Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Softwaretechnik

— Studienarbeit —

Eine Einführung in Statecharts vor dem Hintergrund der objektorientierten Anwendungsentwicklung

Annette Laue Matthias Liedtke

27. November 1998

Betreuung:
Prof. Dr.-Ing. Heinz Züllighoven

Studienarbeit

Eine Einführung in Statecharts vor dem Hintergrund der objektorientierten Anwendungsentwicklung

Annette Laue
Krückauweg 18
22851 Norderstedt

Matrikel-Nr.: 4626296
E-Mail: 3laue@informatik.uni-hamburg.de

und

Matthias Liedtke
Bramfelder Chaussee 373 a
22175 Hamburg

Matrikel-Nr.: 4626169
E-Mail: 3liedtke@informatik.uni-hamburg.de

Betreuung:

Prof. Dr.-Ing. Heinz Züllighoven
Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Softwaretechnik

Inhaltsübersicht

Abbildungsverzeichnis

Tafelverzeichnis

1 Einleitung

1.1 Motivation

Objektorientierte Konzepte für den fachlichen und den technischen Entwurf sowie für die Konstruktion von Softwaresystemen sind inzwischen auch im betrieblichen Umfeld weit verbreitet. Sie konzentrieren sich auf die Gegenstände des jeweiligen Anwendungsbereichs, ermöglichen einen fast nahtlosen Übergang zwischen Analyse und Entwurf und können schließlich zu Implementationen führen, die sich durch einen hohen Grad an Lesbarkeit, Wartbarkeit, Erweiterbarkeit und Wiederverwendbarkeit auszeichnen.

Bei der objektorientierten Anwendungsentwicklung kommt neben Klassenbibliotheken vor allem Frameworks, die sich auf Entwurfsmuster stützen, eine große Bedeutung zu, da sie zusätzliche Abstraktionsebenen schaffen und somit erheblich zu den genannten Qualitätseigenschaften beitragen. Um von einem Framework profitieren zu können, ist es wesentlich, die in ihm festgelegten Entwurfsentscheidungen nachzuvollziehen und die Verantwortlichkeiten der einzelnen Klassen sowie ihre Interaktionen zu verstehen, ohne eine lange Einarbeitungszeit aufwenden zu müssen. Aus diesem Grund ist eine gute Framework-Dokumentation unerläßlich. Da ein Framework ein Modell einer Anwendungsdomäne darstellt und den Kontrollfluß weitgehend festlegt, ist hierbei eine Veranschaulichung der Dynamik besonders wichtig.

Leider wird der dynamische Aspekt bei der Framework-Dokumentation oftmals nicht ausreichend berücksichtigt. Zu den Darstellungsmitteln, die das Zusammenspiel zwischen Framework-Komponenten verdeutlichen können, zählen vor allem Interaktionsdiagramme, Nachrichtendiagramme, Diagramme, die Benutzt-Beziehungen aufzeigen, und Diagramme, die wiedergeben, in welcher Reihenfolge Objekte erzeugt werden. In diesem Bericht wird eine weitere Technik behandelt, mit deren Hilfe Dynamik in kompakter Form veranschaulicht werden kann: der auf David Harel zurückgehende grafische Formalismus der Statecharts, der das Konzept endlicher Automaten und ihrer Zustandsdiagramme erweitert.

Herkömmliche endliche Automaten sind in der Praxis nicht als Darstellungsmittel für komplexe Spezifikationen zu verwenden, da der Zustandsraum in bezug auf die Größe des zu beschreibenden Systems exponentiell anwächst, kaum strukturiert werden kann und keine Wiedergabe von Nebenläufigkeit erlaubt. Statecharts hingegen weisen diese Nachteile nicht auf; sie ermöglichen die Bildung von Zustandshierarchien, können verschiedene Teilaspekte eines Systems visuell voneinander getrennt charakterisieren und stellen einen Kommunikationsmechanismus für Ereignisse zur Verfügung. Deshalb sind Statecharts hervorragend geeignet, um die dynamischen Aspekte hochgradig reaktiver Systeme in knapper, ausdrucksmächtiger Form zu beschreiben, und gelten vielfach zurecht als die am besten geeignete Technik zur Darstellung komplexen Verhaltens.

Da interaktive Softwaresysteme spezielle reaktive Systeme sind, deren Funktionsmechanismen sich oftmals als besonders kompliziert erweisen, können gerade sie in hohem Maße von der Verwendung von Statecharts profitieren. Dies gilt vor allem für

objektorientierte Software, deren Grundlage stets interagierende Objekte sind.

Statecharts können somit einen wichtigen Beitrag dazu leisten, daß die Qualität der Dokumentation von Frameworks und von objektorientierter Software im allgemeinen verbessert werden kann. Ihre Integration in den Softwareentwicklungsprozeß kann dabei helfen, das intendierte Verhalten besser zu erfassen und in besonders geeigneter Form zum Diskussionsgegenstand zu machen.

1.2 Über den vorliegenden Bericht

Am Anfang der Studienarbeit stand der Wunsch nach einem Text, der sich unter Anwendungsgesichtspunkten mit den Möglichkeiten beschäftigt, Statecharts im Rahmen der objektorientierten Softwareentwicklung zu nutzen. Dieser sollte zum einen eine Arbeitsgrundlage sein, die es jedem Interessierten ermöglicht, sich zügig in das Gebiet der Statecharts einzuarbeiten, zum anderen jedoch auch recht präzise die Abbildung des Zustandskonzepts auf Klassen beschreiben. Vor diesem Hintergrund war eine umfassende, nahezu alle Aspekte des grafischen Formalismus behandelnde Darstellung wünschenswert.

Die Recherchen im Internet sowie in zahlreichen gedruckten Literaturverzeichnissen zeigten jedoch schnell, daß Statecharts etliche Fragestellungen aufwerfen. Aus diesem Grund erhielt die Studienarbeit einen ordnenden Charakter. Der vorliegende Bericht liefert einen Überblick und stellt eine Einführung in verschiedene Facetten des Statechart-Formalismus dar, so daß er als eine Basis für weiterführende Arbeiten mit weniger Grundlagen- und mehr Anwendungsaspekten angesehen werden kann. Sein Aufbau orientiert sich an der Chronologie der einzelnen Themen, die im Zusammenhang mit Statecharts in der Literatur behandelt worden sind.

So beschreibt Abschnitt 2 zunächst einmal die Natur reaktiver Systeme durch einen Vergleich mit transformierenden Systemen und stellt auf diese Weise heraus, vor welchem Hintergrund der Formalismus der Statecharts entwickelt wurde.

Abschnitt 3 ist eine Einführung, die einen Überblick über die wesentlichen Eigenschaften von Statecharts liefert. Er ist so informal wie möglich gehalten, da die Stärke von Statecharts in ihren kompakten und ausdrucksmächtigen Zustandsdiagrammen liegt.

Einige Möglichkeiten, wie die unabhängig voneinander vorgestellten Elemente des Formalismus zusammenwirken können, zeigt Abschnitt 4 anhand eines Anwendungsbeispiels auf. Im Vordergrund steht hierbei der Umgang mit der grafischen Syntax in einer konkreten Anwendungssituation und nicht deren Einbettung in eine bestimmte Methodik. Deshalb wurde als Beispiel kein objektorientiertes Softwaresystem, das ohnehin für eine vollständige Verhaltensbeschreibung zu umfangreich wäre, sondern ein Heimtrainer gewählt, der trotz seines geringen Funktionsumfangs bereits zu einer recht komplexen grafischen Darstellung führt.

Abschnitt 5 behandelt in knapper Form die formale Syntax und Semantik von Statecharts. Bei dieser Darstellung wird auf mathematische Ausführungen weitgehend verzichtet, da lediglich ein Überblick über einige wichtige Fragestellungen gegeben werden

soll, die bei der Verwendung von Statecharts auftreten können. Es wird verdeutlicht, daß der grafische Formalismus nicht eindeutig ist und somit verschiedene Semantiken denkbar sind. Abschnitt 5 kann als eine Anregung betrachtet werden, im konkreten Fall für die Konstrukte von Statecharts exakte Bedeutungen festzulegen.

Mit Abschnitt 6 beginnt die Betrachtung von Statecharts unter dem Gesichtspunkt der objektorientierten Anwendungsentwicklung. Er beschreibt, wie Statecharts im Rahmen der *Object Modeling Technique* (*OMT*) verwendet werden. Einer kurzen Einführung in OMT folgt der zentrale Teil, der das dynamische Modell behandelt, zu dem die Statecharts bei der Benutzung von OMT zusammengefaßt werden. Den Abschluß bildet eine Schilderung des Entwurfsprozesses nach OMT in Übersichtsform, die sich auf das dynamische Modell konzentriert.

Abschnitt 7 stellt die Notation der *Objectcharts* vor, eine weitere Möglichkeit, wie Statecharts für die Beschreibung objektorientierter Systeme eingesetzt werden können. Objectcharts sind präziser definiert als die im Rahmen von OMT verwendeten Statecharts und orientieren sich stärker an objektorientierten Konzepten. Es wird verdeutlicht, welche besonderen Eigenschaften Objectcharts gegenüber herkömmlichen Statecharts aufweisen, welche weitere Diagrammtechnik ihre Entwickler vorsehen, um die statischen Systemaspekte zu beschreiben, und wie aus beiden Darstellungen das letztendliche Systemverhalten abgeleitet werden kann.

Gegenstand von Abschnitt 8 ist der jüngste der erläuterten objektorientierten Ansätze, der auf den Entwickler des Statechart-Formalismus, David Harel, zurückgeht. Hier ist die Verknüpfung von Statecharts und Konzepten der Objektorientierung besonders ausgeprägt. Wesentliche Punkte sind die verwendete Beschreibungstechnik für statische Systemaspekte, die für die Interaktion zwischen Objekten vorgesehenen Mechanismen, die Elemente objektorientierter Statecharts sowie die Ausführung mit Hilfe von Statecharts spezifizierter Systeme.

Abschnitt 9 stellt den Abschluß des Berichts dar und verdeutlicht zunächst in kompakter Form die Gemeinsamkeiten und Unterschiede der drei vorgestellten Ansätze. Den Hauptteil bilden eine Bewertung dieser Punkte sowie Vorschläge für die Verwendung von Statecharts im Kontext des am Arbeitsbereich Softwaretechnik gelehrten WAM-Methodenrahmens. Eine kurze Darstellung weitergehender Fragestellungen, die sich im Laufe der Beschäftigung mit dem Statechart-Formalismus ergeben haben, rundet diesen Abschnitt ab.

2 Transformierende und reaktive Systeme

Im vorigen Jahrzehnt entwickelte David Harel, Professor für Angewandte Mathematik und Informatik am Weizmann Institute of Science in Israel, im Rahmen einer Beratertätigkeit für die Forschungs- und Entwicklungsabteilung der Israel Aircraft Industries (IAI) eine grafische Notation für die Modellierung des Verhaltens komplexer Systeme, den Formalismus der *Statecharts*. Harel hatte damals die Aufgabe, gemeinsam mit einer Gruppe von Ingenieuren die Spezifikation für ein Flugelektronik-System eines israelischen Kampfflugzeuges zu erarbeiten.

Obwohl es bereits zahlreiche Bücher und Artikel über Methoden für die Spezifikation und den Entwurf komplexer Systeme gab, erwies sich die gestellte Aufgabe als inhärent schwierig. Den Grund für diese Schwierigkeit sah Harel darin, daß es zwar für viele Arten von Systemen eine grundlegende Entwurfsphilosophie gab, sehr komplexe Systeme, die aus vielen nebenläufigen Hard- und Software-Komponenten bestehen, den bekannten Methoden jedoch kaum zugänglich waren. Gemeinsam mit Amir Pnueli, einem Experten für Systemspezifikation und -verifikation, der ebenfalls am Weizmann Institute of Science tätig ist, führt Harel diese Erkenntnis auf einen wesentlichen Unterschied zwischen zwei Arten von Systemen zurück.

In der Informatik gibt es, so Harel und Pnueli in [HP85], eine Reihe von Dichotomien[1], die aufgrund bestimmter Charakteristika eine Unterscheidung zwischen „recht einfach“ zu entwerfenden und „komplizierten“ Systemen, für die besondere Vorgehensweisen erforderlich sind, erlauben.

Bekannte derartige Dichotomien sind z.B. die Einteilung von Systemen in deterministische und nichtdeterministische, in terminierende und nicht terminierende, in synchrone und asynchrone sowie in sequentielle und nebenläufige Systeme. Alle diese Dichotomien haben nach Ansicht Harels und Pnuelis ihre Berechtigung, und die als „kompliziert“ klassifizierten Systeme stellen ihre Entwickler in der Tat vor erhebliche Probleme. Harel und Pnueli schlagen eine weitere Dichotomie vor, welche sie als diejenige ansehen, die am besten die „relativ einfach“ von den „kompliziert“ zu entwerfenden Systemen abgrenzt: die Unterscheidung zwischen *transformierenden* und *reaktiven* Systemen.

Unter einem transformierenden System verstehen Harel und Pnueli ein System, welches Eingaben akzeptiert, daraufhin Transformationen durchführt und schließlich Ausgaben produziert (siehe Abb. 1(a)). Diese Definition schließt auch solche Systeme ein, die während ihres Einsatzes nach weiteren Eingaben verlangen oder zusätzliche Ausgaben produzieren; entscheidend ist der Umstand, daß ein transformierendes System eine Ein-/Ausgabe-Operation durchführt. Ein reaktives System wird hingegen wiederholt durch seine Umgebung zu Aktivitäten veranlaßt und reagiert ständig auf

[1] Das Deutsche Wörterbuch von Brockhaus-Wahrig, erschienen im Rahmen der 18. Auflage des Grossen Brockhaus, beschreibt den Begriff der *Dichotomie* als einen u.a. in der Philosophie und der Sprachwissenschaft gebräuchlichen Ausdruck; in diesen Bereichen bedeutet er „Zweiteilung, Gliederung nach zwei Gesichtspunkten“.

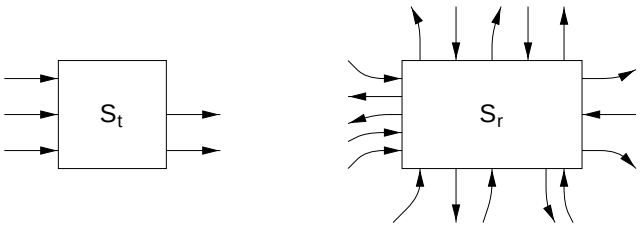


Abb. 1: Ein transformierendes System in (a) sowie ein reaktives System in (b)

externe Ereignisse, ist also ereignisgesteuert (siehe Abb. 1(b)). Es berechnet i. a. keine Funktion und führt auch keine solche aus, sondern erhält in gewisser Hinsicht eine bestimmte Beziehung zu seiner Umgebung aufrecht. Beispiele für reaktive Systeme lassen sich leicht und zahlreich finden; so sind neben Flugelektronik-Systemen u. a. auch Armbanduhren, Mikrowellengeräte, viele moderne medizinische Geräte, Telekommunikationsanlagen, Betriebssysteme sowie die Mensch-Maschine-Schnittstellen zahlreicher Softwareprodukte reaktive Systeme.

Harel und Pnueli machen deutlich, daß sich ihre Sicht des Begriffs „System“ nicht auf softwarebasierte, hardwarebasierte oder eingebettete Systeme beschränkt. Die von ihnen verwendete Terminologie ist sehr allgemein, und die Form, in der ein System schließlich implementiert wird, ist zunächst nicht von Bedeutung.[2]

Es stellt sich die Frage, warum Harel und Pnueli die Einteilung in transformierende und reaktive Systeme als die grundlegendste aller Dichotomien ansehen. In [HP85] werden hierfür im wesentlichen zwei Argumente genannt. Zum einen zieht sich diese Unterscheidung quer durch alle oben genannten anderen Dichotomien: Sowohl transformierende als auch reaktive Systeme können deterministisch oder nichtdeterministisch, terminierend oder nicht terminierend, synchron oder asynchron sowie sequentiell oder nebenläufig sein. Zum anderen liegt in der Natur der reaktiven Systeme eine besondere Problematik, die durch die Notwendigkeit einer Beschreibung des Systemverhaltens zutage tritt. Dieser zweite Punkt soll im folgenden näher betrachtet werden.

Nach Ansicht Harels und Pnuelis können der Entwurf und die Konstruktion eines Systems nicht ohne ein klares Verständnis des intendierten Systemverhaltens durchgeführt werden. Dieses Verständnis ist für jedes Entwicklungsstadium eines Systems wichtig. Ein Entwicklungsstadium ist, so Harel und Pnueli, durch einen bestimmten Stand der Implementation und einen bestimmten Detaillierungsgrad der zum implementierten System gehörigen Verhaltensbeschreibung gekennzeichnet. Durch Verfeinerungen der Implementation und der Verhaltensbeschreibung gelangt man zu weiteren

[2] Wir beschäftigen uns vor dem Hintergrund reaktiver objektorientierter Anwendungssysteme mit den Ideen Harels und Pnuelis.

Stadien der Entwicklung.[3] Für jedes Stadium muß eine Schnittstellenbeschreibung erstellt werden, die die Interaktion des Systems mit seiner Umgebung über Ein- und Ausgabekanäle darstellt. Diese Beschreibung kann für unsere Zwecke als eine Liste von Ereignissen bestimmter Granularität betrachtet werden. Die zugehörige Verhaltensbeschreibung spiegelt dann das Verhalten des Systems unter Verwendung der in dieser Liste aufgeführten Ereignisse wider.

Die Probleme, vor die reaktive Systeme ihre Entwickler stellen, treten im Verlauf des Entwurfs- und Konstruktionsprozesses immer deutlicher zutage: Es muß nicht nur die Konsistenz zweier Verhaltensbeschreibungen eines reaktiven Systems mit unterschiedlichen Detaillierungsgraden überprüft, sondern es muß vor allem eine Möglichkeit gefunden werden, das eventuell äußerst komplexe Verhalten eines reaktiven Systems in kleinere Einheiten zu gliedern. Während zum Erscheinungszeitpunkt von [HP85] für transformierende Systeme geeignete Methoden zur Dekomposition aus dem Bereich der Strukturierten Analyse weit verbreitet waren, gab es für reaktive Systeme nahezu keine entsprechenden Vorgehensweisen. Dennoch war der Bedarf an derartigen Methoden unverkennbar, denn jedes komplexe reaktive System besteht aus reaktiven Subsystemen, deren Verhaltenscharakteristika das Verhalten anderer Subsysteme und des gesamten Systems beeinflussen.

Man kann sich an diesem Punkt die Frage stellen, ob es damals sinnvoll gewesen wäre, reaktive als transformierende Systeme aufzufassen, indem man die Sequenz aller das System von außen beeinflussenden Ereignisse als Eingabe betrachtet, die vom System in die zugehörige Sequenz der ausgesandten Ereignisse überführt wird. Das Argument, das gegen diese Idee spricht, besteht darin, daß die auf ein reaktives System Einfluß nehmenden Ereignisse von zu früheren Zeitpunkten erfolgten Reaktionen des Systems abhängen können; somit ist die Beschreibung einer Relation zwischen Ein- und Ausgabesequenzen von Ereignissen nicht ausreichend (siehe [HdR91]).[4]

Die bisherige Diskussion macht deutlich, daß eine Methode wünschenswert ist, die es ermöglicht, das Verhalten eines reaktiven Systems in kleinere Einheiten zu gliedern. Harel und Pnueli formulieren in [HP85] folgende Anforderungen, die eine derartige Methode erfüllen sollte:

[3] Die in [HP85] dargestellte Sicht ist vor dem Hintergrund des Paradigmas der Strukturierten Analyse (SA) zu betrachten, dessen wesentliches Konzept die Gliederung allgemeiner Aufgaben in speziellere Aufgaben geringerer Komplexität im Sinne eines Top-Down-Entwurfs ist. Dennoch ist die Darstellung in [HP85] unserer Ansicht nach auch im Hinblick auf objektorientierte Vorgehensweisen interessant, da auch bei diesen oftmals Verhaltensbeschreibungen sowie Systeme unterschiedlicher Granularität Gegenstände der Betrachtung sind.

[4] Der Unterschied zwischen reaktiven und transformierenden Systemen ist in etwas anderer Form ein aktueller Diskussionsgegenstand der Informatik: Peter Wegner hat das Konzept der Turing-Maschine um die Möglichkeit der Interaktion mit einer externen Umgebung erweitert (siehe hierzu z. B. [Weg97]). Die auf diese Weise entstehenden *Interaktionsmaschinen* lassen sich als reaktive Systeme auffassen, während Turing-Maschinen Algorithmen repräsentieren und als transformierende Systeme betrachtet werden können. Wegner zeigt, daß Interaktionsmaschinen mächtiger als Turing-Maschinen sind. Hieraus läßt sich eine wichtige Schlußfolgerung ableiten: Interaktive Systeme können nicht vollständig durch Algorithmen modelliert werden. Anders formuliert: Man kann reaktive Systeme nicht auf transformierende Systeme abbilden.

- Sie sollte Beschreibungen zur Verfügung stellen, die wohlstrukturiert, prägnant, unzweideutig, lesbar und leicht zu verstehen sind.
- Sie sollte ausschließlich beschreibend sein und keine Abhängigkeiten von Implementationsaspekten aufweisen oder diese zumindest minimieren.

Der von Harel und Pnueli vorgeschlagenen Methode liegt der grafische Formalismus der *Statecharts* zugrunde, welcher diesen Forderungen genügt und im folgenden vorgestellt werden soll.

3 Der grafische Formalismus der Statecharts

Statecharts stellen eine Erweiterung herkömmlicher endlicher Automaten sowie ihrer Zustandsdiagramme dar und ermöglichen die Beschreibung des Verhaltens komplexer reaktiver Systeme in kompakter, ausdrucksmächtiger Form. Endliche Automaten werden in nahezu allen Teilgebieten der Informatik eingesetzt und sind daher von besonderem Interesse. Sie finden nicht nur bei der Anwendungsentwicklung, sondern z. B. auch in den Bereichen Betriebssysteme (siehe z. B. [Tan92]), Kommunikationsprotokolle (siehe z. B. [Ker93]) und Rechnerarchitekturen (siehe z. B. [Gil93]) Verwendung.

Schon seit langem werden endliche Automaten für Verhaltensbeschreibungen eingesetzt, um den Entwurf von Systemen zu erleichtern und die getroffenen Entscheidungen zu dokumentieren (siehe z. B. [Par69]). Trotz ihrer breiten Akzeptanz weisen endliche Automaten jedoch erhebliche Nachteile auf. Martin und McClure schreiben den Zustandsdiagrammen endlicher Automaten in [MM85] zwar z. B. zu, eher problem- als programmbezogen zu sein, nennen jedoch auch eine Reihe von negativen Eigenschaften. So sind die Zustandsdiagramme endlicher Automaten u. a. oftmals

- nicht leicht zu lesen,
- schwer zu zeichnen und zu ändern,
- nur schwer zugänglich,
- nicht für schrittweise Verfeinerungen geeignet sowie
- als Darstellungsmittel für komplexe Spezifikationen, die einen hohen Grad an Fehlerfreiheit erfordern, nicht zu verwenden.

Harel, Pnueli sowie zwei Koautoren vertreten in [HPSS87] die Auffassung, daß viele, die sich mit dem Entwurf komplexer Anwendungen beschäftigen, es nahezu aufgegeben hätten, herkömmliche endliche Automaten sowie ihre Zustandsdiagramme einzusetzen, und nennen hierfür vier konkrete Gründe:

1. Zustandsdiagramme sind „flach“. Sie sehen keine Konzepte für Tiefe, Hierarchie oder Modularität vor und unterstützen daher keine schrittweisen Top-Down- oder Bottom-Up-Entwicklungen.
2. Ein Ereignis, das einen identischen Übergang von einer großen Anzahl von Zuständen bewirkt, wie z. B. ein Interrupt hoher Abstraktion, muß jedem dieser Zustände separat zugeordnet werden, so daß sich im zugehörigen Zustandsdiagramm unnötig viele Pfeile ergeben.
3. Zustandsdiagramme sind im Hinblick auf die Anzahl der Zustände unökonomisch und daher häufig nahezu unmöglich zu erstellen. Wächst das zu beschreibende System in seiner Größe linear, so wächst die Anzahl der Zustände exponentiell,

da der Formalismus herkömmlicher endlicher Automaten die Entwickler dazu zwingt, alle Systemzustände explizit darzustellen.

4. Zustandsdiagramme sind ihrer Natur nach sequentiell und bieten keine Möglichkeit zur Darstellung von Nebenläufigkeit.

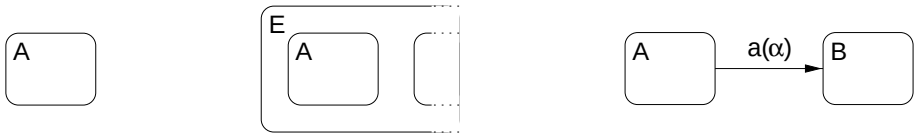
Der von Harel entwickelte Statechart-Formalismus überwindet diese Unzulänglichkeiten, wobei er das visuelle Erscheinungsbild der Zustandsdiagramme konventioneller endlicher Automaten in vielfacher Hinsicht verbessert.

Die folgenden Unterabschnitte bilden eine in sich geschlossene Darstellung der wesentlichen Charakteristika von Statecharts sowie einiger denkbarer Erweiterungen. Dabei orientiert sich der Text weitgehend an den Ausführungen in [Har87], die im Bereich der Statecharts als grundlegend gelten. Es erfolgt im vorliegenden Abschnitt noch keine Fokussierung auf den Einsatz von Statecharts in objektorientierten Kontexten, da die vorgestellten Merkmale von einer solchen Verwendung unabhängig sind.

3.1 Die Bildung von Zustandshierarchien

Eine wichtige Eigenschaft von Statecharts ist die Möglichkeit, Zustandshierarchien bilden zu können. Zustandshierarchien erlauben verschiedene Grade der Detaillierung und lassen in bestimmter Hinsicht zusammengehörige Zustände leicht als solche erkennen. Mit ihrer Hilfe kann somit eine Strukturierung des Zustandsraumes erreicht werden, die für ein klares Verständnis komplexer Verhaltensbeschreibungen unerläßlich ist.

Auf jeder Hierarchieebene werden Zustände durch abgerundete Rechtecke dargestellt, welche mit einem Bezeichner versehen sind (siehe Abb. 2(a)). Die Hierarchie-Relation auf der Menge der Zustände wird durch grafischen Einschluß ausgedrückt. Somit besagt Abb. 2(b), daß Zustand A ein Subzustand des Zustands E ist.



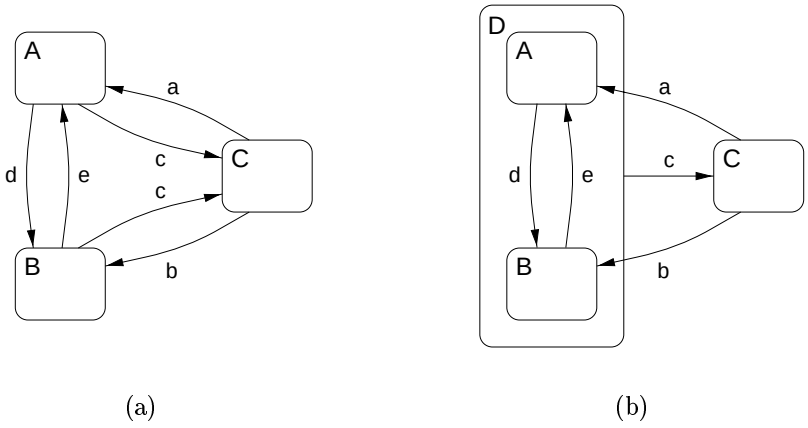
(a) (b) (c)

Abb. 2: Ein Zustand in (a), die Darstellung eines Teils einer Hierarchie-Relation in (b) und ein Zustandsübergang in (c)

Mögliche Zustandsübergänge werden durch beschriftete Pfeile dargestellt, die Zustände beliebiger Hierarchieebenen miteinander verbinden können. Die Beschriftung eines Pfeils besteht aus einem Bezeichner für das Ereignis, das den tatsächlichen Übergang bewirkt. Zustandsübergänge können von einer Bedingung abhängen; in einem solchen Fall wird der entsprechende Pfeil zusätzlich mit einem Bezeichner für diese

Bedingung versehen, welcher in runden Klammern hinter dem Ereignisbezeichner aufgeführt wird. Abb. 2(c) zeigt ein Beispiel für einen Zustandsübergang. In der einführenden Darstellung folgt die Beschriftung der Zustände sowie der die Zustandsübergänge repräsentierenden Pfeile soweit wie möglich einem einheitlichen Schema; fast immer bezeichnen wir Zustände mit lateinischen Großbuchstaben, Ereignisse mit lateinischen Kleinbuchstaben und Bedingungen mit griechischen Kleinbuchstaben.

Betrachtet man Abb. 3(a), so erkennt man, daß ein Zustandsübergang von A nach C oder von B nach C stattfinden kann, wenn das Ereignis c eintritt. Die Zustände A und B weisen also eine gemeinsame Eigenschaft auf, und es ist möglich, sie im Sinne einer Bottom-Up-Entwicklung in einem neuen Zustand D zusammenzufassen, wie es in Abb. 3(b) gezeigt wird.



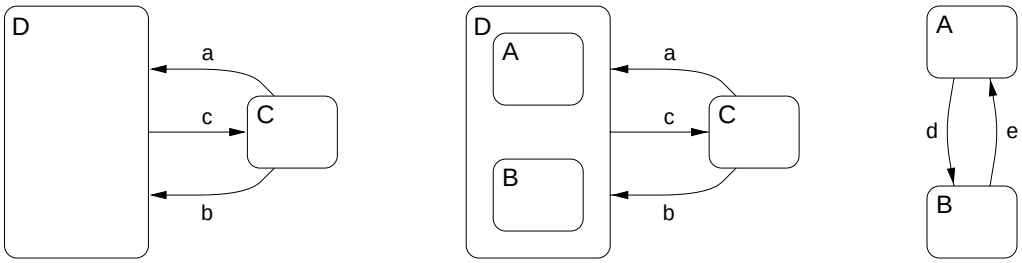
(a) (b)

Abb. 3: Ein einfacher Statechart in (a)
sowie eine Abstraktion in (b)

Der Zustand D hat dann die Bedeutung eines *Exklusiv-Oder* (XOR): Befindet sich das System im Zustand D, so befindet es sich in einem der Zustände A und B, aber nicht in beiden. Somit läßt sich D als *Abstraktion* von A und B auffassen. Man beachte, daß durch die Einführung des Zustands D nicht nur eine Zustandshierarchie gebildet, sondern darüber hinaus die Anzahl der Pfeile im Diagramm reduziert wird. Eine in großem Umfang erfolgende Anwendung des obengenannten Prinzips hat daher zur Folge, daß die Nachteile endlicher Automaten, die in der Aufzählung auf Seite 9 unter den ersten beiden Punkten genannt sind, überwunden werden.

Geht man von Abb. 4(a) aus, so kann man sich D als aus A und B bestehend vorstellen und im Sinne eines Top-Down-Entwurfs zu Abb. 4(b) gelangen; in diesem Fall kann man von einer *Verfeinerung* sprechen. Hierbei ist zu bedenken, daß die Pfeile mit den Beschriftungen a und b nicht ausreichend spezifiziert sind. Verlängert man diese Pfeile bis zu den Zuständen A und B und verfeinert man D mit Hilfe von Abb. 4(c) weiter, so entsteht auf diese Weise der Statechart aus Abb. 3(b), welcher zuvor das Ergebnis eines Abstraktionsvorganges war. Der Folgezustand, der sich in Abb. 4(b) ergibt, wenn Zustand C verlassen wird, kann auch durch einen der in Abschnitt 3.3

beschriebenen Mechanismen (Default-Zustand, bedingter Eintritt, selektiver Eintritt, History-Funktion) erfolgen.



(a) (b) (c)

Abb. 4: Ein einfacher Statechart in (a) sowie eine denkbare Verfeinerung des Zustands D in (b); (c) zeigt eine weitere Verfeinerung von D

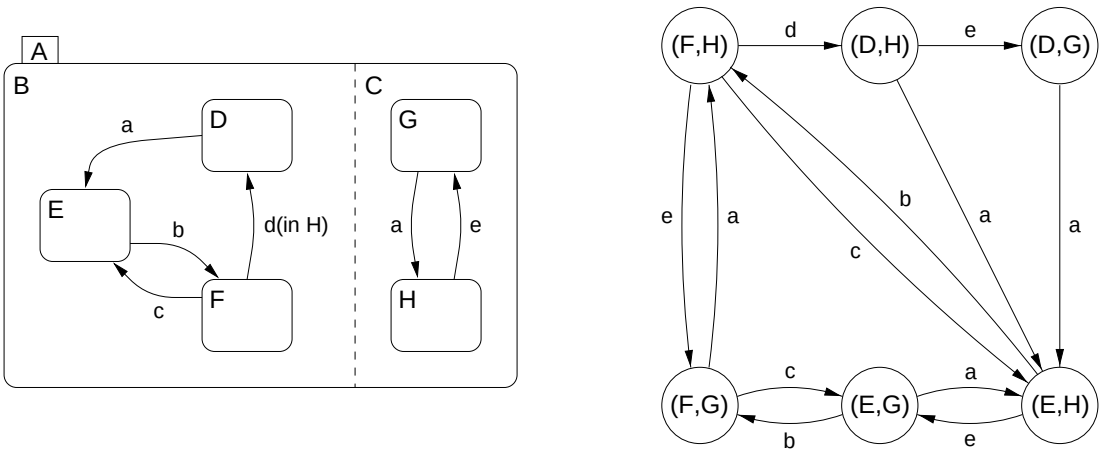
3.2 Orthogonalität

Neben der im Abschnitt 3.1 vorgestellten XOR-Dekomposition von Zuständen stellt der Formalismus der Statecharts auch die Möglichkeit einer AND-Dekomposition zur Verfügung. Durch sie wird festgelegt, daß alle Subzustände eines Zustandes *gleichzeitig* eingenommen werden, wenn dieser betreten wird. Die Subzustände eines mittels der AND-Dekomposition verfeinerten Zustandes nennt man gemäß den Ausführungen in [HP85] *orthogonal*; dementsprechend bezeichnet man die der AND-Dekomposition zugrundeliegende Eigenschaft des Statechart-Formalismus als *Orthogonalität*. Im Falle einer XOR-Dekomposition spricht man von (*einander*) *ausschließenden* Subzuständen. Orthogonalität kann – wie auch die XOR-Dekomposition – auf jeder Hierarchieebene eingesetzt werden, um Zustände genauer zu beschreiben.

Die grafische Repräsentation von Orthogonalität erfolgt mit Hilfe gestrichelter Linien, die die einzelnen AND-Komponenten voneinander trennen. Abb. 5(a) zeigt das Beispiel eines Zustandes A, der aus zwei orthogonalen Komponenten B und C besteht. Geht das System in den Zustand A über, so bedeutet dies, daß die Zustände B und C gleichzeitig betreten werden. Da B und C mittels der XOR-Dekomposition verfeinerte Zustände sind, ist dies gleichbedeutend damit, daß genau eines der Zustandspaare (D, G), (D, H), (E, G), (E, H), (F, G) und (F, H) eingenommen wird.

Im vorliegenden Fall ist das bei einem Zustandsübergang nach A einzunehmende Zustandspaar nicht festgelegt, und somit ist die durch den Statechart in Abb. 5(a) gegebene Spezifikation eines denkbaren Verhaltens nicht ausreichend. Die Möglichkeiten, das initiale Zustandspaar zu bestimmen, sollen an dieser Stelle jedoch nicht näher betrachtet werden, da sie Teil der allgemeinen Darstellung des Themenkomplexes „Einen Zustand einnehmen und verlassen“ im Abschnitt 3.3 sind.

Ist im Beispiel von Abb. 5(a) (D, G) der aktuelle Zustand des Systems, so überführt



(a) (b)

Abb. 5: Ein Beispiel für Orthogonalität in (a); (b) zeigt ein zum Statechart in (a) äquivalentes Zustandsdiagramm eines herkömmlichen endlichen Automaten

das Ereignis a das System in den Zustand (E, H), wobei die beiden Zustandswechsel zu E und H gleichzeitig erfolgen. Auf diese Weise ermöglicht Orthogonalität eine bestimmte Form der *Synchronisation*. Tritt hingegen im Systemzustand (D, H) das Ereignis e auf, so erfolgt in C ein Zustandswechsel von H nach G, während die Komponente B keiner Veränderung unterliegt; es besteht aufgrund der Orthogonalität eine gewisse *Unabhängigkeit*, denn der Zustandswechsel innerhalb von C wird nicht durch den aktuellen Zustand in B beeinflußt.

Somit verdeutlicht das Beispiel in Abb. 5(a), daß mit Hilfe orthogonaler Zustände die den herkömmlichen Zustandsdiagrammen innewohnende Beschränkung auf Sequentialität (siehe Punkt 4 in der Aufzählung der Nachteile endlicher Automaten auf S. 10) aufgehoben und Nebenläufigkeit dargestellt werden kann. Beachtenswert ist in diesem Zusammenhang die in Abb. 5(a) neben dem Ereignis d aufgeführte Bedingung „in H“, welche aufzeigt, daß orthogonale Zustände aufeinander Bezug nehmen können.[1]

Abb. 5(b) zeigt ein zum Statechart in Abb. 5(a) äquivalentes Zustandsdiagramm eines herkömmlichen endlichen Automaten und deutet einen besonders bedeutsamen Aspekt an: Orthogonalität kann zu einer eventuell drastischen Verkleinerung der Zustandsmenge führen und somit die unter Punkt 3 auf Seite 9 genannten Nachteile der Zustandsdiagramme endlicher Automaten überwinden. Man denke hier z. B. an zwei orthogonale Komponenten mit jeweils eintausend Subzuständen, welche in einem konventionellen Zustandsdiagramm mit bis zu einer Million Zuständen resultieren würden.

[1] Das Beispiel in Abb. 5(a) verdeutlicht, daß zwischen orthogonalen Komponenten Abhängigkeiten bestehen können. Da nach unserem Verständnis Nebenläufigkeit mit Unabhängigkeit einhergeht, halten wir es für unangebracht, die Begriffe Orthogonalität und Nebenläufigkeit im gleichen Sinn zu verwenden. Wir schlagen vielmehr vor, Orthogonalität als eine Eigenschaft zu betrachten, welche Nebenläufigkeit ermöglicht, zumal Harel die beiden Begriffe in [Har87] nicht explizit gleichsetzt.

3.3 Einen Zustand einnehmen und verlassen

In den vorangegangenen Abschnitten wurden mit der XOR- und der AND-Dekomposition zwei wesentliche Eigenschaften von Statecharts vorgestellt. Weitgehend offen geblieben ist bisher die Frage, nach welchen Regeln die einzelnen Zustände eines Statecharts, in dem diese beiden Prinzipien Anwendung finden, eingenommen und verlassen werden sollen.

So ist z. B. im Statechart in Abb. 6(a) nicht festgelegt, welcher der beiden Zustände B und C bei einem durch das Ereignis a ausgelösten Übergang von A nach D eingenommen werden soll. Eine naheliegende Lösung dieses Problems besteht darin, den entsprechenden Pfeil zu verlängern; auf diese Weise wird in Abb. 6(b) der Zustand B als nächster Zustand gekennzeichnet.

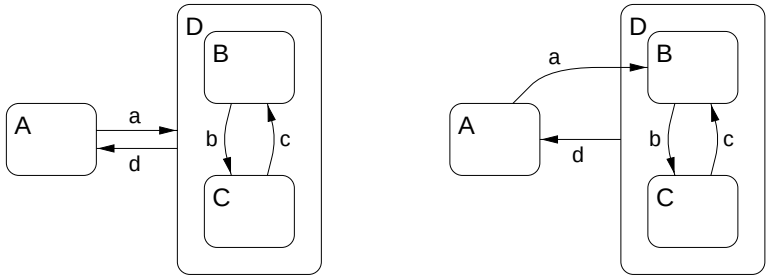


Abb. 6: Ein unterspezifizierter Übergang in einen verfeinerten Zustand in (a) sowie eine denkbare Lösung in (b)

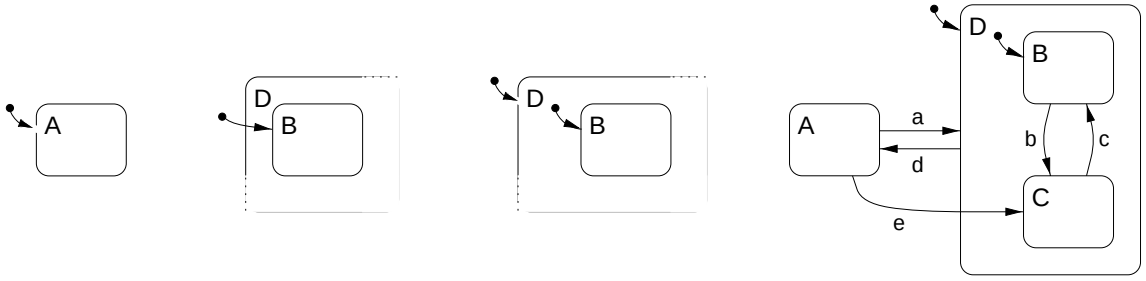
Harel beschreibt in [Har87] weitere Möglichkeiten, den Folgezustand beim Übergang in einen mittels der XOR-Dekomposition verfeinerten Zustand festzulegen:

- Es kann ein *Default-Zustand* unter den einander ausschließenden Subzuständen bestimmt werden, der eingenommen wird, sofern kein anderer Zustand durch einen Pfeil explizit als Folgezustand gekennzeichnet wird.
- Mit Hilfe einer *History-Funktion* kann derjenige Zustand zum Folgezustand erklärt werden, in dem sich das System befand, als der unmittelbar übergeordnete Zustand zuletzt verlassen wurde.
- *Selektive* und *bedingte Eintritte* können verwendet werden, wenn es besondere Beziehungen zwischen Zuständen und Ereignissen oder zwischen Zuständen und Bedingungen gibt (s. u.).

Erklärt man einen Zustand A aus einer Menge von Subzuständen, die alle der gleichen Hierarchieebene zuzuordnen sind, zum Default-Zustand, so bedeutet dies, daß A eingenommen wird, wenn der übergeordnete Zustand betreten und nicht explizit

ein anderer Folgezustand festgelegt wird. Die grafische Kennzeichnung eines Zustandes als Default-Zustand erfolgt mit Hilfe eines kleinen unbeschrifteten Pfeiles; Abb. 7(a) zeigt ein Beispiel. Eine besondere Situation ergibt sich auf der obersten Hierarchieebene: Dort stellt ein Default-Zustand den initialen Zustand eines Systems dar; die Verwandtschaft der Default-Zustände mit den Startzuständen herkömmlicher endlicher Automaten wird hier besonders deutlich.

Da jeder Zustand unabhängig von der Hierarchieebene, zu der er gehört, ein Startzustand des durch den Statechart beschriebenen Systems sein kann, kann in dem Beispiel aus Abb. 6(a) auch der Zustand B der initiale Zustand sein. Für diesen Fall sowie allgemein für den Übergang in einen mittels der XOR-Dekomposition verfeinerten Zustand sind zwei Notationen denkbar, die in Abb. 7(b) und Abb. 7(c) dargestellt sind. Wir schlagen vor, stets die in Abb. 7(c) gezeigte Darstellung zu verwenden, da diese es einfacher macht, zusätzliche Verfeinerungen ein- und auszublenden.



(a) (b) (c) (d)

Abb. 7: Default-Zustand in (a); Alternativen für die Kennzeichnung eines Default-Zustandes einer niedrigeren Hierarchieebene in (b) und (c); Ergänzung des Beispiels aus Abb. 6 in (d)

Abb. 7(d) zeigt eine Ergänzung des Beispiels aus Abb. 6(a). Zusätzlich wurde eine mit dem Ereignis e beschriftete Kante eingeführt, die verdeutlicht, wie die mittels eines Default-Zustandes getroffene Entscheidung für einen Folgezustand aufgehoben werden kann.

Eine weitere Möglichkeit, aus einer Gruppe von Zuständen den Folgezustand zu bestimmen, stellt die History-Funktion dar. Der Einsatz der History-Funktion führt dazu, daß derjenige Zustand eingenommen wird, welcher innerhalb der Zustandsgruppe der zuletzt besuchte ist.

Diese Form des Zustandsübergangs wird im Statechart durch das Symbol „Ⓗ“ dargestellt. Abb. 8(a) zeigt ein Beispiel: Befindet sich das zugehörige System im Zustand A, so überführt das Ereignis a das System in den Zustand B, während im Falle des Ereignisses b der Folgezustand anhand der „System-Geschichte“ bestimmt wird. Abb. 8(b) stellt eine alternative Darstellung vor.

Zu beachten ist, daß durch das Symbol „Ⓗ“ nur eine Anwendung der History-Funktion auf derjenigen Zustandsebene ausgedrückt wird, auf der es auftritt. Es ist

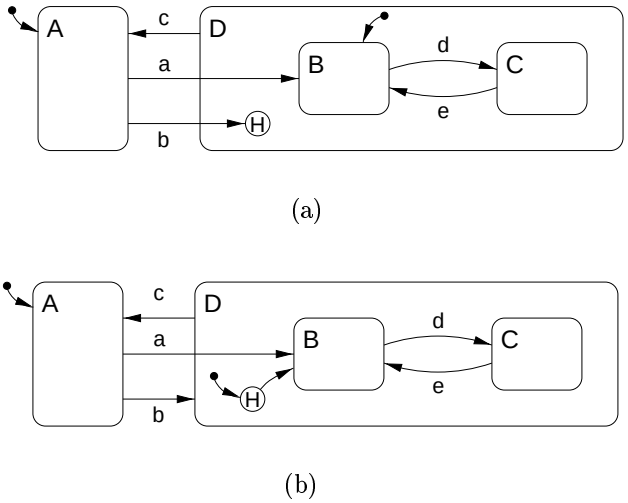


Abb. 8: Zwei Darstellungsformen für die History-Funktion

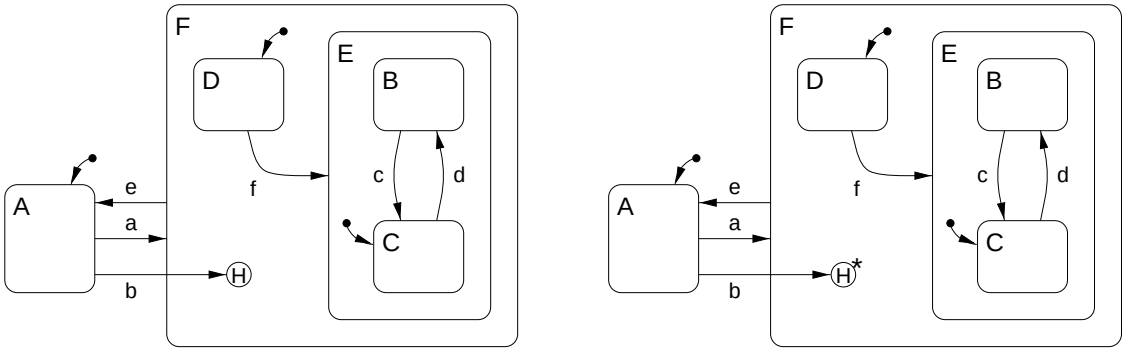
jedoch ebenfalls möglich, die History-Funktion auf alle Verfeinerungsebenen eines Zustands anzuwenden; zu diesem Zweck wird die Notation „Ⓗ*“ verwendet, die an die Darstellung der reflexiven und transitiven Hülle von Relationen erinnert.

Abb. 9 verdeutlicht den Unterschied zwischen diesen beiden Arten der History-Funktion. Für den mit dem Ereignis a beschrifteten Übergang gilt in beiden dargestellten Fällen, daß der Zustand D eingenommen wird – unabhängig von dem zuletzt innerhalb von F besuchten Zustand. Tritt jedoch das Ereignis b ein, so können beim Übergang von A nach F unterschiedliche Folgezustände resultieren: Nimmt man an, daß B der zuletzt eingenommene Zustand war, bevor F verlassen wurde, so erfolgt im Statechart aus Abb. 9(a) ein Übergang nach C, da allein der Umstand, daß das System in E verweilte, als Information genutzt wird. Demgegenüber wird im Statechart aus Abb. 9(b) erneut der Zustand B eingenommen, weil in diesem Fall die entsprechende Information für alle Hierarchieebenen gespeichert wurde.

Das Beispiel aus Abb. 9 läßt auch das Zusammenwirken von Default-Zuständen und Anwendungen der History-Funktion erkennen: Die History-Funktion kommt nur dann wirklich zum Einsatz, wenn der Zustand F bereits betreten worden ist; beim ersten Übergang nach F wird der Default-Zustand D eingenommen.

Die beiden bisher behandelten Varianten der History-Funktion stellen zwei Extreme dar: Entweder wird nur die „System-Geschichte“ der jeweils obersten Hierarchieebene oder auch diejenige aller feineren Ebenen betrachtet. Verwendet man die History-Funktion auf verschiedenen Ebenen, so kann man beliebige Ergebnisse zwischen diesen beiden Polen erzielen. Als eine ausdrucksmächtige Ergänzung der History-Funktion schlägt Harel den Einsatz temporaler Logik vor; in [Har87] findet man eine kurze Betrachtung dieser Idee.

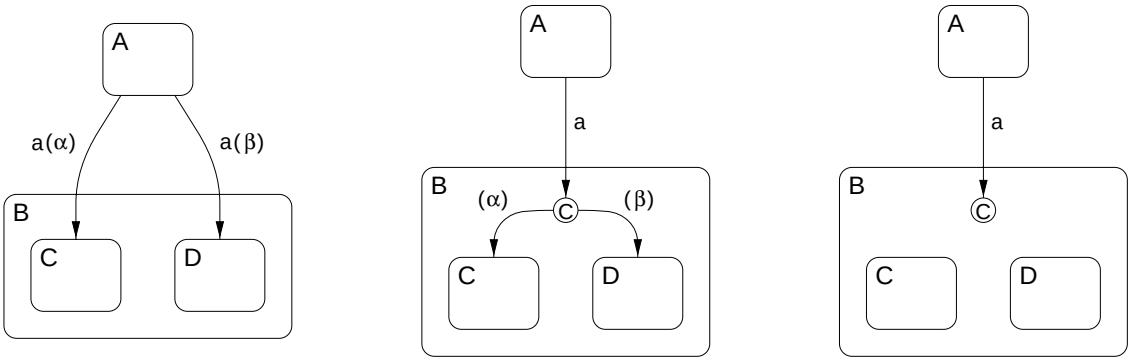
Damit die bis zu einem bestimmten Zeitpunkt vorgenommenen History-Einträge



(a) (b)

Abb. 9: Vergleich der beiden Varianten der History-Funktion

gelöscht werden können, sieht Harel die beiden Aktionen[2] *clear-history*(*state*) und *clear-history*(*state**) vor, von denen die erste nur die für die Ebene des Zustandes *state* gültige History-Information, die zweite hingegen auch alle entsprechenden Vermerke der unterhalb von *state* liegenden Hierarchieebenen löscht. Eine Anwendung dieser beiden Aktionen findet sich in dem Beispiel, das Gegenstand von Abschnitt 4 ist.



(a) (b) (c)

Abb. 10: Ein Statechart in (a); Möglichkeiten der vereinfachten Darstellung mit Hilfe des bedingten Eintritts in (b) und (c)

Zwei weitere Möglichkeiten, einen Zustand einzunehmen, stellen der *bedingte* und der *selektive Eintritt* zur Verfügung, die es erlauben, komplizierte Zustandsübergänge in einer einfachen grafischen Form darzustellen. Der bedingte Eintritt ist für Fälle vorgesehen, die der in Abb. 10(a) dargestellten Situation ähneln: Ein Ereignis a löst

[2] Zur Erläuterung des Begriffs „Aktion“ siehe Abschnitt 3.4

einen Übergang von einem Zustand A in einen Zustand B aus, wobei der einzunehmende Subzustand von B jedoch davon abhängt, ob bestimmte Bedingungen erfüllt sind. Mit Hilfe des Zeichens „©“, das einen bedingten Eintritt kennzeichnet (der Buchstabe „C“ soll an das Wort „conditional“ erinnern), kann man das Diagramm vereinfachen und zu Abb. 10(b) gelangen.

Ist man zunächst an einer groben Darstellung interessiert, so kann man sich auf eine Vereinfachung gemäß Abb. 10(c) beschränken; die erforderlichen Einzelheiten sollten dann separat festgehalten werden, damit sie bei Bedarf zur Verfügung stehen.

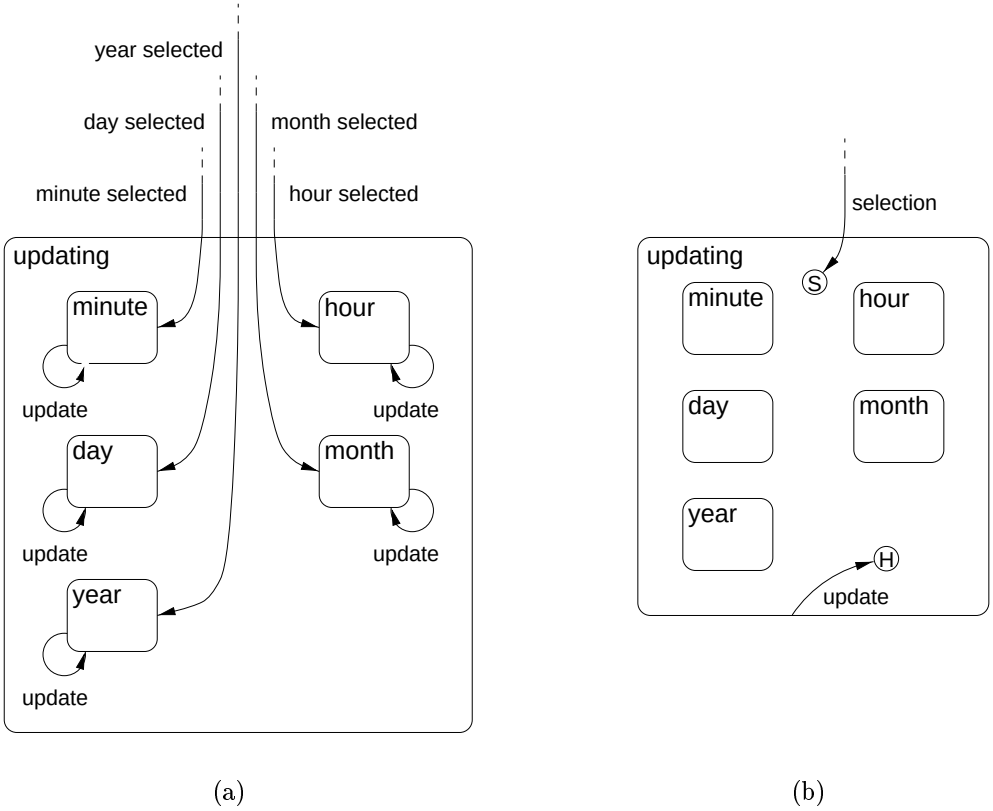


Abb. 11: Ein Statechart in (a) und eine mögliche Vereinfachung durch einen selektiven Eintritt in (b)

Der selektive Eintritt stellt eine weitere Art dar, einen verfeinerten Zustand einzunehmen. Er kann verwendet werden, wenn es eine 1:1-Beziehung zwischen der Menge der denkbaren Ereignisse, die einen Übergang auslösen können, und der Menge der Subzustände, die mögliche Folgezustände sind, gibt. In einem solchen Fall kann man jedes der Ereignisse als Auswahl aus einer Menge von Optionen ansehen, die durch die einzelnen Zustände repräsentiert werden, wobei die Zugehörigkeit eines Ereignisses zu einem bestimmten Zustand anhand der gewählten Bezeichnungen erkennbar ist.

Abb. 11(a) zeigt ein an [Har87] orientiertes Beispiel, das einige Einstellungsmöglichkeiten einer Digitaluhr wiedergibt. Der Benutzer der Uhr kann sich zunächst für eine

einzustellende Komponente entscheiden, die er durch einen Knopfdruck auswählt. Die gewählte Komponente kann er daraufhin mit Hilfe eines anderen Knopfes aktualisieren. Abb. 11(b) zeigt, wie die Situation mittels des selektiven Eintritts, dargestellt durch das Symbol „Ⓢ“, modelliert werden kann. Die History-Funktion wird hier in besonders eleganter Form benutzt, um die Darstellung weiter zu vereinfachen.

Alle bisher in diesem Abschnitt beschriebenen Teile des grafischen Formalismus der Statecharts stellen Möglichkeiten dar, denjenigen Zustand zu charakterisieren, der anfangs oder bei einem Zustandsübergang eingenommen werden soll. Im folgenden wird auch betrachtet, wie Zustände verlassen werden.

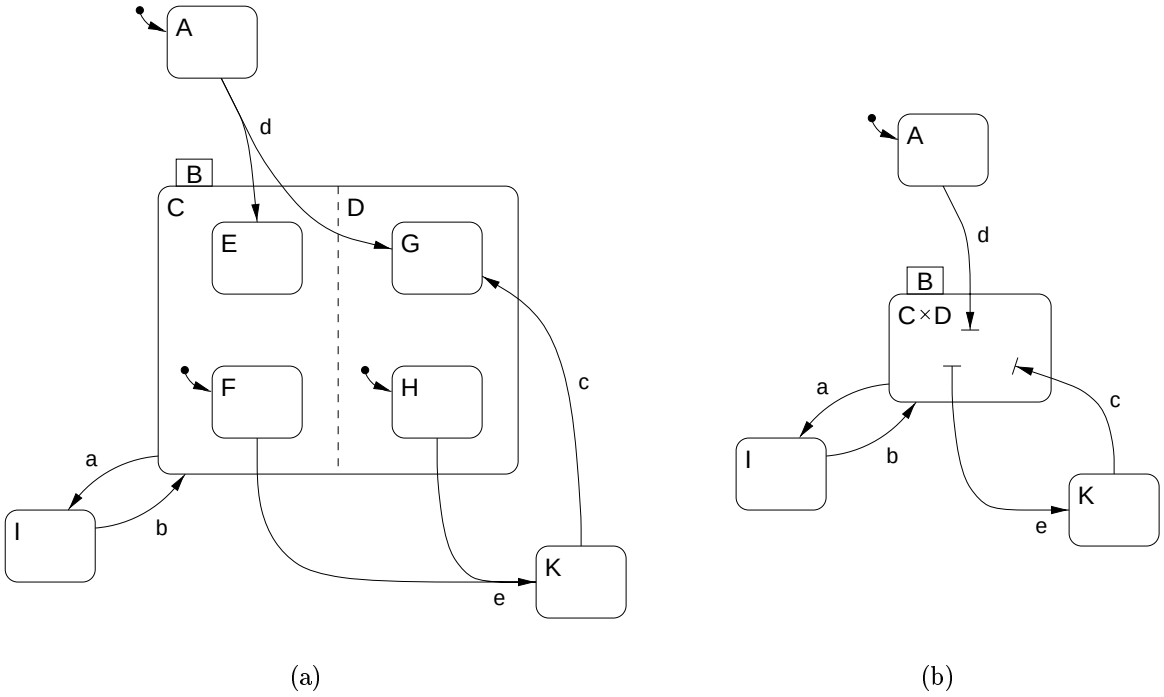


Abb. 12: Einen Zustand mit orthogonalen Subzuständen einnehmen und verlassen (in (a)); möglicher Ausgangspunkt beim Entwurf in (b)

Von besonderem Interesse sind Zustände, die aus orthogonalen Komponenten bestehen, da sie auf vielfältige Weise eingenommen und verlassen werden können. Abb. 12(a) zeigt ein Beispiel, das einige Möglichkeiten verdeutlicht. Einen einfachen Fall stellen diejenigen Zustandsübergänge dar, die durch die Ereignisse a und b ausgelöst werden. Der eine Übergang resultiert im Zustandspaar (F, H), das durch das Charakteristikum „Default-Zustand“ bestimmt wird, während der andere unabhängig vom gegenwärtigen Zustandspaar dazu führt, daß der Zustand B verlassen wird.

Im Falle des durch c ausgelösten Übergangs wird der Zustand G explizit bestimmt, während die andere Komponente des einzunehmenden Zustandspaares wiederum durch den Default-Zustand festgelegt ist. Eine bisher nicht behandelte Darstellungsmöglichkeit zeigen die beiden verbleibenden Übergänge auf. Durch den sich aufspaltenden

Pfeil, der mit dem Ereignis d beschriftet ist, werden beide Komponenten des Folgezustandspaares bestimmt; der zum Zustand K führende Pfeil legt fest, daß im Falle des Ereignisses e das Zustandspaar (F, H) der gegenwärtige Zustand sein muß, damit ein Übergang nach K stattfinden kann. Neben den in Abb. 12(a) dargestellten Möglichkeiten, einen mit Hilfe der AND-Dekomposition verfeinerten Zustand einzunehmen und zu verlassen, sind zahlreiche weitere vorstellbar; man denke z. B. daran, daß auch die History-Funktion, der bedingte und der selektive Eintritt verwendet werden können, um eine Komponente eines Zustandspaares festzulegen.

Entwirft man einen Statechart, so kann die genaue Struktur eines Zustands anfangs unbekannt sein, während die Art und die Anzahl der Übergänge zwischen diesem Zustand und anderen durchaus bereits feststehen kann. In solchen Fällen ist es nützlich, die Übergänge bereits einzuzeichnen, ihre Start- oder Zielzustände aber durch kleine Striche als „noch nicht festgelegt“ zu kennzeichnen. Abb. 12(b) zeigt das Ergebnis eines solchen Vorgehens für den Statechart aus Abb. 12(a).

Abschließend betrachten wir einen Aspekt, der für Realzeitsysteme von Bedeutung ist: Häufig werden Zustände mit zeitlichen Einschränkungen der Art „10 Sekunden warten, bevor die Schranke geschlossen wird“ oder „Nach 5 Minuten ohne Eingabe den Bildschirmschoner aktivieren“ verknüpft. Für solche Fälle sieht der Formalismus der Statecharts eine spezielle Notation vor. Die Grundlage bildet stets ein Ereignis der Form *timeout*(*event*, *number*), das nach einem Ereignis *event* generiert wird, sobald eine bestimmte Anzahl von Zeiteinheiten, die durch *number* festgelegt wird, verstrichen ist. Mit Hilfe eines solchen Ereignisses kann Einfluß darauf genommen werden, wie lange ein System in einem bestimmten Zustand verweilt.

Abb. 13 zeigt ein einfaches Beispiel. Durch die gezackte Linie am oberen Rand des Zustandes wird verdeutlicht, daß der Zustand einer zeitlichen Beschränkung unterliegt, die direkt unter ihr angegeben ist. Im vorliegenden Fall handelt es sich um eine obere Schranke; es ist ebenfalls möglich, ein Zeitintervall in der Form $t_1 < t_2$ oder eine untere Schranke anzugeben, was zur Folge hätte, daß Ereignisse im gegenwärtigen Zustand bis zu einem bestimmten Zeitpunkt nicht berücksichtigt würden, so daß eine Mindestverweildauer gewährleistet wäre. Das Ereignis *timeout* steht für *timeout(„Zustand betreten“, Obergrenze)*, durch das garantiert wird, daß sich das System höchstens für eine bestimmte Zeit, die durch *Obergrenze* vorgegeben wird, im betrachteten Zustand befindet.

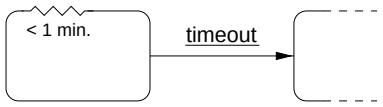


Abb. 13: Ein Zustand mit maximaler Verweildauer

3.4 Kommunikation: Ereignisse, Aktionen und Aktivitäten

Reaktive Systeme arbeiten weitgehend ereignisgesteuert. Sie reagieren auf externe Stimuli, und ihr Verhalten kann nach [Har87] als „Menge der erlaubten Folgen von Ein- und Ausgabe-Ereignissen, Bedingungen und Aktionen“ betrachtet werden. Harel definiert in seinen grundlegenden Artikeln über Statecharts den Begriff „Ereignis“ nicht explizit, jedoch kann man aufgrund seiner Erläuterungen und der von ihm vorgestellten Beispiele davon ausgehen, daß er sich auf die allgemeine Bedeutung dieses Wortes bezieht.[3]

In den vorangegangenen Abschnitten wurden einige Statecharts behandelt, die einfache Systeme beschreiben. Die Reaktionsmöglichkeiten der zugehörigen Systeme sind in diesen Beispielen auf Zustandswechsel beschränkt. Dabei wird keine Verbindung zwischen den einzelnen Zuständen und den Übergängen zwischen ihnen einerseits sowie den tatsächlichen Systemreaktionen andererseits, die Einflüsse auf die reale Welt bedeuten, hergestellt. Die Statecharts fungierten somit bisher ausschließlich als Kontrolleinrichtungen, die in Abhängigkeit von auftretenden Ereignissen sowie geltenden Bedingungen unter Berücksichtigung zeitlicher Gegebenheiten das Verhalten des gesamten Systems lediglich bedingen.

Die geschilderten Unzulänglichkeiten können durch eine Erweiterung des bisherigen Formalismus überwunden werden, die die Möglichkeit beinhaltet, Ereignisse innerhalb von Statecharts zu generieren. Kommt es zu einem Zustandsübergang, so kann dann nicht nur ein neuer Zustand eingenommen, sondern auch ein Ereignis erzeugt werden; ein derartiges Ereignis wird von Harel als *Aktion* bezeichnet.[4] In der grafischen Darstellung können Aktionen durch die Notation „. . . /*aktion*“ ausgedrückt werden, die der Beschriftung des entsprechenden Pfeils hinzuzufügen ist. Abb. 14(a) zeigt ein Beispiel, in dem bei einem Zustandsübergang ein Ereignis b generiert wird.

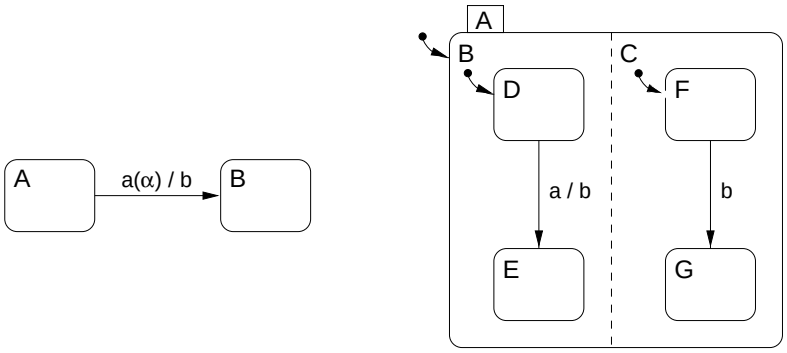
Aktionen sind Geschehnisse von extrem kurzer Dauer, und in Übereinstimmung mit der Brockhaus-Definition für „Ereignis“ können sie als momentane Vorkommnisse, d. h. als augenblicklich, betrachtet werden. Unter den Aktionen lassen sich zwei Arten unterscheiden:

- Aktionen, die unmittelbar auf die Umgebung des reaktiven Systems wirken (z. B. „Fahrkarte und Wechselgeld ausgeben“), sowie
- Aktionen, die Zustandsübergänge in orthogonalen Komponenten auslösen und somit nur einen indirekten Einfluß auf die Umgebung ausüben können.

Die Aktionen der ersten Kategorie sind Ausgaben des Systems, die in derjenigen Weise erfolgen, wie sie für Mealy-Automaten charakteristisch ist (siehe z. B. [HU79]).

[3] Die 18. Auflage des Grossen Brockhaus definiert den Begriff des *Ereignisses* wie folgt: 1) Begebenheit, (bedeutsames) Geschehnis; 2) Physik: Vorgang, dessen räumliche Ausdehnung und zeitliche Dauer vernachlässigbar klein sind; in der Theorie dargestellt als ein Raum-Zeit-Punkt; 3) Vorkommnis

[4] Bei einem Zustandsübergang kann eine *Aktion* ausgeführt werden, die ein *Ereignis* generiert. Da Harel eine Aktion und das durch sie generierte Ereignis ohnehin mit demselben Bezeichner versieht, kann man die beiden Begriffe in diesem Zusammenhang als Synonyme ansehen.



(a) (b)

Abb. 14: Aktion bei einem Zustandsübergang in (a); Aktion, die einen Übergang in einer orthogonalen Komponente bewirkt, in (b)

Aktionen, die der zweiten Gruppe angehören, sind vom Statechart selbst generierte Ereignisse, die in allen orthogonalen Komponenten registriert werden; Harel spricht in diesem Zusammenhang von *broadcast-communication*. Der Bereich, in dem auf Aktionen reagiert werden kann, ist somit stets auf den entsprechenden Statechart begrenzt (siehe [HG96]).[5]

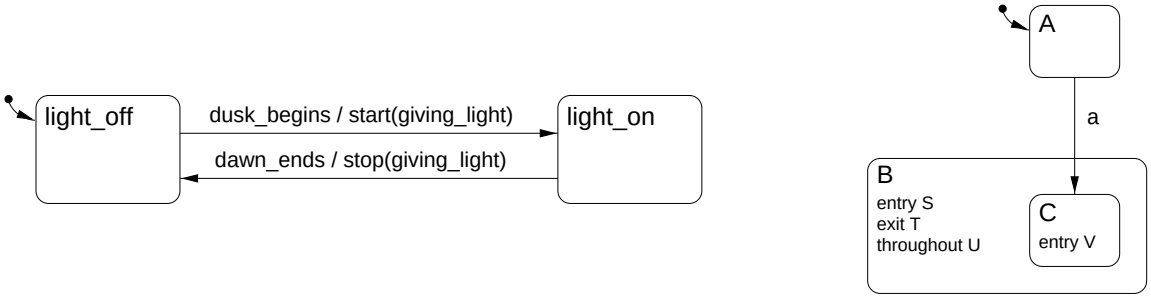
Abb. 14(b) verdeutlicht die Möglichkeit, Transitionen einer Komponente durch Transitionen einer orthogonalen Komponente zu steuern: Befindet sich das System im Zustand (D, F) und tritt das Ereignis a ein, so kommt es in der Komponente B zum Übergang in den Zustand E. Hierbei wird das Ereignis b generiert, das daraufhin in der Komponente C registriert wird und dort dazu führt, daß der Zustand G eingenommen wird. Dieses Beispiel ist sehr einfach, und die Abb. 14(b) läßt nur eine Interpretation zu. Es sind allerdings wesentlich kompliziertere Fälle vorstellbar, in denen es zu Zyklen kommen kann, wenn Ereignisse in einer bestimmten Reihenfolge generiert werden. Statecharts, die derartige Systeme beschreiben, sind zunächst mehrdeutig, so daß ihnen eine präzise Semantik zugeordnet werden muß. Von grundlegender Bedeutung ist in diesem Zusammenhang die Frage, ob in einem Zeitschritt das registrierte externe Ereignis sowie alle sich daraus ergebenden Aktionen abgearbeitet oder generierte Ereignisse erst in späteren Zeitschritten betrachtet werden; auf diesen Aspekt geht Abschnitt 5 näher ein.

Durch die Möglichkeit, Ereignisse selbst zu generieren, können Statecharts Einfluß auf ihre Umgebung ausüben. Da die beschriebenen Aktionen allerdings als augenblicklich betrachtet werden, während reale Systeme stets durch Abläufe gekennzeichnet sind, die eine bestimmte Zeitdauer beanspruchen, ist eine zusätzliche Erweiterung des Formalismus erforderlich. Harel führt zu diesem Zweck den Begriff der *Aktivität* ein,

[5] Der Broadcast-Mechanismus stellt neben der XOR- und der AND-Dekomposition das dritte wesentliche Charakteristikum von Statecharts dar; Harel beschreibt Statecharts in einer Kurzform folgendermaßen: „*statecharts = state-diagrams + depth + orthogonality + broadcast-communication*“.

mit dem Vorgänge bezeichnet werden, die Zeit benötigen.

Damit Statecharts Aktivitäten steuern können, sind zwei Aktionen vonnöten, die den Start und das Ende einer Aktivität zur Folge haben; aus diesem Grund ordnet Harel jeder Aktivität X die beiden Aktionen $start(X)$ und $stop(X)$ zu, die die entsprechenden Bedeutungen besitzen. Darüber hinaus kann mit der Bedingung $active(X)$ geprüft werden, ob die Aktivität X zum betrachteten Zeitpunkt ausgeführt wird. Abb. 15(a) verdeutlicht, wie zwei Aktionen eine Aktivität steuern können: Setzt die Abenddämmerung ein, so wird automatisch eine Außenbeleuchtung eingeschaltet, die während der gesamten Nacht leuchtet. Das Ende der Morgendämmerung führt schließlich dazu, daß das Licht ausgeschaltet wird.



(a) (b)

Abb. 15: Aktionen steuern in (a) nach dem Vorbild von Mealy-Automaten eine Aktivität; von Moore-Automaten stammende Eigenschaften in (b)

Es ist zu beachten, daß die Aktivitäten selbst zunächst nicht durch Statecharts spezifiziert werden. Harel schlägt vor, für die Spezifikation sequentieller Aktivitäten herkömmliche Programmiersprachen zu verwenden. Sind die Aktivitäten hingegen selbst reaktiver Natur, könnten auch sie ihrerseits durch einzelne Statecharts beschrieben werden, so daß Hierarchien von Statecharts und Aktivitäten denkbar sind. Das Programmsystem STATEMATE, das die Entwicklung reaktiver Systeme mit Hilfe von Statecharts ermöglicht und von der Firma i-Logix vertrieben wird, verwendet einen weiteren grafischen Formalismus (sogenannte *activity-charts*), um Aktivitäten zu spezifizieren (siehe [HLN+90]).

Aktionen und Aktivitäten können nicht nur in der bisher beschriebenen Form, sondern darüber hinaus auch auf der Grundlage des Mechanismus, der für Moore-Automaten charakteristisch ist (siehe wiederum [HU79]), ausgeführt werden. In diesem Fall werden Aktionen nicht mit Zustandsübergängen, sondern vielmehr mit einzelnen Zuständen des Statechart verknüpft.

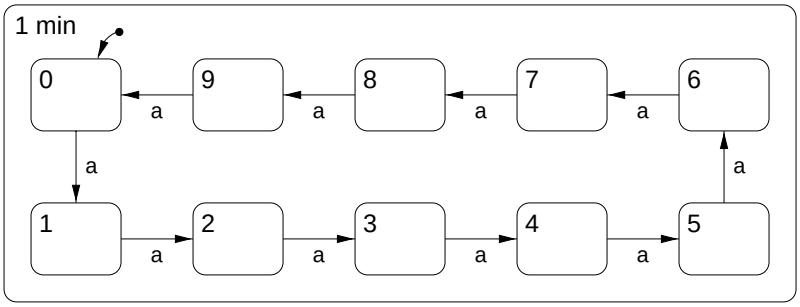
Indem man einen Zustand zusätzlich mit einem der Ausdrücke *entry* S oder *exit* S versieht, kann eine Aktion S ausgeführt werden, wenn er betreten oder verlassen wird. Ferner kann während der gesamten Zeit, in der ein System in einem Zustand verweilt, eine Aktivität X ablaufen; dieses kann man durch die Notation *throughout* X kennzeichnen, die äquivalent zur gleichzeitigen Beschriftung des Zustands mit *entry* $start(X)$

und *exit stop*(X) ist.

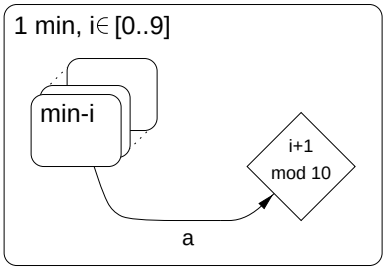
Abb. 15(b) zeigt für eine Aktivität U sowie für Aktionen S, T und V, in welcher Weise man Statecharts um die zusätzlichen Ausdrücke ergänzen kann, und macht außerdem deutlich, daß bereits Zustandshierarchien zu Nebenläufigkeit führen können: Wird nach dem Ereignis a der Zustand C betreten, so werden die Aktionen S und V gleichzeitig ausgeführt.

3.5 Weitere Eigenschaften

Nachdem in den vorangegangenen Abschnitten die wesentlichen Konzepte des Statechart-Formalismus vorgestellt worden sind, sollen nun einige denkbare Erweiterungen behandelt werden, die Harel ebenfalls in [Har87] beschreibt. Zum Erscheinungszeitpunkt des Artikels war den meisten dieser zusätzlichen Eigenschaften keine Syntax oder formale Semantik zugeordnet; es handelt sich überwiegend um Ideen, die durch die Anwendung des Formalismus im Rahmen realer Projekte entstanden sind und zu einer weiteren Vereinfachung der Darstellung beitragen können.



(a)



(b)

Abb. 16: Zustand mit mehreren Subzuständen gleicher Struktur in (a) sowie eine übersichtlichere Darstellung mit Hilfe eines parametrisierten Zustands in (b)

Häufig besitzen verschiedene Zustände eines Statechart dieselbe innere Struktur und weisen ähnliche Möglichkeiten für einen Zustandswechsel auf. Harel schlägt vor,

derartige Zustände in einem sogenannten *parametrisierten Zustand* zusammenzufassen, wobei die ursprünglichen Zustände durch einen Parameter identifiziert werden. In Abb. 16(a) ist ein Beispiel dargestellt, das die möglichen Zustände einer minutengenauen einstelligen Anzeige beschreibt. Allen Zuständen ist gemein, daß sie nach einem Zeitsignal *a* verlassen werden, damit die Folgeziffer angezeigt werden kann. Abb. 16(b) zeigt, wie die Darstellung durch einen parametrisierten Zustand vereinfacht wird.

Neben der Möglichkeit, für Zustände, die gemäß der XOR-Dekomposition verfeinert sind, parametrisierte Zustände zu verwenden, ist eine Anwendung dieser Idee auch im Falle von Zuständen mit orthogonalen Komponenten vorstellbar. Harel nennt als ein Beispiel eine Telefonanlage, die 1000 Telefone umfaßt; das Verhalten einer solchen Anlage könnte durch einen Statechart gemäß Abb. 17 beschrieben werden. Sowohl für die XOR-Dekomposition als auch für die AND-Dekomposition ist jedoch, wie Harel betont, häufig die letztendliche Programmiersprache am besten geeignet, um komplizierte Fälle von Parametrisierung zu spezifizieren.

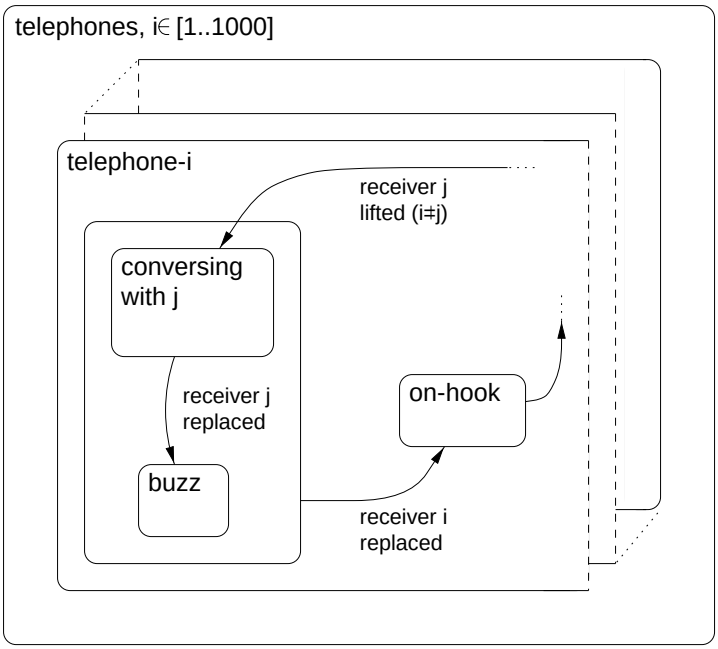


Abb. 17: Eine Telefonanlage mit 1000 Telefonen, beschrieben durch einen parametrisierten Zustand

Eine andere Erweiterung des Statechart-Formalismus betrifft die Organisation des Zustandsraumes. Bisher waren die Zustände eines Statecharts stets in einer baumartigen Struktur angeordnet, so daß jeder Zustand – mit Ausnahme desjenigen der höchsten Hierarchieebene – genau einen ihm direkt übergeordneten Zustand aufwies; diese Anordnung ist dem menschlichen Verständnis leicht zugänglich und bildet somit eine gute Arbeitsgrundlage.

Es ist jedoch auch vorstellbar, daß ein Zustand mehrere direkt übergeordnete Zustände hat, wodurch auf der Basis einer *Oder*-Beziehung (OR) *überlappende Zustände* entstehen. Dieser Fall kann z. B. dann eintreten, wenn zwei Subzustände zweier sich

ausschließender Zustände in anwendungsfachlicher Hinsicht große Ähnlichkeiten aufweisen oder gar identisch sind, kann aber auch einfach herbeigeführt werden, um weniger Transitionen einzeichnen zu müssen und somit die Komplexität des Diagramms zu verringern. In Abb. 18 ist ein Beispiel dargestellt: Dem Zustand E sind die beiden Zustände A und B unmittelbar übergeordnet; er kann nach einem der Ereignisse a, b, c oder d verlassen werden.

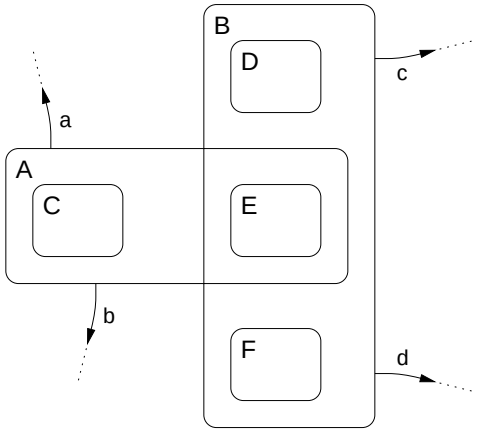
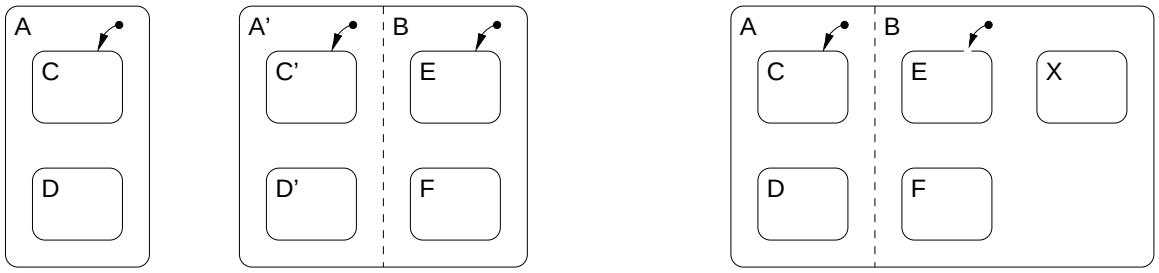


Abb. 18: Überlappende Zustände auf der Grundlage einer Oder-Beziehung

Auch Fälle ganz anderer Art, in denen überlappende Zustände nützlich sein können, sind denkbar. So kann es z. B. sein, daß ein Zustand A unter gewissen Umständen als orthogonale Komponente eines Zustandes B auftritt, in anderen hingegen als eigenständiger Zustand. Die zunächst naheliegende Lösung, im Sinne der Abb. 19(a) zu modellieren, weist den Nachteil der Redundanz auf, die sich besonders bei einer komplexen Struktur von A negativ bemerkbar macht.



(a) (b)

Abb. 19: Zustand mit und ohne zu ihm orthogonalen Zustand (Redundanz in (a) und „künstliche“ Lösung in (b))

Auch eine Lösung gemäß Abb. 19(b) ist nicht zufriedenstellend: Ein zusätzlich eingeführter Zustand X soll eingenommen werden, wenn A als eigenständiger Zustand

und B als „inaktiv“ aufgefaßt werden soll. Harel schlägt für Fälle der vorliegenden Art wiederum die Verwendung überlappender Zustände vor, so daß sich eine Darstellung wie in Abb. 20 ergibt: A und A' sind die Zustände C und D gemein.

Durch das Ereignis a kann ein Übergang nach (C, E) ausgelöst werden, während das Ereignis b dazu führt, daß lediglich A' eingenommen und B nicht aktiv wird. Das Ereignis c gestattet es, den Zustand C unabhängig davon, ob sich das System in A oder A' befindet, zu verlassen; hingegen führt d nur dann dazu, daß C verlassen wird, wenn B aktiv ist. Mittels e kann der Zustand D betreten werden, wobei der Bogen im zum Ereignis e gehörigen Pfeil anzeigt, daß die Begrenzung von A' nicht überschritten, der Produktzustand aus A und B betreten und somit der Folgezustand (D, E) eingenommen wird. Bei dem zum Ereignis f gehörigen Pfeil liegt ein anderer Fall vor: Mit D wird gleichzeitig A' eingenommen, und B ist inaktiv.

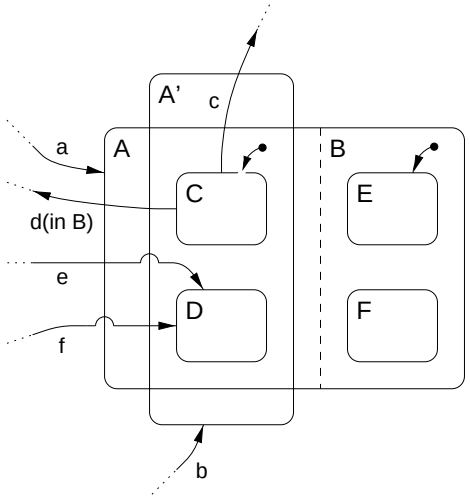


Abb. 20: Verbesserung der Darstellungen aus Abb. 19 mit Hilfe überlappender Zustände

Obwohl die grafische Umsetzung der Idee überlappender Zustände sehr anschaulich ist, bestehen einige Probleme, die die Verwendungsmöglichkeiten dieses Konzeptes erheblich einschränken können. So muß z. B. berücksichtigt werden, daß ein übergeordneter Zustand, dessen Begrenzungslinie übersprungen werden soll, dennoch vom System einzunehmen ist, wenn der zugehörige Pfeil keine Begrenzung eines anderen nicht atomaren Zustands, der ebenfalls dem letztlichen Zielzustand übergeordnet ist, kreuzt. Eine ausführliche Betrachtung der Möglichkeiten, die überlappende Zustände bieten, sowie der durch sie entstehenden Schwierigkeiten findet man in [HK92]. Die dortige Darstellung zeigt auch, daß eine geeignete formale Syntax und vor allem eine passende formale Semantik für überlappende Zustände ausgesprochen umfangreich sind, so daß die resultierende Komplexität einer entsprechenden Beschreibung den Nutzen der eleganten Darstellung zunichte machen kann.

Als eine weitere Ergänzung des Statechart-Formalismus zieht Harel den Gebrauch *temporaler Logik* in Betracht. Es wäre dann möglich, die Spezifikation eines Systemverhaltens um allgemeine Aussagen zu erweitern, die z. B. Verklemmungsfreiheit oder die

Einhaltung bestimmter globaler Zeitbeschränkungen garantieren könnten. Harel hält folgende Arten der Integration temporaler Logik für denkbar:

- Für eine gegebene Spezifikation mit Hilfe von Statecharts kann untersucht werden, ob sie einer Menge von Sätzen der temporalen Logik genügt.
- Menschen denken häufig in einer linearen, auf Szenarien basierenden Weise. Da sich Szenarien mit Hilfe temporaler Logik in geeigneter Form ausdrücken lassen, kann versucht werden, Statecharts aus Sätzen der temporalen Logik abzuleiten.
- Temporale Logik kann in Statecharts selbst verwendet werden, um komplexe Bedingungen zu formulieren. So ist es bisher nur mit Hilfe der History-Funktion möglich, das bisherige Systemverhalten zur Steuerung weiterer Abläufe einzusetzen. Durch die Verwendung temporaler Logik werden Bedingungen der Form

  $$(\text{in } B) \wedge \neg(\text{in } C) \textbf{ seit } (\text{in } A)$$

  möglich.

Auch das Konzept der *Rekursion* kann nach Ansicht Harels in den Statechart-Formalismus integriert werden; bei Zustandsübergängen könnte dann über einen Bezeichner in einen anderen Statechart verzweigt werden, welcher aus einem terminalen Zustand heraus wieder verlassen werden könnte.[6] In Anlehnung an Markov-Ketten, die durch eine besondere Form endlicher Automaten dargestellt werden können, sind darüber hinaus *probabilistische Statecharts* denkbar, die sich dadurch auszeichnen, daß anhand festgelegter Wahrscheinlichkeiten nichtdeterministisch Übergänge wählbar sind.

[6] Ein ähnlicher Ansatz wird im dynamischen Modell von OMT verfolgt (siehe Abschnitt 6 sowie [RBP+91] und [Rum95]).

4 Ein Beispiel

In den vorangegangenen Abschnitten wurden die syntaktischen Konstrukte von Statecharts weitgehend isoliert voneinander betrachtet. Um ihren Nutzen zu verdeutlichen, sollen nun anhand eines Anwendungsbeispiels einige Möglichkeiten aufgezeigt werden, wie sie zusammenwirken können.

Als Beispiel dient ein bereits existierendes Produkt: ein Heimtrainer.[1] Somit wird der Statechart-Formalismus im weiteren nicht für einen Entwurf, sondern zu Dokumentationszwecken verwendet. Der Heimtrainer ist ein in sich verständliches Beispiel. Er ist einerseits überschaubar genug, um hier behandelt zu werden, andererseits trotz seiner vermeintlichen Einfachheit bereits recht komplex.

4.1 Die Funktionen des Heimtrainers

Der Heimtrainer ist ein Trimmrad, das der Beinbewegung des Trainierenden einen Tretwiderstand entgegensetzt. Er ist vor allem für das Training der Beinmuskulatur und des Kreislaufs geeignet. An einer Bedien- und Anzeigeeinheit (siehe Abb. 21) kann man mit vier Tasten Einstellungen vornehmen und Daten ablesen.

Abb. 21: Die Bedien- und Anzeigeeinheit des Heimtrainers

[1] Es handelt sich hierbei um das Gerät „WIMFIT 110“ der Siegmann & Schröder GmbH, Hamburg.

Über die Taste „EIN/AUS“ kann der Heimtrainer ein- und ausgeschaltet werden. Ist er eingeschaltet, so können mit Hilfe der Taste „WAHL“ die bisher benötigte Zeit, die momentane Geschwindigkeit, die zurückgelegte Entfernung, die Gesamtdistanz aller Trainingseinheiten und der ungefähre Kalorienverbrauch angezeigt werden. Ferner ist es möglich, über die „WAHL“-Taste einen „Scan“-Modus zu aktivieren, in dem diese Anzeigen bei einer jeweiligen Verweildauer von ca. 5 Sekunden automatisch nacheinander durchlaufen werden.

Befindet sich die Anzeige im Zeit- oder Entfernungsmodus, so kann mit den beiden „SET“-Tasten eine Zeit oder Entfernung, die man zu absolvieren wünscht, vorgegeben werden. Das Gerät zählt in beiden Fällen bis Null herunter, läßt für ca. 8 Sekunden ein Alarmsignal ertönen und zählt dann im positiven Bereich weiter.

Wird der Heimtrainer für ungefähr 4 Minuten nicht benutzt, so erlischt die Digitalanzeige.[2]

4.2 Eine Verhaltensbeschreibung mit Hilfe von Statecharts

Betrachtet man die Bedien- und Anzeigeeinheit, so lassen sich zunächst einmal zwei Zustände unterscheiden: Der Heimtrainer ist entweder ein- oder ausgeschaltet (siehe Komponente *main* des Statecharts in Abb. 22). Betätigt man im Zustand *off* die Taste „EIN/AUS“, so wird der Heimtrainer eingeschaltet; der Tastendruck ist durch das Ereignis *ein_aus* gekennzeichnet. Ein Benutzer hat darüber hinaus die Möglichkeit, den Heimtrainer dadurch zu aktivieren, daß er einfach zu treten beginnt. Wir gehen davon aus, daß die kontinuierliche Tretbewegung in eine Folge elektrischer Impulse umgesetzt wird. Durch das Ereignis *cycle* verdeutlichen wir, daß ein solcher Impuls gesendet wird.[3]

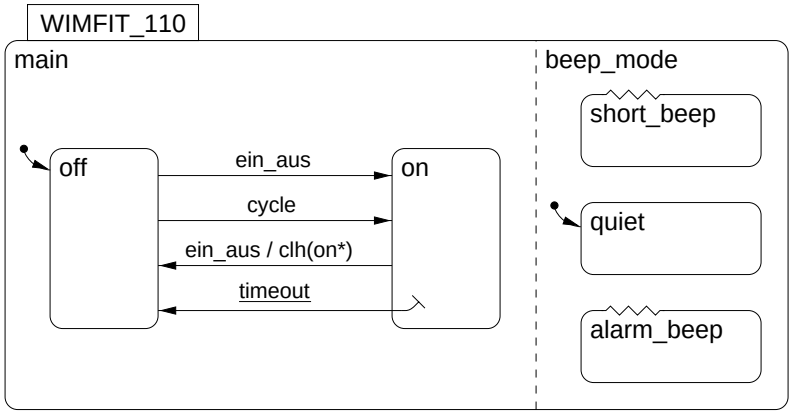


Abb. 22: Der Heimtrainer als Statechart

Ist der Heimtrainer eingeschaltet, so kann er mit Hilfe der Taste „EIN/AUS“ ausge-

[2]Dies gilt allerdings nicht uneingeschränkt (siehe Abschnitt 4.2).

[3]Da wir Ereignisse, die Tastendrücken entsprechen, gemäß den auf der Bedien- und Anzeigeeinheit aufgeführten Begriffen benannt haben, treten englische Bezeichnungen neben deutschen auf; alle übrigen Bezeichner entstammen dem Englischen.

schaltet werden. Wenn dieses Ereignis eintritt, wird die Aktion *clear-history*, abgekürzt durch *clh*, ausgelöst, die die Information über die bisher im Zustand *on* durchlaufene Folge von Subzuständen vollständig löscht (vergleiche Seite 17). Das Gerät wechselt selbständig in den Zustand *off*, wenn es für eine bestimmte Zeitspanne nicht benutzt worden ist. Da dies jedoch nicht uneingeschränkt gilt, beginnt der zum Ereignis *timeout* gehörige Pfeil innerhalb der Begrenzung des Zustandes *on*.

Wird die Taste „EIN/AUS“ gedrückt, so hat dies nicht nur einen Zustandswechsel zur Folge, sondern es ertönt darüber hinaus stets ein kurzer Piepton. Soll das Ereignis *ein_aus* verwendet werden, um in einen zugehörigen Zustand zu wechseln, so kann ein entsprechender Übergang nicht innerhalb eines der Zustände *on* und *off* liegen, sofern man sich an der in [HN96] beschriebenen Semantik orientiert (siehe Abschnitt 5.2.2). Dies ergibt sich aus der Tatsache, daß *on* und *off* im Falle des Ereignisses *ein_aus* verlassen werden; in solchen zunächst nichtdeterministischen Situationen hat stets derjenige Übergang, der von einem Zustand einer höheren Abstraktionsebene ausgeht, Priorität, so daß im vorliegenden Fall der Zustand, der mit dem Piepton verknüpft ist, überhaupt nicht eingenommen würde.[4]

Wir haben uns dafür entschieden, die Zustände *on* und *off* in einer Komponente *main* zu kapseln und eine zu dieser orthogonale Komponente *beep_mode* einzuführen, die u. a. den Zustand *short_beep* beinhaltet, der für den Piepton bei einem Tastendruck steht. Durch den Zustand *alarm_beep* wird erfaßt, daß der eingangs beschriebene Alarmton erklingt. Da dieses Alarmsignal durch einen Tastendruck, der einen kurzen Piepton zur Folge hat, beendet werden kann, ist auch *alarm_beep* Bestandteil von *beep_mode*. Im Anfangszustand des Heimtrainers erklingt kein Ton (Zustand *quiet*). Die Übergänge zwischen den einzelnen Subzuständen von *beep_mode* werden hier noch nicht betrachtet.

Abb. 23 zeigt, wie der Zustand *on* aufgebaut ist. Durch die Komponente *user_mode* wird gesteuert, unter welchen Umständen sich der Heimtrainer automatisch ausschaltet. Zunächst befindet sich der Heimtrainer im Zustand *user_idle*, der nach einer Zeit von 4 Minuten automatisch verlassen wird. Drückt der Benutzer eine der beiden „SET“-Tasten oder den „WAHL“-Knopf oder macht eine Tretbewegung, so erfolgt ein Übergang nach *user_busy*. Dieser Zustand wird im Regelfall sofort wieder verlassen. Hierbei ist zu beachten, daß der entsprechende Übergang nicht durch ein Ereignis ausgelöst wird; er ist lediglich von der Bedingung „*not in t_backwards*“ abhängig, die ausdrückt, daß die Zeit nicht rückwärts gezählt wird (siehe Komponente *time_mode* in Abb. 23 und Abb. 25(a)). Genau dies ist der bereits erwähnte Fall, in dem der Heimtrainer auch nach Ablauf von 4 Minuten weiterhin eingeschaltet bleibt.

Mit Hilfe der zu *user_mode* orthogonalen Komponente *scan_mode* wird festgehalten, ob sich der Heimtrainer im „Scan“-Modus befindet. Dieser ist in die Folge der Anzeigen für Zeit, Geschwindigkeit, Strecke, Gesamtdistanz und Kalorienverbrauch integriert

[4] Auch eine Aktion, ausgelöst bei einem Übergang, der durch *ein_aus* hervorgerufen wird, hilft hier nicht weiter, weil intern generierte Ereignisse nach [HN96] erst im folgenden Zeitschritt abgearbeitet werden. Da der Ausgangszustand dann bereits verlassen worden ist, kann die ausgelöste Aktion bei zumindest einem Zustandswechsel zwischen *on* und *off* nicht den Übergang in den Zustand, der für den Piepton steht, bewirken.

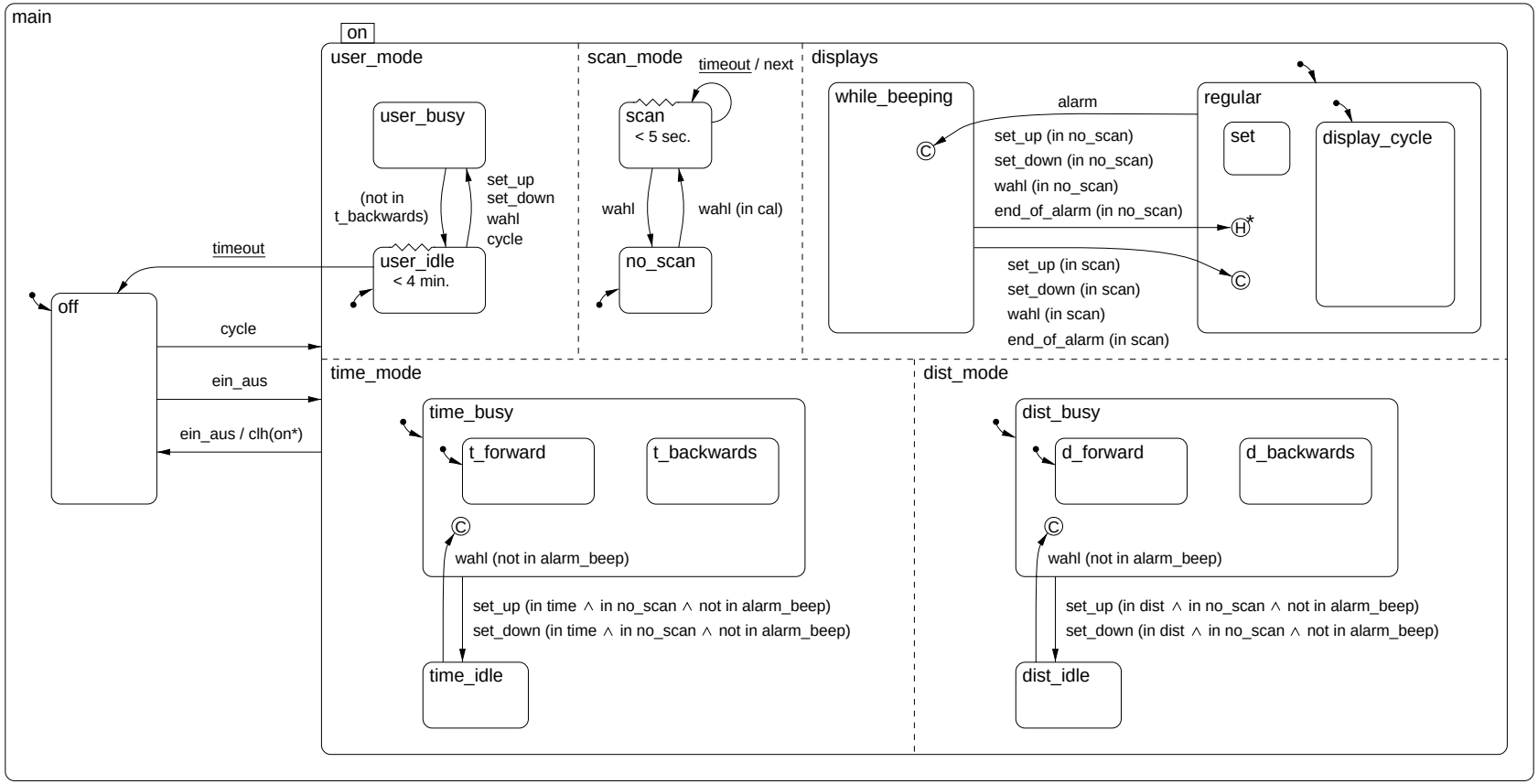


Abb. 23: Der Zustand *on* innerhalb von *main*

und daher ebenfalls über die „WAHL“-Taste zu erreichen: Wird der bisherige Kalorienverbrauch angezeigt (Bedingung „*in cal*“), so kann man mittels der Taste „WAHL“ den „Scan“-Modus aktivieren, der wieder verlassen werden kann, indem erneut „WAHL“ gedrückt wird. Alle 5 Sekunden wird *scan* durch *timeout* automatisch verlassen und erneut eingenommen. Hierbei wird stets das interne Ereignis *next* generiert, das in der Komponente *displays* verwendet wird, um die Anzeige fortzuschalten.

Über die Komponente *displays* wird erfaßt,

- welche Information angezeigt werden soll (z. B. die bisher benötigte Zeit),
- ob der Benutzer im Augenblick eine Zeitdauer oder Streckenlänge vorgibt und
- ob z. Zt. das Alarmsignal, welches anzeigt, daß eine zuvor eingestellte Zeitdauer abgelaufen oder eine festgelegte Distanz absolviert worden ist, ertönt.

Betätigt der Benutzer die „WAHL“-Taste oder eine der beiden „SET“-Tasten, so hängt die Wirkung davon ab, welche dieser Umstände vorliegen, d. h. welcher Subzustand innerhalb von *displays* zuletzt eingenommen wurde. Mit Hilfe des Zustands *regular* wird festgelegt, welche Information anzuzeigen ist: Entweder ist dies eine der Angaben über Zeit, Geschwindigkeit, Strecke, Gesamtdistanz oder Kalorienverbrauch (Zustand *display_cycle*), oder der Benutzer ist im Begriff, eine Zeit oder eine Streckenlänge vorzugeben (Zustand *set*). Das Ereignis *alarm*, welches signalisiert, daß die Zeit oder die Strecke auf Null heruntergezählt worden ist, und in den beiden Komponenten *time_mode* und *dist_mode* ausgelöst werden kann (siehe Abb. 25), führt zu einem Übergang in den Zustand *while_beeping*, durch den festgehalten wird, welche Information nach einem Tastendruck während des Alarmtons oder nach Ende dieses Signals angezeigt werden soll. Der Zustand *while_beeping* kann verlassen werden, indem man die „WAHL“- oder eine der beiden „SET“-Tasten drückt; durch das Ereignis *end_of_alarm*, das in der Komponente *beep_mode* nach dem Ende des Alarmtons erzeugt wird (siehe Abb. 27), wird *while_beeping* automatisch verlassen.

Abb. 24 zeigt den Zustand *displays* im Detail. Mit Hilfe der Subzustände von *display_cycle* wird festgehalten, welche der fünf bereits vorgestellten Angaben anzuzeigen ist. Befindet sich der Heimtrainer nicht im „Scan“-Modus, so kann man zur jeweils nächsten Anzeige wechseln, indem man die „WAHL“-Taste drückt. Durch das interne Ereignis *next*, das in der Komponente *scan_mode* generiert wird (siehe Abb. 23), kann der Anzeige-Zyklus automatisch durchlaufen werden. Wird die Zeit oder die Streckenlänge angezeigt, so kann mittels der beiden „SET“-Knöpfe ein Übergang nach *time_set* oder *dist_set* erfolgen. Diese beiden Zustände ermöglichen es dem Benutzer, die zu absolvierende Trainingszeit oder eine Distanz vorzugeben, und können über die „WAHL“-Taste wieder verlassen werden.

Erfolgt durch das Ereignis *alarm* ein Übergang nach *while_beeping*, so wird die Information über den innerhalb von *regular* zuletzt besuchten Zustand gespeichert, indem ein entsprechender Subzustand eingenommen wird. Befindet sich der Heimtrainer im „Scan“-Modus, so wird diese Information beim Verlassen von *while_beeping* dazu benutzt, über einen bedingten Eintritt zur nächsten Anzeige innerhalb des Zyklus zu schalten. Ist hingegen *no_scan* der aktuelle Zustand innerhalb von *scan_mode*

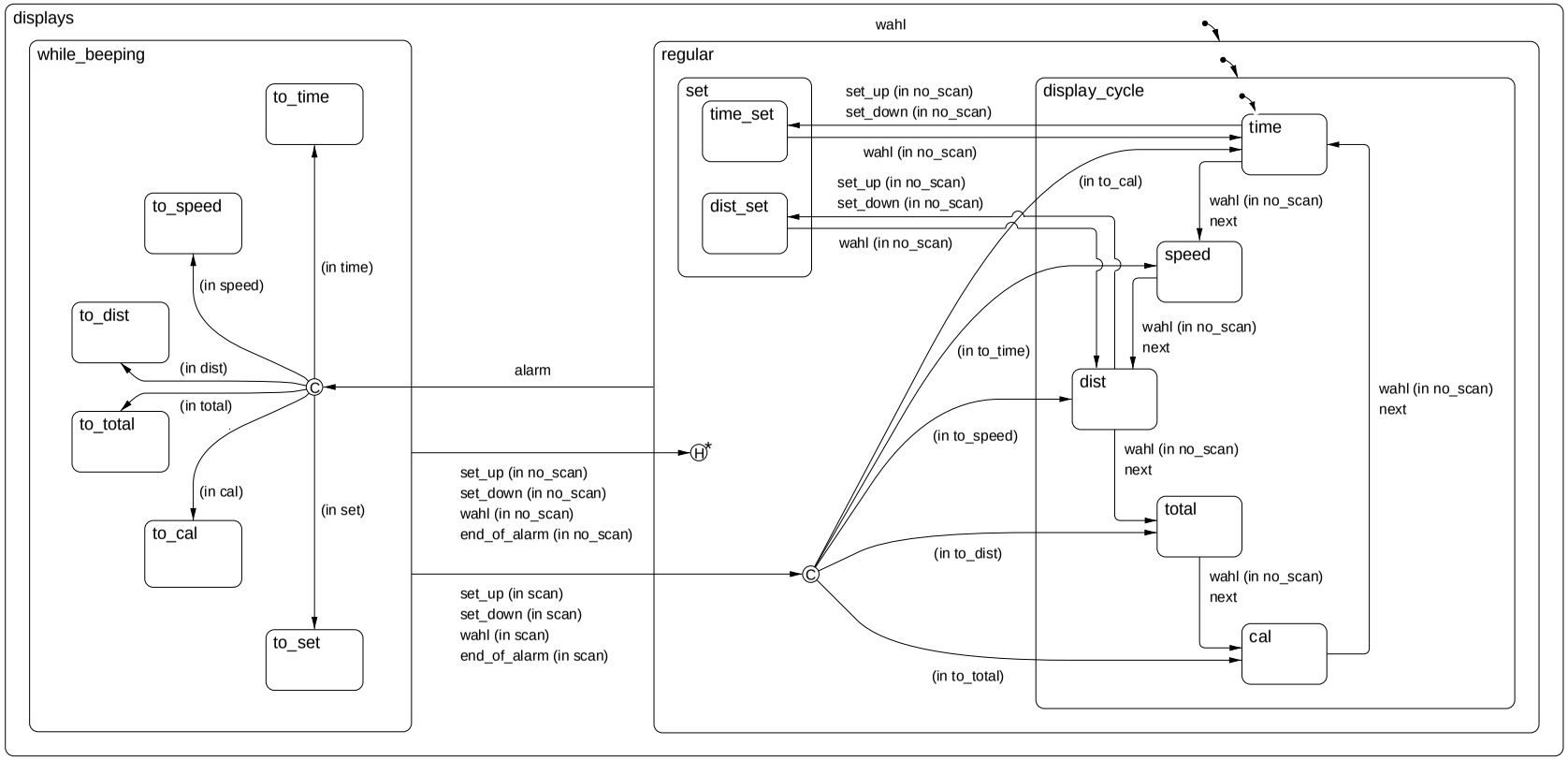
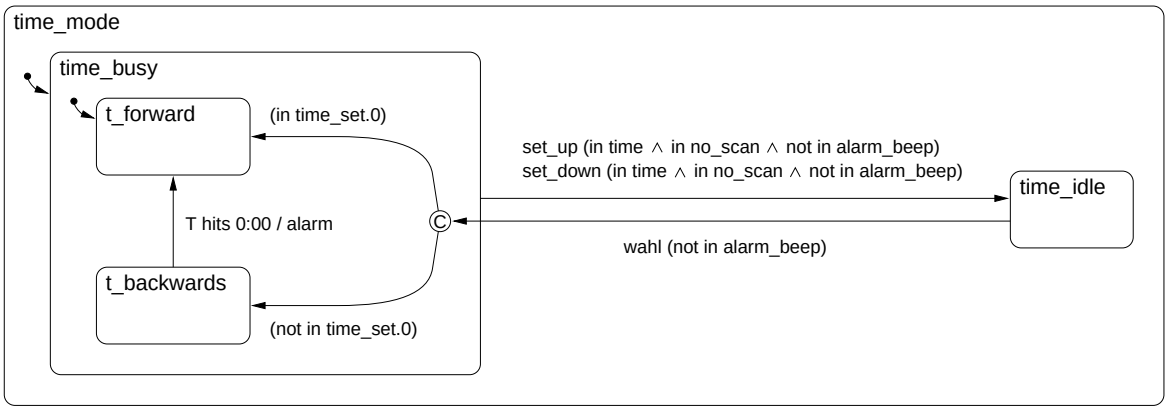
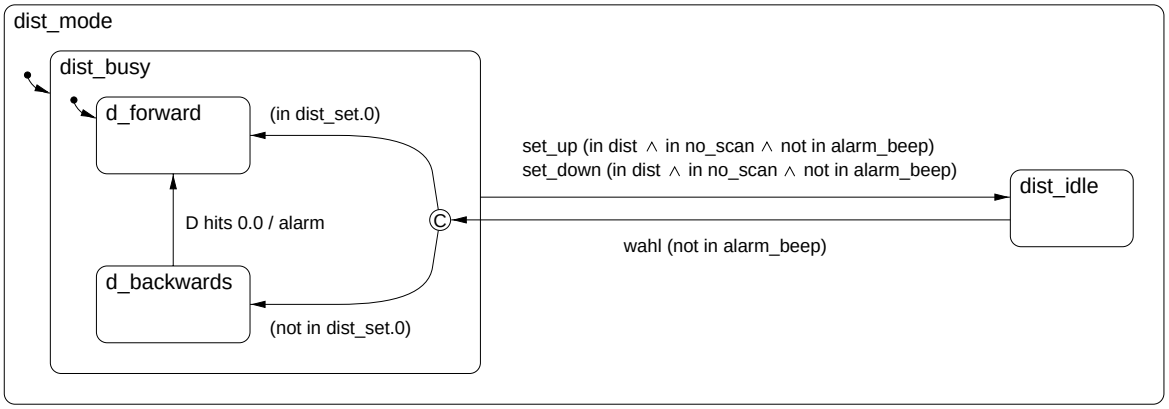


Abb. 24: Die Komponente *displays*

(siehe Abb. 23), so bewirken die Ereignisse, durch die *while_beeping* verlassen wird, eine Rückkehr zur bisherigen Anzeige; in diesem Fall ist die mittels eines Subzustandes von *while_beeping* gespeicherte Information redundant, weil die History-Funktion verwendet werden kann.[5]



(a)



(b)

Abb. 25: Die Komponenten *time_mode* und *dist_mode*

Die beiden bisher noch nicht erläuterten Komponenten des Zustands *on* sind *time_mode* und *dist_mode* (siehe Abb. 23), über die festgehalten wird, ob die Zeit oder die Strecke im Augenblick nicht fortzuschreiben ist, da der Benutzer gerade einen Wert vorgibt, oder ob vor- oder rückwärts zu zählen ist. Da *time_mode* und *dist_mode* die gleiche Struktur besitzen (siehe Abb. 25), beschränkt sich die folgende Darstellung

[5] Da sich der Heimtrainer nicht gleichzeitig im „Scan"- und im „Set"-Modus befinden kann, ist der Zustand *to_set* innerhalb von *while_beeping* ohne Nutzen; er ist dennoch eingezeichnet worden, um alle grundsätzlichen Möglichkeiten aufzuführen.

auf den Zustand *time_mode*.

Wird der Heimtrainer eingeschaltet, so wird die Zeit fortgeschrieben (Zustand *time_busy*), und es wird vorwärts gezählt (Zustand *t_forward*). Während der Zeitanzeige führt jede der beiden „SET"-Tasten in den Zustand *time_idle*, sofern sich der Heimtrainer nicht im „Scan"-Modus befindet und kein Alarmsignal ertönt. Innerhalb von *time_idle* werden keine Zeiteinheiten gezählt. Mittels der „WAHL"-Taste wird die eingestellte Zeit bestätigt (siehe Zustand *regular* in Abb. 24); in *time_mode* führt dies dazu, daß im Regelfall rückwärts gezählt wird (Übergang nach *t_backwards*). Die Bedingung „*not in time_set.0*" drückt hierbei aus, daß der Benutzer einen größeren Wert als Null vorgegeben hat (siehe Zustand *time_set* in Abb. 26).[6] Hat der Benutzer hingegen den Wert Null eingestellt, so wird vorwärts gezählt. Erreicht die rückwärts gezählte Zeit, die in der Variablen T gespeichert ist, die Nullmarke, so wird das Ereignis *alarm* generiert, das das Alarmsignal auslöst.[7]

Abb. 26 zeigt den Aufbau der beiden Subzustände von *set*, der Zustände *time_set* und *dist_set*. Der Zustand *time_set* konnte in eleganter Form als parametrisierter Zustand (siehe Abschnitt 3.5) realisiert werden; die grafische Darstellung drückt den Sachverhalt in kompakter und präziser Form aus:

- Es gibt 100 Subzustände, bezeichnet mit *0* bis *99*, die jeweils einer einstellbaren Zeit in Minuten entsprechen.
- Die Zustände sind zyklisch angeordnet; mit *set_up* wechselt der Heimtrainer in den Zustand mit der nächsthöheren, mit *set_down* in denjenigen mit der nächstniedrigeren Zahl.

Demgegenüber wirkt die für *dist_set* gewählte Struktur ungeeignet; die Subzustände sind einzeln eingezeichnet, und anstelle eines Verhaltensmusters wird jeder Übergang explizit aufgeführt. Den Grund für die Wahl dieser Struktur stellt eine beobachtete Anomalie dar: Über *set_up* sind Entfernungen bis zu 163 km einstellbar, während *set_down* im Falle des Zustands *0* direkt in den Zustand *160* führt. Dieser Umstand macht deutlich, daß eine Erweiterung des Statechart-Formalismus durch eine (grafisch orientierte) Sprache, die solche und wesentlich kompliziertere Strukturen in überschaubarer, auf das Wesentliche reduzierter Form beschreibt, wünschenswert ist.

Die Komponente *main* ist mit dieser abschließenden Betrachtung des Zustands *set* vollständig behandelt worden. Der zu *main* orthogonale Zustand *beep_mode* ist sehr einfach aufgebaut (siehe Abb. 27). Im Zustand *quiet* führt jeder Tastendruck dazu, daß *short_beep* eingenommen wird; dies gilt für den „EIN/AUS"-Knopf uneingeschränkt, während im Falle einer der drei anderen Tasten nur dann ein kurzes Piepsignal ertönt, wenn der Heimtrainer eingeschaltet ist. Beginnt der Benutzer im Zustand *off* mit einer Tretbewegung, so wird über das Ereignis *cycle* ebenfalls ein Piepton erzeugt.

Wenn in einer der Komponenten *time_mode* und *dist_mode* (siehe Abb. 25) das interne Ereignis *alarm* generiert wird, so kommt es in *beep_mode* zu einem Übergang nach

[6] Wie die Bedingung „*not in time_set.0*" zeigt, können qualifizierte Bezeichner verwendet werden, um einen Bezug auf Subzustände zu ermöglichen.

[7] In *dist_mode* wird die Variable D verwendet, um die noch zu absolvierende Strecke zu speichern.

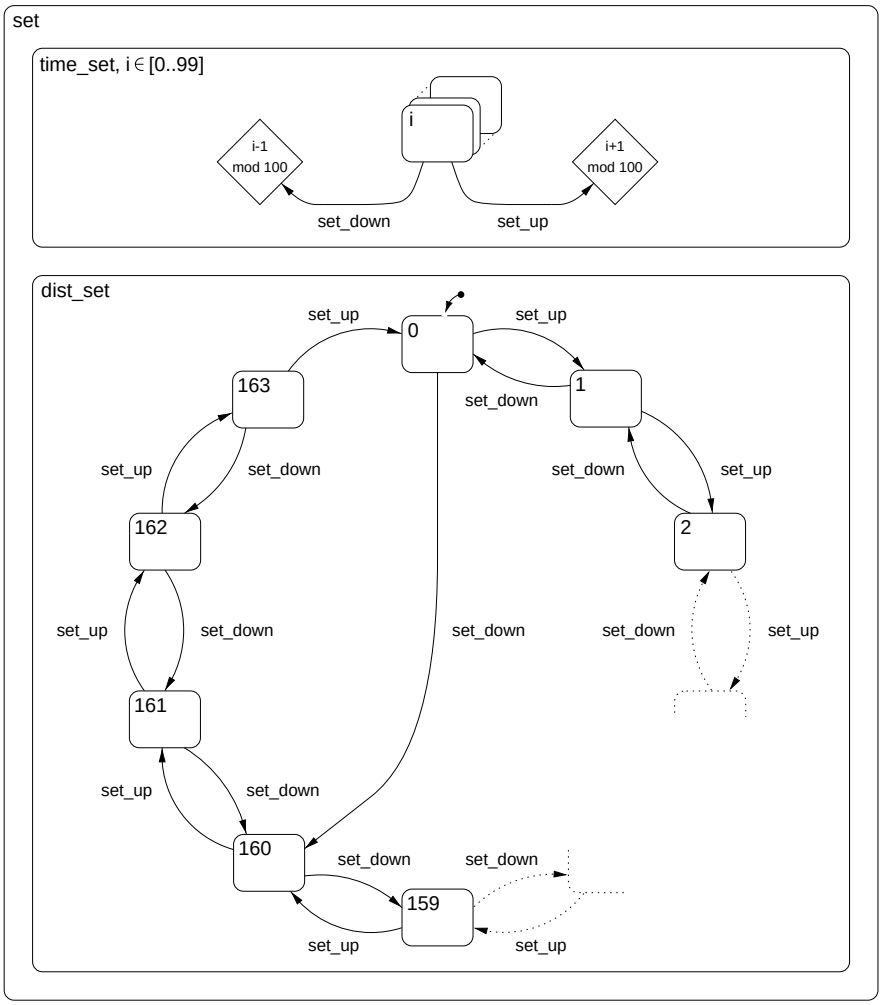


Abb. 26: Die Zustände *time_set* und *dist_set*

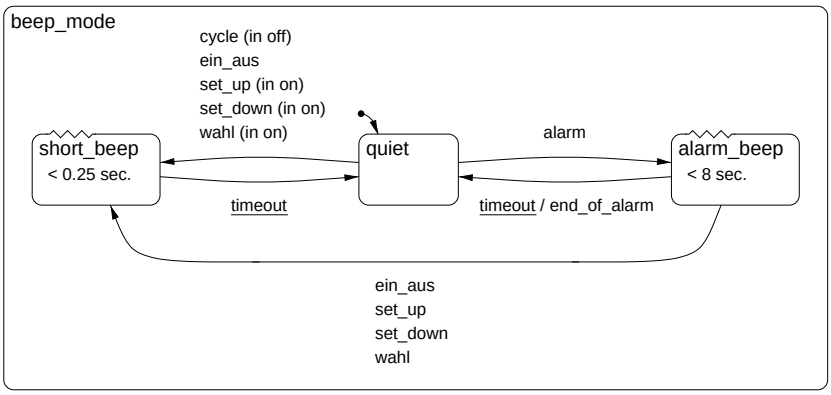


Abb. 27: Die Komponente *beep_mode*

alarm_beep und dazu, daß das bereits mehrfach erwähnte Alarmsignal ertönt. Dieses Signal kann der Benutzer beenden, indem er eine beliebige Taste drückt; auch hierbei ist ein kurzer Piepton zu hören. Die Zustände *short_beep* und *alarm_beep* werden jeweils nach einer festgelegten Zeitspanne verlassen, wobei der Übergang von *alarm_beep* nach *quiet* auch bedeutet, daß das Ereignis *end_of_alarm*, das in der Komponente *displays* die Rückkehr in den Zustand *regular* ermöglicht (siehe Abb. 24), generiert wird.

4.3 Einige abschließende Bemerkungen

Das Beispiel des Heimtrainers verdeutlicht nicht nur, wie die einzelnen Konstrukte des Statechart-Formalismus verwendet werden können, sondern zeigt auch dessen Grenzen auf: Detaillierte Beschreibungen sind für kleine Systeme recht schnell erstellbar, wenn man erst einmal einen Modellierungsansatz entwickelt hat; um komplexe Systeme mit Hilfe von Statecharts beschreiben zu können, ist eine grundlegende Vorgehensweise, in die die Verhaltensbeschreibung durch Statecharts eingebunden werden kann (z. B. ein objektorientierter Ansatz), unverzichtbar.

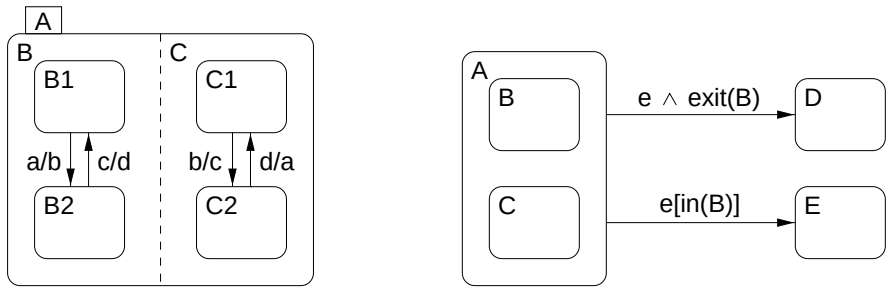
Obwohl der Heimtrainer nur wenige Funktionen aufweist, ist die Beschreibung in Abschnitt 4.2 nicht vollständig. Dies liegt zum einen daran, daß bestimmte Ereignisse nicht berücksichtigt worden sind (hierzu zählt z. B. das Ausfallen der Stromversorgung infolge einer schwachen Batterie), rührt aber vor allem daher, daß grundlegende Aktivitäten nicht mit den Statecharts durch Aktionen verknüpft worden sind. So kann durch den Zustand *time* innerhalb der Komponente *displays* (siehe Abb. 24) zwar festgehalten werden, daß die anzuzeigende Information die Zeit ist; die Zeit wird jedoch nicht tatsächlich fortgeschrieben.[8] Die Initialisierung der benötigten Variablen (z. B. von T für die Zeit) erfaßt die Darstellung in Abschnitt 4.2 ebenfalls nicht; sie könnte durch Aktionen erfolgen, die ausgeführt werden, wenn der Zustand *on* betreten wird.

[8] Dies könnte durch eine orthogonale Komponente modelliert werden.

5 Syntax und Semantik von Statecharts

Den in den vorangegangenen Abschnitten vorgestellten und anhand des Beispiels in Abschnitt 4 verdeutlichten grafischen Notationen liegen eine formale Syntax, die die Struktur von Statecharts exakt beschreibt, und eine formale Semantik, die eine Interpretation der Zustandsdiagramme darstellt und die möglichen Folgen von Zustandsübergängen bestimmt, zugrunde. Beide Aspekte werden in diesem Abschnitt angesprochen, um die wesentlichen Konstrukte vorzustellen und eine der Grundproblematiken im Zusammenhang mit der Semantik von Statecharts zu erläutern. Es wird hierbei weitgehend auf mathematische Ausführungen verzichtet, da im wesentlichen die Konzepte von Interesse sind; für eine ausführlichere Behandlung der Thematik sei auf die angegebene Literatur verwiesen. Nicht weiter betrachtet werden auch viele der in Abschnitt 3.5 dargestellten Erweiterungen; ihnen ist überwiegend noch keine formale Syntax und Semantik zugeordnet worden.[1]

Während die formale Syntax für Statecharts verhältnismäßig leicht zu formulieren und zu verstehen ist, birgt die Entwicklung einer geeigneten formalen Semantik einige Schwierigkeiten in sich. Nach [Har87] liegt die Hauptschwierigkeit hierbei nicht, wie man annehmen könnte, in der Hierarchiebildung oder der Orthogonalität; diese Punkte können durch Umwandlung in endliche Automaten behandelt werden, obwohl ein solches Verfahren nicht unproblematisch ist. Von besonderer Bedeutung ist vielmehr die Frage, wann die innerhalb eines Statecharts generierten Ereignisse in den orthogonalen Komponenten registriert werden, da sich aus einer solchen Festlegung eine Reihe von Problemen, z. B. Schleifen aus generierten und registrierten Ereignissen, ergeben können. Befindet sich beispielsweise das zum Statechart aus Abb. 28(a) gehörende



(a) (b)

Abb. 28: Ein Ereignis-/Aktionszyklus in (a) und die Frage nach dem Registrierungszeitpunkt von Zustandswechseln in (b)

System in den Zuständen *B1* und *C1*, wenn das Ereignis *a* eintritt, so werden die Zu-

[1] Eine Beschreibung der formalen Syntax und Semantik überlappender Zustände findet der interessierte Leser in [HK92]. Kontinuierliche Prozesse werden in [KP91] formal behandelt.

standspaare $(B2, C1)$, $(B2, C2)$, $(B1, C2)$ und $(B1, C1)$ immer wieder durchlaufen, falls die festgelegte Semantik einen derartigen Zyklus nicht verhindert. In diesem Zusammenhang können außerdem die Bedingung $in(state)$ sowie die Ereignisse $entered(state)$ und $exit(state)$, die automatisch erzeugt werden, wenn ein Zustand betreten bzw. verlassen wird, betrachtet werden; in Abb. 28(b) wäre diesbezüglich zu überlegen, ob das Ereignis e einen Übergang nach D oder nach E auslöst, wenn das System im Zustand B verweilt.

In Rahmen dieser Ausführungen werden zunächst einige Aspekte der formalen Syntax sowie zusätzliche Notationen und Definitionen vorgestellt. Darauf aufbauend erläutert Unterabschnitt 5.2 zwei mögliche Semantikansätze, die anhand einer gegebenen Zustandskonfiguration, einer Menge simultaner (externer) Ereignisse und einer Menge simultaner (externer) Bedingungen die Menge der möglichen Folgekonfigurationen bestimmen; ihr wesentlicher Unterschied besteht in dem Zeitpunkt, zu dem Veränderungen, die durch die Zustandsübergänge in einem Statechart bewirkt werden, wahrgenommen werden können.

5.1 Formale Syntax

Seit der Formalismus der Statecharts als grafisches Darstellungsmittel zur Beschreibung des Verhaltens von Systemen eingeführt worden ist, sind eine Reihe von Varianten entstanden, die die ursprüngliche Form ergänzen oder verändern.[2] Da die einzelnen Ausprägungen jedoch überwiegend auf der von Harel und seinen Koautoren vorgeschlagenen Form basieren, sind die in diesem Abschnitt gegebenen Erläuterungen zur formalen Syntax weitgehend an [HPSS87] und [HN96] orientiert und auf die Grundform der Statecharts ohne die in Abschnitt 3.5 vorgestellten Erweiterungen bezogen. Sie dienen hauptsächlich dazu, die Komponenten der Statecharts, ihre Eigenschaften und ihre Verwendung zusammenfassend darzulegen sowie weitere Begriffe und Notationen einzuführen, die die Ausführungen in Abschnitt 5.2 verständlicher machen;[3] spezielle Begriffe und Eigenschaften, die sich nur auf einen der ab Seite 44 erläuterten Semantikansätze beziehen, werden entsprechend gekennzeichnet.

- *Zustände:*
  Jedem Zustand eines Statecharts wird einer der Typen *XOR*, *AND* oder *basic* (*Basiszustand*, d. h. „besitzt keine Subzustände“) zugeordnet, so daß jeweils zwei Zustände in genau einer der Beziehungen „ausschließend“, „orthogonal“ oder „hierarchisch“ stehen; anhand eines Zustandsbaumes zum Statechart ist erkennbar, welche Beziehung auf ein gegebenes Zustandspaar zutrifft. Abb. 29 auf Seite 43 zeigt ein Beispiel: Während die Zustände B und E sowie C und D sich jeweils gegenseitig ausschließen, steht das Zustandspaar $(G, D2)$ in orthogonaler und das Zustandspaar (A, H) in hierarchischer Beziehung. Für den besonderen

[2] Einen Überblick über einen Großteil der bis 1994 entwickelten Ansätze gibt [vdB94].

[3] Hier nicht näher betrachtete formale Einzelheiten, deren Behandlung dem einführenden Charakter dieses Abschnitts widersprechen würde, können [HPSS87] und [HN96] entnommen werden.

Zustand *root*, der die oberste Ebene der Systembeschreibung darstellt, gilt, daß er kein Subzustand eines anderen Zustands ist. Einem Zustand vom Typ *XOR* können ein History-Symbol sowie eine Reihe direkter Subzustände zugewiesen werden. Außerdem erlaubt die Syntax für jeden verfeinerten Zustand die Definition einer sogenannten Default-Menge, die Subzustände oder History-Symbole verschiedener Hierarchieebenen enthalten kann.

- *Ausdrücke:*
  Die Menge der Ausdrücke wird auf der Basis der natürlichen Zahlen und einer Menge von Variablen gebildet, indem algebraische Operationen auf schon erstellte Ausdrücke angewendet werden. Ändert sich der Wert eines solchen Ausdrucks, der in Bedingungen und Aktionen verwendet werden kann, so wird ein entsprechendes Ereignis generiert (z. B. $changed(v)$). Für den Ansatz der Microsteps[4] wurde zudem zu jedem Ausdruck v ein zusätzlicher Ausdruck $current(v)$ definiert, der den aktuellen Wert von v während eines Steps enthält.

- *Bedingungen:*
  Ähnlich wie Ausdrücke werden Bedingungen von einer Basismenge ausgehend definiert; diese besteht aus sogenannten primitiven Bedingungen und Bedingungen der Form $in(s)$, in denen festgehalten wird, ob das System momentan in einem Zustand s verweilt, sowie der Form $active(a)$, um prüfen zu können, ob eine Aktivität a zum gegenwärtigen Zeitpunkt läuft. Im Microstep-Ansatz sind zusätzlich die Bedingungen $not\_yet(e)$ und $current(\gamma)$ vorgesehen, aus denen ersichtlich ist, ob das Ereignis e in der Kettenreaktion schon eingetreten ist und welchen Wert die Bedingung γ zur Zeit hat. Weitere Bedingungen lassen sich erstellen, indem man die Verknüpfungen $\vee$, $\wedge$ und $\neg$ auf durch $R \in \{=, <, >, \neq, \leq, \geq\}$ verbundene Ausdrücke und bereits formulierte Bedingungen anwendet. Für die in Aktionen und Ereignissen verwendbaren Bedingungen werden wie für Ausdrücke Ereignisse generiert, wenn sich ihr Wert ändert. Hierbei bedeutet das Ereignis $true(\gamma)$, daß γ *wahr* wird, während $false(\gamma)$ dafür steht, daß γ den Wahrheitswert *falsch* annimmt.

- *Ereignisse:*
  Auf der Grundlage einer Menge primitiver Ereignisse, der Ereignisse $true(\gamma)$ und $false(\gamma)$ für Bedingungen γ, $changed(v)$ für Ausdrücke v, $exit(s)$ und $entered(s)$ für Zustände s sowie dem speziellen Ereignis *timeout* können durch Verknüpfungen mit $\vee$ und $\wedge$ neue Ereignisse gebildet werden. Zusätzlich gelten auch mit Bedingungen verbundene Ereignisse der Form $e[\gamma]$ als Ereignisse, die sowohl in Bedingungen als auch für Beschriftungen von Transitionen verwendet werden können.

- *Aktionen:*
  Einzelne Aktionen a sowie durch Semikola gebildete Aktionsgruppen $a_1; a_2; \ldots; a_n$

[4] siehe Seite 45ff.

können für die Beschriftung von Transitionen eingesetzt werden. Sie bestehen aus primitiven Ereignissen und aus Zuweisungen von Bedingungen an primitve Bedingungen sowie von Ausdrücken an Variablen. In [HN96] werden die schon im Rahmen von [HPSS87] vorgeschlagenen Aktionen zum Starten und Beenden von Aktivitäten zur Syntax hinzugefügt; durch die Aktion *schedule*(a,t) wird es zusätzlich ermöglicht, die Ausführung der Aktion a um einen bestimmten Zeitraum t zu verschieben.

- *Transitionen:*
  Eine Transition t besitzt eine Quellmenge X von Zuständen und eine Zielmenge Y, die sowohl Zustände als auch History-Symbole umfassen kann.[5] Ihre Beschriftung l, die als *label* bezeichnet wird, besteht aus einem Ereignis-Aktionspaar e/a. Häufig wird auch das Tripel (X, l, Y) als Transition bezeichnet. Transitionen können zur Ausführung ausgewählt werden, wenn sich das System in X befindet[6] und das Ereignis e eintritt; bei dem zugehörigen Zustandsübergang wird X verlassen, die Aktion a ausgeführt und Y betreten.

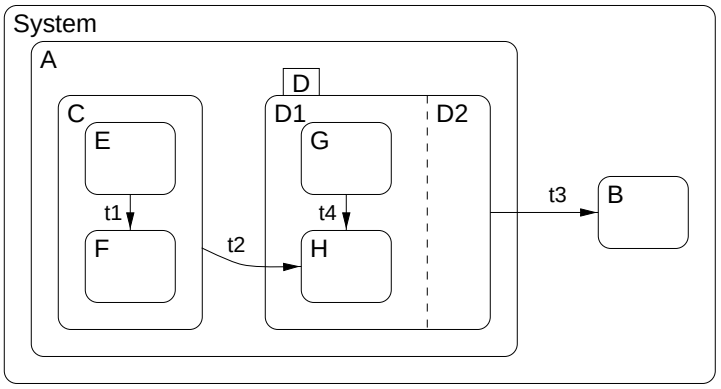
- *Statische Reaktionen:*
  Die in [HN96] eingeführten statischen Reaktionen sind eine einfache Darstellung für einen Spezialfall von Transitionen. Wird in einem Zustand Z eine Aktion a jedesmal, wenn das Ereignis $e[\gamma]$ eintritt und Z nicht verlassen wird, ausgeführt, so könnte man dies durch eine orthogonale Komponente in Z, die nur einen Zustand sowie eine mit $e\,[\gamma]/a$ beschriftete Schleife an diesem Zustand enthält, darstellen. Eine mit der gleichen Bedeutung versehene statische Reaktion erhält man, indem die Beschriftung der Schleife direkt in den Zustand Z eingetragen wird; bei einer Vielzahl derartiger Transitionen in einem Zustand kann so die Anzahl der orthogonalen Komponenten des Zustands verhältnismäßig klein gehalten werden.

Neben den vorgestellten Begriffen zum Aufbau von Statecharts gibt es eine Reihe weiterer Definitionen, die für ein leichteres Verständnis der Inhalte in Abschnitt 5.2 von Bedeutung sind. Für die Mehrheit von ihnen sind in Abb. 29 Beispiele angegeben, die die folgenden Ausführungen verdeutlichen.

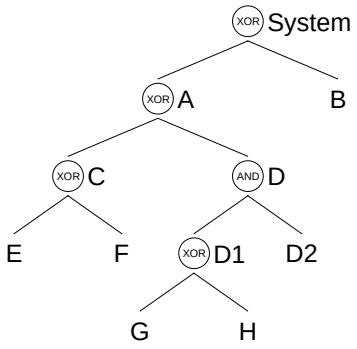
- *Lowest Common Ancestor:*
  Als Lowest Common Ancestor (LCA) einer Menge X von Zuständen wird jener Zustand bezeichnet, der auf der tiefstmöglichen Ebene des Zustandsbaumes liegt und alle Zustände aus X einschließt. Für die wichtigere Variante, den Strict Lowest Common Ancestor (LCA^+), wird zusätzlich gefordert, daß er den Typ *XOR* hat und selbst nicht in X enthalten ist. Auch einer Transition t kann, ausgehend von diesen beiden Definitionen, ein LCA zugeordnet werden; er entspricht dem

[5] Im STATEMATE-System sind zusätzlich „Ⓢ“- und „© “-Symbole sowie das Terminierungssymbol „Ⓣ“, das besagt, daß weitere Übergänge im Statechart nicht zulässig sind und die Ausführung beendet wird, erlaubt.

[6] siehe Punkt „Strukturell relevante Transitionen“ auf S. 44



(a)



(b)

LCA(G,H) = D1	LCA^+(E,F) = C	Basiskonf.: {D2,H}
LCA(D1,D2) = D	LCA^+(D2,H) = A	vervollst. Konf.: {D2,H,D1,D,A,System}
LCA(F,D2) = A	LCA(t1) = LCA^+(E,F) = C	strukt. konsistent: t1 und t4
LCA(B,E) = System	LCA(t2) = LCA^+(C,H) = A	strukt. relevant in {G,D2}: t3 und t4

(c)

Abb. 29: Ein Statechart in (a), der zugehörige Zustandsbaum in (b) sowie Beispiele zu den vorgestellten Begriffen in (c)

LCA^+ für die Vereinigung aus Quell- und Zielmenge von t.[7] $LCA(t)$ ermittelt somit den Zustand, in dem die Transition ausgeführt wird, und grenzt den durch t beeinflußten Teil des Statecharts vom unberührten Teil ab.

- *Orthogonalität:*
  Zwei Zustände werden als orthogonal bezeichnet, wenn sie entweder gleich sind oder ihr LCA vom Typ AND ist. In solchen Zuständen ablaufende Vorgänge können unabhängig voneinander ausgeführt werden. Gilt für eine Menge von Zuständen, daß die Zustände in ihr paarweise orthogonal sind, so wird die Menge selbst ebenfalls als orthogonal bezeichnet. Liegen alle Zustände einer orthogonalen Menge im gleichen Teil des Zustandsbaumes, so nennt man sie orthogonal relativ zu dem Zustand, der diesen Teilbaum aufspannt. Eine relativ zu einem Zustand orthogonale Menge heißt maximal, wenn das Hinzufügen eines beliebigen weiteren Zustands die Orthogonalität der Menge zerstören würde.

- *Zustandskonfiguration:*
  Für die Beschreibung der Menge von Zuständen, in denen ein System zu einem bestimmten Zeitpunkt verweilt, werden in [HPSS87] und [HN96] verschie-

[7] Ein in der Zielmenge enthaltenes History-Symbol wird durch den Zustand, in dem es liegt, ersetzt.

dene Bezeichnungen eingeführt. Die hier gegebene Definition von „Zustandskonfiguration“ berücksichtigt die einzelnen Varianten, indem eine weitere Aufgliederung des Begriffs vorgenommen wird. Es erfolgt eine Trennung zwischen partiellen und maximalen Zustandskonfigurationen sowie eine Unterscheidung anhand des Zustandstyps, der in den Zustandskonfigurationen zulässig ist. Eine *partielle Zustandskonfiguration* ist eine relativ zum Zustand *root* orthogonale Menge von Zuständen beliebiger Ebenen; ist diese Menge maximal, so bezeichnet man sie kurz als *Zustandskonfiguration*. Wenn eine (partielle) Zustandskonfiguration nur aus Basiszuständen besteht, wird sie (*partielle*) *Basiskonfiguration* genannt. Zusätzlich kann man den Begriff der *vervollständigten Konfiguration*[8] einführen; diese enthält neben den Zuständen der Basiskonfiguration auch die entsprechenden Zustände der höheren Ebenen.

- *Strukturell relevante Transitionen:*
  Eine Transition (Y, l, Z) wird als strukturell relevant für eine gegebene Basiskonfiguration X bezeichnet, wenn für jeden Zustand y der Quellmenge Y ein Zustand in X enthalten ist, der in dem von y aufgespannten Teil des Zustandsbaumes liegt. Dies ist gleichbedeutend mit der Forderung, daß Y eine Teilmenge der vervollständigten Konfiguration von X sein soll.

- *Strukturell konsistente Transitionen:*
  Zwei oder mehr Transitionen werden als strukturell konsistent bezeichnet, wenn ihre *LCA*s orthogonal sind. Sie beeinflussen unterschiedliche Teilbereiche des Statecharts und können daher zusammen ausgeführt werden, wenn sie gleichzeitig in einem Schritt aktiviert sind.

5.2 Semantik

Abschnitt 5.1 beschreibt einige wesentliche Komponenten der in [HPSS87] vorgestellten und im Rahmen von [HN96] ergänzten Syntax sowie zusätzliche Definitionen und Notationen für den grafischen Formalismus der Statecharts. Auf dieser Basis werden im folgenden die Möglichkeiten für die Entwicklung einer formalen Semantik erörtert; da die grafische Darstellung keine eindeutige Interpretation vorgibt, sind diesbezüglich verschiedene Ansätze denkbar. Die Vielzahl der möglichen Semantiken führt dazu, daß es im konkreten Anwendungsfall erforderlich ist, die zu verwendende Semantik festzulegen und diese Entscheidung geeignet zu dokumentieren, damit andere Personen die erstellten Statecharts leichter verstehen können und Zweideutigkeiten sowie Mißverständnissen vorgebeugt wird. Es hängt von der gewählten Semantik ab, welche Basiskonfigurationen bei einer gegebenen Menge externer Bedingungen und Ereignisse als Nachfolger der aktuellen Basiskonfiguration zulässig sind; ein System wird allgemein als nichtdeterministisch bezeichnet, wenn mehrere solcher Folgekonfigurationen existieren.

[8] in [HN96] als „Zustandskonfiguration“ bezeichnet

Neben den auf Seite 39 angesprochenen Schwierigkeiten gibt es eine Reihe weiterer Punkte, die für die Entwicklung einer konkreten Semantik von Bedeutung sein können. Hierzu zählen u. a. die Bestimmung der Priorität einzelner Transitionen, wenn gleichzeitig mehrere Zustandsübergänge aktiviert sind,[9] sowie die Frage, auf welche Weise gleichzeitig eintretende Ereignisse behandelt werden; die Nutzung gemeinsamer Variablen wäre ebenfalls ein denkbarer Diskussionspunkt. Diese kleine Auswahl an Fragestellungen verdeutlicht, daß es schon unter der Voraussetzung einer vorgegebenen Syntax vielfältige Möglichkeiten zur Festlegung einer Semantik gibt; durch Variationen in der Syntax läßt sich dieses Feld zusätzlich ausdehnen. In [vdB94] findet sich eine Übersicht über die bis 1994 erstellten Statechart-Varianten.[10]

Harel und seine Koautoren haben im Rahmen der Entwicklung des Formalismus zwei wesentliche Semantik-Ansätze, die sich auf die grundlegende Form der Statecharts ohne Erweiterungen beziehen, formuliert; die Unterschiede zwischen der ersten Variante, dem sogenannten Microstep-Ansatz, und der im Produkt STATEMATE umgesetzten Semantik liegen hauptsächlich in dem Zeitpunkt, zu dem die innerhalb eines Statecharts generierten Ereignisse registriert werden, sowie im Verständnis des Begriffes „Schritt“, der den Übergang zwischen zwei Basiskonfigurationen bezeichnet. Die Zeitdauer, die für einen Schritt benötigt wird, ist zusammen mit dem Zeitpunkt, zu dem die internen Ereignisse wahrgenommen werden, von besonderer Bedeutung, da hierdurch die Unterbrechbarkeit durch externe Ereignisse determiniert wird. Beide Ansätze werden hier vorgestellt, um zu verdeutlichen, daß die Wahl einer geeigneten Semantik von Bedeutung ist; für die Erläuterung soll eine ausführliche Darstellung der Prinzipien ohne Betrachtung der formalen Details genügen.

5.2.1 Das Konzept der Microsteps

Der Grundgedanke dieser Semantik, die von Harel und einigen Kollegen in [HPSS87] veröffentlicht und später in ähnlicher Weise auch von Pnueli und Shalev weiterverwendet wurde (siehe [PS91]), ist, daß die durch einen Zustandsübergang hervorgerufenen Veränderungen und die Ereignisse, die während der Transitionsausführung generiert werden, noch in demselben Schritt weitere Zustandsübergänge auslösen können. Für eine auf diese Weise gebildete Kette von Zustandsübergängen, einen sogenannten *Step*, die sich in eine endliche Anzahl von Teilschritten (*Microsteps*) zerlegen läßt, werden folgende Forderungen aufgestellt:

1. Die Ausführung der Kettenreaktion beansprucht keine Zeit, so daß die nächste Zustandskonfiguration im folgenden Schritt vorliegt und keine Unterbrechung durch extern generierte Ereignisse ermöglicht wird.[11]

2. Trotz der in 1. geforderten Zeitlosigkeit der Kettenreaktion soll die Reihenfolge der einzelnen Transitionen während der Ausführung Einfluß auf die nächste Zustandskonfiguration haben.

[9] Dies wird in Abschnitt 5.2.2 kurz betrachtet.
[10] Die in Abschnitt 5.2.2 vorgestellte Semantik ist in diesem Artikel noch nicht enthalten.
[11] Eine Begründung für eine solche Forderung geben Huizing und de Roever in [HdR91].

Zu berücksichtigen ist in diesem Zusammenhang, daß die hier eingeführten Microsteps lediglich ein theoretisches Konstrukt darstellen und nicht von außen wahrnehmbar sind. Abb. 30 veranschaulicht den Zusammenhang zwischen den einzelnen Microsteps, dem Step sowie dem Systemzustand und verdeutlicht, welche Aspekte des Steps von einem Beobachter registriert werden können.

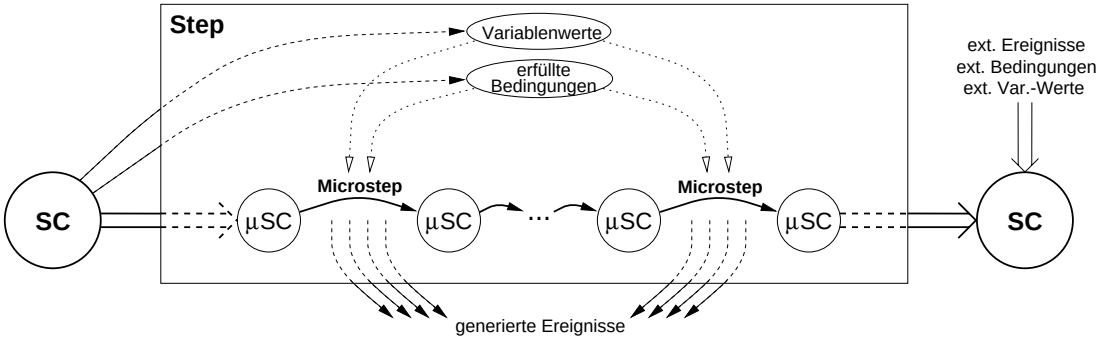


Abb. 30: Das Konzept der Microsteps

Wie in Abb. 30 dargestellt, geht man von der Systemkonfiguration SC zu einem betrachteten Zeitpunkt σ_i aus, um die möglichen Steps zu ermitteln; SC besteht aus

- der Basiskonfiguration X,
- einer Menge externer primitiver Ereignisse Π,
- einer Menge erfüllter externer primitiver Bedingungen Θ und
- einer den Variablen die aktuellen Werte zuweisenden Funktion ξ,

wobei das Tripel (Π, Θ, ξ) als *externer Stimulus* bezeichnet wird. Neben der Frage, wie die denkbaren Folgekonfigurationen bestimmt werden, sind die möglichen *Systemreaktionen*, die sich aus der im Step auszuführenden Menge von Transitionen und den während des Steps generierten Ereignissen zusammensetzen, von Interesse. Existieren für eine Systemkonfiguration zwei oder mehr verschiedene Reaktionen, so wird das System als nichtdeterministisch in dieser Konfiguration bezeichnet.

Zur Bestimmung der Systemreaktionen sowie der Folgekonfigurationen werden zunächst die *relevanten Transitionen* betrachtet, d. h. diejenigen Zustandsübergänge, die in der gegebenen Systemkonfiguration (X, Π, Θ, ξ) aktiviert sind. Hierzu zählen alle strukturell relevanten Transitionen, deren auslösendes Ereignis eintritt, wenn die Ereignisse aus Π vorliegen. Eine geeignete Auswahl aus dieser Menge von Transitionen bildet den ersten Microstep. Durch das Ausführen einzelner Microsteps werden nacheinander sogenannte *Micro System Configurations* (μSC) erreicht, die den momentanen, nach außen nicht sichtbaren Systemzustand beim Durchlaufen der Kettenreaktion beschreiben. Die erste Micro System Configuration wird aus der betrachteten Systemkonfiguration abgeleitet, und entsprechend wird aus der letzten durch die Kettenreaktion erreichten Micro System Configuration die nächste Systemkonfiguration ermittelt.

Ein einzelner Microstep $\mu\Upsilon$ besteht aus einer konsistenten Menge von Transitionen[12], die in der gegebenen Micro System Configuration aktiviert sind. Die durch den Microstep ausgelösten Aktionen werden zu der schon vorhandenen Ereignismenge hinzugefügt und die Werte von $cr(v)$, $cr(\gamma)$ und $ny(e)$ für alle Variablen v, Bedingungen γ und Ereignisse e aktualisiert.[13] Außerdem wird die durch die Transitionsübergänge des Microsteps erreichte partielle Zustandskonfiguration zu einer Basiskonfiguration vervollständigt, indem man die Default- und History-Eintritte wiederholt anwendet.

Aus einer Folge $(\mu\Upsilon_0, \mu\Upsilon_1, \ldots, \mu\Upsilon_m)$ von Microsteps ergibt sich ein Step Υ, wenn die Menge aller Transitionen, die in diesen Microsteps ausgeführt werden, strukturell konsistent ist und die Ausführung einer beliebigen weiteren aktivierten Transition in einem zusätzlichen Microstep diese Eigenschaft verletzen würde.[14] Unter diesem Gesichtspunkt kann ein Step als eine maximale Folge von Microsteps betrachtet werden. Die durch Υ erreichte Systemkonfiguration besteht aus der neuen Basiskonfiguration, den in dem rechtsoffenen Intervall zwischen σ_i und dem nächsten Zeitpunkt σ_{i+1} eingetretenen externen Ereignissen, den externen Bedingungen γ, die in einem Intervall $[\sigma, \sigma_{i+1})$ mit $\sigma \geq \sigma_i$ erfüllt sind, sowie einer Funktion, die jeder Variablen denjenigen Wert, den sie in einem Intervall $[\sigma, \sigma_{i+1})$ mit $\sigma \geq \sigma_i$ hatte, zuordnet. Der Step $\Upsilon = (\mu\Upsilon_0, \mu\Upsilon_1, \ldots, \mu\Upsilon_m)$ stellt zusammen mit den in diesem Step generierten Ereignissen die Systemreaktion dar.

Auf Seite 46 wurde bereits erwähnt, daß die hier beschriebenen Microsteps ein theoretisches Gebilde darstellen und ebenso wie die Micro System Configurations nicht von außen wahrnehmbar sind. Ein externer Beobachter kann von dem beschriebenen Vorgang nur

- die ursprüngliche und die neu entstandene Zustandskonfiguration,
- die während des Steps vom Statechart generierten Ereignisse sowie
- die neuen Werte für die primitiven Bedingungen und Variablen

registrieren. Außerdem ergibt sich durch die sofortige Ausführung der Steps, die keinerlei Zeit beansprucht, daß externe Ereignisse, die während des Steps eintreten, erst im nachfolgenden Step Auswirkung haben und den aktuellen Step nicht beeinflussen. Durch die Transitionen der einzelnen Microsteps hervorgerufene Wertänderungen von Bedingungen, Variablen und Ausdrücken können zudem nur dann Einfluß auf die möglichen Zustandsübergänge im aktuellen Step haben, wenn im Statechart die Ausdrücke $cr(v)$ sowie die Bedingungen $cr(\gamma)$ und $ny(e)$ verwendet werden. Die mit den eigentlichen Bedingungen, Variablen und Ausdrücken verbundenen Werte bleiben innerhalb eines Steps gleich; erst mit Erstellung der neuen Systemkonfiguration werden die Werte von $cr(v)$ und $cr(\gamma)$ auf v und γ übertragen.

[12] Eine Menge von Transitionen wird als *konsistent* bezeichnet, wenn die in ihr enthaltenen Transitionen strukturell konsistent sind (s. S. 44) und die durch sie ausgelösten Aktionen ebenfalls konsistent sind, d. h. daß es keine mehrfachen Wertzuweisungen an primitive Variablen oder Bedingungen gibt.

[13] $ny(e)$ ist die Abkürzung für die Bedingung *not_yet*(e); $cr(v)$ steht für den Ausdruck *current*(v).

[14] Die durch den Step ausgelöste Menge von Aktionen braucht nicht konsistent zu sein.

5.2.2 In STATEMATE realisierter Ansatz

Harel und Naamad stellen in ihrem Artikel [HN96] einen zweiten, nach ihren Angaben stärker als das Konzept der Microsteps an der Realität orientierten Ansatz vor, der auch in dem von der Firma i-Logix vertriebenen Produkt STATEMATE umgesetzt wurde. Er basiert auf der grundsätzlichen Überlegung, daß die durch einen Zustandsübergang hervorgerufenen Veränderungen – z. B. generierte Ereignisse, Wertänderungen von Variablen sowie die Menge der momentan eingenommenen Zustände – erst im nächsten Schritt Auswirkung haben sollen und somit keine weiteren Zustandsübergänge im aktuellen Schritt auslösen können.

Abb. 31 veranschaulicht das diesem Ansatz zugrundeliegende Verständnis des Begriffes *Step*: Während jedes Steps eines sogenannten *Laufs*[15] wird durch die Ausführung

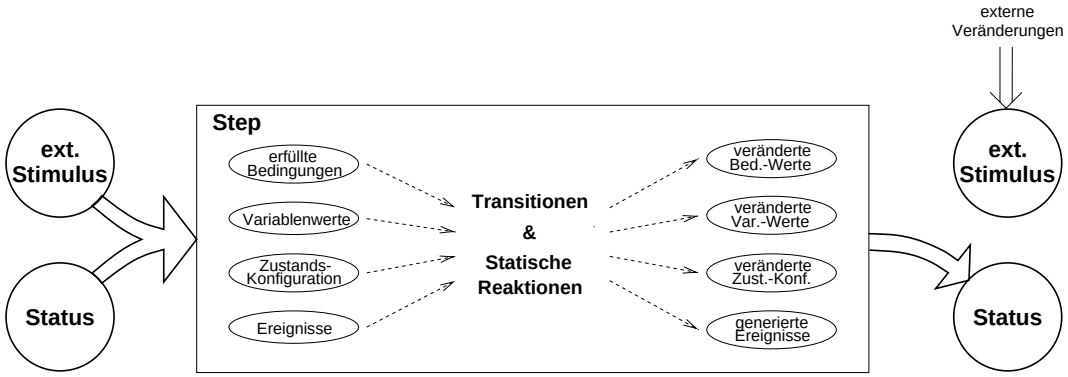


Abb. 31: Aufbau eines Steps bei der in STATEMATE verwendeten Semantik

von Transitionen und statischen Reaktionen der aktuelle Status des Systems in einen neuen Status überführt. Auslöser hierfür sind der externe Stimulus[16] und der aktuelle Status des Systems, der sich aus folgenden Bestandteilen zusammensetzt:

- der Menge aller Zustände, in denen das System z. Zt. verweilt (vervollständigte Konfiguration),
- der Menge laufender Aktivitäten,
- den aktuellen Werten der Bedingungen und Variablen,
- der Menge im letzten Step intern generierter Ereignisse,
- den aufgeschobenen Aktionen und ihren Ausführungszeitpunkten,
- den Timeout-Ereignissen und deren Zeitpunkten sowie

[15] Ein Lauf repräsentiert die Systemantworten auf eine Sequenz externer Stimuli.

[16] In [HN96] erfolgt keine Definition des Begriffs „externer Stimulus“; es ist jedoch anzunehmen, daß von der Beschreibung auf Seite 46 ausgegangen werden kann.

- relevanten Informationen über die Systemgeschichte.

Neben dem schon im Konzept der Microsteps formulierten Umstand, daß für die Ausführung eines Steps keine Zeit benötigt wird – die Zeit zwischen einzelnen Steps ist nicht in der Step-Semantik enthalten –, werden hier weitere Eigenschaften vorausgesetzt:

1. Reaktionen auf externe und interne Ereignisse, z. B. Zustandswechsel, sowie in einem Step auftretende Veränderungen, beispielsweise Wertänderungen von Variablen, können erst nach Beendigung des Steps registriert werden.

2. Tritt ein Ereignis ein, so existiert es nur bis zum Ende des nach seinem Eintreten beginnenden Steps. In allen späteren Steps kann das Ereignis nicht registriert werden; intern generierte Ereignisse können auch im aktuellen Step nicht wahrgenommen werden.

3. Die Berechnungen innerhalb eines Steps basieren immer auf der Situation am Anfang des Steps; dies ist von Interesse, wenn sich Werte von Bedingungen und Variablen sowie die Zustände, in denen das System verweilt, und die laufenden Aktivitäten während des Steps ändern.

4. Es wird immer eine maximale Menge konfliktfreier Transitionen und statischer Reaktionen in einem Step ausgeführt.

Die im letzten Punkt erwähnte Konfliktfreiheit der Transitionen und statischen Reaktionen bezieht sich hierbei auf die zu verlassenden Zustände und nicht auf die Wertzuweisung an Variablen.[17]

[HN96] beinhaltet eine Beschreibung für die Bearbeitung eines Steps, deren einzelne Schritte hier kurz wiedergegeben werden:

Eingabe: Systemstatus, aktuelle Zeit, externe Veränderungen (externer Stimulus)

Ausgabe: neuer Systemstatus

1. Vorbereitung:

   - die internen und externen Ereignisse zusammenfügen
   - die mit den externen Veränderungen verbundenen Aktionen ausführen
   - aufgeschobene Aktionen mit überschrittenem Ausführungszeitpunkt ausführen
   - Timeout-Ereignisse überprüfen und gegebenenfalls generieren

2. Inhalt des Steps:

   - die aktivierten Transitionen herausarbeiten

[17] In [HN96] werden die durch mehrfache Wertzuweisung entstehenden Probleme unter dem Punkt „Racing Conditions“ behandelt.

- alle Transitionen, die mit Transitionen höherer Priorität[18] in Konflikt stehen, aus der Menge der aktivierten Transitionen entfernen
- die maximalen Mengen konfliktfreier Transitionen bestimmen
- die nicht mit den Transitionen dieser Mengen in Konflikt stehenden aktivierten statischen Reaktionen ermitteln

3. Ausführung der Transitionen und statischen Reaktionen:
   - eine der in 2. ermittelten Möglichkeiten für einen Step auswählen
   - für jede statische Reaktion die zugehörige Aktion ausführen
   - für jede Transition t mit der zugehörigen Menge der zu verlassenden (betretenden) Zustände S_x (S_{en}) :
     - Systemgeschichte für Elternzustände der Zustände aus S_x aktualisieren
     - Zustände aus S_x aus der Menge der aktuellen Zustände entfernen
     - die durch t festgelegten Aktionen ausführen
     - zum Betreten der in S_{en} liegenden Zustände gehörende Aktionen ausführen
     - Zustände aus S_{en} zur Liste der aktuellen Zustände hinzufügen

Das in dieser Semantik verwendete Prinzip, nach dem die durch Transitionen und statische Reaktionen hervorgerufenen Veränderungen erst im nachfolgenden Schritt registriert werden können, bedingt u. a., daß es – im Gegensatz zum Konzept der Microsteps – nicht möglich ist, in einem Step einen Zustand sowohl zu betreten als auch wieder zu verlassen, da das Betreten des Zustands erst im nächsten Step bekannt ist. Der umgekehrte Fall, daß ein Zustand erst verlassen und dann sofort wieder betreten wird, ist jedoch weiterhin durch den Einsatz von Schleifen möglich. Abb. 30 und Abb. 31 veranschaulichen die Unterschiede der beiden vorgestellten Semantikansätze.

Über diese grundsätzlichen Überlegungen zum Umfang und Aufbau eines Steps hinaus enthält [HN96] u. a. Betrachtungen zum Zusammenspiel von History-Eintritt und Default-Zuständen sowie zur Festlegung von Prioritäten bei konfliktbehafteten Transitionen. Für einen Zustandsübergang, der an einem „Ⓗ“- oder „Ⓗ*“-Symbol endet, schlagen Harel und Naamad folgende Reihenfolge zur Ermittlung des zu betretenden Subzustandes vor:

1. Wenn sich das System schon einmal in dem Zielzustand der Transition, der das „Ⓗ“- oder „Ⓗ*“-Symbol enthält, befunden hat und die Systemgeschichte dieses Zustands nicht gelöscht wurde, wird der zu betretende Subzustand anhand der Systemgeschichte ermittelt.

2. Andernfalls ist für die vom „Ⓗ“- oder „Ⓗ*“-Symbol ausgehenden Transitionen zu prüfen, ob eine von ihnen ausgewählt werden kann, um den Folgezustand zu bestimmen.

[18] siehe Seite 51

3. Kann keine direkt vom History-Symbol wegführende Transition gewählt werden, so wird der Default-Zustand innerhalb des betretenen Zustands als Folgezustand festgelegt.

Verwendet man eine umfangreichere Konstruktion wie beispielsweise in Abb. 32, ist es sinnvoll sicherzustellen, daß keine Schleifen entstehen können und immer ein zu betretender Subzustand ermittelt werden kann. In dem hier verwendeten Beispiel ist nicht

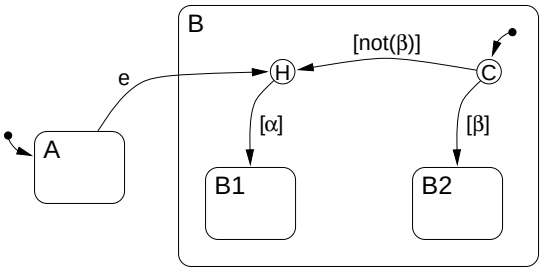


Abb. 32: Zusammenspiel von History-Eintritt und Default-Zustand

sichergestellt, daß bei einem Übergang vom Zustand A zum Zustand B durch das Ereignis e immer ein geeigneter Subzustand gefunden werden kann: Sind die Bedingungen α und β beide nicht erfüllt, so erhält man einen ständigen Wechsel zwischen „Ⓗ“ und „Ⓒ“.

Neben der Prioritätsfestlegung für History-Eintritt und Default-Zustand ist es notwendig, die vorrangige Transition in einer Menge konfliktbehafteter Transitionen zu bestimmen. Folgende Regelungen kommen nach [HN96] in Betracht: Zum einen kann man die Priorität der Transitionen anhand ihrer *strukturellen Prioritätsdeterminanten* bestimmen, wobei die Transition mit der höheren strukturellen Prioritätsdeterminanten vorrangig ist. In der in [HN96] beschriebenen Version von STATEMATE wird der LCA der einzelnen Transitionen für diese Bestimmung herangezogen. In späteren Versionen soll jedoch, so Harel und Naamad, im wesentlichen die Menge der Quellzustände einer Transition für die Festlegung verwenden werden. Sind die strukturellen Prioritätsdeterminanten zweier Transitionen gleich, kann es zu Nichtdeterminismus kommen, da es hierfür keine gesonderte Regelung gibt. Eine zweite Möglichkeit, um den Vorrang einzelner Transitionen festzulegen, besteht darin, *explizite Prioritätsnummern* zu vergeben. Es gilt entsprechend, daß die Transition mit der höheren Prioritätsnummer Vorrang vor den anderen hat und es bei gleicher Priorität zu Nichtdeterminismus kommt. Darüber hinaus haben von einem Zustand ausgehende Transitionen immer eine höhere Priorität als jede statische Reaktion des Zustands; somit werden statische Reaktionen nur solange ausgeführt, wie das System in dem entsprechenden Zustand verweilt und diesen nicht verläßt.

6 Statecharts im Rahmen von OMT

Mit diesem Abschnitt beginnt die Betrachtung von Statecharts vor dem Hintergrund der objektorientierten Anwendungsentwicklung. Die bisherigen Überlegungen sind für den Fall eines objektorientierten Vorgehens allerdings nicht von untergeordneter Bedeutung; sie stellen vielmehr eine Basis dar, die zunächst einmal als unabhängig von dem Paradigma, das der zu entwickelnden oder bereits entwickelten Software zugrunde liegt, verstanden werden sollte. Im konkreten Fall sollten Statecharts jedoch stets als ein Bestandteil des gewählten Ansatzes aufgefaßt werden, der mit anderen Techniken kombiniert werden muß – zumindest dann, wenn es um die Entwicklung komplexer Systeme geht.

Die erste bekannte derartige Integration des Statechart-Formalismus erfolgte auf der Grundlage des Paradigmas der Strukturierten Analyse (SA) und mündete schließlich im kommerziellen Produkt STATEMATE der Firma i-Logix. STATEMATE kombiniert Statecharts mit sogenannten *Activity-Charts*, die die funktionale Dekomposition eines Systems widerspiegeln und Datenflüsse zwischen den einzelnen Funktionen beschreiben, sowie mit *Module-Charts*, die verdeutlichen, wie das System in Module zerlegt werden kann.[1]

Die fortschreitende Verdrängung von SA-Ansätzen durch objektorientierte Vorgehensweisen brachte im Laufe der Zeit neue Anwendungsmöglichkeiten für Statecharts mit sich.[2] Im vorliegenden Abschnitt soll aufgezeigt werden, welche Rolle Statecharts bei einem objektorientierten Vorgehen, das sich an der in [RBP+91] beschriebenen *Object Modeling Technique* (*OMT*) orientiert, spielen können.

Der erste Teilabschnitt gibt einen kurzen Überblick über OMT, damit erkennbar wird, wie sich der Statechart-Formalismus in die Methode einfügt. Im zweiten Teilabschnitt wird das dynamische Modell, zu dem die Statecharts zusammengefaßt werden, erläutert. Dort wird zunächst gezeigt, wie sich Statecharts und objektorientierte Konzepte miteinander verbinden lassen. Anschließend erfolgt eine Darstellung typischer syntaktischer Konstrukte von OMT-Statecharts. Da sich Rumbaugh, der als wesentlicher geistiger Vater von OMT gilt, und seine Koautoren an [Har87] orientieren, werden grundlegende Aspekte wie z. B. Zustandshierarchien oder Orthogonalität nicht wiederholt, sondern vielmehr nahezu ausschließlich spezielle Notationen und Anwendungsmöglichkeiten vorgestellt. Einige Problembetrachtungen bilden den Abschluß des Teilabschnitts. Im dritten Teil wird in knapper Form auf den Entwurfsprozeß eingegangen, wobei das dynamische Modell im Vordergrund steht, da ihm eine besondere Bedeutung zukommt, wenn man reaktive objektorientierte Anwendungssoftware erstellen möchte. Vieles muß bei dieser Darstellung zwangsläufig unberücksichtigt bleiben und im Bedarfsfall [RBP+91] entnommen werden.

---

[1] [HLN+90] beschreibt das STATEMATE-Werkzeug im Überblick; in [Har92] findet man eine Darstellung der damals zugrundeliegenden Motivation.

[2] Harel beschreibt diese Entwicklung aus seiner Sicht in knapper Form in [Har97].

6.1 OMT im Überblick

OMT ist eine objektorientierte Entwicklungsmethode, die inzwischen weit verbreitet ist und vielfach in der Praxis eingesetzt wird. Diese Methode anzuwenden bedeutet, mit Hilfe von grafischen Darstellungen und Textdokumenten ein Modell im Hinblick auf eine Anwendungsdomäne zu konstruieren, das schließlich implementiert wird.

Obwohl die folgende Darstellung an den Charakter von Phasenmodellen erinnert, ist OMT nicht an eine entsprechende Vorgehensweise gebunden. Rumbaugh und seine Koautoren betonen, daß die Techniken, von denen OMT Gebrauch macht, ebensogut für die Softwareentwicklung nach einem zyklischen Modell, die auf dem Konzept des Prototyping basiert, geeignet sind; es kommt ihnen nicht darauf an, diesbezüglich eine Wertung vorzunehmen. Die Punkte *Analyse*, *Systemdesign*, *Objektdesign* und *Implementation* sind nach Rumbaugh et al. in der Praxis nicht derart klar gegeneinander abzugrenzen, wie dies im folgenden vielleicht zunächst erscheinen mag. Der gesamte Prozeß ist iterativ, und die Kommunikation zwischen Auftraggebern, Benutzern und Personen, die für die Entwicklung verantwortlich sind, ist keineswegs auf die Analyse beschränkt. Die vier genannten Punkte können folgendermaßen kurz charakterisiert werden:

- *Analyse:*
  Ausgehend von einer Problembeschreibung, die häufig erst einmal gemeinsam von dem Auftraggeber und dem Auftragnehmer erarbeitet werden muß, konstruiert ein *Analytiker*[3] in Zusammenarbeit mit dem Auftraggeber oder den späteren Anwendern ein Modell der Anwendungsdomäne. Dieses Modell soll in konsistenter und möglichst präziser Form beschreiben, *was* das spätere System leisten soll, und darf keine Aspekte enthalten, die implementationsabhängig sind.

- *Systemdesign:*
  Der *System-Designer* legt die Grundzüge der späteren Systemarchitektur fest. Er trifft allgemeine Entscheidungen für die Lösung des Problems und beschreibt Subsysteme und Kriterien, die das System erfüllen soll, wie Performanzaspekte oder Art und Umfang der benötigten Betriebsmittel.

- *Objektdesign:*
  Der *Objekt-Designer* erstellt ausgehend von der Analyse ein erweitertes Modell, das Implementationsdetails berücksichtigt, die auf die im Rahmen des Systemdesigns getroffenen Entscheidungen zurückgehen. Er fügt den Konzepten der Anwendungsdomäne weitere hinzu, die sich auf die Realisierung des Modells mit Hilfe eines Rechners beziehen und keine Entsprechung in der realen Welt haben, und bestimmt Datenstrukturen und Algorithmen, die verwendet werden sollen, um die einzelnen Klassen zu implementieren.

[3] Bezeichnungen wie *Analytiker*, *System-Designer* usw. sind hier als funktionelle Rollen zu verstehen.

- *Implementation:*
  Die Klassen und die Beziehungen zwischen ihnen (z.B. Assoziationen), die sich aus dem Objektdesign ergeben, werden implementiert. Dies kann nicht nur mit Hilfe einer objektorientierten Programmiersprache erfolgen; es können andere Programmiersprachen oder Datenbanken verwendet werden, und auch Hardware-Implementationen sind denkbar.[4]

In jedem dieser Abschnitte werden drei Arten von Modellen eingesetzt, die gemeinsam das System beschreiben: das *Objektmodell*, das *dynamische Modell* und das *funktionale Modell*. Sie stellen drei verschiedene Sichten auf das System dar und beziehen sich aufeinander; eine vollständige Systembeschreibung besteht aus allen drei Modellen. Auch diese drei Modellarten sollen hier kurz vorgestellt werden:

- *Objektmodell:*
  Das Objektmodell beschreibt die statische Struktur eines Systems auf der Grundlage von Klassen und Objekten und kann als das wichtigste der drei Modelle angesehen werden. Mit Hilfe von Klassen- und Exemplardiagrammen[5] werden neben den Klassen und Objekten sowie ihren Operationen und Attributen zahlreiche weitere Informationen festgehalten; hierzu zählen vor allem

  - Vererbungsbeziehungen,
  - Angaben über abstrakte und konkrete Klassen,
  - Assoziationen zwischen Klassen und sogenannte *Links* als konkrete Ausprägungen dieser Assoziationen,
  - Aggregationen als Spezialfälle von Assoziationen,
  - Angaben darüber, wieviele Exemplare einer Klasse mit wievielen Exemplaren einer potentiell anderen Klasse in Beziehung stehen (*multiplicity*),
  - ergänzende Informationen über Assoziationen und Links (z.B. *Link-Attribute*, die Eigenschaften von Links beschreiben, und *Rollenbezeichner*, die die partizipierenden Exemplare näher charakterisieren) sowie
  - Angaben über die Verteilung der Klassen auf verschiedene Einheiten (*Module*).

- *Dynamisches Modell:*
  Während sich das Objektmodell auf die statische Struktur des Systems bezieht,

[4] Rumbaugh et al. betonen, daß ein objektorientiertes Vorgehen aufgrund seiner ausgeprägten Orientierung an den realen Konzepten der Anwendungsdomäne auch dann Vorteile gegenüber anderen Ansätzen bietet, wenn keine objektorientierte Programmiersprache zur Verfügung steht; in [RBP+91] beschreiben sie, wie man prozedurale Programmiersprachen und relationale Datenbanken einsetzen kann, um objektorientierte Modelle mit Einschränkungen implementieren zu können.

[5] Rumbaugh et al. sprechen von *instance diagrams*. Wir übersetzen *instance* mit *Exemplar* und nicht mit *Instanz*; eine Instanz ist eine „zuständige Stelle bei Behörden od. Gerichten“ oder ein „Dienstweg“ (siehe Duden „Rechtschreibung der deutschen Sprache und der Fremdwörter“, 18. Auflage).

steht im dynamischen Modell das Verhalten des Systems über die Zeit, das sich darin äußert, wie sich die Anzahl der Objekte, ihre Zustände und die Beziehungen zwischen ihnen verändern, im Vordergrund. Das dynamische Modell beschreibt mit Hilfe von Statecharts die möglichen Kontrollflüsse und Interaktionen im System und somit alle denkbaren Sequenzen von Operationsaufrufen und ist der zentrale Betrachtungsgegenstand des vorliegenden Abschnitts.

- *Funktionales Modell:*
  Das funktionale Modell beschreibt die notwendigen Berechnungen und Transformationen von Daten, ohne festzulegen, wie, wann und warum die entsprechenden Werte ermittelt werden. Mit Hilfe von Datenflußdiagrammen werden

  – die Prozesse, die die hierzu erforderlichen Funktionen darstellen,

  – die Daten, die zwischen diesen ausgetauscht werden,

  – unabhängige Objekte (sogenannte *Akteure* oder *Terminatoren*), die im weitesten Sinne Informationen in Form von Werten an das System übermitteln und/oder durch das System mit derartigen Informationen versorgt werden, und

  – benötigte Speicher, die als passive Objekte fungieren und den Datenfluß zwischen Prozessen verzögern,

  erfaßt.

  Die Datenflußdiagramme haben i.a. einen hierarchischen Charakter, denn Prozesse können ihrerseits zu weiteren Datenflußdiagrammen verfeinert werden. Prozesse einer bestimmten Granularität, die nicht näher untergliedert werden sollen, werden schließlich als Operationen aufgefaßt, die Klassen zugeordnet werden.

Die drei beschriebenen Modelle werden während der *Analyse* erstellt und in den nachfolgenden Schritten stets überarbeitet, so daß über den gesamten Entwicklungsprozeß hinweg eine einheitliche Notation verwendet wird. In Abhängigkeit vom konkreten Anwendungskontext sind diese Modelle von unterschiedlicher Bedeutung innerhalb des Gesamtsystems: Während z.B. für Compiler, die verhältnismäßig komplexe Transformationen leisten, das funktionale Modell eine wichtige Rolle spielt, zeichnen sich hochgradig interaktive Systeme oder Realzeitsysteme durch detaillierte dynamische Modelle aus.

Obwohl jedes Modell einen anderen Aspekt des Systems beschreibt, gibt es Verknüpfungen zwischen ihnen: Das Objektmodell beschreibt eine Struktur, auf der das dynamische und das funktionale Modell operieren. Die in den Statecharts des dynamischen Modells aufgeführten Ereignisse schlagen sich ebenso wie die nicht weiter verfeinerten Prozesse des funktionalen Modells in Operationen des Objektmodells nieder. Das dynamische Modell beschreibt, wie in Abhängigkeit von der Belegung der Exemplarvariablen Entscheidungen getroffen werden, die zu Veränderungen an Objekten führen können, indem Operationen aufgerufen werden.

Rumbaugh und seine Koautoren betonen, daß eine klare Trennung zwischen den Modellen aufgrund dieser Beziehungen nicht immer möglich ist. Informationen können häufig nicht eindeutig einem Modell zugeordnet werden, und vieles läßt sich in natürlicher Sprache am besten ausdrücken.

6.2 Das dynamische Modell

6.2.1 Grundlegende Konzepte

Beschreibt man das Verhalten eines Systems, so kann dies immer nur auf der Grundlage einer bereits bestehenden statischen Struktur geschehen, weil zunächst einmal bekannt sein muß, *was* sich im Laufe der Zeit verändern soll, bevor verstanden werden kann, *wie* es sich ändern soll. Daher ist es vernünftig, anfangs ein Objektmodell zu erstellen, das die Strukturen der Objekte und die Beziehungen zwischen ihnen, die zu einzelnen Zeitpunkten vorliegen können, wiedergibt. Anschließend kann untersucht werden, wie sich Objekte und Beziehungen über die Zeit verändern können.

Da man sich bei einem objektorientierten Ansatz verstärkt an den Gegenständen der Anwendungsdomäne orientiert, ist es sinnvoll, sich alle Objekte zunächst als nebenläufig aktiv vorzustellen. Die Objekte reagieren auf Stimuli, indem sie Sequenzen von Operationen initiieren. Das dynamische Modell beschreibt auf der Grundlage von *Ereignissen* und *Zuständen*, wie dieser Prozeß in zeitlicher Hinsicht kontrolliert wird, ohne festzulegen, was die Operationen im einzelnen bewirken oder worauf sie operieren.

Ereignisse können externe Stimuli repräsentieren, die außerhalb des Systems generiert werden und Objekte zu Reaktionen veranlassen, oder können dafür stehen, daß ein Objekt ein anderes oder sich selbst auf bestimmte Weise anregt. Sie werden als augenblicklich betrachtet und daher stets mit einem Zeitpunkt verknüpft. Rumbaugh et al. führen Ereignisklassen ein, die hierarchisch angeordnet werden können. In Anlehnung an Klassenhierarchien werden konkrete Ereignisse als Exemplare von Ereignisklassen aufgefaßt. Es ist ein Modellierungskonzept, daß Hierarchien von Ereignisklassen gebildet werden können. An eine direkte Umsetzung mit Hilfe einer Programmiersprache ist hierbei nicht gedacht; es sollen vielmehr verschiedene Abstraktionsebenen ermöglicht werden. Im Hierarchiebaum[6] dürfen nur die Ereignisklassen an den Blättern konkrete Klassen sein.

Während einige Ereignisse nur eine einfache Signalfunktion haben, beinhalten die meisten Informationen in Form von Belegungen ihrer *Attribute*, die von einem Objekt an ein potentiell anderes übermittelt werden können. Beispielsweise können *Adressat*, *Absender* und *Text* Attribute eines Ereignisses sein, das anzeigt, daß eine E-Mail eintrifft. Attribute werden gemäß der Ereignishierarchie *vererbt*, und die Zeit wird als implizites Attribut aller Ereignisse betrachtet. Im Sinne einer Subtyp-Beziehung kann ein Ereignis an einem Objekt alle Reaktionen auslösen, die Ereignisse hervorrufen können,

[6] Rumbaugh et al. erwähnen nicht, ob sie in dem Graph, der die Hierarchie der Ereignisklassen wiedergibt, auch Ereignisse mit mehr als einem unmittelbaren Vorfahren zulassen; das in [RBP+91] enthaltene Beispiel besitzt eine Baumstruktur.

von denen es direkt oder indirekt erbt.

Zustände sind Abstraktionen der möglichen Belegungen der Exemplarvariablen eines Objekts sowie seiner Links, d. h. der Beziehungen, die zwischen ihm und anderen Objekten bestehen. Wir gehen im folgenden davon aus, daß Links als Zeiger implementiert werden und somit ebenfalls als Belegungen von Exemplarvariablen aufgefaßt werden können.[7] In fast allen Fällen werden zahlreiche Belegungen in einem einzigen Zustand zusammengefaßt; somit steht ein Zustand für eine Menge von denkbaren Wertekombinationen. In der Regel erfolgt hierbei eine Orientierung an qualitativen Gesichtspunkten: Die Reaktion eines Objekts auf ein Ereignis hängt in quantitativer Hinsicht von den exakten Werten ab, die in seinen Exemplarvariablen gespeichert sind, jedoch ist sie qualitativ gleich für alle Wertekombinationen, die durch einen gemeinsamen Zustand repräsentiert werden. Im allgemeinen bedeuten verschiedene Zustände hingegen auch qualitativ unterschiedliche Reaktionen. Exemplarvariablen, die das Verhalten eines Objekts nicht in qualitativer, sondern nur in quantitativer Hinsicht beeinflussen, tragen nicht zu dem Prozeß, in dessen Verlauf Zustände definiert werden, bei; sie können in bezug auf einen gegebenen Zustand als Parameter aufgefaßt werden.

Zustände zu definieren ist eine Aufgabe im Rahmen der Modellbildung, deren Bearbeitung durch Gesichtspunkte bestimmt sein sollte, die im Hinblick auf den Anwendungskontext als wesentlich angesehen werden. Hierbei ist es vor allem wichtig, *abgeleitete Exemplarvariablen*, deren Werte zu jedem Zeitpunkt aus den Belegungen der *Basis-Exemplarvariablen* ermittelt werden können und somit redundante Informationen darstellen, zu erkennen oder festzulegen: Nur die Basis-Exemplarvariablen sollten den Zustand eines Objekts bestimmen.[8]

Innerhalb des Lebenszyklus eines Objekts stellen Zustände Zeitintervalle dar, die zwischen zwei Ereignissen liegen; analog markiert ein Ereignis die Trennung zweier derartiger Zeitintervalle. Empfängt ein Objekt ein Ereignis, so kann es einen neuen Zustand einnehmen und/oder eine Aktion auslösen, die ihrerseits weitere Objekte zu Reaktionen veranlaßt.[9] Somit können sich Objekte ständig gegenseitig beeinflussen und fortwährend ihre Zustände ändern.

Das Verhalten einer Menge nebenläufiger Objekte wird im dynamischen Modell beschrieben, welches aus mehreren nebenläufigen Statecharts, die über gemeinsame Ereignisse miteinander verknüpft sind, besteht. Jeder Statechart beschreibt eine andere Klasse. Rumbaugh et al. halten es nicht für erforderlich, für jede Klasse einen Statechart

[7] In den gängigen objektorientierten Programmiersprachen werden Links mit Hilfe eines Zeigerkonzeptes implementiert; unter diesen Umständen kann ein Zustand einfach als Abstraktion einer Menge von Exemplarvariablenbelegungen aufgefaßt werden. Rumbaugh et al. halten die ursprüngliche Differenzierung jedoch aufrecht, da konzeptionelle Unterschiede zwischen einer Exemplarvariablen, die zu einem Objekt gehört, und einer Beziehung zwischen zwei Objekten, für die es häufig keine eindeutige Zuordnung zu einem Objekt gibt, bestehen und grundsätzlich auch andere Möglichkeiten, Assoziationen zu implementieren, vorstellbar sind.

[8] Ein typisches Beispiel ist eine Exemplarvariable *Alter* eines Personen-Objekts, deren Wert aus der Belegung der Exemplarvariablen *Geburtsdatum* und dem global verfügbaren aktuellen Datum abgeleitet werden kann.

[9] Erst gegen Ende des Softwareentwicklungsprozesses werden Ereignisse und Aktionen auf konkrete Methoden abgebildet. Rumbaugh et al. behalten daher die anfänglich eingeführten Begriffe bei.

zu erstellen. Ihrer Ansicht nach sollte ein Statechart nur dann entworfen werden, wenn das Verhalten einer Klasse verhältnismäßig komplex ist.

Alle Objekte einer Klasse besitzen die gleichen Exemplarvariablen und die gleichen Operationen, aber sie können sich in den Belegungen ihrer Exemplarvariablen unterscheiden. Entsprechend zeigen alle Objekte einer Klasse das gleiche Verhaltensmuster, das durch einen Statechart beschrieben wird, können sich jedoch zu einem Zeitpunkt in verschiedenen Zuständen befinden und somit auf ein bestimmtes Ereignis unterschiedlich reagieren.

6.2.2 Die Statecharts des dynamischen Modells

Rumbaugh und seine Koautoren fordern, daß alle Statecharts des dynamischen Modells deterministisch sein sollten. Sie unterscheiden zwischen Statecharts, die das Verhalten bis ins Unendliche beschreiben, und sogenannten „One-shot"-Statecharts, die Anfangs- und Endzustände besitzen. Während „One-shot"-Statecharts es erlauben, die Erzeugung und Zerstörung von Objekten zu berücksichtigen, werden die Statecharts der erstgenannten Art verwendet, wenn von der durch sie beschriebenen Kontrollschleife (noch) nicht bekannt ist, unter welchen Umständen und in welcher Form sie beginnt oder endet. „One-shot"-Statecharts können als Subroutinen aufgefaßt werden, auf die sich Statecharts einer höheren Abstraktionsebene beziehen können. So können z. B. *Aktivitäten* und *Ereignisse* durch „One-shot"-Statecharts verfeinert werden (s. u.). Anfangszustände von „One-shot"-Statecharts werden durch einen gefüllten Kreis gekennzeichnet, der mit mehreren *Anfangsbedingungen* beschriftet werden kann, die z. B. initiale Werte anzeigen. Endzustände werden zusätzlich von einem Kreis umschlossen und können analog mit *Endbedingungen* versehen werden. Abb. 33 zeigt einen aus [RBP+91] stammenden „One-shot"-Statechart für ein Schachspiel.

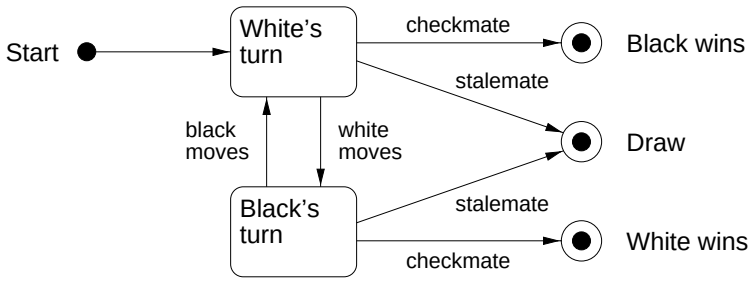


Abb. 33: Ein Schachspiel als „One-shot"-Statechart

Wie die in [Har87] beschriebenen Statecharts können die Statecharts des dynamischen Modells *Bedingungen* enthalten, die als Wächter für Transitionen fungieren. Sie werden als Boolesche Ausdrücke formuliert, die, in eckige Klammern eingeschlossen, den Ereignisbezeichnern folgen. Die Statecharts des dynamischen Modells können sich ebenfalls auf *Aktionen* und *Aktivitäten* beziehen.

Wie Harel betrachten Rumbaugh et al. Aktionen als zeitlos und stets mit einem Ereignis verbunden. Die Notation für Aktionen entspricht im wesentlichen der in Ab-

schnitt 3.4 beschriebenen: Aktionen folgen den Ereignis- und Bedingungsbezeichnern, von denen sie durch einen Schrägstrich getrennt werden. Aktionen, die ausgeführt werden, wenn ein Zustand betreten oder verlassen wird, werden dessen Beschriftung gemäß Abb. 34(a) hinzugefügt. Wird ein Zustand eingenommen, so wird zuerst die Aktion der eingehenden Transition und danach die mit dem Zustand verknüpfte Eingangs-Aktion ausgeführt. Wenn ein Zustand hingegen verlassen wird, findet zunächst die Ausführung der Ausgangs-Aktion statt, bevor die an die ausgehende Transition gebundene Aktion abgearbeitet wird. Zustände können mehrere Ein- und Ausgangs-Aktionen aufweisen. Ist mit einem Zustand zusätzlich eine Aktivität (s. u.) verbunden, so wird diese nach den Eingangs- und vor den Ausgangs-Aktionen ausgeführt. Im Falle einer Zustandshierarchie werden die Eingangs-Aktionen der höchsten Abstraktionsebene zuerst, die der niedrigsten zuletzt berücksichtigt; für Ausgangs-Aktionen gilt die umgekehrte Reihenfolge.[10]

(a) (b)

Abb. 34: Mit einem Zustand verbundene Ein- und Ausgangs-Aktionen in (a) sowie eine Aktion im Sinne einer statischen Reaktion nach Abschnitt 5.1 in (b)

Es sind auch Aktionen möglich, die statischen Reaktionen nach Abschnitt 5.1 entsprechen. Abb. 34(b) zeigt ein Beispiel: Befindet sich ein Objekt im Zustand A und tritt das Ereignis a ein, so wird die Aktion b ausgeführt, ohne daß A verlassen wird oder Aktionen ausgeführt werden, die daran gekoppelt sind, daß A betreten oder verlassen wird.

Aktivitäten nehmen, wie in [Har87] geschildert, Zeit in Anspruch. Sie sind im dynamischen Modell immer mit einem Zustand verbunden. Rumbaugh und seine Koautoren unterscheiden zwei Arten von Aktivitäten: Aktivitäten können Geschehnisse, die nach einer bestimmten Zeitdauer selbständig enden (z. B. Berechnungen), oder ständig andauernde Vorgänge (z. B. Darstellungen von Überwachungsbildern auf einem Bildschirm) sein. Abb. 35(a) zeigt die OMT-Notation für Aktivitäten.

Gehört *activityA* der erstgenannten Kategorie von Aktivitäten an, so startet sie, sobald Zustand A betreten wird, und endet später selbständig. In diesem Fall führt die in Abb. 35(b) dargestellte Transition dazu, daß A nach dem Ende von *activityA* sofort verlassen und B eingenommen wird, da sie nicht mit einem Ereignis oder einer

[10] Rumbaugh et al. erzwingen mit dieser Festlegung, die sich an verschachtelten Funktionsaufrufen orientiert, daß die Ein- und Ausgangs-Aktionen serialisiert werden. Harel spricht hingegen lediglich von Nebenläufigkeit, die sich in der Struktur des Statechart niederschlägt, und läßt somit offen, ob die Aktionen seriell oder parallel ausgeführt werden.

Abb. 35: Eine Aktivität, die ausgeführt wird, wenn sich ein Objekt im Zustand *A* befindet, in (a); (b) zeigt eine Möglichkeit, wie *A* verlassen werden kann

Bedingung beschriftet ist. Ist *activityA* hingegen eine andauernde Aktivität, so startet sie, wenn *A* betreten wird, und endet vorzeitig, sobald ein Ereignis dazu führt, daß *A* verlassen wird. Ist der entsprechende Zustand mit Ausgangs-Aktionen verknüpft, so werden diese hierbei grundsätzlich ausgeführt.

Es ist möglich, den in Abb. 35(a) dargestellten Zustand gemeinsam mit der Aktivität *activityA* zu verfeinern, indem ein „One-shot"-Statechart erstellt wird. Abb. 36 zeigt ein Beispiel: Sobald *event2* eingetreten und somit der Endzustand erreicht worden ist, gilt *activityA* als beendet.

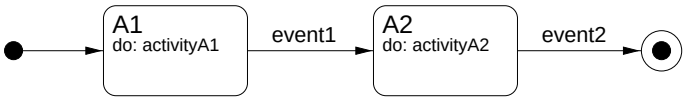


Abb. 36: Verfeinerung des Zustands aus Abb. 35(a) und seiner Aktivität *activityA*

Ebenso können Transitionen mitsamt den sie auslösenden Ereignissen verfeinert werden. Abb. 37 zeigt ein an [RBP+91] orientiertes Beispiel: Der Übergang in (a) kann durch den „One-shot"-Statechart in (b) beschrieben werden. Der Endzustand in Abb. 37(b) ist mit dem Ereignis beschriftet, das den Übergang in Abb. 37(a) auslöst.[11]

Während der Bereich der Aktionen und Aktivitäten gegenüber den Ausführungen in Abschnitt 3.4 nur wenig Neues mit sich bringt, ist der Aspekt der Nebenläufigkeit von besonderem Interesse. Das dynamische Modell beschreibt das Verhalten der zur Laufzeit existierenden Objekte einer Anwendung, die hinsichtlich ihrer Dynamik durch Statecharts beschrieben werden und jeweils einen eigenen Zustand besitzen. Der Zustand eines Systems zur Laufzeit wird durch die Zustände all dieser Objekte bestimmt. Die Objekte können zunächst einmal als nebenläufig betrachtet werden; ihre gegen-

[11] Man könnte auch eine Aktivität definieren, die dafür steht, daß eine Nummer eingegeben wird, zumal die Abb. 37(b) Aktivitäten enthält, die Zeit benötigen, und somit das Ereignis in Abb. 37(a) eigentlich ebenfalls nicht zeitlos ist. Das Beispiel zeigt, daß die Grenze zwischen Ereignissen und Aktionen auf der einen sowie Aktivitäten auf der anderen Seite manchmal schwer zu ziehen ist: In der Realität benötigt jeder Vorgang eine gewisse Zeitdauer; Ereignisse, Aktionen und Aktivitäten sind theoretische Konzepte, die letztlich alle auf Methoden abgebildet werden müssen.

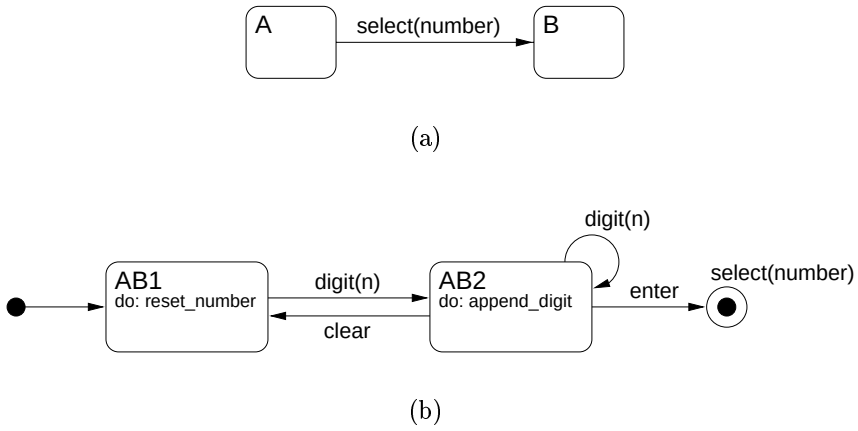


Abb. 37: Eine Transition in (a) und eine mögliche Verfeinerung in (b)

seitige Unabhängigkeit wird durch den Programmcode eingeschränkt, denn nicht jede denkbare Kombination von Objektzuständen soll zulässig sein.

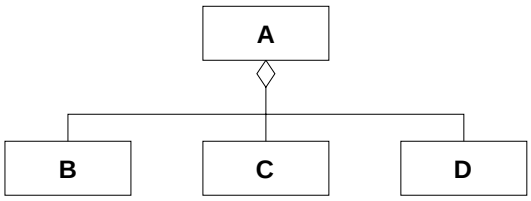


Abb. 38: Ein Beispiel für eine Aggregation

Als ein Spezialfall können Aggregationen betrachtet werden. Abb. 38 zeigt ein Beispiel für eine Aggregationsbeziehung: Jedes Objekt der Klasse A besitzt drei Objekte, jeweils genau eines der Klassen B, C und D, die seinen Zustand maßgeblich beeinflussen. Die Statecharts, die das Verhalten der Objekte der Klassen B, C und D beschreiben, sind nebenläufig und bilden einen wesentlichen Bestandteil der Verhaltensbeschreibung für Objekte der Klasse A. Im allgemeinen bestehen über Bedingungen, die Annahmen über aktuelle Zustände beinhalten, Abhängigkeiten und Wechselwirkungen zwischen diesen Statecharts.

Die nebenläufigen Objekte einer Anwendung können interagieren, indem sie sich gegenseitig Ereignisse senden. Rumbaugh und seine Koautoren sehen hierfür die Aktion „*send* $E(attributes)$“ vor, die ein Ereignis E mit seinen Attributen an ein Objekt oder an mehrere Objekte sendet. Alle Objekte, deren Statechart Transitionen aufweist, die durch E ausgelöst werden können, können nebenläufig auf E reagieren. Abb. 39 zeigt eine alternative Darstellung dafür, daß ein Objekt ein Ereignis sendet. Bei dieser Form ist die Wirkung des gesendeten Ereignisses besser nachzuvollziehen, da der beeinflußte Statechart über die aufgeführte Klasse leicht auszumachen ist.

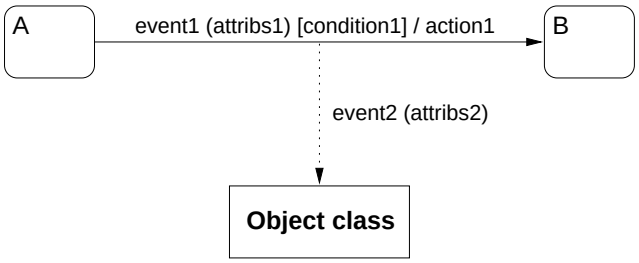


Abb. 39: Während eines Zustandsübergangs wird ein Ereignis an ein anderes Objekt gesendet

Nebenläufigkeit kann nicht nur durch mehrere Statecharts zustandekommen, sondern auch innerhalb eines Objektzustands auftreten. Die Grundlage hierfür ist eine Partitionierung der Menge der Exemplarvariablen; jede Teilmenge wird dann durch eine orthogonale Komponente eines gemäß der AND-Dekomposition nach [Har87] verfeinerten Zustands repräsentiert. Die Regeln dafür, wie ein derartiger Zustand eingenommen und verlassen werden kann, entsprechen den in Abschnitt 3.3 genannten. Rumbaugh et al. bemerken vor dem Hintergrund ihrer praktischen Erfahrungen mit dem dynamischen Modell, daß Nebenläufigkeit hauptsächlich durch Aggregation von Objekten auftrete und seltener explizit in den Zustandsdiagrammen ausgedrückt werden müsse.

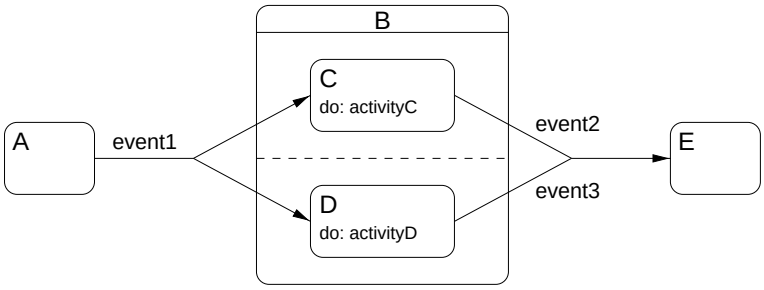


Abb. 40: Ein Zustand mit nebenläufigen Aktivitäten

Eine Besonderheit kann sich im Zusammenhang mit nebenläufigen Aktivitäten ergeben. Abb. 40 zeigt ein Beispiel in der OMT-Notation für AND-Zustände: Die Aktivitäten *activityC* und *activityD* werden nebenläufig ausgeführt; der Zustand E kann erst betreten werden, wenn beide beendet sind. Die Ereignisse *event2* und *event3* können verschieden sein. Sie müssen beide eintreten, damit der Zustand B verlassen werden kann. Die entsprechenden Zeitpunkte sind hierbei unerheblich: *event2* und *event3* können gleichzeitig oder nacheinander registriert werden, wobei im zweiten Fall die Reihenfolge und der zeitliche Abstand keine Rolle spielen.

6.2.3 Erkannte Probleme

Bei der Betrachtung des dynamischen Modells sind uns zwei mögliche Problemkomplexe besonders aufgefallen. Nach Rumbaugh et al. kann ein Ereignis über die Grenzen verschiedener Klassen hinweg definiert und die Ereignishierarchie, die eine zusätzliche Abstraktionsebene darstellt und eine stärkere Orientierung an der realen Welt bedeutet, als unabhängig von der Klassenhierarchie aufgefaßt werden. Im Verlauf des Entwicklungsprozesses werden Ereignisse jedoch auf konkrete, klassenspezifische Methoden abgebildet, so daß diese Unabhängigkeit eingeschränkt werden muß. Rumbaugh et al. gehen allerdings nicht näher darauf ein, in welcher Weise die Ereignishierarchie die Formulierung von Methoden beeinflußt.

Es stellt sich ferner die Frage, in welcher Form sich die Beziehungen zwischen Ober- und Unterklassen in den zugehörigen Statecharts niederschlagen sollten. Durch Vererbung können in Unterklassen z.B. neue Exemplarvariablen eingeführt und Methoden redefiniert werden; die Struktur der Statecharts sollte dies widerspiegeln. Werden in einer Unterklasse lediglich Exemplarvariablen und Methoden, die nur auf diesen arbeiten, hinzugefügt, so kann der Statechart der Unterklasse aus dem Statechart der Oberklasse einfach dadurch gebildet werden, daß eine zusätzliche orthogonale Komponente vorgesehen wird. Im allgemeinen liegt jedoch ein komplizierterer Fall vor, der weitergehende Maßnahmen erfordert und u.a. häufig dazu zwingt, Zustände durch Subzustände zu verfeinern, um Fallunterscheidungen zu ermöglichen. Rumbaugh et al. schlagen vor, die Statecharts für Unterklassen weitgehend unabhängig von denjenigen der Oberklassen zu definieren, und sprechen davon, daß ein Unterklassen-Statechart die Statecharts aller in der Klassenhierarchie höher liegenden Klassen erben solle, ohne hierauf näher einzugehen. Die Beziehung zwischen Ober- und Unterklassen-Statecharts ist nach unserer Ansicht ein wichtiger Punkt; [PR94] beinhaltet einige Betrachtungen hierzu.

6.3 Das dynamische Modell im Entwicklungsprozeß

In diesem Abschnitt wird in komprimierter Form geschildert, wie das dynamische Modell im Verlauf des Entwicklungsprozesses nach OMT erstellt und überarbeitet wird und schließlich die Implementation beeinflußt. Eine klare Trennung der im folgenden beschriebenen Schritte ist nach Rumbaugh et al. nicht immer möglich und häufig auch gar nicht gewünscht.

6.3.1 Analyse

Im Rahmen der Analyse sollte zunächst das Objektmodell erstellt werden, indem Objekte und Klassen identifiziert und in einem *data dictionary* beschrieben sowie Vererbungsbeziehungen, Assoziationen, Exemplarvariablen und Link-Attribute festgehalten werden. Die Klassen des Objektmodells besitzen anfangs noch keine Operationen.

Im Anschluß kann man das dynamische Modell auf der Grundlage von *Szenarien* konstruieren. Unter einem Szenario verstehen Rumbaugh et al. eine Folge von Ereignis-

sen, die in natürlicher Sprache formuliert werden. Jedes Ereignis bedeutet hierbei eine Übertragung von Information zwischen zwei Objekten. Die beteiligten Objekte sowie eventuelle Ereignis-Parameter müssen stets erkennbar sein. Die Szenarien sollen typische Dialoge zwischen einem Benutzer und dem zu entwickelnden System darstellen; einige wesentliche reichen aus. Rumbaugh und seine Koautoren empfehlen, zuerst die üblichen Vorgänge, dann spezielle Fälle und schließlich Fehlersituationen zu berücksichtigen.

Sind Szenarien erstellt, so kann man Ereignisse zu Gruppen zusammenfassen, indem man von den aktuellen Parametern abstrahiert. Daraufhin kann man jedes Ereignis mit denjenigen Klassen in Verbindung bringen, deren Objekte es senden oder empfangen, und diesen Sachverhalt grafisch in einem *event trace* darstellen. Abb. 41 zeigt einen Ausschnitt eines event trace aus [RBP+91], der ein Szenario für einen Telefonanruf wiedergibt.

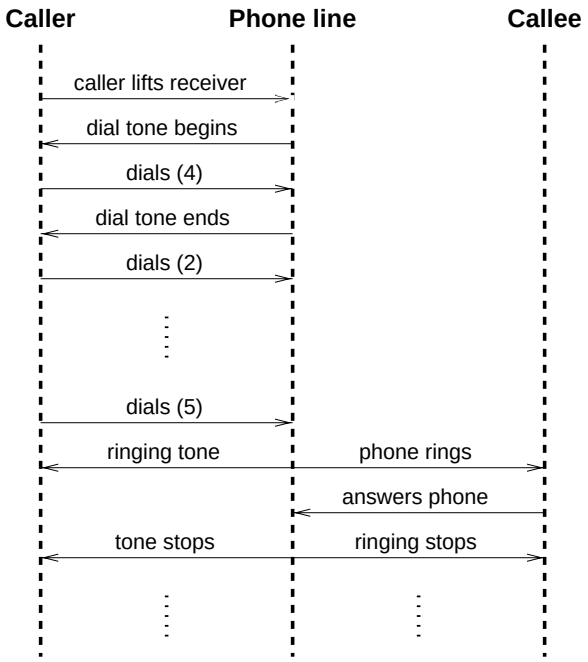


Abb. 41: Ein Teil eines *event trace* für einen Telefonanruf

Event traces sind Interaktionsdiagrammen sehr ähnlich. Objekte werden durch senkrechte Linien dargestellt; sie sind durch beschriftete Pfeile, die Ereignisse repräsentieren, miteinander verbunden. Über diese Darstellung können alle Ereignisse gefunden werden, die mit einem bestimmten Objekt in Verbindung stehen; nur diese können später im Statechart der zugehörigen Klasse auftreten. Event traces halten keine Kontrollflüsse fest, sondern geben nur die Reihenfolge von Ereignissen sowie deren Sender- und Empfänger-Objekte wieder.

Die erstellten event traces können schließlich in einem *Ereignis-Flußdiagramm* zusammengefaßt werden. Die Knoten eines derartigen Diagramms sind die Klassen, die in den event traces vorkommen. Untereinander sind diese Knoten mit Pfeilen verbunden,

die jeweils mit den Ereignissen beschriftet sind, die zwischen den entsprechenden Klassen ausgetauscht werden. An einem Ereignis-Flußdiagramm kann somit leicht abgelesen werden, welche Ereignisse eine Klasse generiert und welche sie empfängt.

Auf der Basis des bisher Erarbeiteten kann nun für jede Klasse, deren Verhalten nicht trivial ist, ein Statechart erstellt werden. In einem Statechart korrespondiert jedes Szenario und somit jeder event trace zu einem Pfad. Möchte man das Verhalten einer bestimmten Klasse beschreiben, so zieht man alle relevanten event traces heran und konzentriert sich auf die Ereignisse, die die Klasse beeinflussen. Man beginnt damit, für einen typischen event trace die Ereignisse in einem Pfad anzuordnen. Die Pfeile werden mit den empfangenen und den gesendeten Ereignissen beschriftet; Zustände müssen zunächst nicht unbedingt einen Namen erhalten. Kann der event trace beliebig oft durchlaufen werden, so kann man den Pfad in einen Zyklus umformen. In ähnlicher Weise können Schleifen in den Graphen integriert werden, wenn Ereignissequenzen immer wieder durchlaufen werden können.

Sind alle Informationen des event trace genutzt worden, so kann der Graph mit Hilfe der anderen event traces nach und nach erweitert werden. Hierbei sollten zunächst die übrigen typischen Szenarien, dann Spezialfälle und zuletzt Fehlersituationen berücksichtigt werden. Nachdem alle event traces in den Graphen integriert worden sind, empfiehlt es sich, im Statechart nach Fehlern zu suchen und über eventuell vergessene Fälle, die nicht durch Szenarien abgedeckt werden, nachzudenken. Im allgemeinen erkennt man bereits während des beschriebenen Prozesses, daß man bestimmte Ereignisse nicht berücksichtigt hat. Nach Rumbaugh et al. kann man Statecharts nach einiger Übung entwickeln, ohne zuvor event traces erstellt zu haben. Szenarien sind jedoch stets eine wichtige Grundlage.

Sind für alle Klassen Statecharts erstellt worden, so sollte man untersuchen, ob diese konsistent sind. So muß z. B. jedes Ereignis einen Sender- und einen Empfänger-Statechart haben, und alle Szenarien müssen durch das Zusammenspiel der Statecharts möglich sein. Die eventuell korrigierten Statecharts bilden gemeinsam das dynamische Modell.

Nach dem dynamischen Modell kann schließlich das funktionale Modell erstellt werden. Zu diesem Zweck werden anfangs alle Parameter von Ereignissen, die zwischen dem System und seiner Umgebung ausgetauscht werden, aufgelistet. Mit Hilfe von Datenflußdiagrammen kann festgehalten werden, welche Schritte nötig sind, damit aus Eingabewerten Ausgabewerte berechnet werden können. Die Prozesse in diesen Diagrammen korrespondieren zu Aktivitäten oder Aktionen, die in den Statecharts vorkommen, und werden fortlaufend durch detaillierte Datenflußdiagramme verfeinert; die Datenflüsse beziehen sich auf Objekte oder Werte von Exemplarvariablen. Wenn ein Prozeß als ausreichend verfeinert angesehen wird, so wird die Funktion, die ihn realisiert, in natürlicher Sprache, durch Pseudocode, mit Hilfe von Gleichungen oder auf eine andere Weise hinsichtlich ihrer Wirkung beschrieben, ohne daß ein Algorithmus festgelegt wird.

6.3.2 Systemdesign, Objektdesign und Implementation

Während des Systemdesigns legt man u. a. die Strategie fest, mit deren Hilfe das dynamische Modell realisiert werden soll. Rumbaugh et al. betrachten verschiedene Arten von Systemen, die sich durch unterschiedliche derartige Strategien auszeichnen. Drei dieser Systemklassen stellen wir im folgenden kurz vor und verweisen für eine vollständige Darstellung auf [RBP+91]:

- *Prozedurgesteuerte sequentielle Systeme:*
  Bei einem prozedurgesteuerten sequentiellen System liegt die Kontrolle über die Dynamik explizit im Programmcode. Eingaben an ein solches System müssen von Prozeduren angefordert werden; bis zur Eingabe befindet sich der Kontrollfluß an der entsprechenden Stelle in der Prozedur. Ist die Eingabe erfolgt, so wird damit fortgefahren, die Befehle der Prozedur auszuführen. Im Falle eines solchen Systems müssen die Ereignisse in Operationen umgesetzt werden, wobei eine Operation häufig zu einem Ereignis-Paar korrespondiert. Ein derartiges Paar besteht i. d. R. aus einem Ausgabe-Ereignis, das eine Ausgabe darstellt und eine Eingabe erfordert, sowie einem Eingabe-Ereignis, das Eingabewerte liefert. Komfortable Benutzungsschnittstellen sind auf diese Weise nur äußerst schwierig zu erstellen.

- *Ereignisgesteuerte sequentielle Systeme:*
  Liegt ein ereignisgesteuertes sequentielles System vor, so wird die Kontrolle über die Dynamik von einer Komponente übernommen, die die Programmiersprache, die Programmierumgebung oder das Betriebssystem zur Verfügung stellt. Ereignisse wie z. B. Mausklicks werden von dieser Komponente registriert, welche Callback-Operationen aufruft, die vor dem Systemstart mit den Ereignissen verknüpft worden sind. Mit Hilfe von Operationsaufrufen kann die Komponente dazu veranlaßt werden, etwas auszugeben oder eine Eingabe entgegenzunehmen. Die Kontrolle wird hierbei wieder der Komponente übergeben. Ereignisgesteuerte Systeme sind prozedurgesteuerten Systemen vorzuziehen, da Ereignisse, die im Statechart vorkommen, mit Hilfe des beschriebenen Kontrollmechanismus elegant auf Programmkonstrukte abgebildet werden können. Angesichts der weiten Verbreitung von Programmierumgebungen mit GUI-Fähigkeiten auf Ein-Prozessor-Systemen kann diese Art der Umsetzung heutzutage als Standard angesehen werden.

- *Nebenläufige Systeme:*
  In einem nebenläufigen System ist die Kontrolle auf mehrere unabhängige Tasks verteilt. Ereignisse können als Nachrichtenübertragungen zwischen diesen Tasks implementiert werden. Während ein Task auf eine Eingabe wartet, können die übrigen in ihrer Ausführung fortfahren. Über das Betriebssystem können Ereignisse in eine Warteschlange eingereiht und Konflikte zwischen den Tasks aufgelöst werden. Nebenläufige Systeme bieten vor allem den Vorteil, daß sich die inhärente Nebenläufigkeit realer Objekte in den Software-Objekten niederschlagen kann.

Im Rahmen des Objektdesigns ergänzt man das während der Analyse erstellte Modell, das sich ausschließlich auf fachliche Konzepte bezieht, um rechnerbezogene Aspekte. Die im Verlauf des Systemdesigns entwickelte Strategie wird umgesetzt, und die drei Sichten auf das System führen gemeinsam zur Definition der Operationen im Objektmodell, wobei sich diese Operationen aus jedem der drei Modelle ergeben können.

Zum einen kann man anhand des Objektmodells selbst Operationen erkennen: Sollen Exemplarvariablen von anderen Objekten gelesen oder geschrieben werden können, so muß man entsprechende Operationen zur Verfügung stellen. Zum anderen ist es notwendig, die in den Statecharts beschriebenen Kontrollstrukturen umzusetzen. Die Ereignisse, die ein Objekt empfängt, werden auf Operationen abgebildet. Häufig treten Ereignisse paarweise auf: Ein Ereignis steuert einen Vorgang an, während ein zweites die entsprechenden Ergebnisse zurückliefert oder einfach nur das Ende des Vorgangs anzeigt. Solche Ereignis-Paare können mit einer Operation identifiziert werden, die den jeweiligen Vorgang ausführt; oftmals muß eine derartige Operation an einem anderen Objekt ausgeführt werden. Aktivitäten werden ebenfalls in Operationen umgesetzt und können – wie auch Aktionen – im funktionalen Modell zu Datenflußdiagrammen expandieren. Darüber hinaus ist es möglich, aus den Datenflußdiagrammen des funktionalen Modells weitere Operationen abzuleiten.

Überprüft man das angereicherte Objektmodell, so stößt man in der Regel auf Fehler, die korrigiert werden müssen. So ist es z. B. denkbar, daß einige Operationen inkonsistent sind, oder es könnte mehrere Operationen geben, die ähnliche Wirkungen haben. Von besonderer Bedeutung sind an diesem Punkt die Umgangsformen der realen Gegenstände, die sich in den Klassen widerspiegeln sollen: Sie können Fehler in bestehenden Operationen verdeutlichen oder aufzeigen, daß bestimmte Operationen schlichtweg vergessen worden sind. Nach den Korrekturen müssen schließlich für alle Operationen, um die man das Objektmodell ergänzt hat, Algorithmen gefunden werden.

Da die wesentlichen Entscheidungen bereits im Rahmen des Designs getroffen werden, kann man sich während der Implementation darauf konzentrieren, welche Sprachmittel eingesetzt werden sollen, um die entwickelte Struktur in Programmcode zu übertragen. Eine objektorientierte Programmiersprache ist selbstverständlich am besten geeignet, um den Entwurf umzusetzen, jedoch unterstützen verschiedene objektorientierte Sprachen einzelne objektorientierte Konzepte i. a. unterschiedlich gut. In [RBP+91] betrachten Rumbaugh et al. die Möglichkeiten, die die Sprachen Eiffel, C++ und Smalltalk bieten, und beschreiben außerdem, wie prozedurale Programmiersprachen oder relationale Datenbanken verwendet werden können, um einen Entwurf nach OMT schließlich auf einem Rechner zu realisieren.

7 Objectcharts

Neben der im vorigen Abschnitt behandelten Methodik, bei deren praktischer Anwendung Statecharts oft als eher informales Beschreibungsmittel genutzt werden, existieren weitere Ansätze zur Lösung der Frage, wie der grafische Formalismus geeignet erweitert oder verändert werden kann, um im objektorientierten Umfeld einsetzbar zu sein. In diesem Abschnitt wird die von Derek Coleman, Fiona Hayes und Stephen Bear in [CHB92] vorgeschlagene Notation der *Objectcharts* erläutert und dargelegt, auf welche Weise aus Sicht der Entwickler dieser Notationsform eine Systembeschreibung erfolgen kann.

Den Ausgangspunkt dieses Ansatzes bildet der 1991 erschienene Artikel [HC91], in dem Hayes und Coleman Modelle für die objektorientierte Analyse, welche die Formulierung und Diskussion objektorientierter Beschreibungen eines Problemgebietes erleichtern sollen, betrachten. Unbestritten ist für die Autoren, daß die zur Modellbeschreibung herangezogenen Notationen so mächtig sein sollten, daß der Analytiker in die Lage versetzt wird, geeignete Beschreibungen zu erstellen, und nicht mit den zur Verfügung stehenden Mitteln zu „kämpfen“ hat. Würde das Ausdrucksvermögen allein jedoch schon ein geeignetes Kriterium zur Bewertung einer Notation darstellen, so wäre die natürliche Sprache für derartige Modellbeschreibungen am besten geeignet. Coleman und Hayes betonen, daß neben der Ausdruckskraft die Möglichkeit zur genauen Verständigung über die zu behandelnde Domäne ein wesentliches Kriterium zur Bewertung von Modellen und eingesetzten Notationen ist. Folglich sollten die gewählten Beschreibungsmittel unzweideutig, abstrakt und konsistent sein. Zur Gewährleistung der Konsistenz ist es erforderlich, für die verschiedenen Modelle, die zur Beschreibung desselben Systems verwendet werden, entscheiden zu können, ob sie sich widersprechen. Grundsätzlich sollte, so die Autoren in [HC91], eine Methode zur objektorientierten Analyse helfen, konsistente Modelle zu entwickeln, die eine begründete und überzeugende Beschreibung der Problemdomäne darstellen.

Wesentlicher Gegenstand der Betrachtungen von Coleman und Hayes ist die Frage, inwieweit die zum Erscheinungszeitpunkt von [HC91] vorliegenden Ansätze zur objektorientierten Analyse das Kriterium der genauen Verständigung über die Anwendungsdomäne erfüllen und welche Erweiterungen und Änderungen vorzunehmen sind, um geeignete Notationen zu erhalten. Hinsichtlich des ersten Aspekts dieser Fragestellung gilt aus Sicht der Autoren, daß es den bestehenden Ansätzen, z. B. in [CY90], [SM88] und [RBP+91], insgesamt an Präzision mangelt. Für die detaillierteren Ausführungen zum objektorientierten Analyseprozeß und zu den während der Analyse entstehenden Modellen orientieren sich Coleman und Hayes an den Beschreibungen zu OMT in [RBP+91]. Als Ergebnis halten sie fest, daß die mittels OMT in der Analyse erstellten Modelle nicht zwangsläufig eine kohärente Menge von Beschreibungen bilden, und führen folgende Punkte als mögliche Gründe an:

1. Im Gegensatz zu den für die Beschreibung von Objektmodellen verwendeten ER-Notationen, die über eine formale Semantik verfügen und eindeutig sind, gilt für

die eingesetzten Zustandsautomaten und Datenflußdiagramme, daß sie weniger formal sind und eher zu Mehrdeutigkeiten neigen, weil sie bei den Definitionen der Grundkonzepte, z. B. Aktion und Prozeß, auf natürliche Sprache zurückgreifen.

2. Jeder Zustandsautomat definiert das Verhalten individueller Objekte; es gibt jedoch keine grundsätzliche Definition der Kommunikation zwischen Automaten, so daß das Verhalten einer Menge miteinander kommunizierender Objekte nicht aus den Verhaltensdefinitionen der einzelnen Objekte abgeleitet werden kann. Somit ist es nicht möglich, das dynamische und das funktionale Modell hinsichtlich ihrer Konsistenz zu überprüfen.

3. Datenflußdiagramme (DFD's) stellen kein adäquates Beschreibungsmittel für das funktionale Modell dar, weil ein in einem Prozeß vorgegebener Datenfluß Teile der Berechnungsreihenfolge festlegt. Die bei Rumbaugh vorgesehene Verwendung, in der DFD's nur der Angabe von benötigten Ein- sowie berechneten Ausgabewerten dient, ist nach [HC91] ebenfalls ungeeignet: Durch die Forderung, daß Prozesse auf unterster Ebene Operationen an Objekten entsprechen sollen, entsteht zum Teil eher eine Beschreibung spezieller Berechnungen als eine Spezifikation des allgemeinen Systemverhaltens.

Aus den genannten Punkten folgern Coleman und Hayes, daß es notwendig ist, den einzelnen in OMT eingesetzten Modellen eine formalere Grundlage zu geben sowie zusätzlich im funktionalen Modell stärker zu abstrahieren. Als mathematische Basis für die Formalisierung schlagen sie eine Ableitung von VDM vor. Das Resultat dieser Überlegungen besteht aus den nachfolgend kurz erläuterten Modellen, die, so Coleman und Hayes, präziser als ihre OMT-Vorgänger sind und hinsichtlich ihrer Konsistenz überprüft werden können.

Das *Objektstrukturmodell*, das den Verbindungspunkt zwischen Objektmodell, dynamischem und funktionalem Modell bildet, stellt eine Verfeinerung des Objektmodells von OMT dar. In dieser Verfeinerung werden binäre Beziehungen zwischen Objekten durch Attribute in jedem Objekt ersetzt; jedes dieser Attribute enthält eine Liste von Verweisen auf diejenigen Objekte, mit denen das betrachtete Objekt über die entsprechende Beziehung verbunden ist.[1] Für die Beschreibung des *funktionalen Modells* verzichten Coleman und Hayes auf die Datenflußdiagramme und spezifizieren die Systemoperationen deklarativ durch Angabe von Vor- und Nachbedingungen über dem Objektstrukturmodell. Das auf diese Weise formal beschriebene funktionale Modell steht mit den anderen Modellen in Verbindung, indem auf das Objektstrukturmodell Bezug genommen wird. Für die im dynamischen Modell vorgenommene Spezifikation der Interaktion zwischen einzelnen Objekten des Systems werden die in Abschnitt 7.2 ausführlich erläuterten *Objectcharts* verwendet, in denen die Auswirkungen von Ereignissen auf die einzelnen Attribute eines Objektes durch Angabe von Vor- und Nachbedingungen beschrieben werden. Die so entstehende Verbindung zum Objektstrukturmodell bildet zusammen mit der Verbindung zwischen funktionalem und

[1] Es werden keine Aussagen hinsichtlich der in OMT ebenfalls möglichen ternären Relationships getroffen.

Objektstrukturmodell die Basis, auf der die Konsistenz der Modelle überprüft werden kann.[2]

Die Erstellung einer solchen kohärenten Menge von Notationen für die objektorientierte Analyse auf der Basis von Objectcharts, Vor- und Nachbedingungen sowie Rumbaughs objektorientierter Analysetechnik stellt ein wesentliches Forschungsgebiet für Coleman und Hayes dar, wie sie in [CHB92] darlegen. In dem genannten Artikel gehen sie zusammen mit Stephen Bear gezielt auf das Beschreibungsmittel „Objectcharts" ein und definieren das Verhalten eines statischen Systems von Objekten – abweichend von dem oben angeführten, an OMT angelehnten Ansatz – anhand eines sogenannten *Configuration Diagram* und einer Menge von Objectcharts. Auf die anderen in [HC91] erläuterten Modelle[3] wird in [CHB92] nicht eingegangen, so daß nicht ersichtlich ist, ob und in welcher Weise diese Modelle in den Beschreibungsansatz aus [CHB92] eingebunden werden sollen.

Im folgenden wird anhand eines stark vereinfachten Beispiels, einer Kesselsteuerung, der Aufbau von Configuration Diagrams und Objectcharts erläutert und dargelegt, welche Informationen in ihnen enthalten sind und auf welche Weise sich aus dem Zusammenspiel zwischen Configuration Diagram und den Objectcharts zu einzelnen Klassen das Verhalten des beschriebenen Systems ableitet. Als Kesselsteuerung sei eine Einrichtung bezeichnet, mittels der die Temperatur in einem Heizkessel gesteuert werden kann.[4] Sie dient dazu, eine in einem Heizkessel befindliche Flüssigkeit auf eine vorgegebene Temperatur zu erhitzen und diese anschließend für eine bestimmte Zeit zu halten. Für einen Benutzer dieser Steuerung stehen folgende Möglichkeiten zur Verfügung, um den Heizvorgang im Kessel zu beeinflussen:

- Er kann die zu erreichende Zieltemperatur festlegen, sofern noch keine Zeit vorgegeben ist, für die die aktuelle Temperatur im Heizkessel gehalten werden soll, oder diese schon abgelaufen ist.

- Ist im Heizkessel die vorgegebene Zieltemperatur erreicht, so hat der Benutzer die Möglichkeit, eine Zeitdauer anzugeben, für die die aktuelle Temperatur im Kessel gehalten werden soll, oder eine neue Zieltemperatur vorzugeben.

- Bei einer vorgegebenen Haltedauer für die aktuelle Temperatur kann die Restzeit, d. h. die bis zum Ende der Haltedauer verstreichende Zeit, auf Null zurückgesetzt werden.

- Der Heizkessel kann an- und ausgeschaltet werden, z. B. für Reparaturen. Des weiteren wird nach Ablauf der Haltedauer, d. h. wenn die Restzeit Null beträgt, die Zieltemperatur automatisch auf Null gesetzt, wenn für eine bestimmte Zeit vom Benutzer keine weiteren Vorgaben gemacht worden sind.

[2] Die Überprüfung selbst erfolgt, indem für jede im dynamischen Modell mögliche Ereignisfolge getestet wird, ob sie wenigstens den Effekt der funktionalen Spezifikation erbringt und mindestens in den Situationen, die die funktionale Spezifikation erlaubt, auftreten kann.

[3] Funktionales Modell sowie Objekt- und Objektstrukturmodell

[4] Die hier gegebene Beschreibung einer Kesselsteuerung erhebt keinen Anspruch auf technische Korrektheit oder Vollständigkeit.

- Über ein Informationsfenster wird der Benutzer benachrichtigt, wenn während des Heizvorganges bestimmte Situationen (z. B. „Zieltemperatur erreicht“) eintreten. Der Benutzer muß kenntlich machen, daß er die Meldungen gelesen hat, da anderenfalls von einem nicht überwachten Heizvorgang ausgegangen und der Heizkessel abgeschaltet wird (siehe vorherigen Punkt).

- Vom Benutzer können die Ziel- sowie die aktuelle Temperatur und die noch abzulaufende Restzeit abgefragt werden.

Der folgende Abschnitt dient dazu, den allgemeinen Aufbau von Configuration Diagrams zu erläutern und ein mögliches Configuration Diagram für das hier beschriebene Beispiel vorzustellen. Anschließend werden in Abschnitt 7.2 anhand dieses Beispiels die Struktur und die Komponenten der Objectcharts veranschaulicht. Abschnitt 7.3 vermittelt schließlich, wie nach [CHB92] das Systemverhalten dargestellt werden kann.

7.1 Configuration Diagram

In der ersten Komponente einer Systembeschreibung nach [CHB92], dem *Configuration Diagram*, werden die im System vorhandenen Objekte sowie die Interaktionsmöglichkeiten zwischen ihnen dargestellt. Coleman et al. beschränken sich in ihrem Artikel auf die Spezifikation statischer Systeme mit einer festen Menge von Objekten, um die Beschreibungen einfacher zu halten.

Ein Configuration Diagram besteht aus einer Menge von Rechtecken, die die im System vorhandenen Objekte repräsentieren und jeweils mit einem im System eindeutigen Objektbezeichner (*object identifier*) sowie dem zugehörigen Klassennamen versehen sind. Jedem dieser Rechtecke können beschrifteten Linien zugeordnet werden (siehe Abb. 42(a)). Durchgezogene Linien an einem Objekt stellen die von ihm angebotenen

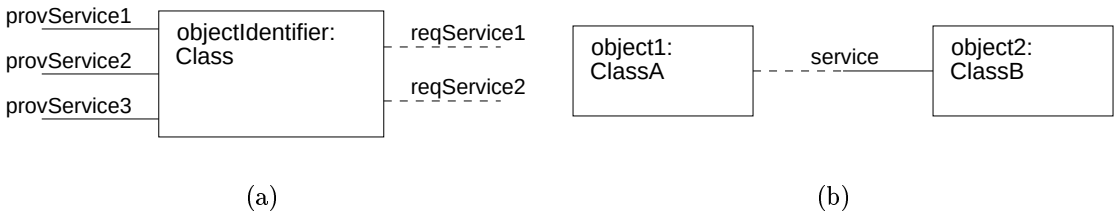


Abb. 42: Ein Objekt mit Interfaces in (a) und Darstellung einer möglichen Objektinteraktion in (b)

Dienste (*provided interface*) dar; ist es für ein Objekt erforderlich, daß andere Objekte des Systems bestimmte Dienste zur Verfügung stellen, so werden diese Dienste durch gestrichelte Linien an dem betrachteten Objekt repräsentiert (*required interface*).

Die grafische Darstellung der vorgesehenen Interaktionsmöglichkeiten zwischen den Objekten des Systems, d. h. des allgemeinen Musters der Kommunikation, erhält man, indem zueinander passende Elemente der required und provided interfaces einzelner

Objekte miteinander verbunden werden (siehe Abb. 42(b)).[5] Anhand der so festgelegten Systemstruktur und möglichen Interaktionswege kann später in Kombination mit den in Abschnitt 7.2 erläuterten Objectcharts das mögliche Systemverhalten ermittelt werden. Das Kommunikationsmuster selbst beinhaltet jedoch noch keine Kontroll- oder Reihenfolgestrukturen, sondern legt ausschließlich fest, welche Dienste die einzelnen Objekte zur Verfügung stellen und von anderen Objekten in Anspruch nehmen können.

Für das hier verwendete Beispiel der Kesselsteuerung ist ein Configuration Diagram wie in Abb. 43, nach dem das System aus fünf Objekten besteht, denkbar. Das

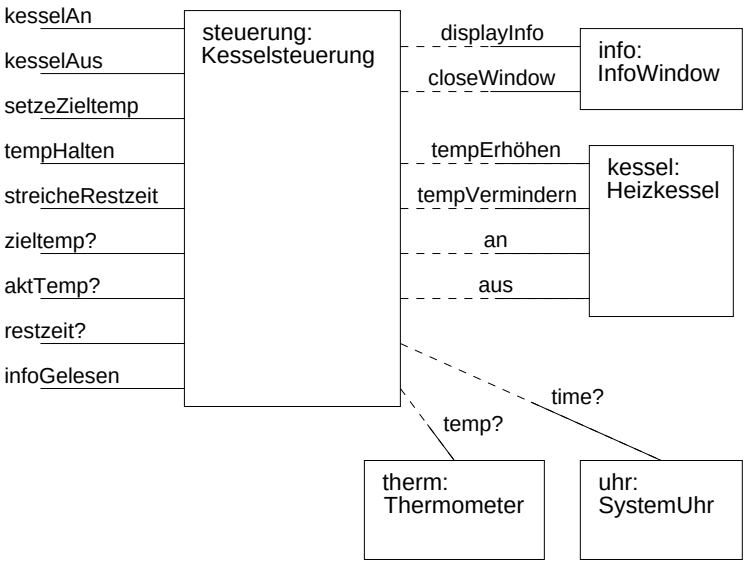


Abb. 43: Ein Configuration Diagram zum Beispiel der Kesselsteuerung

Exemplar steuerung der Klasse *Kesselsteuerung* ist die wesentliche Komponente dieses Systems und stellt dem Benutzer diejenigen Dienste zur Verfügung, mit denen der Heizkessel kessel gesteuert werden kann. Es benötigt eine Reihe von Diensten, die andere Objekte im System bereitstellen. So wird beispielsweise das Objekt kessel durch steuerung an- und ausgeschaltet und dazu veranlaßt, die Temperatur zu erhöhen oder zu vermindern. Um ermitteln zu können, ob die Temperatur in kessel verändert werden muß, greift steuerung auf das Objekt therm zurück, das die aktuelle Temperatur im Heizkessel angibt. Zusätzlich wird das Objekt uhr benötigt, wenn die Temperatur im Heizkessel für eine bestimmte Zeitdauer gehalten werden soll. Über das Exemplar info der Klasse *InfoWindow* zeigt steuerung Benachrichtigungen für den Benutzer an.

[5] Ein angebotener und ein benötigter Dienst passen zusammen, wenn sie mit der gleichen Beschriftung versehen sind.

7.2 Objectcharts als Erweiterung von Statecharts

Während das Configuration Diagram zu einem System Informationen über dessen Aufbau, d. h. die vorhandenen Objekte und ihre möglichen Kommunikationswege, enthält, wird in der zweiten Komponente einer Systembeschreibung nach [CHB92] das sogenannte *Klassenverhalten*[6] definiert. Zu jeder nicht trivialen Klasse wird eine Beschreibung erstellt, die ein allgemeines Muster für das Verhalten der einzelnen Exemplare der jeweiligen Klasse liefert; das konkrete Verhalten eines Objektes innerhalb des Systems läßt sich aus dem definierten Klassenverhalten unter Berücksichtigung des Configuration Diagrams ableiten.

Als Darstellungsmittel für das Klassenverhalten verwenden Coleman et al. die von ihnen entwickelten *Objectcharts*; diese stellen eine Erweiterung der von Harel eingeführten Statecharts dar, in der die Effekte der möglichen Transitionen auf die Attribute der beschriebenen Objekte spezifiziert werden. Ein Objectchart besteht aus einem modifizierten Statechart sowie einer Reihe von Transitions- und Invariantenspezifikationen und wird genau einer Klasse zugeordnet.

7.2.1 Modifizierte Statecharts

In der für Objectcharts eingeführten Variante von Statecharts liegt ein Schwerpunkt auf dem für die Interaktion von Objekten verwendeten Kommunikationsmechanismus. Zur Kommunikation zwischen den erstellten Objectcharts sind bei Coleman et al. ausschließlich direkte Dienstanfragen (*service requests*), z. B. in Form von Methodenaufrufen, vorgesehen. Auf den von Harel in [Har87] eingeführten Broadcast-Mechanismus wird bewußt verzichtet, weil er, so die Autoren in [CHB92], kein passendes Modell für die Objektinteraktion darstelle; der direkte Methodenaufruf entspreche stärker dem Gedanken der Objektorientierung. Jedes Objekt durchläuft einen Lebenszyklus, der durch Zustandsänderungen geprägt ist. Diese werden hervorgerufen, indem das betrachtete Objekt Dienste anderer Objekte im System in Anspruch nimmt oder zustandsändernde Methoden an ihm selbst aufgerufen werden.

Aufgrund der genannten Überlegungen sind die Zustandsübergänge in den für die einzelnen Klassen entwickelten Statecharts mit den entsprechenden Methodennamen zu beschriften. Man kann die von einem betrachteten Objekt zur Verfügung gestellten Dienste als Eingabealphabet des Statecharts betrachten; die von ihm an anderen Objekten aufgerufenen Methoden bilden dann entsprechend das Ausgabealphabet. Um mit dem erstellten Statechart ein allgemeines, vom tatsächlichen Systemaufbau unabhängiges Verhaltensmuster festzulegen, werden in den Transitionsbeschriftungen ausschließlich formale Namen für die anderen Objekte verwendet. Bei einem späteren Systemlauf lassen sich die Namen der Objekte, mit denen tatsächlich kommuniziert wird, anhand des Configuration Diagrams ermitteln. Innerhalb eines Statecharts gilt, daß bei gleichen formalen Namen an verschiedenen Stellen dieselben Objekte angesprochen werden, wohingegen unterschiedliche formale Namen sowohl für verschiedene

[6] Zu den verschiedenen Begriffen bezüglich des „Verhaltens“ siehe Abschnitt 7.3.

als auch für dieselben Objekte stehen können.[7] Für die Zustandsübergänge in einem Statechart zur Klasse *Kesselsteuerung* wäre beispielsweise die nachfolgend zuerst aufgeführte Beschriftung nicht zulässig, die zweite hingegen schon.

infoGelesen/info.closeWindow *infoGelesen/W.closeWindow*

In einer während eines späteren Systemlaufs auftretenden Interaktion zwischen zwei Objekten sendet das sogenannte *Client-Objekt* einen Methodenaufruf an ein benanntes *Server-Objekt*, das in der Lage ist, die entsprechende Methode auszuführen. Es ist zulässig, einen Methodenaufruf an verschiedene Server-Objekte zu schicken, und ebenso kann ein Server-Objekt einen Methodenaufruf von unterschiedlichen Client-Objekten erhalten. Für den gesamten Kommunikationsvorgang wird jedoch vorausgesetzt, daß stets nur *eine* Interaktion zur Zeit stattfindet und diese beendet wird, bevor die nächste beginnt; geschachtelte Methodenaufrufe sind daher nicht möglich.

Neben den zustandsverändernden gibt es für Objekte weitere Methoden: die sogenannten *Beobachter* (*observer*), die den aktuellen Wert eines bestimmten Attributs zurückgeben, ohne diesen oder den Zustand des Objektes zu ändern.[8] Da es nicht zu jedem Zeitpunkt sinnvoll ist, derartige Abfragen vorzunehmen, führen die Autoren von [CHB92] folgende Notation ein: Ein Observer, bestehend aus Bezeichner und Typ des zugehörigen Attributs, wird in diejenigen Zustände des Statecharts eingetragen, in denen seine Verwendung gestattet ist. Befindet sich ein betrachtetes Exemplar der Klasse in einem Zustand, für den gilt, daß weder in ihm noch in einem ihm übergeordneten Zustand für ein bestimmtes Attribut ein Observer eingetragen ist, so bedeutet dies, daß es nicht sinnvoll ist, den Wert dieses Attributs abzufragen. Attribute, die in dem aktuellen Zustand einen sinnvollen Wert besitzen, für den jedoch gilt, daß er nicht über einen Observer zugänglich sein soll, werden wie ein Observer in den entsprechenden Zustand eingetragen und zusätzlich mit eckigen Klammern versehen.

Beiden Arten von Methoden ist gemein, daß sie mit Ein- und Ausgabeparametern versehen werden können, welche, in Klammern eingeschlossen, hinter dem jeweiligen Methodennamen aufgeführt sind; von einer Methode zurückgelieferte Parameter sind durch einen vorangestellten senkrechten Strich gekennzeichnet. Es ist gegebenenfalls sinnvoll, auf diese zusätzlichen Angaben im Statechart zu verzichten, um die Darstellung möglichst einfach und übersichtlich zu halten. Ein Informationsverlust tritt hierdurch nicht ein, da die in einem Statechart angegebenen notwendigen Ein- und Ausgabeparameter zusammen mit anderen Informationen auch in den in Abschnitt 7.2.2 beschriebenen Transitionsspezifikationen enthalten sind.[9]

Abb. 44 zeigt einen Statechart zu der Klasse *Kesselsteuerung*, deren Exemplar steuerung wesentlicher Bestandteil des zum Beispiel erstellten Configuration Diagrams (siehe

[7] In [CHB92] ist nicht ersichtlich, ob und in welcher Weise für einen formalen Namen festgelegt werden kann, welcher Klasse das später hierüber angesprochene Objekt angehören soll.

[8] Die durch diese Interpretation des Begriffs beschriebenen Methoden werden bei anderen Autoren u. a. als *sondierende Operationen* bezeichnet.

[9] Coleman et al. erläutern die Paramterangabe anhand des von ihnen verwendeten Beispiels, setzen sie in der grafischen Repräsentation jedoch ebenfalls nicht um.

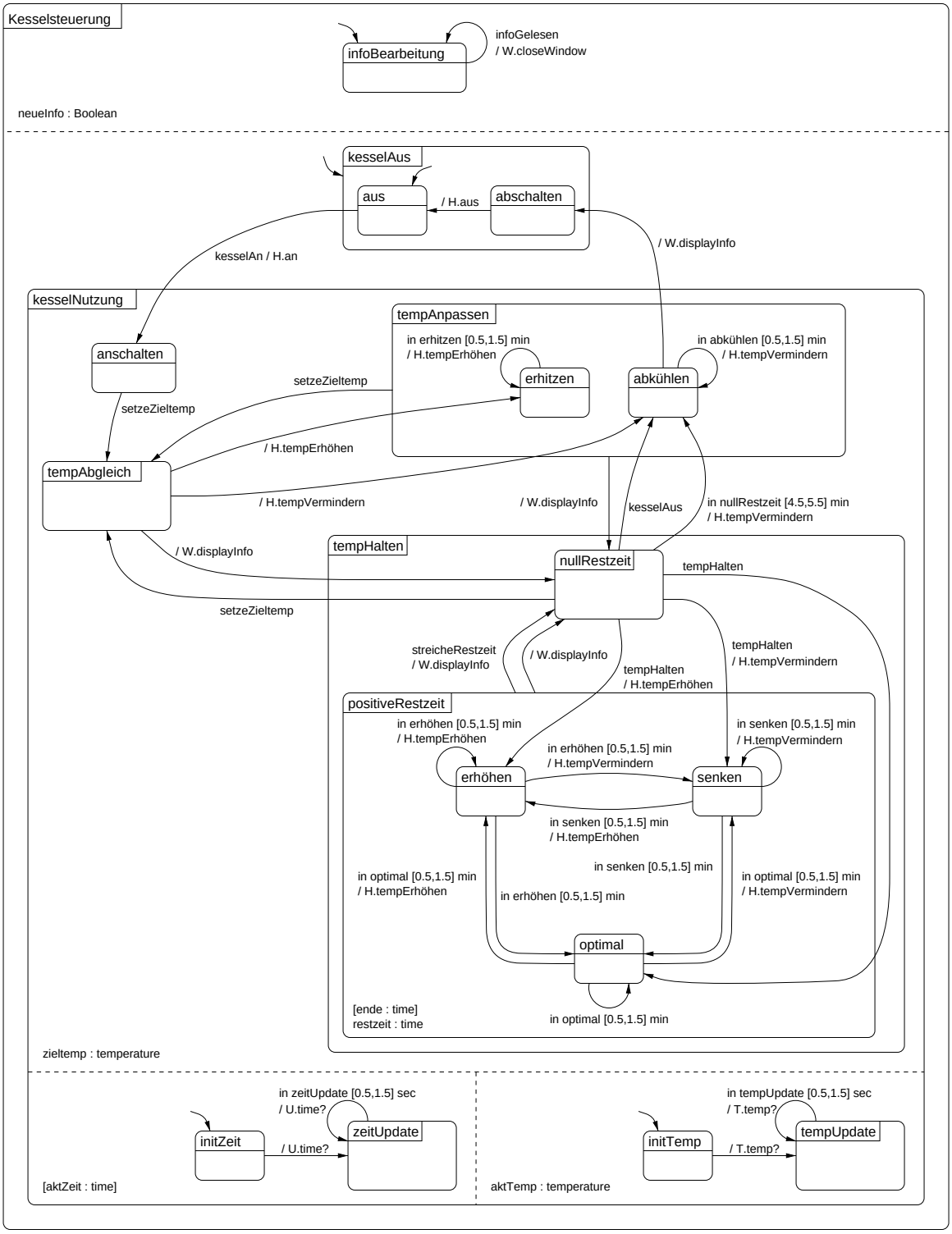


Abb. 44: Erweiterter Statechart für die Klasse *Kesselsteuerung*

S. 73) ist. Neben der im oberen Teil der Darstellung abgebildeten Komponente, die der Bearbeitung von Meldungen für den Benutzer dient, besitzt die AND-Dekomposition des Zustands *Kesselsteuerung* nur eine weitere Komponente, welche für die eigentliche Steuerung des Heizkessels zuständig ist. Sie besteht aus zwei sich ausschließenden Zuständen *kesselAus* und *kesselNutzung*, von denen letzterer detailliert den Heizbetrieb beschreibt und hierfür in drei zueinander orthogonale Subzustände aufgegliedert ist.

Die beiden am unteren Ende der Darstellung aufgeführten Zustände aktualisieren in regelmäßigen Abständen von ungefähr einer Sekunde die aktuelle Temperatur *aktTemp* entsprechend der im Heizkessel vom Thermometer gemessenen Temperatur sowie die aktuelle Zeit *aktZeit*, die u. a. zur Berechnung der noch abzuwartenden Restzeit genutzt wird. An ihnen lassen sich einige Punkte der Verwendung von Statecharts nach [CHB92] erkennen: Für das Attribut *aktTemp* steht ein Observer zur Verfügung, so daß sein Wert zugänglich ist, wenn sich das System im Zustand *kesselNutzung* befindet. Demgegenüber ist es nicht möglich, auf den Wert von *aktZeit* zuzugreifen, wenn der Zustand *kesselNutzung* eingenommen ist, da *aktZeit* ein sogenanntes verstecktes Attribut ist. In beiden Komponenten wird bei Zustandsübergängen auf andere Objekte zugegriffen, wie an den formalen Namen U und T erkennbar ist.

In dem letzten Subzustand von *kesselNutzung* stellt der Zustand *tempAbgleich* einen Übergangszustand dar, der eingenommen wird, wenn der Benutzer eine neue Zieltemperatur eingegeben hat, und anschließend sofort wieder verlassen werden kann, da die von ihm wegführenden Kanten nicht mit Methodenaufrufen verknüpft sind. Von ihm aus wird entsprechend des Wertes der neuen Zieltemperatur in einen der Zustände *erhitzen*, *abkühlen* oder *nullRestzeit* (Zieltemperatur ist gleich der aktuellen Temperatur) gewechselt. Der Zustand *tempAnpassen* mit seinen beiden Subzuständen dient dazu, die angestrebte Zieltemperatur durch Erhitzen oder Abkühlen zu erreichen, und kann verlassen werden, wenn eine neue Zieltemperatur festgelegt wird oder die aktuelle Temperatur der Zieltemperatur entspricht. Aufgabe des dritten Subzustandes dieser AND-Komponente von *kesselNutzung* ist es, die erreichte Zieltemperatur für eine angegebene Dauer zu halten, indem der Heizkessel bei Bedarf dazu veranlaßt wird, die Temperatur zu erhöhen oder zu senken. Nach Ablauf der Restzeit, für die die Temperatur gehalten werden sollte, erfolgt innerhalb dieses Zustandes ein Übergang zum Subzustand *nullRestzeit*. Dieser Teilzustand von *tempHalten* wird u. a. dann wieder verlassen, wenn eine neue Zieltemperatur gesetzt oder eine neue Haltedauer eingegeben wird; ein Übergang zum Zustand *abkühlen* wird dadurch ausgelöst, daß eine bestimmte Verweildauer (ca. 5 Minuten) überschritten oder die Methode *kesselAus* aufgerufen wird. Jeder Übergang zu *nullRestzeit* verursacht eine Meldung im Informationsfenster. Die Default-Zustände dieses Statecharts zur Klasse *Kesselsteuerung* sind *aus* und *infoBearbeitung*.

7.2.2 Transitionsspezifikationen

Mittels der oben erläuterten erweiterten Statecharts ist es möglich, einzelne Attribute von Exemplaren der zugehörigen Klassen bestimmten Zuständen, in denen der Zugriff

auf ihre Werte zulässig ist, zuzuordnen. Außerdem werden Zustandsübergänge mit Methoden, die den Zustand des betrachteten Objektes ändern können oder an anderen Objekten aufrufbar sind, beschriftet; der Aufruf dieser Methoden löst die entsprechenden Übergänge aus. In den so veränderten Statecharts wird jedoch weiterhin nicht die Wirkung der einzelnen Transitionen dargestellt, und es bleibt zudem unberücksichtigt, welche Bedingungen vor der Ausführung erfüllt sein müssen und welche Bedingungen gelten, nachdem der Zustandswechsel erfolgt ist. Zur Festlegung dieser Voraussetzungen und Effekte verwenden Coleman et al. sogenannte *Transitionsspezifikationen*.[10]

Jeder Transition eines Statecharts kann eine Transitionsspezifikation zugeordnet werden. In ihr wird festgehalten, unter welchen Voraussetzungen die Transition ausgeführt werden darf und welche Wirkung sie auf die Attribute des Objektes hat. Sie enthält den Namen des Start- und des Zielzustandes der beschriebenen Transition sowie die zugehörigen Methodennamen und deren Parameter. Des weiteren können in ihr Vor- und Nachbedingungen angegeben werden, um die Voraussetzungen und den Effekt der Transition zu beschreiben, so daß sich insgesamt folgende allgemeine Form für eine Transitionsspezifikation ergibt:

initialState → finalState: {*firingCondition*}
provService(parameters)/reqService(parameters)
{*postCondition*}

Es gilt, daß nicht in den Nachbedingungen aufgeführte Attribute und Observer durch die Ausführung der Transition unbeeinflußt bleiben. Die als Restriktion dienende *Vor-* oder auch *Feuerbedingung* einer Transition wird als Prädikatsausdruck über Observern und versteckten Attributen sowie über Zustandsnamen gebildet. Sie bezieht sich jedoch nicht auf Werte von Eingabeparametern, damit vermieden wird, daß die Interaktion zwischen zwei Objekten aufgrund einer Information, deren Übermittlung erst durch die Kommunikation erfolgt, abgelehnt werden kann. Prädikate über den ursprünglichen und den durch Ausführung der Transition erzeugten Werten von Attributen und Observern werden als Nachbedingung der Transition verwendet und charakterisieren zusammen mit der Vorbedingung den Effekt der Transition.

Das in diesem Abschnitt eingeführte Beispiel der Kesselsteuerung verfügt trotz seiner einfachen Struktur über eine Vielzahl von Transitionsspezifikationen. Eine Auswahl aus dieser Menge gibt Tafel 1 wieder. Mittels der ersten beiden Spezifikationen werden die Default-Pfeile zu den Zuständen *kesselAus* und *initTemp* beschrieben. Während nach dem Übergang zu *kesselAus* keine besonderen Bedingungen erfüllt sind, wie anhand der leeren Klammer dieser Transitionsspezifikation erkennbar ist, gilt nach dem Default-Eintritt in den Zustand *initTemp*, daß die aktuelle Temperatur Null ist. Vorbedingungen werden für Default-Pfeile nicht festgelegt, da die Nutzung eines solchen Übergangs grundsätzlich ohne Erfüllung bestimmter Voraussetzungen möglich ist. Die nächste Gruppe von Transitionsspezifikationen in Tafel 1 beschreibt Zustandswechsel,

[10] Auf die von Harel eingeführte Notation zur Pfeilbeschriftung, bei der zu erfüllende Vorbedingungen in Klammern hinter dem auslösenden Ereignis aufgeführt sind, wird in [CHB92] verzichtet.

	→ kesselAus:	$\{\}$
	→ initTemp:	$\{aktTemp = 0\}$
tempAbgleich	→ erhitzen:	$\{aktTemp < zieltemp\}$ /H.tempErhöhen $\{\}$
tempUpdate	→ tempUpdate:	$\{true\}$ /T.temp?(|t) $\{aktTemp = t\}$
abkühlen	→ abschalten:	$\{aktTemp = zieltemp \wedge zieltemp = 0\}$ /W.displayInfo(„Kessel wird abgeschaltet“) $\{neueInfo = true\}$
abschalten	→ aus:	$\{true\}$ /H.aus $\{\}$
nullRestzeit	→ erhöhen:	$\{aktTemp < zieltemp\}$ tempHalten(t)/H.tempErhöhen $\{ende = aktZeit + t\}$
anschalten	→ tempAbgleich:	$\{true\}$ setzeZieltemp(t) $\{zieltemp = t\}$

Tafel 1: Transitionsspezifikationen zum Statechart der Klasse *Kesselsteuerung*

die mit dem Aufruf einer Methode an einem anderen Objekt verbunden sind. Neben der Darstellungsform für übergebene und als Ergebnis erhaltene Parameter ist die Verwendung von Vor- und Nachbedingungen erkennbar; die im Statechart zur Klasse *Kesselsteuerung* angegebenen zeitlichen Beschränkungen (siehe S. 76) sind hier nicht erneut aufgeführt, könnten jedoch als zusätzliche Vorbedingungen angegeben werden. In den letzten beiden Spezifikationen wird veranschaulicht, wie Zustandsübergänge, die durch Methodenaufrufe an dem betrachteten Objekt ausgelöst werden, darzustellen sind, so daß insgesamt ein Großteil der denkbaren Formen von Transitionsspezifikationen in Tafel 1 aufgeführt ist.

Manche Attribute und Observer sind nur in bestimmten Zuständen verwendbar, weswegen es erforderlich ist sicherzustellen, daß immer, wenn sie in einer zu prüfenden Vorbedingung vorkommen, der Zugriff auf sie auch möglich, d. h. sinnvoll, ist. Beispielsweise ist die Abfrage des Wertes von *restzeit* eines Exemplars der Klasse *Kesselsteuerung* nur dann ratsam, wenn eine Zeitspanne vorgegeben wurde, für die die Temperatur im Kessel gehalten werden soll. Somit ist bei der Formulierung von Vorbedingungen, die sich auf *restzeit* beziehen, zu berücksichtigen, daß sich das Objekt im Zustand *positiveRestzeit* befinden muß. Die Autoren von [CHB92] schlagen für die Bearbeitung solcher Abhängigkeiten vor, die Logik partieller Funktionen von VDM zu verwenden.

7.2.3 Invariantenspezifikationen

Ein Objectchart umfaßt neben einem um Attribute und Observer erweiterten Statechart und den Spezifikationen der in diesem enthaltenen Transitionen als dritte Komponente eine Menge sogenannter *Invariantenspezifikationen* für die Attribute der beschriebenen Klasse. Zweck dieser Form von Spezifikationen ist es, zwischen Attributen und Observern einer Klasse bestehende Zusammenhänge festzuhalten.

Ein Beispiel für eine derartige Beziehung stellen das Attribut *ende* und der Observer *restzeit* im Zustand *positiveRestzeit* des Statecharts zur Kesselsteuerung dar: Der Wert von *restzeit* ergibt sich aus der Differenz von *ende* und der aktuellen Zeit *aktZeit* und ist aus diesem Grund redundant. Bestehen zwischen einzelnen Attributen und Observern einer Klasse Abhängigkeiten dieser Art, so werden sie als Invariantenspezifikationen folgendermaßen formuliert:

state: {*relation that holds in* state}

Eine Invariantenspezifikation setzt sich somit aus einem Zustandsnamen sowie der in dem genannten Zustand gültigen Beziehung zwischen abgeleitetem Attribut oder Observer und anderen Attributen und Observern des Objektes zusammen. Coleman et al. geben in [CHB92] ein Beispiel für eine Invariantenspezifikation, an dem jedoch nicht erkennbar ist, ob abgeleitete Attribute und Observer nur in den Invariantenspezifikationen aufgeführt oder auch im Statechart zur betrachteten Klasse eingetragen werden sollen. In dem hier angeführten Beispiel der Kesselsteuerung sind diese redundanten Informationen zusätzlich im Statechart angegeben, um einen Blick auf alle vorhandenen Attribute zu geben. Für die Klasse *Kesselsteuerung* stellt *restzeit* den einzigen ableitbaren Observer dar; die Invariantenspezifikation für ihn kann folgendermaßen formuliert werden:

positiveRestzeit: {*restzeit* = *ende* - *aktZeit*}

Unter Nutzung dieser drei Bestandteile eines Objectcharts – erweiterter Statechart, Transitions- sowie Invariantenspezifikationen – ist es nach Ansicht der Autoren von [CHB92] möglich, formale Klassenbeschreibungen zu erstellen, in denen Charakteristika der Objektorientierung berücksichtigt werden. Allerdings fehle noch eine Beschreibung für die Erzeugung und Zerstörung von Objekten sowie eine feste Semantik, so Coleman et al.

7.2.4 Vererbungsbeziehungen zwischen Objectcharts

In den vorangegangenen Teilabschnitten ist der grundsätzliche Aufbau von Objectcharts zu einzelnen Klassen erläutert worden. Zwischen verschiedenen Klassen eines zu beschreibenden Systems können jedoch zusätzlich Vererbungsbeziehungen bestehen, durch die Generalisierungs- oder Spezialisierungszusammenhänge dargestellt werden. Es ist denkbar, diese Beziehungen hinsichtlich des in den Objectcharts zu den einzelnen Klassen definierten Verhaltens zu untersuchen und u. a. zu überprüfen, ob eine Klasse zu ihrer Oberklasse verhaltenskonform ist.

Erbt eine Klasse die Beschreibung einer anderen Klasse, so stehen im wesentlichen zwei Wege zur Verfügung, um das Verhalten der abgeleiteten Klasse zu ändern: Zum einen ist es möglich, neue Methoden hinzuzufügen, damit zusätzliche Dienste in der neuen Klasse bereitgestellt werden, zum anderen kann durch Redefinition bestehender Methoden der Oberklasse eine Spezialisierung erfolgen. In den zu den einzelnen Klassen erstellten Objectcharts, mittels derer das Verhalten der entsprechenden Klassen

definiert ist, können Vererbungsbeziehungen berücksichtigt werden, indem der Objectchart zu einer Oberklasse als Ausgangspunkt für die Verhaltensbeschreibungen der abgeleiteten Klassen verwendet wird.

Zur gewünschten Anpassung der Verhaltensbeschreibung für eine Subklasse sehen die Autoren von [CHB92] folgende Änderungsmöglichkeiten an dem von der Oberklasse übernommenen Objectchart vor:

- Dem Statechart des Objectcharts wird eine neue Transition hinzugefügt, die einer zusätzlich angebotenen Dienstleistung entspricht. Statt einer Transition kann der Statechart auch um einen Zustand mit der gleichen Eigenschaft ergänzt werden. Neben diesen Ergänzungen, die für neu angebotene Methoden stehen, ist die Angabe zusätzlicher Attribute möglich.

- Eine Transitionsspezifikation kann geändert werden, indem die zu erfüllende Vorbedingung abgeschwächt oder die nach der Ausführung gültige Nachbedingung verschärft wird. Diese Einschränkung hinsichtlich der Änderbarkeit von Transitionsspezifikationen gewährleistet, daß die modifizierte Transition in mindestens genausovielen Situationen wie die ursprüngliche aktiviert ist und außerdem anschließend auf jeden Fall die gleichen Nachbedingungen erfüllt sind.

- Die gültigen Invariantenspezifikationen können, ähnlich dem Prozeß bei den Transitionsspezifikationen, verschärft werden.

Derartige Änderungen im Objectchart führen zu einer Modifikation des Verhaltens der beschriebenen Klasse. Es ist gewährleistet, daß in den Exemplaren der Subklasse alle Zustandsübergänge der ursprünglichen Klasse erlaubt sind und in ihnen nach Ausführung einer aktivierten Transition wenigstens die gleichen Bedingungen wie in Exemplaren der Oberklasse erfüllt sind. Nicht sichergestellt werden kann hingegen, daß in einer auf diese Weise veränderten Subklasse die gleichen Folgen von Transitionsübergängen wie in der Oberklasse zulässig sind.

Eine abgeleitete Klasse stellt nach [CHB92] einen Subtyp der Oberklasse dar, wenn jedes Exemplar von ihr überall dort eingesetzt werden kann, wo ein Objekt der Oberklasse verwendbar ist, d. h. wenn sie verhaltenskonform zur Oberklasse ist. Hierfür ist es erforderlich, daß die Semantik der einzelnen Methoden nicht durch Redefinition verändert wird und die Spezifikation der Oberklasse auch für die Subklasse gilt, somit das Klassenverhalten[11] der Oberklasse in dem der abgeleiteten Klasse enthalten ist. Dies kann mit den oben angegebenen Modifikationsmöglichkeiten nicht gewährleistet werden, wie Coleman et al. an einem einfachen Beispiel demonstrieren; das semantische Modell der Objectcharts müßte hierfür geeignet erweitert werden.[12]

[11] Zu dem Begriff des Klassenverhaltens siehe Abschnitt 7.3.

[12] In der Erstellung einer formalen semantischen Grundlage sehen die Autoren von [CHB92] einen Schwerpunkt ihrer Arbeit.

7.3 Das Verhalten eines Systems

Aus den in den Abschnitten 7.1 und 7.2 erläuterten Komponenten einer Systembeschreibung – Configuration Diagram und Objectcharts – läßt sich, so die Autoren von [CHB92], das Verhalten des zugehörigen Systems ermitteln, indem man zunächst das durch die Objectcharts festgelegte Klassenverhalten bestimmt und aus diesem anschließend mit Hilfe des erstellten Configuration Diagrams das sogenannte Objektverhalten ermittelt. Als letzter Schritt wird für alle denkbaren Alternativen des Systemverhaltens geprüft, ob sie mit dem möglichen Verhalten der einzelnen Objekte des Systems vereinbar sind.

Von grundlegender Bedeutung für die Beschreibung des Systemverhaltens ist die Form der Objekt-Interaktion. Die hier dargestellte Art der Spezifikation basiert auf folgenden Annahmen über die Kommunikation zwischen den Objekten eines Systems:

- Die Kommunikation erfolgt grundsätzlich über direkte Anforderungen von Dienstleistungen, die einzelne Objekte zur Verfügung stellen (z. B. durch den Aufruf von Methoden), und nicht über den von Harel eingeführten allgemeinen Broadcast-Mechanismus, so daß stets genau zwei Objekte an der Interaktion beteiligt sind.

- Interagieren zwei Objekte miteinander, so wird der Anbieter des angeforderten Dienstes vom anfragenden Objekt direkt benannt. Für den Diensterbringer gilt hingegen, daß ihm der Name seines Klienten nicht bekannt ist.

- Ein Objekt hat die Möglichkeit, Methodenaufrufe an verschiedene Objekte zu senden und ebenso von verschiedenen Objekten des Systems zu erhalten, so daß jede Dienstanfrage als Tripel $\langle \mathrm{r}, s, \mathrm{p} \rangle$ geeignet notiert werden kann. In diesem Tripel steht r für den Namen desjenigen Objektes, das den Methodenaufruf generiert, und p für den des Diensterbringers; s beschreibt die Methode sowie die zugehörigen Ein- und Ausgabeparameter.

- Es ist stets nur eine Interaktion zur Zeit möglich. Zudem stellen Aufruf und Ausführung einer Methode eine atomare Handlung dar und es gilt, daß jede aufgerufene Methode beendet wird, bevor die nächste Interaktion in Form einer weiteren Dienstanfrage beginnt.

Aufgrund der im letzten Punkt erwähnten Serialität der Interaktionen[13] läßt sich ein bestimmtes Systemverhalten durch eine Folge der oben beschriebenen Tripel ausdrücken. Somit stellen die denkbaren Folgen von Methodenaufrufen, die mit dem möglichen Verhalten aller Objekte des System in Einklang stehen, die Verhaltensalternativen des Systems dar.

Im ersten Schritt zur Ermittlung des Verhaltens eines Systems wird das *Klassenverhalten*, das als allgemeines Interaktionsmuster für die Exemplare der jeweiligen Klasse

[13] Verschachtelte Methodenaufrufe sind mit diesem Ansatz nicht beschreibbar.

dient, bestimmt. Hierfür wird jeder in den Objectcharts der einzelnen Klassen vorkommende Methodenaufruf s durch ein Tripel $\langle \mathrm{r}, s, \mathrm{p} \rangle$ ausgedrückt. Die an einem derartigen Methodenaufruf beteiligten Objekte sind in diesem Tripel in folgender Darstellungsform berücksichtigt: Das betrachtete Exemplar der Klasse selbst wird mit *self* bezeichnet, und für andere Objekte, mit denen später im System interagiert wird, verwendet man die im Objectchart eingeführten formalen Namen, sofern die Methode an diesen Objekten ausgeführt werden soll, und das Zeichen „$*$“, wenn die Methode an dem betrachteten Objekt selbst aufgerufen wird.[14] Tafel 2 zeigt einige Tripel für den Objectchart von Seite 76. Die beiden zuunterst aufgeführten Tripel verdeut-

$\langle \mathrm{self}, \mathit{aus}, \mathrm{H} \rangle$	$\langle \mathrm{self}, \mathit{temp?}(\lvert t), \mathrm{T} \rangle$
$\langle \mathrm{self}, \mathit{displayInfo}(\text{„}\mathit{Kessel\ wird\ abgeschaltet}\text{“}), \mathrm{W} \rangle$	$\langle *, \mathit{setzeZieltemp}(t), \mathrm{self} \rangle$
$\langle *, \mathit{tempHalten}(t), \mathrm{self} \rangle$	$\langle \mathrm{self}, \mathit{tempErhöhen}, \mathrm{H} \rangle$

Tafel 2: Einige Tripel zu den Transitionen im Objectchart der Klasse *Kesselsteuerung*

lichen, in welcher Weise bei Coleman et al. Transitionen behandelt werden, die sowohl angebotene als auch an einem anderen Objekt aufgerufene Methoden umfassen: $\langle *, \mathit{tempHalten}(t), \mathrm{self} \rangle$ und $\langle \mathrm{self}, \mathit{tempErhöhen}, \mathrm{H} \rangle$ stellen zusammen die Interaktion dar, die bei einem Zustandsübergang von *nullRestzeit* nach *erhöhen* erfolgt und durch die Transitionsbeschriftung *tempHalten/H.tempErhöhen* ausgedrückt wird. Das Klassenverhalten einer Klasse X, abkürzend als $T_c(X)$ dargestellt, setzt sich aus allen unter Berücksichtigung des Objectcharts zur betrachteten Klasse zulässigen Sequenzen von Methodenaufrufen zusammen; die möglichen Sequenzen werden als Folgen von Tripeln (*Traces*) notiert.

Die in $T_c(X)$ enthaltenen Sequenzen stellen die grundsätzlich denkbaren Interaktionen der Exemplare von X dar, berücksichtigen jedoch nicht die tatsächlich vorhandene Systemstruktur. Erst mit der Bestimmung des *Objektverhaltens* wird auf die im zugrundeliegenden System vorhandenen Objekte eingegangen, indem man die formalen Namen in den Folgen durch die zur Verfügung stehenden und aus dem Configuration Diagram ersichtlichen aktuellen Objektnamen ersetzt (Tafel 3 zeigt einige Beispiele). Als Ergebnis erhält man für jedes Objekt O des Configuration Diagrams die Men-

$\langle \mathrm{self}, \mathit{aus}, \mathrm{kessel} \rangle$	$\langle \mathrm{self}, \mathit{temp?}(\lvert t), \mathrm{therm} \rangle$
$\langle \mathrm{self}, \mathit{displayInfo}(\text{„}\mathit{Kessel\ wird\ abgeschaltet}\text{“}), \mathrm{info} \rangle$	$\langle *, \mathit{setzeZieltemp}(t), \mathrm{self} \rangle$
$\langle *, \mathit{tempHalten}(t), \mathrm{self} \rangle$	$\langle \mathrm{self}, \mathit{tempErhöhen}, \mathrm{kessel} \rangle$

Tafel 3: Die Tripel aus Tafel 2 unter Berücksichtigung des Configuration Diagrams (Objektverhalten)

[14] Das Zeichen „$*$“ wird eingesetzt, weil das betrachtete Objekt keine Informationen über Interaktionspartner, die Methoden an ihm aufrufen, besitzt.

ge $T_o(O)$ derjenigen Interaktionssequenzen, die für dieses Objekt in dem betrachteten System möglich sind.

Um letztendlich das *Systemverhalten*, das vom Verhalten aller Objekte des Systems abhängt, zu bestimmen, werden alle Sequenzen tr von Methodenaufrufen ermittelt, für die gilt, daß sie aus der „Sicht" jedes einzelnen Objektes des Systems möglich sind. Wird eine beliebige Interaktionsfolge dahingehend überprüft, ob sie ein denkbares Systemverhalten beschreibt, so ermittelt man zunächst für jedes Objekt O denjenigen Teil der Sequenz, der von Bedeutung ist, d. h. die Teilfolge von Methodenaufrufen, an denen O als ausführendes oder aufrufendes Objekt beteiligt ist. Diese Teilfolge bezeichnen Coleman et al. mit $tr \triangleright O$. Anschließend wird hieraus ein sogenannter *localized trace* erzeugt, indem jedes Vorkommen von O durch *self* und jeder Klient von O durch $*$ ersetzt wird. Liegt die auf diese Weise ermittelte Sequenz von Methodenaufrufen in $T_o(O)$, so ist die Ausgangsfolge des Systems aus der Sicht des Objektes O möglich. Zum Systemverhalten $T_s(S)$ zählen alle Folgen, die für jedes Objekt des Systems $S = \{O_1, \ldots, O_n\}$ möglich sind, also

$$T_s(S) = \{tr \mid \forall o \in S : \mathit{local}(tr \triangleright o) \in T_o(o)\}.$$

Coleman et al. schlagen in [CHB92] folgende Schritte vor, um systematisch eine auf Objectcharts basierende Systembeschreibung zu entwickeln:

1. Ein Configuration Diagram für das zu beschreibende System erstellen:
   (a) Zu jeder Klasse das required und das provided interface bestimmen
   (b) Die im System vorhandenen Objekte festlegen sowie ihre angebotenen und benötigten Methoden geeignet miteinander verbinden
2. Einen Statechart pro Klasse angeben:
   (a) Zustandsverändernde Methoden und sondierende Operationen der Klasse identifizieren
   (b) Den Statechart, in dem die Wirkungen der zustandsverändernden Methoden und der von anderen Objekten benötigten Methoden auf den Zustand eines Exemplars der Klasse beschrieben sind, entwickeln
3. Die erstellten Statecharts zu Objectcharts erweitern:
   (a) Den einzelnen Zuständen Attribute und Observer zuordnen
   (b) Auf Zuständen sowie Attribut- und Observer-Werten basierende Vor- und Nachbedingungen für die Transitionen des erweiterten Statecharts definieren
   (c) Invariantenspezifikationen für ableitbare Attribute und Observer formulieren
4. Den Vererbungsbaum für die Objectcharts erstellen

Mit der hier beschriebenen Variation von Statecharts steht ein interessanter Ansatz für die Verwendung im objektorientierten Kontext zur Verfügung. Nach Aussage der Autoren von [CHB92] sind die Ergebnisse einer Anzahl von Fallstudien sowie einiger größerer realitätsnaher Beispiele wie der Simulation einer Telekommunikationsanwendung vielversprechend und lassen vermuten, daß Objectcharts ein geeignetes Mittel zur Systembeschreibung darstellen. Trotz dieser ersten Erfolge ist zu berücksichtigen, daß der bisher von Coleman et al. vorgeschlagene Ansatz für die Verwendung von Objectcharts keine Möglichkeit bietet, sich dynamisch in ihrem Aufbau verändernde Systeme zu beschreiben. Des weiteren ist es als problematisch anzusehen, daß diese Form keine verschachtelten Methodenaufrufe gestattet, sondern diese explizit durch entsprechend viele zusätzliche Zustände im Objectchart formuliert werden müssen.

8 Objektorientierte Statecharts nach Harel

Der dritte hier erläuterte Ansatz für die Verwendung von Statecharts in der objektorientierten Anwendungsentwicklung wird von David Harel und Eran Gery in ihrem 1996 erschienenen Artikel [HG96] vorgestellt. Während die ursprüngliche, vor dem Hintergrund der Strukturierten Analyse entwickelte Variante der Statecharts entworfen wurde, um eine isolierte Verhaltensdefinition für ein System zu ermöglichen, sind die objektorientierten Statecharts als Beschreibungsmittel für die dynamische Komponente eines allgemeinen Ansatzes zur Systemmodellierung zu betrachten. Damit Differenzen zwischen den einzelnen Aspekten einer solchen Systembeschreibung aufgezeigt werden können, sollte der gewählte Ansatz detailliert und präzise genug sein, um eine Modellausführung und Analyse der Dynamik zu ermöglichen, so die Autoren in [HG96].[1]

Als Basis für die Modellierung verwenden sie, ähnlich anderen Ansätzen (siehe z. B. [RBP+91], [SM92]), mehrere diagrammbasierte Sprachen, die sie als intuitiv verwendbar und gut strukturiert ansehen. Zur Beschreibung der Systemstruktur werden an die Objektmodelle von OMT angelehnte Diagramme, sogenannte *O-Charts*, eingesetzt, in denen die Klassen der im System vorkommenden Objekte sowie die Beziehungen zwischen ihnen dargestellt sind. Mittels objektorientierter Statecharts wird das Verhalten des zu modellierenden Systems festgehalten; hierfür können alle in [Har87] erläuterten Möglichkeiten sowie einige Erweiterungen (z. B. für die Erzeugung/Zerstörung von Objekten) genutzt werden. Die Interaktion zwischen den Objekten des Systems erfolgt sowohl über die Generierung und Registrierung von Ereignissen als auch durch direkte Operationsaufrufe.

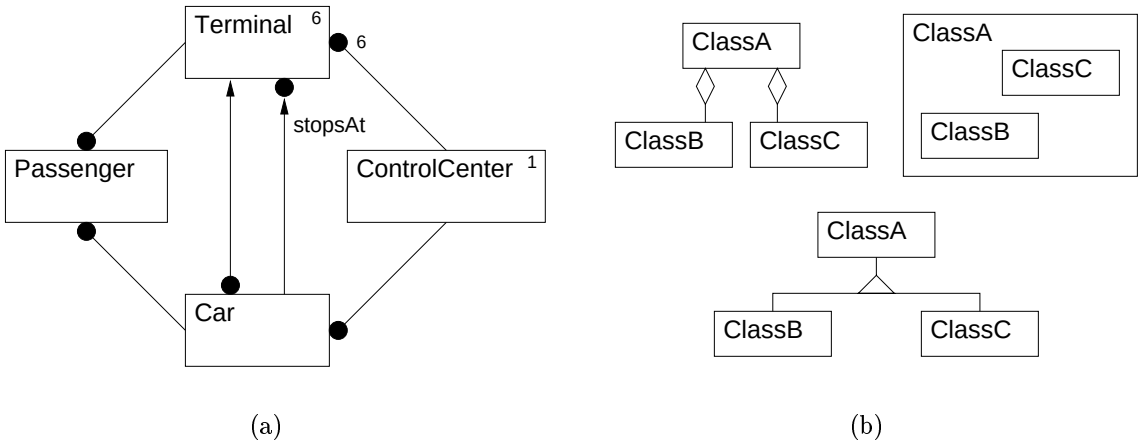
8.1 O-Charts zur Beschreibung der Systemstruktur

Harel und Gery orientieren sich mit ihrem Darstellungsmittel für die Struktur eines Systems, dem *O-Chart*, weitgehend an dem in OMT verwendeten Objektmodell, da diese Beschreibungsform zum Erscheinungszeitpunkt ihres Artikels bereits eine breite Akzeptanz gefunden hat. Eine zunächst erwogene Trennung von Klassen- und Objektbeschreibungen zur flexibleren Spezifikation nichttrivialer Verhältnisse (z. B. Anzahl von Objekten, Objekterzeugung, Referenzen) wurde verworfen, da eine gemeinsame Darstellung kompakter und leichter verständlich schien. Neben den Klassen der im System vorhandenen Objekte selbst umfaßt ein O-Chart auch Angaben zur Anzahl der Exemplare dieser Klassen sowie zu den strukturellen Beziehungen zwischen ihnen.

Ein O-Chart ist ein Entity-Relationship-ähnliches Diagramm, in dem Rechtecke die verschiedenen Klassen des Systems symbolisieren und Beziehungen zwischen Klassen durch gerichtete Kanten ausgedrückt werden. Abb. 45(a) zeigt ein aus [HG96] entnommenes Beispiel zu einem schienengebundenen Beförderungsnetz, in dem eine Reihe

[1] Ein weiteres Ziel von Harel und Gery ist die automatische Codeerzeugung aus den erstellten Diagrammen, die für die Sprache C++ in dem Produkt RHAPSODY realisiert ist.

von Wagen (*Car*) Personen (*Passenger*) zu verschiedenen Stationen (*Terminal*) transportieren; die Beförderung wird durch ein Kontrollzentrum (*ControlCenter*) gesteuert. Für jede Klasse kann die Anzahl ihrer Exemplare im System festgelegt werden; das



(a) (b)

Abb. 45: Ein O-Chart zum Beförderungsnetz in (a) sowie die Darstellung von Aggregation, Komposition und Vererbung in (b)

Beförderungsnetz verfügt über ein ControlCenter und sechs Terminals. Von Klassen ohne eine solche Angabe, z. B. *Passenger*, können beliebig viele Objekte existieren. Beziehungen zwischen Klassen werden durch gerichtete Kanten, die mit dem Namen der Beziehung und Rollenbezeichnungen beschriftet werden können, ausgedrückt. Ungerichtete Kanten stehen als abkürzende Schreibweise für bidirektionale Beziehungen zur Verfügung. Es gibt zwei Möglichkeiten, Beziehungspartner anzusprechen: zum einen über den Beziehungs- oder Rollennamen, zum anderen mittels *its*<*ClassName*>, wenn eine Beziehung nicht benannt ist. Ein Exemplar der Klasse *Car* greift somit mittels *itsControlCenter* auf das Kontrollzentrum zu. Durch Verknüpfung derartiger Bezeichnungen entstehen sogenannte *Navigationsausdrücke*, mit denen ein Objekt auf andere, über mehrere Stufen mit ihm verbundene Objekte zugreifen kann. Harel und Gery führen ein besonderes Objekt *System* ein, welches das gesamte Modell als Komposition umfaßt und von dem aus über Navigationsausdrücke alle Objekte des Systems ansprechbar sind. Die in einem O-Chart aufgeführten Beziehungen können um Angaben zur zulässigen Anzahl von Exemplaren der beteiligten Klassen ergänzt werden, wodurch eine Differenzierung in folgende drei Gruppen möglich wird:

- *unambigious:*
  Für Beziehungen dieser Art gilt, daß sie unzweideutig durch die Angaben zur Objektanzahl der beteiligten Klassen und zur Relation festgelegt sind. Ein Beispiel ist die Beziehung zwischen den Klassen *ControlCenter* und *Terminal*.

- *ambigious but bounded:*
  Beziehungen dieser Gruppe zeichnen sich dadurch aus, daß mehrere Ausprägungen möglich sind, die Anzahl der Varianten jedoch begrenzt ist. Wäre beispiels-

weise die Kante zwischen *ControlCenter* und *Terminal* nicht mit einer Zahl versehen, so könnten dem *ControlCenter* zwar verschieden viele, jedoch maximal sechs Terminals zugeordnet sein.

- *unworkable:*
  Zur letzten Gruppe von Beziehungen ist es nicht möglich, genauere Informationen zu gewinnen, da keine genügend einschränkenden Angaben vorliegen. Hierzu zählt z. B. die nicht näher benannte Beziehung zwischen den Klassen *Car* und *Terminal*, da beliebig viele Exemplare von *Car* im System vorhanden sein können.

In Abb. 45(b) sind die zusätzlich zur Verfügung stehenden, ebenfalls aus OMT übernommenen Symbole für Aggregation und Vererbung dargestellt. Zudem ist die grafische Einschließung als Notation für die Komposition aufgeführt; eine auf diese Weise gebildete zusammengesetzte Klasse (*composite class*) kann direkt auf ihre Komponenten zugreifen, ohne den Zusatz *its* zu benötigen.

8.2 Interaktionsmechanismen

Bevor auf die Eigenschaften objektorientierter Statecharts als Beschreibungsmittel für das Verhalten der in einem System vorhandenen Objekte eingegangen wird, sollen zunächst die von Harel und Gery zur Interaktion zwischen Objekten vorgesehenen Mechanismen betrachtet werden. Im Gegensatz zu anderen Autoren verwenden sie sowohl Ereignisse als auch Operationsaufrufe für die Kommunikation zwischen Objekten und erlauben Interaktionen nicht nur durch direkte Kommunikation, sondern auch mittels des in [Har87] eingeführten Broadcast-Mechanismus. Neben Operationen auch Ereignisse zu verwenden, halten die Autoren von [HG96] für ratsam, weil nach ihrer Erfahrung Ereignisse eher in der Analyse Verwendung finden, während Operationen näher am Design orientiert sind; zunächst als Ereignisse eingeführte Vorkommnisse können in späteren Schritten durchaus durch Operationen ersetzt werden. Ein wesentlicher Unterschied zwischen Ereignissen und Operationen liegt in der Art ihrer Bearbeitung, wie sich im folgenden zeigen wird.

8.2.1 Ereignisse

Jedes Objekt im System hat die Möglichkeit, Ereignisse zu generieren und diese an andere, von ihm addressierbare Objekte zu senden. Der Adressat kann vom Sender eines Ereignisses über zulässige Navigationsausdrücke (siehe S. 88) oder, wenn es sich um eine Komponente des Senders (Aggregation, Komposition) handelt, direkt mit Namen angesprochen werden. Im Statechart zur zugehörigen Klasse wird die Generierung eines solchen Ereignisses mittels

$$objectname \rightarrow \mathsf{gen}\,(eventname\,(parameters))$$

formuliert; steht an der Stelle des Zielobjektes die Bezeichnung *this* oder ist kein Objektname angegeben, so entspricht dies dem Broadcast-Mechanismus aus [Har87] und

das Ereignis wird mit dem generierenden Objekt als Ziel erzeugt. Das ausgelöste Ereignis wird in eine systemweite Warteschlange eingereiht und erst dann an das Zielobjekt weitergeleitet, wenn alle vorher generierten Ereignisse bearbeitet worden sind. Damit ein Objekt auf ein solches Ereignis reagieren kann, ist folgende Beschriftung als Auslöser mindestens einer Transition[2] im Statechart erforderlich:

eventname (*parameters*)

Zwei interessante Aspekte gibt es hinsichtlich der Generierung von Ereignissen und der Reaktion auf sie. Zum einen ist festzuhalten, daß der Kontrollfluß auch dann bei einem Objekt verbleibt, wenn es ein Ereignis generiert. Erst nachdem alle möglichen Zustandsübergänge im Statechart des Objektes ausgeführt worden sind und das Objekt somit eine stabile Situation (die nächste Zustandskonfiguration) erreicht hat, geht der Kontrollfluß an das oberste Objekt *System* über, welches das nächste Element der Ereigniswarteschlange bearbeitet und an seinen Adressaten weiterleitet. Das zweite beachtenswerte Detail ist in der Möglichkeit zu sehen, daß mittels Komposition zusammengesetzte Klassen die Ereignisse, die sie erhalten, an ihre Komponenten weiterleiten können. In Abschnitt 8.3.1 wird näher auf dieses Mittel der *Ereignis-Delegation* eingegangen, das die Ereignisübermittlung in allen Varianten, von direkter Objekt-Objekt-Kommunikation bis zum vollständigen Broadcast, gestattet.

Für die Autoren von [HG96] stellen Ereignisse bestimmte Einheiten innerhalb des Modells eines Systems dar, die untereinander in einer Generalisierungs-/Spezialisierungs-Hierarchie angeordnet werden können. Als Konsequenz reagiert ein Objekt nicht nur auf das Ereignis, das in seinem Statechart eingetragen ist, sondern auch auf alle Subereignisse dieses Ereignisses.

8.2.2 Operationen

Zusätzlich zu der Generierung von Ereignissen steht mit der Möglichkeit von Operationsaufrufen eine weitere Interaktionsform zur Verfügung, die von Harel und Gery näher am Design angesiedelt wird. Diese Art der Kommunikation gestattet einem Objekt, direkt eine Operation an einem anderen, benannten Objekt aufzurufen, woraufhin das Zielobjekt die entsprechende Methode ausführt und gegebenenfalls Rückgabewerte an das aufrufende Objekt liefert. Der Aufruf einer Operation bildet die konkretere Form der Interaktion zwischen Objekten und wird im Statechart durch

objectname $\rightarrow$ *operationname* (*parameters*)

ausgedrückt; im Gegensatz zur Ereignisgenerierung ist hier der Objektname zwingend erforderlich. Innerhalb des Statecharts zum Zielobjekt beschreibt

operationname (*parameters*)

die Reaktion auf den Operationsaufruf, d. h. die Ausführung der zugehörigen Methode. Eine solche Ausführung terminiert, wenn eine Wertrückgabe erfolgt oder das Objekt

[2] oder einer statischen Reaktion (siehe S. 42)

eine stabile Situation erreicht. Wegen des zweiten Punktes ist es erforderlich, daß die Ausführung der gerufenen Methode in ihrer Gesamtheit von *einer* Transition angeboten wird, da anderenfalls die Methodenausführung stets frühzeitig beendet wird, sobald das Objekt die nächste Zustandskonfiguration erreicht hat.

Bei einem Operationsaufruf geht, anders als bei der Generierung eines Ereignisses, der Kontrollfluß gleich zum gerufenen Objekt über und verbleibt dort bis zur Beendigung der sofort beginnenden Methodenausführung. Für den Transitionsübergang des rufenden Objektes bedeutet dies, daß er für die Dauer der Methodenausführung „eingefroren“ und erst nach ihrer Beendigung fortgeführt wird. Da die Methodenausführung im Zielobjekt selbst wieder einen Methodenaufruf an einem anderen Objekt mit sich bringen kann, ist bei diesem Ansatz eine Verschachtelung von Aufrufen möglich. Nicht vorgesehen für Operationen ist hingegen ein Delegations-Mechanismus, wie ihn die Autoren von [HG96] für Ereignisse eingeführt haben, so daß die Wahl der Kommunikationsform Einfluß auf den Grad der Weiterleitung hat.

8.3 Objektorientierte Statecharts

Ergänzend zum O-Chart, der die Struktur des zu modellierenden Systems beschreibt und eher statische Aspekte anspricht, kann für jede im System vorkommende Klasse ein objektorientierter Statechart erstellt werden, durch den das Verhalten der Exemplare dieser Klasse spezifiziert wird.[3] Ein solcher Statechart dient nicht nur der Aufbewahrung des Objektzustandes im Sinne der Bereitschaft, auf Ereignisse oder Operationsaufrufe zu reagieren, sondern beinhaltet zusätzlich Informationen über die Dynamik des internen Verhaltens, wenn das Objekt auf derartige Vorkommnisse antwortet. Außerdem werden Beziehungen zu anderen Objekten entsprechend der in ihm enthaltenen Vorgaben gepflegt. Objektorientierte Statecharts umfassen im wesentlichen alle in [Har87] beschriebenen Eigenschaften des Formalismus, einschließlich Zustandshierarchien, orthogonale Komponenten und die (interne) Broadcast-Kommunikation. Darüber hinaus existieren einige Erweiterungen, z. B. zur Erzeugung und Zerstörung von Objekten sowie für die Verwaltung der Beziehungen zu anderen Objekten.

Für die Beschriftung der Zustandsübergänge wird weiterhin die aus der ursprünglichen Form des Statechart-Formalismus bekannte Form

$$trigger\,[condition]/action$$

verwendet. Der Auslöser *trigger* kann jetzt jedoch sowohl aus einem ankommenden Ereignis als auch aus einem Operationsaufruf bestehen und die Aktions-Sequenz *action* entsprechend aus Ereignisgenerierungen und Operationsaufrufen zusammengesetzt sein.

Hinsichtlich der verwendeten Semantik orientieren sich Harel und Gery bei ihrem Ansatz an STATEMATE, so daß generierte Ereignisse erst im nachfolgenden Schritt registriert werden. Verzichtet wird allerdings auf simultane Ereignisse, da die in der

[3] Exemplare von Klassen, für deren Verhalten kein Statechart „erforderlich“ ist, werden als *primitive Objekte* bezeichnet.

systemweiten Warteschlange eingereihten Ereignisse stets nacheinander bearbeitet werden, wie in Abschnitt 8.2.1 dargestellt wurde. Zudem liegt die Priorität bei gleichzeitig möglichen Zustandsübergängen nun bei der niedrigeren Ebene.

8.3.1 Ereignis-Delegation

Wie oben erwähnt, besteht ein objektorientierter Statechart hauptsächlich aus den bereits in Abschnitt 3 erläuterten grafischen Komponenten, die in einigen Fällen etwas abgewandelt oder ergänzt wurden. Neben den Erweiterungen für die Erzeugung und Zerstörung von Objekten (siehe Abschnitt 8.3.2) stellt die Einführung eines Mechanismus zur Ereignis-Delegation die wichtigste Änderung dar.

Den Ausgangspunkt für diesen Mechanismus bildet die Frage, welches Objekt auf ein Ereignis e reagiert, das ein mittels Komposition aus verschiedenen Objekten zusammengesetztes Objekt A erhält. Da diesbezüglich verschiedene Ansichten möglich sind – nur das Objekt A, einige oder alle seiner Komponenten reagieren –, kann im obersten Zustand des Statecharts einer zusammengesetzten Klasse eine sogenannte *forwarding specification* definiert werden, in der die Delegationsstrategien für Ereignisse festgelegt sind. Eine Delegationsstrategie kann mittels der zwei Ausdrücke

delegate (*eventname*, *component1*, *component2*, ...)

broadcast (*eventname*)

formuliert werden; die zweite Alternative stellt jedoch lediglich eine abkürzende Schreibweise der ersten dar, in der das Ereignis *eventname* an alle Komponenten von A weitergereicht wird. In der Menge der einzelnen Elemente einer solchen forwarding specification ist das zusammengesetzte Objekt A selbst grundsätzlich implizit enthalten, und es gilt, daß nicht aufgeführte Ereignisse nur im Statechart von A bekannt sind und bearbeitet werden können. Durch die Ereignis-Delegation stehen beliebige Varianten der Ereignisverbreitung, von direkter Objekt-Objekt-Kommunikation bis zu vollständigem Broadcast, zur Verfügung, so daß Ereignisse auf vielfältige Weise zwischen Objekten übermittelt und zudem sehr weit verteilt werden können.

8.3.2 Erzeugung und Zerstörung von Objekten

Objektorientierte Systeme zeichnen sich u. a. dadurch aus, daß sich ihre Struktur im allgemeinen während der Zeit verändert, z. B. durch wechselnde Partner in Relationen oder durch die Erzeugung und Zerstörung von Objekten. Um die aktuelle Menge von Objekten und Beziehungen zu modifizieren, stehen in den Statecharts nach [HG96] folgende Aktionen zur Verfügung:

new *classname* (*parameters*)	*rolename* → add (*objectname*)
delete *objectname*	*rolename* → remove (*objectname*)

Unter Berücksichtigung des O-Charts von Seite 88 könnte im Statechart der Klasse *Car* als Aktion beispielsweise *stopsAt* → add(*term*) für ein Exemplar *term* der Klasse *Terminal* aufgeführt sein.

Mit der Erzeugung eines neuen Exemplars einer Klasse ist in der Regel auch die Ausführung einer Initialisierung verbunden. In dem Statechart zu einer Klasse ist der Einstiegspunkt für neu erzeugte Objekte durch das besondere Zeichen „Ⓝ“ gekennzeichnet; der hiervon wegführende Übergang zum ersten eingenommenen Zustand kann wie jede Transition beschriftet werden und wird von Harel und Gery als *Initialisierungsskript* bezeichnet. Neben der expliziten Erzeugung und Zerstörung eines Objektes durch new *classname* (*parameters*) und delete *objectname* ist zusätzlich die Möglichkeit der Terminierung und Selbstzerstörung eines Objektes vorgesehen. Ausgedrückt wird die selbsttätige Vernichtung eines Objektes durch eine Transition, die an dem Terminierungssymbol „Ⓣ“ endet.

8.3.3 Vererbungsbeziehungen

Über den allgemeinen Aufbau objektorientierter Statecharts hinausgehend betrachten Harel und Gery die Darstellung und Umsetzung von Vererbungsbeziehungen mit dem Ziel, bereits entwickelte Spezifikationen weiterzuverwenden. Hinsichtlich dieser Beziehung gibt es, so die Autoren in [HG96], verschiedene Möglichkeiten für die Bedeutung der Aussage, daß ein Exemplar der Klasse B gleichzeitig ein Objekt der Klasse A ist. Oftmals werde in diesem Zusammenhang ausschließlich auf Protokoll-Konformität[4] geachtet, gegebenenfalls zusätzlich auf strukturelle Konformität, d. h. darauf, daß B's interne Struktur (z. B. aggregierte Objekte) mit der von A konsistent ist. Aus der Sicht von Harel und Gery ergibt sich durch dieses Verständnis von Vererbung stets nur eine schwache Form des Subtyping, in der keinerlei Informationen über die Verhaltens-Konformität von A und B vorliegen. Somit kann sich ein Exemplar der Klasse B vollkommen anders als eines der Klasse A verhalten, wenn ein Ereignis e eintritt.

Die sehr schwer zu realisierende *volle* Verhaltens-Konformität zwischen zwei miteinander durch Vererbung verbundenen Klassen A und B erfordert noch viel Forschungsarbeit auf diesem Gebiet. Es ist jedoch fraglich, so die Autoren von [HG96], ob ein derartiges Ziel sinnvoll ist, wenn Vererbung zum Zweck der Wiederverwendung eingesetzt wird, da es der Forderung entspricht, daß ein Exemplar der Klasse B alles ausführen kann, was auch einem Objekt der Klasse A möglich ist, und dies zudem in genau der gleichen Weise vollbringt. In den meisten Fällen reicht es aus, wenn an einem Exemplar der Klasse B die gleichen Methoden aufrufbar sind wie an einem der Klasse A und diese in identischer Weise ausgeführt zu werden scheinen; die Ausführung selbst kann hingegen durchaus unterschiedlich sein und zu anderen konkreten Ergebnissen führen.

Von Harel und Gery wird ein verhaltensorientierter Ansatz für die Vererbung vorgeschlagen, mit dem eine gewisse Konformität erreicht und Wiederverwendung gefördert werden soll. In [HG96] wird betont, daß es leicht zu zeigen sei, daß keiner der zum Erscheinungszeitpunkt des Artikels vorliegenden Ansätze vor radikalen Veränderungen des Verhaltens bewahrt, wobei diese von den entsprechenden Autoren auch nicht unbedingt auf ein solches Ziel ausgerichtet seien. Die nachfolgend aufgeführten Über-

[4] Unter Protokoll-Konformität ist zu verstehen, daß die Klasse B mindestens die gleichen Operationen wie die Klasse A zur Verfügung stellt.

legungen entstanden im wesentlichen vor dem Hintergrund der von Harel und Gery angestrebten automatischen Codegenerierung und mit dem Ziel, möglichst hilfreich bei der Wiederverwendung schon generierten Codes zu sein.

Ausgangspunkt des hier erläuterten Vorgehens ist es, beide Statecharts auf der gleichen Zustands-/Transitions-Topologie basieren zu lassen, indem der Statechart der Oberklasse die Grundlage für den der abgeleiteten Klasse bildet. Es werden alle Zustände und Transitionen des Statecharts der Oberklasse übernommen und können nicht entfernt werden. Zur anschließenden Modifikation stehen folgende Möglichkeiten zur Verfügung:

- für Zustände:
  - Ein Basiszustand kann mittels AND- oder XOR-Dekomposition in Subzustände aufgesplittet werden.
  - Zu einem durch XOR-Dekomposition aus Subzuständen zusammengesetzten Zustand können weitere Subzustände hinzugefügt werden.
  - Ebenso kann ein mittels AND-Dekomposition verfeinerter Zustand um weitere orthogonale Komponenten ergänzt werden.
- für Transitionen:
  - Zum bestehenden Statechart können neue Transitionen hinzugefügt werden.
  - Es ist möglich, den Zielzustand einer vorhandenen Transition zu ändern; der Quellzustand bleibt jedoch stets erhalten.
  - Sowohl die zu erfüllende Vorbedingung als auch die auszuführende Aktion einer Transition können geändert werden. Somit ist es möglich, ererbte Transitionen, die nicht entfernt werden können, implizit zu beseitigen, indem die Vorbedingung auf *falsch* gesetzt wird.

8.4 Dynamik und Modellausführung

Ein durch einen O-Chart und eine Menge von Statecharts spezifiziertes System besteht, wenn es angestoßen wird und läuft, aus einer Menge konkreter Objekte, die miteinander über aktuelle Verweise (*Links*) kommunizieren. Im Gegensatz zu Hardware-Systemen ist hier die Topologie der Objekte und ihrer Beziehungen oft sehr dynamisch, so daß es gegebenenfalls schwierig ist, das Verhalten eines solchen Systems zu beschreiben. Um eine Verhaltenssemantik so zu definieren, daß sie Modellausführung oder automatische Codegenerierung gestattet, sind u. a. die folgenden zwei wesentlichen Punkte zu betrachten:

1. *Initialisierung*:
   Hinsichtlich der Initialisierung ist die Frage zu klären, wie das durch O-Chart und Statecharts formulierte Modell des beschriebenen Systems anfängt zu existieren, d. h. welche Objekte am Start erzeugt werden und wie ihre Attributwerte und Beziehungen zu anderen Objekten initialisiert werden.

2. *Dynamik im Zeitverlauf*:
   Der zweite Punkt betrifft die Veränderungen, denen das Modell im Laufe der Zeit unterliegt. Hierzu zählen zum einen schrittweise Reaktionen auf auslösende Vorkommnisse wie Ereignisse und Operationsaufrufe, zum anderen Strukturänderungen am Modell, die durch Objekterzeugung und -zerstörung sowie Änderung von Verweisen hervorgerufen werden.

Das zentrale Darstellungsmittel zur Spezifizierung des Verhaltens eines Systems und seiner Objekte sind die Statecharts, die zu jeder Klasse mit nichttrivialem Verhalten erstellt werden. Darüber hinaus enthält jedoch auch der O-Chart des Systems Informationen zum Verhalten; Teile der Dynamik (im wesentlichen Strukturänderungen) und der Initialisierung hängen sogar ausschließlich von diesen Informationen ab. Beim Start des Systems werden zunächst alle Exemplare von Klassen mit fest vorgegebener Objektanzahl erzeugt. Diese Erzeugung beginnt auf der obersten Ebene der Systembeschreibung im O-Chart und wird für alle durch Komposition aus Exemplaren anderer Klassen gebildeten Objekte entsprechend auf den tieferen Ebenen fortgeführt. Es werden keine Objekte erzeugt von Klassen, die im O-Chart keine Angaben zur Anzahl ihrer Exemplare aufweisen. Diese Erzeugungsstrategie wird auch im weiteren Verlauf des Modells eingesetzt, wenn durch Komposition zusammengesetzte Objekte explizit erzeugt werden; umgekehrt beinhaltet die Zerstörung eines solchen Objektes entsprechend auch die Zerstörung aller seiner Komponenten-Objekte.

Hinsichtlich der Beziehungen zwischen Objekten (siehe S. 88) bieten sich für diejenigen mit begrenzt mehrdeutigem Charakter zwei mögliche Semantiken an: Zum einen können stets so viele Exemplare wie möglich von den beteiligten Klassen erzeugt werden (*greedy*, in [HG96] eingesetzt), als Alternative jedoch auch nur so viele, wie unbedingt erforderlich sind (*nonchalant*). Der gewählte Ansatz kann nicht nur Auswirkungen auf die initiale Objekterzeugung zu Beginn des Systemlaufs haben, sondern gegebenenfalls auch für die Dynamik während des Laufs, da alle relevanten Beziehungen erneut ausgewertet werden, sobald ein Objekt erzeugt oder zerstört wird. In den einzelnen Statecharts verwendbare Aktionen zur Veränderung der aktuellen Objektmenge sowie der Beziehungspartner sind auf Seite 92 angegeben.

Anhand der erwähnten Informationen und Vorgehensweisen ist ein Großteil des initialen Systemzustandes nach dem Start ermittelbar. Weitere Angaben, z. B. zur Anzahl der Exemplare einer Klasse ohne im O-Chart festgelegte Objektanzahl, können vom Benutzer beim Start des Systems vorgegeben werden. Zudem ist es möglich, beim Beginn des Systemlaufs oder bei der Erzeugung eines Objektes notwendige Schritte im Initialisierungsskript des Statecharts zum System oder zum Objekt anzugeben.

9 Abschließende Betrachtungen und Ausblick

Der Vergleich von reaktiven und transformierenden Systemen am Anfang des vorliegenden Berichts verdeutlicht, daß die Natur reaktiver Systeme eine besondere Form der Verhaltensbeschreibung erfordert. Mit dem grafischen Formalismus der Statecharts steht ein geeignetes Beschreibungsmittel für das Verhalten derartiger Systeme zur Verfügung, das leicht verständlich ist und kompakte Darstellungen ermöglicht. Die ausführliche Erläuterung der einzelnen Konstrukte sowie das anschließende Beispiel, in dem viele der erwähnten Elemente verwendet werden, gestatten einen umfassenden Einblick in die grundlegenden Möglichkeiten, die der Formalismus bietet. Ergänzend wurde der Bereich der formalen Syntax und Semantik behandelt, um die Mehrdeutigkeit der rein grafischen Darstellung aufzuzeigen und die allgemeine Einführung abzurunden, auch wenn er für Dokumentationszwecke eher von untergeordneter Bedeutung ist.

Mit einer Übersichtsdarstellung von OMT, die sich auf das dynamische Modell konzentriert, beginnt die Betrachtung von Statecharts vor dem Hintergrund der objektorientierten Anwendungsentwicklung. Statecharts werden bei OMT für den Systementwurf eingesetzt und in eine ganzheitliche Entwicklungsmethode eingebunden. OMT orientiert sich im Bereich des dynamischen Modells stark an dem Grundlagenartikel [Har87], fügt nur wenige objektorientierte Konstrukte hinzu und behandelt die Beziehung zwischen Klassen und Statecharts eher vage. Im Gegensatz hierzu beinhaltet der zweite erläuterte Ansatz mit den Objectcharts eine wesentliche Erweiterung des ursprünglichen Formalismus. Objectcharts zeichnen sich durch eine größere Nähe zum Klassenkonzept aus und besitzen einen eher formalen Charakter, da sie bei den verwendeten Begriffen weniger auf die natürliche Sprache zurückgreifen. Der jüngste der beschriebenen Ansätze, zu dem Harel selbst maßgeblich beigetragen hat, fügt dem grafischen Formalismus ebenfalls zahlreiche objektorientierte Erweiterungen hinzu, greift aber die ursprünglichen Konzepte aus [Har87] in größerem Umfang auf. Auch hier bilden Statecharts die dynamische Komponente eines Systemmodells, allerdings steht die automatische Codeerzeugung im Vordergrund, so daß diese Variante stärker auf Aspekte der objektorientierten Programmierung ausgerichtet ist.

Im folgenden stellen wir zunächst die drei erläuterten Ansätze zur Verbindung von Statecharts mit objektorientierten Konzepten anhand ausgewählter Punkte gegenüber, um die wesentlichen Gemeinsamkeiten und Unterschiede in kompakter Form zu verdeutlichen und gegebenenfalls vorhandene Unzulänglichkeiten sowie Besonderheiten der einzelnen Ansätze aufzuzeigen. Anschließend legen wir dar, in welcher Form wir die Verwendung von Statecharts bei der objektorientierten Anwendungsentwicklung für sinnvoll halten und welche Charakteristika der einzelnen Ansätze uns für diesen Einsatz besonders geeignet scheinen. Wir beziehen uns bei dieser Betrachtung vor allem auf den am Arbeitsbereich Softwaretechnik gelehrten WAM-Methodenrahmen für die Entwicklung von Software.[1] Den Abschluß bildet eine Übersicht über weitergehende

[1] Zum WAM-Methodenrahmen siehe [KGZ94].

Fragestellungen, die sich im Laufe der Beschäftigung mit dem Statechart-Formalismus im allgemeinen und insbesondere vor dem Hintergrund des WAM-Leitbildes ergeben haben.

9.1 Gegenüberstellung der drei objektorientierten Ansätze

Bei der Betrachtung der drei Ansätze fällt zunächst einmal auf, daß sie Statecharts nicht isoliert behandeln, sondern stets in Beziehung zu weiteren Darstellungstechniken setzen, die andere Systemaspekte wiedergeben. OMT ist in dieser Hinsicht der ausführlichste Ansatz, in dem neben Statecharts für das dynamische Modell zusätzlich Klassendiagramme für das Objektmodell sowie Datenflußdiagramme für das funktionale Modell verwendet werden. Statecharts werden hierbei eindeutig gegen die anderen Techniken abzugrenzen versucht; sie sollen nur die Dynamik und nicht die Systemfunktionen an sich oder die Strukturen, auf denen diese operieren, wiedergeben.

In den beiden anderen Ansätzen wird auf eine Beschreibung im Sinne des funktionalen Modells nach OMT verzichtet. Coleman et al. verwenden ein Configuration Diagram, um die Systemstruktur auf der Basis tatsächlicher Objekte und der zwischen ihnen denkbaren Interaktionsmöglichkeiten festzuhalten, und definieren mit Hilfe von Objectcharts das Verhalten dieser Objekte auf Klassenebene. Da die Menge der Objekte im System und die Beziehungen zwischen ihnen somit fest vorgegeben sind, ist dieser Ansatz nur sehr stark eingeschränkt verwendbar. Harel und Gery beschreiben Systemstrukturen mit Hilfe von O-Charts, die den Klassendiagrammen von OMT ähneln und sich verändernde Mengen von Objekten berücksichtigen. Die konkrete Dynamik eines Systems hängt nicht nur von den Statecharts der einzelnen Klassen ab, sondern wird auch durch den jeweiligen O-Chart bestimmt.

Ein interessanter Aspekt, der sich für einen Vergleich der drei Ansätze anbietet, ist der für die Interaktion zwischen Objekten verwendete Kommunikationsmechanismus. OMT orientiert sich in dieser Hinsicht sehr stark an [Har87], und die Begriffe *Ereignis* und *Aktion* werden nahezu unverändert übernommen. Ereignisse besitzen allerdings nicht mehr nur eine einfache Signalfunktion, sondern können über Attribute weitere Informationen übermitteln. In den OMT-Statecharts kommen keine Methodenaufrufe vor; vielmehr werden die Ereignisse und Aktionen im Rahmen der Implementation auf informale Weise auf Methoden abgebildet (vgl. Seite 68). Konkrete Ereignisse sind Exemplare von Ereignisklassen und können direkt an andere Objekte gesendet oder über einen systemweiten Broadcast-Mechanismus[2] verbreitet werden. Die Ereignisklassen werden in einer Ereignishierarchie angeordnet, die unabhängig von der Klassenhierarchie ist. Transitionen eines Statecharts können zu „One-Shot"-Statecharts verfeinert werden (vgl. S. 61).

Eine vollständig andere Ansicht hinsichtlich des Kommunikationsmechanismus vertreten die Entwickler der Objectcharts: Anstelle von Ereignissen werden im Statechart ausschließlich Methodenaufrufe zur Beschreibung der Interaktion verwendet. Sie un-

[2]Dies bedeutet, daß alle Statecharts auf das gesendete Ereignis reagieren können.

terscheiden zwischen Observern[3], die in Zustände eingetragen werden, und zustandsverändernden Methoden, durch deren Aufruf Transitionen ausgelöst werden können. Die Aktion eines solchen Übergangs umfaßt den direkten Aufruf maximal einer Methode an einem anderen Objekt; auf den Broadcast-Mechanismus wird verzichtet. Eine Beschreibung geschachtelter Methodenaufrufe ist nicht möglich, da vorausgesetzt wird, daß stets nur eine Interaktion zur Zeit stattfindet und diese beendet wird, bevor die nächste beginnt. Hängen Transitionen von Bedingungen ab, so können diese entweder der Transitionsbeschriftung hinzugefügt oder innerhalb einer Transitionsspezifikation formuliert werden.

In dem Ansatz von Harel und Gery werden schließlich sowohl Ereignisse als auch Operationsaufrufe eingesetzt. Dementsprechend ist als Auslöser eines Zustandsübergangs ein Ereignis oder ein Methodenaufruf zulässig, und die zugehörige Aktion kann sich aus Ereignisgenerierungen und Methodenaufrufen zusammensetzen. Wie bei OMT sind Ereignisse parametrisierbar und können in einer Hierarchie angeordnet werden. Sie werden jedoch stets gezielt an Objekte versendet[4] und können durch Delegation weitergegeben werden (vgl. Abschnitt 8.3.1). Während Ereignisse über eine systemweite Warteschlange verwaltet und erst zu einem späteren Zeitpunkt bearbeitet werden, bedeutet ein Operationsaufruf die sofortige Methodenausführung im gerufenen Objekt. Für die Dauer der Methodenausführung wird die Transition im rufenden Objekt „eingefroren“. Da weitere Aufrufe durch das zweite Objekt denkbar sind, können – anders als bei den Objectcharts – geschachtelte Methodenaufrufe abgebildet werden. Offen bleibt jedoch, welche Auswirkungen ein weiterer Methodenaufruf am ursprünglich rufenden Objekt während der Methodenausführung hat.

Möchte man das Verhalten einer sich im Laufe der Zeit verändernden Menge von Objekten mit Hilfe von Statecharts beschreiben können, so müssen die Erzeugung und die Zerstörung von Objekten berücksichtigt werden. Bei OMT wird dieser Punkt nicht gesondert erwähnt, und auch Coleman et al. betrachten ihn nicht explizit, da sie nur statische Systeme behandeln. Objectcharts könnten jedoch zumindest die Initialisierung vorhandener Objekte durch Transitionsspezifikationen für Default-Pfeile beschreiben. Harel und Gery sehen die Erzeugung und Zerstörung von Objekten durch spezielle Aktionen vor. Zudem gibt es in ihren Statecharts einen Einstiegspunkt „Ⓝ“ für neue Objekte, der einem Default-Pfeil auf oberster Zustandsebene entspricht, sowie einen Endpunkt „Ⓣ“ für die Zerstörung eines Objektes (vgl. S. 92).

Der Bereich der Vererbung ist bei objektorientierten Herangehensweisen von besonderem Interesse und wird bei allen drei Ansätzen betrachtet. Die Unterscheidung Harels und Gerys zwischen Protokoll-, Struktur- und Verhaltens-Konformität (vgl. S. 93) zeigt jedoch bereits auf, daß es schwierig ist, geeignete Kriterien für eine Vererbungsbeziehung zwischen zwei Klassen zu formulieren. Vor diesem Hintergrund ist es nicht verwunderlich, daß sich die Autoren von OMT nur sehr vage äußern. Ihnen zufolge sollte der Statechart zu einer Unterklasse weitgehend unabhängig von dem der Oberklasse

[3] (speziellen) sondierenden Operationen

[4] Wenn das sendende Objekt zugleich das Ziel ist, entspricht dies dem Broadcast-Mechanismus aus [Har87], welcher nicht systemweit ist, sondern sich stets auf einen Statechart beschränkt.

formuliert werden und alle Statecharts höherer Ebenen des Hierarchiebaumes „erben“. Im Rahmen der beiden anderen Ansätze werden zwar konkrete Regeln für die Bildung von Unterklassen- aus Oberklassen-Statecharts angegeben (vgl. Abschnitte 7.2.4 und 8.3.3); diese führen jedoch nicht zwangsläufig zu voller Verhaltens-Konformität.

9.2 Statecharts und der WAM-Methodenrahmen

Durch die Beschäftigung mit den drei erläuterten Ansätzen wird deutlich, daß Statecharts stets nur einen Teil zu einer Systembeschreibung beitragen können und weitere Darstellungstechniken unerläßlich sind. So beinhalten Klassendiagramme Informationen über die Systemstruktur, die durch Statecharts nicht ausgedrückt werden können. Umgekehrt veranschaulichen Statecharts grundsätzlich dynamische Aspekte, die Klassendiagramme nicht wiederzugeben vermögen.

Für einen konkreten Einsatz von Statecharts ist abzuwägen, welche syntaktischen Konstrukte dafür geeignet sind, eine dem jeweiligen Kontext angemessene Beschreibung der Dynamik zu ermöglichen. Hierbei spielen besonders die mit der Verwendung von Statecharts verfolgten Ziele eine große Rolle. Ein in der Literatur kaum erwähntes Einsatzfeld für Statecharts ist der Bereich der Dokumentation von Software. Wir halten diese Verwendungsmöglichkeit für besonders interessant, denn dynamische Aspekte werden bei Beschreibungen von existierenden Systemen häufig vernachlässigt. Möchte man Statecharts zur Dokumentation objektorientierter Software einsetzen, so stellt sich die Frage, in welchem Umfang Dynamik wiedergegeben werden soll. Die bisherigen Erfahrungen am Arbeitsbereich Softwaretechnik zeigen, daß in der Regel nur wenige klassenübergreifende Folgen von Methodenaufrufen von Interesse sind. Eine größere Bedeutung kommt den möglichen Aufrufreihenfolgen der einzelnen Klassen zu, die geschlossene fachliche Konzepte realisieren. In diesem Zusammenhang ist vor allem eine sinnvolle, an fachlichen Gesichtspunkten orientierte Gliederung der Zustandsräume auf Klassenebene wichtig.

Bei der Erarbeitung eines Statecharts sollten qualitative Aspekte im Vordergrund stehen, so daß sich ausschließlich Zustände ergeben, deren Bedeutung in aussagekräftigen Bezeichnern zusammengefaßt werden kann. Um die fachlichen Aussagen, die mit Zuständen verbunden sind, exakt fassen zu können, ist ggf. die Formulierung von Invariantenspezifikationen wie bei Coleman et al. ein geeignetes Mittel. Auf diese Weise kann eine Beziehung zwischen den fachlich orientierten Zuständen eines Statecharts und den konkreten Belegungen der Exemplarvariablen hergestellt werden. Vor dem Hintergrund des Leitbildes von Werkzeug und Material ist vor allem der fachliche Zustand von Funktionskomponenten sowie von Materialien wesentlich; deshalb können gerade sie mit Hilfe von Statecharts gut dokumentiert werden.

Da im Falle der Verwendung von Statecharts zu Dokumentationszwecken der Programmcode bereits vorliegt, sollte eine Transition zwischen zwei Zuständen mit dem Namen derjenigen Methode beschriftet werden, deren Aufruf den Zustandsübergang auslöst. Wir halten es nicht für sinnvoll, Transitionen zusätzlich mit Methoden, die Objekte der betrachteten Klasse an anderen Objekten aufrufen, zu beschriften, weil

die Statecharts hierdurch einen beträchtlichen Teil des Programmtextes wiedergäben und der Vorteil der kompakten grafischen Darstellung leicht zunichte gemacht werden könnte. Eine solche Beschriftung mit dem Ziel, den Zusammenhang zwischen den einzelnen Klassen aufzuzeigen, verspricht ebenfalls keinen Gewinn, da sich die Interaktionspartner bereits aus Klassendiagrammen ergeben. Beispielsweise sind die Formen der Interaktionen zwischen Kontextwerkzeugen, Subwerkzeugen und Materialien innerhalb des WAM-Methodenrahmens auf diese Weise beschrieben. Wie bereits erwähnt, sind in den meisten Fällen nur einige ausgewählte klassenübergreifende Folgen von Methodenaufrufen zu dokumentieren. Für diesen Zweck haben sich Interaktionsdiagramme, die die Darstellung auf die im konkreten Fall wichtigen Interaktionen reduzieren und darüber hinaus die beteiligten Objekte schnell erkennen lassen, bestens bewährt.

Neben den zustandsverändernden Methoden, die in den einzelnen Transitionsbeschriftungen aufgeführt sind, müssen für eine vollständige Dokumentation auch sondierende Methoden berücksichtigt werden. Wir halten es für ratsam, diese unter Ausnutzung der Zustandshierarchie im Sinne der von Coleman et al. beschriebenen Observer direkt in diejenigen Zustände einzutragen, in denen ihr Aufruf aus fachlicher Sicht zulässig ist. Auf diese Weise kann einfach festgestellt werden, unter welchen Umständen sondierende Methoden sinnvolle Ergebnisse liefern.

Harel und Gery lassen nicht nur Methodenaufrufe, sondern auch Ereignisse als Auslöser von Transitionen zu. Grundsätzlich kann es für den Bereich der Dokumentation durchaus nützlich sein, auch das Konzept der Ereignisse in Statecharts aufzugreifen, denn Ereignisse führen eine zusätzliche Abstraktionsebene ein, die ein besseres Verständnis ermöglichen kann. Sie sollten allerdings nicht zwischen beliebigen Objekten versendet, sondern vielmehr gezielt eingesetzt werden. Die typische Verwendung im WAM-Kontext ist die lose Kopplung zwischen der Funktions- und der Interaktionskomponente eines Werkzeuges. Benachrichtigt eine Funktionskomponente ihre Interaktionskomponente über eine Zustandsänderung, so geschieht dies über einen Ereignis-Mechanismus. Im Statechart zur Funktionskomponente kann eine solche Ereignisauslösung berücksichtigt werden, um die Verknüpfung zwischen einem neu eingenommenen Zustand und einem speziellen Ereignis zu verdeutlichen. Wir schlagen vor, auf Ereignisse als Transitionsbeschriftungen vollständig zu verzichten und eine Ereignisgenerierung direkt in einen Zustand einzutragen, denn für alle Transitionen mit gleichem Zielzustand wird das gleiche Ereignis ausgelöst.

Statecharts können nicht nur zur Dokumentation von Software, sondern auch während des gesamten Entwicklungsprozesses eingesetzt werden, wie beispielsweise die weite Verbreitung von OMT zeigt. Hier ist es ggf. sinnvoll, sich zunächst an Ereignissen im allgemeinen Sinne zu orientieren und diese im Statechart aufzuführen, um mit ihrer Hilfe Vorgänge im System zu formulieren. Harel und Gery legen dar, daß sogar eine automatische Codeerzeugung möglich ist. Zu den Vorteilen, die Statecharts durch ihre Verwendung beim tatsächlichen Prozeß der Softwareentwicklung evtl. mit sich bringen, können wir aufgrund mangelnder Erfahrungen keine Angaben machen. Wir wollen jedoch abschließend kurz zum Bereich der automatischen Codegenerierung Stellung beziehen.

Unserer Ansicht nach sind mit einer derartigen Zielsetzung etliche Probleme ver-

bunden, die insbesondere im Falle komplexer Systeme von Bedeutung sind. Zum einen müßte ein System bis ins kleinste Detail spezifiziert werden; dies könnte dazu führen, daß die grafische Darstellung großer Systeme nicht mehr überschaubar wäre. Zum anderen wäre die Architektur der resultierenden Software streng festgelegt und kaum flexibel, und der generierte Code wäre voraussichtlich nur schlecht lesbar sowie schwierig von Hand zu warten. Als besonders kritisch müssen nachträgliche manuelle Änderungen an den erzeugten Code-Rümpfen angesehen werden, denn diese schlagen sich bei existierenden CASE-Tools oftmals nicht in der ursprünglichen Spezifikation nieder. Trotz all dieser Punkte ist die automatische Codeerzeugung aus einer grafischen Darstellung, die auf Statecharts basiert, in unseren Augen von Bedeutung; wir meinen, daß sie besonders im Anwendungsfeld technischer Systeme, das sich durch einen hohen Detaillierungsgrad und häufig auch durch eine enorm große Anzahl von Ereignissen auszeichnet, nützlich sein kann.

9.3 Weitergehende Fragestellungen

Während der intensiven Beschäftigung mit dem grafischen Formalismus der Statecharts haben sich zahlreiche Fragestellungen ergeben, die mögliche Ausgangspunkte für weitere Arbeiten auf diesem Gebiet sind.

Wie wir im vorigen Teilabschnitt erläutert haben, können Statecharts unserer Ansicht nach im Rahmen der Dokumentation objektorientierter Software verwendet werden, um die fachlichen dynamischen Aspekte einer Klasse wiederzugeben. Insbesondere für erweiterbare Komponenten von Klassenbibliotheken und Frameworks scheint eine Dokumentation sinnvoll. Eventuell können hier die Bedürfnisse von Endanwendern und Weiterentwicklern unterschieden werden; für Weiterentwickler mag eine umfangreichere Dokumentation mit mehr oder detaillierteren Statecharts von großem Wert sein. Nur die konkrete Anwendung unserer Ideen kann jedoch zeigen, wie groß der tatsächliche Nutzen ist. Des weiteren ist es sicherlich interessant, anhand eines Beispiels zu untersuchen, ob Statecharts auch eine Rolle im fachlichen oder technischen Entwurf nach dem WAM-Methodenrahmen spielen können. Möglicherweise kämen hier auf einer informalen Basis verstärkt Ereignisse zum Tragen.

Die geschilderte Eintragung von Ereignissen in die Zustände der Statecharts von Funktionskomponenten kann vielleicht zu einem Kriterium für einen guten Entwurf führen: Müßten von einem Zustand mehrere Ereignisse ausgesendet werden, so könnte dies auf eine nicht fachlich motivierte Strukturierung des Zustandsraumes hindeuten.

Eine weitere Fragestellung werfen AND-Komponenten auf. Da sie mit einer Partitionierung der Menge der Exemplarvariablen einhergehen, sind sie unter Umständen ein Hinweis darauf, daß verschiedene fachliche Konzepte, die durch mehrere unabhängige Klassen repräsentiert werden sollten, in einer Klasse zusammengefaßt worden sind. Enthält ein Statechart mehrere AND-Komponenten, so bedeutet dies vielleicht auch, daß ein eventuell fachlich unabhängiger Teil in eine Oberklasse ausgelagert werden kann.

Es könnte zudem hinsichtlich der Entwicklung eines Statecharts zu einer bereits

existierenden Klasse geprüft werden, in welchem Umfang im Programmcode formulierte Vorbedingungen dazu beitragen, die Zustände der Klasse leichter herauszuarbeiten.

Literatur

[CHB92] D. Coleman, F. Hayes, and S. Bear. Introducing objectcharts or how to use statecharts in object-oriented design. *IEEE Transactions on Software Engineering*, 18(1):9–18, January 1992.

[CY90] P. Coad and E. Yourdon. *Object-Oriented Analysis*. Yourdon Press Computing Series. Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, second edition, 1990.

[Gil93] W. K. Giloi. *Rechnerarchitektur*. Springer-Verlag, Berlin Heidelberg New York, zweite Auflage, 1993.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[Har92] D. Harel. Biting the silver bullet: Toward a brighter future for system development. *Computer*, 25(1):8–20, January 1992.

[Har97] D. Harel. Some thoughts on statecharts, 13 years later. In O. Grumberg, editor, *Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 226–231. Springer-Verlag, Berlin Heidelberg New York, 1997.

[HC91] F. Hayes and D. Coleman. Coherent models for object-oriented analysis. *ACM SIGPLAN Notices*, 26(10):171–183, October 1991.

[HdR91] C. Huizing and W. P. de Roever. Introduction to design choices in the semantics of statecharts. *Information Processing Letters*, 37(4):205–213, February 1991.

[HG96] D. Harel and E. Gery. Executable object modeling with statecharts. In *18th International Conference on Software Engineering*, pages 246–257. IEEE Computer Society Press, 1996. Also in *Computer*, 30(7):31–42, July 1997.

[HK92] D. Harel and C.-A. Kahana. On statecharts with overlapping. *ACM Transactions on Software Engineering and Methodology*, 1(4):399–421, October 1992.

[HLN+90] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[HN96] D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. In K. R. Apt, editor, *Proceedings of the NATO Advanced Study Institute on Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI Series, Series F, Computer and System Sciences*, pages 477–498, Heidelberg, 1985. Springer-Verlag.

[HPSS87] D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman. On the formal semantics of statecharts. In *Proceedings of the Symposium on Logic in Computer Science*, pages 54–64. IEEE Computer Society Press, 1987.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to automata theory, languages and computation.* Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1979.

[Ker93] H. Kerner. *Rechnernetze nach OSI.* Addison-Wesley (Deutschland) GmbH, zweite Auflage, 1993.

[KGZ94] K. Kilberth, G. Gryczan und H. Züllighoven. *Objektorientierte Anwendungsentwicklung.* Friedr. Vieweg & Sohn Verlagsgesellschaft mbH, Braunschweig/Wiesbaden, zweite, verbesserte Auflage, 1994.

[KP91] Y. Kesten and A. Pnueli. Timed and hybrid statecharts and their textual representation. In J. Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 571 of *Lecture Notes in Computer Science*, pages 591–620. Springer-Verlag, Berlin Heidelberg New York, 1991.

[MM85] J. Martin and C. McClure. *Diagramming Techniques for Analysts and Programmers.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1985.

[Par69] D. L. Parnas. On the use of transition diagrams in the design of a user interface for an interactive computer system. In *Proceedings of the 24th National Conference of the ACM*, pages 379–385, New York, 1969. Association for Computing Machinery.

[PR94] B. Paech and B. Rumpe. A new concept of refinement used for behaviour modelling with automata. In M. Naftalin, T. Denvir, and M. Bertran, editors, *FME '94: Industrial Benefit of Formal Methods*, volume 873 of *Lecture Notes in Computer Science*, pages 154–174. Springer-Verlag, Berlin Heidelberg New York, 1994.

[PS91] A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 244–264. Springer-Verlag, Berlin Heidelberg New York, 1991.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1991.

[Rum95] J. Rumbaugh. OMT: The dynamic model. *Journal of Object-Oriented Programming*, 7(9):6–12, February 1995.

[SM88] S. Shlaer and S. J. Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data*. Yourdon Press Computing Series. Yourdon Press, Englewood Cliffs, New Jersey, 1988.

[SM92] S. Shlaer and S. J. Mellor. *Object Lifecycles: Modeling the World in States*. Yourdon Press, Prentice Hall Building, Englewood Cliffs, New Jersey 07632, 1992.

[Tan92] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, Inc., Upper Saddle River, New Jersey 07458, 1992.

[vdB94] M. von der Beeck. A comparison of statecharts variants. In H. Langmaack, W. P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 128–148. Springer-Verlag, Berlin Heidelberg New York, 1994.

[Weg97] P. Wegner. Why interaction is more powerful than algorithms. *Communications of the ACM*, 40(5):80–91, May 1997.