7

Modellbasierte Tests

Testen ist eine extrem kreative und
intellektuell herausfordernde Aufgabe.
Glenford J. Myers, [Mye01]

Auf Basis des vorangegangenen Kapitels 6 stehen in diesem Kapitel praktische Fragestellungen zur Umsetzung von Tests mit der UML im Vordergrund. Dabei wird demonstriert, wie unter Nutzung der UML/P effizient Testfälle definiert werden und welche Diagrammarten sich dafür eignen.

B.Rumpe, *Agile Modellierung mit UML*, 2. Aufl., Xpert.press
DOI 10.1007/978-3-642-22430-0_7, © Springer-Verlag Berlin Heidelberg 2012

Nach der Einführung der beim Testen eingesetzten Terminologie und der Diskussion der Probleme und ihrer Lösungsansätze unter Nutzung von Modellen wird in diesem Kapitel auf die konkreten Ansätze zum Einsatz der Notationen der UML/P zur Definition und Entwicklung von Testfällen eingegangen.

Die Definition von Testfällen mit der UML/P basiert auf der Umsetzung der Diagramme und der OCL in Codeteile in Kapitel 5, die zu Testfällen und Konsistenzprüfungen zusammengesetzt werden. Darauf aufbauend wird die Testfallmodellierung mit Objektdiagrammen in Abschnitt 7.1, mit OCL-Invarianten in Abschnitt 7.2, mit Methodenspezifikationen in Abschnitt 7.3, mit Sequenzdiagrammen in Abschnitt 7.4 und mit Statecharts in Abschnitt 7.5 anhand von Beispielen demonstriert.

7.1 Testdaten und Sollergebnis mit Objektdiagrammen

Abbildung 7.1 zeigt den sich aus der in Abbildung 6.7 ergebenden Einsatz von zwei Objektdiagrammen zur Darstellung der Testdaten und des Sollergebnisses. Damit ein Objektdiagramm für mehrere Testfälle verwendet werden kann, sind gegebenenfalls kleinere Anpassungen für einen spezifischen Testfall notwendig. Deshalb ist es möglich, nach Aufbau der entsprechenden Objektstruktur zusätzlich Java-Code einzubauen.

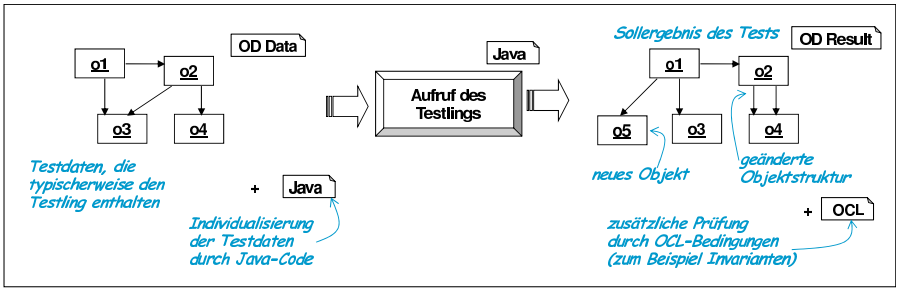


Abbildung 7.1. Standardform des Tests mit Objektdiagrammen, OCL und Java-Testtreiber

Nach Ausführung des Testlings steht die Istdatenstruktur zur Verfügung und kann mit der durch das zweite Objektdiagramm vorgegebenen Solldatenstruktur verglichen werden. Zusätzliche Eigenschaften können durch OCL-Bedingungen geprüft werden. Dies können allgemein gültige OCL-Invarianten sein, die immer geprüft werden sollten, aber auch für den Test spezifische Bedingungen.

Einer der typischen im Auktionssystem verwendeten Testdatensätze besteht aus einem Objekt der Klasse `AllData`, mehreren Auktionen mit allen

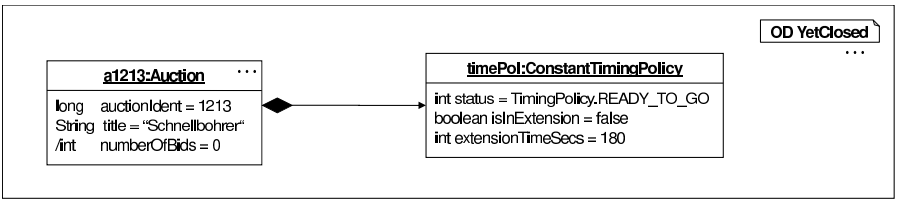


Abbildung 7.2. Testdatensatz als Objektdiagramm

davon abhängigen Objekten, mehreren Personen, die in unterschiedlichen Situationen angemeldet sind, und einer Reihe von Nachrichten, die einigen offenen und abgelaufenen Auktionen zugeordnet sind. Von diesen Grundstrukturen sind nur relativ wenige Datensätze notwendig, denn durch Anpassung mit zusätzlichem Java-Code lassen sich diese für viele Testfälle wiederverwenden.

Auf dem in Abbildung 7.2 dargestellten Testdatensatz wird der Effekt der Methode zum Eröffnen einer Auktion `start()` durch das Sollergebnis (ausschnittsweise) in Abbildung 7.3 beschrieben.

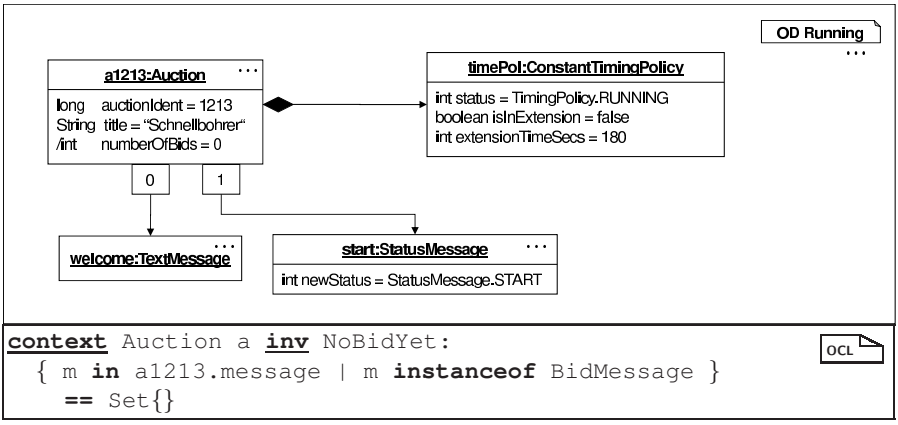


Abbildung 7.3. Sollergebnis als Objektdiagramm

Neben der als `NoBidYet` angegebenen OCL-Bedingung, die besagt, dass nach der Eröffnung noch kein Gebot abgegeben ist, existieren auch einzuhaltende Invarianten. Beispielsweise ist `Bidders1` für Auktionen allgemeingültig:

```
context Auction a inv Bidders1:
  a.activeParticipants <= a.bidder.size
```

OCL

und muss natürlich auch nach der Eröffnung einer Auktion gelten. Wird eine Codegenerierung für die beschriebenen Diagramme nach Kapitel 5

zugrunde gelegt, so kann ein Test mit der in Anhang B, Band 1 beschriebenen Java/P-Erweiterung wie folgt formuliert werden:

```
testStart() {                                                    Java/P
  Auktion a1213 = setupYetClosed();        // Testdaten erzeugen
  a1213.start();                           // Test ausführen
  ocl isStructuredAsRunning(a1213);        // Sollergebnis erfüllt?
  ocl checkNoBidYet(a1213);                // Eigenschaft NoBidYet
  ocl checkBidders1(a1213);                // Invariante Bidders1
  ocl checkTime1(a1213);                   // weitere Invariante Time1
}
```

Die UML/P bietet eine eigenständige Dokumentart an, mit der diese Tests kompakter formuliert werden können:

```
test Auction.start() {                                           Test
  testdata: OD YetClosed;
  driver:   a1213.start();
  assert:   OD Running; inv NoBidYet; inv Bidders1; inv Time1;
}
```

Um die Fähigkeiten der beiden oben beschriebenen Frameworks JUnit und VUnit richtig zu nutzen, ist allerdings eine angepasste Codegenerierung für die prädikative Abfrage `isStructuredAs` auf Basis von Objektdiagrammen und `check` für OCL-Bedingungen sinnvoll. Dabei entsteht nicht nur eine boolesche Aussage, sondern im Fehlerfall eine bei JUnit übliche Beschreibung des Istergebnisses, die idealerweise den Namen der verletzten OCL-Bedingung beziehungsweise Namen und Inhalt des vom Sollergebnis abweichenden Objekts und Attributs beschreibt.

In Abbildung 7.3 wurde das Sollergebnis ebenso detailliert dargestellt wie die Testdaten. Dies muss im Allgemeinen nicht der Fall sein. Während das Objektdiagramm zur Bereitstellung der Testdaten eine gewisse Vollständigkeit benötigt, um damit konstruktiv Objektstrukturen bilden zu können, muss das Sollergebnis nur den Teil der Objektstruktur darstellen, der für den behandelten Testfall von Interesse ist. Dabei können einzelne Attribute ausgelassen aber auch abgeleitete Attribute eingesetzt werden. Fehlt im Sollergebnis eine Teilstruktur der Objekte, so bedeutet das entsprechend der Semantik eines Objektdiagramms nicht, dass diese im Istergebnis zu löschen war, sondern dass ihre tatsächliche Form nicht von Interesse ist.

Durch ein Hinzufügen des Stereotyps «complete» aus Tabelle 5.15 kann gefordert werden, dass das Objektdiagramm als vollständige Beschreibung einer Objektstruktur inklusive aller Links verstanden wird. Der Vergleich ist dann, wie in Abschnitt 5.2 beschrieben, wesentlich restriktiver. Allerdings müssen in diesem Objektdiagramm auch alle Attributwerte angegeben werden.

Die in Abschnitt 4.3, Band 1 diskutierte Kombinierbarkeit von Objektdiagrammen, die durch OCL-Logikoperatoren gesteuert wird, kann bei der

Modellierung von Testdaten und von Sollergebnissen hilfreich zur Modularisierung eingesetzt werden. Nach dem in Abschnitt 5.2 diskutierten Verfahren können Objektdiagramme kombiniert und damit die Gesamtstruktur der Testdaten aus mehreren Einzeldiagrammen zusammengesetzt werden. Bei der Definition des Sollergebnisses kann die in Kapitel 4, Band 1 diskutierte, sehr flexible Kombinierbarkeit von Objektdiagrammen mithilfe der OCL-Operatoren genutzt werden. Dabei kann zum Beispiel mit negierten Objektdiagrammen geprüft werden, dass eine bestimmte Situation nicht auftritt.

Der hier vorgeschlagene Weg zur Modellierung von Testfällen mit der UML findet sich in Ansätzen auch in [CCD02] wieder, wo ebenfalls Objektdiagramme zur Modellierung von Testdaten eingesetzt werden. Dort werden mehrere Stereotypen vergeben, um im Objektdiagramm direkt zu markieren, wofür es eingesetzt werden soll, wodurch jedoch die Wiederverwendbarkeit für verschiedene Zwecke sinkt.

Die Verwendung von Objektdiagrammen wird in Kapitel 8 anhand von Testmustern detailliert demonstriert.

7.2 Invarianten als Codeinstrumentierungen

Einer der Nachteile einer Testfallmodellierung im Stil der Abbildung 6.7 ist die fehlende Möglichkeit, während des Testablaufs Invarianten zu prüfen. Wird zum Beispiel ein komplexer Algorithmus bearbeitet, so kann es sinnvoll sein, statt nur das Ergebnis zu prüfen und damit auf interne Zwischenzustände rückschließen zu müssen, direkt die zwischendurch geltenden Eigenschaften zu formulieren und zu prüfen. Zu diesem Zweck wurde in Anhang B, Band 1 die bereits mehrfach genutzte `ocl`-Anweisung mit einer OCL-Bedingung als zu prüfende Zusicherung eingeführt. Ergänzt wird diese durch eine `let`-Anweisung zur Definition lokaler Variablen, die in späteren OCL-Invarianten verwendet werden können, um damit auf frühere Zustände zurückzugreifen.[1]

Das Beispiel in Abbildung 7.4 demonstriert die Verwendung von `ocl`- und `let`-Anweisungen anhand einer Methode der Klasse `Auction` zur Versendung von Nachrichten.

Eine `ocl`-Anweisung ist, wie in Abbildung 7.4 gezeigt, mit einer OCL-Bedingung oder einer Referenz auf eine benannte OCL-Invariante versehen. Im Beispiel wurde auf die folgende OCL-Invariante Bezug genommen, die eine allgemeine Beziehung zwischen einer Auktion und der eine Nachricht empfangenden Person beschreibt:

[1] Java bietet ab Version 1.4 mit der `assert`-Anweisung eine ähnliche Form für Zusicherungen. Jedoch fehlt eine Analogie für die `let`-Anweisung, die es erlaubt lokale Variablen nur zu Testzwecken einzuführen.

```
class Auction {                                                    Java/P
addMessage(Message m) {
  ocl !this.message.contains(m);

  let oldMessageSize = message.size;
  message.add(m);
  ocl message.size == oldMessageSize +1;

  for(Iterator<Person> ip = bidder.iterator();
                                          ip.hasNext(); ) {
    Person p = ip.next();
    p.addMessage(m);
    ocl MessagesDelivered(this,p);
  }
}}
```

Abbildung 7.4. Zusicherungen als `ocl`-Anweisungen

```
import Auction a, Person p inv MessagesDelivered:                  OCL
  p in a.bidder implies
    forall m in a.message: m in p.message
```

Da Objektdiagramme sich nach Abschnitt 4.4, Band 1 hervorragend als prädikative Beschreibung eines Zustands eignen, lassen sich damit natürlich auch Objektdiagramme sowie die in Abschnitt 4.3, Band 1 diskutierte Kombination von Objektdiagrammen und der OCL als Zusicherungen einsetzen.

Es ist auch möglich, lokale Variablen innerhalb von Schleifen zu definieren. Die mit `let` eingeführten Variablen sind zwar unveränderbar, haben aber denselben Sichtbarkeitsbereich wie normale lokale Java-Variablen und werden daher bei jedem Schleifendurchlauf neu belegt. So lässt sich zum Beispiel die Terminierung der in Abbildung 7.5 gegebenen (nicht ganz trivialen) Schleife prüfen.

```
  int n = ...;                                                     Java/P
  while( n>0 ) {
    let nOld = n;
    if( n % 1 == 0 ) n = n/2; else n = n-1;
    // Verwendung von n ...
    ocl nOld > n;       // absteigende Werte
    ocl nOld > 0;       // Begrenzung durch 0
  }
  ocl n<=0;             // Nach der Schleife gilt
```

Abbildung 7.5. Zusicherungen, die die Terminierung zeigen

Zur Terminierung der Schleife in Abbildung 7.5 wird die stetig absteigende und nach unten durch 0 begrenzte Variable `n` verwendet, deren alter Wert in `nOld` zwischengespeichert wird.

Die Erweiterung von Java um die beiden Anweisungen ist relativ stark angelehnt an die Zusicherungslogik, die sich derselben Techniken bedient, um Aussagen über Programme zu beweisen. Tatsächlich können die beschriebenen Hilfsmittel weit mehr als nur exemplarische Tests unterstützen. Ein geeigneter Beweiskalkül für Java, wie er zum Beispiel in [vO01, RWH01] diskutiert wird, kann damit verifizieren, dass Invarianten immer gelten. Allerdings ist in einem objektorientierten System die Entwicklung einer ausreichend vollständigen Menge von Zusicherungen im Code, so dass ein Verifikationswerkzeug die Korrektheit prüfen kann, sehr aufwändig. Ein solches Vorgehen kann aber vor allem für spezifische Aufgabenstellungen, wie komplexe Algorithmen, Protokolle oder Kernelemente der Sicherheitsarchitektur von Interesse sein.

Die Instrumentierung des Produktionscodes mit Prüfungen für Invarianten muss vom Compiler flexibel gehandhabt werden können. Im laufenden Betrieb sollte die Instrumentierung unterbleiben, im Probebetrieb eine für den Anwender unsichtbare Instrumentierung mit Ausgabe im Hintergrund (Protokoll) möglich sein und im Testsystem eine für Testfälle geeignete Instrumentierung durch Ausgabe und Fehleranzeige mittels JUnit erfolgen.

7.3 Methodenspezifikationen

Eines der wesentlichen Anwendungsgebiete der OCL ist die Beschreibung des Verhaltens einzelner Methoden auf abstrakte Weise, indem für die Methode eine Vorbedingung und eine Nachbedingung angegeben werden. Eine Methodenspezifikation kann als Codeinstrumentierung, als Teil eines Testfalls und als Ausgangspunkt zur Ableitung von Testdatensätzen dienen.

7.3.1 Methodenspezifikationen als Codeinstrumentierung

Das Paar `CC2pre/CC2post` ist ein typisches Beispiel für eine Methodenspezifikation. Es wurde aus Abschnitt 3.4.3, Band 1 übernommen. Dessen Kontext ist bereits dort beschrieben:

```
context Person.changeCompany(String name)                    OCL
pre  CC2pre:  company.name != name &&
              exists Company co: co.name == name
post CC2post:
  company.name          == name &&
  company.employees     == company.employees@pre     +1 &&
  company@pre.employees == company@pre.employees@pre -1
```

Eine typische Umsetzung dieser Methodenspezifikation ist die Instrumentierung des Produktionscodes analog der in Abschnitt 7.2 beschriebenen Verwendung von Zusicherungen. Abbildung 7.6 zeigt einen Methodenrumpf der Methode `changeCompany` für einen der drei nachfolgend noch diskutierten Fälle, der um `ocl`-Anweisungen angereichert wurde.

```
class Person {                                                    Java/P
  changeCompany(String name) {
    // pre CC2pre:
    ocl company.name != name &&
        exists Company co: co.name == name;

    // Methodenimplementierung
    Company oldCo = company;
    Company newCo = AllData.instance().getCompany(name);
    if(newCo==null) ... // Company existiert nicht

    company = newCo;
    newCo.employees++;
    oldCo.employees--;

    // post CC2post:
    ocl company.name == name &&
      company.employees     == company.employees@pre    +1 &&
      company@pre.employees == company@pre.employees@pre-1;
}}
```

Abbildung 7.6. Instrumentierung mit Vor-/Nachbedingung

Die `ocl`-Anweisungen und ihre OCL-Argumente werden, wie bereits in Abschnitt 7.2 diskutiert, in JUnit-fähige Laufzeitprüfungen umgesetzt. Die für die Laufzeit problematische Existenzquantifizierung in der OCL-Bedingung kann effizienter gestaltet werden, indem die Methodenspezifikation so umgestaltet wird, dass statt der Existenzquantifizierung mittels `let`-Konstrukt direkt das infrage kommende `Company`-Objekt festgelegt wird.

7.3.2 Methodenspezifikationen zur Testfallbestimmung

Mit der obigen Umsetzung ist zwar die Methodenspezifikation zur Instrumentierung des Produktionscodes verwendet worden, ein Testfall beziehungsweise dessen operative Umsetzung in einen Testtreiber ist aber damit nicht entstanden. Die Entwicklung von Tests aus einem Vor-/Nachbedingungspaar ist generell nicht einfach. Jedoch lassen sich für bestimmte Formen von Methoden aus der Struktur der Spezifikation geeignete Testdatensätze ableiten.

Ausgehend von der disjunktiven Normalform der Vorbedingung kann eine Partitionierung vorgenommen werden, die es erfordert, pro erfüllbarer Klausel der Normalform einen Testfall zu definieren. In [BW02a, BW02b] wurde dies an einem Beispiel durch Transformation nach Isabelle/HOL [NPW02] vorgenommen um mit einem Verifikationswerkzeug zumindest teilweise automatisiert nicht erfüllbare Anteile zu erkennen und zu eliminieren. Das Beispiel `changeCompany` mit seinen drei Spezifikationsteilen (siehe Abschnitt 3.4.3, Band 1) kann ebenfalls als Disjunktion verstanden werden. Diese Methode ist durch drei Vor-/Nachbedingungspaare beschrieben, die drei Äquivalenzklassen `CC1pre`, `CC2pre` und `CC3pre` von Eingaben festlegen:

```
// Liste von Vorbedingungen als OCL-Teile
context Person.changeCompany(String name)

pre  CC1pre:  !exists Company co: co.name == name

pre  CC2pre:  company.name != name &&
              exists Company co: co.name == name

pre  CC3pre:  company.name == name
```

OCL

Diese Äquivalenzklassen sind paarweise disjunkt und partitionieren den gesamten möglichen Eingabebereich. Die Disjunktion der drei Bedingungen ergibt:

```
CC1pre || CC2pre || CC3pre  <=>  true
```

OCL

Zumindest ein Testfall sollte daher für jede dieser drei Äquivalenzklassen zur Verfügung gestellt werden. Die Identifikation von Äquivalenzklassen für Testdaten ist ein wesentlicher Schritt zur systematischen Entwicklung von Tests. Auf Basis einer manuellen oder werkzeuggestützten Analyse der Spezifikation können interessante Testfälle ermittelt werden. Leider ist davon auszugehen, dass die Generierung von Testdatensätzen, in diesem Beispiel also Objektstrukturen, in denen jeweils eine der angegebenen Bedingungen gilt, nicht ohne weiteres automatisierbar ist. Beispielsweise ist die konstruktive Umsetzung des Existenzquantors `exists x: P`, also die Generierung von Code, der ein Objekt `x` erzeugt, das die Bedingung `P` erfüllt, nur für Spezialfälle von `P` lösbar. Dennoch ist es hilfreich, bei der Analyse einer gegebenen Testsammlung einen Hinweis zu erhalten, wenn eine der angegebenen Äquivalenzklassen durch Tests nicht abgedeckt wird.

Wie in [Mye01, Lig90, Bal98] beschrieben, kann die Partitionierung des Testdatenraums durch eine Analyse der Nachbedingungen verfeinert werden. Dazu wird folgende Spezifikation betrachtet:

```
context int abs(int val)
pre:  true
post: if (val>=0) then result==val else result==-val
```

OCL

Da die Vorbedingung **`true`** ist, kann sie für keine Partitionierung des Testdatenraums genutzt werden. In diesem Fall kann aber eine Analyse der Nachbedingung helfen, die sofort zwei Äquivalenzklassen erkennen lässt: `val>=0` und `val<0`.

Eine weitere Möglichkeit zur Testfalldefinition bilden die standardmäßige Äquivalenzklassenbildung und die Betrachtung von Grenzwertfällen. Dies eignet sich besonders für die von der Programmiersprache angebotenen Datentypen. Zum Beispiel bietet es sich für ganze Zahlen an, Testdaten aus `Set{-n,-10,-2,-1,0,1,2,3,4,10,11,n+1}` (für ein großes `n` des Wertebereichs) zu verwenden, da meist kritische Sonderfälle im Bereich um die `0` zu finden sind. Für Container-Datenstrukturen, boolesche Werte und Fließkommazahlen lassen sich ähnliche Standards festlegen.

Diese Standardfälle leiten sich aus der Erkenntnis her, dass in diesen Datentypen ausgezeichnete Werte existieren, bei denen die Implementierung einen anderen Pfad nimmt, als bei benachbarten Werten. Das Verfahren der Grenzwertanalyse [Mye01, Bal98] grenzt explizit solche Wertebereiche ein und deckt sie durch Testdatensätze auf jeder Seite der Grenze ab. Im Fall der Absolutfunktion ist die `0` eine solche Grenze und erfordert `-1`, `0`, `+1` als Testdaten. Meist ist jedoch die Feststellung der Grenzwerte komplexer, da Fallunterscheidungen die Parameter in Beziehung setzen können.

Neben der Entwicklung von Black-Box-Tests aus der Spezifikation darf aber auch die Entwicklung von White-Box-Tests aus einer gegebenen Implementierung nicht vernachlässigt werden. Erst durch eine Analyse der Implementierung, in diesem Fall also vor allem der Java-Coderümpfe und Transitions-Aktionen, werden zusätzliche Fälle offensichtlich, die in der Spezifikation eventuell nicht erkennbar waren. Hier sind klassische Techniken zur Testüberdeckung einzusetzen [Bei95, Bal98].

In manchen Fällen lässt sich die Menge und Art notwendiger Testfälle aus einer Spezifikation nicht vorhersagen, weil es eine Reihe unterschiedlicher Implementierungen gibt. Dazu gehören zum Beispiel Sortierverfahren, wie Mergesort, Quicksort oder Bubblesort, die jeweils sehr unterschiedlich funktionieren und deshalb unterschiedliche Testdatensätze erfordern. Besonders aufwändig werden Tests für Kombinationen, indem etwa Bubblesort für das Vorsortieren kleiner Reihungen vor der Anwendung von Mergesort verwendet wird.

Allgemein ist es aber wichtig, dass Testfälle sowohl auf Basis einer Codeanalyse als auch aus der Spezifikation entwickelt werden. Denn codebasierte Tests sind vor allem zur Sicherung der Robustheit geeignet, während spezifikationsbasierte Tests die Übereinstimmung des implementierten und des spezifizierten Verhaltens, also der Spezifikationskonformität, prüfen. Ist zum Beispiel bei der Implementierung ein Spezifikationsfall vergessen worden (eine *Auslassung*), so kann dies durch codebasierte Testfälle nicht entdeckt werden.

Jedoch ist auch bei der schematischen Verwendung von Metriken zur Testfallüberdeckung Vorsicht geboten, da diese dazu führen können, die

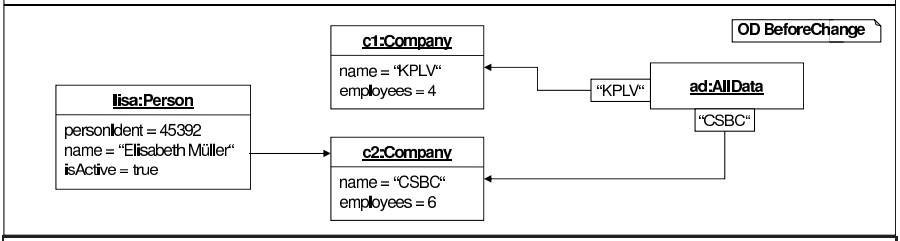
Metriken zu schematisch zu erfüllen und einerseits doch wichtige Fälle zu übersehen, andererseits aber viel, eventuell unnötigen Zusatzaufwand erzeugen. Generell ist daher eine an die Komplexität des Testlings und die notwendige Qualität des Systems angepasste Kombination aus verschiedenen Vorgehensweisen zur Testfalldefinition als optimal anzusehen.

7.3.3 Testfalldefinition mit Methodenspezifikationen

Eine Methodenspezifikation stellt noch keinen vollständigen Testfall dar, sondern benötigt zusätzlich einen Testdatensatz. Abbildung 7.7 zeigt eine tabellarische Darstellung einer Testfallsammlung, bestehend aus fünf Testfällen, aus der automatisch eine Testsuite für JUnit generiert werden kann.

Testling: Person.changeCompany(String name) — Test
Methodenspezifikation (methodspec): CCpre/CCpost

Test#	Objektdiagramm für Testdaten (testdata)	zusätzlicher Java-Code (tune)	Parameter des Methodenaufrufs (driver)	zusätzlich geprüfte Eigenschaften und Invarianten (assert)
(1)	OD BeforeChange	-	("CSBC")	-
(2)	OD BeforeChange	-	("KPLV")	-
(3)	OD BeforeChange	-	("NECO")	exists co in ad.company: co.name=="NECO"
(4)	OD BeforeChange	c1.employees=7; c2.employees=9;	("KPLV")	-
(5)	OD BeforeChange	c1.employees=7; c2.employees=9;	("NECO")	c2.employees==9;



OCL

```
context Person.changeCompany(String name)
pre  CCpre:  true
post CCpost: company.name == name &&
  (company@pre.name != name implies
  company@pre.employees == company@pre.employees@pre -1 &&
   (company.employees   == company.employees@pre     +1
     || (isnew(company) && company.employees == 1)))
```

Abbildung 7.7. Definition einer Testfallsammlung

Dabei kann für die ersten drei Fälle dasselbe Objektdiagramm als Testdatensatz verwendet werden, denn es können alleine durch den Aufruf-

Parameter alle drei Fälle variiert werden. Für die Fälle (4) und (5) wird zusätzlicher Java-Code verwendet, der nach Aufbau des Objektdiagramms, aber vor dem Test selbst ausgeführt wird. Beide Fälle stellen Varianten bereits vorhandener Fälle dar. Sie prüfen vor allem die korrekte Änderung der Zahl der Angestellten. Dies erscheint notwendig, denn sonst hätte eine Implementierung die Angestelltenzahl immer auf denselben Wert setzen können, ohne dass dies entdeckt worden wäre.

Durch die Möglichkeit, zusätzliche OCL-Bedingungen oder Invarianten (per Namen) anzugeben, können weitere Eigenschaften geprüft werden. So wird in Fall (3) gefordert, dass die neue Firma auch im Singleton `ad` angemeldet ist, und im Fall (5) beschrieben, dass die unbeteiligte Firma „KPLV" die Anzahl ihrer am System angemeldeten Angestellten beibehält. Beides hätte auch jeweils durch Objektdiagramme ausgedrückt werden können.

Das angegebene Objektdiagramm `BeforeChange` wird konstruktiv eingesetzt, denn daraus werden die Testdaten generiert. Das Objektdiagramm gibt hier also sogar die komplette Umgebung an, es existieren keine weiteren `Person`- und `Company`-Objekte. Deshalb verletzt der im Attribut `employee` angegebene Wert Invarianten, die deshalb bei diesem Test nicht geprüft werden dürfen.

7.4 Sequenzdiagramme

Unter einer Testsequenz wird eine Folge von Eingaben an den Testling verstanden, die dieser während eines Tests bearbeitet. Bei schlecht manipulierbaren Systemen sind einige Anstrengungen zu unternehmen, um zu verstehen, wie Testsequenzen zu entwerfen sind, um bestimmte Situationen herzustellen und das Verhalten des Systems in solchen Situationen zu testen. Demgegenüber hat sich gemeinsam mit der Objektorientierung verstärkt eine Vorgehensweise durchgesetzt, nur die Testdaten direkt vor der Testausführung zu erstellen, statt einen Pfad von einem Startzustand bis zu diesem Datensatz anzugeben. Dies hat mehrere Gründe. Zum einen sind aufgrund geänderter Codierungsstandards im objektorientierten Ansatz Methoden typischerweise sehr viel kleiner als die Prozeduren der imperativen Welt. Zum zweiten kann mit dynamischer Bindung und Bildung von Unterklassen der Testling besser kontrolliert und manipuliert werden, indem ihm an kritischen Stellen Dummies statt der echten Implementierung angeboten werden. Zum dritten hat sich die Erkenntnis durchgesetzt, dass der Code durchaus so gestaltet oder umgestaltet werden sollte, dass er gut testbar ist. Zum Beispiel wird in dem in Kapitel 8 vorgestellten Ansatz die Erzeugung neuer Objekte in eine Factory ausgelagert, um in einem Factory-Dummy auch diese Erzeugung kontrollieren zu können. Durch diese Maßnahmen ist keine Sequenz von Aufrufen mehr notwendig, um eine bestimmte Situation im System herzustellen. Diese kann direkt als Testdatensatz konstruiert werden. Deshalb kann ein Großteil der Testfälle durch einen einfachen

Methodenaufruf gesteuert werden, nachdem der adäquate Testdatensatz erstellt wurde. Diese Vorgehensweise birgt allerdings das Risiko, dass der benutzte Testdatensatz im realen System gar nicht auftreten kann.

Durch die Zerlegung der Funktionalität in viele objektorientierte Methoden entstehen jedoch komplexere Aufrufhierarchien. Die Objekte bilden damit komplexere Interaktionsmuster, die durch die Vor- und Nachbedingung einer Methode oft nicht adäquat erfasst werden. Für das Verständnis des Zusammenspiels zwischen verschiedenen Objekten oder Systemteilen ist es daher sinnvoll, diese Interaktionsmuster in Form von Sequenzdiagrammen zu modellieren. Ein Sequenzdiagramm ist eine exemplarische Darstellung eines möglichen Ablaufs. Deshalb eignet sich ein Sequenzdiagramm in idealer Weise, den internen Ablauf eines Tests zu beschreiben. Beispielsweise beschäftigt sich [PJH+01] mit der Verwendung eines an die UML angelehnten MSC-Dialekts für die Konformitätstests von Telekommunikationsprotokollen.

Im Folgenden wird anhand einiger Beispiele demonstriert wie Sequenzdiagramme zur Modellierung von Testfällen verwendet werden können.

7.4.1 Trigger

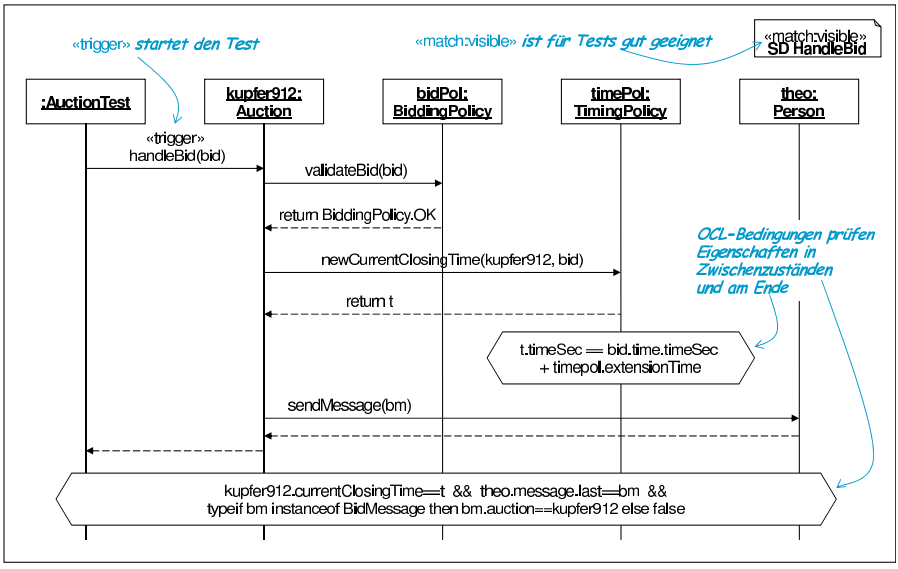


Abbildung 7.8. Sequenzdiagramm als Testfallbeschreibung

Das in Abbildung 7.8 gezeigte Beispiel nutzt den in Abschnitt 6.1, Band 1 definierten Stereotyp «trigger», um den ersten Methodenaufruf als Auslöser

zu kennzeichnen. Damit müssen notwendigerweise die weiteren angegebenen Methodenaufrufe und Returns stattfinden, damit der beschriebene Test erfolgreich ist. Zusätzliche OCL-Bedingungen können innerhalb des Sequenzdiagramms angegeben werden, um Zwischenzustände zu prüfen sowie eine finale Prüfung des Ergebnisses vorzunehmen. Das Sequenzdiagramm zeigt nicht alle stattfindenden Methodenaufrufe. Es abstrahiert zum Beispiel von Interaktionen zwischen dem Auktionsobjekt und den anderen teilnehmenden Personen, die ebenfalls eine Mitteilung erhalten. Dieser Aspekt wird also mit dem angegebenen Sequenzdiagramm nicht getestet.

Um den Test zu vervollständigen, ist für den Ablauf neben dem Sequenzdiagramm ein Testdatensatz notwendig, der zum Beispiel durch ein Objektdiagramm beschrieben werden kann. Durch die Verwendung eines Objektdiagramms werden die im Sequenzdiagramm angegebenen Objekte in eine strukturelle Beziehung durch Links gesetzt, die im Sequenzdiagramm nicht dargestellt werden kann. Aufgrund der Komplexität der konstruktiv und damit vollständig zu modellierenden Struktur, empfiehlt es sich im Beispiel, diese Struktur durch zwei Objektdiagramme zu beschreiben. Im Beispiel wird angenommen, dass eine vervollständigte und damit für konstruktive Testdatengenerierung verwendbare Fassung des Objektdiagramms aus Abbildung 5.13 unter dem Namen `Kupfer912` zur Verfügung steht. Abbildung 7.9 beinhaltet damit die vollständige Beschreibung des Tests auf Basis des Sequenzdiagramms `HandleBid`.

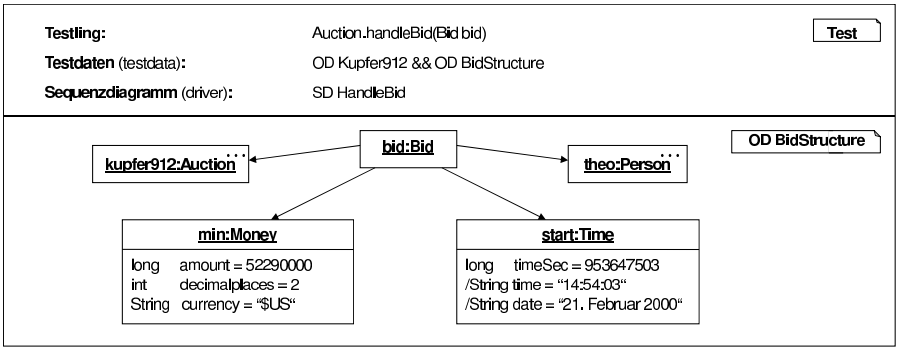


Abbildung 7.9. Test nutzt Sequenzdiagramm `HandleBid`

Da es ein Sequenzdiagramm erlaubt, OCL-Bedingungen zu verschiedenen Punkten im Ablauf des Systems anzugeben, ist die Verwendung einer zusätzlichen Methodenspezifikation nicht notwendig, obwohl eine solche im Test zusätzlich angegeben werden kann.

Für die Prüfung der angegebenen Interaktionen sowie der OCL-Bedingungen während des Ablaufs eines Sequenzdiagramms ist es notwendig, den Code geeignet zu instrumentieren. So muss, wie in Abschnitt 5.5

beschrieben, jeder Methodenaufruf und jede `return`-Anweisung protokolliert werden. Zusätzlich sind die OCL-Bedingungen zu prüfen. Die Prüfung, der Aufbau der Testdaten und die Ausführung des Tests werden im Testtreiber lokalisiert, der als von `JUnit` aufrufbare Methode in der Klasse `AuctionTest` abgelegt wird.

7.4.2 Vollständigkeit und Matching

Gemäß Abschnitt 6.3, Band 1 gibt es mehrere Interpretationsmöglichkeiten für ein Sequenzdiagramm. So kann durch Awendung des Stereotyps ≪match:complete≫ festgelegt werden, dass alle Interaktionen des Objekts im dargestellten Zeitraum im Sequenzdiagramm gezeigt werden. Diese für Spezifikationen meist unpraktikabel starke Aussage hat im Test seine Berechtigung, da die getesteten Objekte ausschließlich zum Zweck des Tests erzeugt wurden und weder davor, noch danach benutzt werden.

Der Stereotyp ≪match:visible≫ wirkt etwas schwächer, indem er nur fordert, dass alle Interaktionen zwischen den im Diagramm angegebenen Objekten gezeigt werden, aber weitere Methodenaufrufe an andere Objekte möglich sind. Damit ist ≪match:visible≫ ebenfalls eine nützliche Interpretation von Sequenzdiagrammen für Tests.

Die Verwendung von ≪match:initial≫ erlaubt den getesteten Klassen weitere Freiheiten, da das Sequenzdiagramm nach Auslösung des Triggers jeweils nur die angegebenen Interaktionen fordert, aber weitere zulässt. Damit gehen aber einige Kontrollmöglichkeiten verloren, die gerade für Tests von Interesse sind. Die Interpretation mit dem Stereotyp ≪match:free≫ ist daher für Tests kaum mehr geeignet, wenn der Stereotyp auf das komplette Sequenzdiagramm angewandt wird. Eine interessante Technik bietet allerdings die kombinierte Verwendung von Stereotypen, wie in Abbildung 7.10 gezeigt.

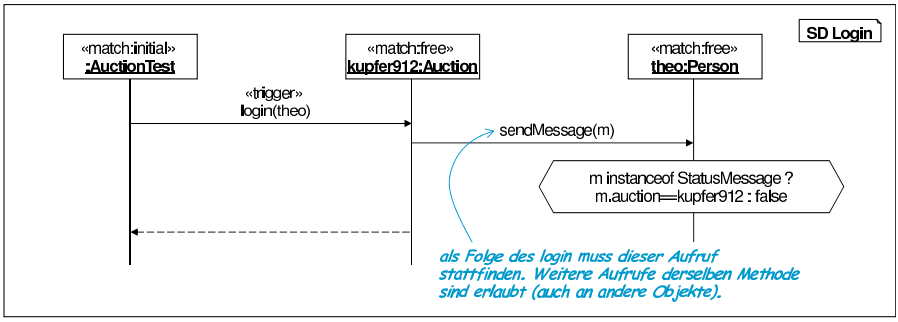


Abbildung 7.10. Kombinierte Verwendung von Stereotypen

Bei Anmeldung einer neuen Person in einer Auktion werden dem Personenobjekt durch die Methode `sendMessage` nacheinander alle

aufgetretenen Nachrichten zugestellt. Dabei wird `sendMessage` mehrfach aufgerufen. Das Sequenzdiagramm fordert nun, dass ein Aufruf dabei ist, der die angegebenen Eigenschaften erfüllt, bevor die übergeordnete `login`-Methode abgearbeitet ist.

7.4.3 Nicht-kausale Sequenzdiagramme

Wie in Abschnitt 6.4, Band 1 beschrieben, kann ein Sequenzdiagramm unvollständig und damit nicht-kausal sein. Beispielsweise kann gewollt sein, dass ein an der Ausführung beteiligtes Objekt oder einzelne Methodenaufrufe nicht beobachtet werden sollen, diese aber weitere beobachtete Methodenaufrufe zur Folge haben. Derartige Sequenzdiagramme sind für Tests ebenfalls geeignet, wie das Beispiel in Abbildung 7.11 zeigt.

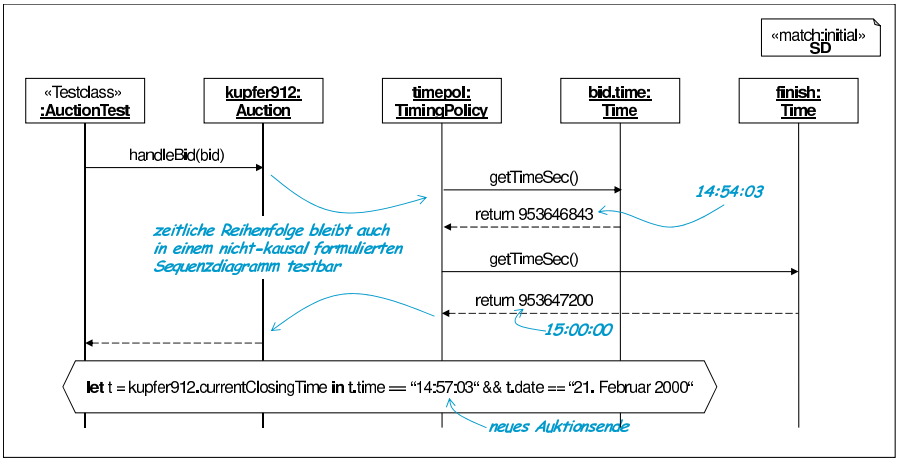


Abbildung 7.11. Nicht-kausales Sequenzdiagramm als Testablaufbeschreibung

Die zeitliche Folge der Methodenaufrufe wird alleine durch deren Reihenfolge entlang der Zeitachse festgelegt und kann somit überprüft werden. Die Kausalität kann allerdings nicht beliebig aufgehoben werden. Natürlich können `return`-Pfeile nur angegeben werden, wenn der zugehörige Methodenaufruf angegeben ist.

7.4.4 Mehrere Sequenzdiagramme in einem Test

Da ein Sequenzdiagramm eine Beobachtung beschreibt, ist es möglich, mehrere Sequenzdiagramme für verschiedene Teilabläufe innerhalb eines Tests einzusetzen. So ergänzen sich die beiden Diagramme aus den Abbildungen

7.8 und 7.11. Dabei werden die Sequenzdiagramme unabhängig voneinander geprüft, also jeweils so, als wenn kein anderes Sequenzdiagramm vorhanden wäre. Die in beiden Sequenzdiagrammen gemeinsam auftretenden Methodenaufrufe, im Beispiel also `handleBid`, werden daher im Testablauf durch denselben Methodenaufruf erfüllt. Eine sequentielle oder alternative Komposition oder Iteration von Sequenzdiagrammen, wie sie MSC's [IT11], der UML-Standard [OMG10] oder auch YAMS [Krü00] erlauben, wäre eine mögliche Erweiterung, ist jedoch, wie in Kapitel 6, Band 1 begründet, in UML/P nicht vorgesehen. Stattdessen unterliegt der Interpretation mehrerer Sequenzdiagramme eine lose Form der „Verschmelzung", die gleiche Methodenaufrufe identifizieren kann und so die Sequenzdiagramme unabhängig voneinander einsetzt.

7.4.5 Mehrere Trigger im Sequenzdiagramm

Testklassen werden entsprechend mit den in JUnit üblichen Codierungsstandards mit dem Suffix „Test" versehen oder, wie in Abschnitt 2.5.2, Band 1 vorgeschlagen, durch einen Stereotyp wie ≪Testclass≫ markiert. Weil die Methodenaufrufe zum Start des Tests ausschließlich vom Testtreiber vorgenommen werden können, umgekehrt aber auch jeder Methodenaufruf, ausgehend von der Testklasse als Trigger zu verstehen ist, ist bereits einer der Stereotypen ≪trigger≫ oder ≪Testclass≫ bei der Definition von Testfällen ausreichend.

Wie Abbildung 7.12 zeigt, kann ein Sequenzdiagramm auch mehrere mit dem Stereotyp ≪trigger≫ markierte Methodenaufrufe beinhalten. Ein solches Sequenzdiagramm beschreibt einen Testtreiber, der nacheinander mehrere Methodenaufrufe durchführt.

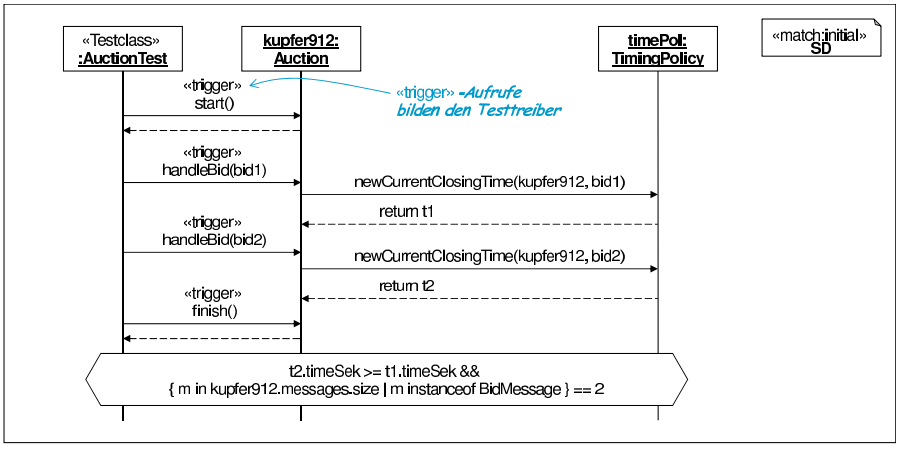


Abbildung 7.12. Sequenzdiagramm mit aufeinanderfolgenden Triggern

7.4.6 Interaktionsmuster

Die Verwendung eines Sequenzdiagramms zur Definition eines Testfalls hat den wesentlichen Vorteil, dass durch das Sequenzdiagramm das geprüfte Interaktionsmuster explizit dargestellt ist. Durch Abgleich mit einer vorhandenen Implementierung lässt sich relativ leicht analysieren, ob und wie gut eine gegebene Menge von solchen Testfällen die verschiedenen möglichen Abläufe im Testling abdeckt.

Eine vollständige Überdeckung aller möglichen Abläufe eines Testlings ist aber meist nicht möglich, denn ganz ähnlich dem Pfadüberdeckungstest einer Prozedur [Bal98, Mye01] führen Schleifen und Rekursion zu einer unbeschränkten Anzahl von Varianten möglicher Abläufe. Für praktische Belange ist es deshalb notwendig, eine endliche, möglichst repräsentative Menge von Abläufen festzulegen, die in Testfällen umgesetzt wird. Im Auktionsprojekt sind dies zum Beispiel Auktionen mit verschiedenen Mengen an Geboten (0,1,2,3,5 und sehr viele), die in Testfällen geprüft worden sind. Solche Tests, die den Ablauf einer Teilphase oder sogar der gesamten Auktion prüfen, gehen über den einzelnen Methodentest hinaus und bilden damit einen ersten Schritt zur Integration verschiedener Systemteile. Sequenzdiagramme sind daher auch geeignet, Integrationstests zu modellieren. Dabei ist es sinnvoll, bei Integrationstests vor allem die Schnittstellen zwischen den Systemteilen zu modellieren und die Kapselung der Systemteile selbst zu respektieren. Dafür sind Sequenzdiagramme mit Stereotypen wie ≪match:free≫ geeignet.

Weitere Probleme, eine Überdeckung der Interaktionsmuster zu erzielen, ergeben sich aus der explodierenden Anzahl von möglichen Objektstrukturen, die als Grundlage für die Abläufe dienen:

- Mengenwertige Assoziationen erlauben grundsätzlich eine unendliche Anzahl verschiedener Objektstrukturen, die bei einer Bearbeitung ebenfalls Schleifen benötigen und daher wieder zu unendlich vielen möglichen Ablaufstrukturen führen.
- Die sich aus der Vererbung ergebende dynamische Bindung von Methoden erfordert es, für ein gegebenes Objekt einer Klasse auch alle Unterklassen zu testen. Dies führt bei mehreren Objekten im Testdatensatz zu einer Explosion der Testfallanzahl.
- Die dynamische Veränderbarkeit von Objektstrukturen macht das Ablaufverhalten eines Testlings auch von den Parametern abhängig. Beispielsweise kann ein Parameter bestimmen, wieviele Objekte in einer Datenstruktur erzeugt oder wie diese durch Links verbunden werden.

Deshalb ist es für praktische Belange unrealistisch, eine vollständige Überdeckung möglicher Abläufe durch Testfälle auf Basis von Sequenzdiagrammen zu erwarten. Wie das folgende Beispiel zeigt, ist damit außerdem auch nicht gesichert, dass eine Anweisungsüberdeckung der getesteten Methoden stattfindet:

```
void methode(int i) {
  if(i >= 1)
    foo(i);
  else {
    attribut = i;
    foo(i+1);
  }
}
```

Alle Sequenzdiagramme zum Test von `method` zeigen dieselbe Ablaufstruktur. Es ist daher falsch, anzunehmen, dass zwei Sequenzdiagramme mit derselben Ablaufstruktur dieselben Abläufe der Implementierung testen.

Deshalb ist die Definition von Integrationstests mit Sequenzdiagrammen nur eines von mehreren Instrumenten zur Testfallmodellierung, das gemeinsam mit den bereits besprochenen Testfällen aus Methodenspezifikationen und der Prüfung von Invarianten einzusetzen ist.

7.5 Statecharts

Die in Kapitel 5, Band 1 definierten Statecharts haben folgende grundsätzlichen Anwendungsgebiete:

1. *Ausführbare Statecharts* werden konstruktiv in Code übersetzt und finden im Produktionssystem oder als Orakelfunktion Verwendung.
2. *Für Tests einsetzbare Statecharts* werden zur Überprüfung des korrekten Ablaufs eines Tests genutzt.
3. Statecharts dienen als allgemein gültige *Verhaltensspezifikationen* und werden zur manuellen oder automatisierten Ableitung von Testfällen eingesetzt.
4. *Abstrakte Statecharts* dienen als Dokumentation, sind aber für Tests und Codegenerierung zu abstrakt oder zu informell.[2]

Ein abstraktes, informelles Statechart kann durch Detaillierung und Präzisierung der darin enthaltenen Information zu einem für Tests geeigneten oder ausführbaren Statechart transformiert werden. Da in einem Statechart gleichzeitig konstruktive Elemente wie Aktionen und deskriptive Elemente wie OCL-Nachbedingungen verwendet werden können, ist auch ein kombinierter Einsatz sinnvoll. So kann ein konstruktiver Anteil eines Statecharts zur Generierung von Methoden eingesetzt werden, während zum Beispiel Zustands- und Nachbedingungen parallel dazu verwendet werden, Tests zu unterstützen und Invarianten zu generieren.

[2] In [Wil01] werden diese als Skizzen („Sketches“) und nicht als echte Modelle bezeichnet.

7.5.1 Ausführbare Statecharts

Ein ausführbares Statechart wird als konstruktives Modell genutzt und, wie in Abschnitt 5.4 beschrieben, direkt in Code umgesetzt. Ausführbare Statecharts besitzen typischerweise prozedurale Aktionen in Form von Java-Code. Der bei der Umsetzung entstehende Java-Code kann durch im Statechart eingetragene Zustandsinvarianten erweitert sein, die während der Testphase des Systems im Code zur Laufzeit geprüft werden. Konstruktive Statecharts eignen sich jedoch nicht als Testbeschreibung, da der Effekt einer prozeduralen Aktion nicht getestet, sondern die Aktion nur ausgeführt werden kann.

Im Prinzip können Statecharts auch als Testtreiber eingesetzt werden. Es ist jedoch ein Prinzip bei der Definition von Testtreibern, diese möglichst einfach und damit im Wesentlichen ohne verzweigte Kontrollstruktur zu entwickeln. Die lineare Struktur eines Testtreibers aber lässt ein Statechart zu einer linearen Form entarten, die auch durch ein Sequenzdiagramm dargestellt werden kann.

Interessant ist der Einsatz eines konstruktiven Statecharts als Orakelfunktion für das Testergebnis. Dies empfiehlt sich zum Beispiel, wenn der durch ein Statechart generierbare Code `g` für das Produktionssystem zu langsam ist und deshalb eine alternative Implementierung `f` realisiert wurde. Im Test wird der Testdatensatz `x` kopiert, auf beiden Testdatensätzen die jeweilige Funktion angewandt und das Ergebnis passend verglichen. Die Vergleichsform kann, wie in Abbildung 6.6 gezeigt, durch die Angabe eines `comparator`-Eintrags auch explizit definiert werden, da es eine ganze Reihe von Vergleichsmöglichkeiten gibt, etwa ob Reihenfolgen in Listen eine Rolle spielen.

Ein Black-Box-Vergleich der Ergebnisse `f(x)=g(x)` hat den Vorteil, dass die interne Realisierung einer Methode wesentlich von der Beschreibung durch das Statechart abweichen und damit einem Refactoring unterworfen werden kann, ohne das Statechart in seiner Funktion als Orakel zu beeinträchtigen. Bei dieser Vorgehensweise werden aber die im Statechart angegebenen OCL-Bedingungen während des Ablaufs nicht geprüft. Um das zu erreichen ist eine gleichzeitige Verwendung der konstruktiven Anteile des Statecharts für die Generierung von Produktionscode und der deskriptiven Anteile für die Generierung von Prüfungen in Tests notwendig.

Sowohl bei der Verwendung eines Statecharts als Testtreiber als auch als Orakel ist Nichtdeterminismus im Statechart kritisch. Nichtdeterministische Statecharts, also solche mit überlappenden Schaltbereichen, sind nur sinnvoll, wenn das Statechart als Ablaufbeschreibung im Test eingesetzt wird. Der Nichtdeterminismus wirkt dann als Unterspezifikation und wird durch die tatsächliche Implementierung aufgelöst.

In Abbildung 7.13 sind drei der Strategien des Auktionssystems skizziert, die gemäß der Beschreibung in Abschnitt D.2, Band 1 bei Gebotsabgabe

unterschiedliche Verlängerungen (delta) in Abhängigkeit des Zeitpunkts des jeweils aktuellen Gebots bewirken.

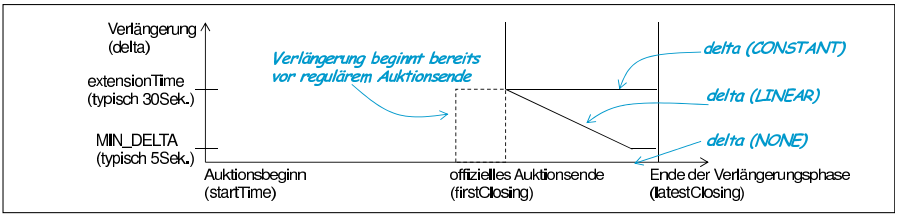


Abbildung 7.13. Verlängerungsstrategien bei Geboten in der Auktion

Die historisch gewachsene Berechnungsstruktur für das Auktionsende nach Abgabe eines Gebots ist durch das Methoden-Statechart in Abbildung 7.14 beschrieben.[3] Es zeigt die Berechnung für eine konstante Verlängerung, keine Verlängerung sowie eine linear absteigende Verlängerung mit minimaler Untergrenze.[4]

Die Berechnung ist im tatsächlichen Auktionssystem in verschiedene Methoden auf mehrere Unterklassen von `TimingPolicy` verteilt worden. Die in Abbildung 7.14 gezeigte ursprüngliche Modellierung der Funktionalität ist deshalb als konstruktive Implementierung nicht geeignet, kann aber aufgrund ihrer Ausführbarkeit und der inhaltlichen Korrektheit als Orakelfunktion eingesetzt werden.[5]

Daher können Tests unter Nutzung der modellierten Funktionalität als Orakel, wie in Abbildung 7.15, formuliert werden.

7.5.2 Statechart als Ablaufbeschreibung

Ein Statechart, das einen Lebenszyklus deskriptiv modelliert, kann ähnlich wie ein Sequenzdiagramm zur Prüfung der Korrektheit eines Systemablaufs eingesetzt werden. Das Statechart wird also als Prädikat für einen Testfall verstanden, der in einem vorgegebenen Zustand beginnt, und dessen Interaktionen durch das Statechart überwacht werden. Damit kann ein Statechart in einem Test zusätzlich als einzuhaltendes Prädikat eingesetzt werden. Das Statechart selbst stellt allerdings keinen kompletten Testfall dar, da sowohl die Testdaten als auch der Testtreiber fehlen.

[3] Alle `Time`-Objekte wurden in diesem Beispiel vereinfachend durch `long`-Werte ersetzt.

[4] Ein so entwickeltes Statechart dient typischerweise als erster Entwurf und kann wie auch hier meistens vereinfacht werden. In diesem Fall ist auch eine tabellarische Darstellung statt eines Diagramms sinnvoll.

[5] Parallel zum Refactoring der Funktionalität von `newCurrentClosingTime` mit der Verschiebung auf die `TimingPolicy`-Klassen sind Anpassungen der verwendeten Attribute notwendig. Auf diese wird im Beispiel aber verzichtet.

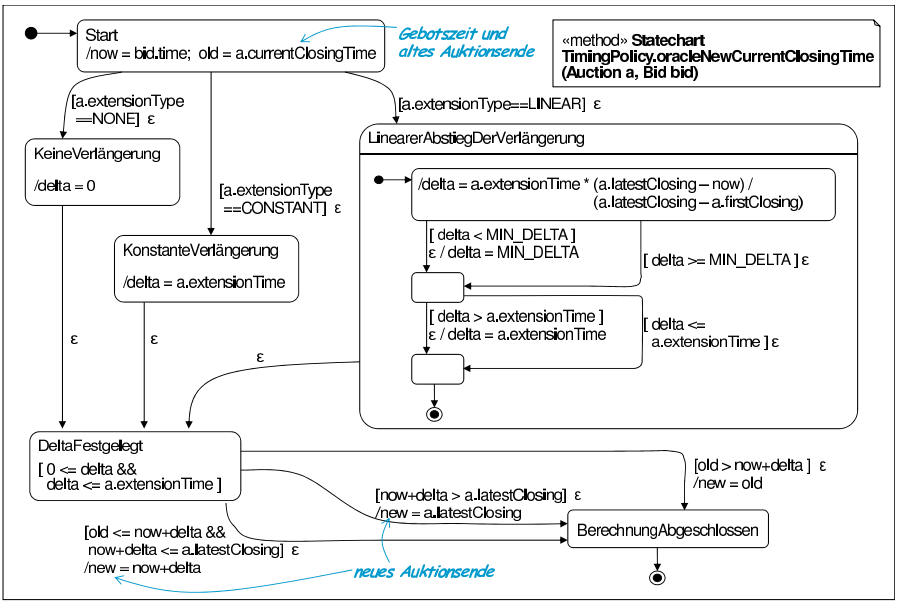


Abbildung 7.14. Statechart berechnet neues Auktionsende

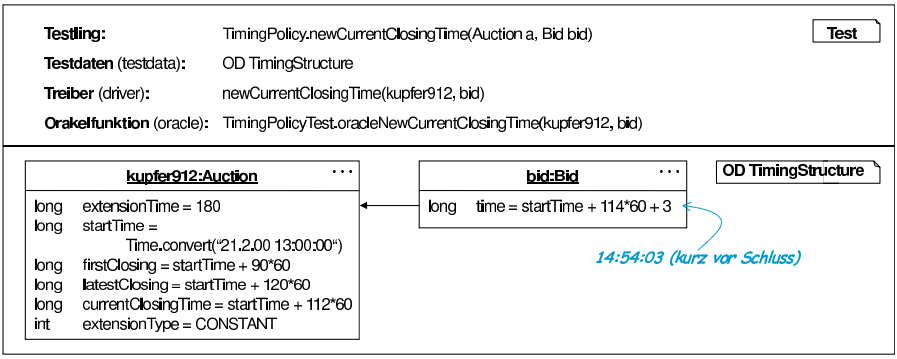


Abbildung 7.15. Test mit Methoden-Statechart als Orakelfunktion

Ein zugehöriger Testablauf wird typischerweise durch ein Sequenzdiagramm beschrieben. Die beteiligten Objekte werden in der üblichen Form durch eine Objektstruktur modelliert. Einem oder mehreren dieser Objekte kann nun jeweils ein Statechart zugeordnet sein, dessen Gültigkeit zu prüfen ist. Die Objekte müssen sich dabei nicht in einem durch das Statechart vorgegebenen Startzustand befinden oder während des Testablaufs einen Endzustand erreichen. Deshalb werden für jedes Objekt zusätzlich der zu Beginn des Tests relevante Zustand und die am Ende des Tests erlaubten Zustände angegeben. Abhängig von der in Abschnitt 5.4 beschriebenen Umsetzung

der Zustände durch ein Aufzählungsattribut oder Prädikate werden diese Zustände eingestellt oder geprüft. Da ein Statechart verschiedene Pfade darstellt ist eine Wiederverwendung eines Statecharts für mehrere Testfälle sinnvoll.

Abbildung 7.16 zeigt ein geeignetes Statechart und einen Testtreiber, der alternativ auch als Sequenzdiagramm dargestellt sein könnte. Weitere Testfälle können in anderen Zuständen starten oder andere Pfade nehmen, indem sie zum Beispiel Gebote (`bid`-Aufrufe) berücksichtigen.

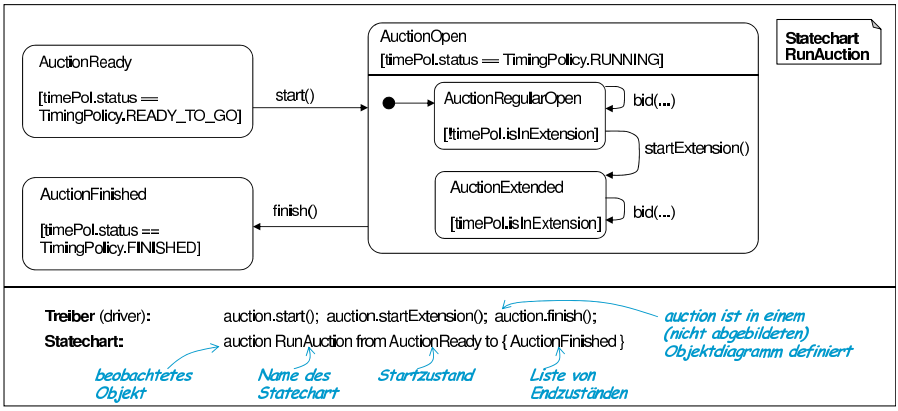


Abbildung 7.16. Statechart als Lebenszyklusbeschreibung im Test

Für den Vergleich des Testablaufs mit den Transitionen und Zuständen des Statecharts ist eine Instrumentierung des Testlings in ähnlicher Form wie beim Sequenzdiagramm notwendig. Das heißt, zu Beginn und am Ende jeder Methode wird eine geeignete Instrumentierung eingebaut, um unter anderem die Zustände, Zustandsinvarianten und die Schaltbereitschaft prüfen zu können. Nicht alle im Statechart formulierbaren Eigenschaften können allerdings zur Prüfung verwendet werden. Zum Beispiel sind prozedurale Aktionen als Prüfvorgaben ungeeignet und müssen daher (falls angegeben) ignoriert werden.[6]

Außerdem können bei Methoden-Statecharts beziehungsweise generell die Kontrollzustände (Stereotyp «controlstate») eines Statecharts nicht geprüft werden, da deren automatische Zuordnung zu Programmstellen im Code nicht möglich ist. Nur wenn die Implementierung aus dem prozeduralen Teil eines Methoden-Statecharts konstruktiv erzeugt wird, ist es möglich,

[6] Es ist möglich, prozedurale Aktionen als Orakelfunktion g für Tests der Form $f(x) = g(x)$ des Effekts jeder einzelnen Transition zu verwenden. Das ist allerdings wegen des für jede Transition notwendigen Kopierens der jeweils aktuellen Datenstruktur nur bedingt effizient.

den deskriptiven Teil gleichzeitig als Zusicherungen in die generierte Methode einzusetzen.

Auch für die in Abschnitt 5.5.2, Band 1 diskutierten, abschnittsweise komponierten Aktionen (Stereotyp ≪action:sequential≫) ist eine Prüfung der zwischendurch gültigen Bedingungen nur bei einer konstruktiven Umsetzung des Statecharts möglich.

7.5.3 Testverfahren für Statecharts

Da Statecharts nicht einen einzelnen Ablauf, sondern eine unendliche Menge von potentiellen Abläufen für ein Objekt beschreiben, ist ein Statechart eine hervorragende Ausgangsbasis für die Entwicklung von Tests und für die Messung der Überdeckung der im Statechart modellierten Verhaltensteile. Im Gegensatz zu den bisher diskutierten Formen des Einsatzes der UML zur Testfallmodellierung, wird hier also nicht ein Diagramm für einen Testfall eingesetzt, sondern es werden aus einem Diagramm viele Testfälle abgeleitet.

Wird ein Statechart konstruktiv eingesetzt, so können die aus dem prädikativen Anteil (OCL-Bedingungen) generierten Zusicherungen als White-Box-Tests verstanden werden, da das Statechart die Implementierung darstellt. Existiert jedoch eine unabhängig entstandene Implementierung, so wird das Statechart als Spezifikation mit der Implementierung verglichen. Die Tests sind dann also Black-Box-Tests.

Es ist heute Gegenstand der Forschung aus gegebenen Statechart-Spezifikationen möglichst kompakte, aber doch vollständige Sammlungen von Testdatensätzen zu generieren, die eine Überdeckung nach einer vorgegebenen Überdeckungsmetrik erreichen. Für verschiedene Varianten flacher Automaten wurden bereits Verfahren entwickelt [PYvB96, RDT95]. In Abschnitt 7.5.6 werden weiterführende Ansätze diskutiert.

Wie bereits beschrieben, ist es im generellen Fall unentscheidbar, ob eine OCL-Bedingung erfüllbar ist. Wurde eine unerfüllbare OCL-Bedingung als Vorbedingung einer Transition eingesetzt, so kann kein Pfad gefunden werden, der die zugehörige Transition beinhaltet. Eine Überdeckung aller Transitionen ist daher unmöglich. Wegen diesen Unentscheidbarkeitsproblemen gibt es für die in der UML/P definierten Statecharts wie für viele andere Varianten von Automaten kein automatisches Verfahren, das für alle Formen eine nach einem bestimmten Kriterium vollständige Testfallsammlung generiert.

Ein weiteres Problem entsteht durch den möglichen Nichtdeterminismus im deskriptiven Statechart zum Beispiel durch überlappende Schaltbereiche. Hat ein Statechart zwei Transitionen mit identischen Schaltbereichen, aber unterschiedlichen Zielzuständen, wie zum Beispiel in Abbildung 3.38(a) dargestellt, so ist es möglich, dass eine Implementierung immer nur eine Alternative auswählt. Die zweite Transition wird daher nicht gewählt und kann

durch Tests nicht überdeckt werden. Auch die Überlappung von Schaltbereichen und die Bevorzugung einer Transition in einer Implementierung kann im Allgemeinen nicht automatisiert erkannt werden. Es ist daher bereits eine sinnvolle Unterstützung, wenn ein Werkzeug auf Basis vorhandener Tests misst, zu welchem Grad ein Statechart überdeckt wurde und gegebenenfalls auf Defizite hinweist.

Beiden Formen von Statecharts ist gemeinsam, dass die damit beschriebenen Transitionsfolgen eine gewisse Abstraktion der vollständigen Implementierung darstellen. So kann

- in einer Aktionen eine komplexe Anweisungsfolge einschließlich Schleifen und Verzweigungen eingebettet sein,
- ein einzelner Diagrammzustand einer Menge von Objektzuständen entsprechen und
- eine OCL-Bedingung aus mehreren in eine Disjunktion zusammengefassten alternativ erfüllbaren Klauseln bestehen.

Für detaillierte Tests sollte dies berücksichtigt werden. Dies kann geschehen, indem ähnlich zu [GH99] Überdeckungsmetriken für diese Elemente mit den nachfolgend genannten Verfahren für Statecharts kombiniert werden. Das heißt also, dass nach der Entwicklung der Tests auf Basis der Statechart-Struktur diese so verfeinert werden, dass die einzelnen Bestandteile von Bedingungen und Aktionen im Statechart ebenfalls überdeckt werden. Eine Alternative ist es, die Überdeckung nicht auf Basis des Statecharts, sondern des daraus generierten Codes zu messen.

Ein weiterer Aspekt ist die Einbeziehung der Abläufe in aufgerufenen Methoden desselben oder anderer Objekte. Insbesondere, wenn eine Kompositionsbeziehung zu anderen Objekten besteht, wie im Auktionsbeispiel zwischen dem `Auction`- und seinen `Policy`-Objekten, dann wird das Verhalten der abhängigen Objekte in die Tests des Kompositums einbezogen. Dadurch wächst aber die Zahl der notwendigen Tests stark an, da verschiedene Konfigurationen von Objektstrukturen und in jeder Konfiguration jeweils die Zustandsmodelle aller Objekte des Kompositums betrachtet werden können. Eine Abschätzung zeigt schnell, ob ein Versuch einer Überdeckung praktisch durchführbar ist. Die adäquate Wahl des zu testenden Teilsystems, des gewünschten Überdeckungskriteriums und die intelligente, aber dadurch leider manuelle Auswahl von Testfällen wird daher notwendig. Bereits in [Mye79] wird vermerkt, dass erfahrene Tester auch ohne explizit angewandte Systematik durch „Error Guessing“, also dem Erraten von potentiellen Fehlerquellen, gute Ergebnisse erzielen. [PKS02] aber fordern für viele Anwendungen oder zumindest kritische Systemteile, dass „Error Guessing“ vor allem als zusätzliche Technik zu mehr systematischen Vorgehensweisen zu sehen ist.

7.5.4 Überdeckungsmetriken

In Vernachlässigung der diskutierten Komplexitäten bei Aktionen und OCL-Bedingungen lassen sich folgende Metriken für die Überdeckung eines Statecharts durch eine Testsammlung identifizieren. Diese sind als Kontrollflussbasierte Metriken bereits in [FHNS02, Bal98] genannt:

Zustandsüberdeckung erfordert einen Testfall für jeden erreichbaren Zustand im Diagramm. Ein Testfall legt dabei eine Sequenz von Eingaben fest, die von einem Startzustand in diesen Zustand führt.

Transitionsüberdeckung erfordert einen Testfall für jede schaltbereite Transition. Die Sequenz von Eingaben beschreibt dabei den Pfad von einem Ausgangszustand bis zur Ausführung der Transition.

Pfadüberdeckung erfordert einen Test für jeden möglichen Pfad von einem Start- in einen Endzustand. Diese Form der Überdeckung umfasst die beiden vorhergehenden Formen. Wenn Schleifen im Automat existieren, ist die Menge der Pfade jedoch unendlich und damit eine Pfadüberdeckung praktisch undurchführbar.

Minimale Schleifenüberdeckung ist eine reduzierte Form der Pfadüberdeckung. Dabei wird bei Schleifen darauf verzichtet, diese mehrmals zu durchlaufen. Jede im Statechart vorkommende Schleife muss jedoch mindestens einmal durchlaufen werden.

Nach [Bal98] hat die Pfadüberdeckung keine praktische Relevanz, da sie bei Schleifen sofort zu einer unendlichen und damit nicht mehr durchführbaren Aufgabe wird. Demgegenüber ist die minimale Schleifenüberdeckung ein starkes und dennoch praktikables Kriterium, das für qualitativ hochwertige Software eingesetzt werden sollte. Bei Statecharts ohne Schleifen sind jedoch beide Metriken äquivalent. Ein Problem beider Metriken ist die Frage, ob ein im Statechart erkennbarer Pfad in einer Implementierung tatsächlich durchgeführt werden kann. Wird die Vorbedingung einer Transition durch Invarianten und dem bisherigen Verlauf eines Teilpfades nie erfüllbar, so lässt sich diese Transition nicht ausführen. Dies gilt um so mehr, wenn das durch ein Statechart modellierte Objekt in einen Testling aus mehreren Objekten eingebettet und daher nicht direkt zugänglich ist. Die Umgebung kann dann auch verhindern, dass eine erforderliche Eingabesequenz auftritt. Deshalb ist die Testumgebung jeweils geeignet zu wählen.

Um die Überdeckungsmetriken bezüglich einer Testsammlung zu ermitteln, werden die Abläufe der Tests auf Basis einer Instrumentierung des Codes protokolliert.

Die ursprünglich für flache Automaten entwickelten Überdeckungsmetriken können in dieser Form auch auf Statecharts mit hierarchischem Zustandskonzept angewandt werden. Durch die in Abschnitt 5.6.2, Band 1 diskutierten Transformationen auf Statecharts ist es allerdings auch möglich, die Tests auf Basis der expandierten Hierarchie durchzuführen. Dadurch wird die notwendige Testüberdeckung verfeinert, da zum Beispiel beim Auflösen

einer Transition in einen hierarchischen Zustand eine Vervielfachung dieser Transition stattfindet. Die Expansion kann also genutzt werden, um eine feinere Testüberdeckung zu erhalten.

Anhand des Statecharts aus Abbildung 7.14 werden die vier genannten Metriken in Abbildung 7.17 illustriert. Dabei werden strukturell identische, ikonisierte Formen des Ausgangsstatecharts verwendet, die den Durchlauf des Tests durch das Statechart illustrieren. Die drei Pfade der Zustandsüberdeckung reichen für die Transitionsüberdeckung nicht aus: Es muss ein vierter Ablauf hinzu genommen und zwei der vorhandenen Abläufe so modifiziert werden, dass auch die am Ende stattfindenden Transitionen überdeckt sind.

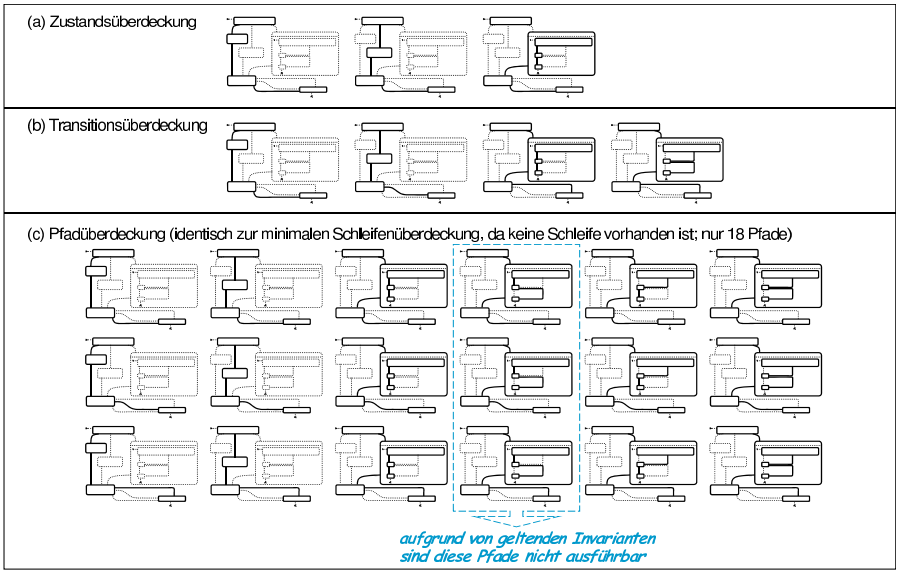


Abbildung 7.17. Testfallpfade zur Erfüllung der Metriken

Die Pfadüberdeckung erfordert 18 Testfälle. Da es sich bei dem zu testenden Statechart um ein Methoden-Statechart handelt, besteht die Eingabe aus nur einem Methodenaufruf. Gegebenenfalls werden weitere Interaktionen mit der Umgebung vorgenommen, indem Methoden der Umgebung aufgerufen werden und durch `return`-Werte eine weitere Steuerung des Transitionsverlaufs möglich ist. In diesem Beispiel aber hängt der Verlauf des Testfalls ausschließlich von der initialen Eingabe und der Wertebelegung in der initialen Objektstruktur ab. Um eine 100%ige Pfadüberdeckung zu erreichen, sind also 18 verschiedene Belegungen für die im Statechart benutzten Variablen zu finden. Die Invariante

```
context Auction a inv:
  MIN_DELTA <= a.extensionTime
```

OCL

fordert zum Beispiel, dass die `extensionTime` einer Auktion immer über der minimalen Erweiterungszeit von `MIN_DELTA` (5 Sekunden) liegt. Deshalb sind die markierten drei Pfade nicht durchführbar. Für die anderen 15 Pfade lassen sich geeignete Testdaten finden. Zusätzlich ist es sinnvoll, Randfälle, wie die Ankunft eines Gebots exakt zum Zeitpunkt des Auktionsendes oder minimal danach, zu behandeln. Weitere Randfälle lassen sich durch die Analyse der Schaltbedingungen von Transitionen identifizieren und so weitere Tests ableiten.

Als zweites Beispiel werden die Metriken anhand des in Abbildung 7.16 dargestellen Statecharts demonstriert. Dort ist jede Transition durch einen Methodenaufruf zu steuern, die Eingabe ist also eine mehrteilige Sequenz.

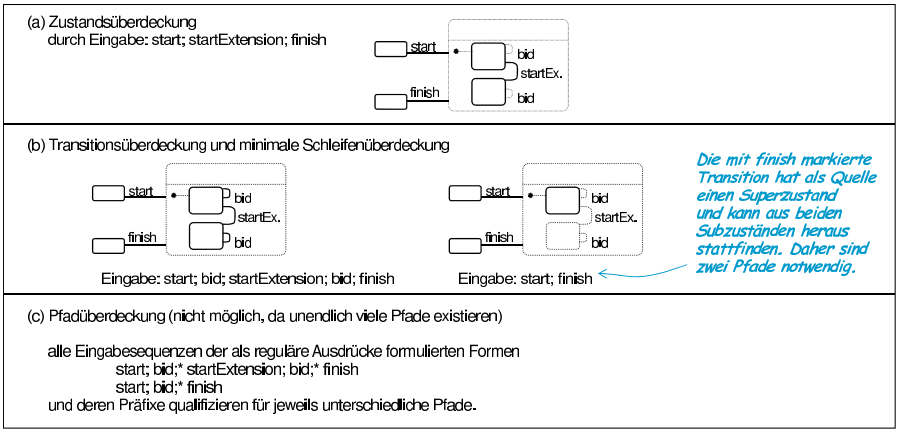


Abbildung 7.18. Testfälle zur Erfüllung der Metriken

Die Möglichkeit, Statecharts durch Verwendung verschiedener Stereotypen eine unterschiedliche Semantik in Bezug auf Vervollständigung und Fehlersituationen zu geben, erfordert bei der Definition von Tests die Berücksichtigung dieser Stereotypen. Folgende Situationen und Strategien sind dabei möglich:

1. Das Statechart ist vollständig, indem zum Beispiel ein Fehlerzustand eingeführt wurde. Dadurch existieren implizite Transitionen, die in den Fehlerzustand führen und deren Schaltbereiche alles abdecken, was nicht durch explizite Transitionen abgedeckt ist. Diese Transitionen können in die Überdeckung einbezogen werden. Da Statecharts einen zweiten Fehlerzustand für die Behandlung aufgetretener Exceptions besitzen können, ist die hier auftretende Situation analog. Es ist wieder zu

entscheiden, ob und wie detailliert die Verarbeitung der Exceptions getestet werden soll.
2. Das Statechart wurde mit «completion:ignore» vervollständigt. Dadurch entstehen Transitionsschleifen, die bei Transitions- und Pfadüberdeckung getestet werden können.
3. Ein mit «completion:chaos» markiertes Statechart basiert auf der Annahme, dass das Statechart nur einen Teil des Verhaltens festlegt. Es beschreibt das Verhalten des Objekts bis zu dem ersten Auftreten einer Situation, in der das Statechart nicht schaltbereit ist und der Testling beliebiges Verhalten annehmen kann. Derartige Abläufe müssen nicht getestet werden.

Bei einer Vervollständigung mittels «completion:ignore» wird besonders offensichtlich, dass ein Test dieser zusätzlichen Transitionen für das Verhalten des mit dem Statechart beschriebenen Objekts nicht sehr wesentlich ist, da diese Transitionen in uniformer Weise generiert sind. Wesentlich interessanter ist es daher meistens, Objekte der Umgebung darauf zu testen, ob deren Umgang mit dem beschriebenen Objekt soweit korrekt ist, dass sie auf das Ignorieren eines Methodenaufrufs oder das Betreten eines Fehlerzustands ihrerseits robust reagieren. Bei der Annahme, dass ein Statechart durch die Unvollständigkeit alle erlaubten Abläufe eines Objekts beschreibt, entstehen dadurch sogar wesentliche Einschränkungen an die Umgebung, die zumindest in Form von Tests zu prüfen sind. Das kann zum Beispiel durch eine Vervollständigung mit Fehlerzustand umgesetzt werden, bei der im Fehlerzustand ein Scheitern des Tests gemeldet wird.

7.5.5 Transitionstests statt Testsequenzen

Bei den bisher diskutierten Testsequenzen wird grundsätzlich davon ausgegangen, dass als Testdatensatz das beschriebene Objekt in einem Initialzustand vorliegt. Um daher eine Transition zu testen, ist zunächst ein Pfad zu finden, der zum Quellzustand dieser Transition führt.

Die mit Abbildung 6.7 charakterisierte Grundstruktur von Testfällen zeigt, dass ein Testfall von einem beliebigen Testdatensatz ausgehen kann. Dies bedeutet, dass eine Transition eines Statecharts auch dadurch getestet werden kann, dass ein Testdatensatz gefunden wird, der dem Quellzustand der Transition entspricht und diese schaltbereit macht. Dies geht vor allem bei dem Lebenszyklus eines Objekts, bei dem ein Methodenaufruf einer einzelnen Transition entspricht und somit isoliert ausgeführt werden kann. Statt also eine Testsequenz beginnend mit einem Initialzustand zu erfordern, reicht es für einen Transitionstest aus, einen einfachen Methodenaufruf durchzuführen.

Bei einer Automatisierung der Generierung von Tests wird durch die Definition von Testdaten, die einem Quellzustand einer Transition entsprechen,

das bereits in Abschnitt 7.4 diskutierte Problem umgangen, dass eine Testsequenz zu finden ist, die zu einem bestimmten Zustand beziehungsweise einer Transition führt. Stattdessen ist eine Objektstruktur zu finden, in der der Testling einem Statechart-Zustand entspricht. Es ist deshalb notwendig Diagrammzustände mit OCL-Bedingungen ausreichend genau zu charakterisieren. Sonst kann es bei diesem Ansatz vorkommen, dass ein Objektzustand für die Ausgangsdaten festgelegt wird, der im Produktionssystem nicht erreichbar ist. Bei der Suche von Objektzuständen, die einem Diagrammzustand entsprechen, sind außerdem auch die allgemein gültigen Invarianten zu beachten.

Damit ist die Suche nach Testdaten für Transitionen aus Statecharts äquivalent mit der bereits in Abschnitt 7.3.2 diskutierten Suche nach Testdaten, die die Vorbedingung einer OCL-Methodenspezifikation erfüllen. Wird ein Verfahren zur Identifikation von Testdaten, die eine OCL-Vorbedingung erfüllen, verwendet, so kann dieses Verfahren aufgrund der in Abschnitt 5.6.2, Band 1 vorgestellten Transformation von Statecharts in OCL auch für Transitionen in Lebenszyklen-Statecharts verwendet werden.

Ein Vorteil des hier skizzierten transitionsbasierten Ansatzes ist die einfache Prüfbarkeit jeder Transition, die auch in effizienter ablaufenden Tests mündet. Nachteil ist, dass nicht erkannt wird, ob ein Zustand überhaupt erreichbar ist. Von solchen Zuständen ausgehende Transitionen müssen nicht getestet werden, da sie keinen Beitrag zum Systemverhalten leisten.

Der skizzierte Ansatz ist äquivalent zur Transitionsüberdeckung, ignoriert aber die Pfade vom Startzustand bis zum Quellzustand der eigentlich durchzuführenden Transition.

7.5.6 Weiterführende Ansätze

Wie bereits erwähnt, ist es weiterhin ein Forschungsgebiet, aus automatenartigen Beschreibungstechniken mit verschiedenen Verfahren Testfälle zu ermitteln. Exemplarisch werden die folgenden Ansätze kurz skizziert, die den Fortschritt in diesem Bereich demonstrieren.

Für einfachere Varianten von zustandsbasierten Systemen ist zum Beispiel in [GH99] ein Verfahren angegeben, mit dem aus einem ausführbaren Automaten mithilfe von Model-Checking automatisierte Tests generiert werden können. Diese Tests prüfen eine Implementierung gegen den als Orakelfunktion verwendeten Automaten. Diese Automaten haben aber keine Zustandshierarchie, keinen Nichtdeterminismus und eine im Vergleich zu den hier präsentierten Statecharts eingeschränkte Form von Transitionsmarkierungen. In dem Verfahren wird eine Transitionsüberdeckung des Automaten erreicht. Model-Checking wird dabei vor allem verwendet, um Aufrufsequenzen für den Automaten zu finden, die Vorbedingungen der jeweils zu testenden Transition erfüllen. Dieses Verfahren wird außerdem auf zwei Arten verfeinert. Zum einen werden aus Disjunktionen bestehende Vorbedingungen zerlegt und damit implizit die Transitionen geteilt, um so eine

Überdeckung aller Klauseln einer Disjunktion zu erreichen. Zum anderen wird eine einfache Grenzwertanalyse gemacht, indem Randvergleiche wie etwa `a>=b` in zwei Fälle `a>b` und `a==b` zerlegt werden.

In [FHNS02] wird ein Ansatz beschrieben, aus einem Zustandsautomaten durch Projektion eine Verringerung des Zustandsraums zu erhalten. Dadurch wird eine Überdeckung des als Projektion erhaltenen Automaten durch Tests nach den Kriterien Zustands- und Transitionsüberdeckung möglich. Allerdings hängt es von der Wahl der Projektion ab, welche Pfade durch das System wirklich getestet werden und welche Fehler nicht entdeckt werden, weil sie über mehrere der projezierten Automaten verteilt sind. Dieses Verfahren ist hilfreich, wenn man ein Objekt mit vielen Zuständen hat. In objektorientierten Systemen ist es in diesem Fall ratsam (wenn auch nicht immer möglich), durch Auslagerung von Teilfunktionalität diese Projektion bereits beim Entwurf vorzunehmen. Als Nebeneffekt entstehen mehrere Objekte, deren Zustandsraum jeweils einer Projektion entsprechen kann. Das Projektionsverfahren aus [FHNS02] bietet aber zusätzliche Flexibilität, da es überlappende Projektionen erlaubt und so überlappende Sichten für Tests zulässt.

Der Ansatz in [CCD02] diskutiert ähnlich wie hier den Einsatz einer restringierten Form der UML-Statecharts für die Gewinnung von Tests. Für die Testdaten werden ebenfalls Objektdiagramme eingesetzt. Jedoch erklären die Autoren der noch in Weiterentwicklung befindlichen Arbeit mehr die Architektur einer auf XML basierenden Werkzeugkopplung, als die Vorgehensweise aus den hierarchischen Statecharts Testfälle zu generieren.

7.6 Zusammenfassung und offene Punkte beim Testen

Zusammenfassung

Qualitätssicherung ist ohne systematisches und diszipliniertes Entwickeln von Tests nicht möglich. Die Definition von Testfällen ist aber aufwändig, insbesondere da für geschäftskritische Systeme eine gute Überdeckung durch Testfälle notwendig ist. Die effiziente Entwicklung und Darstellung von *Testfällen* spielt daher eine wesentliche Rolle. Die Techniken der UML/P erlauben eine kompakte und übersichtliche Darstellung von Tests durch die Kombination verschiedener Diagramme zur Darstellung der *Testdaten*, des *Testtreibers* und des *Sollergebnisses* sowie von *Invarianten* des Systems. Diese Diagramme und Spezifikationen können unabhängig voneinander entwickelt werden, sind kompakter und leichter verständlich als Testcode und erleichtern daher die Wiederverwendung.

In einer Test-First-Vorgehensweise können mittels Sequenzdiagrammen zunächst Verhaltensmuster spezifiziert werden, die gegebenenfalls mit Ergänzungen als Testfallbeschreibungen einsetzbar sind, bevor eine Implementierung vorgenommen wird. Damit ist dieser Ansatz eine Erweiterung des

klassischen Vorgehens Sequenzdiagramme bei der frühen Anforderungserhebung einzusetzen.

Nach einer Implementierung ist die Definition weiterer Testfälle sinnvoll, um Rand- und Sonderfälle zu prüfen. Wie intensiv getestet wird, hängt von der gewünschten Qualität und der zugrunde liegenden Methodik ab. In einem agilen Ansatz werden vor allem an besonders kritischen und komplexen Stellen Metriken zur Analyse der Testqualität eingesetzt und sonst auf die Erfahrung der Testentwickler vertraut.

Ein UML/P-Modell kann Gegenstand des Tests sein, wenn das Modell *konstruktiv* zur Beschreibung des *Systems* verwendet wird. Ein Modell kann aber auch als *testbare Spezifikation* eingesetzt werden, wenn bereits eine Implementierung gegeben ist. Wie bereits die Verfahren zur Codegenerierung in Kapitel 4 und Kapitel 5 gezeigt haben, können einzelne Konzepte des Modells auch unterschiedliche Rollen übernehmen und je nach Ansatz der Generierung konstruktiv oder als Testcode eingesetzt werden.

Nach einer Einführung in die Begriffswelt der Testverfahren und einer kurzen Beschreibung zweier wesentlicher Testwerkzeuge wurde in diesem Kapitel demonstriert, wie UML/P-Diagramme zur Modellierung von Tests eingesetzt werden.

Trotz vorhandener Literatur zum Thema Testen von Software gibt es gerade beim *Konformitätstest* von Code, der aus graphisch notierten Implementierungsmodellen generiert wurde, eine Reihe von Verbesserungsmöglichkeiten und offenen Enden, von denen einige nachfolgend diskutiert werden. In diesem Buch sind außerdem Lasttests, Qualitätssicherungsmaßnahmen wie Inspektionen und Modellreviews oder die Vorgehensweisen für interaktive Tests unter Nutzerpartizipation zum Zweck der Abnahme nicht behandelt.

Entwicklungspotential bei der Testfallgenerierung

Die Anwendung von UML/P-Diagrammen und insbesondere OCL-Spezifikationen zur Generierung automatisierter Testfälle bietet noch großes Entwicklungspotential. Wie in Abschnitt 7.3 beschrieben, wäre eine möglichst effektive Generierung einer den Code ausreichend überdeckenden, aber möglichst kleinen Menge von Testfällen aus einer OCL-Methodenspezifikation hilfreich, um in noch effizienterer Form Testfälle zu entwickeln. Das gleiche gilt für die in Abschnitt 7.5 diskutierten Statecharts.

Im Kontext der UML und der OCL sind noch wenig Arbeiten zum Thema agiles Testen bekannt. Das liegt teilweise daran, dass lange davon ausgegangen wurde, dass die Testverfahren, die im Wesentlichen alle bereits für prozedurale Programmierung entwickelt wurden [Bei04, Lig90], nur auf Objektorientierung zu übertragen wären. Durch die Vererbung und die dynamische Bindung von Methoden entstehen aber grundsätzlich neue Problemstellungen.

Als weitere Problemquelle wird die Entkopplung der Sprache UML und der vielen, teilweise sehr unterschiedlichen Vorgehensmodelle auf Basis der UML genannt [BL02], die sehr unterschiedlich detaillierte und damit unterschiedlich testbare Modelle einsetzen.

Derzeit ist dennoch eine steigende Anzahl von Arbeiten zu erkennen, die verschiedene Testverfahren auf die UML übertragen, ohne allerdings das Potential für Code- und Testgenerierung bereits auszuschöpfen. [PJH+01] beschäftigen sich etwa mit der Übertragung der aus TTCN [ISO92] bekannten Verwendung von Sequenzdiagrammen zur Testfallmodellierung. [BL02] beschreiben die systematische und teilweise automatisierte Entwicklung von Testfällen aus UML-Analysedokumenten wie Use Case Diagrammen, Sequenz-, Kommunikations- und das Domänenmodell darstellenden Klassendiagrammen. In [BB00] wird ein ähnlicher Ansatz verfolgt, der ebenfalls eine Form von Sequenzdiagrammen zur Beschreibung von Interaktionen zwischen Objekten verwendet, und dies mit einem bereits bekannten Verfahren zur Kategorie-Partitionierung [OB88] kombiniert.

In [PLP01] wird beschrieben, wie für eine für verteilte Systeme geeignete, allerdings nicht an der UML angelehnte, graphische Modellierungssprache Techniken des Constraint Reasoning angewandt werden, um aus einer abstrakten *Testfallspezifikation* eine Sammlung von Testfällen (dort als „*Testsequenzen*“) bezeichnet, zu generieren. Allerdings ist die Form der betrachteten Systeme dort statisch und deshalb die Übertragung der Algorithmen auf dynamische, objektorientierte Systeme nicht kanonisch.

Bereits in den Publikationen [DN84, HT90] wurde beschrieben, dass die zufallsbasierte Generierung von Testdaten für Entwicklung von Zutrauen in die Korrektheit der Implementierung ähnlich gute Ergebnisse aufweist wie partitionsbasierte Testverfahren. Wenn diese Ergebnisse sich auch bei den heutigen Sprachen und Testverfahren bewahrheiten, dann bedeutet dies, dass die zufallsbasierte Generierung von Tests ein wesentliches Hilfsmittel für Testverfahren sein kann. Insbesondere Statecharts und OCL-Nachbedingungen sind dann als Orakel für solchermaßen generierte Tests hilfreich. Ist es erwünscht, die generierten Tests zu speichern, so können dafür Objekt- und Sequenzdiagramme verwendet werden. Wie bereits in [HT90] vermerkt, muss bei den Testverfahren eine weitere Verlagerung von der arbeitsaufwändigen manuellen Erstellung zur automatisierten Testgenerierung stattfinden.

Dieses Kapitel konnte nur einen kleinen Teil der insgesamt zum Thema Testen existierenden Konzepte, Techniken und Vorgehensweisen diskutieren. An verschiedenen Stellen wurde deshalb auf entsprechende Literatur verwiesen. Generell ist festzustellen, dass für eine effektive Unterstützung der Testfallentwicklung nicht nur für die UML gute und parametrisierte Generatoren existieren müssen, sondern auch Unterstützung für Testmuster sowie für einzubindende Frameworks und Komponenten anzubieten sind.

In objektorientierten und insbesondere agilen Vorgehensmodellen ist der Trend zu beobachten, dass das Produktionssystem explizit so strukturiert

wird, dass es gut testbar ist. Dadurch ist es nicht mehr notwendig eine Testsequenz zu definieren, die ausgehend von einem Initialzustand eine bestimmte Transition, Anweisung oder Methode prüft. Stattdessen kann eine Objektstruktur angegeben werden, die direkt zu dessen Prüfung führt. Objektstrukturen sind viel leichter zu finden, der Test wird effektiver in der Ausführung und besser verständlich.

Symbolisches Testen

Ein Beispiel für die potentielle Weiterentwicklung der Testverfahren basiert auf der Interpretation von symbolischen statt echten Werten. So kann die Verwendung von abstrakten, durch Symbole repräsentierten Elementen in Objektdiagrammen, wie in Abschnitt 4.2.2, Band 1 diskutiert, als Grundlage für eine interessante Erweiterung des Einsatzes dieser Diagramme dienen, die hier als Ausblick skizziert wird. Durch die Nutzung symbolischer Werte können Testfälle verallgemeinert werden. Dabei wird in einer Vorbedingung ein symbolischer Wert angegeben, der innerhalb der getesteten Methode verändert oder für die Berechnung anderer Attribute genutzt wird. Dabei wird der Ausdruck nicht ausgewertet, sondern unausgewertet als Term abgelegt. In der Nachbedingung kann dann statt eines konkreten Werts geprüft werden, ob der zur Berechnung verwendete Term der gewünschten Intention entspricht. Die einfachste Form des Vergleichs ist die syntaktische Gleichheit, jedoch kann durch geeignete algebraische Umformung auch ein semantisch äquivalenter Term angegeben werden. Ein Beispiel ist die folgende Berechnung, die auf umständliche Weise das Maximum beider Argumente ergibt:

```java
int foo(x,y) {
  int a = x;
  int b = y;
  a = a-b;
  if(x < y) b = b-a;
  return a+b;
}
```

Dabei wird statt mit konkreten Werten mit den symbolischen Werten `x` und `y` gerechnet. Unter der zusätzlichen Annahme `x<y` wird als Testergebnis für den Funktionswert `y` vorgegeben. Die symbolische Interpretation lässt sich durch folgende Annotation beschreiben:

```java
int foo(x,y) {                  // Wert a=      b=
  int a = x;                    //      x
  int b = y;                    //      x       y
  a = a-b;                      //      x-y     y
  if(x < y) b = b-a;            //      x-y     y-(x-y)
  return a+b;                   // Ergebnis: (x-y)+(y-(x-y))
}
```

Tatsächlich ist das Ergebnis `(x-y)+(y-(x-y))` äquivalent zu `y`. In analoger Form kann die Negation der Bedingung `!(x<y)` symbolisch „getestet" werden. Bei dieser Form der Interpretation erfolgt die Berechnung zumindest teilweise auf Basis symbolischer Werte. Damit entsteht eine Verallgemeinerung von exemplarischen Testfällen auf Äquivalenzklassen von Testfällen. Im obigen Beispiel sind dies nur zwei Äquivalenzklassen, beschrieben durch `x<y` und `!(x<y)`, so dass dadurch eine vollständige Testüberdeckung möglich wird. Deshalb kann symbolisches Testen ein Mittel sein, eine endliche Menge verallgemeinerter Tests zu entwerfen, die eine Überdeckung der gesamten Systemfunktionalität darstellen. Damit entsteht eine Brücke zwischen exemplarischen Testverfahren und allgemeinen Verifikationstechniken [Lig90]. Allerdings sind solche Verfahren gerade im objektorientierten und von Seiteneffekten behafteten Programmierstil aufwändig. Bereits die Verwendung von Schleifen macht die endliche Äquivalenzklassenbildung zunichte, so dass die Testüberdeckung in Bezug auf der Anzahl durchlaufener Schleifen unvollständig bleibt. Heute werden unausgewertete Datenstrukturen noch kaum beim Testen, sondern vor allem in Implementierungen für Logikprogrammiersprachen wie Prolog [Llo87], funktionalen Programmiersprachen mit „lazy"-Auswertung wie Gofer [Jon96], mathematisch algebraischen Systemen wie Maple [Viv01] oder Mathematica [Wol99], Termersetzungssystemen wie OBJ [GWM+92] oder Verifikationssystemen mit Termersetzung [NPW02, Pau94] erfolgreich eingesetzt. Sie werden dort aber auch für wesentlich komplexere Aufgaben wie der Verifikation eingesetzt.

Statistische, anwendungsbezogene Tests

Eine besondere Form der Testentwicklung ist beispielsweise im *Cleanroom*-Ansatz [PTLP98] zur Softwareentwicklung zu finden. Dort werden statistische Markov-Modelle verwendet, um die Wahrscheinlichkeiten für Anwendungsfälle zu beschreiben. Da Cleanroom primär mit Ein- und Ausgabesequenzen arbeitet, die durch Zustands-Transitions-Modelle beschrieben werden, werden durch das statistische Modell die potentiell vorkommenden Eingabesequenzen nach Wahrscheinlichkeiten gewichtet. Dadurch wird es möglich, die wahrscheinlichsten Anwendungen verstärkt zu testen. Im Kontext der UML kann dieses Verfahren auf Statecharts angewendet werden. Dazu werden auch hier die Transitionen gemäß der erwarteten auftretenden Häufigkeit gewichtet. Das heißt, es wird ein Modell der getesteten Klasse entwickelt, das beschreibt, welche Methodenaufrufe und Nachrichten mit welcher Wahrscheinlichkeit ankommen. Auf dieser Gewichtung basiert dann die Auswahl der Testpfade für das Statechart.

Alternativ, allerdings in Cleanroom selbst nicht propagiert, wäre auch die Gewichtung der Zustände nach der erwarteten Häufigkeit des Auftretens von Interesse. Außerdem lässt sich dasselbe Verfahren möglicherweise auch bei Klassendiagrammen gewinnbringend einsetzen, um eine

Gewichtung der Objektstrukturen zu erreichen, die als Testdatensätze einzusetzen sind.

Die Verwendung statistischer Verfahren erlaubt in Cleanroom die Vorhersage von durchschnittlichen Fehlerhäufigkeiten in Abhängigkeit der Anzahl durchgeführter Tests und der dabei gefundenen Fehler. Insbesondere kann so ein präzises Kriterium für das Testende in Abhängigkeit der gewünschten Qualität (Fehlerhäufigkeit) entwickelt werden. Eine der wesentlichen Grundvoraussetzungen dafür ist allerdings die Erhebung einer aussagekräftigen Menge an Daten, wie gut die so entwickelten Tests tatsächlich Fehler entdecken. Solche Daten hängen von der verwendeten Programmiersprache ab und können nur in geeigneten Feldversuchen erhoben werden.