Diplomarbeit

UML-Zustandsdiagramme von Java-Klassen auf der Basis des Vertragsmodells

MATTHIAS WITT

April 2003

Betreuer:

PROF. DR.-ING. HEINZ ZÜLLIGHOVEN[*]
DR. DANIEL MOLDT[**]

Universität Hamburg
Fachbereich Informatik
[*]Arbeitsbereich Softwaretechnik
[**]Arbeitsbereich Theoretische Grundlagen der Informatik
Vogt-Kölln-Straße 30, 22527 Hamburg

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Zustandsdiagramme sind ein mächtiges Hilfsmittel zur Beschreibung des dynamischen Verhaltens von Klassen in objektorientierten Systemen. In der Literatur zur Objektorientierung wird Zustandsdiagrammen (und den Beschreibungen von Zuständen allgemein) allerdings eine untergeordnete Bedeutung zugemessen; auf die Beschreibung der statischen Aspekte der Systeme wird ungleich mehr Wert gelegt. Zudem herrscht kein Konsens darüber, wie die beim Entwurf entwickelten Zustandsdiagramme auf eine konkrete Implementierung in einer Programmiersprache abgebildet werden sollen. Bezeichnenderweise sind die Beispiele, die in der Literatur gebracht werden, meistens weit entfernt von Klassen, wie man sie in einem realistischen Anwendungssystem antrifft: Da werden Zustandsdiagramme entworfen für Mikrowellengeräte [SM92], Schienenwagensysteme [HG96], Heimtrainer [LL98] oder Thermostate [BRJ99].

Sofern überhaupt Hinweise gegeben werden, wie Zustandsdiagramme in Klassen umzusetzen sind, so sind die Vorschläge meist nicht sehr akzeptabel, da viel Code-Overhead entsteht. Die meisten Vorschläge beruhen auf einer ausprogrammierten Ereignisbehandlung, die vergleichsweise viel Programmcode erzeugt (siehe z. B. [SM92] oder [Köhler00]). Auch wenn dieser Code – wie in [Köhler00] – automatisch von einem grafischen Modellierungswerkzeug generiert wird, so ist diese Möglichkeit der Zustandsmodellierung sicherlich nicht das Optimum, da zu viel Overhead entsteht. – Andere Ansätze vertreten bei der Umsetzung von Zustandsdiagrammen in Programmcode eine sehr prozedurale Sichtweise; [RBPEL91] legt beispielsweise die Verwendung von vielen geschachtelten Schleifen nahe und toleriert sogar ausdrücklich den Gebrauch von `Goto`-Anweisungen.

1.1 Zielsetzung

Ziel dieser Arbeit soll es sein, einen Ansatz zur Beschreibung von Zustandsdiagrammen von Klassen zu entwickeln, der in der Literatur bislang vernachlässigt wurde. Dieser Ansatz beruht auf dem Vertragsmodell von Meyer [Meyer97], welches vorsieht, dass zu jeder Operation Vor- und Nachbedingungen angegeben werden; das sind boolesche Ausdrücke, die vor bzw. nach Ausführung der Operation wahr sein müssen. Mithilfe dieser Zusicherungen ist es möglich zu entscheiden, in welchen Zuständen sich ein Exemplar dieser Klasse vor bzw. nach Ausführung der Operation befinden kann. Die Zustandsdiagramme dieser Arbeit beschreiben nicht das Verhalten von Klassen (also deren Implementierung), sondern lediglich die Klassenprotokolle, also die möglichen Aufrufreihenfolgen der Operationen. Bei der Implementierung der Zustandsdiagramme in Klassen entsteht somit lediglich ein minimaler Overhead, da außer dem Code zur Überprüfung der Zusicherungen kein weiterer Programmcode anfällt.

In dieser Arbeit soll eine Abbildung zwischen den Zusicherungen einer Klasse und dem zugehörigen Zustandsdiagramm entwickelt werden, und zwar in beide Richtungen. Die Zusicherungen in der Klasse werden dazu mathematisch modelliert und aus diesem Modell die Zustände und Transitionen des Zustandsdiagramms abgeleitet. Umgekehrt soll aus der Beschreibung der Zustände und Transitionen ein Modell der Zusicherungen hergeleitet werden. Zu diesem Zweck muss bei Zustandsdiagrammen auf einige höhere Modellierungskonzepte verzichtet werden (z. B. Gedächtniszustände), da diese sich nicht effektiv auf Zusicherungen abbilden lassen (Zusicherungen können nicht ohne weiteres auf vorige Zustände Bezug nehmen). Beim Vertragsmodell hingegen ist keine Einschränkung notwendig.

Ferner soll untersucht werden, welchen Nutzen die Verbindung von Vertragsmodell und Zustandsdiagrammen für die Softwareentwicklungspraxis bietet. Es wird der Frage nachgegangen, ob diese

beiden Konzepte sich gegenseitig ergänzen können, d. h. ob eine durchdachte Wahl der Zusicherungen zu einem wohldefinierten Klassenprotokoll verhilft und ob ein anschauliches Zustandsdiagramm wiederum zu robusten Zusicherungen führt. So könnte die in dieser Arbeit vorgenommene Verbindung in doppelter Hinsicht nützlich sein.

Im Rahmen dieser Arbeit wurde der *Statechartgenerator* entwickelt, ein Werkzeug, welches den Quelltext einer beliebigen Java-Klasse (die auf der Basis des Vertragsmodells erstellt worden sein muss) einliest, ein Zustandsdiagramm erzeugt und darstellt. Am Zustandsdiagramm können manuell Änderungen durchgeführt werden, woraufhin die Zusicherungen per Knopfdruck entsprechend angepasst werden können.

Zur Untersuchung des in dieser Arbeit entwickelten Verfahrens wird das JWAM-Rahmenwerk [URL1] in der Version 1.8.0 herangezogen. JWAM ist ein Rahmenwerk zur Entwicklung von objektorientierter Anwendungssoftware in Java [GJSB00] nach dem WAM-Ansatz (Werkzeug-Automat-Material, siehe [Züllighoven98]). Da es vollständig auf der Grundlage des Vertragsmodells entworfen wurde, eignet es sich hervorragend zur genaueren Untersuchung. Für das Verständnis dieser Arbeit ist keine detaillierte Kenntnis von JWAM notwendig, da lediglich die Zusicherungen und Klassenprotokolle betrachtet werden, nicht die Implementierung oder Architektur.

An dieser Stelle noch ein Wort zur Terminologie: Mit dem Begriff *Operation* werden in dieser Arbeit sowohl *Methoden* als auch *Konstruktoren* bezeichnet. Bei den Methoden wird wiederum zwischen *Funktionen* und *Prozeduren* unterschieden, wobei Funktionen diejenigen Methoden sind, die einen Rückgabewert liefern; alle übrigen Methoden sind Prozeduren.

Das folgende UML-Diagramm verdeutlicht diese Zusammenhänge:

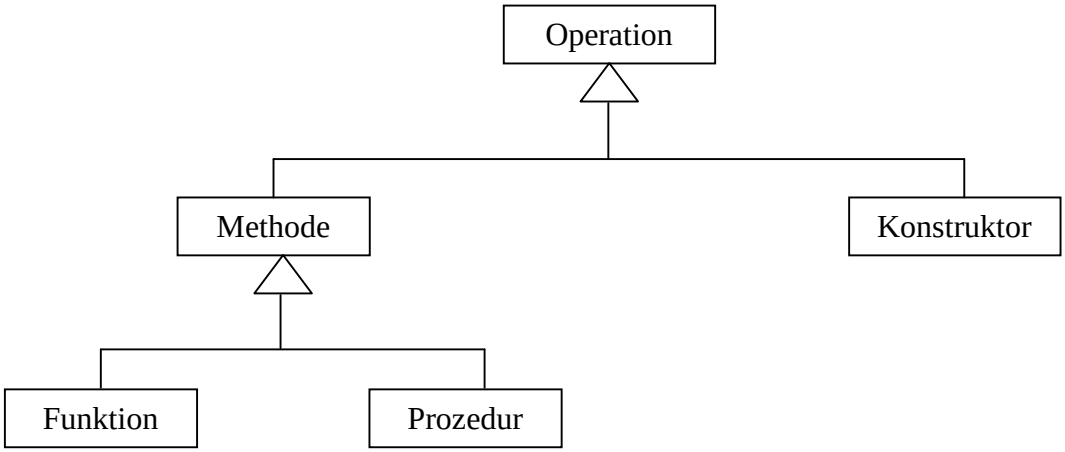


Abbildung 1.1: Zur Terminologie

1.2 Aufbau der Arbeit

Der Aufbau dieser Arbeit ist wie folgt: In *Kapitel 2* wird die Syntax von Zustandsdiagrammen erläutert. *Kapitel 3* gibt einen Überblick über verschiedene Ansätze zur Verwendung von Zustandsdiagrammen in der objektorientierten Softwareentwicklung. In *Kapitel 4* erfolgt zunächst eine Einführung in das Vertragsmodell; anschließend wird ein Zusammenhang zwischen Zustandsdiagrammen und Zusicherungen hergestellt. Die Kapitel 2 bis 4 bilden somit den Grundlagenteil der Arbeit.

Kapitel 5 erklärt ausführlich, wie aus einer Klasse mit explizit angegebenen Zusicherungen das zugehörige Zustandsdiagramm erzeugt werden kann. Außerdem wird untersucht, wie die Zustandsdiagramme von Ober- und Unterklassen zueinander in Beziehung stehen. In *Kapitel 6* wird beschrieben, wie aus einem gegebenen Zustandsdiagramm die Zusicherungen generiert werden können. Im *7. Kapitel* wird der Statechartgenerator vorgestellt, der die gegenseitige Abbildung von Zusicherungen und Zustandsdiagrammen vornehmen kann.

In *Kapitel 8* wird ein Bezug zur Praxis hergestellt, indem z. B. die Klassen des JWAM-Rahmenwerks im Hinblick auf die generierten Zustandsdiagramme untersucht werden und der praktische Nutzen des Verfahrens erwogen wird. Im abschließenden *9. Kapitel* erfolgt schließlich eine Zusammenfassung der Ergebnisse.

2 Zustandsdiagramme

In diesem Kapitel wird eine Einführung in Zustandsdiagramme gegeben. Diese Einführung bezieht sich hauptsächlich auf die syntaktische Beschreibung der Elemente, wobei von der Notation nach der UML (Unified Modeling Language [BRJ99, URL2]) ausgegangen wird. Dabei wird berücksichtigt, dass Zustandsdiagramme in dieser Arbeit zur Beschreibung von Klassenprotokollen eingesetzt werden.

2.1 Die Idee der Zustandsdiagramme

Endliche Automaten [Schöning01] wurden in den achtziger Jahren von David Harel zu Zustandsdiagrammen weiterentwickelt um das Verhalten komplexer Systeme zu beschreiben.[1] Endliche Automaten werden jedoch aufgrund ihrer sehr einfachen Struktur bei komplexeren Systemen sehr schnell unübersichtlich, namentlich durch ein exponentielles Wachstum der Anzahl der Zustände. Harels Zustandsdiagramme bieten dagegen Möglichkeiten zur Zusammenfassung von semantisch ähnlichen Zuständen, beispielsweise die Bildung von Zustandshierarchien, parallele Zustände usw.

Harel entwarf Zustandsdiagramme ursprünglich nicht im Hinblick auf Objektorientierung, sondern zur Beschreibung allgemeiner komplexer Systeme. Mit der zunehmenden Bedeutung des objektorientierten Paradigmas in der Softwaretechnik wurde auch das Interesse geweckt, Zustandsdiagramme im Rahmen der objektorientierten Analyse und des Entwurfs einzusetzen [RBPEL91, CHB92, HG96, BRJ99]; immerhin ist die objektorientierte Programmierung eine Form der zustandsorientierten Programmierung. Zustandsdiagramme können hierbei eingesetzt werden um das Zustandsverhalten einzelner Klassen zu beschreiben. In Kapitel 3 werden einige bekannte Ansätze vorgestellt, Zustandsdiagramme in die objektorientierte Programmierung einzubeziehen.

2.2 Die Syntax von Zustandsdiagrammen

Die Syntax von Zustandsdiagrammen wird hier von der UML übernommen. Sämtliche dort vorgesehenen Details hier anzuführen würde jedoch den Rahmen sprengen; deshalb erfolgt hier nur die Darstellung der wichtigen Elemente und Konzepte, von denen wiederum auch nicht alle im weiteren Verlauf dieser Arbeit Verwendung finden. Für eine vollständige Auflistung aller Details zu UML-Zustandsdiagrammen sei auf [URL2] verwiesen.

2.2.1 Zustände

Ein Zustand repräsentiert eine Bedingung während des Lebenszyklus eines Objekts. Der Zustand ist genau dann aktiv, wenn die zugehörige Bedingung wahr ist.

Zustände werden durch abgerundete Rechtecke dargestellt. Der optionale Name des Zustands steht im Inneren:

[1] [Harel87]; zur weiteren Entwicklung siehe auch [Harel97] und [Beeck94]

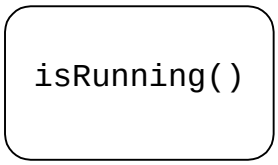


Abbildung 2.1: Zustand

In dieser Arbeit werden Zustände mit den Bedingungen beschriftet, die in ihnen gelten. Im obigen Beispiel liefert die Funktion `isRunning` den Wert `true`, wenn sich das Objekt in diesem Zustand befindet.

Mehrere Bedingungen werden untereinander geschrieben:

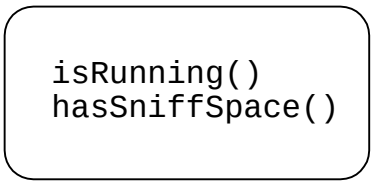


Abbildung 2.2: Zustand mit mehreren Bedingungen

Ein Zustand kann interne Transitionen enthalten. Eine interne Transition ist ein Paar von einem Ereignis und einer Aktion. Tritt das Ereignis ein, so wird die zugehörige Aktion ausgeführt. Um interne Transitionen darzustellen wird der Zustand durch eine horizontale Linie unterteilt, unter der dann die internen Transitionen in folgender Form aufgelistet werden:

Ereignis / *Aktion*

Einige Ereignisse sind vordefiniert:

- **entry**: Betreten des Zustands
- **exit**: Verlassen des Zustands
- **do**: Ausführung der Aktion während der gesamten Verweildauer in diesem Zustand
- **include**: Aufrufen einer Unterzustandsmaschine (die Aktion enthält deren Namen)

2.2.2 Transitionen

Transitionen werden durch einen Pfeil dargestellt und beschreiben Zustandsübergänge. Die Transition verläuft von einem Ursprungs- zu einem Zielzustand. Sind diese identisch, so spricht man von einer Selbsttransition.

In der UML werden Pfeile mit offener und geschlossener Spitze unterschieden. Pfeile mit offener Spitze repräsentieren das Senden von Nachrichten, wobei der Kontrollfluss direkt zum Sender der Nachricht zurückkehrt. Ein Pfeil mit geschlossener Spitze steht dagegen für den Aufruf einer Operation. Dieser Aufruf ist synchron und die Kontrolle kehrt erst nach vollständiger Ausführung der Operation zum Auslöser zurück. In dieser Arbeit sind die auslösenden Ereignisse immer Aufrufe

von Operationen, da die Zustandsdiagramme zur Spezifikation von Klassenprotokollen eingesetzt werden; deshalb werden hier Pfeile mit geschlossener Spitze verwendet.

Die Transition wird mit dem Ereignis beschriftet, das ihr Auslöser ist.[2] Tritt dieses Ereignis ein, so schaltet die Transition und der Zustandsübergang findet statt.

Auslösende Ereignisse sind hier immer Aufrufe von Operationen. Operationen werden in der UML folgendermaßen notiert:

Sichtbarkeit Name (*Parameterliste*) : *Rückgabetyp* { *Eigenschaftswerte* }

Alle Elemente bis auf den Namen und die Klammern um die Parameterliste sind optional.
Die Bedeutung im Einzelnen ist:

Sichtbarkeit: Ein Zeichen, das die Sichtbarkeit der Operation angibt. Es gibt vier Möglichkeiten:

UML-Zeichen	**Sichtbarkeit (Java)**
+	`public`
#	`protected`
~	`package`
–	`private`

Name: Der Name der Operation (auch im Falle eines Konstruktors).

Parameterliste: Eine durch Kommata getrennte Liste der Parameter dieser Operation in der Form:
Parametername : *Parametertyp*

Rückgabetyp: Der Typ des Rückgabewerts (entfällt bei Prozeduren und Konstruktoren).

Eigenschaftswerte: Eine durch Kommata getrennte Liste von Eigenschaftswerten dieser Operation („property list“). Möglich sind bei Verwendung von Java:

Eigenschaftswert	**Java-Modifier**
`abstract`	`abstract`
`leaf`	`final`
`concurrency = guarded`	`synchronized`

Im Anschluss an die Angabe der Operation kann noch in eckigen Klammern eine Wächterbedingung folgen. Dies ist ein boolescher Ausdruck, der wahr sein muss, damit die Transition schalten kann. Nun kann nach einem Schrägstrich eine Folge von Aktionen angegeben werden. Diese Aktionen werden ausgeführt, wenn die Transition schaltet. Am Ende kann in geschweiften Klammern eine Zusicherung („Constraint“) spezifiziert werden. Dabei handelt es sich um einen booleschen Ausdruck, der immer wahr ist, nachdem der Zustandsübergang stattgefunden hat.

[2] Eine Transition kann auch mehrere Ereignisse haben, die sie auslösen.

Die Syntax für die booleschen Ausdrücke ist in der UML nicht zwingend vorgeschrieben. Es wird die Verwendung von OCL[3] empfohlen; falls man sich jedoch nahe an einer bestimmten Programmiersprache bewegt, sollte man deren Syntax benutzen. In dieser Arbeit wird daher bevorzugt Java-Syntax verwendet.

Hier ein Beispiel für eine Transitionsbeschriftung:

```
+foo(o: Object): int {abstract} [!equals(o)] {result != 0}
```

Die zugehörige Deklaration der Operation lautet[4]:

```
public abstract int foo(Object o);
```

2.2.3 Anfangs- und Endzustände

Der erste Zustand, der eingenommen wird, ist der Anfangszustand. Der Anfangszustand darf nicht Zielzustand einer Transition sein. Ein Endzustand hingegen kann nicht verlassen werden (kein Ursprungszustand einer Transition), markiert also das Ende einer Folge von Zustandsübergängen. Anfangszustände werden durch kleine ausgefüllte Kreise dargestellt; Endzustände haben noch einen umschließenden Kreis:

Abbildung 2.3: Anfangs- und Endzustand

Wenn es mehrere Transitionen gibt, die aus dem Anfangszustand herausführen, so können mehrere Anfangszustände gezeichnet werden. Konzeptionell gibt es aber nur einen, und sobald eine der Transitionen schaltet, die als Ursprungszustand einen Anfangszustand hat, ist der Zielzustand dieser Transition der aktive Zustand.

Transitionen, die aus dem Anfangszustand herausführen, werden durch Konstruktoren ausgelöst, da sie die Initialisierung des Objekts beschreiben. Der Konstruktor legt fest, welche Zustände das Objekt anfangs einnehmen kann.

2.2.4 Sequenzielle Unterzustände

Der Formalismus der Zustandsdiagramme erlaubt die Bildung von Zustandshierarchien. Ein Zustand kann beliebig viele sequenzielle Unterzustände besitzen, wobei gilt, dass das System sich im Oberzustand befindet, wenn es sich in genau einem der Unterzustände befindet.[5] Ober- und Unterzustand sind also gleichzeitig aktiv. Eine Transition, deren Ursprungszustand der Oberzustand ist[6],

[3] Object Constraint Language, siehe [URL2, Kapitel 6]
[4] Wächterbedingung und Constraint sind hieraus nicht ersichtlich.
[5] Aus diesem Grund bezeichnet man die Bildung von sequenziellen Unterzuständen auch als XOR-Dekomposition.
[6] in Abbildung 2.4 vom Oberzustand zu Zustand 2

verlässt auch den aktiven Unterzustand, wenn sie schaltet. Es ist auch möglich, dass eine Transition aus einem Unterzustand nach außerhalb des Oberzustands führt[7]; auch in diesem Fall werden sowohl Ober- als auch Unterzustand verlassen.

Transitionen können von außerhalb zu einem Unterzustand führen.[8] Sie können aber auch zum Oberzustand führen[9]; in diesem Fall muss es als Unterzustand einen Anfangszustand geben, der dann aktiv wird und dessen Transition sofort schaltet.

Zustandshierarchien werden durch grafischen Einschluss dargestellt[10]:

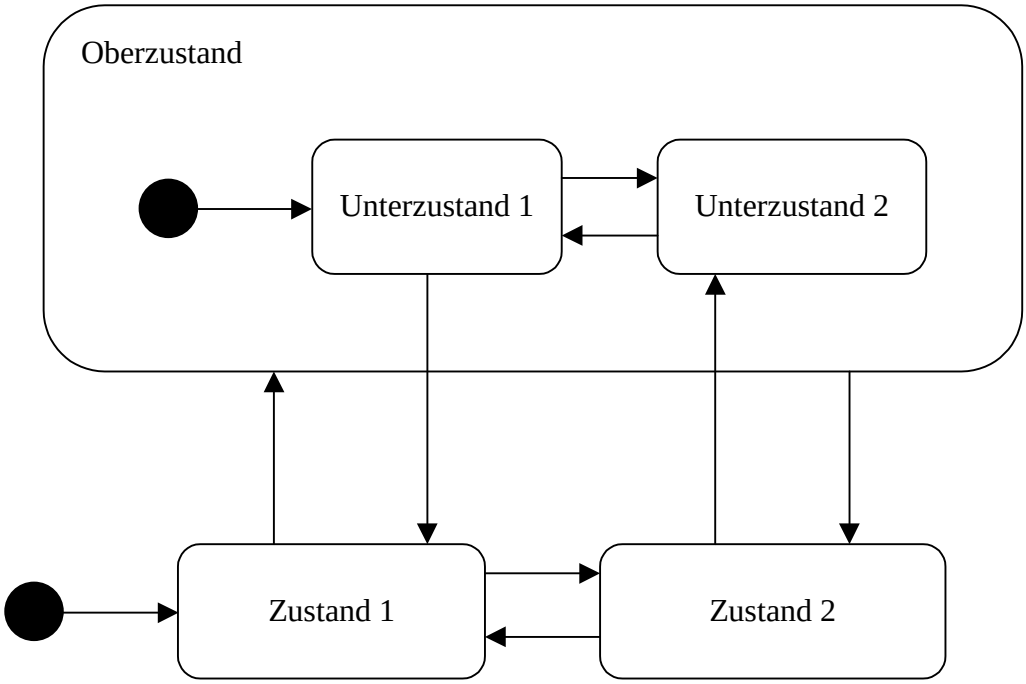


Abbildung 2.4: Sequenzielle Unterzustände

2.2.5 Parallele Unterzustände

Ein Zustand kann beliebig viele parallele Unterzustände besitzen. Parallele Zustände bilden die zweite Form von Unterzuständen, wobei alle parallelen Zustände gleichzeitig aktiv sind.[11] Die einzelnen parallelen Zustände dürfen keine eigenen Transitionen haben, da sie immer alle gleichzeitig betreten oder verlassen werden.

Meistens besteht ein paralleler Zustand aus sequenziellen Unterzuständen inklusive eigenem Anfangszustand. Führt aus einem dieser Unterzustände eine Transition heraus, so beendet diese alle parallelen Zustände.

Parallele Zustände werden durch gestrichelte Linien voneinander getrennt:

[7] von Unterzustand 1 zu Zustand 1
[8] von Zustand 2 zu Unterzustand 2
[9] von Zustand 1 zum Oberzustand
[10] In diesen Beispielen wird aus Gründen der Übersichtlichkeit auf die Beschriftung der Transitionen verzichtet.
[11] Die Bildung von parallelen Zuständen wird daher auch als AND-Dekomposition bezeichnet.

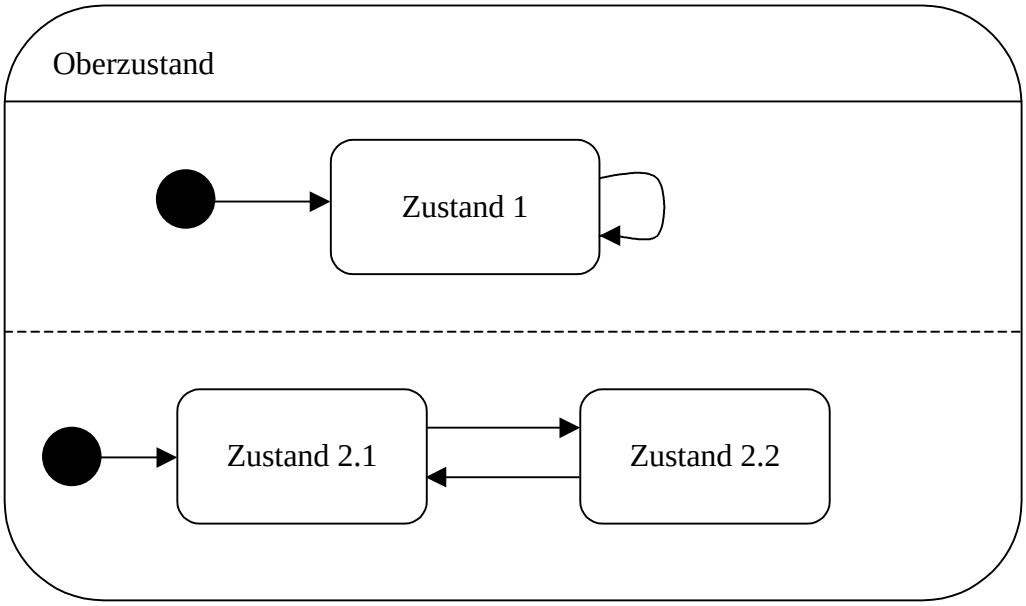


Abbildung 2.5: Parallele Unterzustände

2.2.6 Gedächtniszustände

Gedächtniszustände („history states“) können in Zusammenhang mit sequenziellen Unterzuständen auftreten. Beim Verlassen des zusammengesetzten Zustands wird sich der zuletzt aktive Unterzustand gemerkt. Wird nun der zusammengesetzte Zustand wieder aktiv, indem zu einem Gedächtniszustand verzweigt wird, so geht die Kontrolle auf den zuletzt aktiven Unterzustand über. War zuvor noch kein Unterzustand aktiv, so geht es mit dem Zielzustand der Transition weiter, die aus dem Gedächtniszustand herausführt; fehlt diese Transition, ist das Verhalten undefiniert.

Gedächtniszustände werden durch ein H innerhalb eines kleinen Kreises gekennzeichnet:

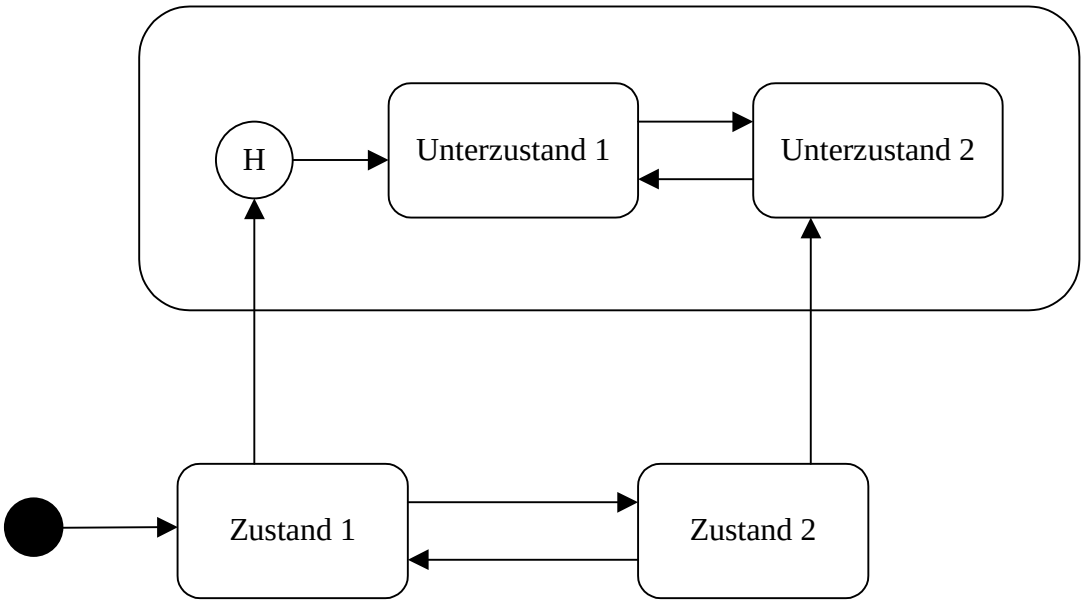


Abbildung 2.6: Gedächtniszustand

Erfolgt in Abbildung 2.6 ein Zustandsübergang von Zustand 1 zum zusammengesetzten Zustand, so wird Unterzustand 1 oder 2 aktiv, je nachdem, welcher von beiden zuletzt eingenommen wurde. Wurde bislang noch keiner eingenommen, so wird Unterzustand 1 aktiv. Zustand 2 dagegen verzweigt immer zu Unterzustand 2, unabhängig vom zuletzt aktiven Zustand.

Bei Gedächtniszuständen unterscheidet man zwischen flachen und tiefen Gedächtnissen. Bei einem tiefen Gedächtnis werden rekursiv in allen Unterzuständen innerhalb der Zustandshierarchie des zusammengesetzten Zustandes die zuletzt aktiven Zustände eingenommen; bei einem flachen Gedächtnis gilt dies nur für die Hierarchiestufe, in der sich der Gedächtniszustand befindet. Gibt es – wie im obigen Beispiel – in dieser Hierarchiestufe keine weiteren Unterzustände, so hat diese Unterscheidung keine Auswirkung. Tiefe Gedächtniszustände werden durch einen Stern hinter dem H gekennzeichnet; ohne diesen Stern handelt es sich um einen flachen Gedächtniszustand.

2.2.7 Nebenläufige Transitionen

Nebenläufige Transitionen haben mehrere Ursprungs- oder Zielzustände (oder beides). Sie repräsentieren eine Synchronisation oder Aufspaltung der Ablaufsteuerung und werden als Balken dargestellt:

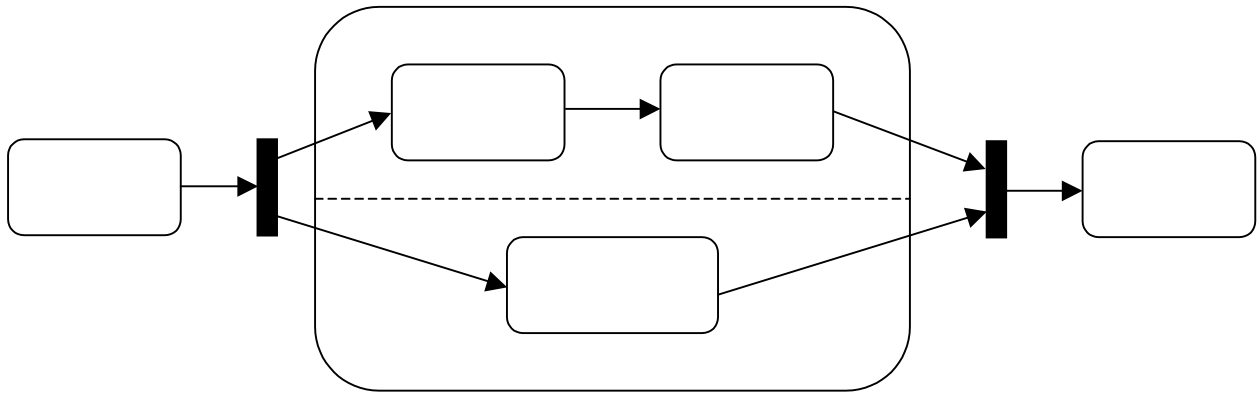


Abbildung 2.7: Nebenläufige Transitionen

2.2.8 Entscheidungsknoten

Führen von einem Zustand mehrere Transitionen, die sich nur in den Wächterbedingungen unterscheiden, zu unterschiedlichen Zielzuständen, so kann ein Entscheidungsknoten eingesetzt werden. Dort wird je nach gültiger Wächterbedingung der passende Zielzustand herausgesucht. Die Wächterbedingungen schließen sich im Normalfall gegenseitig aus; ist dies jedoch nicht der Fall, so ist der Zielzustand nicht eindeutig.

Entscheidungsknoten werden durch einen kleinen ausgefüllten Kreis dargestellt[12]:

[12] Alternativ kann ein unausgefülltes Diamantensymbol verwendet werden.

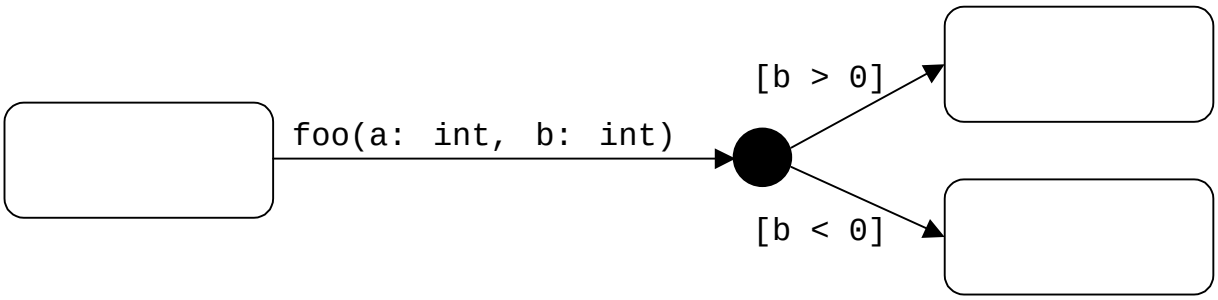


Abbildung 2.8: Entscheidungsknoten

Ist im obigen Beispiel `b` gleich null, so erfolgt kein Zustandsübergang.

Alle Bedingungen werden ausgewertet, bevor eine Transition schaltet; daher wird diese Form auch als *statischer Entscheidungsknoten* bezeichnet. Die UML definiert auch *dynamische Entscheidungsknoten*, bei denen die Bedingungen der herausführenden Transitionen erst ausgewertet werden, wenn der Knoten betreten wird. Auf diese Weise kann eine Transition, die zum Entscheidungsknoten führt, noch eine Aktion auslösen, die die Bedingungen ändert. Dabei wird die Verwendung der speziellen Wächterbedingung `[else]` empfohlen um Laufzeitfehler zu vermeiden. Dynamische Entscheidungsknoten werden als ungefüllte Kreise dargestellt.

2.2.9 Synchronisationszustände

Synchronisationszustände dienen dazu, mehrere nebenläufige Transitionen zu synchronisieren. Sie werden dargestellt als Kreis mit einer positiven Ganzzahl oder einem Stern. Die Zahl bezeichnet die maximale Differenz zwischen der Anzahl der Schaltvorgänge der Eingangs- und Ausgangstransitionen; ein Stern steht für unbegrenzte Differenz. Hier besteht also ein enger Zusammenhang zur Kapazität einer Stelle in einem Petrinetz [Reisig86, Baumgarten96]:

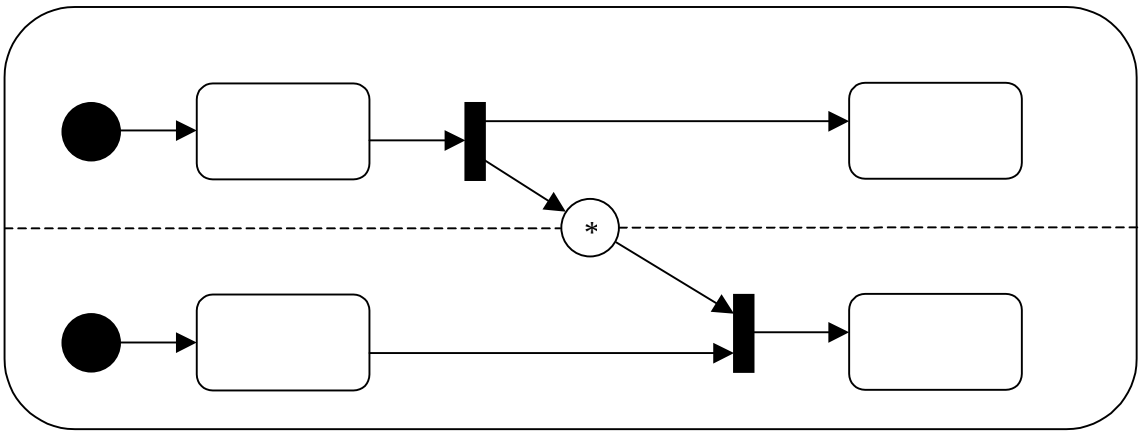


Abbildung 2.9: Synchronisationszustand

3 Zustandsdiagramme im Kontext der Objektorientierung

In der objektorientierten Softwareentwicklung kommt den Zustandsdiagrammen im Regelfall eine verhältnismäßig geringe Bedeutung zu. Klassen- und Anwendungsfalldiagramme zum Beispiel werden in der entsprechenden Fachliteratur meist sehr ausführlich behandelt, während Zustandsdiagramme häufig sehr knapp besprochen werden [RBPEL91, Jacobson92, Booch94, JBR99, Oestereich01]. Wie Zustandsdiagramme in einer Programmiersprache umzusetzen sind, ist in den meisten Fällen nicht klar; einen einheitlichen Ansatz gibt es nicht.

In diesem Kapitel sollen einige etablierte Ansätze vorgestellt werden, wie Zustandsdiagramme im Kontext der Objektorientierung eingesetzt werden können, wobei es weniger um die Syntax der Diagramme als um deren Semantik und Pragmatik geht. Die Vorstellung erfolgt hier deshalb, um den in dieser Arbeit gewählten Ansatz einordnen zu können.

3.1 OMT-Zustandsdiagramme

In der Object Modeling Technique (OMT, siehe [RBPEL91]) finden Zustandsdiagramme eine ihrer frühesten Anwendungen im Kontext der objektorientierten Softwareentwicklung. Es werden drei verschiedene Modelle zur Beschreibung von Systemen verwendet: das *Objektmodell*, das *dynamische Modell* und das *funktionale Modell*.

Das *Objektmodell* beschreibt die statische Struktur des Systems und die Attribute, Operationen und Beziehungen der Objekte. Dieses Modell kann durch Objektdiagramme visualisiert werden, und zwar durch Klassen- und Exemplardiagramme[13].

Das *dynamische Modell* beschreibt die zeitlichen und sequenziellen Aspekte des Systems. Ereignisse können die Zustände von Objekten beeinflussen. Hier kommen Zustandsdiagramme zum Einsatz.

Das *funktionale Modell* beschreibt die Transformationen und Abhängigkeiten von Daten. Dabei werden Datenflussdiagramme verwendet.

Zustandsdiagramme haben im dynamischen Modell eine zentrale Bedeutung.[14] Ein Zustandsdiagramm gehört zu genau einer Klasse, wobei nicht zu jeder Klasse ein Zustandsdiagramm entworfen werden muss. Dabei wird im Wesentlichen die originale Notation aus [Harel87] beibehalten. Von großer Bedeutung sind die Ereignisse, die als Exemplare von Ereignisklassen verstanden werden, wodurch auch Ereignishierarchien gebildet werden können. Ereignisse können von außerhalb des Systems kommen oder zwischen den Objekten ausgetauscht werden. Zustände sind Abstraktionen der Attributwerte und Referenzen eines Objekts. Zustandsübergänge werden durch Ereignisse ausgelöst.

In [RBPEL91, S. 239–241] werden nur wenige Informationen darüber gegeben, wie das dynamische Modell implementiert werden kann. Es werden lediglich drei verschiedene Ansätze grob angerissen:

---

[13] engl. *class diagrams* bzw. *instance diagrams*; die Bezeichnung von Klassendiagrammen als Form von Objektdiagrammen ist allerdings ungewöhnlich und irreführend.

[14] In [Rumbaugh95] wird das dynamische Modell noch einmal mit einigen kleineren Ergänzungen dargestellt.

Prozedurgesteuertes System: Der aktuelle Zustand ergibt sich aus der gerade durchlaufenen Stelle des Programms. Folgen von Zuständen werden durch Anweisungsfolgen implementiert, Schleifen im Zustandsdiagramm werden durch Programmschleifen realisiert. Selbst der Gebrauch von `Goto`-Anweisungen wird nahe gelegt, wenn auch nur im Ausnahmefall. Es wird empfohlen, den Hauptpfad aus dem Zustandsdiagramm sequenziell zu implementieren; Verzweigungen werden durch Auswahlkonstrukte realisiert; Fehlerfälle können durch Fehlerroutinen, Status-Flags oder `Goto`-Anweisungen behandelt werden. Eingabeereignisse (also sämtliche Transitionen) werden durch blockierende I/O-Lesevorgänge oder Warte-Anweisungen (in multitaskingfähigen Sprachen wie Ada) realisiert. – Diese Form der Implementierung hat mit Objektorientierung allerdings nichts mehr zu tun.

Ereignisgesteuertes System: Eine Zustandsmaschine wird explizit implementiert und ausgeführt. Eine gesonderte Klasse könnte z. B. die Möglichkeit bieten, ein Zustandsdiagramm auszuführen. Ein Objekt könnte dann seine Zustandsmaschine den Folgezustand und etwaige Aktionen bestimmen lassen. – Diese Implementierungsmöglichkeit wird auch in [SM92] verfolgt und ist die direkteste Abbildung eines Zustandsdiagramms auf Programmcode. In eine ähnliche Richtung geht auch das Zustandsmuster aus [GHJV95].

Nebenläufige Tasks: In diesem Ansatz wird ein Objekt durch einen eigenständigen Task der Programmiersprache oder des Betriebssystems realisiert. Ereignisse werden als Interaktionsbeziehungen zwischen den Tasks implementiert. Die einzelnen Tasks arbeiten dann allerdings wieder prozedurgesteuert, also nicht objektorientiert. – Diese Realisierungsform wird jedoch nicht von allen Programmiersprachen unterstützt und ist außerdem wegen des Overheads für die Verwaltung der Tasks sehr ineffizient.

Im Kontext der Objektorientierung ist der einzige sinnvolle Ansatz der zweite, da die beiden anderen nicht zu objektorientierten Systemen führen. Allerdings resultiert auch aus dem zweiten Ansatz ein Overhead, der nicht zu vernachlässigen ist, da viel zusätzlicher Code für die Implementierung der Zustandsmaschinen anfällt.

3.2 Objectcharts

In [CHB92] werden Objectcharts als Erweiterung von Zustandsdiagrammen eingeführt. Während Zustandsdiagramme noch recht allgemein einsetzbar sind, modellieren Objectcharts immer eine Klasse eines objektorientierten Systems. Der Ansatz der Objectcharts ist formal besser ausgearbeitet als die meisten übrigen objektorientierten Ansätze, die den Autoren zufolge zu wenig Präzision haben.

Zustandsänderungen erfolgen in Objectcharts durch Objektinteraktion (Dienstanfragen eines Objekts an ein anderes durch Methodenaufruf). Der Broadcast-Mechanismus, der in [Harel87] eingeführt wurde, wird nicht erlaubt, da im gängigen Modell der Objektorientierung ein Objekt nicht mit allen Objekten gleichzeitig interagiert.[15] Die Zustandsänderungen werden durch Methodenaufrufe von außen oder durch Zeitereignisse ausgelöst (z. B. Verweilen in einem Zustand über eine bestimmte Zeitdauer). Als Aktionen bei Zustandsübergängen sind Aufrufe von Methoden anderer Objekte möglich.

[15] Harel führt in [HG96] korrekt dagegen an, dass ein Objectchart nur zu einer einzigen Klasse gehört und somit die Broadcast-Kommunikation innerhalb eines Objectcharts auch nur diese Klasse betrifft.

Das folgende Beispiel eines Software-Weckers aus [CHB92, S. 13] verdeutlicht die Möglichkeiten der Objectcharts:

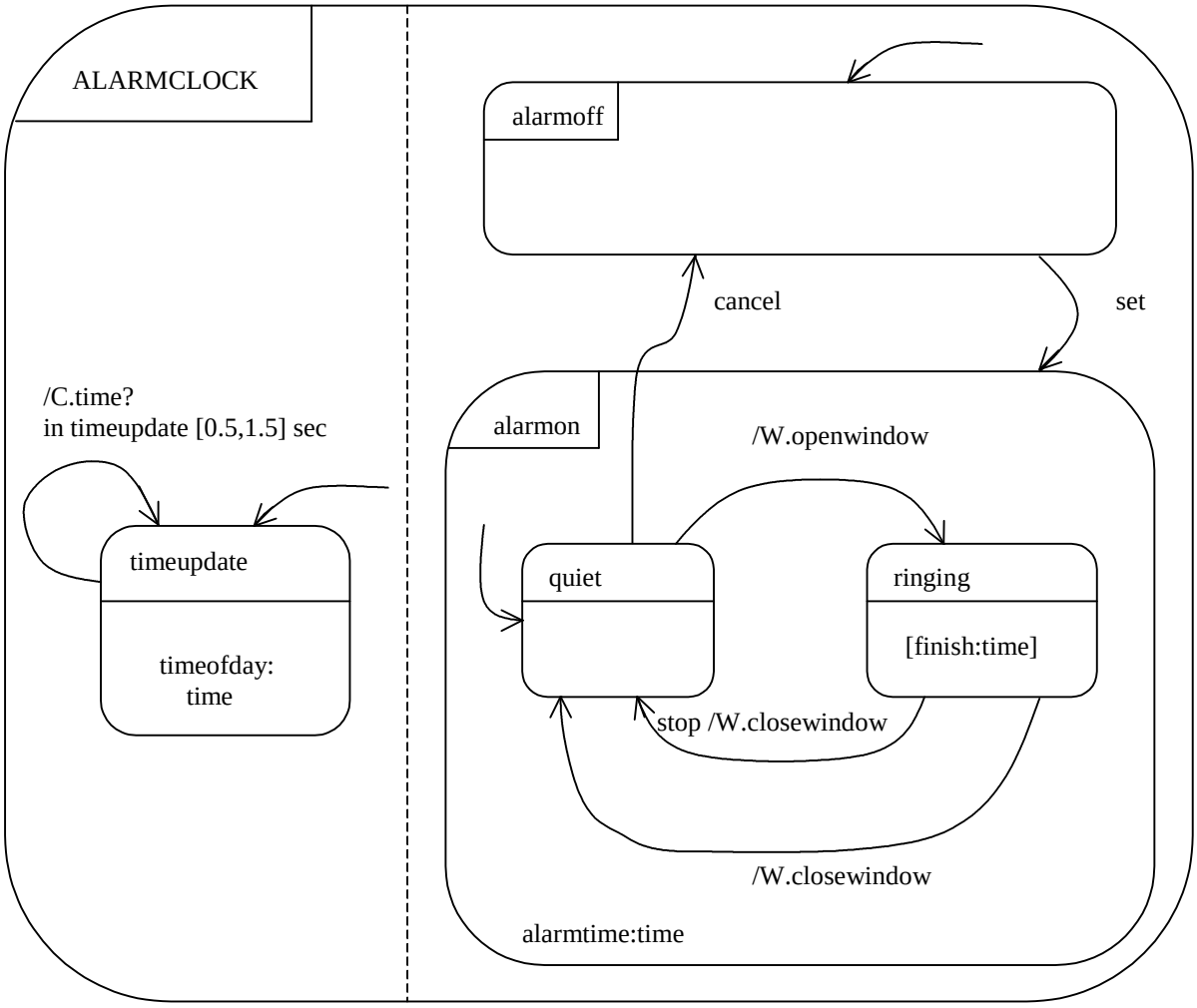


Abbildung 3.1: Objectchart aus [CHB92]

Hierbei sieht man, dass Zustände auch mit Attributen versehen werden können, die für sie von Bedeutung sind. Der Zustand `timeupdate` hat z. B. ein Attribut `timeofday` vom Typ `time`. Ist ein solches Attribut in eckige Klammern gesetzt (z. B. `finish` im Zustand `ringing`), so hat es privaten Zugriff, d. h. ist von außen nicht sichtbar.[16]

Transitionen können auf zwei Weisen ausgelöst werden:

1. **Aufruf einer Methode:** Ein Beispiel hierfür ist die Transition von `alarmoff` nach `alarmon`, ausgelöst durch einen Aufruf von `set`.

2. **Erreichen einer festgelegten Zeit:** Hier gibt es mehrere Möglichkeiten. Ein Beispiel ist das Verweilen in einem Zustand über eine festgelegte Dauer; dies trifft für eine Selbsttransition im

[16] Die anderen Attribute sollen auch nicht direkt von außen zugreifbar sein, sondern über sondierende Operationen (so genannte *Observer*). Ein Aufruf einer solchen Operation soll nur innerhalb des Zustandes, in dem der Observer steht, möglich und sinnvoll sein.

Zustand `timeupdate` zu, die nach etwa einer Sekunde Verweilen in diesem Zustand ausgelöst wird. Eine andere Möglichkeit zeigt die Transition von `quiet` nach `ringing`, bei der das auslösende Ereignis nicht explizit angegeben ist. Es muss anderweitig spezifiziert werden und wird aus dem Attribut `alarmtime` ermittelt.

Zu einigen Transitionen sind Aktionen angegeben, die Methodenaufrufe an anderen Objekten darstellen. Beim Übergang von `quiet` zu `ringing` wird beispielsweise die Methode `openwindow` am Objekt `W` aufgerufen. Das Fragezeichen hinter `C.time` bei der Selbsttransition von `timeupdate` deutet an, dass es sich hierbei um den Aufruf einer Funktion handelt.

Zu jedem Objectchart gehört noch eine Menge von so genannten Transitionsspezifikationen. Zu jeder Transition gibt es eine solche Spezifikation, welche die Wirkung der Transition auf die Attribute festlegt. Die Transition hat eine Vorbedingung, die erfüllt sein muss, damit die Transition ausgelöst werden kann, und eine Nachbedingung, welche die Wirkung der Transition auf die Attribute spezifiziert. Die Vorbedingung darf zusätzlich noch Zustandsnamen enthalten.

Zum obigen Beispiel wird in [CHB92] folgende Spezifikation für die Transition von `quiet` nach `ringing` genannt:[17]

$$\text{quiet} \rightarrow \text{ringing} : \{alarmtime \le timeofday < alarmtime + 1.5\} \,/\, W.openwindow\{\, finish = \overleftarrow{timeofday} + 60 \wedge timeofday = \overleftarrow{timeofday}\}$$

Nun ist ersichtlich, warum an der Transition kein auslösendes Ereignis stehen muss: Es wird einfach in der Vorbedingung angegeben. In der Nachbedingung werden die Belegungen von `finish` und `timeofday` nach Beendigung des Zustandsübergangs angegeben. Hierbei gilt, dass Attribute, die in der Nachbedingung nicht erwähnt werden, unverändert bleiben.

Die hier verwendeten Vor- und Nachbedingungen sind keine Zusicherungen im Sinne von [Meyer97], da es sich um Zusicherungen bei *Transitionen* handelt, nicht um Zusicherungen bei *Methoden*. Die Zusicherungen in den Transitionsspezifikationen lassen sich nicht unmittelbar auf Zusicherungen von Methoden abbilden. Man betrachte z. B. die Spezifikation für die Transition von `alarmoff` nach `alarmon`:

$$\text{alarmoff} \rightarrow \text{alarmon} : \{true\}\ set(t)\ \{alarmtime = t\}$$

Die Vorbedingung der Methode `set` ist nicht `true`, da ein Aufruf von `set` nur im Zustand `alarmoff` möglich ist. Allerdings kann auch nicht davon ausgegangen werden, dass die Vorbedingung `alarmoff` lautet, da es noch mehr Transitionen geben könnte, die durch `set` ausgelöst werden.

Neben den Transitionsspezifikationen wird noch die Angabe von Invarianten erlaubt. Eine Invariante gehört zu einem Zustand und stellt einen Ausdruck dar, der zu jedem Zeitpunkt wahr ist.

[17] Der übergestellte Pfeil über *timeofday* liefert den Wert, den das bezeichnete Attribut *vor* Ausführung der Methode hatte.

Die Autoren von [CHB92] geben keine Hinweise darauf, wie Objectcharts in Programmcode umgesetzt werden können. Sie formalisieren zwar gewisse Folgen von Methodenaufrufen über so genannte *Traces*, wobei sie jedoch nur prüfen, ob bestimmte Traces gültige Reihenfolgen von Methodenaufrufen darstellen; es wird nicht untersucht, wie bestimmt werden kann, ob ein *beliebiger* Trace gültig ist. Lediglich die Entscheidbarkeit dieses Problems wird gezeigt.

Mit den Objectcharts ist es nicht möglich, geschachtelte Methodenaufrufe zu modellieren. Jede Transition (und somit jeder Methodenaufruf) wird als atomar angesehen, weshalb Methoden lediglich nacheinander aufgerufen werden können. Andernfalls würde die Bestimmung des aktuellen Zustands problematisch werden: Angenommen während des Schaltens einer Transition würde eine andere Methode aufgerufen werden. Welche Transition soll diese nun auslösen? Es ist nicht klar, in welchem Zustand sich das System befindet. Ebenso ist unklar, ob der Zielzustand der ursprünglichen Transition am Ende trotzdem noch eingenommen werden soll.

3.3 OO-Zustandsdiagramme

Harels Zustandsdiagramme entstanden zunächst unter dem Paradigma der Strukturierten Analyse [DeMarco78]. Beim Erscheinungszeitpunkt von [Harel87] fand die Objektorientierung noch keine allgemeine Verbreitung. Mit der zunehmenden Bedeutung des objektorientierten Paradigmas entstand das Bedürfnis, Zustandsdiagramme auch in diesen Kontext einzubinden. Dies versucht Harel zusammen mit Eran Gery in dem Artikel [HG96]. Dabei führen sie keinen neuen Namen für die Diagramme ein; sie sprechen von OO-Zustandsdiagrammen oder einfach von Zustandsdiagrammen.

Die Autoren setzen viel Wert auf die automatische Codegenerierung und erwähnen ein selbst entwickeltes Tool *Rhapsody* [URL3], das hierzu in der Lage ist. Das Objektmodell des zu entwerfenden Systems muss hierfür in einem so genannten Objektmodelldiagramm spezifiziert werden, einer Mischung aus Klassen- und Objektdiagramm, die sich an OMT anlehnt, auf die hier jedoch nicht näher eingegangen werden soll.

Als Beschreibungsmittel für Objektinteraktion sind sowohl Ereignisse als auch Operationsaufrufe vorgesehen. Dies wird dadurch gerechtfertigt, dass Ereignisse leichter zu modellieren sein sollen und deshalb eher in der Analyse Verwendung finden, während Operationsaufrufe sich effizient in Code umsetzen lassen und daher näher am Design liegen. Ein ausgelöstes Ereignis wird in eine Warteschlange des Systems (bzw. des aktiven Threads) eingereiht; bei der nächsten stabilen Situation werden die eingereihten Ereignisse abgearbeitet. Der Unterschied zwischen Ereignissen und Operationsaufrufen ist, dass bei der Auslösung eines Ereignisses der Kontrollfluss sofort zum auslösenden Zustandsdiagramm zurückkehrt, während er beim Operationsaufruf so lange angehalten wird, bis die Operation beendet ist (d. h. bis das Zustandsdiagramm des aufgerufenen Objekts eine stabile Situation erreicht hat). Aus diesem Grund werden zyklische Aufrufbeziehungen ausgeschlossen.

Die Zustandsdiagramme, die zu jeder nicht-primitiven Klasse entworfen werden können, dürfen die volle in [Harel87] vorgestellte Mächtigkeit verwenden. Einschränkungen sehen die Autoren deshalb nicht für notwendig an, weil die Interaktion zwischen den einzelnen Objekten lediglich über Ereignisse und Operationsaufrufe stattfindet, die im objektorientierten Design leicht abgebildet werden können; die restlichen Mechanismen (z. B. interne Broadcast-Kommunikation) finden nur innerhalb des betrachteten Objekts statt. Von daher sind u. a. parallele Zustände, Zustandshierarchien, Gedächtniszustände und bedingte Verzweigungen möglich. Weiterhin kann auch die Erzeugung oder Zerstörung von Objekten angegeben werden und bei der Erzeugung eines neuen Objekts kann ein Codeblock als Initialisierungsskript spezifiziert werden.

Als Auslöser für Transitionen kommen Operationsaufrufe und ankommende Ereignisse in Frage (die jedoch nicht näher unterschieden zu werden brauchen). Ereignisse kommen vom System (aus der Warteschlange) und Operationsaufrufe von anderen Objekten.

Es ist deutlich ersichtlich, dass das Hauptaugenmerk von Harel und Gery auf der automatischen Codegenerierung liegt. Ihre OO-Zustandsdiagramme eignen sich zur Beschreibung von Systemen mit einem charakteristischen Zustandsverhalten, nicht jedoch zur Betrachtung des Verhaltens einzelner isolierter Klassen, da alle interagierenden Klassen modelliert werden und keine Klasse für sich allein betrachtet wird.

3.4 UML-Zustandsdiagramme

In der UML (Unified Modeling Language [BRJ99, URL2]) wurden mehrere objektorientierte Modellierungsansätze vereinigt. Im Wesentlichen sind dies die Booch-Methode [Booch94], OMT von James Rumbaugh (siehe [RBPEL91] und Abschnitt 3.1) sowie Objectory (OOSE) von Ivar Jacobson [Jacobson92], die sich in der Praxis bereits weltweit bewährt hatten. Hinzu kamen noch einige andere Ansätze, so auch Harels Zustandsdiagramme, die bereits in den drei genannten Ansätzen Verwendung fanden.

Die UML ist eine Modellierungssprache, die sowohl für Menschen als auch für Maschinen leicht lesbar sein soll und deren Modelle eine enge Affinität zu (objektorientiertem) Programmcode haben sollen. Es gibt neun Diagrammtypen, die zu unterschiedlichen Sichten passen.[18] Die Sichten werden wiederum in drei Hauptgebiete unterteilt: strukturelle Klassifikation, dynamisches Verhalten und Modellmanagement.

Zustandsdiagramme gehören zur Zustandsmaschinensicht im Bereich des dynamischen Verhaltens und sind stark an Harels Zustandsdiagramme angelehnt. Die Beschreibungen bewegen sich größtenteils auf der Ebene von Ereignissen und Aktionen und geben wenig Aufschluss darüber, wie Zustandsmaschinen in einer Programmiersprache zu implementieren sind. Dies ist auch nicht das erklärte Ziel der UML; zu diesem Zweck wird auf [JBR99] verwiesen – die Beschreibung des „Unified Process“ – wo jedoch Zustandsdiagramme nur am Rande erwähnt werden.

In [BRJ99, S. 337 f.] wird ein Beispiel gegeben, wie ein Zustandsdiagramm mithilfe der `Switch`-Kontrollstruktur auf Java-Code abgebildet werden kann; es wird jedoch darauf hingewiesen, dass der umgekehrte Weg in der Praxis nicht sehr nützlich sei, da ein Programm nicht von sich aus entscheiden könne, was sinnvolle Zustände seien.

Als Bestandteil der UML gilt die Object Constraint Language (OCL [URL2, Kapitel 6]), eine formale Sprache zur Beschreibung von Zusicherungen („Constraints“). Mit ihrer Hilfe lassen sich präzise Bedingungen formulieren.

In [RJB99, S. 447 f.] werden zwei verschiedene Verwendungsmöglichkeiten von Zustandsdiagrammen erwähnt: Zum einen lässt sich mit ihnen ausführbares Verhalten spezifizieren. Mithilfe von Aktionen (darunter Signale an andere Objekte) wird auf externe Ereignisse reagiert; die Zustandsmaschinen spezifizieren einen Teil des Systemverhaltens. – Die andere Möglichkeit ist die Verwendung zur Protokollspezifikation: Zustandsdiagramme werden dazu benutzt, erlaubte Aufrufreihenfolgen von Operationen anzugeben. Hierbei werden die Transitionen durch Operationsaufrufe ausgelöst. Eine solche Zustandsmaschine ist nicht direkt ausführbar, da sie lediglich die möglichen

[18] Eine Sicht ist eine Teilmenge von UML-Modellierungskonstrukten, die einen bestimmten Aspekt eines Systems repräsentiert [RJB99].

Folgen von Operationsaufrufen spezifiziert und auch Nichtdeterminismus enthalten kann. Diese Verwendungsmöglichkeit wird jedoch kaum näher beschrieben, sie findet sich aber auch an anderen Stellen in der Literatur, z. B. in [LLMT00a]. Dort werden Zustandsdiagramme auf Petrinetze [Reisig86, Baumgarten96] abgebildet, die dann die Objektprotokolle beschreiben. Die aktivierten Transitionen entsprechen den Operationen, die im gegenwärtigen Zustand aufgerufen werden dürfen.

4 Beschreibung von Objektzuständen mit dem Vertragsmodell

Der erste Abschnitt dieses Kapitels beinhaltet eine allgemeine Einführung in das Vertragsmodell und dessen Anwendung in Java. Im zweiten Abschnitt wird dann erläutert, wie Objektzustände mit dem Vertragsmodell beschrieben werden. Es wird ein direkter Zusammenhang zwischen Zustandsdiagrammen und dem Vertragsmodell hergestellt.

4.1 Einführung in das Vertragsmodell

Das Vertragsmodell [Meyer92a, Meyer92b, Meyer97] betrachtet die Bereitstellung von Operationen als ein Anbieten von Dienstleistungen. Erhält eine Klasse eine Anfrage zur Ausführung einer Operation, so fungiert diese Klasse als *Lieferant*, während die aufrufende Klasse die Rolle des *Kunden* einnimmt. Beide Klassen gehen ein Vertragsverhältnis ein und müssen sich an festgelegte Regeln halten, damit der Vertrag erfüllt wird. Diese Regeln werden als Zusicherungen bezeichnet und in Vorbedingungen, Nachbedingungen und Klasseninvarianten eingeteilt. Der Kunde muss die Vorbedingungen einhalten, während der Lieferant die Erfüllung der Nachbedingungen und Klasseninvarianten sicherstellen muss.

Das Nicht-Erfüllen einer Zusicherung wird im Vertragsmodell als Programmierfehler angesehen. Zusicherungen dürfen also nicht als Ersatz für Kontrollstrukturen (wie beispielsweise `if`-Abfragen) verwendet werden. Wird eine Zusicherung nicht eingehalten, so ist das Programm fehlerhaft und muss korrigiert werden. Laut Meyer darf ein Programm im Falle einer nicht erfüllten Zusicherung Beliebiges machen, u. a. abstürzen oder in eine Endlosschleife gehen [Meyer97, S. 343]. Angemessener ist jedoch ein Programmabbruch mit detaillierter Fehlermeldung.

Die Verwendung des Vertragsmodells hilft dabei, Programmierfehler frühzeitig aufzudecken. Wird eine Zusicherung nicht eingehalten, so ist der Fehler in der Regel leicht lokalisierbar. Zusicherung helfen dabei, korrekte Software zu bauen, obwohl die vollständige Korrektheit durch Zusicherungen selbstverständlich nicht gewährleistet werden kann. – Die Verwendung von Vorbedingungen erspart auch viele `if`-Abfragen, da die Operationen sich nun darauf verlassen können, dass die Vorbedingungen gelten. Ein kompakterer und besser lesbarer Code ist die Konsequenz. Obwohl der Einsatz des Vertragsmodells noch nicht allzu verbreitet ist, wird er doch mittlerweile vielfach empfohlen [Züllighoven98, HT00].

Java bietet derzeit keine Sprachunterstützung für Zusicherungen, wie das z. B. bei der Programmiersprache Eiffel [Meyer92c] der Fall ist. Die `assert`-Anweisung, die ab der Java-Version 1.4 Bestandteil der Sprache ist, ist zwar ein Schritt in die Richtung von Zusicherungen, allerdings von einer vollwertigen Umsetzung des Vertragsmodells noch weit entfernt, da z. B. nicht zwischen Vor- und Nachbedingungen unterschieden wird und die Besonderheiten des Vertragsmodells bei Vererbung nicht berücksichtigt werden. Im JWAM-Rahmenwerk [URL1] wurde daher ein eigener Standard für Zusicherungen integriert, der im Folgenden beschrieben wird. Er schränkt das Vertragsmodell jedoch insbesondere im Hinblick auf Vererbung ein.

4.1.1 Vorbedingungen

Eine Vorbedingung ist ein boolescher Ausdruck, der erfüllt sein muss, wenn eine Operation aufgerufen wird. Die aufrufende Klasse in der Rolle des Kunden muss sicherstellen, dass die Vorbedin-

gungen einer Operation erfüllt sind, wenn sie diese Operation aufruft. Ansonsten hat sie den Vertrag gebrochen und dies ist – wie oben beschrieben – ein Programmierfehler.

Vorbedingungen werden in JWAM in einem `@require`-Tag innerhalb des Dokumentationskommentars[19] angegeben. Dies ist kein Java-Standard-Tag, wird aber vom JWAM-Doclet[20] unterstützt.

Die Überprüfung der Zusicherungen zur Laufzeit erfolgte bis zur JWAM-Version 1.7 mit der Klasse `Contract`. Diese Klasse enthält zu diesem Zweck verschiedene statische Methoden. Für Vorbedingungen gibt es beispielsweise:

```
public static void require(boolean precondition,
     Object contractor, String description);
```

Dabei bezeichnet `precondition` die Vorbedingung, `contractor` das Objekt, das die Dienstleistung anbietet, und `description` eine Beschreibung der Vorbedingung, die im Fehlerfall ausgegeben wird.

Das Überprüfen einer Vorbedingung geschieht typischerweise wie folgt:

```
Contract.require(size() > 0, this, "size() > 0");
```

Wird die Vorbedingung zu `false` ausgewertet, so wird eine Runtime-Exception geworfen. – Der Aufruf von `Contract.require` erfolgt immer zu Beginn der Operation.[21] Bei Konstruktoren darf er allerdings erst hinter `super(...)` bzw. `this(...)` stehen, da diese Anweisungen die ersten in einem Konstruktor sein müssen.

Ab der JWAM-Version 1.8.0 werden Zusicherungen durch `assert`-Anweisungen geprüft, da dies effizienter ist und ein bequemes Ein- und Ausschalten der Laufzeitüberprüfung durch Setzen von Optionen der virtuellen Maschine ermöglicht. Eine Vorbedingung wird wie folgt ausgewertet:

```
assert size() > 0 : "Precondition violated: " + "size() > 0";
```

Die Zeichenkette hinter dem Doppelpunkt wird an den Konstruktor der Klasse `AssertionError` übergeben, sodass bei Nichterfüllung der Vorbedingung eine passende Fehlermeldung ausgegeben wird.

Die Zusicherungen werden in JWAM also doppelt formuliert, einmal im Dokumentationskommentar (zur Dokumentation der Zusicherungen) und einmal im Programmcode (zu ihrer Auswertung). In den Beispielen zu Zusicherungen werden im Folgenden nur die entsprechenden Tags angegeben; der Code zur Auswertung wäre redundant.

Ein Beispiel aus JWAM für Vorbedingungen ist Folgendes:

[19] der in /** ... */ eingeschlossene Kommentar vor der Deklaration der Operation, der von Javadoc ausgewertet wird um die Operation zu dokumentieren

[20] Ein Doclet generiert aus Dokumentationskommentaren Beschreibungen der Operationen, indem die Tags ausgewertet werden.

[21] Es sind auch mehrere Aufrufe von `Contract.require` hintereinander möglich, wenn es mehrere Vorbedingungen gibt.

```
/**
 * @require s != null
 * @require canCreateFromString()
 */
public boolean isValid(String s);
```

(de.jwam.lang.domainvalue.DomainValue)

Die Methode `isValid` hat zwei Vorbedingungen:

1. `s != null`
2. `canCreateFromString()`

Beide Vorbedingungen hätten auch zu einer einzigen zusammengefasst werden können, indem sie durch logisches Und verknüpft worden wären. Die angegebene Variante lässt allerdings besser zur Geltung kommen, dass die beiden Vorbedingungen nicht zueinander in Beziehung stehen.

4.1.2 Nachbedingungen

Eine Nachbedingung ist ein boolescher Ausdruck, der erfüllt sein muss, wenn eine Operation beendet ist. Für die Sicherstellung der Erfüllung der Nachbedingungen ist die Klasse verantwortlich, die die Rolle des Lieferanten einnimmt, die also die Operation anbietet.

In JWAM werden Nachbedingungen durch ein `@ensure`-Tag angegeben. Im Programmcode erfolgte bis zu JWAM 1.7 ein Aufruf der Methode `Contract.ensure`, die ähnlich wie `Contract.require` aufgerufen wird[22], allerdings am Ende der Operation steht (jedoch vor einer eventuellen `return`-Anweisung). Ab JWAM 1.8.0 erfolgt die Überprüfung durch eine `assert`-Anweisung mit dem Hinweis `"Postcondition violated"`.

Bei Nachbedingungen gibt es laut JWAM-Konvention das aus Eiffel [Meyer92c] übernommene Schlüsselwort `result`, das den Rückgabewert einer Funktion bezeichnet.

Ein Beispiel mit Nachbedingungen ist:

```
/**
 * @ensure result != null
 * @ensure ArrayUtil.containsNoNulls(result)
 */
public JwamAction[] actions();
```

(de.jwam.handling.toolconstruction.basicconstruction.ToolFunctionalityImpl)

Eine Besonderheit bei Nachbedingungen bildet das Schlüsselwort `old`, das ebenfalls aus der Sprache Eiffel übernommen wurde. Mit ihm ist es möglich, auf den Wert eines Teilausdrucks *vor* Aus-

[22] Lediglich in der Meldung der `ContractBrokenException` wird zwischen Vor- und Nachbedingung unterschieden.

führung der Operation zuzugreifen. Will man derartige Nachbedingungen zur Laufzeit überprüfen, so muss der Wert des Ausdrucks in einer lokalen Variable gespeichert werden.

In JWAM kommen Nachbedingungen mit `old`-Ausdrücken nur in der Klasse `Account` aus dem Cookbook vor:

```
/**
 * @require amount > 0.0
 * @ensure balance() == old balance() + amount
 */
public void deposit(float amount);
```

(de.jwamexample.cookbook.step01_material.Account)

Hier wird zugesichert, dass der Saldo des Kontos sich beim Einzahlen um den eingezahlten Betrag erhöht.

Bei `old`-Ausdrücken gelten folgende Regeln:[23]

- `old !a` ist äquivalent zu `!old a`.
- `old (a && b)` ist äquivalent zu `old a && old b` (ebenso bei `||` und `^`).
- `old true` ist äquivalent zu `true`; `old false` ist äquivalent zu `false`.
- `old old a` ist unzulässig, da zu Beginn der Operationsausführung kein `old`-Ausdruck erlaubt ist.

Es sei noch erwähnt, dass die Verwendung von `old`-Ausdrücken eher ungewöhnlich ist (s. o.) und die Lesbarkeit von Nachbedingungen oft unnötig erschwert. Sie werden jedoch gebraucht um eine vollständige gegenseitige Abbildung von Zusicherungen und Zustandsdiagrammen zu ermöglichen.

4.1.3 Invarianten

Eine Invariante ist eine Zusicherung, die nicht an eine Operation gebunden ist, sondern an eine Klasse. Zu jedem Zeitpunkt, an dem keine Operation der Klasse ausgeführt wird, muss die Invariante erfüllt sein. Der erste Zeitpunkt, zu dem sie gültig ist, ist nach Ausführung des Konstruktors. Weiterhin ist sie vor und nach jedem Methodenaufruf erfüllt, jedoch nicht notwendigerweise während der Ausführung einer Operation.

Invarianten werden durch `@invariant` gekennzeichnet und dürfen im Dokumentationskommentar einer Klasse, eines Interface oder eines Attributs stehen.

In JWAM gibt es keine einzige Invariante, die bei der Klasse bzw. dem Interface angegeben ist. Sämtliche Invarianten beziehen sich lediglich auf ein Attribut:

[23] Diese Regeln werden in Abschnitt 4.2.2 benötigt. Dass sie gelten, ist unmittelbar einsichtig.

```
/**
 * @invariant _name != null
 */
private String _name;
```

(de.jwam.handling.thing.ThingDescriptionDV)

Derartige Invarianten sind für Klassen in der Rolle des Kunden unbrauchbar, da sie auf die privaten Attribute keinen Zugriff haben.

Jede Operation der Klasse muss sicherstellen, dass nach ihrer Ausführung die Invariante erfüllt bleibt. Theoretisch könnte sie die Invariante als Nachbedingung übernehmen (und gegebenenfalls noch weitere Nachbedingungen hinzufügen). Dies ist aber im Vertragsmodell nicht vorgeschrieben, da jede Operation in der Formulierung ihres Vertrages frei ist.

4.1.4 Zusicherungen bei Vererbung

Wird eine Methode in einer Unterklasse redefiniert, so können sich deren Zusicherungen ändern. Diese Änderungen dürfen allerdings nicht beliebig sein, da das Subtypenkonzept beachtet werden muss: Aufgrund des dynamischen Bindens muss eine Unterklasse überall dort einsetzbar sein, wo der Typ der Oberklasse verlangt wird. Deshalb darf eine Vorbedingung nicht verschärft werden, da sonst ein Klient, der sich an die Vorbedingung aus der Oberklasse hält, die verschärfte Vorbedingung möglicherweise nicht einhält. Ebenso dürfen Nachbedingungen und Invarianten nicht abgeschwächt werden, weil ein Klient sich auf die entsprechenden Zusicherungen in der Oberklasse verlassen können muss.

Konkret bedeutet dies, dass die Vorbedingung einer Methode der Oberklasse die entsprechende Vorbedingung in der Unterklasse implizieren muss. Bei Nachbedingungen und Invarianten ist es genau umgekehrt.[24] Bei diesen Betrachtungen werden sämtliche Vorbedingungen einer Methode durch Und-Verknüpfung zu einer einzigen zusammengefasst, ebenso die Nachbedingungen und Invarianten.

Invarianten dürfen in Unterklassen verschärft werden (vgl. [Meyer97, S. 570] und [Meyer92b, S. 48]). Die Behauptung, dass die Invariante implizit mit den Vor- und Nachbedingungen der Methoden konjunktiv verknüpft wird und deshalb die Verschärfung der Invariante zugleich eine (illegale) Verschärfung der Vorbedingungen mit sich zieht, ist nicht aufrechtzuerhalten. Erstens bezieht sich eine Invariante niemals auf Methodenparameter (da sie zur Klasse gehört und auf Methodenparameter keinen Zugriff hat) und zweitens geschieht das Verknüpfen mit den Vor- und Nachbedingungen *implizit*; es ist eher technischer, nicht konzeptioneller Natur. Außerdem hat der Klient keine Möglichkeit, die Invariante zu verletzen, da sie ohnehin vor und nach jedem Methodenaufruf gilt. Wäre das Verschärfen von Invarianten verboten, so müsste ferner jede Java-Klasse die Invariante `true` besitzen, da dies die Invariante von `java.lang.Object` ist, der Wurzelklasse jeder Klassenhierarchie.

In JWAM gibt es derzeit keine Unterstützung für Zusicherungen im Hinblick auf Vererbung. Der Programmierer muss selber darauf achten, dass die Regeln des Vertragsmodells bei Vererbung eingehalten werden.

[24] siehe auch [Meyer97, Abschnitt 16.1], [LW93] und [LW94]

4.2 Zusicherungen und Objektzustände

In diesem Abschnitt werden die grundlegenden Ideen vorgestellt, wie Zusicherungen und Objektzustände zusammenhängen. Die Zusicherungen werden hierzu in Prädikate zerlegt, wobei zwischen Prädikaten, die die Parameter der Operation betreffen, und solchen, die den Objektzustand betreffen, unterschieden wird. Es wird erläutert, auf welche Elemente von Zustandsdiagrammen die verschiedenen Prädikate abgebildet werden. Somit wird ein genereller Zusammenhang zwischen dem Vertragsmodell und Zustandsdiagrammen hergestellt.

4.2.1 Prädikate

Grundlegend bei der Betrachtung von Zusicherungen ist der Begriff des Prädikats. Ein Prädikat ist in diesem Fall eine atomare Einheit einer Zusicherung, die einen booleschen Wert liefert. Der Begriff *Prädikat* wird auch in [Daugherty98] verwendet um Bedingungen zu beschreiben, die in Zuständen gelten (dort ist sogar schon von *Zustands-Prädikaten* die Rede). Transitionen in Zustandsdiagrammen werden als *Prädikaten-Transformierer* bezeichnet (siehe auch [Gries81, S. 109]); das kommt dem Ansatz in dieser Arbeit bereits sehr nahe.

Definition 4.1: *Prädikate* sind die booleschen Teilausdrücke einer Zusicherung, die sich selbst nicht unmittelbar aus booleschen Teilausdrücken zusammensetzen.

Teilausdrücke, die sich *unmittelbar* aus booleschen Teilausdrücken `a` und `b` zusammensetzen, sind:

- `!a`
- `a && b`
- `a || b`
- `a ^ b`
- `old a`

In all diesen Fällen handelt es sich *nicht* um Prädikate im Sinne der obigen Definition.

Allerdings können Prädikate indirekt doch boolesche Teilausdrücke oder das Schlüsselwort `old` enthalten, wenn sie nicht auf eine der obigen Arten verknüpft sind. Bei folgenden Beispielen handelt es sich um Prädikate:

- `function(a && b)`
- `balance() == old balance() + amount`

4.2.2 Vor- und Nach-Prädikate

Bei der Angabe der Prädikate in den Zusicherungen wird zwischen Vor- und Nach-Prädikaten unterschieden. Die Unterscheidung beruht hierbei auf dem Zeitpunkt, zu dem der Wert der Prädikate abgefragt wird. Bei Vor-Prädikaten wird der Wert des Prädikats vor Ausführung der Operation betrachtet, bei Nach-Prädikaten der Wert danach:

Definition 4.2: *Vor-Prädikate* sind entweder Prädikate in Vorbedingungen oder durch vorangestelltes `old` gekennzeichnete Prädikate in Nachbedingungen. *Nach-Prädikate* sind Prädikate in Nachbedingungen, die nicht durch vorangestelltes `old` gekennzeichnet sind.

Vorbedingungen enthalten ausschließlich Vor-Prädikate. Nachbedingungen enthalten überwiegend Nach-Prädikate, aber auch durch vorangestelltes `old` kenntlich gemachte Vor-Prädikate.

Nicht immer weist das Schlüsselwort `old` auf ein Vor-Prädikat hin. Befindet sich `old` innerhalb eines Prädikats, so bezieht es sich nicht auf das gesamte Prädikat, sondern nur auf einen Teilausdruck. Das Prädikat

```
balance() == old balance() + amount
```

ist daher ein Nach-Prädikat, denn die Auswertung dieses Prädikats kann erst nach Ausführung der Operation erfolgen.

Bei Vor-Prädikaten in Nachbedingungen ist zu beachten, dass diese nicht beliebig auftreten dürfen. Sichert eine Methode beispielsweise `old isDefined()` zu, ohne dass es eine Vorbedingung oder eine Invariante gibt, so kann die Erfüllung dieser Nachbedingung nicht sichergestellt werden, da die Methode keine Kontrolle über den Wert von `isDefined()` vor der Methodenausführung hat.[25]

Es muss also sichergestellt werden, dass Nachbedingungen mit Vor-Prädikaten erfüllbar sind. Dazu geht man wie folgt vor:

1. Zuerst werden alle Invarianten durch Und-Verknüpfung zu einer einzigen Invariante *inv* zusammengefasst. Das Gleiche gilt für die Vor- und Nachbedingungen der entsprechenden Operation. Die Vorbedingung heiße nun *pre*, die Nachbedingung *post*.
2. Die Nachbedingung wird nun in disjunktive Normalform umgeformt.
3. Jeder Minterm wird so umgeformt, dass er genau einen `old`-Teilausdruck enthält. Notfalls wird ein neuer Teilausdruck `old true` eingeführt.
4. Sämtliche Teilausdrücke, die keine `old`-Teilausdrücke sind (also die Nach-Prädikate), werden aus der Nachbedingung entfernt.
5. Alle Vorkommen von `old` werden entfernt, d. h. alle Vor-Prädikate durch Nach-Prädikate ersetzt. Der neu entstandene Ausdruck heiße *post'*.
6. Die Nachbedingung ist genau dann korrekt in Bezug auf Vor-Prädikate, wenn gilt:
   $inv \wedge pre \rightarrow post'$

Dass obiger Satz gilt, kann man sich leicht klar machen: Aus der Nachbedingung wurden sämtliche Nach-Prädikate entfernt (Punkt 4). Das Schlüsselwort `old` wurde entfernt um den Ausdruck an die Vorbedingung und die Invariante anzupassen (Punkt 5). Der Satz besagt nun, dass, wenn Invariante *inv* und Vorbedingung *pre* gelten, auch der aus den Vor-Prädikaten der Nachbedingung gebildete Ausdruck *post'* gilt, also kein Fall auftritt, in dem die Vor-Prädikate der Nachbedingung mehr zusichern als die Vorbedingung und die Invariante.

Das folgende Beispiel geht die obigen Schritte zur Verdeutlichung durch (die Invariante sei `true`)[26]:

[25] Über den Wert von `isDefined()` *nach* der Methodenausführung macht diese Nachbedingung keine Aussage.
[26] Man beachte, dass `old` stärker als `&&` bindet und `&&` stärker als `||`.

Vorbedingung: `a || (b && c)`
Nachbedingung: `old (a || (b && c)) && d || old (a && b)`

1. Es gibt bereits jeweils eine Vor-, Nachbedingung und Invariante.
2. Nachbed.: `old a && d || old b && old c && d || old a && old b`
3. `old a && d || old (b && c) && d || old (a && b)`
4. `old a || old (b && c) || old (a && b)`
5. `a || (b && c) || (a && b)`
6. Da der obige Ausdruck äquivalent zur Vorbedingung ist, ist die Nachbedingung in Bezug auf Vor-Prädikate korrekt.

4.2.3 Parameter-Prädikate

Die Prädikate sollen in den beiden folgenden Abschnitten in Parameter- und Zustands-Prädikate eingeteilt werden. Parameter-Prädikate sagen etwas über einen oder mehrere Parameter der Operation aus:

Definition 4.3: Ein *Parameter-Prädikat* ist ein Prädikat, das mindestens einen Parameter (Eingabe- oder Rückgabeparameter) der zur Zusicherung gehörenden Operation enthält.

Parameter-Prädikate sind also daran erkennbar, dass sie den Namen eines Parameters oder das Schlüsselwort `result` enthalten.

Ein Beispiel mit Parameter-Prädikaten ist:

```
/**
 * @require user != null
 * @ensure creator().equals(user)
 */
public void setCreation(UserIdentificatorDV user);
```

(de.jwamx.handling.registry.Registerable)

In diesem Beispiel bestehen beide Zusicherungen aus genau einem Parameter-Prädikat.

Bei der Nachbedingung handelt es sich um ein Nach-Parameter-Prädikat, da die Auswertung nach Ausführung der Methode stattfindet. Parameter-Prädikate, die das Schlüsselwort `result` enthalten, dürfen ausschließlich als Nach-Parameter-Prädikate auftreten.

Vor-Parameter-Prädikate werden im Zustandsdiagramm auf Wächterbedingungen abgebildet, da sie Bedingungen angeben, die beim Zustandsübergang erfüllt sein müssen. Im obigen Beispiel würde jede durch `setCreation` ausgelöste Transition die Wächterbedingung `[user != null]` erhalten.

Nach-Parameter-Prädikate werden dagegen auf Constraints abgebildet, weil sie Bedingungen definieren, die nach dem Schalten der Transition gelten. Die Transitionen von `setCreation` würden daher als Constraint {`creator().equals(user)`} erhalten.

Nach-Parameter-Prädikate, die etwas über Eingabeparameter aussagen, könnten durchaus so angesehen werden, dass sie eine Änderung des Objektzustands beschreiben. Eine Nachbedingung wie

```
@ensure creator().equals(user)
```

beschreibt ja eine Änderung des Objekts. Der resultierende Zustand lässt sich jedoch nicht im Vorhinein bestimmen, da er vom Parameter abhängt. Wollte man solche Nachbedingungen in die Zustände einbeziehen, so müssten die entsprechenden Parameter ebenfalls den Zuständen zur Verfügung gestellt werden. Dies macht aber insbesondere bei Referenzparametern wenig Sinn, weil es nahezu beliebig viele Belegungsmöglichkeiten gibt. Auch die Prüfung solcher Bedingungen in Zuständen bei der Auswertung von Vorbedingungen ist nicht ohne Weiteres möglich. Bei einer Vorbedingung wie

```
@require creator().equals(user2)
```

wäre zu prüfen, ob `user` und `user2` auf dasselbe Objekt verweisen, also `user==user2` gilt. Unter Zuhilfenahme der OCL wäre eine solche Prüfung möglich. In diesem Fall wäre jedoch `user.equals(user2)` ausreichend, was aber eine Kenntnis der Semantik dieser Ausdrücke voraussetzt.[27]

Außerdem haben Untersuchungen am JWAM-Rahmenwerk ergeben, dass es in der Praxis sehr selten vorkommt, dass dasselbe Parameter-Prädikat sowohl in einer Vorbedingung als auch in einer Nachbedingung (nicht notwendigerweise derselben Operation) vorkommt. Selbst der Fall, dass sowohl die positive als auch die negative Form eines Parameter-Prädikats anzutreffen ist, tritt praktisch kaum auf. Von daher lohnt sich der zusätzliche Aufwand, Parameter-Prädikate in die Zustände einzubeziehen, nicht. Die Lesbarkeit der Diagramme sänke um einiges, ohne dass ein praktischer Nutzen erkennbar wäre.

4.2.4 Zustands-Prädikate

Alle übrigen Prädikate sind Zustands-Prädikate:

Definition 4.4: Ein *Zustands-Prädikat* ist ein Prädikat, das kein Parameter-Prädikat ist.

Zustands-Prädikate sind diejenigen Prädikate, die sich auf den Objektzustand beziehen. Beispiele für Zustands-Prädikate sind[28]:

```
/**
 * @require hasLock()
 * @require !isLocked()
 * @ensure !hasLock()
 * @ensure hasRemovedLock()
 */
public void removeLock();
```

(de.jwam.handling.accesscontrol.Lockable)

[27] Transitivität und Symmetrie der `equals`-Methode
[28] Die Funktionen (ohne die Negationen) sind die Zustands-Prädikate.

Dieser Fall (parameterlose boolesche Funktionen) ist die bei weitem häufigste Form von Zustands-Prädikaten.

Weitere Möglichkeiten sind:

```
/**
 * @ensure elementCount() == 0
 */
public void removeAll();
```

(de.jwamx.technology.iafpf.swingpf.JMultipleSelectionPanelPF)

```
/**
 * @require materials().length > 0
 */
public void setupSubTools();
```

(de.jwambeta.handling.formtools.FormEditorTool)

Vorbedingungen von Konstruktoren können keine Zustands-Prädikate enthalten, weil der Objektzustand vor Ausführung des Konstruktors nicht vom Klienten beeinflussbar ist. Daher machen in Vorbedingungen von Konstruktoren ausschließlich Parameter-Prädikate Sinn. Invarianten hingegen enthalten ausschließlich Zustands-Prädikate, da sie sich auf die gesamte Klasse beziehen und auf die Parameter der Operationen keinen Zugriff haben.

4.2.5 Objektzustände

Der gängige Begriff des Objektzustands bezeichnet die momentane Belegung der Attribute (vgl. [RBPEL91, S. 84]). In [Meyer97, S. 756 ff.] wird beim Objektzustand zwischen dem *konkreten* und dem *abstrakten Zustand* unterschieden. Der konkrete Zustand wird durch die Belegungen der Attribute charakterisiert; der abstrakte Zustand ist der von außen sichtbare Zustand. Liefern alle öffentlichen Funktionen vor und nach einer Änderung des konkreten Zustands dieselben Werte, so hat sich der abstrakte Zustand nicht geändert.

Für Zustandsdiagramme sind diese Sichtweisen jedoch höchst unpraktikabel, da die Anzahl der Zustände dann viel zu groß ist. Wird jedes Mal ein anderer Zustand eingenommen, wenn eine (öffentliche) Funktion einen anderen Wert liefert, dann ist die Anzahl der Zustände nicht mehr überschaubar; auch die Unterschiede zwischen den einzelnen Zuständen sind von geringer Relevanz, da es sehr viele Zustände gibt, die konzeptionell ähnlich sind.

Eine differenziertere Betrachtung findet sich in [Jacobson92, S. 234], wo zwischen dem *internen Zustand* („internal state“) und dem *Berechnungszustand* („computational state“) unterschieden wird. Während sich der interne Zustand aus der Belegung der Attribute zusammensetzt, beschreibt der Berechnungszustand den aktuellen Stand der Ausführung. Berechnungszustände legen die Operationen fest, die in diesen Zuständen aufgerufen werden dürfen, und eignen sich zur Entwicklung von Zustandsdiagrammen.

Die UML-Definition eines Zustands ist sehr universell [URL2, Abschnitt 3.75.1]:

“A state is a condition during the life of an object or an interaction during which it satisfies some condition, performs some action, or waits for some event.”

Da in dieser Arbeit Interaktionen nicht betrachtet werden und Aktionen sowie das Warten auf Ereignisse nichts mit dem Protokoll der Klasse zu tun haben, bleibt hier nur die Aussage, dass ein Zustand eine Bedingung während des Lebenszyklus eines Objekts ist. Dies ist sehr allgemein, denn die Bedingungen, die einen Zustand ausmachen, sind individuell festlegbar.

Die Bedingungen, die in dieser Arbeit zur Beschreibung von Objektzuständen verwendet werden, sind die Zustands-Prädikate der Klasse. Der Zustand muss sich dabei nicht ausschließlich aus den eigenen Attributen der Klasse zusammensetzen, wie das letzte Beispiel aus dem vorigen Abschnitt zeigt: Die Vorbedingung

```
@require materials().length > 0
```

nimmt auf ein Attribut eines anderen Objekts Bezug. Der Objektzustand kann sich somit ändern, wenn die Attribute des eigenen Objekts unverändert bleiben, aber ein Attribut in einem Objekt modifiziert wird, das aus dem eigenen Objekt heraus referenziert wird.

Der hier verwendete Zustandsbegriff bezieht sich demnach auf Berechnungszustände im Sinne von [Jacobson92]:

Definition 4.5: Der *Objektzustand* wird beschrieben durch die momentane Belegung sämtlicher in der Klasse vorkommenden Zustands-Prädikate.

Der Objektzustand wird also nicht dadurch geändert, dass sich die Belegung eines Attributs ändert, sondern dadurch, dass sich der Wert eines Zustands-Prädikats ändert. Diese Sichtweise reduziert die Anzahl der Zustände auf ein für Zustandsdiagramme geeignetes Maß.

Eine Klasse mit n Zustands-Prädikaten hat daher maximal 2^n Zustände, von denen nicht alle erreichbar sein müssen. Die Belegungen der Zustands-Prädikate werden in die Zustände geschrieben:

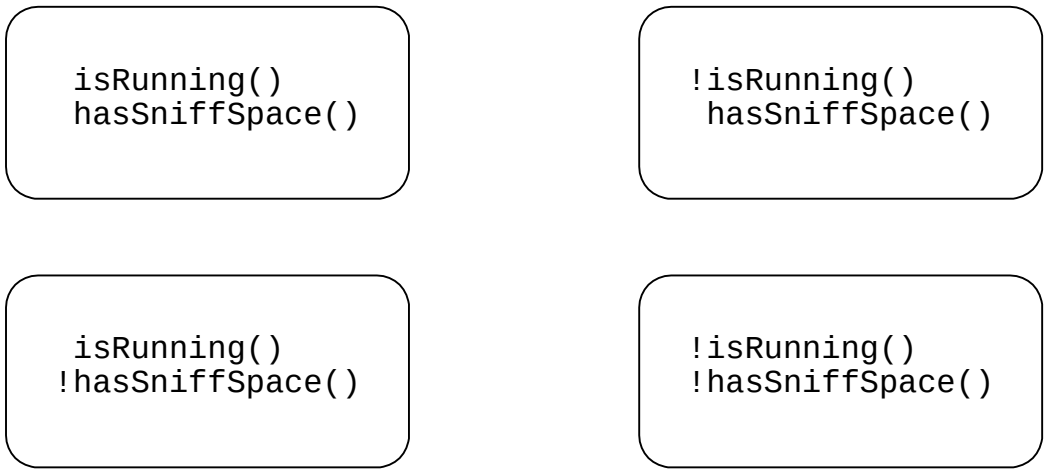


Abbildung 4.1: Zustände der Klasse de.jwam.handling.containerconstruction.SnifferImpl

Gibt es nur einen Zustand – d. h. keine Zustands-Prädikate – so wird dieser mit `true` beschriftet. In diesem Zustand verweilt das Objekt dann für seine gesamte Lebensdauer.

4.2.6 Prädikate mit Seiteneffekten

Funktionen mit Seiteneffekten werfen häufig Probleme auf. In [Meyer97] wird zwischen *konkreten* und *abstrakten Seiteneffekten* unterschieden.[29] Konkrete Seiteneffekte sind Zuweisungen oder Zuweisungsversuche an Attribute oder Prozeduraufrufe. Abstrakte Seiteneffekte sind konkrete Seiteneffekte, die den abstrakten Zustand[30] des Objekts verändern, d. h. die dazu führen, dass mindestens eine öffentliche Funktion einen anderen Wert liefert. Funktionen sollten keine abstrakten Seiteneffekte produzieren [Meyer97, S. 751]; konkrete Seiteneffekte sind jedoch erlaubt, sofern sie den abstrakten Zustand nicht beeinflussen.

Übertragen auf den in dieser Arbeit verwendeten Zustandsbegriff bedeutet das, dass Funktionen keine Prädikate ändern dürfen. Wir können aber noch einen Schritt weiter gehen und lediglich fordern, dass bei der Auswertung von Prädikaten keine Prädikate geändert werden. Funktionen, die nicht in Prädikaten vorkommen, können den Zustand ändern (auch wenn dies nicht zu empfehlen ist); wenn allerdings Funktionen, die in Zustands-Prädikaten vorkommen, ein Zustands-Prädikat ändern dürften, so gäbe es instabile Zustände. Betrachten wir beispielsweise folgenden Zustand:

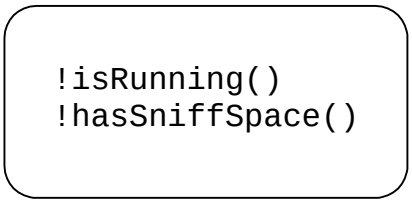


Abbildung 4.2: Beispielzustand

Würde ein Aufruf der Funktion `isRunning` dazu führen, dass `isRunning()` oder `hasSniffSpace()` `true` liefert, so fände bei der Auswertung des Zustands-Prädikats ein spontaner Zustandsübergang statt.[31] Noch schlimmer wäre es, wenn `isRunning` abwechselnd `true` und `false` liefern würde, weil dann der Zustand permanent gewechselt würde. Abgesehen davon ist nicht spezifiziert, wann und wie oft ein Zustands-Prädikat ausgewertet wird.

Ähnliches gilt für Parameter-Prädikate. Dürften Parameter-Prädikate andere Parameter-Prädikate ändern, so wäre die Auswertung der Vor- bzw. Nachbedingungen von der Reihenfolge abhängig. Hierzu ein Beispiel:

```
/**
 * @require id != null
 * @require has(id)
 * @require isRemovable(id)
 * @ensure !has(id)
 */
public void remove(IdentificatorDV id);
```

(de.jwam.handling.containerconstruction.Container)

[29] S. 749 bzw. 757
[30] zum Begriff des abstrakten Zustands siehe voriger Abschnitt
[31] Alternativ würde das Objekt weiterhin in diesem Zustand verweilen, aber die Bedingung würde nicht mehr gelten.

Würde z. B. die Funktion `isRemovable` Änderungen durchführen, sodass `has(id)` anschließend `false` zurückgibt, so wären die Vorbedingungen nur bei der ersten Auswertung gültig. Dies ist jedoch nicht im Sinne des Vertragsmodells, da die Methode `remove` sich auf die Gültigkeit der Vorbedingungen verlässt, egal wie oft sie ausgewertet werden.

In obigem Beispiel spielt es keine Rolle, ob es sich um Parameter- oder Zustands-Prädikate handelt; sie dürfen sich nicht gegenseitig beeinflussen. Daher wird in dieser Arbeit gefordert, dass sämtliche Prädikate (sowohl Zustands- als auch Parameter-Prädikate) keine Auswirkungen auf irgendwelche Prädikate haben dürfen. Diese Einschränkung ist allerdings ganz im Sinne eines sauberen objektorientierten Entwurfs gemäß [Meyer97], sodass man nicht von einer echten Einschränkung sprechen kann.

Eine weitere Einschränkung, die vorgenommen werden muss, ist, dass Funktionen, die in Prädikaten vorkommen, keine Vorbedingungen mit Zustands-Prädikaten haben dürfen. Das liegt daran, dass Prädikate in jedem beliebigen Zustand abgefragt werden sollen; dies darf nicht durch Vorbedingungen eingeschränkt sein. Auch dies ist keine echte Einschränkung, da die Funktionen, aus denen Prädikate zusammengesetzt sind, praktisch immer universelle Abfragefunktionen sind, die zu jedem Zeitpunkt aufgerufen werden dürfen.

4.2.7 Änderungssemantiken

Die Zustands-Prädikate der Vorbedingungen bestimmen die Zustände, in denen ein Aufruf der Methode möglich ist. Ähnlich legen die Zustands-Prädikate (genauer: Nach-Zustands-Prädikate) der Nachbedingungen fest, welche Zustände anschließend eingenommen werden können.

In den Nachbedingungen müssen allerdings nicht sämtliche Zustands-Prädikate der Klasse vorkommen. Die Frage ist, was mit denjenigen Zustands-Prädikaten geschieht, über die in den Nachbedingungen nichts ausgesagt wird: Bleiben sie unverändert oder ist ihre Belegung nach Ausführung der Operation ungewiss?

Man betrachte folgendes Beispiel:

```
/**
 * @require hasLock()
 * @require ring != null
 * @require fitsIn(ring)
 * @ensure isLocked()
 */
public void lock(KeyRing ring);
```

(de.jwam.handling.accesscontrol.LockableThingImpl)

Es gibt keine Nachbedingung, die etwas über das Zustands-Prädikat `hasLock()` aussagt, das in der ersten Vorbedingung vorkommt. Bei der Untersuchung der Funktionalität der Methode `lock` stellt man aber fest, dass es keinen Grund gibt anzunehmen, dass die Belegung von `hasLock()` sich durch einen Aufruf von `lock` ändert. Ähnliches gilt in fast allen Fällen, in denen ein Zustands-Prädikat in den Nachbedingungen nicht vorkommt. Wird ein Zustands-Prädikat von einer Operation geändert, so dokumentiert der Programmierer diese Änderung in den meisten Fällen in einer Nachbedingung.

Es gibt allerdings auch Ausnahmen. Das folgende Beispiel könnte aus der Implementierung eines Stacks stammen:

```
/**
 * @require !isEmpty()
 */
public void pop();
```

Die Methode `pop`, die ein Element vom Stack entfernt, darf nur aufgerufen werden, wenn der Stack nicht leer ist. Ob er allerdings hinterher weiterhin nicht leer ist, kann von Fall zu Fall unterschiedlich sein. Das Zustands-Prädikat `isEmpty()` kann also von `pop` geändert werden.

Das Problem besteht demnach darin, dass keine Annahmen darüber gemacht werden können, was mit derartigen Zustands-Prädikaten passiert. Würden in den Zielzuständen sämtliche möglichen Belegungen berücksichtigt, so gäbe es viel zu viele Zielzustände, die größtenteils auch gar nicht tatsächlich eingenommen werden (siehe das Beispiel mit `hasLock()` bei `lock`). Auf der anderen Seite darf nicht einfach angenommen werden, dass die nicht erwähnten Zustands-Prädikate sich nicht ändern.

In [Meyer97] wird dieses Problem nicht angesprochen, da dort die Beschreibung von Zuständen durch Zusicherungen nicht thematisiert wird.

Das Problem wird in dieser Arbeit mehrstufig gelöst. Zunächst werden zwei verschiedene Änderungssemantiken eingeführt, die beschreiben, was mit Zustands-Prädikaten geschieht, die in den Nachbedingungen einer Operation als Nach-Prädikate nicht vorkommen. Die *Nochange-Semantik* besagt, dass derartige Zustands-Prädikate ihre Belegung nicht ändern, während in der *Canchange-Semantik* die Belegung undefiniert ist, d. h. alle möglichen Belegungen werden auf Zielzustände abgebildet. Bei Konstruktoren wird in jedem Fall die Canchange-Semantik angewendet, da hier die Belegung von Zustands-Prädikaten, über die nichts ausgesagt wird, ungewiss ist.

Im Normalfall wird man mit der Nochange-Semantik arbeiten, da diese in den meisten Fällen die beabsichtigten Ergebnisse liefert. Es muss allerdings eine Möglichkeit geschaffen werden, die Canchange-Semantik für ein bestimmtes Zustands-Prädikat einzuschalten.

Einen Ausgangspunkt hierzu bildet der boolesche Ausdruck `Nochange`, der in [Meyer88, S. 115] erwähnt wird. Dieser Ausdruck, der nur in Nachbedingungen erlaubt ist, ist genau dann wahr, wenn „kein Attribut des aktuellen Objekts seit dem Aufruf den Wert geändert hat“[32]. `Nochange` ist aber nicht flexibel genug; in [Meyer97] wird es nicht mehr erwähnt und aus der Sprache Eiffel wurde es ab Version 3 entfernt [Meyer92c].

Auf unser Problem bezogen könnte ein neues Tag `@nochange` eingeführt werden, welches die Zustands-Prädikate angibt, die sich bei Ausführung der Methode nicht ändern. Da der Nochange-Fall jedoch der Standardfall ist, ist es wesentlich praktischer, genau den umgekehrten Fall explizit anzugeben. Es wird also ein neues Tag `@canchange` eingeführt, das ein Zustands-Prädikat bezeichnet, dessen Belegung nach Ausführung der Methode ungewiss ist. Die Methode `pop` aus dem obigen Beispiel würde dann noch folgendes Tag erhalten:

```
@canchange isEmpty()
```

[32] deutsche Übersetzung S. 124

Genau das Gleiche würde folgende Nachbedingung bewirken:

```
@ensure isEmpty() || !isEmpty()
```

Auch diese Nachbedingung (die in der Nochange-Semantik nicht identisch mit `@ensure true` ist) sagt aus, dass die Belegung von `isEmpty()` ungewiss ist. Die Formulierung mit `@canchange` ist jedoch eleganter und daher zu bevorzugen. In der Canchange-Semantik ist das `@canchange`-Tag nicht von Bedeutung, da dort für jedes nicht erwähnte Zustands-Prädikat die Canchange-Semantik gilt. – Um das Subtypenkonzept einzuhalten darf eine Unterklasse beim Redefinieren einer Methode kein neues `@canchange`-Tag einführen für ein Prädikat, das in der Oberklasse sichtbar ist, sonst könnte das Prädikat mehr Belegungen annehmen als in der Oberklasse; ein Klient, der die Unterklasse mit der Schnittstelle der Oberklasse anspricht, würde das nicht mitbekommen. Eine Unterklasse darf aber `@canchange`-Tags entfernen, da dies einer genaueren Angabe über die Belegungen der Prädikate entspricht und somit einer verschärften Nachbedingung gleichkommt.

Das `@canchange`-Tag ist der `modifies`-Klausel aus Larch-Spezifikationen [GHW85, LW93, LW94] ähnlich. Larch ist eine Familie von Spezifikationssprachen und beinhaltet u. a. Zusicherungen. Die `modifies`-Klausel listet Attribute auf, die von der Operation geändert werden; andere Attribute müssen unverändert bleiben. Möglich ist auch die Angabe von `modifies at most`, wobei die Attribute angegeben werden, die sich zwar ändern dürfen, aber nicht müssen. Dies kommt dem `@canchange`-Tag am nächsten, wobei sich jedoch auch Zustands-Prädikate ändern dürfen, die nicht im `@canchange`-Tag, wohl aber in einer Nachbedingung angegeben sind.

Damit das Subtypenkonzept gilt, darf ein Klient sich nicht darauf verlassen, dass die Nochange-Semantik gilt. Hat eine Methode beispielsweise die Vorbedingung `a` und die Nachbedingung `b`, so darf ein Klient nicht davon ausgehen, dass aufgrund der Nochange-Semantik nach der Methodenausführung `a` gilt. Eine Unterklasse könnte nämlich die Nachbedingung zu `b && !a` verschärfen, ohne dass der Klient dies mitbekommt. Ein Klient muss daher stets von der Canchange-Semantik ausgehen. Aus diesem Grund bezieht sich die Wahl der Änderungssemantik für eine Klasse auch nicht auf die Benutzung der Klasse, sondern lediglich auf das zugehörige Zustandsdiagramm, wo die Nochange-Semantik viele unerwünschte Transitionen entfallen lässt. Beim Programmieren hingegen darf die Nochange-Semantik nicht verwendet werden.

4.2.8 Einfluss von Invarianten

Im Folgenden wird davon ausgegangen, dass eine Klasse genau eine Invariante besitzt. Gibt es mehrere `@invariant`-Tags, so werden die verschiedenen Invarianten konjunktiv verknüpft; ist keine Invariante angegeben, so lautet diese `true`.

Invarianten enthalten ausschließlich Zustands-Prädikate, da sie auf Operationsparameter keinen Zugriff haben. Da die Invariante vor der Ausführung jeder Methode gilt, kann sie allen Methoden als Vorbedingung hinzugefügt werden[33]; Zustände, in denen die Invariante nicht gilt, können ohnehin nicht eingenommen werden. Aus diesem Grunde ist es für das Zustandsmodell gleichgültig, ob die Invariante den Vorbedingungen hinzugefügt wird oder nicht.

[33] Dies gilt lediglich für die Generierung von Zustandsdiagrammen, allerdings nicht für die Benutzung der Klasse, da eine Unterklasse zwar die Invariante verschärfen darf, nicht jedoch die Vorbedingungen.

Beim Zusammenspiel von Invarianten und Nachbedingungen ist jedoch Vorsicht geboten. Hier kommt es auf die Änderungssemantik an. In jedem Fall ist für die nach Ausführung der Operation eingenommenen Zustände zu prüfen, ob die Invariante gilt.

Bei der **Canchange-Semantik** kann die Invariante einfach den Nachbedingungen hinzugefügt werden. Zustands-Prädikate, die in den Nachbedingungen nicht vorkommen, dürfen sich dann nicht mehr beliebig ändern, sondern nur noch so, dass die Invariante erfüllt ist. Da bei Konstruktoren grundsätzlich die Canchange-Semantik gilt, kann hier die Invariante einfach als zusätzliche Nachbedingung betrachtet werden.

Bei der **Nochange-Semantik** kann die Invariante dazu führen, dass es zu einer Methode keinen gültigen Zielzustand gibt. Folgende Klasse beschreibt einen Behälter, der maximal ein Element aufnehmen kann, der also entweder leer oder voll ist[34]:

```
/**
 * @invariant isEmpty() ^ isFull()
 */
public class OneElementContainer
{
  /**
   * @ensure !isFull()
   */
  public OneElementContainer();

  /**
   * @require !isFull()
   * @ensure !isEmpty()
   */
  public void store(Object x);

  /**
   * @require !isEmpty()
   * @ensure !isFull()
   */
  public void clear();

  /**
   * @require !isEmpty()
   */
  public Object get();

  public boolean isEmpty();

  public boolean isFull();
}
```

Bei Verwendung der Canchange-Semantik sieht das zugehörige Zustandsdiagramm folgendermaßen aus[35]:

[34] Das kann z. B. eine Zwischenablage (Clipboard) sein.
[35] Die Funktionen sind im Diagramm nicht berücksichtigt.

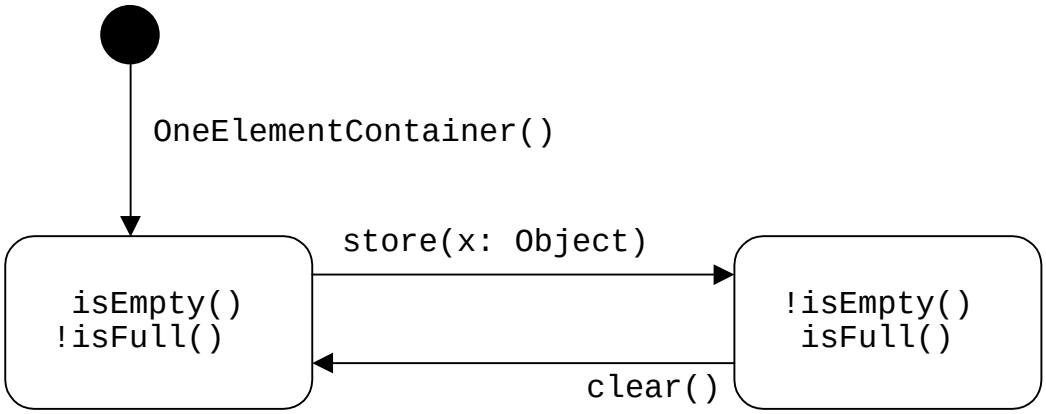


Abbildung 4.3: Zustandsdiagramm der Klasse OneElementContainer

Bei der Nochange-Semantik gäbe es die Transitionen von `store` und `clear` nicht. Die Methode `store` sichert `!isEmpty()` zu; da über `isFull()` nichts ausgesagt wird, bleibt `isFull()` weiterhin auf `false` und der Zielzustand wäre `!isEmpty()&&!isFull()`; da dieser Zustand aber die Invariante nicht erfüllt, gibt es keinen gültigen Zielzustand. Entsprechendes gilt für `clear`.

Eine Möglichkeit zur Behebung dieses Problems bestünde darin, in so einem Fall automatisch auf Canchange-Semantik umzuschalten. Diese Lösung ist allerdings unbefriedigend, da einerseits eine einheitliche Änderungssemantik für die ganze Klasse zu bevorzugen ist und andererseits eine implizite Änderung der Semantik verwirrend sein kann. Besser ist eine Vervollständigung der Nachbedingungen; in obigem Beispiel sollte z. B. `store` zusätzlich `isFull()` zusichern.

4.2.9 Nicht verwendete Elemente von Zustandsdiagrammen

In diesem Abschnitt sollen die Elemente von Zustandsdiagrammen zusammengefasst werden, die in dieser Arbeit keine weitere Verwendung finden. Zu jedem Element wird die Entscheidung begründet, es nicht zu verwenden.

Die Elemente, die bislang unerwähnt blieben, aber dennoch verwendet werden, sind Entscheidungsknoten, Unterzustände und nebenläufige Transitionen. Entscheidungsknoten können dann eingesetzt werden, wenn aus einem Zustand mehrere Transitionen, die durch dieselbe Operation ausgelöst werden, mit unterschiedlichen Wächterbedingungen herausführen. Da die Transitionen keine Aktionen auslösen, ist die Unterscheidung zwischen statischen und dynamischen Entscheidungsknoten in dieser Arbeit irrelevant. – Der Einsatz von sequenziellen und parallelen Unterzuständen (und den damit verbundenen nebenläufigen Transitionen) wird in den Kapiteln 5 und 6 behandelt.

Die nicht verwendeten Elemente sind im Einzelnen:

- **Aktionen:** Weder Zustände noch Transitionen lösen in dieser Arbeit Aktionen aus. Das liegt darin begründet, dass die Zustandsdiagramme lediglich die Klassenprotokolle beschreiben, nicht die Implementierung von Operationen. Aktionen würden jedoch die Implementierung beschreiben, da sie auszuführenden Code beinhalten.

- **Endzustände:** Die Zustandsdiagramme haben keine Endzustände, weil es in Java keine Destruktoren gibt.[36] Objekte lassen sich nicht explizit löschen. Ein Löschen der Referenz auf das Objekt oder ein tatsächliches Löschen durch den Garbage Collector sind jedoch in jedem Zustand möglich. Endzustände sind also überflüssig.

- **Gedächtniszustände:** Dass Gedächtniszustände nicht verwendet werden, hängt mit dem Vertragsmodell zusammen. Zusicherungen können nicht ohne Weiteres auf zuvor eingenommene Zustände Bezug nehmen. Dazu ist eine Modellierung dieser Zustände durch Zustands-Prädikate notwendig, die dann allerdings auf gewöhnliche Zustände abgebildet werden. Da Zusicherungen immer nur Bedingungen prüfen können, die aktuell gültig sind, können Gedächtniszustände nicht direkt durch Zusicherungen modelliert werden.

- **Synchronisationszustände:** Der Grund dafür, dass Synchronisationszustände nicht unterstützt werden, liegt ebenfalls daran, dass das Konzept der Zusicherungen nicht mächtig genug ist um Synchronisation zu beschreiben. Es ist nicht vorgesehen, dass Zusicherungen verschiedener Operationen aufeinander Bezug nehmen. Das wäre aber für eine Synchronisation notwendig.

[36] Die Methode `finalize` ist kein echter Destruktor, weil sie nur von der Laufzeitumgebung aufgerufen wird. Das geschieht allerdings zu einem undefinierten Zeitpunkt; es wird nicht einmal garantiert, dass sie überhaupt aufgerufen wird.

5 Von den Zusicherungen zum Zustandsdiagramm

In diesem Kapitel wird ausführlich erläutert, wie man von den gegebenen Zusicherungen einer Klasse zum zugehörigen Zustandsdiagramm gelangt. Das Verfahren arbeitet mit logischen Ausdrücken und wird anhand eines Beispiels nachvollzogen. Dabei wird auch untersucht, wie Unterzustände gebildet werden können. Zum Abschluss wird diskutiert, wie sich Vererbungsbeziehungen zwischen Klassen auf Zustandsdiagramme auswirken.

5.1 Die Beispielklasse

Der Algorithmus zur Bildung des Zustandsdiagramms zu einer gegebenen Klasse wird an einem hinreichend komplexen Beispiel nachvollzogen. Dabei geht es nicht um ein möglichst realistisches Beispiel; dafür sind einige Zusicherungen sicher zu komplex. Um die Feinheiten des Verfahrens zu erläutern sind jedoch derartig komplizierte Zusicherungen notwendig.

Bei der Beispielklasse handelt es sich um eine Klasse zum Versenden von Nachrichten über ein Netzwerk. Die Schnittstelle sieht wie folgt aus:

```
public class MessageService
{
  /**
   * @ensure !sent()
   */
  public MessageService();

  /**
   * @require connected()
   * @require msg != null
   * @ensure sent()
   * @ensure hasBeenSent(msg)
   */
  public void send(Object msg);

  /**
   * @require msg == null || msg.toString() != null
   * @ensure (old (msg == null) && result == false)
   *         || !old (msg == null)
   */
  public boolean hasBeenSent(Object msg);

  /**
   * @require !connected()
   * @canchange connected()
   */
  public void tryToConnect();
```

```
  /**
   * @require connected()
   * @ensure !connected()
   */
  public void disconnect();

  public boolean sent();

  public boolean connected();
}
```

Die Methode `send` dient zum Senden einer Nachricht, `tryToConnect` versucht eine Verbindung herzustellen, `disconnect` baut sie wieder ab, `hasBeenSent` prüft, ob eine gegebene Nachricht bereits gesendet wurde, `sent` prüft, ob überhaupt schon etwas gesendet wurde, und `connected` prüft, ob eine Verbindung besteht.

Der Konstruktor sichert zu, dass am Anfang noch nichts gesendet worden ist. Über `connected()` wird nichts ausgesagt; der Konstruktor könnte z. B. `tryToConnect` aufrufen.

Die Methode `send` hat als Vorbedingungen, dass eine Verbindung bestehen muss und die Nachricht nicht `null` sein darf. Die erste Nachbedingung besagt, dass nach Ausführung der Methode etwas gesendet wurde, die zweite Nachbedingung, dass `msg` gesendet wurde.

Die Funktion `hasBeenSent` prüft, ob die angegebene Nachricht bereits versendet wurde. Es kann auch `null` übergeben werden; in diesem Fall soll das Ergebnis immer `false` sein. Die Vorbedingung besagt, dass die `toString`-Methode des Parameters nicht `null` liefern darf, falls der Parameter ungleich `null` ist. Die Nachbedingung sagt aus, dass das Ergebnis `false` ist, falls der Parameter `null` ist; ist er ungleich `null`, so wird über das Ergebnis nichts zugesichert.

Die Methode `tryToConnect` versucht eine Verbindung herzustellen, wobei zuvor keine Verbindung bestehen darf. Da das Verbinden fehlschlagen kann, also die Belegung von `connected()` ungewiss ist, wird hier ein `@canchange`-Tag verwendet.

Mit `disconnect` wird die Verbindung wieder getrennt, wobei sie vorher bestehen muss und hinterher garantiert nicht mehr besteht.

5.2 Erzeugen des Zustandsdiagramms

Nun sollen die Schritte zum Erzeugen des Zustandsdiagramms aufgezeigt werden. Nach der Bestimmung und Einteilung der Prädikate werden die Zusicherungen modelliert, woraufhin die Zustände und Transitionen bestimmt werden können und somit ein flaches Zustandsdiagramm entsteht. Anschließend wird noch erläutert, wie Zustände zu komplexen Zuständen mit sequenziellen oder parallelen Unterzuständen zusammengefasst werden können.

5.2.1 Einteilung der Prädikate

Zuerst ist es notwendig, die Prädikate zu finden und einzuteilen. Hierzu werden die Teilausdrücke der Zusicherungen herausgesucht, die sich nicht unmittelbar aus booleschen Teilausdrücken zusammensetzen (vgl. Abschnitt 4.2.1). Durch Überprüfen, ob die Prädikate einen Parameter der zugehörigen Operation oder das Schlüsselwort `result` enthalten, werden die Prädikate in Parameter- und Zustands-Prädikate unterteilt. Die Zustands-Prädikate werden nun auf z_1 bis z_n verteilt (n sei die Anzahl der Zustands-Prädikate). Die Verteilung ist beliebig, muss jedoch im gesamten Verfahren einheitlich bleiben.

Während die Zustands-Prädikate für die ganze Klasse einheitlich sind, besitzt jede Operation ihre eigenen Parameter-Prädikate. Diese werden mit p_1 bis p_m bezeichnet (wobei m die Anzahl der Parameter-Prädikate der zugehörigen Operation ist). Dabei ist jedoch stets anzugeben, zu welcher Operation die Parameter-Prädikate gehören. Die Verteilung auf die einzelnen p_i ist wiederum beliebig, muss allerdings im Folgenden einheitlich bleiben.

Wenn für die Verteilungen die Reihenfolge des Auftretens der Prädikate im Quelltext genommen wird, sieht die Zuordnung für die Beispielklasse wie folgt aus:

Zustands-Prädikate:	z_1	`sent()`
	z_2	`connected()`
Parameter-Prädikate:		
`MessageService`	–	
`send`	p_1	`msg != null`
	p_2	`hasBeenSent(msg)`
`hasBeenSent`	p_1	`msg == null`
	p_2	`msg.toString() != null`
	p_3	`result == false`
`tryToConnect`	–	
`disconnect`	–	
`sent`	–	
`connected`	–	

5.2.2 Die Zustände

Gemäß Abschnitt 4.2.5 gibt es im zu konstruierenden Zustandsdiagramm $2^2 = 4$ Zustände[37], da es 2 Zustands-Prädikate gibt. Die Zustände sind die folgenden:

[37] den Anfangszustand nicht mit einberechnet

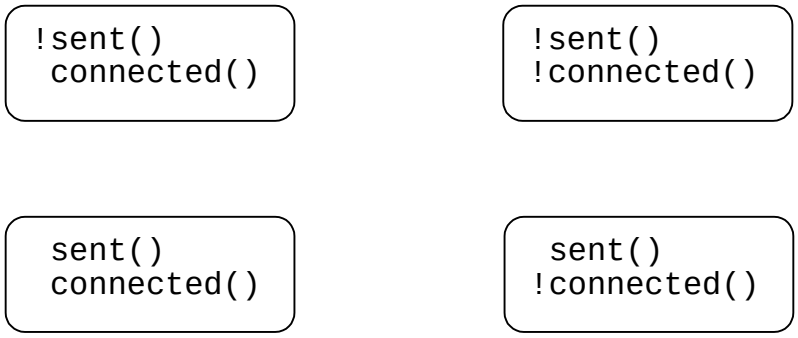


Abbildung 5.1: Zustände der Klasse MessageService

Bei diesen Betrachtungen wird von flachen Zustandsdiagrammen (d. h. solchen ohne Unterzustände) ausgegangen. Bei der Erzeugung von Zustandsdiagrammen mit Unterzuständen kann die Anzahl der Zustände abweichen. Solche Zustandsdiagramme werden in den Abschnitten 5.2.9 und 5.2.10 behandelt.

5.2.3 Modellierung der Zusicherungen

Um die Zusicherungen zu modellieren ist es notwendig, Vor- und Nach-Prädikate entsprechend zu kennzeichnen. Vor-Prädikate werden durch einen nachgestellten Strich kenntlich gemacht; ohne Strich sei das Prädikat ein Nach-Prädikat. Im Beispiel steht also z_1' für `old sent()` und z_1 für `sent()`.

Nun ist es wichtig, die jeweils relevanten Prädikate zu bestimmen. Ein Prädikat ist für eine Operation relevant, wenn eine Änderung seiner Belegung auf die Auswertung der Zusicherungen Einfluss haben kann. Dies soll im Folgenden genauer definiert werden.

Vor-Zustands-Prädikate sind genau dann relevant, wenn die betrachtete Operation kein Konstruktor ist. Bei Konstruktoren ist der Ursprungszustand immer der Anfangszustand, weshalb Vor-Zustands-Prädikate dort nicht von Bedeutung sind. Bei Methoden hingegen sind alle Vor-Zustands-Prädikate relevant. Das liegt daran, dass in den Zuständen sämtliche Zustands-Prädikate stehen, sodass für alle Belegungen der Vor-Zustands-Prädikate geprüft werden muss, ob die Zusicherungen erfüllt sind.

Bei den Nach-Zustands-Prädikaten muss zwischen Canchange- und Nochange-Semantik unterschieden werden. Bei Canchange-Semantik (also auch bei allen Konstruktoren) sind alle Nach-Zustands-Prädikate relevant, da sich laut dieser Semantik alle Zustands-Prädikate ändern können. Bei Nochange-Semantik dagegen sind nur die Nach-Zustands-Prädikate relevant, die in mindestens einer Nachbedingung als solche vorkommen oder für die es ein `@canchange`-Tag gibt. Für die übrigen Nach-Zustands-Prädikate gilt, dass ihre Belegungen die gleichen wie die der entsprechenden Vor-Zustands-Prädikate sind.

Bei den Vor- und Nach-Parameter-Prädikaten sind jeweils genau diejenigen relevant, die in den Zusicherungen vorkommen. Für Vor-Parameter-Prädikate, die nicht vorkommen, gilt, dass ihre Belegungen beliebig sind und die Prädikate somit nicht in die Wächterbedingung einbezogen werden müssen. Ebenso sind die Belegungen von Nach-Parameter-Prädikaten beliebig, wenn sie nicht

angegeben werden, sodass sie nicht im Constraint genannt zu werden brauchen. Die Mengen der relevanten Vor-/Nach-Zustands- bzw. Parameter-Prädikate seien Z', Z, P' und P.

Von dieser Stelle an ist die Unterscheidung zwischen Vor- und Nachbedingungen nicht mehr von Bedeutung. Es kommt nunmehr lediglich darauf an, ob die Bedingungen gelten oder nicht. Durch die Unterteilung der Prädikate in Vor- und Nach-Prädikate ist die korrekte Auswertung bereits gesichert. Dadurch können sämtliche Zusicherungen einer Operation durch konjunktive Verknüpfung zu einer einzigen Bedingung *b* zusammengefasst werden. Für eine konkrete Belegung der Vor- und Nach-Prädikate ist dann lediglich zu prüfen, ob *b* gilt; genau dann ist ein Zustandsübergang unter dieser Belegung möglich.

Da die Invariante sowohl vor als auch nach der Ausführung von Methoden gelten muss, wird eine entsprechende Bedingung sowohl mit Vor- als auch mit Nach-Prädikaten benötigt; diese seien i' bzw. i. Für die Beispielklasse gilt $i' = i = true$.

Die folgende Tabelle zeigt die relevanten Prädikate sowie die aus den Zusicherungen resultierende Bedingung *b* für die Operationen der Beispielklasse in der Nochange-Semantik:[38]

	Z'	Z	P'	P	b
`MessageService`	$\varnothing$	$\{ z_1, z_2 \}$	$\varnothing$	$\varnothing$	$\neg z_1$
`send`	$\{ z_1', z_2' \}$	$\{ z_1 \}$	$\{ p_1' \}$	$\{ p_2 \}$	$z_2' \wedge p_1' \wedge z_1 \wedge p_2$
`hasBeenSent`	$\{ z_1', z_2' \}$	$\varnothing$	$\{ p_1', p_2' \}$	$\{ p_3 \}$	$(p_1' \vee p_2') \wedge ((p_1' \wedge p_3) \vee \neg p_1')$
`tryToConnect`	$\{ z_1', z_2' \}$	$\{ z_2 \}$	$\varnothing$	$\varnothing$	$\neg z_2'$
`disconnect`	$\{ z_1', z_2' \}$	$\{ z_2 \}$	$\varnothing$	$\varnothing$	$z_2' \wedge \neg z_2$
`sent`	$\{ z_1', z_2' \}$	$\varnothing$	$\varnothing$	$\varnothing$	*true*
`connected`	$\{ z_1', z_2' \}$	$\varnothing$	$\varnothing$	$\varnothing$	*true*

5.2.4 Mathematische Grundlagen

Um den Algorithmus zur Bestimmung der Transitionen im nächsten Abschnitt sauber zu formulieren sind noch einige mathematische Definitionen nötig. Sie sollen an dieser Stelle eingebracht werden.

Zunächst soll der aussagenlogische Begriff der Belegung geklärt werden [Schöning00, S. 15]. Zu einer Menge von Prädikaten *A* ist eine *Belegung* eine totale Funktion $\mathcal{A}: A \rightarrow \{true, false\}$, die also allen Prädikaten einen Wahrheitswert zuweist. Üblicherweise wird der Definitionsbereich von $\mathcal{A}$ auf sämtliche Ausdrücke, die ausschließlich aus Prädikaten aus *A* aufgebaut sind, erweitert. Dadurch kann $\mathcal{A}$ auch zur Auswertung von Ausdrücken verwendet werden.

Da der Wertebereich einer Belegungsfunktion aus genau zwei Elementen besteht, gibt es zu einer Menge von Prädikaten *A* genau $2^{|A|}$ Belegungen. Wir bezeichnen die Belegungen von Z mit $\mathcal{Z}$, von Z′ mit $\mathcal{Z}'$ usw.

Weiterhin wird im nächsten Abschnitt die *Substitution von Mengenelementen* benötigt. Dazu wird folgende Schreibweise eingeführt:

[38] Der Unterschied in der Canchange-Semantik bestünde darin, dass dort für alle Operationen $Z = \{ z_1, z_2 \}$ gälte.

Definition 5.1: Zu einer Menge A bezeichne $A[\sigma(a_i', a_i)]$ die Menge, die aus A entsteht, wenn alle Elemente der Form a_i' durch a_i ersetzt werden (für alle $i \in \mathbb{N}$).

Zu einer Menge $A = \{a_1', a_2'\}$ gilt also $A[\sigma(a_i', a_i)] = \{a_1, a_2\}$.

Außerdem wird noch eine Funktion *str* eingeführt, die Prädikate in Zeichenketten umwandelt:

Definition 5.2: Die Funktion *str* sei wie folgt definiert: $str(p, b) = \begin{cases} "p", \text{falls } b = true \\ "\neg p", \text{sonst} \end{cases}$

Beispielsweise gilt $str(p_1', true) + "\wedge" + str(p_2, false) = "p_1' \wedge \neg p_2"$, wobei das Pluszeichen den Verkettungsoperator darstellt.

5.2.5 Die Transitionen

Die Bestimmung der Transitionen erfordert einigen Aufwand. Zunächst soll der Algorithmus vorgestellt werden, woraufhin eine ausführliche Erläuterung folgt:

(1) für alle $\mathcal{Z}'$
(2) wenn ein Konstruktor betrachtet wird oder $\mathcal{Z}'(i') = true$
(3) für alle $\mathcal{Z}$
(4) $\mathcal{Z}_2: Z \cup Z'[\sigma(z_i', z_i)] \rightarrow \{true, false\}$ mit $\mathcal{Z}_2(z) = \begin{cases} \mathcal{Z}(z), \text{falls } z \in Z \\ \mathcal{Z}'(z'), \text{sonst} \end{cases}$
(5) wenn $\mathcal{Z}_2(i) = true$
(6) $W := \emptyset$
(7) für alle $\mathcal{P}'$
(8) $w := str(p_1', \mathcal{P}'(p_1')) + "\wedge" + ... + "\wedge" + str(p_n', \mathcal{P}'(p_n'))$ $\quad (n = |P'|)$
(9) $c :=$ "*false*"
(10) für alle $\mathcal{P}$
(11) $\mathcal{B}: Z \cup Z' \cup P \cup P' \rightarrow \{true, false\}$ mit $\mathcal{B}(x) = \begin{cases} \mathcal{Z}'(x), \text{falls } x \in Z' \\ \mathcal{Z}(x), \text{falls } x \in Z \\ \mathcal{P}'(x), \text{falls } x \in P' \\ \mathcal{P}(x), \text{falls } x \in P \end{cases}$
(12) wenn $\mathcal{B}(b) = true$
(13) $c := c + "\vee" + str(p_1, \mathcal{P}(p_1)) + "\wedge" + ... + "\wedge" + str(p_n, \mathcal{P}(p_n))$ $\quad (n = |P|)$
(14) wenn $c <>$ "*false*"
(15) wenn es $(w_i, c_i) \in W$ gibt mit $c_i = c$
(16) $w_i := w_i + "\vee" + w$
(17) sonst
(18) $W := W \cup \{(w, c)\}$
(19) für alle $(w, c) \in W$
(20) vereinfache w und c nach den Regeln für logische Ausdrücke
(21) erzeuge Transition von $\mathcal{Z}'$ (bzw. dem Anfangszustand, falls ein Konstruktor betrachtet wird) nach $\mathcal{Z}$ mit Wächterbedingung w und Constraint c

Im Folgenden werden sämtliche Schritte detailliert erklärt:

(1) Es wird über alle Belegungen der Menge Z' iteriert. Konkret heißt dies, dass sämtliche Belegungen der Vor-Zustands-Prädikate ausprobiert werden. Es wird also für alle Zustände geprüft, ob sie als Ursprungszustände für Transitionen in Frage kommen. Da Z' bei Konstruktoren leer ist, wird dort nicht über die Vor-Zustands-Prädikate iteriert; stattdessen gilt dann lediglich $Z'(true) = true$ und $Z'(false) = false$.
(2) Die folgenden Schritte werden nur vollzogen, wenn die Invariante im aktuellen Ursprungszustand erfüllt ist, sofern der Ursprungszustand nicht der Anfangszustand ist (bei einem Konstruktor). Ist sie nicht erfüllt, so kann der betreffende Zustand gar nicht eingenommen werden und somit auch nicht Ursprungszustand einer Transition sein.
(3) Nun wird über die relevanten Nach-Zustands-Prädikate iteriert. Alle Prädikate, die in Z enthalten sind, können ihre Belegungen ändern (über sie wird iteriert), die übrigen müssen unangetastet bleiben. Dies sind die möglichen Zielzustände für Transitionen.
(4) Hier wird eine neue Belegungsfunktion eingeführt, die für alle Nach-Zustands-Prädikate definiert ist.[39] Die Belegung entspricht für Prädikate aus Z der Belegung Z, ansonsten wird aufgrund der Nochange-Semantik die entsprechende Belegung der Vor-Prädikate genommen.
(5) Die nachfolgenden Schritte werden wiederum nur durchlaufen, wenn auch im potenziellen Zielzustand die Invariante erfüllt ist.
(6) Die Menge W soll Tupel bestehend aus Wächterbedingung und Constraint enthalten, welche die Transitionen vom aktuellen Ursprungs- zum aktuellen Zielzustand beschreiben.
(7) Nun wird über alle relevanten Vor-Parameter-Prädikate iteriert.
(8) Die aktuelle Wächterbedingung w setzt sich aus der momentanen Belegung der Vor-Parameter-Prädikate zusammen.
(9) c soll das Constraint sein, das gültig ist, wenn eine Transition mit Wächterbedingung w schaltet.
(10) Schließlich wird auch über alle relevanten Nach-Parameter-Prädikate iteriert.
(11) Hier wird eine neue, quasi allumfassende Belegungsfunktion eingeführt, die für Ausdrücke gültig ist, die aus beliebigen Prädikaten bestehen.
(12) An dieser Stelle werden die Zusicherungen der betrachteten Operation ausgewertet, die zu einem einzigen Ausdruck b zusammengefasst wurden.
(13) Wenn die Zusicherungen gelten, dann wird das Constraint c disjunktiv um die aktuelle Belegung der Nach-Parameter-Prädikate erweitert.
(14) Nachdem über alle Nach-Parameter-Prädikate iteriert wurde, wird geprüft, ob c gesetzt wurde (d. h. ob die Zusicherungen irgendwann erfüllt waren).
(15) Wenn das der Fall ist, dann wird geprüft, ob es in der Menge W bereits ein Tupel aus Wächterbedingung und Constraint gibt, bei dem das Constraint mit dem aktuellen identisch ist.
(16) In diesem Fall wird die zugehörige Wächterbedingung um die aktuelle erweitert, da auch unter der aktuellen Wächterbedingung dasselbe Constraint gilt.
(17) Wenn das aktuelle Constraint noch nicht vorgekommen ist,
(18) dann wird W um ein Tupel aus aktueller Wächterbedingung und aktuellem Constraint erweitert.
(19) Nachdem über alle Vor-Parameter-Prädikate iteriert wurde, werden nun sämtliche Paare von Wächterbedingung und Constraint aus W betrachtet.
(20) Bei w und c handelt es sich um Zeichenketten, die logische Ausdrücke repräsentieren. Diese können nach den bekannten Regeln vereinfacht werden.
(21) Vom aktuellen Ursprungs- zum Zielzustand existiert nun eine Transition mit Wächterbedingung w und Constraint c.

[39] Das liegt daran, dass bei Methoden Z' alle Vor- und bei Konstruktoren Z alle Nach-Prädikate enthält.

Um den Algorithmus an der Beispielklasse nachzuvollziehen betrachten wir die Methode `hasBeenSent`, wobei nur die Selbsttransition des Zustands `sent() && connected()` untersucht werden soll. Die Belegungen der Zustands-Prädikate sind also z_1', z_2', z_1 und z_2. Mit dieser Ausgangsbasis starten wir bei Schritt (6).

Die folgende Tabelle zeigt die Werte von *b* bei den verschiedenen Belegungen der relevanten Parameter-Prädikate:

	$\neg p_1' \wedge \neg p_2'$	$p_1' \wedge \neg p_2'$	$\neg p_1' \wedge p_2'$	$p_1' \wedge p_2'$
$\neg p_3$	*false*	*false*	*true*	*false*
p_3	*false*	*true*	*true*	*true*

In Schritt (7) des Algorithmus werden nun nacheinander die Spalten dieser Tabelle untersucht. In der ersten Spalte wird noch keine Belegung gefunden, die zu *true* ausgewertet wird; erst in der zweiten Zeile der zweiten Spalte ist dies der Fall. Die Wächterbedingung *w* wird in Schritt (8) auf "$p_1' \wedge \neg p_2'$ " gesetzt, während das Constraint *c* in Schritt (13) auf "*false* $\vee$ p_3" gesetzt wird. Die Auswertung von Schritt (15) ergibt *false*, weil *W* noch leer ist; daher wird *W* := {("$p_1' \wedge \neg p_2'$ ", "*false* $\vee$ p_3")} gesetzt.

Bei Untersuchung der nächsten Tabellenspalte wird bereits in der ersten Zeile das Ergebnis *true* gefunden, woraufhin *c* := "*false* $\vee$ $\neg p_3$" gesetzt wird. Da die zweite Zeile ebenfalls *true* liefert, wird *c* erweitert zu "*false* $\vee$ $\neg p_3 \vee p_3$". Die Auswertung von (15) ergibt wiederum *false*, da "*false* $\vee$ $\neg p_3 \vee p_3$" als Constraint bisher noch nicht vorkam. In Schritt (18) wird daher *W* auf {("$p_1' \wedge \neg p_2'$ ", "*false* $\vee$ p_3"), ("$\neg p_1' \wedge p_2'$ ", "*false* $\vee$ $\neg p_3 \vee p_3$")} erweitert.

In der dritten Spalte ergibt nur die letzte Zeile *true*, weshalb zur Wächterbedingung "$p_1' \wedge p_2'$ " das Constraint "*false* $\vee$ p_3" gesetzt wird. Die Auswertung von (15) liefert nun *true*, da "*false* $\vee$ p_3" als Constraint schon vorkam. Die zu "*false* $\vee$ p_3" gehörende Wächterbedingung wird also zu "$p_1' \wedge \neg p_2' \vee p_1' \wedge p_2'$ " erweitert.

Am Ende gilt *W* = {("$p_1' \wedge \neg p_2' \vee p_1' \wedge p_2'$ ", "*false* $\vee$ p_3"), ("$\neg p_1' \wedge p_2'$ ", "*false* $\vee$ $\neg p_3 \vee p_3$")}. Der Ausdruck $p_1' \wedge \neg p_2' \vee p_1' \wedge p_2'$ lässt sich zu p_1' vereinfachen, aus *false* $\vee$ p_3 wird p_3 und aus *false* $\vee$ $\neg p_3 \vee p_3$ wird *true*. Daraus folgt, dass die Selbsttransition des betrachteten Zustands durch `hasBeenSent` folgendermaßen ausgelöst werden kann[40]:

1. `[msg == null] {result == false}`
2. `[!(msg == null) && msg.toString() != null]`

Es wird also zugesichert, dass der Rückgabewert `false` ist, wenn als Parameter `null` übergeben wird. Wird ein Parameter ungleich `null` übergeben, so muss `msg.toString() != null` gelten; über den Rückgabewert wird dann jedoch nichts zugesichert.

Es sei noch erwähnt, dass der Algorithmus zu disjunkten Wächterbedingungen führt, aber nicht unbedingt zu disjunkten Constraints (die Constraints im obigen Beispiel sind nicht zueinander disjunkt). Würde man in Schritt (7) über alle $\mathcal{P}$ iterieren und in Schritt (10) über alle $\mathcal{P}'$ (bei gleichzeitigem Vertauschen der Bildung von *w* und *c*), so wären die Constraints zueinander disjunkt und die

[40] Wächterbedingungen und Constraints, die *true* sind, können entfallen.

Wächterbedingungen nicht. Das ist aber nicht zu bevorzugen, da in manchen Fällen nicht entschieden werden kann, welche Transition schalten soll. Beim gewählten Algorithmus kann dies immer entschieden werden, da die Wächterbedingungen sich gegenseitig ausschließen.[41]

5.2.6 Unerreichbare Zustände

Zustände, die vom Anfangszustand aus nicht erreichbar sind, sollten entfernt werden. Unerreichbare Zustände lassen sich mithilfe von graphentheoretischen Verfahren [Turau96] leicht auffinden. Folgender Algorithmus entfernt Zustände, die nicht erreichbar sind:

(1) Z sei die Menge der Zielzustände von Anfangszuständen.
(2) Markiere alle Zustände aus Z.
(3) $Z_2 := \emptyset$
(4) für alle Zustände z aus Z
(5) für alle Transitionen t mit Ursprungszustand z
(6) z_2 sei der Zielzustand von t
(7) wenn z_2 noch nicht markiert ist
(8) $Z_2 := Z_2 \cup \{ z_2 \}$
(9) $Z := Z_2$
(10) wenn $Z \neq \emptyset$
(11) gehe zu (2)
(12) Entferne alle Zustände, die nicht markiert wurden, und alle Transitionen, deren Ursprungszustand nicht markiert wurde.

Im Beispiel gibt es keine unerreichbaren Zustände. – Unerreichbare Zustände hängen mit Invarianten zusammen, da die Bedingungen, die in solchen Zuständen stehen, nie wahr sind. Deren Negation ist somit eine Invariante. Um eine Invariante zu erhalten müssen also nur die Bedingungen aller *erreichbaren* Zustände disjunktiv verknüpft werden.

5.2.7 Das Zustandsdiagramm der Beispielklasse

Mit diesen Ergebnissen kann das Zustandsdiagramm der Beispielklasse bereits konstruiert werden. Es sieht folgendermaßen aus:

[41] Dies gilt nur für einen bestimmten Zielzustand. Nichtdeterminismus kann trotzdem auftreten, wenn Transitionen zu mehreren Zielzuständen aktiviert sind.

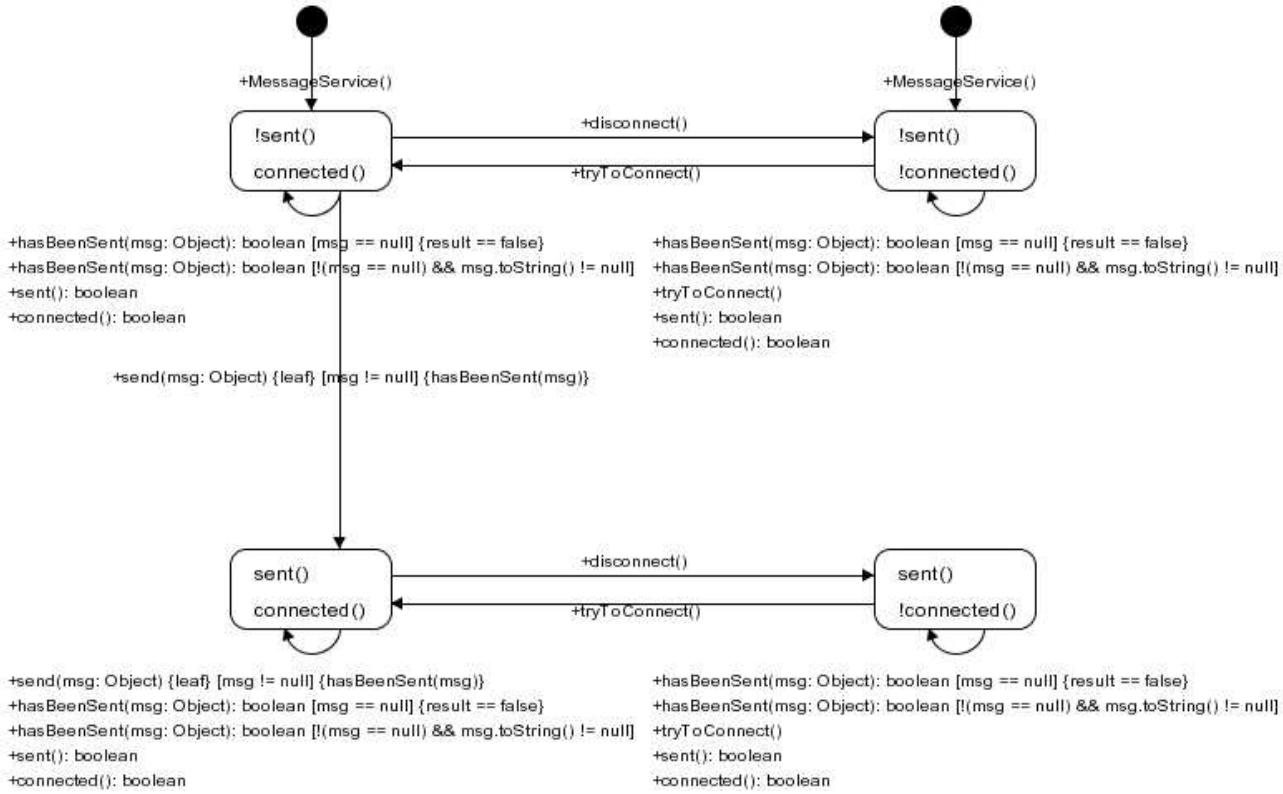


Abbildung 5.2: Zustandsdiagramm der Klasse MessageService

5.2.8 Entscheidungsknoten

Wenn aus einem Zustand mehrere Transitionen mit unterschiedlichen Wächterbedingungen, aber der gleichen Operation zu verschiedenen Zielzuständen herausführen, können auch Entscheidungsknoten gezeichnet werden. Gleiches gilt bei unterschiedlichen Constraints zu verschiedenen Wächterbedingungen. Im Zustand `sent() && connected()` könnte die Transition von `hasBeenSent` z. B. zu einem Entscheidungsknoten führen, der wie folgt aufgebaut ist:

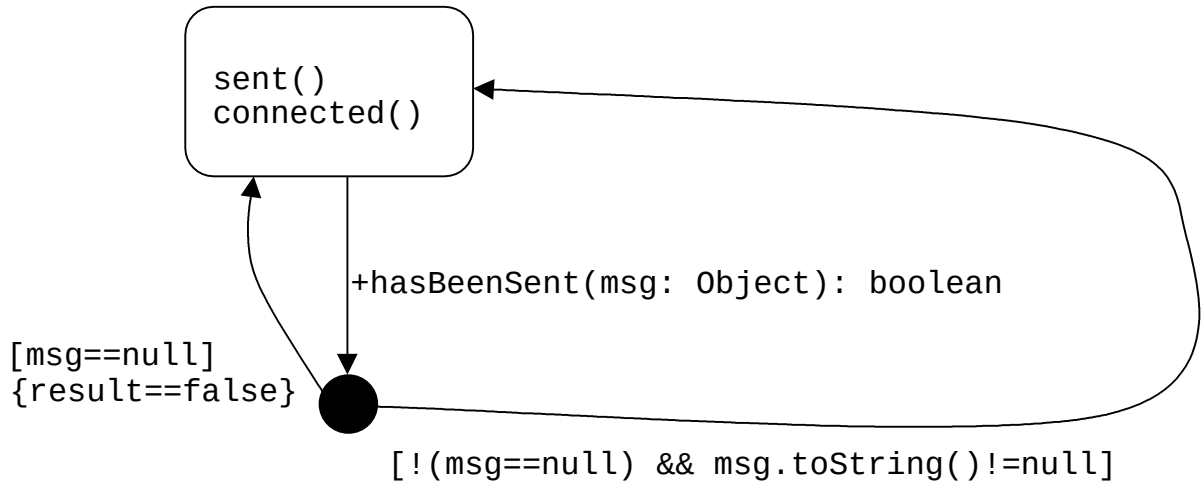


Abbildung 5.3: Verwendung von Entscheidungsknoten

Diese Form der Darstellung macht das Verhalten des Zustandsdiagramms bei einzelnen Operationen deutlicher, führt jedoch durch das Hinzufügen neuer Elemente zu erhöhter Unübersichtlichkeit bei größeren Diagrammen. Daher wird die Darstellung ohne Entscheidungsknoten hier bevorzugt.

5.2.9 Sequenzielle Unterzustände

Sequenzielle Unterzustände können dadurch gebildet werden, dass mehrere Zustände zusammengefasst werden, in denen ein oder mehrere Zustands-Prädikate dieselben Belegungen haben. Welche Zustände dabei genau zusammengefasst werden, spielt im Prinzip keine Rolle, da sequenzielle Unterzustände immer das direkte Ein- und Austreten von Transitionen unterstützen. Auf diese Weise ist die Bildung eines sequenziellen zusammengesetzten Zustands zunächst nur ein grafischer Einschluss von mehreren Zuständen, wobei die Belegung eines oder mehrerer Zustands-Prädikate im zusammengesetzten Zustand angegeben und von den Unterzuständen quasi „geerbt“ wird.

Im Beispiel könnten die beiden Zustände, in denen `sent()` gilt, zusammengefasst werden. Der Ausschnitt sähe dann so aus (ohne Transitionsbeschriftungen):

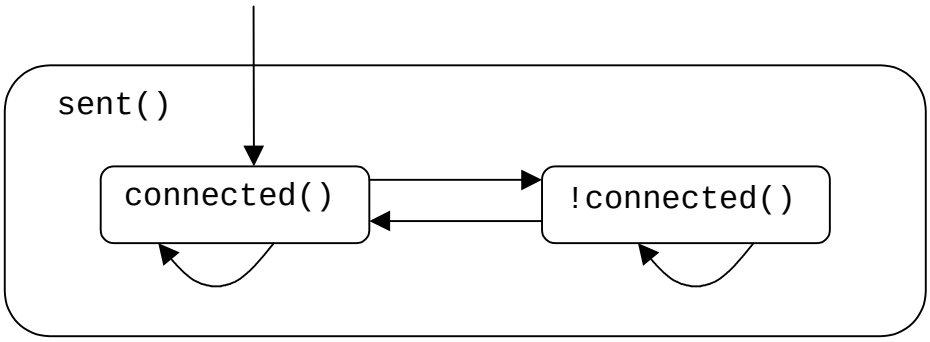


Abbildung 5.4: Verwendung von sequenziellen Unterzuständen

Da nur eine einzige Transition in den zusammengesetzten Zustand hineinführt, ist auch folgender Aufbau denkbar:

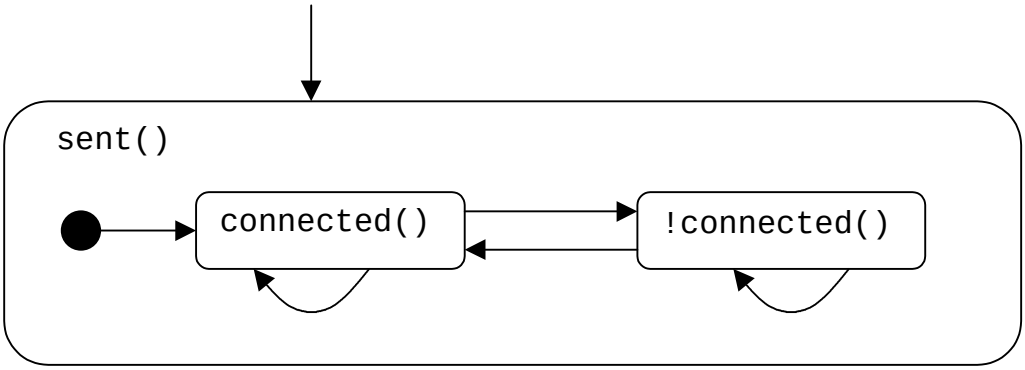


Abbildung 5.5: Alternative Verwendung von sequenziellen Unterzuständen

Ein Anfangszustand innerhalb eines komplexen Zustands kann im Prinzip zu einem beliebigen Unterzustand führen; von außen kommende Transitionen, die zu einem anderen Unterzustand führen sollen, können diesen Unterzustand direkt als Zielzustand benutzen. Der besseren Übersicht wegen sollte der Anfangszustand zu dem Unterzustand führen, der Zielzustand der meisten Transitionen von außerhalb ist.

Eine Transition, die den komplexen Zustand (nicht einen seiner Unterzustände) als Ursprungszustand hat, kann nur dann erzeugt werden, wenn alle Unterzustände Transitionen mit denselben Ereignissen zum Zielzustand dieser Transition haben.

Es gibt keine Regel, die besagt, welche Zustände zusammengefasst werden sollen. Hier sind die unterschiedlichsten Lösungen möglich. Im Beispiel könnten auch die Zustände, in denen `connected()` gilt, zusammengefasst werden, was ein ganz anderes Bild ergeben hätte. Die Wahl hängt hier in besonderem Maße von den Präferenzen des Betrachters ab. Eine Automatisierung soll daher nicht unterstützt werden.

5.2.10 Parallele Unterzustände

Parallele Unterzustände repräsentieren in diesem Modell nicht Nebenläufigkeit, wie man annehmen könnte, sondern Unabhängigkeit von Zustands-Prädikaten. Da hierfür besondere Zustandsdiagramme nötig sind, folgt hier ein eigenes Beispiel; es handelt sich um einen Ausschnitt des Zustandsdiagramms zur JWAM-Klasse `de.jwam.technology.action.JwamActionImpl`:

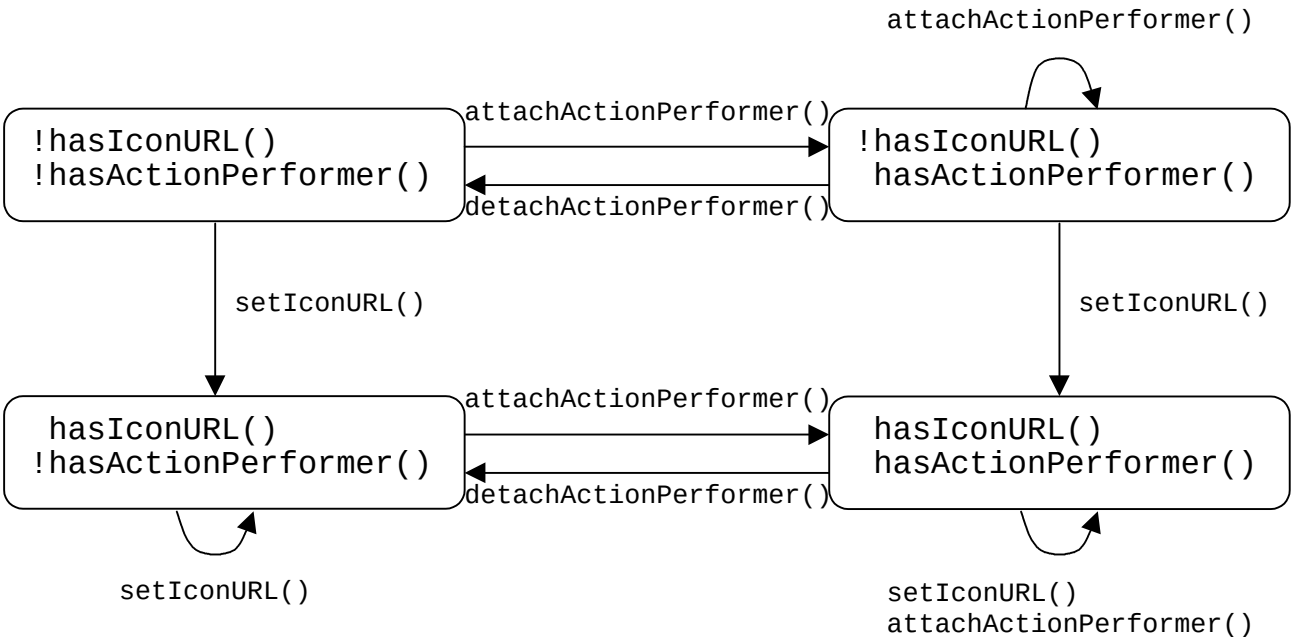


Abbildung 5.6: Beispieldiagramm zur Bildung von parallelen Unterzuständen

Diese vier Zustände können zu einem komplexen Zustand mit zwei parallelen Unterzuständen zusammengefasst werden. Das liegt daran, dass die beiden Zustands-Prädikate voneinander unabhängig sind. Dies ist der Fall, weil für alle Transitionen, die innerhalb dieser vier Zustände liegen (in diesem Beispiel alle), folgende Bedingungen gelten:

1. Alle Transitionen zwischen verschiedenen Zuständen, in denen `hasIconURL()` gilt (die beiden unteren), haben exakte Gegenstücke bei den Zuständen, in denen `!hasIconURL()` gilt.
2. Alle Transitionen zwischen verschiedenen Zuständen, in denen `hasActionPerformer()` gilt (die beiden rechten), haben exakte Gegenstücke bei den Zuständen, in denen `!hasActionPerformer()` gilt.
3. Es gibt keine Transitionen, die beide Zustands-Prädikate ändern (keine diagonal verlaufenden Transitionen).
4. Für jede Methode gilt: Verknüpft man die Bedingungen der Zustände, an denen eine Selbsttransition mit dieser Methode steht, disjunktiv und vereinfacht den Ausdruck, so enthält kein Minterm mehr als ein Prädikat, das nicht in allen Zuständen, die zusammengefasst werden sollen, dieselbe Belegung hat.[42]

Immer wenn diese Bedingungen gelten, können parallele Unterzustände gebildet werden. Transitionen, die in den zusammengesetzten Zustand hinein- oder aus ihm herausführen, können immer durch nebenläufige Transitionen realisiert werden. Wie bei sequenziellen Unterzuständen können aber auch hier Anfangszustände gebildet werden, wodurch der komplexe Zustand als Zielzustand einer Transition verwendet werden kann. Als Ursprungszustand einer Transition kann er hingegen nur benutzt werden, wenn im flachen Zustandsdiagramm aus allen Zuständen, die zusammengefasst werden, eine Transition mit derselben Operation zum gleichen Zustand führt.

Die vierte Bedingung muss gelten, weil auch bei Selbsttransitionen nicht mehrere Prädikate voneinander abhängen dürfen. Im Beispiel gibt es für `setIconURL` Selbsttransitionen bei `hasIconURL() && !hasActionPerformer()` und bei `hasIconURL() && hasActionPerformer()`. Die disjunktive Verknüpfung und anschließende Vereinfachung ergibt `hasIconURL()`, womit nur ein Prädikat übrig bleibt. Die entsprechende Selbsttransition wird also dem Unterzustand `hasIconURL()` beigefügt.

Folgendes Zustandsdiagramm mit parallelen Unterzuständen kann somit aus dem obigen gebildet werden:

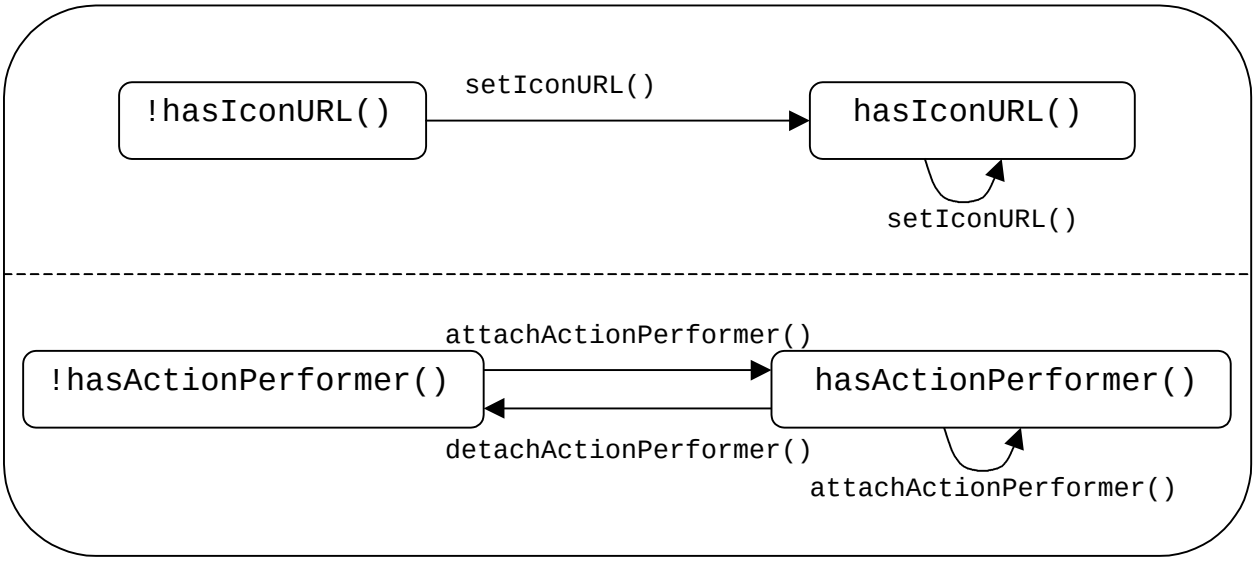


Abbildung 5.7: Verwendung von parallelen Unterzuständen

[42] In diesem Beispiel gibt es kein Prädikat, das in allen Zuständen dieselbe Belegung hat.

Dieses Beispiel verdeutlicht sehr anschaulich die Unabhängigkeit der beiden Zustands-Prädikate. Es macht auch auf den ersten Blick ersichtlich, dass die Methode `setIconURL` zu `hasIconURL()` gehört und `attachActionPerformer` sowie `detachActionPerformer` zu `hasActionPerformer()`, was auch von der Namensgebung her intuitiv ist.

Man beachte, dass bei Methodenaufrufen in parallelen Zuständen in diesem Modell immer nur eine Transition zur Zeit schaltet. Dies gilt für den Fall, dass verschiedene parallele Zustände Transitionsereignisse enthalten, die durch dieselbe Methode ausgelöst werden. Im obigen Beispiel trifft das jedoch nicht zu.

In vielen Fällen werden sich die entstandenen Zustandsdiagramme allerdings nicht in solche mit parallelen Unterzuständen umformen lassen, da die hierzu nötigen Voraussetzungen zu speziell sind. Eine automatische Überprüfung, welche Zustände sich zu parallelen Unterzuständen zusammenfassen lassen, wäre zwar prinzipiell möglich, allerdings recht aufwändig. Außerdem kann es vorkommen, dass es mehrere Möglichkeiten der Unterzustandsbildung gibt, wobei es von den Intentionen des Benutzers abhängt, welche vorzuziehen ist. Daher soll auch hier von einer automatisierten Unterstützung abgesehen werden.

5.3 Vererbung

In diesem Abschnitt soll untersucht werden, wie sich Zustandsdiagramme in Vererbungshierarchien verhalten. Die Benutzung des Vertragsmodells in Unterklassen ist klar vorgeschrieben (s. Abschnitt 4.1.4); es herrscht aber in der Fachliteratur kein allgemeiner Konsens darüber, wie sich Zustandsdiagramme von Unterklassen zu denen ihrer Oberklassen verhalten sollen. Während diese Frage vielfach offen gelassen wird, wird z. B. in [RBPEL91] und [HG96] empfohlen, dass eine Unterklasse das Zustandsdiagramm ihrer Oberklasse erben soll, wobei neue Elemente hinzugefügt werden dürfen.

In [HG96] wird vorgeschlagen, dass Zustandsdiagramme von Unterklassen folgende Erweiterungen vornehmen dürfen:

- Ein Zustand kann zu einem sequenziellen oder parallelen Unterzustand verfeinert werden.
- Zu einem sequenziellen oder parallelen Zustand dürfen weitere Unterzustände hinzugefügt werden.
- Neue Transitionen können eingeführt werden.
- Der Zielzustand einer Transition darf sich ändern.
- Die Aktionen einer Transition können geändert werden.
- Die Wächterbedingung einer Transition darf geändert werden.

Es wird darauf hingewiesen, dass eine Transitionen implizit entfernt werden kann, indem ihre Wächterbedingung auf *false* gesetzt wird.

Der erklärte Schwerpunkt von [HG96] liegt allerdings auf automatischer Codegenerierung; obige Regeln basieren nicht auf einem ausgearbeiteten theoretischen Fundament. Andere Ansätze, die besser theoretisch ausgearbeitet sind, gehen von deterministischen Zustandsdiagrammen aus (d. h. der Zielzustand einer Operation ist bei gleichen Anfangsbedingungen eindeutig). In [KS94] wird z. B. unter diesen Voraussetzungen gearbeitet, wodurch zwar einfache Vererbungsregeln entstehen[43], aber aufgrund der Annahme des deterministischen Falles die Zustandsdiagramme zu unflexi-

[43] Zustände und Transitionen werden an das Zustandsdiagramm der Unterklasse vererbt.

bel sind. In [MD93] werden die Zustandsdiagramme sogar aufgrund von Vor- und Nachbedingungen gebildet; allerdings ist der Ansatz formal wenig ausgearbeitet und enthält außerdem Fehlschlüsse.[44]

Im folgenden Abschnitt soll ein bereits bestehender flexibler theoretischer Ansatz zur Vererbung von Verhalten vorgestellt werden, woraufhin untersucht wird, wie sich Vererbungshierarchien im Modell dieser Arbeit auf die Zustandsdiagramme auswirken.

5.3.1 Vererbung von Verhalten

Eine detaillierte Untersuchung zum Thema Vererbung von Verhalten findet sich in [BA01].[45] Objektlebenszyklen werden hier mithilfe von Prozessalgebra sowie Petrinetzen modelliert. Es werden vier unterschiedliche Vererbungsbeziehungen eingeführt, wobei für Methoden, die in der Unterklasse neu eingeführt werden, zwei Prinzipien betrachtet werden: *blockieren* und *verbergen*. Blockieren heißt, dass die neuen Methoden nicht aufgerufen werden dürfen; verbergen bedeutet, dass sie zwar aufgerufen werden können, aber nur die Effekte von Methoden der Oberklasse berücksichtigt werden (dies gilt für den Fall, dass man anhand des Zustandsverhaltens der Oberklasse mit einem Exemplar der Unterklasse arbeitet). Die vier Vererbungsbeziehungen lauten wie folgt:

- *Protokoll-Vererbung* liegt vor, wenn das externe Verhalten der Unterklasse sich nicht von dem der Oberklasse unterscheiden lässt, sofern nur Methoden aufgerufen werden, die auch in der Oberklasse vorhanden sind.

- *Projektions-Vererbung* liegt vor, wenn das externe Verhalten der Unterklasse sich nicht von dem der Oberklasse unterscheiden lässt, sofern beliebige Methoden aufgerufen werden, aber nur die Effekte von Methoden betrachtet werden, die auch in der Oberklasse vorhanden sind.

- *Protokoll-/Projektions-Vererbung* liegt vor, wenn sowohl Protokoll- als auch Projektions-Vererbung vorliegt.

- *Lebenszyklus-Vererbung* liegt vor, wenn das externe Verhalten der Unterklasse sich nicht von dem der Oberklasse unterscheiden lässt, sofern einige der Methoden, die nur in der Unterklasse vorhanden sind, nicht aufgerufen werden und zu einigen anderen dieser Methoden nur die Effekte von Methoden betrachtet werden, die auch in der Oberklasse vorhanden sind.

Die dritte Vererbungsbeziehung impliziert die ersten beiden, während jede der ersten drei die vierte impliziert.

Auf diesen Beziehungen aufbauend werden vier Vererbungsregeln eingeführt, mit deren Hilfe man einen Lebenszyklus erweitern kann, sodass eine dieser Vererbungsbeziehungen gilt. Jede dieser Regeln entspricht einem Design-Konstrukt, und zwar Iteration, sequenzieller und paralleler Komposition sowie alternativer Verzweigung. Die vier Vererbungsregeln werden hier kurz vorgestellt, wobei jeweils beispielhafte Ausschnitte aus Zustandsdiagrammen angegeben werden:

- Die erste Regel bewahrt Protokoll-/Projektions-Vererbung und entspricht einer Iteration. Von einem Zustand kann eine neue Transition zu einer neuen Teil-Zustandsmaschine führen, die abschließend wieder eine Transition zum ursprünglichen Zustand enthält. – In den folgenden Bei-

[44] Es wird behauptet, dass in Zustandsdiagrammen von Unterklassen keine Transitionen entfernt werden dürften, weil dies die Abschwächung von Nachbedingungen mit sich zöge. Das ist falsch, wie noch gezeigt werden wird.
[45] Die dort gewonnenen Ergebnisse werden u. a. in [AB01] und [AHT02] weiterverfolgt.

spielen sind die gestrichelten Elemente die im Zustandsdiagramm der Unterklasse neu hinzugekommenen. Auf die Beschriftung der Zustände und Transitionen wird verzichtet, soweit nicht zur Zuordnung erforderlich. Protokoll-Vererbung liegt hier vor, weil sich durch das Blockieren der neuen (gestrichelten) Transitionen keine Änderung am Zustandsdiagramm ergibt; Projektions-Vererbung liegt vor, weil durch das Schalten der neuen Transitionen kein anderer Zustand des Oberklassendiagramms erreicht werden kann:

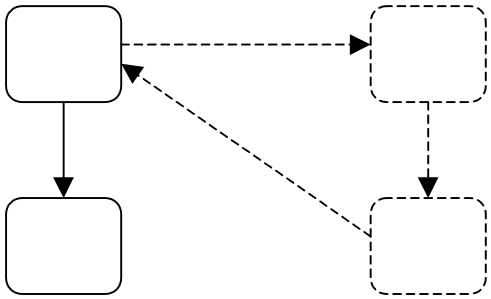


Abbildung 5.8: Erste Vererbungsregel

- Die zweite Regel entspricht einer sequenziellen Komposition und hält Projektions-Vererbung ein. Sie besagt, dass zwischen zwei Zuständen, die durch eine Transition verbunden sind, weitere Zustände sequenziell eingefügt werden können. Diese Regel hält nicht die Protokoll-Vererbung ein, weil neue Methoden, die in der Sequenz auftreten, blockiert sind und der Zielzustand somit nicht erreicht werden kann. – Hier wieder ein Beispiel:

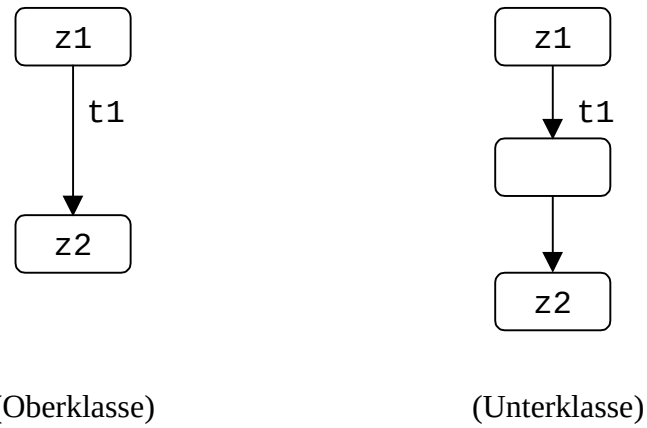


Abbildung 5.9: Zweite Vererbungsregel

- Mit der dritten Regel wird Protokoll-Vererbung eingehalten. Sie entspricht einer alternativen Verzweigung und besagt, dass von einem Zustand aus eine Transition zu einer neuen Teil-Zustandsmaschine führen kann. Im Gegensatz zur ersten Regel muss die letzte Transition nicht wieder zum ursprünglichen Zustand zurückführen, sondern kann einen beliebigen Zustand als Zielzustand haben, weshalb diesmal Projektions-Vererbung nicht eingehalten wird:

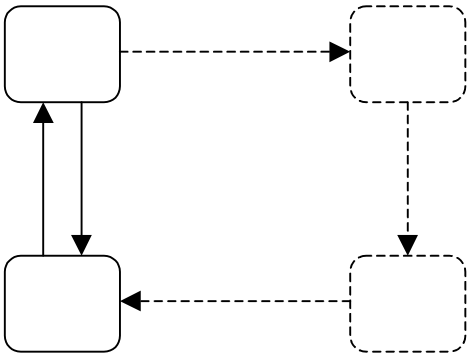
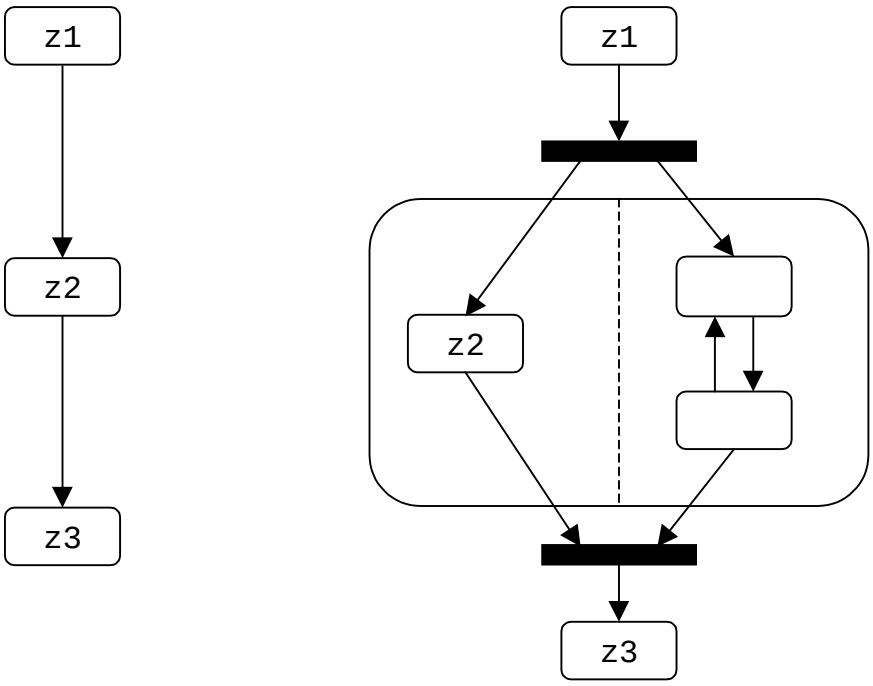


Abbildung 5.10: Dritte Vererbungsregel

- Die vierte Regel hält wiederum Projektions-Vererbung ein und steht für eine parallele Komposition. Zu einer Teil-Zustandsmaschine des ursprünglichen Zustandsdiagramms kann ein paralleler Zustand hinzugefügt werden:



(Oberklasse) (Unterklasse)

Abbildung 5.11: Vierte Vererbungsregel

Unter Anwendung dieser vier Vererbungsregeln können die Objektlebenszyklen von Unterklassen so entwickelt werden, dass sie gültige Erweiterungen der Objektlebenszyklen der Oberklassen bilden.

5.3.2 Zustandsdiagramme von Unterklassen

Es soll nun untersucht werden, wie Zustandsdiagramme von Unterklassen aufgebaut sind, wenn das Verfahren aus diesem Kapitel angewendet wird. Dabei wird davon ausgegangen, dass das Vertragsmodell in der Unterklasse korrekt benutzt wird, d. h. Vorbedingungen werden nicht verschärft, Nachbedingungen und Invarianten werden nicht abgeschwächt. In Abschnitt 5.3.3 werden die entstandenen Zustandsdiagramme im Hinblick auf die im vorigen Abschnitt vorgestellten Vererbungsregeln untersucht.

Es werden jeweils die Transitionen zu einer Methode betrachtet, die sowohl in der Oberklasse als auch in der Unterklasse vorhanden ist. Grundsätzlich gibt es bei Vererbung drei zu untersuchende Möglichkeiten für die Zusicherungen: Abschwächen einer Vorbedingung, Verschärfen einer Nachbedingung und Verschärfen der Invariante.[46] Diese Möglichkeiten werden im Folgenden getrennt voneinander betrachtet.

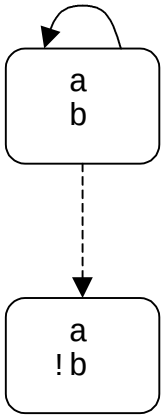
Es ist zu berücksichtigen, dass in der Unterklasse neue Zustands-Prädikate definiert werden können und sich somit die Menge der möglichen Zustände ändern kann. In diesem Fall muss auch das Zustandsdiagramm der Oberklasse um die neuen Zustands-Prädikate erweitert werden, damit die nachfolgenden Ergebnisse gültig bleiben. Dies erreicht man, indem für jedes neue Zustands-Prädikat[47] und jede Transition Folgendes gemacht wird: Die Transition verlaufe von Zustand z_1 zu Zustand z_2.[48] Das neue Zustands-Prädikat sei p. Im neuen Zustandsdiagramm verlaufen nun Transitionen von z_1 `&&` p nach z_2 `&&` p und von z_1 `&&!`p nach z_2 `&&!`p. In der Canchange-Semantik verlaufen zusätzlich Transitionen von z_1 `&&` p nach z_2 `&&!`p und von z_1 `&&!`p nach z_2 `&&` p.

Da Konstruktoren grundsätzlich nicht vererbt werden, entfallen Transitionen, die vom Anfangszustand ausgehen, im Zustandsdiagramm der Unterklasse generell. Die Unterklasse hat jedoch eigene Konstruktoren, sodass neue Transitionen vom Anfangszustand hinzukommen. Diese Transitionen brauchen mit denen der Oberklasse in keinerlei Beziehung zu stehen, da für Zusicherungen bei Konstruktoren keine Regeln in Bezug auf Oberklassenkonstruktoren gelten, auch wenn ein Konstruktor (explizit oder implizit) grundsätzlich einen Oberklassenkonstruktor aufrufen muss[49]. Dadurch kann sich die Menge der erreichbaren Zustände für die Unterklasse ändern; es können also Transitionen entfallen oder hinzukommen.

Im Folgenden wird die Änderung von Zusicherungen in der Unterklasse untersucht. Zunächst wird der Fall betrachtet, dass eine Vorbedingung abgeschwächt wird. Hier muss wiederum unterschieden werden, ob die Vorbedingung aus Zustands- oder Parameter-Prädikaten oder beiden besteht. Bei einer reinen Zustands-Vorbedingung sieht es folgendermaßen aus[50]:

[46] Sind die Zusicherungen in Ober- und Unterklasse identisch, was ebenfalls erlaubt ist, so sind die Transitionen gleich.
[47] Bei mehreren neuen Zustands-Prädikaten ist jeweils das Zustandsdiagramm, das Ergebnis eines Zwischenschritts ist, als Ausgangsbasis für den nächsten Schritt zu nehmen.
[48] Hierbei stehen z_1 und z_2 für die Bedingungen, die in diesen Zuständen gelten.
[49] bis auf den Konstruktor der Klasse `java.lang.Object`, der keinen anderen Konstruktor aufruft
[50] In allen folgenden Beispielen befinden sich links Zusicherungen und Zustandsdiagramm der Oberklasse, rechts der Unterklasse. Die durchgezogenen Transitionen sind diejenigen, die in der Nochange-Semantik erzeugt werden; in der Canchange-Semantik kommen zusätzlich die gestrichelten Transitionen hinzu.

```
require a && b
ensure a
```




```
require b
ensure a
```

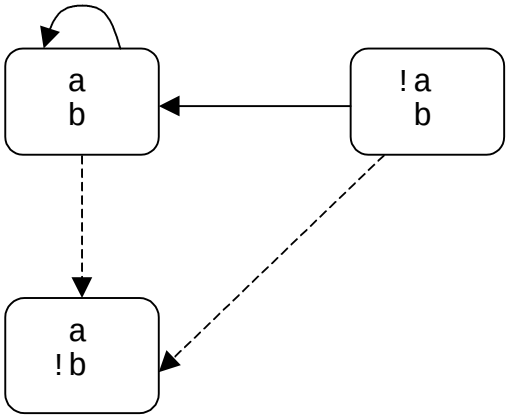
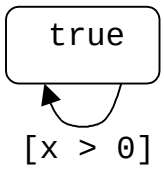


Abbildung 5.12: Abschwächen der Vorbedingung bei Zustands-Prädikaten

Hier kommen durch die Abschwächung der Vorbedingung neue Transitionen hinzu. Das kommt daher, weil in der Unterklasse mehr Ursprungszustände erlaubt sind. Die Ursprungszustände der Oberklasse erhalten keine weiteren Transitionen mehr; es kommen aber neue Ursprungszustände hinzu, von denen neue Transitionen ausgehen.

Die Abschwächung einer rein aus Parameter-Prädikaten bestehenden Vorbedingung ist unkompliziert:

```
require x > 0
```




```
require x > 0 || y == 0
```

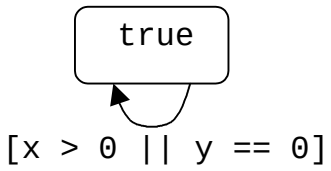


Abbildung 5.13: Abschwächen der Vorbedingung bei Parameter-Prädikaten

Hier wird lediglich die Wächterbedingung abgeschwächt; die Transitionen bleiben ansonsten unverändert.

Der interessanteste Fall ist eine gemischte Vorbedingung:

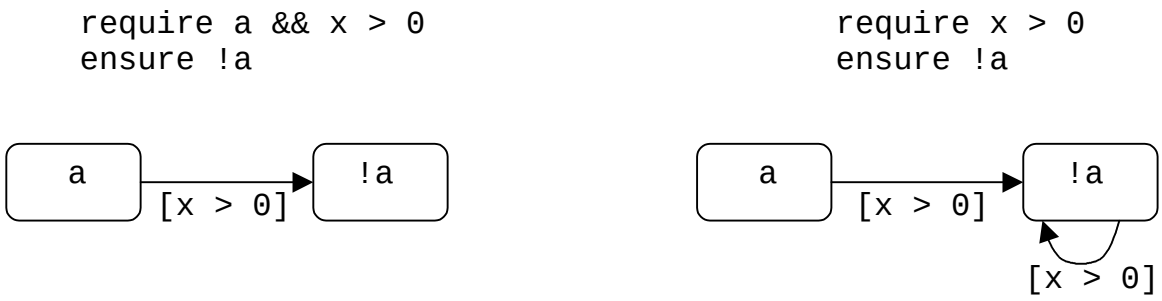


Abbildung 5.14: Abschwächen der Vorbedingung bei gemischten Prädikaten, 1. Fall

Eine andere Möglichkeit für die Unterklasse ist, das Zustands-Prädikat beizubehalten:

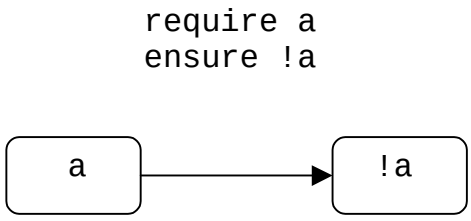


Abbildung 5.15: Abschwächen der Vorbedingung bei gemischten Prädikaten, 2. Fall

Auch in diesem Fall kommen neue Transitionen ausschließlich bei neuen Ursprungszuständen vor. Die alten Ursprungszustände erhalten keine neuen Transitionen, wobei bei den bisherigen – wie im letzten Beispiel – die Wächterbedingung abgeschwächt werden kann.

Nun soll das Verschärfen von Nachbedingungen betrachtet werden. Zunächst ein Beispiel, das ausschließlich aus Zustands-Prädikaten besteht, wobei für die Unterklasse zwei Möglichkeiten angeboten werden:

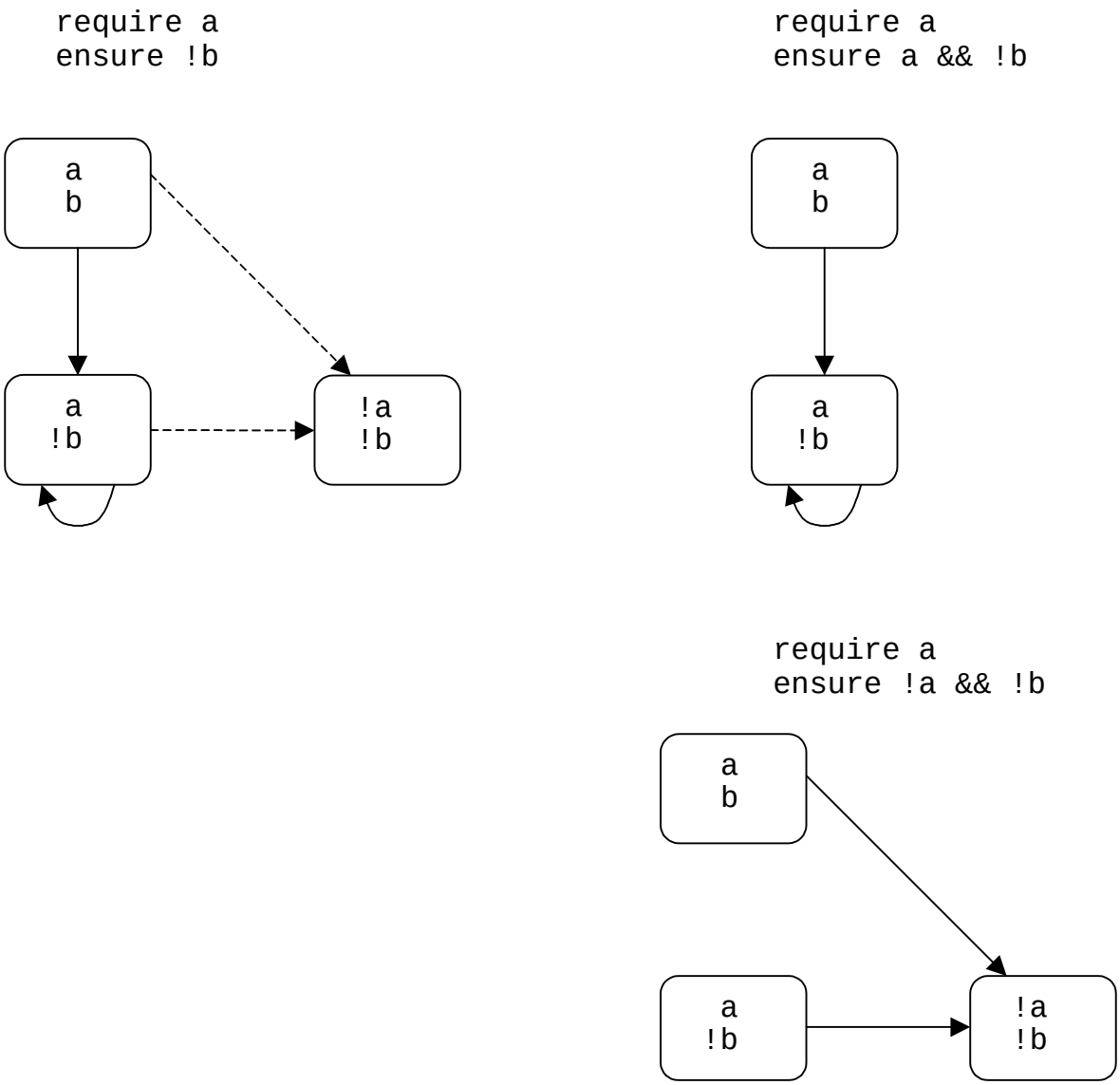


Abbildung 5.16: Verschärfen der Nachbedingung bei Zustands-Prädikaten

Bei der ersten Unterklasse gibt es in der Nochange-Semantik keinen Unterschied in den Zustandsdiagrammen. Das liegt daran, dass in der Nachbedingung der Oberklasse das Zustands-Prädikat `a` nicht vorkommt und daher seine Belegung erhalten bleibt. Bei der zweiten Unterklasse ist der Zielzustand in der Nochange-Semantik jedoch ein anderer, weil hier `!a` zugesichert wird.

Es wird deutlich, dass durch die Verschärfung einer Nachbedingung Transitionen entfallen können, und zwar sowohl in der Nochange- als auch in der Canchange-Semantik. Die entfallenen Transitionen sind diejenigen, die bei der neuen Nachbedingung nicht mehr möglich sind. Neue Transitionen kommen in der Canchange-Semantik nicht hinzu, da es durch die Verschärfung einer Nachbedingung nicht mehr Zielzustände als vorher geben kann; in der Nochange-Semantik können allerdings neue Transitionen hinzukommen, wie im Beispiel deutlich wurde. Die Ursprungszustände dieser neuen Transitionen sind dabei immer Ursprungszustände von bereits existierenden Transitionen, weil die Vorbedingung erhalten bleibt.

Bei aus Parameter-Prädikaten bestehenden Nachbedingungen ist der Fall klar: Hier werden lediglich die Constraints verschärft. Bei gemischten Nachbedingungen sieht es folgendermaßen aus (hier werden zwei mögliche Oberklassen angeboten):

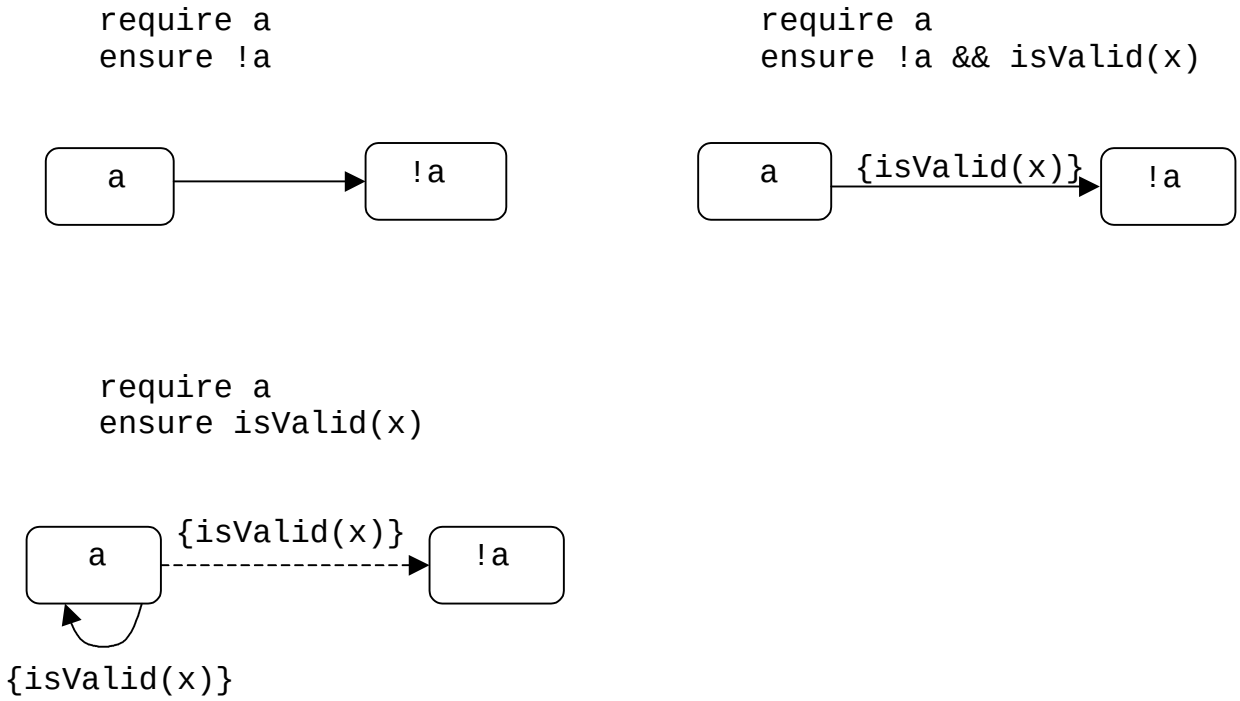
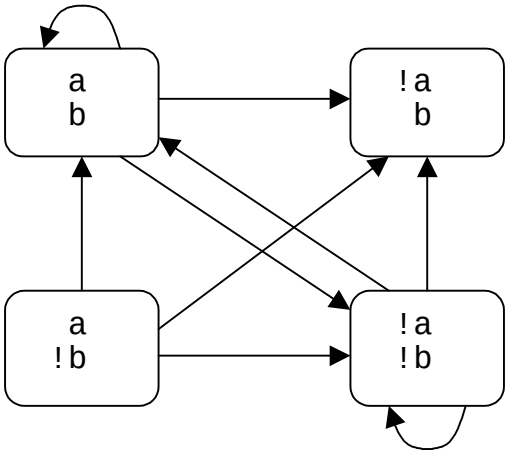


Abbildung 5.17: Verschärfen der Nachbedingung bei gemischten Prädikaten

Man sieht, dass auch hier die bereits erwähnten Fälle eintreten können: Transitionen können entfallen oder (in der Nochange-Semantik) hinzukommen und Constraints können verschärft werden.

Eine Besonderheit bei Nachbedingungen stellen die Vor-Prädikate dar, die durch das Schlüsselwort `old` gekennzeichnet sind. Dass auch bei Verwendung solcher Nachbedingungen die obigen Fälle eintreten können, wird an den folgenden Beispielen deutlich, die aufgrund der für `old`-Ausdrücke geltenden Regeln (s. Abschnitt 4.2.2) etwas komplizierter aufgebaut sein müssen; die Zustandsdiagramme lassen sich jedoch sehr einfach vergleichen:

```
require a || !b
ensure b || !a
```




```
require a || !b
ensure old a && b || old !b && !a
```

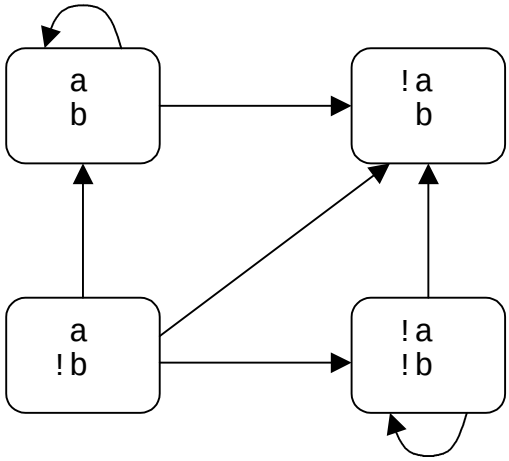
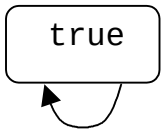


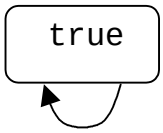
Abbildung 5.18: Verschärfen der Nachbedingung bei Vor-Zustands-Prädikaten

```
require f1(x) || f2(x)
ensure f3(x) || f4(x)
```




```
[f1(x)||f2(x)] {f3(x)||f4(x)}
```

```
require f1(x) || f2(x)
ensure old f1(x) && f3(x) ||
       old f2(x) && f4(x)
```




```
[f1(x) && !f2(x)] {f3(x)}
[!f1(x) && f2(x)] {f4(x)}
[f1(x) && f2(x)]  {f3(x)||f4(x)}
```

Abbildung 5.19: Verschärfen der Nachbedingung bei Vor-Parameter-Prädikaten

```
require f(x) || !a
ensure a || b
```

```
require f(x) || !a
ensure old f(x) && a || old !a && b
```

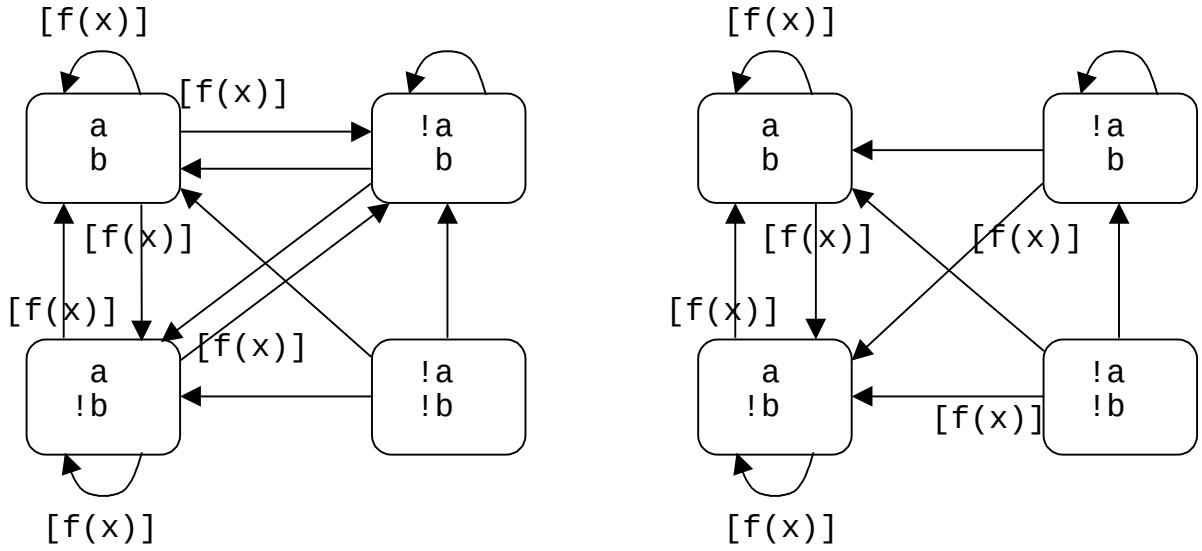


Abbildung 5.20: Verschärfen der Nachbedingung bei gemischten Vor-Prädikaten

Aus dem letzten Beispiel ist ersichtlich, dass Transitionen nicht nur entfallen können, sondern auch deren Wächterbedingung verschärft werden kann: Die Transitionen von den Zuständen `!a&&b` bzw. `!a&&!b` zu `a&&!b` haben in der Oberklasse keine explizite Wächterbedingung (und somit die Wächterbedingung `true`); in der Unterklasse hingegen lautet die Bedingung `f(x)`. Tatsächlich kann das Entfallen einer Transition als Spezialfall der Verschärfung der Wächterbedingung angesehen werden, indem die Wächterbedingung auf `false` gesetzt wird.

Die letzte Änderungsmöglichkeit besteht darin, die Invariante zu verschärfen. Da in Invarianten ausschließlich Zustands-Prädikate vorkommen und somit Wächterbedingungen und Constraints unberührt bleiben, genügt hier ein Beispiel:

```
require a
ensure b
```

```
invariant !a || !b
require a
ensure b
```

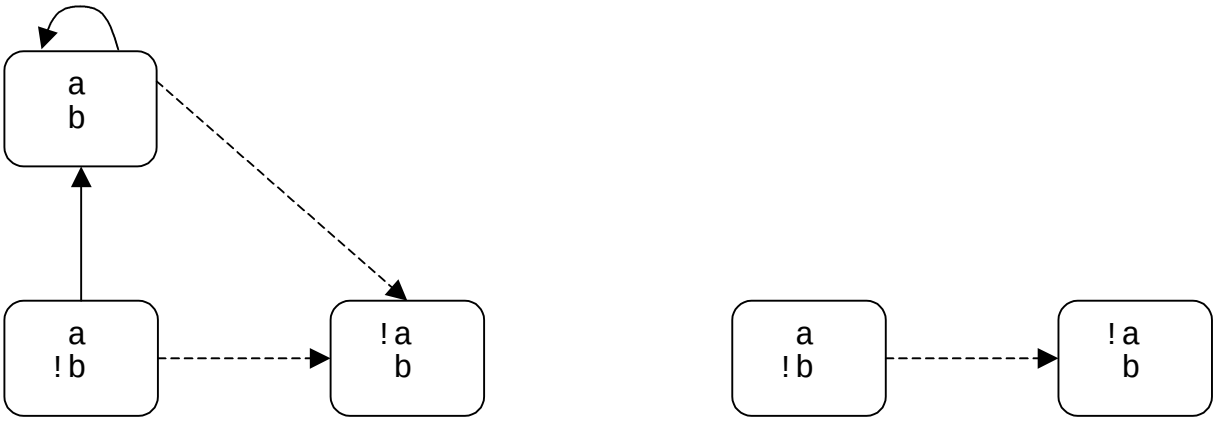


Abbildung 5.21: Verschärfen der Invariante

Der Zustand `a&&b` entfällt in der Unterklasse, da er die Invariante nicht erfüllt. Dadurch verschwinden auch die Transitionen dieses Zustands. In der Nochange-Semantik existiert kein gültiger Zielzustand mehr; somit kann die Methode ihre Nachbedingung in dieser Semantik nicht erfüllen. Würde die Invariante `!b` lauten, so wäre die Nachbedingung auch in der Canchange-Semantik nicht erfüllbar, da sie im Widerspruch zur Invariante stände.

Neue Transitionen können durch eine verschärfte Invariante nicht hinzukommen, weil das Verschärfen der Invariante einer Einschränkung der möglichen Zielzustände entspricht, keiner Ausweitung; daher können die möglichen Zielzustände höchstens weniger werden, aber nicht mehr.

Die folgende Tabelle fasst noch einmal die Möglichkeiten zusammen, die bei Änderungen der Zusicherungen in einer Unterklasse bestehen:

	Transition entfällt	Transition entsteht	Wächterbedingung abgeschwächt	Wächterbedingung verschärft	Constraint verschärft
Vorbedingung abgeschwächt		X	X		
Nachbedingung verschärft	X	X[51]		X	X
Invariante verschärft	X				

5.3.3 Ergebnisse in Bezug auf Vererbung von Verhalten

Die Ergebnisse des letzten Abschnitts sollen nun mit den Regeln über Vererbung von Verhalten aus Abschnitt 5.3.1 in Verbindung gebracht werden. Da Wächterbedingungen und Constraints in den Vererbungsregeln nicht vorkommen, gibt es dort allerdings keine Entsprechung.

Es ist festzustellen, dass keine der vier eingeführten Vererbungsbeziehungen durch das in diesem Kapitel entwickelte Verfahren eingehalten wird. Projektions-Vererbung würde bedeuten, dass zwischen den Aufrufen von Methoden der Oberklasse beliebige Methoden der Unterklasse aufgerufen werden dürften, ohne dass sich an den Effekten etwas ändert. Neue Methoden der Unterklasse können jedoch beliebige Zielzustände haben, sodass ein anschließender Aufruf einer Methode der Oberklasse unter Umständen nicht mehr möglich ist. Aus diesem Grund wird Projektions-Vererbung nicht eingehalten.

Protokoll-Vererbung wird auch nicht eingehalten, denn selbst wenn in der Unterklasse nur Methoden aufgerufen werden, die auch in der Oberklasse vorhanden sind, kann sich das Protokoll ändern. Beispielsweise kann durch eine abgeschwächte Vorbedingung der Aufruf einer Methode in der Unterklasse möglich sein, während er an selber Stelle in der Oberklasse untersagt ist.

Weil sowohl Protokoll- als auch Projektions-Vererbung nicht eingehalten wird, scheidet auch Protokoll-/Projektions-Vererbung aus, die eine Kombination aus beiden darstellt.

Dass sogar Lebenszyklus-Vererbung als allgemeinste dieser Vererbungs-Beziehungen nicht eingehalten wird, verdeutlicht folgendes Beispiel (links die Oberklasse, rechts die Unterklasse):

[51] nur in der Nochange-Semantik

```
/**
 * @require a
 * @ensure !a
 */
public void m1();
```

```
/**
 * @require true
 * @ensure !a
 */
public void m1();

/**
 * @require !a
 * @ensure !a
 */
public void m2();
```

Abbildung 5.22: Beispiel für Nichteinhaltung von Lebenszyklus-Vererbung

In der Unterklasse ist ausgehend von Zustand `a` die Aufrufreihenfolge `m1`, `m2`, `m1` möglich. Lässt man `m2`, das nur in der Unterklasse vorkommt, unberücksichtigt, so ergibt sich die Reihenfolge `m1`, `m1`, die jedoch in der Oberklasse nicht möglich ist. Das Verbergen von `m2` führt also nicht zu einer Verhaltenskonformität der Unterklasse bezüglich der Oberklasse, ebenso wenig das Blockieren, denn die Aufrufreihenfolge `m1`, `m1` ist in der Unterklasse erlaubt, in der Oberklasse jedoch nicht.

Dieses Beispiel zeigt, dass Lebenszyklus-Vererbung nicht eingehalten wird und damit auch keine der anderen Vererbungsbeziehungen, die allesamt Lebenszyklus-Vererbung implizieren.

An dieser Stelle wirft sich die Frage auf, welche Regeln für den Aufbau eines Zustandsdiagramms gelten müssen, damit es das Zustandsdiagramm einer Unterklasse zu einer gegebenen Oberklasse sein kann. Die Antwort ist, dass die Kombination folgender Bedingungen bereits hinreichend ist[52]:

1. Die Zustände enthalten mindestens die Zustands-Prädikate der Oberklasse.
2. Transitionen, die durch einen Konstruktor der Oberklasse ausgelöst werden, entfallen.
3. Wenn eine in der Oberklasse vorhandene Transition entfällt, so existiert noch mindestens eine durch dieselbe Methode ausgelöste Transition mit demselben Ursprungszustand, wobei die entfallene Wächterbedingung deren Wächterbedingung impliziert.
4. Transitionen, die im Zustandsdiagramm der Oberklasse nicht vorkommen, aber durch eine Methode ausgelöst werden, die auch in der Oberklasse vorhanden ist, haben Zielzustände und Constraints, die der Nachbedingung dieser Methode in der Oberklasse genügen.
5. Ursprungs- und Zielzustände neuer Transitionen genügen der Invariante der Oberklasse.
6. Wächterbedingungen und Constraints können verschärft werden, wobei bei Wächterbedingungen darauf zu achten ist, dass noch mindestens eine durch dieselbe Methode ausgelöste Transition vom selben Ursprungszustand ausgeht, deren Wächterbedingung von der ursprünglichen Wächterbedingung impliziert wird.

[52] Bei den Bedingungen 2 bis 6 sind jeweils die Zustandsdiagramme zu nehmen, die die gesamten Zustands-Prädikate aus Ober- und Unterklasse enthalten.

Die erste Bedingung ist unmittelbar einsichtig. Die zweite muss deshalb gelten, weil Konstruktoren nicht vererbt werden.

Die nächsten Bedingungen scheinen auf den ersten Blick nicht restriktiv genug zu sein. Es ist aber tatsächlich möglich, beliebige Transitionen zu entfernen, sofern nicht alle von einem Zustand ausgehenden Transitionen einer Methode entfernt werden.

Das Entfernen einer Transition von z_1 nach z_2 mit Wächterbedingung w[53] ist möglich, indem die entsprechende Methode folgende Nachbedingung erhält (die alte Nachbedingung sei n):

```
@ensure old ( z1 && w) && !z2 && n || old !( z1 && w) && n
```

Diese Nachbedingung sorgt dafür, dass nach der Methodenausführung z_2 nicht gilt, wenn vorher z_1 && w gegolten hat, wobei n jedoch weiterhin in jedem Fall gilt. Wenn allerdings `!`z_2 && n unerfüllbar ist, so kann der erste Minterm dieser Nachbedingung entfallen (weil er stets `false` ist); dann ist die Nachbedingung nicht mehr korrekt in Bezug auf Vor-Prädikate (s. Abschnitt 4.2.2). Der Teilausdruck `!`z_2 && n ist genau dann unerfüllbar, wenn keine Transition, die die entfallene Wächterbedingung erfüllt, einen Zielzustand außerhalb von z_2 hat; deshalb gilt die erwähnte Einschränkung. – Es ist zu beachten, dass durch die Einführung der obigen Nachbedingung die Nochange-Semantik für einige Prädikate verloren gehen kann, da in z_2 alle Zustands-Prädikate erwähnt werden. Dadurch können neue Transitionen entstehen, die jedoch alle einzeln mithilfe von Nachbedingungen des obigen Schemas entfernt werden können.

Das Hinzufügen einer durch eine in der Oberklasse vorhandene Methode ausgelösten Transition von z_1 nach z_2 mit Wächterbedingung w und Constraint c kann durch Setzen folgender Vor- und Nachbedingung geschehen (v sei die alte Vor-, n die alte Nachbedingung):

```
@require v || (z1 && w)
@ensure old (z1 && w)&& z2 && c && n || old !(z1 && w) && n
```

z_2 und c müssen hierbei die alte Nachbedingung n erfüllen, weil sonst der Teilausdruck z_2 && c && n nicht erfüllbar ist. Daher ist eine neue Transition, die von einer in der Oberklasse vorhandenen Methode ausgelöst wird, nur zu einem Zielzustand möglich, der die alte Nachbedingung erfüllt. c kann nicht abgeschwächt werden, weil es konjunktiv mit n verknüpft ist und das stärkere Constraint der Oberklasse somit bestehen bleibt. – Auch durch diese Nachbedingung geht die Nochange-Semantik verloren, weil in z_2 alle Zustands-Prädikate vorkommen. Die neuen Transitionen müssen gegebenenfalls wieder entfernt werden. – Die Invariante muss ohnehin eingehalten werden.

Das Verschärfen einer Wächterbedingung w oder eines Constraints c einer Transition von z_1 nach z_2 ist möglich, indem die Nachbedingung n durch folgende Nachbedingung ersetzt wird (w oder c sei dabei gegenüber der Oberklasse verschärft):

```
@ensure old z1 && n && (z2 && old w && c || !z2 ) || old !z1 && n
```

Diese Nachbedingung gewährleistet, dass bei gleichzeitiger Einhaltung der alten Nachbedingung n die Transition von z_1 nach z_2 mit w und c arbeitet, während die anderen Transitionen unberührt bleiben. Beim Verschärfen einer Wächterbedingung ist darauf zu achten, dass es noch mindestens eine Transition geben muss, die die alte Wächterbedingung erfüllt; ansonsten kann keine Transition

[53] Die Einbeziehung der Wächterbedingung ist notwendig, weil es mehrere Transitionen mit unterschiedlichen Wächterbedingungen und verschiedenen Constraints geben kann.

schalten, obwohl die Vorbedingung der Methode erfüllt ist.[54] – Zum Abschwächen einer Wächterbedingung oder eines Constraints ist dieser Ausdruck allerdings nicht geeignet, da aufgrund der konjunktiven Verknüpfung mit *n* die in *n* enthaltene stärkere Bedingung bestehen bleibt.

Dass das Abschwächen eines Constraints nicht möglich ist, ist einsichtig, da hierfür eine Nachbedingung abgeschwächt werden müsste. Erstaunlich ist hingegen, dass auch das Abschwächen einer Wächterbedingung bei einer beliebigen Transition nicht möglich ist. Der Grund hierfür ist, dass die Wächterbedingung auch Teil der Nachbedingung sein kann, wie folgendes Beispiel zeigt:

```
@require param != null
@ensure old (param != null)
```

Während es unproblematisch ist, die Vorbedingung zu `true` abzuschwächen[55], ist dies bei der Nachbedingung nicht möglich, weil Nachbedingungen nicht abgeschwächt werden dürfen. Um die Zusicherungen konsistent zu halten muss daher die Vorbedingung erhalten bleiben; ein Abschwächen der Wächterbedingung ist nicht möglich.

Diese Ergebnisse zeigen, dass Zustandsdiagramme von Unterklassen sich recht deutlich von denen ihrer Oberklassen unterscheiden können; die erwähnten Einschränkungen gestatten viel Spielraum. So ist sowohl das Hinzufügen als auch das Entfernen von Transitionen möglich (unter gewissen Rahmenbedingungen); ebenso können Wächterbedingungen und Constraints bestehender Transitionen verschärft werden. Die Möglichkeiten sind also viel weitreichender als in der Literatur im Allgemeinen festgelegt (u. a. in Abschnitt 5.3.1). Darin ist vermutlich auch der Grund zu suchen, warum kein allgemeiner Konsens darüber herrscht, wie Zustandsdiagramme von Unterklassen auszusehen haben: Unterklassen haben in Bezug auf Verhalten so viel Freiräume, dass sich die meisten Richtlinien als zu restriktiv erweisen.

Als Ergebnis für die Softwareentwicklungspraxis ergibt sich, dass Vererbungsbeziehungen nicht allein anhand von Zustandsdiagrammen spezifiziert werden sollten, da die Regeln zur Erhaltung der Verhaltenskonformität für den täglichen Gebrauch zu komplex sind. Die Regeln, die beim Vertragsmodell in Bezug auf Vererbung gelten, sind bedeutend einfacher und sollten daher beim Design der Klassen angewandt werden. Die Zustandsdiagramme der Ober- und Unterklassen sollten dann allerdings auf jeden Fall automatisch generiert und anschließend verglichen werden, da sich die Diagramme erheblich unterscheiden können, was möglicherweise nicht beabsichtigt ist; beim Überschreiben einer Methode will man im Allgemeinen nicht deren Verhalten völlig ändern. Als wichtiges Ergebnis dieses Abschnitts ist festzuhalten, dass beim Spezifizieren des Zustandsverhaltens von Unterklassen große Vorsicht geboten ist.

5.3.4 Auswirkungen auf Polymorphie

Die obigen Ergebnisse können zu der Vermutung führen, dass ein stark geändertes Zustandsdiagramm einer Unterklasse eine polymorphe Benutzung erschwert, wenn nicht gar unmöglich macht. Ein Klient, der die Unterklasse mit der Schnittstelle der Oberklasse benutzt, ruft die Operationen gemäß dem Zustandsdiagramm der Oberklasse auf. Da dieses sich sehr vom Zustandsdiagramm der Unterklasse unterscheiden kann, stellt sich die Frage, ob die Zustandsdiagramme hinsichtlich einer polymorphen Benutzung kompatibel sind.

[54] Die Nachbedingung wäre dann nicht korrekt in Bezug auf Vor-Prädikate.
[55] allerdings nicht in Kombination mit dieser Nachbedingung, da diese dann nicht mehr korrekt in Bezug auf Vor-Prädikate wäre

Zur Verdeutlichung wird an dieser Stelle ein Beispiel gebracht. Es handelt sich um eine Klasse, die eine sortierbare Liste implementiert[56]:

```
/**
 * @invariant !isEmpty() || isSorted()
 */
public class List
{
  /**
   * @ensure !isEmpty()
   */
  public void add(Object o);

  /**
   * @require !isEmpty()
   * @ensure isSorted()
   */
  public void sort();

  public boolean isEmpty();

  public boolean isSorted();
}
```

Die Invariante erklärt sich dadurch, dass eine leere Liste immer sortiert ist.

Das Zustandsdiagramm dieser Klasse sieht folgendermaßen aus, wobei die gestrichelten Transitionen bei Verwendung der Canchange-Semantik hinzukommen[57]:

[56] Natürlich fehlen noch Methoden zum Entfernen bzw. Auslesen der Elemente; hier werden nur die für das Beispiel relevanten Methoden angegeben.

[57] Die Anfangszustände wurden der Übersichtlichkeit wegen weggelassen; da es keinen explizit deklarierten Konstruktor gibt, existieren Transitionen vom Anfangszustand aus zu allen Zuständen.

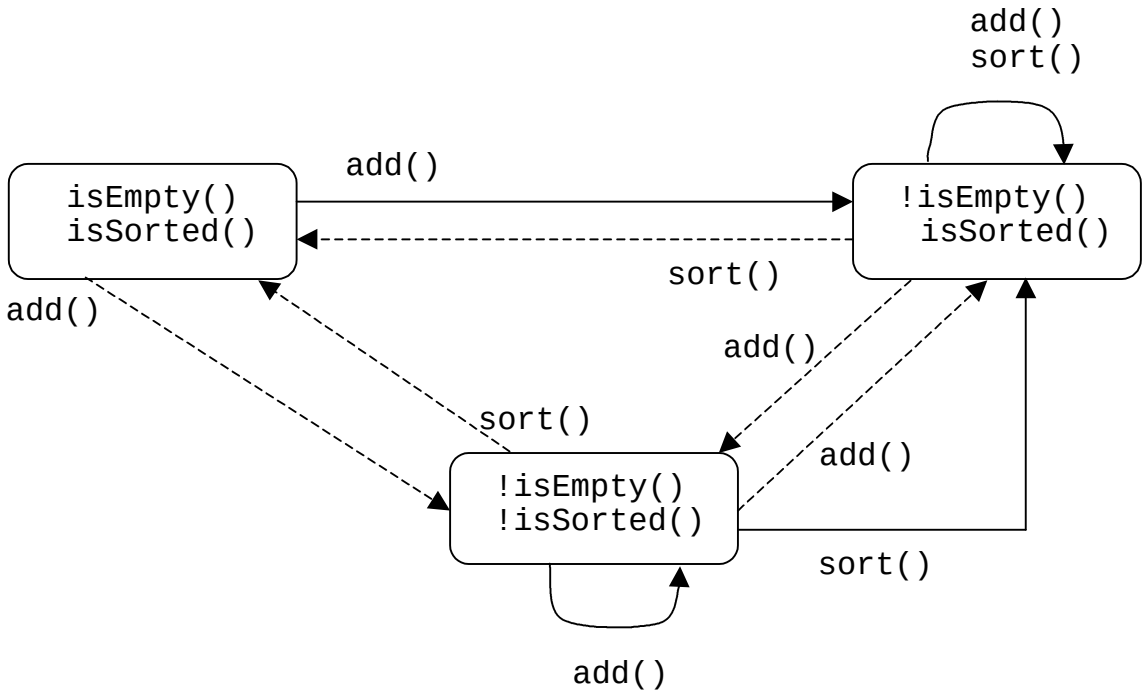


Abbildung 5.23: Zustandsdiagramm der Klasse List

Für die Methode `sort` scheint die Nochange-Semantik Sinn zu machen, denn es gibt keinen Grund anzunehmen, dass durch einen Aufruf von `sort` das Prädikat `isEmpty()` geändert wird. Bei `add` hingegen kann man sich nicht sicher sein, ob `isSorted()` nicht geändert wird. Aber selbst wenn der Programmierer dieser Klasse die Nochange-Semantik eingehalten hat, so kann ein anderer Programmierer folgende Unterklasse schreiben:

```
/**
 * @invariant !isEmpty() || isSorted()
 */
public class ImprovedList extends List
{
  /**
   * @ensure !isEmpty()
   * @ensure isSorted()
   */
  public void add(Object o);

  /**
   * @ensure isSorted()
   */
  public void sort();
}
```

Die Methode `add` sichert nun zu, dass nach einem Aufruf `isSorted()` gilt, während `sort` nun auch auf eine leere Liste angewandt werden kann.

Die Unterklasse besitzt folgendes Zustandsdiagramm:

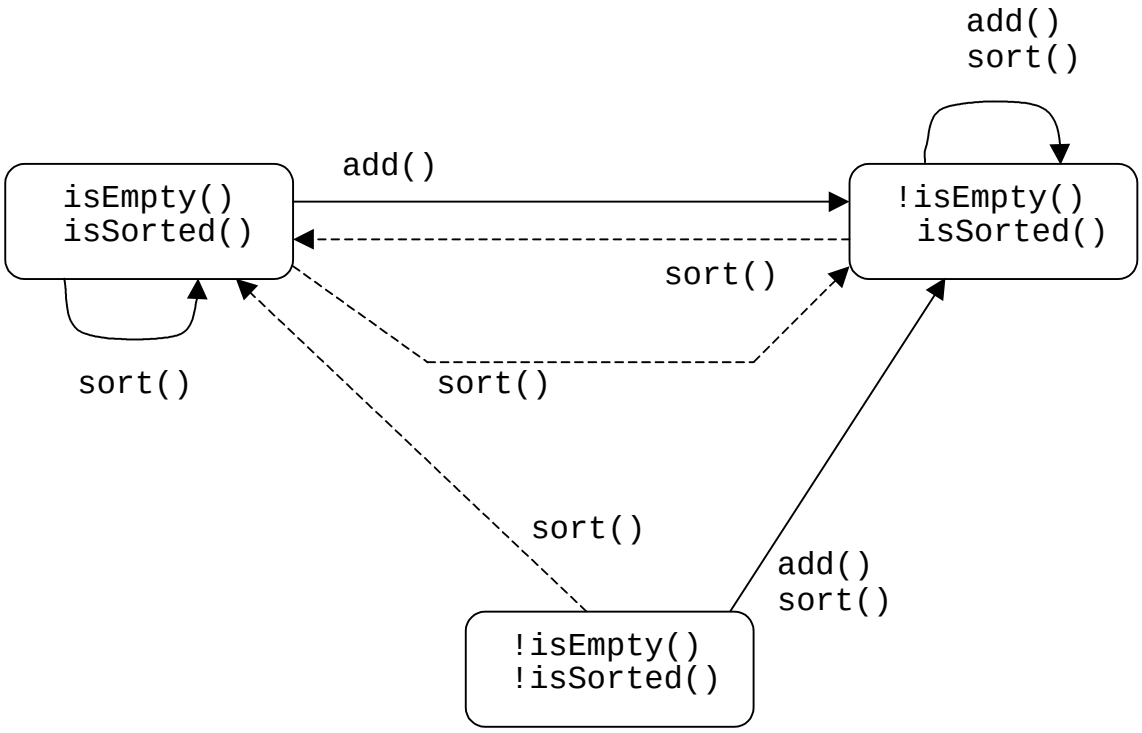


Abbildung 5.24: Zustandsdiagramm der Klasse ImprovedList

Was passiert nun bei einer polymorphen Benutzung der Unterklasse mit der Schnittstelle der Oberklasse? Problematisch für die Polymorphie sind die im Zustandsdiagramm der Unterklasse entfallenen sowie neu hinzugekommenen Transitionen. Die Transitionen der Methode `sort` aus dem Diagramm der Oberklasse sind im Diagramm der Unterklasse alle noch vorhanden, und zwar in beiden Änderungssemantiken. Die abgeschwächte Vorbedingung hat nur dazu geführt, dass mehr Transitionen entstanden sind. In diesem Fall ist das unproblematisch, da der Ursprungszustand dieser Transitionen neu ist und somit die Methode `sort` laut Protokoll der Oberklasse in diesem Zustand gar nicht aufgerufen werden kann. Bei `add` hingegen sind in beiden Änderungssemantiken Transitionen entfallen, beispielsweise die Selbsttransition des unteren Zustands. Die Auswirkungen einer polymorphen Benutzung sind folgende: Angenommen ein Exemplar der Klasse `ImprovedList` befindet sich im Zustand `!isEmpty() && !isSorted()`; der Klient sieht nur das Zustandsdiagramm von `List`. Nun wird die Methode `add` aufgerufen. In der Nochange-Semantik geht der Klient davon aus, dass anschließend noch derselbe Zustand aktiv ist, in Wahrheit jedoch lautet der aktuelle Zustand `!isEmpty() && isSorted()`, wie aus dem Zustandsdiagramm von `ImprovedList` abzulesen ist. In der Canchange-Semantik hingegen weiß der Klient, dass der Zustand nach Ausführung von `add` entweder `!isEmpty() && !isSorted()` oder `!isEmpty() && isSorted()` ist. Hier verhält sich das Zustandsdiagramm der Unterklasse konform zu dem der Oberklasse, da spezifiziert wird, welcher der beiden Zustände tatsächlich eingenommen wird.

Der Klient einer Klasse darf also nicht von der Nochange-Semantik ausgehen; diese Überlegung wurde bereits am Ende von Abschnitt 4.2.7 thematisiert. Geht er allerdings von der Canchange-Semantik aus, so verhält sich die Klasse auch bei polymorpher Benutzung konform zum Zustandsdiagramm. Wenn Transitionen entfallen, so sind dies immer Transitionen, die einen Nichtdeterminismus betreffen, wo also der Zielzustand nicht eindeutig ist. In Unterklassen kann die Menge der möglichen Zielzustände eingeschränkt werden, aber es bleibt immer mindestens eine Transition übrig, die auch im Zustandsdiagramm der Oberklasse vorhanden ist. Dies wird durch die dritte der

in Abschnitt 5.3.3 eingeführten Bedingungen sichergestellt. Bei Verwenden der Canchange-Semantik ergeben sich also keine Probleme im Hinblick auf Polymorphie.

Als wesentliches Ergebnis ist somit festzuhalten, dass ein Klient eine Klasse unter dem Zustandsdiagramm in der Canchange-Semantik benutzen muss, damit es bei Polymorphie keine Zustandsübergänge gibt, die im Diagramm nicht vorkommen. Es genügt ebenfalls, wenn sich der Klient an das Vertragsmodell hält, d. h. die Vorbedingungen einhält und sich auf die Gültigkeit der Nachbedingungen und Invarianten verlässt. Als Pragmatik für die Softwareentwicklungspraxis ergibt sich Folgendes:

- Benutze beim Aufrufen einer Operation das Vertragsmodell, also stelle die Einhaltung der Vorbedingungen sicher und verlasse dich nach Ausführung der Operation auf die Nachbedingungen und Invarianten.
- Betrachte das Zustandsdiagramm der verwendeten Klasse in der Canchange-Semantik. Dort lässt sich ablesen, in welchen Zuständen sich ein Exemplar der Klasse (oder einer Unterklasse!) nach Ausführung der Operation befinden kann. Diese Zustände geben die Bedingungen wieder, die nun gelten können.

Der erste Punkt ist nichts Neues; er beinhaltet die Pragmatik des Vertragsmodells. Der zweite Punkt hingegen kann davor schützen, Bedingungen zu übersehen, die nach Ausführung der Operation gelten. Es wäre sicher übertrieben, für jeden Operationsaufruf im Zustandsdiagramm nachzusehen; da die meisten Klassen nur ein sehr kleines Zustandsdiagramm besitzen (vgl. Kapitel 8), lohnt es sich dort nicht. Bei den wenigen Klassen mit komplexeren Zustandsdiagrammen ist jedoch ein Blick auf die Diagramme zu empfehlen um die jeweils gültigen Bedingungen abzulesen.

6 Vom Zustandsdiagramm zu den Zusicherungen

Nachdem gezeigt wurde, wie zu einer gegebenen Klasse das zugehörige Zustandsdiagramm erzeugt werden kann, soll es in diesem Kapitel darum gehen, wie zu einem Zustandsdiagramm der Rumpf einer Klasse gebildet werden kann, d. h. die Signaturen der Operationen einschließlich Zusicherungen. Der gesamte Vorgang wird wieder anhand eines Beispiels nachvollzogen; allerdings wird hier ein anderes Beispiel als in Kapitel 5 gewählt um die verschiedenen Besonderheiten des Verfahrens abzudecken, die sich durchaus von den Besonderheiten des umgekehrten Vorgangs aus dem letzten Kapitel unterscheiden.

6.1 Das Beispieldiagramm

Das Zustandsdiagramm, das in diesem Kapitel als begleitendes Beispiel dienen soll, beschreibt eine einfache Behälterklasse. Es sieht folgendermaßen aus:

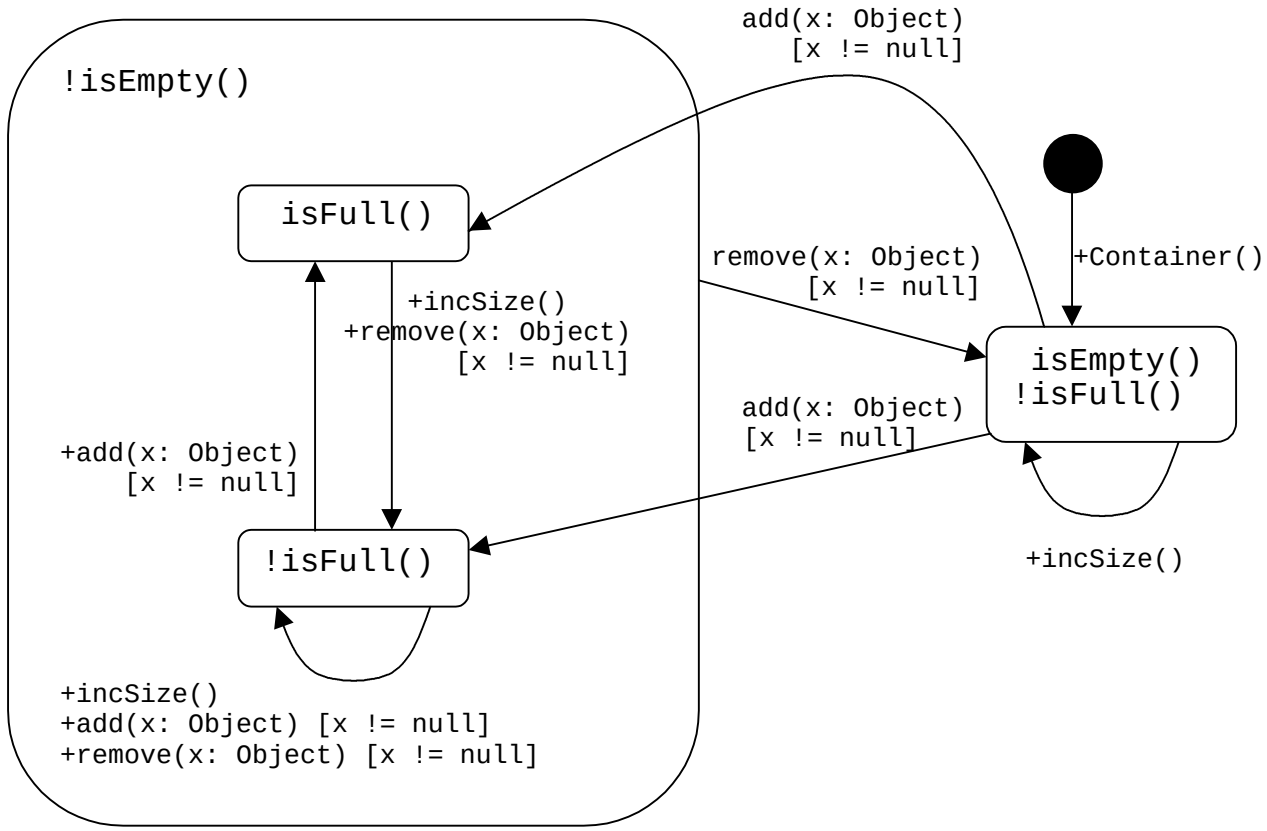


Abbildung 6.1: Zustandsdiagramm der Klasse Container

Es gibt Methoden `add` und `remove` zum Hinzufügen bzw. Entfernen eines Elements. Mit `incSize` kann die Maximalgröße des Behälters erhöht werden.

Die Funktionen `isEmpty` und `isFull` sind an den Transitionen der besseren Übersichtlichkeit wegen weggelassen worden; sie sollen in jedem Zustand aufgerufen werden können und nicht zustandsändernd sein.

6.2 Erzeugen der Zusicherungen

Es soll nun Schritt für Schritt gezeigt werden, wie aus einem Zustandsdiagramm passende Zusicherungen erzeugt werden können. Dazu wird das Zustandsdiagramm zuerst in ein flaches umgewandelt, dann werden die Zusicherungen bestimmt und anschließend vereinfacht, da die erzeugten Zusicherungen zunächst lang und unhandlich sind.

6.2.1 Umwandlung in ein flaches Zustandsdiagramm

Das Zustandsdiagramm muss als erstes in ein flaches umgewandelt werden. Das beinhaltet die Aufteilung von Entscheidungsknoten in mehrere einfache Transitionen sowie die Aufspaltung von sequenziellen und parallelen Unterzuständen in einzelne flache Zustände. In den Abschnitten 5.2.8 bis 5.2.10 wurde dargelegt, wie komplexe und flache Zustandsdiagramme zusammenhängen.

Das Beispieldiagramm enthält lediglich sequenzielle Unterzustände; das entsprechende flache Zustandsdiagramm sieht aus wie folgt:

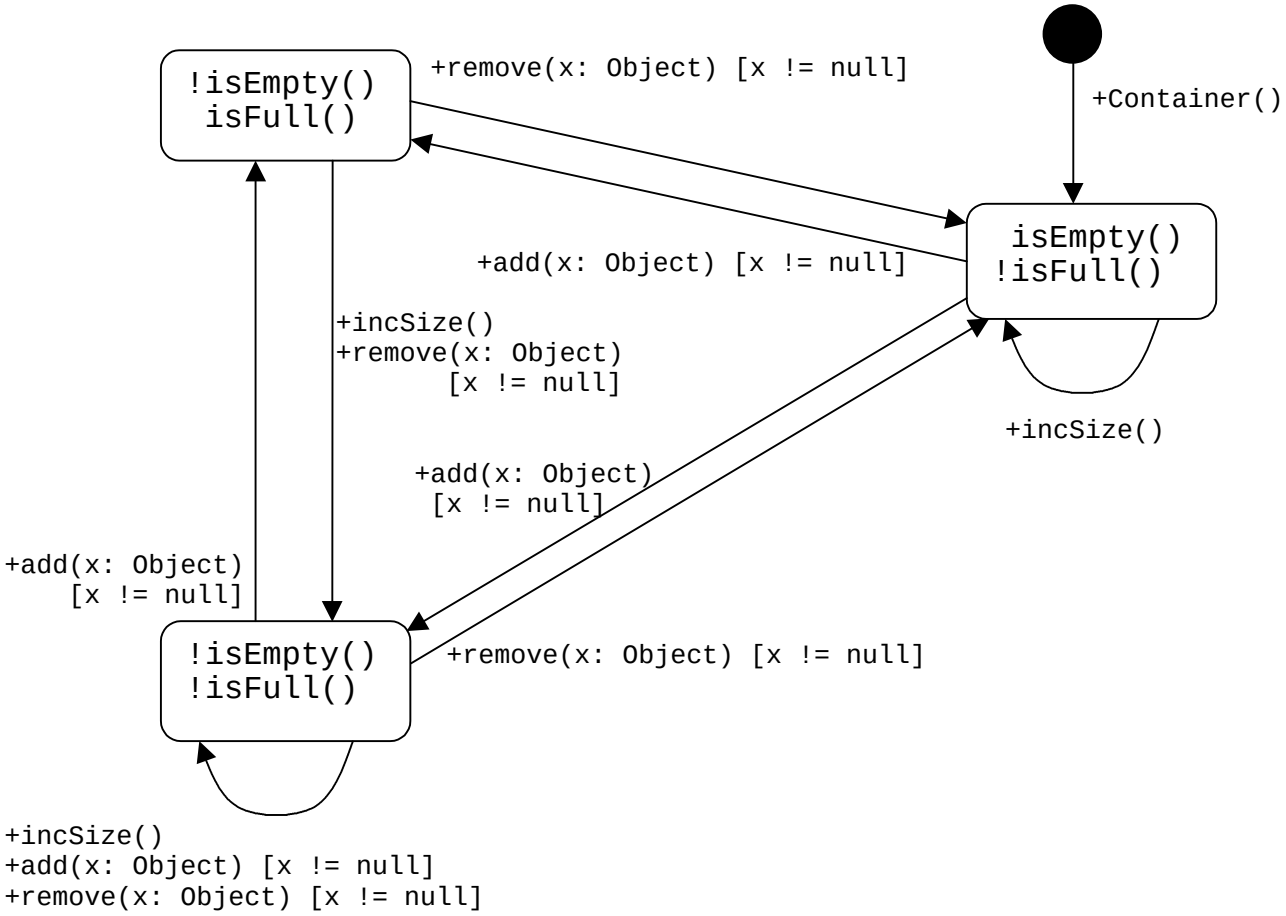


Abbildung 6.2: Flaches Zustandsdiagramm

Es ist wichtig, dass jeder der flachen Zustände exakt dieselben (Zustands-)Prädikate enthält. Ist das nicht der Fall, so ist das Zustandsverhalten nicht ausreichend spezifiziert. Im Folgenden wird davon ausgegangen, dass alle Zustands-Prädikate in jedem Zustand eine eindeutige Belegung haben.

6.2.2 Einteilung der Prädikate

Die Einteilung der Prädikate in Zustands- und Parameter-Prädikate ist auch hier unerlässlich. Die Zustands-Prädikate können direkt aus den Zuständen abgelesen werden. Sie werden mit z_1 bis z_n bezeichnet (n sei die Anzahl der Zustands-Prädikate und kann auch 0 sein, wenn es nur einen Zustand gibt, der mit `true` bezeichnet ist).

Die Parameter-Prädikate jeder Operation sind die Prädikate, die in einer Wächterbedingung oder einem Constraint eines Transitionsereignisses vorkommen, das durch diese Operation ausgelöst wird. Die Parameter-Prädikate werden mit p_1 bis p_m bezeichnet (wobei m die Anzahl der Parameter-Prädikate der zugehörigen Operation ist). Hierbei ist darauf zu achten, dass auch tatsächlich alle Parameter-Prädikate einen Parameter oder das Schlüsselwort `result` enthalten; ist dies nicht der Fall, so entspricht das Zustandsdiagramm nicht den in dieser Arbeit vorausgesetzten Konventionen.[58]

Für das Beispieldiagramm ergeben sich folgende Prädikate, wobei die Nummerierung willkürlich ist:

Zustands-Prädikate:	z_1	`isEmpty()`
	z_2	`isFull()`
Parameter-Prädikate:		
`Container`	–	
`incSize`	–	
`add`	p_1	`x != null`
`remove`	p_1	`x != null`

6.2.3 Finden einer Invariante

Um eine Invariante zu finden müssen zunächst die nicht erreichbaren Zustände entfernt werden. Dies kann mit dem in Abschnitt 5.2.6 angegebenen Algorithmus erreicht werden. Die disjunktive Verknüpfung der Bedingungen der erreichbaren Zustände ergibt dann eine Invariante.

Da im Beispieldiagramm alle Zustände vom Anfangszustand aus erreichbar sind, gilt die Invariante $(\neg z_1 \wedge z_2) \vee (z_1 \wedge \neg z_2) \vee (\neg z_1 \wedge \neg z_2)$ bzw. vereinfacht $\neg z_1 \vee \neg z_2$, das entspricht `!isEmpty() || !isFull()`.

6.2.4 Finden der Vorbedingungen

Das Finden der Vorbedingungen ist unproblematisch: Eine Methode kann in einem Zustand mit einer bestimmten Parameter-Belegung genau dann aufgerufen werden, wenn eine durch diese Methode ausgelöste Transition mit entsprechendem Ursprungszustand und passender Wächterbedingung existiert; bei einem Konstruktor muss eine Transition vom Anfangszustand aus mit passender

[58] Das Zustandsdiagramm kann allerdings in ein gültiges umgewandelt werden, indem das in einer Wächterbedingung oder einem Constraint vorkommende Zustands-Prädikat in die Zustände übernommen wird und die Transitionen entsprechend geändert werden; auf weitere Details wird hier jedoch verzichtet.

Wächterbedingung existieren. Die Vorbedingung einer Operation lässt sich also wie folgt bestimmen:

Ist die Operation ein Konstruktor, so werden sämtliche Wächterbedingungen, die an durch diesen Konstruktor ausgelösten Transitionen stehen, disjunktiv verknüpft. Der resultierende Ausdruck ist die Vorbedingung des Konstruktors.

Handelt es sich um eine Methode, so werden mit den Wächterbedingungen die Bedingungen der Ursprungszustände der Transitionen konjunktiv verknüpft. Die disjunktive Verknüpfung dieser Ausdrücke ergibt dann die Vorbedingung. So ergibt sich z. B. für die Methode `add`, für die es 4 Transitionen gibt, als Vorbedingung $(z_1 \wedge \neg z_2 \wedge p_1) \vee (z_1 \wedge \neg z_2 \wedge p_1) \vee (\neg z_1 \wedge \neg z_2 \wedge p_1) \vee (\neg z_1 \wedge \neg z_2 \wedge p_1)$ oder vereinfacht $\neg z_2 \wedge p_1$; das entspricht `!isFull() && x != null`.

Ist die Vorbedingung nach dem Schema $b \wedge i$ aufgebaut, wobei i die Invariante ist, so kann sie zu b vereinfacht werden, da die Invariante ohnehin gilt. Dies kann bei `incSize` angewandt werden, deren Vorbedingung genau der Invariante entspricht; sie vereinfacht sich somit zu `true`.

Die folgende Tabelle zeigt alle Vorbedingungen der Operationen des Beispieldiagramms:

`Container`	*true*	`true`
`incSize`	*true*	`true`
`add`	$\neg z_2 \wedge p_1$	`!isFull() && x != null`
`remove`	$\neg z_1 \wedge p_1$	`!isEmpty() && x != null`

6.2.5 Finden der Nachbedingungen

Das Finden von Nachbedingungen ist an sich nicht wesentlich schwieriger; allerdings enthalten die Bedingungen zunächst viele Vor-Prädikate. Um diese, soweit möglich, zu entfernen muss einiger Aufwand getrieben werden.

Löst eine Operation eine Transition vom Zustand z_1 zum Zustand z_2 mit Wächterbedingung w und Constraint c aus, so lautet eine passende Nachbedingung $z_1' \wedge w' \wedge z_2 \wedge c$.[59] Dieser Ausdruck besagt, dass vor Ausführung der Transition $z_1 \wedge w$ galt und nachher $z_2 \wedge c$. Die disjunktive Verknüpfung all dieser Ausdrücke zu allen durch die Operation ausgelösten Transitionen ergibt die Nachbedingung.[60] Die Umformung muss zunächst jedoch so erfolgen, dass keine Nach-Zustands-Prädikate entfallen, da dies in der Nochange-Semantik zu einer semantisch anderen Nachbedingung führt. Diese Ausdrücke werden später noch vereinfacht.

Für die Methode `add` erhält man beispielsweise $(z_1' \wedge \neg z_2' \wedge p_1' \wedge \neg z_1 \wedge \neg z_2) \vee (z_1' \wedge \neg z_2' \wedge p_1' \wedge \neg z_1 \wedge z_2) \vee (\neg z_1' \wedge \neg z_2' \wedge p_1' \wedge \neg z_1 \wedge \neg z_2) \vee (\neg z_1' \wedge \neg z_2' \wedge p_1' \wedge \neg z_1 \wedge z_2)$ oder vereinfacht $\neg z_2' \wedge p_1' \wedge \neg z_1 \wedge (z_2 \vee \neg z_2)$; das entspricht `!old isFull() && old (x != null) && !isEmpty() && (isFull() || !isFull())`. Dabei darf `isFull() || !isFull()` (noch) nicht zu `true` vereinfacht werden, da sonst `isFull()` nicht mehr als Nach-Prädikat vorkäme, was in der Nochange-Semantik Auswirkungen hätte.

[59] Der nachgestellte Strich deutet an, dass die Prädikate im jeweiligen Teilausdruck als Vor-Prädikate einzusetzen sind.
[60] Bei Konstruktoren fällt z_1' weg, da es dort keine Vor-Zustands-Prädikate gibt.

Die folgende Tabelle listet die Nachbedingungen zu den Operationen des Beispieldiagramms auf:

`Container`	$z_1 \wedge \neg z_2$	`isEmpty() && !isFull()`
`incSize`	$\neg z_2 \wedge ((z_1' \wedge \neg z_2' \wedge z_1) \vee (\neg z_1' \wedge \neg z_2' \wedge \neg z_1) \vee (\neg z_1' \wedge z_2' \wedge \neg z_1))$	`!isFull() && ((old isEmpty() && !old isFull() && isEmpty()) || (!old isEmpty() && !old isFull() && !isEmpty()) || (!old isEmpty() && old isFull() && !isEmpty()))`
`add`	$\neg z_2' \wedge p_1' \wedge \neg z_1 \wedge (z_2 \vee \neg z_2)$	`!old isFull() && old (x != null) && !isEmpty() && (isFull() || !isFull())`
`remove`	$\neg z_1' \wedge p_1' \wedge \neg z_2 \wedge (z_1 \vee \neg z_1)$	`!old isEmpty() && old (x != null) && !isFull() && (isEmpty() || !isEmpty())`

Die so gebildeten Nachbedingungen bestehen aus zwei konjunktiv miteinander verbundenen Teilausdrücken. Der erste Teilausdruck ist eine reine Konjunktion von Prädikaten[61]; der zweite Teilausdruck ist eine Disjunktion von weiteren Teilausdrücken. Die Teilausdrücke werden im Folgenden weiter vereinfacht.

Im ersten, rein konjunktiv verknüpften Teilausdruck können die Vor-Prädikate entfallen. Sie gelten in jedem Fall, wenn die Nachbedingung erfüllt ist, und werden daher stets bereits in der Vorbedingung erfasst. Bei der Methode `add` kann also `!old isFull() && old (x != null)`entfallen und bei `remove` `!old isEmpty() && old (x != null)`. Ein Blick auf die Vorbedingungen zeigt, dass diese Teilausdrücke tatsächlich redundant sind.

Bei der Vereinfachung des zweiten, aus Disjunktionen bestehenden Teilausdrucks muss zwischen Canchange- und Nochange-Semantik unterschieden werden. In der Canchange-Semantik kann der Ausdruck so weit wie möglich vereinfacht werden; es muss nicht darauf geachtet werden, dass alle Nach-Prädikate im Endausdruck vorkommen.

In der Nochange-Semantik ist eine weitere Vereinfachung möglich. Zunächst wird der Ausdruck so vereinfacht, dass die Vor-Zustands-Prädikate nicht zusammengefasst werden. Hierbei können einige Nach-Zustands-Prädikate verschwinden; für diese wird ein `@canchange`-Tag generiert. Dies ist bei `add` für z_2, also `isFull()`, der Fall, ebenso bei `remove` für z_1, also `isEmpty()`. Nun müssen die Zustands-Prädikate bestimmt werden, für die die Nochange-Semantik gilt. Das sind genau diejenigen Zustands-Prädikate, die in jedem der disjunktiv verknüpften Teilausdrücke als Vor- und Nachprädikate mit derselben Belegung konjunktiv verknüpft sind und ansonsten in der Nachbedingung nicht vorkommen. Bei `incSize` ist dies z_1 bzw. `isEmpty()`. Die anderen Methoden haben keine Zustands-Prädikate, für die die Nochange-Semantik gilt. – Von diesen Prädikaten können nun die Nach-Prädikate aus der Nachbedingung entfernt werden, da in diesem Fall definitionsgemäß die Nochange-Semantik angewandt wird. Anschließend kann eine weitere Vereinfachung erfolgen. Der zweite Teilausdruck von `incSize` vereinfacht sich somit zu $(z_1' \wedge \neg z_2') \vee (\neg z_1' \wedge \neg z_2') \vee (\neg z_1' \wedge z_2')$. Dieser Ausdruck enthält ausschließlich Vor-Prädikate und kann entfallen, da er bereits durch Vorbedingung und Invariante abgedeckt wird. Damit erhält man die erheblich einfachere Nachbedingung $\neg z_2$ bzw. `!isFull()`.

[61] Bei `remove` lautet er z. B. `!old isEmpty() && old (x != null) && !isFull()`.

Ist die Nachbedingung nach dem Schema $b \wedge n$ aufgebaut und lautet die Vorbedingung $b' \wedge v$, so kann b in der Nochange-Semantik aus der Nachbedingung entfallen, wenn b ausschließlich Zustands-Prädikate enthält und diese in n nicht vorkommen (für diese Prädikate gilt dann die Nochange-Semantik). Ist b hingegen die Invariante, so kann b in jedem Fall entfallen, da es ohnehin gilt.

Die erzeugten Nachbedingungen (in der Nochange-Semantik) sind schließlich folgende:

`Container`	$z_1 \wedge \neg z_2$	`isEmpty() && !isFull()`
`incSize`	$\neg z_2$	`!isFull()`
`add`	$\neg z_1$	`!isEmpty()`
`remove`	$\neg z_2$	`!isFull()`

Zu `add` wird außerdem ein `@canchange`-Tag für `isFull()` generiert, zu `remove` für `isEmpty()`.

Nicht immer entfallen alle `old`-Teilausdrücke aus den Nachbedingungen. In der Canchange-Semantik hätte beispielsweise die Nachbedingung von `incSize` nicht weiter vereinfacht werden können und enthielte somit noch mehrere `old`-Teilausdrücke. Ohne das Schlüsselwort `old` könnten also zu einigen Zustandsdiagrammen keine passenden Nachbedingungen generiert werden.

6.2.6 Die Zusicherungen zum Beispieldiagramm

Die Zusicherungen zum Beispieldiagramm können nun komplett angegeben werden. Beim Erzeugen der Tags ist es ratsam, konjunktiv verknüpfte Teilausdrücke auf mehrere Tags zu verteilen um lange Programmzeilen nach Möglichkeit zu vermeiden.

Die Schnittstelle der Klasse sieht folgendermaßen aus:

```
/**
 * @invariant !isEmpty() || !isFull()
 */
public class Container
{
  /**
   * @ensure isEmpty()
   * @ensure !isFull()
   */
  public Container();

  /**
   * @ensure !isFull()
   */
  public void incSize();
```

```
  /**
   * @require !isFull()
   * @require x != null
   * @ensure !isEmpty()
   * @canchange isFull()
   */
  public void add(Object x);

  /**
   * @require !isEmpty()
   * @require x != null
   * @ensure !isFull()
   * @canchange isEmpty()
   */
  public void remove(Object x);

  public boolean isEmpty();

  public boolean isFull();
}
```

6.3 Round-Trip Engineering

In den letzten Jahren von steigender Bedeutung im Softwareentwicklungsprozess ist das so genannte *Round-Trip Engineering* geworden, d. h. die wechselseitige Abbildung von Code und Modell, in unserem Fall von Zusicherungen und Zustandsdiagramm. Die letzten beiden Kapitel haben gezeigt, dass eine solche gegenseitige Abbildung möglich ist.

Interessant ist an dieser Stelle die Frage, inwieweit sich nach Generierung und anschließender Regenerierung die ursprünglichen Zusicherungen bzw. die ursprünglichen Diagramme von den über den Zwischenschritt erzeugten unterscheiden. Diese Frage soll im Folgenden untersucht werden.

Generiert man nach gegebenen Zusicherungen ein Zustandsdiagramm und gewinnt daraus wiederum die Zusicherungen, so können sich die Nachbedingungen unterscheiden. Man betrachte eine Methode, die als Vorbedingung `a` und als Nachbedingung `old a` hat. Das Zustandsdiagramm sieht folgendermaßen aus[62]:

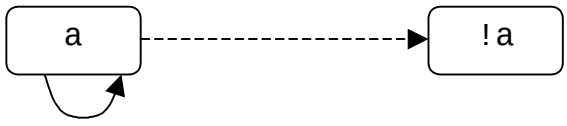


Abbildung 6.3: Zustandsdiagramm bei der Nachbedingung old a

[62] Die gestrichelten Transitionen in den folgenden Beispielen sind diejenigen, die in der Canchange-Semantik gegenüber der Nochange-Semantik hinzukommen.

In beiden Änderungssemantiken wird nach dem vorgestellten Verfahren aus diesem Diagramm die Nachbedingung `true` generiert; `old a` wird nicht zugesichert, da die Vorbedingung es bereits voraussetzt. Der Sinn einer Nachbedingung `old a` ist allerdings nicht recht einzusehen; der einzige erkennbare Nutzen ist, die Vorbedingung in Unterklassen nicht abschwächbar zu machen, da dies eine (illegale) Abschwächung der Nachbedingung mit sich ziehen würde. Diese Anwendung ist allerdings nicht als vorbildlich anzusehen, weil sie dem Erweiterungsprinzip der Objektorientierung entgegensteht.

Lautet die ursprüngliche Nachbedingung hingegen `a`, so sieht das Zustandsdiagramm wie folgt aus:

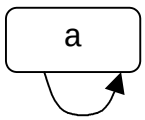


Abbildung 6.4: Zustandsdiagramm bei der Nachbedingung a

Die neue Nachbedingung lautet nun in der Canchange-Semantik `a` und in der Nochange-Semantik `true`. Aufgrund der Semantik ist es im letzteren Fall nicht notwendig, `a` zuzusichern, da sich dieses Zustands-Prädikat dann nicht ändert. Die ursprüngliche Nachbedingung ist also in der Nochange-Semantik überflüssig.

Weitere Differenzen kann es bei der Verwendung von Invarianten geben. Man betrachte eine Klasse mit der Invariante `a || b`. Eine Methode habe die Vorbedingung `!a` und die Nachbedingung `!b`. Das Zustandsdiagramm ist folgendes:

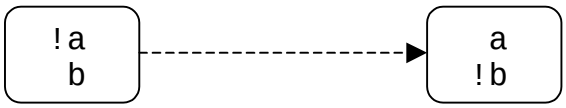


Abbildung 6.5: Zustandsdiagramm einer Klasse mit Invariante

In der Nochange-Semantik kann die Methode ihre Zusicherungen nicht erfüllen, da der Zielzustand `!a && !b` aufgrund der Invariante nicht eingenommen werden kann. Als Vorbedingung wird also `false` generiert.[63] In der Canchange-Semantik hingegen wird als Vorbedingung `!a && b` und als Nachbedingung `a && !b` erzeugt, was zwar auch nicht den ursprünglichen Zusicherungen entspricht, aber aufgrund der Invariante auch dort gelten muss. – Die aus dem Zustandsdiagramm generierte Invariante kann schließlich stärker sein als die ursprünglich angegebene, da sie die tatsächlich erreichbaren Zustände beschreibt. Die vom Programmierer formulierte Invariante muss nicht zwingend die möglichst stärkste sein; auch ohne eine explizit angegebene Invariante kann es unerreichbare Zustände geben.

Bei der Zurückgewinnung der Zusicherungen aus einem generierten Zustandsdiagramm kann es also zu geringen Unterschieden kommen, die jedoch dasselbe Zustandsdiagramm beschreiben und lediglich Bedingungen ergänzen oder fortlassen, die ohnehin in beiden Fällen gelten.

[63] Streng genommen ist dies ein Unterschied zu den ursprünglichen Zusicherungen, da nun ein Aufrufer der Methode die Vorbedingung nicht einhalten kann, während ursprünglich die Methode ihre Nachbedingung nicht erfüllen konnte. Der Unterschied besteht darin, wer den Vertrag gebrochen hat. Dies ändert jedoch nichts daran, dass die Methode nicht ohne Vertragsbruch aufgerufen werden kann.

Bei der Rückgewinnung eines Zustandsdiagramms über zwischendurch generierte Zusicherungen sind die Unterschiede noch geringer, wenn man lediglich flache Zustandsdiagramme ohne spezielle Elemente wie Entscheidungsknoten betrachtet. Hier kann es höchstens zu Differenzen kommen, wenn das ursprüngliche Diagramm nicht erreichbare Zustände enthält. Ansonsten entsprechen sich die Diagramme. Der Grund ist die Festlegung des Klassenprotokolls durch das ursprüngliche Diagramm; dieses Protokoll wird durch das Zustandsdiagramm vollständig und eindeutig ausgedrückt und kann somit lückenlos wiedergewonnen werden.

Als Ergebnis kann also festgestellt werden, dass eine gegenseitige Abbildung von Zusicherungen und Zustandsdiagrammen möglich ist (bei unwesentlichen Unterschieden). Beim Programmieren von Klassen ist es daher problemlos möglich, zwischen den Zusicherungen und dem Zustandsdiagramm einer Klasse hin- und herzuschalten und Änderungen dort vorzunehmen, wo sie gewünscht sind. Das kann sowohl bei den Zusicherungen sein (z. B. wenn sie zu lang und unübersichtlich sind) als auch beim Zustandsdiagramm (z. B. wenn unerwünschte Zustandsübergänge möglich sind). Diese parallele Pflege von Zusicherungen und Zustandsdiagramm führt gleichermaßen zu robusten Zusicherungen wie zu einem stabilen Klassenprotokoll.

7 Der Statechartgenerator

Im Rahmen dieser Arbeit entstand der in Java programmierte *Statechartgenerator*, der in der Lage ist, zu einer Java-Klasse mit Zusicherungen ein Zustandsdiagramm zu erzeugen. Das Diagramm kann vom Anwender mit der Maus verändert werden, woraufhin die neuen Zusicherungen sich wiederum auf den Quellcode anwenden lassen können. Der Statechartgenerator wurde als Erweiterung (OpenTool) zum JBuilder [URL4] entwickelt, einer integrierten Java-Entwicklungsumgebung von Borland.

In Abschnitt 6.3 wurde deutlich, dass eine gegenseitige Abbildung von Zusicherungen und Zustandsdiagrammen möglich und sinnvoll ist. Eine parallele Pflege der Zusicherungen und des Zustandsdiagramms einer Klasse sollte angestrebt werden, insbesondere bei Klassen mit komplexem Zustandsverhalten. Für Klassen ohne Zustands-Prädikate (dies sind die meisten Klassen, vgl. Kapitel 8) ist der Statechartgenerator nicht besonders hilfreich; bei Klassen mit mehreren Zustands-Prädikaten hingegen sollte der Statechartgenerator auf jeden Fall eingesetzt werden um den Überblick über das Zustandsverhalten der Klasse zu behalten. Entspricht dieses nicht dem intendierten Verhalten, kann das Zustandsdiagramm manuell angepasst werden, woraufhin die Zusicherungen automatisch geändert werden. Sind die Zusicherungen daraufhin zu komplex, können diese wiederum modifiziert werden; so können Zusicherungen und Zustandsdiagramm schrittweise angepasst werden, sodass am Ende sowohl robuste und einfache Zusicherungen als auch ein wohldefiniertes Klassenprotokoll gegeben sind.

7.1 Integration in JBuilder

Da die Generierung von Zustandsdiagrammen eine Unterstützung im Softwareentwicklungsprozess ist, bietet sich eine Einbindung in eine integrierte Entwicklungsumgebung an. Der JBuilder bietet hierbei die folgenden signifikanten Vorteile:

- weite Verbreitung
- großer Funktionsumfang
- einfache Einbindung von Erweiterungen über die OpenTools-API
- freie Verfügbarkeit in der Grundversion

Aus diesen Gründen fiel die Wahl auf eine Einbindung des Statechartgenerators in den JBuilder.[64] Der folgende Screenshot gibt einen Eindruck des Werkzeugs:

[64] Mittlerweile ist die Eclipse-Plattform [URL5] ebenso gut für eine Einbindung geeignet, da die angeführten Gründe auch dort zutreffen. Da Eclipse zum Zeitpunkt der Implementierung des Statechartgenerators noch nicht die Bedeutung hatte, die es heute hat, fiel die Entscheidung für den JBuilder.

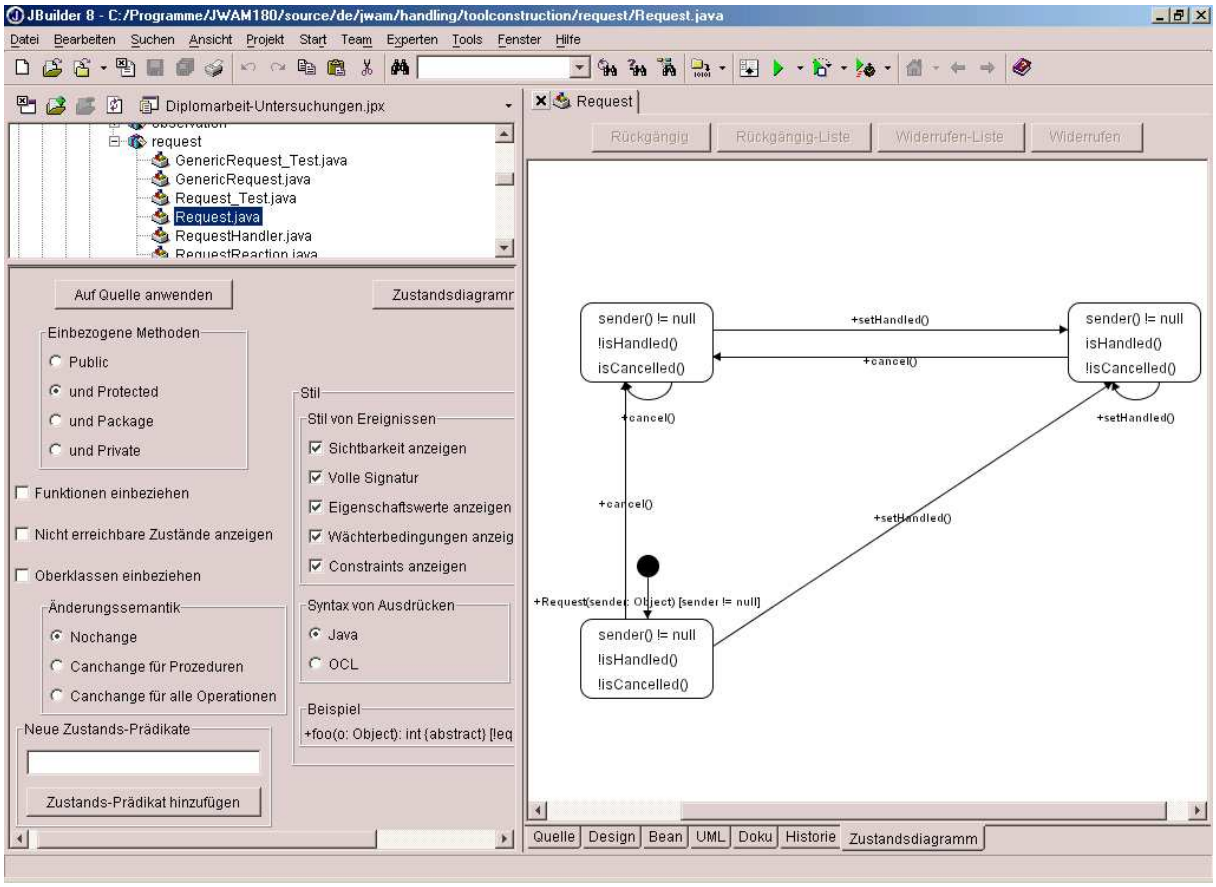


Abbildung 7.1: JBuilder 8 Enterprise mit Statechartgenerator

Links können diverse Optionen gesetzt werden, die später noch vorgestellt werden; rechts ist das Zustandsdiagramm der aktuellen Klasse zu sehen. Zu den Registern, die verschiedene Ansichten der Klasse widerspiegeln (Quelle, Design etc.), wurde ein Register „Zustandsdiagramm“ hinzugefügt. Mit der Maus kann nun dieses Zustandsdiagramm manipuliert werden. Im Einzelnen können:

- Transitionen und Anfangszustände hinzugefügt werden
- Transitionen bearbeitet werden (d. h. Ereignisse mit Wächterbedingungen und Constraints hinzugefügt und entfernt werden)
- beliebige Elemente entfernt werden
- beliebige Elemente verschoben werden

Das Bearbeiten einer Transition geschieht in einem eigenen Dialogfenster:

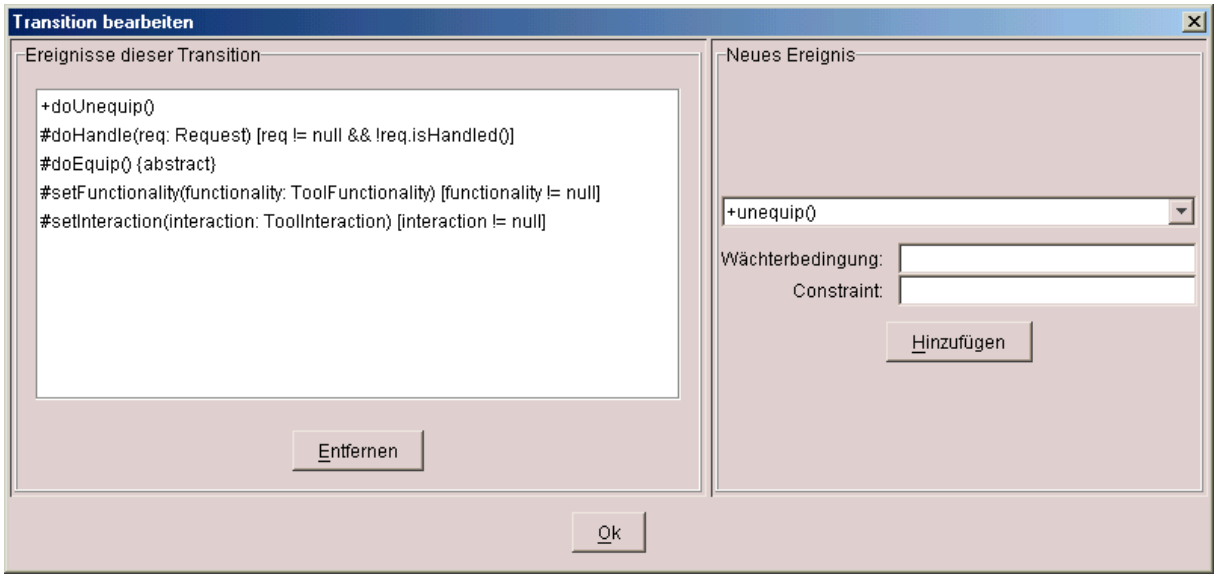


Abbildung 7.2: Dialog „Transition bearbeiten“

Zusätzlich steht ein mehrfacher Undo-/Redo-Mechanismus („Rückgängig“ und „Widerrufen“) zur Verfügung.

Neue Zustände können implizit hinzugefügt werden, indem neue Zustands-Prädikate erzeugt werden (im linken Fensterbereich) und das Zustandsdiagramm neu aufgebaut wird.

Nach den Änderungen kann der Knopf „Auf Quelle anwenden“ gedrückt werden, woraufhin die neuen Zusicherungen in den Quelltext der Klasse geschrieben werden.

Das Benutzungsmodell des Statechartgenerators sieht also so aus, dass zunächst der Code der Klasse geschrieben wird, wobei die Zusicherungen in den Dokumentationskommentaren angegeben werden. Bei Bedarf lässt man sich das Zustandsdiagramm anzeigen und nimmt dort gegebenenfalls Anpassungen vor, die man wiederum auf die Quelle anwendet. Dabei kann, falls erwünscht, auch gleich der Code zum Überprüfen der Zusicherungen generiert werden. Hier muss allerdings beachtet werden, dass statische Methoden vom Statechartgenerator generell nicht betrachtet werden.

7.2 Die Optionen

Der Statechartgenerator bietet diverse Optionen an, die Einfluss auf das zu erzeugende Zustandsdiagramm nehmen. Die Optionen können für jede Klasse individuell gesetzt werden:

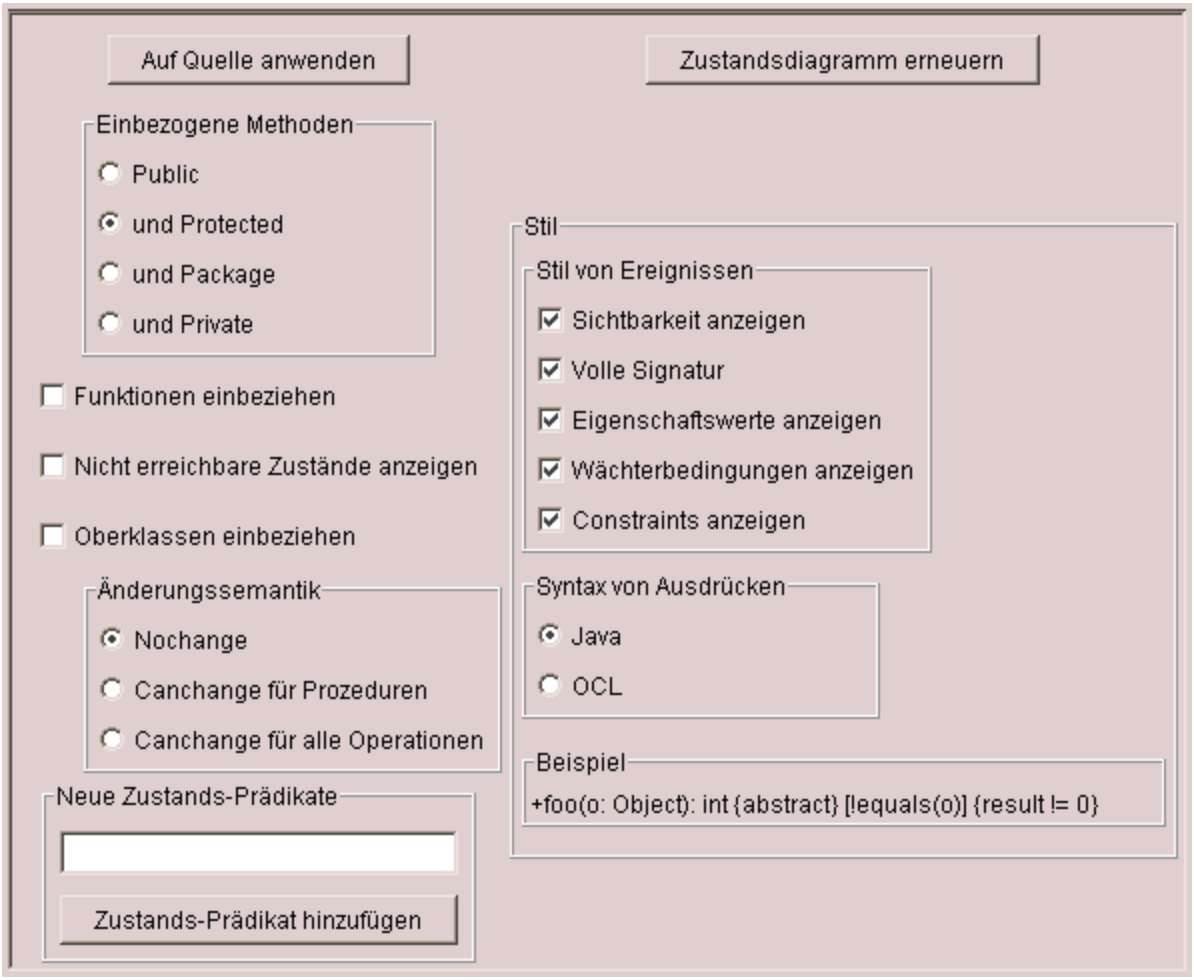


Abbildung 7.3: Optionen des Statechartgenerators

Die Optionen sollen nun im Einzelnen erläutert werden:

- **Einbezogene Methoden:** Hier wird festgelegt, Methoden welcher Sichtbarkeit in das Zustandsdiagramm aufgenommen werden. *Package* steht dabei für die Standard-Sichtbarkeit, die zur Anwendung kommt, wenn zur Methode keine Sichtbarkeit explizit angegeben ist. Es ist immer genau einer der vier Knöpfe ausgewählt, wobei die höheren Sichtbarkeitsstufen immer mit einbezogen werden; es ist also z. B. nicht möglich, nur Methoden einzubeziehen, die als *protected* oder *private* deklariert sind. – Konstruktoren werden unabhängig von ihrer Sichtbarkeit immer mit einbezogen. Das liegt daran, dass ansonsten mögliche Anfangszustände verloren gehen könnten, was dazu führen könnte, dass einige erreichbare Zustände als nicht erreichbar eingestuft würden.

- **Funktionen einbeziehen:** Ist dieser Schalter aktiviert, so werden Funktionen mit einbezogen, ansonsten nicht. Der Grund, warum man Funktionen oft nicht betrachten möchte, ist, dass sie in einer konzeptionell „sauberen“ Klasse den Objektzustand nicht verändern und somit ausschließlich an Selbsttransitionen stehen. Interessant wäre dann nur noch, in welchen Zuständen sie aufgerufen werden dürfen, eventuell auch die Wächterbedingungen und Constraints. Da es aber meist eher auf Prozeduren ankommt und Zustandsdiagramme mit vielen Ereignissen dazu neigen, unübersichtlich zu werden, wird man häufig auf die Einbeziehung von Funktionen verzichten.

- **Nicht erreichbare Zustände anzeigen:** Ist diese Option gesetzt, so werden auch Zustände angezeigt, die nicht erreichbar sind. Das kann sinnvoll sein, wenn man das Zustandsdiagramm ändern möchte, sodass auch einige dieser Zustände erreichbar sind.

- **Oberklassen einbeziehen:** Beim Setzen dieser Option werden die Methoden sämtlicher Oberklassen (bis hin zu `java.lang.Object`) in den Generierungsprozess einbezogen, sofern der Quellcode gefunden werden kann.

- **Änderungssemantik:** Hier kann eingestellt werden, welche Änderungssemantik benutzt wird, wenn ein Nach-Zustands-Prädikat in einer Nachbedingung nicht vorkommt. Bei Canchange kann unterschieden werden, ob diese Semantik nur für Prozeduren oder für alle Operationen verwendet werden soll. Da man in einem „sauberen“ Modell davon ausgeht, dass Funktionen den Objektzustand nicht ändern, kann es zuweilen sinnvoll sein, die Canchange-Semantik nur bei Prozeduren anzuwenden, bei Funktionen hingegen die Nochange-Semantik zu benutzen.

- **Neue Zustands-Prädikate:** Es ist möglich, in das Textfeld neue Zustands-Prädikate einzugeben, die im Quellcode bislang noch nicht vorkommen. Sie erscheinen dann in den Zuständen zusätzlich zu den bereits vorhandenen. Das kann nützlich sein, wenn man das Zustandsdiagramm ändern möchte.

- **Stil von Ereignissen:** Hier wird festgelegt, wie Ereignisse von Transitionen angezeigt werden. Folgende Optionen stehen zur Auswahl:

  - **Sichtbarkeit anzeigen:** Das UML-Zeichen für die Sichtbarkeit wird der Operation vorangestellt.

  - **Volle Signatur:** Parameternamen und -typen der Operationen und Typ des Rückgabewertes werden angezeigt.

  - **Eigenschaftswerte anzeigen:** Die Eigenschaftswerte werden angezeigt (`abstract`, `leaf` bzw. `concurrency = guarded`).

  - **Wächterbedingungen anzeigen:** Legt fest, ob Wächterbedingungen angezeigt werden.

  - **Constraints anzeigen:** Legt fest, ob Constraints angezeigt werden. Man beachte, dass die Deaktivierung der letzten beiden Optionen zu einem Zustandsdiagramm mit weniger Ereignissen führen kann. Das passiert, wenn es Transitionen derselben Operation mit verschiedenen Wächterbedingungen und unterschiedlichen Constraints gibt. Die Deaktivierung mindestens einer der letzten beiden Optionen führt dann dazu, dass nur noch ein Ereignis übrig bleibt.

- **Syntax von Ausdrücken:** Ausdrücke werden entweder in Java- oder in OCL-Syntax angezeigt. Das betrifft sowohl die Zustands-Prädikate, die in den Zuständen angezeigt werden, als auch Wächterbedingungen und Constraints.

Über den Menüpunkt *Tools* → *Statechartgenerator konfigurieren* können weitere Optionen gesetzt werden, die die Codegenerierung für Zusicherungen betreffen:

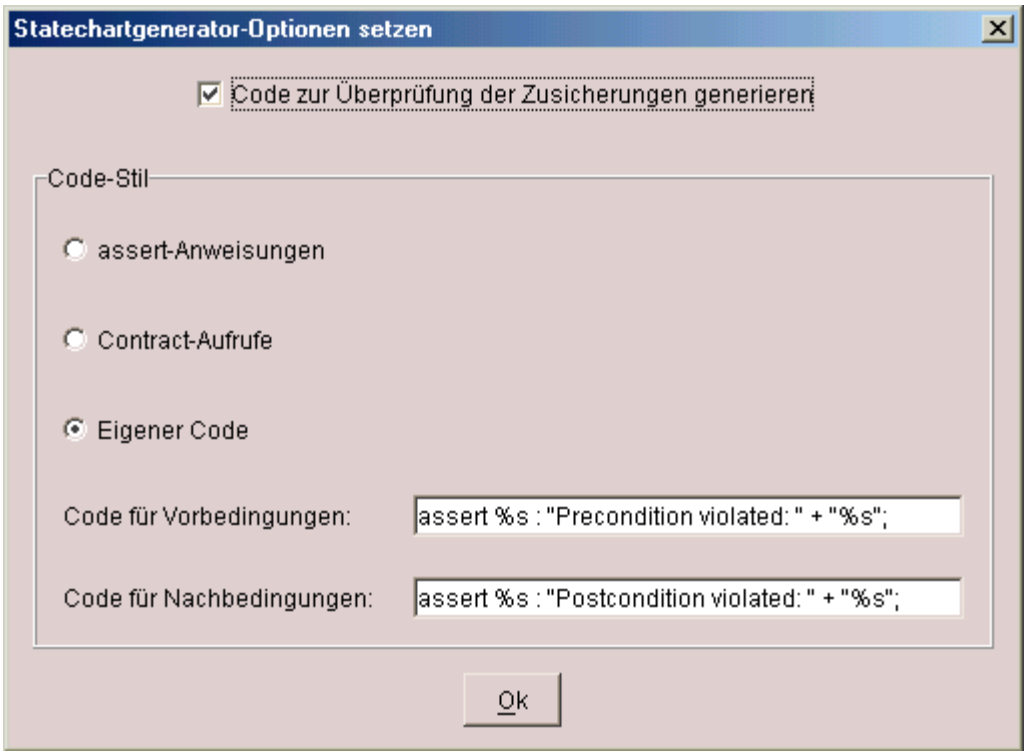


Abbildung 7.4: Dialog „Statechartgenerator-Optionen setzen“

Falls Code generiert werden soll, kann zwischen drei Möglichkeiten ausgewählt werden: `assert`-Anweisungen der Form "`assert` *`Bedingung`*`;`", den aus JWAM bekannten `Contract`-Aufrufen und einem selbst definierten Format. Bei der letzten Möglichkeit werden in den beiden Textfeldern Muster für den Code zur Auswertung der Vor- und Nachbedingungen angegeben, wobei "%s" durch die aktuelle Zusicherung ersetzt wird.

7.3 Die Generierungsprozesse

Im Folgenden soll die Vorgehensweise des Programms genauer beschrieben werden. Dabei gibt es genau genommen zwei Generierungsprozesse: die Generierung eines Zustandsdiagramms aus den Zusicherungen der Klasse und die Generierung der Zusicherungen aus dem Zustandsdiagramm. Diese beiden Prozesse werden nun erläutert.

7.3.1 Die Generierung des Zustandsdiagramms

Die Generierung des Zustandsdiagramms besteht aus den folgenden Schritten: Parsen der Klasse(n), Erzeugen des Modells des Zustandsdiagramms und Anzeigen des Zustandsdiagramms.

1. **Parsen der Klasse(n):** Sollen Oberklassen einbezogen werden, so wird zunächst (rekursiv) die Oberklasse der aktuellen Klasse geparst, sofern die aktuelle Klasse nicht `java.lang.Object` ist und der Quellcode gefunden wird. Das Parsen beinhaltet das Aufbauen einer Liste sämtlicher Konstruktoren (nur der untersten Klasse) und einer Liste der nichtstatischen Methoden mit passender Sichtbarkeit. Außer den Signaturen dieser Operationen werden die Zusicherungen ausgelesen, indem aus den Dokumentationskommentaren die `@require`-, `@ensure`- und `@canchange`-Tags ausgewertet werden. Dabei wird beachtet,

dass ein Standardkonstruktor ohne Parameter und mit Vor- und Nachbedingung `true` erzeugt wird, wenn die unterste Klasse keinen explizit deklarierten Konstruktor enthält. Aus dem Dokumentationskommentar der Klasse werden außerdem die `@invariant`-Tags ausgewertet. – Aus den Zusicherungen werden bereits die Prädikate bestimmt und eingeteilt, sodass die Zusicherungen als boolesche Verknüpfungen von Parameter- und Zustands-Prädikaten modelliert werden. Jede Operation hält eine Liste ihrer Parameter-Prädikate, während eine einzige Liste für die Zustands-Prädikate gehalten wird, zu der auch die vom Anwender explizit neu erzeugten Zustands-Prädikate hinzugefügt werden.

2. **Erzeugen des Modells des Zustandsdiagramms:** Aus den Zusicherungen wird nun ein Modell des Zustandsdiagramms erzeugt, bestehend aus Zuständen und Transitionen. Dazu werden zunächst die Konstruktoren, dann die Methoden nacheinander abgearbeitet. Zur schnelleren Abarbeitung werden die Vor- und Nachbedingungen jeweils in *feste* und *variable* Bedingungen unterteilt. Feste Bedingungen sind solche, die ausschließlich aus Konjunktionen von (eventuell negierten) Prädikaten bestehen. Diese Bedingungen sind deshalb „fest“, weil es nur eine Belegung der enthaltenen Prädikate gibt, in der die Bedingung gilt. Andere Belegungen brauchen daher nicht ausprobiert zu werden. Die Prädikate, die fest sind, werden markiert und ihre Belegung gemerkt. Als nächstes erfolgt die Bestimmung der relevanten Vor- und Nach-Prädikate. Dies geschieht gemäß Abschnitt 5.2.3. – Nun werden gemäß dem Algorithmus aus Abschnitt 5.2.5 die Transitionen bestimmt, wobei die Belegungen der festen Prädikate nicht verändert werden. Existiert bereits eine Transition vom selben Ursprungs- zum selben Zielzustand, so wird zu dieser ein neues Ereignis hinzugefügt, ansonsten wird eine neue Transition erzeugt. Wurde der Ursprungs- oder Zielzustand der Transition noch nicht generiert, so geschieht das beim Erzeugen der Transition. Die Zustände setzen sich aus den Zustands-Prädikaten zusammen. Sollen alle Zustände angezeigt werden, so werden am Schluss die fehlenden Zustände hinzugefügt; ansonsten werden mithilfe des Algorithmus aus Abschnitt 5.2.6 die nicht erreichbaren Zustände bestimmt und entfernt.

3. **Anzeigen des Zustandsdiagramms:** Jetzt muss das Zustandsdiagramm noch angezeigt werden. Hierzu werden die Zustände in annähernd gleich vielen Zeilen und Spalten auf der Anzeigefläche positioniert. Die Transitionen werden so platziert, dass sie nicht durch ihre Ursprungs- oder Zielzustände hindurchlaufen. Ist der Ursprungszustand einer Transition der Anfangszustand, so wird zu jedem Zielzustand ein eigener Anfangszustand gezeichnet, und zwar über dem Zielzustand. Dies sorgt für eine bessere Übersichtlichkeit.

7.3.2 Die Generierung der Zusicherungen

Bei der Generierung der Zusicherungen aus dem Zustandsdiagramm werden folgende Schritte ausgeführt: Erzeugen von Zusicherungen aus den Transitionsereignissen, Vereinfachen der Zusicherungen und Schreiben der Zusicherungen.

1. **Erzeugen von Zusicherungen aus den Transitionsereignissen:** Zuerst werden aus sämtlichen Transitionsereignissen Zusicherungen generiert, indem Ursprungszustand und Wächterbedingung zur Vorbedingung, Ursprungs-, Zielzustand, Wächterbedingung und Constraint zur Nachbedingung und Ursprungs- und Zielzustand zur Invariante hinzugefügt werden. Konkret bedeutet das, dass für eine Transition von z_1 nach z_2 mit Wächterbedingung w und Constraint c zur Vorbedingung z_1 `&&` w, zur Nachbedingung `old` z_1 `&&` z_2 `&&` `old` w `&&` c sowie zur Invariante z_1 `||` z_2 disjunktiv hinzugefügt werden.

2. **Vereinfachen der Zusicherungen:** Das Vereinfachen betrifft vor allem die Nachbedingungen und geschieht gemäß Abschnitt 6.2.5. Dabei wird die eingestellte Änderungssemantik berücksichtigt. Außerdem werden konjunktiv verknüpfte Teilausdrücke in mehrere einzelne Zusicherungen zerlegt, sodass aus b_1 && b_2 zwei Zusicherungen b_1 und b_2 werden.

3. **Schreiben der Zusicherungen:** Schließlich werden die Zusicherungen in die Klasse zurückgeschrieben. Dazu werden die alten `@require`-, `@ensure`-, `@canchange`- und `@invariant`-Tags aus den Dokumentationskommentaren entfernt und die neuen hineingeschrieben. Ist die Codeerzeugung eingeschaltet, so werden außerdem nach dem Löschen des alten Codes `Contract`-Aufrufe bzw. `assert`-Anweisungen in den Rumpf der Operation geschrieben, und zwar Vorbedingungen an den Anfang (aber nach einen eventuellen `this`- oder `super`-Aufruf eines Konstruktors) und Nachbedingungen an das Ende (aber vor eine eventuelle `return`-Anweisung). Zu Nachbedingungen, die das Schlüsselwort `old` enthalten, wird allerdings kein Code generiert, da der Code sonst zu umständlich und ineffizient würde (die Anfangsbelegungen der `old`-Teilausdrücke müssten in lokalen Variablen abgelegt werden). Zu Nachbedingungen, die das Schlüsselwort `result` enthalten, wird nur dann Code erzeugt, wenn die Methode mit der Anweisung "`return result;`" endet; dann kann die Nachbedingung im Code direkt übernommen werden.

7.4 Effizienzbetrachtungen

Programme sollen möglichst effizient arbeiten. Dabei unterscheidet man zwischen Platz- und Zeitbedarf.

Der Platzbedarf des Statechartgenerators ergibt sich hauptsächlich aus der Anzahl der Zustände und Transitionen, da die Zahl der Zustände exponentiell mit der Zahl der Zustands-Prädikate wächst, wobei demgegenüber die Zahl der Zusicherungen (und damit auch der Operationen) im realistischen Fall vergleichsweise gering ist (riesige Klassen sind im Regelfall nicht erwünscht). Mehr Platzbedarf als für das resultierende Zustandsdiagramm tatsächlich nötig entsteht lediglich durch nicht erreichbare Zustände und Transitionen. Bei n Zustands-Prädikaten gibt es maximal 2^n Zustände und 2^{2n} Transitionen (von jedem Zustand zu jedem), jede mit möglicherweise mehreren Ereignissen. Berücksichtigt man, dass zu jedem Zustand noch ein Anfangszustand kommen kann, erhöht sich die Anzahl der Zustände und Transitionen jeweils noch um 2^n. Die maximale Zahl der Transitionen führt somit dazu, dass der Platzbedarf für das Zustandsdiagramm von der Ordnung 2^{2n} ist; der Statechartgenerator braucht jedoch nicht größenordnungsmäßig mehr Platz.

Der Zeitbedarf für das Generieren des Zustandsdiagramms wird bestimmt durch den Algorithmus zur Erzeugung der Transitionen (siehe Abschnitt 5.2.5). Dort ergibt er sich aus der (maximalen) Anzahl der Durchläufe der „für alle"-Schleifen. Diese ergeben sich wie folgt (n sei die Anzahl der Zustands-Prädikate, p die Anzahl der Parameter-Prädikate der betrachteten Operation):[65]

(1) Bei Konstruktoren wird nicht über die Vor-Zustands-Prädikate iteriert, daher erfolgt hier nur ein Schleifendurchlauf. Bei Methoden werden alle Belegungen durchgespielt, dort erfolgen 2^n Durchläufe.
(3) Hier erfolgen maximal 2^n Durchläufe, auch wenn es in der Nochange-Semantik oft weniger sind, weil nicht alle Nach-Zustands-Prädikate relevant sind.

[65] Die Nummern in Klammern entsprechen den Nummern aus Abschnitt 5.2.5 und bezeichnen ausgewählte Schritte des Algorithmus.

(7) Die maximale Zahl der Durchläufe ist 2^p. Es können aber weniger sein, weil nicht alle Vor-Parameter-Prädikate relevant sind.
(10) Auch hier erfolgen maximal 2^p Durchläufe, auch wenn oft nicht alle Nach-Parameter-Prädikate relevant sind.
(19) Die Anzahl der Wächterbedingung-Constraint-Paare kann die Anzahl der Durchläufe von Schleife (10) nicht übersteigen, weil innerhalb dieser Schleife höchstens ein Paar hinzugefügt wird (in (18)). Im Regelfall ist die Anzahl dieser Paare zudem sehr klein.

Für die maximale Anzahl der Durchläufe ergibt sich somit $2^n \cdot 2^n \cdot 2^p \cdot 2^p = 2^{2n+2p}$. Diese exponentielle Zeitkomplexität würde im Regelfall als ineffizient angesehen werden, da für größere Werte von *n* oder *p* der Zeitbedarf sehr schnell wächst. Folgende Gründe rechtfertigen dennoch den Gebrauch dieses Algorithmus:

1. Untersuchungen in Kapitel 8 werden ergeben, dass die Werte für *n* und *p* in der Praxis immer sehr klein sind. Für solche kleinen Werte besteht kein großer Unterschied zwischen exponentiellem und polynomiellem Zeitbedarf.

2. Die Schleifen werden vom Programm nur für die variablen Prädikate (s. Abschnitt 7.3.1) mehr als einmal durchlaufen. In der Praxis sind die meisten Prädikate jedoch fest, da andere Verknüpfungen als Konjunktion und Negation in Zusicherungen selten anzutreffen sind. Dies wird ebenfalls in Kapitel 8 dargelegt.

3. Eine weitere Reduktion ergibt sich dadurch, dass nicht alle Prädikate relevant sind. Dies betrifft z. B. viele Nach-Prädikate in der Nochange-Semantik. Bei Parameter-Prädikaten gilt außerdem, dass in aller Regel entweder das entsprechende Vor-Prädikat oder das Nach-Prädikat relevant ist, da es sehr selten ist, dass dasselbe Parameter-Prädikat sowohl in der Vorbedingung als auch in der Nachbedingung vorkommt.

Der tatsächliche Zeitbedarf ist also in der Regel vertretbar. Dies zeigt auch die praktische Erfahrung. Selbst für JWAM-Klassen mit 8 Zustands-Prädikaten wird das Zustandsdiagramm innerhalb weniger Sekunden generiert. Bei Klassen mit 3 Zustands-Prädikaten (was auch schon viel ist) ist bereits keine wesentliche Verzögerung bemerkbar.

Bei der Generierung der Zusicherungen aus dem Diagramm müssen die Transitionen abgearbeitet werden; dies sind, wie bereits oben dargelegt, größenordnungsmäßig 2^{2n}. Der Zeitbedarf zur Vereinfachung der Nachbedingungen ist ebenfalls im schlechtesten Fall exponentiell, da die Länge der Nachbedingungen exponentiell sein kann (wenn alle Belegungen der Vor-Prädikate explizit vorkommen). Das Entfernen der Nach-Zustands-Prädikate, für die die Nochange-Semantik gilt, kann jedoch in zwei Durchläufen geschehen: Im ersten werden die Prädikate bestimmt, die in jedem Teilausdruck als Vor- und Nach-Prädikate mit derselben Belegung vorkommen, im zweiten werden die entsprechenden Nach-Prädikate entfernt. Das Vereinfachen der Ausdrücke erfordert wiederum exponentiellen Zeitaufwand, ebenso das Schreiben der Zusicherungen, was an der möglicherweise exponentiellen Länge der Zusicherungen liegt.

Obwohl auch das Erzeugen der Zusicherungen exponentiellen Zeitaufwand benötigt, geht es in der Regel sehr zügig, was zum einen daran liegt, dass die Zahl der Prädikate meist sehr begrenzt ist (vgl. Kapitel 8), zum anderen daran, dass die obigen Betrachtungen jeweils für den schlechtesten Fall gelten, die meisten Zusicherungen jedoch kürzer sind und nicht alle Prädikate enthalten.

8 Bezug zur Praxis

In diesem Kapitel soll ein Bezug zur Praxis hergestellt werden. Zunächst wird anhand eines Fallbeispiels die Arbeit mit dem Statechartgenerator demonstriert. Weiterhin erfolgt eine systematische Untersuchung der Zusicherungen und Zustandsdiagramme existierender Java-Klassen. Hierzu bietet sich das JWAM-Rahmenwerk an. Schließlich wird eine Einordnung des vorgestellten Ansatzes in bereits existierende Ansätze vorgenommen.

8.1 Ein Fallbeispiel

Die Arbeit mit dem Statechartgenerator soll anhand eines Fallbeispiels demonstriert werden. Dazu wird die JWAM-Klasse `de.jwam.handling.toolconstruction.request.Request` betrachtet. Ihre Schnittstelle sieht folgendermaßen aus:

```
public class Request
{
  /**
   * @require sender != null
   * @ensure sender() != null
   * @ensure !isHandled()
   * @ensure !isCancelled()
   */
  public Request(Object sender);

  /**
   * @ensure sender != null
   */
  public Object sender();

  /**
   * @ensure isHandled()
   * @ensure !isCancelled()
   */
  public void setHandled();

  /**
   * @ensure !isHandled()
   * @ensure isCancelled()
   */
  public void cancel();

  public boolean isHandled();

  public boolean isCancelled();
}
```

Hierzu ist zunächst anzumerken, dass mit der Nachbedingung der Funktion `sender` wahrscheinlich `result != null` gemeint war, da in der Implementierung der Funktion das Objekt nicht geändert wird. Die Implementierung sieht aus wie folgt:

```
public Object sender()
{
  assert _sender != null : "Postcondition violated: "
                         +  "_sender";
  return _sender;
}
```

Hier ist ersichtlich, dass die Zusicherung eigentlich `_sender != null` lauten sollte; die tatsächliche Intention war aber keine Zusicherung über das Attribut `_sender`, sondern über den Rückgabewert, da das Attribut von der Funktion nur gelesen wird.

Wird nun das Zustandsdiagramm generiert, wobei Funktionen nicht einbezogen werden und Nochange-Semantik angewendet wird, so erhält man folgendes Diagramm:

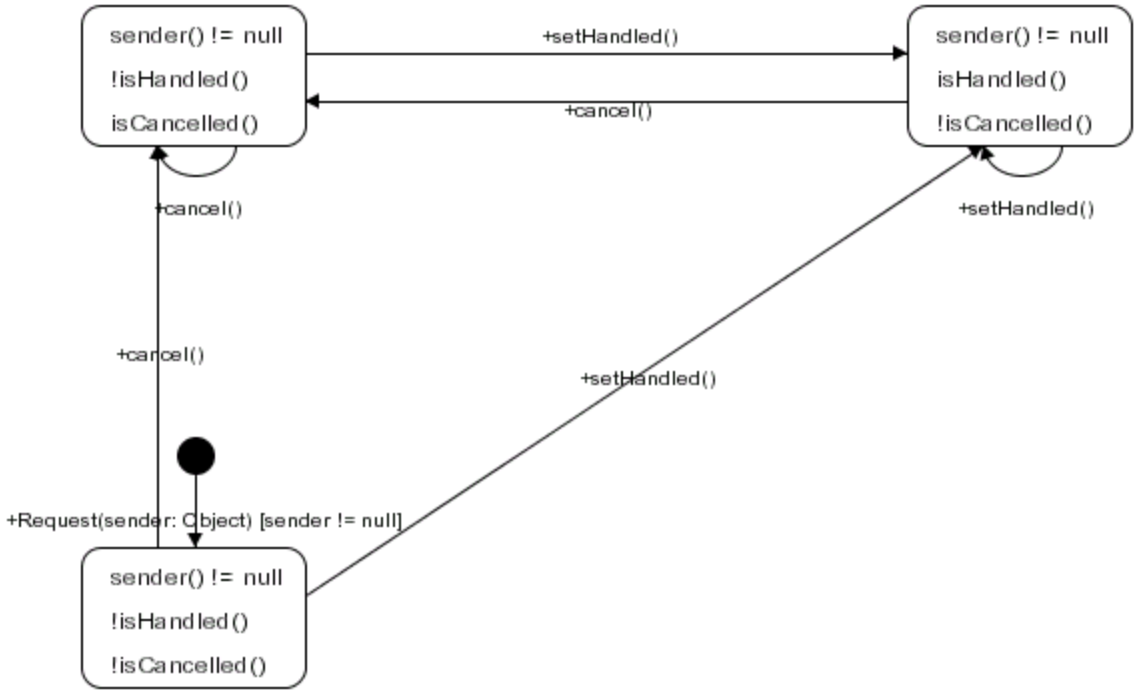


Abbildung 8.1: Zustandsdiagramm der Klasse Request

Dieses Diagramm gibt das Protokoll der Klasse recht übersichtlich wieder. Dabei wird ein konzeptioneller Fehler offenkundig, denn es ist fraglich, ob `setHandled` oder `cancel` noch aufgerufen werden dürfen, wenn `isHandled()` oder `isCancelled()` gilt. Daher werden die entsprechenden Transitionen manuell entfernt, was zu folgendem Diagramm führt:

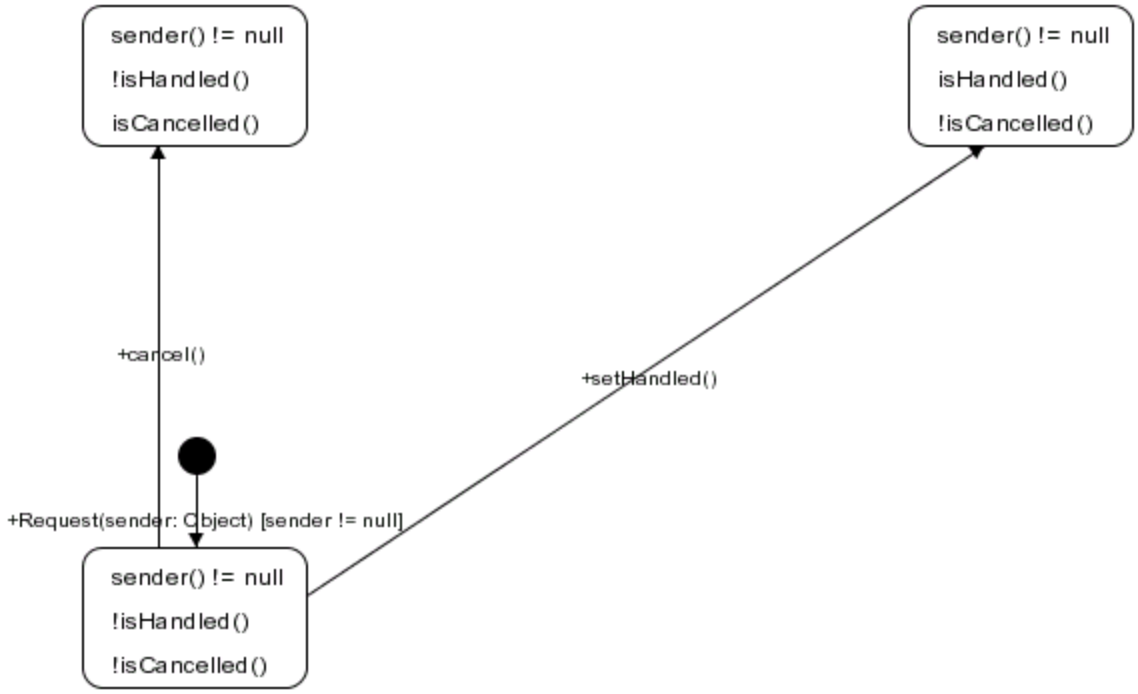


Abbildung 8.2: Geändertes Zustandsdiagramm der Klasse Request

Die Änderungen können nun auf den Quelltext angewendet werden. Die Zusicherungen der Methoden `setHandled` und `cancel` werden dadurch geändert. Die neu generierten Zusicherungen lauten[66]:

```
/**
 * @require !isHandled()
 * @require !isCancelled()
 * @ensure isHandled()
 */
public void setHandled();

/**
 * @require !isHandled()
 * @require !isCancelled()
 * @ensure isCancelled()
 */
public void cancel();
```

Dieses Fallbeispiel zeigt, dass die Verwendung des Statechartgenerators eine gute Hilfe zum Schreiben von konzeptionell korrekten Zusicherungen sein kann.

[66] Die `assert`-Anweisungen werden durch den Statechartgenerator natürlich auch angepasst, falls die entsprechende Option gesetzt ist. Aus Redundanzgründen werden sie hier nicht angeführt.

8.2 Untersuchung der JWAM-Klassen

Das JWAM-Rahmenwerk wurde vollständig auf der Basis des Vertragsmodells entworfen und eignet sich daher hervorragend zur genaueren Untersuchung. In der Version 1.8.0 hat JWAM insgesamt 931 Klassen und Interfaces; darunter fallen aber 273 Testklassen, die in die folgenden Untersuchungen nicht mit einbezogen werden, sodass 658 Klassen und Interfaces übrig bleiben. Es gibt insgesamt 4147 Vor- und 1910 Nachbedingungen, die sich auf 5518 Methoden (davon 2691 Funktionen) und 661 Konstruktoren verteilen. Lässt man die statischen Methoden außen vor, so bleiben 4127 Vor- und 1869 Nachbedingungen bei 5331 Methoden, darunter 2619 Funktionen.

Es fällt auf, dass es keine einzige Invariante gibt, die im Dokumentationskommentar der Klasse (bzw. des Interface) angegeben ist. Sämtliche `@invariant`-Tags beziehen sich lediglich auf ein Attribut, das in seinem Wertebereich eingeschränkt wird. Invarianten werden in JWAM also nur auf einer recht primitiven Ebene eingesetzt.

`@canchange`-Tags kommen in JWAM nicht vor, da sie erst in dieser Arbeit eingeführt wurden.

8.2.1 Anzahl der Prädikate

Zuerst soll untersucht werden, wie viele Zustands-Prädikate die JWAM-Klassen enthalten. Dabei ist zu berücksichtigen, dass durch Fehler in den Zusicherungen vom Statechartgenerator manchmal mehr Zustands-Prädikate erzeugt werden als tatsächlich konzeptionell in der Klasse vorhanden sind. Dies geschieht typischerweise dann, wenn ein Schreibfehler bei einem Parameter-Prädikat vorliegt; dieses wird dann nicht als solches erkannt, sondern als Zustands-Prädikat eingestuft. Detailliertere Angaben bezüglich Fehlern in den Zusicherungen folgen in Abschnitt 8.2.3.

Berücksichtigt man Methoden jeder Sichtbarkeit und bezieht Funktionen mit ein, so ergibt sich folgende Verteilung:

Anzahl Zustands-Prädikate	Klassen absolut	Klassen prozentual
0	402	61,1%
1	131	19,9%
2	53	8,1%
3	32	4,9%
4	18	2,7%
5	5	0,8%
6	4	0,6%
7	4	0,6%
8	1	0,2%
9	2	0,3%
12	1	0,2%
ungültig[67]	5	0,8%

[67] Als ungültig gelten Klassen, die ungültige Prädikate enthalten, die z. B. mehr öffnende als schließende Klammern haben; solche Klassen werden vom Statechartgenerator abgelehnt.

Diese Verteilung ist allerdings aufgrund der oben genannten Gründe verfälscht. Korrigiert man die fehlerhaften Zustands-Prädikate und berücksichtigt man somit nur die Zustands-Prädikate, die tatsächlich als solche beabsichtigt waren, so ergibt sich folgende Verteilung:

Anzahl Zustands-Prädikate	Klassen absolut	Klassen prozentual
0	452	68,7%
1	123	18,7%
2	44	6,7%
3	16	2,4%
4	12	1,8%
5	6	0,9%
6	3	0,5%
8	1	0,2%
9	1	0,2%

Diese Ergebnisse zeigen, dass die meisten Klassen (etwa zwei Drittel) überhaupt keine Zustands-Prädikate haben (ihre Zustandsdiagramme bestehen aus nur einem Zustand) und auch ansonsten die Anzahl der Zustands-Prädikate sehr überschaubar ist. Fast 95 % der Klassen haben zwei oder weniger Zustands-Prädikate und ihre Zustandsdiagramme somit maximal vier Zustände. Die meisten Klassen haben ein einfaches Zustandsmodell, in dem die Operationen in beliebiger Reihenfolge aufgerufen werden dürfen. Das ist durchaus als positiv anzusehen, denn es erleichtert die Benutzung dieser Klassen.

Parameter-Prädikate gibt es insgesamt 5181; bei 5992 nicht-statischen Operationen ist das ein Durchschnitt von 0,86 pro Operation. Es gibt insgesamt 4472 Parameter und damit durchschnittlich 1,16 Parameter-Prädikate pro Parameter. Die Verwendung von Parameter-Prädikaten ist demnach in JWAM sehr gebräuchlich. Der bei weitem häufigste Fall ist eine Vorbedingung, dass ein Referenzparameter ungleich `null` ist[68]; derartige Vorbedingungen sind bei sehr vielen Operationen mit Referenzparametern anzutreffen.

Vor-Prädikate in Nachbedingungen treten in JWAM überhaupt nicht auf. Das ist nicht weiter verwunderlich, weil ihre Verwendung eher ungewöhnlich ist. Sie sind hauptsächlich dazu da, beliebige Zustandsübergänge in Zusicherungen ausdrücken zu können, führen aber zu einer erschwerten Verständlichkeit der Zusicherungen.

In den Zusicherungen gibt es 201 Und-Verknüpfungen[69], 56 Oder-Verknüpfungen und 4086 Negationen. Es gibt keine Exklusiv-Oder-Verknüpfungen. Diese Zahlen zeigen, dass die Prädikate mit fester Belegung (s. Abschnitt 7.3.1) bei weitem überwiegen und dass die Zusicherungen meist einfach aufgebaut sind.

8.2.2 Qualitative Aspekte

Neben den quantitativen Ergebnissen des letzten Abschnitts sollen nun einige qualitative Untersuchungen folgen. Es soll geprüft werden, inwieweit das in dieser Arbeit vorgestellte Verfahren sinn-

[68] 3428 der 5181 Parameter-Prädikate (66,2 %) sind von dieser Art.
[69] Dies sind die explizit angegebenen Und-Verknüpfungen; in den meisten Fällen sind Und-Verknüpfungen jedoch über eine Aufteilung in mehrere einzelne Zusicherungen realisiert.

volle Zustandsdiagramme zu den JWAM-Klassen liefert. Dazu wird auch ein Beispiel herangezogen.

Eine sehr häufige Schwachstelle der JWAM-Zusicherungen ist, dass Konstruktoren keine Nachbedingungen zu bestimmten Zustands-Prädikaten haben. Aufgrund der Canchange-Semantik bei Konstruktoren ist dann deren Belegung beliebig und vom Anfangszustand aus verlaufen Transitionen zu allen Zuständen. Einige Zustände enthalten dann Bedingungen, die in Wahrheit nie gelten, sodass die Zustände gar nicht erreichbar wären, wenn die Nachbedingungen explizit angegeben wären. Das Zustandsdiagramm wird nicht nur unübersichtlich, sondern ist auch größer als notwendig.

Das folgende Zustandsdiagramm der Klasse `de.jwam.handling.toolconstruction.basicconstruction.ToolFpIpImpl` gibt hiervon einen Eindruck:

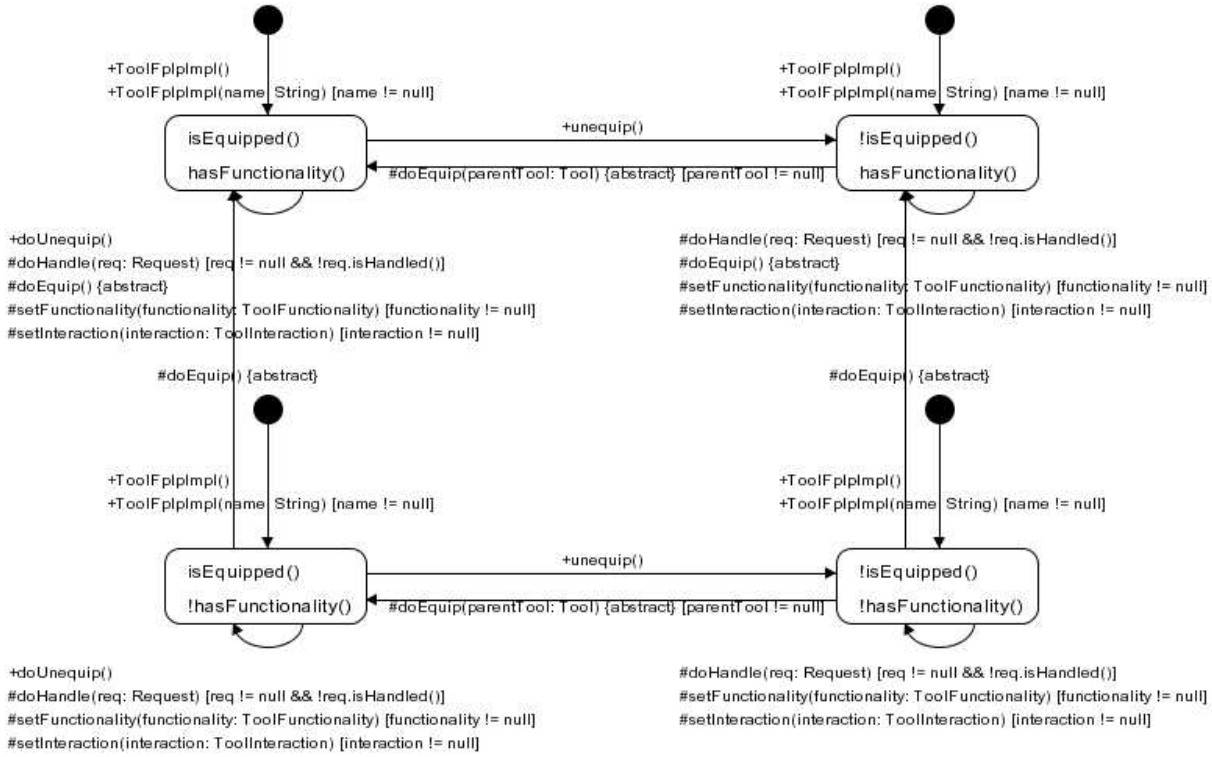


Abbildung 8.3: Zustandsdiagramm der Klasse ToolFpIpImpl

Da nach dem Aufruf eines der beiden Konstruktoren immer `!isEquipped() && !hasFunctionality()` gilt, befindet sich ein Objekt anfangs immer im rechten unteren Zustand. Weil die Konstruktoren aber nichts zusichern, existieren Transitionen vom Anfangszustand aus zu allen Zuständen.

Dieses Diagramm zeigt noch etwas anderes. Wird in einem der beiden unteren Zustände die Methode `setFunctionality` aufgerufen, so bleibt das Objekt laut Zustandsdiagramm in diesem Zustand. Ein Aufruf von `setFunctionality` führt jedoch dazu, dass `hasFunctionality()` anschließend `true` liefert, d. h. das Objekt wäre in einem Zustand, in dem die entsprechende Bedingung nicht mehr gilt. In diesem Fall liegt das daran, dass bei `setFunctionality` die Nachbedingung

`@ensure hasFunctionality()`

vergessen wurde; es kann aber auch sein, dass der Programmierer von der Canchange-Semantik ausgegangen ist, während das obige Diagramm der Nochange-Semantik folgt. In der Canchange-Semantik würden durch `setFunctionality` ausgelöste Transitionen zu allen Zuständen führen. – Ähnliche Probleme ergeben sich für `doUnequip` und die parameterlose Variante von `doEquip` (die zwar `hasFunctionality()` zusichert, aber nicht `isEquipped()`).

Derartige Fälle kommen in den JWAM-Klassen zwar recht häufig vor; es ist aber oft ein Leichtes, die Zustandsdiagramme mithilfe des Statechartgenerators dem gewünschten Verhalten anzupassen und so das Zustandsverhalten exakter zu spezifizieren. Aus den obigen Beispielen ist ersichtlich, dass das intendierte Klassenprotokoll oft einfach zu erkennen ist, sodass die entsprechenden Änderungen leicht durchgeführt werden können.

Schwierig wird es bei Zustandsdiagrammen mit vielen Zuständen. Bei diesen sieht es allerdings oft so aus, dass die meisten Zustands-Prädikate nur bei ganz wenigen Operationen vorkommen (nicht selten nur bei einer einzigen), sodass die Zustandsdiagramme zwar viele Zustände haben (jeder mit einer Selbsttransition mit verschiedenen Ereignissen), aber nur sehr wenige Transitionen zwischen verschiedenen Zuständen, denn oft haben die Operationen, die die Zustands-Prädikate ändern, keine entsprechenden Nachbedingungen. Solche Zustandsdiagramme sind (schon aufgrund ihrer Größe) wenig hilfreich, kommen aber nur recht selten vor, wie schon aus Abschnitt 8.2.1 hervorgeht.

8.2.3 Schreibfehler in den Zusicherungen

Da die Zusicherungen in Kommentaren angegeben werden, findet nirgendwo eine Prüfung auf ihre Korrektheit statt, nicht einmal auf syntaktische Korrektheit. Das führt fast zwangsläufig zu fehlerhaften Zusicherungen, die wiederum zu inkorrekten Zustandsdiagrammen führen. Es sind unterschiedliche Arten von Fehlern möglich, die im Folgenden beschrieben werden, wobei jeweils Beispiele aus dem JWAM-Rahmenwerk angeführt und die Auswirkungen der Fehler beschrieben werden.

1. Schreibfehler bei Zustands-Prädikaten:

```
/**
 * @require hasLock()
 * @require !isLocked()
 * @ensure !hasLock
 * @ensure hasRemovedLock()
 */
public void removeLock();
```

(de.jwam.handling.accesscontrol.LockableThingImpl)

In diesem Beispiel wurden bei der ersten Nachbedingung die Klammern vergessen, was zu einem weiteren Zustands-Prädikat `hasLock` führt, das von `hasLock()` unterschieden wird. Es wird also fälschlicherweise ein neues Zustands-Prädikat eingeführt.

2. Schreibfehler bei Parameter-Prädikaten:

```
/**
 * @require name != null
 * @require default != null
 * @ensure result != null
 */
public String string(String name, String defaultValue);
```

(de.jwam.technology.configuration.Configuration)

Hier wurde in der zweiten Vorbedingung `default` statt `defaultValue` geschrieben, was dazu führt, dass das Prädikat `default != null` nicht als Parameter- sondern als Zustands-Prädikat eingestuft wird. Auch hier wird also ein neues Zustands-Prädikat eingeführt, das in Wirklichkeit keines ist.

Statt eines Parameters kann auch das Schlüsselwort `result` falsch geschrieben bzw. verwechselt worden sein:

```
/**
 * @ensure return != null;
 */
public ThingDescriptionDV thingDescriptionOfWorkspace();
```

(de.jwamx.handling.desktop.DesktopFP)

Auch dies führt zu einem zusätzlichen Zustands-Prädikat, da das Prädikat nicht als Parameter-Prädikat erkannt wird.

3. Schreibfehler beim Namen von Tags:

```
/**
 * @rquire toolPrototype != null
 */
public void setToolPrototype(Tool toolPrototype);
```

(de.jwamalpha.handling.mmi.toolMMI)[70]

Wird das Tag falsch geschrieben, so wird es nicht erkannt und die Zusicherung bleibt unberücksichtigt.

Während die Fehler der dritten Kategorie nicht auf den ersten Blick aus dem Zustandsdiagramm abzulesen sind, fallen Fehler der Kategorien 1 und 2 sofort ins Auge, weil die fälschlicherweise generierten Zustands-Prädikate in sämtlichen Zuständen stehen. Es ist dann sehr leicht, die fehler-

[70] aus JWAM 1.7.0

haften Zusicherungen im Quelltext zu korrigieren, sodass anschließend ein korrektes Zustandsdiagramm erzeugt werden kann. So kann der Statechartgenerator auch dabei helfen, Schreibfehler in den Zusicherungen aufzudecken.

8.3 Einordnung in andere Ansätze

In Kapitel 3 wurden andere Ansätze zur Verwendung von Zustandsdiagrammen in der objektorientierten Softwareentwicklung vorgestellt. Es soll nun versucht werden, den in dieser Arbeit entwickelten Ansatz in diese Ansätze einzuordnen. Dabei ist ein entscheidender Unterschied, dass Zustandsdiagramme in dieser Arbeit lediglich dazu eingesetzt werden, Klassenprotokolle zu beschreiben, während in den anderen Ansätzen das Zustandsverhalten der Objekte modelliert wird.

In den **OMT-Zustandsdiagrammen** werden die Ereignisse als Exemplare von Ereignisklassen modelliert, was zu einer recht umständlichen Implementierung führt; die Implementierungsalternativen, die angegeben werden, führen entweder zu nicht objektorientierten Systemen oder zu übermäßigem Code-Overhead. In dieser Arbeit bilden Aufrufe von Operationen die Ereignisse; die Diagramme werden durch Zusicherungen implementiert, was zu minimalem Code-Overhead führt.

Die **Objectcharts** beschreiben ebenfalls mehr als nur Klassenprotokolle, da an den Zustandsübergängen Methodenaufrufe als Aktionen stehen dürfen. Das Auslösen der Transitionen geschieht durch Methodenaufrufe (wie in dieser Arbeit), aber auch durch das Erreichen einer festgelegten Zeit. – Durch die Angabe von Vor- und Nachbedingungen für Transitionen ist eine gewisse Nähe zum Ansatz in dieser Arbeit gegeben; diese Zusicherungen sind jedoch – wie bereits in Abschnitt 3.2 beschrieben – keine Zusicherungen im Sinne von [Meyer97], da sie nicht die Operationen betreffen, sondern die einzelnen Transitionen, wobei es mehrere Transitionen pro Operation geben kann. In Abschnitt 3.2 wurde auch erläutert, warum sich die Transitionsspezifikationen nicht auf Zusicherungen von Operationen abbilden lassen.

Die **OO-Zustandsdiagramme** von Harel und Gery haben mit den Diagrammen dieser Arbeit den engen Zusammenhang zum zugehörigen Code durch den Fokus auf automatische Codegenerierung gemeinsam, auch wenn sich der Code in dieser Arbeit auf Zusicherungen beschränkt. Die Verarbeitung von Ereignissen erfolgt über die Abarbeitung einer Ereignis-Warteschlange, was wie bei OMT-Zustandsdiagrammen zu einem Code-Overhead führt; daneben werden aber auch Operationsaufrufe als Objektinteraktionen unterstützt. Durch die Verwendung der vollen Mächtigkeit von Zustandsdiagrammen, wie sie in [Harel87] eingeführt wurden, ist recht viel Code zur Umsetzung der Diagramme in eine Implementierung notwendig.

UML-Zustandsdiagramme werden in dieser Arbeit verwendet. Die UML ist aber lediglich eine Modellierungssprache; sie gibt wenig Aufschluss darüber, wie Zustandsdiagramme zu implementieren sind. Immerhin wird in [RJB99, S. 447 f.] darauf hingewiesen, dass Zustandsdiagramme dazu verwendet werden können, Klassenprotokolle zu beschreiben (siehe auch Abschnitt 3.4).

Der wesentliche Unterschied zu den anderen Ansätzen besteht also darin, dass Zustandsdiagramme in dieser Arbeit nicht das Verhalten bzw. die Implementierung von Klassen beschreiben, sondern die Protokolle der Klassen. Es wird spezifiziert, unter welchen Umständen die Operationen aufgerufen werden dürfen (Ursprungszustände und Wächterbedingungen) und welche Bedingungen nach Ausführung der Operationen gelten (Zielzustände und Constraints). Dabei wird von der konkreten Implementierung abstrahiert.

9 Zusammenfassung

In dieser Arbeit wurde der Frage nachgegangen, wie ein Zusammenhang zwischen dem Vertragsmodell und Zustandsdiagrammen hergestellt werden kann. Es wurde festgestellt, dass sich diese beiden Konzepte gegenseitig Nutzen bringen können, d. h. dass eine sorgfältige Wahl der Zusicherungen zu einem wohldefinierten Klassenprotokoll verhilft und dass ein anschauliches Zustandsdiagramm zu robusten Zusicherungen führt. Somit ist es gelungen, eine fruchtbare Verbindung dieser beiden unterschiedlichen Konzepte herzustellen, die in der Softwareentwicklungspraxis von Nutzen ist.

9.1 Zusammenfassung der Vorgehensweisen

Es wurde ein neuer Ansatz vorgestellt, wie Zustandsdiagramme zu Klassen gebildet werden können, die auf der Basis des Vertragsmodells entwickelt wurden. Die Implementierung der Zustandsdiagramme in Klassen geschieht mit minimalem Code-Overhead, da lediglich Programmcode zur Überprüfung der Zusicherungen anfällt. Eine gegenseitige Abbildung ist möglich, d. h. zu jeder Klasse mit explizit angegebenen Zusicherungen existiert ein Zustandsdiagramm und zu jedem Zustandsdiagramm, das den in dieser Arbeit geltenden Einschränkungen genügt[71], existiert eine Schnittstellenbeschreibung mit Zusicherungen. Die gegenseitige Abbildung ist unter Zuhilfenahme logischer Ausdrücke automatisierbar. Die Zusicherungen werden dazu als Verknüpfungen von Prädikaten modelliert, wobei die Prädikate in Zustands- und Parameter-Prädikate eingeteilt werden. Parameter-Prädikate sichern etwas über einen Parameter der betreffenden Operation zu, während Zustands-Prädikate sich auf den Objektzustand beziehen; die Belegungen der Zustands-Prädikate stehen somit in den Zuständen, die Belegungen der Parameter-Prädikate in den Wächterbedingungen und Constraints der Transitionen. In den Zusicherungen kommen die Prädikate als Vor- und Nach-Prädikate vor, wobei diese sich darin unterscheiden, ob die Belegung vor oder nach Ausführung der Operation gemeint ist. In Vorbedingungen kommen ausschließlich Vor-Prädikate vor; in Nachbedingungen stehen hauptsächlich Nach-Prädikate, wobei Vor-Prädikate durch vorangestelltes `old` gekennzeichnet werden. Für Vor-Prädikate in Nachbedingungen gibt es einige Regeln (s. Abschnitt 4.2.2), damit die Nachbedingungen sinnvoll bleiben.

Der vorgestellte Ansatz unterscheidet sich insofern von den bereits etablierten Ansätzen (s. Kapitel 3), als dass die in dieser Arbeit verwendeten Zustandsdiagramme nicht direkt ausführbar sein sollen und kein dynamisches Verhalten spezifizieren. Sie dienen stattdessen der Beschreibung der Klassenprotokolle, legen also fest, welche Operationen in welcher Reihenfolge aufgerufen werden dürfen. Hier ist eine Nähe zur Automatentheorie im Bereich der formalen Sprachen [Schöning01] erkennbar, wo Automaten zum Akzeptieren von Eingabewörtern verwendet werden. Eine Aufrufreihenfolge kann als ein solches Wort angesehen werden, das vom Zustandsdiagramm genau dann akzeptiert wird, wenn sie laut Klassenprotokoll erlaubt ist. Da Zustandsdiagramme in der hier verwendeten Form – bis auf Wächterbedingungen und Constraints – endlichen Automaten entsprechen[72], ist die Sprache der möglichen Aufrufreihenfolgen regulär (sieht man von den Methodenparametern ab).

Da das Vertragsmodell ursprünglich nicht dazu gedacht war, Zustände zu beschreiben, war die Einführung der Canchange- und der Nochange-Semantik notwendig. Die verwendete Semantik

[71] Verzicht auf End-, Gedächtnis-, Synchronisationszustände und Aktionen sowie Erwähnung sämtlicher Zustands-Prädikate in allen Zuständen

[72] Diagramme mit Zustandshierarchien lassen sich immer in flache Zustandsdiagramme umformen, siehe Abschnitt 6.2.1.

definiert, was mit Zustands-Prädikaten geschieht, deren Belegungen in den Nachbedingungen einer Operation nicht explizit angegeben sind. In der Canchange-Semantik ist die Belegung nach Ausführung der Operation unbestimmt, während sie in der Nochange-Semantik von der Operation nicht geändert wird. Bei Konstruktoren gilt die Canchange-Semantik generell, während bei Methoden eher die Nochange-Semantik verwendet werden sollte um zu viele Transitionen zu vermeiden. Mithilfe des neuen `@canchange`-Tags kann die Canchange-Semantik für ein bestimmtes Zustands-Prädikat außerdem explizit eingeschaltet werden. – Bei der Nochange-Semantik können sich allerdings in zwei Bereichen Probleme ergeben, nämlich bei Verwendung von Invarianten und bei Vererbung. Durch Invarianten kann es dazu kommen, dass es keine Transitionen mehr zu bestimmten Methoden gibt, weil die möglichen Zielzustände nicht der Invariante genügen. Abhilfe schafft hier die Einführung von zusätzlichen Nachbedingungen. Bei Vererbung können durch das Verschärfen von Nachbedingungen neue Transitionen entstehen, sodass eine Unterklasse Zielzustände einnehmen kann, die laut Oberklasse nicht möglich sind. Beim Benutzen der Klasse muss daher die Canchange-Semantik zugrunde gelegt werden.

Die Untersuchung an real existierenden Klassen hat ergeben, dass die Zustandsdiagramme zu den meisten Klassen recht klein sind. Viele Klassen haben keine Zustands-Prädikate und somit nur einen einzigen Zustand. Sehr wenige Klassen haben mehr als zehn Zustände. Die Zustandsdiagramme sind also überschaubar und größtenteils auch leicht lesbar.

Ähnliches gilt für die generierten Zusicherungen zu vorgegebenen Diagrammen. Allerdings kann hier bei den Nachbedingungen leicht eine gewisse Komplexität entstehen, weil die im ersten Schritt generierten Nachbedingungen recht lang sind. Eine Vereinfachung ist nicht immer möglich. Die Vorbedingungen dagegen sind im Allgemeinen leicht lesbar.

9.2 Ergebnisse für die Softwareentwicklungspraxis

Im Zusammenhang mit dieser Arbeit entstand der Statechartgenerator, der die gegenseitige Abbildung von Zusicherungen und Zustandsdiagrammen automatisch vornimmt und somit bei der Softwareentwicklung eingesetzt werden kann. Die Programmierung einer Klasse sollte nun folgendermaßen vor sich gehen:

Man beginnt wie gewohnt mit der Programmierung der Klasse. Zu den Operationen schreibt man Vorbedingungen (die Bedingungen, die die Operation erwartet) und Nachbedingungen (die Bedingungen, die nach Ausführung der Operation gültig sind); eventuell formuliert man noch eine Klasseninvariante. Die Implementierung der Operationen braucht an dieser Stelle noch nicht zu geschehen; es genügt, wenn die Zusicherungen formuliert sind. Mit ein wenig Erfahrung wird man schnell abschätzen können, wie viele Zustands-Prädikate die Klasse in etwa enthält. Bei Klassen ohne Zustands-Prädikate ist das Zustandsdiagramm uninteressant und braucht nicht generiert zu werden. In den meisten Fällen trifft dies auch zu.[73] Je mehr Zustands-Prädikate die Klasse jedoch enthält, desto wichtiger ist ein Blick auf das zugehörige Zustandsdiagramm. Dieses sollte zunächst in der Nochange-Semantik generiert werden um grobe Fehler ausfindig zu machen. Beispielsweise könnte man feststellen, dass eine Transition einen falschen Zielzustand besitzt, weil eine Nachbedingung vergessen wurde. Man hat nun entweder die Möglichkeit, dies am Diagramm zu korrigieren und daraufhin die Zusicherungen neu zu erzeugen, oder die fehlende Nachbedingung selbst einzugeben, falls der Fehler offensichtlich ist; das Diagramm kann anschließend neu generiert werden. In Abschnitt 8.1 wurde diese Vorgehensweise bereits an einem Fallbeispiel verdeutlicht.

[73] Der Statechartgenerator kann hier trotzdem nützlich sein, weil er den Code zur Überprüfung der Zusicherungen automatisch einfügen kann; dies braucht dann nicht mehr per Hand zu geschehen.

Anders sieht es aus, wenn man das Zustandsdiagramm einer Klasse betrachtet, die man *benutzt*, d. h. eine Operation aufruft. Hier hat man wie üblich dafür Sorge zu tragen, dass das Vertragsmodell eingehalten wird. Im Regelfall wird dies auch genügen. Das Zustandsdiagramm der verwendeten Klasse zu betrachten lohnt sich nur, wenn es ein komplexeres Diagramm ist und man wissen möchte, welche Bedingungen zu welchem Zeitpunkt gelten. Zu diesem Zweck muss das Zustandsdiagramm in der Canchange-Semantik generiert werden, weil sonst in Unterklassen neue Zielzustände hinzukommen könnten und auch die Operationen selbst nicht an die Nochange-Semantik gebunden sind (d. h. sie könnten weniger zusichern als tatsächlich gilt). Das Untersuchen des Zustandsdiagramms kann auch nützlich sein, wenn man lediglich das Klassenprotokoll verstehen möchte. In diesem Fall reicht oft die Nochange-Semantik aus; die Canchange-Semantik würde durch zu viele Transitionen zu Verwirrung führen.

Die obigen Ausführungen ergeben, dass man sich bei der Wahl der Änderungssemantik an folgende Faustregel halten kann: Betrachtet man die Klasse für sich allein, so benutze man die Nochange-Semantik, betrachtet man sie aus der Sicht eines Klienten, so benutze man die Canchange-Semantik. Bei Funktionen sollte man grundsätzlich die Nochange-Semantik verwenden, falls man sie in das Zustandsdiagramm einbezieht; als Rechtfertigung hierfür sollte man auf zustandsändernde Funktionen verzichten.

Dass die Verwendung des Statechartgenerators bei der Entwicklung von JWAM eine große Bereicherung gewesen wäre, zeigen die Untersuchungen aus Abschnitt 8.2. Viele fehlerhafte Zusicherungen und ein undurchsichtiges Protokoll diverser Klassen hätten dadurch verhindert werden können. Der Statechartgenerator ist also gleich in doppelter Hinsicht nützlich: Er verhilft sowohl zu korrekten Zusicherungen als auch zu robusten Klassenprotokollen.

9.3 Grenzen und Schwachstellen des Verfahrens

Das in dieser Arbeit entwickelte Verfahren hat auch seine Grenzen. In diesem Abschnitt soll untersucht werden, wo die Grenzen und Schwachstellen in Bezug auf eine gegenseitige Abbildung von Zusicherungen und Zustandsdiagrammen sind. Dieser Abschnitt soll gleichzeitig als Ausblick dienen, was in diesem Gebiet noch an Forschung möglich ist.

Folgende Punkte sind zu nennen:

1. **Zusicherungen müssen explizit angegeben werden, wenn sie gelten.** Dies ist nicht ganz konform mit der Idee des Vertragsmodells. Zumindest bei Vorbedingungen ist es jedoch erforderlich, denn eine Operation muss explizit als Vorbedingung angeben, was sie voraussetzt. In der Formulierung ihrer Nachbedingung hat eine Operation laut Vertragsmodell allerdings alle Freiheiten, denn sie kann selbst bestimmen, was sie zusichert. So spricht aus der Sicht des Vertragsmodells nichts dagegen, dass alle Operationen lediglich die Nachbedingung `true` haben. – Für das in dieser Arbeit entwickelte Verfahren ist es aber notwendig, dass eine Operation genau das zusichert, was tatsächlich gilt. Ansonsten hängt das Ergebnis von der Änderungssemantik ab: In der Nochange-Semantik kann es passieren, dass die Operation ein Zustands-Prädikat ändert, ohne dass es im Zustandsdiagramm eine entsprechende Transition gibt[74]; in der Canchange-Semantik kann dies zwar nicht passieren, allerdings gibt es dann viel zu viele Transitionen zu Zielzuständen, die von dieser Operation nie tatsächlich eingenommen werden.

[74] siehe das Beispiel aus Abschnitt 8.2.2

2. **Ähnliche Prädikate, die zueinander in Beziehung stehen, werden nicht als zusammenhängend erkannt.** Beispielsweise werden `o!=null` und `o==null` als zwei voneinander unabhängige Prädikate betrachtet und nicht beachtet, dass `o!=null` immer dieselbe Belegung hat wie `!(o==null)`. Auch die Prädikate `x>0` und `x<0` werden als unabhängig angesehen und nicht beachtet, dass ein Zustand, in dem `x>0 && x<0` gilt, in der Praxis nie eingenommen werden kann. Um diesem Problem Abhilfe zu verschaffen müssten Beziehungen zwischen den Prädikaten hergestellt werden.

3. **Bislang existiert keine Unterstützung für die Interoperabilität zwischen verschiedenen Klassen.** Das Protokoll einer Klasse wird für sich alleine betrachtet, jedoch nicht im Zusammenspiel mit anderen Klassen, die zueinander in einer Benutzt-Beziehung stehen. In diesem Zusammenhang kann es hinderlich sein, dass Parameter-Prädikate nicht zur Beschreibung von Zuständen verwendet werden, da die Kommunikation zwischen verschiedenen Objekten über Parameter verläuft. Wollte man allerdings Zustände mit Parameter-Prädikaten definieren, so wären die resultierenden Zustandsdiagramme weitaus komplexer.

4. **Oft werden lange Nachbedingungen mit vielen Vor-Prädikaten generiert.** Das liegt daran, dass sich viele Nachbedingungen nicht deutlich vereinfachen lassen. Die Folge ist, dass häufig viele Vor-Prädikate in Nachbedingungen stehen bleiben. Die Verwendung von Vor-Prädikaten in Nachbedingungen ist unüblich (vgl. Abschnitt 8.2.1) und erschwert die Lesbarkeit. Ohne Vor-Prädikate lassen sich jedoch nicht alle Zustandsdiagramme auf Zusicherungen abbilden.

5. **Einige Elemente von Zustandsdiagrammen werden nicht unterstützt.** Dies wurde bereits in Abschnitt 4.2.9 beschrieben. Dort wurde auch erläutert, warum diese Elemente im Zusammenhang mit Zusicherungen wegen einiger Einschränkungen keinen Sinn machen.

6. **Die Klassenprotokolle können nicht dynamisch verändert werden.** Sie liegen statisch vor und können zur Laufzeit nicht mehr angepasst werden. Hierzu sind allerdings Zustandsdiagramme nicht das richtige Modellierungswerkzeug, da dort eine dynamische Anpassung nicht vorgesehen ist. Möchte man eine dynamische Änderung von Klassenprotokollen modellieren, so muss man entweder die Semantik von Zustandsdiagrammen erweitern oder auf andere Modelle zurückgreifen.[75]

Trotz dieser Einschränkungen ist zu sagen, dass das hier entwickelte Verfahren bereits recht ausgereift ist und für viele realitätsnahe Klassen eine brauchbare Abbildung von Zusicherungen und Zustandsdiagrammen liefert. Bei der Softwareentwicklung kann es eine große Hilfe sein, die Protokolle der entworfenen Klassen zu überprüfen und gegebenenfalls anzupassen.

[75] In [LLMT00a] und [LLMT00b] wird z. B. auf eine Variante von Petrinetzen zurückgegriffen.

Literaturverzeichnis

[AB01] W. M. P. van der Aalst, T. Basten: Identifying Commonalities and Differences in Object Life Cycles using Behavioral Inheritance. In: J. M. Colom, M. Koutny [Hrsg.]: *Application and Theory of Petri Nets 2001*, Lecture Notes in Computer Science, Vol. 2075, S. 32–52. Berlin: Springer-Verlag, 2001

[AHT02] W. M. P. van der Aalst, K. M. van Hee, R. A. van der Toorn: Component-Based Software Architectures: A Framework Based on Inheritance of Behavior. *Science of Computer Programming* 42 (2–3), S. 129–171, 2002

[BA01] T. Basten, W. M. P. van der Aalst: Inheritance of Behavior. *Journal of Logic and Algebraic Programming* 47(2), S. 47–145, 2001

[Baumgarten96] B. Baumgarten: *Petri-Netze: Grundlagen und Anwendungen*. Heidelberg [u. a.]: Spektrum, Akademischer Verlag, 2. Auflage, 1996

[Beeck94] M. von der Beeck: A Comparison of Statecharts Variants. In: H. Langmaack, W.-P. de Roever, J. Vytopil [Hrsg.]: *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lecture Notes in Computer Science, Vol. 863, S. 128–148. Berlin; Heidelberg: Springer-Verlag, 1994

[Booch94] G. Booch: *Object-oriented analysis and design with applications*. Redwood City, California: Benjamin/Cummings, Second Edition, 1994

[BRJ99] G. Booch, J. Rumbaugh, I. Jacobson: *The Unified Modeling Language User Guide*. Reading, Massachusetts: Addison-Wesley, 1999

[CHB92] D. Coleman, F. Hayes, S. Bear: Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design. *IEEE Transactions on Software Engineering* 18(1), S. 9–18, Januar 1992

[Daugherty98] G. Daugherty: Unification of the Models for Types, Classes and State Machines. In: J. Bosch, S. Mitchell [Hrsg.]: *Object-Oriented Technology*, ECOOP ’97 Workshop Reader, Lecture Notes in Computer Science, Vol. 1357, S. 169–172. Berlin; Heidelberg; New York: Springer-Verlag, 1998

[DeMarco78] T. DeMarco: *Structured Analysis and System Specification*. New York: Yourdon Press, 1978

[Dijkstra75] E. Dijkstra: Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM*, Vol. 18, S. 453–457, 1975

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison-Wesley, 1995
(deutsch: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Bonn: Addison-Wesley, 1996)

[GHW85] J. Guttag, J. Horning, J. Wing: The Larch Family of Specification Languages. *IEEE Software* 2(5), S. 24–36, September 1985

[GJSB00] J. Gosling, B. Joy, G. Steele, G. Bracha: *The Java Language Specification*. Boston, Massachusetts [u. a.]: Addison-Wesley, Second Edition, 2000

[Gries81] D. Gries: *The Science of Programming*. New York: Springer-Verlag, 1981

[Harel87] D. Harel: Statecharts: A visual formalism for complex systems. *Science of Computer Programming* 8, S. 231–274, 1987

[Harel97] D. Harel: Some Thoughts on Statecharts, 13 Years Later. In: O. Grumberg [Hrsg.]: *Computer Aided Verification*, Lecture Notes in Computer Science, Vol. 1254, S. 226–231. Berlin; Heidelberg; New York: Springer-Verlag, 1997

[Heyden00] O. Heyden: *Ausführung von UML-Zustandsdiagrammen durch Referenz-Netze*. Studienarbeit, Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Verteilte Systeme, 2000

[Heyden01] O. Heyden: *Konzepte zur automatisierten Abbildung von UML-Entwurfsmodellen in eine ausführbare Implementierung am Beispiel einer eCommerce-Businesslogik*. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Verteilte Systeme, September 2001

[HG96] D. Harel, E. Gery: Executable Object Modeling with Statecharts. *18th International Conference on Software Engineering*, S. 246–257, IEEE Computer Society Press, 1996. Auch in: *Computer* 30(7), S. 31–42, Juli 1997

[Hoare69] C. A. R. Hoare: An Axiomatic Basis for Computer Programming. *Communications of the ACM*, Vol. 12, S. 576–580/583, 1969

[HT00] A. Hunt, D. Thomas: *The Pragmatic Programmer*. Reading, Massachusetts: Addison-Wesley, 2000

[Jacobson92] I. Jacobson et al.: *Object-Oriented Software Engineering: A Use Case Driven Approach*. Reading, Massachusetts: Addison-Wesley, 1992

[JBR99] I. Jacobson, G. Booch, J. Rumbaugh: *The Unified Software Development Process*. Reading, Massachusetts: Addison-Wesley, 1999

[Köhler00] H.-J. Köhler: *Codegenerierung für UML-Collaborations-, -Sequenz- und -Statechart-Diagramme*. Diplomarbeit, Universität-Gesamthochschule Paderborn, Fachbereich Informatik, 2000

[KS94] G. Kappel, M. Schrefl: Inheritance of Object Behavior – Consistent Extensions of Object Life Cycles. *Proceedings of the 2nd Int. East/West Database Workshop Series in Computer Science*, S. 289–300, 1994

[LL98] A. Laue, M. Liedtke: *Eine Einführung in Statecharts vor dem Hintergrund der objektorientierten Anwendungsentwicklung*. Studienarbeit, Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, November 1998

[LLMT00a] A. Laue, M. Liedtke, D. Moldt, I. Tričković: Statecharts as Protocols for Objects. In: *Proceedings of the Third Workshop on Rigorous Object-oriented Methods*. University of York, 2000

[LLMT00b] A. Laue, M. Liedtke, D. Moldt, I. Tričković: Modeling Intra- and Inter-Object Control Using Reference Nets. In: J. Ebert, U. Frank [Hrsg.]: *Modelle und Modellierungssprachen in Informatik und Wirtschaftsinformatik: Beiträge des Workshops „Modellierung 2000“*, St. Goar, 5. – 7. April 2000 (Band 15 von Koblenzer Schriften zur Informatik), S. 89–102. Koblenz: Fölbach, 2000

[LW93] B. Liskov, J. Wing: Specifications and Their Use in Defining Subtypes. *ACM SIGPLAN Notices* 28(10), S. 16–28, 1993

[LW94] B. Liskov, J. Wing: A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems* 16(6), S. 1811–1841, November 1994

[MD93] J. McGregor, D. Dyer: A Note on Inheritance and State Machines. *ACM SIGSOFT Software Engineering Notes* 18(4), S. 61–69, Oktober 1993

[Meyer88] B. Meyer: *Object-oriented Software Construction*. Englewood Cliffs, New Jersey: Prentice-Hall, First Edition, 1988
(deutsch: *Objektorientierte Softwareentwicklung*. Wien: Carl Hanser Verlag; London: Prentice-Hall, 1990)

[Meyer92a] B. Meyer: Design by Contract. In: D. Mandrioli, B. Meyer [Hrsg.]: *Advances in Object-Oriented Software Engineering*, S. 1–50. New York; London: Prentice-Hall, 1992

[Meyer92b] B. Meyer: Applying “Design by Contract”. *Computer* 25(10), S. 40–51, Oktober 1992

[Meyer92c] B. Meyer: *Eiffel: The Language*. New York [u. a.]: Prentice-Hall, 1992

[Meyer97] B. Meyer: *Object-oriented Software Construction*. Upper Saddle River, New Jersey: Prentice-Hall, Second Edition, 1997

[Oestereich01] B. Oestereich: *Objektorientierte Softwareentwicklung: Analyse und Design mit der Unified Modeling Language*. München; Wien: Oldenbourg, 5. Auflage, 2001

[PR94] B. Paech, B. Rumpe: A new Concept of Refinement used for Behaviour Modelling with Automata. In: M. Naftalin, T. Denvir, M. Bertran [Hrsg.]: *FME ’94: Industrial Benefit of Formal Methods*, Lecture Notes in Computer Science, Vol. 873, S. 154–174. Berlin; Heidelberg; New York: Springer-Verlag, 1994

[RBPEL91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen: *Object-Oriented Modeling and Design*. Englewood Cliffs, New Jersey: Prentice-Hall, 1991
(deutsch: *Objektorientiertes Modellieren und Entwerfen*. München; Wien: Carl Hanser Verlag; London: Prentice-Hall, 1993)

[Reisig86] W. Reisig: *Petrinetze: Eine Einführung*. Berlin: Springer-Verlag, 2. Auflage, 1986

[RJB99] J. Rumbaugh, I. Jacobson, G. Booch: *The Unified Modeling Language Reference Manual*. Reading, Massachusetts: Addison-Wesley, 1999

[Rumbaugh95] J. Rumbaugh: OMT : The dynamic model. *Journal of Object-Oriented Programming* 7(9), S. 6–12, Februar 1995

[Schöning00] U. Schöning: *Logik für Informatiker*. Heidelberg; Berlin: Spektrum, Akademischer Verlag, 5. Auflage, 2000

[Schöning01] U. Schöning: *Theoretische Informatik – kurzgefasst*. Heidelberg; Berlin: Spektrum, Akademischer Verlag, 4. Auflage, 2001

[SK98] T. Systä, K. Koskimies: Extracting State Diagrams from Legacy Systems. In: J. Bosch, S. Mitchell [Hrsg.]: *Object-Oriented Technology*, ECOOP ’97 Workshop Reader, Lecture Notes in Computer Science, Vol. 1357, S. 272–273. Berlin; Heidelberg; New York: Springer-Verlag, 1998

[SM92] S. Shlaer, S. Mellor: *Object Lifecycles: Modeling the World in States*. Englewood Cliffs, New Jersey: Prentice-Hall, 1992

[Turau96] V. Turau: *Algorithmische Graphentheorie*. Bonn; Paris; Reading, Massachusetts: Addison-Wesley, 1996

[Witt01] M. Witt: *Automatische Generierung von UML-Zustandsdiagrammen aus Java-Klassen*. Studienarbeit, Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, 2001

[Züllighoven98] H. Züllighoven et al.: *Das objektorientierte Konstruktionshandbuch nach dem Werkzeug & Material-Ansatz*. Heidelberg: dpunkt-Verlag, 1998

URL-Verzeichnis

[URL1] `http://www.jwam.de`

JWAM ist ein Rahmenwerk für objektorientierte Anwendungssoftware in Java nach dem WAM-Ansatz (Werkzeug-Automat-Material, siehe [Züllighoven98]).

[URL2] `http://www.omg.org/cgi-bin/doc?formal/03-03-01.pdf`

Dies ist die Spezifikation der Unified Modeling Language in der Version 1.5.

[URL3] `http://www.ilogix.com/products/rhapsody`

Rhapsody ist ein von David Harel mitentwickeltes Werkzeug zum Erstellen von Zustandsdiagrammen inklusive automatischer Codegenerierung.

[URL4] `http://www.borland.com/jbuilder`

Der JBuilder ist eine integrierte Entwicklungsumgebung für Java.

[URL5] `http://www.eclipse.org`

Eclipse ist eine erweiterbare und sehr flexible integrierte Entwicklungsumgebung.

Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig durchgeführt zu haben und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Hamburg, im April 2003 Matthias Witt