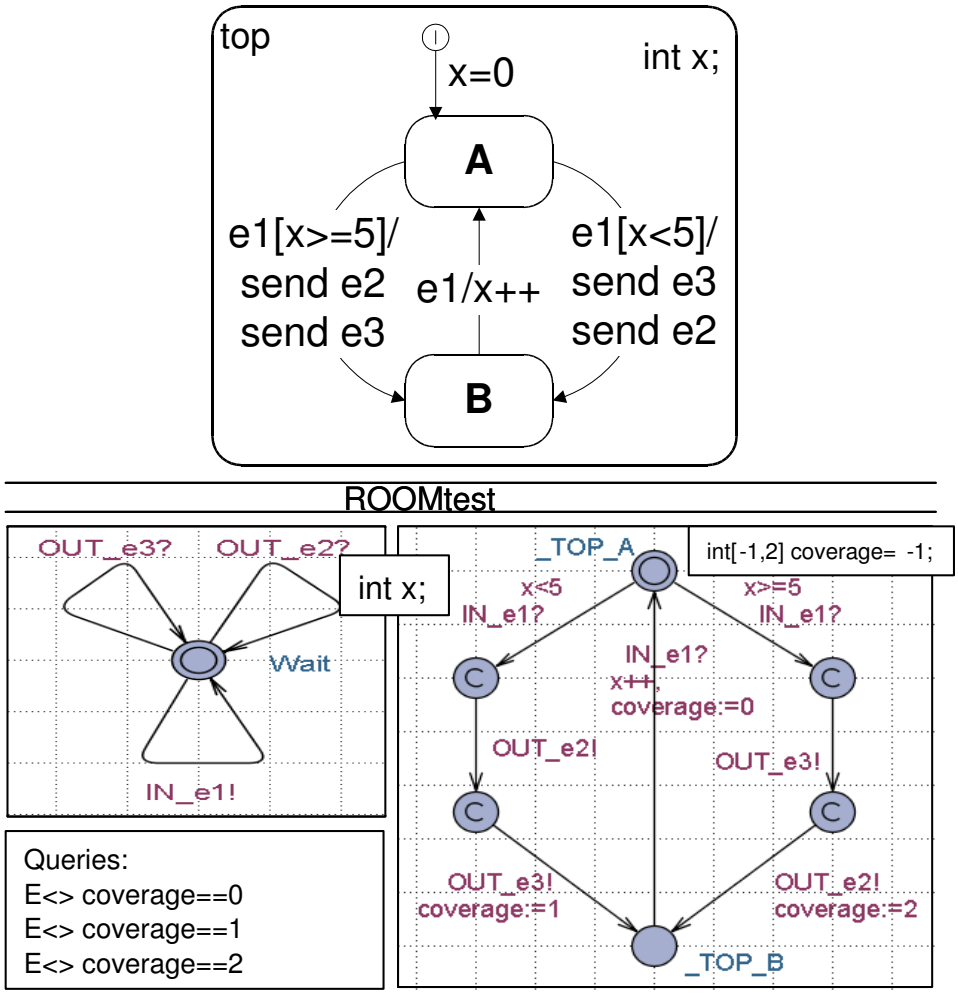


Abbildung 9.13: Komplexität von Wächterbedingungen

vermindert, aber nicht vollständig kompensiert werden. Verschiedene Techniken zur Komplexitätsreduktion werden im Anschluss diskutiert.

Vollständigkeit Die Vollständigkeit einer Spezifikation wird von verschiedenen automatenbasierten Testverfahren gefordert [18, 33, 37, 51]. Die Anwendung der *Completeness Assumption* [18] verändert nicht die Anzahl erreichbarer oder nicht erreichbarer Zustände. Daher bleibt derer Zeitbedarf nach O_s und O_{rs} unverändert. Die Pfadlänge wird nicht verändert. Da die zusätzlichen Transition den nichtdeterministischen Freiheitsgrad erhöhen können, wird sich der Zeitbedarf nach O_{nd} möglicherweise erhöhen. Dies ist insbesondere dann der Fall, wenn das System in allen Zuständen unvollständig spezifiziert war. Nach O_{cc} wird der Zeitbedarf erhöht, wenn mindestens ein Zustand unvollständig spezifiziert war.

9.3.3 Betrachtung der Raumkomplexität

Die Raumkomplexität wird durch die Repräsentation der Zustände im Zustands-Transitions-System bestimmt. Daher erhöhen zusätzliche Variablen und Zustände den Raumbedarf linear. Die Generierung von Testfällen zur Strukturüberdeckung mit einem *Model Checker* macht in der Regel zusätzliche Instrumentierungsvariablen notwendig, die den Speicherbedarf linear erhöhen.

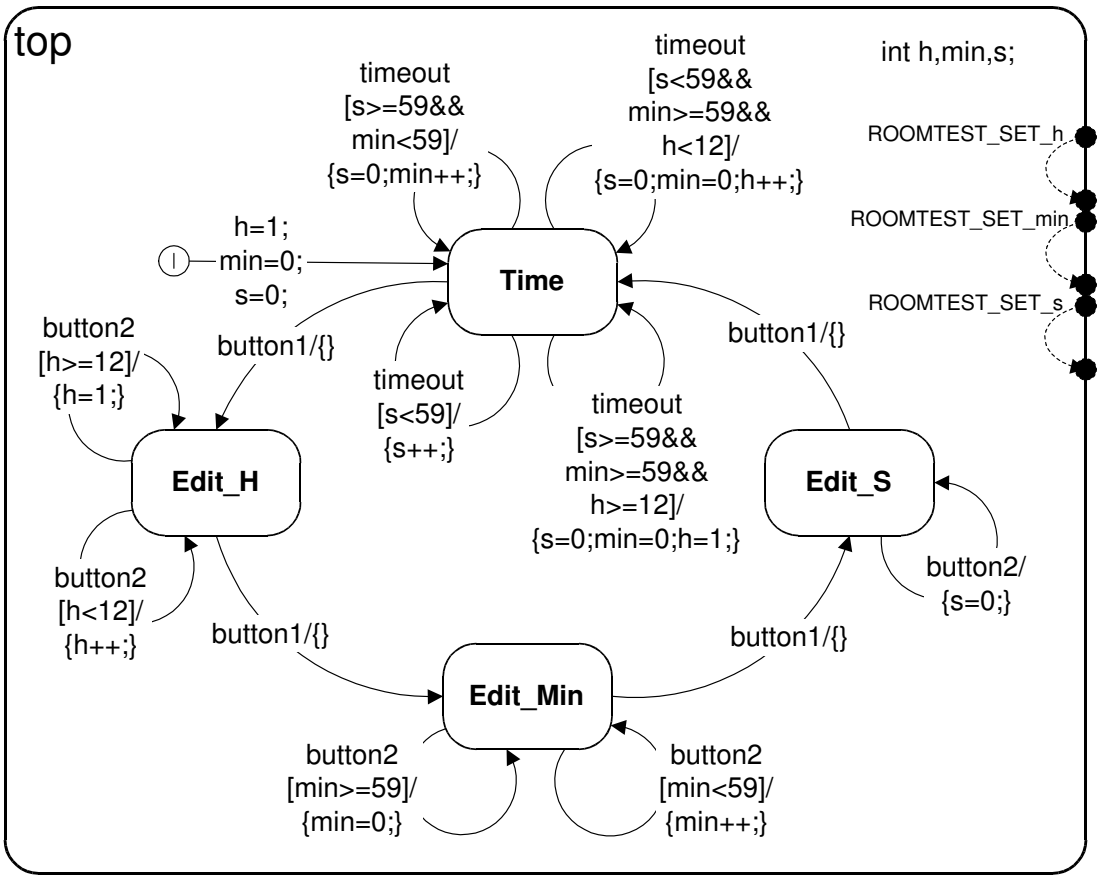


Abbildung 9.14: ROOMchart *Clock Controller*

Wächterbedingungen

Die Definition von Wächterbedingungen oder Auswahlpunkten hat keinen Einfluss auf den Raumbedarf.

Parametrisierte Ereignisse

Die Ermittlung von Ereignisparametern hat keinen Einfluss auf den Raumbedarf.

Gedächtnisfunktion

Für die Generierung einer Testfallmenge für Transitionsüberdeckung mittels *Model Checking* macht eine zusätzliche Variable für jeden hierarchischen Zustand mit Gedächtnisfunktionen notwendig, die den Speicherbedarf linear erhöhen.

Vollständigkeit

Die Vervollständigung eines Systemmodells hat keinen Einfluss auf den Raumbedarf.

9.3.4 Experimentelle Anwendung

Die in dieser Arbeit präsentierte Methode zur Testfallgenerierung mit Uppaal wurde in dem integrierten Werkzeug *ROOMtest für Rational Rose Realtime* realisiert. In dem Beispiel in Abbildung 9.14 ist das ROOMchart der Steuerung einer Digitaluhr mit Editierungsfunktion gegeben. Das Modell wurde mit *Rational Rose Realtime* realisiert und ein Uppaal-Testmodell wurde mit *ROOMtest* erzeugt.

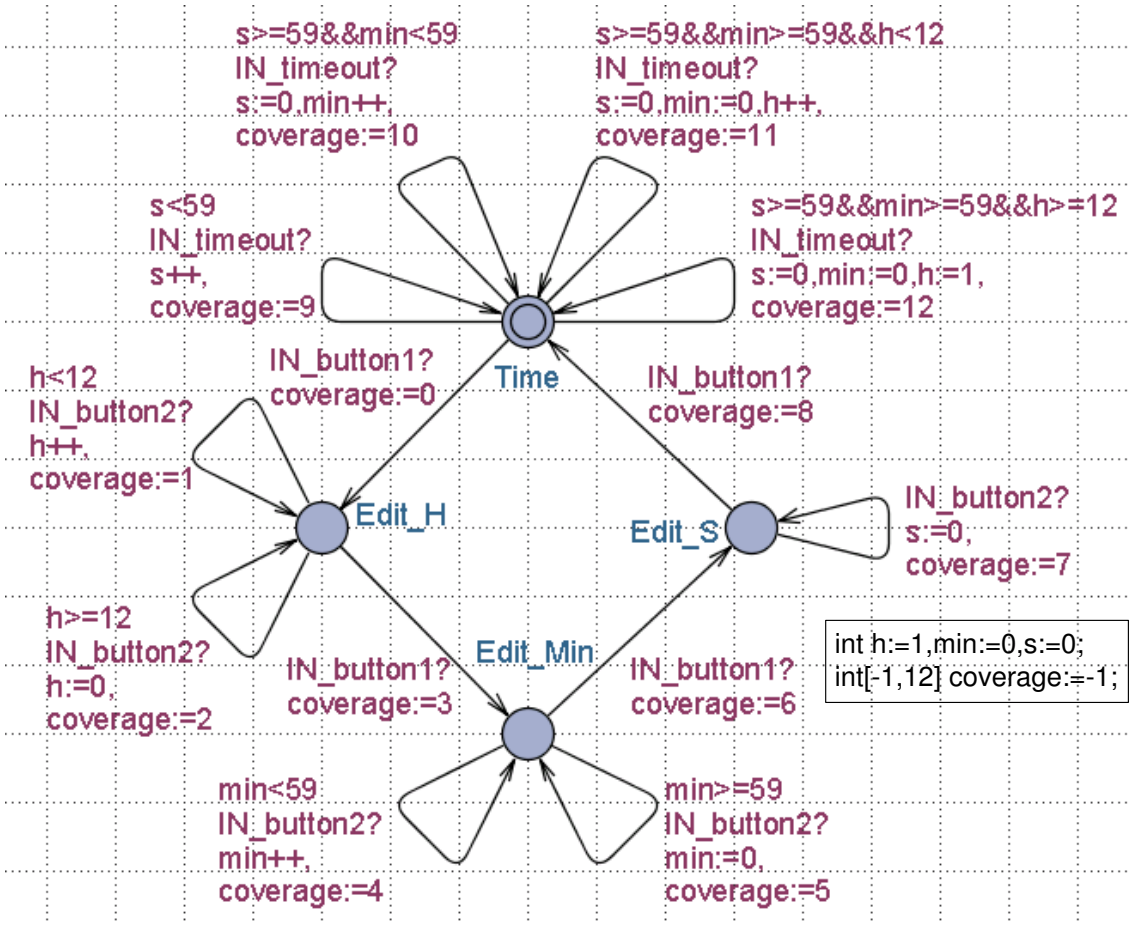


Abbildung 9.15: UPPAAL Systemautomat *Clock* o. Instrumentierung

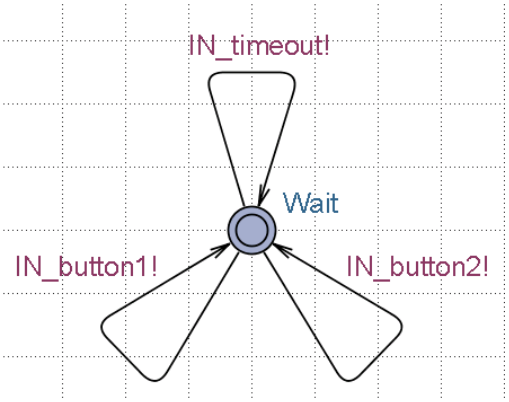


Abbildung 9.16: UPPAAL Treiberautomat *Clock Controller*

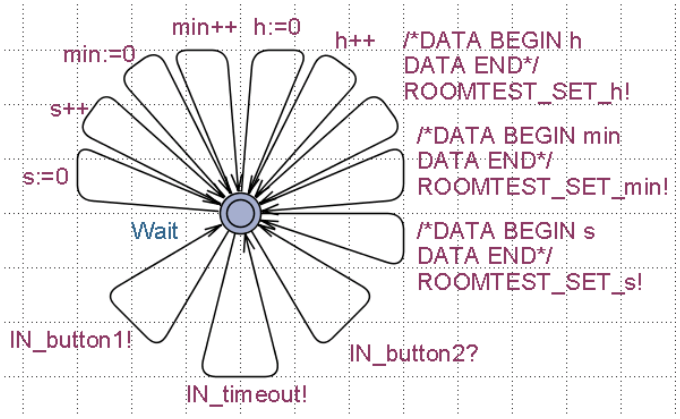


Abbildung 9.17: UPPAAL Treiberautomat Clock Controllermit Parameterbestimmung

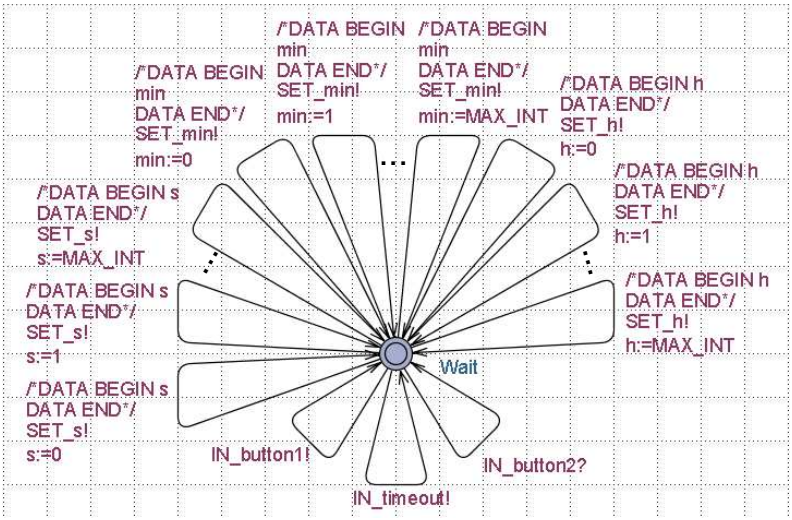


Abbildung 9.18: Alternativer UPPAAL Treiberautomat *Clock Controller* mit separierter Parameterbestimmung

Nach der Initialisierung befindet sich das Uhrwerk im Zustand *Time* und erwartet die Ereignisse *timeout* oder *button1*. Ein *timeout* wird nach dem 12 Stunden Schema verarbeitet, das durch die vier reflexiven Transitionen im Zustand *Time* realisiert ist. Die beiden Ereignisse *button1* und *button2* repräsentieren das Drücken von zwei Knöpfen. Der erste Knopf ändert den Editierungsmodus, repräsentiert durch *Edit_H*, *Edit_Min* und *Edit_S*. Die ganzzahligen Variablen *h*, *min* und *s* werden durch den zweiten Knopf im jeweiligen Modus editiert. Die Stunden und Minuten werden inkrementiert und die Sekunden werden zu 0 gesetzt. Die gestrichelt dargestellten internen Gruppentransitionen können zu Testzwecken automatisch generiert werden.

Testfallgenerierung

Das Testmodell in Abbildung 9.15 hat die gleiche Kontrollstruktur wie das ROOMchart in Abbildung 9.14. Der Systemautomat ist um die ganzzahlige Variable *coverage* mit dem Wertebereich [-1,12] erweitert. Die zusätzliche Variable erhöht linear den Speicherbedarf. Da infolge der unvollständigen Spezifikation der nichtdeterministische Freiheitsgrad konstant ist, wird bei der folgenden Komplexitätsbetrachtung die Abschätzung über Folgezustände nicht angewendet, da $O_{nd} = O_{cc}$ gilt. Im Folgenden werden verschiedene, alternative Treiberautomaten für die Lösung des *Model Checking* Problems der Generierung einer Transitionsüberdeckung für das Beispiel in Abbildung

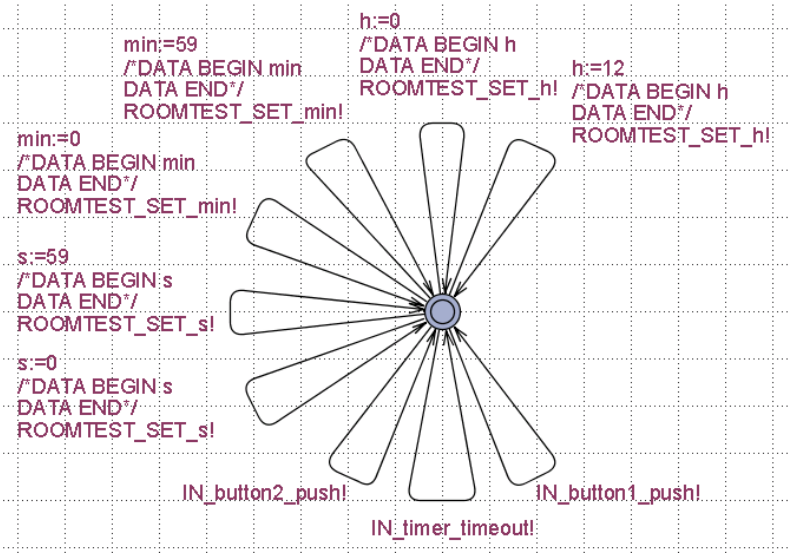


Abbildung 9.19: Abstrakter UPPAAL Treiberautomat *Clock Controller* mit Parameterbestimmung

9.14 vorgestellt und deren Auswirkung auf die Komplexität diskutiert.

Der Treiberautomat in Abbildung 9.16 besitzt drei reflexive Transitionen mit denen die Transitionen im Systemautomaten in Abbildung 9.15 synchronisieren. Für die Generierung eines Testpfads für *coverage*=12 wird der Suchalgorithmus 1+12+59+1+1+60=134 Schritte benötigen. Die effektive Komplexität wird für dieses Beispiel nicht weniger als 2^{134} betragen. Von den

65536*65536*65536*4=1125899906842624

Zuständen sind 12*59*59*4=167088 Zustände erreichbar.

Der alternative Treiberautomat in Abbildung 9.17 enthält reflexive Transitionen, um die Werte der globalen Variablen *h*, *min* und *s* zu ändern und parametrisierte Eingabeereignisse zu generieren. Diese Konstruktion erhöht die Ausführbarkeit der Transitionen im Systemmodell, beispielsweise kann die Transition für coverage=12 in 12+59+59+1=131 Schritten ermittelt werden. Weiterhin können die zusätzlichen Transitionen im Testtreiber in jedem Suchschritt ausgelöst werden und erhöhen den nichtdeterministischen Freiheitsgrad von 2 auf 11. In diesem Beispiel ist der Einfluss der Erhöhung des nichtdeterministischen Freiheitsgrads deutlich höher als die Ersparnis durch einen kürzeren Testpfad, obwohl der Testpfad deutlich größeren Einfluss auf die Zeitkomplexität besitzt.

Ein automatisierbarer Ansatz zur Komplexitätsreduktion im Treiber ist die Verwendung von jeweils einer Transition für jeden Variablenwert der erweiterten Zustandsvariablen. Der Treiberautomat in Abbildung 9.18 enthält für jeden möglichen Wert der Variablen *h*, *min* und *s* eine reflexive Transition zum Setzen und Senden parametrisierter Ereignisse. Jede dieser Transitionen weist jeweils einer Variablen einen Wert zu und synchronisiert. Durch diese Konstruktion wird die notwendige Pfadlänge von 131 auf 1+1+1+1=4 deutlich reduziert. Der nichtdeterministische Freiheitsgrad wird dagegen von 11 auf maximal 65536+65536+65536+2=196612 erhöht. Ohne Datenabstraktion ist dieser Ansatz nicht sinnvoll, besitzt aber das Potential einer signifikanten Komplexitätsreduktion, wenn der nichtdeterministische Freiheitsgrad reduziert werden kann.

Eine deutliche Komplexitätsreduktion kann durch die Einschränkung der Wertebereiche der Variablen *h*, *min* und *s* erzielt werden. Der Treiberautomat in Abbildung 9.19 enthält Transitionen, welche die relevanten Werte (0,12,59) der Variablen *h*, *min* und *s* setzen und senden können. Weiterhin wird der Wertebereich jeder einzelnen Variablen zu *h*[1,12], *min*[0,59], und *s*[0,59] reduziert. Die notwendige Anzahl Schritte für die Ermittlung eines Pfades für die *coverage*=12 ist auch hier 4. Der nichtdeterministische Freiheitsgrad konnte durch die manuelle Datenabstraktion allerdings auf 9 reduziert werden.

Durch den alternativen Treiberautomaten in Abbildung 9.19 ist das Beispiel in Abbildung 9.15 innerhalb der Berechenbarkeitsgrenze, die nach Erfahrungen mit dem SMV *Model Checker* bei 10^{120} liegt. Die Raumkomplexität wird nur durch die Variable *coverage* linear um den Faktor 13 erhöht.

DS Generierung

Die ermittelten Eingabeereignisse für Transitionsüberdeckung des Beispiels in Abbildung 9.14 sind in den Tabellen C.3 und C.4 dargestellt. In Tabelle C.3 sind die Eingabeereignisse für das Beispiel ohne Instrumentierung veranschaulicht. Für diese Testeingaben müssen für einen Test ohne Beobachtbarkeit der Modulinterna (engl. *Black Box Test*) die *Distinguishing Sequences* für die Zustände mit Variablenwerten ermittelt werden. Im Anhang in den Abbildungen C.2, C.1, C.3, C.4, C.5, C.6, C.7, C.8, C.9, C.10 und C.11 ist das vollständige DS-Modell für das *Clock*-Beispiel dargestellt. In Tabelle C.4 sind die Testfälle unter Verwendung der Instrumentierung zum Setzen der erweiterten Zustandsvariablen dargestellt. Für diese Testfälle können theoretisch ebenfalls *Distinguishing Sequences* ermittelt werden. Da allerdings angenommen werden kann, dass die vollständige Beobachtbarkeit der Modulinterna durch weitere Instrumentierung hergestellt werden kann, wird dies nicht weiter erläutert. Die Testfälle in Abbildung C.4 demonstrieren aber sehr gut die Verkürzung der Testfälle infolge der Instrumentierung. In Tabelle C.5 sind die *Distinguishing Sequences* für die Zustände und Variablenwerte der Testfälle in Tabelle C.3 dargestellt. Da die *Distinguishing Sequences* auch Transitionen ohne Ausgaben enthalten, muss ein zeitliches Raster für deren Anwendung vorhanden sein. Ohne zeitliches Raster würde die Eingabesequenz in jedem Zustand nur die Ausgabe *time* generieren, ohne die Unterscheidbarkeit der Zustände zu gewährleisten. Ein zeitliches Raster kann pragmatisch durch die längste, zu erwartende Antwortzeit der Eingabeereignisse generiert werden. In dem vorliegenden Fall ist ein Raster mit einer Schrittweite von einer Sekunde passend, da innerhalb dieser Zeit die reflexiven Transitionen in *Time* abgeschlossen sein müssen, um die rechtzeitige Verarbeitung des nächsten *timeout*-Signals zu gewährleisten. In den anderen Zuständen liegen dagegen keine zeitlichen Zwänge vor. Hier sollte die Reaktionszeit auf ein *ButtonX*-Signal schnell genug erfolgen, um eine komfortable Bedienung zu ermöglichen. Für das Beispiel *Clock* sind die *Distinguishing Sequences* nicht von den Variablenwerten abhängig. Trotzdem demonstriert das Beispiel in ausreichender Weise die notwendige Vorgehensweise. Die Verkettung der transitionsüberdeckenden Eingaben mit den *Distinguishing Sequences* liefert die Testsequenzen in den Tabellen C.1 und C.2. Mit einem Zeitraster von $1s$-Intervallen benötigt die Testsequenz 495 Sekunden. Da dieser Test vollautomatisch erfolgen kann, spielt die Ausführungsdauer nur eine untergeordnete Rolle. Weiterhin kann die Testdauer durch die Wahl eines Zeitrasters mit kürzeren Intervallen oder durch Instrumentierung, zu Lasten der Portabilität, erheblich reduziert werden.

Kapitel 10

Realisierung

Der in dieser Arbeit präsentierte Ansatz der Ableitung von Testmodellen und Testfallgenerierung mittels *Model Checking* für *Real-Time Object Oriented Modeling* wurde in Form einer integrierten Komponente für *Rational Rose Real-Time* realisiert. In diesem Kapitel werden Aufbau und Verfahrensweise des Testwerkzeugs *ROOMtest* erläutert und dargestellt.

10.1 Analyse

Die grundlegende Anforderung an das zu entwickelnde Werkzeug ist die Realisierung der in dieser Arbeit präsentierten Methode unter Verwendung von *Rational Rose Real-Time* und dem *Model Checker Uppaal*. Die zu realisierende Methode wurde bereits ausführlich dargestellt. Aus der Verwendung von *Rational Rose Real-Time* und Uppaal ergeben sich dagegen noch einige neue Anforderungen.

Neben den funktionalen Anforderungen soll das zu entwickelnde Testwerkzeug eine hohe Änderbarkeit und Erweiterbarkeit aufweisen, um die spätere Anpassung an andere Entwicklungsmethoden und -werkzeuge zu erleichtern und die Anbindung an andere *Model Checker* zu erlauben.

Rational Rose Real-Time

Das modellbasierte Softwareentwicklungswerkzeug *Rational Rose Real-Time* besitzt zu Erweiterungszwecken eine detaillierte COM-Schnittstelle. Diese Schnittstelle soll für die Realisierung von ROOMtest genutzt werden, um

- Struktur- und Verhaltensmodelle zu exportieren
  Die in *Rational Rose Real-Time* verwendete Semantik ist kompatibel zu der Semantik von *Real-Time Object-Oriented Modeling*.
- Struktur- und Verhaltensmodelle zu instrumentieren
  Zur Erhöhung der Steuer- und Beobachtbarkeit sollen die Modelle automatisch instrumentiert werden können. Die generierten Veränderungen sollen automatisch wieder entfernt werden können.
- Testfälle in Form von *Message Sequence Charts* zu erzeugen
  Der in *Rational Rose Real-Time* enthaltene *Quality Architect Real-Time* ist ein Testwerkzeug zur automatischen Ausführung, Protokollierung und Auswertung von Testfälle in Form von nachrichtenbasierten Sequenzdiagrammen (*engl. message sequence chart*). Die mittels ROOMtest generierten Testfälle sollen als Sequenzdiagramme in *Rational Rose Real-Time* verfügbar und ausführbar sein.

Uppaal Model Checker

Der Model Checker Uppaal erlaubt den Im- und Export von Systemdateien in XML-Format und die skriptbasierte Verarbeitung von textuellen Anforderungsdateien. Für die Anbindung von Uppaal an ROOMtest sind wenige Anforderungen zu beachten:

- Export eines *Uppaal Timed Automata* in XML-Format
  Die Semantik von Uppaal-Testmodellen geht aus der in dieser Arbeit beschriebenen Methode hervor.

- Export von Anforderungen als Textdatei
  Die Semantik von Uppaal-Testanforderungen geht aus der in dieser Arbeit beschriebenen Methode hervor.

- Export von Skriptdateien zur automatisierten Steuerung von Uppaal
  Die Automatisierung der Testfallermittlung mit Uppaal macht die Generierung von Skripten notwendig, die den *Model Checker* mit geeigneten Parametern, der generierten Systemdatei und der Anforderungsdatei, starten.

- Import von *Trace*-Dateien
  Der *Model Checker Uppaal* produziert auf Basis der Testanforderungen und der Systemdatei die gewünschten Testdaten in einer Sequenzdiagrammen ähnlichen Form. Diese Daten werden kodiert in *Trace*-Dateien (*.xtr*) gespeichert und müssen unter Verwendung der Systemdatei dekodiert werden.

10.2 Entwurf

Die Realisierung der im Vorangegangenen dargestellten Anforderungen erfolgt für die *Windows*-Plattform mittels *Microsoft Visual Studio .NET*. Diese Entscheidung basiert auf der Verfügbarkeit einer umfangreichen COM-Schnittstelle [104, 24] von *Rational Rose Real-Time.* Die Wahl von C# erfolgt aufgrund der vergleichsweise guten Mischung von konstruktiver Sicherheit und Komfortabilität sowie guten Erfahrungen in vorangegangenen Projekten mit dieser objektorientierten Programmiersprache. In Abbildung 10.1 ist der Aufbau von ROOMtest schematisch dargestellt.

HEFSM Modell

Die Kommunikation über die COM-Schnittstelle soll auf eine Interface-Komponente in ROOMtest begrenzt werden, um spätere Anpassungen an andere Entwicklungswerkzeuge zu erleichtern. Zu diesem Zweck wird ein Datenmodell benötigt, das die ausgelesenen Informationen für die weitere Verarbeitung speichert. Um Erweiterbarkeit und Änderbarkeit der Software zu gewährleisten, soll dieses Datenmodell möglichst unabhängig zu *Rational Rose Real-Time* entworfen werden. Daher wird die Semantik von hierarchischen, erweiterten Zustandsautomaten (*engl. hierarchical extended finite state machine, HEFSM*) für dieses Datenmodell gewählt.

Um die Nutzung von Fremdwerkzeugen für Prüfzwecke zu ermöglichen soll ein XML-Export dieser Modelle möglich sein. Da die Implementierung in einer objektorientierten Sprache erfolgt, sollte das Datenmodell diese Exportfunktionen kanonisch implementieren.

EFSM Modell

Die meisten *Model Checker* und die meisten Testfallgenerierungsmethoden basieren auf Systembeschreibungen in Form flacher Zustandsautomaten. Daher wird ein Datenmodell benötigt, das die flachen Systemmodelle speichert und die Entwicklung von Exportfiltern für die Anpassung an verschiedene Testfallgenerierungsmethoden und -werkzeuge ermöglicht.

Um die Nutzung von Fremdwerkzeugen für Prüfzwecke zu erleichtern, soll ein XML-Export dieser Modelle möglich sein. Da die Implementierung in einer objektorientierten Sprache erfolgt, sollte das Datenmodell diese Exportfunktionen kanonisch implementieren. Der Export eines Uppaal-Testmodells soll auf die gleiche Art erfolgen.

Testfallmodell

Das Ergebnis der Testfallgenerierung sind Nachrichtensequenzen zwischen der Testumgebung und dem Modul. Der *Model Checker* produziert *Trace*-Dateien, die zur Erzeugung von Test-Sequenzdiagrammen in *Rational Rose Real-Time* dienen. Eine *Trace*-Datei enthält kodierte Informationen, die

erst in Verbindung mit dem Uppaal-Testmodell die notwendigen Testdaten liefern. Die erzeugten Testsequenzen werden in einem Testfallmodell gespeichert, auf das von der Schnittstellenkomponente für die Erzeugung der Sequenzdiagramme zugegriffen wird.

Interface

Alle Zugriffe über COM auf *Rational Rose Real-Time* sollen innerhalb einer Schnittstellenkomponente gekapselt werden , um eine hohe Unabhängigkeit von *Rational Rose Real-Time* zu gewährleisten. Die Schnittstellenkomponente soll die automatische Generierung von HEFSM-Modellen aus den statischen und dynamischen Systemmodellen von *Rational Rose Real-Time* erlauben. Die Schnittstellenkomponente soll die automatische Generierung von Sequenzdiagrammen auf Basis von Testfallmodellen ermöglichen.

Testmodell-Generator

Für die Verwendung von Uppaal zur Testfallgenerierung müssen alle hierarchischen und konditionalen Elemente aus dem HEFSM-Modell entfernt werden. Der Testmodell-Generator produziert einen flachen, erweiterten Zustandsautomaten. Die notwendigen Schritte für die Entfernung aller hierarchischen Strukturen wird in dieser Arbeit ausführlich erläutert. Die Entfernung aller konditionalen Strukturen bezieht sich auf Strukturen aus Auswahlpunkten, die durch bedingte Transitionen ersetzt werden. Dieser Vorgang wird ebenfalls ausführlich in dieser Arbeit erläutert.

Trace-Import

Der Uppaal *Model Checker* produziert *Trace*-Dateien, die unter Verwendung der Systemdateien alle Informationen zu den gewünschten Testfällen vermitteln. Aus einer *Trace*-Datei können Initialzustände der Kommunikationskomponenten, Folgezustände und Variablenwerte nach Ausführung einer Transition sowie Synchronisationsbeziehungen abgeleitet werden. Diese Daten dienen zur Erzeugung von Testfällen im Testfallmodell.

10.3 Implementierung

Die Implementierung von ROOMtest erfolgte in der objektorientierten Programmiersprache C# und mittels der Entwicklungsumgebung *Microsoft Visual Studio*. Jedes Verarbeitungs- und Datenmodell wurde als eine separate Komponente implementiert. Eine *Visual Basic* Komponente mit einem grafischen Interface dient als COM-Objekt und zur Steuerung des Ablaufs.

Die Anbindung einer anderen Entwicklungsumgebung kann durch Implementierung einer angepassten Schnittstellenkomponente erfolgen.

Für den XML-Export stehen XML-Schemata für den XML-Editor *Altova XMLSpy* zur Verfügung, die im Anhang dargestellt sind. Der Export von XML-Dateien erfolgt mittels Objektoperationen, die jede Klasse eines Datenmodell implementiert. Ein solches Objekt hat Zugriff auf eine *toXML*- und eine *toUppaal*-Operationen, die kanonisch diese Operationen aller weiteren betroffenen Objekte aufruft. Eine *toXML*-Operation liefert eine allgemeine XML-Repräsentation des Objekts. Eine *toUppaal*-Operation liefert eine spezielle XML-Repräsentation des Objekts mit der entsprechenden Semantik von *Uppaal Timed Automata*.

Für die Erzeugung der Testmodelle können in der grafischen Benutzeroberfläche diverse Parameter variiert werden, beispielsweise ob eine Instrumentierung vorgenommen oder entfernt werden soll. Die Auswahl verschiedener Überdeckungskriterien ist ebenfalls möglich. Auf Basis dieser Daten werden das Testmodell und die Uppaal-Anforderungen erzeugt. Der Import von Testdaten erfolgt ebenfalls mittels der grafischen Oberfläche durch den Systemnutzer. Weitere Implementierungsdetails sollen an dieser Stelle nicht präsentiert werden.

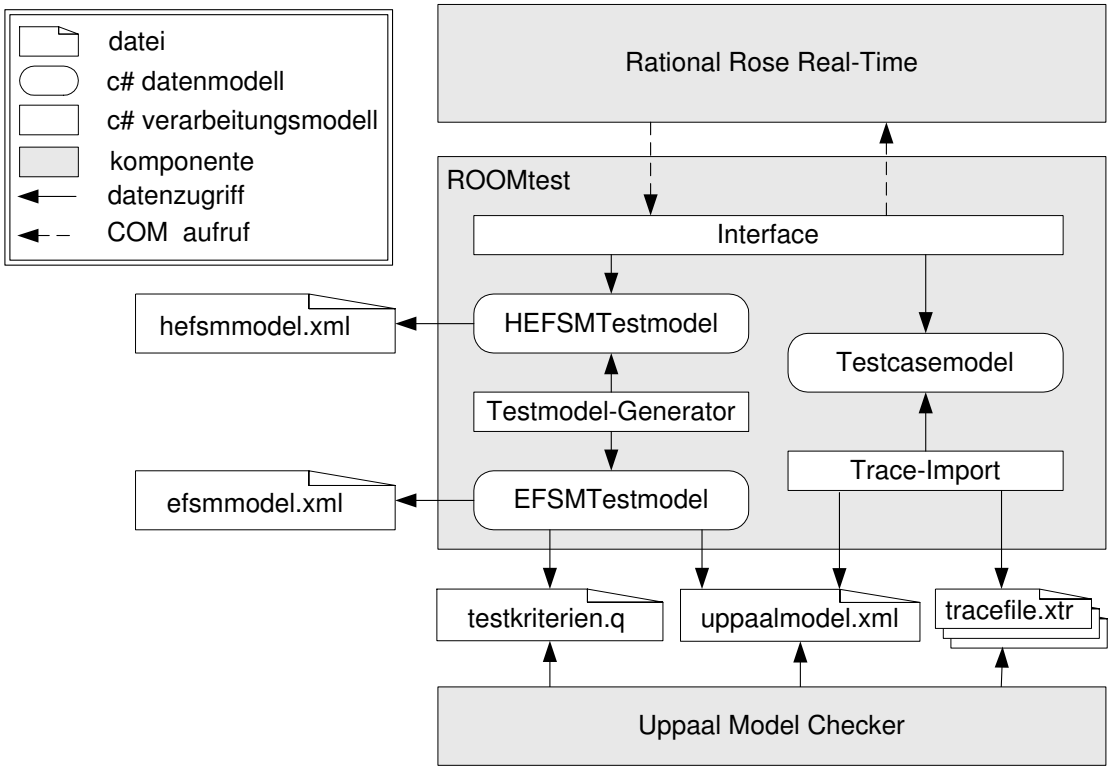


Abbildung 10.1: Aufbau *ROOMtest*

Kapitel 11

Zusammenfassung

In dieser Arbeit wurden ein umfassender Ansatz und eine Methode für die Modulprüfung auf Basis von *Real-Time Object-Oriented Modeling* vorgestellt und in einen modellbasierten Softwareentwicklungsprozess für technische Softwaresysteme eingeordnet.

Ein Kernthema dieser Arbeit ist die Ableitung von Testmodellen in Form flacher Zustandsautomaten aus ROOMcharts. Für die Testfallgenerierung basierend auf ROOMcharts und den abgeleiteten Testmodellen wird der *Model Checker Uppaal* genutzt. Zu diesem Zweck wurde ein leicht abgewandeltes Verfahren für die Konstruktion von Uppaal-Testmodellen vorgestellt.

Die Testfallgenerierung kann auf Basis von Transitions-, Bedingungs- und Datenflussüberdeckungskriterien erfolgen. Das Pfadüberdeckungskriterium wurde für die Testfallgenerierung als unzweckmäßig demonstriert. Bei der Ableitung von zyklischen, kontinuierlichen Kontrollstrukturen dient allerdings das *Boundary Interior* Kriterium zur Komplexitätsbeherrschung. Für alle verwendeten Verfahren der Testmodell- und Testfallgenerierung wurde die Zweckmäßigkeit anhand einer Untersuchung der Raum- und Zeitkomplexitäten diskutiert. Im Rahmen der Komplexitätsuntersuchung wurden die Effekte der wesentlichen Entwurfselemente von ROOMcharts auf die Komplexität der Testfallgenerierung mittels Uppaal und Breitensuche demonstriert. Weiterhin wurden Maßnahmen für die Komplexitätsreduktion vorgeschlagen und anhand von Beispielen veranschaulicht.

Ein typisches Problem bei der Durchführung von *Black Box* Tests, beispielsweise in einem *Hardware-In-The-Loop* Entwicklungsansatz [88], ist die oftmals mangelhafte Beobachtbarkeit der internen Zustände und Abläufe der zu testenden Software. Diesem Problem wurde durch die Entwicklung eines neuartigen Verfahrens für die Generierung von *Distinguishing Sequences* mittels *Model Checking* begegnet.

Die Anwendung von Verfahren zur Testfallgenerierung und die Semantik des verwendeten *Model Checkers* führen zu einer Anzahl von Testbarkeitsanforderungen, die erläutert und diskutiert wurden. Erst die Existenz eines ROOMcharts in Normalform erlaubt einen Rückschluss von der erzielten Modellüberdeckung auf die implizierte Codeüberdeckung. Für die Erfüllung dieser Anforderung, muss bereits während der Entwicklung auf die Modellierung einer Normalform geachtet werden. Zur Vereinfachung dieses Vorgangs wurde das Entwurfselement des *Hierarchischen Pseudozustands* vorgestellt.

Die auf Basis von *Real-Time Object-Oriented Modeling* entwickelte Testmethode wurde als Testwerkzeug für die Softwareentwicklungsumgebung *Rational Rose Real-Time* realisiert, welche auf der *Unified Modeling Language Real-Time* basiert.

Beurteilung

Der in dieser Arbeit vorgestellte Ansatz eines modellbasierten Modultestverfahrens für *Real-Time Object-Oriented Modeling* kann gut auf aktuelle Entwicklungsansätze elektronischer Komponenten, beispielsweise *Hardware-In-The-Loop*, angewendet werden. In diesem Umfeld ist eine automatische Strukturüberdeckung der Steuersoftware von großem Vorteil, da kontinuierlich und entwicklungsbegleitend auf der Zielplattform getestet wird und der zeitliche Aufwand zur Testfallentwicklung begrenzt werden muss.

Die Abbildung von ROOMcharts auf Testmodelle mit der Semantik endlicher Zustandsautomaten oder *Uppaal Timed Automata* ist problematisch, wenn komplexe Aktionen in Transitionen und Zuständen vorhanden sind. Insbesondere die Verwendung von Schleifenanweisungen kann zu einer Explosion des Testmodells führen, wenn eine große Anzahl von Pfaden durch die Schleife impliziert wird. Auch die Verwendung von *Hierarchischen Pseudozuständen* löst dieses Problem nicht. Durch die Reduktion auf k Schleifendurchläufe, mittels strukturiertem Pfadtest oder *Boundary Interiour* Test, kann diesem Problem nur begegnet werden, wenn die Schleife innerhalb von k verlassen werden kann. Die algorithmische Bestimmung eines k, innerhalb dessen die Schleife beendet werden kann, ist in ähnlicher Form als *Turings Halteproblem* bekannt. Da für dieses Problem keine allgemeine Lösung gefunden werden kann, wird es auch praktische Anwendungen geben, für die kein Testmodell nach der hier präsentierten Methode existiert. In diesem Fall kann nur eine Anpassung des Entwurfs erfolgen, um ein Testmodell abzuleiten.

Die Nutzung von *Model Checking* zur Testfallgenerierung bringt immer das Problem der *Explosion des Zustandsraums* mit sich. Es konnte in dieser Arbeit demonstriert werden, dass durch verhältnismäßig einfache Maßnahmen, beispielsweise manuelle Datenabstraktion, eine erhebliche Komplexitätsreduktion erzielt werden kann. Der manuelle Eingriff an dieser Stelle ist kritisch hinsichtlich der erhöhten Fehleranfälligkeit der Testfallgenerierung. Es ist allerdings möglich, durch systematische Werkzeugunterstützung dieses Fehlerpotential auf ein akzeptables Maß zu reduzieren.

In Experimenten im Rahmen dieser Arbeit konnte für den *Model Checker Uppaal* in der Version 3.4.6 mit einem maximal adressierbaren Speicherbereich von 4 Gigabyte eine Obergrenze für berechenbare Systeme im Bereich zwischen 40 Millionen *Locations* ohne Variablen und einer *Location* mit 20 Millionen Variablenwerten ermittelt werden. Realistische Anwendungen für die Testfallgenerierung mittels Uppaal sollten daher im Bereich einer zweistelligen Anzahl von *Locations* und einer einstelligen Anzahl von erweiterten Zustandsvariablen liegen. Bei dieser Abschätzung wurden keine Maßnahmen zur Komplexitätsreduktion berücksichtigt. Mit dem *Model Checker* SMV wurden bereits deutlich größere Systeme von maximal 10^{120} Zuständen berechnet. Obwohl die Übertragbarkeit dieser Ergebnisse auf das vorhandene Problem nicht gewährleistet ist, da es sich um sehr dünn besetzte Systeme handelte, sollte trotzdem die Verwendung anderer *Model Checker* erwogen werden.

Eine aktuelle experimentelle Anwendung der modellbasierten Softwareentwicklungsmethoden in Kooperation der Technischen Universität Braunschweig und der Volkswagen AG in Wolfsburg, dient der Entwicklung der Steuerungssoftware eines Fensterhebersystems im Automobil [50]. Die Entwicklung basiert auf ausführbaren Modellen, die manuell implementiert werden. Dieses System besteht aus elf Komponenten, die gemeinsam ein kombinatorisches System von $1,4 \cdot 10^{16}$ Zuständen aufspannen. Die höchste Komplexität besitzt die zentrale Fensterheberkontrolle mit 107 Zustände, die sich aus der Kombination von *Locations* und Daten zusammensetzen. Die Anwendung der Testfallgenerierung mittels Uppaal ist nach dieser Komplexitätsabschätzung möglich.

Weiterhin ist auch die eingeschränkte Semantik der Systembeschreibungssprache negativ zu bewerten. Durch den Einsatz anderer *Model Checker* ist hier möglicherweise eine Verbesserung zu erzielen, die allerdings mit hoher Wahrscheinlichkeit eine erhöhte Komplexität mit sich bringt. In der praktischen Anwendung kann diesem Problem durch die konsequente Verwendung von ganzzahligen Daten begegnet werden. Da im *Model Checker* nur Daten problematisch sind, die den Kontrollfluss und Ereignisparameter bestimmen, können eingeschränkt auch andere Datentypen in weniger problematischen Bereichen verwendet werden. Eine weitere Möglichkeit besteht in der Abbildung der Fließkommadaten auf Ganzzahlen zum Zweck der Testfallgenerierung. Die Praxistauglichkeit dieser Ansätze sollte in nachfolgenden Forschungsarbeiten untersucht werden.

Der Vorteil des Ansatzes der Testfallgenerierung mittels *Model Checking* liegt in der strikten Trennung von Testanforderungen und Generierungsalgorithmus. Der Vorgang der Testfallgenerierung beschränkt sich im Wesentlichen auf die Formulierung eines Überdeckungskriteriums. Die komplexitätsreduzierenden Maßnahmen werden daher vorwiegend auf das Testmodell angewendet. Durch die Wahl eines geeigneten Algorithmus kann ebenfalls die Komplexität reduziert werden. Im Rahmen dieser Arbeit fiel die Wahl auf *CTL Model Checking* und Breitensuche. Es ist allerdings möglich durch die Wahl effizienterer Algorithmen weitere Komplexitätsreduktionen zu erzielen, ohne wesentliche Änderungen am Testmodell vornehmen zu müssen. Mit der Verwendung verbesserter *Model Checking* Algorithmen wird somit auch der hier präsentierte Ansatz verbessert.

Weiterhin ist dieser vergleichsweise leicht an neue Testanforderungen anzupassen, da keine neuen Lösungsalgorithmen implementiert werden müssen.

Insgesamt überwiegen die Vorteile der Generierung von Testfällen mittels *Model Checking* gegenüber allen Nachteilen. Im Vergleich mit der manuellen Implementierung von analytischen oder heuristischen Lösungsansätzen sticht *Model Checking* hier durch Wirtschaftlichkeit, Änderbarkeit und Anpassbarkeit hervor. Dagegen besitzen manuell implementierte Speziallösungen oftmals eine höhere Effizienz des Lösungsalgorithmus, die in der Regel allerdings auf lineare Verbesserungen beschränkt ist, da die Komplexitätsklasse des Testproblems identisch ist.

Für die praktische Nutzung ist die Ableitung des modulinternen Zustands über das Ein-/Ausgabeverhalten von großem Vorteil, da das Laufzeitverhalten nicht durch eine Instrumentierung verfälscht und die Beobachtbarkeit nicht durch Restriktionen der Testumgebung eingeschränkt werden. Möglicherweise existiert nicht für jeden Minimalautomaten eine *Distinguishing Sequence*. In diesem Fall kann auf die Generierung von *Wp-Sets* ausgewichen werden. Die Existenz eines Minimalautomaten ist für den praktischen Einsatz eine sehr strenge Anforderung an die Systemstruktur. Der Konstruktionsprozess sollte zur Erfüllung und Überwachung dieser Anforderung durch Werkzeuge unterstützt werden, die den Entwickler zu dem Entwurf eines Minimalautomaten anleiten. Es kann auch untersucht werden, ob die Ableitung von Minimalautomaten zu Testzwecken in der Praxis vorteilhaft ist. In der Regel lässt sich dieses Problem auf eine Teilmenge nicht identifizierbarer Zustände reduzieren. Durch Inspektion der Testdaten kann in diesen Fällen möglicherweise die Modulprüfung ergänzt und erfolgreich abgeschlossen werden. Die Minimierung eines Automaten ist dem Problem der Generierung von *Characterization Sets* sehr ähnlich und verursacht daher die gleichen Komplexitätsprobleme.

Die Konstruktion eines Minimalautomaten kann aus den verschiedensten Gründen in der Praxis unmöglich sein. Zum gegenwärtigen Stand der vorgestellten Generierungsmethode muss in diesen Fällen auf die Verwendung von *Characterization Sets* verzichtet werden. Ein möglicher Ausweg ist die Abstraktion eines minimalen Testmodells vom nicht minimalen Automaten. Welche Aussagekraft ein Test mit *Characterization Sets* auf Basis eines abstrakten Modells hat, sollte in nachfolgenden Forschungsarbeiten untersucht werden.

Die Verwendung von *Model Checking* für die Generierung von *Characterization Sets* beschränkt die Anwendbarkeit dieses Ansatzes auf minimale ROOMcharts mit ganzzahligen erweiterten Zustandsvariablen. In nachfolgenden Forschungsarbeiten sollten Ansätze zur Behandlung nicht minimaler Systeme mit beliebigen Datentypen untersucht werden.

Ausblick

Der vorgestellte Ansatz einer modellbasierten Modulprüfung für *Real-Time Object-Oriented Modeling* kann möglicherweise erfolgreich auf andere modellbasierte Softwareentwicklungsansätze übertragen werden. In einem anschließenden Forschungsvorhaben sollten verschiedene Arbeiten die Übertragbarkeit der Ergebnisse prüfen. Viel versprechende Kandidaten für die Übertragung des Ansatzes sind SDL-2000 und *Stateflow*. Im Rahmen von zwei Forschungs- und Entwicklungsarbeiten kann die Eignung von SDL-2000 und Stateflow untersucht werden. Eine dritte Arbeit sollte die Gemeinsamkeiten der drei Entwicklungsansätze darstellen. Die vierte Arbeit sollte die Entwicklung eines gemeinsamen Prüfkonzepts behandeln. Als Ziel des gesamten Vorhabens könnte langfristig die Entwicklung von Konzepten, Methoden und Werkzeugen für die modellbasierte Modulprüfung erfolgen, die auf alle genannten modellbasierten Softwareentwicklungsansätze gleichermaßen anwendbar sind.

Weiterhin kann diese Arbeit als Ausgangspunkt für die Entwicklung neuartiger Konzepte, Methoden und Werkzeuge für eine modellbasierte Integrationsprüfung dienen. Insbesondere die Verwendung von *Model Checking* in diesem Umfeld ist sinnvoll. Eine erste Arbeit auf dem Gebiet des Integrationstest von *Info- und Entertainmentsystemen* in Automobilen [8] hat bereits viel versprechende Ergebnisse geliefert und den Bedarf an effizienten Konzepten und Methoden für einen systematischen Test verteilter Systeme verdeutlicht. Auch dieser Forschungsansatz sollte verschiedene modellbasierte Entwicklungsansätze umfassen, um die Durchgängigkeit zu den Methoden der Modulprüfung zu gewährleisten. Zusätzliche Forschungsarbeiten könnten auf die Entwicklung umfassender Konzepte und Methoden zielen. Weiterhin sind in diesem Umfeld Randbedingungen

aus Hardware- und Systemumgebung zu berücksichtigen, die das Zeitverhalten der zu testenden Software maßgeblich beeinflussen. Insbesondere die Prüfung des Zeitverhaltens der Software ist beim Integrationstest von hoher Bedeutung. Hier sollte eine weitere Arbeit der Entwicklung von Konzepten und Methoden für die Behandlung von Zeitproblemen im modellbasierten Integrationstest dienen. Auch dieses Thema besitzt das Potenzial langfristig und erfolgreich ausgebaut zu werden. Weiterhin wurde bereits das Interesse der Industrie an leistungsfähigen Lösungen dieser Problematik deutlich.

Langfristig könnten diese Ansätze zu Forschungsthemen ausgebaut werden, die Raum für Dissertationsvorhaben und Industriekooperationen bieten würden.

Schlusswort

In dieser Arbeit wurden ein vollständiges Konzept und eine Methode für den modellbasierten Modultest für *Real-Time Object-Oriented Modeling* präsentiert und die Anwendbarkeit sowie die Praxistauglichkeit bezüglich einer eingeschränkten Art von Systemen demonstriert. Die Entwicklung eines praxistauglichen Werkzeugs sollte in naher Zukunft vorgenommen werden. Insbesondere die Übertragbarkeit der hier präsentierten Ergebnisse auf andere modellbasierte Entwicklungsmethoden und die Entwicklung ausgereifter Konzepte, Methoden und Werkzeuge für die systematische, modellbasierte Qualitätssicherung sind angesichts der rasanten Entwicklung und der hohen Akzeptanz modellbasierter Entwicklungsmethoden empfehlenswerte, zukünftige Forschungsansätze. Diese Arbeit hat weiterhin den hohen Nutzen von *Model Checking* als universelles und flexibles Mittel zur Lösung einer Vielzahl von Test- und Analyseproblemen demonstriert.

Literaturverzeichnis

[1] ADA: Annotated Ada Reference Manual, Version 6.0 / Intermetrics Inc. 1995. – Documentation

[2] Aho, A. V. ; Dahbura, A. T. ; Lee, D. ; Uyar, M. U.: An Optimization Technique for Protocol Conformance Test Generation Based on UIO Sequences and Rural Chinese Postman Tours. In: *IEEE Transactions on Communications* 39 (1991), November, Nr. 11

[3] Alur, R. ; Dill, D. L.: A theory of timed automata. In: *Journal of Theoretical Computer Science* 126 (1990)

[4] Alur, R. ; Dill, D. L.: A theory of timed automata. In: *Theoretical Computer Science* 126 (1994), Nr. 2

[5] Angermann, A. ; Beuschel, M. ; Rau, M. : *Matlab, Simulink, Stateflow*. Oldenbourg, 2005

[6] Balzert, H. : *Lehrbuch der Software-Technik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*. Bd. 2. Heidelberg, Berlin : Spektrum Akademischer Verlag, 1998

[7] Balzert, H. : *Lehrbuch der Software-Technik: Software-Entwicklung*. Bd. 1. 2. Auflage. Heidelberg, Berlin : Spektrum Akademischer Verlag, 2000

[8] Bauer, T. ; Herrmann, J. ; Liggesmeyer, P. ; Robinson-Mallett, C. : A Flexible Integration Strategy for In-Car Telematics Systems (SEAS). In: *ICSE workshop on Software Engineering for Automotive Systems*. St. Louis, May 2005

[9] Becker, L. B. ; Pereira, C. E.: From Design to Implementation: Tool Support for the Development of Object-Oriented Real-Time Systems

[10] Beizer, B. : *Software Testing Techniques*. International Thomson Computer Press, 1990

[11] Bengtsson, J. ; Larsen, K. G. ; Larsson, F. ; Pettersson, P. ; Yi, W. : Uppaal - a Tool Suite for Automatic Verification of Real-Time Systems. In: *Proceedings of the 4th DIMACS Workshop on Verification and Control of Hybrid Systems*, 1995

[12] Berry, G. ; Gonthier, G. : The Esterel synchronous programming language: Design, semantics, implementation. In: *Science of Computer Programming* 19 (1992)

[13] Bichler, L. ; Radermacher, A. ; Schürr, A. : Combining Data Flow Equations with UML/Realtime. In: *Proceedings of the Fourth International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'01)*, IEEE Computer Society, 2001

[14] Binder, R. : *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 1999

[15] Birolini, A. : *Reliability Engineering*. 4th. Springer, July 2003

[16] v. Bochmann, G. : Hardware Specification with Temporal Logic: An Example. In: *IEEE Transactions on Computers* C-31(3)

[17] v. Bochmann, G. : Finite State Description of Communications Protocols / Departement d'Informatique, Universite de Montreal. 1976 (236). – Forschungsbericht

[18] Bochmann, G. V. ; Petrenko, A. : Protocol testing: review of methods and relevance for software testing. In: *Proceedings of the 1994 international symposium on Software testing and analysis*, ACM Press, 1994

[19] Bogdanov, K. ; Holcombe, M. ; Singh, H. : Automated Test Set Generation for Statecharts. In: *Proceedings International Workshop on Current Trends in Applied Formal Methods*, 1998

[20] Bogdanov, K. : *Automated Testing of Harel's Statecharts*, University of Sheffield, Diss., January 2000

[21] Bogdanov, K. ; Holcombe, M. : Statechart testing method for aircraft control systems. In: *Software Testing Verification Reliability* 11 (2001), Nr. 2

[22] Bogdanov, K. ; Holcombe, M. : Testing from statecharts using Wp method. In: *FATES'02: Formal Approaches to Testing of Software* FORTEST, 2002

[23] Bosch, J. ; Molin, P. : A Model for Flexible and Predictable Object-Oriented Real-Time Systems. In: *Proc. of the 3rd Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'97)*, IEEE Press, 1997

[24] Box, D. : *Essential COM.* Addison-Wesley, 1997

[25] Briand, L. C. ; Labiche, Y. ; Wang, Y. : Using Simulation to Empirically Investigate Test Coverage Criteria Based on Statecharts / Software Quality Engineering Laboratory, Carleton University, Systems and Computer Engineering. Ottawa, Canada, October 2002 (TR SCE-02-09). – Forschungsbericht

[26] Bund: V-Modell XT / Koordinierungs und Beratungsstelle der Bundesregierung für Informationstechnik in der Bundesverwaltung. Berlin, 2004. – Guideline

[27] Bundesamt für Wehrtechnik und Beschaffung: Entwicklungsstandard für IT-Systeme des Bundes, Vorgehensmodell, Teil 1: Regelungsteil, Teil 3: Handbuchsammlung, Allgemeiner Umdruck Nr. 250/1 / IT 15. Koblenz, Juni 1997. – Guideline

[28] Burch, J. R. ; Clarke, E. M. ; Long, D. E. ; McMillan, K. L. ; Dill, D. L.: Symbolic Model Checking for Sequential Circuit Verification. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits* 13 (4)

[29] Burstall, R. M.: Program proving as hand simulation with a little induction. In: *IFIP Congress 74*, North Holland, 1974

[30] Burton, S. : *Automated Generation of High Integrity Test Suites from Graphical Specifications*, University of York, Dept. of Computer Science, Diss., March 2002

[31] Campos, S. ; Clarke, E. : Real-Time Symbolic Model Checking for Discrete Time Models. In: *AMAST Series in Computing: Theories and Experiences for Real-Time System Development* (1995)

[32] Cardell-Oliver, R. : Conformance test experiments for distributed real-time systems. In: *ISSTA 2002*. Rome : ACM Press, August 2002

[33] Chan, W. Y. ; Vuong, S. T. ; Ito, M. R.: An Improved Protocol Test Generation Procedure based on UIOs. In: *Proceedings of the ACM Symposium on Communication Architectures and Protocols*, ACM, 1989

[34] Chen, W.-H. : Test Sequence Generation from the Protocol Data Portion Based on the Selecting Chinese Postman Algorithm. In: *Information Processing Letters* 65 (1998), Nr. 1

[35] Chen, W.-H. ; Lu, C.-S. ; Brozovsky, E. R. ; Wang, J.-T. : An Optimization Technique for Protocol Conformance Testing Using Multiple UIO Sequences. In: *Information Processing Letters* 36 (1990), Nr. 1

[36] Chen, W.-H. ; Tang, C. Y.: Computing the Optimal IO Sequences of a Protocol in Polynomial Time. In: *Information Processing Letters* 40 (1991), Nr. 3

[37] Chow, T. S.: Testing Software Design Modeled by Finite-State Machines. In: *IEEE Transactions On Software Engineering* 4 (1978), Nr. 3

[38] Clarke, E. M. ; Emerson, E. A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: *LNCS 131, Workshop on Logic of Programs*, Springer, 1981

[39] Clarke, E. M. ; Emerson, E. A. ; Sistla, A. P.: Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. In: *ACM Transactions on Programming Languages and Systems* 8 (2)

[40] Clarke, E. M. ; Grumberg, O. ; Long, D. E.: Model Checking and Abstraction. In: *Proc. 19th Annual ACM Symposium on Principles of Programming Languages*, 1992

[41] Diethers, K. ; Goltz, U. ; Huhn, M. : Model Checking UML Statecharts with Time. In: *UML 2002, Workshop on Critical Systems Development with UML*, 2002

[42] DIN: EN 50128: Bahnanwendungen - Telekommunikationstechnik, Signaltechnik und Datenverarbeitungssysteme - Software für Eisenbahnsteuerungs- und Überwachsungssysteme. Berlin : Deutsches Institut für Normung e.V., 2001. – Guideline

[43] Domschke, W. ; Drexl, A. : *Einführung in Operations Research*. Berlin : Springer, 2004

[44] Douglass, B. : *Doing hard time: developing realtime systems with UML, objects, frameworks, and patterns*. Addison Wesley Longman, 1999

[45] Emerson, E. A. ; Clarke, E. M.: Characterizing Correctness Properties of Parallel Programs using Fixpoints. In: *LNCS 85, International Colloquium on Automata, Languages and Programming*, Springer, July 1980

[46] En-Nouaary, A. ; Dssouli, R. ; Khendek, F. : Timed Wp-Method: Testing Real-Time Systems. In: *IEEE Transactions on Software Engineering* 28 (2002), November, Nr. 11

[47] Engels, A. ; Feijs, L. ; Mauw, S. : Test Generation for Intelligent Networks using Model Checking. In: Brinksma, E. (Hrsg.): *Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'97*. Enschede : Springer, April 1997

[48] Feijs, L. ; Goga, N. ; Mauw, S. ; Tretmans, J. : Test selection, trace distance, and heuristics. In: *International Federation for Information Processing (IFIP)* Bd. 210, 2002

[49] Fernandez, J. ; Jard, C. ; Jeron, T. ; Viho, G. : Using on the fly verification techniques for the generation of test suites. In: Alur, A. (Hrsg.) ; Henzinger, I. (Hrsg.): *Conference on Computer-Aided Verification* Bd. 1102. New Brunswick, New Jersey : Springer, July 1996

[50] Firley, T. : *Computing Abstract Models for Verifying Reactive Systems*, Technical University of Brunswick, Germany, Diss., 2004

[51] Fujiwara, S. ; v. Bochmann, G. ; Khendek, F. ; Amalou, M. ; Ghedamsi, A. : Test Selection Based on Finite State Models. In: *IEEE Transactions on Software Engineering* 17 (1991), Nr. 6

[52] Gargantini, A. ; Heitmeyer, C. : Using Model Checking to Generate Tests from Requirements Specification. In: *Proceedings of the 7th European Software Engineering Conference*. Toulouse : ACM Press, 1999

[53] Gaudreau, D. ; Freedman, P. : Temporal Analysis and Object-Oriented Software Development: a Case Study with ROOM/ObjectTime. In: *Real-Time Systems Symposium* IEEE, 1996

[54] Gericke, J. ; Liggesmeyer, P. : Eine Erweiterung der Unified Modeling Language zur Verfolgung von Software-Anforderungen in sicherheitskritischen Systemen. In: *Informatik Forschung und Entwicklung* 17, 2002, Nr. 2

[55] Gill, A. : *Introduction to the Theory of Finite-State Machines.* New York, USA : McGraw-Hill, 1962

[56] Gnesi, S. ; Latella, D. ; Massink, M. : Formal Test-Case Generation for UML Statecharts. In: *Proc. of the 9th IEEE International Conf. on Engineering Complex Computer Systems*, IEEE Press, 2004

[57] Godefroid, P. ; Pirottin, D. : Refining Dependencies Improves Partial-Order Verification Methods. In: *LNCS 697: Proceedings of the 5th Conference on Computer-Aided Verification*, Springer, 1993

[58] Gomaa, H. : *Designing Concurrent, Distributed, and Real-Time Applications with UML.* Boston : Addison-Wesley, 2000

[59] Gönenc, G. : A Method for the Design of Fault Detection Experiments. In: *IEEE Transactions on Computers* C-19 (1970), June, Nr. 6

[60] Gosling, J. ; Joy, B. ; Steele, G. ; Bracha, G. : *The Java Language Specification Second Edition.* Boston, Mass. : Addison-Wesley, 2000

[61] Group, O. M.: Guide to Model Driven Architecture / OMG. 2003 (v1.0.1). – Documentation

[62] Halang, W. A. ; Pereira, C. E. ; Frigeri, A. H.: Safe Object-Oriented Programming of Distributed Real-Time Systems in PEARL*. In: *Proc. of the 4th International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'01)*, IEEE Press, 2001

[63] Hamon, G. ; de Moura, L. ; Rushby, J. : Generating Efficient Test Sets with a Model Checker. In: *Proc. of the 2nd Intern. Conf. on Software Engineering and Formal Methods (SEFM'04)*, IEEE Press, 2004

[64] Harel, D. : Statecharts: A visual formalism for complex systems. In: *Science of Computer-Programming, North-Holland* 8 (1987)

[65] Hennessy, M. C. B. ; Plotkin, G. D.: A Term Model for CCS. In: Dembinski, P. (Hrsg.): *Mathematical Foundations of Computer Science 1980, Proceedings of the 9th Symposium* Bd. 88. Rydzyna, Poland : Springer, 1–5 September 1980

[66] Hennie, F. C.: *Finite State Models for Logical Machines.* New York, USA : Wiley, 1968

[67] Herrmann, D. S.: *Software Safety and reliability: techniques, approaches, and standards of key industrial sectors.* IEEE Computer Society Press, 1999

[68] Hessel, A. ; Larsen, K. ; Nielsen, B. ; Pettersson, P. ; Skou, A. : Time-Optimal Real-Time Test Case Generation using UPPAAL. In: *FATES2003*, 2003

[69] Hessel, A. ; Larsen, K. ; Nielsen, B. ; Pettersson, P. ; Skou, A. : Time-Optimal Test Cases for Real-Time Systems. In: *Workshop on Formal Modeling and Analysis of Timed Systems*, 2003

[70] Hierons, R. M. ; Kim, T.-H. ; Ural, H. : Expanding an Extended Finite State Machine to aid Testability. In: *Proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC'02)*, IEEE Computer Society, 2002

[71] Hierons, R. M. ; Sadeghipour, S. ; Singh, H. : Testing a System specified using Statecharts and Z. In: *Information and Software Technology* 43 (2001), Nr. 2

[72] Hierons, R. ; Ural, H. : Reduced Length Checking Sequences. In: *IEEE Transactions on Computers* 51 (2002), September, Nr. 9

[73] Higashino, T. ; Nakata, A. ; Taniguchi, K. ; Cavalli, A. R.: Generating Test Cases for a Timed I/O Automaton Model. In: *Proceeding of the IFIP 12th International Workshop on Testing of Communicating Systems*, Springer, 1999

[74] Hoare, C. : *Communicating Sequential Processes.* Englewood Cliffs, NJ : Prentice-Hall, 1984

[75] Hong, H. ; Lee, I. ; Sokolsky, O. ; Cha, S. : Automatic Test Generation from Statecharts Using Model Checking. In: *FATES.* Aalborg, Denmark, 2001

[76] Hu, E. Y. ; Bernat, G. ; Wellings, A. : Addressing Dynamic Dispatching in WCET Analysis for Object-Oriented Hard Real-Time Systems. In: *Proc. of the 5th International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'02)*, IEEE Press, 2002

[77] Ipate, F. ; Holcombe, M. : Generating test sets from non-deterministic stream X-Machines. In: *Formal Aspects of Computer Science* (2001)

[78] ITU-T: Z.100 (08/02) Specification and description language (SDL). International Telecommunication Union, August 2002. – Documentation

[79] for Java Expert Group, T. R.-T. : *The Real-Time Specification for Java.* Addison-Wesley, 2000

[80] Jéron, T. ; Morel, P. : Test generation derived from model-checking. In: *International Conference on Computer Aided Verification* Bd. 1633, 1999

[81] Kim, S. ; Clark, J. ; McDermid, J. : Class Mutation: Mutation Testing for Object-Oriented Programs. In: *Object-Oriented Software Systems, Net.ObjectDays'2000*, 2000

[82] Knuth, D. E.: Big Omicron and big Omega and big Theta. In: *SIGACT News* 8 (1976), Nr. 2. – ISSN 0163–5700

[83] Kap. Acta Informatica 8(3) In: Kröger, F. : *LAR: A logic for algorithmic reasoning*

[84] Kung, D. ; Lu, Y. ; Venugopalan, N. ; Hsia, P. ; Toyoshima, Y. ; Chen, C. ; Gao, J. : Object State Testing and Fault Analysis for Reliable Software Systems. In: *Proc. o. t. 7th International Symposium on Software Reliability Engineering, ISSRE 96*, IEEE Press, 1996

[85] Kung, D. C. ; Suchak, N. ; Gao, J. ; Hsia, P. : On Object State Testing. In: *Proc. of IEEE COMPSAC '94*, 1994

[86] Küster, J. M. ; Stroop, J. : Consistent Design of Embedded Real-Time Systems with UML-RT. In: *Proceedings of the Fourth International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'01)*, IEEE Computer Society, 2001

[87] Larsen, K. G. ; Pettersson, P. ; Yi, W. : UPPAAL in a Nutshell. In: *International Journal on Software Tools for Technology Transfer* 1 (1997), Nr. 1-2

[88] Ledin, J. : Hardware-In-The-Loop Simulation. In: *Embedded Systems Programming* 12 (1999), February, Nr. 2

[89] Lee, D. ; Yannakakis, M. : Testing Finite-State Machines: State Identification and Verification. In: *IEEE Transactions on Computers* 43 (1994), March, Nr. 3

[90] Leuween, J. V.: *Handbook of Theoretical Computer Science - Formal Models and Semantics.* Bd. B. Elsevier Science Publishers, 1990

[91] Lichtenstein, O. ; Pnueli, A. : Checking that Finite-State Concurrent Programs Satisfy their Linear Specification. In: *Proceedings of the 12th Annual ACM Symposium on Logic in Computer Science*, ACM, 1985

[92] Liggesmeyer, P. : *Software-Qualität: Testen, Analysieren und Verifizieren von Software.* Heidelberg, Berlin : Spektrum Akademischer Verlag, 2002

[93] Lilius, J. ; Paltor, I. : Formalising UML State Machines for Model Checking. In: *UML'99*, 1999

[94] Luo, G. ; v. Bochmann, G. ; Petrenko, A. : Test selection based on communicating nondeterministic finite state machines using a generalized Wp-method. In: *IEEE Transactions On Software Engineering* 20 (1994), Nr. 2

[95] Malachi, Y. ; Owicki, S. S.: Temporal Specifications of Self-Timed Systems. In: Kung, H. T. (Hrsg.) ; Sproull, B. (Hrsg.) ; Steele, G. (Hrsg.): *VLSI Systems and Computations.* Computer Science Press, 1981

[96] McMillan, K. L.: *Symbolic Model Checking - An Approach to the State Explosion Problem*, Carnegie Mellon University, Diss., 1992

[97] Mellor, S. J. ; Scott, K. ; Uhl, A. ; Weise, D. : *MDA Distilled:Principle of Model-Driven Architecture.* Boston : Addison-Wesley, 2004

[98] Meyer, B. ; The, E. : *Eiffel: The Language; International Series in Computer Science.* Englewood Cliffs : Prentice-Hall, 1993

[99] Meyer, B. : *Object-Oriented Software Construction 2nd Ed.* Santa Barbara : Prentice-Hall, 1997

[100] Middendorf, S. ; Singer, R. : *JAVA: Programmierhandbuch und Referenz für die Java2-Plattform.* Heidelberg : dpunkt-Verlag, 1999

[101] Mikucionis, M. ; Nielsen, B. ; Larsen, K. G.: Real-time System Testing On-the-fly. In: Sere, K. (Hrsg.) ; Waldén, M. (Hrsg.): *the 15th Nordic Workshop on Programming Theory.* Turku, Finland : Abo Akademi, Department of Computer Science, Finland, 2003 (B 34). – Abstracts

[102] Moore, E. F.: Gedanken-Experiments on Sequential Machines. In: *Automata Studies (Annals of Mathematics Studies)* (1956), Nr. 34

[103] Mücke, T. ; Huhn, M. : Generation of Optimized Testsuites for UML Statecharts with Time. In: Groz, R. (Hrsg.) ; Hierons, R. M. (Hrsg.): *Testing of Communicating Systems.* Berlin : Springer, March 2004. – ISBN 0302–9743

[104] Nathan, A. : *.NET and COM.* SAMS, 2002

[105] Nehmer, J. ; Sturm, P. : *Systemsoftware: Grundlagen moderner Betriebssysteme.* 1. Auflage. Heidelberg : dpunkt.verlag, 1998

[106] Nielsen, B. ; Skou, A. : Automated Test Generation from Timed Automata. In: *Tools and Algorithms for the Construction and Analysis of Systems*, 2001

[107] Nielsen, B. ; Agha, G. : Towards Reusable Real-Time Objects. J. C. Baltzer AG, Science Publishers, 1999. – Forschungsbericht

[108] Nilsen, K. : Real-Time Programming with Java Technologies. In: *Proc. of the 4th International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'01)*, IEEE Press, 2001

[109] NIST: The Economic Impacts of Inadequate Infrastructure for Software Testing / National Institute of Standards and Technology. 2002 (PR 02-3). – Planning Report

[110] Offutt, J. ; Abdurazik, A. : Generating Tests from UML Specifications. In: *UML'99*. Fort Collins, USA, 1999

[111] Offutt, J. ; Liu, S. ; Abdurazik, A. ; Ammann, P. : Generating test data from state-based specifications. In: *Software Testing, Verification and Reliability* 13 (2003), January/March, Nr. 1

[112] OMG: Guide To Model Driven Architecture / Object Management Group. 2000. – White Paper

[113] OMG: Unified Modeling Language 2.0 Proposal. OMG, 2002. – Documentation

[114] OMG: Unified Modeling Language Specification. Object Management Group, 2003. – Documentation

[115] Pahl, P. J. ; Damrath, R. : *Mathematische Grundlagen der Ingenieurinformatik*. Heidelberg : Springer, 2000

[116] Papadimitriou, C. H.: *Computational Complexity*. San Diego : Addison Wesley, 1994

[117] Peled, D. A.: *Software Reliability Methods*. Bd. ISBN 0-387-95106-7. Springer Verlag, 2001

[118] Peled, D. A. ; Edmund ; Clarke, M. ; Grumberg, O. : *Model Checking*. Boston : MIT Press, 2000

[119] Pereira, C. E.: Applying Object-Oriented Concepts to the Development of Real-Time Industrial Automation Systems. In: *Proceedings of the 3rd Workshop on Object-Oriented Real-Time Systems*, IEEE Press, 1997

[120] Perry, D. E. ; Kaiser, G. E.: Adequate testing and object-oriented programming. In: *Journal of Object-Oriented Programming* 2 (1990), Nr. 5

[121] Petrenko, A. : Fault Model-Driven Test Derivation vrom Finite State Models. In: *Proc. of Modeling and Verification of Parallel Processes MOVEP '2P*. Nantes : Springer, June 2000 (LNCS 2067)

[122] Petrenko, A. ; Boroday, S. ; Groz, R. : Confirming Configurations in EFSM Testing. In: *IEEE Transactions on Software Engineering* 30 (2004), January, Nr. 1

[123] Petrenko, A. ; von Bochmann, G. ; Yao, M. Y.: On Fault Coverage of Tests for Finite State Specifications. In: *Computer Networks and ISDN Systems* 29(1) (1996)

[124] Petrenko, A. ; Yevtushenko, N. ; Bochmann, G. V.: Fault Models for Testing in Context. In: *FORTE*. Kaiserslautern, Germany : Kluwer, 1996 (IFIP Conference Proceedings)

[125] Pnueli, A. : A Temporal Logic of Programs. In: *18th IEEE Symposium on Foundation of Computer Science*, IEEE Computer Society Press, 1977

[126] Pretschner, A. : Classical search strategies for test case generation with Constraint Logic Programming. In: *FATES*. Aalborg, Denmark, 2001

[127] Pretschner, A. ; Lötzebeyer, H. : Model based testing with constraint logic programming. In: *Workshop on Automated Program Analysis, Testing and Verification (WAPATV)*, 2001

[128] Puschner, P. ; Wellings, A. : A Profile for High-Integrity Real-Time Java Programs. In: *Proc. of the 4th International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'01)*, IEEE Press, 2001

[129] Quielle, J. P. ; Sifakis, J. : Specification and Verifikation of Concurrent Systems in CESAR. In: *Proc. of the 5th International Symposium on Programming*, 1981

[130] Rayadurgan, S. ; Heimdahl, M. : Coverage Based Test-Case Generation using Model Checkers. In: *Intl. Conf. and Workshop on the Engineering of Computer Based Systems*, 2001

[131] Robinson-Mallett, C. : ROOMtest: Structural Module Testing based on ROOMcharts using Rational Rose Realtime. University of Kent : Fortest, April 2004. – Presentation

[132] Robinson-Mallett, C. : Structural Testing based on ROOMcharts and UML-RT Statecharts. In: *Position Paper Proceedings TestCom 2004, Oxford*, 2004

[133] Robinson-Mallett, C. : ROOMtest: An Approach for the Structural Testing of ROOMcharts and UML-RT State Diagrams. In: *Doctoral Symposium Proceedings of the 4th International Conference on Integrated Formal Methods (IFM 2004)*. Canterbury : University of Kent, April 2005

[134] Robinson-Mallett, C. ; Mücke, T. : Complexity Issues of Generating Test-Cases from Statecharts with Uppaal. Uxbridge : Fortest, October 2004. – Presentation

[135] Robinson-Mallett, C. ; Mücke, T. ; Liggesmeyer, P. : Generating Optimal Distinguishing Sequences with a Model Checker. In: *ICSE Workshop on Advances in Model Based Software Testing (A-MOST)*. St. Louis : ACM, May 2005

[136] RTCA: Software Consideration in Airborne Systems and Equipment Certification / Radio Technical Commission for Aeronautics. Washington, USA, 1992 (DO-178B). – Guideline

[137] Sabnani, K. ; Dahbura, A. : A new Technique for Generating Protocol Tests. In: *ACM Computer Communication Review* 15 (1985), September, Nr. 4

[138] Sabnani, K. ; Dahbura, A. : A Protocol Test Generation Procedure. In: *Computer Networks and ISDN systems* 15 (1988)

[139] Saksena, M. ; Freedman, P. ; Rodziewicz, P. : Guidelines for Automated Implementation of Executable Object-Oriented Models for Real-Time Embedded Control Systems. In: *Proceedings of the 18th IEEE Real-Time Systems Symposium*, IEEE Computer Society, December 1997

[140] Saksena, M. ; Freedman, P. ; Rodziewicz, P. : Guidelines for Automated Implementation of Executable Object-Oriented Models for Real-Time Embedded Control Systems. In: *Proc. of the IEEE Real-Time Systems Symposium*, IEEE Computer Society, December 1997

[141] Saksena, M. ; Ptak, A. ; Freedman, P. ; Rodziewicz, P. : Schedulability Analysis for Automated Implementations of Real-Time Object-Oriented Models. In: *Proc. of the IEEE Real-Time Systems Symposium*, IEEE Computer Society, 1998

[142] Sari, B. ; v. Bochmann, G. ; Cerny, E. : A Test Design Methodology for Protocol Testing. In: *IEEE Transactions on Software Engineering* 13 (1987), Nr. 5

[143] Selic, B. ; Gullekson, G. ; Ward, P. T.: *Real-Time Object-Oriented Modeling*. Wiley, 1994

[144] Shaw, A. C.: *Real-Time Systems and Software*. Washington : Wiley, 2001

[145] Sidhu, D. ; kau Leung, T. : Experience with test generation for real protocols. In: *Symposium proceedings on Communications architectures and protocols*, ACM Press, August 1988

[146] Sidhu, D. ; Leung, T.-K. : Formal methods for protocol testing: a detailed study. In: *IEEE Transactions on Software Engineering* 15 (1989), April, Nr. 4

[147] Siebert, F. : The Impact of Realtime Garbage Collection on Realtim Java Programming. In: *Proc. of the 7th International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'04)*, IEEE Press, 2004

[148] Simons, A. J. H.: On the Compositional Properties of UML Statecharts Diagrams. In: Clark, A. N. (Hrsg.) ; Evans, A. (Hrsg.) ; Lano, K. (Hrsg.): *Proceedings of the 3rd. Conf. Rigorous Object-Oriented Methods*, 2000

[149] Sistla, A. P. ; Clarke, E. M.: Complexity of Propositional Temporal Logics. In: *Journal of the ACM* 32 (3) (1985)

[150] Stroustrup, B. : *The C++ Programming Language.* Boston, Mass. : Addison-Wesley, 2000

[151] Sun, D. ; Vinnakota, B. ; Jiang, W. : Fast state verification, ACM Press, 1998

[152] Tretmans, J. : Test Generation with Inputs, Outputs and Repetitive Quiescence. In: *Software – Concepts and Tools* 17 (1996), Nr. 3

[153] Tretmans, J. : Testing Concurrent Systems: A Formal Approach. In: *10th International Conference on Concurrency Theory(CONCUR'99)* Bd. 1664, Springer-Verlag, 1999

[154] Turner, C. D. ; Robson, D. : A State-Based Approach to the Testing of Class-Based Programs. In: *Software-Concepts and Tools* 16 (1995)

[155] Ural, H. : Formal methods for test sequence generation. In: *Computer Communications* 15 (1992), June, Nr. 5

[156] Ural, H. ; Wu, X. ; Zhang, F. : On Minimizing the Length of Checking Sequences. In: *IEEE Transactions on Computers* 46 (1997)

[157] Ümit Uyar, M. ; Fecko, M. A. ; Sethi, A. S. ; Amer, P. D.: Testing Protocols Modeled as FSMs with Timing Parameters. In: *Computer Networks* 31 (1999)

[158] Valmari, A. : A Stubborn Attack on State Explosion. In: *LNCS 531: Proceedings of the 16th International Colloquium on Automata, Languages, and Programming*, Springer, 1990

[159] Vasilevskii, M. P.: Failure Diagnosis of Automata. In: *Kibernetika* (1973), Nr. 4

[160] Vuong, S. T. ; Chan, W. ; Ito, M. R.: The UIOv-Method for Protocol Test Sequence Generation. In: *Proceedings of 2nd IFIP Workshop on Protocol Test Systems (IWPTS'89)*, 1989

[161] Yevtushenko, N. ; Lebedev, A. ; Petrenko, A. : On the Checking Experiments with Nondeterministic Automata. In: *Automatic Control and Computer Sciences* 25 (1991), Nr. 6

[162] Yovine, S. : Kronos: A Verification Tool for Real-Time Systems. In: *Journal on Software tools for Technology Transfer* (1997), October

[163] Zariopulo, P. : A new Approach to Protocol Validation. In: *Proceedings of the International Communications Conference Volume II (ICC77).* Chicago, 1977

Anhang A

Mathematische Grundlagen

A.1 Graphen

Die folgenden graphentheoretischen Grundlagen sind [115] entnommen.

A.1.1 Schlichte Graphen

Ein Gebilde *G=(V;R)* heißt schlichter, gerichteter Graph, wenn V die Knotenmenge und $R \subseteq V \times V$ die Kantenmenge des Graphen ist. Eine Kante vom Knoten $x \in V$ zum Knoten $y \in V$ wird mit dem geordneten Paar $(x, y) \in R$ bezeichnet. Die Kante *(x,y)* heißt von x nach y gerichtet. Der Knoten x heißt Anfangsknoten der Kante. Der Knoten y heißt Endknoten der Kante.

A.1.2 Zyklen in schlichten Graphen

Ein nicht leerer Weg, dessen Anfangs- und Endknoten identisch sind, heißt Zyklus. Eine Schlinge eines Knoten ist ein Zyklus der Länge 1. Ein Zyklus, der keine Schlingen besitzt, heißt echter Zyklus. Existieren ein nicht leerer Weg von x nach y und ein nicht leerer Weg von y nach x, so liefert die Verkettung beider Wege einen Zyklus durch x und y.

A.1.3 Äquivalente Zyklen

Zwei Zyklen a und b sind äquivalent, wenn jeder Weg in a auch ein Weg in b ist und jeder Weg in b auch ein Weg in a ist.

A.1.4 Multigraphen

Ein Gebilde *G=(V,K;A,B)* heißt Multigraph, wenn V die Knotenmenge, K die Kantenmenge und $A, B \subseteq K \times V$ rechtseindeutige, binäre Relationen sind. Die Relation A spezifiziert die Anfangsknoten der Kanten und heißt *Ausgangsinzidenz*. Die Relation B spezifiziert die *Endknoten* der Kanten und heißt *Eingangsinzidenz*.

A.2 Endliche Zustandsautomaten

Die Definitionen und die Theorie endlicher Zustandsautomaten sind [55] entnommen. Ein endlicher Zustandsautomat M ist ein synchrones System mit einem endlichen Eingabealphabet $X = \{\xi_1, \xi_2, ..., \xi_p\}$, einem endlichen Ausgabealphabet $Z = \{\zeta_1, \zeta_2, ..., \zeta_p\}$, einer endlichen Menge Zustände $S = \{\sigma_1, \sigma_2, ..., \sigma_p\}$ und einem Paar von charakteristischen Funktionen f_z und f_s. Die charakteristischen Funktionen werden definiert durch:

$z_\nu = f_\nu(x_\nu, s_\nu)$

$s_{\nu+1} = f_s(x_\nu, s_\nu)$

wobei Eingabesymbol x_ν, Zustand s_ν und Ausgabesymbol z_ν auf einen Zeitpunkt $t_\nu (\nu = 1, 2, ...)$ bezogen sind.

Die Struktur eines endlichen Zustandsautomaten kann zweckmäßig durch einen Multigraphen oder schlichte Graphen dargestellt werden. Wenn die Zustände und Transitionen bijektiv auf die Menge der Knoten bzw. die Menge der Kanten abgebildet werden, existiert für jeden Zustand und jede Transition genau ein Knoten bzw. genau eine Kante im Multigraphen. Wenn die Zustände bijektiv auf die Menge der Knoten abgebildet werden und die Transition zwischen den Zuständen x und y auf genau eine Kante abgebildet werden, existiert ein schlichter Graph.

äquivalente Zustände Ein Zustand σ_i eines Automaten M_1 und ein Zustand σ_j eines Automaten M_2 werden *äquivalent* genannt, wenn M_1 in σ_i und M_2 in σ_j, dargestellt durch $M_1|\sigma_i$ bzw. $M_2|\sigma_j$, auf jede Folge von Eingaben die gleichen Ausgaben liefern. Nicht äquivalente Zustände werden unterscheidbar genannt. M_1 und M_2 können auf denselben Automaten verweisen. Wenn $M_1|\sigma_i$ und $M_2|\sigma_j$ für alle Eingabesequenzen der Länge k äquivalent sind, werden sie k-äquivalent genannt. Die k-äquivalenten Zustände sind auch l-äquivalent für alle $l \leq k$, können aber unterscheidbar sein für $l>k$. Alle Zustände, die durch die Eingabesequenzen der Länge k unterscheidbar sind, werden k-unterscheidbar genannt.

quasi-äquivalente Zustände Die Zustände $\sigma_i$und σ_j werden *quasi-äquivalent* genannt, wenn jede in $M_1|\sigma_i$ akzeptierte Eingabesequenz in $M_2|\sigma_j$ die gleichen Ausgaben bietet.

A.2.1 Konformität und Äquivalenz

Zwei endliche Zustandsautomaten M_1 mit der Zustandsmenge S_1 und M_2 mit der Zustandsmenge S_2 sind äquivalent, wenn für jeden Zustand $\sigma_i \in S_1$ ein äquivalenter $\sigma_j \in S_2$ existiert, und wenn für jeden Zustand $\sigma_j \in S_2$ ein äquivalenter Zustand $\sigma_i \in S_1$ existiert.

Die endlichen Zustandsautomaten M_1 und M_2 sind quasi-äquivalent, wenn für jeden Zustand $\sigma_i \in S_1$ ein quasi-äquivalenter Zustand $\sigma_j \in S_2$ existiert.

strenge Konformität Zwei äquivalente, endliche Zustandsautomaten werden *streng konform* genannt.

schwache Konformität Zwei quasi-äquivalente, endliche Zustandsautomaten werden *schwach konform* genannt.

A.2.2 Minimale endliche Zustandsautomaten

Ein endlicher Zustandsautomat, der keine äquivalenten Zustände besitzt, wird *minimal* genannt. Für die Minimierung von Zustandsautomaten kann beispielsweise der in [55] verwendete Algorithmus verwendet werden.

Anhang B

XML Schemata

EFSM

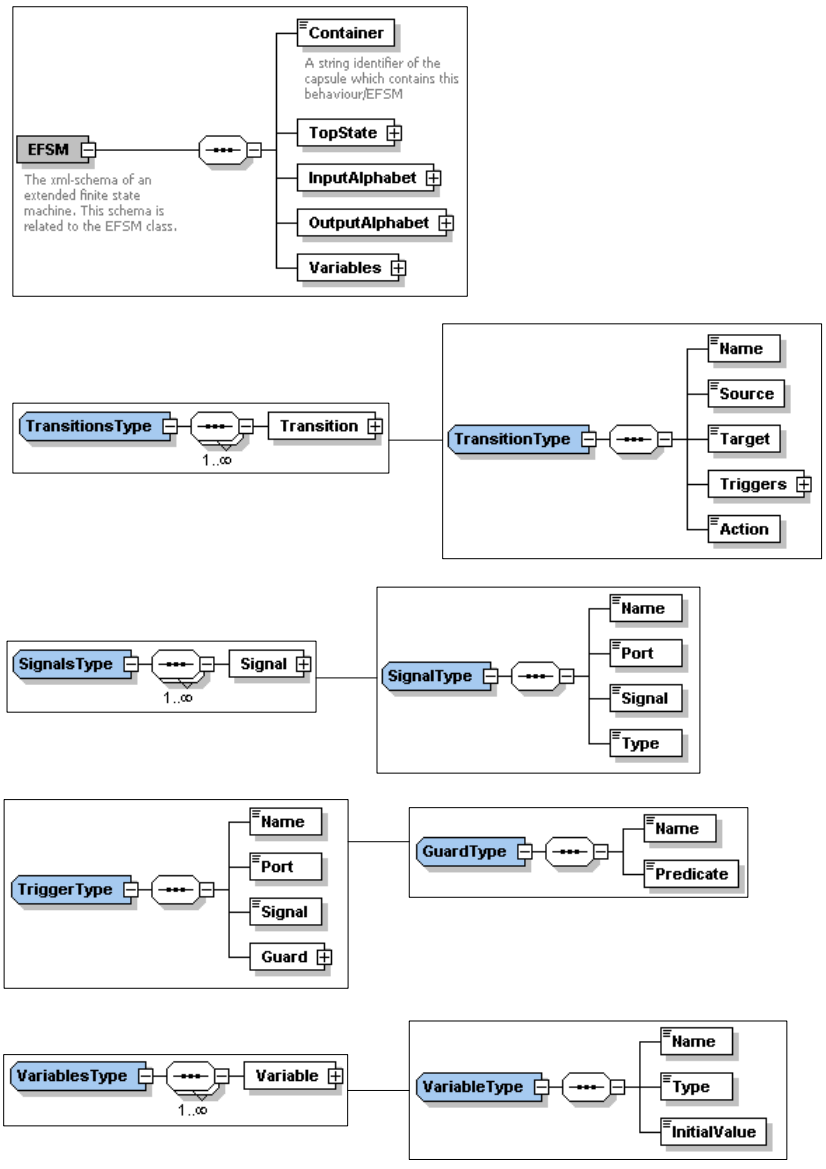


Abbildung B.1: EFSM XML Schema

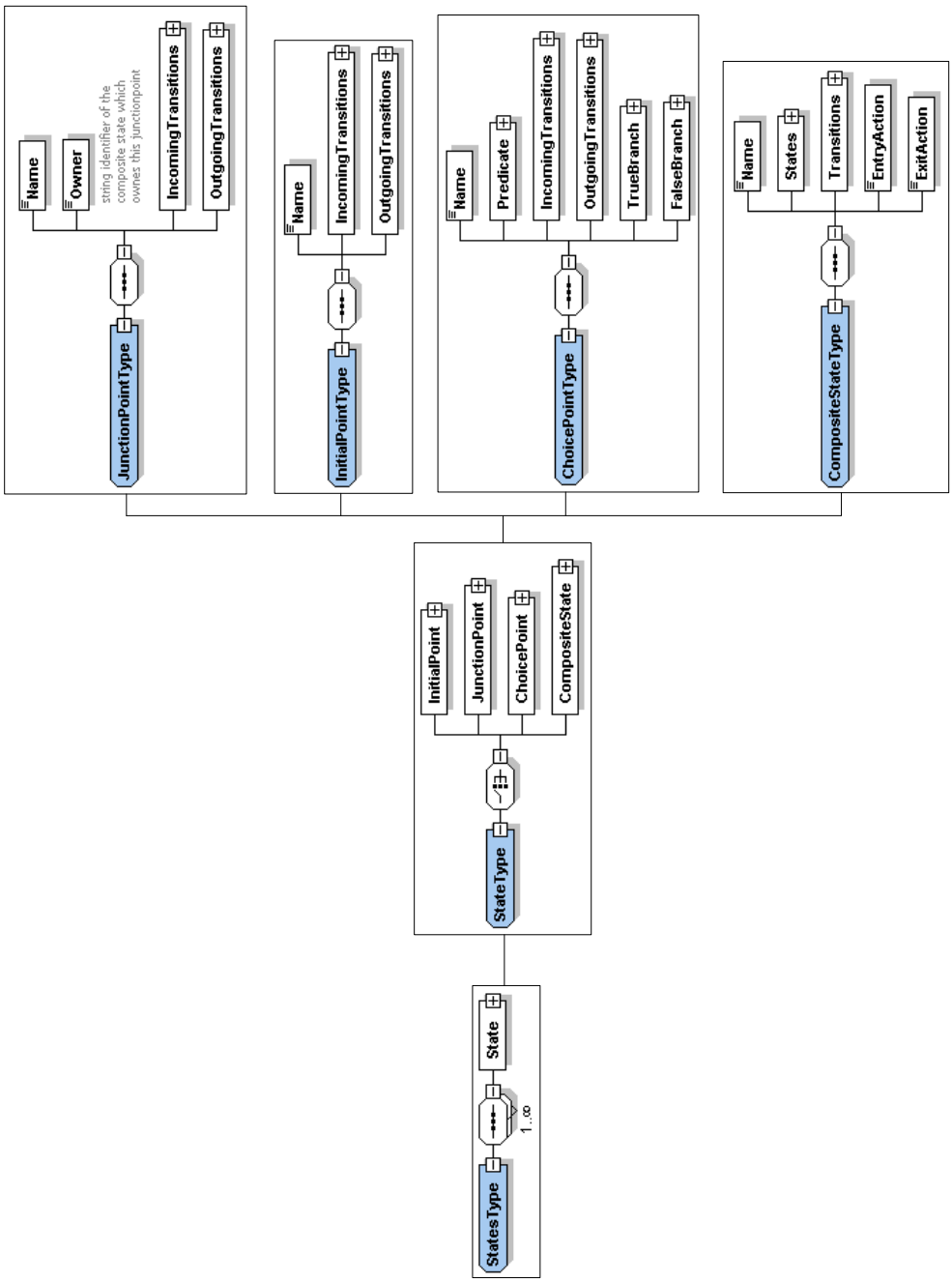


Abbildung B.2: EFSM XML Schema

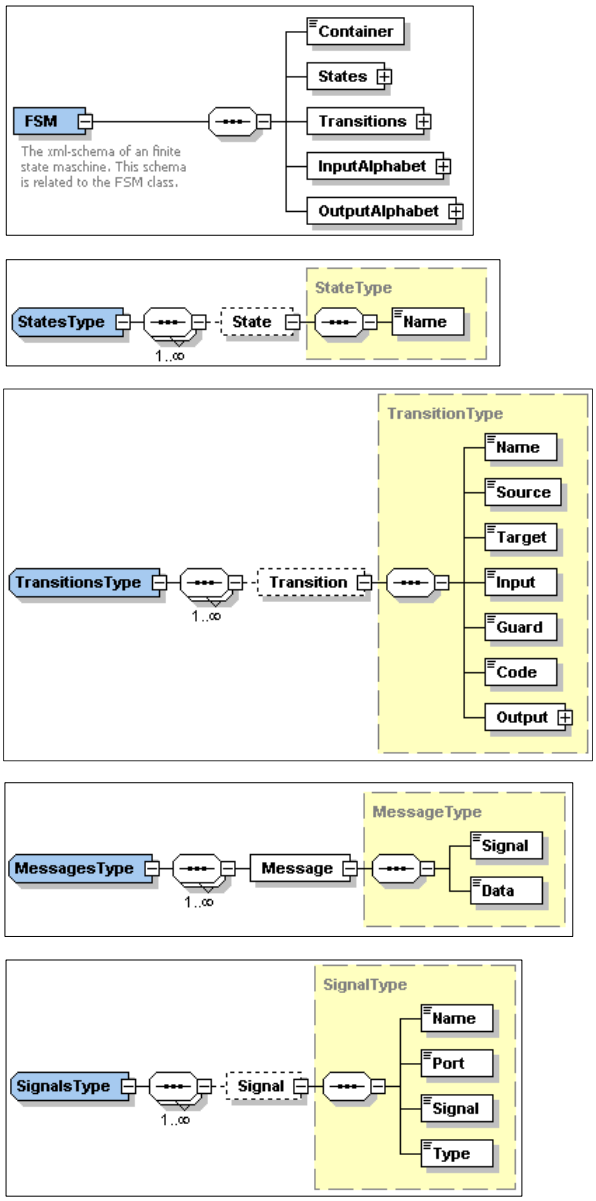


Abbildung B.3: EFSM XML Schema

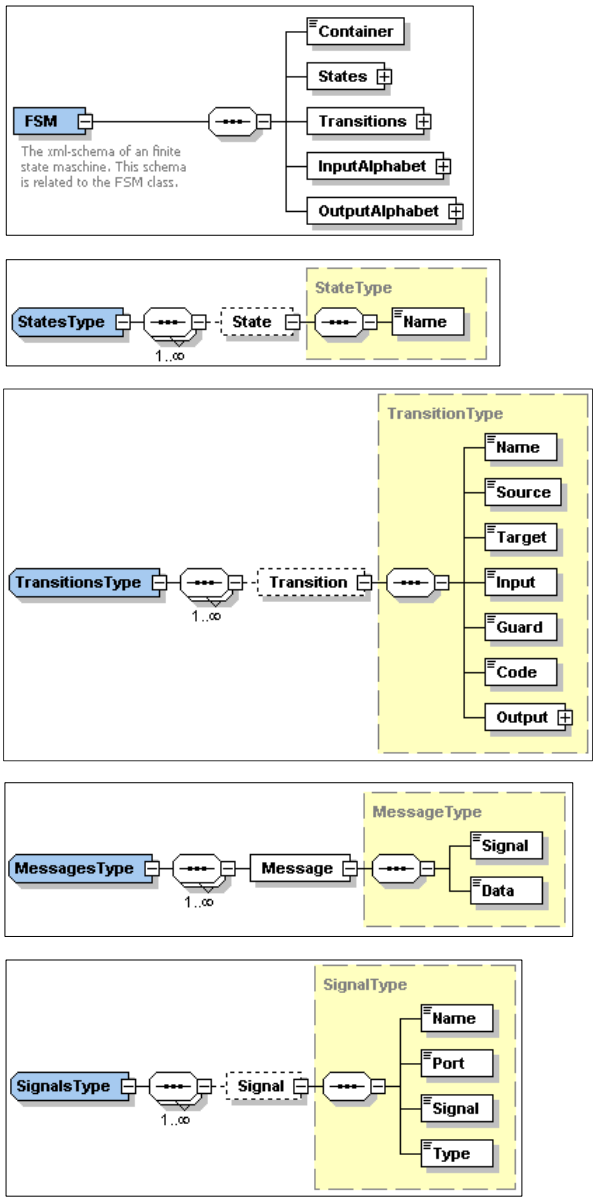


Abbildung B.4: FSM XML Schema

Anhang C

Beispiele

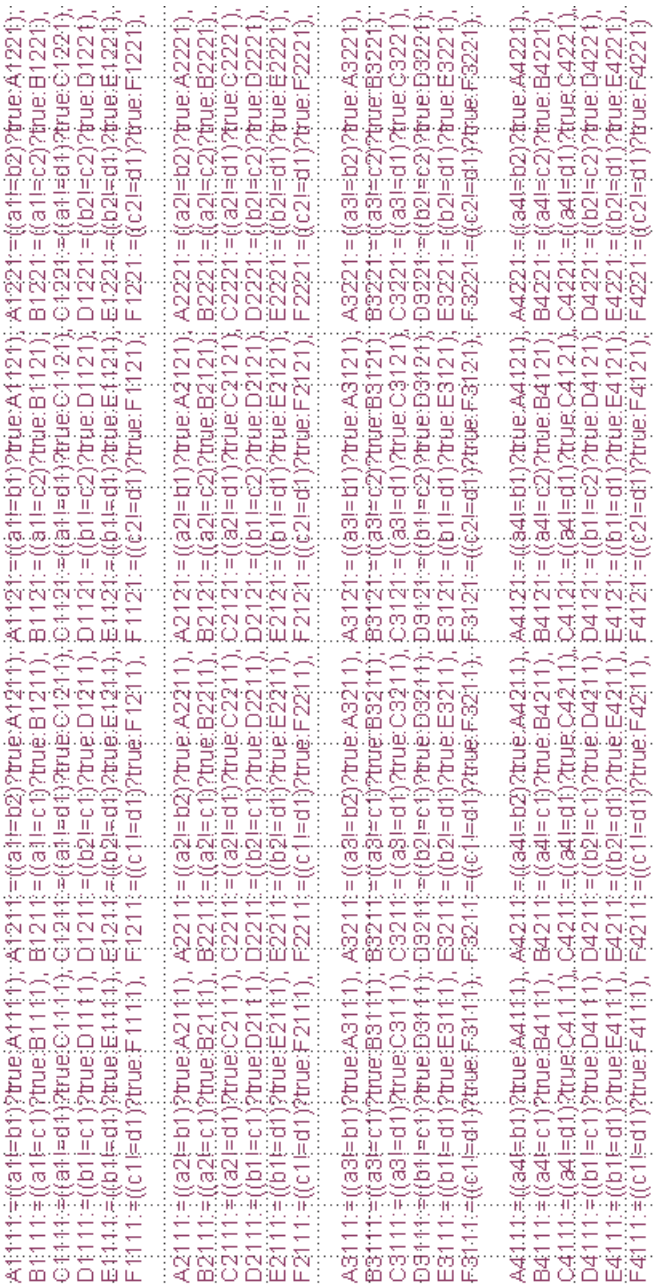


Abbildung C.1: Auswertung der Ausgabevariablen für Clock

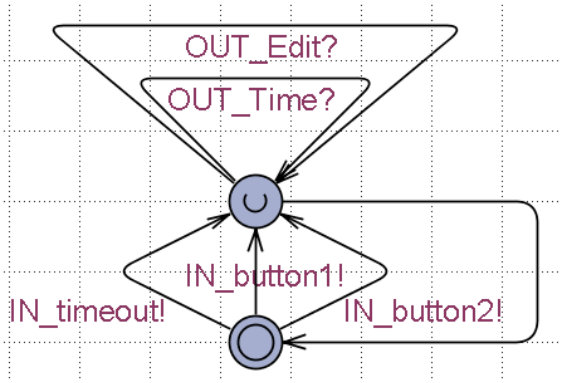


Abbildung C.2: DS-Treiberautomat für Clock

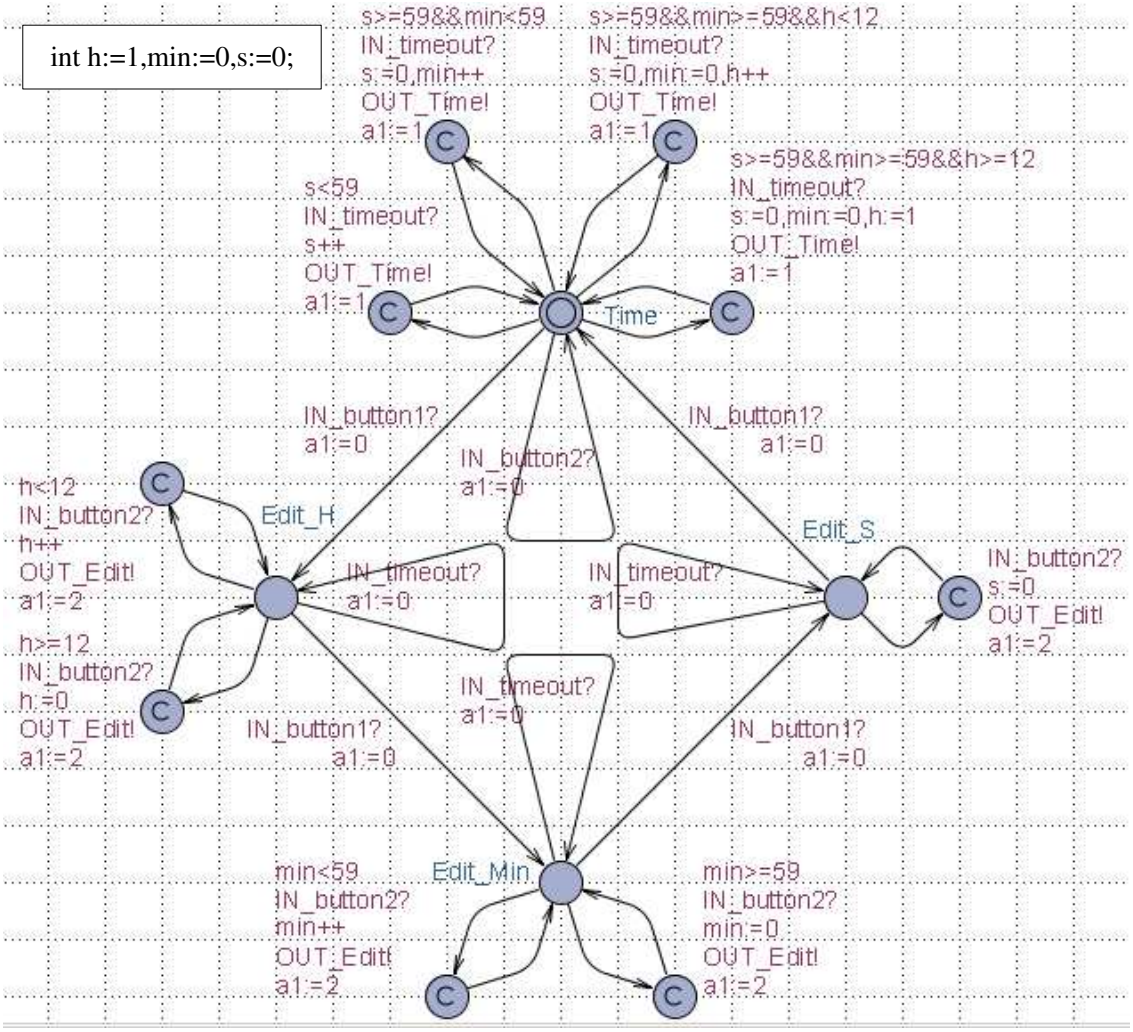


Abbildung C.3: DS-Edit_H-Automat für Clock

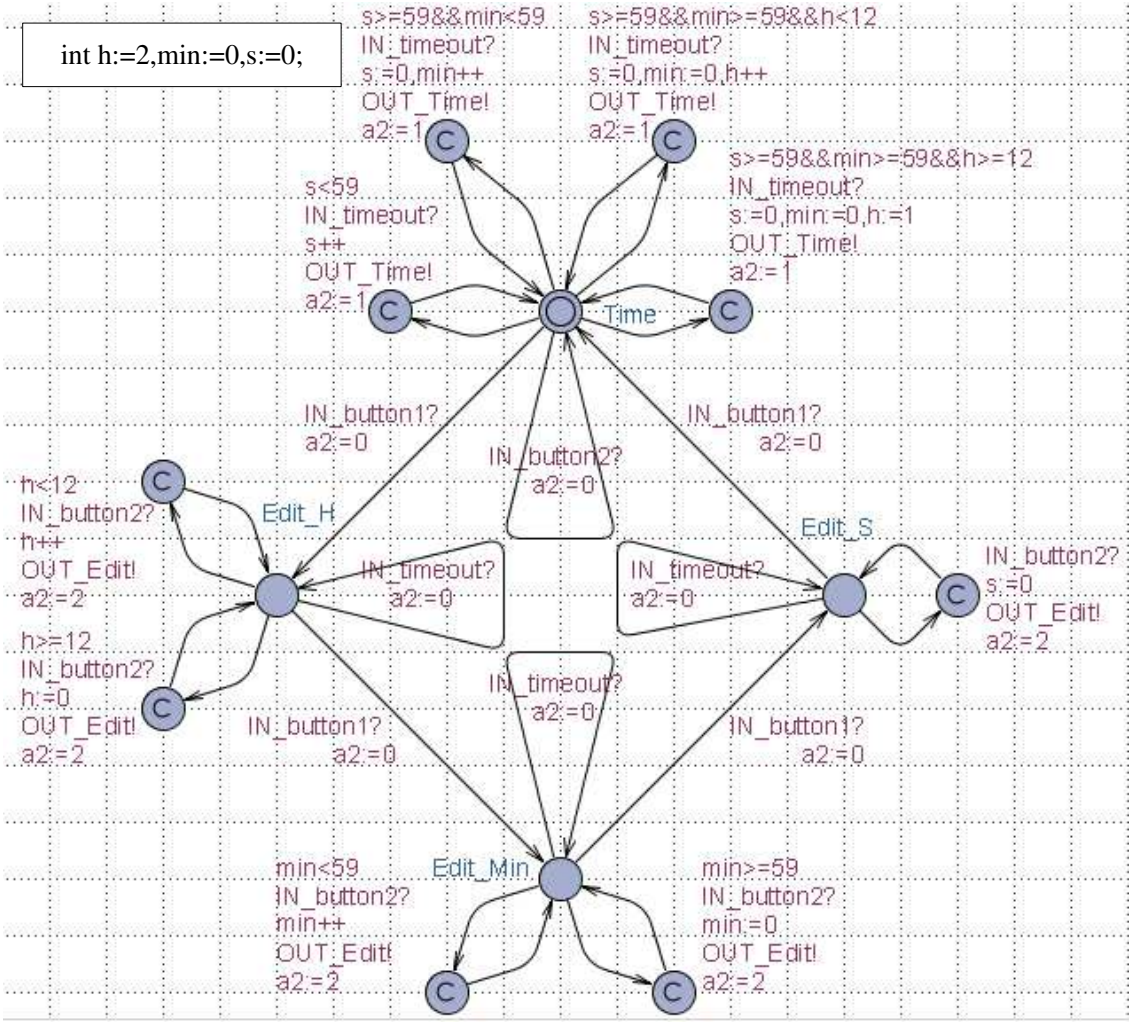


Abbildung C.4: DS-Edit_H-Automat für Clock

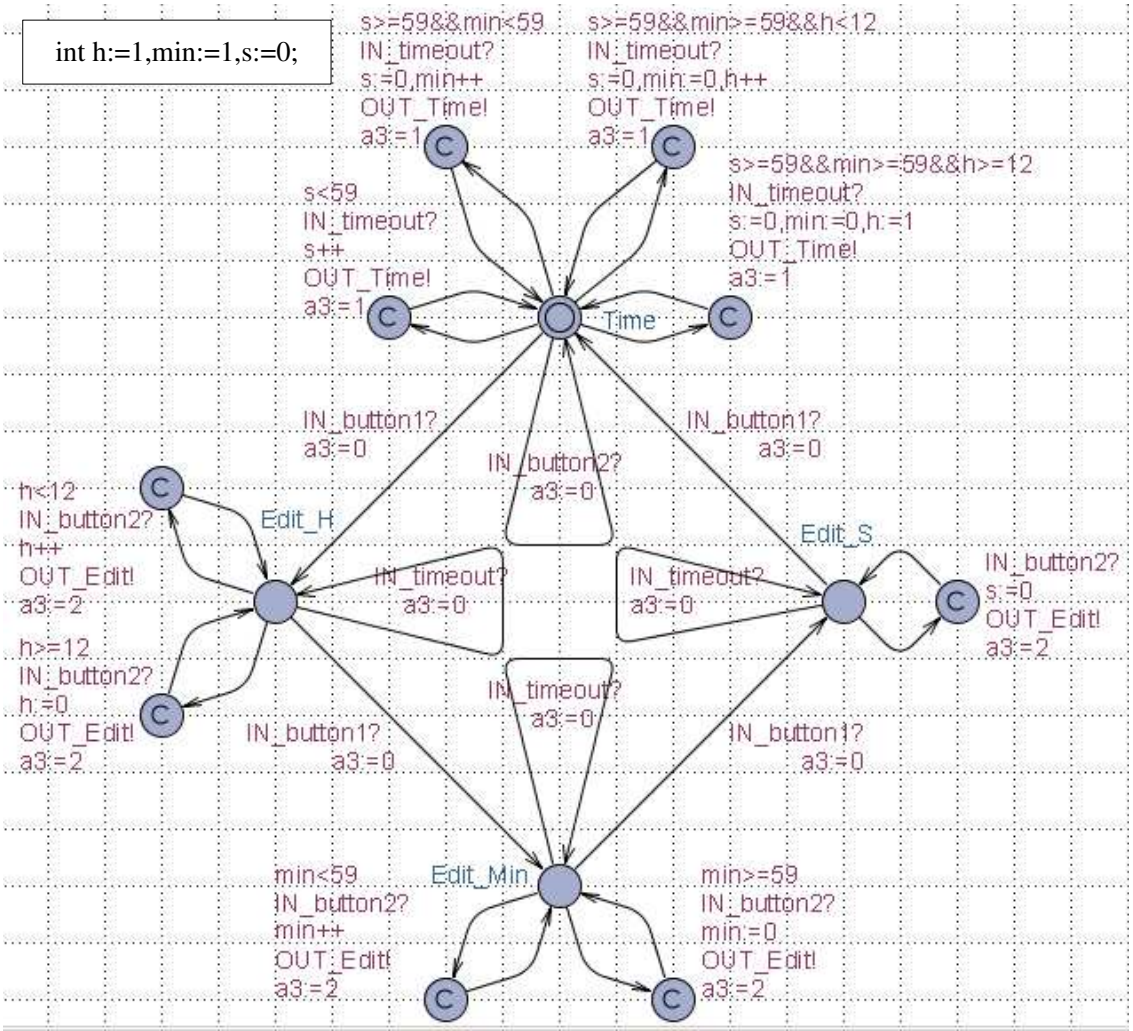


Abbildung C.5: DS-Edit_H-Automat für Clock

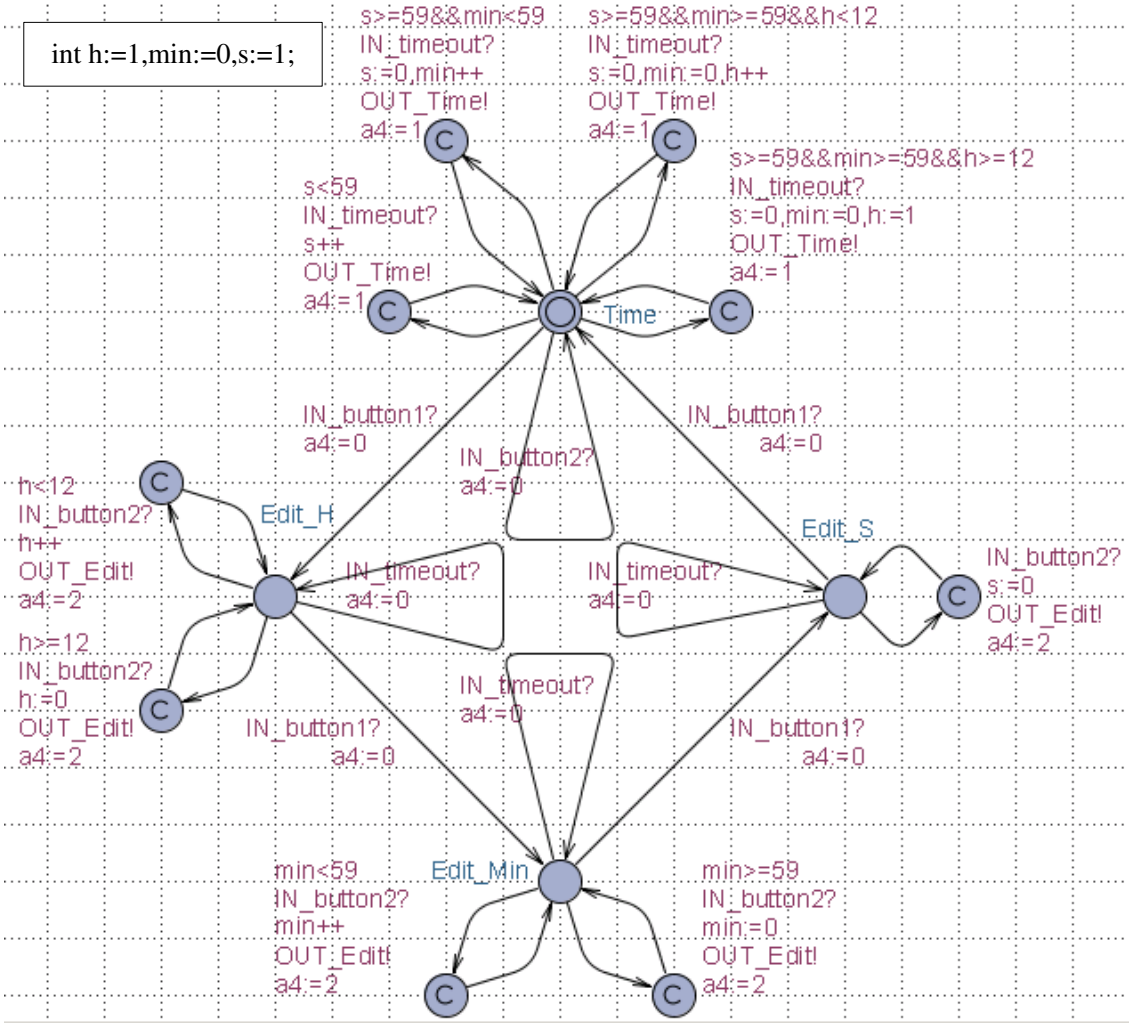


Abbildung C.6: DS-Edit_H-Automat für Clock

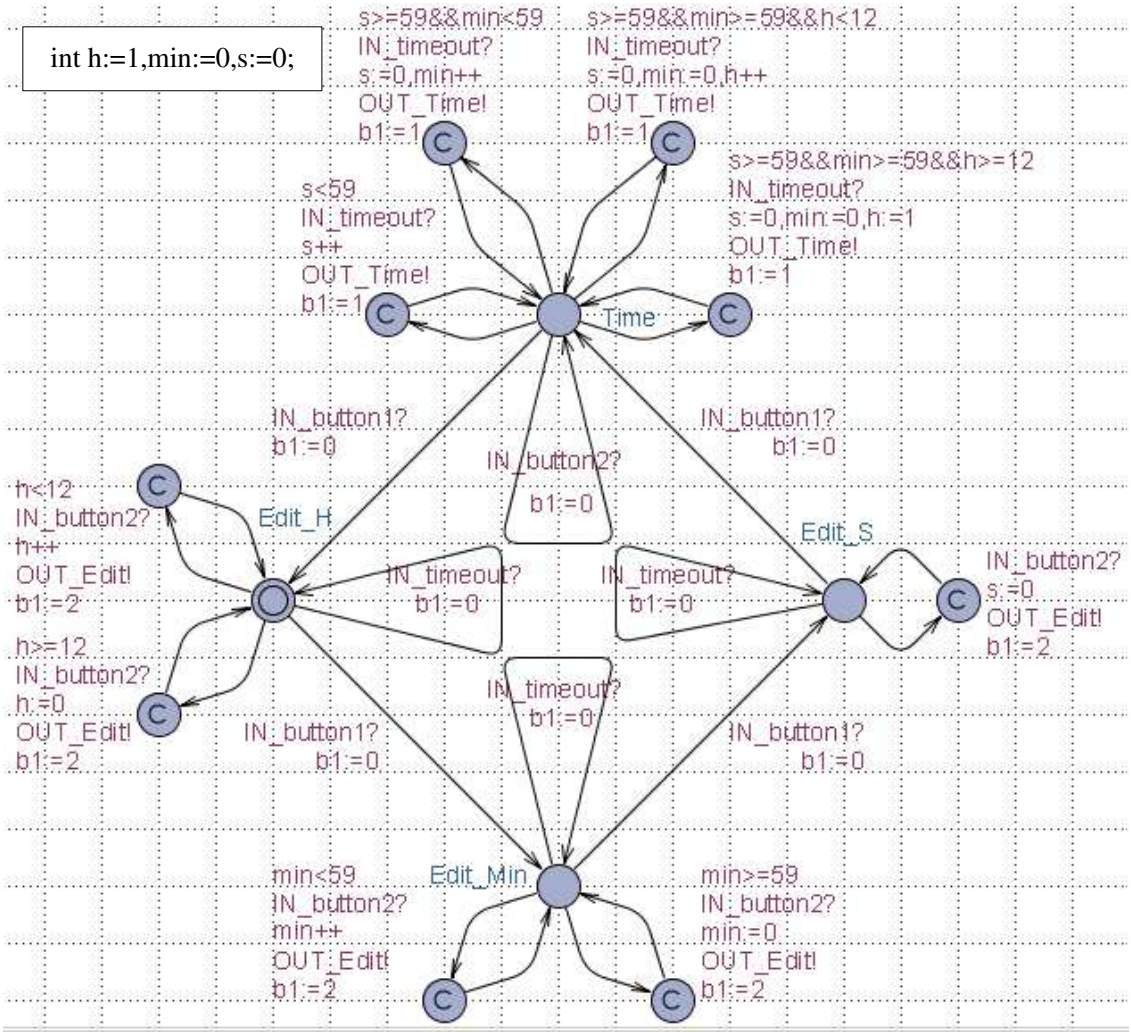


Abbildung C.7: DS-Edit_H-Automat für Clock

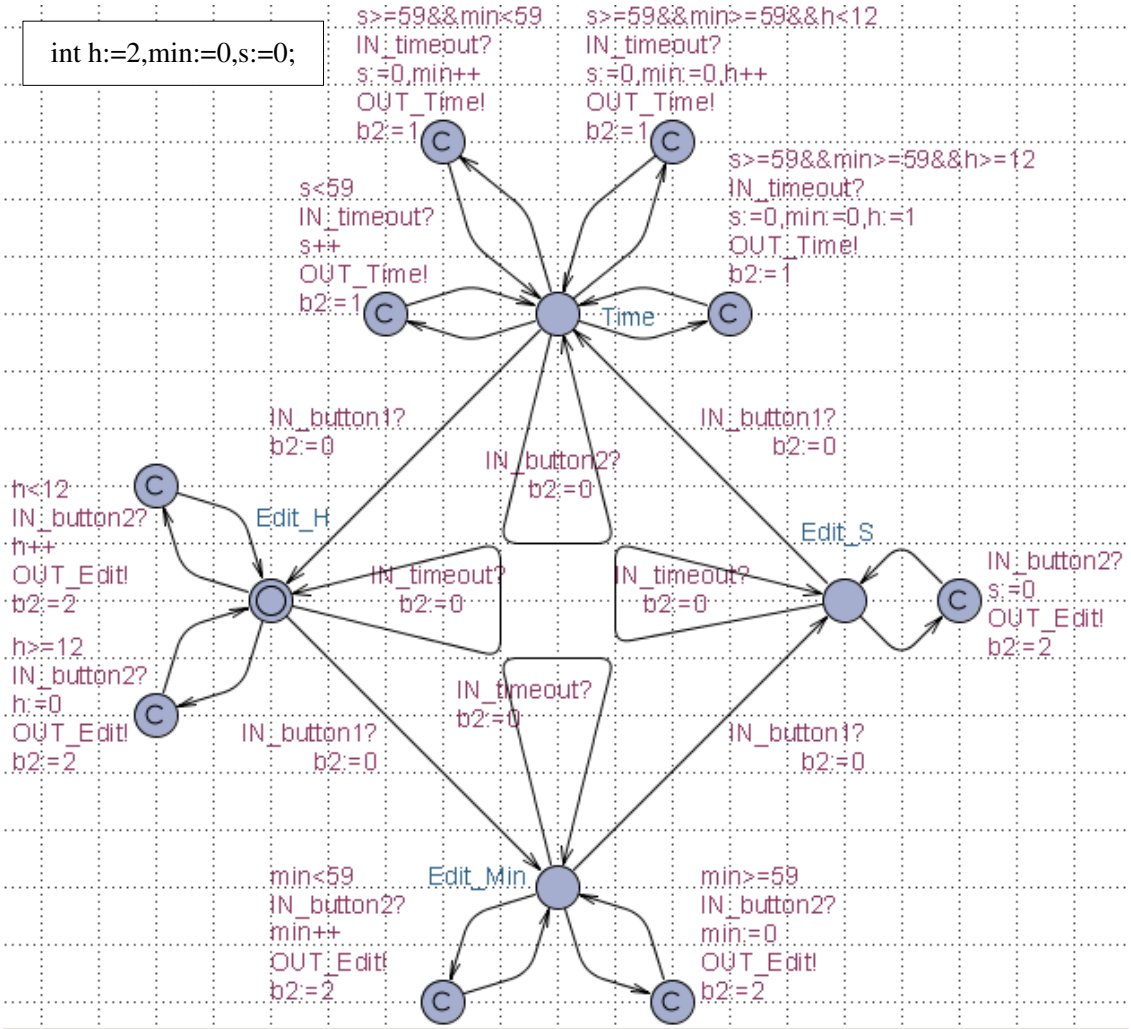


Abbildung C.8: DS-Edit_H-Automat für Clock

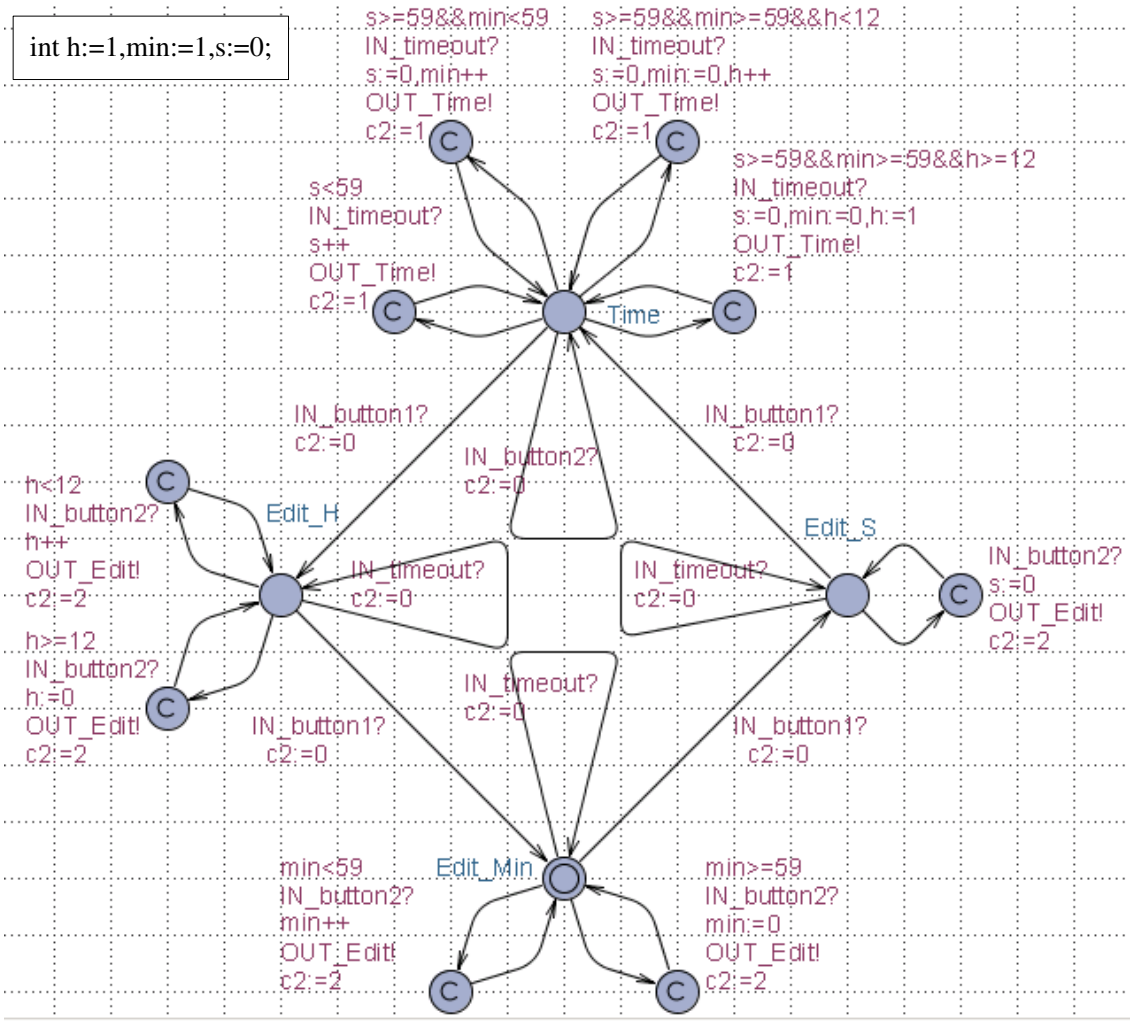


Abbildung C.9: DS-Edit_Min-Automat für Clock

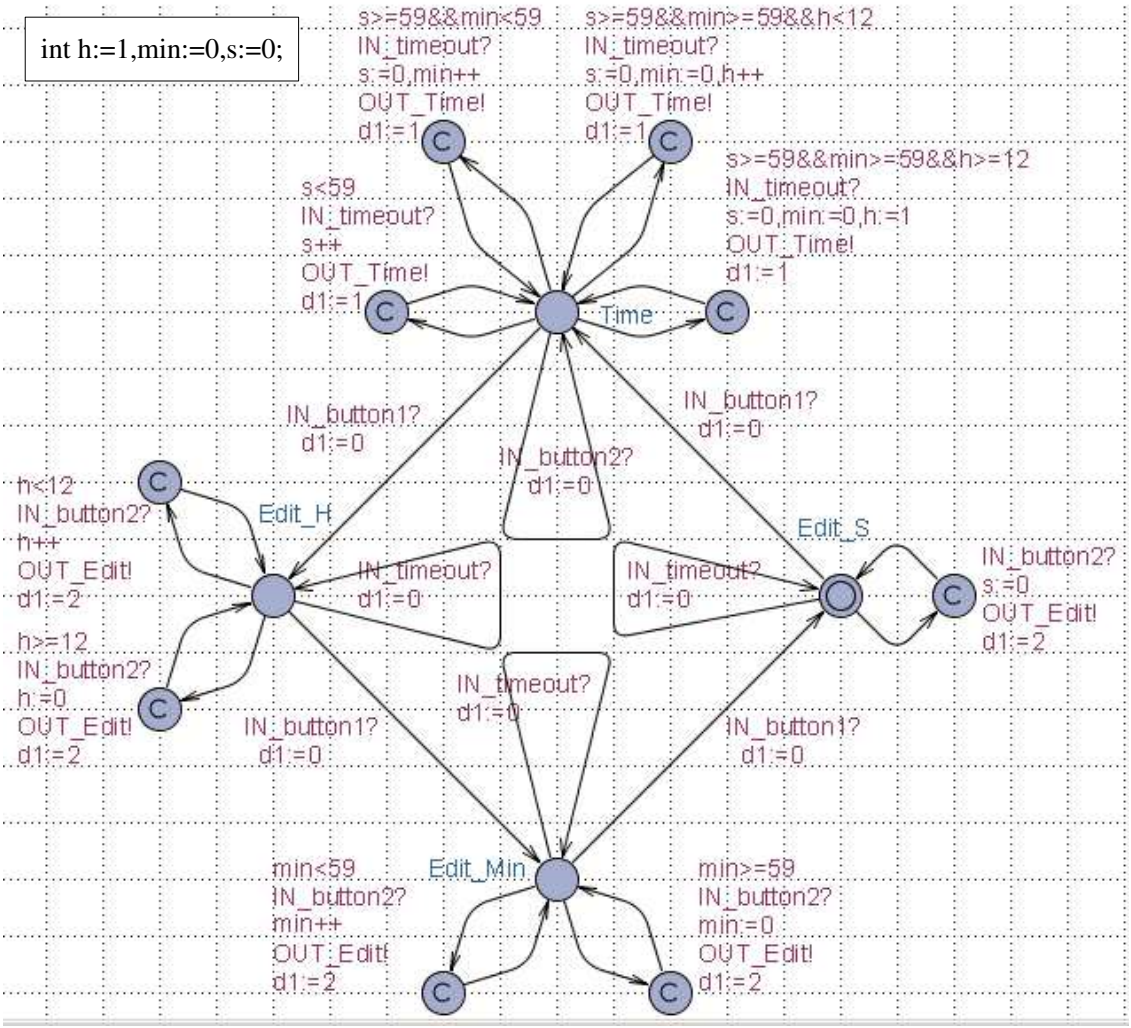


Abbildung C.10: DS-Edit_S-Automat für *Clock*

```
E<>
A1111 && B1111 && C1111 && D1111 && E1111 && F1111 &&
A2111 && B2111 && C2111 && D2111 && E2111 && F2111 &&
A3111 && B3111 && C3111 && D3111 && E3111 && F3111 &&
A4111 && B4111 && C4111 && D4111 && E4111 && F4111 &&

A1211 && B1211 && C1211 && D1211 && E1211 && F1211 &&
A2211 && B2211 && C2211 && D2211 && E2211 && F2211 &&
A3211 && B3211 && C3211 && D3211 && E3211 && F3211 &&
A4211 && B4211 && C4211 && D4211 && E4211 && F4211 &&

A1121 && B1121 && C1121 && D1121 && E1121 && F1121 &&
A2121 && B2121 && C2121 && D2121 && E2121 && F2121 &&
A3121 && B3121 && C3121 && D3121 && E3121 && F3121 &&
A4121 && B4121 && C4121 && D4121 && E4121 && F4121 &&

A1221 && B1221 && C1221 && D1221 && E1221 && F1221 &&
A2221 && B2221 && C2221 && D2221 && E2221 && F2221 &&
A3221 && B3221 && C3221 && D3221 && E3221 && F3221 &&
A4221 && B4221 && C4221 && D4221 && E4221 && F4221
```

Abbildung C.11: DS-Anforderungen für *Clock*

$P \circ DS$
$< timeout, button1, timeout, button1, timeout, reset,$
$button1, timeout, button1, timeout, button1, timeout, reset,$
$button1, button2, timeout, button1, timeout, button1, timeout, reset,$
$button1, button2^{12}, timeout, button1, timeout, button1, timeout, reset,$
$button1, button1, timeout, button1, timeout, button1, timeout, reset,$
$button1, button1, button2, timeout, button1, timeout, button1, timeout, reset,$
$button1, button1, button2^{60}, timeout, button1, timeout, button1, timeout, reset,$
$button1, button1, button1, timeout, button1, timeout, button1, timeout, reset,$
$button1, button1, button1, button2, timeout, button1, timeout, button1, timeout, reset,$
$button1, button1, button1, button1, timeout, button1, timeout, button1, timeout, reset,$
$timeout, timeout, button1, timeout, button1, timeout, reset,$
$timeout^{60}, timeout, button1, timeout, button1, timeout, reset,$
$button1, button1, button2^{59}, button1, button1, timeout^{60}, timeout, button1, timeout, button1, timeout, reset,$
$button1, button2^{11}, button1, button2^{59}, button1, button1, timeout^{60}, timeout, button1, timeout, button1, timeout >$

Tabelle C.1: Test-Eingabesequenz *Clock*

$Ausgabe(P \circ DS)$
$< time, -, -, -, -, -,$
$-, -, -, -, -, -,$
$-, edit, -, -, -, -,$
$-, edit(h <= 12)^{11}, -, -, -, -,$
$-, -, -, -, -, time, -,$
$-, -, edit, -, -, -, time, -,$
$-, -, edit(min \leq 59)^{59}, -, -, -, time, -,$
$-, -, -, -, time, -, -,$
$-, -, -, edit, -, time, -, -,$
$-, -, -, time, -, -, -,$
$time, time, -, -, -, -,$
$time(s \leq 59)^{59}, time(min = 1), time, -, -, -,$
$-, -, edit(min \leq 59)^{59}, -, -, time(s \leq 59), time(h = 2), time, -, -, -,$
$-, edit(h <= 12)^{11}, -, edit(min \leq 59)^{59}, -, -, time(s \leq 59)^{59}, time(h = 0, min = 0, s = 0), time, -, -, -, - >$

Tabelle C.2: Transitionüberdeckung *Clock* ohne Instrumentierung

Nr.	coverage	Eingaben	Endzustand	Variablenwerte
I	0	$< button1 >$	*Edit_ H*	h=1,min=0,s=0
II	1	$< button1, button2 >$	*Edit_ H*	h=2,min=0,s=0
III	2	$< button1, button2^{12} >$	*Edit_ H*	h=1,min=0,s=0
IV	3	$< button1, button1 >$	*Edit_ Min*	h=1,min=0,s=0
V	4	$< button1, button1, button2 >$	*Edit_ Min*	h=1,min=1,s=0
VI	5	$< button1, button1, button2^{60} >$	*Edit_ Min*	h=1,min=0,s=0
VII	6	$< button1, button1, button1 >$	*Edit_ S*	h=1,min=0,s=0
VIII	7	$< button1, button1, button1, button2 >$	*Edit_ S*	h=1,min=0,s=0
IX	8	$< button1, button1, button1, button1 >$	*Time*	h=1,min=0,s=0
X	9	$< timeout >$	*Time*	h=1,min=0,s=1
XI	10	$< timeout^{60} >$	*Time*	h=1,min=1,s=0
XII	11	$< button1, button1, button2^{59}, button1, button1, timeout^{60} >$	*Time*	h=2,min=0,s=0
XIII	12	$< button1, button2^{11}, button1, button2^{59}, button1, button1, timeout^{60} >$	*Time*	h=1,min=0,s=0

Tabelle C.3: Transitionüberdeckung *Clock* ohne Instrumentierung

	coverage	Eingaben	Endzustand	Variablenwerte
I	0	$< button1 >$	$Edit\_H$	h=1,min=0,s=0
II	1	$< button1, button2 >$	$Edit\_H$	h=2,min=0,s=0
III	2	$< button1, setH(11), button2 >$	$Edit\_H$	h=1,min=0,s=0
IV	3	$< button1, button1 >$	$Edit\_Min$	h=1,min=0,s=0
V	4	$< button1, button1, button2 >$	$Edit\_Min$	h=1,min=1,s=0
VI	5	$< button1, button1, setMin(59), button2 >$	$Edit\_Min$	h=1,min=0,s=0
VII	6	$< button1, button1, button1 >$	$Edit\_S$	h=1,min=0,s=0
VIII	7	$< button1, button1, button1, button2 >$	$Edit\_S$	h=1,min=0,s=0
IX	8	$< button1, button1, button1, button1 >$	$Time$	h=1,min=0,s=0
X	9	$< timeout >$	$Time$	h=1,min=0,s=1
XI	10	$< setS(59), timeout >$	$Time$	h=1,min=1,s=0
XII	11	$< setS(59), setMin(59), timeout >$	$Time$	h=2,min=0,s=0
XIII	12	$< setS(59), set(59), set(11), timeout >$	$Time$	h=1,min=0,s=0

Tabelle C.4: Transitionüberdeckung *Clock* mit Instrumentierung

Erw. Zustände	Eingaben	<timeout,button1,timeout,button1,timeout>
Time h=1,min=0,s=0	IX,XIII	<time,-,-,-,->
Time h=1,min=0,s=1	X	<time,-,-,-,->
Time h=1,min=1,s=0	XI	<time,-,-,-,->
Time h=2,min=0,s=0	XII	<time,-,-,-,->
Edit_H h=1,min=0,s=0	I,III	<-,-,-,-,->
Edit_H h=2,min=0,s=0	II	<-,-,-,-,->
Edit_Min h=1,min=0,s=0	IV,VI	<-,-,-,-,time>
Edit_Min h=1,min=1,s=0	V	<-,-,-,-,time>
Edit_S h=1,min=0,s=0	VII,VIII	<-,-,time,-,->

Tabelle C.5: DS *Clock* ohne Instrumentierungunter *Completeness Assumption*