Ein UML-basierter Testansatz zum Klassen- und Integrationstest objektorientierter Systeme

Dehla Sokenou
Technische Universität Berlin, Fachgebiet Softwaretechnik,
Sekr. FR 5-6, Franklinstr. 28/29, D-10587 Berlin, dsokenou@cs.tu-berlin.de

Abstract: Vorgestellt wird eine Technik für den Klassen- und Integrationstest objektorientierter Systeme. Es werden Sequenzdiagramme und UML-Statecharts zur Testfallgenerierung und Statecharts und OCL-Constraints zur Generierung des Testorakels kombiniert. Zur Einbindung des Testorakels in das zu testende System werden aspektorientierte Programmiertechniken verwendet.

1 Motivation

Der Zustandsraum eines objektorientierten Systems ist meist zu groß, um einen Test auch nur ansatzweise vollständig durchzuführen. Damit ist die Aussagekraft eines Tests gering. Um dieses Problem zu lösen, ist eine geeignete Auswahl der Testfälle nötig. Ein pragmatischer Ansatz ist die Konzentration auf typische Anwendungsfälle des Systems.

UML-Sequenzdiagramme beschreiben mögliche Nachrichtenfolgen zwischen Objekten in einem System. Sie stellen in der Regel typische Normal- und Fehlerfälle dar. Werden sie als Grundlage für den Test verwendet, kann insbesondere das Kommunikationsverhalten des zu testenden Systems untersucht werden. Auf den ersten Blick scheint der Test auf Basis von Sequenzdiagrammen intuitiv zu sein. Jedes Sequenzdiagramm definiert einen Testfall bzw. eine Menge von Testfällen, wenn einer der in der UML 2.0 [UML04] definierten Operatoren wie *alt* (Alternative), *opt* (Option), *loop* (Iteration) oder *par* (Parallelität) verwendet wird. Sequenzdiagramme definieren die an einem Szenario beteiligten Objekte und somit die Beziehungen von Objekten innerhalb eines Systems.

Es ist jedoch festzustellen, daß die modellierten Szenarien nur einen Teilausschnitt des Systemverhaltens darstellen und somit keine Aussage über die Vollständigkeit des Modells getroffen werden kann. Vielmehr ist bei großen Softwaresystemen in der Regel nur ein Bruchteil aller möglichen Szenarien modelliert worden. Hinzu kommen weitere Probleme. Meist gibt es weder einen zeitlichen Bezug (wann tritt das modellierte Verhalten auf?) noch sind die Voraussetzungen für ein Szenario gegeben (in welchem Zustand müssen sich die beteiligten Objekte befinden?).

Sequenzdiagramme enthalten in der Regel zu wenig Informationen, um als Basis für ein Testorakel zu dienen. Es gibt zwar den Operator *neg*, der Nachrichtenfolgen kennzeichnet, die nicht erlaubt sind, so daß das Auftreten einer entsprechenden Sequenz als fehlgeschla-

gener Test zu werten ist. Bei positiven Sequenzen[1] kann das Auftreten einer entsprechenden Sequenz als bestandener Test interpretiert werden. Reagiert das System aber auf eine Nachricht mit einer Sequenz, die nicht modelliert wurde, kann keine Aussage getroffen werden (*nicht entscheidbar*), da es sich sowohl um eine vergessene negative als auch um eine vergessene positive Sequenz handeln kann. Eine grundsätzliche Bewertung des Auftretens einer nicht modellierten Sequenz scheint ein nicht praktikables Vorgehen zu sein, wenn man berücksichtigt, daß meist ein Großteil aller Sequenzen nicht modelliert ist.

Andererseits können Sequenzdiagramme für den Integrationstest Informationen liefern, da sie Nachrichten modellieren, die zwischen verschiedenen Objekten im System ausgetauscht werden. Im Gegensatz dazu können Zustandsdiagramme (auch UML-Statecharts genannt) zwar als Basis sowohl für die Testfallgenerierung als auch für ein Testorakel dienen, beschreiben aber in der Regel nur das Verhalten eines Objekts oder einer Komponente und stellen das Kommunikationsverhalten der einzelnen Teile des Systems nicht dar.

Einen Teil der Probleme beim Test auf Basis von Sequenzdiagrammen oder auf Basis von Zustandsdiagrammen versucht der vorgestellte Ansatz zur Testfallgenerierung zu lösen. Die Sequenzdiagramme werden zur Testfallgenerierung und im begrenzten Maße auch für das Testorakel verwendet. Zusätzliche Informationen werden aus Zustandsdiagrammen und OCL-Constraints in Form von Vor- und Nachbedingungen für Methoden gewonnen. Diese erweitern das Testorakel und werden zur Initialisierung der an einem Szenario beteiligten Objekte herangezogen.

Um einen Test durchführen zu können, ist eine Integration von Testcode in das zu testende System notwendig. Testtreiber müssen das System gezielt steuern können, in dem sie es in einen bestimmten Zustand bringen und anschließend die Testfälle ausführen. Das Testorakel vergleicht Ausgaben und Zustände des zu testenden Systems mit den erwarteten Ausgaben und Zuständen und leitet daraus ein Testurteil ab. Sowohl Testtreiber als auch Testorakel müssen Zugriff auf das zu testende System besitzen und nach dem Test unter Umständen deaktiviert oder gar wieder entfernt werden. Klassischerweise wird der zusätzliche Code durch Instrumentierung in das zu testende System integriert. Unser Ansatz verwendet aspektorientierte Programmiertechniken [KLM+97] zur Integration des Testcodes, da so eine Unabhängigkeit von Testcode und zu testendem System gewahrt bleibt und der Testcode später sehr leicht wieder entfernt werden kann.

Zunächst führt Abschnitt 2 die verwendeten UML-Diagramme ein. In Abschnitt 3 wird die Technik der Testfallgenerierung vorgestellt, die auf Sequenzdiagrammen und Zustandsdiagrammen basiert. Anschließend wird in Abschnitt 4 ein Testorakel vorgestellt, das aus UML-Statecharts und OCL-Constraints in Form von Vor- und Nachbedingungen abgeleitet wird. In Abschnitt 5 wird ein kurzer Einblick in die aspektorientierte Integration des Testcodes in das zu testende System gegeben. Abschnitt 6 vergleicht unseren Ansatz mit verwandten Arbeiten und Abschnitt 7 faßt die Ergebnisse zusammen und gibt eine Ausblick auf unsere zukünftige Forschungsarbeit.

[1] Alle Sequenzen, die nicht mit dem Operator *neg* gekennzeichnet sind, werden hier als positive Sequenzen bezeichnet.

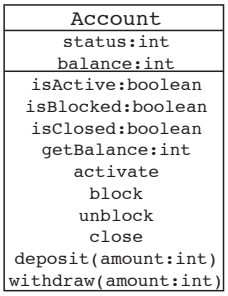

Abbildung 1: Klasse `Account`

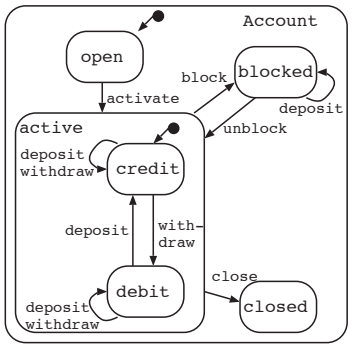

Abbildung 2: Statechart der Klasse `Account`

2 Sequenzdiagramme, UML-Statecharts und OCL-Constraints

Zunächst sollen die verwendeten UML-Diagramme und OCL-Constraints eingeführt und ihre Beziehung zu dem zu testenden System verdeutlicht werden. Dabei wird die Modellierung eines Banksystems als Beispiel verwendet. Abb. 1 zeigt eine statische Sicht der Klasse `Account`, die ein Bankkonto mit den Attributen `status` und `balance` und den Query-Methoden `isActive`, `isBlocked`, `isClosed` und `getBalance` modelliert. Bei den anderen Methoden handelt es sich um Update-Methoden. Die Unterscheidung zwischen Update- und Query-Methoden wird automatisch aus dem Statechart abgeleitet.

UML-Modelle können je nach Einsatzzweck unterschiedlich interpretiert werden. Die UML [UML04] läßt durch semantische Variationspunkte (*Semantic Variation Points*) verschiedene Auslegungen zu. Um auf der Basis von UML-Modellen testen zu können, ist eine Präzisierung dieser Variationspunkte nötig. Wir schränken darüber hinaus die Menge der zu verwendenden Diagramme ein.

So verwenden wir von den zwei Arten der UML-Statecharts die *Protocol State Machines*, die zur Spezifikation von Objektlebenszyklen benutzt werden. Protocol State Machines unterscheiden sich in einigen Punkten von den ebenfalls in der UML definierten *Behavioral State Machines*. Ereignisse (*Events*) in einer Protocol State Machine sind Methodenaufrufe (*Call Events*). Methoden, die zustandsverändernd sind (*Update-Methoden*), können nur in den Zuständen aufgerufen werden, in denen sie im Statechart eine Transition auslösen, in allen anderen Zuständen verletzt ihr Aufruf eine (implizite) Vorbedingung. Die UML definiert jedoch kein Verhalten des Systems im Falle der Verletzung einer impliziten Vorbedingung, so kann das System z.B. mit einem Fehler, einer Ausnahme oder dem Ignorieren des Ereignisses reagieren. Zustandsneutrale Methoden (*Query-Methoden*) werden im Statechart nicht modelliert und sind in jedem Zustand aufrufbar. Sie dürfen den Zustand des Objekts nicht verändern. Alle Methoden einer Klasse, die im zugehörigen Statechart nicht modelliert werden, behandeln wir als Query-Methoden. Die im Statechart referenzierten Methoden sind Update-Methoden. Betrachtet man die Form des Labels einer Transition, `Event[Condition]/Action`, so fehlt bei den Protocol State Machines der `Action`-Teil, da als implizite Aktion die Ausführung der Methode gilt, die dem

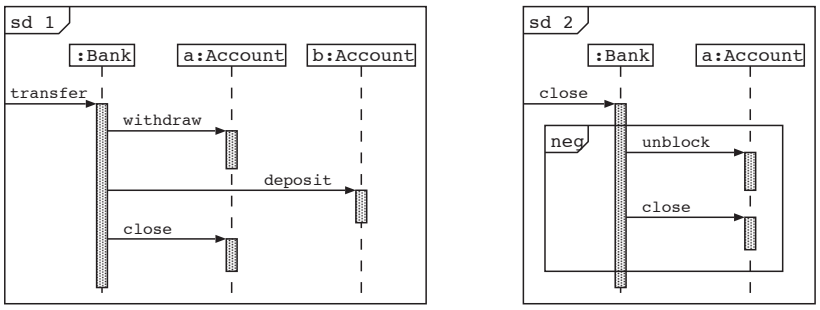

Abbildung 3: Positives und negatives Sequenzdiagramm

```
context Account::withdraw (amount:int)
   pre : true
   post: self.balance = self.balance@pre - amount
```

Abbildung 4: Vor- und Nachbedingungen für `withdraw`

Ereignis zugeordnet ist. Statt einer Aktion kann eine Nachbedingung spezifiziert sein. Wir setzen voraus, daß als `Event` ein Methodenname angegeben ist.

Das Beispiel in Abb. 2 zeigt die Protocol State Machine der Klasse `Account`. Die Query-Methoden `isActive`, `isBlocked` und `isClosed` sind den Zuständen des Toplevel-Statecharts zugeordnet. Befindet sich ein Objekt der Klasse `Account` beispielsweise im Zustand `active`, liefert die Query-Methode `isActive` den Wert `wahr` zurück. Der von der Query-Methode `getBalance` zurückgelieferte Wert bestimmt den Zustand im Sublevel-Statechart `active`. Ist dieser Wert größer oder gleich Null, befindet sich das Objekt im Zustand `credit`, ansonsten im Zustand `debit`. Diese Bedingungen sind in Form von Zustandsinvarianten in OCL spezifiziert.

Auch bei den Sequenzdiagrammen betrachten wir ausschließlich Nachrichten in Form von Methodenaufrufen. Das bedeutet eine Fokussierung auf synchrone Kommunikation. Abb. 3 zeigt zwei Sequenzdiagramme[2], ein positives (`sd 1`) und ein negatives (`sd 2`). Das positive Sequenzdiagramm modelliert ein Szenario, bei dem Geld von einem Konto auf ein anderes transferiert und das erste Konto anschließend geschlossen wird. Das negative Szenario soll verhindern, daß ein gesperrtes Konto geschlossen wird.

OCL-Constraints können in diversen Kontexten in UML-Modellen auftauchen, jedoch auch außerhalb eines Modells definiert sein. In jedem Fall sind sie immer einem Modellelement, z.B. einer Klasse, zugeordnet. Wir benutzen OCL-Constraints primär zur Spezifikation von Vor- und Nachbedingungen und Klasseninvarianten. Abb. 4 zeigt exemplarisch die Vor- und Nachbedingung der Methode `withdraw` der Klasse `Account` (siehe `context`-Deklaration). Die Vorbedingung ist wahr (im `pre`-Teil). Die Nachbedingung sagt aus, daß der Kontostand nach Ausführung der Methode um den ausgezahlten Betrag verringert sein muß (im `post`-Teil).

[2]Im Beispiel sind aus Übersichtsgründen keine Parameter gegeben. Die Methode `withdraw` besitzt z.B. den Parameter `amount` (siehe auch Abb. 4).

Ein wichtige Voraussetzung zur Anwendung unserer Technik ist die Korrektheit der Modelle und die Konsistenz zwischen den einzelnen Modellen. Einfachstes Kriterium ist die gleiche Benennung desselben Modellelements in allen Modellen und in der Implementierung. Auf Basis einer inkorrekten oder inkonsistenten Spezifikation ist ein Test nur wenig bis gar nicht aussagekräftig, die Ableitung von Testfällen und Testorakel kann sogar verhindert werden.

Nachdem nun alle verwendeten Formalismen eingeführt sind, wird im nächsten Abschnitt unsere Technik zur Generierung von Testfällen vorgestellt.

3 Testfallgenerierung aus Sequenzdiagrammen und Statecharts

Wir generieren Testfälle hauptsächlich für den Integrationstest, auf den hier näher eingegangen werden soll. Testfälle können jedoch auch für den Klassentest generiert werden, wobei die an einem Szenario beteiligten Objekte einzeln betrachtet werden. Die im Sequenzdiagramm an ein Objekt geschickten Nachrichten definieren dabei einen Testfall, wobei das zu testende Objekt zunächst, wie auch für den Integrationtest, zu initialisieren ist. Objekte, die Nachrichten an das zu testende Objekt schicken, werden durch einen Testtreiber ersetzt. Empfangen Objekte Nachrichten vom zu testenden Objekt, können diese wahlweise als reale Objekte oder als Minimalimplementierung (Stub oder Mock-Objekt) eingebunden werden.

Beim Integrationstest wird versucht, die modellierten Szenarien zu provozieren. Dabei wird die die Sequenz auslösende Nachricht vom Testtreiber an das erste zu testende Objekt der Sequenz geschickt. Im Beispiel in Abb. 3 wäre es im Sequenzdiagramm `sd 1` die Nachricht `transfer` an das unbenannte Objekt vom Typ `Bank` und im Sequenzdiagramm `sd 2` die Nachricht `close` an das unbenannte Objekt vom Typ `Bank`.

Wie bereits in [FL02] beschrieben, ist eine Initialisierung der am Szenario beteiligten Objekte notwendig. Anders als die dort beschriebenen Lösungen verfolgt der hier vorgestellte Ansatz die Gewinnung der Information zur Initialisierung aus den UML-Statecharts. Wird ein positives Sequenzdiagramm als Basis für die Testfallgenerierung verwendet, werden zunächst alle beteiligten Objekte passend initialisiert. Dazu werden durch Analyse des Statecharts Zustände gesucht, in denen ein bestimmtes Szenario ausführbar ist. Dabei gelten folgende Regeln:

- Handelt es sich um eine Query-Methode, so ist diese in jedem Zustand aufrufbar und damit ausführbar.
- Update-Methoden sind in einem Zustand nur dann ausführbar, wenn sie von einer Transition referenziert werden, die diesen Zustand als Ausgangszustand besitzt.
- Bei einer Folge von Nachrichten an ein Objekt muß die erste Nachricht in einem Zustand ausführbar sein und in einen Zustand führen, in dem die zweite Nachricht ausführbar ist. Dies ist rekursiv für alle Nachrichten der Nachrichtenfolge anzuwenden.

Für das Szenario `sd 1` aus Abb. 3 ist es nötig, daß das `Account`-Objekt `a` sich in einem Zustand befindet, in dem die Methodenfolge `withdraw` gefolgt von `close` ausgeführt

```
sd 1, Testfall 1
x:Bank{new};
a:Account{new};
        a.activate;
b:Account{new};
        b.activate;
x.transfer
```

```
sd 1, Testfall 2
x:Bank{new};
a:Account{new};
        a.activate;
b:Account{new};
        b.activate;
        b.block;
x.transfer
```

```
sd 2, Testfall 1
x:Bank{new};
a:Account{new};
        a.activate;
        a.block;
x.close
```

Abbildung 5: Reguläre Testsequenzen

werden kann, dies gilt analog für das `Account`-Objekt `b` und die Methode `deposit`. Aus dem Statechart für die Klasse `Account` in Abb. 2 läßt sich ermitteln, daß sich das Objekt `a` im Zustand `active` befinden muß, das Objekt `b` in einem der Zustände `active` oder `blocked`. Damit gibt es zwei mögliche gültige Testfälle für das Szenario: jeweils einen für jeden möglichen gültigen Zustand des Objekts `b`. Der Übersichtlichkeit halber wurde von den Zuständen der Bank abstrahiert.

Zu berücksichtigen ist, daß eine korrekte Initialisierung der beteiligten Objekte keine Garantie darstellt, daß das System sich verhält, wie die Modellierung der Sequenzdiagramme es vorsieht. Dies ist nur dann der Fall, wenn die modellierten Sequenzdiagramme das System vollständig modellieren. Unser Testorakel, vorgestellt in Abschnitt 4, ist unabhängig von der tatsächlichen Sequenz, mit der das System auf die Stimulation reagiert. Eine Erweiterung des Testorakels ist geplant, so daß auch die Sequenz einbezogen wird, mit der das System reagiert.

Um auch negative Sequenzen zu berücksichtigen, werden weitere Testfälle generiert. Reagiert das System auf die Stimulation mit einer negativen Sequenz, so ist der Test fehlgeschlagen. Dazu sind analog zu der beschriebenen Technik für positive Sequenzen alle beteiligten Objekte in einen Zustand zu bringen, in dem die negative Sequenz (potientiell) ausgeführt werden könnte. Im Beispiel ist dieses für das Objekt `a` im Zustand `blocked` der Fall.

Die beteiligten Objekte werden nun initialisiert, indem aus ihren Statecharts Informationen über die Initialisierung gewonnen werden. Dazu wird zunächst eines der zu testenden Objekte erzeugt und eine Methodensequenz an das Objekt geschickt, bis es sich in dem gewünschten Zustand befindet. Für alle Zustände eines Objekts werden die entsprechenden Sequenzen zur Initialisierung vorab erzeugt und gespeichert. Es wird ein einfacher Traversierungsalgorithmus benutzt, der vom Initialzustand aus allen Transitionen folgt und dadurch alle (erreichbaren) Zustände besucht. Für jeden besuchten Zustand wird die berechnete Transitionsfolge in einer Tabelle gespeichert. Alle weiteren beteiligten Objekte werden analog initialisiert. Die entsprechenden Testsequenzen zeigt Abb. 5.

Die bisher beschriebenen erzeugten Testfälle werden von uns *reguläre Testfälle* oder, in Anlehnung an die positiven Sequenzdiagramme, *positive Testfälle* genannt. Sind Bedingungen an den Transitionen des Statecharts spezifiziert, werden diese im Testfall vermerkt (Beispiel: `a.withdraw[amount>0]`). Als Basis kann auch das um die OCL-Constraints angereicherte Statechart aus Abschnitt 4 verwendet werden. Die annotierten Bedingungen dienen der leichteren Bestimmung von Testdaten.

Es ist für den Tester möglich, neben diesen regulären Testfällen weitere Testfälle zu erzeugen, von uns *komplementäre Testfälle* genannt, um das System auf Robustheit zu testen. Komplementäre Testfälle berücksichtigen alle Zustände, in denen eine Sequenz die implizite Vorbedingung aus dem Statechart verletzt. Das sind alle Zustände, die die Objekte nicht in den regulären Testfällen annehmen, z.B. für das Objekt `a` im Sequenzdiagramm `sd 1` die Zustände `open`, `blocked` und `closed`.

Probleme bestehen hauptsächlich bei der automatischen Erzeugung von Testdaten, sei es die Erzeugung von Objektwelten oder die Bestimmung von aktuellen Parametern. Für beide Arten der Testdatenerzeugung sind Lösungen angedacht, jedoch noch nicht vollständig integriert. Testdaten für Basistypen können bereits aus den OCL-Constraints durch Äquivalenzklassenbildung gewonnen werden, Objekte müssen bisher manuell erzeugt werden. Objektwelten könnten zukünftig aus Objektdiagrammen automatisch erzeugt werden.

Die erzeugten Testfälle lassen sich auch für Objekte von Unterklassen der am Szenario beteiligten Klassen verwenden, unter der Voraussetzung, daß konform vererbt wurde. Das Testorakel für die Testfälle wird durch die in Abschnitt 4 vorgestellte Technik erzeugt.

4 Kombination von Statecharts und OCL-Constraints als Testorakel

Sowohl Statecharts als auch OCL-Constraints scheinen eine gute Basis zu sein, um sie als Testorakel zu verwenden. Statecharts, insbesondere die UML Protocol State Machines zur Spezifikation von Objektlebenszyklen, definieren erlaubte Zustände und Zustandsübergänge. Ein Aufruf einer Methode in einem Zustand, der im Statechart nicht modelliert wurde, verletzt die implizite Vorbedingung dieser Methode. Auch OCL-Constraints können zur Spezifikation von Vorbedingungen verwendet werden. Beide Vorbedingungen können kombiniert werden. Da die UML die Reaktion des Systems auf eine verletzte Vorbedingung als semantischen Variationspunkt deklariert, wird diese in unserem Test nicht als fehlgeschlagener Test gewertet, sondern aufgezeichnet, mit einer Warnung versehen und der Auswertung durch den Tester überlassen. Zukünftig ist die Auswahl einer Semantik vor dem Test denkbar, so daß der Test an dieser Stelle automatisch ausgewertet werden kann.

Eine Transition, die im Statechart in einen anderen Zustand führt als im zu testenden Objekt, ist als fehlgeschlagener Test zu werten. Der Nachfolgezustand einer Transition ist eine implizite Nachbedingung der die Transition auslösenden Methode. Auch die Verletzung einer in OCL spezifizierten Nachbedingung (unter der Annahme, daß die Vorbedingung vom Client der Methode erfüllt wurde), führt zu einem Fehlschlagen des Tests.

Vor- und Nachbedingungen aus dem Statechart und den OCL-Constraints müssen laut UML-Semantik beide gelten und werden mit einem logischen *UND* verknüpft. Gleichzeitig muß natürlich sowohl vor als auch nach dem (externen) Aufruf einer Methode die Klasseninvariante gelten. Auf die Darstellung der Integration der Klasseninvarianten wurde hier verzichtet, sie erfolgt analog zu der Integration von Vor- bzw. Nachbedingung.

Die Verknüpfung von Vor- und Nachbedingungen kann prinzipiell auf zwei Arten erfolgen. Entweder werden die Vor- und Nachbedingungen aus dem Statechart gewonnen und

Zustand	active	credit	debit
Invariante	self.isActive	self.balance >= 0	self.balance < 0

Abbildung 6: Zustandsinvarianten der Zustände `active`, `credit` und `debit`

```
context Account::withdraw (amount:int)
  pre : self.isActive
  post: (self.balance@pre >= 0 implies self.balance >= 0 or self.balance < 0)
         and (self.balance@pre < 0 implies self.balance < 0)
         and self.isActive
```

Abbildung 7: Vor- und Nachbedingungen für `withdraw` aus dem Statechart

mit den ursprünglichen OCL-Constraints zu neuen Constraints verknüpft oder es werden Vor- und Nachbedingungen aus den OCL-Constraints in das Statechart integriert. In unserer Arbeit haben wir beide Wege betrachtet, da sich unterschiedliche Vor- und Nachteile ergeben.

Zunächst wird die erste Variante vorgestellt, die Gewinnung von Vor- und Nachbedingungen aus dem Statechart. Als Beispiel betrachten wir wieder die Methode `withdraw`. Im Statechart (siehe Abb. 2) werden zunächst alle Zustände identifiziert, in denen die Methode `withdraw` als Ereignis einer Transition referenziert wird. Im Beispiel sind das alle Unterzustände des Substatecharts `active`. Die Zustandsinvarianten des Zustands `active` und seiner Unterzustände `debit` und `credit` sind in Abb. 6 dargestellt. Anschließend werden für alle Ausgangszustände der Transitionen Implikationen generiert, die den Zusammenhang zwischen dem Zustand vor und den Zuständen nach Ausführung der Transition beschreiben.

Die gewonnene Vor- und Nachbedingung ist in Abb. 7 dargestellt. Die Vorbedingung (im `pre`-Teil) erlaubt ein Abheben nur, wenn sich das `Account`-Objekt im Zustand `active` befindet (Zustandsinvariante `self.isActive`). Die Nachbedingung im `post`-Teil besteht aus zwei Implikationen. Die erste Implikation behandelt den Fall, daß sich das Objekt vor dem Aufruf im Zustand `credit` befand (Zustandsinvariante vor dem Aufruf `self.balance@pre >= 0`). Anschließend befindet sich das Objekt entweder im Zustand `debit` (Zustandsinvariante nach Ausführung `self.balance < 0`) oder weiterhin im Zustand `credit` (Zustandsinvariante nach Ausführung `self.balance >= 0`). Die zweite Implikation behandelt den Fall für den Ausgangszustand `debit` analog, wobei es hier nur einen Folgezustand gibt. Gleichzeitig zu den Implikationen muß sich das System aber weiterhin im Zustand `active` befinden. Der Übersichtlichkeit halber ist die Zustandsinvariante des Zustands `active` herausgezogen worden. Der Algorithmus zur Generierung der Constraints erzeugt sie als Teil der Implikationen. Zur Kombination werden nun die aus dem Statechart gewonnene Vorbedingung und die ursprüngliche Vorbedingung mit einem logischen *UND* verknüpft (analog die Nachbedingungen).

Bei der Integration von Vor- und Nachbedingungen aus dem Statechart in die OCL-Constraints verliert man einen Teil der Statechart-spezifischen Information über Zustände und Transitionen, die zur Auswertung des Tests und Behebung der Fehler hilfreich sein kann. Vorteil dieser Vorgehensweise ist die Generierung eines relativ kleinen Testorakels, das

für jeden Methodenaufruf einer Klasse nur die Einhaltung der Bedingungen vor und nach der Ausführung der Methode überprüft.

Die zweite Variante ist die Integration der OCL-Constraint in das Statechart. Das so erzeugte Statechart nennen wir angereichertes Statechart. Seien `withdraw` die betrachtete Methode, `pre(withdraw)` die Vorbedingung und `post(withdraw)` die Nachbedingung von `withdraw` in OCL. Jedes Label der Form `withdraw[c1]/[c2]` wird in das Label `withdraw[c1` *and* `pre(withdraw)]/[c2` *and* `post(withdraw)]` im angereicherten Statechart überführt. Hat das ursprüngliche Label keine Bedingungen, so wird zunächst `true` für die fehlenden Bedingungen eingesetzt.

Bei der Integration der OCL-Constraints in die Statechart-Spezifikation bleibt die Statechart-spezifische Information erhalten und wird nur durch zusätzliche Bedingungen ergänzt. Zudem kann das angereicherte Statechart als Basis der Testfallgenerierung dienen. Ein Nachteil ist, daß das Statechart aufgebläht wird, da an jedem Event, also jedem Methodenaufruf, Information zu ergänzen ist. Die Implementierung des Testorakels erhält, soweit möglich, alle Statechart-Eigenschaften (siehe auch Abschnitt 5), so daß ein eher komplexes Testorakel bereits aus den ursprünglichen Statecharts erzeugt wird. Die Integration der OCL-Constraints erhöht die Komplexität weiter. Bei der Entscheidung für eine der beiden Varianten haben wir immer auch die ursprüngliche Spezifikation betrachtet. So macht es bei einem Statechart mit nur einem Zustand wenig Sinn, die (doch recht mangelhafte) Information zu erhalten. Um eine heuristische Aussage über die Wahl der Variante zu treffen, ist jedoch zunächst eine umfangreiche Fallstudie nötig, in der beide Varianten gegenübergestellt werden. Die Durchführung einer größeren Fallstudie steht noch aus.

5 Aspektorientierte Integration des Testorakels

Nachdem sowohl Testfälle als auch Testorakel generiert wurden, werden diese in das zu testende System integriert. Die Integration erfolgt mit Hilfe aspektorientierter Programmiertechniken. Hier stellen wir die Integration des Testorakels vor.

Aspektorientierte Programmierung [KLM+97] ist eine Erweiterung objektorientierter Programmierung und wird eingesetzt, um verschiedene Concerns eines Systems in einzelne Module, die Aspekte genannt werden, zu kapseln und auf diese Weise von der eigentlichen Business-Logik des Systems zu trennen. Ein sogenannter Aspektweber integriert die einzelnen Aspekte wieder in die Business-Logik. Vorteil der aspektorientierten Integration des Codes ist die Trennung einzelner Concerns und die unabhängige Weiterentwicklung und Pflege von Concerns und Business-Logik, da der Source-Code durch den Aspektweber nicht verändert wird, sondern nur der kompilierte oder der Laufzeit-Code. Grundlage unserer Arbeiten ist die aspektorientierte Sprache ObjectTeams/Java [Her02, Obj], die Vorteile gerade in Bezug auf die Integration von Testcode bietet. Eine Adaption[3] von Bytecode ist möglich. Dieser muß anders als z.B. in AspectJ [KHH+01] nicht einmal neu kompiliert werden, sondern kann zum Beispiel auch in Form eines Java-Archiv (jar)

[3] Als Adaption wird die Veränderung von bestehenden Systemen durch zusätzlichen Code bezeichnet, der durch einen Aspektweber eingefügt wird.

vorliegen, das durch den Weber nicht verändert wird. Dies erleichtert das Versionsmanagement, da nicht zwei verschiedene Versionen des zu testenden Systems zu pflegen sind, die Version mit und die Version ohne Testcode. Zudem können Aspekte, also die Wirkung des zusätzlichen Codes, zur Laufzeit explizit ein- und ausgeschaltet werden.

Zur Integration der erweiterten OCL-Constraints als Testorakel wird zunächst durch einen einfachen OCL-2-Java-Generator Java-Code erzeugt. Dieser Code wird durch Methoden ausgewertet, die durch das ObjectTeams/Java-Laufzeitsystem vor und nach der Ausführung der zu testenden Methode aufgerufen werden. Diese Zwei-Schritt-Strategie hat mehrere Vorteile. Es kann sehr einfach ein anderer OCL-2-Java-Generator eingesetzt werden[4]. Auch die eigentliche Zielsprache, ObjectTeams/Java, kann durch eine andere Sprache ersetzt werden, z.B. durch AspectJ.

Die angereicherten Statecharts werden ebenfalls in ObjectTeams/Java implementiert und aspektorientiert in das zu testende System eingebunden. Grundlage sind die in [SH04] vorgestellten Ideen zur Implementierung eines Testorakels, das aus Statecharts generiert wird und die Statecharts-Eigenschaften Hierarchie, Historie, Parallelität und begrenzt auch Nichtdeterminismus unterstützt. Dabei werden Statecharts ausführbar gemacht und überwachen das System zur Laufzeit. Alle Methodenaufrufe an ein Objekt des Systems werden durch das ObjectTeams/Java-Laufzeitsystem an das korrespondierende Statechart-Objekt weitergeleitet und dort sowohl vor der Methodenausführung (Einhaltung der impliziten Vorbedingung) und nach der Methodenausführung (Einhaltung der impliziten Nachbedingung) überprüft.

Die Nutzung aspektorientierter Programmiertechniken bietet einige Vorteile gegenüber klassischer Instrumentierung, vor allem sinkt der Aufwand für Versionsverwaltung.

6 Verwandte Arbeiten

In [FL02] wird eine Technik zum Test objektorientierter Systeme auf Basis von Sequenzdiagrammen vorgestellt. Die Definition von fehlgeschlagenen oder bestandenen Tests basiert dort auf Sequenzdiagrammen in Kombination mit Vor- und Nachbedingungen von Methoden. Nicht modellierte Sequenzen werden als fehlgeschlagener Test (anstatt als nicht entscheidbarer Test) gewertet. Die Initialisierung der Testszenarien muß von außen erfolgen. Anders als in unserer Arbeit wird dort jedoch eine Lösung für das Testdatenproblem vorgeschlagen.

In [BL01] wird ein Testansatz zum objektorientierten Systemtest auf der Basis von UML-Modellen vorgestellt. Anders als in unserem Ansatz werden dort zusätzlich Klassendiagramm und Use-Cases verwendet, jedoch keine Statecharts. Der Ansatz fokussiert auf den Systemtest statt auf Klassen- und Integrationstest. Auch diese Technik ist noch nicht vollständig automatisiert, obwohl dies in Zukunft angestrebt wird.

Testansätze auf der Basis von Statecharts gibt es viele, examplarisch seien hier nur zwei

[4] Da unser OCL-2-Java-Generator nicht alle OCL-Konstrukte beherrscht, ist ein Einsatz von DresdenOCL [Dre] angedacht.

erwähnt, die zwei häufig verwendete Strategien zur Generierung von Testfällen aus Statecharts repräsentieren. In [KSG+94] werden Testfälle durch Traversierung des Statecharts erzeugt und führen zu einer vollständigen Überdeckung aller Zustände und Transitionen. [SHS03] beschreibt eine Technik, bei der durch nichtdeterministische Auswahl der Transitionen die Menge der Testfälle beliebig groß und Zyklen mehrfach durchlaufen werden können. Unsere Technik dagegen trifft die Auswahl der Transitionen, die im Statechart durch den Test ausgeführt werden sollen, auf Basis der Sequenzdiagramme.

Die aspektorientierte Integration von Testcode wird auch von [BAMR03], [Les], [RG03] und [BDL04] vorgeschlagen. [Les] integriert Mock-Objects mit Hilfe von AspectJ in das zu testende System. Zur Integration eines Testorakels benutzen [RG03] und [BDL04] AspectJ zur Laufzeitüberprüfung von OCL-Constraints und [BAMR03] zur Implementierung eines Statechart-basierten Testorakels. Während diese Ansätze sich auf die Integration der Testorakel konzentrieren, generiert unsere Technik darüber hinaus Testfälle für das zu testende System. Eine genauere Abgrenzung zu unseren Arbeiten findet der interessierte Leser in [SH04] und [VS05].

7 Zusammenfassung und Ausblick

Die vorgestellte Technik kombiniert Sequenzdiagramme und Statecharts zur Generierung von Testfällen und Statecharts und OCL-Constraints zur Erzeugung eines Testorakels. Der Vorteil ist die Berücksichtigung unterschiedlicher Sichten auf ein System und die Zusammenführung der über verschiedene Modelle verteilten Information zum Test. Der Tester kann sehr leicht durch die Modellierung weiterer Sequenzdiagramme neue Testfälle erzeugen. Auch wenn nur Sequenzdiagramme vorliegen, ist durch die Spezifikation eines Statecharts oder von Vor- und Nachbedingungen ein Test möglich. Beide Techniken, die Testfallgenerierung und das Testorakel, werden unabhängig voneinander aus den UML-Modellen abgeleitet und eingesetzt. So können Testfallgenerierung und Testorakel unabhängig voneinander erweitert werden, indem weitere UML-Modelle herangezogen werden. Die aspektorientierte Integration des Testcodes in das zu testende System bietet ebenfalls Vorteile. So kann der Testcode unabhängig vom zu testenden Code verwaltet werden. Durch den Einsatz von ObjectTeams/Java ist sogar der Test von Systemen möglich, die nicht im Source-Code vorliegen.

Unsere zukünftige Arbeit konzentriert sich darauf, die Schwächen unseres Ansatzes zu beheben. Ein Problem sind Testdaten. Bisher können Testdaten aus den OCL-Constraints nur für Basistypen durch Äquivalenzklassenbildung gewonnen werden. Andere Testdaten müssen manuell vom Tester definiert werden. Darüber hinaus planen wir den Einsatz effizienterer Werkzeuge. So soll die Generierung von Java-Code aus OCL-Constraints durch Einbindung von DresdenOCL [Dre] erfolgen. Ein Modelchecker soll zukünftig die Zustände bestimmen, in denen eine Nachrichtenfolge ausgeführt werden kann, und ein Homing-Algorithmus den Weg zu einem bestimmten Zustand im Statechart finden. Wir planen außerdem die Erweiterung unserer Techniken auf weitere UML-Diagramme, insbesondere die anderen Interaktionsdiagramme, und die Berücksichtigung anderer OCL-Constraints als Vor-, Nachbedingungen und Invarianten.

Literatur

[BAMR03] Jean-Michel Bruel, Joao Araújo, Ana Moreira und Albert Royer. Using Aspects to Develop Built-In Tests for Components. In *AOSD Modeling with UML Workshop, 6th UML Conference*, San Francisco, USA, 2003.

[BDL04] Lionel C. Briand, Wojciech Dzidek und Yvan Labiche. Using Aspect-Oriented Programming to Instrument OCL Contracts in Java. Bericht, Carlton University, Kanada, 2004.

[BL01] Lionel C. Briand und Yvan Labiche. A UML-Based Approach to System Testing. In *Proc. of 4th UML Conference*, Toronto, Kanada, 2001.

[Dre] Dresden OCL-Homepage. http://dresden-ocl.sourceforge.net/.

[FL02] F. Fraikin und T. Leonhardt. SeDiTeC - Testing Based on Sequence Diagram. In *Proc. of ASE*, Edinburgh, Großbritannien, 2002.

[Her02] Stephan Herrmann. Object Teams: Improving Modularity for Crosscutting Collaborations. In *Objects, Components, Architectures, Services, and Applications for a Networked World (Net.ObjectDays Conference)*, Jgg. 2591 of *Lecture Notes In Computer Science*, Erfurt, Deutschland, 2002. Springer-Verlag.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm und William G. Griswold. An Overview of AspectJ. In *Proc. of 15th ECOOP*, Jgg. 2072 of *Lecture Notes in Computer Science*, Budapest, Ungarn, 2001. Springer-Verlag.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Christina Videira Lopes, Jean-Marc Loingtier und John Irwin. Aspect-Oriented Programming. In *Proc. of 11th ECOOP*, Jgg. 1241 of *Lecture Notes in Computer Science*, Jyväskylä, Finnland, 1997. Springer-Verlag.

[KSG+94] David C. Kung, Nimish Suchak, Jerry Gao, Pei Hsia, Yasufumi Toyoshima und Cris Chen. On Object State Testing. In *Proc. of 18th COMPSAC*, Taipeh, Taiwan, 1994. IEEE Computer Society Press.

[Les] Nicholas Lesiecki. Test Flexibility with AspectJ and Mock Objects. http://www-106.ibm.com/developerworks/java/library/j-aspectj2/?loc=j.

[Obj] Object Teams-Homepage. http://www.objectteams.org.

[RG03] Mark Richters und Martin Gogolla. Aspect-Oriented Monitoring of UML and OCL Constraints. In *AOSD Modeling With UML Workshop, 6th UML Conference*, San Francisco, USA, 2003.

[SH04] Dehla Sokenou und Stephan Herrmann. Using Object Teams for State-Based Class Testing. Bericht, Technische Universität Berlin, Fakultät IV - Elektrotechnik und Informatik, Deutschland, 2004.

[SHS03] D. Seifert, S. Helke und T. Santen. Test Case Generation for UML Statecharts. In *Perspectives of System Informatics (PSI)*, Jgg. 2890 of *Lecture Notes In Computer Science*, Novosibirsk, Russland, 2003. Springer-Verlag.

[UML04] *Unified Modeling Language Specification, Version 2.0.* Object Management Group (OMG), *http://www.uml.org*, 2004.

[VS05] Matthias Voesgen und Dehla Sokenou. Aspektorientierte Programmiertechniken im Unit-Testen. *eingereicht*, 2005.