Übersetzung von Statecharts nach AspectJ

Projektarbeit

Liu, Yu

Matrikelnummer: 22495

26.Juli 2006

Betreuer der Arbeit

Prof. Dr. Ralf Möller
STS – TUHH

M.Sc. Miguel GARCIA
STS - TUHH

Deklaration

Ich deklariere:

Alle Zitate, ob wörtliche oder sinngemäße Quellenangaben, habe ich schone hingewiesen.

26.07.2006, Hamburg

Liu, Yu

Inhaltsverzeichnis

1 Einleitung

Die Object Modelling Technique (OMT) stellt eine objektorientierte Entwicklungsmethodologie vor, bei der ein Softwaresystem mit drei Modellen beschrieben wird. Das erste Objektmodell beschreibt mit Hilfe eines Objektdiagramms die statische Struktur, Objekte und Relationen zwischen Objekten des Systems. Das zweite dynamisches Modell beschreibt mit Hilfe von Zustandsdiagrammen die zeitlichen Veränderungen des Systems. Das dritte funktionale Modell beschreibt schließlich mit Hilfe von Datenflussdiagrammen die Transformation der Daten innerhalb des Systems. Verwendet man die Idee von den 1. und 2. Modellen, kann beispielweise für ein Objektmodell verschiedene dynamische Modelle entwickelt werden, das heißt auch, verbindet ein Objektmodell mit unterschiedlichen dynamischen Modellen, hat es in der Laufzeit auch unterschiedlich zeitlichen Veränderungen. Das dynamische Modell wird mit Hilfe von Zustandsdiagrammen beschrieben. Zustandsdiagrammen setzen Zustände und Ereignisse voraus. Es wird daher eine Methode benötigt um die Methodenaufrufe nach Ereignissen zu übersetzen. AspectJ ist eine aspektorientierte Programmierungssprache, mit Hilfe von AspectJ Compiler können Aspekte mit normalen Java-Bytecode verbunden werden. Die Nutzung von Pointcuts und Advices in AspectJ kann normale Methodenaufrufe nach Ereignisse übersetzen. Das heißt auch, wir können mit Hilfe von Aspekten die Systeme, die nicht ereignisbasiert sind, in ereignisbasierte Systeme umzuwandeln. Wird ein Zustandsdiagramm in AspectJ implementiert, kann es mit Hilfe von AspectJ Compiler in normalen Java-Bytecode eingebunden werden, damit Objekte und Komponente mit Zustandsdiagrammen angereichert werden können. Statecharts sind die Erweiterungen für flache Zustandsdiagramme, um Hierarchie und Parallelität einzuführen. Damit können komplizierte Systeme modelliert werden. Der Schwerpunkt dieser Projektarbeit ist die Entwicklung eines Frameworks für Statecharts mit Hilfe von AspectJ.

2 Grundlage

In diesem Kapitel wird die Grundlage der Zustandsdiagramme erläutern. Es wird zuerst die Elemente in den flachen Zustandsdiagrammen vorstellen, z. B. Zustände, Ereignisse, Bedingungen und Zustandsübergänge. Danach werden die Erweiterungen in Statecharts für die flachen Zustandsdiagramme vorgestellt.

2.1 Zustände

Ein Zustand [2, 3] beschreibt die aktuellen Werte und Verknüpfungen eines Objektes und besitzt einen Zeitraum. In einem Objektmodell werden die Objekte, einschließlich ihrer Attribute und der Relationen zwischen ihnen beschrieben. Im seinem Leben eines Objektes kann er verschiedene Zustände annehmen, und jeder Zustand wird beschrieben mit Hilfe von einer notwendigen Untermenge von ihren Attributwerten und Relationen. Eine Änderung davon kann daher als eine Zustandsänderung angesehen werden. Beispielweise kann der Zustand eines Produkts in einem Onlineshop „verfügbar“ oder „nicht verfügbar“ sein, je nachdem, ob das Attribut „Gebote“ null ist oder nicht. Im Zustandsdiagramm wird ein Zustand durch ein Rechteck mit abgerundeten Ecken dargestellt und durch einen für ihn eindeutigen Namen definiert. Gleichnamige Zustände innerhalb eines Zustandsdiagramms beschreiben den selben Zustand eines Objektes. Start- und Endzustand sind Zustände von insbesondere Typen, da ein Startzustand niemals einen Übergang zu selbem hat und keine Änderung einem Endzustand folgt.

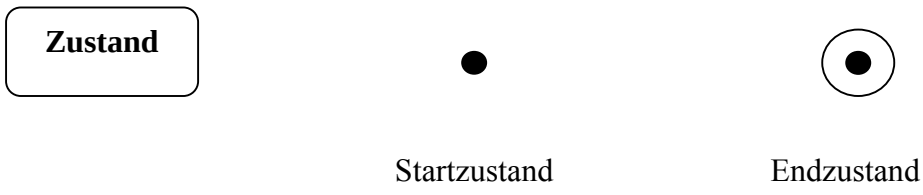


Abbildung 2-1

2.2 Ereignisse

Ein Ereignis [2,3] stellt einen „äußeren Reiz“ auf ein Objekt dar, der das Objekt veranlasst, seinen Zustand zu ändern. Im Vergleich zu dem Konzept von Zuständen, das den Zeitraum zwischen dem Eintreffen von Ereignissen modelliert, hat ein Ereignis keine Zeitdauer. Es gibt eigentlich in der Praxis keine zeitlosen Vorgänge, aber man kann die Reaktionszeit auf ein Ereignis im Vergleich zu der Zeitdauer zwischen dem Eintreffen von zwei Ereignissen als sehr geringer betrachten. Durch ein Ereignis und seine Attribute wird das System benachrichtigt über die Informationen, die aus der Umgebung kommen (externes Ereignis) oder innerhalb des Systems (internes Ereignis) generiert werden.

2.3 Zustandsübergänge

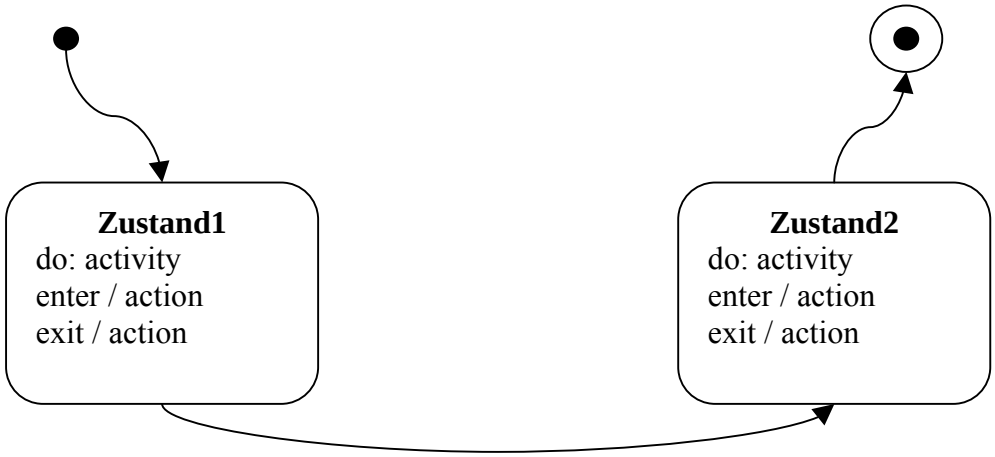
Ein Zustandsübergang [2, 3] beschreibt, wie sich ein System auf das Eintreten eines Ereignisses reagieren kann. Das System kann in jedem Zustand auf verschiedene

Ereignisse mit unterschiedlichen Zustandübergängen reagieren. Wenn das System sich im Quellzustand befindet, und tritt ein Ereignis ein, wird ein Zustandsübergang genommen, so wird das System im Zielzustand des Zustandsübergangs wechseln. Im Fall, dass mehrere Zustandsübergänge gleichzeitig feuern könnten, sollen die Konflikte der Zustandsübergänge behandelt werden.

2.4 Bedingungen

Bedingungen [2,3] sind boolesche Funktionen, damit wird das Konzept des Zustandsübergangs angereichert. Die in dieser Weise erweiterten Zustandsübergänge können nur feuern, wenn entsprechende Ereignisse eintreten und gleichzeitig die Bedingungen erfüllt sind. Beispielweise kann ein Produkt in einem Onlineshop nur verkauft werden, wenn die Attribute „Gebote“ größer als null ist.

2.5 Zustandsdiagramme



Ereignis(Attribute)[Bedingung]/Aktion

Abbildung 2-2

Ein flaches Zustandsdiagramm [2, 3] wird mit Hilfe von den in Abschnitten 2.4 bis 2.5 vorgestellten Elemente zusammengebaut und unterstützt keine Hierarchie und Parallelität. Jedes Zustandsdiagramm ist wie ein gerichteter Graph, Zustände werden als Knoten beschrieben und Zustandsübergänge als Kanten, die mit der Kombination *Ereignis (Attribute) [Bedingung] / Aktion* beschriftet.

Die Abbildung 2-2 beschreibt, Wenn sich das System im Zustand1 befindet, tritt ein Ereignis mit den Attributen ein, und die Bedingung ist gleichzeitig erfüllt, dann wird der Übergang vom Zustand1 nach dem Zustand2 genommen, und während des Zustandsübergangs wird die Aktion ausgeführt. Die Aktivität wird durch do: gekennzeichnet. Die Aktivitäten sollen nur mit Zuständen verbinden, weil sie gewisse Zeitdauer haben. Im Vergleich zu den Aktivitäten können Aktionen sowohl mit Zuständen als auch mit Übergängen verbunden, da sie keine Zeitdauer haben. Die *exit / aktion* vom Zustand1 wird nur ausgeführt, wenn der Zustand1 verlassen wird. Wird den Zustand2 betreten, dann wird *enter / aktion* vom Zustand2 ausgeführt.

2.6 Mealy- vs. Moore-Automaten

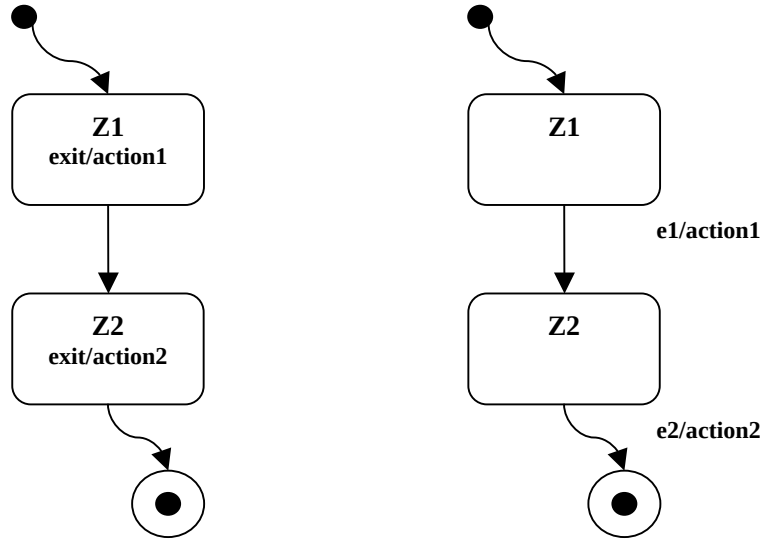


Abbildung 2-3

Man verwendet häufig zwei Type von endlichen Automaten. Beim ersten Ansatz werden Aktionen mit den Zustandsübergängen verbunden, d.h. die Aktionen erfolgen bei der Ausführung eines Zustandsübergangs. Ein Automat dieses Typs heißt Mealy-Automat. Beim zweiten Ansatz werden die Aktionen mit Zuständen verbunden, d.h. die Aktionen erfolgen nur, wenn die Zustände betreten werden. Ein Automat dieses Typs heißt Moore-Automat. Jede Aktion hat in der Praxis eine zeitliche Verzögerung. Der Zustand eines Mealy-Automaten kann daher nicht fest sein, während sich das System bei dem Zustandsübergang befindet. Normalweise hat ein Moore-Automat aber kein gleiches Problem, weil die Aktionen mit den Zuständen verbunden sind. Die auszuführenden Aktionen eines Moore-Automaten sind im Prinzip von dem aktuellen Zustand abhängig, während die auszuführenden Aktionen des Mealy-Automaten sowohl von dem aktuellen Zustand und als auch von der Eingabe abhängen. Es wurde bewiesen, dass die beiden Automaten mathematisch gleich sind, d.h. für jeden Mealy-Automaten besteht ein gleicher Moore-Automat und für jeden Moore-Automaten besteht auch ein gleicher Mealy-Automat. Im allgemein, das mit Hilfe von den Moore-Automaten beschriebenes System wird mehrere Zustände benötigt als das gleiche System, das mit Hilfe von den Mealy-Automaten beschrieben wird. Der Grund liegt darin, dass die Moore-Automaten mehrere Zustände verwenden sollten, um die Ausführung von unterschiedlichen Aktionen darzustellen. Jeder Zustand in den Mealy-Automaten kann aber mehrere Zustandsübergänge besitzen, deren Aktionen während der Übergänge ausgeführt werden könnten. In der Praxis werden daher die Mealy-Automaten häufig verwendet um die Zustandsdiagramme zu beschreiben.

2.7 Zustandsdiagramme vs. Ablaufsdiagramme

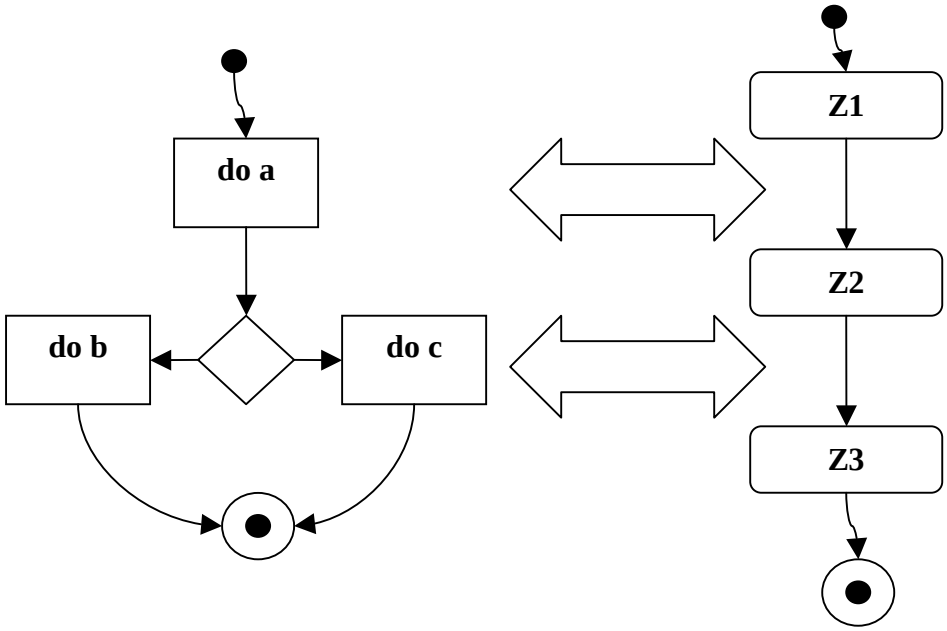


Abbildung 2-4

Der Zustandsübergang von einem Knoten zu einem anderen in Zustandsdiagrammen benötigt deutlich das Auslösen eines Ereignisses, das aus der Umgebung kommt (externes Ereignis) oder innerhalb des Systems (internes Ereignis) generiert wird. In Ablaufdiagrammen sind die Übergänge aber von der Ausführung der Aktionen abhängig, d.h. Wenn die Ablaufdiagramme mit den Zustandsdiagrammen verglichen werden, sind die Bedeutungen von Knoten und Kanten in Ablaufdiagrammen umgekehrt. Beispielweise funktioniert ein Ablaufdiagramm wie ein Produktfließband, aber ein Zustandsdiagramm funktioniert passiv und reaktive.

2.8 Statecharts vs. Flache Zustandsdiagramme

Statecharts sind die Erweiterungen der flachen Zustandsdiagramme. Der hauptsächliche Vorteil der Statecharts ist die Einführung der Konzepte von Hierarchie und Parallelität. Wie es ist vorher erwähnt, dass die flachen Zustandsdiagramme im Gegensatz dazu diese Konzepte nicht unterstützen. In der Praxis haben sie daher viele Nachteile [4] bei der Modellierung komplizierter Systeme. Unter anderem sind die Gründe:

1. Wegen der fehlenden Modularität und Hierarchie unterstützen flache Zustandsdiagrammen die schrittweise „von oben nach unten“ oder „von unten nach oben“ Entwicklung nicht. Beispielweise besteht ein System aus vielen in sich geschlossen Modulen, so wäre es günstig, dass für jedes Modul ein Zustandsdiagramm definiert wird. Jedes Modul kann somit bei der Modellierung des gesamten Systems als „Schwarze Kiste“ betrachtet wird. Damit werden die Wiederverwendbarkeit der Module erhöht, das Tempo der Softwareentwicklung beschleunigt und den Test der Module erleichtert. Statecharts unterstützen Hierarchie und somit Modularität. Ein Zustand in Statecharts kann Unterzustände aufnehmen, die wieder andere Unterzustände

besitzen könnten. Beispielweise kann ein Zustand im Statecharts des gesamten Systems die Wurzelzustände der Zustandsdiagramme der Module als seine Unterzustände aufnehmen, somit wird die Modularität des gesamten Systems erheblich verbessert.

2. Die Anzahl der Zustandsübergänge nimmt schnell zu, wenn mehrere Anfangszustände für ein gleiches Ereignis mehrere Übergänge mit der gleichen Beschriftung zu einem einzelnen Zielzustand besitzen. In Abbildung 2-5 wird dieser Sachverhalt verdeutlicht. Das Ereignis E verursacht in den Zuständen B, C und D eine Rückkehr nach dem Zustand A. Es müssen also drei Zustandsübergänge erzeugt werden. Aber mit Hilfe von Statecharts können beispielweise wie in Abbildung 2-6 die Zustände C und D als Unterzustände vom Zustand B aufgenommen werden, somit kann der Vaterzustand B von den Zuständen C und D die Übergänge zum Zustand A übernehmen, und die Anzahl der Übergänge wird somit vermindert werden. In der Praxis kann es vorkommen, ein hierarchisches Zustandsdiagramm in ein mathematisch gleiches flaches Zustandsdiagramm zu transformieren. Diese führt häufig dazu, dass sich die Anzahl der Zustandübergänge schnell zunimmt. In diesem Fall werden die Start- und Endzustände mehrere Zustandsübergänge besitzen.

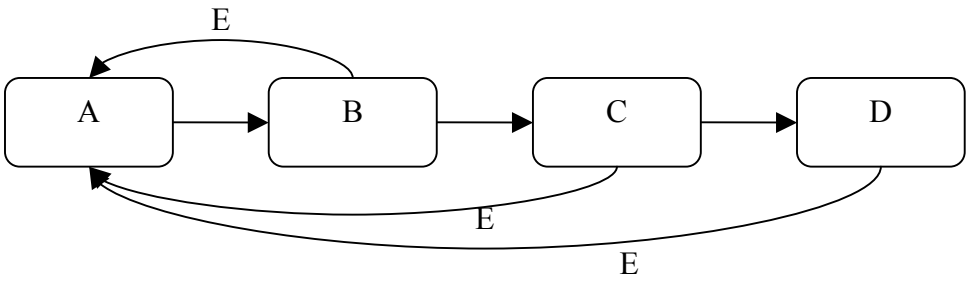


Abbildung 2-5

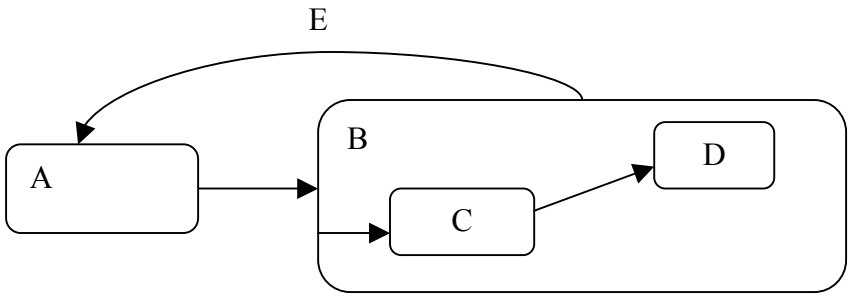


Abbildung 2-6

3. Wenn ein System beispielweise n boolesche Attributen besitzt, die voneinander unabhängig sind, so muss das System 2^n Zustände in das Diagramm aufnehmen. Dies führt zum schnellen Zunehmen der Anzahl von Zuständen. Mit Hilfe von Statecharts können die n unabhängigen booleschen Attribute zwischen dem Vaterzustand und seinen Unterzuständen entsprechend verteilt werden, somit wird die Anzahl der Zustände des Systems erheblich vermindert. Das bedeutet auch, dass die Hierarchische Struktur in der Praxis mehr deskriptiver ist und eine Optimierungsmöglichkeit mit sich hat.

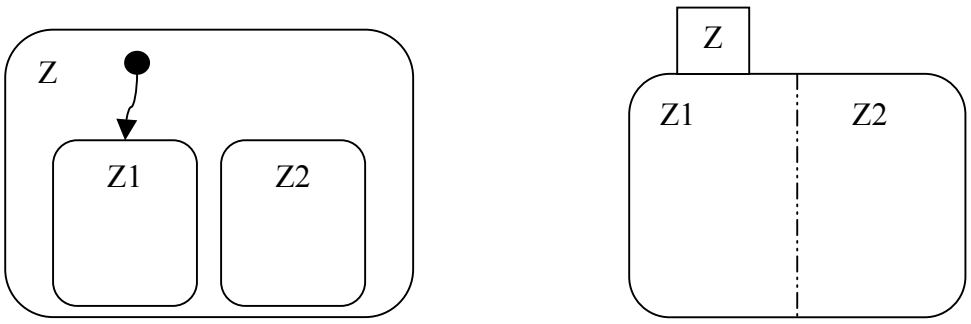
4. Für flache Zustandsdiagramme ist es nicht möglich, die in einem System benötige Parallelität direkt zu modellieren, weil sich das System zu jedem Zeitpunkt in genau einem Zustand befindet, also sequentiell ist. In Statecharts könnte ein Zustand zugleich mehrere aktive Unterzustände haben, damit wird die Modellierung eines parallelen Systems unterstützt. Es wird im nächsten Abschnitt auf die wichtigen Konzepte von Statcharts eingehen.

2.9 Statecharts

Gewisse Nachteile der flachen Zustandsdiagramme wurden im letzten Abschnitt gezeigt. In diesem Abschnitt werden die Type von Zuständen, Transitionen und Konnektoren in Statecharts erläutern.

2.9.1 Arten von Zuständen

Zur Erweiterung für flache Zustandsdiagramme gibt es in Statecharts drei Arten [4] von Zuständen, Basis-Zustände, OR-Zustände und AND-Zustände. Die Basis-Zustände sind mit den Zuständen der flachen Zustandsdiagramme identisch. OR- und AND-Zustände wurden eingeführt, um die Konzepte der Hierarchie und Parallelität zu unterstützen. Im Unterschied zu den Basis-Zuständen sind OR- und AND-Zustände zusammengesetzten Zustände, die weitere Unterzustände besitzen könnten.



(a)OR-Zustand

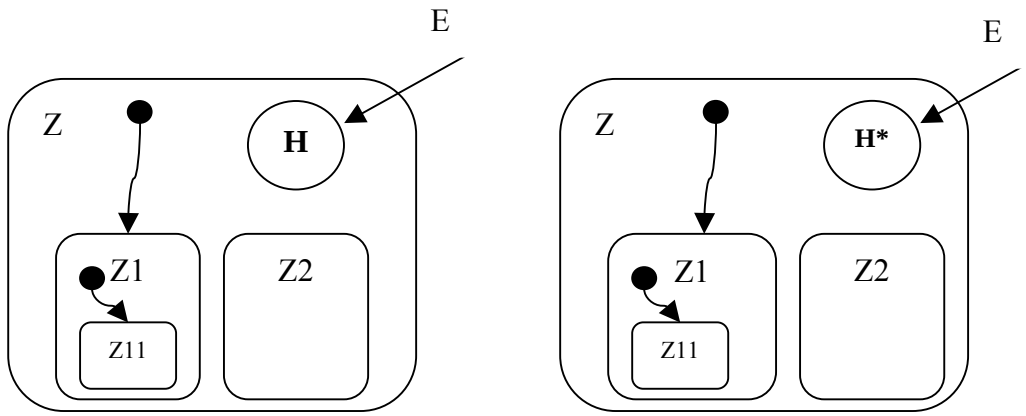
(b)AND-Zustand

Abbildung 2-7

Ein OR-Zustand wie der Zustand Z in Abbildung 2-7 (a) kann eine Menge von Unterzuständen wie die Zustände Z1 und Z2 enthalten. Die Unterzustände können Basis-, OR- und AND-Zustände sein. Die OR-Zustände heißen noch exklusive OR-Zustände, weil genau ein Unterzustand vom OR-Zustand in der Laufzeit aktiv sein darf, wenn der OR aktiv ist. Die AND-Zustände haben eine Menge von Unterzuständen, die nicht von demselben Typ AND sein dürfen. Wenn ein AND-Zustand wie der Zustand Z in Abbildung 2-7 (b) aktiv ist, sind seine alle direkten Unterzustände wie die Zustände Z1 und Z2 gleichzeitig aktiv. Die direkten Unterzustände eines AND-Zustands heißen noch orthogonale Komponente. Der Begriff der direkten Unterzustände bedeutet, dass sich ein Zustand in der ersten Stufe der Hierarchie von einem zusammengesetzten Zustand befindet. Die sich nicht in der ersten Hierarchiestufe befindenden Unterzustände heißen im Gegensatz dazu indirekte Unterzustände. Nach der Definition vom OR-Zustand muss ein Unterzustand eines OR-Zustands aktiv sein, wenn der OR-

Zustand aktiv ist. Ein solcher Unterzustand im OR-Zustand heißt Default-Zustand wie der Zustand Z1.

Ein OR-Zustand kann eine nützliche Erweiterung [4] haben, die Historys heißen. Ein History ermöglicht beim Betreten eines OR-Zustands den letzt verlassenen aktiven Unterzustand dieses OR-Zustands zu aktivieren. Dabei unterschieden sich zwei Typen von Historys. Einer heißt Shallow-History, die sich nur auf die direkten Unterzustände eines OR-Zustands bezieht. Der andere Typ heißt Deep-History, die sich auf alle mögliche Unterzustände eines OR-Zustands bezieht. Graphisch wird in Abbildung durch H und H* dargestellt wie in Abbildung 2-8.



(a) Einfache History (b) Deep-History

Abbildung 2-8

Wenn der Zustand Z in beiden Statecharts in Abbildung 2-8 noch nie aktiv war, führt das zur indirekten Aktivierung des Zustands Z1 (Default-Zustand). Shallow-History und Deep-History werden in diesem Fall nicht beachtet. Wenn der Zustand Z bereits einmal aktiv war, dann unterscheiden sich die beide Historys. Z. B. es wird angenommen, dass der letzte Zustand Z11 in beiden Statecharts aktiv war. Wenn das Ereignis E bei den beiden Historys auftritt, hier unterscheiden sich die zu aktivierenden Unterzustände. Für das einfache History wird der Zustand Z1 aktiviert, weil sich die Shallow-Historys nur auf direkte Unterzustände eines OR-Zustands beziehen. Im Gegensatz dazu wird für das Deep-History der Zustand Z11 aktiviert, da sich die Deep-Historys auf alle Unterzustände eines OR-Zustands beziehen.

2.9.2 Zusammengesetzte Zustandsübergänge

Einfache Zustandsübergänge zwischen zwei Zuständen wurden in den flachen Zustandsdiagrammen vorgestellt. Statecharts erweitern die flachen Zustands-diagramme durch zusammengesetzte Zustandsübergänge. Ein zusammengesetzter Zustandsübergang könnte von mehreren separaten Übergängen zusammengefasst werden, die gleichzeitig auf unterschiedlichen orthogonalen Regionen auftreten könnten. Jeder Zustandsübergang besteht wieder aus einer Reihe von Segmenten, die sind beschriftet und verbinden Zustände und Konnektoren. In diesem Abschnitt werden die verschiedenen Arten von Konnektoren und deren Nutzungen vorgestellt. Die Konnektoren in Statecharts unterscheiden sich in zweit Typen [4,6,7]: AND und OR.

Bei den AND-Konnektoren teilnehmen sich alle Segmente, die demselben AND-Konnektor angehören. Fork- und Join-Konnektor sind vom Typ AND und werden in folgenden vorgestellt.

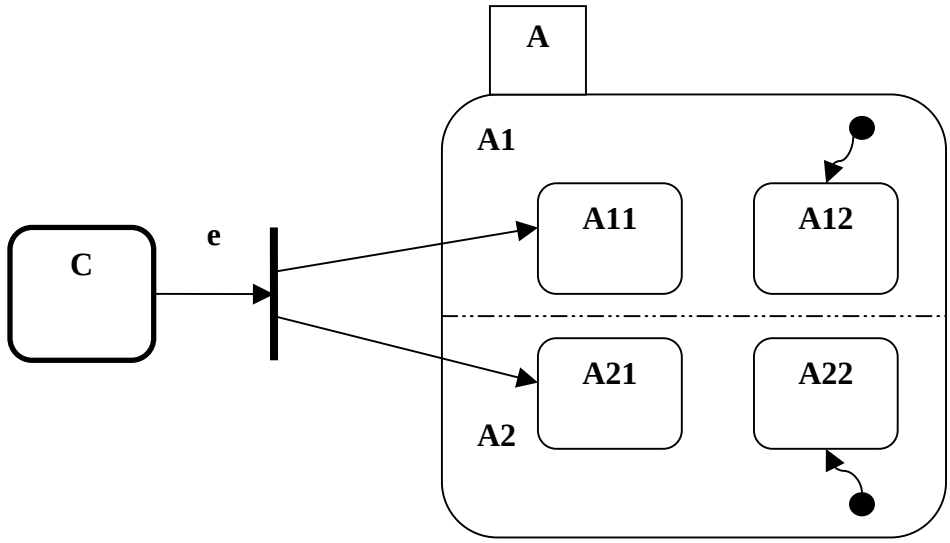


Abbildung 2-9

Ein Fork-Konnektor ist wie in Abbildung 2-9. Wir nehmen an, ein Objekt befindet sich jetzt im Zustand C und das Ereignis e tritt ein, das löst aus, den Fork-Zustandsübergang auszuführen. Während dieses Übergangs wird zuerst der aktive Zustand C verlassen, dann werden die Zustände in der Sequenz von {A, A1, A2, A11, A21} betreten. Dieser Fork-Zustandsübergang bedeutet, dass sich die beiden Segmente nach den A11 und A21 gleichzeitig an diesem zusammengesetzten Übergang teilnehmen müssen. Der Zielzustand des Fork-Zustandsübergangs muss ein Zustand oder ein History sein. Die Segmente von Fork-Zustandsübergängen zu Zielzustände dürfen nicht beschriftet werden.

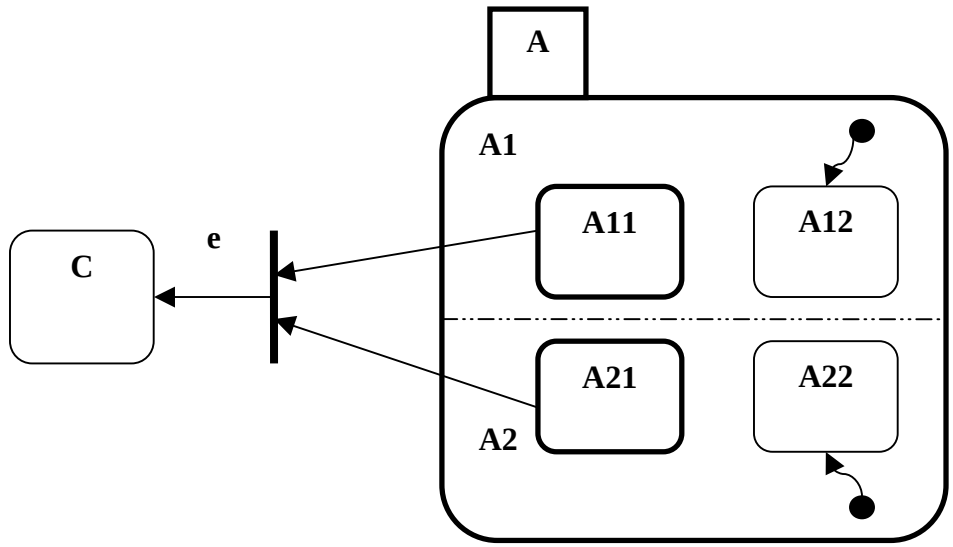


Abbildung 2-10

Ein Join-Konnektor ist wie in Abbildung 2-10 dargestellt. Es wird angenommen, dass sich ein Objekt in Zustände A11 im A1 und A21 im A2 befindet und ein Ereignis e eintritt, das löst aus, den Zusammengesetzten Übergang zum C auszuführen. Während des Übergangs werden zuerst die Zustände in der Sequenz von {A11, A21, A1, A2, A} verlassen, dann wird der Zustand C betreten. Der Join-Zustandsübergang bedeutet, dass sich alle Segmente von den A11 und A21 nach dem Join-Konnektor an diesem zusammengesetzten Übergang teilnehmen müssen. Die Segmente von den aktiven Zuständen nach dem Join-Konnektor dürfen nicht beschriftet werden.

Bei den OR-Konnektoren teilnehmen sich genau ein einkommendes Segment und ein auskommendes Segment an einem zusammengesetzten Übergang, obwohl dieser Übergang mehrere Segmente besitzen könnte. Junktion- und Bedingung-Konnektor sind von Typ OR, werden in folgenden vorgestellt.

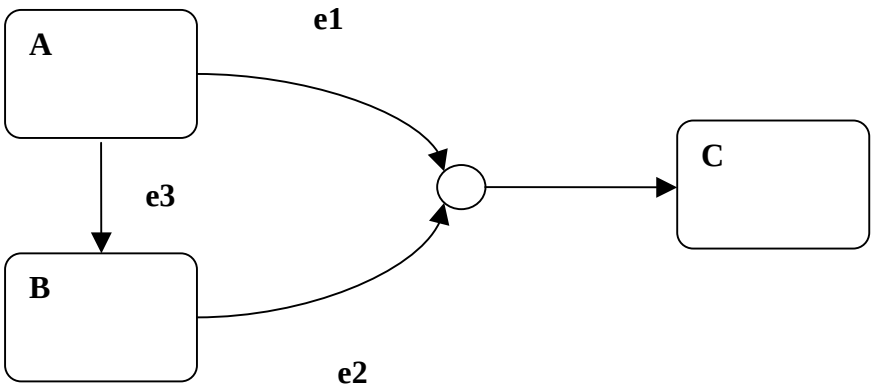


Abbildung 2-11

Ein Junktion-Konnektor ist wie in Abbildung 2-11 dargestellt. Der besitzt genau ein auskommendes Segment und mehrere einkommende Segmente. Befindet sich ein Objekt im Zustand A und tritt das Ereignis e1 ein, dann wird der Übergang vom Zustand A zum Zustand c ausgeführt. Während dieses Übergangs wird zuerst der Zustand A verlassen und dann den C betreten. Eigentlich ist das in Abbildung 2-11 dargestellte Statechart mathematisch gleich wie das Statechart in Abbildung 2-12.

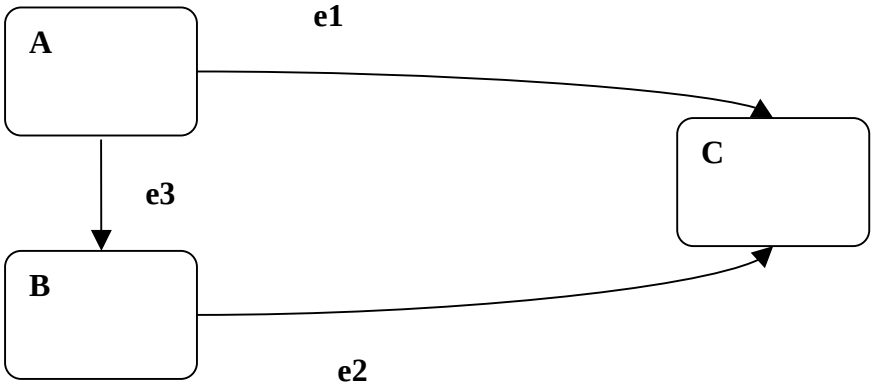


Abbildung 2-12

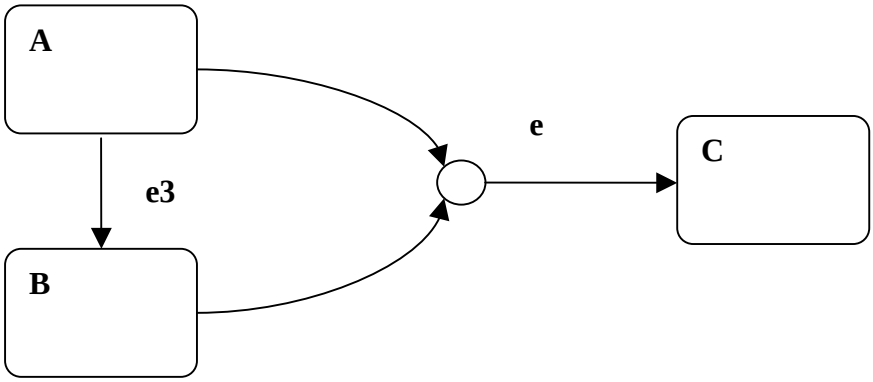


Abbildung 2-13

Ein Bedingung-Konnektor ist wie in Abbildung 2-14 dargestellt. Der hat mehrere einkommende Segmente und genau ein auskommendes Segment. Abzweige sind mit Bedingungen beschriftet. Ein Abzweig kann nur ausgewählt werden, wenn seine Bedingung wahr ist. Da der Bedingung-Konnektor vom Typ OR ist, wenn mehrere Abzweige gleichzeitig wahr sind, wird genau ein Abzweig daraus auswählt. Jeder Bedingung-Konnektor kann einen besonderen „else“ Abzweig besitzen, der wird ausgewählt, wenn alle andere Abzweige nicht wahr sind. Ein Abzweig kann wieder mehrere Abzweige besitzen, jeder Abzweig in einem Bedingung-übergang kann auch mehrere Aktionen besitzen.

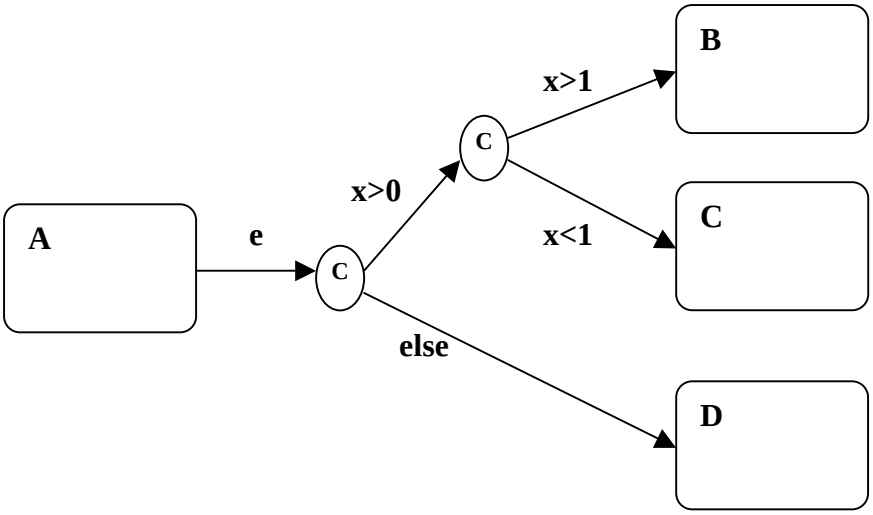


Abbildung 2-14

3 Klassendiagramme von Statecharts

Im letzten Kapitel wurden die Grundlage von Statecharts vorgestellt, z. B. die Arten von Zuständen sind Basis-Zustände, OR-Zustände, AND-Zustände, Startzustände und Historys, und die Arten von Transitionen sind einfache Transitionen, Join-Transitionen und Fork-Transitionen. In diesem Kapitel wird erläutern, wie diese Elemente von Statecharts implementiert werden. Für Transitionen liegt der Schwerpunkt bei den Algorithmen, da die hierarchischen Zustände und Orthogonale Regionen ermöglichen, dass mehrere Transitionen gleichzeitig möglich sind, und jede Transition könnte wieder auch mehrere Segmente besitzen. Deswegen sollen die Konflikte der Transitionen beachtet werden. Manche Konflikte können mit ihren Prioritäten behandelt werden, manche können dagegen nicht. Andererseits bei den Transitionen soll das Prinzip run-to-completion (RTC) beachtete werden, da andere Ereignisse während der Transitionen eintreten könnten.

3.1 Generalisierung von Zuständen

Da die Zustände in Statecharts Unterzuständen besitzen dürfen und jeder Zustand daher genau einen Vaterzustand haben darf, soll hier der *composite design pattern* [8] verwendet werden, um die Beziehungen zwischen dem Vaterzustand und seinen Unterzuständen zu beschreiben. Die realen Zustände, die mehrere Unterzustände haben dürfen, sind AND-Zustände und OR-Zustände. Die Zustände, die keine Unterzustände haben dürfen, sind Basis-Zustände. Ein Objektdiagramm ist wie in Abbildung 3-1.

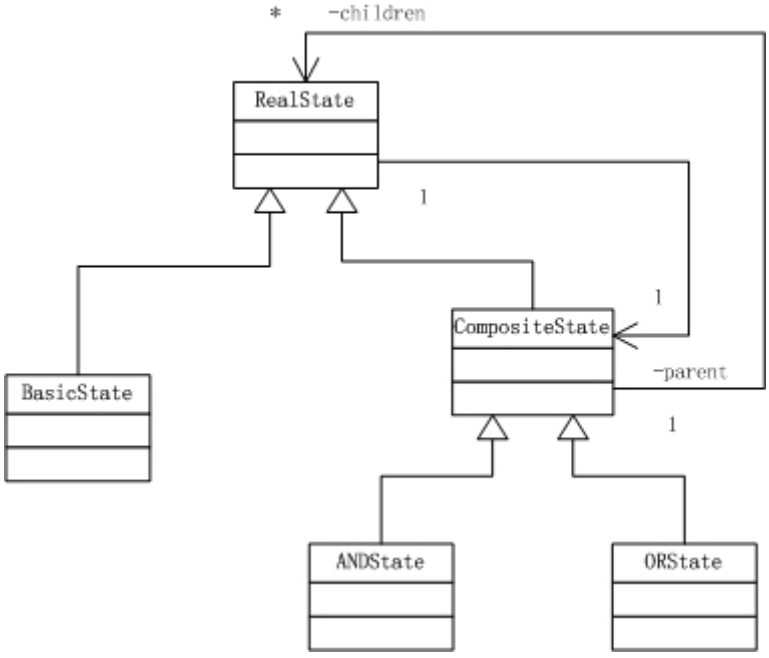


Abbildung 3-1

In Statecharts gibt es außer den realen Zuständen noch Pseudozustände wie z. B. Initial und Hisotry. Die beide sollen keine direkten Unterklassen von der Klasse RealState sein. Deswegen wird eine neue Klasse State als der Wurzel dieser statischen Struktur eingeführt. Die Abbildung 3-1 soll daher wie in Abbildung 3-2 abgeändert werden. Die Klasse History, Initial und RealState liegen jetzt auf der gleichen Schicht

als direkte Unterklasse von der Klasse State. Es wird somit dargestellt, dass die History und Initial keine realen Zustände sind und im Vergleich zu realen Zuständen unterschiedlich behandelt werden.

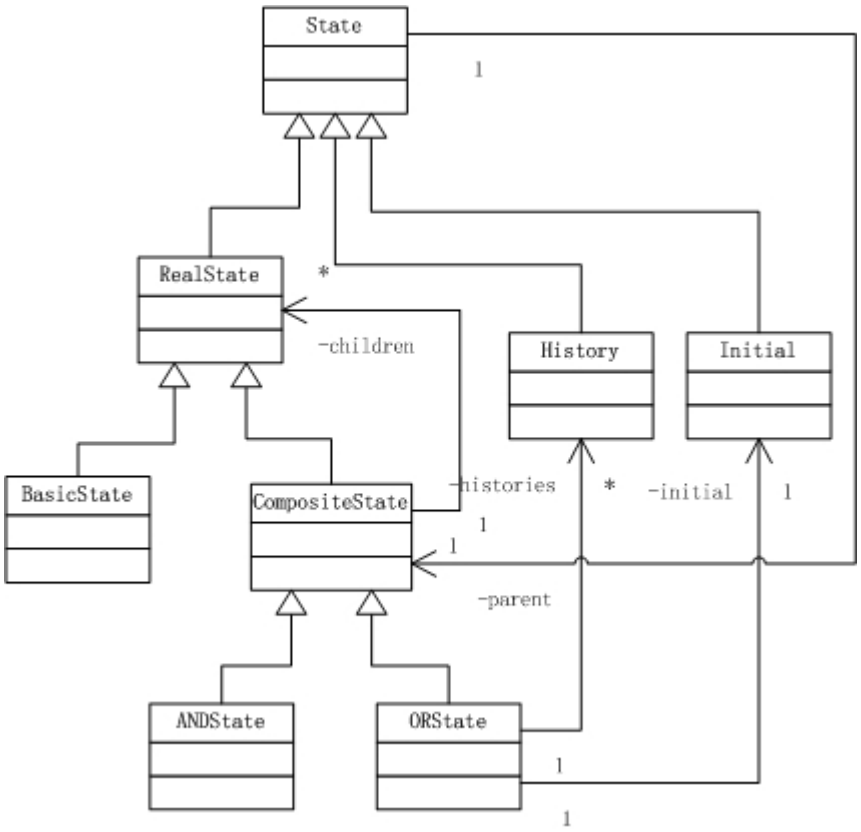


Abbildung 3-2

3.2 Zusammengesetzte Transitionen

Im Vergleich zu den flachen Zustandsdiagrammen ermöglichen Statecharts zusammengesetzte Transitionen, die mehrere Anfangszustände oder Zielzustände zugelassen werden. Damit kann eine Transition dazu führen, dass sich ein System in mehreren aktiven Anfangszuständen befindet, und dass am Ende der Transition mehrere Zielzustände gleichzeitig aktiv sind. Zusammengesetzte Transitionen dürfen mehrere Segmente besitzen, die mit den Transitionskonnektoren verbindet werden. Dabei unterscheiden sich zwei Typen von Konnektoren, AND-Konnektoren und OR-Konnektoren. Die AND-Konnektoren und OR-Konnektoren können sowohl eine Menge von mehreren UpSegmenten (vom Anfangzustand nach dem Konnektor) als auch eine Menge von mehreren DownSegmenten (vom Konnektor nach dem Zielzustand) besitzen. Das Objektdiagramm ist wie in Abbildung 3-3.

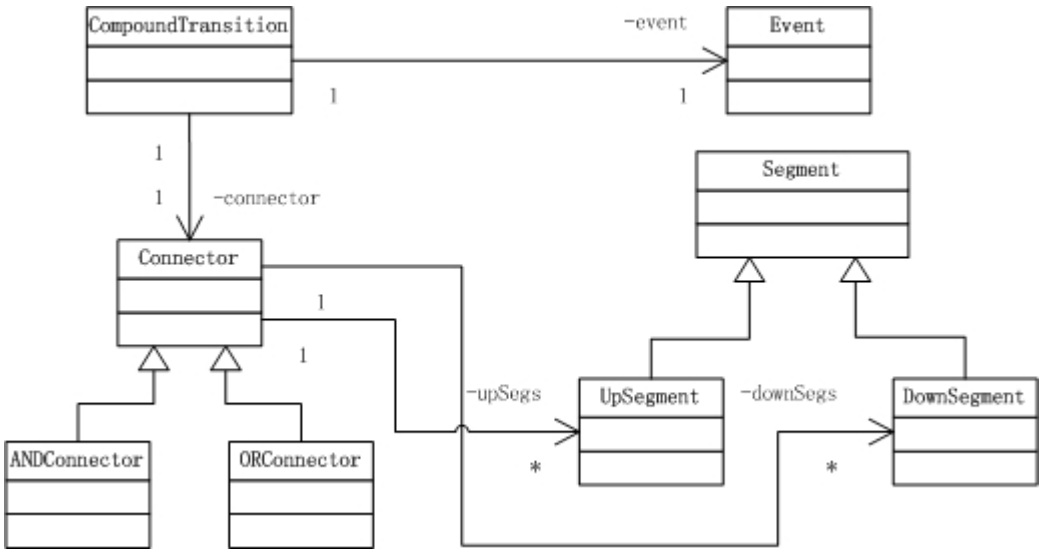


Abbildung 3-3

Der Typ von zusammengesetzten Transitionen hängt eigentlich vom Typ des Konnektor ab. Das heißt, wenn eine Transition den AND-Konnektor hat, ist der Typ dieser Transition die AND-Transition, und wenn eine Transition den OR-Konnektor hat, ist der Typ dieser Transition die OR-Transition. Im letzten Kapitel wurde der Unterschied zwischen AND-Transitionen und OR-Transitionen schon vorgestellt. In einer OR-Transition sollen sich genau ein UpSegment aus der Menge von UpSegmenten und ein DownSegment aus der Menge DownSegmenten teilnehmen. Im Gegensatz dazu sollen sich alle Segmente einer AND-Transition an dieser Transition teilnehmen, das heißt, erfüllen nur alle Segmente seine Bedingungen, dann kann diese Transition erfolgen. In der Laufzeit soll die Prüfung der Bedingung einer Transition an ihrem Konnektor delegiert werden, dieser Konnektor delegiert die Prüfung weiter an alle Segmente. Die Sequenz ist wie in Abbildung 3-4.

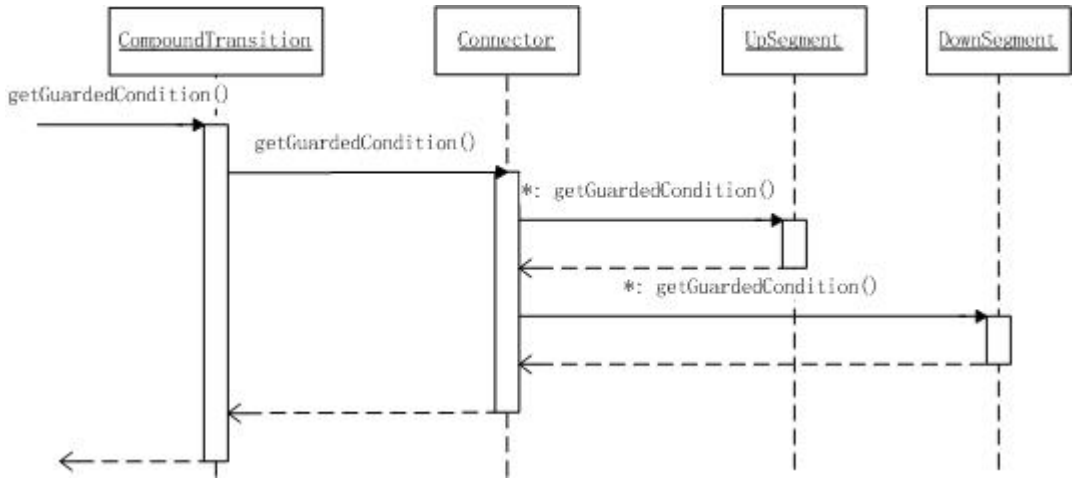


Abbildung 3-4

3.3 Schritte und Status

Die Ausführung einer Zustandmaschine besteht aus einer Reihe von Schritten [7] wie in Abbildung 3-5. Um jeden Schritt auszuführen wird zuerst der aktuelle Status des Systems ermittelt. Der Status kann als einen Schnappschuss des Systems verstanden werden, er besteht zumindest aus einer Menge von aktiven Zuständen des Systems und interne ausgelöste Ereignisse vom letzten Schritt, wenn die Ausführung der Schritte dem Prinzip RTC folgt.

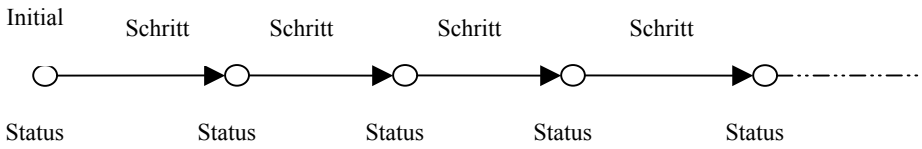


Abbildung 3-5

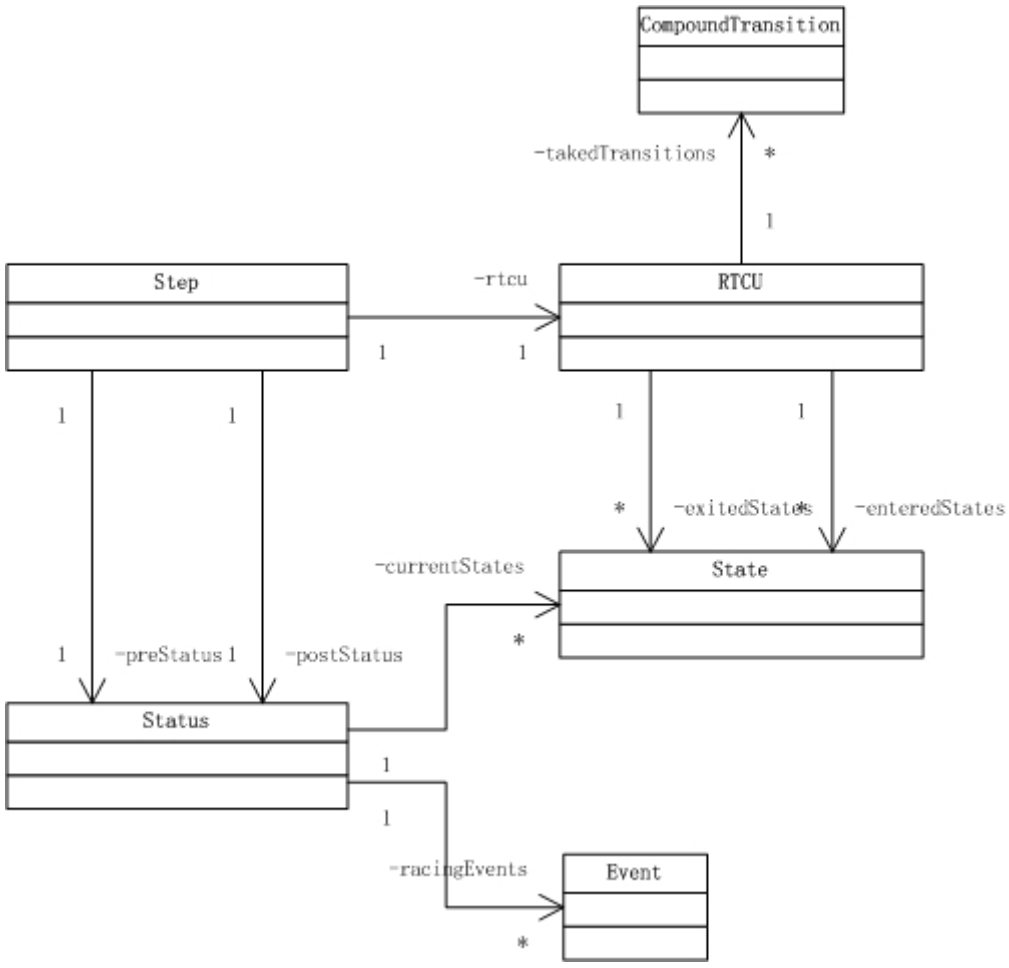


Abbildung 3-6

Das Objektdiagramm in Abbildung 3-6 erläutert die statischen Strukturen und Beziehungen zwischen den Klassen Step, Status und run-to-completion-unit (RTCU). Die Klasse Status hat zwei gerichtete Assoziationen.

Die currentStates-Assoziation beschreibt die Menge der aktiven Zustände, in den sich das System zurzeit befindet. Die racingEvent-Assoziation beschreibt die während der Transition entstehenden Ereignisse, die wegen des Prinzips RTC nicht sofort ausgelöst werden können. Die preStatus-Assoziation der Klasse Step beschreibt den aktuellen Status des Systems, während die postStatus-Assoziation den Status nach der Ausführung eines Schritts des Systems beschreibt. In die Klasse RTCU werden die auszutretende und einzutretende Zustände und die auszuführende Transitionen zusammengefasst, die während der Ausführung eines Schritts nicht unterbrechbar sind.

3.4 Zustandmaschine

In den obigen Abschnitten wurden viele Klassen und deren Beziehungen vorgestellt, die zur Implementierung von Statecharts benötigt werden. Eine Zustandmaschine kann daher als einen Verwalter betrachtet werden, der die Objekte von den obigen Klassen in der Laufzeit verwaltet und den Status vor jedem Schritt ermittelt. Nach der Ermittelung führt die Zustandmaschine die Schritte in der Sequenz aus und aktualisiert sie den Status des Systems nach jedem Schritt. Das Objektdiagramm ist in Abbildung 3-7.

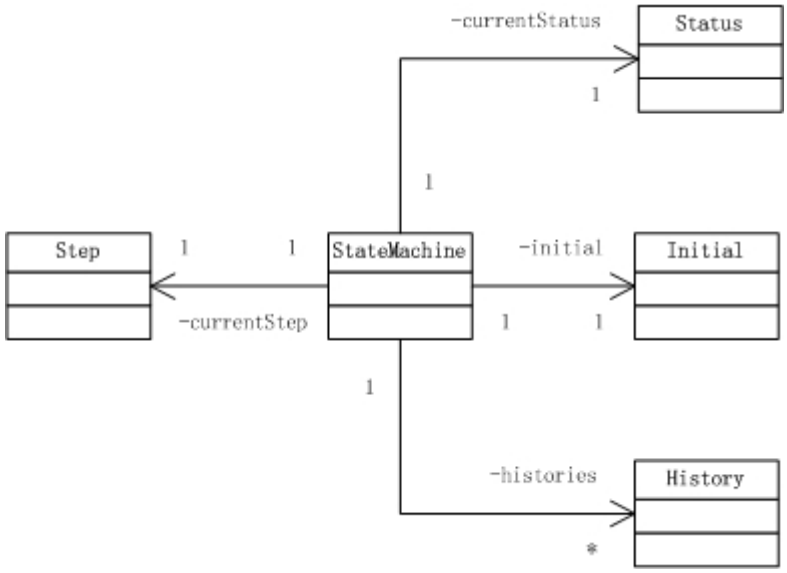


Abbildung 3-7

Die currentStatus-Assoziation beschreibt den aktuellen Status des Systems wie in dem letzten Abschnitt erklärt wurde, und die currentStep-Assoziation beschreibt den aktuellen Schritt. Der Zustand Initial beschreibt den Startzustand des Systems, wenn die Zustandmaschine hochfährt. Die histories-Assoziation beschreibt alle Histroy-Zustände im System, die nach der Ausführung jedes Schritts aktualisiert werden soll, wenn ihre Vaterzustände eingetreten werden.

3.5 Aktionen und Bedingungen

Für die Aktionen und Bedingungen sind in dieser Implementierung noch keine konkreten Klassen entworfen geworden. Denn die Aktionen und Bedingungen können mit Hilfe von pointcuts und advices in AspectJ realisiert und erweitert werden. Diese Methode ist sehr praktisch und effizient. Die wird in dem letzen Kapitel vorgestellt.

3.6 Übersicht aller Klassendiagramme

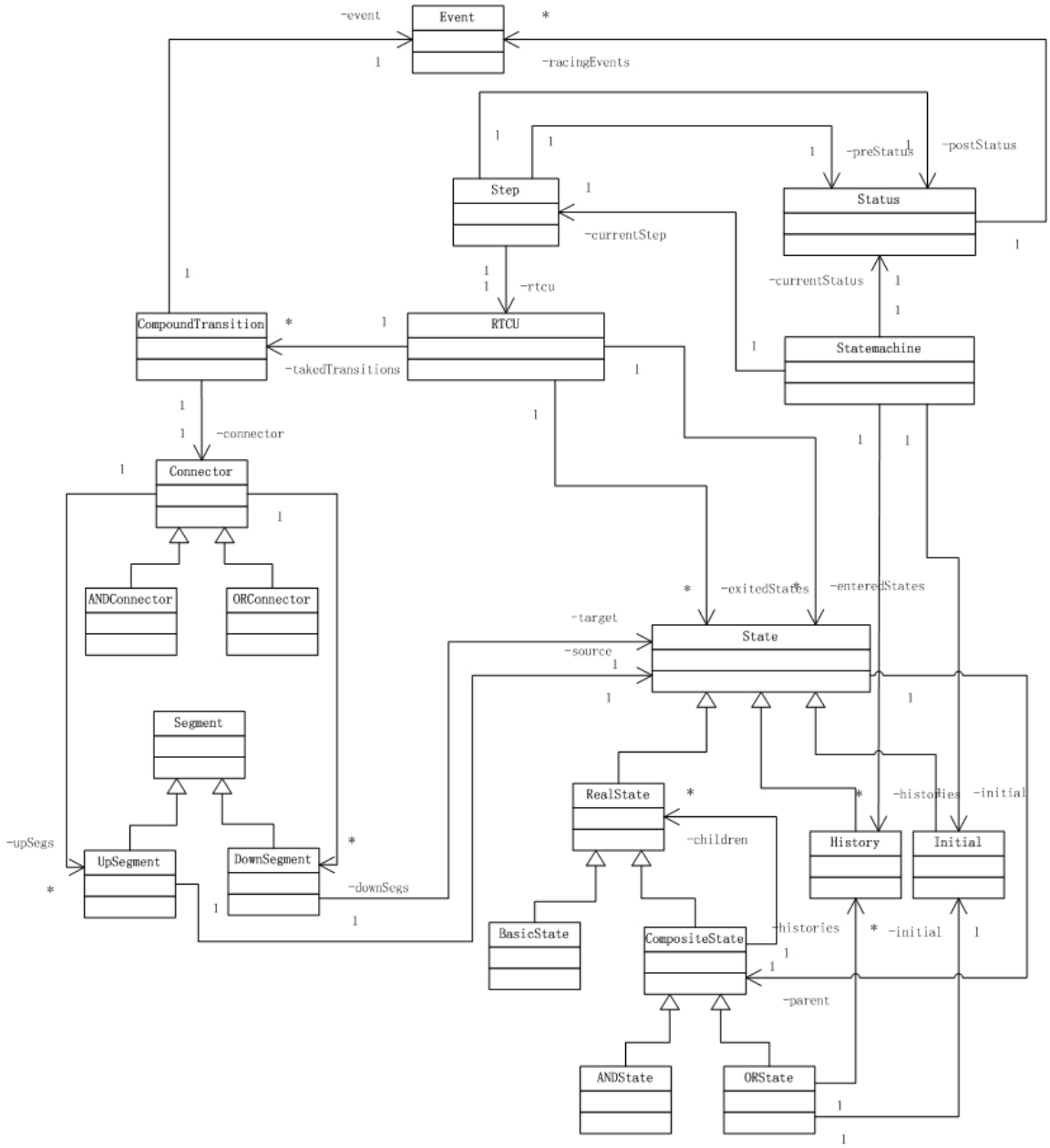


Abbildung 3-8

4 Algorithmen der Transitionen

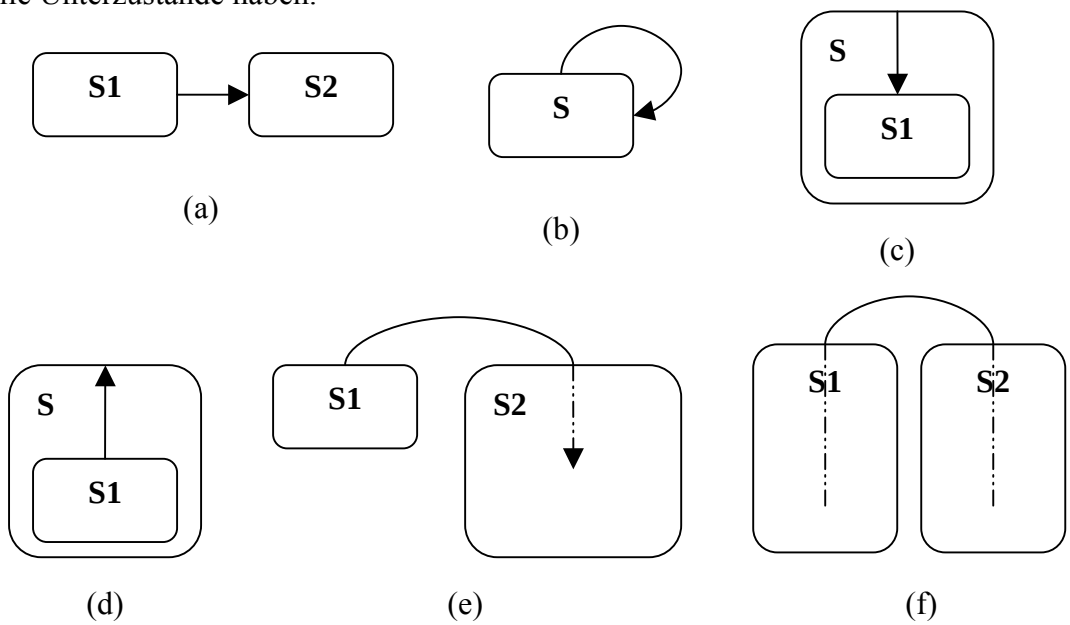
Im letzten Kapitel wurden die Klassendiagramme von Statecharts vorgestellt, die zur Transformation von Statecharts nach AspectJ notwendig sind. Dieser Kaptitel wird die Algorithmen erläutern, die zur Ausrechnung des Pfads von den aktiven Zuständen nach den Zielzuständen notwendig sind. Zur Ausrechnung wird zuerst eine maximale Menge von Transitionen ermittelt, die für die aktuellen aktiven Zustände möglich sind. Dabei soll es beachtet werden, dass ein Vaterzustand die Transitionen von seinem allen Unterzuständen übernehmen kann, wenn die Unterzustände selbe die Ereignisse nicht behandeln könnten. Nach der Ermittlung der maximalen Menge wird nur eine Teilmenge von Transitionen daraus ausgewählt, wenn die Bedingungen, also die Bedingungen der ermittelten Transitionen wahr sind. Dabei soll es auch beachtet werden, Wenn es gleichzeitig mehrere mögliche Transitionen gibt, sollen die Konflikte der Transitionen auch behandelt werden.

4.1 Lowest Common Ancestor

Die Zustände in Statecharts haben eine hierarchische Struktur wie in einem Baum. Ein LCA n der Menge X von Zuständen kann daher als den tiefsten Vorfahren verstanden werden, mathematisch dargestellt [5] wie im folgenden:

$$X \subseteq children^*(n)$$
$$\text{for all } n' \in N : X \subseteq children^*(n') \Rightarrow n \in children^*(n')$$

Die Funktion children*(n) stellt dabei die Menge, die den Zustand n selben und seine alle Unterzustände haben.



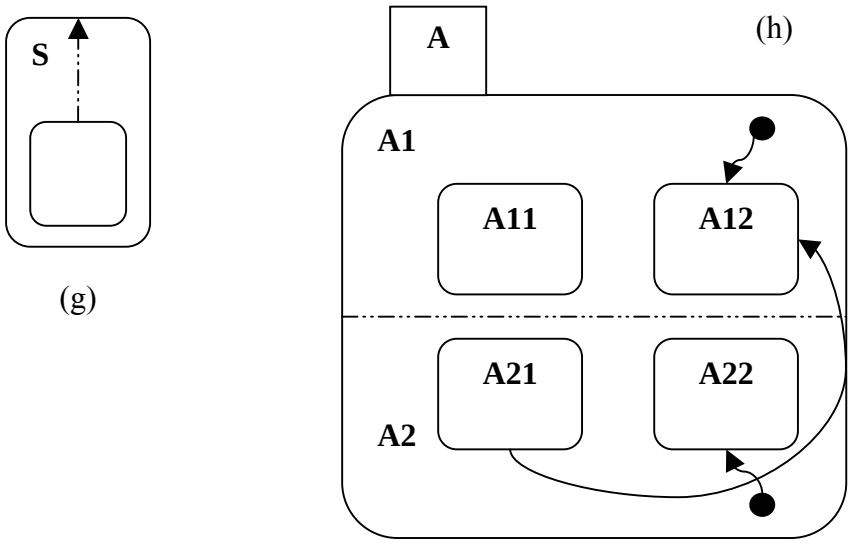


Abbildung 4-1

Der LCA ist sehr wichtig, weil der Ausführungskontext Scope in Abschnitt 4.2 einer Transition in Zusammenhang mit dem LCA steht. Es wird daher der Begriff LCA durch Beispiele in Abbildung 4-1 weiter erläutert. Die punktierte Linie in Abbildung 4-1 bedeutet, dass die Zielzustände nicht die direkten Unterzustände sind. In Abbildung 4-1 (a), (e) und (f) ist der LCA der Vaterzustand der Zustände S1 und S2. In (b) ist der LCA der Zustand S selber. Der LCA in (c), (d) und (g) ist der Zustand S. Der LCA der Zustände A21 und A12 in (h) ist der AND-Zustand A.

Der Algorithmus: LCA
Eingabe: s1 – ein Zustand
s2 – ein Zustand
Ausgabe: LCA – ein Zustand (Lowst Common Ancestor)

```
Begin
    if (s1 == s2) return s1
    else if (s2 is a descendant of s1) return s1
    else if (s1 is a descendant of s2) return s2
    else
        set s1a := all ancestors of s1
        set s2a := all ancestors of s2
        set s12a := s1a ∩ s2a
        if (s12a is not empty)
            lca := the least element of s12a
            if (lca exist) return lca
            else return null
            endif
        endif
    endif
    return null
End
```

4.2 Scope

Im letzten Abschnitt wurde erwähnt, dass das Scope einer Transition im Zusammenhang mit dem LCA steht. Das Scope [6, 7] ist auch der tiefste Zustand in der Hierarchie von Zuständen während einer zusammengesetzten Transition. Das Scope (Proper LCA) unterscheidet sich mit dem LCA, wobei das Scope bei einer Transition nicht betreten und verlassen werden darf und gleichzeitig ein OR-Zustand sein muss. Mit Hilfe von Beispielen in Abbildung 4-1 wird der Unterschied erläutert. Das Scope in (a), (c), (d), (e), (f) und (g) ist gleich mit dem LCA. Das Scope in (b) ist nicht mehr das LCA, sondern der Vaterzustand vom Zustand S, weil der Zustand S zuerst betreten und danach wieder verlassen wird. Das Scope in (h) ist der Vaterzustand vom Zustand A, weil ein Scope ein OR-Zustand muss. Das Scope kann mit Hilfe vom LCA ausgerechnet werden. Der Algorithmus wird im folgenden beschrieben.

Der Algorithmus: Scope
Eingabe: s1 – ein Zustand
s2 – ein Zustand
Ausgabe: scope – ein Zustand (das Scope)

```
Begin
    If (s1 == s2) scope := the parent of s1
    else
        scope := lca(s1, s2)
    endif
    if (scope != null)
        if (scope is an ANDState)
            scope := the parent of scope
        endif
    endif
    return scope
End
```

4.3 Aktive Konfiguration

Eine aktive Konfiguration [6, 7] ist eine maximale Menge von Zuständen, die in einem System gleichzeitig aktiv sein können. Diese Menge muss den Wurzel Zustand enthalten, und genau ein direkter Unterzustand jedes OR-Zusands gehört zu dieser Menge, und alle direkte Unterzustände eines AND-Zustands gehören zu dieser Menge. Ein Beispiel in Abbildung 4-2 wird diese Definition erläutern. Beispielweise ist eine Menge von Zustände {C, Top} eine richtige aktive Konfiguration, aber eine Menge von Zustände {A, Top} ist keine richtige aktive Konfiguration, da alle ihre direkten Unterzustände auch aktiv sein, wenn ein AND-Zustand aktive ist. Die Menge {A1, A2, A, Top} ist auch keine richtige aktive Konfiguration, da genau ein direkter Unterzustand jedes OR-Zustands aktive sein muss, wenn ein OR-Zustand aktiv ist. Die Menge {A12, A22, A1, A2, Top} ist eine aktive Konfiguration. Der Algorithmus wird im folgenden beschrieben.

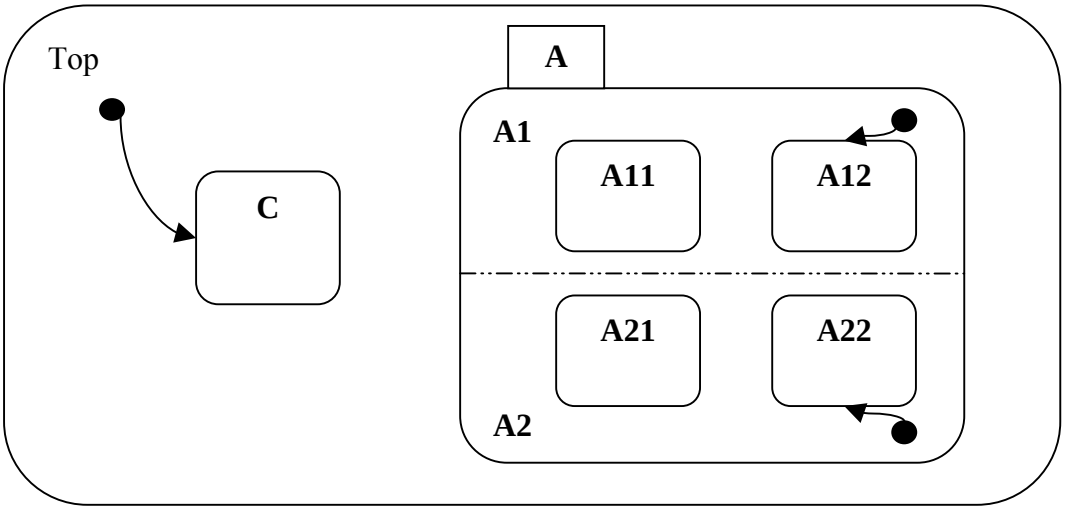


Abbildung 4-2

Der Algorithmus: MaxConfiguration
Eingabe: states – eine Menge von Zuständen
root – ein Wurzelzustand
Ausgabe: maxConfiguration – eine Menge von Zuständen

```
Begin
    for all s in states
        if (s is not in maxConfiguration)
            while( s is descendant of root)
                add s to maxConfiguration
                s := the parent of s
            endwhile
        endif
    endfor
    return maxConfiguration
End
```

Der Algorithmus: MaxConfigurations
Eingabe: states – eine Menge von Zuständen
roots – eine Menge von Wurzelzuständen
Ausgabe: maxConfigurations – eine Menge von Zuständen

```
Begin
    for all r in roots
        set ss := MaxConfiguration(states, r)
        add ss to maxConfigurations
    endfor
    return maxConfigurations
End
```

4.4 Das Segment von Anfangszuständen nach Scope

Das Segment ist der Pfad von den Anfangszuständen nach dem Scope. Das Scope ist exklusiv. Das heißt, dass sich das Scope nicht in diesem Pfad befindet, weil das Scope während der Transition nicht verlassen wird. Beispielweise das Segment in Abbildung 4-3 ist {A11, A21, A}, und das Scope ist der Zustand Top, wenn sich das System in den Zuständen A11 und A21 befindet und das Ereignis e eintritt.

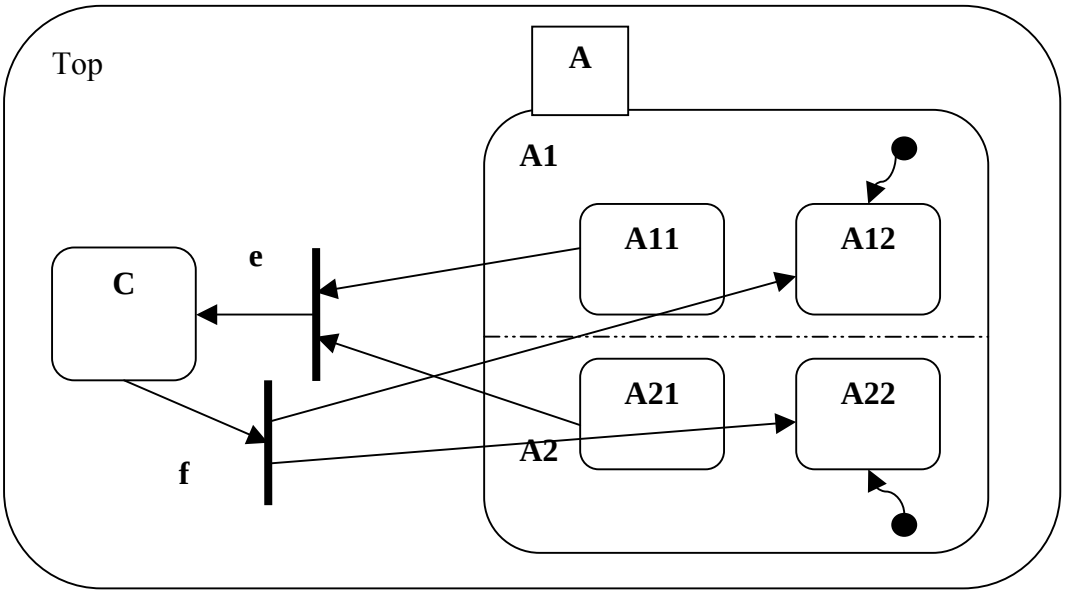


Abbildung 4-3

Der Algorithmus: UpSegment
Eingabe: sourceStates – die Menge der Anfangszustände
scope – der Zustand scope
Ausgabe: upSeg – die Menge des Pfads von Anfangszuständen nach dem Scope
Begin
upSeg = MaxConfiguration(sourceStates, scope)
remove scope from upSeg
return upSeg
End

4.5 Das Segment von Scope nach Zielzuständen

Im Gegensatz zu dem letzten Abschnitt ist das Segment der Pfad vom Scope nach den Zielzuständen. Das Scope befindet sich nicht in diesem Pfad, weil das Scope nicht eingetreten werden darf. Beispielweise befindet sich das System wie in Abbildung 4-3 im Zustand C und tritt das Ereignis f auf. Das Segment ist {A, A12, A22}.

Der Algorithmus: DownSegment
Eingabe: targetStates – die Menge der Anfangszustände
scope – der Zustand scope
Ausgabe: downSeg – die Menge des Pfads von den Anfangszuständen nach dem Scope
Begin
downSeg = MaxConfiguration(targetStates, scope)

remove scope downSeg
return downSeg
End

4.6 Verlassene Zustände

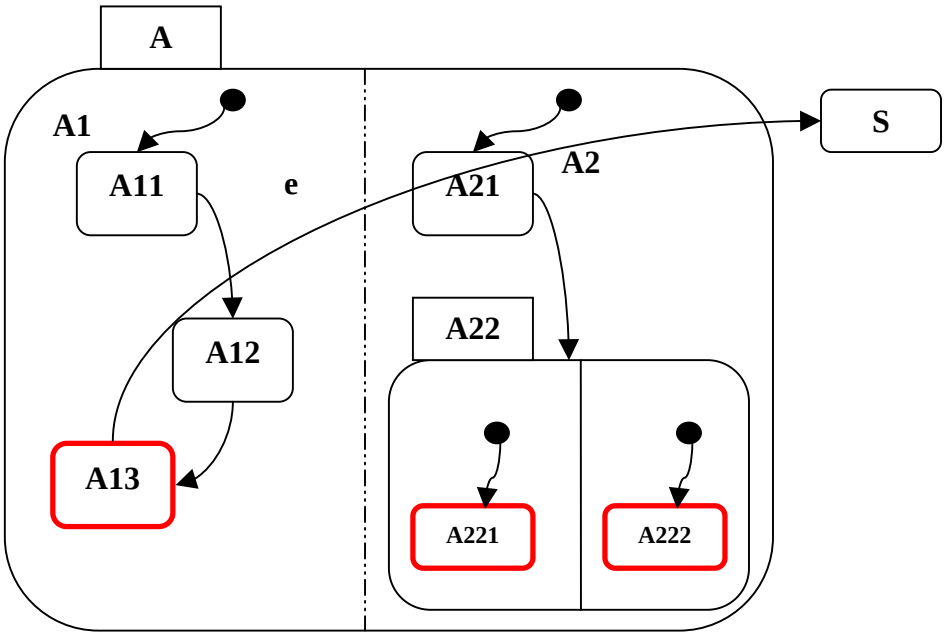


Abbildung 4-3

In den flachen Zustandsdiagrammen muss bei den Transitionen nur genau einen aktiven Zustand verlassen werden. Bei den Transitionen in Statecharts sollen aber nicht nur die Anfangzustände der Transitionen sonder auch andere aktive Zustände behandelt werden. Weil es groß möglich ist, dass eine Transition parallele Regionen durchquert, die sind Vorfahren von anderen aktiven Zuständen sind.

Es wird angenommen, die aktuelle aktive Zustände in Abbildung 4-3 sind A13, A221 und A222. Wenn ein Ereignis e auftritt, die Transition vom Zustand A13 nach dem S ist möglich. Aber die zu verlassene Zustände sind nicht nur {A13, A1, A} sonder auch {A221, A222, A22, A2}. Der Algorithmus dafür wird im folgenden beschrieben.

Der Algorithmus: ExitedStates
Eingabe: upSeg – die Menge der Zustände des Pfads vom Anfangs- zustand nach dem Scope (exklusiv) in einer Transition
currentStates – die Menge der aktiven Zustände
Ausgabe: exitedStates – die Menge der auszutretende Zuständen

Begin
Add all states in Upseg to exitedStates
for all s in upSeg
if (s is an ANDState)
add all direct substates to exitedStates

```
        endif
    endfor
    Add all states in exitedStates to a new set toDo
    for all s in toDo
        for all c in currentStates
            if (c is a descendant of s)
                while(c != s)
                    add c to exitedStates
                    c := the parent of c
                endwhile
            endif
        endfor
    endfor
    return exitedStates
End
```

4.7 Betretene Zustände

In Statecharts sind Pseudozustände wie Initial-Zustand, History möglich, die sind aber keine echten Zielzustände. Beispielweise, ein History hängt von dem letzten aktiven Unterzustand in einem OR-Zustand ab, und ein Unterzustand aktiv sein muss, wenn ein OR-Zustand aktiv ist. Das führt zum Problem der Bestimmung der Zielzustände.

Der Algorithmus: ResolveTargetStates

Eingabe: targets – die Menge der Zielzustände
histories – die Geschichte im letzten Status des Systems.

Ausgabe: resolvedTargetStates – die Menge der endlichen Zielzustände

```
begin
    set toDo := targets
    for all t in toDo
        if (t is a BasicState)
            add t to resolvedTargetStates
        else if (t is a ORState)
            add initial to toDo
        else if (t is an ANDState)
            add all substates to toDo
        else if (t is a History)
            if (t in histories)
                add t to toDo
            else
                add initial of its parent to toDo
            endif
        endif
    endfor
    return resolvedTargetStates
end
```

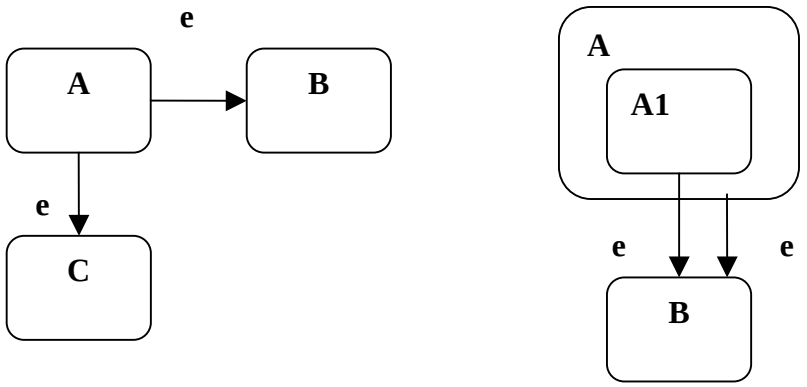
Der Algorithmus: EnteredStates

Eingabe: downSeg – die Menge von Zuständen des Pfads vom Scope (exklusiv) nach den Zielzuständen in einer Transition
targets – die Menge der Zielzustände
histories – die Geschichte vom letzten Status des Systems.
Ausgabe: enteredStates – die Menge der einzutretenden Zielzuständen

Begin
set regions := null
set resolvedTargets : = null
for all s in downSeg
if (s is an ANDState)
add all substates of the ANDState to regions
endif
endfor
regions := regions – downSeg
enteredStates := ResolvedTargetStates(regions ∪ targets, histories)
enteredStates := MaxConfigurations(enteredStates, scope)
remove the scope from enteredStates
return enteredStates
End

4.8 Konflikte der Transitionen

In Statecharts könnten mehrere Transitionen gleichzeitig möglich sein. Es soll daher vor der Ausführung der Transitionen geprüft werden, ob die Transitionen in Konflikt stehen könnten. Nach der Definition von UML stehen zwei Transitionen nur in Konflikt, wenn die beiden Transitionen gleiche Zustände verlassen.



(a) (b)

Abbildung 4-4

In Abbildung 4-4 (a) [6, 7], wenn sich das System im Zustand A befindet und das Ereignis e eintritt, dann sind zwei Transitionen {A, B} und {A, C} gleichzeitig möglich. Weil die beiden Transitionen den gleichen Zustand A verlassen, stehen die beiden Transitionen in Konflikt. Für deterministische Algorithmen soll eine beliebige Transition daraus ausgewählt und ausgeführt werden. In Gegensatz dazu soll ein System, das die nicht deterministischen Algorithmen verwendet, den Konflikt zu zeigen oder Ausnahmen auszulösen. Es wird somit dem Benutzer den Konflikt benachrichtigt.

In Abbildung 4-4 (b), befindet sich das System im Zustand A1 und tritt das Ereignis e auf, dann sind zwei Transitionen {A1, B} und {A, B} gleichzeitig möglich. Beispielweise in Rhapsody [7] soll der Unterzustand die höhere Priorität als seinem Vaterzustand haben, das heißt, dass die Transition {A1, B} in diesem Fall genommen wird.

Der Algorithmus: inConflict

Eingabe: upSeg1 – die Menge der Zustände des Pfads vom Anfangszustand nach dem Scope (exklusiv) in einer Transition
upSeg2 – die Menge der Zustände des Pfads vom Anfangszustand nach dem Scope (exklusiv) in einer anderen Transition
currentStates – die Menge der aktiven Zustände

Ausgabe: boolean – ob die beide Transition in Konflikt stehen

```
begin
    exitedStates1 := ExitedStates(upSeg1, currentStates)
    exitedStates2 := ExitedStates(upSeg2, currentStates)
    intersection := exitedStates1 ∩ exitedStates2
    if ( intersection is empty )
        return false
    else
        return true
    endif
end
```

5 Übersetzung von Statecharts nach AspectJ

In diesem Kapitel wird erläutern, wie Statecharts nach AspecJ übersetzt werden. AspectJ ist eine aspektorientierte Programmierungssprache, mit Hilfe von AspectJ Compiler können Aspekte mit normalen Java-Bytecode verbunden werden. Die Nutzung von Pointcuts und Advices in AspectJ kann normale Methodenaufrufe nach Ereignisse übersetzen. Das heißt auch, wir können mit Hilfe von Aspekte die Systeme, die nicht ereignisbasiert sind, in ereignisbasierte Systeme umzuwandeln.

5.1 Übersetzung von Aufrufen nach Ereignissen

In dem 1. Kapitel wurde erwähnt, die Object Modelling Technique (OMT) stellt eine objektorientierte Entwicklungsmethodologie vor, bei der ein Softwaresystem mit drei Modellen beschrieben wird. Das 1. Objektmodell beschreibt mit Hilfe der Objektdiagramme die statische Struktur, Objekte und Relationen zwischen Objekten des Systems. Das zweite dynamisches Modell beschreibt mit Hilfe von Zustandsdiagrammen die zeitlichen Veränderungen des Systems. Das dritte funktionale Modell beschreibt schließlich mit Hilfe von Datenflussdiagrammen die Transformation der Daten innerhalb des Systems. Beispielweise gibt es ein Klasse One wie in Abbildung 5-1,

```
public class One {
    public void a() {
        System.out.println("a()");
    }
    public void b() {
        System.out.println("b()");
    }
    public void c() {
        System.out.println("c()");
    }
    public void d() {
        System.out.println("d()");
    }
}
```

Abbildung 5-1

Zum Zweck der Wiederverwendbarkeit wird die zeitliche Veränderung des Objekts One nicht vorher definiert. Die Idee ist, Wenn ein Statechart A mit dem Objekt One verbunden wird, soll sich das Objekt auf den Methodenaufrufen reagieren wie im Statechart A beschrieben. Wenn das Statechart A später mit einem neuen Statechart B getauscht wird, soll sich das Objekt reagieren wie im Statechart B beschrieben. Mit Hilfe von AspectJ können die Methodenaufrufe des Objekts One nach Ereignisse zu übersetzen. Solche Ereignisse werden in UML als Aufrufensereignisse (CallEvents) genannt. Der Empfang der Methodenaufrufe wird durch Pointcuts und Advices in AspectJ erreicht, beispielweise wie in Abbildung 5-2.

```
public aspect OneStatechart extends StateMachine {

    pointcut aPointcut(One o) : execution(public void a()) && this(o);

    pointcut bPointcut(One o) : execution(public void b()) && this(o);

    pointcut cPointcut(One o) : execution(public void c()) && this(o);

    pointcut dPointcut(One o) : execution(public void d()) && this(o);

    void around(One o)  : aPointcut(o) {
        proceed(o);
    }

    void around(One o)  : bPointcut(o) {
        proceed(o);
    }

    void around(One o)  : cPointcut(o) {
        proceed(o);
    }

    void around(One o)  : dPointcut(o) {
        proceed(o);
    }


}
```

Abbildung 5-2

Ein Poincut identifiziert einen oder auch mehrere Join Points. Z. B. das Pointcut

```
pointcut aPointcut(One o) : execution(public void a()) && this(o);
```

identifiziert die Methode a() vom Objekt der Klasse One.

Das Around Advice bedeutet, wenn die Methode a() aufgerufen wird, dies Advice

```
void around(One o)  : aPointcut(o) {
    proceed(o);
}
```

wird anstelle der Methode a() ausgeführt. In einem Around Advice kann weiter entscheiden, ob die ursprünglichen Methodenaufrufe noch ausgeführt werden sollen. Die ursprünglichen Methodenaufrufe können mit Hilfe von der Funktion proceed() weiter aufgerufen werden.

Es wird angenommen, dass die Objekte von der Klasse One nach dem Statechart in Abbildung 5-3 funktionieren sollen. Um ein Statechart zu definieren soll das definierte Statechart die Klasse Zustandmaschine wie in Abbildung 5-4 vererben.. Die Klasse Zustandmaschine verwaltet die definierten Zustände und Transitionen. Wenn ein Ereignis eintritt, findet sie entsprechende Transitionen und führt diese aus.

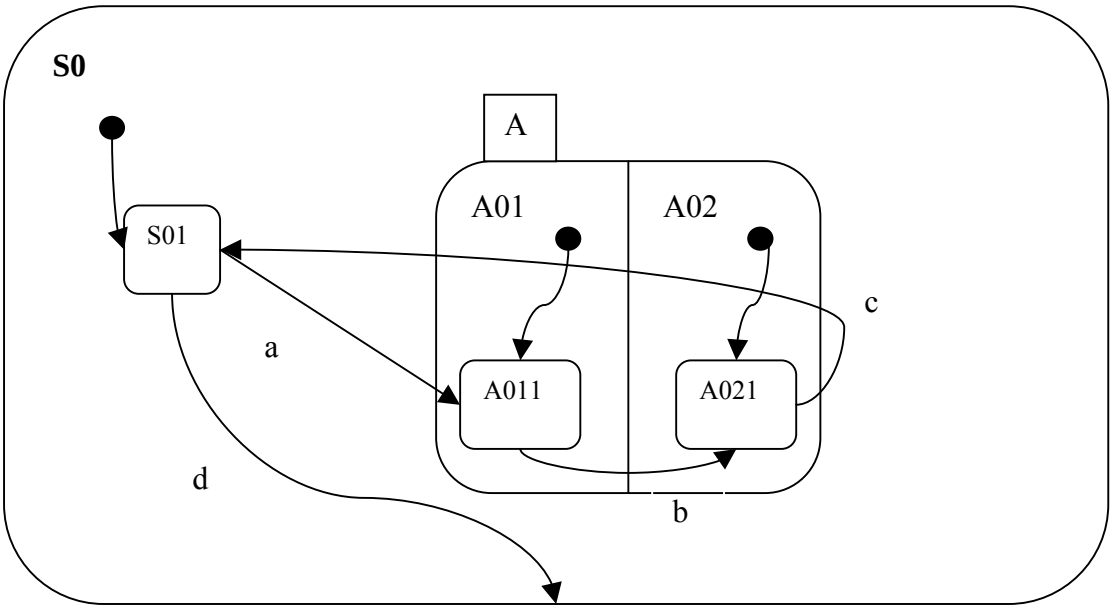


Abbildung 5-3

```
package test;

import sta.framework.*;

public aspect OneStatechart extends StateMachine {

    static Event a = new Event("a");
    static Event b = new Event("b");
    static Event c = new Event("c");
    static Event d = new Event("d");

    static ORState S0 = new ORState("S0", null);
    static BasicState S01 = new BasicState("S01", s0);
    static ANDState A = new ANDState("A", S0);
    static ORState A01 = new ORState("A01", A);
    static ORState A02 = new ORState("A02", A);
    static BasicState A011 = new BasicState("A011", A01);
    static BasicState A021 = new BasicState("A021", A02);

    static CompoundTransition t1 = createSimpleTransition(S01, A011, a);
    static CompoundTransition t2 = createSimpleTransition(A011, A021, b);
    static CompoundTransition t3 = createSimpleTransition(A021, S01, c);
    static CompoundTransition t4 = createSimpleTransition(S01, S0, d);

    static {
        setInitial(S0, s01);
        setInitial(A01, A011);
        setInitial(A02, A021);
    }

    pointcut aPointcut(One o) : execution(public void a()) && this(o);
    pointcut bPointcut(One o) : execution(public void b()) && this(o);
    pointcut cPointcut(One o) : execution(public void c()) && this(o);
    pointcut dPointcut(One o) : execution(public void d()) && this(o);
```

```
    void around(One o)  : aPointcut(o) {
        trigger(a);
        proceed(o);
    }
    void around(One o)  : bPointcut(o) {
        trigger(b);
        proceed(o);
    }
    void around(One o)  : cPointcut(o) {
        trigger(c);
        proceed(o);
    }
    void around(One o)  : dPointcut(o) {
        trigger(d);
        proceed(o);
    }

    public State getInitial() {
        return S01;
    }

}
```

Abbildung 5-4

Beim Eintreten und Austreten der Zustände sollten die Aktionen onExit() und onEntry() ausgeführt werden. Die beiden Methoden sind schon in der Klasse State als leere Methoden definiert und während der Transitionen ausgeführt werden. Exit/Action und Entry/Action können daher mit Hilfe von Pointcuts und Advices erweitert werden wie in Abbildung 5-5.

```
public aspect EntryExitAspect {

    pointcut entryPointcut(State s) : execution(public void onEntry()) && this(s);
    pointcut exitPointcut(State s) : execution(public void onExit()) && this(s);


    void around(State s)  : entryPointcut(s) {
        // do something else
        proceed(s);
    }
    void around(State s)  : exitPointcut(s) {
        // do something else
        proceed(s);
    }

}
```

Abbildung 5-5

Referenz

[1] James Rumbaugh. Objektorientiertes Modellieren und Entwerfen. Verlag: Hanser Fachbuch (Dezember 1993).

[2] UML-Tutorial. http://ivs.cs.uni-magdeburg.de/~dumke/UML/21.htm

[3] Das dynamische Modell http://www.fh-wedel.de/~si/seminare/ws97/Ausarbeitung/1.Schlueter/omt_30.htm

[4] D. Harel, A. Pnueli, J.P. Schmidt and R. Sherman, On the formal semantics of statecharts, Proc. 2nd IEEE Symposium on Logic in Computer Science (1987).

[5] On Nets with Structured Concurrency. Rik Eshuis. Eindhoven University of Technology, Department of Technology Management. http://fp.tm.tue.nl/beta/publications/working%20papers/Beta_wp155.pdf

[6]The STATEMATE Semantics of. Statecharts. DAVID HAREL. http://www.cs.mcgill.ca/~bcheun1/harel.pdf

[7] The Rhapsody Semantics of Statecharts. David Harel and Hillel Kugler. http://www.cs.nyu.edu/~kugler/charts04.pdf

[8] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Publisher: Addison-Wesley Professional; 1st edition (January 15, 1995)