Master Thesis

Integration von Verhaltensmodellen in ein MDSD-Framework am Beispiel von UML Aktivitäten

Gerrit Beine

geboren am 01.08.1980 in Plauen

Studiengang International Software Product Engineering

Westsächsische Hochschule Zwickau
Fachbereich Physikalische Technik / Informatik
Fachgruppe Informatik

Betreuer: Prof. Dr. Georg Beier

Abgabetermin: 25.01.2010

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1 Einführung

In den letzten Jahren wird eine seit langem schon bekannte Vorgehensweise zur Effizienzsteigerung in der Softwareentwicklung langsam salonfähig: modellgetriebene Softwareentwicklung. Auch wenn schon vor vielen Jahren Werkzeuge existieren, die diese Vorgehensweise ermöglichen, gibt es nach wie vor (im Großen und Ganzen unbegründete) Skepsis in der Wirtschaft, was die Möglichkeiten und Potentiale angeht.

Auch heute werden vielfach Modelle nur zur Entwurfszeit oder zu Dokumentationszwecken genutzt und führen ein Dasein am Rand der eigentlichen Entwicklungsarbeit. Gleichzeitig existiert mittlerweile ein riesiger Markt für offene wie auch proprietäre Werkzeuge und Methoden, die sich mit dem Thema der (teil)automatisierten Erzeugung von Software aus formalen Modellen beschäftigen.

Allerdings unterstützen viele der verfügbaren Generatoren lediglich die Erzeugung von Infrastrukturcode wie z.B. Klassen oder Datenbankschemata aus statischen Modellen.

Wesentlicher Bestandteil von Softwaresystemen sind aber die dynamischen Modelle, auch als Verhaltensmodelle bezeichnet. Diese sind im Allgemeinen weitaus schwieriger zu verstehen und komplizierter zu transformieren als die statischen Modelle. Das mag zum Einen daran liegen, dass bei der Transformation statischer Modelle oftmals nur identische Strukturen in eine andere Repräsentation überführt werden, zum Anderen aber auch an den teilweise sehr komplizierten mathematischen Modellen, auf denen die Verhaltensmodelle basieren.

Die UML kennt drei verschiedene Möglichkeiten, Verhaltensmodelle zu erstellen: Aktivitäten, Interaktionen und Zustandsautomaten. Das Augenmerk dieser Arbeit liegt auf den Aktivitäten. Diese sind erst relativ spät Bestandteil der UML geworden und seit ihrer Integration einem deutlich stärkerem Wandel unterworfen als das für andere Bestandteile der UML der Fall ist. Aktivitäten besitzen ein großes Potential für die Verhaltensmodellierung, denn sie können sowohl sehr abstrakt verwendet werden, um z.B. Geschäftsprozesse zu modellieren, als auch sehr konkret auf dem Niveau von Programmiersprachen. Die graphische Darstellung von Aktivitäten ist dabei sehr

einfach gehalten, so dass sie hervorragend geeignet sind, um zur Entwurfszeit einer Software oder während der Abstimmung mit Personen, die keine detaillierten Kenntnisse über Softwaremodellierung haben, eingesetzt zu werden.

1.1 Motivation und Ziel der Arbeit

Trotz dieser offensichtlichen Vorteile von Aktivitäten, wurden sie im Kontext der modellgetriebenen Softwareentwicklung bisher eher vernachlässigt. Die vorliegende Arbeit soll zeigen, dass Aktivitäten aber gerade in diesem Bereich immense Vorteile gegenüber den komplizierten Zustandsautomaten und den weitaus weniger formalen Interaktionen haben.

Insbesondere in Hinblick auf den möglichen Einsatz von Aktivitäten für Softwaretests wird im Rahmen dieser Arbeit auf das seit 2006 an der Westsächsischen Hochschule Zwickau entwickelte GeneSEZ Generator Framework Bezug genommen. Das GeneSEZ Generator Framework unterstützt, wie viele der verfügbaren Generatoren, lediglich die Erzeugung von Infrastrukturcode aus statischen Modellen und muss für den hier verfolgten Zweck erweitert werden. Dabei soll zunächst das GeneSEZ Metamodell eine möglichst minimale Erweiterung erfahren, um Aktivitäten wie sie in der UML definiert sind, sinnvoll transformieren zu können. In einem zweiten Schritt dienen die Aktivitäten als Grundlage für ein weiteres Metamodell, das zur Generierung von Softwaretests in das GeneSEZ Framework integriert werden soll.

1.2 Aufteilung der Arbeit

Der Hauptteil der Arbeit ist in drei Kapitel gegliedert. Zunächst wird ein detaillierter Überblick zum Thema Aktivitäten erfolgen. Dabei wird vor allem auf die Bedeutung und die Entwicklung von Aktivitäten im Rahmen der UML eingegangen und die in der aktuellen Version der UML enthaltenen Elemente erläutert. Das Kapitel endet mit einer Betrachtung der wesentlichen Änderungen am Aktivitätenmodell der UML in den Versionen 1.5 und 2.0 und den Zusammenhängen von Aktivitäten und Zustandsautomaten, die in der aktuellen Version der UML noch enthalten sind.

Im nächsten Kapitel wird die Integration von Aktivitäten in das GeneSEZ Metamodell erklärt, die im Rahmen dieser Arbeit stattgefunden hat. Das Kapitel beginnt mit einer genauen Zielstellung, was

die Aktivitäten im Rahmen des GeneSEZ Metamodells leisten sollen und erklärt dann zunächst welche grundlegenden Änderungen für die Integration von Verhaltensmodellen im GeneSEZ Metamodell überhaupt notwendig waren. Im Anschluss werden die einzelnen Elemente der Aktivitäten detailliert vorgestellt sowie die Änderungen an der Modell-zu-Modell-Transformation von UML2 auf das GeneSEZ Metamodell gezeigt.

Das letzte Kapitel des Hauptteils erläutert zunächst die Möglichkeiten modellgetriebener Softwaretests, wobei der Schwerpunkt auf UML-basierten Softwaretests liegt. Nach einem kurzen Überblick zum Thema Softwaretests wird das UML Testing Profile analysiert. Anschließend wird eine prototypische Realisierung mit dem GeneSEZ Generator Framework gezeigt, die aus UML Aktivitäten Testscripten für die Software QF-Test erzeugt.

Im Ausblick werden noch einige Möglichkeiten gezeigt, die sowohl den Umgang mit Aktivitäten im Rahmen von GeneSEZ perspektivisch erleichtern und verbessern können als auch Erweiterungsmöglichkeiten in Hinsicht auf Softwaretests und entsprechende Generierungsmöglichkeiten für weitere Werkzeuge.

1.3 Abgrenzung

Die Arbeit soll sich nicht mit Verhaltensmodellen im Allgemeinen beschäftigen und auch keinen Vergleich zwischen den Möglichkeiten unterschiedlicher Verhaltensmodelle liefern. Es besteht auch nicht der Anspruch, eine den Aktivitäten der UML gleichwertige Lösung zu schaffen, vielmehr soll eine pragmatische Realisierung erfolgen. Insbesondere soll das Metamodell lediglich so formal sein, dass es eine effiziente Übersetzung in Quellcode oder Softwaretests erlaubt. Keinesfalls soll es erlauben, Aktivitäten in virtuellen Maschinen auszuführen, wie das Aktivitätemodell der UML 2 erlaubt. Einige Aspekte von Aktivitäten, die in der UML enthalten sind, laufen diesem Ziel eines minimalen Metamodells für Aktivitäten entgegen. Diese werden an geeigneten Stellen im Text erwähnt und so weit erläutert, wie es der Arbeit dient.

Ebenfalls erwähnt werden Erkenntnisse zu den verwendeten Modellierungswerkzeugen, wobei im Rahmen dieser Arbeit keine Evaluierung von Werkzeugen erfolgt.

Auf eine detaillierte Beschreibung des GeneSEZ Generator Frameworks wird im Rahmen dieser Arbeit ebenfalls verzichtet, da es dazu bereits ausreichend frei verfügbare Literatur gibt.

Da Aktivitäten in breitem Maße für die modellgetriebene Softwareentwicklung verwendet werden

können, gibt es auch viele mögliche Beispiele. Diese Arbeit beschränkt sich auf das Beispiel der Generierung einfacher Softwaretests für die Anwendung QF-Test. Im Rahmen dessen wird auf eine ausführliche Erklärung der Rolle und Bedeutung von Softwaretests verzichtet.

1.4 Hinweis zu Literatur, Zitaten und Quellenverwendung

Direkte Zitate sind als solche kenntlich gemacht. Basieren längere Abschnitte auf einigen wenigen Werken, wie es insbesondere bei Kapitel 2 der Fall ist, sind diese Werke aus Gründen der Lesbarkeit nur einmal am Beginn des Kapitels explizit erwähnt.

Für Abbildungen und Tabellen, die anderen Werken entstammen, sind jeweils die Quellen als Verweise auf das Literaturverzeichnis angegeben.

1.5 Textkennzeichnungen

Beschriftungen von Abbildungen und Tabellen erfolgen untenstehend.

Quellcode, Annotationen und Stereotypen werden im Text `dicktengleich` hervorgehoben. Namen von Modellelementen sind im Text nicht explizit kenntlich gemacht, um den Lesefluss nicht zu behindern.

1.6 Begrifflichkeiten

Da in der deutschsprachigen Literatur im Zusammenhang mit UML die englischen Begriffe Control Flow und Control Node in der Regel mit Kontrollfluss und Kontrollknoten übersetzt sind, werden diese beiden Begriffe auch im Rahmen dieser Arbeit verwendet, obwohl der Begriff des Steuerns der Bedeutung von Control in diesem Zusammenhang deutlich näher käme.

Innerhalb der Aktivitäten wird zwischen zwei Arten von Flüssen unterschieden: Kontrollflüssen, die Kontrolltoken transportieren und Objekt- bzw. Datenflüssen, die Datentoken transportieren. Die beiden Begriffe Objektfluss und Datenfluss sowie Objekttoken und Datentoken werden in dieser Arbeit gleichberechtigt verwendet.

Ebenso werden in Bezug auf die Ausführung von Aktivitäten die Begriffe Laufzeitumgebung und virtuelle Maschine gleichberechtigt verwendet.

1.6.1 Verwendung englischsprachiger Begriffe

Englischsprachige Begriffe werden nach Möglichkeit übersetzt. Wenn die deutschsprachige Literatur Begriffe uneinheitlich oder gar nicht übersetzt, werden die englischen Begriffe verwendet.

1.7 Verwendete Werkzeuge

Als Werkzeuge, insbesondere zur Visualisierung der Beispiele, kommen ausschließlich MagicDraw 16 und Enterprise Architect 7.5 zum Einsatz, wobei der Magic Draw bevorzugt wurde.

Beide Werkzeuge bieten eine gute Unterstützung bei der Modellierung von Aktivitäten, jedoch implementiert der Enterprise Architect das Metamodell der UML nur sehr rudimentär.

Insgesamt ist die Unterstützung von Aktivitäten in diversen Modellierungswerkzeugen sehr uneinheitlich und teilweise auch schlecht. Eine Werkzeuganalyse ist jedoch nicht Bestandteil dieser Arbeit.

Als Zielplattform für die Generierung der Softwaretests kam QF-Test 3 zum Einsatz, da sich dieses Testwerkzeug durch ein einfaches XML-Format für Testscripten sehr gut eignet.

2 Aktivitäten in der Unified Modelling Language

Das folgende Kapitel soll eine Einführung in die Bedeutung und Anwendung von Aktivitäten in der Softwaremodellierung liefern. Dabei wird insbesondere auf die Aktivitäten in der Unified Modeling Language eingegangen. Sonstige Modellierungssprachen, in denen Aktivitäten oder verwandte Konzepte Verwendung finden, sind nicht Bestandteil dieser Arbeit.

Als Grundlage für das Kapitel dienen vor allem die Unified Modeling Language Specification in der Version 1.5 vom 01.03.2003 [OMG03] sowie die Unified Modeling Language Superstructure in der Version 2.2 vom 02.02.2009 [OMG09]. Weiterhin sind einige Absätze gestraffte Wiedergaben der Artikelserie von Conrad Bock [BOC04], [BOC03B], [BOC03C], [BOC03D], [BOC04F] über das neue Aktivitätenmodell der UML 2, die die einzige brauchbare Literatur zum Thema Aktivitäten darstellt.

Sofern nicht explizit andere Quellen benannt werden, bezieht sich das Kapitel auf die oben genannten Werke.

2.1 Aktivitäten als Verhaltensmodelle in der UML

Aktivitäten werden in der UML neben Zustandsautomaten und Interaktionen für die Beschreibung des Verhaltens von Softwaresystemen genutzt. Die Aktivitäten bieten dabei den Vorteil, dass sie bereits auf einem sehr abstrakten Niveau verwendet werden können, z.B. zur Modellierung des Ablaufs von Anwendungsfällen. Gleichzeitig ist es mit Aktivitäten aber auch möglich, Softwareverhalten auf Implementierungsniveau zu beschreiben, da viele der von objektorientierten Programmiersprachen unterstützten Konzepte direkt mit Aktivitäten modelliert werden können.

Die aktuell verwendeten Aktivitäten haben im Laufe der Entwicklung der UML einen drastischen Änderungsprozess durchlaufen. Einige Details zum Verlauf dieser Änderungen sind in 2.2 beschrieben.

Die mit Aktivitäten beschriebenen Verhaltensmodelle entsprechen klassischen Ablaufmodellen.

Aktivitäten in der UML koordinieren Aktionen sowie Kontroll- und Datenflüsse zwischen diesen. Konzeptionell basieren sie auf Petri-Netzen, auf die beim Entwurf der Aktivitäten in der UML 2 explizit verwiesen wurde (siehe u.a. [BOC04], [STO05] Seiten 194ff.). Die Aktivitäten in der UML 2 gehen in ihrer Komplexität allerdings über die klassischen Petri-Netze hinaus. Das wird schon durch die Menge unterschiedlicher Knoten deutlich, die zum Teil historischen Ursprungs sind, zum Teil auch eingeführt werden mussten, um die Ausführung von Aktivitäten in einer virtuellen Maschine zu ermöglichen. Als entfernte Vorläufer der momentanen Aktivitäten kann man Nassi-Shneidermann-Diagramme und Jackson-Diagramme betrachten, wobei der Abstraktionsgrad von Aktivitäten ein deutlich höheres Niveau erreichen kann.

Verwandte Sprachen bzw. Varianten der Aktivitäten sind u.a. in den erweiterten Ereignis-Prozess-Ketten (eEPK) oder in der SysML[1] zu finden. Beides sind Modellierungssprachen, die domänenspezifisch ausgerichtet sind, sollen aber im Rahmen dieser Arbeit nicht weiter behandelt werden.

In Aktivitäten gilt, wie in anderen Ablaufmodellen ebenfalls, die Regel, dass ein Schritt genau dann ausgeführt wird, wenn der vorangegangene Schritt beendet wurde und die notwendigen Eingabedaten vorliegen. In [BOC04] wird erklärt, dass die Aktivitäten der UML exakte Aussagen über ihr Laufzeitverhalten erlauben, weil davon ausgegangen wird, dass sie in einer virtuellen Maschine ablaufen. Diese virtuelle Maschine basiert auf der den Petri-Netzen zugrunde liegenden zentralen Idee: Token bewegen sich zwischen Knoten auf gerichteten Kanten entlang. Die Aktivitäten der UML unterscheiden dabei zwei Arten von Token: Kontroll- und Objekttoken. Im Unterschied zu den Token in Petri-Netzen sind diese Token aber lediglich virtuell, sie existieren weder während der Modellierung der Aktivitäten noch während der Ausführung in der virtuellen Maschine, da das Aktivitätenmodell der UML keinerlei Aussagen über die Implementierung und damit die Ausführung der Aktivitäten macht.

1 Eine gute Einführung in die SysML stellt [WEI06] dar.

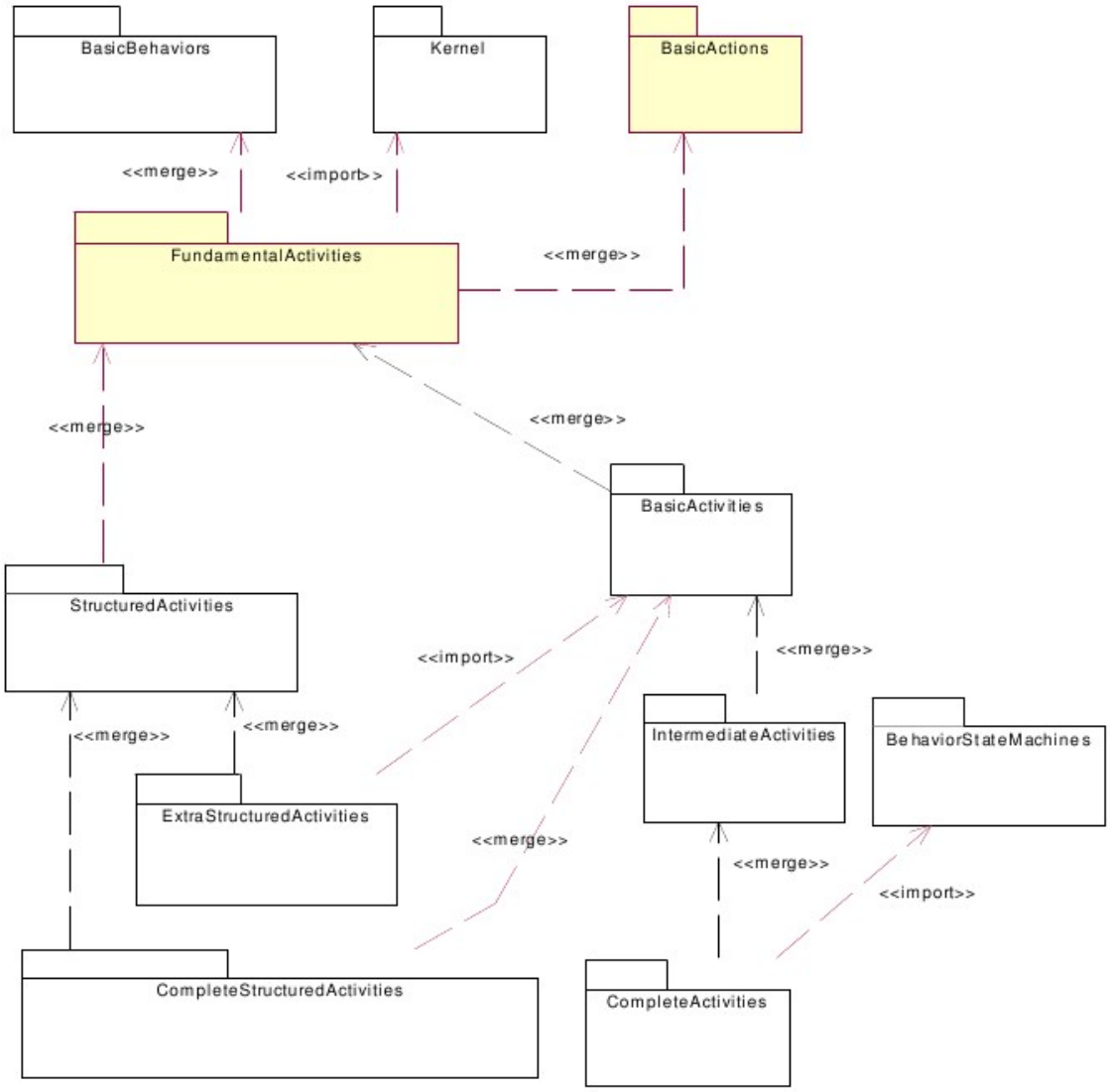

Abbildung 2.1: Übersicht der auf Aktivitäten bezogenen Pakete im UML Metamodell

Quelle: [OMG09]

Wie bereits erwähnt, können Aktivitäten auf sehr unterschiedlichen Abstraktionsniveaus verwendet werden, z.B. um die Methode einer Klasse zu beschreiben, als Verhalten für die Entry- und Exit-Ereignisse bei Zuständen oder rekursiv eingebettet in anderen Aktivitäten, wobei sie hier über eine CallBehaviorAction referenziert werden müssen (siehe 2.1.2). Um diese unterschiedlichen Anwendungsgebiete sinnvoll abbilden zu können, sind Aktivitäten im Metamodell der UML, anders als Zustandsautomaten und Interaktionen, in mehrere Pakete aufgeteilt, die jeweils einen bestimmten Bereich abdecken (siehe Abbildung 2.1).

Insgesamt existieren sieben Pakete für Aktivitäten und vier Pakete für Aktionen im Metamodell.

Eine kurze Übersicht der Aufteilung der Elemente von Aktivitäten auf die einzelnen Pakete ist in Tabelle 2.1 zu finden. Die Pakete des Metamodells der Aktivitäten können, wie in Abbildung 2.1 zu sehen, kombiniert werden.

Paket	Elemente
FundamentalActivities	Aktivitäten und abstrakte Klassen für Knoten, Aktionen
BasicActivities	Objektknoten, Pins, Initial Node und Activity Final Node sowie Kontroll- und Objektfluss (2.1.3 und 2.1.4)
IntermediateActivities	CentralBuffer (2.1.3), sämtliche Kontrollknoten (2.1.4), Partitionen
CompleteActivities	Lokale Bedingungen für Aktionen, Erweiterungen für Objektflüsse und -knoten, Data Stores (2.1.3), Parametergruppen und Effekte
StructuredActivities	Strukturierte Aktivitäten: Sequenzen, Schleifen, Bedingungen, Abschnitte
CompleteStructured Activities	ConditionalNode (2.1.5), LoopNode (2.1.5)
ExtraStructured Activities	ExceptionHandler (2.1.5), ExpansionRegion (Fehler: Referenz nicht gefunden), ExpansionNode

Tabelle 2.1 Inhalte der Pakete zu Aktivitäten im UML Metamodell

Quelle: Eigene Darstellung

Aktivitäten in der UML können problemlos Abläufe über Systemgrenzen hinweg beschreiben. Damit wird es möglich, Abläufe zu modellieren, die sowohl automatische wie auch manuelle Schritte enthalten und die Ablaufmodellierung unabhängig von der Realisierung in Hard- oder Software zu machen.

Wie bei Zustandsautomaten und Interaktionen handelt es sich auch bei den Aktivitäten um anwenderdefiniertes Verhalten. Die UML erlaubt es, jedes anwenderdefinierte Verhalten zu parametrisieren um Daten entgegenzunehmen bzw. zurückzugeben. Die Funktionsweise dieser Aufrufe wird in 2.1.2 und 2.1.3 erklärt. In der Laufzeitumgebung der UML ist festgelegt, dass jedes Verhalten eine Klasse ist, von der zum Zeitpunkt der Ausführung eine Instanz erzeugt und nach Beendigung wieder gelöscht wird (siehe [BOC04]).

Im Folgenden wird zunächst die graphische Notation von Aktivitäten vorgestellt. Danach werden die einzelnen Elemente erläutert. Partitionen (Swimlanes) werden im Rahmen dieser Arbeit nicht betrachtet. Sie stehen konzeptionell dem Ziel der Arbeit, ein minimales Metamodell für Aktivitäten zur modellgetriebenen Softwareentwicklung zu entwickeln, entgegen. Außerdem sind sie für ein vollständiges und formales Metamodell für Aktivitäten nicht notwendig. Eine Einführung in die Konzepte von Partitionen ist in [BOC04E] zu finden.

2.1.1 Graphische Darstellung von Aktivitäten

Die graphische Darstellung ist ohne tiefere Kenntnisse der UML eingängig, da sie stark an Ablauf- und Flussdiagramme erinnert, die auch in anderen Wirtschaftszweigen eingesetzt werden.

Im Wesentlichen bestehen Aktivitäten aus drei Arten von Knoten und zwei Arten von Kanten. Die Knoten sind Kontrollknoten, Objektknoten und Aktionen, bei den Kanten handelt es sich um Kontroll- und Objektfluss.

Abbildung 2.2: Aktionsknoten
(Action, SendAction, AcceptAction)
Quelle: Eigene Darstellung

Aktionen werden als Vierecke mit abgerundeten Ecken dargestellt, die eine Bezeichnung enthalten. Damit ist die Notation von Aktionen identisch mit der, die für Zustände in Zustandsautomaten verwendet wird. Das bedeutet für viele Benutzer zunächst eine Irritation, denn eine semantische Identität geht damit nicht einher. Die Semantik der dargestellten Aktion ist aus der Notation nicht in jedem Fall ersichtlich. In vielen Modellierungswerkzeugen werden die Spezialisierungen der Aktionen aber sichtbar gemacht, in MagicDraw z.B. auf die gleiche Weise wie Stereotypen. Lediglich Timer sowie Send- und Accept-Aktionen werden nicht durch Vierecke dargestellt (siehe Abbildung 2.2). Timer werden durch eine stilisierte Sanduhr visualisiert.

Abbildung 2.3: Kontrollknoten
(Decision/Merge, Fork/Join, Initial, ActivityFinal, FlowFinal)
Quelle: Eigene Darstellung

Die UML unterscheidet sieben Arten von Kontrollknoten. Für diese gibt es insgesamt fünf verschiedene Darstellungen (siehe Abbildung 2.3). Decision und Merge werden jeweils durch eine Raute, Fork und Join durch einen schwarzen Strich dargestellt. Für InitialNodes werden voll ausgefüllte schwarze Punkte verwendet, ActivityFinalNodes werden durch einen Kreis mit einem schwarzen Punkt dargestellt und FlowFinalNodes durch einen Kreis mit einem Kreuz. An der

Auswahl der Symbole für die Kontrollknoten wird wieder deutlich, dass die Aktivitäten der UML lange Zeit als Variante der Zustandsautomaten betrachtet wurden.

Zur Darstellung von Objektknoten können zwei unterschiedliche Varianten benutzt werden. Sie können entweder durch frei stehende Rechtecke, die eine Bezeichnung enthalten oder durch kleine Rechtecke, sogenannte Pins, die an den Seiten von Aktionen kleben, dargestellt werden (siehe Abbildung 2.4 unten). Die Bedeutung ist in beiden Fällen identisch. Sonderfälle der Objektknoten stellen Data Stores und Central Buffer dar, die graphisch durch einen Objektknoten mit den Stereotypen `<<datastore>>` bzw. `<<centralbuffer>>` repräsentiert werden.

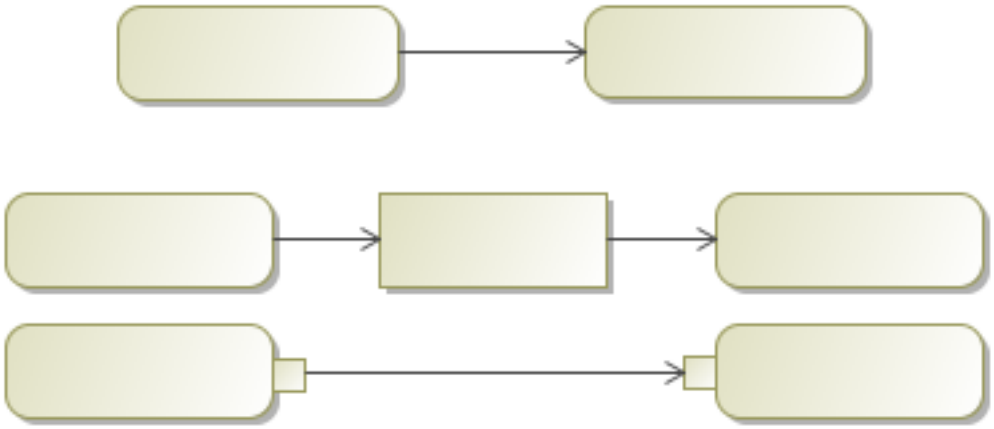

Abbildung 2.4: Darstellung von Kontrollfluss und Objektfluss

Quelle: Eigene Darstellung

Kontrollflusskanten verbinden verschiedene Aktionen miteinander, während Objektflusskanten Objektknoten mit Aktionen verbinden. Kontrollknoten können durch beiden Arten von Kanten verbunden werden.

Da beide Kanten als Pfeile dargestellt werden, ergibt sich die Art des Flusses im Allgemeinen aus der Verwendung der Kante. Die Existenz von Objektknoten bzw. Pins ist damit die einzige Möglichkeit in der graphischen Darstellung zwischen Kontroll- und Objektflüssen zu unterscheiden (siehe Abbildung 2.4).

Aktivitäten selbst müssen in der UML nicht zwangsläufig graphisch repräsentiert werden. Ein Aktivitätsdiagramm bezieht sich implizit auf eine Aktivität. Unter Umständen sind weitere Aktivitäten in eine Aktivität eingebettet oder die Aktivität benötigt Parameter. Dann wird die Aktivität ebenso wie Aktionen als Rechteck mit abgerundeten Ecken dargestellt. Ein Aktvititätsknoten besitzt jedoch keine Pins, sondern sogenannte ActivityParameterNodes, die eine

besondere Form der Objektknoten darstellen und auf dem Rand der Aktivität gezeichnet werden.

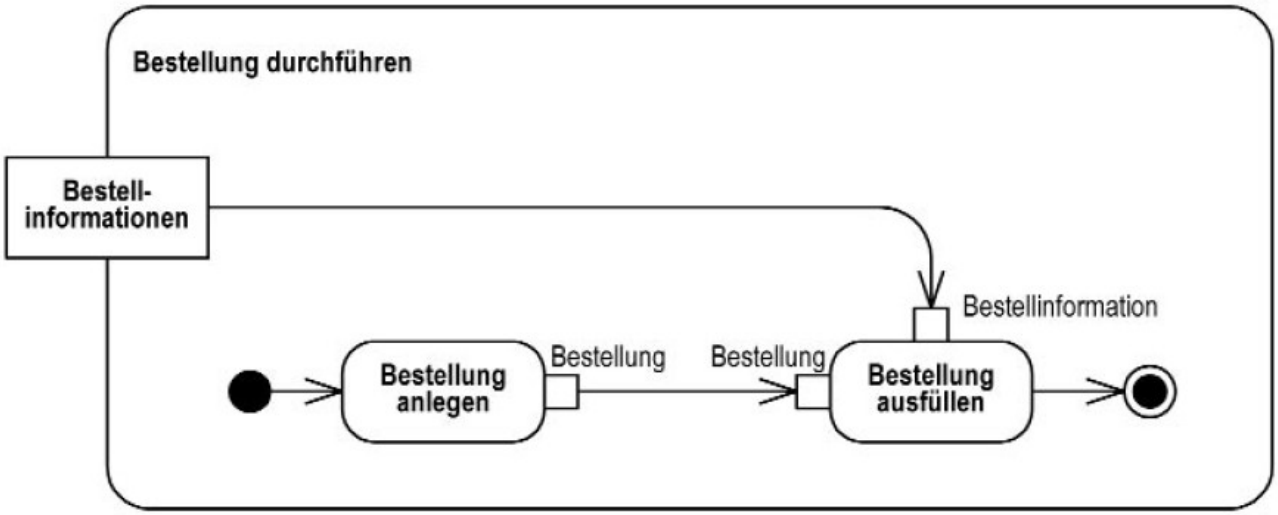

Abbildung 2.5: Explizite Darstellung einer Aktivität mit Parameter

Quelle: [BOC04]

Leider sind die Hersteller von Modellierungswerkzeugen nicht konsequent in der Umsetzung der UML Spezifikation, sodass die Darstellung von Aktivitätsdiagrammen sehr unterschiedlich sein kann. Eine Darstellung wie in Abbildung 2.5 ist in MagicDraw z.B. nicht möglich.

2.1.2 Aktionen

Aktionen sind das Basiselement des Aktivitätenmodells der UML. Jede Verhaltensbeschreibung in der UML muss in Aktionen zerlegbar sein, um auf Objekte wirken zu können. Der Grund dafür ist, dass Aktionen die einzigen Elemente der UML sind, die Objekte lesen, schreiben und Operationen auf ihnen ausführen können. Außerdem ist es Aktionen möglich, andere Verhalten direkt aufrufen.

Dabei handelt es sich bei Aktionen nicht um Verhalten an sich, da die Aktionen Elemente der UML sind, Verhalten aber vom Anwender beschrieben wird. Direkt enthalten sind Aktionen nur in Aktivitäten. Aus anderen Verhaltensmodellen können sie nur indirekt über Aktivitäten referenziert werden. Die in einer Aktivität enthaltenen Aktionen, sind in deren Kontext atomar. Das bedeutet, sie müssen immer vollständig ausgeführt und können nicht unterbrochen werden. Unterbrechungen sind allenfalls zwischen den Aktionen möglich, wobei die Dauer vom Ende einer Aktion bis zum Start der nächsten mit 0 angenommen wird. Der Grund dafür wird weiter unten in diesem Abschnitt erläutert.

Damit eine Aktion ausgeführt werden kann, müssen zwei Bedingungen erfüllt sein: Der Zeitpunkt des Starts muss bekannt sein und die notwendigen Eingabedaten müssen vorliegen. Diese beiden Bedingungen werden durch Kontroll- und Datenflusstoken beschrieben.

Eine Aktion startet mit ihrer Ausführung, wenn alle eingehenden Kontroll- und Datenflusstoken verfügbar sind. Wenn eine Aktion mehrere eingehende Kontrollflüsse hat, muss die Aktion von jedem der eingehenden Kontrollflüsse ein Token erhalten, um zu starten. Gleiches gilt für Objektflüsse und für Aktionen, die sowohl eingehende Kontroll- wie auch Datenflüsse haben. Allerdings gibt es bei Objektflüssen eine Ausnahme, nämlich Datenströme, die weiter unten erklärt werden.

Verfügt eine Aktion über keinerlei eingehende Kontrollflüsse, sondern nur über eingehende Datenflüsse, wird die Aktion genau dann mit der Ausführung beginnen, wenn alle benötigten Daten vorliegen. Es vereinfacht Diagramme massiv, wenn Kontrollflüsse durch den Graphen mit den Datenflüssen identisch sind, da damit alle Datenflüsse implizit zu Kontrollflüssen werden. Letzteres ist in [RJB05] auf Seite 276 explizit beschrieben und für die praktische Umsetzung im Rahmen dieser Arbeit von Bedeutung, denn damit tragen Datentoken aus Sicht der UML implizit Kontrolltoken mit sich.

Sind weder eingehende Kontroll- noch eingehende Datenflüsse vorhanden, kann die Aktion nicht mit ihrer Ausführung starten, da sie kein Token erhält.

Wie bereits in 2.1.1 erwähnt, verbinden Kontrollflüsse direkt Aktionen oder Kontrollknoten miteinander, während Objektflüsse Objektknoten einbeziehen. Aktionen können über eine unbegrenzte Anzahl von eingehenden Kontroll- sowie Objektflüssen verfügen. Der Datentyp, der auf einem Objektfluss transportiert wird, hat auf die Ausführung der Aktion keinen Einfluss, d.h. sollten mehrere Objektflüsse mit dem gleichen Datentyp vorliegen, wird trotzdem für jeden dieser Objektflüsse ein separates Token erwartet.

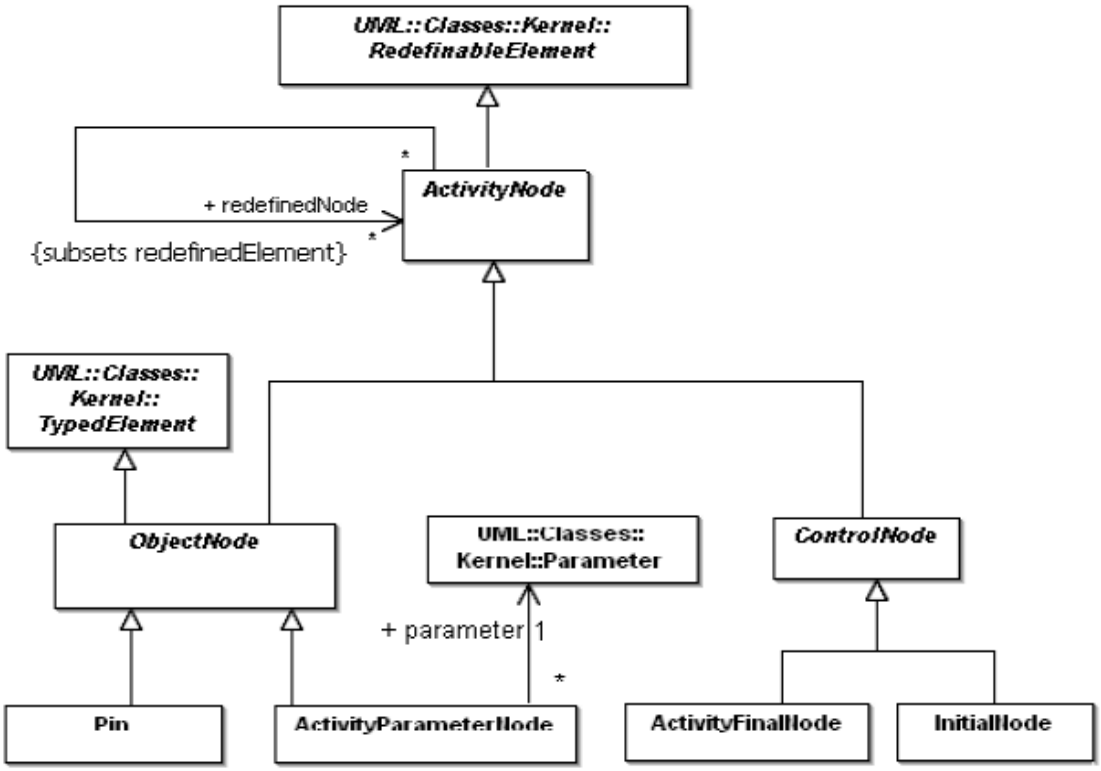

Abbildung 2.6: UML 2 Metamodell: Knoten in Aktivitäten

Quelle: [OMG09], Seite 299

Eine Unterscheidung zwischen Objektknoten und Pins gibt es nur auf graphischer Ebene. Sobald eine Aktion einen eingehenden bzw. ausgehenden Objektfluss hat, verfügt sie über einen Pin des entsprechenden Typs. Im Metamodell der UML sind Pins Spezialisierungen der Objektknoten, die als abstrakte Klasse definiert sind (siehe Abbildung 2.6).

Die Pins funktionieren für die Aktionen als eine Art Puffer und halten Eingangsdaten so lange vor, bis die Token aller eingehenden Flüsse vorliegen. Da Kontrolltoken keine Daten sind oder enthalten und somit keinen Typ haben, wird für diese auch kein Puffer benötigt, wenn eine Aktion mit ihrer Ausführung auf weitere eingehende Token warten muss. Implizit werden aber auch Kontrolltoken so lange gepuffert, bis alle Token vorliegen. Um die Ausführung von Aktivitäten in der virtuellen Maschine der UML zu ermöglichen, wurde bei der Realisierung keine Möglichkeit für den Anwender geschaffen, die beschriebenen Startbedingungen von Aktionen zu verändern. Wenn ein Anwender Einfluss auf den Beginn der Ausführung einer Aktion nehmen kann, muss das explizit modelliert werden.

Die von einer Aktion zu verarbeitenden Objekte und Daten existieren nur zur Laufzeit. Alle anderen

Informationen (wie z.B. die Attribute eines Objekts), die durch die Aktion verändert werden, sind explizit im Modell beschrieben.

Wenn die Ausführung einer Aktion beendet ist, werden sofort alle Kontroll- und Datentoken an den entsprechenden ausgehenden Kontroll- und Datenflüssen bereitgestellt. Es ist möglich, Aktionen ohne ausgehende Kontroll- und Datenflüsse zu modellieren, allerdings können diese Aktionen dann keine anderen Aktionen starten, da es keine Möglichkeit gibt, die entsprechenden Token weiterzuleiten.

Mechanismen zu Rekursion, Wiederverwendung und Polymorphismus

Wie weiter oben erwähnt, sind spezialisierte Aktionen in der UML vordefiniert und können vom Anwender nicht geändert werden. Aktionen sind die einzigen Elemente der UML, die Objekte direkt verändern können. Um dem Anwender die Möglichkeit zu geben, mit Aktionen präzise auszudrücken, was innerhalb einer Aktivität geschehen soll, werden in UML-Modellen immer spezifische Aktionen modelliert. Eine Übersicht dieser Aktionen mit einer kurzen Erklärung ist in Tabelle 2.2 zu sehen.

Von besonderer Bedeutung sind dabei die CallBehaviorAction und die CallOperationAction. Beide erlauben es, direkt bzw. indirekt andere Verhalten aufzurufen. Durch diese beiden Aktionen wird die einführend erwähnte Rekursion von Aktivitäten und damit eine vollständige Beschreibung von Softwaresystemen durch Aktivitäten ermöglicht.

Der Unterschied zwischen CallBehaviorAction und CallOperationAction ist, dass die CallBehaviorAction direkt anderes Verhalten adressiert, während die CallOperationAction eine Methode einer Klasse aufruft. Bei der Modellierung von CallOperationActions muss sichergestellt werden, dass ein Objekt der entsprechenden Klasse vorliegt, wenn die Aktion startet. Dies kann

Aktion	Bemerkung	Quelle
AcceptCallAction	Diese Aktion stellt einen Empfänger für synchrone Aufrufe dar. Sie gibt ein Datentoken an das aufrufende Element zurück.	[RJB05], [OMG09]
AcceptEventAction	Wartet auf das Auftreten eines bestimmten Ereignisses. Sie kann als graphische AcceptAction (siehe Abbildung 2.2, rechts) oder als TimeAction (stilisierte Sanduhr) dargestellt werden.	[RJB05], [OMG09]
AddStructuralFeature ValueAction	Fügt Werte zu einem Objekt hinzu, z.B. ein Element in eine Collection.	[OMG09]
AddVariable ValueAction	Setzt eine Variable auf einen bestimmten Wert, z.B. eine Instanzvariable eines Objektes.	[OMG09]
ApplyFunctionAction	Berechnet mathematische Funktionen; nicht in der UML Superstructure enthalten	[RJB05]
BroadcastEvent Action	Alternativer Name für BroadcastSignalAction; teilweise in der Literatur verwendet	[RJB05]
BroadcastSignal Action	Sendet ein Signal an alle Elemente innerhalb der Laufzeitumgebung, die das Signal empfangen können. Damit können beispielsweise Transitionen in Zustandsautomaten getriggert werden.	[OMG09]
CallBehaviorAction	Konkrete Realisierung der CallAction, die direkt anderes Verhalten aufrufen kann. Im Implementierungskontext z.B. der Aufruf einer statischen Methode.	[RJB05], [OMG09]
CallOperationAction	Konkrete Realisierung der CallAction, die indirekt anderes Verhalten durch Methoden aufrufen kann. Im Implementierungskontext z.B. der Aufruf der Methode einer konkreten Instanz.	[RJB05], [OMG09]
ClearAssociation Action	Entfernt alle Assoziationen, die ein bestimmtes Objekt besitzt.	[OMG09]
ClearStructural FeatureAction	Entfernt alle Elemente aus einem Objekt, bspw. alle Elemente aus einer Collection.	[OMG09]
ClearVariableAction	Löscht alle Werte einer Variablen.	[OMG09]
CreateAction	Alternativer Name für CreateObjectAction; teilweise in der Literatur verwendet	[RJB05]
CreateLinkAction	Erzeugt eine Assoziation zwischen Objekten.	[OMG09]
CreateLink ObjectAction	Erzeugt ein Objekt, das eine Assoziation repräsentiert.	[OMG09]
CreateObjectAction	Erzeugt ein Objekt und gibt ein entsprechendes Datentoken auf einen ausgehenden Datenfluss zur Weiterleitung.	[OMG09]
DestroyAction	Allgemeine Aktion zum Löschen; nicht in der UML Superstructure enthalten	[RJB05]
DestroyLinkAction	Löscht Assoziationen zwischen Objekten und entsprechende Assoziationsobjekte.	[OMG09]

Aktion	Bemerkung	Quelle
DestroyObjectAction	Löscht Objekte.	[OMG09]
LinkEndCreationData	Keine echte Aktion im Sinne der UML, wird benötigt, um Assoziationsenden für die CreateLinkAction zu finden.	[OMG09]
LinkEndData	Keine echte Aktion im Sinne der UML, wird benötigt, um Assoziationsenden für die CreateLinkAction zu finden.	[OMG09]
LinkEnd DestructionData	Keine echte Aktion im Sinne der UML, wird benötigt, um Assoziationsenden für die DestroyLinkAction zu finden.	[OMG09]
OpaqueAction	Platzhalter für noch nicht definierte Aktionen oder Container für plattformspezifischen Code. Auf diesem Weg können z.B. bestimmte Algorithmen direkt in der Zielsprache implementiert und im Modell hinterlegt werden.	[OMG09]
RaiseException Action	Erzeugt eine Exception aus dem Objekt, das über den eingehenden Datenfluss entgegenommen wird.	[RJB05], [OMG09]
ReadExtentAction	Stellt die Menge aller Instanzen eines bestimmten Typs zur Verfügung.	[RJB05], [OMG09]
ReadIsClassified ObjectAction	Prüft, ob ein Objekt Instanz eines bestimmten Typs ist.	[OMG09]
ReadLinkAction	Navigiert über Assoziationen, um Objekte eines der Assoziationsenden zu lesen.	[OMG09]
ReadLinkObject EndAction	Navigiert über Assoziationsobjekte, um Objekte eines der Assoziationsenden zu lesen.	[OMG09]
ReadLinkObject EndQualifierAction	Liest einen Wert von einem Objekt an einem Assoziationsende.	[OMG09]
ReadSelfAction	Liefert ein Token für den Kontext zu dem die Aktivität gehört.	[OMG09]
ReadStructural FeatureAction	Liest Elemente aus einem Objekt, z.B. einer Collection.	[OMG09]
ReadVariableAction	Liest Variablen eines Objektes.	[OMG09]
ReclassifyObject Action	Ändert den Typ eines Objektes; weitreichender als ein Cast im Implementierungskontext, da mehrere Typen gleichzeitig geändert werden können.	[OMG09]
ReduceAction	Kombiniert alle Elemente einer Collection zu einem einzigen Objekt.	[OMG09]
RemoveStructural FeatureValueAction	Entfernt Elemente z.B. aus einer Collection.	[OMG09]
RemoveVariable ValueAction	Entfernt Werte von Variablen.	[OMG09]
ReplyAction	Nimmt Datentoken von AcceptActions an und gibt diese an das rufende Element zurück.	[RJB05], [OMG09]

Aktion	Bemerkung	Quelle
SendObjectAction	Sendet ein Objekt an ein anderes Objekt um dort ein Verhalten zu triggern.	[RJB05], [OMG09]
SendSignalAction	Erzeugt ein Signal aus den eingehenden Datentoken und sendet dieses an ein konkretes Objekt, um z.B. die Transition eines Zustandsautomaten zu triggern.	[RJB05], [OMG09]
StartClassifier BehaviorAction	Führt das Verhalten des Typs des eingehenden Datenflusses aus. Wird nur implementierungsnah verwendet.	[OMG09]
StartObject BehaviorAction	Führt das Verhalten einer Instanz des Typs des eingehenden Datenflusses aus. Wird nur implementierungsnah verwendet.	[OMG09]
StartOwned BehaviorAction	Alternativer Name für StartObjectBehaviorAction; teilweise in der Literatur verwendet	[RJB05]
TestIdentityAction	Prüft zwei Objekte auf Gleichheit	[RJB05], [OMG09]
TimeAction	Familie von Aktionen, die die momentane Zeit liefern oder die Zeit zwischen dem Senden und dem Empfangen einer Nachricht protokollieren.	[RJB05]
UnmarshallAction	Teilt ein Objekt in seine strukturellen Bestandteile auf und liefert diese zurück.	[OMG09]
ValueSpecification Action	Liefert einen Wert, entweder als Konstante oder als Ergebnis eines logischen Ausdrucks (z.B. eines OCL-Ausdrucks).	[OMG09]
WriteAction	Ändert einen Wert einer Variable; nicht in der UML Superstructure enthalten	[RJB05]

Tabelle 2.2: Aktionen in der UML 2

Quelle: Eigene Darstellung

durch einen eingehenden Objektfluss realisiert werden, der ein Token des entsprechenden Typs transportiert.

Die eingehenden Objektflüsse beider CallActions müssen mit den Parameterlisten des adressierten Verhaltens bzw. der aufzurufenden Methode in Anzahl und Typen übereinstimmen. Da die Ausführung einer CallOperationAction in der Ausführung des mit der Methode verknüpften Verhaltens resultiert, muss bei der Verwendung von CallOperationActions zusätzlich sichergestellt werden, dass diese Bedingung auch auf das letztendlich adressierte Verhalten zutrifft.

Die CallActions erlauben es, sowohl eine Wiederverwendung von Verhalten als auch Polymorphismus während der Modellierung zu realisieren. Die Wiederverwendung ist gegeben, wenn die Beschreibung des Verhaltens mit der Deklaration einer Prozedur in einer Programmiersprache verglichen wird und die CallAction mit deren Verwendung durch den

Programmierer. Der Polymorphismus wird implizit möglich, weil die UML es erlaubt, auch Interfaces als Typen für die Objektflüsse zu verwenden.

Abbildung 2.7: CallBehaviorAction und CallOperationAction

Quelle: Eigene Darstellung

Zur graphischen Darstellung von CallBehaviorActions werden die Aktionsknoten um einen nach unten gerichteten Dreizack erweitert (siehe Abbildung 2.7, oben); CallOperationActions werden nicht graphisch kenntlich gemacht. Bei der Beschriftung von CallBehaviorActions wird hinter den Namen der Aktion, getrennt durch einen Doppelpunkt, der Name des adressierten Verhaltens geschrieben. Bei CallOperationActions wird unter den Namen der Aktion, in runden Klammern und durch zwei Doppelpunkte getrennt, der Name der Klasse sowie der aufzurufenden Methode geschrieben (siehe Abbildung 2.7, unten).

Sonderfälle im Aktionenmodell der UML

Wie weiter oben bereits erwähnt, sind Aktionen aus Sicht der Aktivität, in der sie enthalten sind, atomar und haben festgelegte Start- und Endbedingungen. Diese Konsequenz findet allerdings bei CallActions nur eingeschränkt Anwendung, da beim Aufruf von Verhalten (implizit oder explizit) immer dessen Spezifikation berücksichtigt werden muss.

Aus diesem Grund wurden drei Sonderfälle für CallActions geschaffen, um kontinuierliche Datenströme, Ausnahmebehandlungen und Parametergruppen modellieren zu können.

Im Falle **kontinuierlicher Datenströme** (Streams) startet eine Aktion, wenn an allen eingehenden Flüssen die entsprechenden Token vorliegen. Normalerweise kann während der Ausführung eine Aktion keine weiteren Daten mehr annehmen. Bei Flüssen, die mit `{STREAM}` annotiert sind, gelten diese Regeln jedoch nicht. Die Aktion muss auch nicht auf einen ausstehenden Datenstrom warten, wenn alle anderen Token bereits vorliegen. Sollten während der Ausführung der Aktion weitere

Daten über einen solchen Datenfluss die Aktion erreichen, können diese von der Aktion direkt weiterverarbeitet werden.

Datenströme sind explizit nicht optional: Um ihre Ausführung zu beenden, muss die Aktion mindestens ein Datentoken aus dem Datenstrom verarbeitet haben. Der Unterschied zwischen normalen Datenflüssen und Datenströmen liegt lediglich darin, dass letztere ihre Datentoken auch noch liefern können, wenn die Aktion bereits ausgeführt wird. Im Gegensatz dazu müssen an ausgehenden Datenströmen keine Token anliegen, damit eine Aktion terminieren kann. In [BOC03B] wird erläutert, dass diese Realisierung von Datenströmen eher recht als schlecht funktioniert und mit Sicherheit noch überarbeitet werden wird.

Mögliche **Ausnahmen** (Exceptions), die während der Ausführung einer CallAction auftreten können, werden durch einen Pin mit einem Dreieck dargestellt. Tritt während der Ausführung der Aktion eine Ausnahme auf, wird die Ausführung der Aktion sofort beendet. In diesem Fall wird nur an den ausgehenden Datenfluss des Pins, der die Ausnahme repräsentiert, ein Token geliefert. Alle anderen Daten- und Kontrollflüsse werden ignoriert. Eine CallAction kann mehrere verschiedene Ausnahmen auslösen, von denen jedoch jeweils nur eine auftreten kann.

Sollte eine Ausnahme auftreten, werden über einen Datenstrom eventuell bereits gesendete Datentoken davon nicht mehr beeinflusst.

Der letzte Sonderfall für CallActions sind die sogenannten **Parametergruppen**. Diese können verwendet werden, wenn beim Start einer Aktion nicht sicher davon ausgegangen werden kann, welche Datentoken sie am Ende liefern wird, etwa weil die Aktion selbst für eine Prüfung der Eingangsdaten verantwortlich ist. Eine Parametergruppe wird durch ein Rechteck um die Pins, die sie umfasst, dargestellt. Dabei muss die Parametergruppe mindestens einen Pin umfassen. Wenn die Aktion terminiert, werden nur in eine der modellierten Parametergruppen Datentoken bereitgestellt. Die Datentoken der einzelnen Pins sind aber auch bei Parametergruppen im weiteren Verlauf voneinander unabhängig, die Parametergruppe hat also keinen Einfluss darauf, was mit den in ihr enthaltenen Parametern weiter geschehen wird. Parametergruppen können keine Kontroll- sondern nur Datenflüsse enthalten.

In [BOC03B] werden folgende Bedingungen für den Start und das Ende von CallActions definiert:

- An allen eingehenden Flüssen, die keine Datenströme sind, müssen Token vorliegen, damit die Aktion starten kann; gibt es nur eingehende Datenströme, muss an mindestens einem ein

Token vorliegen

- An allen eingehenden Flüssen muss mindesten ein Token vorgelegen haben, damit die Aktion beendet werden kann; es gibt keine optionalen Eingangsdaten.
- An allen ausgehenden Datenflüssen, die keine Datenströme und keine Ausnahmen sind, müssen Token vorliegen, wenn die Aktion beendet wird. Alternativ wird die Aktion beendet, sobald an einem ausgehenden Ausnahme-Datenfluss ein Token vorliegt.
- Eine Aktion, die eingehende Parametergruppen besitzt, kann für eine Ausführung nur Token einer dieser Gruppen verarbeiten. Besitzt die Aktion ausgehende Parametergruppen, können nur Token für eine dieser Gruppen vorliegen, wenn die Aktion beendet wird. Die drei vorigen Regeln gelten für jede Parametergruppe separat.

Die beschriebenen Sonderfälle werden normalerweise explizit mit Kontrollknoten modelliert. Sobald sie von den gängigen Modellierungswerkzeugen vollständig unterstützt werden, stellen sie jedoch eine gute Alternative dazu dar, insbesondere, wenn die Entscheidungsfindung im Sinne der Wiederverwendung in separat definiertes Verhalten ausgelagert werden soll.

Lokale Bedingungen

Alle Aktionen in einer Aktivität können mit sogenannten Lokalen Bedingungen versehen werden. Dies geschieht im Modell über eine Annotation als `<<localPrecondition>>` und `<<localPostcondition>>` an der Aktion. Üblicherweise werden diese Bedingungen als OCL-Ausdrücke formuliert. Da die UML aber keine Aussage darüber macht, ob die lokalen Bedingungen zur Entwurfs- oder Laufzeit ausgewertet werden, ist es auch denkbar, statt OCL-Ausdrücken anderes Verhalten zu referenzieren.

Eine mögliche Verwendung lokaler Bedingungen kann beispielsweise die Einführung von kontextabhängigen Aspekten sein. Der Kontext wäre dann die Aktivität.

2.1.3 Objektknoten

Die Aufgabe von Objektknoten in Aktivitäten ist es, Datentoken so lange zu speichern, bis die Weiterleitung der Token möglich ist.

Objektknoten in der UML verfügen über folgende Charakteristika:

- Aktiv: Sind Datentoken in den Objektknoten enthalten, initiiert der Objektknoten die

Weiterleitung der Token.

- Reduzierend: Datentoken, die eventuell über einen ausgehenden Datenfluss weitergeleitet werden können, stehen nicht zur Weiterleitung auf anderen ausgehenden Datenflüssen zur Verfügung.
- Flüchtig: Datentoken bleiben nicht in Objektknoten gespeichert, wenn die Aktivität, die die Objektknoten enthält, beendet wird.

Diesen Charakteristika gegenüber stehen andere Eigenschaften aus früheren Versionen der UML, die in 2.2 erläutert werden.

Activity Parameter Nodes und Pins

In 2.1.2 wurde erklärt, dass die Parameterlisten von CallActions mit denen des adressierten Verhaltens übereinstimmen müssen. Dies wird durch zwei Arten von Objektknoten realisiert: auf Seite der CallAction durch die Pins, auf Seite einer Aktivität durch Activity Parameter Nodes. Die graphische Darstellung von Pins ist in Abbildung 2.4 ganz unten zu sehen, die von Activity Parameter Nodes in Abbildung 2.5.

Eine Aktivität kann mehrere Activity Parameter Nodes für eingehende und ausgehende Objektflüsse besitzen. Für ausgehende Objektflüsse gilt, dass Datentoken so lange in den jeweiligen Activity Parameter Nodes gepuffert werden, bis alle anderen ebenfalls vorliegen. Sobald alle Datentoken vorliegen, ist die Aktivität beendet und die Datentoken werden an das aufrufende Modellelement übergeben. Im Falle von eingehenden Objektflüssen liegen die Datentoken alle in dem Moment vor, in dem die Aktivität startet. Je nachdem, ob eine Weiterverarbeitung direkt möglich ist, können die Datentoken in den eingehenden Activity Parameter Nodes ebenfalls zwischengespeichert werden.

Die Activity Parameter Nodes müssen den Parametern des Verhaltens entsprechen. Die Parameter des Verhaltens sind im UML Metamodell in der Klasse Behavior angesiedelt. Somit kann das Verhalten jede der drei in UML enthaltenen Arten sein, nämlich Aktivitäten, Zustandsautomaten und Interaktionen. Unglücklicherweise wird in keiner der hier verwendeten und im Literaturverzeichnis aufgeführten Werke eine Aussage gemacht, wie die Modellierung einer Zustandsmaschine oder einer Interaktion in Zusammenhang mit Activity Parameter Nodes aussehen könnte, so dass sich sämtliche weiteren Untersuchungen im Rahmen dieser Arbeit auf Aktivitäten beziehen.

Allen Objektknoten ist gemein, dass sie den Typ der Objekte spezifizieren, für die sie Token

speichern können. Diese Angabe ist allerdings optional; wurde kein Typ für einen Objektknoten angegeben, kann dieser Objektknoten alle Formen von Datentoken speichern. Zusätzlich zum Typ können Objektknoten noch einen Zustand voraussetzen, in dem sich Objekte, die gespeichert werden können, befinden müssen.

Objektknoten können auch mehrere Token speichern, sowohl für verschiedene als auch für identische Objekte.

Für die Pins von Aktionen gilt im Wesentlichen das Gleiche wie für die Activity Parameter Nodes. Im Falle, dass mehrere eingehende Pins existieren, speichern diese Datentoken so lange, bis an allen Pins Datentoken vorliegen. Die genauen Bedingungen, die erfüllt sein müssen, damit eine Aktion starten kann, sind in 2.1.2 beschrieben. Falls durch die Aktion eine Aktivität aufgerufen wird, werden die eingehenden Datentoken direkt an die Activity Parameter Nodes weitergeleitet.

Ein Spezialfall der Pins sind Value Pins, die genutzt werden, um konstante Werte bereitzustellen. Diese Werte werden auf die gleiche Weise beschrieben, wie die in den Guards verwendeten Ausdrücke, die in 2.1.4 beschrieben werden. Die Value Pins werden graphisch als normale Pins mit einem zusätzlich notierten Wert dargestellt.

Zusätzlich können Pins noch mit einem Effekt annotiert werden. Dieser Effekt soll beschreiben, was die Aktion mit den Token tun wird. Mögliche Effekte sind `create`, `read`, `update` und `delete`. Create ist nur für ausgehende Pins, delete nur für eingehende Pins anwendbar.

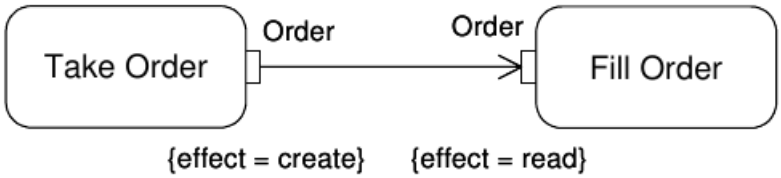

Abbildung 2.8: Darstellung von Effekten an Pins

Quelle: [BOC03D]

Auch wenn Effekte zwar bereits seit 2003 Bestandteil der UML 2 sind, können sie noch nicht in allen Modellierungswerkzeugen benutzt werden. Sowohl MagicDraw als auch der EnterpriseArchitekt bieten keine Unterstützung dafür. Ein Beispiel für Effekte ist in Abbildung 2.8 zu sehen.

Verarbeitung mehrerer Token

Wie bereits erwähnt, haben Objektknoten die Fähigkeit, mehrere Datentoken zu speichern. Diese können sowohl unterschiedliche als auch identische Objekte referenzieren. Die obere Grenze für die Menge an Datentoken, die ein Objektknoten aufnehmen kann, wird als `upperBound` bezeichnet und in der Form `{ upperBound = Anzahl }` annotiert.

Diese obere Grenze kommt zur Laufzeit zum Tragen, denn wenn ein Objektknoten keine neuen Token mehr aufnehmen kann, wird eine vorausgegangene Aktion eventuell blockiert, da sie keine Datentoken mehr absetzen kann. Die Datentoken werden dann so lange in den ausgehenden Pins gepuffert, bis deren obere Grenze erreicht ist, danach kann die Aktion nicht mehr ausgeführt werden. Dies ist die einzige Situation, in der eine Aktion nicht starten wird, auch wenn alle Bedingungen zum Start, wie sie in 2.1.2 erklärt wurden, erfüllt sind.

Gegenstück zum `upperBound` ist der `lowerBound`, der eine Mindestmenge an Token spezifiziert, die ein Objektknoten enthalten muss, bevor er beginnt, Token weiterzuleiten.

Diese Puffer werden werden nur in Objektknoten in Aktivitäten annotiert. Parameter von Verhalten oder Operationen können nicht mit Puffern versehen werden, da sie lediglich Typen und Richtung des Datenflusses spezifizieren. Im Falle von CallActions müssen also die Pufferkapazitäten an die Pins der Aktionen angehängt werden.

Durch puffernde Objektknoten kann auch eine implizite Parallelverarbeitung erreicht werden. Dazu muss die entsprechende Aktion mit { `reentrant` } annotiert werden, was bewirkt, dass mehrere Kopien der Aktion parallel gestartet werden. Die Aktion wird für jedes eingehende Datentoken neu gestartet, unabhängig davon, wie oft sie bereits ausgeführt wurde oder wie viele Kopien sich noch in Ausführung befinden. Es ist diesen Aktionen nicht möglich, Datenströme zu empfangen, da es keine Möglichkeit zu Voraussage gibt, welche Aktion zur Laufzeit ein Datentoken aus dem Datenstrom erhält.

Die Reihenfolge, in der von den Objektknoten gepufferte Datentoken weitergeleitet werden, kann ebenfalls beeinflusst werden. Normalerweise werden die Datentoken in der gleichen Reihenfolge, in der sie ankommen, weitergeleitet (first-in, first-out). Dieses Verhalten kann jedoch umgekehrt werden, so dass die zuletzt angekommenen Datentoken als erste weitergeleitet werden (last-in, first-out). Der Anwender hat zudem noch die Möglichkeit, die Reihenfolge der Weiterleitung durch ein selbst definiertes Verhalten zu bestimmen, das zu diesem Zweck an den entsprechenden Objektknoten mit Hilfe eines Kommentars als `<<selection>>` oder durch eine Annotation nach

dem Schema `{ selection = Verhalten }` gebunden wird.

Letzterer Mechanismus kann sowohl auf eingehende wie auch auf ausgehende Pins angewendet werden. Die beiden anderen Mechanismen sind nur bei eingehenden Pins anwendbar. Damit wird es möglich, eine Sortierung nicht erst bei der Auswahl eines Tokens aus der gepufferten Menge zu treffen, sondern eingehende Token direkt an der richtigen Stelle einzusortieren. Je nach Anwendung kann dies zu Vorteilen wie z.B. einem Performancegewinn führen.

Durch die Art und Weise wie Verhalten und Methoden mit CallActions angesteuert werden können, wird klar, dass diese Aktionen die minimale und maximale Anzahl an Datentoken, die die Parameter des Verhaltens oder der Methode pro Aufruf entgegennehmen können, berücksichtigen müssen. Dabei ist festgelegt, dass eine CallAction erst starten kann, wenn die minimal erforderliche Anzahl an Datentoken vorliegt. Für die minimale Anzahl ist auch der Wert 0 zulässig, was theoretisch bedeuten würde, dass die CallAction nicht auf ein Datentoken warten muss. Die sich daraus ergebende Unklarheit wurde bereits in [BOC03D] erläutert, jedoch haben Recherchen in der aktuellen Spezifikation [OMG09] keine aktuelle Lösung in der UML ergeben.

Die Anzahl der Parameter wird in eckigen Klammern an den entsprechenden Pin geschrieben, wie es für die Notation von Multiplizitäten in der UML üblich ist. Sind keine Anzahlen für einen Pin hinterlegt, sollte der Anwender nach [BOC03D] von einer Multiplizität von 1 ausgehen, was allerdings nicht in der UML definiert wird. Ist die maximale Anzahl größer als 1, kann die Menge der Datentoken zum einen als sortiert annotiert werden und zum anderen kann erlaubt werden, dass es mehrere Datentoken gibt, die das gleiche Objekt referenzieren.

Bisher nicht festgelegt wurde ein möglicher Zusammenhang zwischen Datenströmen und der Multiplizität von Parametern. Mögliche Interpretationen werden an dieser Stelle nicht gemacht, da dies den Rahmen dieser Arbeit sprengt.

Eine andere Art minimale Mengen an Datentoken zu gewährleisten besteht darin, den Datenfluss zwischen zwei Pins mit einer Gewichtung zu versehen. Dazu wird der Datenfluss mit `{ weight = Anzahl }` annotiert. Es müssen in diesem Fall erst entsprechend viele Datentoken vorliegen, bevor sie alle auf einmal vom Datenfluss transportiert werden können. Standardmäßig hat weight für Datenflüsse den Wert `1`. Einen besonderen Wert stellt `all` dar. Ein Objektfluss mit `{ weight = all }` transportiert alle Datentoken auf einmal.

Konkurrierende Datentoken

Die UML erlaubt es, dass ein Objektknoten mehrere ausgehende Datenflüsse besitzt. Allerdings können Objektknoten nicht, wie die in 2.1.4 erklärten Fork Nodes, Datentoken kopieren. Da die UML in diesem Fall keine Aussage darüber macht, welcher der ausgehenden Datenflüsse das Token bekommen wird, kann auf diesem Weg eine zufällige Verteilung der Datenflüsse modelliert werden.

Das Konzept der UML sieht in diesem Fall vor, dass ein Objektknoten allen ausgehenden Flüssen ein Token anbietet und diese das Token wiederum den Objektknoten anbieten, zu denen sie führen. Sobald einer der Objektknoten das Token entgegennehmen kann, wird dieser Fluss aktiv. Damit wird sichergestellt, dass nicht eine Aktion, deren Ausführung lange Zeit dauert, alle Datenflüsse blockiert, weil ein Datentoken nicht zu ihr weitergeleitet werden kann.

Ein weiterer interessanter Fall tritt auf, wenn ein Objektfluss durch eine Decision Node (siehe 2.1.4) geleitet wird. Dazu muss die Decision Node einen durch einen Guard blockierten Datenfluss und einen Datenfluss für den Fall, dass der Guard fehlschlägt, besitzen. Schlägt der Guard fehl und kann das Datentoken vom Objektknoten des alternativen Weges nicht angenommen werden, bleibt es im Objektknoten vor der Decision Node. Der Objektknoten versucht nun permanent, das Datentoken weiterzuleiten, bis entweder der Guard erfolgreich ausgewertet wird oder der Objektknoten des alternativen Datenflusses das Token annehmen kann. Dadurch, dass Kontrollknoten nicht die Fähigkeit zur Speicherung und Veränderung von Datentoken besitzen, können die Kontrollknoten immer wieder von den Datentoken passiert werden. Sobald es dann eine Möglichkeit dazu gibt, werden die Datentoken an den ersten verfügbaren Objektknoten weitergeleitet.

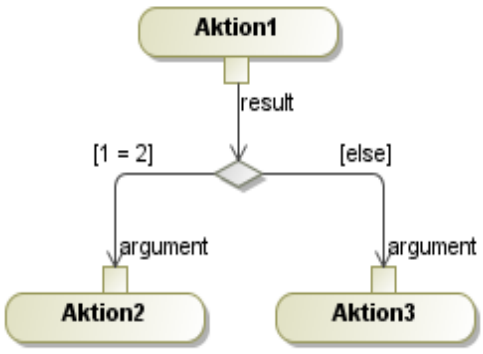

Abbildung 2.9: Weiterleitung von Datentoken

Quelle: Eigene Darstellung

Diese Situation ist in Abbildung 2.9 dargestellt. Wird Aktion1 abgeschlossen, erhält der Pin result

ein Datentoken. Der Guard auf dem Zweig zu Aktion2 schlägt fehl, also wird der else-Zweig der Decision Node ausgewählt. Wenn aber der Pin argument der Aktion3 bereits ein Token enthält, kann das Token nicht weitergeleitet werden und verbleibt im Pin result der Aktion1. Sollte irgendwann einmal die Auswertung des Guard nicht mehr fehlschlagen (was in diesem Fall eher unwahrscheinlich ist), bevor der Pin argument von Aktion3 wieder frei ist, würde das Token zu Aktion2 geschickt.

Ähnlich wird bei einer Transformation von Datentoken während des Transports über einen Datenfluss verfahren. An jeden Datenfluss kann ein Verhalten als `<<transformation>>` gebunden werden, das auf jedes Datentoken angewendet wird. In diesem Fall wird das Datentoken aus dem Objektknoten aus dem der Datenfluss kommt erst entfernt, wenn das transformierte Datentoken vom Objektknoten, in dem der Datenfluss endet, angenommen wurde.

Eine besondere Form der Objektknoten stellen Central Buffers dar. Diese können verwendet werden, um parallel aus mehreren Objektknoten Datentoken entgegenzunehmen, zwischenzuspeichern und nach Bedarf an mehrere Objektknoten weiterzuleiten. Central Buffer müssen immer explizit modelliert werden.

Um Deadlocks zu verhindern, ist das Verhalten von eingehenden Pins von Aktionen so festgelegt, dass ein Pin erst dann ein Token annehmen kann, wenn alle anderen Pins ebenfalls Token annehmen können. Es ist den eingehenden Pins aber nach wie vor möglich, Datentoken zu puffern, sofern auch alle anderen Pins eine entsprechende Menge an Datentoken puffern.

In [BOC03D] ist dies an einem Beispiel gut verdeutlicht: Zwei Schreiner benötigen sowohl ein Verlängerungskabel als auch einen Bohrer, um mit dem Bohren zu beginnen. Keiner der beiden kann mit dem Bohren anfangen, wenn er nur einen Bohrer oder nur ein Verlängerungskabel hat. Erst wenn beides verfügbar ist, kann er mit der Arbeit beginnen. Die UML stellt sicher, dass die Token für Kabel und Bohrer erst dann gleichzeitig an einen der beiden Schreiner weitergeleitet werden, wenn beide Token verfügbar sind und dieser bereit ist, diese zu empfangen (siehe Abbildung 2.10).

Leider gilt auch für den Central Buffer, dass nicht alle Modellierungswerkzeuge dieses Element der UML unterstützen.

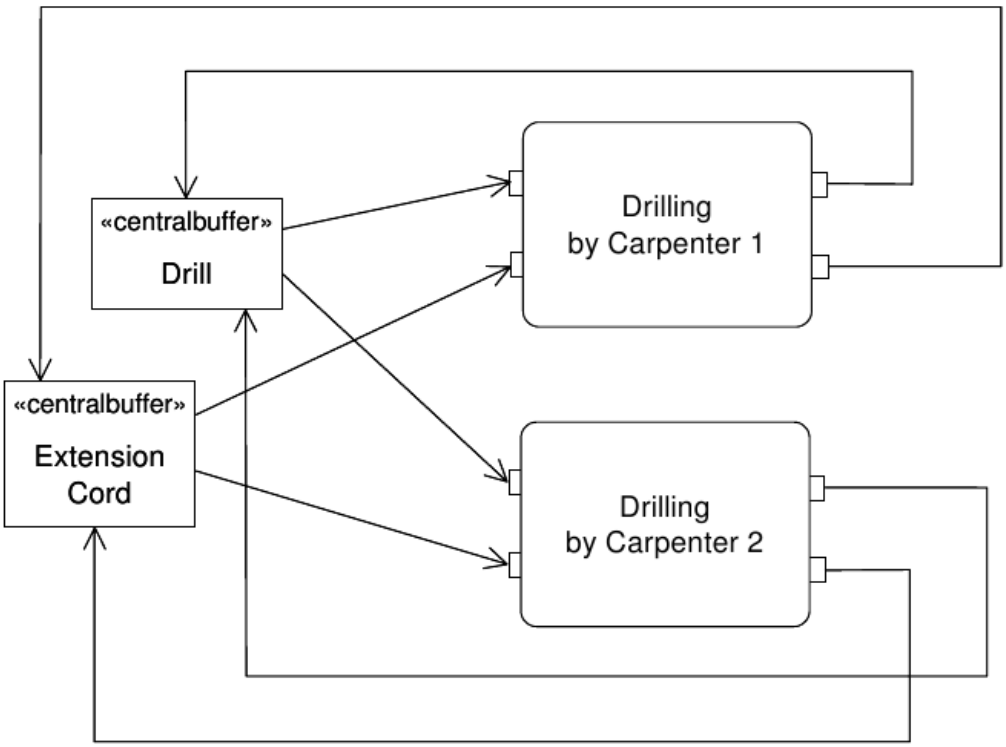

Abbildung 2.10: Vermeidung von Deadlocks mit Datentoken und CentralBuffer

Quelle: [BOC03D]

Data Stores

Normale Objektknoten können Datentoken nicht kopieren, wenn sie über einen ausgehenden Datenfluss weitergeleitet werden, da diese dann nicht mehr im Objektknoten gespeichert sind. Data Stores unterliegen dieser Einschränkung nicht. Sie haben eine Menge an Datentoken gespeichert und können nach Bedarf Kopien dieser Token erstellen und über ausgehende Objektflüsse weiterleiten. Die Datentoken im Data Store bleiben erhalten.

Es ist nicht möglich, explizit Token aus einem Data Store zu löschen. Dies geschieht allerdings implizit mit dem Ende der Aktivität, in der der Data Store enthalten ist.

Sofern ein Datentoken über einen eingehenden Datenfluss einen Data Store erreicht und ein identisches Token, d.h. ein Token, das das gleiche Objekt referenziert, im Data Store vorhanden ist, wird dieses Token ersetzt. Damit unterscheiden sich Data Stores ebenso deutlich von Objektknoten, die die Fähigkeit zum Puffern haben, wie von den Central Buffers.

An ausgehenden Datenflüssen von Data Stores kann Verhalten wie auch bei normalen Objektknote sowohl als `<<selection>>` als auch als `<<transformation>>` gebunden werden.

2.1.4 Kontrollknoten

Kontrollknoten werden in Aktivitäten genutzt, um den Weg, den Token durch den Flussgraph nehmen, zu steuern. Die Kontrollknoten können dabei sowohl für Daten- als auch Kontrolltoken eingesetzt werden, wobei einige Regeln zu beachten sind, die weiter unten in diesem Abschnitt erklärt werden.

Initial und Final Nodes

Wie bereits in 2.1.1 erklärt, gibt es sieben verschiedene Arten von Kontrollknoten. Eine besondere Rolle kommt dabei den Initial Nodes und den Final Nodes zu.

Initial Nodes sind die Startpunkte von Aktivitäten. Sobald eine Aktivität startet, wird ein Kontrolltoken auf die Initial Node der Aktivität gesetzt und von dieser sofort über einen ausgehenden Kontrollfluss weitergeleitet. Dieses Verhalten von Initial Nodes kann nicht beeinflusst werden. Initial Nodes können lediglich ausgehende Kontrollflüsse haben, ausgehende Datenflüsse sind aufgrund des Charakters der Initial Nodes ebensowenig möglich, wie eingehende Flüsse jedweder Art.

Wenn in einer Aktivität mehr als eine Initial Node enthalten ist, wird bei Start der Aktivität automatisch auf jeder dieser Initial Nodes ein Kontrolltoken platziert und somit mehrere parallele Kontrollflüsse gestartet.

Initial Nodes können mehr als einen ausgehenden Kontrollfluss besitzen. Diese sollten dann allerdings mit Bedingungen versehen werden, da die Initial Node nicht die Fähigkeit hat, Token zu kopieren. In so einem Fall verhält sich die Initial Node dann wie die weiter unten in diesem Abschnitt beschriebene Decision Node.

Final Nodes beenden Flüsse in Aktivitäten. Dabei wird zwischen zwei Arten von Final Nodes unterschieden. Activity Final Nodes beenden die gesamte Aktivität, Flow Final Nodes lediglich den aktuellen Kontrollfluss. Final Nodes können beliebig viele eingehende Kontroll- und Datenflüsse besitzen, aber keine ausgehenden.

Die Flow Final Node nimmt dabei alle Token an, die sie erreichen und löscht sie. Da Kontrolltoken lediglich virtuell existieren und Datentoken nur Referenzen auf Objekte sind, beeinflusst das Löschen der Token nicht die Objekte, auf die diese verwiesen haben. Es kann beliebig viele Flow Final Nodes in einer Aktivität geben, es können aber auch mehrere parallele Flüsse in der gleichen Flow Final Node enden. Sobald alle Token in einer Aktivität gelöscht sind, terminiert die Aktivität.

Im Unterschied zu Flow Final Nodes, die lediglich Token löschen, wird beim Erreichen einer Activity Final Node die gesamte Aktivität beendet. Damit werden nicht nur der eingehende Fluss der Final Node, sondern auch alle parallelen Flüsse in der Aktivität sofort gestoppt. Somit können Activity Final Nodes nicht als Alternative zu den Flow Final Nodes verwendet werden, da dies zu Race Conditions führen würde.

Decision und Merge Nodes

Decision Nodes werden verwendet, um die Richtung der Flüsse in der Aktivität zur Laufzeit zu bestimmen. Dazu werden die ausgehenden Flüsse der Decision Node normalerweise mit Guards versehen, die für jedes Kontroll- und Datentoken, das die Decision Node erreicht, neu ausgewertet werden.

Da die UML keinerlei Aussagen darüber macht, in welcher Reihenfolge diese Guards ausgewertet werden und sogar eine parallele Auswertung möglich ist, dürfen sie keine Seiteneffekte haben. Üblicherweise wird zur Notation der Guards OCL verwendet, dies ist jedoch nicht verpflichtend.

Soll ein bestimmter Fluss immer dann ausgewählt werden, wenn alle anderen Flüsse durch ihre Guards blockiert sind, wird dazu der Guard `else` verwendet. Dies ist der einzige Guard, den die UML im Zusammenhang mit Decision Nodes explizit vorgibt. Der Guard `else` darf nur an einem einzigen ausgehenden Fluss der Decision Node verwendet werden.

Es liegt beim Anwender, sicherzustellen, dass nur ein Guard erfolgreich ausgewertet wird, da andernfalls Race Conditions entstehen.

Wenn alle Guards fehlschlagen, werden die Kontroll- und Datentoken an den Objektknoten zurückgeschickt, von dem sie kommen, da Kontrollknoten keine Token puffern können (siehe 2.1.3). Es ist vorgesehen, dass genau ein Guard erfolgreich ausgewertet wird, jedoch wird kein Mechanismus definiert, der das gewährleisten könnte. Hier besteht die Gefahr, bereits im Modell Deadlocks zu verursachen. Im Falle, dass alle Guards erfolgreich ausgewertet werden, ist innerhalb der virtuellen Maschine, in der die Aktivität ausgeführt wird, unklar, was passieren wird.

Während der Zeit, die für die Auswertung der Guards benötigt wird, verbleiben die Token in ihren Aktionen oder Objektknoten. Erst wenn der erste mögliche Pfad zur nächsten Aktion oder zum nächsten Objektknoten ermittelt ist, wird ein Token weitergeleitet. Somit muss zwar die Zeit für die Auswertung der Guards, jedoch nicht die Zeit für den Transport der Token berücksichtigt werden.

Die UML erlaubt es auch, die Bedingungen für die Guards durch ein Verhalten zu beschreiben und

dieses an die Decision Node zu binden. Das ermöglicht die Wiederverwendung von Auswertungen auf die gleiche Weise, wie bei CallBehaviorActions. Die Bindung des Verhaltens erfolgt dabei über einen Kommentar, der mit `<<decisionInput>>` annotiert ist oder über eine Annotation nach dem Schema `{ decisionInput = Verhalten }`. Jedes Token, das die Decision Node erreicht, wird dann zunächst an das entsprechende Verhalten weitergeleitet, bevor die Guards ausgewertet werden. Darüber, wie das Resultat der Auswertung an die Guards übergeben wird, macht die UML keine Aussage.

Die UML stellt mittlerweile auch einige Alternativen zu Decision Nodes bereit, wie sie z.B. in 2.1.5 beschrieben sind.

Das Gegenstück zu den Decision Nodes bilden die Merge Nodes, die unterschiedliche Kontrollflüsse zusammenführen. Merge Nodes leiten dabei sämtlich Token, die an den eingehenden Flüssen angekommen, augenblicklich weiter zum ausgehenden Fluss, von dem sie nur einen besitzen dürfen. Üblicherweise werden Merge Nodes eingesetzt, um die alternativen Flüsse von vorausgegangenen Decision Nodes wieder zusammenzuführen, können aber auch genutzt werden, um parallele Flüsse zu serialisieren.

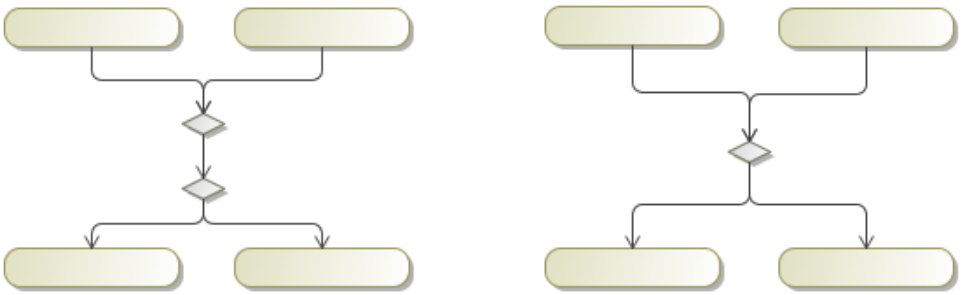

Abbildung 2.11: Zusammenfassung aufeinander folgender Merge/Decision Nodes

Quelle: Eigene Darstellung

Direkt aufeinanderfolgende Decision und Merge Nodes können graphisch zusammengefasst werden, wobei dies im Allgemeinen aus Gründen der Übersichtlichkeit nicht empfohlen werden kann (siehe Abbildung 2.11).

Fork und Join Nodes

Um die parallele Ausführung unterschiedlicher Flüsse zu ermöglichen, werden Fork Nodes verwendet. Fork Nodes kopieren alle Kontroll- und Datentoken, die sie erreichen, entsprechend der

Anzahl ihrer ausgehenden Flüsse. Die Kopie von Datentoken stellt lediglich eine Kopie von Objektreferenzen dar. Wenn die parallelen Flüsse nach einem Fork also das gleiche Objekt bearbeiten, muss der Anwender sicherstellen, dass es nicht zu Synchronisationsproblemen kommt.

Simple Forks können auch erreicht werden, wenn eine Aktion mehrere ausgehende Flüsse besitzt. Diese werden genau dann aktiv, wenn die Aktion beendet ist und laufen parallel ab. Allerdings gibt es auf diesem Weg keine Möglichkeit, Datentoken zu vervielflältigen. Auch kann es bei ausgehenden Datenflüssen von Aktionen passieren, dass die Pins die Datentoken puffern, was bei einer Fork Node nicht geschieht.

Das Gegenstück zu Fork Nodes können wie bereits erwähnt Merge Nodes bilden, um eine Serialisierung paralleler Flüsse zu erreichen, oder Join Nodes, die parallele Flüsse zu einem einheitlichen Fluss zusammenführen und damit synchronisieren.

Join Nodes besitzen mehrere eingehende Flüsse und einen ausgehenden Fluss. Damit ein Token an den ausgehenden Fluss übergeben werden kann, müssen an allen eingehenden Flüssen die Kontroll- und Datentoken vorliegen. Die Join Node kombiniert dabei ein Token von jedem eingehenden Fluss zu einem neuen Token für den ausgehenden Fluss nach folgenden Regeln:

- Handelt es sich bei allen Token um Kontrolltoken, wird ein Kontrolltoken an den ausgehenden Fluss übergeben
- Sind einige (oder alle) der Token Datentoken und einige (oder keine) Kontrolltoken, werden Kontrolltoken, sofern vorhanden, gelöscht und die Datentoken an den ausgehenden Fluss übergeben.

Kommen mehrere Kontrolltoken nacheinander bei der Join Node an, finden diese Regel für jedes Kontrolltoken erneut Anwendung.

Sind mehrere Datentoken vorhanden und referenzieren dasselbe Objekt, werden diese Token durch die Join Node zu einem einzigen Token zusammengefasst. Dieses Verhalten der Join Node kann aber unterdrückt werden.

Handelt es sich bei den Datentoken nicht um Referenzen auf dasselbe Objekt, sollte eventuell auf eine Join Node verzichtet und die parallelen Datenflüsse direkt mit einer Aktion verbunden werden, die eine implizite Synchronisation durchführt, da sie erst ausgeführt wird, wenn an allen eingehenden Flüssen Token anliegen (siehe Abbildung 2.12).

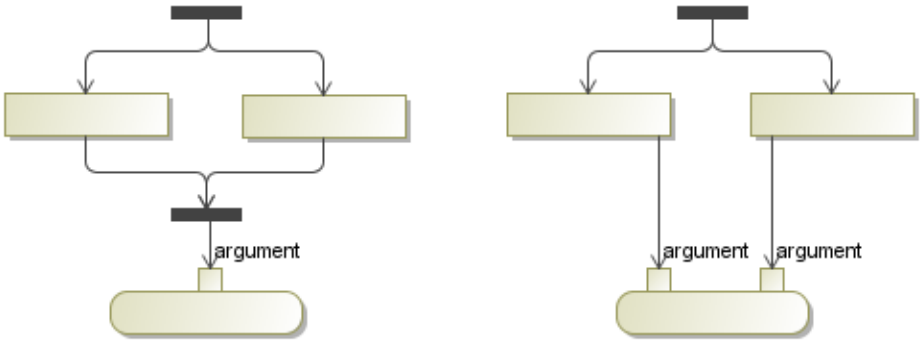

Abbildung 2.12: Expliziter und impliziter Join

Quelle: Eigene Darstellung

Die UML überlässt es dem Anwender, sicherzustellen, dass eine Join Node keine eingehenden Flüsse hat, auf denen eventuell niemals Token geliefert werden.

Parallele Flüsse in Aktivitäten müssen nicht zwangsläufig wieder zusammengeführt werden. Sie können alternativ serialisiert werden oder durch Final Nodes beendet werden.

Normalerweise wird die Join Node aktiv, wenn an allen eingehenden Flüssen Token anliegen. Der Anwender hat aber darüber hinaus die Möglichkeit, mit einer Join Specification die Bedingungen

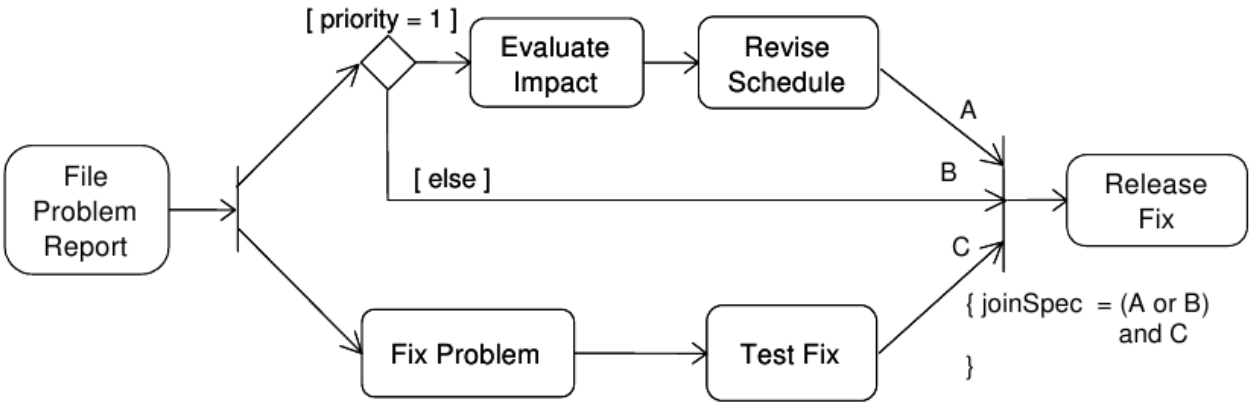

Abbildung 2.13: Beispiel für Join Specification

Quelle: [BOC03C]

festzulegen, unter denen Token in welchen Kombinationen angenommen werden sollen. Diese Join Specification kann als Annotation nach dem Schema `{ joinSpec = Ausdruck }` an die Join Node geschrieben werden. Dabei kann die OCL für die Beschreibung der Ausdrücke zum Einsatz kommen, wobei dies nicht explizit vom Anwender verlangt wird (siehe Abbildung 2.13).

Wie bei vielen anderen graphischen Notationen gilt auch bei der Join Specification, dass nicht alle Modellierungswerkzeuge diese unterstützen.

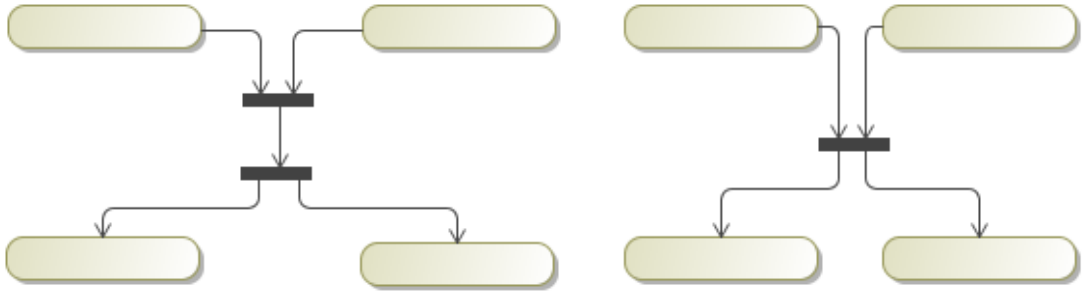

Abbildung 2.14: Zusammenfassung aufeinanderfolgender Join/Fork Nodes

Quelle: Eigene Darstellung

Wie auch Decision und Merge Nodes können Fork und Join graphisch zusammengefasst werden. Auch hier gilt, dass diese Notation aus Gründen der Übersichtlichkeit nur in Ausnahmefällen verwendet werden sollte (siehe Abbildung 2.14).

2.1.5 Strukturierte Aktivitäten

Die bisher vorgestellten Elemente von Aktivitäten können sehr gut genutzt werden, um Abläufe auf einer hohen Abstraktionsebene zu beschreiben. Für technische Abläufe, wie sie z.B. konkrete Implementierungen in einer Programmiersprache darstellen, sind sie zumeist weniger gut geeignet.

Um dennoch Abläufe modellieren zu können, die in einer textuellen Version leichter fassbar sind, bietet die UML die Structured Activities an. Die Einordnung der Structured Activities in die Paketstruktur des UML Metamodells ist in Abbildung 2.1 zu sehen. Sie stehen zum Teil neben den ablauforientierten Elementen der Basic Activities, können aber mit diesen kombiniert werden.

Zu den Structured Activities gehören Sequenzen, Bedingungen, Schleifen, Auffaltungsbereiche für die Arbeit mit Collections und Ausnahmebehandlung. Einige dieser mit Structured Activities explizit eingeführten Elemente lassen sich auch mit den bereits vorgestellten Elementen der UML beschreiben (siehe 2.1.2), allerdings gelten für Structured Activities einige andere Regeln, die im Folgenden erläutert werden sollen.

In der ursprünglichen Fassung der UML 2 existierte keine graphische Notation für die Structured Activities; mittlerweile hat sich jedoch so etwas wie ein Standard dafür ergeben.

Die Structured Activities stehen als Paket neben den Basic Activities und realisieren einige

Konzepte deutlich anders. Während in den normalen Aktivitäten auf den Flüssen ein Push herrscht (d.h. Token werden an die nachfolgenden Aktionen gesendet; sobald diese alle notwendigen Token besitzen, starten sie), wird bei StructuredActivities ein Pull-Prinzip verwendet. Eine Aktion startet erst dann, wenn Token von ihr benötigt werden.

Damit befinden sich die Structured Activities deutlich näher an den üblichen Konstrukten der Programmiersprachen, als die flussbasierten BasicActivities. In Abbildung 2.1 ist jedoch zu sehen, dass es Pakete gibt, die sowohl Konzepte aus den BasicActivities wie auch den Structured Activities nutzen.

Generell sind Structured Activity Nodes Knoten, die andere Knoten enthalten können. Andere Activity Nodes nämlich Aktionen, Kontroll- und Objektknoten können das nicht. Structured Activities sind beliebig verschachtelbar, allerdings wird dies in der graphischen Ansicht schnell unübersichtlich. Die Verschachtelung von Structured Activities entspricht im weitesten Sinne den Codeblöcken in Programmiersprachen.

Structured Activities sind, was die in ihnen enthaltenen Knoten angeht, weniger restriktiv als normale Aktivitäten. Aktionen innerhalb der Structured Activity können sowohl ein- als auch ausgehende Flüsse zu Knoten außerhalb der Structured Activity besitzen. Dies kann mit dem Konstrukt „go to“ in Programmiersprachen verglichen werden und sollte ebenso wie dieses, nicht zur Anwendung kommen.

Die Structured Activity Nodes können ebenso wie Aktionen mit Kontroll- und Objektflüssen verbunden werden. Es gelten für sie auch die gleichen Start- und Endbedingungen wie für normale Aktionen. Startet die Structured Activity, gelten für die enthaltenen Aktionen und Structured Actvities die gleichen Regeln als wären sie in einer normalen Aktivität enthalten, d.h. Knoten ohne eingehenden Kontrollfluss bekommen automatisch ein Kontrolltoken und beginnen mit ihrer Ausführung.

Structured Activities haben zusätzlich die Möglichkeit, Objekte für den Zugriff von Aktionen außerhalb zu sperren. Dies geschieht mit der Annotation `{ mustIsolate = true }`.

Sequenzen

Sequenzen (Sequence Nodes) stellen die einfachste Structured Activity dar, die modelliert werden kann. Sie definieren ebenso wie normale Aktivitäten eine Abfolge von Aktionen, die durch Kontroll- und Datenflüsse verbunden sind. Sequenzen können Parameter entgegennehmen und

zurückgeben. Allerdings muss dies explizit vom Anwender realisiert werden, da sie nicht automatisch die ausgehenden Pins der letzten Aktion mit ihren eigenen ausgehenden Pins verbinden.

Bedingungen

Die Bedingungen (Conditional Nodes) der Structured Activities sind an das in vielen Programmiersprachen verwendete Konstrukt `if-then-else` angelehnt.

Die Conditional Nodes bestehen aus Abschnitten mit je einem Test und einem Body. Im Test können eine Reihe von Aktionen definiert werden, die bestimmen, ob der dazugehörige Body zur Ausführung kommt.

Es können prinzipiell beliebig viele Abschnitte in einer Conditional Node enthalten sein. Zur Laufzeit wird dabei als erstes der Test des Abschnitts ausgewertet, der keinen Vorgänger hat, anschließend alle Nachfolger. Um einen Abschnitt für else zu modellieren, darf dieser Abschnitt keinen Nachfolger besitzen und der Test muss immer true liefern, was durch eine ValueSpecificationAction realisiert werden kann.

Sowohl Tests als auch Bodies können beliebige Aktionen und auch Strukturierte Aktivitäten enthalten. Die ersten Aktionen in beiden Bereichen dürfen aber nicht über eingehende Kontroll- und Datenflüsse verfügen. In der Praxis kann eine Schachtelung mehrere Structured Activities schnell zur sehr unübersichtlichen Diagrammen führen.

Die Auswertung der Tests geschieht über einen ausgehenden Pin des letzten Knotens des Tests der einen booleschen Wert zurückliefert und anhand dessen entschieden wird, ob der Body zur Ausführung kommt oder nicht.

Die Conditional Nodes entsprechen zwar im weitesten Sinne den Fallunterscheidungen in Programmiersprachen, haben jedoch mehr Möglichkeiten. So können innerhalb einer Conditional Node mehrere Tests den gleichen Vorgänger haben, was zu einer parallelen Auswertung der Tests führt. Es besteht somit die Möglichkeit, dass mehrere Tests erfolgreich ausgewertet werden, was allerdings nicht dazu führt, dass alle Bodies dieser Tests zur Ausführung kommen. Die UML spezifiziert nicht, welcher Body in so einem Fall ausgeführt wird (siehe [BOC04F]).

Die UML spezifiziert ebenfalls nicht, ob alle Tests immer oder nur so lange, bis ein Test erfolgreich ist, ausgeführt werden müssen. Da solche Charakteristika aber häufig in Implementierungen benötigt werden, bietet die UML die Möglichkeit für eine Conditional Node zwei Eigenschaften zu

setzen: `isAssured` und `isDeterminate`, die besagen, dass mindestens bzw. genau ein Test erfolgreich sein wird. Das Setzten dieser Eigenschaften befreit einen Anwender aber nicht davon, sicherzustellen, dass eine entsprechende Auswertung der Tests möglich ist. Die Eigenschaften dienen lediglich als Hilfestellung bei der Transformation in eine entsprechende Implementierung, um z.B. parallele Tests abzubrechen, sobald ein Test erfolgreich war.

Conditional Nodes können ausgehende Pins besitzen, die automatisch mit den Datentoken des ausgeführten Bodies belegt werden. Es liegt auch hier beim Anwender sicherzustellen, dass die ausgehenden Pins aller Bodies mit denen der Conditional Node kompatibel sind. Im Gegensatz zu den Sequenzen müssen hierfür keine Datenflüsse zu den ausgehenden Pins der Structured Activity modelliert werden.

Schleifen

Schleifen (Loop Nodes) bestehen aus drei Bereichen: Setup, Test und Body.

Der Setup-Bereich wird beim ersten Start der Loop Node ausgeführt. Tests werden, je nach Art der Schleife, vor oder nach jeder Iteration ausgeführt und der Body wird in jeder Iteration ausgeführt. Wann der Test ausgeführt wird, bestimmt die Eigenschaft `isTestedFirst` der Loop Node.

Test und Body können, wie auch bei Bedingungen, Aktionen und andere Structured Activities enthalten.

Schleifen können eingehende Pins haben, die innerhalb der Schleife konstante Variablen bereitstellen. Ebenso können Schleifen ausgehende Pins besitzen. Die Werte dieser Pins werden beim Beenden der Schleife aus den sogenannten Loop Variables gesetzt. Bei diesen handelt es sich um eine Art virtueller Pins, die bei jeder Iteration geändert werden können, da sie ihre Werte direkt aus den ausgehenden Pins des Bodies beziehen. Die Werte einer Loop Variable werden während einer Iteration nicht geändert, sondern erst, wenn diese abgeschlossen ist und neue Werte an den ausgehenden Pins des Bodies zur Verfügung stehen.

Auffaltungsbereiche

Die Expansion Region wird verwendet, um Collections sequentiell, parallel oder als Datenstrom zu verarbeiten. Sowohl die eingehenden als auch die ausgehenden Daten einer Expansion Region sind Collections. Diese werden durch vier direkt nebeneinander liegende Pins auf dem Rand der Structured Activity dargestellt. Die Pins nehmen eine Collection entgegen und stellen alle Elemente,

die in der Collection enthalten sind, einzeln über einen Datenfluss innerhalb der Structured Activity zur Verfügung. Ausgehende Pins fügen ankommende Elemente wieder zu einer Collection zusammen. Aus Sicht der Aktivität, zu der sie gehören, sind Expansion Regions Objektknoten.

Die Expansion Regions können genutzt werden, um Mappings für Collections zu modellieren. Ist eine Expansion Region als `<<iterative>` gekennzeichnet, wird ein und die selbe Region für jedes Element der Collection erneut ausgeführt. Falls sie als `<<parallel>>` gekennzeichnet ist, werden zur Laufzeit Kopien der Region für alle Elemente der Collection erstellt, die die Elemente dann verarbeiten. Im Modus `<<stream>>` werden alle Elemente direkt nacheinander durch die gleiche Kopie der Region verarbeitet.

Eingehende Collections können als `ordered` gekennzeichnet werden, was Einfluss auf die Reihenfolge der Verarbeitung hat, wenn die Region als iterativ oder datenstromverarbeitend gekennzeichnet ist. Außerdem kann eine Expansion Region sowohl mehrere eingehende wie auch ausgehende Collections verarbeiten. Dies kann z.B. dazu genutzt werden, Collections zusammenzuführen oder aufzuteilen. Bedingung dafür ist aber, dass eingehende Collections die gleiche Menge an Elementen enthalten, da in jeder Ausführung oder Kopie der Region ein Element aus jeder Collection verarbeitet wird.

Ausnahmebehandlung

Neben dem in 2.1.2 beschriebenen flussbasierten Weg zur Ausnahmebehandlung bieten die Structured Activities eine Alternative an, die an das try/catch-Konzept angelehnt ist, das in vielen Programmiersprachen realisiert ist.

Die Ausnahmebehandlung in Structured Activities wird durch einen Objektfluss aus der Structured Activity gekennzeichnet, der mit einer Zickzack-Linie annotiert ist. Wird innerhalb der Structured Activity oder in einem von einer CallAction innerhalb der Structured Activity aufgerufenen Verhalten eine RaisExceptionAction gestartet, werden sofort alle Token in der Structured Activity und eventuell weiteren synchron ausgeführten Verhalten gelöscht und die entsprechende Exception an den ausgehenden Objektfluss gegeben.

2.2 Neuerungen in der UML 2

Während in der ersten Version der UML Aktivitäten als spezielle Arten von Zustandsautomaten

beschrieben waren, brachte die UML 2 ein vollständig neues Aktivitätsmodell mit sich. Dieses neue Aktivitätenmodell ersetzte das bis UML 1.4 existierende semiformale Aktivitätenmodell.

Erste Arbeiten für eine maßgebliche Erweiterung des Aktivitätsmodells waren bereits in der Version 1.5 der UML zu sehen. Die UML 1.5 fand allerdings nie eine hohe Verbreitung, da zum Zeitpunkt ihres Erscheinens die UML 2 bereits in Arbeit war.

2.2.1 Erweiterungen in der UML 1.5

In der Verison 1.5 der UML wurde zum ersten Mal eine Unterscheidung zwischen Kontroll- und Datenflüssen eingeführt. Allerdings waren die Aktivitäten der UML 1.5 nach wie vor lediglich eine besondere Sicht auf Zustandsautomaten. Dennoch war die UML 1.5 damit ein erster Schritt von einer eher skizzenhaften Verwendung hin zu einer starken Formalisierung, die dann mit UML 2.0 vervollständigt wurde.

Die UML 1.5 führte auch einige spezielle Aktionen ein, die eine explizite semantische Bedeutung haben. Einige dieser Aktionen sind in der UML 2.0 nach wie vor enthalten. Alle diese Aktionen wurden in einem separaten Package im Metamodell eingeführt, eine Praxis die in UML 2 ebenfalls erhalten geblieben ist. Da die Aktivitäten der UML 1.5 noch auf Zustandsautomaten basieren, sind alle Aktionen, die sie mit sich bringt, Action States, die durch Rechtecke mit abgerundeten Seiten dargestellt werden. Die Relevanz der einzelnen Aktionen wird sehr unterschiedlich bewertet. Einige Hersteller von UML Modellierungswerkzeugen ignorieren einige der Aktionsklassen (im Enterprise Architect lassen sich z.B. standardmäßig nur CallOperationAction und CallBehaviorAction modellieren) ebenso wie Autoren verschiedener Bücher zum Thema UML. So wird in [RJB05] nur ein Bruchteil der verfügbaren Aktionsklassen beschrieben, in anderen Werken die Existenz unterschiedlicher Arten von Aktionen gänzlich verschwiegen, sofern sie nicht mit unterschiedlichen graphischen Notationen versehen sind.

Eine wesentliche Neuerung, die in UML 1.5 eingeführt wurde, ist die Möglichkeit, Verhalten direkt ohne Objekt aufzurufen. Dies ist die Grundlage für die Unterscheidung in CallOperationAction und CallBehaviorAction.

2.2.2 Änderungen von UML 1.5 zu UML 2.0

Die umfassendste Neuerung in der UML 2 ist sicherlich, dass Aktivitäten nicht länger ein Sonderfall von Zustandsautomaten sind, sondern vielmehr als eine Variante von Petrinetzen betrachtet werden. Auch wenn die Aktivtäten der UML 2 semantisch abwärtskompatibel zur UML 1.5 sind, wurden die

Metamodelle von Zustandsautomaten und Aktivitäten vollständig getrennt [RJB05]. Damit stellt die UML 2.0 einen radikalen Paradigmenwechsel dar, wie er vorher in der Entwicklung der UML eher unüblich war.

Im Übergang von UML 1.5 zu UML 2 sind einige Elemente vollständig entfernt worden, so z.B. die Subactivity States, die ohne Ableitung einer Aktivität von einem Zustandsautomaten keinen Zweck erfüllen. Ersetzt wurde beispielsweise die JumpAction durch die Structured Activity zur Ausnahmebehandlung.

Ein weiterer wesentlicher Unterschied zur UML 1 besteht darin, dass Aktionen nicht mehr nur durch ein eingehendes Token gestartet werden, wie dies bei den Action States der Fall war. Die eingehenden Flüsse haben nicht den Charakter einer Transition in einem Zustandsautomat (siehe 2.1.2). Ähnlich ist die Situation bei ausgehenden Flüssen: während beim Verlassen eines Zustands nur eine Transition ausgeführt werden kann, werden in den Aktivitäten der UML 2 alle ausgehenden Flüsse mit Token versehen. Ebenfalls nicht realisierbar war die mehrfache Hintereinanderausführung einer Transition zwischen zwei Action States. Diese ist erst durch die Verwendung der Token in der UML 2 möglich geworden und erlaubt z.B. die einfache Modellierung von Serialisierung (siehe 2.1.4) und mehreren Token in Datenflüssen (siehe 2.1.3).

Damit einher geht auch, dass die Flüsse in Aktivitäten der UML 2 parallel ausgeführt werden können, was bei Aktivitäten der UML 1 so nicht möglich war. Dort mussten Fork und Join jeweils paarweise auftreten und der Anwender war gezwungen die Synchronisation zwischen Aktionen sicherzustellen. Auch die Eingabe- und Ausgabe-Pins, die in der UML 2 eingeführt wurden, erlauben eine Form der Nebenläufigkeit, die so in der UML 1 mit Aktivitäten nicht realisiert werden konnte.

Statt Objektknoten wurden in der UML 1.5 die ObjectFlowStates verwendet. Diese hatten gänzlich andere Charakteristiken bezüglich der Speicherung von Daten als die Objektknoten der UML 2, nämlich die folgenden:

- Passiv: Daten in ObjectFlowStates führen nicht zum Start von Aktionen, diese holen sich die Daten vielmehr nach Bedarf
- Nicht-Reduzierend: die Verwendung von Daten aus einem ObjectFlowStates entfernt die Daten nicht von dort
- Persistent: wird die Aktivität beendet, verbleiben Daten in den ObjectFlowStates

Die UML 2 ändert alle drei Charakteristiken für Objektknoten wie in 2.1.3 beschrieben. Um Zugriffe auf persistente Objekte modellieren zu können, stellt die UML 2 explizite Aktionen bereit.

In Details wurden noch etliche weitere Änderungen auf dem Weg von UML 1 zur UML 2 vorgenommen, deren Auflistung allerdings nicht Bestandteil dieser Arbeit sein soll.

2.3 Aktivitäten und Zustandsautomaten

Wie bereits erwähnt, waren die ursprüngliche Realisierung von Aktivitäten eine Ableitung der Zustandsautomaten, die in UML 2 durch ein vollständig eigenständiges Metamodell ersetzt wurde.

Dennoch existieren nach wie vor Zusammenhänge zwischen beiden Verhaltensbeschreibungen. Zunächst wäre da die Tatsache, dass es sich bei beiden um parametrisierbares Verhalten handelt. D.h. beide Verhaltensmodelle erlauben die Beschreibung von Verhalten, das von bestimmten Eingangs- und Ausgangsdaten abhängt.

Zustandsautomaten sind dabei in [RJB05] als Sequenzen von Zuständen eines Objektes definiert. Diese Zustände werden von dem Objekt während seiner Existenz angenommen, wobei die Übergänge zwischen den Zuständen durch Ereignisse ausgelöst werden. In der UML können Zustandautomaten, wie andere Verhaltensbeschreibungen auch, an Klassen, Anwendungsfälle und Methoden gebunden werden.

Im Kontext von Zustandsautomaten ist häufig von Actions die Rede, die allerdings nicht mit den Aktionen identisch sind, die in Aktivitäten verwendet werden. Die Actions der Zustandsautomaten sind aus Sicht des jeweiligen Zustandsautomaten zwar ebenfalls atomar, führen aber in jedem Fall ein Verhalten aus. Somit entsprechen sie im weitesten Sinne dem, was die CallActions in Aktivitäten tun, sind im Metamodell der UML aber vollständig unabhängig realisiert. Vielmehr stellen im Metamodell der UML sowohl die als entry-, exit- und do-Actions bezeichneten Assoziationen der Zustände als auch die effect-Action von Transitionen zwischen Zuständen Verknüpfungen zu Verhalten her (siehe [OMG09], Seiten 552 bzw. 572).

Ob diese Realisierung in jedem Fall sinnvoll und logisch ist, wäre zu diskutieren, insbesondere unter Berücksichtigung der Tatsache, dass Verhalten in der UML eigentlich als nicht-atomar, Aktionen in Aktivitäten explizit als atomar definiert sind. Diese Diskussion soll aber nicht im Rahmen dieser Arbeit geführt werden.

In jedem Fall haben aber sowohl Zustandsautomaten als auch Aktivitäten die Möglichkeit, anderes Verhalten anzustoßen.

Ein weiterer direkter Zusammenhang besteht in der Möglichkeit von Objektknoten, nicht nur den Typ der Objekte zu spezifizieren, für die sie Token speichern können, sondern auch zu bestimmen, in welchem Zustand sich diese Objekte befinden müssen. Dazu ist es notwendig, die Klassen der Objekte mit einer Verhaltensbeschreibung in Form eines Zustandsautomaten zu modellieren. Die UML überlässt es auch hier dem Anwender sicherzustellen, dass diese Bedingungen zur Laufzeit erfüllt werden.

Aktivitäten haben außerdem die Möglichkeit, Zustandsautomaten direkt durch SendSignalActions zu triggern.

Der gemeinsame Ursprung von Aktivitäten und Zustandsautomaten in der UML wird an den identischen graphischen Repräsentationen einiger Elemente nach wie vor deutlich. So werden z.B. für Initial- und FinalNodes bei Aktivitäten die gleichen Symbole verwendet, wie für Inital- und FinalStates in Zustandsautomaten, auch wenn die Semantik dieser Elemente nichts mehr gemein hat.

3 Integration von Aktivitäten in GeneSEZ

Das GeneSEZ Generatorframework stellt eine auf dem Eclipse Modeling Framework und openArchitectureWare basierende Plattform für modellgetriebene Softwareentwicklung dar.

Es ist unterteilt in ein Metamodell, welches hinsichtlich einer einfachen Generierung von Quellcode optimiert ist, und in verschiedene Plattformen für die Generierung von Codefragmenten in unterschiedlichen Programmiersprachen.

Die grundlegende Arbeitsweise von GeneSEZ besteht darin, im Rahmen eines Workflows Softwaremodelle zunächst in das GeneSEZ Metamodell zu transformieren und aus dem entstandenen Modell den Quellcode oder andere textuelle Artefakte mit Hilfe von Templates zu erzeugen.

GeneSEZ kann dabei momentan lediglich Modelle, die als EMF UML 2 vorliegen, verarbeiten. Die Verwendung anderer Modellierungsprachen wie z.B. Domänenspezifische Sprachen ist zwar prinzipiell möglich, eine Unterstützung wurde aber bisher nicht realisiert. Auch für die Implementierung des Adapters für den Enterprise Architect im Rahmen dieser Arbeit wurde der Umweg über EMF UML 2 gewählt.

Das Metamodell von GeneSEZ unterstützt bislang lediglich die Generierung von Infrastrukturcode aus Klassenmodellen und muss für die Integration von Aktivitäten erweitert werden. Parallel zu dieser Arbeit wurden auch Zustandsautomaten in das Metamodell integriert, was einige Synergieeffekte mit sich brachte, auf die im Weiteren hingewiesen wird.

Da das Ziel der Arbeit eine möglichst einfache Unterstützung von Aktivitäten ist, basiert das GeneSEZ Metamodell nicht wie das der UML auf Petri-Netzen und geht auch nicht von Token aus, die den Aktvitätengraphen durchlaufen. Die Aktivitäten im GeneSEZ Metamodell unterliegen somit nicht den hohen formalen Anforderungen, wie sie für die Ausführung in einer virtuellen Maschine notwendig sind.

Das GeneSEZ Metamodell muss es lediglich erlauben, die Aktivitätenmodelle der UML sinnvoll in

textuelle Artefakte zu transformieren. Bei diesen Artefakten soll es sich um Quellcode in konkreten Programmiersprachen und um Ablaufanweisungen für Softwaretests handeln.

Ein wesentlicher Aspekt der Entwicklung von GeneSEZ ist die Reduzierung des Metamodells auf absolut notwendige Elemente, um die Modell-zu-Text-Transformation, die ganz am Ende eines jeden Workflows steht, so effizient wie möglich zu gestalten. Dieser Aspekt wird bei der Erweiterung um Aktivitäten berücksichtigt. Letztendlich werden nur minimale Änderungen am GeneSEZ Metamodell vorgenommen, die aber ausreichend sind, um die in 2 vorgestellten Elemente der UML sinnvoll weiterverarbeiten zu können.

Eine vollständige Umsetzung von Aktivitäten in Programmiersprachen, also eine Unterstützung für graphisches Programmieren ist mit dem derzeitigen Metamodell zwar denkbar, allerdings noch nicht praktikabel.

3.1 Änderungen am Kern des Metamodells

Ziel der Arbeit war es, eine möglichst einfache Realisierung der Aktivitäten im Metamodell zu erreichen, die dennoch mächtig genug sein sollte, um sie sinnvoll für die Generierung im Rahmen von GeneSEZ einsetzen zu können.

Um dieses Ziel zu erreichen, waren erfreulich wenige Änderungen am Kern des bisher existierenden Metamodells notwendig. Es mussten lediglich drei neue Metamodellklassen eingeführt werden, die im Folgenden erläutert werden: MBehavior, MDefinitionContext und MUseCase.

3.1.1 MBehavior

Die Metamodellklasse MBehavior soll als abstrakte Klasse die Grundlage von Verhaltensmodellen innerhalb des GeneSEZ Metamodells bilden. Die Idee, die Metamodelle der Verhaltensbeschreibungen auf einer gemeinsamen Basis zu realisieren, wurde von der UML übernommen, wo Aktivitäten, Interaktionen und Zustandsautomaten ebenfalls von einer entsprechenden abstrakten Klasse abgeleitet sind.

Der Grund dafür ist einleuchtend: Auf diese Weise wird dadurch ermöglicht, Verhaltensbeschreibungen für Klassen und Methoden in der für den Anwender jeweils praktikableren Form zu modellieren. Zusätzlich ist auch eine Transformation zwischen den

unterschiedlichen Arten von Verhalten auf Modellebene denkbar, z.B. können aus einem Zustandsautomaten Aktivitäten generiert und transparent verwendet werden. Dies kann u.A. für die Generierung von Testfällen aus Zustandsautomaten, wie sie in 4.1.3 beschrieben ist, eingesetzt werden.

Die Klasse MBehavior hat keine speziellen Attribute, erbt aber von der Klasse MDefinitionContext, die im Rahmen der Integration von Aktivitäten und Zustandsautomaten ebenfalls neu ins Modell aufgenommen wurde.

3.1.2 MDefinitionContext

Verhalten muss im Metamodell der UML nicht zwangsläufig an Klassen oder Methoden geknüpft sein. Aktivitäten in UML Modellen können beispielsweise in Paketen enthalten sein und von anderen Aktivitäten referenziert werden. Diese Aktivitäten können dann zwar nicht direkt ausgeführt werden, aber durchaus als eine Bibliothek für Verhalten dienen.

Um diese Möglichkeit abzubilden, wurde mit den Zustandsautomaten die Klasse MDefinitionContext eingeführt. Von dieser erben nun sowohl MPackage als auch MClassifier. Während MClassifier weiterhin direkt von MElement und MType erbt, wurde die Generalisierung von MPackage zu MElement entfernt.

Die Beziehung owningPackage von MClassifier zu MPackage wurde durch die Beziehung owningContext von MClassifier zu MDefinitionContext ersetzt.

Die Einführung von MDefinitionContext in dieser Form bietet den Vorteil, dass Verhalten sowohl in MPackages enthalten sein kann, als auch direkt in einem MClassifier. Gleichzeitig ergibt diese Änderung des Metamodells die Möglichkeit, innere Klassen zu unterstützen, was vorher für MClassifier nicht möglich war.

Da MBehavior von MDefinitionContext erbt, gleichzeitig aber auch in einem MDefinitionContext enthalten sein kann, wird die die Rekursion von Verhaltensmodellen und damit eine Verschachtelung von Aktivitäten möglich, wie sie auch in der UML modelliert werden kann.

Ein MDefinitionContext kann durchaus mehrere Verhaltensbeschreibungen direkt enthalten. Dies ist sinnvoll, wenn während einer Modelltransformation zusätzliche Verhalten für einen MDefinitionContext generiert werden, wie es z.B. bei der Generierung von Testfällen aus Zustandsautomaten der Fall sein kann.

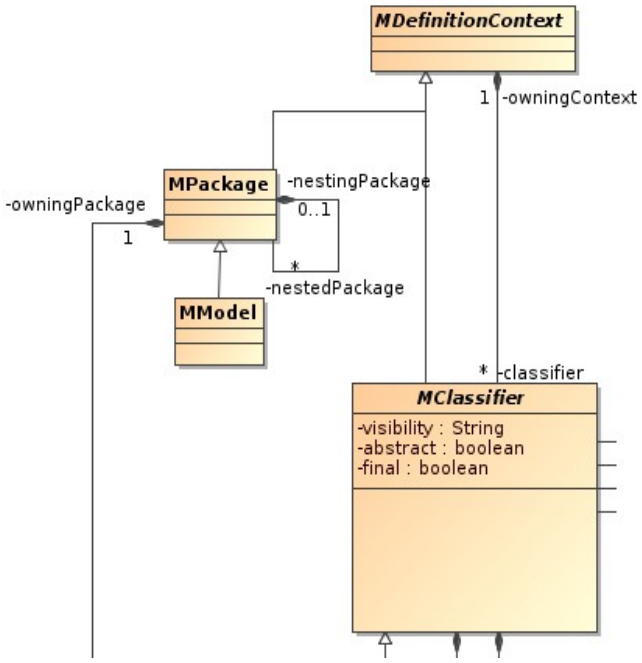

Abbildung 3.1: GeneSEZ Core Version 1.7

Quelle: GeneSEZ Metamodell
(Ausschnitt nach Anpassung)

In Abbildung 3.1 ist die Position von MDefinitionContext im neuen GeneSEZ Metamodell zu sehen; die alte Version zum Vergleich in Abbildung 3.2.

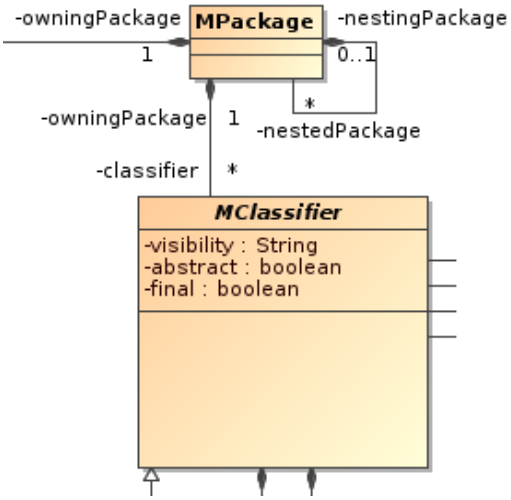

Abbildung 3.2: GeneSEZ Core Version 1.6

Quelle: GeneSEZ Metamodell
(Ausschnitt vor Anpassung)

3.1.3 MUseCase

Wie bereits in 2.1 erläutert, können Aktivitäten auf sehr unterschiedlichen Abstraktionsebenen für die Modellierung eingesetzt werden. Eine Bindung an Klassen und Operationen ist durch die Beziehung zwischen MBehavior und MDefinitionContext gegeben. Um die Aktivitäten aber auch für abstraktere Beschreibungen nämlich für Abläufe aus Anwendersicht verwenden zu können, wurde die Klasse MUseCase eingeführt.

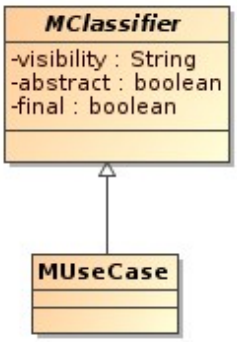

Abbildung 3.3: MUseCase im GeneSEZ Metamodell

Quelle: GeneSEZ Metamodell, eigene Darstellung

Diese Klasse repräsentiert Anwendungsfälle, wie sie auch in UML Modellen definiert werden können. MUseCase ist dabei eine spezielle Form von MClassifier, die in GeneSEZ Metamodell lediglich zur Strukturierung verwendet wird. Die Positionierung von MUseCase im GeneSEZ Metamodell ist in Abbildung 3.3 zu sehen.

3.2 Spezifische Erweiterungen des Metamodells für Aktivitäten

Da an die Aktivitäten in GeneSEZ weniger strikte Anforderungen gestellt werden als dies bei den Aktivitäten in der UML der Fall ist, kann die Realisierung im GeneSEZ Metamodell mit deutlich weniger Elementen auskommen, als die UML. Insgesamt sind nur fünf spezifische Klassen für Aktivitäten notwendig, um Aktivitätenmodelle aus der UML in GeneSEZ abbilden zu können.

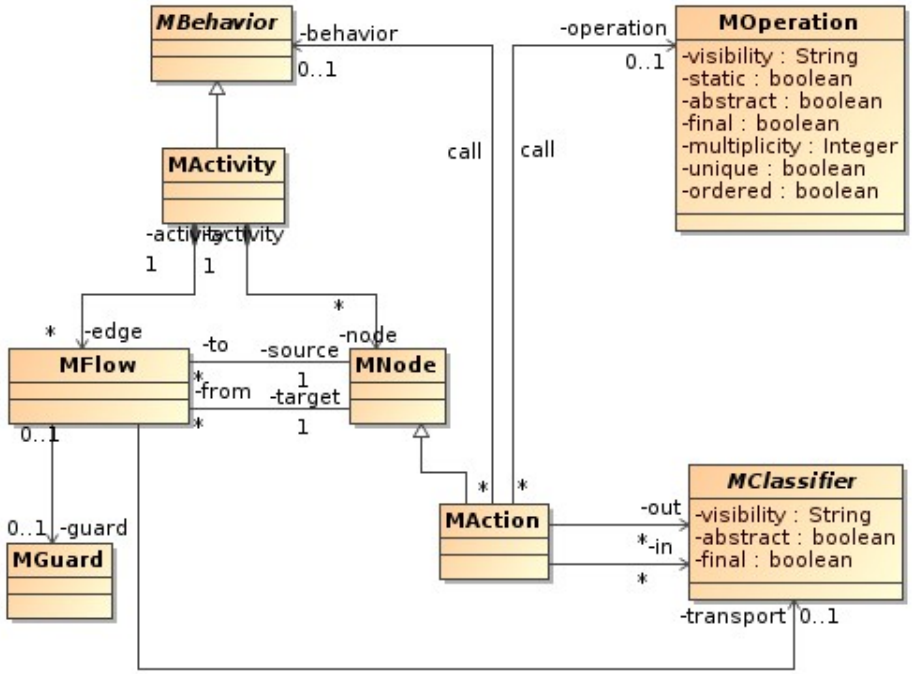

Abbildung 3.4: Metamodell der Aktivitäten in GeneSEZ

Quelle: GeneSEZ Metamodell, eigene Darstellung

Damit geht notwendigerweise eine Reduzierung der Semantik im Metamodell einher. Diese ist allerdings durchaus vertretbar, da nicht der Anspruch erhoben wird, ein Metamodell zu schaffen, das sowohl für die Geschäftsprozessmodellierung als auch zum graphischen Programmieren nutzbar sein soll. Vielmehr können solche Anwendungsfälle in GeneSEZ durch die Nutzung von Profilen und Stereotypen abgebildet werden.

3.2.1 MActivity

Neben MStateMachine stellt MActivity eine Basisklasse für Verhalten im GeneSEZ Metamodell dar. Die Klasse ist von MBehavior abgeleitet und erbt daher auch die Eigenschaften von MDefinitionContext und MElement. Instanzen von MActivity besitzen einen Namen (Attribut name aus MElement) und einen eindeutigen Identifier (Attribut xmiGuid aus MElement). Sie sind in einem Kontext (Assoziation owningContext von MBehavior) enthalten und können gleichzeitig Kontext für andere Verhalten sein (Assoziation ownedBehavior von MDefinitionContext).

Die Positionierung von MActivity im Rahmen des GeneSEZ Metamodells ist in Abbildung 3.5 zu sehen.

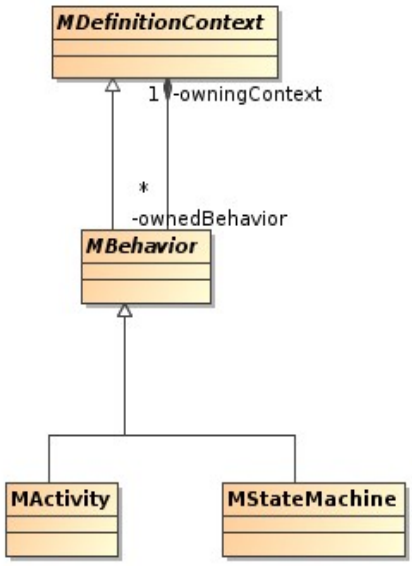

Abbildung 3.5: MBehavior im GeneSEZ Metamodell

Quelle: GeneSEZ Metamodell

Die Fähigkeit des MDefinitionContext, auch Elemente von Typ MClassifier aufzunehmen, wird zwar ebenfalls geerbt, sollte aber im Zusammenhang mit Aktivitäten nicht zum Einsatz kommen, da die Definition von Datentypen in einer Aktivität nicht sinnvoll ist.

In der Aktivität sind Objekte der Typen MFlow und MNode enthalten, die beide im Folgenden erläutert werden.

3.2.2 MFlow

Die Metamodellklasse MFlow soll die aus den Aktivitäten der UML 2 bekannten Objekt- und Kontrollflüsse abbilden. Da in der UML 2 Datenflüsse implizit Kontrollflüsse sind (siehe 2.1.2), wird auf eine Unterscheidung zwischen beiden im GeneSEZ Metamodell verzichtet. Statt zweier Metamodellklassen wird nur eine eingeführt, die dafür die optionale Assoziation `transport` zur Metamodellklasse MClassifier besitzt. Damit wird ermöglicht, einem Fluss zuzuordnen, welchen Datentyp er transportieren kann.

Jeder MFlow kann mit einen MGuard verknüpft sein, um seine Ausführung zu prüfen. Weiterhin besitzt jeder MFlow eine Beziehung zu (`to`) und von (`from`) einer MNode.

3.2.3 MGuard

Bei der im Aktivitätenteil des GeneSEZ Metamodells verwendeten Klasse MGuard handelt es sich

um die gleiche Klasse die auch in den Zustandsautomaten verwendet wird. Aufgabe im Kontext der Aktivitäten ist es, die Ausführung eines Flusses nur unter bestimmten Bedingungen zu erlauben.

Die Klasse ist in den Aktivitäten momentan nur der Vollständigkeit halber enthalten und wird noch nicht aktiv ausgewertet. Perspektivisch ist die Integration einer Unterstützung von OCL-Ausdrücken in GeneSEZ denkbar, durch die dann auch MGuards eine größere Bedeutung erlangen würden.

Da MGuard sowohl in Zustandsautomaten als auch in Aktivitäten Anwendung findet, muss das bereits existierende Metamodell der Zustandsautomaten geändert werden. Die Multiplizität der Assoziation zwischen MGuard und MTransition wird von `1..*` auf `0..*` angepasst.

3.2.4 MNode

Mit der Metamodellklasse MNode sollen sämtliche Knoten, die keine Aktionen sind, abgebildet werden. Das schließt neben den Objektknoten sämtliche Arten von Kontrollknoten ein.

Da das GeneSEZ Metamodell nicht den gleichen Anspruch der Allgemeingültigkeit erhebt wie das der UML, sondern vielmehr ein pragmatisches Minimalmodell für Entwickler darstellt, kann eine solche Zusammenfassung durchgeführt werden. Die Modellelemente des Typs MNode müssen natürlich mit einer Charakteristik ausgestattet werden, die eine Unterscheidung ermöglicht. Diese kann jedoch ausreichend durch ein entsprechendes Attribut abgebildet werden, das bei der Modell-zu-Text-Transformation ausgewertet wird.

Relevant für die Instanzen von MNode sind vor allem die Flüsse, die ein- bzw. ausgehen (Assoziationen `source` und `target` im Metamodell).

Herkömmliche Objektknoten werden im GeneSEZ Metamodell nicht verwendet, da diese in der Form, wie sie in der UML enthalten sind, lediglich benötigt werden, um die Aktivitäten in der virtuellen Maschine laufen lassen zu können und dort Datentoken zu puffern. Der Typ der Daten, die über einen Fluss transportiert werden können, ist durch die bereits angesprochene Assoziation `transport` zwischen MFlow und MClassifier festgelegt.

3.2.5 MAction

MAction soll im GeneSEZ Metamodell verwendet werden, um atomare Aktionen abzubilden. Da eine Aktion alle Charakteristiken eines ActivityNode aufweist, wurde MAction von MNode abgeleitet. Damit besitzen MActions ein- und ausgehende Flüsse und gehören zu einer Aktvität.

Zusätzlich zu den Assoziationen die von MNode geerbt werden, erhalten Elemente vom Typ MAction noch Assoziationen zu MBehavior, MOperation und MClassifier. Die beiden Assoziationen zu MBehavior und MOperation erlauben es, die Semantik von CallActions abzubilden, während zu MClassifier zwei Assoziationen bestehen, um ein- und ausgehende Datentypen festlegen zu können.

Die Unterstützung einer Menge expliziter Aktionen in GeneSEZ ist nicht erstrebenswert, denn damit würde die Philosophie des minimalen Metamodells aufgegeben. Dazu kommt, dass nicht alle der in der UML enthaltenen Aktionen sinnvoll in jede Programmiersprache transformiert werden können aber gleichzeitig sehr oft Aktionen modelliert werden, die nicht explizit in der UML enthalten sind. Daher wird die explizite Semantik einer Aktion in GeneSEZ durch einen Stereotyp festgelegt. Nur für CallActions ist dies nicht notwendig, da diese durch ihre Assoziationen zu MBehavior oder MOperation festgelegt sind.

Durch den konsequenten Einsatz von Stereotypen zur Spezialisierung von Aktionen stellt man sowohl dem Modellierer als auch dem Entwickler der Modell-zu-Text-Transformation einen einheitlichen Mechanismus mit vielen Möglichkeiten zur individuellen Anpassung zur Verfügung. Würden Teile oder die gesamte Menge der verwendbaren Aktionen durch das Metamodell vorgegeben, müssten entweder immer verschiedene Varianten der Spezialisierung berücksichtigt werden oder aber es stünden nur eine begrenzte Menge an Aktionen zur Verfügung.

3.3 Änderungen der uml2genesez-Transformation

Wie zu Beginn dieses Kapitels genannt, werden zur Zeit ausschließlich Modelle, die als EMF UML 2 vorliegen, für die Generierung mit GeneSEZ verwendet. Für diese Modelle existieren ein Adapter und mehrere Workflow-Komponenten, die die Modell-zu-Modell-Transformation für UML 2 Modelle durchführen.

Diese Transformation besteht im Wesentlichen aus einigen Xtend-Scripten, einer Java-Klasse und einem Check-Script.

Das Check-Script prüft, ob das Eingabe-Modell Namens- und Typkonventionen einhält. Die Java-Klasse stellt einige Hilfmethoden zur Verfügung, u.a. um auf die Global Unique Identifier (GUID) von Modellelementen zuzugreifen.

Die eigentliche Transformationsarbeit wird durch das Xtend-Script `uml2genesez` ausgeführt.

Dieses enthält die vollständige Transformation von UML2-Klassenmodellen in das GeneSEZ-Metamodell. Im Rahmen der Erweiterung um Zustandsautomaten wurde zusätzlich das Transformations-Script `uml2genesezdynamicmodel` eingeführt. Die Transformation von Aktivitäten ist im Rahmen dieser Arbeit ebenfalls in diesem Script realisiert.

3.3.1 Änderungen an uml2genesez

Wie bereits zu Beginn dieses Kapitles erläutert, sind drei neue Klassen ins GeneSEZ Metamodell eingeführt worden. Diese müssen natürlich bei der Transformation von UML Modellen berücksichtigt werden. Die Klasse MDefinitionContext ist allerdings abstrakt, es wird also keine Elemente dieses Typs geben, sondern nur Elemente von Typen, die von MDefinitionContext erben. Auf MDefinitionContext muss also in der Transformation keine Rücksicht genommen werden.

Anders verhält es sich für MUseCase und MBehavior. MUseCase wird zur Strukturierung von Verhalten verwendet. Verhalten kann prinzipiell auch in Paketen enthalten sein, um beispielsweise Funktionsbibliotheken abzubilden. Für die Anwendung des modellgetriebenen Testens ist eine Struktureinheit wie sie Anwendungsfälle darstellen, aber sinnvoll.

Da MUseCase eine Spezialisierung von MClassifier ist, ist eine Transformation des UML Usecase in MUseCase als `makeClassifier` in uml2genesez implementiert.

Durch die Einführung von MDefinitionContext erhalten sämtliche MClassifier die Assoziation ownedBehavior, durch die sie mit Verhaltensbeschreibungen verknüpft werden können.

Die Transformation des Verhaltens wird in uml2genesezdynamicmodel realisiert, allerdings ist die Verknüpfung in uml2genesez enthalten. Für MClass findet diese Verknüpfung in `transform(Classifier, MPackage)` bzw. `transform(Classifier, MModel)` statt, für MUseCase in `makeClassifier(UseCase)`. Verhalten, das direkt in Paketen enthalten ist, wird in `makePackage(Package)` mit dem MPackage verknüpft.

Weitere Änderungen an uml2genesez sind rein technischer Natur, wie die Deklaration von `de::genesez::adapter::uml2::uml2genesezdynamicmodel` als Extension und das Entfernen von `private` bei Funktionen, die auch in der Transformation der dynamischen Modelle sichtbar sein sollen. Dazu gehört insbesondere die grundlegende Transformation von MElement, die den GUID für alle Modellelemente setzt, aber auch die Transformationen von Stereotypen, Tagged Values und Kommentaren.

Insgesamt müssen an der Transformation, ebenso wie auch am Kern des Metamodells, wenige Änderungen durchgeführt werden. Damit kann die Einfachheit des GeneSEZ Metamodells sowie der Transformation beibehalten werden.

3.3.2 Transformation von Aktivitäten mit uml2genesezdynamicmodel

Die eigentliche Transformation der Verhaltensmodelle ist nicht im Script uml2genesez enthalten. Der Grund dafür ist, dass die Transformation von Verhaltensmodellen, insbesondere der Aktivitäten sehr viel Code benötigt und sehr schnell unübersichtlich würde. Da bereits die Transformation von Zustandsautomaten in das Script uml2genesezdynamicmodel ausgelagert wurde, wird auch die Transformation von Aktivitäten in diesem Script realisiert.

Sowohl Zustandsautomaten als auch Aktivitäten in der UML ebenso wie im GeneSEZ Metamodell sind Spezialisierungen von Verhalten. Daher kann für die Definitionsregeln in Xtend sehr gut Polymorphismus angewendet werden. Für eine MClass, einen MUseCase oder auch einem MPackage ist es unbedeutend, welche Ausprägung das mit `ownedBehavior` verknüpfte Verhalten besitzt. Während der Transformation dieser Elemente wird immer nur `transformBehavior` aufgerufen. Diese Methode ist dann der Einstieg in `uml2genesezdynamicmodel`. Die Methode existiert in fünf verschiedenen Ausprägungen. Zunächst gibt es eine Implementierung für die Transformation von Zustandsautomaten, wobei für diese davon ausgegangen wird, dass sie immer zu einer Instanz von MClassifier gehören. Bei Aktivitäten kann davon nicht ausgegangen werden, denn sie können zu Instanzen von MClassifier, MPackage oder auch von MActivity gehören.

Da sich die Transformation in jedem dieser Fälle unterscheidet, gibt es insgesamt drei unterschiedliche Methoden für ihre Realisierung. Für Verhalten, das zwar in einem MDefinitionContext enthalten, aber nicht sinnvoll positioniert ist, existiert eine fünfte Implementierung von `transformBehavior`, die eine Fehlermeldung ausgibt.

Den sieben Arten von Kontrollknoten entspricht jeweils eine Transformation, die diese Kontrollknoten in Elemente des Typs MNode überführt und mit einem entsprechenden Stereotypen annotiert. Objektknoten werden nur dann transformiert, wenn sie als Data Store ausgewiesen sind. Da im GeneSEZ Metamodell keine Datentoken verwendet und Elemente des Typs MAction direkt mit ein- und ausgehenden Objekten verknüpft werden, sind die einfachen Objektknoten und damit auch die Pins überflüssig. Sie existieren im GeneSEZ Metamodell implizit, für eine weitere Verarbeitung sind die aber nicht nötig. Selbst wenn für eine Zielplattform ein Konzept benötigt

wird, wie es die Objektknoten darstellen, sind die Informationen durch die Assoziationen `in` und `out` zwischen MAction und MClassifier ausreichend, um diese zu generieren. Data Stores hingegen werden in Elemente des Typs MNode transformiert und wie die Kontrollknoten mit einem Stereotyp versehen. Der Grund für die besondere Behandlung von Data Stores liegt darin, dass diese Token nicht nur zwischenpuffern. Dieser besondere Charakter eines Data Store kann nicht implizit aus anderen Assoziationen gewonnen werden.

Alle Aktionen des UML-Modells werden in Elemente des Typs MAction transformiert. Auch hier werden Stereotypen zur Spezialisierung eingesetzt, was dem Anwender Möglichkeiten zur Definition von Aktionen über den Umfang der UML hinaus bietet. Dies ist bei der Definition domänenspezifischer Aktionen sehr hilfreich, denn diese können eingeführt werden, ohne das Metamodell des Generators zu ändern.

Als Sonderfälle werden lediglich die CallActions behandelt. Diese benötigen keinen expliziten Stereotyp, weil sie durch eine Assoziation mit MBehavior oder MOperation als solche identifizierbar sind.

Sobald die Transformation der Knoten abgeschlossen ist, werden die Flüsse der Aktivität transformiert. Sämtliche Flüsse werden in Elemente des Typs MFlow transformiert, da im GeneSEZ Metamodell wie bereits beschrieben nicht zwischen Kontroll- und Datenfluss unterschieden werden soll. Im ersten Schritt werden die Datenflüsse transformiert und mit den entsprechenden Elementen von MClassifier verknüpft, die sie transportieren sollen. Dies ist über die Assoziation `transport` abgebildet. Kontrollflüsse werden nach den Datenflüssen transformiert. Für beide Flüsse gilt, dass zunächst ihre beiden Enden anhand der GUID gesetzt werden und danach eventuelle Elemente von Typ MGuard.

Deutlich aufwändiger als die Transformation einfacher Aktivitäten und deren Aktionen und Flüssen stellt sich die Transformation von Structured Activities dar. Diese Transformation ist in der aktuellen Version von `uml2genesezdynamicmodel` nicht enthalten, da im Kontext dieser Arbeit keine sinnvolle Verwendung für Structured Activities gegeben war.

Für eine direkte Unterstützung von Structured Activities müsste das GeneSEZ Metamodell sehr stark erweitert werden, was der Philosophie der Einfachheit entgegen läuft. Es existieren zwei Möglichkeiten Structured Activities sinnvoll in das einfache Aktivitätenmodell von GeneSEZ zu integrieren, die beide im Ausblick am Ende der Arbeit erwähnt werden.

4 Verwendung von Aktivitäten zur Testgenerierung

4.1 Kurzüberblick Softwaretests

Die Anwendung von Tests als Mittel zur Sicherung der Qualität von Softwaresystemen hat in den letzten Jahren deutlich zugenommen. Grundsätzlich können Tests aber nur indirekt Einfluss auf die Qualität einer Software nehmen. Die Ursache dafür liegt in einer Erkenntnis begründet, die Dijkstra treffend formuliert hat: Man kann durch Testen generell nur die Anwesenheit von Fehlern beweisen, aber niemals ihre Abwesenheit.

In der Praxis wird von Testmanagern aber die Aussage erwartet, dass eine Software fehlerfrei läuft. Eben diese Aussage kann aber so eigentlich nicht getroffen, sondern bestenfalls festgestellt werden, dass zu einem Termin keine neuen Fehler mehr gefunden werden konnten - was in der Praxis aber sogar ein Zeichen für schlechte Testqualität sein kann. Um den Wert der zweiten Aussage trotzdem weit an den der (gewünschten) ersten Aussage anzunähern, ist es einleuchtend, dass möglichst viele Tests in möglichst kurzer Zeit erstellt, durchgeführt und ausgewertet werden müssen.

Die Durchführung von Softwaretests ist oftmals in einem großen Maße automatisierbar und kann an Software delegiert werden, die die zu testende Software auf verschiedenen technischen Ebenen unter vielerlei Gesichtspunkten prüft. Auch die Auswertung kann zu einem großen Teil automatisiert durchgeführt werden, solange es um Auswertungen anhand von Metriken und Protokollierung der Fehler geht. Die Suche nach Fehlern und ihren Ursachen ist nach wie vor Handarbeit und bleibt den Entwicklern überlassen.

Ebenso ist das Erstellen von Testfällen momentan fast ausschließlich Handarbeit. In der Praxis werden Testfälle oft prosaisch beschrieben, manchmal auch mit Hilfe proprietärer Testwerkzeuge. Die Testfälle werden dann – insbesondere auf den weniger technischen Ebenen, wie GUIs, die näher am Anwender sind – von Menschen manuell durchgeführt, sobald die Software vorliegt. In einigen Fällen werden diese manuellen Tests aufgezeichnet, um sie bei einer neuen Version

wiederholen zu können. Allerdings haben diese aufgezeichneten Regressionstests den Nachteil, dass sie z.B. bei Tests unter Verwendung des GUI davon ausgehen müssen, dass alle Elemente des GUI identisch geblieben sind. Diese Bedingung kann in der Praxis oft nur sehr eingeschränkt erfüllt werden, was zu hohen Aufwänden bei der Wartung der Tests führt.

4.1.1 Klassifizierung nach V-Modell

Eine sinnvolle Möglichkeit der Klassifizierung ist die durch das V-Modell vorgegebene 4-Ebenen-Einteilung in Komponenten-, Integrations-, System- und Akzeptanztests. Die Komponententests entsprechen dabei den allgemein als Unit-Tests bezeichneten Tests, die zumeist direkt vom Entwickler mit Frameworks wie JUnit [JUNIT], NUnit [NUNIT] oder verwandten Werkzeugen implementiert werden. Diese Tests beziehen sich auf Klassen und deren Zusammenarbeit (Klassentests) bzw. die technischen Komponenten der Software (Modultests).

Während die Komponententests isoliert voneinander ablaufen, wird mit Integrationstests versucht, festzustellen, ob die Komponenten erwartungsgemäß zusammenarbeiten. Beispielsweise könnte die Zusammenarbeit einer Datenbankabstraktionsschicht mit einer grafischen Bedienoberfläche und einer Datenbank getestet werden.

Sowohl bei Komponenten- als auch bei Integrationstests steht die technische Korrektheit im Vordergrund. Fachliche Belange werden hier noch nicht berücksichtigt. Diese kommen erst auf der Ebene der Systemtests zum Tragen. Ein Systemtest prüft, ob eine Software den spezifizierten Anforderungen entspricht. Dazu können z.B. die Integration in eine heterogene Umgebung, die Nutzung bereits bestehender Datenbanken oder ähnliche Aspekte gehören.

Die letzte Stufe der Tests im V-Modell stellen die Akzeptanz- oder Abnahmetests dar. Hier ist es das Ziel zu prüfen, ob die Software die spezifizierten Anwendungsfälle korrekt realisiert.

4.1.2 Dynamische Tests

Sämtliche durch das V-Modell klassifizierten Tests sind dynamische Tests, für die eine ausführbare Version der Komponente oder Software vorliegen muss.

Den dynamischen Tests gegenüber stehen die statischen Tests. Bei den statischen Tests handelt es sich um Verfahren wie Code Reviews oder statische Analysen des Quellcodes. Für diese Tests gibt es bereits eine große Anzahl von Methoden und Werkzeugen. Im Fokus der statischen Tests steht handgeschriebener Quellcode in einer definierten Zielsprache, so dass im Rahmen dieser Arbeit

nicht weiter darauf eingegangen wird. Es sei jedoch erwähnt, dass es interessante Ansätze gibt, Pendants zu statischen Tests auf Ebene von Modellen durchzuführen, z.B. MMUnit [MMUNIT].

Die dynamischen Tests werden in der Praxis in drei Varianten unterteilt, die sich durch ihren Kenntnisstand über und die Einflussnahme auf die zu testende Software unterscheiden.

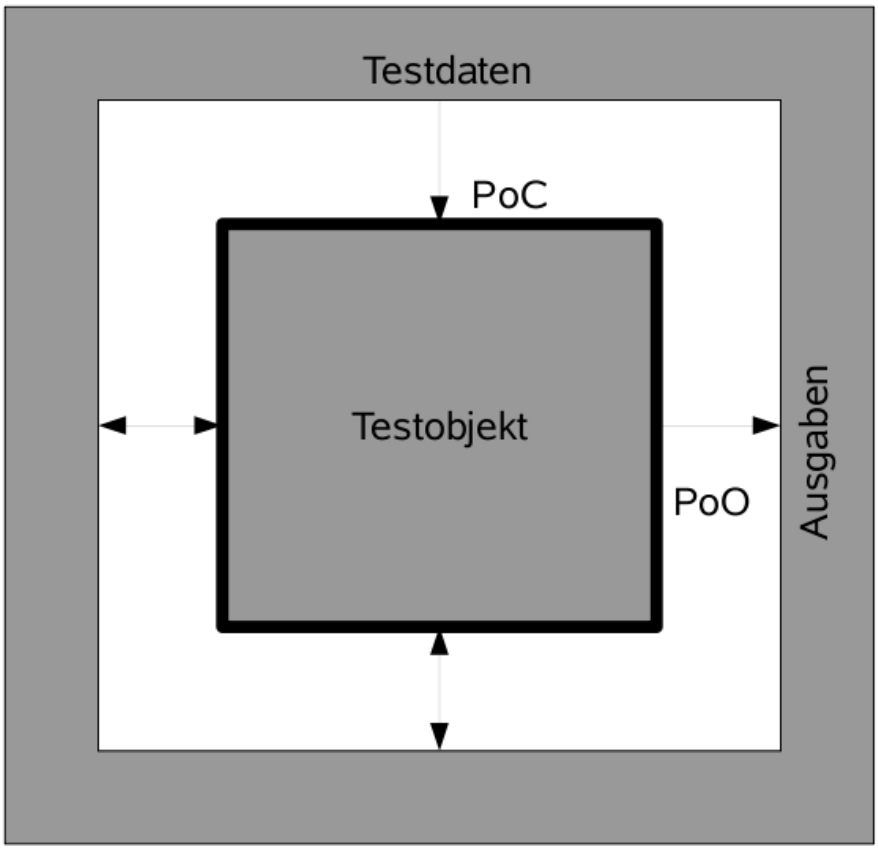

Abbildung 4.1: Schematische Darstellung eines Blackbox-Test

Quelle: Eigene Darstellung, [SPI05]

Die sogenannten Blackbox-Tests setzen keinerlei Kenntnisse über die Implementierung der zu testenden Software voraus und werden nur gegen die Schnittstellen durchgeführt (siehe Abbildung 4.1). Das bedeutet der Point of Control (PoC) und der Point of Observation (PoO) liegen außerhalb des Testsobjekts. Klassische Beispiele für Blackbox-Tests sind Akzeptanztests, aber auch Last- und Performancetests.

Da das Ziel des Testens das Finden von Fehlern ist und ein reiner Schnittstellentest relativ einfach

erfüllt werden kann, kommen in der Testpraxis i.d.R. Greybox-Tests zum Einsatz. Diese setzen eine (eingeschränkte) Kenntnis der inneren Zusammenhänge der Software (Implementierung, Plattformen, Netzwerkfähigkeiten o.ä.) voraus, testen jedoch auch ausschließlich gegen die Schnittstellen. Als Greybox-Tests lassen sich Last- und Performancetests deutlich gezielter ausführen. Außerdem können Stress- und Robustheitstests nur mit Kenntnissen über den Aufbau der Software realisiert werden.

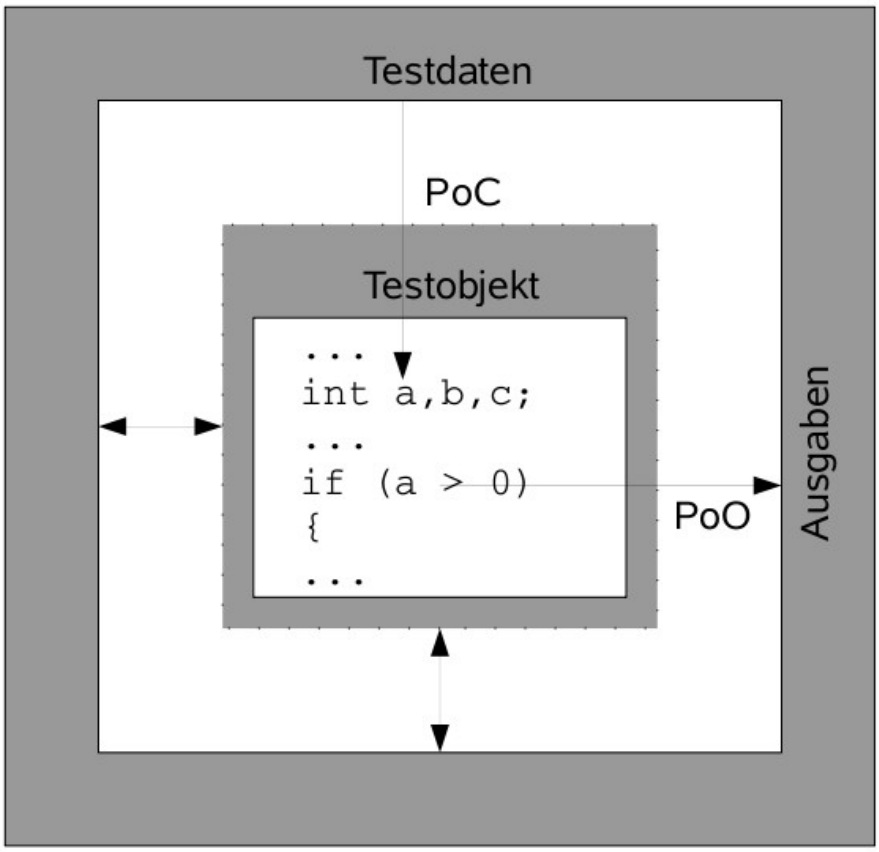

Abbildung 4.2: Schematische Darstellung eines Whitebox-Test

Quelle: Eigene Darstellung, [SPI05]

Die dritte Variante von dynamischen Tests sind die Whitebox-Tests (siehe Abbildung 4.2). Diese setzen nicht nur Kenntnisse über die inneren Abläufe der Software voraus, sondern bieten auch die Möglichkeit, diese gezielt während des Tests zu verfolgen bzw. sogar zu beeinflussen. In diesem Fall liegen Point of Control und Point of Observation innerhalb des Testobjekts.

4.1.3 Testfallerstellung für dynamische Tests

Die folgenden Verfahren zur Testfallerstellung sind allesamt systematisch anwendbar. Verfahren zur intuitiven Testfallerstellung wie das explorative Testen oder Trial-and-Error werden nicht erläutert, da sie für modellbasiertes Testen nicht anwendbar sind.

Weitere Informationen zu den einzelnen Testfallerstellungsverfahren sind in [SPI05] zu finden.

Äquivalenzklassenbildung und Grenzwertanalyse

Die Äquivalenzklassenbildung beruht auf der Idee, dass es für jede Schnittstelle Mengen möglicher Ein- und Ausgabedaten gibt, die identisches Verhalten in der zu testenden Software provozieren. Diese Ein- und Ausgabedaten werden in sogenannte Äquivalenzklassen aufgeteilt. Für fehlerhafte Daten müssen ebenfalls Äquivalenzklassen erstellt werden. Die gefundenen Äquivalenzklassen werden dann als Datengrundlage für die Testfälle verwendet.

Die Grenzwertanalyse beruht auf der Äquivalenzklassenbildung. Hier werden die Grenzwerte der Daten von Äquivalenzklassen betrachtet und Testfälle jeweils innerhalb und außerhalb der Grenzen realisiert.

Beide Testfallerstellungsverfahren sind sehr gut für Komponententests geeignet, eingeschränkt bzw. in Kombination mit anderen Verfahren auch für andere Testebenen.

Zustandsbasiertes Testen

Grundlage für das zustandsbasierte Testen ist, dass die zu testende Software durch einen Zustandsautomaten beschrieben werden kann. Ziel des Tests ist entweder die Ausführung aller möglichen Transitionen oder das Erreichen aller definierten Zustände.

Das Verfahren ist vor allem auf der Ebene der Komponententests geeignet, da für höhere Ebenen i.A. nur sehr schwer Zustandsautomaten mit der nötigen Vollständigkeit modelliert werden können.

Ursache-Wirkung-Graph-Analyse

Bei der Ursache-Wirkung-Graph-Analyse ist die grundlegende Idee, Testfälle durch Abhängigkeiten zwischen den Eingabedaten herzuleiten. Voraussetzung dafür ist, dass diese aus der Spezifikation ermittelbar sind.

Die Eingabedaten bzw. deren Kombinationen werden als Ursachen betrachtet. Diese Ursachen werden dann grafisch mit den entsprechenden Wirkungen verknüpft. Der entstandene Graph wird

dann in eine Entscheidungstabelle umgewandelt, aus der die Testfälle direkt abgelesen werden können.

Geeignet ist dieses Verfahren zur Testfallerstellung vor allem auf der Ebene der Integrations- und Systemtests.

Anwendungsfallbasierte Testerstellung

Die anwendungsfallbasierte Testerstellung erfolgt anhand der spezifizierten Anwendungsfälle.

Diese müssen Vor- und Nachbedingungen sowie einen Ablauf enthalten, woraus dann Testfälle abgeleitet werden können.

In der Regel kann das Verfahren auf der Ebene der Akzeptanztests angewendet werden. Da Anwendungsfälle normalerweise keine Ein- und Ausgabedaten enthalten, ist eine Kombination mit der Äquivalenzklasseanalyse sinnvoll.

Anweisungsüberdeckung

Für die Anweisungsüberdeckung ist ein sogenannter Kontrollflussgraph (siehe Abbildung 4.3) notwendig, aus dem die möglichen Anweisungssequenzen abgeleitet werden können. Diese

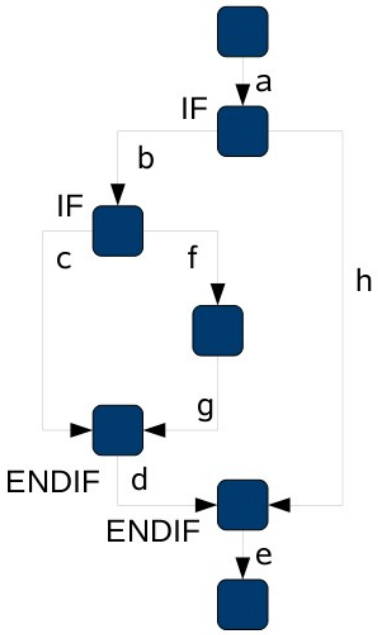

Abbildung 4.3: Beispiel eines Kontrollflussgraphen

Quelle: Eigene Darstellung nach [SPI05]

Sequenzen dienen dann als Grundlage für Tests, wobei das Ziel ist, jede Anweisung einmal zu durchlaufen.

Die Anweisungsüberdeckung kann nicht erreichbaren Code finden und wird klassischerweise für Modultests eingesetzt, ist jedoch auch auf andere Ebenen übertragbar.

Zweigüberdeckung

Etwas umfangreicher als die Anweisungsüberdeckung ist die Zweigüberdeckung. Auch hier dient der Kontrollflussgraph als Grundlage, Ziel der Tests ist es jedoch, alle Zweige einmal zu durchlaufen. Mit Zweigüberdeckungstests können fehlende Anweisungen gefunden werden.

Die Anwendungsmöglichkeiten der Zweigüberdeckung sind analog zur Anweisungsüberdeckung, wobei eine Zweigüberdeckung von 100% auch eine Anweisungsüberdeckung von 100% garantiert.

Pfadüberdeckung

Grundlage der Pfadüberdeckung ist die Zweigüberdeckung. Bei der Pfadüberdeckung werden jedoch auch Schleifen berücksichtigt, wobei jeder Schleifendurchlauf einem neuen Pfad entspricht. Die Pfadüberdeckung sollte eingesetzt werden, wenn das mehrfache Durchlaufen von Kontrollflüssen relevant ist.

Bedingungstest

Abgeleitet von Anweisungs- und Zweigüberdeckung, aber schwächer als diese Verfahren sind die Bedingungstests. Dieses Verfahren zur Testfallerstellung hat das Ziel, im Ablauf enthaltene Zweigbedingungen auf Korrektheit zu prüfen.

Es gibt drei unterschiedliche Varianten des Bedingungstests, die in [SPI05] ausführlich erläutert, hier jedoch nicht von Interesse sind.

4.2 Modellbasiertes Testen

Von den vorgestellten Testfallerstellungsverfahren und Testvarianten eignen sich nicht alle gleichermaßen für ein modellbasiertes Vorgehen. Prinzipiell kann zwar für jedes Verfahren eine eigene Modellierungssprache entwickelt werden, im Rahmen dieser Arbeit soll aber die UML als Standardsprache dienen.

Sehr gut mit UML kombinierbar ist die zustandsbasierte Testfallerstellung, da die UML eine

ausgereifte Unterstützung für die Modellierung von Zustandsautomaten bereithält. Vorausgesetzt die Modellierung erfolgte korrekt und vollständig, ist für die Ableitung von Testfällen aus den Zustandsautomaten nur eine Modelltransformation notwendig.

Weitaus weniger gut lassen sich Ursache-Wirkung-Graphen mit UML dargestellen bzw. aus UML generiert. Alternativ können, sofern möglich, Zustandsautomaten modelliert werden oder ein Überdeckungsverfahren auf Aktivitätsdiagramme angewendet werden.

Die Überdeckungsverfahren, die in der Literatur häufig als Whitebox-Verfahren mit Bezug zum Quellcode erklärt werden, eignen sich hervorragend dazu, Tests aus Aktivitätsmodellen zu gewinnen. Der Zusammenhang zwischen dem Kontrollflussgraph und einem Aktivitätendiagramm ist offensichtlich.

Die einzelnen Testfälle können mit der Anwendung unterschiedlicher Transformationsregeln für Anweisungs-, Zweig- und Pfadüberdeckung gewonnen werden. Gleiches gilt für Bedingungstests als spezifische Variante der Zweigüberdeckung.

Leider stellt die UML nur eine sehr rudimentäre Unterstützung für Anwendungsfallmodellierung zur Verfügung. Allerdings können jedem Anwendungsfall ein oder mehrere Verhalten zugewiesen werden. Dieses Verhalten ist als Zustandsautomat oder als Aktivität modellierbar, was es erlaubt, auch anwendungsfallbasierte Tests aus UML zu generieren – diese sind dann nichts anderes als beispielsweise die entsprechende Pfadüberdeckung.

Die bisher genannten Tests decken alle vier Testebenen des V-Modells ab, berücksichtigen aber nicht die Ein- und Ausgabedaten. Für derartige Tests kommen normalerweise die Äquivalenzklassen oder Grenzwertanalyse zum Einsatz. Für diese bietet die UML leider auch keine direkte Lösung an, sondern zwingt zu zusätzlicher Modellierung. Eine mögliche Lösung dafür ist im UML Testing Profile enthalten, das im nächsten Kapitel vorgestellt wird.

4.3 Das UML Testing Profile

Die folgende Übersicht des UML Testing Profile (UTP) bezieht sich auf die von der OMG als Standard verabschiedete Version 1.0 vom 07.07.2005. Bei dieser Version des UTP handelt es sich um ein Profil zur Erweiterung der Modellierungssprache UML 2.0.

Ziel des UTP ist es, eine Modellierungssprache auf Basis von UML zu schaffen mit der

Testsartefakte entworfen, visualisiert, spezifiziert, analysiert, konstruiert und dokumentiert werden können. Um dieses Ziel zu realisieren, haben die Entwickler des UTP die Domäne der Softwaretests in vier Konzeptbereiche unterteilt: Testarchitektur (Test Architecture), Testverhalten (Test Behavior), Testdaten (Test Data) und Zeit (Time Concepts).

Eine detaillierte Beschreibung zur Einordnung des UTP innerhalb der von der OMG herausgegebenen Standards, insbesondere der UML ist in [UTP05] zu finden.

4.3.1 Kurzbeschreibungen der Elemente des UTP

Bei den Kurzbeschreibungen der Elemente des UTP, die im Folgenden aufgeführt sind, handelt es sich um sinngemäße Übersetzungen aus [UTP05], wo alle Elemente sehr ausführlich beschrieben sind.

Die für die vorliegende Arbeit relevanten Details der Elemente werden – sofern erforderlich – an den entsprechenden Stellen weiter unten erläutert.

Testarchitektur

Innerhalb des Bereichs Testarchitektur werden die strukturellen Aspekte eines Testkontextes beschrieben (siehe Abbildung 4.4). Dazu gehören

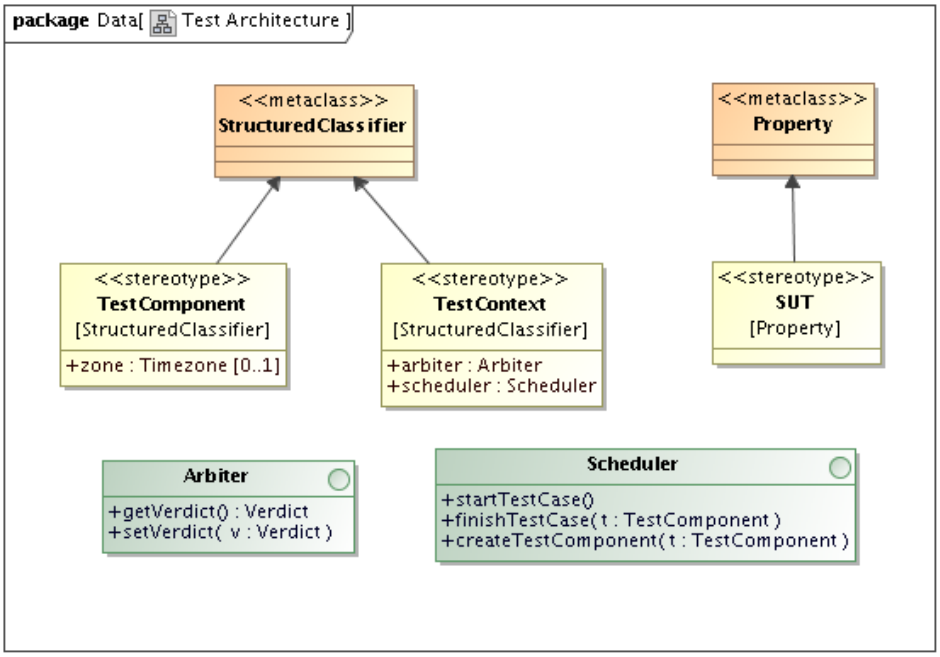

Abbildung 4.4: Profilbereich Test Architecture des UTP

Quelle: Eigene Darstellung nach [UTP05]

SUT (System under Test)

Das System under Test ist der zu testenden Teil eines Systems. Formal kann es sich dabei sowohl um das Gesamtsystem als auch um einen Teil davon, ein Subsystem oder eine Komponente des Systems handeln. Das SUT wird in jedem Fall als Black-Box betrachtet, eine Kommunikation ist also nur über die öffentlichen Schnittstellen möglich.

Test Context

Eine Sammlung von Testfällen (Test Cases) sowie eine Testkonfiguration (Test Configuration) auf deren Basis die Testfälle ausgeführt werden stellt der Test Context dar.

Test Configuration

Die Testkonfiguration ist eine Sammlung von Objekten des Typs Test Component, den Assoziationen zwischen diesen Objekten sowie deren Assoziationen zum SUT. Dabei spezifiziert die Testkonfiguration sowohl die initiale Konfiguration zum Zeitpunkt des Starts eines Testfalles als auch die maximale Anzahl der Objekte und Assoziationen, die während der Ausführung auftreten können.

Test Component

Eine Testkomponente ist eine Klasse eines Testsystems, deren Objekte das Verhalten eines Testfalles realisieren. Testkomponenten besitzen eine Reihe von Schnittstellen, über die sie mit anderen Testkomponenten oder dem SUT kommunizieren können.

Arbiter

Der Arbiter (dt. Schiedsrichter) ist eine Eigenschaft von Testfällen oder Testkomponenten. Er dient dazu die Testergebnisse zu bewerten und das allgemein ermittelte Ergebnis den jeweiligen Testfällen oder Testkomponenten zuzuweisen.

Scheduler

Der Scheduler ist eine Eigenschaft von Testkontexten und wird benutzt, um die Ausführung der unterschiedlichen Testkomponenten zu steuern. Außerdem speichert der Scheduler Informationen darüber, welche Testkomponenten zu welchem Zeitpunkt existieren und an welchen Testfällen sie beteiligt sind. Der Scheduler aktiviert den Arbiter und steuert die Erzeugung und das Löschen von Testkomponenten.

Utility Part

Als Utility Part werden die Teile des SUT bezeichnet, die notwendig sind, um einer Testkomponente das Ausführen ihres Testverhaltens zu ermöglichen.

Testverhalten

Die Konzepte des Bereichs Testverhalten erweitern die in der UML 2.0 enthaltenen Verhaltenskonzepte um die Möglichkeit testspezifisches Verhalten sowie Testziele zu modellieren und das SUT bewerten zu können (siehe Abbildung 4.5).

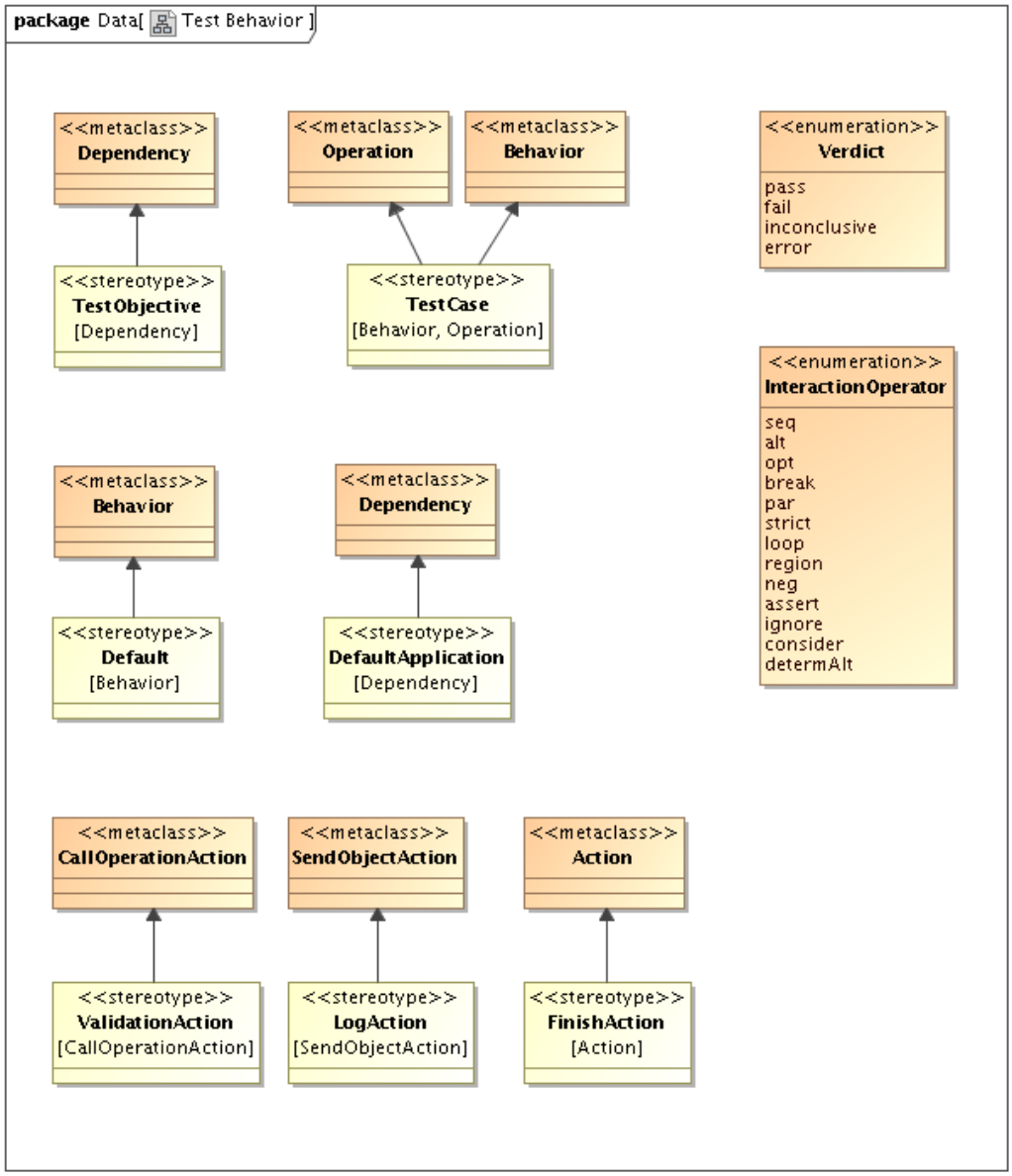

Abbildung 4.5: Profilbereich Test Behavior des UTP

Quelle: Eigene Darstellung nach [UTP05]

Test Control

Teststeuerungen bestimmen, wie Testfälle in einen Testkontext eingebracht werden. Sie spezifizieren aus technischer Sicht, wie ein SUT durch einen Testkontext getestet wird.

Test Case

Testfälle bestehen aus Eingaben, erwarteten Ergebnissen und den Bedingungen, die für ihre Ausführung erfüllt sein müssen sowie aus Schleifen, Alternativen und Sequenzen sowie Standard-Eingaben und erwarteten Resultaten für bzw. vom SUT. Ein Testfall ist damit die vollständige technische Spezifikation, wie das SUT mit einem bestimmten Ziel getestet wird. Er hat ein konkretes Testziel und kann andere Testfälle einschließen. Zur Auswertung der Ergebnisse nutzt der Testfall einen Arbiter. Jeder Testfall ist eine Eigenschaft eines Testkontextes. Testfälle spezifizieren, wie eine Reihe von Testkomponenten mit dem SUT interagiert, um ein bestimmtes Testziel zu erreichen. Sowohl das SUT als auch die entsprechenden Testkomponenten müssen Bestandteile des Testkontextes sein, zu dem der Testfall gehört.

Test Invocation

Ein Testaufruf erfolgt von einem definierten Testkontext aus. Der Testaufruf resultiert in der Ausführung eines Testfalles mit bestimmten Parametern. Testaufrufe werden im Testlog protokolliert.

Test Objective

Ein Testziel beschreibt einen zu testenden Sachverhalt. Testziele müssen Testfällen zugeordnet sein.

Stimulus

Als Stimulus (Impuls) werden Testdaten bezeichnet, die an das SUT gesendet werden um es zu steuern und die Reaktionen des SUT zu bewerten.

Observation

Oberservations sind Testdaten, die benutzt werden, um die Reaktionen des SUT vergleichen und bewerten zu können.

Coordination

Das Element Coordination wird verwendet um verteilte oder konkurrierende Testkomponenten

sowohl funktional als auch im zeitlichen Verlauf zu koordinieren. So wird sichergestellt, dass die Testausführung deterministisch und reproduzierbar verläuft und zu definierten Ergebnissen führt. Die Koordination kann implizit über die Ausführungsreihenfolge oder explizit durch Nachrichtenaustausch zwischen den beteiligten Komponenten erfolgen.

Default

Default repräsentiert Verhalten, das von einer Observation getriggert wird, aber zu keinen Testfall gehört. Es wird von Testkomponenten ausgeführt.

Verdict

Ein Verdict (Urteil) ist die Beurteilung der Korrektheit des SUT und wird von einem Testfall erzeugt. Verdicts können ebenso genutzt werden, um Fehler zu protokollieren. Als vordefinierte Verdicts sind pass (durchlaufen), fail (fehlgeschlagen), inconclusive (ergebnislos) und error (defekt) im UTP enthalten. Das Verdict eines Testfalls wird vom Arbiter bestimmt.

Validation Action

Mit der Validation Action wird der Status der Ausführung eines Testfalles durch eine Betrachtung des SUT oder zusätzlicher Parameter des SUT ausgewertet. Sie wird von einer Testkomponente ausgeführt und und bestimmt das lokale Verdict dieser Testkomponente.

Log Action

Die Log Action schreibt Informationen ins Test Log.

Test Log

Ein Test Log ist das Ergebnis der Ausführung eines Testfalles. Es repräsentiert die unterschiedlichen Nachrichten, die zwischen den Testkomponenten und dem SUT ausgetauscht worden sind bzw. die unterschiedlichen Zustände der beteiligten Testkomponenten.

Testdaten

Die Konzepte der Testdaten erweitern die in der UML 2.0 definierten Konzepte zur Modellierung von Daten. So wird es möglich, spezifische Daten für Stimuli und Beobachtungen des SUT sowie die Koordination zwischen Testkomponenten modellieren zu können (siehe Abbildung 4.6).

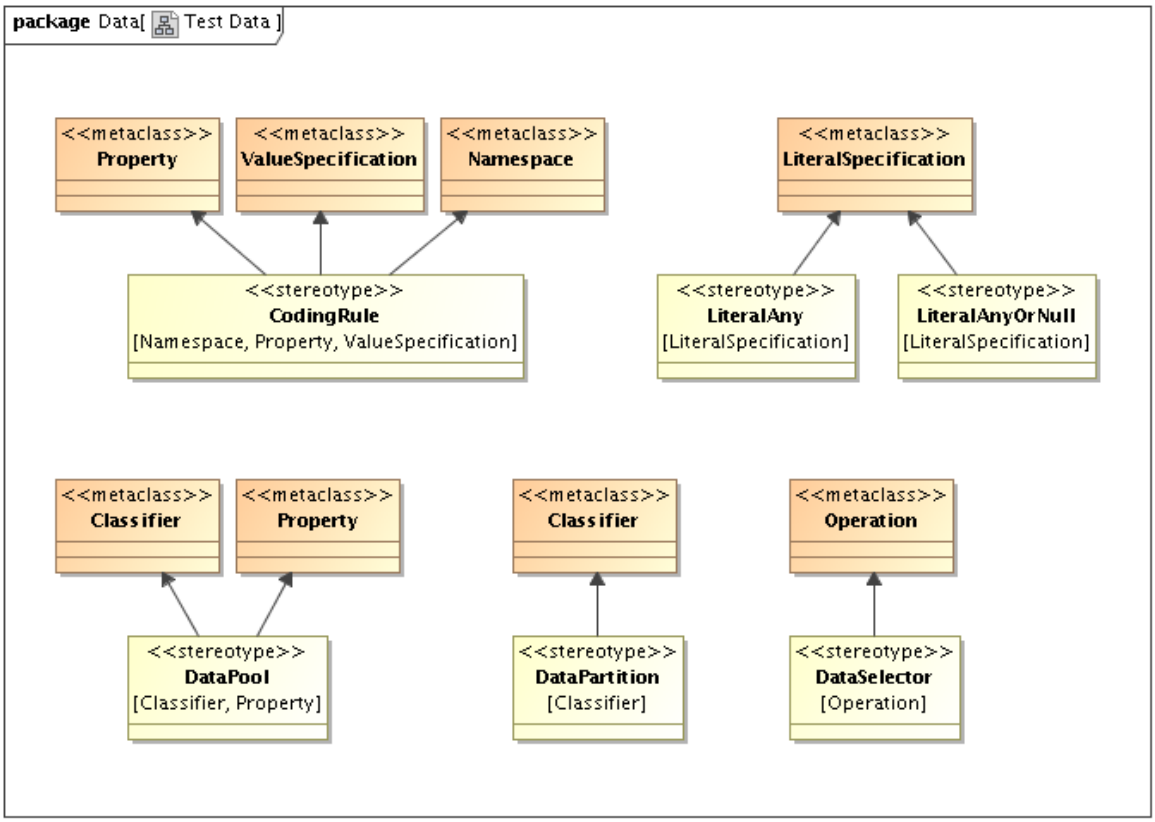

Abbildung 4.6: Profilbereich Test Data des UTP

Quelle: Eigene Darstellung nach [UTP05]

Wildcard

Die Wildcards erlauben es dem Anwender, explizit zu definieren, ob ein Wert vorhanden sein muss oder nicht. Sie sind spezielle Symbole, die Werte oder Wertbereiche repräsentieren. Sie ersetzen konkrete Symbole in Instanzspezifikationen. Es existieren Wildcards für: irgendein Wert, irgendein Wert oder kein Wert sowie für das Auslassen eines Wertes.

Data Pool

Ein Datenpool ist eine Sammlung von Datenpartitionen oder festgelegten Werten. Er kann von einem Testkontext oder einer Testkomponente während der Auswertung von Testkontexten oder Testfällen benutzt werden. In der Praxis hält ein Datenpool Werte oder Datenpartitionen vor, die für Regressionstests genutzt werden.

Data Partition

Eine Data Partition ist ein logischer Wert für einen Parameter, der in Stimulus oder Observation

genutzt wird. Typischerweise handelt es sich dabei um Äquivalenzklassen oder Mengen zulässiger Werte.

Data Selector

Der Datenselektor bestimmt wie Werte oder Äquivalenzklassen aus einem Datenpool oder einer Datenpartitionen ausgewählt werden.

Coding Rule

Sofern Schnittstellen des SUT verschiedene Kodierungen benutzen, die vom Testsystem berücksichtigt werden müssen (z.B. CORBA GIOP/IIOP, IDL, XML), sollten diese Teil der Testspezifikation sein.

Zeit

Die Konzepte aus dem Bereich Zeit ergänzen die in der UML 2.0 bereits enthaltenen Konzepte um die Möglichkeit, zeitliche Einschränkungen, zeitlich basierende Beobachtungen sowie Zeitbegrenzer in der Spezifikation von Testverhalten zu nutzen (siehe Abbildung 4.7). Damit wird die Bewertung der Ausführung von Testfällen anhand der zeitlichen Beobachtungen möglich.

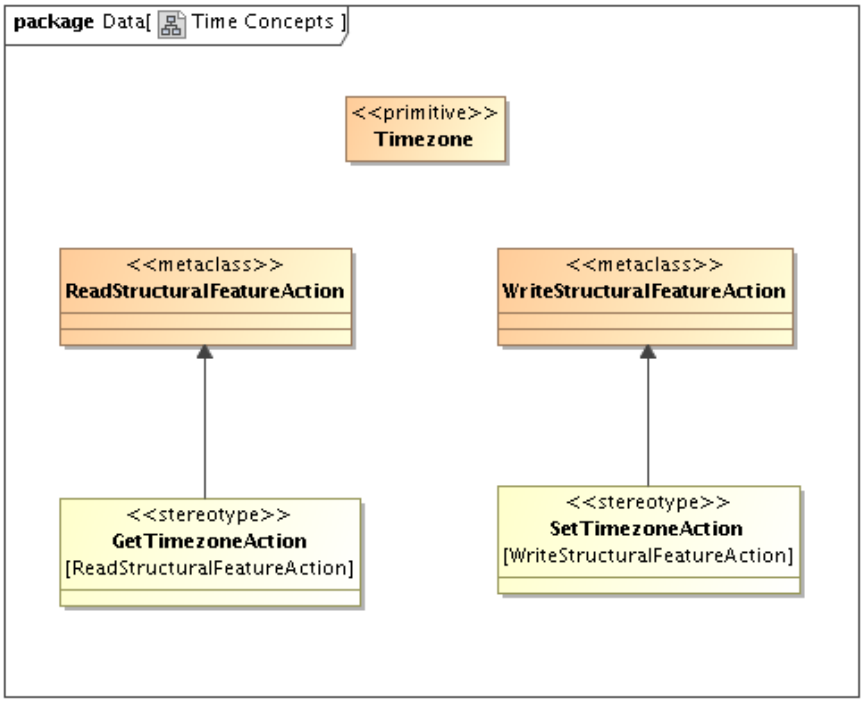

Abbildung 4.7: Profilbereich Time Concepts des UTP

Quelle: Eigene Darstellung nach [UTP05]

Timer

Timer (Zeitbegrenzer) sind Eigenschaften von Testkomponenten. Sie ermöglichen es, ein Timeout Ereignis auszulösen, wenn ein definierter Zeitraum abgelaufen ist und werden beim Start mit einem Zeitpunkt initialisiert zu dem der Timeout ausgelöst werden soll. Sobald ein Timeout ausgelöst wurde, ist der entsprechende Timer abgelaufen. Timer können auch angehalten werden.

Die Ablaufzeit und der aktuelle Status eines Timers können ausgelesen werden.

Timezone

Zeitzonen gruppieren Testkomponenten, wobei jede Testkomponente zu exakt einer Zeitzone gehören muss. Alle Testkomponenten einer Zeitzone haben die gleiche Zeit, damit ist gewährleistet, dass die Zeit zwischen den Testkomponenten synchronisiert ist.

4.3.2 Zusammenhang des UTP mit anderen Testwerkzeugen und -notationen

Bei der Entwicklung des Metamodells des UTP haben die – nicht explizit als solche beschriebenen – Metamodelle des JUnit Testframeworks [JUNIT] und der TTCN-3 [TTCN3] Pate gestanden.

Somit finden sich Aspekte beider Testwerkzeuge bzw. -notationen im UTP wieder.

Eine ausführliche Diskussion sowie die Darstellung eines Mappings mit JUnit und TTCN-3, wie es der Vorstellung der Autoren des UTP entspricht, ist sowohl in [UTP05] als auch in [SCH08] zu finden. Die Referenzimplementierung des UTP in Form des MOF-basierten Metamodells ist als XMI Schema Bestandteil von [UTP05].

Im Bereich der modellbasierten Testgenerierung finden sich zum Zeitpunkt der Recherche (August 2009) keine Werkzeuge, die das UTP direkt unterstützen. Es gibt zwar einige Testwerkzeuge, die eigene Möglichkeiten zur Testmodellierung bereits anbieten oder planen (z.B. Tosca [TOSCA] oder GUIdancer [GUIDAN]), allerdings sind diese proprietär. So modellierte Tests können aber nur im entsprechenden Werkzeug ausgeführt werden.

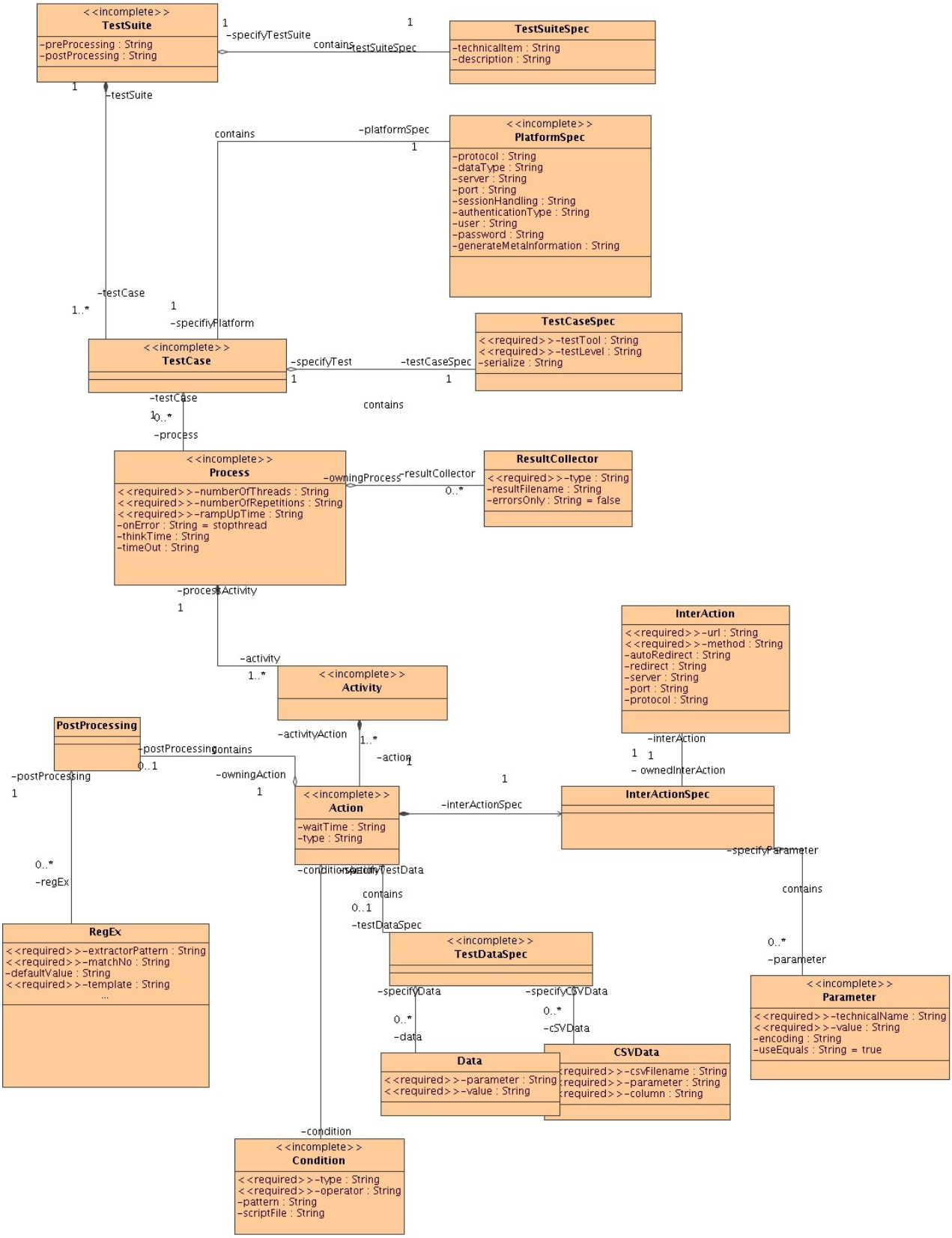

Abbildung 4.8: Das Metamodell von openArchitectureWare-Test

Quelle: http://www.mdtd.de/

Die frei verfügbare Testgenerierungssoftware openArchitectureWare-Test [OAWT] verwendet ein eigenes, nicht auf dem UTP basierendes Metamodell. Ein Mapping zwischen UTP und selbigem ist prinizipiell vorstellbar, da die Domäne des Testens einige so grundlegende Konzepte enthält, dass diese in beiden Metamodellen (siehe Abbildung 4.8) zu finden sind. Allerdings bringt das Metamodell von openArchitectureWare-Test einige äußerst technische Konzepte wie CSV-Dateien als Testdatenquellen und reguläre Ausdrücke zur Validierung mit, die zwar durchaus in einem Testwerkzeug Verwendung finden können, im Metamodell jedoch fehl am Platze sind. Das Metamodell von oaw-Test wurde daher nicht weiter betrachtet.

Der Enterprise Architect von Sparx Systems, eines der für die Beispiele benutzten Modellierungswerkzeuge, stellt ein UML-Profil zur Testgenerierung zur Verfügung. Dieses wird allerdings bereits seit mehreren Jahren parallel zum UTP entwickelt, so dass hier ähnliches wie für das Metamodell von openArchitectureWare-Test gilt: Die Konzepte ähneln sich, im Detail existieren jedoch Unterschiede. Im konkreten Fall darauf beziehen sie sich vor allem darauf, dass der Enterprise Architect kein vollständiges Metamodell der UML für die Profilerstellung bereitstellt. So müssen z.B. Stereotypen, die die Metaklassen Behavior erweitern sollen, statt dessen die Metaklassen State Chart und Activity erweitern.

Für Magic Draw, den zweiten verwendeten Modellierer, war ebenfalls keine Umsetzung des UTP erhältlich.

Die im UTP enthaltenen Elemente ermöglichen eine sehr detaillierte Spezifikation von Tests auf den Ebenen der Unit- und Komponententest. In [SCH08] ist zusätzlich auch beschrieben, wie System- und Akzeptanztests mit Hilfe des UTP modelliert werden können. Aus praktischer Sicht erscheint die von den Autoren bevorzugte Vorgehensweise jedoch unnötig kompliziert.

Sowohl für Akzeptanztests als auch für die meisten Systemtests können Testfälle im Allgemeinen aus den Anwendungsfällen oder den Anforderungen hergeleitet werden, so dass hier eine Spezifikation bzw. Modellierung top-down angemessen erscheint. Hinzu kommt noch, dass die meisten Werkzeuge, die insbesondere im Bereich der Akzeptanztests Anwendung finden, eher an Anwendungsfällen orientiert sind und den Ablauf aus Sicht des Nutzers darstellen.

Diese Akzeptanztests lassen sich mit Anwendungsfall- und Aktivitätsdiagrammen deutlich leichter modellieren als mit Sequenz-, Interaktions- und Objektdiagrammen.

Ein großer Teil der im UTP umgesetzten Aspekte ist optimal für das Spezifizieren von Echtzeit- und Performancetests sowie von Tests der Kommunikation technischer Systeme geeignet. Die Elemente,

die diese Aspekte realisieren, spielen jedoch im Bereich der Akzeptanz- und Systemtests eine eher untergeordnete Rolle. Andererseits stellen diese Arten von Tests Anforderungen an das UTP, die durch Erweiterungen am leichtesten realisierbar sind.

Die im Rahmen dieser Arbeit vorgenommenen Erweitererungen des UTP werden im folgenden Kapitel beschrieben.

4.4 Das GeneSEZ Testing Metamodell

Die Integration eines Metamodells für Softwaretests in GeneSEZ ist im Rahmen dieser Arbeit vorgesehen. GeneSEZ stellt bereits eine vollständige Infrastruktur für modellgetriebene Softwareentwicklung bereit, so dass die Integration in eine stabilen Infrastruktur stattfinden kann.

So wie bei GeneSEZ im Allgemeinen der Fokus auf dem Entwickler als Endanwender liegt, steht beim Testing Metamodell der Testarchitekt im Vordergrund. Es soll diesem möglichst einfach gemacht werden, aus einem UML-Diagramm Testscripten für das jeweils verwendete Werkzeug zu generieren, ohne sich tiefer mit Metamodellen oder der Notation von Testscripten beschäftigen zu müssen.

Für viele Testwerkzeuge (auch das verwendete QF-Test) ist eine gleichzeitige Generierung von Anwendungs- und Testcode vorteilhaft. Bei der parallelen Generierung können Identifikatoren, die bei der Transformation in Anwendungscode generiert werden, in die Testscripten übernommen werden. Mit Hilfe dieser Identifikatoren wird es möglich zur Laufzeit der Anwendung, die Komponenten auf die die einzelnen Aktionen eines Tests wirken sollen, zu finden. Dies spielt insbesondere bei Tests, die Anwendungen auf Ebene der Benutzeroberfläche testen, eine wichtige Rolle.

4.4.1 Elemente des GeneSEZ Testing Metamodell

Das GeneSEZ Testing Metamodell ist im Wesentlichen eine Realisierung des UTP. Da der Schwerpunkt es UTP aber auf Echtzeitsystemen liegt und die Autoren des UTP sich stärker auf Interaktionen statt auf Aktvitiäten konzentrieren, muss das UTP erweitert werden.

Dies betrifft insbesondere den Bereich der Test Behavior. Die meisten Testwerkzeuge arbeiten ablauforientiert und führen eine Reihe von Testschritten nacheinander aus, um schließlich eine Validierung durchzuführen. Im UTP sind zwar eine ValidationAction, eine LogAction und eine

FinishAction enthalten, jedoch keine Elemente, die die einzelnen Testschritte repräsentieren.

Das GeneSEZ Testing Metamodell versucht diese Lücke zu schließen, in dem es die UTP Elemente vom Typ TestCase als Spezialisierung der Aktivitäten betrachtet und die im Folgenden beschriebenen Klassen ins Metamodell einführt.

Um die Transformation des UML-Modells in das Testing Metamodell wiederholbar zu machen, sind alle Elemente des Testing Metamodell, einschließlich der Elemente, die aus dem UTP übernommen werden, von der Klasse TElement abgeleitet. Diese Klasse hat den gleichen Zweck wie die Klasse MElement im GeneSEZ Metamodell. Sie besitzt als Attribute den Namen des Elements und die GUID. Somit kann die eindeutige Identifizierung alle Elemente im Modell sichergestellt werden.

Um Tests sinnvoll strukturieren zu können, werden in das GeneSEZ Testing Metamodell zwei weitere Klassen des GeneSEZ Metamodells übernommen. Das sind TPackage als Pendant zu MPackage und die davon abgeleitete Klasse TModel, die wiederum MModel im GeneSEZ Metamodell entspricht.

Ein TPackage enthält eine Anzahl von TTestSuites, die den MUseCase Elementen des GeneSEZ Metamodells entsprechen. Pro Anwendungsfall kann es mehrere Testfälle geben, die im GeneSEZ Testing Metamodell zu einer Testsuite zusammengefasst werden. Diese Variante der Gruppierung von Testfällen wurde gewählt, weil sie zum Einen sehr einfach abzubilden ist und zum Anderen transparent in das Metamodell vom UTP integriert werden kann, ohne dass dessen Elemente wesentlich verändert werden müssen.

Die Testfälle in der Testsuite werden durch Elemente der Klasse TTestCase repräsentiert. Ein Testcase kann mehrere Elemente der Klasse TSequence enthalten. Eine TSequence kann als SetUp oder TearDown spezialisiert werden, die den klassischen Vor- und Nachbereitungsschritten für Tests entspricht. Jedes Element der Klasse TSequence beinhaltet mehrere Elemente der Klassen TTestStep und TFlow. TTestStep entspricht dabei den Knoten in Aktivitäten, TFlow den Flüssen.

TestStep kann noch weiter spezialisiert werden. Die momentan realisierten Spezialisierungen sind an den Testschritt-Knoten von QF-Test orientiert.

4.4.2 Transformation von UML ins GeneSEZ Testing Metamodell

Aus einem UML Modell wird durch eine Modell-zu-Modell-Transformation zunächst ein GeneSEZ Modell erzeugt. Dieses kann dann unter Verwendung einer GeneSEZ-Plattform in Quellcode einer unterstützten Programmiersprache transformiert werden. Parallel dazu werden die für das Testen

relevanten Modellteile in ein GeneSEZ Testing Modell transformiert, aus dem dann die Testscripten generiert werden können (siehe Abbildung 4.9). Natürlich kann auch nur die Transformation in das GeneSEZ Testing Metamodell durchgeführt werden.

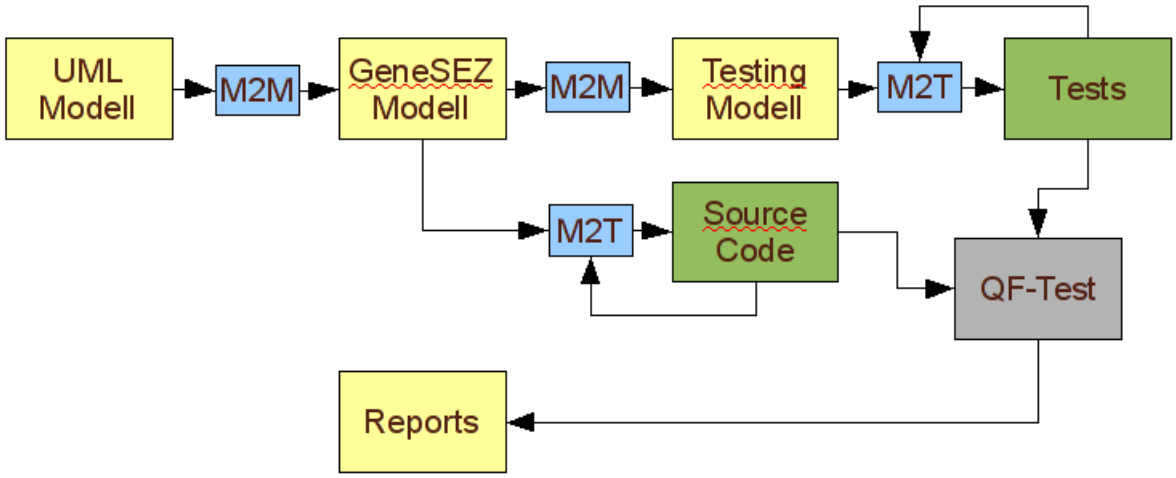

Abbildung 4.9: Parallele Erzeugung von Anwendung und Tests mit GeneSEZ

Quelle: Eigene Darstellung

Unabhängig davon besteht jederzeit die Möglichkeit, Modelle direkt in das GeneSEZ Testing Metamodell zu transformieren, wenn für das Ursprungsmodell ein formales Metamodell existiert. Damit ist es beispielsweise denkbar, Testscripten von proprietären Werkzeugen für ein anderes Werkzeug nutzbar zu machen.

Um die dargestellte Vorgehensweise umzusetzen, wird die gleiche Strategie wie bei der Implementierung von GeneSEZ angewendet: das Testing Metamodell ist in UML beschrieben und wird mit Hilfe des Eclipse Modeling Frameworks in ein Ecore-Modell transformiert. Dieses ist in die GeneSEZ Infrastruktur als Eclipse-Plugin unter dem Namen `de.genesez.testing` integriert. Zusammen mit dieser Realisierung des Metamodells wird ein Transformationsscript implementiert.

Für die eigentliche Erzeugung von Testscripten wurde am Beispiel QF-Test eine Modell-zu-Text-Transformation implementiert, die an den prinzipiellen Aufbau der GeneSEZ-Plattformen für Java und .NET angelehnt ist. Diese Plattform ist ebenfalls als Eclipse-Plugin unter dem Namen `de.genesez.platforms.qftest` in die GeneSEZ Infrastruktur integriert.

Weitere Plattformen für andere Testwerkzeuge sind jederzeit realisierbar. Beide Eclipse-Plugins stellen neben den Transformationen auch die entsprechenden Workflow-Komponenten für

openArchitectureWare bereit, um sie transparent in einem MDSD-Prozess nutzen zu können.

4.4.3 Exemplarische Anwendungen des GeneSEZ Testing Metamodell

Zum Abschluss von Kapitel 4 soll nun noch eine kurze Anwendung des GeneSEZ Testing Metamodell aufgezeigt werden. Als Grundlage dient ein Test einer Webanwendung, die E-Mails versenden kann. Dazu werden zwei Testfälle modelliert, zunächst die Anmeldung eines Nutzers und danach der Vorgang des eigentlichen E-Mail-Versands.

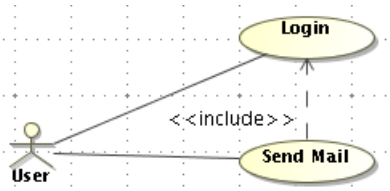

Abbildung 4.10: UseCase Diagramm für die Testfölle

Quelle: Eigene Dargestellung

Ausgangspunkt für die Testgenerierung sind die in Abbildung 4.10 dargestellten Anwendungsfälle. Das Senden einer E-Mail schließt den Anwendungsfall des Logins mit ein. Jeder dieser Anwendungsfälle ist mit einer Aktivität genauer beschrieben, die als `<<testcase>>` annotiert wurde.

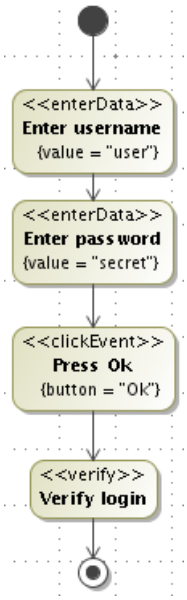

Abbildung 4.11: Ausführlicher Testfall für Login

Quelle: Eigene Darstellung

Der Testfall für den Anwendungsfall Login ist in Abbildung 4.11 zu sehen. Für jeden Schritt des Anwendungsfalles gibt es einen Testschritt. Die Testschritt sind Aktionen, die durch den Stereotyp <<enterData>> genauer bestimmt werden. Solche Aktionen können durch einen Tagged Value einen Wert (`value`) erhalten, der für den Test verwendet werden soll. Nach der Eingabe der Daten wird das Anclicken eines Buttons durch das Testwerkzeug angestoßen. Dies geschieht durch die als `<<clickEvent>>` annotierte Aktion, die den Namen des Buttons als Parameter erhält.

Zum Abschluss des Testfalls kann durch eine als `<<verify>>` deklarierte Aktion eine Überprüfung des Vorgangs durchgeführt werden.

Was genau `<<enterData>>`, `<<clickEvent>>` und `<<verify>>` tun und wie sie realisiert werden, ist abhängig vom Testwerkzeug.

Da das ausführliche Modellieren solcher Testfälle sehr zeitaufwändig und für den Anwender unbefriedigend ist, gibt es auch die Möglichkeit, solche Tests in einer Aktivität komprimiert zu modellieren. Eine solche Aktivität ist in Abbildung 4.12 dargestellt.

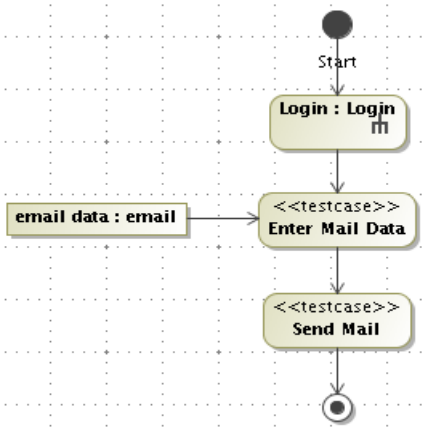

Abbildung 4.12: Komprimierte Darstellung eines Tests mit Dateneingabe

Quelle: Eigene Darstellung

Hier wird ein Objektknoten zur Eingabe der Daten, aus denen eine E-Mail besteht, verwendet. Dazu muss eine Klasse mit entsprechender Struktur modelliert werden. Aus den Attributen dieser Klasse werden dann einzelne Aktionen vom Typ `<<enterData>>` generiert und so der Testfall Enter Email Data erzeugt. Dieses Vorgehen setzt eine etwas unkonventionelle Anwendung von Objektknoten voraus, ist aber in der UML zulässig und auch nach der Transformation in der GeneSEZ Metamodell und das GeneSEZ Testing Metamodell technisch sehr leicht zu beherrschen.

Auf dieser Weise lassen sich auch Datentreiber, wie sie in fast allen gängigen Testwerkzeugen zur Verfügung stehen, bereits auf Modellebene einbinden. Als Datentreiber wird ein Objektknoten verwendet, der als Data Store spezialisiert ist. Damit lassen sich eine ganze Reihe Tests, die viele Eingabedaten benötigen, sehr elegant modellieren und sinnvoll in Testscripten für diverse Testwerkzeuge transformieren. Ein Beispiel dafür ist in zu sehen.

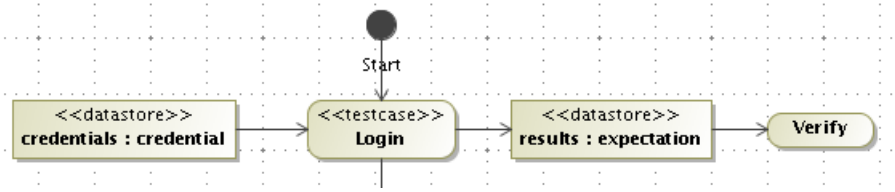

Abbildung 4.13: Anwendung von Data Strores als Datentreiber

Quelle: Eigene Darstellung

Es werden in diesem Beispiel zwei Data Stores verwendet. Für die Eingabedaten der Data Store `credentials`, der Objekte der Klasse `credential` enthält und ein zweiter für die Testresultate. Die Verifikation der Tests erfolgt aus dem Resultatespeicher und kann parallel zur weiteren Ausführung anderer Testfälle erfolgen.

Dem Anwender wird hiermit eine Modellierungsmöglichkeit angeboten, die weit über die Fähigkeiten des UTP hinausgeht und es ermöglicht, Tests auch mit Personen zu modellieren, die mit Softwaretests und -entwicklung nicht vertraut sind.

5 Ausblick und Zusammenfassung

5.1 Aktivitäten in der modellgetriebenen Softwareentwicklung

Die vorliegende Arbeit verfolgte das Ziel einen adäquaten, einfachen und dabei breit anwendbaren Ansatz zur Beschreibung von Verhalten durch die Integration von Aktivitäten in die modellgetriebene Softwareentwicklung zu schaffen.

Die in Kapitel zwei erläuterten Eigenschaften von Aktivitäten machen deutlich, dass sie sich sehr gut zum Einsatz in der modellgetriebenen Softwareentwicklung eignen. Die Aktivitäten in der UML sind mittlerweile stark formalisiert. Damit ist eine eindeutige Interpretation möglich, was wiederum eine Transformation in andere Modelle oder auch in Anwendungsquellcode erlaubt.

Die Möglichkeit, Aktivitäten auf ganz unterschiedlichen Ebenen einzusetzen, führt zu einer breiten Palette an Verwendungen in Softwaregeneratoren. Aktivitäten, die die einzelnen Schritte eines Anwendungsfalles spezifizieren, eignen sich sehr gut als Grundlage für die Verknüpfung von Seiten einer Webanwendung oder für die Steuerung des Kontrollflusses in anderen Benutzeroberflächen. Zudem können Aktivitäten auch sehr gut zur plattformunabhängigen Beschreibung von Algorithmen verwendet werden.

Gleichzeitig besitzen Aktivitäten in ihrer graphischen Notation den Vorteil der leichten Verständlichkeit für Personen, die nicht mit Softwareentwicklung vertraut sind.

Während die UML Aktivitäten so formal spezifiziert, dass Modelle in virtuellen Maschinen ablaufen können, ist die Integration in GeneSEZ deutlich einfacher gehalten. Hier liegt der Fokus ausdrücklich darauf, die Fähigkeiten von Aktivitäten möglichst einfach für die generative Softwareentwicklung verfügbar zu machen. Die damit einhergehende Reduzierung der Anzahl der Metamodellklassen für Aktivitäten erlaubt keine Ausführung in virtuellen Maschinen und stellt durch den Verzicht auf Objektknoten auch keine Möglichkeit mehr zur Verfügung, die Konzepte von Petri-Netzen anzuwenden.

Es konnte jedoch gezeigt werden, dass dieses minimale Aktivitätenmodell für den Zweck einer Generierung mit GeneSEZ ist ausreichend und aus Sicht des Anwenders sogar eine deutlich höhere Freiheit als die Aktivitäten der UML erlaubt. Die Spezialisierung von Aktionen durch Stereotypen, wie sie in GeneSEZ realisiert ist, bietet deutlich größere Einsatzmöglichkeiten.

Natürlich besteht auch die Möglichkeit, das Metamodell der Aktivitäten auszubauen, sollte sich dieses als notwendig erweisen. Ein Aspekt, der eine solche Erweiterung nötig machen könnte, ist die Integration von Structured Activities. Diese werden in der aktuellen Version nicht transformiert. Eine Unterstützung auf Metamodellebene würde bedeuten, dass zusätzliche Klassen zur Beschreibung von Structured Activities im Metamodell vorhanden sein müssten. Da Structured Activities ein sehr junges Konzept sind und in UML Werkzeugen noch nicht einheitlich unterstützt werden, bleibt zunächst abzuwarten, welche Bedeutung sie erlangen werden. Zudem liefe eine explizite Unterstützung von Structured Activities im Metamodell dem Ziel der Einfachheit entgegen.

Eine zweite Variante für die Integration von Structured Activities in GeneSEZ wäre die Zerlegung in einfache Aktivitäten. Dazu müssten Structured Activity Nodes in CallBehaviorActions transformiert werden und die eigentliche Structured Activity in eine normale Aktivität, in der Kontrollknoten die Verzweigung der einzelnen Abschnitte der Structured Activity steuern.

Ebenfalls nicht berücksichtigt sind Partitionen von Aktivitäten. Diese erlauben es, innerhalb einer Aktivität Teile des Verhaltens unterschiedlichen Elementen wie Klassen und Anwendungsfällen zuzuordnen. Knoten gehören dann nicht mehr nur zu einer Aktivität und eventuell implizit zu einem solchen Modellelement. Sie müssten explizit mit einer Klasse oder einem Anwendungsfall verbunden sein, was erneut zu einer Komplexitätserhöhung im GeneSEZ Metamodell führen würde.

Andererseits besteht auch bei Partitionen die Möglichkeit, sie während der Transformation von UML ins GeneSEZ Metamodell auf einfache Aktivitäten zurückzuführen. Dieser Vorgang muss aber formal sehr gut abgesichert werden und ist auch in Verbindung mit anderen komplexen Elementen der Aktivitäten nicht trivial.

5.2 Einsatz von Aktivitäten für modellbasierte Softwaretests

Die Implementierung des GeneSEZ Testing Metamodells zeigt, dass Aktivitäten sehr gut als Basis für Softwaretests dienen können. Die meisten Testwerkzeuge arbeiten ablauforientiert, sind also

prinzipiell eher an Aktivitäten als an anderen Verhaltensbeschreibungen orientiert. Zudem bieten Aktivitäten eine ideale Grundlage für die Beschreibung von Softwaretests auf allen technischen Ebenen. Die meisten vorgestellten Verfahren zur Testfallermittlung, die zum modellbasierten Testen eingesetzt werden können, erzeugen im Grunde die simpleste Form von Aktivitäten, nämlich einfache Abläufe ohne Verzweigungen.

Dies trifft sowohl auf die Verfahren zu, die Aktivitäten als Eingaben nutzen, wie die Anweisungs- und Zweiüberdeckung als auch auf die zustandsbasierten Testfallermittlungen, bei denen sämtliche möglichen Transitionsfolgen von Zustandsautomaten als Abläufe generiert werden.

Dies stellt auch einen der nächsten Schritte dar, der GeneSEZ in Hinsicht auf Testgenerierung komplettieren wird: Aus Aktivitäten und Zustandsautomaten müssen während der Transformation vom GeneSEZ Metamodell auf das Testing Metamodell mit den formalen Verfahren Testfälle generiert werden. Damit kann der Wert von Softwaremodellen zusätzlich gesteigert werden.

Eine andere Richtung, in der sich GeneSEZ weiterentwicklen muss, ist die Unterstützung weiterer Testwerkzeuge. Das im Rahmen der Arbeit benutzte QF-Test ist eine kommerzielle Anwendung, die Java-Anwendungen mit Swing- oder SWT-Benutzeroberflächen testen kann. Für Webanwendungen sollte ein Testframework für Selenium unterstützt werden und für die Generierung von Unit Tests können aufbauen auf den bisher existierenden GeneSEZ Plattformen für .NET, Java und PHP die entsprechenden Unit-Test-Frameworks NUnit, JUnit und PHPUnit unterstützt werden.

Da das GeneSEZ Testing Metamodell vollständig auf dem UTP basiert, besteht natürlich auch die Möglichkeit einer Weiterentwicklung in Richtung von TTCN-3, SOA und anderen Bereichen, die die Entwickler von UTP im Fokus hatten.

5.3 Weitere Entwicklungsmöglichkeiten

Die Integration von Aktivitäten in GeneSEZ bietet zahlreiche Möglichkeiten zum Ausbau des Generatorframeworks. Zunächst können Aspekte der Aktivitäten, die im Rahmen dieser Arbeit nicht weiter untersucht worden sind, ergänzt werden. Das betrifft die Partitionierung von Aktivitäten, eine vollständige Unterstützung von Structured Activities und den Einsatz von Aktivitäten über die Generierung von Softwaretests hinaus.

Insbesondere letzterer Anwendungsfall für Aktivitäten ist im Rahmen des GeneSEZ Frameworks interessant. Denkbar ist hier eine Verwendung innerhalb der PHP-Plattform für das GeneSEZ PHP

Metaframework. Dieses verknüpft Plugins zu vollständigen Applikationen. Diese Verknüpfung wird momentan nur strukturell beschrieben und könnte durch die Verwendung von Aktivitäten deutlich erweitert und verbessert werden.

Es ist weiterhin denkbar, die bereits in 3.2.3 angesprochene Integration von OCL in GeneSEZ zur Aufwertung der MGuards voranzutreiben.

Ein anderer wichtiger Aspekt, der weiterverfolgt werden sollte, die die parallele Generierung von Tests und Software mit dem Ziel Komponenten der Software, insbesondere in Benutzerschnittstellen, innerhalb der Tests ohne viel Aufwand zu referenzieren. Dies ist eine Problematik, die nicht nur die modellbasierte Generierung von Tests betrifft, sondern auch alle Hersteller von Werkzeugen für GUI-basierte Tests. Dort werden zumeist Heuristiken angewendet, um Komponenten über verschiedene Versionen der Software hinweg wiedererkennen zu können. Das ist im Rahmen der modellgetriebenen Software- und auch Testentwicklung unbefriedigend und stellt ein sehr spannendes Aufgabenfeld für weitere Arbeiten dar.

Literaturverzeichnis

BOC04: Conrad Bock, UML2: Aktivitäten und Aktionen, 2004

BOC03B: Conrad Bock, UML 2 Activity and Action Model: Actions, 2003

BOC03C: Conrad Bock, UML 2 Activities and Actions: Control Nodes, 2003

BOC03D: Conrad Bock, UML 2 Activity and Action Models: Object Nodes, 2004

BOC04E: Conrad Bock, UML 2 Activity and Action Models: Partitions, 2004

BOC04F: Conrad Bock, UML 2 Activities and Action Models: Structured Activities, 2004

OMG03: Object Management Group, Inc., Unified Modeling Language Specification, 2003

OMG09: Object Management Group, Inc., Unified Modeling Language Superstructure, 2009

RJB05: James Rumbaugh, Ivar Jacobson, Grady Booch, The Unified Modeling Language Reference Manual, 2005

SCH08: Schieferdecker et al., Model-Driven Testing, 2008

SPI05: Spillner et al., Basiswissen Softwaretest, 2005

STO05: Harald Störrle, UML 2 für Studenten, 2005

UTP05: Object Management Group, UML Testing Profile, 2005

WEI06: Tim Weilkiens, Systems Engineering mit der SysML/UML, 2006

GUIDAN: Bredex GmbH, GUIdancer, http://www.guidancer.com/

JUNIT: JUnit, , http://www.junit.org/

MMUNIT: MMUnit, http://mmunit.sourceforge.net/

NUNIT: NUnit, http://www.nunit.org/

OAWT: openArchitectureWare-Test, http://www.mdtd.de/

TOSCA: TRICENTIS Technology & Consulting GmbH, Tosca Testsuite, http://www.tricentis.com/

TTCN3: Test & Test Control Notation, http://www.ttcn-3.org/

Thesen

1. Aktivitäten, wie sie mit der UML modelliert werden können, können viel zur Effizienz von modellgetriebener Softwareentwicklung beitragen, sowohl was technische Realisierungen als auch die Kommunikation mit Personen, die nicht in der Softwareentwicklung tätig sind, angeht.
2. Das Metamodell der UML für Aktivitäten ist sehr umfangreich und soll vielerlei Aspekte abdecken, die im Rahmen der modellgetriebenen Softwareentwicklung nicht benötigt werden. Diese können in für eben jenen Zweck optimieren Metamodellen ersatzlos entfernt werden.
3. Aktivitäten können von einer eleganten Beschreibung von Anwendungsfällen bis hin zu einer abstrakten graphischen Programmierung innerhalb eines Modells eingesetzt werden und erlauben es damit, Verhalten auf diversen Ebenen zu modellieren.
4. Aktivitäten eignen sich hervorragend zur automatisierten Testfallfindung. Zum Einen als Quelle in Form von Kontrollflussgraphen, zum Anderen als Ziel wie der Testfallgenerierung aus Zustandsautomaten.
5. Die meisten Testwerkzeuge im Bereich von Akzeptanzt- wie auch Modultests realisieren Testfälle in einer Form, die weitestgehend Aktivitäten entspricht. Daher sind diese ein wichtiges Instrument, wenn es darum geht, aus Modellen heraus Testfälle zu generieren.

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Diplomarbeit eigenständig und nur unter Verwendung der genannten Literatur und den aufgeführten Hilfsmitteln verfasst habe.

Gerrit Beine