Philipp Graf

Entwurf eingebetteter Systeme: Ausführbare Modelle und Fehlersuche

Institut für Technik der Informationsverarbeitung
Universität Karlsruhe

Entwurf eingebetteter Systeme: Ausführbare Modelle und Fehlersuche

Zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs

von der Fakultät für Elektrotechnik und Informationstechnik
der Universität Fridericiana Karlsruhe
genehmigte

Dissertation

von

Dipl.-Ing. Philipp Graf
geb. in: Freiburg im Breisgau

Tag der mündlichen Prüfung:	16. Juli 2008
Referent:	Prof. Dr.-Ing. Klaus D. Müller-Glaser
Koreferent:	Prof. Dr. rer. nat. Ralf H. Reussner

Kurzfassung

Modellgetriebene Entwicklung gewinnt mit der steigenden Komplexität technischer Systeme eine wachsende Bedeutung. Wurden graphische Modelle ursprünglich in erster Linie zu Dokumentationszwecken eingesetzt, werden Modellierungssprachen bei der Beschreibung von Software semantisch zunehmend zu Programmiersprachen präzisiert.

Diese Arbeit geht von der Frage aus, welche Anforderungen eine flexible Entwicklungsumgebung für ausführbare Modelle erfüllen muss. Betrachtet werden die *Unified Modeling Language (UML)*, im Bereich eingebetteter Software verbreitete CASE-Werkzeuge und domänenspezifische Modellierungsansätze.

Alle Ansätze wurden in ein Gesamtmodell integriert. Dieses kann über eine integrierte Werkzeugkette ausführbar gemacht werden. Eine Lücke bei der vollständigen Verhaltensbeschreibung in der UML wurde durch eine Modell-zu-Code-Transformationen aus *UML Actions* geschlossen.

Die Fehlersuche mit Hilfe eines den Entwickler unterstützenden Debuggers ist ein zentrales Element während der Software-Entwicklung. Diese Arbeit beschreibt eine flexible Architektur zum Debugging ausführbarer graphischer Modelle. Diese geht von einer Reversierung der bei der Modell-zu-Executable-Transformation durchlaufenen Abstraktionsschichten aus.

Die Softwareplattform *ModelScope* ermöglicht die Anbindung verschiedenster Ausführungsplattformen und die Visualisierung des Systemzustands mit unterschiedlichen Sichten. Beide Heterogenitäts-Achsen wurden exemplarisch für einige beim Entwurf eingebetteter Software/Configware wichtige Zielsysteme und Modellierungsparadigmen realisiert.

Schlagworte: Debugging, Modellgetriebene Entwicklung, Ausführbare Modelle, Eingebettete Systeme, Metamodellierung

Abstract

With increasing complexity of technical systems, model-driven development gains increasing significance. In the past, graphical models were mainly used for documentation purposes. Today the semantics of modeling languages are specified in a way that allows to use them as programming languages to describe software.

This thesis evaluates the question which requirements must be fulfilled by a flexible development environment for executable models. It focuses on the *Unified Modeling Language (UML)*, CASE-tools that are common for embedded software development and domain specific modeling approaches.

All approaches were combined into an integrated model. Using an integrated tool-chain this model can be rendered to an executable binary. A model-to-code tranformation closes a gap for UML models by providing a code generator for *UML Actions*.

Searching for defects is an important task a developer has to fulfil during software development. This thesis describes a flexible architecture for debugging executable graphical models. It is based on reversing through the abstraction layers that were traversed during model-to-executable transformations.

The flexible software-platform *ModelScope* allows to integrate diffent types of execution platforms and visualizations of system state using different views on the running system. Both axes of heterogeneity were evaluated and implemented for a number of target systems and modeling paradigms that are important in the area of embedded software/configware development.

Keywords: Debugging, Model-driven Development, Executable Models, Embedded Systems, Metamodeling

Vorwort

Die vorliegende Dissertation entstand während meiner Tätigkeit am Institut für Technik der Informationsverarbeitung der Universität Karlsruhe (TH) und wurde von Herrn Prof. Dr.-Ing. Klaus D. Müller-Glaser betreut. Ihm verdanke ich ein spannendes Promotionsthema, die Förderung durch wichtige Anregungen sowie ein angenehmes Arbeitsklima mit vielen Freiheiten bei der Ausgestaltung meiner wissenschaftlichen Arbeit.

Mein Dank gilt ebenso Herrn Prof. Dr. rer. nat. Ralf H. Reussner für die spontane Übernahme des Korreferats meiner Arbeit und die damit verbundene Unterstützung, sowie die interessante und hilfreiche Diskussion der Inhalte.

Dankbar bin ich auch allen Mitarbeitern des ITIV. Ohne den immer wieder neu motivierenden Austausch, die Unterstützung in gemeinsamen Projekten und bei der „täglichen Arbeit“ als wissenschaftlicher Mitarbeiter und ein Umfeld in dem Kollegen vielfach zu Freunden wurden, wäre das Gelingen dieser Arbeit nicht möglich gewesen. Weiterhin danke ich allen „meinen“ Studenten, welche mit ihren Arbeiten wertvolle Beiträge für die Plattform *ModelScope* lieferten.

Mein besonderer Dank gilt nicht zuletzt allen Menschen in meinem persönlichen Umfeld, die mir während meiner Promotion ein ständiger Rückhalt waren und mich begleitet haben. Durch die Unterstützung und Motivation haben meine Familie und meine Freunde wesentlich zum Gelingen dieser Arbeit beigetragen.

Karlsruhe, Januar 2009

Philipp Graf

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

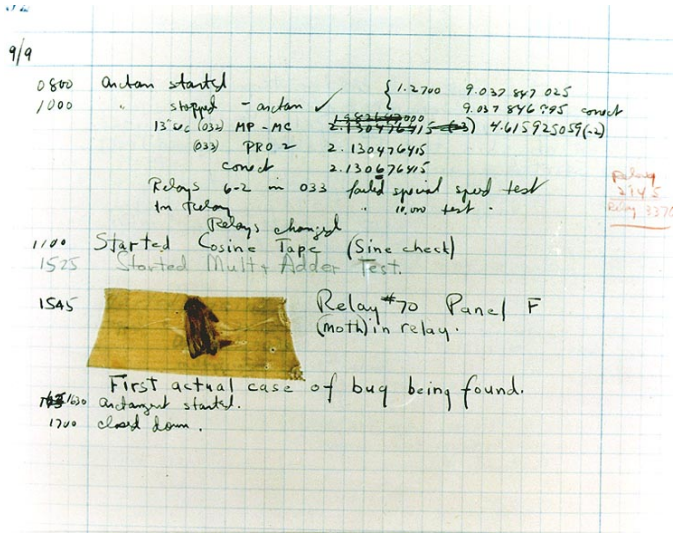

Abbildung 1.1: Möglicherweise der erste „Computer Bug“ [Naval Historical Centre 1988]

Als die Ingenieurin Grace Hopper 1946 an der Universität Harvard an den Rechnern Mark II (siehe [Danis 1997]) und Mark III arbeitete, verfolgte sie eine Fehlfunktion des Computers zurück auf eine Motte, welche sich in einem Relais verfangen hatte. Diese Geschichte gilt heute häufig fälschlicherweise als Ursprung des englischen Begriffs *Bug* für Fehler in technischen Systemen. Vermutlich entfernte sie das Insekt mit einer Pinzette aus der Mechanik und

behob so die Fehlfunktion des Rechners. Es findet sich heute im Logbuch des Rechners – die entsprechende Seite zeigt Abbildung 1.1.

Seitdem hat sich der Aufbau technischer Systeme stark verändert: Zunehmend wurden mechanische Komponenten durch elektronische Komponenten ersetzt; seit einigen Jahren ist auch ein weiterer Übergang zu immer mehr als Software in elektronischen Standardbausteinen realisierter Funktionalität zu beobachten. Ein aus Elektronik und Software bestehendes Teilsystem wird als Eingebettetes System bezeichnet.

Die steigende Systemkomplexität kann am Beispiel Automobil aufgezeigt werden. Die Anzahl der vernetzten Steuergeräte ist in den letzten Jahren in Luxusausführungen auf ca. 80 angewachsen. Diese weisen ihrerseits eine immens wachsende Softwarekomplexität auf. So genügten vor einigen Jahren noch einige 10.000 Quelltextzeilen, heute sind bereits einige Millionen üblich.

Die steigende Systemkomplexität tritt zudem mit weiteren nichtfunktionalen Anforderungsparametern in Konkurrenz: kurze Produktzyklen, begrenzte Kosten, konstante Sicherheit und Qualität des Produkts sind betroffen und widersprechen sich teilweise. So zeigen häufig durch Softwarefehler bedingte Rückrufaktionen der Automobilindustrie, dass diese Problematik aktuell ist.

Zahlreiche Softwareprojekte laufen Gefahr zu scheitern. Eine Softwareentwicklung wird als erfolgreich bezeichnet, wenn sie das Produkt pünktlich bereitstellt und dieses die spezifizierten Eigenschaften aufweist. Diese Definition wird auch von der Standish Group in [Standish Group International 2004] angenommen. Sie konstatierte 2004 nur eine Erfolgsrate von 29%. 53% der Projekte waren beendet, aber über dem Budget, der Zeit oder unter den projektierten Eigenschaften. Die übrigen 18% scheiterten vollständig oder das Produkt wurde nie genutzt. Obwohl die Methodik der Befragung in einzelnen Arbeiten in Frage gestellt wurden [Glass 2006], ist die komplexitätsbedingte Qualitätsproblematik in der Softwareentwicklung auch heute präsent.

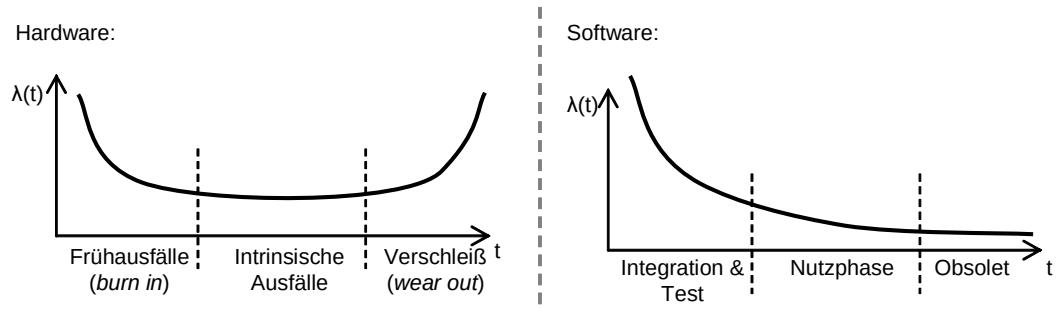

Abbildung 1.2: Fehlerraten von Hardware (links) und Software (rechts) im Lebenszyklus [Rosenberg u. a. 1998]

Ausfälle aufgrund von Softwaredefekten unterscheiden sich jedoch in einem Aspekt fundamental von allgemeinen Fehlfunktionen technischer Systeme. Abbildung 1.2 verdeutlicht den Unterschied: Der linke Bereich zeigt die bekannte „Badewannenkurve“, welche technischen Systemen eine verstärkte Ausfallwahrscheinlichkeit in der Frühphase ihrer Lebenszeit und durch Verschleiß zum Ende der Lebenszeit zuordnet. Da bei Software Verschleiß keine Rolle spielt, findet hier ein exponentieller Abfall der Fehlerrate über die Lebenszeit statt. Für Entwicklungsprozesse bedeutet diese Erkenntnis, dass Qualitätssteigerung für Software nur zur Entwurfszeit geschehen kann und durch Produktion oder Wartung nicht direkt beeinflusst werden kann.

Der steigenden Komplexität kann dadurch begegnet werden, indem der Abstraktionsgrad und damit die Mächtigkeit der verwendeten Beschreibungsmittel erhöht wird. Bei der Entwicklung von Software für eingebettete Systeme ist nach dem Übergang von der Programmierung mit Assemblersprachen zur Nutzung von Hochsprachen (z.B. C oder C++), ein weiterer Umbruch im Gang: Die Nutzung graphischer Modellierungssprachen vor allem der UML aber auch weiterer Ansätze wie Statecharts und Signalflussgraphen zur Systemspezifikation und die automatische Überführung des entworfenen Systems in ausführbaren Code.

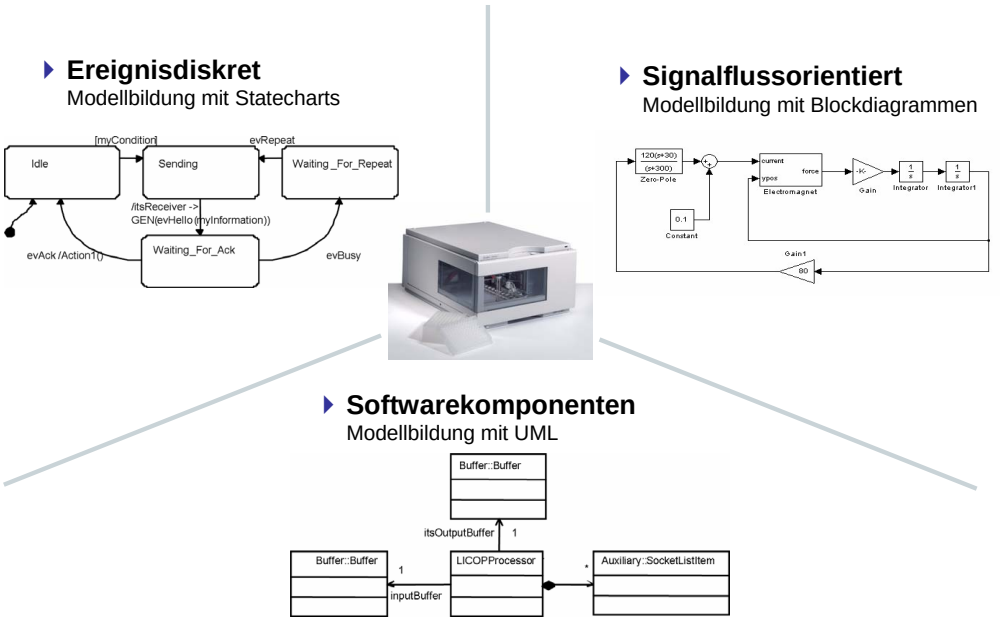

Abbildung 1.3: Wichtige Modellierungsparadigmen beim Entwurf eingebetteter Software

Abbildung 1.3 zeigt drei heute verbreitete Modellierungsparadigmen beim Entwurf eingebetteter Systeme: Modellierung ereignisdiskreter, reaktiv steuern-

der Systemteile mit hierarchischen Zustandsautomaten (Statecharts), Modellierung von (quasi-)zeitkontinuierlichen Signalverarbeitungs- und Regelungssystemen mit Blockdiagrammen und schließlich die systemweite Modellierung von Architektur und Softwarekomponenten mit der *Unified Modeling Language* (UML) [Object Management Group (OMG) 2005d].

Modelle dienen der Dokumentation und Evaluation von technischen Systemen, können diese jedoch auch vollständig in Struktur und Verhalten beschreiben. Solche Modelle werden als ausführbare Modelle bezeichnet. Sie können simuliert, validiert und direkt in Code der Ausführungsplattform übersetzt werden.

Die Object Management Group folgt diesem Ansatz mit ihrer *Model Driven Architecture* (MDA), welche ihre restlichen Standardisierungsbestrebungen umfasst. Dabei werden Modelle entsprechend ihrer Abstraktionsebene von der angestrebten Plattform angeordnet und durch Transformationen von abstrakteren zu realisierungsnäheren Ebenen verbunden.

In letzter Konsequenz bedeutet ein solches Vorgehen die Nutzung von (graphischen) Modellen als diagrammatische Programmiersprache. Von solchen Modellen wird in der vorliegenden Arbeit ausgegangen.

Beim Übergang von graphischen Modellen zu niedrigerer Abstraktion und schließlich in Quelltext wird zusätzliche Information hinzugefügt. Dies verdeutlicht Abbildung 1.4: Aus einem einfachen Zustandsautomaten entstehen bei Nutzung kommerzieller Codegeneratoren mehrere hundert Zeilen Quelltext.

Ein ausführbares Modell wird wie jedes Programm nicht fehlerfrei sein und Defekte enthalten. Um Fehler zu suchen, stehen heute auf Quelltext- und Signalebene ausgereifte Werkzeuge zur Verfügung welche dem Entwickler bei der Suche nach Fehlern, dem Debugging, behilflich sind.

Für ausführbare Modelle sind diese Werkzeuge jedoch nicht angemessen: Die Codeerzeugung fügt Information hinzu und verändert die Struktur des ursprünglichen Modells möglicherweise vollständig. Erschwerend wirkt sich aus, dass der Code nicht selbst geschrieben wurde und dem Entwickler daher unbekannt ist. In ein Softwareprojekt für eingebettete Systeme können zahlreiche Teilmodelle mit unterschiedlichen Transformationen und Zielplattformen einfließen.

[Lieberman 1997] weist provokant darauf hin: “Debugging is the dirty little secret of computer science”. Methoden und Werkzeuge zur Suche von Defekten in Software folgen neuen Entwurfs- und Programmierungsparadigmen stets mit einer gewissen Verzögerung. Trotzdem nimmt Debugging einen konstant großen Zeitraum bei der Entwicklungstätigkeit ein. Entwurfsprozesse und mächtigere Konstrukte können die Anzahl an Defekten reduzieren, sie jedoch

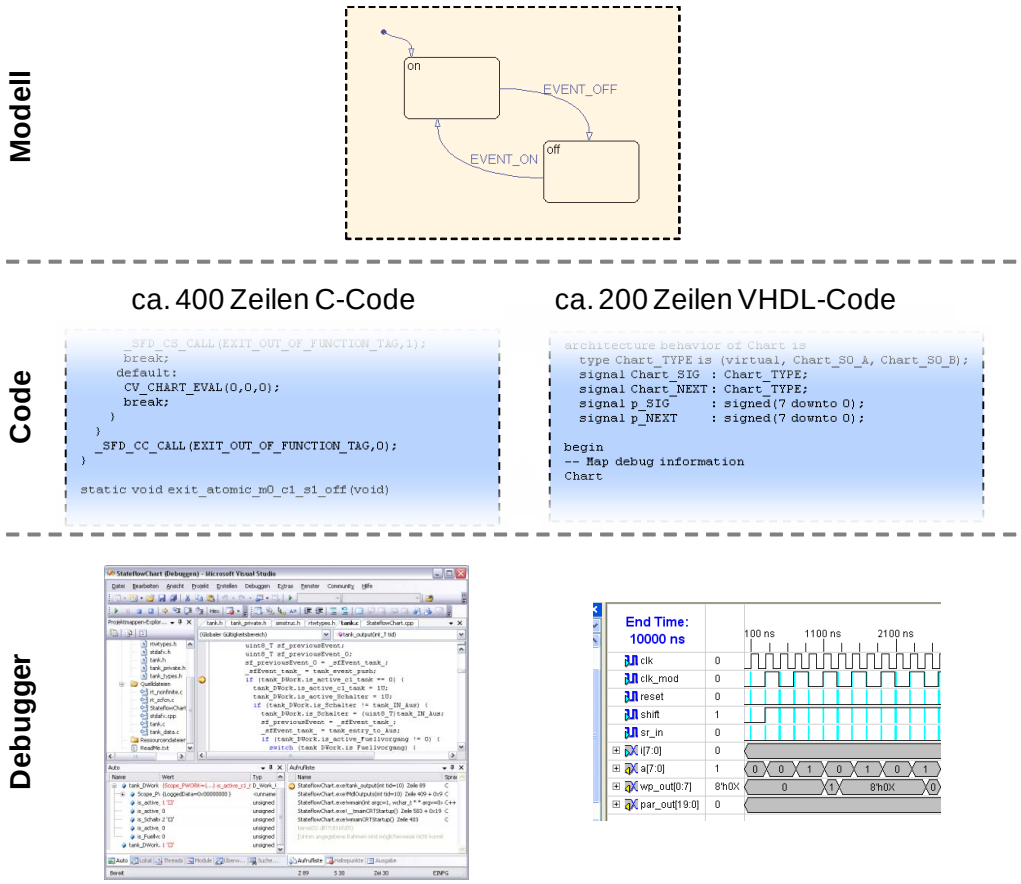

Abbildung 1.4: Transformation eines Zustandsautomaten in C- und VHDL-Code und Werkzeuge zum Debugging

nie vollständig ausschließen. Als Prozess, welcher ein Verständnis für das System benötigt, bleibt Debugging eine Aufgabe, welche einem menschlichen Entwickler nicht ganz abgenommen, nur erleichtert werden kann. Die Nutzung der eingangs erwähnten Pinzette wäre heute kein geeignetes Werkzeug zur Fehlersuche mehr.

Neben reinen Simulationsansätzen existieren bisher nur vereinzelte Werkzeuge, die den Prozess der Fehlersuche von der Ebene des Quelltexts auf die Modellierungsebene heben. Vorhandene Ansätze erlauben meist nur das Debugging auf spezialisierten Rapid-Prototyping-Systemen; ein Einblick in den Ablauf des Modells im Zielsystem oder im Rahmen einer Processor-in-the-Loop (PiL) / Hardware-in-the-Loop (HiL) Simulation ist nicht möglich.

Die Herausforderung liegt demnach in der Schaffung eines Ansatzes, welcher das Debugging von ausführbaren Modellen sowohl über die Grenzen einzelner Notationen als auch unterschiedlicher Ausführungseinheiten hinweg erlaubt.

1.2 Eigener Beitrag

Inhalt dieser Arbeit war die Zusammenführung dreier Bereiche. Sie bewegt sich am thematischen Schnittpunkt der Themengebiete eingebettete Software, ausführbare graphische Modelle und Fehlersuche mit Debugging-Werkzeugen. Eigene Beiträge sind dabei primär zwei Bereichen zuzuordnen:

1.2.1 Ausführbare Modelle

Um eine Durchgängigkeit vom Modell zum Code für eine große Anzahl von Modellierungsmitteln zu erreichen, wurden existierende Ansätze untersucht und eine fehlende Unterstützung für die Generierung von Quelltext aus Modellteilen identifiziert, welche auf Basis von UML Actions spezifiziert werden. Diese sind essentiell für die enge Verzahnung von Struktur und Verhalten in UML Modellen.

Daher entstand eine Abbildung von UML Actions Modellen auf Quelltext. Da UML Actions eng mit dem UML Gesamtmodell verzahnt sind, jedoch keine eigene Notation besitzen, wurde eine weitere Abbildung aus einer Java-ähnlichen Syntax auf Actions entwickelt.

Die Verwaltung und die Ausführung von Modellen wurde basierend auf der in Abschnitt 3.1.5 beschriebenen Plattform Aquintos.GS in eine eigene auf der Eclipse Plattform basierende Entwicklungsumgebung für Modelle mit dem Namen *ModelScope* integriert.

Um auch Durchgängigkeit und Konsistenz während der Transformation des Gesamtmodells in ein ausführbares System zu gewährleisten entstand eine datenflussorientierte Modellierung für Werkzeugketten. Eine modellierte Werkzeugkette ist in das Gesamtmodell integriert und kann selbst ausgeführt werden.

1.2.2 Debugging

Der Bereich des Debugging in ausführbaren Modellen ist, wie in den Abschnitten 1.1 und 4.5 herausgearbeitet, noch im Entstehen begriffen. Werkzeuge sind rar und nur auf Insellösungen zugeschnitten.

Im Rahmen dieser Arbeit entstand ein Konzept zum Debugging von ausführbaren Modellen. Dieses beruht auf der Reversierung der bei der Codeerzeugung durchlaufenen Abstraktionsebenen hin zu Laufzeitinformationen, welche dem Entwurfsmodell entsprechen. Dazu wurde eine Erweiterung des Modells konzi-

piert, welche dynamische Laufzeitdaten aufnehmen und mit dem zur Ausführungszeit statischen Entwurfsmodell verknüpft.

Dieses Konzept wurde in eine Softwarearchitektur umgesetzt, welche den Zustand graphischer Modelle zur Laufzeit darstellen und ihren Ablauf steuern kann. Einen zentralen Aspekt der Architektur stellt ihre Erweiterbarkeit und Flexibilität entlang mehrerer Achsen dar:

- Entwicklung mit unterschiedlichen domänenspezifischen Werkzeugen: Durch angepasste Systemsichten können dem Benutzer adäquate Einblicke in einzelne Subsysteme angeboten werden.
- Verschiedene Werkzeugketten und Zielsysteme: Durch Implementierung eines geeigneten Treibers können zahlreiche Zielplattformen und Werkzeugketten, welche unterschiedliche Debuggingschnittstellen auf Quelltextebene besitzen, angebunden werden. Modellbasiertes Hardware-/Software-Codesign kann durch geeignete Debug-Schnittstellen in Hardware und Erweiterung mit einem geeigneten Treiber durch Co-Debugging im Zielsystem unterstützt werden.

Die Leistungsfähigkeit des Ansatzes wurde durch Anwendung auf mehrere Zielsysteme und Systemsichten validiert. Die integrierten Zielsysteme umfassen Software in Desktop-Rechnern, eingebetteten Prozessoren und virtuellen Maschinen, sowie Configware auf rekonfigurierbaren Bausteinen. Dabei können mehrere Zielsysteme und Systemsichten in einer Debugging-Sitzung kombiniert werden.

Der Ansatz zum Debugging wurde in der Plattform *ModelScope* realisiert. Dabei wurde auf eine konsistente und schlanke Benutzerführung geachtet. Abbildung 1.5 zeigt die Benutzeroberfläche während ein Debugging-Sitzung mit einem ausführbaren hierarchischen Zustandsautomaten.

1.3 Gliederung der Arbeit

Das folgende **Kapitel** 2 beginnt mit einer Beschreibung der Grundlagen modellbasierter Entwicklung. Dabei wird nach Klärung grundlegender Begriffe das verwendete Konzept der Metamodellierung beschrieben. Das Kapitel schließt mit einer Beschreibung der im Umfeld eingebetteter Software wichtigen Modellierungssprachen mit Schwerpunkt auf der *Unified Modeling Language*.

Kapitel 3 befasst sich mit ausführbaren Modellen, welche durch eine vollständige Verhaltensspezifikation gekennzeichnet sind und ihrer Verwaltung und In-

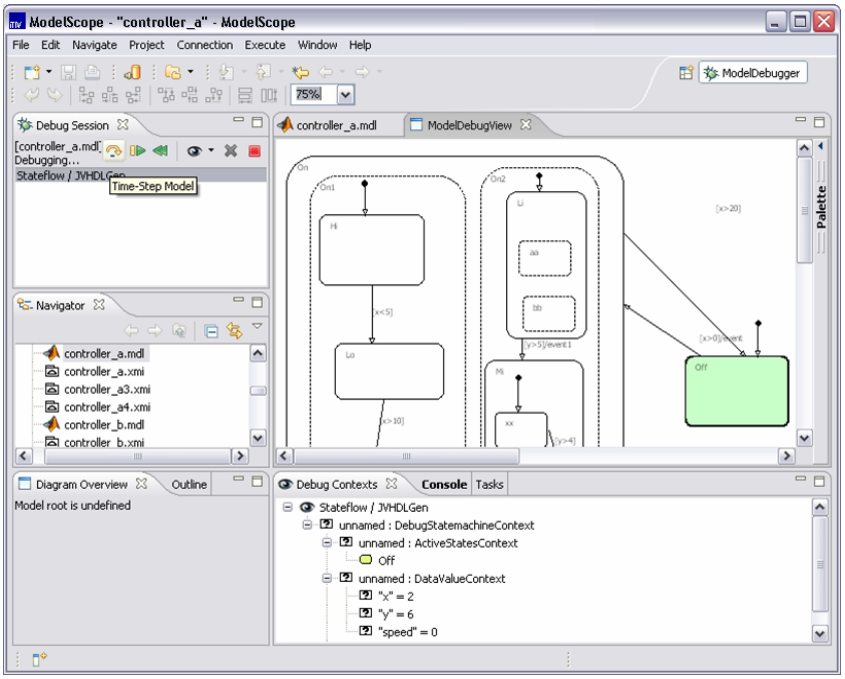

Abbildung 1.5: ModelScope

tegration. Zur Verhaltensspezifikation in UML wird eine Transformation von und aus Java vorgestellt.

In **Kapitel** 4 wird der Begriff des Debugging definiert und abgegrenzt. Weiterhin werden Methoden und Ansätze zum Debugging untersucht. Aspekte des Debugging eingebetteter Software und ausführbarer graphischer Modelle werden benannt.

Kapitel 5 stellt die Plattform *ModelScope* vor. *ModelScope* bietet ein Umfeld für ausführbare Modelle und stellt dabei eine Architektur zum Debugging in Modellen auf dem Zielsystem bereit. Das Kapitel behandelt die relevanten Teile der Plattform.

Unterschiedliche Ausführungsplattformen, welche in die Debug-Plattform integriert wurden, beschreibt **Kapitel** 6. Diese unterteilen sich in nichtechtzeitfähige und echtzeitfähige Debugger für Prozessoren und virtuelle Maschinen, sowie konfigurierbare Hardware.

Kapitel 7 untersucht drei realisierte graphische Systemsichten in *ModelScope* näher: Animation von *Statecharts*, eine Debug-Ansicht für UML *Actions* und eine Darstellung von Instanzen im Objektdiagramm.

Die Arbeit schließt in **Kapitel** 8 mit einer Zusammenfassung und dem Ausblick.

Kapitel 2

Modellgetriebene Entwicklung

Dieses Kapitel führt in die modellgetriebene Entwicklung ein. Es dient dazu, Grundbegriffe und -konzepte zu erläutern, soweit sie in dieser Arbeit verwendet wurden.

Modellgetriebene Entwicklung kann auf zahlreiche unterschiedliche Arten geschehen, welche in ihrer Gesamtheit zu beschreiben den Rahmen dieser Arbeit bei weitem sprengen würden. Das Kapitel legt daher einen starken Schwerpunkt auf für das weitere Verständnis relevante Ansätze und Methoden, grenzt diese aber wo möglich zu anderen Verfahren ab.

Nachdem im ersten Abschnitt grundlegende Begriffe und Verfahrensweisen erläutert wurden, wird in Abschnitt 2.2 detailliert auf Konzepte der Metamodellierung mitsamt der Technologien MOF, JMI, XMI und Diagrammaustausch eingegangen. Darauf aufbauend werden die in dieser Arbeit genutzten Modellierungsansätze beschrieben. Dabei liegt der Schwerpunkt insbesondere auf der *Unified Modeling Language* UML, es werden jedoch auch domänenspezifische Konzepte und die Modellierung mit *Computer Aided Software Engineering* (CASE)-Werkzeugen untersucht.

2.1 Begriffe und Grundlagen

Ein **Modell** ist eine abstrahierende Darstellung von Struktur, Funktion oder Verhalten eines betrachteten realen Systems. Es dient dazu, aus einem komplexen unübersichtlichen Sachverhalt wesentliche Aspekte zu extrahieren und in dieser vereinfachten Form handhabbar zu machen. Umgekehrt werden bei der Interpretation eines Modells seine Bestandteile auf Elemente des modellierten

Systems abgebildet. Diese Abbildung ist jedoch nicht surjektiv, es werden nur Teilaspekte des Systems modelliert.

Modelle können unterschiedliche Zwecke erfüllen. Fowler unterscheidet in [Fowler 2003] zwischen Modellen welche als Skizze, Bauplan oder Programm dienen. **Skizzen** werden genutzt um Aspekte des Systems anderen Menschen zu kommunizieren. Dies kann sowohl zur Entwicklung (*forward-engineering*) als auch zum Verständnis existierender Systeme (*reverse-engineering*) beitragen. **Baupläne** legen Wert auf Vollständigkeit, zumindest für einzelne Aspekte des Systems. Sie folgen einer klaren Sprachdefinition in Syntax und Semantik und dienen der Simulation und Verifikation. Werden Modelle als **Programme** genutzt, enthalten sie alle Informationen, um das modellierte System automatisiert zu erstellen.

Als **graphische Modelle** werden Modelle bezeichnet, welche eine nicht in Textform annotierte Syntax besitzen. Die meisten graphischen Modellierungssprachen ordnen den Elemente Knoten und Kanten in einem gerichteten Graphen zu. Andersartige Notationen, beispielsweise Baumstrukturen sind ebenso möglich.

Eine **Domäne** bezeichnet ein klar umrissenes technisches oder fachliches Wissens- oder Interessensgebiet. Eine Domäne nimmt die Perspektive des Kunden ein und benutzt dessen Terminologie zur Beschreibung des Problemraums.

Eine **Modellierungssprache** fasst eine Menge zulässiger Modelle zu einer Gesamtheit zusammen, mit deren Hilfe Modelle beschrieben werden können. Die Modellierungssprache kann eine oder mehrere Domänen abdecken (siehe Abschnitt 2.3.2)

Der Begriff **Plattform** beschreibt den Lösungsraum, also das Umfeld des Ergebnisses eines Entwurfprozesses. In einem Software-System können Plattformen unterschiedliche Hardwareplattformen, aber auch verwendete Betriebssysteme oder Rahmenwerke bedeuten.

Ein modellgetriebener Entwicklungsansatz versucht Domäne und Plattform mit Hilfe von **Transformationen** zu verknüpfen, welche Modelle in weitere Modelle umwandeln. Der Schritt der Veränderung eines Modells erfolgt meist in Richtung der Plattform, kann aber auch horizontal auf der gleichen Abstraktionsebene erfolgen (siehe dazu auch [Reichmann 2005]).

Die Abgrenzung zwischen Modellierungssprachen, Transformation und Plattform geschieht nicht zwingend im Voraus. Häufiger ist das in Abbildung 2.1 illustrierte Vorgehen. Ein System, hier ein Softwareprodukt wurde in der Vergangenheit ganz oder teilweise manuell in der Lösungsdomäne realisiert. Eine Analyse zeigt, dass diese Umsetzung teilweise aus generischem, immer gleichem Code besteht. Andere Bereiche sind individuell und benötigen individuelle Lö-

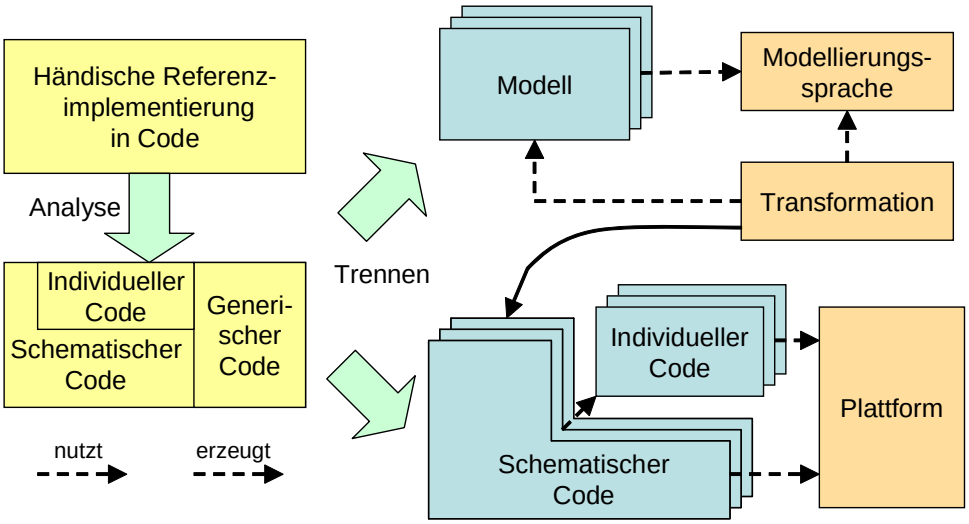

Abbildung 2.1: Grundkonzept modellgetriebener Softwareentwicklung (nach [Stahl u. Völter 2005])

sungen. Ein dritter Teil beinhaltet schematische Teile, welche zwar nie exakt gleich sind, aber Muster aufweisen.

Es werden nun diese drei Bereiche getrennt. Der generische Teil kann ab sofort über eine festgelegte Schnittstelle die Plattform bilden. Individueller Code wird weiterhin manuell umgesetzt.

Der große schematische Bereich kann jedoch in ein Modell in passender Modellierungssprache abstrahiert werden, welches nur noch die Unterschiede erfasst. Eine automatisierte Transformation erzeugt den schematischen Code. Mit Werkzeugen und Methoden zur Fehlersuche für diesen Bereich befasst sich die vorliegende Arbeit.

2.2 Metamodellierung

Metamodelle bilden die Grundlage der in Kapitel 3 beschriebenen Verhaltensmodelle und der im Rahmen dieser Arbeit entwickelten und in Kapitel 5 beschriebenen Plattform zur Modellintegration und zum Debugging von Modellen.

Als Metamodell bezeichnet man nach [Pomberger u. Rechenberg 1997] „die Konstruktionsregeln [...] für die Erstellung des Modellsystems; Metamodelle spezifizieren die verfügbaren Arten von Bausteinen (Meta-Objekte), die Bezie-

hungen zwischen den Bausteinen (Meta-Beziehungen) sowie Konsistenzbedingungen für die Verwendung von Bausteinen und Beziehungen".

Ein Metamodell fungiert als Grammatik einer Modellierungssprache und beschreibt die Modellartefakte und alle möglichen Verknüpfungen zwischen ihnen. Jedes Modell, welches in einer speziellen Modellierungssprache verfasst ist, muss den im Metamodell festgelegten Regeln genügen. Ein Modell stellt dabei eine Instanz des Metamodells dar. Diese Instanzbeziehung ist, wie in Abschnitt 2.2.1 beschrieben, über mehrere Stufen möglich.

Aufbauend darauf wird im folgenden Abschnitt auf die begriffliche Unterscheidung zwischen (abstrakter) Syntax, Semantik und Notation eingegangen und die von der *Object Management Group* (OMG)[1] definierte und in der vorliegenden Arbeit verwendete *Meta Object Facility* (MOF) als Meta-Metasprache zur Modellierung von Modellierungssprachen beschrieben.

Hauptvorteil der Verwendung eines Metadaten-Rahmenwerks wie der MOF ist die Möglichkeit zahlreiche Aspekte der Arbeit mit Modellen von der verwendeten Modellierungssprache zu abstrahieren und generisch zu lösen. In [Karagiannis u. Kühn 2002] werden diese Aspekte als *Mechanismen* bezeichnet und in *generische*, *spezifische* und *hybride Mechanismen* unterteilt. Beispiele für generische Mechanismen sind die Serialisierung von Modelldaten in XML-Dateien (XMI-Format, siehe [Object Management Group (OMG) 2005a]) oder in Datenbanken (siehe Abschnitt 3.1.5), Mechanismen zur Codeerzeugung oder die (halb-)automatische Erstellung von Benutzerschnittstellen zur Modellbearbeitung in Form von graphischen Editoren.

Um auch die Modellnotation in Form von Diagrammen unabhängig vom verwendeten Metamodell generisch in das Modell zu integrieren, wurde im Zuge der Standardisierung der UML 1.5 das *Diagram Interchange* Metamodell ([Object Management Group (OMG) 2005c]) definiert, welches in Abschnitt 2.2.6 gezeigt wird. Dieses kann in beliebige MOF-basierte Metamodelle integriert werden und wird in dieser Arbeit bei der graphischen Visualisierung des Modellzustands genutzt (siehe Abschnitt 5.3.4.3).

2.2.1 Die vierschichtige Hierarchie der Metamodellierung

Abbildung 2.2 verdeutlicht die vierschichtige Modellierungshierarchie, wie sie von der OMG definiert wurde.

Die Schichten sind zur einfacheren Benennung als M0 bis M3 bezeichnet. Die Abstraktion steigt dabei von unten nach oben an.

[1] Object Management Group Webseite: http://www.omg.org

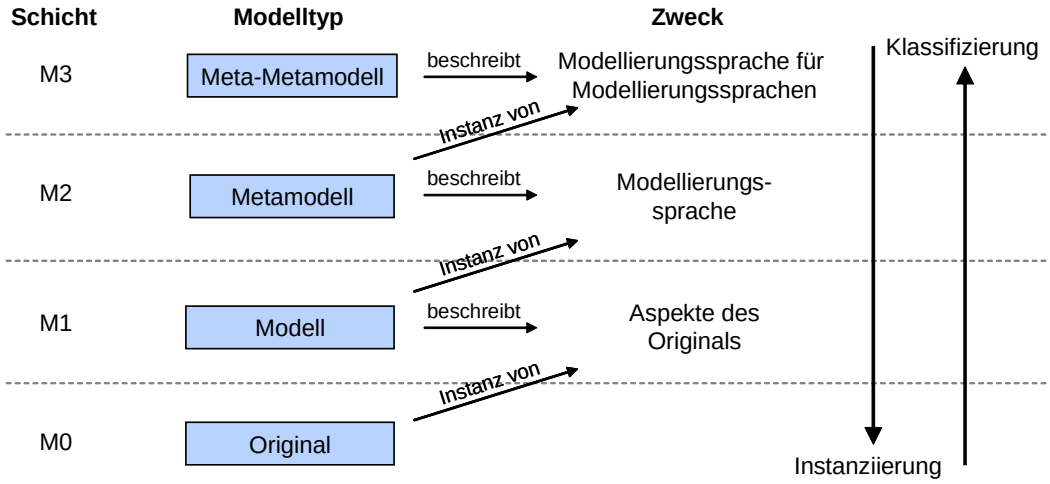

Abbildung 2.2: Metamodellierung basierend auf einer 4-schichtigen Hierarchie

Die Schichten sind im Einzelnen:

- **Originalschicht M0:** Die Laufzeitschicht bildet eine ganzheitliche Repräsentation der Realität. Im Zusammenhang mit der Modellierung technischer Systeme bezeichnet sie meist das Gesamtsystem oder bestimmte zur Laufzeit interessierende Aspekte und wird dann als *Laufzeitschicht* bezeichnet. Diese Aspekte sind weder klassifiziert, abstrahiert, noch anderweitig intellektuell erfasst.
- **Modellschicht M1:** Die Modellschicht M1 abstrahiert von einzelnen Entitäten der Originalschicht M0. Sie verallgemeinert (klassifiziert) dabei einzelne Aspekte der Realität zum Modell. Originale der Schicht M0 instanziieren durch ein Modell beschriebene Aspekte des Originals. Weiterhin ist ein Modell der Schicht M1 eine Instanz einer Modellierungssprache der Ebene M2. Objektorientierte Software kann durch UML-Modelle beschrieben werden [Object Management Group (OMG) 2007]. Regelungstechnische und Signalverarbeitungsaspekte können durch Signalflussmodelle, beispielsweise mit Matlab Simulink dargestellt werden (siehe Abschnitt 2.3.3.1). Statecharts modellieren reaktives ereignisgetriebenes Verhalten (siehe [Harel 1987] und [Object Management Group (OMG) 2007]).
- **Metamodellschicht M2:** Ein Metamodell ist ein Modell das eine spezielle Modellierungssprache beschreibt. Es definiert dabei folgende Aspekte:
  - Aus welchen Elementen kann ein Modell aufgebaut werden?

  - Welche Eigenschaften können diese Elemente aufweisen?
  - Wie können die Modellelemente miteinander in Beziehung stehen? Welche Rolle nehmen sie dabei ein und in welcher Vielfachheit können die Elemente als Instanzen des Metamodells verknüpft sein?

  Das Metamodell beschreibt einen Teil einer Modellierungssprache, nämlich die abstrakte Syntax aller möglichen Modelle, nicht jedoch Notation und Bedeutung (siehe auch Abschnitt 2.2.2). Die durch ein Metamodell spezifizierte Modellierungssprache der Ebene M2 kann durch Modelle auf Ebene M1 instanziiert werden. Das Metamodell selbst ist Instanz einer weiteren Modellierungsprache auf der darüberliegenden Ebene M3.

- **Meta-Metamodellschicht M3:** Ein Meta-Metamodell dient der Beschreibung von Metamodellen der Schicht M2. Es stellt also eine Modellierungssprache für die abstrakte Syntax von Modellierungssprachen dar. Diese Modellierungssprache für Modellierungssprachen kann durch Metamodelle der Schicht M2 instanziiert werden. Um die im vorigen Absatz angeführten Aspekte Elemente, Eigenschaften und Relationen eines Metamodells modellieren zu können, bietet sich eine an Klassendiagramme der UML angelehnte Modellierung an. Im Umfeld der UML ist die *Meta Object Facility* (MOF) der OMG gebräuchlich, welche in Abschnitt Abbildung 2.5 beschrieben wird. Die Elemente eines Metamodells werden in Anlehnung an das Klassendiagramm als Metaklassen, Metaattribute und Metaassoziationen bezeichnet.

In neueren Dokumenten rückt die OMG von einer starren vierstufigen Hierarchie ab und stellt sie in den Kontext eines flexibleren Ansatzes[2]. Die Metamodellierungshierarchie basiert jedoch immer auf zwei grundlegenden Eigenschaften ([Jeckle u. a. 2003]):

- Jede Schicht erfüllt den Zweck der Beschreibung der direkt unter ihr liegenden Schicht.
- Jede Schicht bildet eine Instanz der direkt über ihr liegenden Schicht.

Der Begriff der Instanz ist dabei als Übergang zwischen zwei Metaebenen zu verstehen und darf nicht mit der Instantiierung eines *Classifiers* zu einem Objekt in einem UML-Modell verwechselt werden. Dieser findet ausschließlich innerhalb der Modellschicht M1 statt und wird seit der UML 2.0 als *Snapshot* bezeichnet.

[2] vgl. [Object Management Group (OMG) 2005a], S. 8

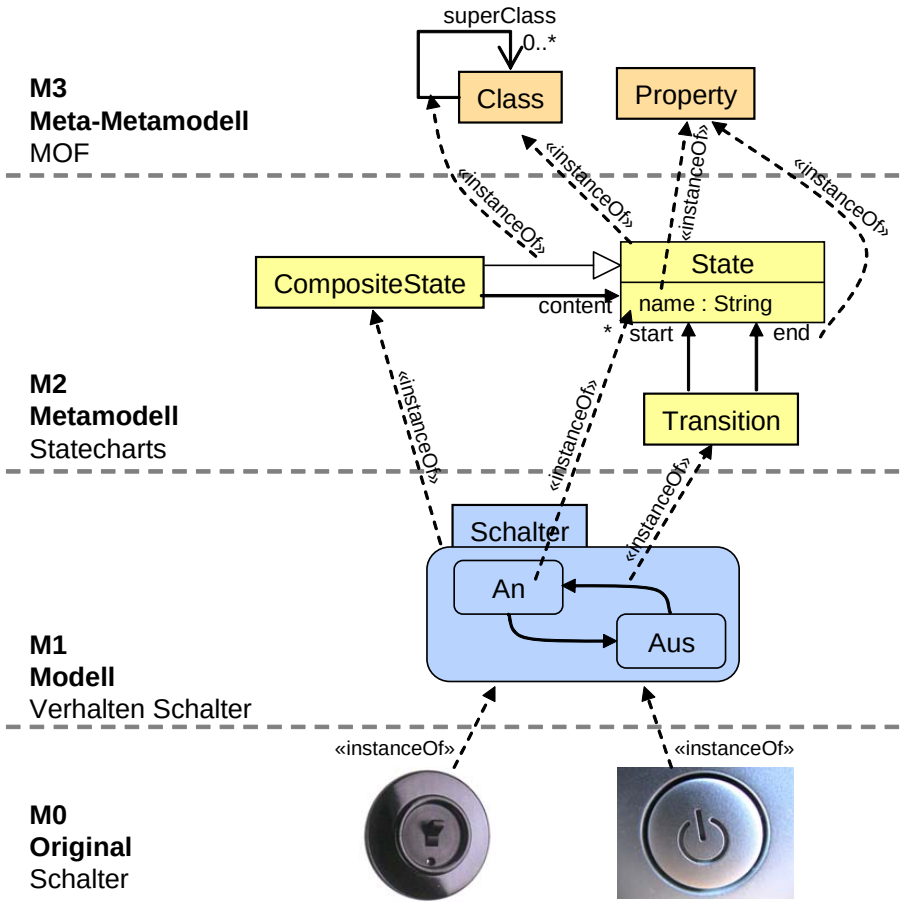

Abbildung 2.3: Beispiel für 4-Schichten-Metamodellierung -Artefakte und Instanziierungsbeziehungen

Abbildung 2.3 verdeutlicht die Zusammenhänge anhand eines Beispiels. Dabei handelt es sich um ein einfaches Beispiel für die Verhaltensmodellierung eines Schalters. Das Verhalten unterschiedlicher Schalter (Schicht M0) kann durch einen hierarchisch untergliederten Zustandsautomaten modelliert und damit abstrahiert werden. Dieses Modell befindet sich in der Schicht M1. Das Metamodell der Schicht M2 beschreibt die Struktur aller möglichen Zustandsmaschinen. Ein Modell besteht aus *States* welche Start- und Endpunkte für *Transitionen* darstellen. *CompositeStates* als Spezialisierung von *States* können eine beliebige Anzahl weiterer Zustände in sich aufnehmen. Modellelemente folgen den durch das Metamodell festgelegten Randbedingungen und instanziieren Elemente des Metamodells. Das Metamodell bildet die Instanz des Meta-Metamodells, welches im Fall der MOF rekursiv durch sich selbst spezifiziert ist.

2.2.2 Syntax, Semantik und Notation

Die **Syntax** behandelt die Muster und Regeln, nach denen Einheiten (in der Sprache Wörter, in Modellierungssprachen Modellierungselemente) zu größeren funktionellen Einheiten zusammengestellt und Beziehungen wie Teil-Ganzes, Abhängigkeit etc. zwischen diesen formuliert werden.

Die **konkrete Syntax** einer Modellierungssprache definiert, aus welchen Symbolen sie besteht und wie diese zusammengestellt werden dürfen. Eine konkrete Syntax kann sowohl die graphischen Elemente eines Diagramms bestimmen, aber auch die in einem in Textform vorliegenden Modell erlaubten Schlüsselworte definieren. Für das Beispiel in Abbildung 2.3 definiert die konkrete Syntax die Darstellung eines Zustands als abgerundetes Rechteck und einer Transition als Pfeil mit geschlossener Spitze. Synonym kann der Begriff **Notation** verwendet werden. Wird ein Modell basierend auf der für seine Modellierungssprache festgelegten konkreten Syntax beschrieben (Abbildung 2.3, Schicht M1), wird dies als **Darstellung in konkreter Syntax** bezeichnet.

Als **abstrakte Syntax** wird die Struktur einer Modellierungssprache definiert. Die abstrakte Syntax kann wie in Abschnitt 2.2.1 beschrieben durch ein Metamodell definiert werden. Sie beschreibt die nicht sichtbare Struktur und Randbedingungen und bedeutet im Beispiel der Zustandsmaschine, dass zusammengesetzte Zustände weitere Zustände enthalten können. Transitionen können strukturell keine Zustände enthalten.

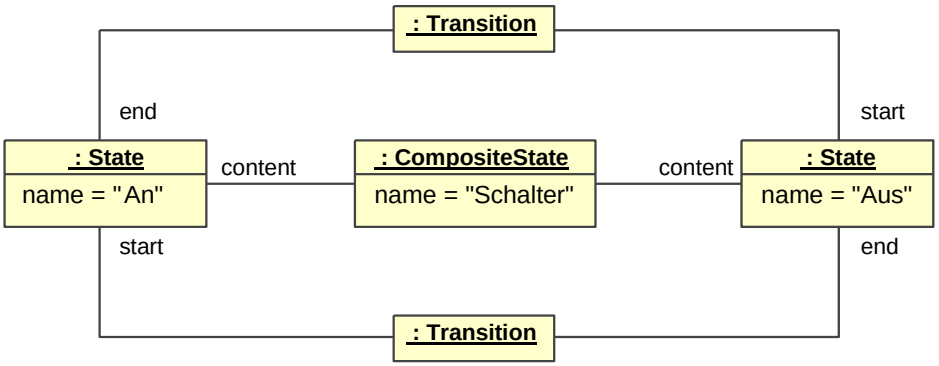

Abbildung 2.4: Darstellung einer Zustandsmaschine in abstrakter Syntax

Ein Modell kann ohne Nutzung seiner Notation auf Basis der abstrakten Syntax dargestellt werden. Da ein Modell eine Instanz eines Metamodells ist und die auf Basis der MOF notierten Metamodelle an eine Darstellung für Klassendiagramme angelehnt ist, kann das Modell als Objektdiagramm bestehend aus „Metaobjekten“, also Instanzen von Metaklassen, dargestellt werden. Eine solche Notation wird als **Darstellung in abstrakter Syntax** bezeichnet.

Abbildung 2.4 zeigt die Zustandsmaschine aus Abbildung 2.3 in abstrakter Syntax, basierend auf dem Metamodell der Schicht M2. Aus ihr sind alle strukturellen Informationen des Modells ersichtlich; Notation und räumliche Anordnung in einem Diagramm fehlen jedoch. Um die räumliche Anordnung dem Modell beizufügen, kann die in Abschnitt 2.2.6 beschriebene *Diagram Interchange* Spezifikation verwendet werden. Die Symbolik der Notation, also die konkrete Syntax, wird im Umfeld der OMG informell innerhalb des Standardisierungsdokumentes beschrieben.

Als **Semantik** einer Modellierungssprache bezeichnet man die *Bedeutung* der syntaktischen Sprachelemente, aus denen sich Modelle zusammensetzen, unter Berücksichtigung des jeweiligen Kontexts [Pomberger u. Rechenberg 1997]. Die Semantik einer Zustandsmaschine ist durch die zeitlichen reaktiven Abläufe des durch sie beschriebenen Systems gegeben.

Die Semantik von Modellierungssprachen wird innerhalb der UML nicht modelliert, sondern ebenso wie die Notation informell im Spezifikationsdokument beschrieben. Durch den Profilmechanismus der UML kann sich die Semantik von Modellelementen durch Anwendung eines Stereotyps gegenüber dem Standard bei gleicher Notation völlig ändern.

2.2.3 Meta-Object-Facility (MOF)

Die *Meta Object Facility* (kurz MOF) ist ein von der Object Management Group spezifiziertes Meta-Metamodell zur Metamodellierung. Sie ist in der Schicht M3 angesiedelt und dient der Modellierung von Modellierungssprachen der Schicht M2. Wie eine Modellierungssprache als Instantiierung aus der MOF hervorgeht zeigt der obere Teil in Abbildung 2.3.

Entwurfskriterien bei der Definition der MOF waren nach [Object Management Group (OMG) 2004] unter anderem:

- Einfache Verwendbarkeit: Das Metametamodell ist verhältnismäßig einfach und nutzt über den Paket-Importmechanismus der UML 2.0 gemeinsam mit der UML verschiedene grundlegende Pakete der *UML Infrastructure* (siehe Abschnitt 2.3.1 und [Object Management Group (OMG) 2007]). Ziel dieses Vorgehens ist es, dadurch die Metamodellierung der geläufigen Modellierung anzunähern und die Metamodellierung auf Basis von UML Werkzeugen zu vereinfachen.

- Modularität: Das MOF Meta-Metamodell ist in mehrere Pakete modularisiert, welche teilweise optional sind.

- Plattformunabhängigkeit: Die MOF bietet größtmögliche Orthogonalität zwischen Modellen und den Diensten, welche mit Modellen arbeiten. So ist auch die Repräsentation eines Modells mittels eines Java-Datenmodells (*Java Metadata Interface*, siehe Abschnitt 2.2.4) oder einer XML-Datei (*XML* Metadata Interchange, siehe Abschnitt 2.2.5) in gesonderten Standards definiert. Eine Abbildung der MOF auf eine Implementierungstechnologie wird als *Mapping* bezeichnet.

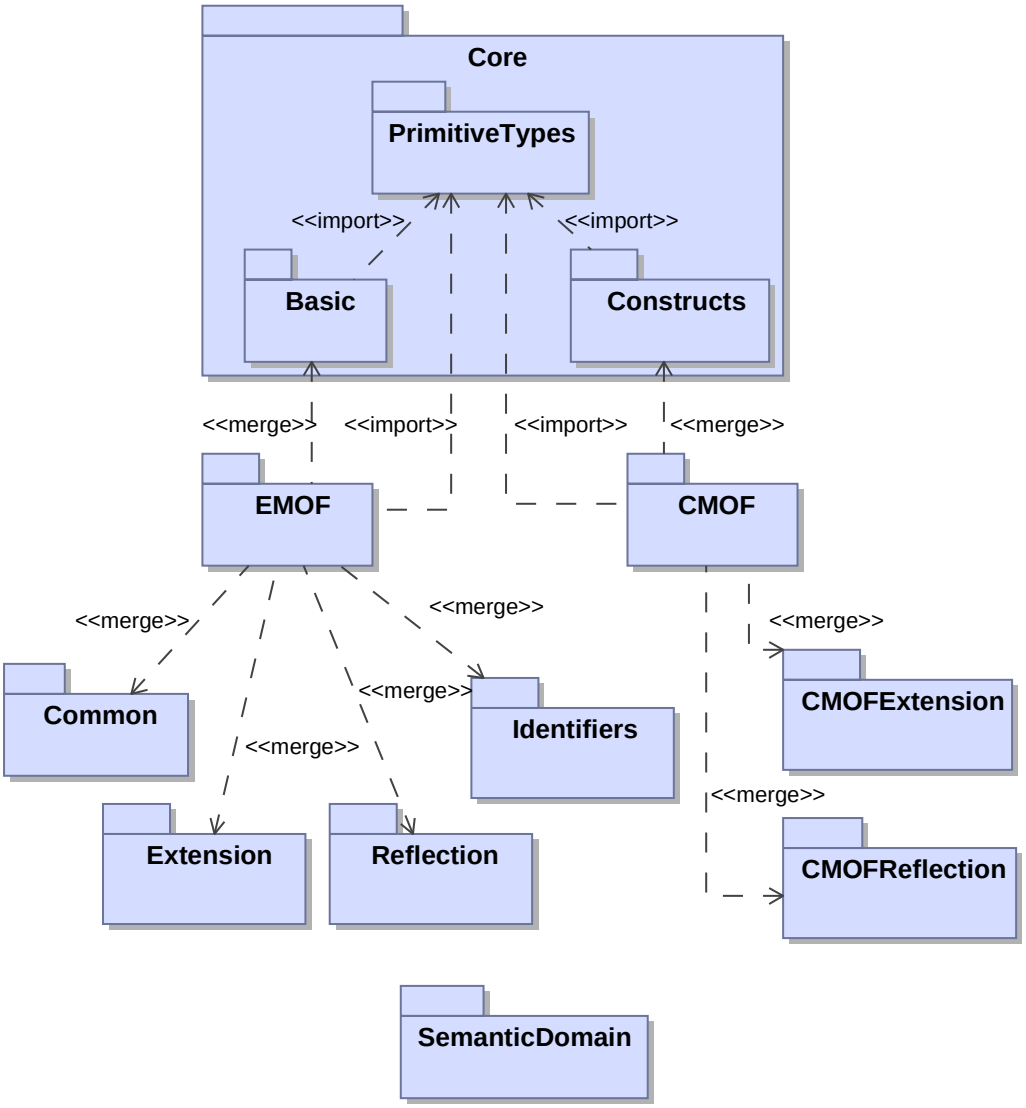

Abbildung 2.5: Paketstruktur des Meta-Metamodells der Meta Object Facility [Object Management Group (OMG) 2004]

Der Standard liegt aktuell in der Version 2.0 vor und wurde zur Beschreibung der UML 2.1.1 genutzt. Abbildung 2.5 zeigt die Paketstruktur des selbstbeschreibend durch die MOF notierten Meta-Metamodells der *Meta Object Facility*.

Grundlegend für die Definition der MOF ist die Nutzung der (Meta-)Metaklassen aus dem Paket *Core* der UML2. In diesem befinden sich:

- einfache Datentypen (*PrimitiveTypes*) wie z.B. ganzzahlige und boolesche Typen oder Zeichenketten.
- im *Basic*-Paket einfache Klassen (*Classifier*) mit Vererbung, Eigenschaften, Operationen, Modellierungsartefakten zur Definition komplexer Datentypen und Pakete zur Modularisierung von (Meta-)Modellen.
- im *Constructs*-Paket weitere für die MOF benötigten Konzepte zur Metamodellierung wie Assoziationen mitsamt Rollennamen und Multiplizitäten, Sichtbarkeiten und Namensräumen. Weiterhin enthält dieses Paket die Abhängigkeiten «import»[3] und «merge»[4].

Die MOF 2.0 beschreibt zwei unterschiedliche MOF-Standards. Die *Essential Meta-Object Facility* (EMOF) und aufbauend auf dieser die *Complete Meta-Object Facility* (CMOF).

Beiden gemein ist, dass sie die Kernkonstrukte der Klassenmodellierung um folgende wesentliche Punkte erweitern:

- **Reflektion:** Aus einem Modellelement kann zur zugehörigen Metaklasse navigiert werden. Die Ganzes-Teil-Hierarchie des Modells, welche immer einen Modellbaum aufspannt kann navigiert werden. Eigenschaften können über ihren Namen referenziert werden. Weiterhin existieren Fabriken

[3]Die «import»-Abhängigkeit bezeichnet das Einbinden von Modellierungsartefakten aus einem anderen Paket. Die importierten Metaklassen stehen in dem importierenden Paket unverändert zur Verfügung.

[4]Über die «merge»-Abhängigkeit kann eine Paketverschmelzung erreicht werden, welche die Inhalte zweier Pakete miteinander kombiniert und in dem Paket, welches am Ausgangspunkt des «merge»-Pfeils steht vereinigt. In beiden Paketen definierte Metaklassen werden vereinigt und die Vereinigungsmenge ihrer Operationen, Attribute und Assoziationen gebildet. Ein «merge» zweier Pakete kann demnach auch als eine implizite Generalisierungsbeziehung zwischen den in den Paketen enthaltenen Metaklassen verstanden werden und erlaubt eine aspektorientierte Definition des Metamodells unter Vermeidung einer tiefen Vererbungshierarchie. Nachteil ist jedoch eine erschwerte Übersicht über die Eigenschaften einer speziellen Metaklasse, wenn diese verteilt definiert wurde.

(*Factories*) welche entsprechend dem *Factory*-Entwurfsmuster Modellelemente basierend auf der Metaklasse oder ihrem Namen erzeugen können. Die CMOF ermöglicht zusätzlich die Aufzählung aller Instanzen einer Metaklasse.

- **Bezeichner:** Diese ermöglichen die Bezeichnung von Metamodellen über ihren *Uniform Resource Identifier* (URI) und die Anreicherung von Modellelementen mit eindeutigen Identifizierungsmerkmalen. Bezeichner ermöglichen das Verfolgen von Modellartefakten über mehrere Transformationsschritte hinweg.

- **Erweiterung:** Jedem Modellartefakt können beliebige Schlüssel-Wert-Paare (*Tags*) zugeordnet werden, welche Modelle unabhängig von der Modellierungssprache um werkzeugspezifische Eigenschaften, beispielsweise zur Versionierung, anreichern.

Die EMOF entspricht in ihren Fähigkeiten der für die UML bis zur Version 1.5 verwendeten MOF 1.4 [Object Management Group (OMG) 2001], während die CMOF weitere komplexere Fähigkeiten hinzufügt. Darunter fallen die aus dem *Constructs*-Paket importierten Eigenschaften und Erweiterungen des reflektiven Mechanismus, wie die Aufzählung aller Artefaktinstanzen und den Aufruf von Metaoperationen.

Als konkrete Syntax der MOF als Modellierungssprache für Modellierungssprachen dient in Anlehnung an den verwendeten Kern der UML die Modellierung von Klassendiagrammen. Ein Metamodell kann jedoch prinzipiell auch in abstrakter Syntax in Form einer Objektstruktur aus Meta-Metaklassen notiert werden.

2.2.4 Java Metadata Interface (JMI)

Das Java Metadata Interface (kurz *JMI*) ist eine Abbildung der Möglichkeiten der *Meta Object Facility* (siehe Abschnitt 2.2.3) auf die Programmiersprache Java und in [Dirckze 2002] spezifiziert.

Inhalt der Spezifikation ist ein Satz aus Java-Schnittstellen, welche ein mit der MOF modelliertes Metamodell widerspiegeln. Dieses Datenmodell trägt alle beschriebenen Eigenschaften eines MOF-konformen Metamodells. Es bildet den Aufbau des Metamodells mitsamt seiner Eigenschaften, Verknüpfungen und Vererbungshierarchien ab und ist darüber hinaus reflexiv, besitzt eindeutige Bezeichner und garantiert verschiedene Konsistenzeigenschaften des Datenmodells.

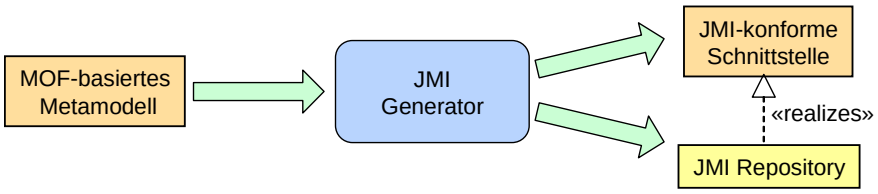

Abbildung 2.6: Abbildung von MOF auf Java

Abbildung 2.6 zeigt die Einordnung dieser Abbildung in eine tatsächliche Werkzeugkette. Die Überführung eines MOF-basierten Metamodells geschieht im Allgemeinen mit Hilfe eines Generators, welcher das MOF-Modell in seiner serialisierten Form aus einem XMI-Dokument liest und die zugehörigen Java-Quelldateien generiert. Diese können weiter compiliert und in eine Bibliothek zusammengefasst werden. Die JMI-Spezifikation beschreibt die Schnittstelle zum Zugriff auf Modelle und ihr nach außen zugesichertes Verhalten. Die Schnittstelle kann durch unterschiedliche *Repositories* implementiert werden. Ein Generator erzeugt neben den Schnittstellen im Allgemeinen immer auch die Implementierung, um das Datenmodell verwendbar zu machen.

Verbreitete Repositories sind das auf EMOF basierende *Eclipse Modeling Framework* EMF[5], *Metadata Repository* MDR[6] von Sun Microsystems und das im Rahmen der vorliegenden Arbeit modifiziert verwendete *Novosoft MetaData Framework* NSMDF[7].

Abbildung 2.7 verdeutlicht zentrale Aspekte der Übertragung von MOF-Elementen nach Java. Die Abbildung zeigt ein MOF-Modell in konkreter Syntax, welches aus dem Paket P1 und einem darin enthaltenen Paket P2 besteht. Das erste Paket P1 enthält die Metaklassen C1 und C2, wobei C2 eine Unterklasse von C1 darstellt. C1 und C2 sind über eine Metaassoziation miteinander verknüpft.

Der rechte Teil der Abbildung zeigt die Vererbungshierarchie der generierten Java-Klassen.

Die Wurzel des Vererbungsgraphen wird durch eine Gruppe vordefinierter Schnittstellen gebildet, welche die reflexiven Eigenschaften von MOF widerspiegeln. Gemeinsam bieten sie Operationen, welche Objektidentität implementieren und generischen Zugriff auf Modell und Metamodell unabhängig vom tatsächlichen Datenmodell erlauben.

Die im rechten unteren Teil gezeigten Klassen werden durch den Generator er-

[5]Projektseite http://www.eclipse.org/modeling/emf

[6]Projektseite http://mdr.netbeans.org

[7]Projektseite http://nsuml.sourceforge.net

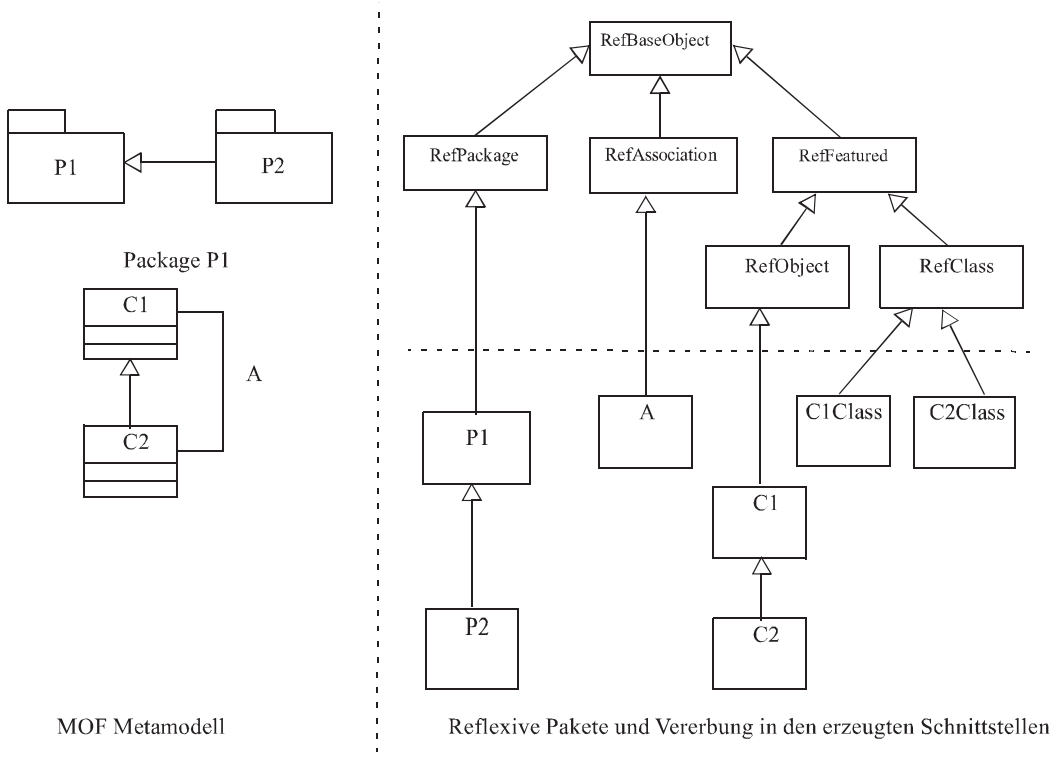

Abbildung 2.7: Übertragung von MOF-Elementen nach Java [Dirckze 2002]

zeugt und repräsentieren einzelne Artefakte des Metamodells. Diese erben von den untersten reflexiven Klassen und repräsentieren durch diese verschiedene Teile des Datenmodells:

- *RefPackage*: Ein Element dieses Typs ist ein Paket im Metamodell. Es stellt einen erweiterten Ordner dar, durch welchen zu weiteren Elementen im Metamodell navigiert werden kann. Ein Paket stellt auch die Wurzel des Metamodells, den *Extent* dar. Alle weiteren Instanzobjekte, Klassenproxies, Metaassoziationen und weitere Pakete sind rekursiv enthalten.
- *RefObjekt*: Ein Instanzobjekt speichert den Zustand einer konkreten Instanziierung einer Metaklasse. Es erlaubt den Zugriff auf Eigenschaften des Modellelements und die Navigation über seine Assoziationen im Metamodell zu weiteren Instanzobjekten. Weiterhin können diese Eigenschaften und Verknüpfungen verändert werden.
- *RefClass*: Modellartefakte als Instanzen von *RefObject* können nicht direkt instanziiert werden, sondern werden immer über Klassenproxies erzeugt. Es stellt also eine Fabrik für Instanzobjekte dar und verwaltet weiterhin statische Attribute und Operationen auf Metaklassenebene.

- *RefAssociation*: Ein Assoziationsobjekt enthält immer eine Sammlung von Verknüpfungen im Modell welche auf der zugehörigen Metaassoziation beruhen und verwaltet diese. Im Gegensatz zum Dualismus *RefObject/RefClass* existiert kein Datenobjekt für Verknüpfungen. Diese sind implizit in den beteiligten *RefObjects* abgelegt.

RefPackage, *RefClass* und *RefAssociation* dienen der Navigation im Metamodell. Instanzen von *RefObject* sind demnach im Gegensatz zu den übrigen Metaklassen Teil des Modells und erlauben die Arbeit mit diesem. Instanzobjekte sind immer an einen Klassenproxy gebunden und werden über seine Operationen erzeugt.

Neben der Definition der Metadatenhierarchie spezifiziert die JMI weitere semantische Randbedingungen für das Datenmodell, beispielsweise für die Gleichheit zweier Modell- und Metamodellobjekte und die Interpretation der NULL-Referenz in Java als undefinierte Eigenschaft im Modell. Weiterhin werden Multiplizitäten größer als Eins auf Instanzen von *java.util.Collection* und im sortierten Fall auf *java.util.List* abgebildet.

Operationen zum Zugriff auf Eigenschaften und Assoziationen der Metaklasse folgen einem festen Benennungsschema. Primitive Datentypen werden nach Java abgebildet.

Zentral ist auch die Spezifikation einer Semantik für den Lebenszyklus der Modell- und Metamodellobjekte, welche sicherstellt, dass das Modell jederzeit konsistent bleibt. So müssen beispielsweise beim Zerstören eines Modellobjektes auch alle Verknüpfungen zu weiteren Modellobjekten entfernt werden.

2.2.5 Modellserialisierung mit XMI

XMI, welches die Abkürzung für *Extensible Markup Language (XML) Metadata Interchange* ist, ist ein Standard zur Serialisierung objektorientierter Information auf Basis der XML. Die *Object Management Group* (OMG) verabschiedete XMI 1.0 im Januar 1999. Die aktuelle Version 2.1, welche auf MOF 2.0 (siehe Abschnitt 2.2.3) aufbaut, erschien im September 2005.

Sie stellt eine Abbildung der MOF auf ein XML-Format dar, indem Instanzen von MOF-Artefakten (also Modelle) in einer festgelegten Form abgelegt werden. Im Folgenden werden zunächst Kernaspekte der XML angerissen; Darauf aufbauend wird in Abschnitt 2.2.5.2 das XMI-Format selbst beschrieben.

Listing 2.1: Beispiel für ein XML-Dokument

```
<?xml version="1.0" encoding="iso-8859-1"?>

<Strasse xmlns="http://www.fzi.de/2002/Strasse">
  <Haus Nummer="10">
    <Farbe>rot</Farbe>
  </Haus>
  <Auto Kennzeichen="KA-RL-1234"/>
</Strasse>
```

2.2.5.1 Extensible Markup Language (XML)

Die in [Bray u. a. 1998] definierte Extensible Markup Language (XML) ist ein Werkzeug zur Schaffung, Gestaltung und Benutzung von Auszeichnungssprachen. Dabei wird ein syntaktischer Rahmen vorgegeben, in dem Strukturen zur Speicherung von Informationen festgelegt werden können. Die Grundidee von XML ist es, die Möglichkeit zu bieten, einem Dokument eine Struktur zu geben. Ein Beispiel für ein einfaches XML Dokument ist in Listing 2.1 gezeigt.

Für eine umfassende Behandlung von XML sei auf die Spezifikation und [Ray 2001] verwiesen. Im Rahmen dieser Arbeit sollen nur die für XMI bedeutenden Aspekte kurz vorgestellt werden. Ein Dokument besteht dabei aus folgenden Bestandteilen:

- Die in eckigen Klammern (auch *tags* genannt) eingefassten **Elemente** wie *Straße* oder *Auto* bilden das Gerüst des Dokuments. Jedes Element umfaßt den zwischen <*Element*> und </*Element*> stehenden Teil des Dokuments. Innerhalb eines Elements können sich weitere Elemente oder Text befinden. Das Dokument erhält dadurch eine hierarchische Gliederung. Leere Elemente, wie *Auto*, dürfen mit <*Element/*> abgekürzt werden.
- Jedes XML Dokument hat genau ein alles andere umfassendes **Wurzel-Element**. Im Beispiel ist dies *Strasse*.
- Einzelnen Elementen können ***Attribute*** zugeordnet werden. Dabei wird jedem Attribut ein Wert zugewiesen. Attribute ermöglichen es, die Eigenschaften eines Elements genauer zu beschreiben. Dabei müssen die Attribute eines Elements unterschiedliche Namen aufweisen. Im Beispiel in Listing 2.1 sind *Kennzeichen* und *Nummer* Attribute.
- In der ersten Zeile eines Dokuments steht die **XML-Deklaration**. Dabei bezeichnet das erste Attribut die Version und das zweite Attribut die verwendete Zeichenkodierung.

- **Kommentare** werden durch die Zeichenfolgen <!– und –> begrenzt und dürfen überall im Dokument erscheinen. Text und Markup in Form von Tags, die innerhalb der Kommentarbegrenzungen liegen, werden dabei vom Parser ignoriert.

Die Namen, Typen und möglichen Inhalte der Elemente selbst sind durch XML zunächst nicht vorgegeben. Das Dokument muss jedoch einige Syntaxbeschränkungen erfüllen, damit es als **wohlgeformt** bezeichnet werden kann. So muß zu jedem Start-Tag ein schließendes Tag gehören, oder das Tag selbst ist von der Form <*Element*/>. Attributwerte müssen immer in Anführungszeichen stehen und Elemente dürfen sich nicht überlappen, eine Anordnung

<A><B></A></B>

ist also nicht erlaubt. Außer als Markup-Begrenzungen, dürfen die Zeichen *'*, *"*, <, > und *&* nicht in ihrer literalen Form verwandt werden und müssen durch die Zeichenentities *'*, *"*, *<*, *>* und *&* ersetzt werden.

Ein wohl geformtes XML Dokument wird durch seine Elemente in einzelne hierarchisch strukturierte Bestandteile zerlegt. Diese Hierarchie kann in eine Baumstruktur überführt werden. Die Ansicht als Markup-Text und die Baumstruktur sind nur zwei verschiedene Darstellungen desselben Dokuments, weshalb im Folgenden die Begriffe Knoten und Element synonym das Selbe bezeichnen.

Der Baum besitzt genau einen Wurzelknoten, nämlich das Wurzel-Element. Die anderen Elemente bilden weitere Knoten des Baums, denen ein oder mehrere Attribute zugeordnet sein können. Die Knoten unterhalb eines Elements, werden als dessen **Kinder** bezeichnet. Das Element über einem Knoten wird als **Eltern-Element** bezeichnet. Mehrere Knoten, die das gleiche Eltern-Element besitzen, werden als **Geschwister** bezeichnet.

Für ein Dokument können über die Wohlgeformtheit hinaus weitere Einschränkungen vorgenommen werden. Die Festlegung einer solchen genauen Syntax nennt man Dokumententyp-Deklaration. Diese kann über Dokumententyp-Definitionen (DTD) oder über ein XML Schema erfolgen (siehe [Ray 2001; Grose u. a. 2001]). Mehrere Dokumententypen können in einem XML-Dokument durch Nutzung von Namensräumen kombiniert werden.

2.2.5.2 XML Metadata Interchange (XMI)

Der XML Metadata Interchange (XMI)-Standard definiert die Abbildung von MOF auf ein XML-Format, welches Instanzen von MOF-Artefakten (also Mo-

delle) in einer festgelegten Form darstellen kann. Eine ausführliche Einführung gibt [Grose u. a. 2001], die aktuelle Version 2.1 des Standards ist durch [Object Management Group (OMG) 2005a] spezifiziert.

Listing 2.2 zeigt hierzu die die Serialisierung des Schaltermodells, welches in Abbildung 2.4 in abstrakter Syntax dargestellt wurde, in XMI 2.1.

Das Wurzelelement *xmi:XMI* enthält das gesamte weitere Dokument. Das Wurzelelement verweist auf zwei Namensräume, welche im XML-Dokument genutzt werden.

Der *XMI*-Namensraum enthält Artefakte, welche unabhängig vom dargestellten Modell und vom genutzten Metamodell sind. Dies beinhaltet die im Folgenden beschriebenen Metadaten, Typisierung, aber auch den Identifikations- und Referenzmechanismus. Jedes Metamodell besitzt einen eigenen Namensraum, im Beispiel der *UML-Namespace* für die UML 2.0.

Das *xmi:Documentation*-Element bündelt Metadaten über das Modell. Dazu gehören im Beispiel die Benennung des Werkzeugs (inklusive Version), durch welches das XMI-Dokument erzeugt wurde. Auch Informationen über Benutzer, Kontaktadressen und Anmerkungen können hier eingebunden werden.

Das eigentliche Modell ist direkt unter dem *xmi:XMI*-Element angesiedelt und besteht aus mindestens einem Wurzelknoten für das eigentliche Modell, welches über die Kompositionsbeziehungen im Metamodell eine Baumstruktur aufspannt. Im Beispiel der UML ist dieser eine Instanz des Metamodellelements *Model*.

Jedem Modellobjekt wird eine innerhalb des Dokuments eindeutige Identifikation über das XML-Attribut *xmi:id* zugewiesen. Dies erlaubt es, diese Elemente über *xmi:idref*-Attribute eindeutig zu referenzieren und damit das Modell über die Kompositionsbaumstruktur hinaus zu vernetzen.

Die Abbildung der Artefakte aus dem Metamodell in das XML-Format geschieht wie folgt:

- **Metaobjekte:** Metaobjekte, also Modellelemente als Instanzen von Metaklassen, werden auf Wurzelebene als eigenständiges XML-Element im UML-Namensraum dargestellt. Im Beispiel in Listing 2.2 gilt das für *xmi:Model*. Ist das Element Teil einer Kompositionsbeziehung, ist der Name des XML-Elements durch den Rollennamen des untergeordneten Elements gegeben. Der Name des Datentyps des tatsächlichen Modellknotens ist durch das *xmi:type*-Attribut gegeben. Im Beispiel gilt dies für den *uml:State* mit dem Namen *Aus*, der als *subvertex* im übergeordneten Zustand enthalten ist.

Listing 2.2: Serialisierung des Schalters als XMI

```
<?xml version='1.0' encoding='UTF-8'?>
<xmi:XMI xmi:version='2.1'
         xmlns:uml='http://schema.omg.org/spec/UML/2.0'
         xmlns:xmi='http://schema.omg.org/spec/XMI/2.1'>
  <xmi:Documentation xmi:Exporter='MagicDraw UML'
                     xmi:ExporterVersion='12.0'/>
  <uml:Model xmi:id='1' name='Data' visibility='public'>
    <ownedMember xmi:type='uml:StateMachine' xmi:id='231'
                 name='Ohne Titel1' visibility='public'
                 context='231'>
      <region xmi:type='uml:Region' xmi:id='232'
              visibility='public'>
        <subvertex xmi:type='uml:State' xmi:id='327'
                   name='Schalter' visibility='public'>
          <region xmi:type='uml:Region' xmi:id='328'
                  visibility='public'>
            <subvertex xmi:type='uml:State' xmi:id='266'
                       name='Aus' visibility='public'>
              <outgoing xmi:idref='296'/>
              <incoming xmi:idref='280'/>
            </subvertex>
            <subvertex xmi:type='uml:State' xmi:id='252'
                       name='An' visibility='public'>
              <outgoing xmi:idref='280'/>
              <incoming xmi:idref='296'/>
            </subvertex>
            <transition xmi:type='uml:Transition' xmi:id='280'
                        visibility='public' kind='external'
                        source='252' target='266'/>
            <transition xmi:type='uml:Transition' xmi:id='296'
                        visibility='public' kind='external'
                        source='266' target='252'/>
          </region>
        </subvertex>
      </region>
    </ownedMember>
  </uml:Model>
</xmi:XMI>
```

- **Metaeigenschaften:** Abhängig von der Art der Eigenschaft im Metamodell werden Instanzen von Eigenschaften unterschiedlich gehandhabt:
  - Attribute: Jeder ein Metaobjekt darstellender Knoten, enthält seine Eigenschaftswerte, also die Werte von Attributen im MOF-Modell als XML-Attribute. So ist in Listing 2.2 für jeden *uml:State* ein Attribut *name* vorhanden.
  - Aggregationsbeziehungen: Assoziationen im Metamodell, welche als Aggregationsbeziehung, also eine Teil-Ganzes-Beziehung, gekennzeichnet sind, werden, wie bereits beschrieben, als untergeordnetes XML-Element mit dem Rollennamen des enthaltenen Modellelements gekennzeichnet.
  - Referenzen: Referenzen im Metamodell, also Assoziationen welche dieses nicht hierarchisieren, verweisen auf Modellelemente, die an anderer Stelle im XMI-Dokument dargestellt sind. Dies kann entweder durch ein enthaltenes XML-Element geschehen, welches über das *xmi:idref*-Attribut auf das gegenüberliegende Modellelement mit der entsprechenden *xmi:id* verweist. Bei einer Multiplizität größer als eins werden mehrere Unterelemente verwendet. Alternativ kann der Rollenname auch als Attribut aufgenommen werden, dessen Wert die *xmi:id* des referenzierten Metaobjekts angibt. Sollen mehrere Element über eine Rolle referenziert werden, werden die *xmi:id* durch Leerzeichen getrennt angegeben. Im Beispiel verweisen die *uml:State*-Elemente mit Hilfe der ersteren Methode über *incoming/outgoing*-Elemente auf einkommende und ausgehende Transitionen. Die Transitionen verweisen auf ihre Quelle (*source*) und ihr Ziel (*target*) über die entsprechenden XML-Attribute.

Neben Modell- und Metadaten können weitere werkzeugspezifische Inhalte in das Dokument eingebunden werden. Dies geschieht über das *xmi:Extension*-Element, welches über ein *extender*-Attribut das erweiternde Werkzeug bezeichnet. Alle enthaltenen Elemente müssen wohlgeformtem XML entsprechen, sind jedoch ansonsten nicht begrenzt.

2.2.6 Diagrammaustausch mit Diagram Interchange

UML-Modelle können mittels verschiedener Softwarewerkzeuge erzeugt, bearbeitet und gespeichert werden. Dafür stellt der in Abschnitt 2.2.5 vorgestellte Formatierungsstandard XMI (XML Metadata Interchange) das Format dar, mit welchem unterschiedliche UML-Werkzeuge solche Modelle austauschen können. Die XMI ermöglicht jedoch nicht, das gewünschte Ziel eines

kompletten Modellaustauschs gänzlich zu realisieren. Es ist zwar möglich, alle modellierten UML-Artefakte und deren Beziehungen in XMI-Dateien zu speichern und diese in andere Programme zu importieren, für das Aussehen der Diagramme aber wurden proprietäre Formate verwendet. Somit können die graphischen Informationen nicht Tool-übergreifend benutzt werden. Diese Einschränkung ist nicht durch XMI bedingt, sondern dem UML-Metamodell mangelt es an einem standardisierten Format zum Diagrammaustausch.

Aus diesem Grund wurde im Zuge der UML 2.0 die *Diagram Interchange* (kurz *DI*) Spezifikation der UML hinzugefügt, welche dem UML-Metamodell Diagramminformationen hinzufügt und diese mit dem eigentlichen Modell konsistent verknüpft. Die aktuelle Version 1.0 ist Teil der UML 2.1 (siehe [Object Management Group (OMG) 2005c]), kann aber auch dem Metamodell älterer Versionen der UML hinzugefügt und mit minimalen Änderungen auch in beliebigen Metamodellen dem Diagrammaustausch dienen.

Das im folgenden Abschnitt genauer beschriebene Metamodell zum Diagrammaustausch, befindet sich in einem gesonderten Paket (*DiagramInterchange*), in welchem sich die DI-Klassen befinden. Außer Assoziationen aus dem DI Paket zum *Top-Level* Element und Datentypen der UML existieren keine Verbindungen zu weiteren Modellartefakten, was eine Nutzung für andere, auf MOF basierende Modellierungssprachen einfach ermöglicht.

Die DI beschreibt den Aufbau eines Diagramms über Knoten und Kanten. Im Modell werden dabei Position und Größe der Knoten und Start, Ziel und Wegpunkte von Kanten beschrieben. Knoten können hierarchisch untergliedert sein, sie dürfen also weitere Knoten und Kanten graphisch enthalten.

Alle Diagramme werden ausschließlich durch einheitliche Metaklassen für Knoten und Kanten beschrieben. Diese Grundstruktur gilt für jedes Diagramm der UML. Es existieren somit keine gesonderten Modellartefakte für Klassen-, Sequenz- oder Anwendungsfalldiagramme.

Die *Diagram Interchange* Spezifikation verändert das existierende Metamodell nicht, um dessen Struktur nicht zu beeinflussen oder zu stören.

Um das Modell jederzeit konsistent halten zu können, sind die Diagramminformationen zwar unabhängig vom verwendeten Metamodell, jeder Knoten und jede Kante kann mit dem Modell jedoch auf eine der folgenden Arten verknüpft sein:

- Die Verknüpfung erfolgt zu einem Modellelement. So kann ein Knoten beispielsweise ein Element der Metaklasse *Class* oder der Metaklasse *State* im Diagramm repräsentieren.
- Die Verknüpfung erfolgt zu einem Attribut eines Modellelements. So kann

beispielsweise ein Knoten, welcher zu einem *State* gehört, einen weiteren Knoten enthalten, der mit seinem Namen verknüpft ist (vgl. auch Abbildung 2.3, Schichten M1 und M2).

- Der Knoten ist nicht direkt mit einem Modellelement verknüpft. Ein Klassensymbol im Klassendiagramm ist beispielsweise in drei Bereiche (*Compartments*) unterteilt, welche Eigenschaften, Attribute und Operationen aufnehmen. Jeder dieser Bereiche kann durch einen Knoten repräsentiert werden.

Kennzeichnend für den Standard ist hierbei, dass nur die graphische Anordnung und Ausmaße des Diagramms festgehalten werden. Die graphische Syntax einzelner Bausteine ist auch in der DI nicht enthalten. Dass eine Klasse von einem Rechteck dargeellt wird, ist nicht Teil der Diagrammbeschreibung. Diese Information muss das Modellierungswerkzeug entsprechend dem verknüpften Modellelement selbst bereitstellen. Hinweise zur Visualisierung bei alternativ möglichen Darstellungsweisen können jedoch gegeben werden (siehe Abschnitt 2.2.6.1).

Ein solches Vorgehen ermöglicht es, Redundanz im Modell zu vermeiden – es muss also nicht für jedes Modellelement im Diagramm abgelegt werden, wie es graphisch gezeichnet werden kann.

2.2.6.1 Metamodell

Abbildung 2.8 zeigt das Diagram Interchange Metamodell als MOF-Diagramm.

Ein Diagramm besteht aus graphischen Elementen (Metaklasse *GraphElement*), welche sich in die zentralen Metaklassen *GraphNode* und *GraphEdge*, die Knoten und Kanten mitsamt ihrer Eigenschaften repräsentieren, unterteilen.

Die durch *GraphElemente* beschrieben Graphen können über eine *„container-contained“*-Beziehung hierarchisch untergliedert werden. Es liegt hierbei ein Kompositum-Entwurfsmuster vor [Gamma u. a. 2004], welches eine beliebig tiefe Schachtelung von Knoten und Kanten ermöglicht.

Um Kanten mit Knoten zu verknüpfen existieren für jede *GraphEdge* genau zwei Elemente vom Type *GraphConnector*, die jeweils den Punkt kennzeichnen an welchem eine Kante einen der verknüpften Knoten graphisch berührt.

GraphNode-Elemente sind neben ihrer Position durch ihre Größe (Eigenschaft *size*) gekennzeichnet. Auch *GraphConnector*-Elemente weisen ein Attribut *position* zur Bestimmung des Schnittpunktes mit der zugehörigen *GraphNode*

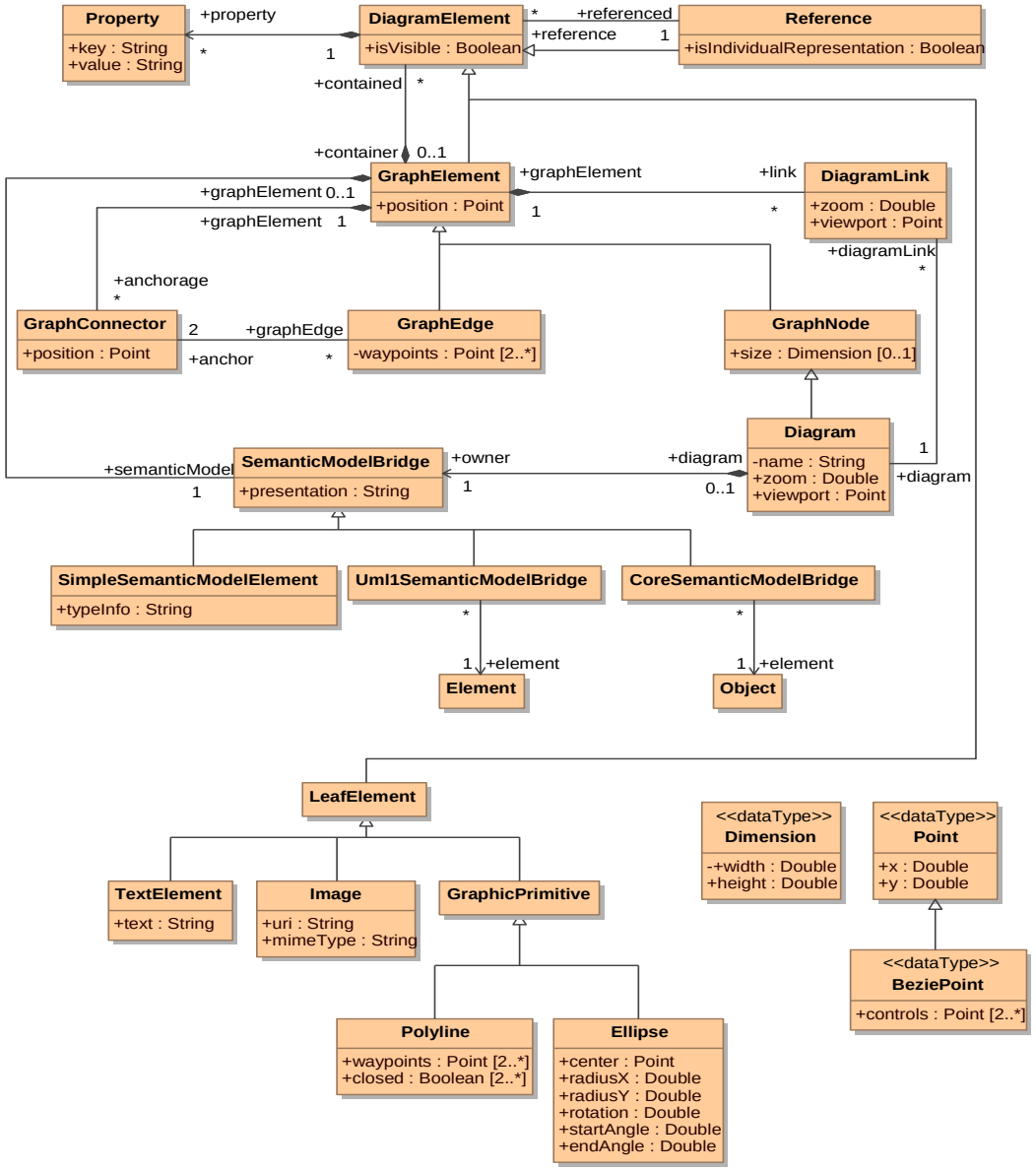

Abbildung 2.8: Das Diagram Interchange Metamodell

auf. Eine *GraphEdge* enthält über die Eigenschaft *waypoints* eine List von Wegpunkten zur Darstellung als Polygonzug oder stetige Kurve (z.B. Bezier-Kurve), welche den Pfad zwischen die Knoten genauer spezifizieren.

Die Verknüpfung der *GraphElemente* mit den eigentlichen Modellartefakten erfolgt über eine Instanz der abstrakten Metaklasse *SemanticModelBridge*, welche die Brücke zwischen Diagramm und Modell schlägt. Als konkrete Brücken stehen die *Uml1SemanticModelBridge* zur UML 1.x und die *CoreSemantic-*

ModelBridge zur UML 2 zur Verfügung. Weiterhin können Knoten Metaattribute zugeordnet werden, oder Knoten definiert werden, welche keine direkte Verknüpfung zum Modell besitzen. Dies geschieht über *SimpleSemanticModelElements*. Eine *ModelBridge* weist ein Attribut *presentation* auf, mit welchem spezifiziert werden kann, wie das verknüpfte Modellelement dargestellt werden soll. So kann beispielsweise ein Aktor in einem Anwendungsfalldiagramm als durch eine stereotypisierte Klasse dargestellt werden oder mit Hilfe eines graphischen Symbols.

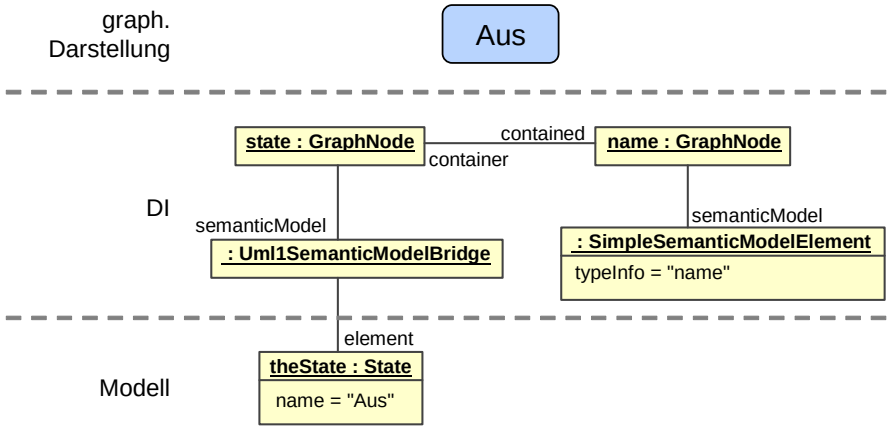

Abbildung 2.9: Ein State und seine Darstellung im Diagram Interchange Metamodell

Abbildung 2.9 zeigt die graphische Darstellung (oben) eines einzelnen *States* aus dem Beispiel in Abbildung 2.3. Die abstrakte Syntax basierend auf dem UML Metamodell ist im unteren Bereich gezeigt. Das Objekt *theState* enthält jedoch keine Diagramminfomationen. Diese werden durch die Modellobjekte in der Mitte hinzugefügt. Der Zustand selbst wird durch einen *GraphNode* dargestellt, welcher Informationen über Größe und Position trägt. Dieser Knoten ist mit dem Modellelement über eine Instanz der Metaklasse *Uml1SemanticModelBridge* verknüpft. Der Zustandsknoten enthält einen weiteren *GraphNode*, der keinem Modellelement zugeordnet ist, sondern über ein *SimpleSemanticModelElement* ein Attribut (hier den Namen) des übergeordneten Knotens bezeichnet.

Die Diagrammrepräsentation in DI kann in unterschiedlicher Granularität erfolgen. So kann beispielsweise der Knoten *state* als atomar angenommen werden und der enthaltene Name des Zustands als impliziter Teil des Symbols betrachtet werden. In diesem Fall entfallen die Objekte *name:GraphNode* und *:SimpleSemanticModelElement* in Abbildung 2.9. Um diese Unschärfe in der Spezifikation der DI handhaben zu können, empfiehlt es sich, bei der Erzeugung von Diagrammdaten so viele Informationen wie möglich zu generieren

und nicht benötigte Knoten gegebenenfalls im einlesenden Werkzeug zu verwerfen oder neu zu generieren.

Das Diagramm (Metaklasse *Diagram*) selbst existiert als spezieller *GraphNode*. Diesem kann über ein *SimpleSemanticModelElement* eine Diagrammtyp und über eine weitere Brücke ein Namensraum zugewiesen werden.

Alle Diagrammelemente können beliebig viele weitere Eigenschaften als Instanz der Metaklasse *Property* besitzen. Diese Eigenschaften bestehen aus Schlüssel-Wert-Paaren und dienen der Beschreibung weiterer werkzeugspezifischer Eigenschaften wie z.B. Farbe oder verwendeter Schriftart.

Der Übergang zur Nutzung des DI ist bei zahlreichen UML-Werkzeugen noch nicht abgeschlossen. Viele sonst UML 2-konforme Tools nutzen zum Zeitpunkt dieser Arbeit nach wie vor den Erweiterungsmechanismus der XMI (siehe Abschnitt 2.2.5.2) um eigene XML-basierte Diagrammdaten abzulegen.

2.3 Modellierungssprachen

Dieser Abschnitt beschreibt die Grundkonzepte der für diese Arbeit relevanten Modellierungssprachen. Dies ist in erster Linie die UML, die in späteren Kapiteln untersuchten Ansätze sind jedoch auch auf domänenspezifische Sprachen und spezialisierte CASE-Werkzeuge anwendbar. Möglich ist dies durch Abbildung der Modellierungsmöglichkeiten auf die in den bisherigen Abschnitten beschriebenen Grundkonzepte und –techniken.

2.3.1 Unified Modeling Language (UML)

Die *Unified Modeling Language* (UML) bezeichnet eine Familie graphischer Notationen, welche in ein gemeinsames MOF-konformes Metamodell integriert wurden.

Die UML liegt aktuell in der Version 2.1.1 vor und ist in drei Teildokumenten spezifiziert. Die *UML Infrastructure* definiert in [Object Management Group (OMG) 2007] fundamentale Konzepte und abstrakte Metamodelltypen, welche von weiteren Modellierungselementen generalisieren. Konkrete Syntax und Diagramme werden nicht spezifiziert. Dies geschieht in der *UML Superstructure* [Object Management Group (OMG) 2005d], welche darauf aufbauend die eigentliche Modellierungsmöglichkeiten und –diagramme spezifiziert. Das UML-Metamodell umfasst zahlreiche Randbedingungen für zulässige Modelle. Diese Randbedingungen werden mittels der *Object Constraint Language* (OCL) definiert (siehe [Object Management Group (OMG) 2005b]).

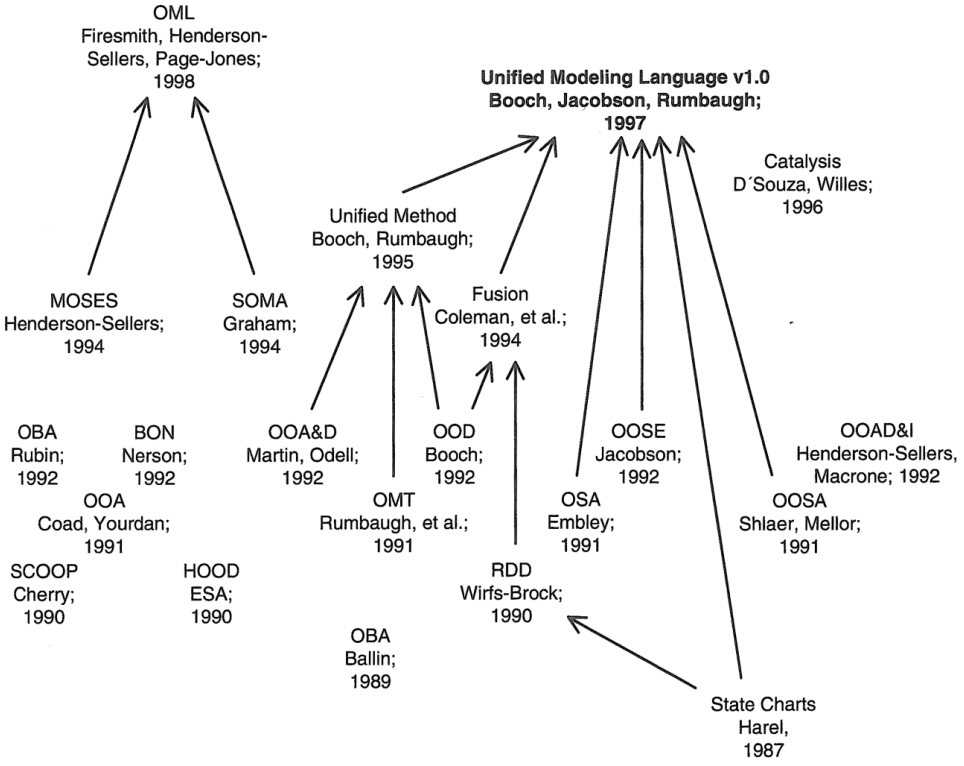

Abbildung 2.10: Entwicklungsgeschichte der UML als Vereinheitlichung zahlreicher Methoden [Jeckle u. a. 2003]

In ihrer, in Abbildung 2.10 skizzierten historischen Entwicklung ging die UML aus einer großen Zahl unterschiedlicher Modellierungsmethoden für objektorientierte Software hervor. Diese Ansätze entstanden alle zu Beginn der 90-er Jahre als Antwort auf den sich abzeichnenden Paradigmenwechsel von strukturierter Softwareentwicklung hin zu objektorientierten Ansätzen und vereinigten unterschiedlichste Entwicklungsprozesse und Notationsarten. Diese Ansätze hatten unterschiedlichste Ziele, jedoch auch zahlreiche Überschneidungen, die zu tatsächlichen konzeptionellen Alternativen lästige Inkompatibilitäten hinzufügten. Jede der sich hinter den jeweiligen Autoren versammelnden Gruppen bevorzugte den eigenen Ansatz, weshalb auch von „Methodenkriegen der Softwareentwicklung“ gesprochen wird.

Neben wissenschaftlichen Differenzen spielten jedoch auch wirtschaftliche Interessen der jeweiligen Werkzeughersteller eine Rolle. So führte die Vereinigung der Ansätze von Rumbaugh und Booch zur *Unified Method*, welche unter dem Dach von *Rational* einen kritischen Marktanteil zu gewinnen schien, zum Ruf nach einer Vereinheitlichung der Beschreibungsmittel. Unter dem Dach der

OMG entstand unter Einbeziehung weiterer Methoden, insbesondere des Ansatzes OOSE von Jacobson, die UML 1.0.

Im weiteren Verlauf übernahm die OMG das Copyright der UML. Auch sind die ursprünglichen Sprachschöpfer der UML nur noch peripher an neuen Versionen beteiligt.

Entscheidend ist der Ursprung der UML als Modellierungssprache für objektorientierte Software mit einer Zentrierung um die Begriffe *Klasse* und *Objekt*. Dieser Anspruch wurde insbesondere durch die UML 2.0 und die SysML auf die Modellierung der Entwurfsdomäne, nicht objektorientierter Software und genereller technischer Systeme ausgeweitet.

Weiterhin definiert die UML nur Datenmodell und Notation der Modellierungssprache, nicht jedoch Entwurfsprozess und Zusammenhänge der Diagramme. Daher kann ein UML-Modell mit leicht unterschiedlicher Semantik sowohl als Skizze, als auch als Bauplan und Programmiersprache genutzt werden (vergleiche Abschnitt 2.1). Die für diese Arbeit zentrale Nutzung als Programmiersprache wird in Abschnitt 3.1 beschrieben.

Die exakte semantische Bedeutung einzelner Elemente ist variabel und in zahlreichen Fällen bewusst offen gelassen. Die UML kapselt einige, jedoch bei weitem nicht alle solche Stellen der Spezifikation als *Semantic Variation Points*, die explizit Freiräume für unterschiedliche Interpretationen lassen. Dies spielt auch insbesondere bei Nutzung der UML als Programmiersprache eine wichtige Rolle, da hier Teile der Semantik oft erst durch die Transformation in Code implizit in das Modell einfließen.

Die UML unterscheidet strikt zwischen dem MOF-basierten Modell und den Sichten auf das Modell, den Diagrammen. Ein Modellelement kann in beliebig vielen Diagrammen enthalten sein, teilweise in unterschiedlicher Syntax. So kann beispielsweise ein Objekt Teil eines Objektdiagramms sein, jedoch auch als Interaktionspartner in einem Sequenzdiagramm erscheinen.

Zentral für die Strukturierung der *UML Superstructure* ist die Unterteilung in Spracheinheiten (*language units*), welche eine Menge von zusammengehörenden Modellierungselementen bündeln. Innerhalb dieser Einheiten sind die Modellierungsmöglichkeiten in vier *Compliance Levels* unterteilt, in welche sich Modellierungswerkzeuge entsprechend ihrer Fähigkeiten einsortieren lassen.

Zur Modellierung mit der UML existiert eine unüberschaubar große Anzahl an Werkzeugen, welche häufig auf bestimmte Kundenkreise zugeschnitten sind und ihren Fokus beispielsweise entweder Richtung Analyse und Modellierung oder auf gute Konsistenz mit erzeugtem Quellcode legen.

2.3.1.1 Diagramme

UML 1.5	UML 2.1
Klassendiagramm	Klassendiagramm
	Paketdiagramm
Objektdiagram	Objektdiagramm
	Kompositionsstrukturdiagramm
Komponentendiagramm	Komponentendiagramm
Verteilungsdiagramm	Verteilungsdiagramm
Anwendungsfalldiagramm	Anwendungsfalldiagramm
Aktivitätsdiagramm	Aktivitätsdiagramm
Zustandsdiagramm	Zustandsdiagramm
Sequenzdiagramm	Sequenzdiagramm
Kollaborationsdiagramm	Kommunikationsdiagramm
	Zeitdiagramm
	Interaktionsübersichtsdiagramm

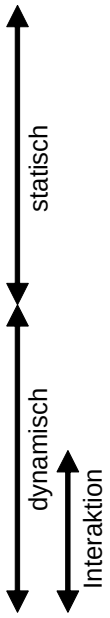

Abbildung 2.11: Übersicht und Klassifikation der Diagramme der UML 1.5 und UML 2.1. Grau hinterlegt sind in der UML 2.0 hinzugekommene oder stark überarbeitete Diagrammtypen.

Für einen Anwender der UML sind Diagramme nach wie vor von zentraler Bedeutung. Abbildung 2.11 listet alle 9 Diagramme der UML 1.5 und alle 13 Diagramme der UML 2.1 auf. Die grau hinterlegten Bereiche kennzeichnen neue oder stark veränderte Diagramme der UML 2.

Die Diagramme unterteilen sich in statische und dynamische Diagramme. Erstere treffen Aussagen über die Struktur des zu entwerfenden Systems. Es handelt sich dabei immer um statische Entitäten, ihre Eigenschaften und Verknüpfungen untereinander. Dies geschieht auf unterschiedlichen Ebenen im Anwendungsfalldiagramm (Anforderungen), Klassendiagramm (Mögliche Eigenschaften und Verknüpfungen von Objekten) und Objektdiagramm (Objekte zu einem bestimmten Zeitpunkt).

Dynamische Diagramme beschreiben das Verhalten des Systems, also zeitliche oder ereignisgetriebene Abläufe. Eine Untergruppe dynamischer Diagramme ist durch Interaktionsdiagramme gegeben, welche die Kommunikation zwischen Entitäten (*Classifier*) im Modell in unterschiedlichen Sichten darstellen.

Die Diagramme sind im Einzelnen (in Klammern die geläufigen englischen Begriffe):

- **Klassendiagramm *(Class diagram)*:** Das Klassendiagramm ist das geläufigste Diagramm zur Darstellung der objektorientierten Softwarestruktur in Form von Klassen und ihren Eigenschaften. Es kann sowohl in der Analysephase, aber auch zur Codegenerierung genutzt werden. Abschnitt 2.3.1.3.1 beschreibt Klassendiagramme detaillierter.

- **Objektdiagramm *(Object diagram)*:** Das in Abschnitt 2.3.1.3.2 genauer beschriebene Objektdiagramm kann als Schnappschuss des Systems zu einem bestimmten Zeitpunkt verstanden werden. Es stellt die Instanzen von Klassen sowie ihre Attributwerte und Relationen zu anderen Objekten dar.

- **Paketdiagramm *(Packet diagram)*:** Ein Modell kann aus vielen tausend Modellartefakten bestehen. Es ist also nötig Modelle modular strukturieren zu können. Dies geschieht mit Hilfe von Paketen und Subsystemen, welche einen Modellbaum aufspannen. Pakete erhöhen auch die Möglichkeit der Wiederverwendung funktional zusammengehörender Modellteile. Die Pakete und ihre Zusammenhänge werden im Paketdiagramm dargestellt. Abbildung 2.5 zeigt ein Paketdiagramm auf MOF-Ebene. Neben der funktionalen Gliederung des Modells kann das Paketdiagramm zur Modellierung von Schichtenarchitekturen verwendet werden.

- **Kompositionsstrukturdiagramm *(Composite structure diagram)*:** Das Kompositionsstrukturdiagramm zeigt die interne Struktur einer Klasse und die Kollaborationen, die diese ermöglicht. Eine solche Struktur besteht aus Teilen (*parts*), welche sich rekursiv modularisieren lassen. Die Teile besitzen Ein- und Ausgänge (*ports*) über welche die Teile verknüpft werden können. Die Teile interagieren zur Laufzeit, wobei sie eine bestimmte Aufgabe erfüllen. Dieses neue Diagramm der UML 2.0 ähnelt in seiner ingenieursnahen Semantik Signalflussmodellen und elektronischen Schaltplänen.

- **Komponentendiagramm *(Component diagram)*:** Mehrere zusammenarbeitende Klassen können in der UML zu einer logischen Komponente gruppiert werden, welche über eine öffentliche Schnittstelle oder *Ports* ein gemeinsames Verhalten zur Verfügung stellt. Das Komponentendiagramm kann sowohl dafür genutzt werden, die interne Struktur von Komponenten zu modellieren, als auch die Wechselbeziehungen, also die Verknüpfung der Schnittstellen, darstellen.

- **Verteilungsdiagramm *(Deployment diagram)*:** Die physikalische Verteilung auf eine Hardware-Topologie geschieht mit Hilfe des Verteilungsdiagramms. Dies steht im Gegensatz zum Komponentendiagramm,

welches das System logisch gruppiert. Den physikalischen Knoten können sowohl Komponenten als auch Artefakte, wie Bibliotheken oder ausführbare Dateien, zugeordnet werden. Seine Anwendung ist nur bei verteilten Systemen sinnvoll.

- **Anwendungsfalldiagramm *(Use case diagram)*:** Das Anwendungsfalldiagramm beschreibt das nach außen sichtbare Verhalten des Systems. Das Diagramm stellt Anwendungsfälle dar und zeigt außerhalb des Systems stehende Rollen, welche mit diesen interagieren. Diese Rollen können sowohl ein Benutzer, als auch andere (Software-)Systeme sein. Das Anwendungsfalldiagramm wird üblicherweise in sehr frühen Projektphasen zur Analyse und Gruppierung der Nutzeranforderungen auf hohem Abstraktionsniveau genutzt und kann auch zur Kommunikation mit Kunden eingesetzt werden.

- **Aktivitätsdiagramm *(Activity diagram)*:** Das Aktivitätsdiagramm dient zur Beschreibung von Verhalten im System. Es kann zur Darstellung von komplexen Algorithmen, aber auch von Geschäftsabläufen mit mehreren beteiligten Akteuren und zur detaillierteren Beschreibung von Anwendungsfällen dienen. Kern des Aktivitätsdiagramms sind Aktionen, welche über Kontrollflusspfeile sequentialisiert sind. Neben Kontrollelementen zur Ablaufsteuerung bietet das Aktivitätsdiagramm die Möglichkeit zur Parallelisierung und ein an Petri-Netze angelehntes *Token*-Konzept.

- **Zustandsdiagramm *(State diagram)*:** Das Zustandsdiagramm erlaubt eine Beschreibung reaktiver Aspekte des modellierten Systems durch eine hierarchisch untergliederte Zustandsmaschine und kann zur Beschreibung von Klassenverhalten, Protokollen oder von Anwendungsfällen verwendet werden. Abschnitt 2.3.1.3.3 geht auf Zustandsdiagramme genauer ein.

- **Sequenzdiagramm *(Sequence diagram)*:** Sequenzdiagramme sind die geläufigsten und ausdruckstärksten Interaktionsdiagramme. Sie stellen entlang einer vertikalen Zeitachse die Kommunikation von Objekten dar. Diese Interaktion erfolgt über Nachrichten, welche synchron (Operationsaufruf) oder asynchron (Signal/Ereignis) sind. Interaktionsdiagramme allgemein können in der Analysephase zur detaillierteren Beschreibung von Anwendungsfällen dienen, aber auch Detailabläufe der Implementierung oder Testfälle modellieren.

- **Kommunikationsdiagramm *(Communication diagram)*:** Das Kommunikationsdiagramm, welches bis zur UML 1.5 als Kollaborationsdiagramm benannt war, beschreibt eine Interaktion mit einem Schwer-

punkt auf die beteiligten Objekte und ihre Verknüpfung. Da keine Zeitachse vorhanden ist, kann der Ablauf selbst nur anhand einer Nummerierung der Interaktionen nachvollzogen werden.

- **Zeitverlaufsdiagramm *(Timing diagram)*:** Das Zeitverlaufsdiagramm kombiniert Interaktionsmodellierung und Zustandsdiagramm. Es zeigt eine horizontale Zeitachse, an der abgelesen werden kann, zu welchem Zeitpunkt die Interaktionspartner in welchem Zustand befinden. Dies geschieht mit einer Zeitverlauflinie, wie sie aus der Digitaltechnik bekannt ist. Zusätzlich können Interaktionen, welche zur Zustandsänderung führen eingetragen werden.

- **Interaktionsübersichtsdiagramm *(Interaction overview diagram)*:** Dieses Diagramm kombiniert ein Aktivitätsdiagramm mit untergeordneten Interaktionen. Es ermöglicht, zahlreiche Interaktionen in einen übergeordneten Ablauf zu integrieren. Dabei können kleinere Interaktionen integriert werden oder auf externe Interaktionen verwiesen werden.

Für die vorliegende Arbeit sind vor allem Klassen-, Objekt- und Zustandsdiagramme bedeutend. Diese werden im Folgenden ausführlicher beschrieben und der für das weitere Verständnis der Arbeit wichtige zugehörige Teil des Metamodells vorgestellt.

2.3.1.2 Grundlagen Metamodell der UML

Das Metamodell der UML ist in zahlreiche Pakete unterteilt. Abhängig vom *Compliance Level* des eingesetzten Metamodells umfasst die UML *Superstructure* zwischen 3 (Level 0) und 41 Paketen (vollständige Spezifikation, Level 3), welche vollständig in der Spezifikation behandelt werden (siehe [Jeckle u. a. 2003]).

Im Folgenden werden die Pakete für Struktur- und Verhaltensmodelle stark vereinfacht dargestellt. Einzelne Pakete können bestimmten Diagrammen entsprechen, müssen dies jedoch nicht. Elemente des *Interactions*-Pakets können beispielsweise in zahlreichen Diagrammen erscheinen. Alle gezeigten Metamodelldiagramme der UML entstammen der *UML Superstructure*-Spezifikation [Object Management Group (OMG) 2005d].

Abbildung 2.12 zeigt die Hierarchie zur Strukturmodellierung. Diese verdeutlicht, dass die Pakete, und damit die Modellierungsmöglichkeiten der UML, aufeinander aufbauen.

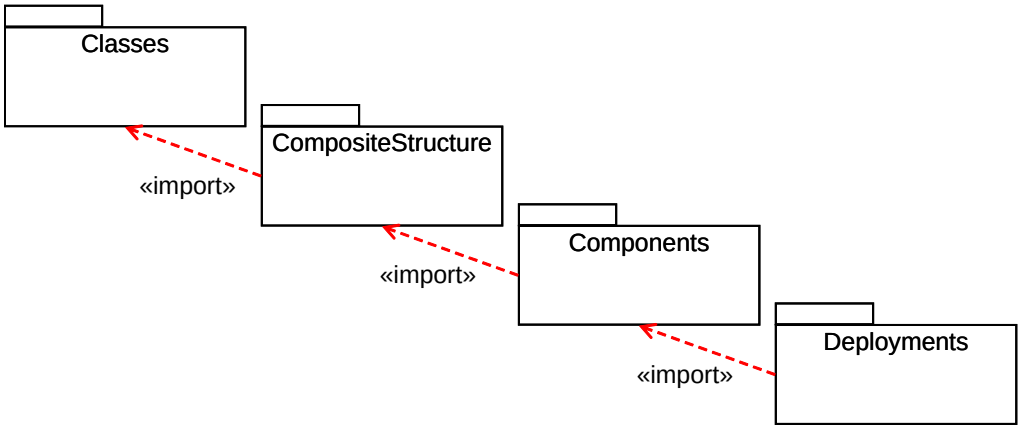

Abbildung 2.12: Pakete zur Strukturmodellierung

Die grundlegende zentrale Modellierungsart ist die Klassenmodellierung, auf welcher, unter Nutzung des Begriffs der Klasse die Kompositionsstruktur (Paket *CompositeStructure*) aufbaut. Komponenten, wie sie in einem Komponentendiagramm dargestellt werden können, spezialisieren Kompositionsstrukturen weiter, indem sie deren Schnittstellen und ihre Realisierungen bündeln. *Deployments* sind im Metamodell spezielle *CompositeStructures*, welche alle anderen Struktureigenschaften des Modells physikalischen Einheiten zuordnen können.

Die zentralen Pakete zur Modellierung von Verhalten präsentiert Abbildung 2.13. Ihre gemeinsamen Eigenschaften sind im Paket *CommonBehaviors* festgelegt. Darunter fallen die Verknüpfung mit dem statischen Modell, aber auch Konzepte für Ereignisdefinitionen, Ereignisse (*Trigger*) und Zeitbegriffe.

CommonBehaviors wird durch alle weiteren Pakete spezialisiert. *StateMachines* beschreibt hierarchische Zustandsautomaten und *UseCases* Anwendungsfälle. *Interactions* fasst alle Modelle zusammen, welche Abläufe zwischen Objekten beschreiben. Die unterschiedlichen Interaktionsdiagramme präsentieren das gleiche Modell auf unterschiedliche Weise. *Actions* erlauben die Modellierung von Abläufen und Algorithmen auf niedriger detaillierter Ebene und besitzen keine spezifizierte syntaktische Entsprechung (siehe auch Abschnitt 3.2). Aktivitäten (in *Activities*) abstrahieren einige Actions und bieten die Modellierung von Abläufen auf hoher Ebene. Dargestellt werden sie im Aktivitätsdiagramm.

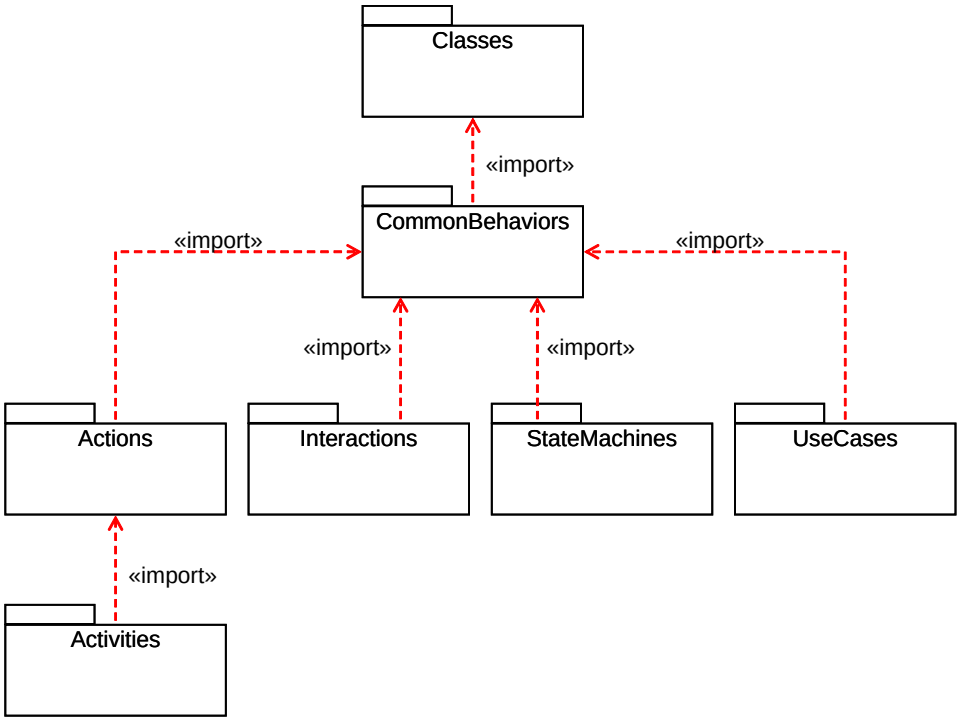

Abbildung 2.13: Verhaltensmodellierung mit UML

2.3.1.3 In dieser Arbeit genutzte Diagramme

2.3.1.3.1 Klassendiagramme Klassendiagramme stellen die grundlegende objektorientierte Struktur einer Software dar. Sie sind implementierungsnah und können gut zur Codeerzeugung genutzt werden, da ihre Elemente einfach auf objektorientierte Programmiersprachen abgebildet werden können.

Abbildung 2.14 zeigt beispielhaft ein einfaches Klassendiagramm. Das Diagramm besteht aus vier Klassen *Schalter*, *Fensterheber*, *Motor*, *Einklemmschutz*. Jedes der Klassensymbole besteht aus drei Bereichen für Titelzeile, Attribut- und Operationenbereich.

Ausgangspunkt in diesem Beispiel ist die Klasse *Fensterheber*, welche vier verschiedene Operationen besitzt und über drei Assoziationen mit den weiteren Klassen verknüpft ist. Die Beschriftung auf der Seite der entfernten Klasse zeigt die Rolle an, welche das Element (z.B. *Schalter*) für die Klasse auf der gegenüberliegenden Seite (*Fensterheber*) einnimmt. Die Multiplizität besagt, dass eine Instanz von *Fensterheber* zur Laufzeit immer mit einer Instanz von *Motor* assoziiert ist (*1*), mit einer Instanz *Einklemmschutz* optional assoziiert sein kann (*0..1*) und mit beliebig vielen Instanzen von *Schalter* verknüpft

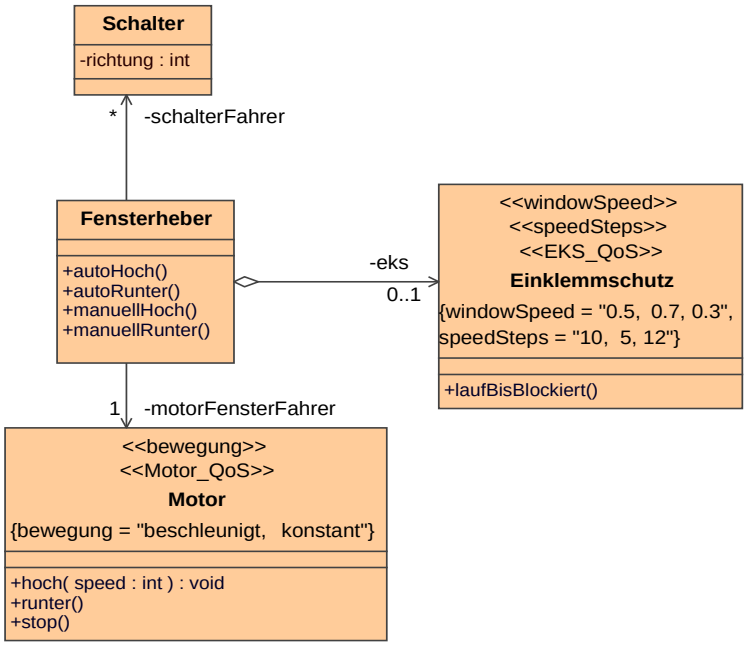

Abbildung 2.14: Beispiel eines Klassendiagramms

werden kann (*). Die durch eine Raute gekennzeichnete Aggregation zwischen *Fensterheber* und *Einklemmschutz* bezeichnet eine Teil-Ganzes-Beziehung.

Schalter besitzt ein Attribut *richtung* vom ganzzahligen Datentyp *int*. Die Operation *hoch* der Klasse *Motor* besitzt einen Parameter *speed* vom Typ *int* und den Rückgabetyp *void*.

Sowohl Attributen, als auch Rollen kann eine Sichtbarkeit zugeordnet sein, welche den Zugriff auf diese reglementiert.

Die Klassen Motor und Einklemmschutz besitzen Stereotypen (z.B. *«speedSteps»*) und zugeordnete *TaggedValues*, welche sich in geschweiften Klammern unter dem Klassennamen befinden.

Abbildung 2.15 zeigt den Ausschnitt des Metamodells, der sich mit *Features* befasst. Diese sind Merkmale, welche beliebigen *Classifiern* zugeordnet sein können. Ein *Classifier* ist ein Element, welches im Modell instanziiert werden kann. *Features* unterteilen sich in strukturelle Eigenschaften (*StructuralFeature*) und Verhaltenseigenschaften (*BehavioralFeature*). *StructuralFeatures* besitzen Typ und Multiplizität. Ein *BehavioralFeature* besitzt eine geordnete Parameterliste, welche typisierte Parameter und ihre Standardwerte enthält.

In Abbildung 2.16 ist der grundlegende Aufbau der Klassenmodellierung abzu-

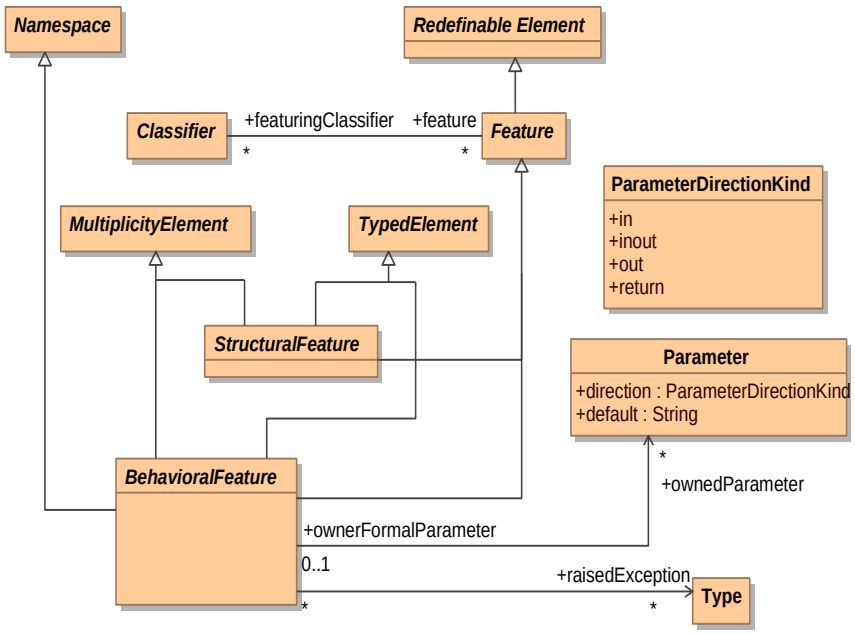

Abbildung 2.15: Features im Metamodell

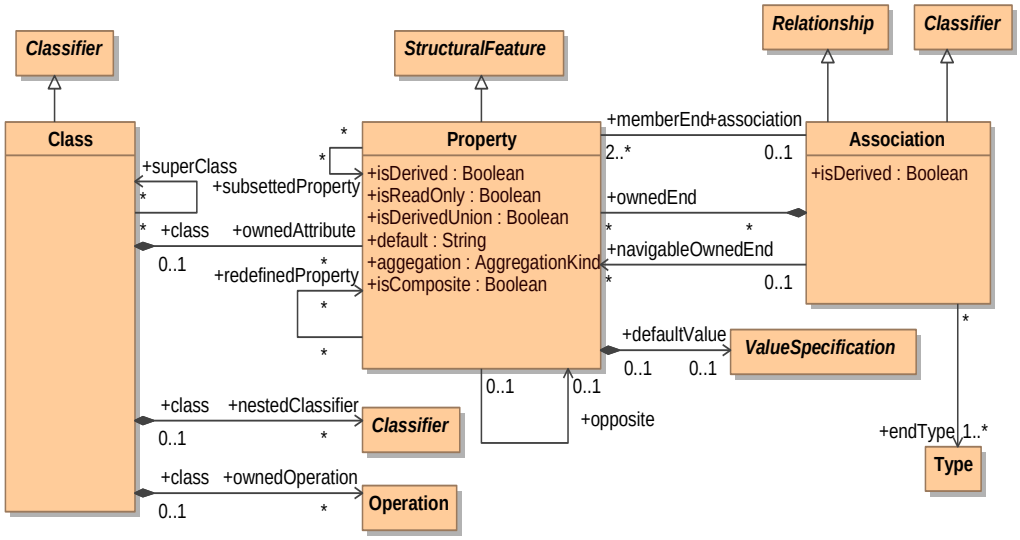

Abbildung 2.16: Klassen, Assoziationen und Attribute

lesen. Eine Klasse (Metaklasse *Class*) ist ein spezieller *Classifier* für Objekte und wird in der Klassenmodellierung genutzt.

Eine Klasse besitzt über die Rolle *ownedAttribute* strukturelle Eigenschaften,

die als *Properties* bezeichnet sind. *Properties* haben als *StructuralFeatures* einen Typ, Multiplizität und zusätzlich einen Standardwert (*defaultValue*).

Im Datenmodell wird dabei nicht zwischen Attributen und Assoziationsenden unterschieden. Gehört eine *Property* zu einer Assoziation muss ihr Datentyp dem des Classifiers am gegenüberliegenden Ende entsprechen. Einer Klasse gehört demnach immer das entfernte Assoziationsende. Im Modell zum Beispiel in Abbildung 2.14 besitzt *Fensterheber* eine *Property* mit dem Namen *schalterFahrer*.

Eine Klasse kann verschachtelt interne Klassen besitzen (Rolle *ownedClassifier*) und neben *Properties* zusätzliche Operationen aufweisen.

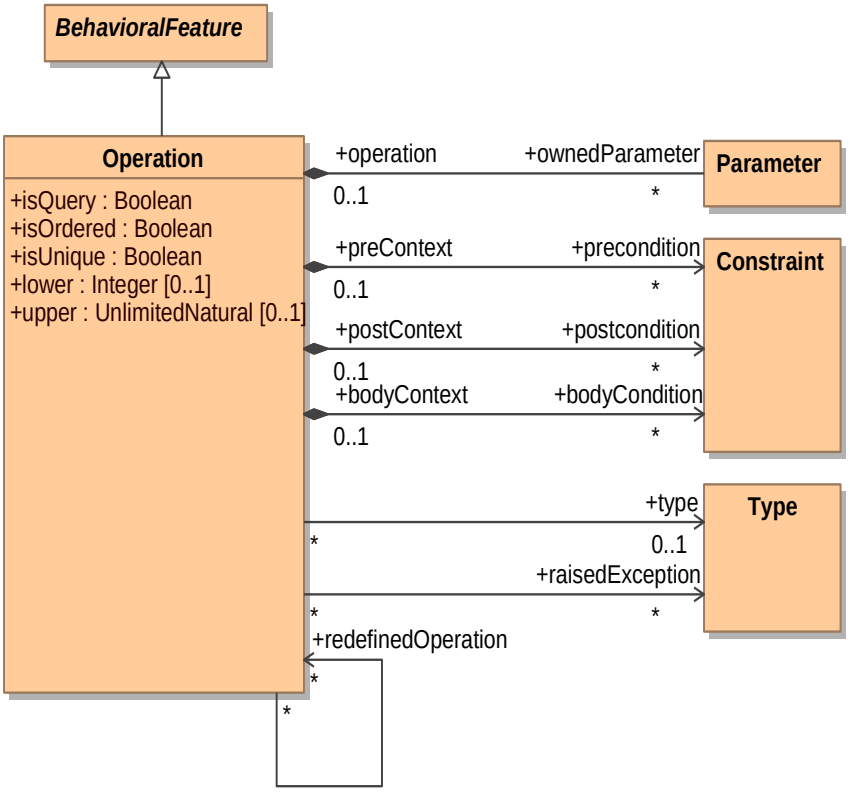

Abbildung 2.17: Operationen

In Abbildung 2.17 sind die in Klassen verwendeten *Operationen* dargestellt, welche spezielle *BehavioralFeatures* sind. Operation besitzen ihre Parameter, einen Rückgabetyp (*type*) und definieren die Ausnahmen, die in ihnen auftreten können. Weiterhin können Randbedingungen (*Constraints*) angegeben, welche Vor- und Nachbedingungen, sowie eine Bedingung für den Rückgabewert angeben.

2.3.1.3.2 Objektdiagramme Während das Klassenmodell den möglichen Aufbau und die Anordnung der Instanzen zur Laufzeit beschreibt, zeigen Objektdiagramme eine Momentaufnahmen zu einem bestimmten Zeitpunkt zur Laufzeit, sie zeigen Instanzen, Attributwerte und Verbindungen zwischen diesen Instanzen.

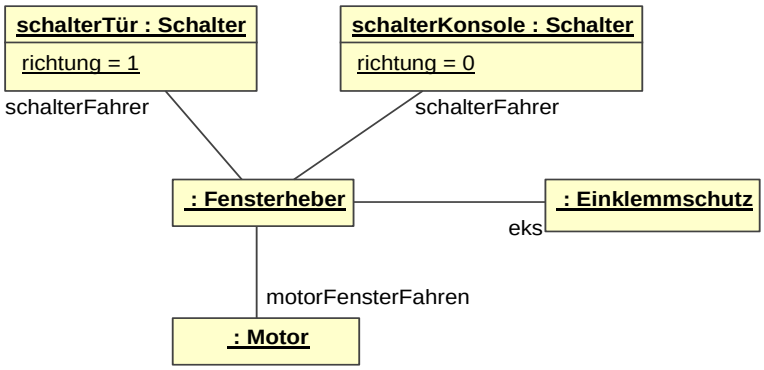

Abbildung 2.18: Beispiel eines Objektdiagramms

In Abbildung 2.18 ist ein Objektdiagramm gezeigt. Es stellt einen Schnappschuss des Klassendiagramms aus Abbildung 2.14 dar und enthält Objekte. Die Kopfzeile jedes Objektes enthält die ursprüngliche Klasse der Instanz. Optional kann ein Objekt mit einem eindeutigen Namen bezeichnet werden. Im Beispiel besitzen die Instanzen von *Schalter* einen Namen, die anderen Objekt nicht. Die Verknüpfungen zwischen den Objekten werden als *Links* bezeichnet und sind Instanzen der Assoziationen. Die Schalter besitzen *Slots*, welche Attributwerte der einzelnen Objekte zeigen. Die Anzeichnung der Rollennamen und Assoziationstypen aus dem Klassendiagramm ist optional.

Im vorliegenden Beispiel besitzt der *Fensterheber* wie gefordert genau ein Objekt *Motor* und den optionalen *Einklemmschutz*. Er kann durch zwei *Schalter* bedient werden.

Abbildung 2.19 zeigt das Metamodell zur Instanzspezifikation. Eine Momentaufnahme einer Klasse oder einer Assoziation ist eine Instanz von *InstanceSpecification*. Unterschieden werden die beiden Typen über den *Classifier*, mit dem die Instanz verbunden ist. Die Abbildung zeigt weiterhin die Metaklasse *Slot*, welche einen Eintrag in der Werteliste der Instanz darstellt. Jeder *Slot* gehört zu einer strukturellen Eigenschaft (*definingFeature*), für welche ein Wert über die Rolle *value* verknüpft ist.

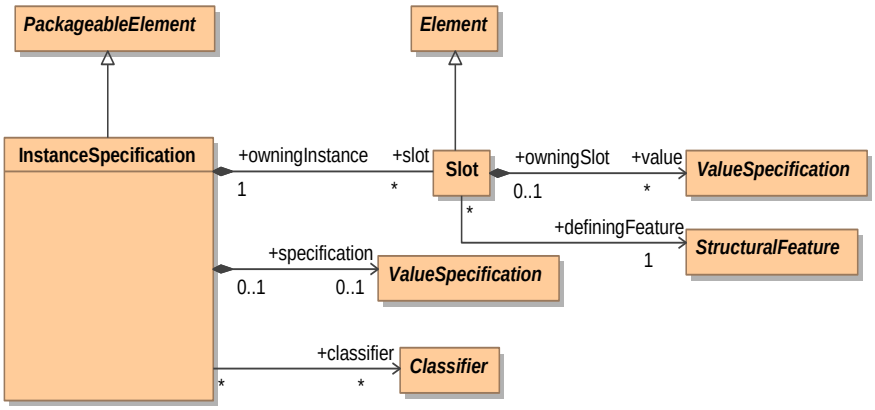

Abbildung 2.19: Metamodell Instanzspezifikation [Object Management Group (OMG) 2005d]

2.3.1.3.3 Zustandsdiagramme Mit Zustandsdiagrammen kann reaktives ereignisdiskretes Verhalten modelliert werden. Sie bilden eine Erweiterung endlicher Automaten um Hierarchie und Nebenläufigkeit.

Die in der UML verwendete Variante besitzt eine festgelegte Semantik, welche der ursprünglichen *Statechart*-Semantik von David Harel entspricht (siehe [Harel 1987]). Die Beschreibung der genauen Semantik überschreitet den Rahmen dieser Arbeit, daher wird das Zustandsdiagramm an einem Beispiel veranschaulicht.

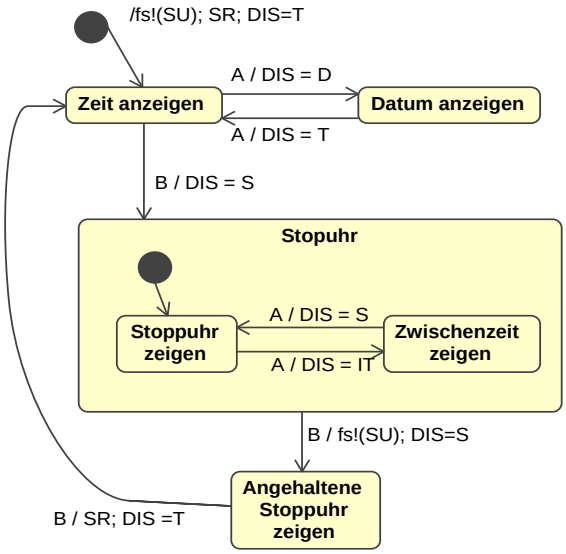

Abbildung 2.20: Beispiels eines UML Zustandsdiagramms

Das Zustandsdiagramm, das in Abbildung 2.20 gezeigt ist, beschreibt das ereignisdiskrete Verhalten einer Stoppuhr. Die Uhr kennt fünf unterschiedliche diskrete einfache Zustände, wobei die Zustände *Stoppuhr zeigen* und *Zwischenzeit zeigen* in einem zusammengesetzten Zustand gruppiert sind. Das System kann über als Pfeile dargestellte Transitionen zwischen den Zuständen wechseln. Die Beschriftung der Transitionen folgt der Syntax *Event [Condition] / Action*. Das *Event* ist ein Ereignis, welches die Transition schalten lässt, wenn auch die *Condition* als Bedingung erfüllt ist. Schaltet eine Transition, wechselt das System in einen neuen aktiven Zustand und führt die *Action* als Aktion aus. Initiale Transitionen bezeichnen die Unterzustände, welche beim Betreten zusammengesetzter Zustände aktiviert werden.

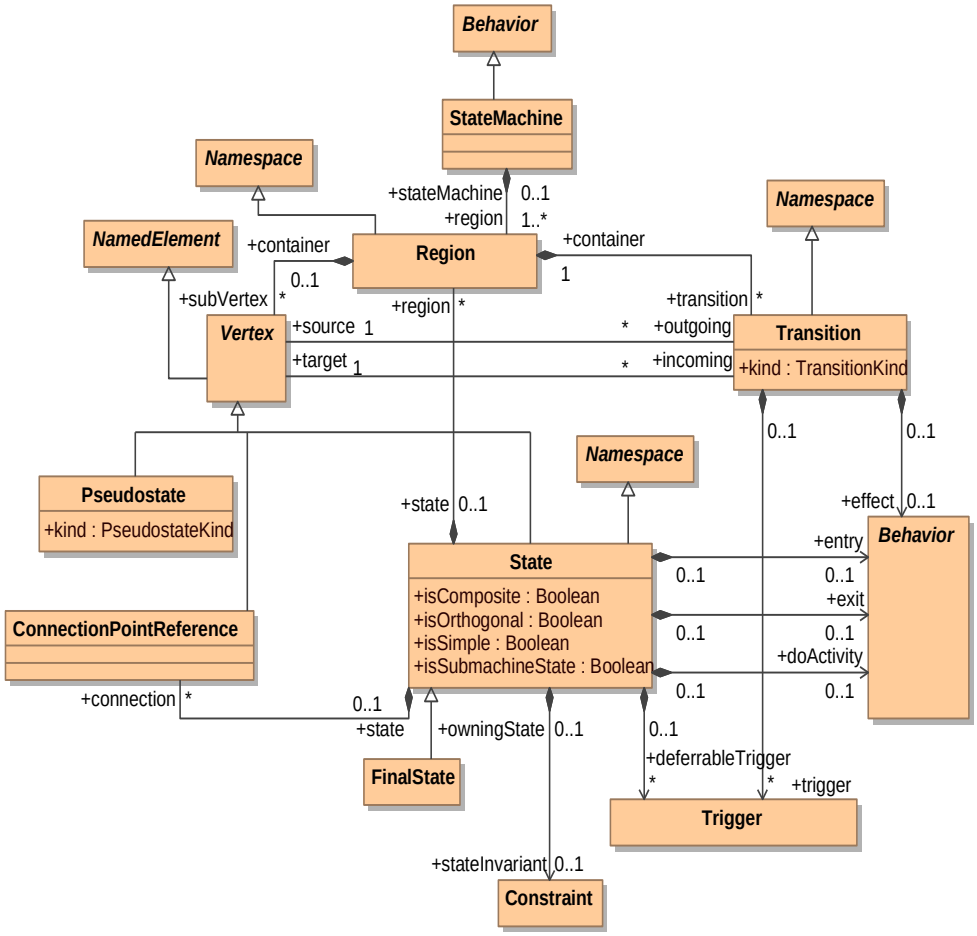

Abbildung 2.21: Metamodell für StateMachines [Object Management Group (OMG) 2005d]

In Abbildung 2.20 ist das Metamodell für Zustandsautomaten gezeigt. Zentral

ist die Metaklasse *State*, welche einfache, zusammengesetzte und nebenläufige zusammengesetzte Zustände beinhaltet. Die abstrakte Superklasse des Zustands ist der *Vertex*, welcher neben Zuständen *PseudoStates* beinhaltet. Ein *PseudoState* ist ein Knoten in der Zustandsmaschine, welcher kein Zustand ist, z.B. Ausgangspunkte von Initialtransitionen oder *HistoryStates*, welche den Zustand eines zusammengesetzten Zustands bei Verlassen desselben speichern. Die Metaklasse *StateMachine* bezeichnet den Zustandsautomaten selbst und enthält mindestens eine Wurzelregion.

Jeder *Vertex* kann über Instanzen von *Transition* mit weiteren *Vertices* verbunden werden. Eine Transition besitzt entsprechend ihrer Beschriftung ein oder mehrere auslösende Ereignisse (Rolle *trigger*), eine Bedingung (Rolle *guard*) und eine ausgelöste Aktion (Rolle *effect*).

2.3.2 Domänenspezifische Modellierungssprachen

2.3.2.1 Grundlagen

Eine domänenspezifische Sprache (engl. *domain-specific language*, DSL) ist eine auf eine bestimmte Anwendungsklasse zugeschnittene Modellierungs- oder Programmiersprache. Domänenspezifische Sprachen werden auch als applikationsorientierte, *special-purpose*, spezialisierte, aufgabenspezifische oder Andwendungssprachen bezeichnet [Mernik u. a. 2005]. Diese können sowohl in textueller als auch in graphischer Form vorliegen. Graphische domänenspezifische Sprachen werden als *domain-specific modeling languages* (DSML) bezeichnet

Bei ihrer Definition wird der Anspruch aufgegeben, mehr als eine definierte Klasse von Problemlösungen beschreiben zu können. Sie stehen damit im Gegensatz zu *General-Purpose Languages* (GPL) wie Java oder UML, welche Turing-komplett sind und für viele, wenn nicht alle Anwendungsfälle einsetzbar sein sollen.

Die Ausdrucksstärke kann in modernen Programmiersprachen jedoch auch durch einfache Anwendungsbibliotheken (engl. *application programming interface*, API) oder *Frameworks* erhöht werden; auch in Modellierungssprachen kann dies durch geeignete Mechanismen, wie z.B. Profile[8] ermöglicht werden. Dediziert domänenspezifische Sprachen bieten jedoch darüber hinaus den Vor-

[8]Profile sind der Erweiterungsmechanismus der UML. Dabei können Modellelemente durch Stereotype in Ihrer Ausdrucksstärke, Semantik und Notation erweitert oder verändert werden. Randbedingungen können dazu genutzt werden, die Mächtigkeit der Modellierung einzuschränken (*tailoring*) [Object Management Group (OMG) 2007].

teil, dass eine angemessene domänenspezifische Notation die Sprache näher an die Welt des Domänenexperten rückt.

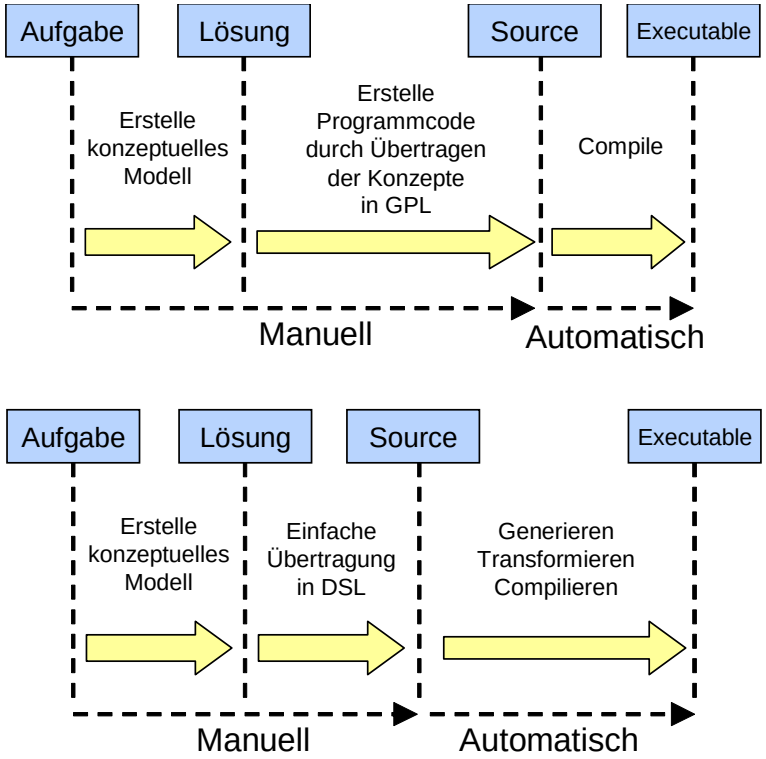

Abbildung 2.22: Übergänge aus der Problem- in die Lösungsdomäne bei Nutzung von GPL (oben) und DSL (unten)

Domänenspezifische Modellierungssprachen tauschen also Allgemeingültigkeit gegen Ausdrucksstärke und einfachere Nutzbarkeit durch einen Domänenexperten in einem klar umrissenen Spezialgebiet ein. Abbildung 2.22 illustriert dies: Vereinfacht bedeutet der Weg von der Aufgabe zu einem Softwaresystem immer eine Formalisierung der Problemlösung. Bei einer GPL ist der Weg von der informellen Lösung in den formalisierten Quelltext recht aufwändig und benötigt sowohl Wissen über die verhältnismäßig komplexe Sprache als auch über die Problemdomäne. Eine DSL rückt die Formalisierung näher an die informelle Lösung. Erkauft wird dies dadurch, dass für jede Problemdomäne ein Satz an Werkzeugen neu entwickelt werden muss, welche in den Bereichen Analyse, Verifikation, Optimierung, Parallelisierung und Transformation (AVOPT) eine automatisierte Überführung in ausführbaren Code ermöglichen.

[Mernik u. a. 2005] beschreiben einen Satz sich überlappender „Entscheidungsmuster“, die die grundlegenden Anliegen ökonomischerer Systementwicklung und der Ermöglichung der Modellierung für Personen mit geringen

Entwicklungs- und Programmierkenntnissen aufschlüsseln und Anwendungsfälle strukturieren, welche die Verwendung von DSL motivieren können:

- *Notation*: Eine domänenspezifische Sprache ermöglicht dabei entweder die Anreicherung einer existierenden API mit einer nutzerfreundlichen Notation oder beschreibt umgekehrt die Tranformation einer visuellen Notation in eine textuelle Syntax.

- *AVOPT*: Eine domänenspezifische Sprache kann so konstruiert sein, dass sie die Implementierung der Achsen Analyse, Verifikation, Optimierung, Parallelisierung und Transformation gegenüber einer GPL erleichtert. Im Hardware-Entwurf werden beispielsweise Modellierungssprachen wie VHDL eingesetzt, welche Parallelität modellierbar machen und gegenüber einer GPL manche der genannten Achsen erst ermöglicht.

- *Automatisierung*: Die Nutzung von GPL bedingt häufig eine umständliche, fehleranfällige Modellierung, welche durch eine ausdrucksstärkere DSL vereinfacht werden kann. Die redundanten Arbeitsschritte können automatisiert werden. [Jones 2006] stellt geläufige GPL und DSL gegenüber und bewertet diese mit einem *language level*, welcher linear mit der Produktivität korreliert. Hier erreichen DSL einen bis zu einen Faktor 10 höheren Wert als GPL.

- *Produktlinien*: Falls die Mitglieder einer Produktlinie eine gemeinsame Architektur und gemeinsame Komponenten teilen, kann eine DSL die Produktspezifikation aus diesen Bauteilen ermöglichen.

- *Traversierung von Datenstrukturen*: Das Durchlaufen und Durchsuchen komplexer Datenstrukturen wird durch geeignete DSL erleichtert. Ein Beispiel ist die *Structured Query Language (*SQL), mit welcher Teile von Datenbanken selektiert und manipuliert werden können

- *Systemfassade und Interaktion:* Domänenspezifische Sprachen können in eine Anwendung eingebettet sein, um dem Benutzer eine einfache Konfiguration zu ermöglichen.

Derzeit existieren mehrere wissenschaftliche und kommerzielle Ansätze für domänenspezifische Modellierung. Das im Rahmen dieser Arbeit verwendete Aquintos.GS (siehe Abschnitt 3.1.5) erlaubt die Definition von Datenmodellen und Transformationen. Einen ähnlichen Ansatz verfolgt das Projekt *Eclipse Modeling Foundation*, welches zur Metamodellierung dient. Es kann jedoch

durch weitere Plug-Ins wie *Graph Modeling Framework* und *OpenArchitectureWare* zu einem DSL-Werkzeug erweitert werden[9]. Als kommerzielle Produkte sind zusätzlich *MetaEdit+* der Firma *MetaCase*[10] und *Microsoft DSL-Tools*[11] zu nennen.

2.3.2.2 Bewertung

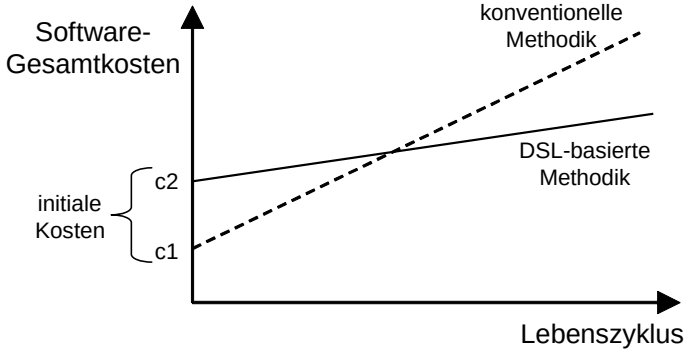

Abbildung 2.23: Amortisation von DSL-basiertem Vorgehen gegenüber konventionellen Methoden (nach [Hudak 1998])

Eine stark vereinfachte Betrachtung zur Abwägung zwischen konventionellem und domänenspezifischem Vorgehen zeigt Abbildung 2.23. Die Initialkosten eines domänenspezifischen Ansatzes können durchaus höher als bei einem traditionellen Entwicklungsszenario sein, da beispielsweise Entwicklungswerkzeuge nicht direkt benutzbar sind, sondern auf die DSL angepasst werden müssen und Entwickler sich mit der unbekannten Notation vertraut machen müssen.

Die aggregierten Softwarekosten werden jedoch ab einem gewissen Punkt im Lebenszyklus niedriger sein. Gründe hierfür sind die größere Nähe zur Anwendungsdomäne und die dadurch für die Problemstellung angepasste Abstraktion und Granularität. Einzelne Probleme werden nur einmal gelöst und diese Lösungen können über Modelle immer wieder neu kombiniert werden.

[Tolvanen u. a. 2002] verweisen auf Fallstudien, welche den Produktivitätsgewinn zu belegen scheinen: Nokia setzt domänenspezifische Modellierung und Codeerzeugung bei der Entwicklung von Mobiltelefonen ein und postuliert eine Multiplikation der Produktivität um den Faktor 10. Lucent Technologies

[9] Webseiten: http://www.eclipse.org/emf, http://www.eclipse.org/gmf und http://www.openarchitectureware.org

[10] Webseite: http://www.metacase.com/mep

[11] Webseite: http://msdn2.microsoft.com/en-us/teamsystem/aa718368.aspx

modelliert Produktfamilien und gewinnt dabei einem Faktor 3-10 in der Produktivität. In einem Forschungsvorhaben der US Air Force wurden 130 Entwicklungsaufgaben sowohl klassisch als auch über DSL gelöst. Es ergaben sich hier ein Produktivitätsgewinn um den Faktor 3 und 50% weniger auftretende Fehler.

2.3.3 CASE-Werkzeuge

Die zeitlich vor dem Aufkommen der UML und domänenspezischen Sprachen entstandenen CASE-Werkzeuge, stellen im industriellen Entwurfsumfeld eingebetteter System auch heute noch zentrale Entwurfswerkzeuge dar. CASE steht für *Computer Aided Software Engineering*, ein Synonym ist 4GL-Tool[12]. Es bezeichnet ein Werkzeug, welches eine (meist domänenspezifische) Modellierungssprache, Transformationen in Code und einen eingeschränkten Satz an Zielplattformen bündelt. Der Vorteil eines solchen Vorgehens ist die möglicherweise robustere Codeerzeugung im Vergleich zu domänenspezifischen Sprachen.

Kritisiert wird jedoch, dass zahlreiche Werkzeuge eben keine wirklich domänenspezifische Modellierung bieten und einen „One Size Fits All"-Ansatz verfolgen[13].

Im Umfeld eingebetteter Systeme im Automobil- und Automatisierungsbereich ist das CASE-Werkzeug *Mathworks Matlab Simulink/Stateflow*[14] weit verbreitet. Der folgenden Abschnitt gibt eine Übersicht über das Werkzeug.

2.3.3.1 Matlab Simulink / Stateflow

Das Programm Matlab entstand ursprünglich zur Beschreibung und numerischen Lösung rechnerischer Aufgabenstellungen, welche durch Matrizen beschreibbar sind. Hirzu kann das Paket *Simulink / Stateflow*, dessen Simulationsfähigkeiten später um Möglichkeiten zur Codeerzeugung erweitert wurden.

Simulink und *Stateflow* werden derzeit im Umfeld eingebetteter Systeme häufig zusammen mit der Generierung von C-Code verwendet.

Simulink ist dabei eine interaktive Umgebung zur Modellierung und Simulation mit einfach zu handhabenden Blockdiagrammen, welche im Falle eines

[12] 4-GL steht für *Fourth Generation Language*. Nach strukturierten (2-GL) und objektorientierten (3-GL) Programmiersprachen sollen 4-GL-Sprachen und Werkzeuge die nächste Stufe der Abstraktion bei der Programmierung bieten.

[13] siehe dazu auch [Stahl u. Völter 2005], S. 81f

[14] Homepage The Mathworks: http://www.mathworks.com

zeitkontinuierlichen Modells semantisch einen Satz nichtlinearer Differenzialgleichungen, bei einem zeitdiskreten Modell einen Satz von Differenzengleichungen beschreiben.

Mit *Stateflow* können innerhalb eines *Simulink*-Modells zusätzlich ereignisdiskrete Zustandsdiagramme integriert werden. *Stateflow* ist ein interaktives Entwurfswerkzeug für die Modellierung und Simulation eines ereignisgesteuerten Systems. *Stateflow* erweitert klassische Statecharts um Fähigkeiten zur Ablaufsteuerung, graphische Funktionen, zeitliche Operatoren, Einbettung in das Simulink-Modell und die automatische Generierung von C Code [Wang 2007].

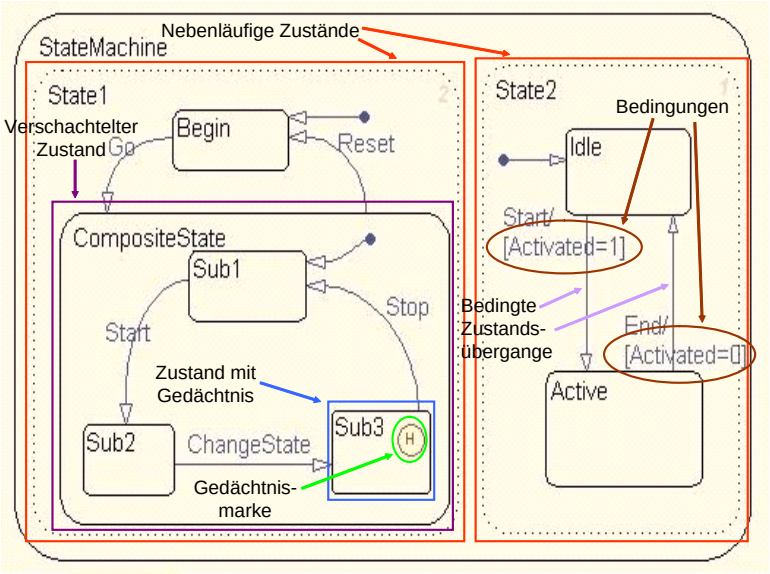

Abbildung 2.24: Zusätzliche Eigenschaften von StateFlow-Diagrammen

Im Vergleich zu UML Zustandsdiagrammen besitzen StateFlow-Diagramme eine leicht unterschiedliche Syntax und eine andere, teilweise zeitdiskrete Ausführungssemantik.

Abbildung 2.24 illustriert anhand eines StateFlow-Diagramms syntaktische Unterschiede. Bei den Zuständen *State1* und *State2* handelt es sich um parallele Zustände, d.h. sie sind gleichzeitig aktiv. Der Zustand *StateMachine* wird als And-Zustand bezeichnet. *CompositeState* ist ein verschachtelter Xor-Zustand (ebenso wie *StateMachine*, *State1* und *State2*), welcher eine Hierarchiestufe höher liegt als die in ihm enthaltenen Zustände *Sub1*, *Sub2* und *Sub3*.

Verbreitete Codegeneratoren, welche zur Erzeugung von C-Quelltext aus *Stateflow* dienen, sind *EmbeddedCoder* und *RealTimeWorkshop* von The Mathworks

und *TargetLink* der Firma dSpace[15]. In der vorliegenden Arbeit wurde *Real-TimeWorkshop* genutzt.

Aus Simulink- und Stateflow-Modellen kann auch synthetisierbares VHDL erzeugt werden. Dazu steht das Werkzeug *Simulink HDLCoder*[16] zur Verfügung. In dieser Arbeit, wurde aus Stateflow-Modellen mit dem Werkzeug *JVHDLGen* C-Quelltext erzeugt (siehe [Dreier u. a. 2003b]). Diese Möglichkeit wird in Abschnitt 6.2 beim Funktionsdebugging für konfigurierbare Logikbausteine genutzt.

[15]dSpace Webseite: http://www.dspace.com

[16]Simulink HDL Coder Webseite: http://www.mathworks.com/products/slhdlcoder

Kapitel 3

Ausführbare Verhaltensmodelle

Diese Arbeit verwendet Modellierungssprachen zur Programmierung. Dazu müssen Modelle ausführbar sein. Um ein Modell ausführen zu können, muss es sowohl Struktur als auch Verhalten des entworfenen Systems beschreiben. Abschnitt 3.1 beschreibt ausführbare Modelle ausgehend vom Umfeld modellgetriebener Softwareentwicklung. Dabei wird die Problematik der Heterogenität untersucht und auf die Plattform Aquintos.GS zur Modellintegration eingegangen.

Abschnitt 3.2 befasst sich eingehend mit der Nutzung von Verhaltensbeschreibungen auf Basis von UML *Actions*, welche im Bereich der UML nötig sind, um Modelle ausführen zu können. Neben der Einführung in *Actions* und das zugehörige Metamodell, wird auf den in dieser Arbeit genutzten und erweiterten Ansatz zur Codeerzeugung und auf die Nutzung von Java als Notation für UML *Actions* eingegangen.

3.1 Ausführbare Modelle

3.1.1 Modellgetriebene Softwareentwicklung

Die Modellgetriebene Softwareentwicklung (MDSD/MDSE[1]) bezeichnet ein Vorgehensmodell für die Entwicklung von Software, bei dem nicht direkt ausführbarer Programmcode entwickelt wird, sondern zunächst auf die Problem-

[1]MDSD: *Model Driven Software/Systems Development*MDSE: *Model Driven Software/-Systems Engineering*

stellung angepasste meist graphische Modelle erstellt werden [Stahl u. Völter 2005].

Das Modell dient dabei als Ausgangspunkt für den Softwareentwicklungsprozess. Treten Änderungen auf, wird das Modell geändert. Durch automatische Transformationen werden Modell und Code synchron gehalten. Informationen bezüglich der Anwendung oder der Systemumgebung werden im Modell statt im Code gepflegt.

Ein zentrales Element ist die Bereitstellung von Transformationstechnologien, welche es ermöglichen, das Modell über seinen Dokumentationscharakter hinaus zu verwerten.

3.1.2 Model Driven Architecture (MDA)

Die *Model Driven Architecture* MDA ist eine Kernstrategie der Object Management Group seit 2000 und gruppiert die weiteren Aktivitäten der OMG in ein Gesamtgerüst. Sie bildet eine spezielle Ausprägung modellgetriebener Entwicklung und ist Gegenstand von Standardisierungsbemühungen in diesem Bereich. Insbesondere die in den Standarddokumenten verwendeten Begriffe haben sich im Umfeld modellgetriebener Entwicklung durchgesetzt [Stahl u. Völter 2005].

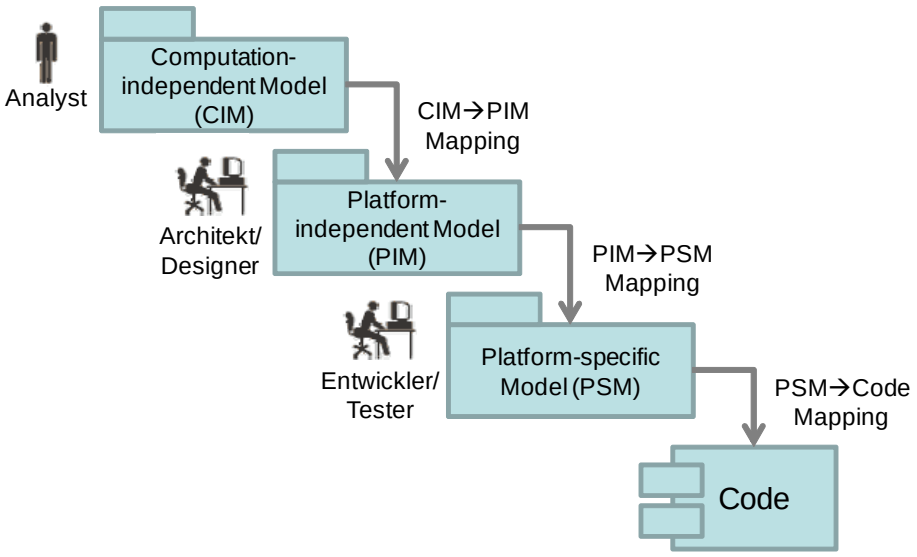

Abbildung 3.1: Grundprinzip der Model Driven Architecture [Dodani 2006]

Grundgedanke der MDA ist eine strikte Trennung zwischen Funktionalität und Technik. Diesen Grundgedanken verdeutlicht Abbildung 3.1: Gegenstand der

Modellierung ist ein Plattformunabhängiges Modell (PIM), welches zur Anpassung an verschiedene Plattformen in Plattformspezifische Modelle (PSM) überführt wird.

Die Abbildung kann automatisiert erfolgen, dies ist jedoch keine notwendige Bedingung. Der Entwurfsfluss sollte dabei soweit wie möglich nur in eine Richtung erfolgen; Änderungen im PSM und Rückführung dieser in das PIM sind jedoch vorgesehen. Auch besteht im Gegensatz zu CASE-Werkzeugen kein Anspruch auf vollständige Beschreibung durch das Modell und vollständige Generierung des Codes aus einem Modell.

Die MDA bündelt einige weitere Technologien der OMG:

- MOF: Die *Meta Object Facility* (siehe Abschnitt 2.2.3) bildet als Metametamodell das Rückgrat des Konzepts zur Definition von domänenspezifischen Modellierungssprachen oder zur Erstellung leichtgewichtiger Erweiterungen zur UML. Ergänzend zur MOF selbst können Technologien wie XMI und JMI eingesetzt werden.
- UML: Abwohl die OMG mit der MDA von der Vorgabe einer vereinheitlichten Modellierungssprache zur Systemspezifikation abgerückt ist, bleibt die *Unified Modeling Language* zentral und wird in zahlreichen Realisierungen der MDA verwendet. Alternativ kann auch nur die *UML Infrastructure* oder das *Diagram Interchange* Metamodell als Basis für eine eigene Modellierungssprache verwendet werden.
- Action-Sprachen zur Beschreibung von Modellverhalten. Siehe dazu Abschnitt 3.2.
- Modelltransformation

Die Spezifikation der MDA definiert folgende wichtige und weit verbreitete Begriffe [Object Management Group (OMG) 2003a]:

- **Plattform:** Eine Plattform ist eine Menge von Subsystemen und Technologien, welche eine kohärente Zusammenstellung von Funktionalität bieten. Dies geschieht über Schnittstellen und spezifizierte Nutzungsmuster, welche jede die Plattform nutzende Applikation ohne Wissen über die Realisierung der Plattform nutzen kann.
- ***Computation Independent Model* (CIM):** Ein CIM ist ein Blick auf das System aus einem von Rechensystemen unabhängigen Blickwinkel und zeigt keine Details über die funktionale Struktur des Systems. Es kann von einem Domänennutzer verstanden werden und beschreibt seine Nutzeranforderungen (*Requirements*).

- ***Platform Independent Model* (PIM):** Das PIM beschreibt die funktionale Realisierung des Systems, spart jedoch alle Aspekte des Systems aus, welche Kenntnis über die Zielplattform vorraus setzen.

- ***Platform Specific Model* (PSM):** Das PSM erweitert das PIM um Realisierungsdetails der verwendeten Zielplattform.

Der Begriff Plattformunabhängigkeit wird in der MDA qualitativ verwendet, ist aber kein absoluter Wert. Ein Modell wird Plattformunabhängigkeit immer nur zu einem gewissen Grad besitzen. Die Kette von einem PIM zum PSM kann beliebig viele Zwischenschritte umfassen.

3.1.3 Executable UML

Die *Executable UML* (kurz xUML oder xtUML) ist eine spezielle Ausprägung der MDA. Sie wurde von Stephen Mellor und Marc Balcer erstmals 2002 in [Mellor u. Balcer 2002] beschrieben.

Sie beschreibt die Konstruktion ausführbarer Modelle mit UML mit dem Ziel der Nutzung der UML als graphische Programmiersprache. Die *Executable UML* beschränkt dabei die Anzahl der verwendbaren Diagramme und Modellierungsartefakte auf eine überschaubare Zahl. Die Semantik der verbleibenden Modellierungsmittel wird fixiert und wo nötig leicht erweitert.

Die Executable UML nutzt das Paketdiagramm zur Identifizierung orthogonaler Aspekte des Systems, das Klassendiagramm zur Beschreibung der statischen Struktur und Zustandsdiagramme zur Beschreibung des Lebenszyklus von Klassen.

Ein weiterer wichtiger Bestandteil einer Realisierung der Executable UML ist eine Oberflächensprache für das *Actions*-Metamodell der UML (siehe Abschnitt 3.2). Mit ihrer Hilfe kann das Verhalten der Operationen im Klassendiagramm vollständig beschrieben werden.

Die semantische Präzisierung des UML Modells eröffnet einige interessante Möglichkeiten. Neben der Zugänglichkeit des Modells für formale Methoden zum Zweck der statischen Verifikation, kann das Modell simuliert und damit funktional validiert werden.

Aus dem Modell können jedoch auch direkt Prototypen für die Plattform erzeugt und ausgeführt werden. Werkzeuge, welche aus *Executable UML* Modellen ausführbaren Code erzeugen können, werden als Modell-Compiler bezeichnet.

UML-Modellierungswerkzeuge, welche die *Executable UML* realisieren sind unter anderem Telelogic Tau[2] und Kennedy Carter iUML[3].

3.1.4 Heterogenität

Der Begriff der Heterogenität umfasst im Umfeld der modellbasierten Entwicklung zwei Dimensionen: Unterschiedlichkeit der Beschreibungsmittel und der ihnen zugrunde liegenden Techniken und Verschiedenheit der untersuchten Zielplattformen.

Zum Entwurf eines umfangreichen eingebetteten Systems ist die Nutzung einer Modellierungssprache und eines Werkzeugs nicht ausreichend. Es bietet sich daher an, mehrere Möglichkeiten zur Modellierung zu kombinieren und miteinander zu verknüpfen. Darunter fallen unter anderem:

- Unified Modeling Language und Erweiterungen für modellgetriebene Softwareentwicklung und Codeerzeugung
- Datenflussorientierte Blockdiagramme: Reglungstechnische Entwurfsaufgaben werden mit Hilfe von Blockdiagrammen und Datenflussdiagrammen wie sie in MathWorks Matlab Simulink oder ETAS ASCET-SD[4] in CASE-Werkzeugen umgesetzt sind, modelliert.
- Reaktive Systemteile: Diese können effizient mit Statecharts (hierarchischen parallelen Zustandsautomaten) modelliert werden. Werkzeuge in diesem Umfeld sind Telelogic Rhapsody[5] und MathWorks Matlab Stateflow.
- Domänenspezifische MOF-basierte Modellierungsansätze

Häufig kann ein Übergang zu rein modellbasierter Entwicklung nicht in einem Schritt bewältigt werden. In zahlreichen Fällen ist dies auch nicht sinnvoll, da sich aus Gründen der Optimierung die Nutzung von Codeerzeugung nicht lohnt oder weil keine dem Entwurfsproblem angemessene Modellierungssprache existiert. Daher werden die meisten Projekte, wie in Abbildung 2.1 bereits angedeutet, teilweise auf Quellcode (*Legacy Code*) zurückgreifen.

[2]Telelogic Tau Webseite: http://www.telelogic.de/products/tau/index.cfm

[3]Kennedy Carter Webseite: http://www.kc.com

[4]ETAS ASCET Webseite: http://www.etas.com/de/products/ascet_software_products.php

[5]Telelogic Rhapsody Webseite: http://modeling.telelogic.com/products/rhapsody/index.cfm

Nachteil eines solchen Vorgehens ist jedoch die Preisgabe der Plattformunabhängigkeit, die in einem Modell durch die höhere Abstraktion gegeben ist. Lösung kann hier der Versuch sein, für die nicht modellierbaren Teile eine Schnittstelle zu definieren und sie in die Plattform zu integrieren. Die Erfahrung hat gezeigt, dass eine unidirektionale Abbildung vom Modell in den Code einer Lösung vorzuziehen ist, bei der spätere Änderungen im Code ins Modell zurück propagiert werden (sogenanntes *Round-Trip Engineering*) [Grönniger u. a. 2006].

Die zweite Dimension der Heterogenität bezieht sich auf die addressierten Zielsysteme. Eine Kernidee der MDA ist ja gerade die möglichst plattformunabhängige Spezifikation von Modellen. Für eingebettete Software ergibt sich damit ein großes Feld möglicher Plattformen, für welche möglichst große Teile eines Modells wieder verwendbar sein sollen:

- Verschiedene Zielsprachen (C, C++, Java)
- Unterschiedliche Betriebssysteme (Windows, Unix-Derivate, OSEK, proprietäre Echtzeitbetriebssysteme)
- Unterschiedliche Prozessoren (x86, ARM, Softcoreprozessoren, 8-64 Bit)
- Unterschiedliche Architekturen von Ausführungseinheiten (CPU in von-Neumann-Architektur, Digitaler Signalprozessor DSP in Harvard-Architektur, feingranular und grobgranular konfigurierbare Bausteine (zum Beispiel FPGA, siehe Abschnitt 6.2).

Ein Ansatz zur modellgetriebenen Entwicklung für eingebettete Software muss diese beiden Achsen der Heterogenität integrieren können. Die im folgenden Abschnitt beschriebene Plattform Aquintos.GS leistet diese Integration bis zur Generierung des Prototyps. Die in Kapitel 5 beschriebene Plattform *ModelScope* erweitert diese mit Fähigkeiten zur Modellausführung und zum graphischen Debugging auf Modellebene in dem beschriebenen heterogenen Umfeld.

3.1.5 Aquintos.GS

Aquintos.GS (früher: *GeneralStore*) ermöglicht einen durchgängig modellbasierten Entwurf. Mehrere Entwickler können zur gleichen Zeit und mit unterschiedlichen Werkzeugen an verschiedenen Teilen eines integrierten Modells arbeiten. Implementiert wurde eine Anbindung an zahlreiche UML-Werkzeuge, sowie an Matlab Simulink/Stateflow und Statemate. Unterschiedliche Aspekte

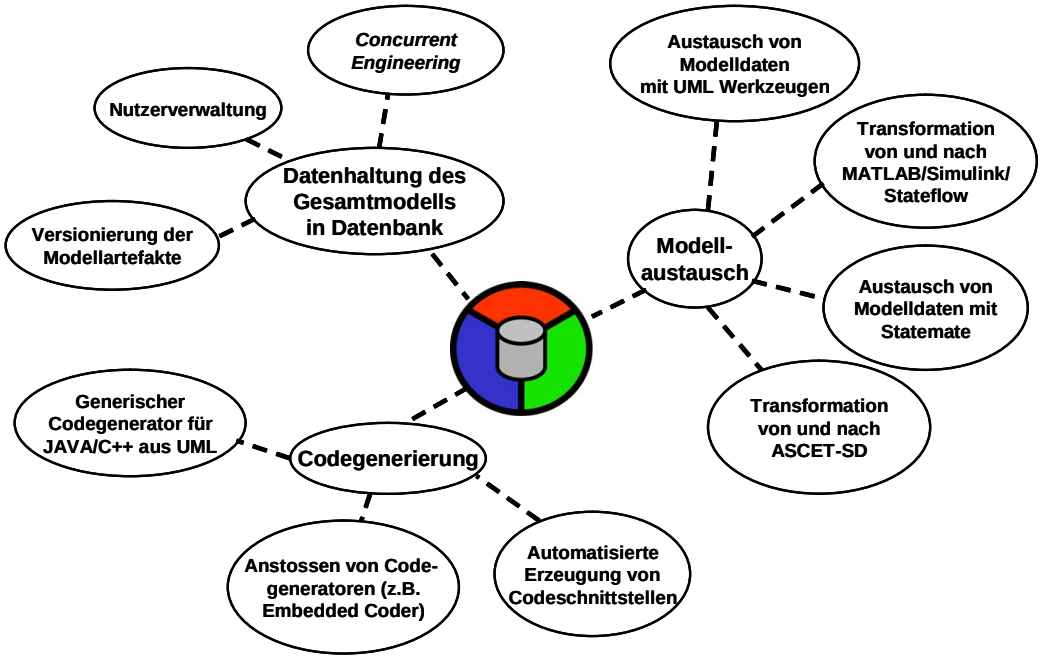

Abbildung 3.2: Überblick über Aquintos.GS

der Plattform sind in Abbildung 3.2 strukturiert und werden in [Müller-Glaser u. a. 2004; Reichmann u. a. 2004a,b, 2003a,b] genauer behandelt.

Um paralleles Arbeiten mehrerer Entwickler zu ermöglichen, ist die Modellverwaltung datenbankbasiert. Zudem implementiert Aquintos.GS Benutzerverwaltung und einen Versionierungsalgorithmus, der das Modell auf verschiedensten Granularitätsebenen verwalten kann.

Aquintos.GS ermöglicht die automatische Generierung von Schnittstellenklassen, die sowohl Simulink/Stateflow-, als auch Statemate-Modelle in transparenter Weise aus der UML-Welt zugänglich machen. Dies ist ein Ansatz, der verglichen mit manueller Programmierarbeit, deutlich unanfälliger gegenüber Fehlern und viel robuster gegenüber Änderungen in einzelnen Modellteilen ist.

Der Endpunkt eines durchgängig entworfenen Systems soll ein lauffähiger Prototyp sein. Aquintos.GS kann für UML-Modelle selbstständig und mit Hilfe anpassbarer Templates Code generieren. Für in Simulink/Stateflow oder Statemate modellierte Teilsysteme werden kommerzielle Codegeneratoren angesteuert. Die Kommunikation zwischen den Modellteilen geschieht über die beschriebenen automatisch generierten Schnittstellen.

Unterstützte Modellierungswerkzeuge sind:

- Softwaremodellierung mit UML: Austausch von Modelldaten über eine an zahlreiche CASE-Werkzeuge angepasste XMI-Schnittstelle.
- Ereignisdiskrete Modellierung: Austausch von Modelldaten mit Telelogic Statemate[6] und Matlab Stateflow.
- Signalfluss: Verwaltung von Modelldaten aus Matlab Simulink durch bidirektionale Transformation in das UML-Metamodell.

Eigenschaften der Modellverwaltung:

- Speicherung auf Basis des UML-Standards.
- Zentralisierte Datenhaltung des gesamten Modells: Modellierungsdaten bestehen nicht aus zahlreichen Dateien, sondern sind in einem zentralen Repository abgelegt.
- Concurrent Engineering: Teile des Modells können gesperrt und isoliert von verschiedenen Entwicklern gleichzeitig in den unterstützten Werkzeugen bearbeitet werden.
- Versionsverwaltung: Die Modellverwaltung protokolliert Veränderungen im Modell. Ein Einblick in bzw. eine Rückkehr zu einer älteren Version ist jederzeit möglich.

Codegenerierung und Modellkopplung:

- Codegenerierung für ereignisdiskrete und regelungstechnische Systemteile durch kommerzielle Codegeneratoren (z. B. Rhapsody in MicroC oder Embedded Coder).
- Template-gesteuerte Erzeugung von Quelltext aus dem UML-Modell durch ein eigenes Plug-In, das strukturelle und Verhaltensaspekte von UML-Modellen in Code umsetzt. Die Verhaltensspezifikation einzelner Methoden erfolgt mit Hilfe einer an Java angelehnten Modellierungssprache, aus der anschließend C++/C/Java-Code generiert wird.
- Kopplung der Modellierungsdomänen über automatische Generierung von Schnittstellenklassen aus generiertem Code [Reichmann u. a. 2003c].

[6] Telelogic Statemate Webseite: http://modeling.telelogic.com/products/statemate/index.cfm

Die Tragfähigkeit des Ansatzes wurde in einer industrienahen Fallstudie nachgewiesen [Reichmann u. a. 2005]. Für die vorliegende Arbeit, speziell die Plattform *ModelScope* bildet Aquintos.GS das grundlegende Fundament zur Modellverwaltung, zum Austausch mit Modellierungswerkzeugen und zur Codegenerierung.

3.2 Action Semantics

Durch Verhaltensmodelle wie Statecharts oder Blockdiagramme können reaktive und datenflusszentrierte Abläufe gut dargestellt werden. Das detaillierte Verhalten einzelner Operationen in Klassendiagrammen oder von Aktionen in Zustandsmaschinen der UML ist jedoch schlecht abbildbar. Aktivitätsdiagramme der UML sind dazu nur bedingt geeignet und dienen meist nur der Visualisierung grundlegender Algorithmen. Diese Situation führte im Umfeld der UML dazu, dass das Verhalten als Zeichenkette dem Modellelement in einer beliebigen Programmiersprache beigefügt werden kann, ein Vorgehen welches in aktuell marktgängigen Modellierungswerkzeugen verbreitet ist.

Damit ist die Verhaltensbeschreibung jedoch kein Teil des Modells mehr. Modellartefakte (z.B. Operationen oder Attribute) werden über ihren Namen referenziert, was dazu führt, dass bei Änderungen des Modells die Konsistenz nicht garantiert werden kann. Auch verliert man die Möglichkeit, das Modell durch formale oder semi-formale Methoden zu verifizieren, welche auch die dynamischen Aspekte beachten.

Bei Betrachtung ausführbarer Modelle, wie sie in Abschnitt 3.1 vorgestellt wurden, verliert man so einen wichtigen Teil der Plattformunabhängigkeit, nämlich die Abstraktion von der verwendeten Programmiersprache.

3.2.1 Verhaltensbeschreibung mit Actions

Das in der Version 1.5 der UML eingeführte Konzept der Modellierung mit *Actions* löst die beschriebenen Nachteile auf und integriert beliebige aktivitätsorientierte Verhaltensbeschreibungen in die UML. Der Begriff *Action Semantics* wird in diesem Umfeld synonym verwendet.

Für UML Actions wurde keine eigene Sprache definiert, es existiert also keine standardisierte konkrete Syntax. Festgelegt wurden aber Semantik und abstrakte Syntax. Dies bedeutet, dass Actions im UML Metamodell gemeinsam mit ihrer Bedeutung definiert sind, eine konkrete Notation jedoch nicht. Actions stehen dabei im Gegensatz zum sonstigen Vorgehen bei der Definition

der UML; die Artefakte des Metamodells, welche beispielsweise eine Zustandmaschine mitsamt Stellen und Transitionen beschreiben, besitzen eine klar definierte Syntax. Bei der Spezifikation der Action Semantics wurde bewusst darauf verzichtet, eine Notation zu schaffen oder die Nutzung einer bestehenden vorzuschreiben (siehe auch [Sunye u. a. 2001]).

Das Action Metamodell kann als niedrigere Abstraktionsebene verstanden werden, vergleichbar mit plattformunabhängigem Maschinencode oder dem Bytecode einer virtuellen Maschine in der Programmiersprache Java. Die Artefakte des Metamodells sind nur bedingt dazu geeignet, direkt mit ihnen zu modellieren. So setzt sich das Erhöhen einer Zählervariablen, welches in einer modernen Hochsprache mit einer Anweisung geschieht, aus insgesamt fünf Actions zusammen. Zuerst muss die entsprechende Variable gelesen werden, anschließend wird das Literal erzeugt, um das erhöht wird, dann wird addiert und schließlich muss das Ergebnis zurückgeschrieben werden.

Für UML-Entwicklungswerkzeuge, die die Action Semantics unterstützen hat dies zur Folge, dass zusätzlich eine *Action Language* (auch Oberflächensprache oder *Surface Language*) zu Verfügung gestellt werden muss, in der Verhalten spezifiziert werden kann und für die eine (notwendigerweise weder injektive noch surjektive) Abbildung auf das Action Metamodell existiert. Ein mit Hilfe der Action Language formuliertes Konstrukt wird im Allgemeinen mächtige, sich aus mehreren Actions zusammensetzende Befehle anbieten. Dabei ist eine Abbildung aus allen geläufigen imperativen *General Purpose Languages* (z.B. Java, C++) möglich. Eine Abbildung aus Java wurde im Rahmen dieser Arbeit entwickelt und wird in Abschnitt 3.2.4 vorgestellt. Im akademischen Umfeld wurden weitere textuelle Oberflächensprachen entworfen und vorgestellt, so SMALL von Mellor et al. und TALL von Balcer et al. (s. [Mellor u. Balcer 2002]). Diese fokussieren wie die Action Semantics selbst anstelle des Kontrollflusses auf den Datenfluss und unterstützen daher die Modellierung von Parallelität. Auch domänenspezifische graphische oder textbasierte Beschreibungssprachen, welche beispielsweise signalverarbeitende Algorithmen modellieren, können auf UML Actions abgebildet werden. Ein weiteres Beispiel ist die in Abschnitt 5.1.2 vorgestellte Modellierung von Werkzeugketten, welche mit Nutzung von UML Actions in das Gesamtmodell integriert wird.

Die Integration von Actions in die UML ermöglicht es, UML-Modelle universell ausführbar zu machen. Modelle können simuliert und getestet werden; Code für unterschiedlichste Zielplattformen kann automatisch generiert werden. Eine anpassbare Codeerzeugung aus dem Action-Modell nach Java wurde im Rahmen dieser Arbeit entwickelt und in das Gesamtsystem integriert. Abschnitt 3.2.3 beschreibt das Vorgehen genauer. Eine Generierung für andere Sprachen und Programmierparadigmen (z.B. VHDL) ist möglich.

Durch die Integration von Actions in das UML-Metamodell wird die Interoperabilität von UML-Entwicklungswerkzeugen erhöht. Die vom Benutzer angefertigten Modelle mitsamt ihrer Actions können in andere Programme geladen werden und zum Beispiel die dort vorhandene Codeerzeugung nutzen, auch wenn eine andere Oberflächensprache genutzt wird.

Weitere Einsatzmöglichkeiten bieten sich bei der Modelltransformation. Da das UML-Metamodell selbst durch ein UML-Modell repräsentiert wird, können die Actions genutzt werden, um UML-Modelle zu verändern. [Sunye u. a. 2001] liefert Beispiele zur Modelltransformation, wie das Refactoring von Modellen oder dem Einsatz von Design Patterns, mit Action Semantics.

Mellor führt in [Mellor u. a. 1998] folgende Argumente zur Begründung der Notwendigkeit von Action Semantics in der UML an:

- Das Modell kann auf einer höheren Abstraktionsebene als in einer Programmiersprache erstellbar sein, ist aber dennoch vollständig und präzise.
- Formale Nachweise der Korrektheit der Verhaltens (nicht nur der Struktur) sind prinzipiell möglich.
- Das Modell kann exakt simuliert und verifiziert werden, es ist also ausführbar.
- Modelle sind wieder verwendbar.
- Eine automatisierte Codeerzeugung für unterschiedliche Zielplattformen aus dem Modell wird möglich.

Diese Vorteile fügen sich auch in das in Abschnitt 3.1.2 beschriebene Konzept der *Model Driven Architecture* (MDA) der OMG zur modellgetriebenen Softwareentwicklung. Actions ermöglichen eine plattformunabhängige, vollständige und konsistente Modellierung, welche die Vorraussetzung für die Definition eines von technischen Aspekten abgelösten plattformunabhängigen Modells im Sinne der MDA bildet. Um Transformationen aus dem Modell in ausführbaren Code zu ermöglichen, ist es essentiell, das Verhalten von Modellelementen exakt wie in einer Programmiersprache beschreiben zu können.

3.2.2 Das Actions Metamodell

Der folgende Abschnitt bietet einen Abriss über die Verhaltensbeschreibung mit Actions und stellt die notwendigen Konstrukte vor. Zunächst werden die

grundlegenden Konzepte und die semantischen Unterschiede zu herkömmlichen Programmiersprachen beschrieben. Darauf aufbauend werden neben der Struktur des Actions Paketes die konkreten Artefakte, welche in der UML 1.5 zur Verfügung stehen, klassifiziert und kurz vorgestellt. Für detaillierte Informationen sei hierbei auf die Spezifikation verwiesen [Object Management Group (OMG) 2003b]. Im letzten Unterabschnitt wird schließlich auf Details der Weiterentwicklung der Action Semantics der UML in der Version 2.1 eingegangen.

3.2.2.1 Grundlegende Prinzipien

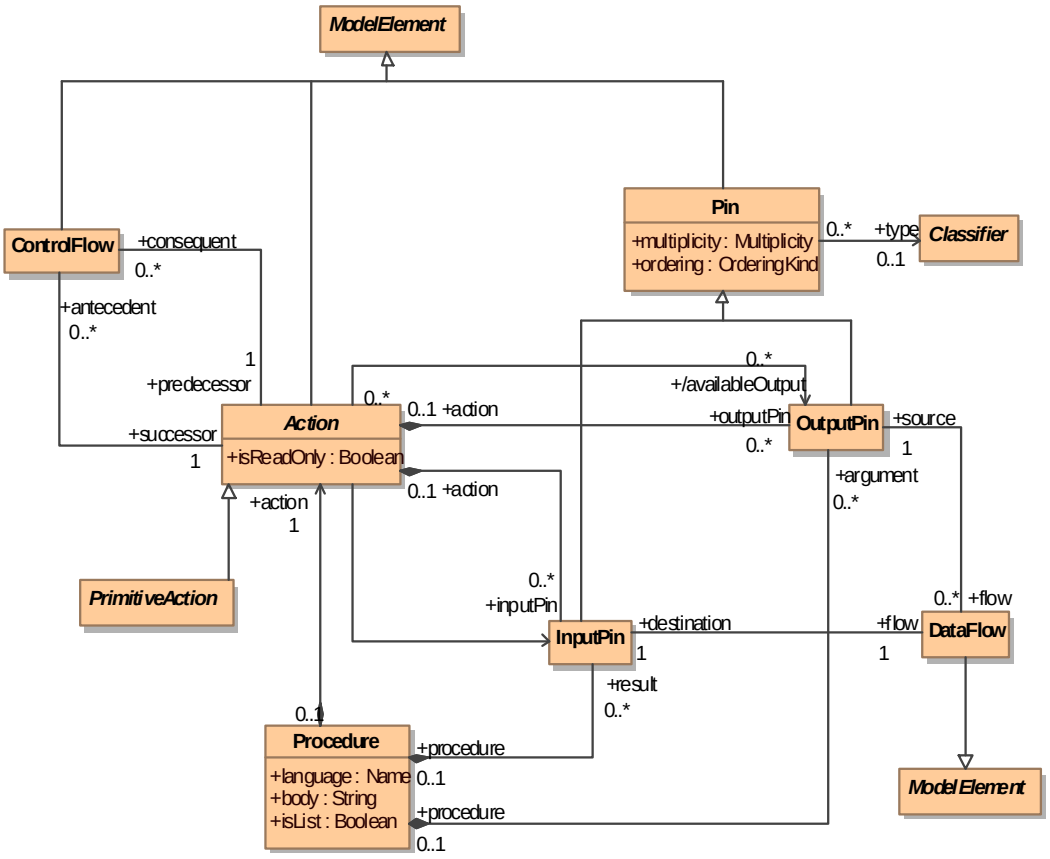

Abbildung 3.3: Modell der Action Foundation

Abbildung 3.3 zeigt das Paket *Action Foundation* aus dem UML Action Metamodell. Es definiert die Grundlagen der Action Semantics.

Actions stellen die elementaren Einheiten für Verhaltensbeschreibungen dar, die wie gezeigt innerhalb von Prozeduren (*Procedures*) geschehen. Ihre grundle-

gende Aufgabe besteht darin, typisierte Eingangswerte, die über Eingabepins (*InputPin*) anliegen, zu verarbeiten. Ergebnisse der Aktionen werden durch Ausgabepins (*OutputPin*) nach außen gegeben und können durch weitere Aktionen verarbeitet werden. Abbildung 3.4 zeigt eine sehr einfache graphische Darstellung dieses Sachverhalts. Diese soll im Folgenden in Beispielen zur Illustration als einfache Oberflächensprache dienen.

Werte werden von Ausgabepins zu Eingabepins immer über Datenflüsse (*DataFlow*) weitergegeben, welche ein Ausgangspin als Quelle mit einem Eingangspin als Ziel verbindet. Die Typen der verbundenen Pins müssen gleich oder zumindest kompatibel sein. Es ist ein sogenannte *fan-out* möglich: Jedes Ausgangspin kann sein Ergebnis beliebig vielen Eingangspins zur Verfügung stellen. Pins, welche innerhalb einer Verbindung von mehreren *Actions* nicht über Datenfluss miteinander verbunden werden, können von *Actions* als verfügbare Ein- bzw. Ausgaben (*availableInput/availableOutput*) deklariert werden und ermöglichen dadurch eine Kapselung, indem von einer angenommenen Gruppierung von untereinander verbundenen *Actions* die verfügbaren Ein- bzw. Ausgaben zur Weitergabe von Werten verwendet werden, ohne dabei die einzelnen Unteraktionen auf ihre verfügbaren Pins hin untersuchen zu müssen.

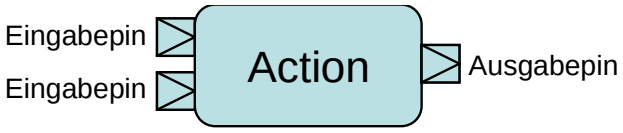

Abbildung 3.4: Beispiel für eine Action mit zwei Eingabepins und einem Ausgabepin

Neben dem Datenfluss, welcher die Reihenfolge der Ausführung von *Actions* auf der Ebene der für die Durchführung notwendigen Eingangswerte begrenzt, bietet die *Action Foundation* den Kontrollfluss (*ControlFlow*), mit dessen Hilfe *Actions* explizit untereinander in Vorgänger-Nachfolger-Beziehungen gesetzt werden können. Dies ermöglicht auf einfache Art, Abläufe aus herkömmlichen sequentiellen Programmiersprachen abzubilden. Nimmt eine Action an einem Kontrollfluss teil, so kann sie erst ausgeführt werden, wenn einerseits die benötigten Werte über Datenflüsse an den Eingabepins liegen und andererseits sämtliche über Kontrollflüsse verbundene Vorgänger-*Actions* abgeschlossen sind. Dadurch entsteht eine Sequenz aus sonst hinsichtlich Ausführungszeitpunkt voneinander unabhängigen *Actions*.

In folgenden werden die grundlegenden Prinzipien und Modellelemente des Action Metamodells genauer beschrieben:

3.2.2.1.1 Pins Die Aufgabe einer Action in der UML ist die einer Einheit, welche Eingabewerte zu Ausgabewerten weiterverarbeitet. Die Eingabewerte liegen an Input-Pins an. Ausgabewerte werden an Output-Pins weitergegeben. Die Pins sind ähnlich wie Assoziationsenden typisiert und besitzen einen Datentyp sowie eine Multiplizität. Der Datentyp kann ein beiliebiger Classifier sein, also einen primitiven Datentyp, eine Aufzählung oder eine Klasse sein. Über die Multiplizität wird die Menge der parallel verarbeiteten Instanzen des Datentyps bezeichnet. Über das Attribut *ordering* kann angegeben werden, ob die Werte in einer festen Reihenfolge übergeben werden.

Um zusammengesetzte Actions sinnvoll kapseln zu können, kann auf Ebene der umgebenden Action angegeben werden, welche internen Input- und Output-Pins nach außen sichtbar sind. Dies geschieht über die Referenzen *availableInput* und *availableOutput*, mit denen die entsprechenden Pins der enthaltenen Action, welche die Schnittstelle der *Composite Action* darstellen sollen im Modell gekennzeichnet werden.

3.2.2.1.2 Datenfluss Output- und Input-Pins werden über Datenflüsse miteinander verknüpft. Im Metamodell ist hierfür die Metaklasse *DataFlow* vorgesehen. Ein Output-Pin kann mit mehreren Input-Pins verbunden werden. Input-Pins erhalten nur einmal genau einen Wert, der sich im Verlauf nicht ändern kann. Der Typ eines Output-Pins, der einen Datenfluss zu einem Input-Pin hat, muss den gleichen oder einen kompatiblen Typ des Input-Pins haben. Weiterhin müssen sich die beiden Multiplizitäten von Input- und Output-Pin entsprechen. Dies bedeutet, dass die Multiplizität, welche für ein Output-Pin spezifiziert ist, auch für den angeschlossenen Input-Pin valide sein muss.

3.2.2.1.3 Kontrollfluss Über den im folgenden Abschnitt beschriebenen Lebenszyklus der Actions stellt der Datenfluss eine implizite Ausführungsreihenfolge der Actions innerhalb einer Prozedur her: Mit der Ausführung einer Action kann erst dann begonnen werden, wenn alle Eingaben an den Input-Pins zur Verfügung stehen. Ansonsten können alle Actions in beliebiger Reihenfolge ausgeführt werden. Möchte man die Ausführungsreihenfolge weiter beschränken, kann der Kontrollfluß explizit modelliert werden. Dies geschieht mit Hilfe der Metaklasse *ControlFlow*, welche zwei Actions eine Reihenfolge aufprägen kann. Der Nachfolger (*successor*) wird erst dann zur Ausführung gelangen, wenn die Ausführung seines Vorgängers (*predecssor*) abgeschlossen wurde.

3.2.2.1.4 Ausführung der Actions Prinzipiell können alle Aktionen parallel ausgeführt werden, soweit es die Einschränkungen durch Daten- und Kontrollflussbeziehungen zulassen. Dem Grundgedanken folgend, dass eine Sequentiali-

sierung parallel spezifizierten Verhaltens bedeutend einfacher als nachträgliche Parallelisierung ist, wird die Überspezifizierung von Kontrollfluss, wie sie in prozeduralen Programmiersprachen durch die sequentielle Abarbeitung jeder Anweisung zu finden ist, umgangen. Trotzdem ist für die Umsetzung der Action Semantics in eine prozedurale Programmiersprache die Bestimmung einer Ausführungsreihenfolge ein wichtiger Aspekt.

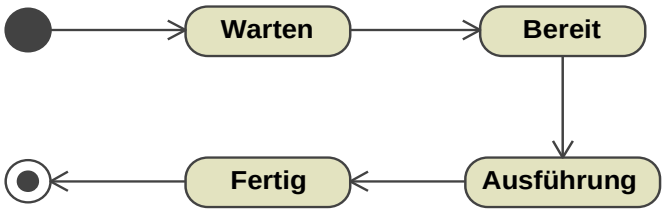

Abbildung 3.5: Lebenszyklus einer Action als Aktivitätsdiagramm nach [Object Management Group (OMG) 2003b]

Für die Ausführung von *Actions* wird in der UML-Spezifikation der Lebenzyklus einer Action festgelegt (s. Abbildung 3.5). Der Lebenszyklus einer Action durchläuft folgende Zustände:

- **Warten**: Eine *Action* befindet sich nach Erzeugung zu Beginn der Ausführung einer umhüllenden Prozedur im wartenden Zustand. Es liegen keine oder nicht alle benötigten Werte an ihren Eingabepins.
- **Bereit**: Sobald alle *Actions*, deren Ausgabepins über Datenfluss mit den Eingabepins der betrachteten *Action* verbunden sind, die Ergebnisse an ihre Ausgabepins gelegt haben und alle Vorgänger-*Actions*, die mit der betrachteten *Action* über Kontrollfluss zusammenhängen, ihre Ausführung beendet haben, wechselt die betrachtete *Action* in den Zustand „Bereit“.
- **Ausführung**: Eine sich im Zustand „bereit“ befindende *Action* kann zu jedem Zeitpunkt ausgeführt werden. Weder dieser Zeitpunkt noch eine mögliche Zeitverzögerung zwischen den Wechseln in den Zustand „Bereit“ und in den Zustand „Ausführung“ sind spezifiziert.
- **Fertig**: Diesen Zustand erreicht eine *Action*, sobald sie die Ausführung abgeschlossen hat. In diesem Fall liegen an allen Aus- und Eingabepins (soweit vorhanden) die Ergebnisse der Ausführung.

Der Lebenszyklus beinhaltet keine Schleifen. Eine Aktion kann nur einmal ausgeführt werden. Daher können sich die Werte von Ausgangspins, mit Ausnahme von Bedingungen in Schleifenstrukturen nie ändern.

3.2.2.2 Typen von Actions

Das *Actions* Package der UML 1.5 unterteilt sich in sechs Gruppen. Zusätzlich existiert die *Action Foundation*, die für sämtliche Gruppen grundlegende Artefakte vorstellt. Die Struktur des *Actions Package* lässt sich anhand der Abbildung 3.6 nachvollziehen.

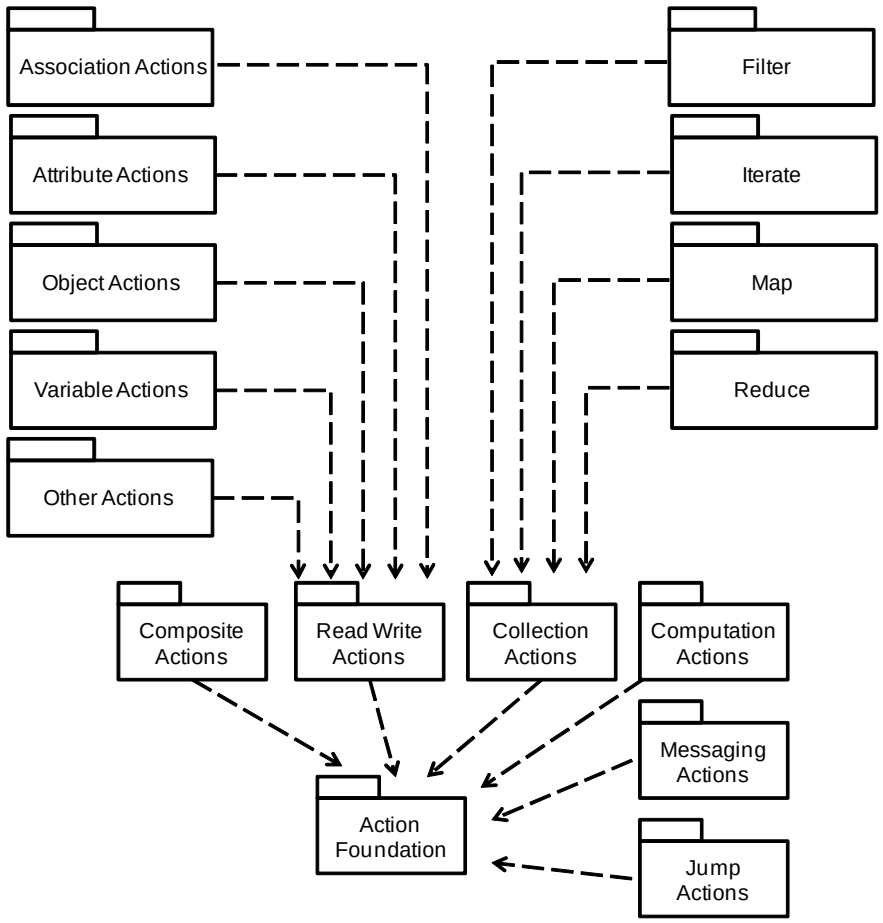

Abbildung 3.6: Actions Package

Die Unterpakete klassifizieren die Actions in unterschiedliche Typen. Es wird dabei kurz die jeweilige Funktion beschrieben. Für eine ausführliche Darstellung aller Aktionen und ihrer Effekte sei auf [Object Management Group (OMG) 2003b] verwiesen.

- **Composite Actions:** Composite Actions gruppieren mehrere atomare oder wieder zusammengesetzte Actions über eine „enthält"-Beziehung und stellen Funktionalität zur Ausführungswiederholung mit Schleifen

(ähnlich *for*, *while, do while* in Java) und zur bedingter Ausführung (ähnlich *if*) zur Verfügung.

- **Read und Write Actions:** Unter *Read and Write Actions* befinden sich Aktionen, die auf Attribute, Variablen, Assoziation und Objekte lesend und schreibend zugreifen können. Dabei werden beim Lesen Werte als *OutputPin* anderen Actions zur Verfügung gestellt. Beim Schreiben in Attribute und Variablen, werden Werte, die an *InputPins* anliegen verwendet. Links können als Instanzen von Assoziationen erzeugt und mit Objekten verknüpft werden.

- **Computation Actions**: Computation Actions nehmen Eingabewerte über *InputPins* entgegen und produzieren für ihre *OutputPins* daraus Ausgabewerte. Die Berechnungen sind reine Funktionen; Ausgabewerte hängen nur von den Eingabewerten und nicht vom Zustand des Systems ab. Neben der Anwendung primitiver Funktionen, finden sich in diesem Paket Vergleichsoperationen (*TestIdentityAction*) und Aktionen zur Bereitstellung von Konstanten (*LiteralValueAction*). Die verfügbaren primitiven Funktionen sind nicht spezifiziert, können aber mittels der *PrimitiveFunction*-Metaklasse im Modell definiert werden.

- **Collection Actions:** *Collection Actions* erlauben die Anwendung einer Unteraktion auf eine beliebige Datenstruktur gleichartiger Elemente, genannt *Collection*. Explizite Indexbildung und Extraktion der einzelnen Elemente werden dabei vermieden, um Kontrollfluss nicht überzuspezifizieren und damit eine mögliche Parallelisierung zu erschweren. Die Eingabe einer *CollectionAction* besitzt immer ein Multiplizität größer als Eins. Auf jedes, als *Slice* bezeichnetes Element der Eingabe wird die Unteraktion angewendet. Konkrete Aktionen beinhalten neben der allgemeinen *MapAction* Iteration (*IterateAction*), Filterung (*FilterAction*) und Reduktion der Eingabemenge auf ein Element (*ReduceAction*).

- **Messaging Actions:** Mit Messaging Actions können Nachrichten zwischen Objekten ausgetauscht werden, welche beim Empfänger ein bestimmtes Verhalten auslösen. Der Empfänger kann eine eingegangene Nachricht mit dem Ausführen einer Prozedur oder mit dem Auslösen eines Zustandsautomaten behandeln. Der Sender der Nachricht kann direkt nach dem Senden auf eine Antwort warten (synchroner Aufruf, vergleichbar mit einem Operationsaufruf in sequentiellen Programmiersprachen) oder direkt mit der Ausführung fortfahren (asynchroner Aufruf). Neben dem Aufruf von Operationen in Klassen (*CallOperationAction*), können Signale an einzelne (*SendSignalAction*) oder viele Empfänger (*BroadCastSignalAction*) gesendet werden.

- **Jump Actions:** Der normale Kontrollfluss kann mit Hilfe von *Jump Actions* unterbrochen werden. Jump Actions sind allgemeiner Art, so dass sich die Funktionalität, wie sie aus modernen Programmiersprachen bekannt ist mit ihnen realisieren lässt. Neben *break* und *continue*-Anweisungen, finden *Jump Actions* als Ausnahmebehandlungsmechanismus ihre Anwendung.

3.2.2.3 Verknüpfung mit dem Gesamtmodell

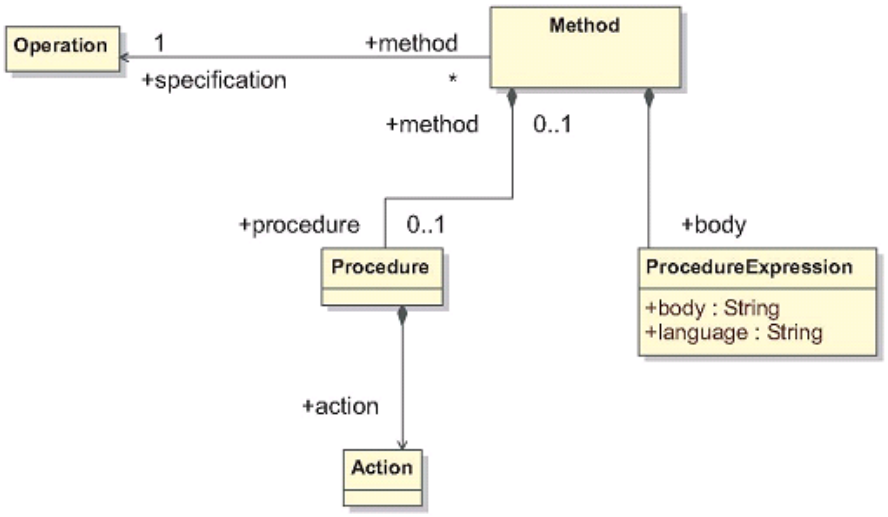

Abbildung 3.7: Actions im Kontext des Gesamtmodells

Abbildung 3.7 zeigt wie Actions in das restliche UML-Metamodell eingebettet sind. In der UML gibt es zwei Modellelemente zur Beschreibung von Verhalten: Operationen und Methoden. Eine Operation stellt einen Dienst zur Verfügung, der von einem Objekt in Anspruch genommen werden kann, um ein bestimmtes Verhalten zu erzielen. Eine Methode stellt die konkrete Implementierung einer Operation dar.

Eine Methode ist mit einer *ProcedureExpression* verbunden, in der ihre konkrete Implementierung in einer beliebigen Programmiersprache als String abgelegt werden kann. Auf diese Weise konnte vor Erweiterung der UML um Actions das Verhalten einer Operation außerhalb der UML festgelegt werden. Die Verhaltensbeschreibung kann nun aber auch mit Hilfe der UML Action Semantics stattfinden, indem einer Methode eine Prozedur zugeordnet wird, der wiederum eine Action zugeordnet ist. Prozeduren und damit auch Actions können neben der Beschreibung von Verhalten von Methoden auch zur Definition von durch UML-Zustandsmaschinen ausgelöste Aktionen genutzt werden.

3.2.2.4 Neuerungen bei Verhaltensdarstellung unter UML 2.1

Der Versionswechsel der UML von 1.5 nach 2.1 spiegelt sich in einer Vielzahl von grundlegenden Veränderungen in der Spezifikation wieder. In diesem Abschnitt soll auf die Veränderungen, die das Konzept der *Actions* betreffen, eingegangen werden und deren Bedeutung für die vorliegende Arbeit analysiert werden (siehe auch [Koschuhar 2007]).

In der UML 2.1 dienen Aktivitäten (*Activities*) allgemein der Abbildung von prozeduralen und signalflussorientierten Verhaltensbeschreibungen mittels Daten- sowie Kontrollfluss. Die Nutzung von Actions durchzieht dabei das gesamte Metamodell. Weiterhin wurde die Unterteilung von Aktivitäten und Action weitestgehend aufgehoben: Die Ausführung des Verhaltens wird von den dafür geeigneten *Actions* übernommen, welche in der Aktivität enthalten sind. Im Aktivitäten-Paket der Spezifikation werden Aktivitätsknoten und –kanten definiert, welche auch Strukturen zwischen den sonst konkurrierenden Aktivitäten etablieren. Es gibt insgesamt sieben Aktivitätsschichten:

- *FundamentalActivities* definiert Aktivitäten, welche Aktionen enthalten.
- In der Basisschicht (*BasicActivities*) werden Kontroll- und Datenflüsse zwischen *Actions* festgelegt. Dabei wird den Aktivitätskanten ihre Bedeutung als Kontroll- oder Datenfluss zugewiesen.
- Die Zwischenschicht (*IntermediateActivities*) ermöglicht die Modellierung mit Aktivitätsdiagrammen und Petri-Netzen und definiert nebenläufigen Kontroll- und Datenfluss sowie Entscheidungen.
- Die vollständigen Aktivitäten (*CompleteActivities*) erweitern die oben genannten Schichten um Kantengewichte und Flüsse.
- *StructuredActivities* erweitern die Aktivitätsknoten um Schleifen, Sequenzen und Bedingungen.
- Die vollständige Strukturschicht (*CompleteStructuredActivities*) reichert die Strukturschicht um die Unterstützung von Ausgabepins an.
- Die Sonderstrukturschicht (*ExtraStructuredActivities*) definiert die Modellierung von Ausnahmeverhalten (*Exceptions*).

Das gesamte Verhalten wird demnach mit einem Netz bestehend aus Aktivitätsknoten beschrieben. Dieses besteht aus Anfangsknoten, ausführbaren Knoten und Endknoten sowie Aktivitätskanten, welche die Knoten entweder als Daten- oder als Kontrollfluss verbinden.

Eine *Action* bildet in UML 2.1 nur noch einen einzelnen Schritt innerhalb einer Aktivität (*Activity*), welcher0 nicht weiter in Aktivitäten aufgelöst werden kann. *Actions* übernehmen nur noch atomare Aufgaben, wie die Beschreibung von Zugriffen auf Objekte, Variablen und Attribute sowie von Aufrufen von anderen *Actions* oder Aktivitäten.

Die *Actions* sind dabei in drei Schichten gruppiert: die grundlegende Schicht (*Basic Concepts*) spezifiziert *Actions* mit Ein- und Ausgabepins sowie Konstrukte, welche Operationsaufrufe und Signalübermittlungen durchführen, und nutzt dafür Elemente des *Kernels*. Darauf aufbauend geben die Elemente der Zwischenschicht (*Intermediate Concepts*) dem Modellierer neben den primitiven Funktionen die Möglichkeit, die Modellstruktur optimal im Hinblick auf Performanz in verschiedene parallele Teilprozesse zu unterteilen, ohne dabei das modellierte Verhalten zu verändern. Die strukturierte Schicht (*Structured Concepts*) enthält diejenigen *Actions*, deren Ausführung unmittelbar im Kontext von Aktivitäten und Aktivitätsknoten stattfindet. Sie werden als *CompleteActions* bezeichnet und bauen auf der grundlegenden und der Zwischenschicht auf.

Im Vergleich zu der in der vorliegenden Arbeit verwendeten Modellierung auf Basis von Actions der UML 1.5, besteht die wichtigste Änderung in UML 2.1 im Transfer prozeduraler Konstruktionen wie Schleifen oder Bedingungen in den Verantwortungsbereich von Aktivitäten. *Actions* dienen nun nur noch dem grundlegenden Zugriff auf Objekte und Aufrufen. Variablen samt Gruppen sind ebenfalls bei Aktivitätsknoten angesiedelt. Daten- und Kontrollfluss werden in UML 2.1 auf Aktivitäts- und nicht mehr auf Action-Ebene definiert.

Actions, die unter UML 2.1 nicht mehr Teil des *Actions Package* sind, wie es beispielsweise bei *ConditionalAction* in UML 1.5 der Fall ist, werden nun als *ConditionalNode* abgebildet, wobei die Syntax hinsichtlich der *Clauses* ähnlich geblieben ist, wogegen im Falle von *LoopActions* durch das Fehlen eines *Clauses* nicht einmal syntaktische Ähnlichkeit zwischen *LoopAction* und der korrespondierenden *LoopNode* festgestellt werden kann.

Schlussendlich lassen sich die im Rahmen dieser Arbeit entwickelten Konzepte zur Arbeit mit Actions jedoch von UML 1.5 auf Aktivitäten und *Actions* aus UML 2.1 übertragen. Der signalflussorientierte Leitgedanke mit Datenfluss über Pins und Kontrollfluss in einer abgewandelten Form besteht auch in UML 2.1 fort. Auch die überwiegende Mehrzahl der Konzepte der Aktionen bleibt erhalten oder erhält ein korrespondierendes Modellartefakt in der UML 2.1. Das neue Metamodell ist jedoch umfassend inkompatibel zum bisherigen Aufbau. Auch durch die Zusammenfassung von Aktionen und Aktivitäten der UML ergeben sich strukturelle Veränderungen im Modell, die eine Überarbeitung der Transformationen notwendig machen.

3.2.3 Codeerzeugung aus dem Action-Modell

Um eine vollständige Ausführbarkeit im Sinne der *Executable UML* (siehe Abschnitt 3.1.3) zu erreichen, muss die Modellierungssprache die vollständige Beschreibung von Verhalten ermöglichen. Dies wird in der UML durch das *Action*-Metamodell erreicht, durch welches Verhalten präzise und plattformunabhängig als Teil des Gesamtmodells beschrieben werden kann.

Um die Verhaltensbeschreibung tatsächlich ausführbar zu machen, muss eine automatische Codeerzeugung für Programmiersprachen von Zielsystemen konzipiert werden. Dies geschah im Rahmen dieser Arbeit. Die erarbeiteten und im folgenden vorgestellten Methoden wurden in der in Kapitel 5 vorgestellten Plattform *ModelScope* realisiert.

3.2.3.1 Sequentialisierung

Die Codeerzeugung bedeutet eine Übersetzung des inhärent parallelen Ausführungsparadigmas auf eine sequentielle Ausführung durch einen Mikroprozessor. Actions sind, wie in Abschnitt 3.2.2.1 beschrieben, im Action-Metamodell durch Daten- sowie Kontrollflüsse verbunden, welche Randbedingungen an die mögliche Ablaufreihenfolge der Aktionen definieren. In einer imperativen Programmiersprache werden einzelne Anweisungen nacheinander abgearbeitet. Eine Parallelisierung auf mehrere Ausführungseinheiten geschieht durch den Compiler oder zur Laufzeit durch einen Interpreter oder ein Laufzeitsystem.

Eine Verhaltensbeschreibung mit Actions innerhalb einer Prozedur besteht aus einem gerichteten azyklischen Graphen, dessen Knoten durch die Actions gebildet werden und dessen Kanten die einzelnen Kontroll- und Datenflüsse sind[7]. Abbildung 3.8 zeigt ein in [Gutstein 2007] erarbeitetes Beispiel in der graphischen Darstellung aus Abbildung 3.4. Die durchgezogenenen Kanten stellen dabei Datenflüsse dar; die gestrichelten Kanten Kontrollflüsse. Im Beispiel sind die Actions in einer *Group Action* zusammengefasst.

Um die Ausführungsreihenfolge von Actions innerhalb einer *GroupAction* zu bestimmen, wird aus ihnen ein Abhängigkeitsgraph erzeugt. Der Abhängigkeitsgraph für das beschriebene Beispiel ist in Abbildung 3.9 gezeigt. Eine Kante von Action A nach Action B bezeichnet die Existenz einen Kontroll- oder Datenflusses. Action A muss daher vor Action B abgearbeitet werden. Der

[7] Wie in Abschnitt 3.2.2.1.4 beschrieben durchläuft eine Action die Zustände ihres Lebenszyklus genau einmal. Ihre Ausgänge werden berechnet und ändern sich dann nicht mehr. Daher bilden die Actions einer Prozedur einen azyklischen Graphen, um unauflösbare Kontrollstrukturen zu vermeiden (vgl. auch [Object Management Group (OMG) 2003b], Seite 2-220).

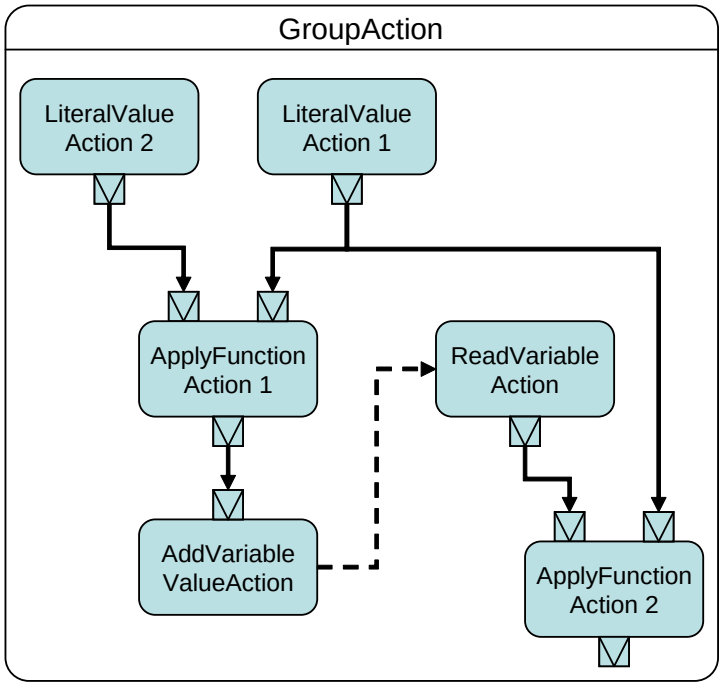

Abbildung 3.8: Beispiel mit graphischer Darstellung von Actions

Graph ermöglicht also die Ableitung einer möglichen Ausführungsreihenfolge, die allen Daten- und Kontrollflüssen gerecht wird. Dazu wird der azyklische Graph unter Berücksichtigung der hierarchischen Struktur durch zusammengesetzte Actions topologisch sortiert.

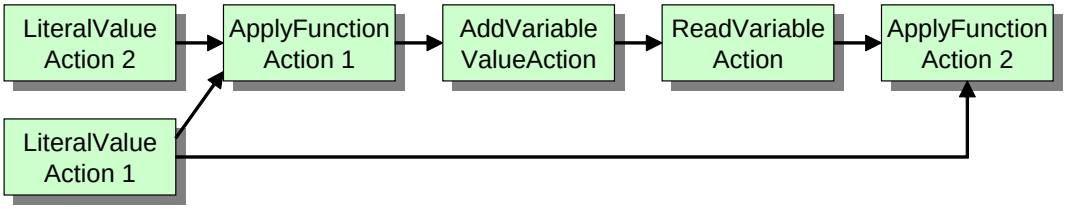

Abbildung 3.9: Abhängigkeitsgraph der Actions

Dazu werden zunächst alle Knoten extrahiert, welche keine eingehenden Kanten besitzen. Für diese gibt es keine Vorgänger im Kontroll- oder Datenfluss, auf deren Ergebnisse sie angewiesen wären. Sie werden an den Anfang der Liste der sortierten Actions gestellt und mitsamt ihrer ausgehenden Kanten aus dem Abhängigkeitsgraphen entfernt. Auf dem Rest-Abhängigkeitsgraphen wird diese Suche nun so lange wiederholt, bis dieser keine Actions mehr enthält.

Zusammengesetzte Actions (beispielsweise *CompositeActions*) werden zur Sequentialisierung als gewöhnliche Action behandelt. Kommt eine dieser Actions

zur Ausführung, wird der Algorithmus zur Sequentialisierung rekursiv auf den Inhalt angewendet.

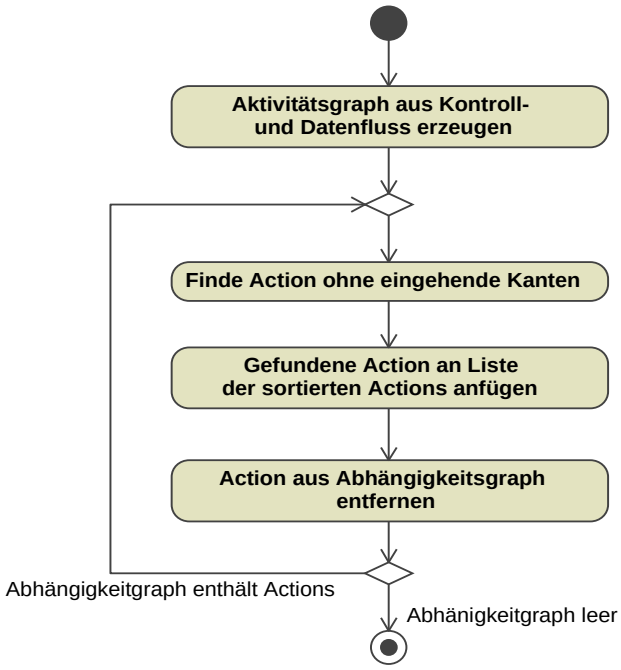

Abbildung 3.10: Algorithmus zur Sequentialisierung von Actions als Aktivitätsdiagramm

Abbildung 3.10 verdeutlicht den iterativen Algorithmus zur Sequentialisierung anhand eines UML Aktivitätsdiagramms.

3.2.3.2 Pins und Symboltabelle

Actions transferieren Werte an die Eingabe-Pins anderer Actions über Ihre Ausgabe-Pins. Die Verknüpfung zwischen Ein- und Ausgabe-Pins erfolgt über Datenflüsse. Eingabe-Pins können dabei nicht nur einfache Werte, sondern auch Sammlungen, also Eingaben mit größerer Multiplizität engegennehmen. Liest beispielsweise eine *ReadAttributeAction* Werte eines Attributs mit einer Multiplizität größer als eins aus, muss auch das Ausgabe-Pin, sowie alle Eingabe-Pins mehrere Werte aufnehmen können.

Der Codegenerator erzeugt je nach Komplexität aus jeder Action eine oder mehrere Zeilen Quelltext. Um die Berechnungsergebnisse einer Action für die folgenden Actions nutzbar zu machen, werden die Ausgaben, also die Werte an den Ausgabe-Pins, in Variablen der Zielsprache abgelegt. Einfache Actions

(beispielsweise Vergleich oder Multiplikation) könnten zwar prinzipiell zusammengefasst werden, um so den Quelltext lesbarer zu gestalten. Moderne Compiler entfernen jedoch selbständig unnötige Variablen aus dem Quelltext, so dass keine Vorteile in Bezug auf Kompaktheit und Ausführungsgeschwindigkeit gegeben wären. Weiterhin würde die Generizität des Ansatzes eingeschränkt, da diese Optimierungen von der Zielsprache abhängig sind und ein Debugging (siehe Abschnitt 7.2) erschweren.

Daher erhält jeder der *available* Ausgabe-Pins (siehe Abschnitt 3.2.2.1) einer Action eine Variable zugeteilt. Vor Ausführung der Action, werden zu den Ausgabe-Pins entsprechende Variablen im Quelltext deklariert, deren Name sich aus einem Prefix und einer fortlaufenden Nummer zusammensetzt und über eine Symboltabelle mit dem jeweiligen Pin in Verbindung gebracht.

Bei Erzeugung einer Action, welche über einen Datenfluss das entsprechende Output-Pin nutzt, wird über den Variablennamen auf die entsprechende Datenquelle zugegriffen.

Da ein Pin bei mehreren Actions in der Gesamtmenge der verfügbaren Ausgabe-Pins enthalten sein kann, muss vor der Deklarierung überprüft werden, ob das Pin nicht bereits in die Symboltabelle aufgenommen wurde, um eine doppelte Deklaration zu vermeiden.

Um Werte höherer Multiplizität aufnehmen zu können, muss zwischen der Nutzung einfacher Variablen (ober Schranke der Multiplizität kleiner oder gleich eins) in der Zielsprache und der Nutzung von Collections (Multiplizität größer als eins) unterschieden werden. Sind die Werte ungeordnet, wird das Pin auf eine Datenstruktur, die das Collection-Interface von Java (*java.util.Collection*, [Krüger 2002]) implementiert, abgebildet. Sind die Werte geordnet, wird eine Datenstruktur, die das Java List-Interface (*java.util.List*) implementiert, genutzt.

3.2.3.3 Operationen und Prozeduren

Jede Operation des UML-Modells erzeugt eine Methode im Quelltext der Zielplattform. Actions modellieren immer nur das Verhalten einer einzelnen Methode.

Operationen besitzen jeweils eine Signatur, welche aus Parametern besteht, die folgende Typen aufweisen können:: Parameter zur Eingabe, zur Ausgabe, zur Ein- und Ausgabe sowie zur Rückgabe. Der Codegenerator lässt dabei nur einen Rückgabe-Parameter zu - eine Beschränkung die in allen imperativen Programmiersprachen gilt. Beim Aufruf von Methoden in Java oder C++

werden Referenzen auf Objekte übergeben und nicht Kopien der Objekte, so dass eine Veränderung der Objekte immer möglich ist.

Im Gegensatz zu Operationen besitzen Prozeduren Ein- und Ausgabe-Pins, um Eingabewerte an enthaltene Actions weiterzugeben bzw. um Ausgabewerte von Actions aufzunehmen. Eingabewerte werden über *argument*- und *result*-Pins einer an die enthaltenen Actions weiter gegeben. Bei der Codeerzeugung werden demnach die Paramter einer Operation den *argument*- und *result*-Pins der zugehörigen Prozedur zugeordnet. Diese Abbildung muss passend zu den Parametern der Operation dargestellt sein, um eine eindeutige Zuordnung zu ermöglichen. Diese Zuordnung geschieht stets in der Reihenfolge der Parameter der Operation und entsprechend ihres Typs: Parameter mit Richtung Eingabe und Ein-/Ausgabe erhalten je einen *argument*-Pin. Parameter mit Richtung Ausgabe und Ein-/Ausgabe erhalten je einen *result*-Pin. Der Rückgabewert der Operation erhält den letzen *result*-Pin.

3.2.3.4 Datentypen, Variablen und Attribute

Datentypen werden in der UML über die Metaklasse *DataType* abgebildet. Instanzen von Datentypen sind identitätslos und damit unveränderbar. Auch Operationen können keine Manipulation am Wert selbst durchführen. Die vom Codegenerator zugelassenen einfachen Datentypen sind die Schnittmenge der Typen in C++ und Java, also: int, long, boolean, char, float, double und eine String-Klasse.

Datentypen, welche nur in einer Zielsprache existieren, können bei Bedarf vom Modellierer hinzugefügt werden. Der Code ist dann aber nur noch für diese Sprache generierbar. Ob diese weiteren Datentypen über eigene Klassen im Modell oder als Instanz der Metamodellklasse *ProgrammingLanguageDataType* modelliert werden ist für den Codegenerator jedoch unerheblich, da nur der Name des Datentyps relevant ist.

Action können auch Variablen zum Austausch von Werten nutzen. Der Zugriff erfolgt über *VariableActions*, die das lesen und schreiben erlauben. Variablen können in *GroupActions* deklariert werden und unterliegen damit den gleichen Regeln für den Gültigkeitsbereich wie die Variablen von Ausgabe-Pins. Die Multiplizität entscheidet analog über ihre Ausprägung als einfache Variable oder Datencontainer. Zugriffe erfolgen nicht über die Symboltabelle, sondern den Variablennamen.

Attribute beinhalten Werte, die zu einem Objekt gehören. Sie werden in der instanziierten Klasse definiert. Der Zugriff ist über *AttributeActions* möglich. Die Verfahrensweise zum Zugriff auf Attribute entspricht weitestgehend dem Vorgehen bei Variablen.

3.2.3.5 Assoziationen

Die Codeerzeugung von Aquintos.GS beschränkt sich derzeit auf binäre Assoziationen. Möglichkeiten Assoziationen mit beliebig vielen Klassen zuzulassen, sind in [Jeckle u. a. 2003] beschrieben. Assoziationsklassen und Qualifier werden ebenfalls nicht unterstützt.

Weder Java noch C++ kennen eine direkte Beschreibung von Assoziationen. Eine unidirektionale Assoziation zwischen zwei Klassen wird mittels eines Attributs realisiert. Ist die Assoziation bidirektional, , erhalten beide Klassen ein entsprechendes Attribut, das auf die Instanz der anderen Klasse verweist. Die Attributwerte bidirektionaler Assoziationen müssen stets synchronisiert werden. Für die Typisierung der Datencontainer ist auch hier wieder die ober Schranke der Multiplizität entscheidend.

3.2.3.6 Objekterzeugung

Objekte können durch *CreateObjectActions* und durch *MarshalActions* erzeugt werden. Die *MarshalAction* erzeugt ein Objekt und initialisiert seine Attribute. In den Zielsprachen muss für die Objekterzeugung ein Konstruktor aufgerufen werden, bevor die Attribute mit Werten belegt werden können. Bei der *MarshalAction* wird deshalb zuerst das Objekt mit Hilfe des Standard-Konstruktors erzeugt. Anschließend werden die Werte der Attribute belegt.

Die *CreateObjectAction* erzeugt ein Objekt ohne Aufruf eines Konstruktors. Dies ist jedoch in Java und C++ nicht möglich. Der Codegenerator erzeugt daher zunächst nur die Referenz auf das zu erzeugende Objekt. Beim ersten Zugriff auf das Objekt wird es erzeugt. Der Modellierer kann also mit einer *CallOperationAction* als ersten Zugriff einen Konstruktor aufrufen. Falls der erste Zugriff auf das Objekt nicht durch Aufruf eines Konstruktors geschieht, wird bei der Generierung noch die Erzeugung über den Standardkonstruktor eingeschoben. Diese Vorgehensweise bietet größtmögliche Flexibilität.

3.2.3.7 Prototypische Umsetzung des Codegenerators

Der Codegenerator ist als Java-Anwendung implementiert und in das existierende Codeerzeugungsumfeld eingebunden.

Abbildung 3.11 zeigt die wichtigsten Pakete und Klassen der Softwarearchitektur der Implementierung. Das Paket *Action Semantics* enthält für jede Metaklasse aus dem Action-Paket eine von GeneralAction erbende Generatorklasse, welche den entsprechenden Code der Action erzeugt. Auch die Paketstruktur folgt der der Metaklassen. In der Vaterklasse *GeneralAction* ist die abstrakte

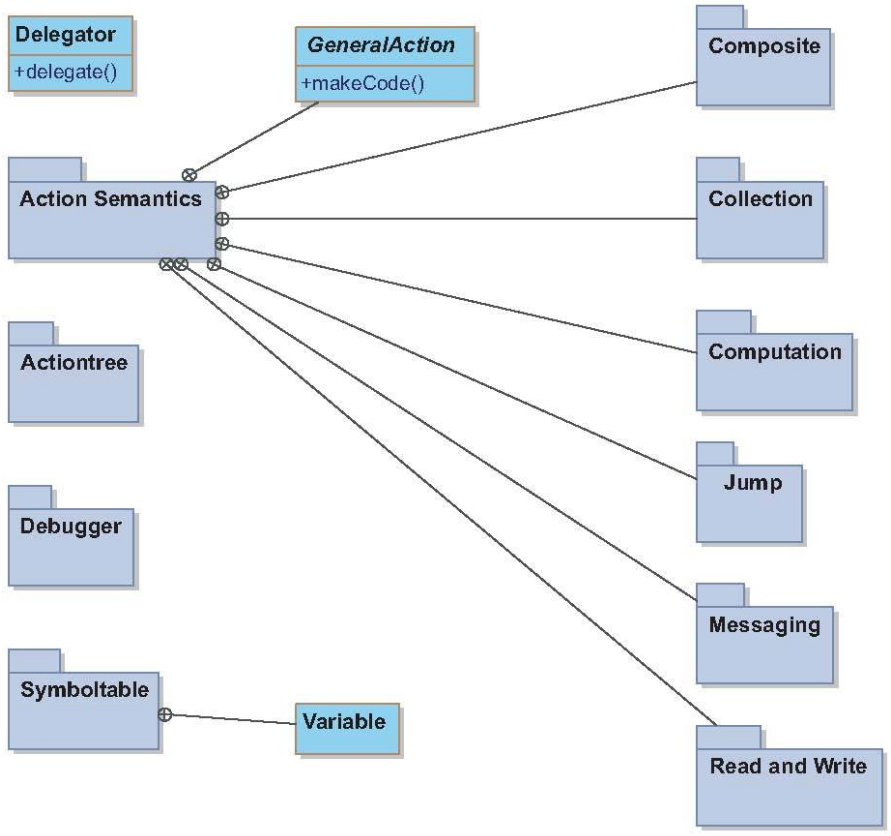

Abbildung 3.11: Pakete und Klassen der Implementierung

Operation *makeCode* deklariert, welche den Code einer Action liefert und für jeden Action-Typ implementiert werden muss. Auch übergeordnete Vorgänge wie das das Jump-Handling (siehe Abschnitt 3.2.2.2) oder die Deklaration von Pins wird von *GeneralAction* behandelt.

Das Paket *Actiontree* enthält Klassen zur Bestimmung der Ausführungsreihenfolge von Actions. Im Paket *Symboltable* ist die Symboltabelle implementiert. Die Klasse *Variable* dient als Schnittstelle zur Symboltabelle und bietet Funktionen zur Deklarationen von Pins und Variablen und zum Zugriff auf Pins und Objekte.

Im Paket *Debugger* ist die in Abschnitt 7.2 näher beschriebene Einbettung des Action Modells in die Plattform *ModelScope* implementiert.

Die Gesamtablaufsteuerung übernimmt die Klasse *Delegator*. Sie ordnet dem Action-Typ die zugehörige Generator-Klasse zu. Ihr wird jeweils das Modellelement oder die Action übergeben, für welche Code generiert werden soll. Der Delegator prüft die Art der Action und instanziiert die entsprechende Genera-

torklasse und führt diese aus. Die Generatorklassen rekurrieren den Delegator wiederum.

3.2.4 Java als Oberflächensprache

Als Oberflächensprache wurde Java in das Modellierungsumfeld integriert. Trotz der fehlenden Sprachmerkmale zur Parallelisierung und Verwaltung von Assoziationen, ist durch seine weite Verbreitung eine hohe Akzeptanz bei potentiellen Benutzern zu erwarten.

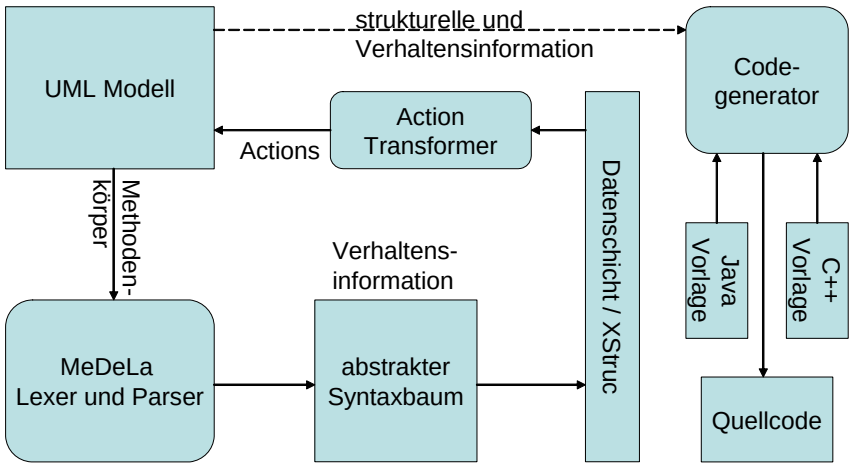

Abbildung 3.12: Funktionaler Zusammenhang bei Codegenerierung mit Actions

Abbildung 3.12 zeigt die Integration in das in Abschnitt 3.1.5 beschriebene Codeerzeugungsumfeld in Aquintos.GS. Die im UML Modell in Java abgelegten Verhaltensbeschreibungen werden in einen abstrakten Syntaxbaum umgewandelt, welcher durch eine interne Datenschicht repräsentiert wird. Mit Hilfe des *ActionTransformers* wird diese Struktur in ein Actionsmodell umgewandelt. Sowohl die Modellstruktur in Form von Klassen, als auch das Verhalten sind nun zu einem Modell verschmolzen und können mittels des im vorigen Abschnitt beschriebenen Codeerzeugungsmechanismus in Quellcode umgesetzt werden.

Der durch Parsen des Java-Klartexts entstandene abstrakte Syntaxbaum enthält Knoten, welche Anweisungen (*Statements*) repräsentieren. Neben einfachen gruppierenden Blockanweisung (Kürzel BLOCK), existieren Bedingungen (IF), Schleifen (FOR, WHILE, DOWHILE), Rücksprunganweisungen (RETURN), als auch Anweisungen, welche aus einem Ausdruck bestehen (EXPRESSION). Abbildung 3.13 listet alle *Statements* mitsamt Beispielen auf.

Anweisungen setzen sich demnach wieder aus Ausdrücken (*Expressions*) zusammen. Dabei werden wie in Abbildung 3.14 gezeigt neun Typen von Ausdrücken unterschieden. Identifier-Ausdrücke (IDENT) tragen dabei die elementarste Bedeutung – sie dienen der Darstellung von Bezeichnern und Literalen und stellen somit die Blätter im abstrakten Syntaxbaum von Java dar. Unäre Ausdrücke (UNARY) werden durch einen Operanden und einen Operator spezifiziert. Eine Teilmenge dieser Ausdrücke bilden so genannte Postfix-Ausdrücke (POSTFIX), bei denen der Operator am Ausdrucksende steht. Binäre Ausdrücke (BINARY) bestehen hingegen aus zwei Operanden – einem linken und rechten – und einem Infix-Operator. Durch Referenzierungsausdrücke (REFERENCE) werden Methoden oder Attribute anderer oder der eigenen Klasse identifiziert. Der prinzipielle Aufbau ist dabei durch <*target.member*> gegeben, wobei *target* ein Ausdruck, der das referenzierte Objekt repräsentiert ist, während durch *member* die Methode oder das Attribut, auf das zugegriffen werden soll, spezifiziert wird. Ein CALL-Methodenaufruf wird auf einer Klasse, welche durch eine Referenz bezeichnet wird, eine Methode aufgerufen, während im Falle von SUPERCALL die Angabe der Zielklasse entfällt, da die Klasse mit der gesuchten Methode eine Superklasse der aufrufenden Klasse ist. In beiden Fällen erfolgt eine Auflistung von Argumenten für die gesuchte Methode. In *Expressions* des Typs CREATE geschieht die Erzeugung von Objekten, wobei deren Typ von der *XClass* spezifiziert wird. Die Variablendeklaration erfolgt durch die *Expression* des Typs VARIABLE_DECL.

Typ des *Statement*	**Beschreibung/Aufbau**	**Beispiel in Java**	**Transformation in Action**
BLOCK	besteht aus einer Liste von Unteranweisungen *(Statement_0, … , Statement_N-1)*	`{a=b+c; … d++;}`	*GroupAction* (erhält Variablen)
EXPR	besteht aus einem einzigen Ausdruck (Expression) und verursacht keinen Datenfluss außerhalb der Anweisung *(Expression_0)*	`a = b*c+d/e; //oder int a;`	*GroupAction* (erhält keine Variablen)
IF	besteht aus einem Testausdruck und zwei Block-Anweisungen – einer Then- und einer Else-Anweisungsliste. *(Expression_test, Statement_then, Statement_else)*	`if (a>0) { a=10-a*2; a=-a; } else { a++; }`	*ConditionalAction*
FOR	besteht aus einem Initialisierungs-ausdruck, einem Testausdruck, einem Inkrementalausdruck und einer Anweisungsliste als Block-Anweisung *(Expression_init, Expression_test, Expression_incr, Statement_body)*	`for(x:=0;x<10; x++) {a=b+c; … d++;}`	*GroupAction* innerhalb einer *LoopAction*
WHILE	besteht aus einem Testausdruck und einer Anweisungsliste als Block-Anweisung (kopfgesteuert) (*Expression_test, Statement_body)*	`While(x < 10) {a=b+c; … x++;}`	*LoopAction*
DOWHILE	besteht aus einem Testausdruck und einer Anweisungsliste als Block-Anweisung (fußgesteuert) (*Expression_test, Statement_body)*	`Do {a= b + c; … x++;} while(x < 10)`	*GroupAction* innerhalb einer *LoopAction*
RETURN	besteht aus einem einzigen Ausdruck (*Expression*), dessen Auswertung zurückgegeben wird *(Expression_return)*	`return z+3;`	*JumpAction*

Abbildung 3.13: Überblick über Typen von Anweisungen und ihre Entsprechung als Action

Typ der Expression	**Beschreibung/Aufbau**	**Beispiel in Java**	**Transformation in Action**

LITERAL		`3`
		`//oder`
		`"hallo"`
			LiteralValue Action

VARIABLE	lokale Variable lesen oder schreiben	`x`	*ReadVariable ValueAction,, AddVariable ValueAction*
ATTRIBUTE	Attribut eines Objekts lesen oder schreiben	`Object o;` `o.x`	*ReadAttribute ValueAction, AddAttribute ValueAction*

PARA-METER	Parameter der umschließenden Operation	`f(int x) {`
		`x`
		`}`
			keine - Parameter als Pin vorhanden

UNARY	einseitige Operationen mit einem Operator und einem Operanden	`-3`
		`//oder`
		`--c`
			ApplyFunction Action

POSTFIX	ein spezifischer unärer Ausdruck, bei dem der Operator am Ende des Ausdrucks steht	`d++`
		`//oder`
		`c--`
			ApplyFunction Action

BINARY	zweiseitige Ausdrücke mit zwei Operanden und Operator	`a < 10`
		`//oder`
		`d=7`
			CallOperation Action

CALL / SUPERCALL	Funktionsaufruf unter Angabe von Klasse (Target), Name und einer Auflistung von Argumenten	`T.meth(arg1,arg2)`	*ReadVariable ValueAction*

VARIABLE-DECL	stellt Variablendeklaration mit Angaben zu Typ, Name und Initialwert der Variablen dar	`int i = 0;`
		`//oder`
		`int i;`
			AddVariable ValueAction

CREATE	Objekterzeugung mittels Angabe des Typs	`new T;`	*CreateObject Action*

Abbildung 3.14: Überblick über Typen von Ausdrücken und ihre Entsprechung als Action

Kapitel 4

Fehlersuche in eingebetteter Software

Das folgende Kapitel beschreibt Konzepte und Methoden bei der Fehlersuche mit Debugging-Methoden. Diese werden an den Anforderungen bei der Softwareentwicklung für eingebettete Systeme gespiegelt. Anschließend wird untersucht, wie Debugging auf ausführbare graphische Modelle angewendet werden kann.

Dazu werden zunächst in Abschnitt 4.1 grundlegende Begrifflichkeiten geklärt, bevor in Abschnitt 4.2 auf den Begriff des *Debugging* eingegangen und dieser von anderen Methoden zur Qualitätssicherung abgegrenzt wird. Abschnitt 4.3 beschreibt konkrete Debugging-Methoden und –Werkzeuge, für welche im darauf folgenden Abschnitt die Randbedingungen beim Debugging in eingebetteten Systemen untersucht werden.

Abschnitt 4.5 untersucht schließlich die Anforderungen und schon im Vorfeld dieser Arbeit existierende Methoden zum Debugging von ausführbaren graphischen Modellen. Der Abschnitt sammelt somit auch die Randbedingungen und den Stand der Forschung für die in Kapitel 5 vorgestellte Debug-Plattform für Modelle *ModelScope*.

4.1 Grundlegende Begriffe

Fehlersuche wird im Folgenden als Überbegriff für einen mehrstufigen Prozess verstanden. Dabei folgen wir im Wesentlichen den Begrifflichkeiten aus [Frank 1994], wobei dieser die Schritte in das Umfeld der Diagnose nach der Auslieferung integriert. Es fallen, mit steigender Komplexität, drei Aufgaben an:

- Die *Fehlerdetektion* soll erkennen, wenn ein Fehler aufgetreten ist. Dies

sollte möglichst bald nach dem Auftreten des Fehlers geschehen. Außerdem sollen möglichst alle Fehlersituationen erkannt und Fehlalarme vermieden werden.

- Der zweite Schritt besteht aus der *Fehlerlokalisierung*, also der Frage, wo genau in welchem Teil des Systems der Fehler auftrat.
- Der letzte Schritt ist die *Fehleranalyse*, bei der die Fehlerart und seine Ursachen bestimmt werden sollen.

Für die Aufgabe der Fehleranalyse bedarf es dabei im Allgemeinen des menschlichen Experten oder zumindest eines wissensbasierten Systems, z.B. eines Expertensystems. Fehlerdetektion und Fehlerlokalisierung können aber durchaus automatisiert und mit geeigneten Methoden bewältigt werden.

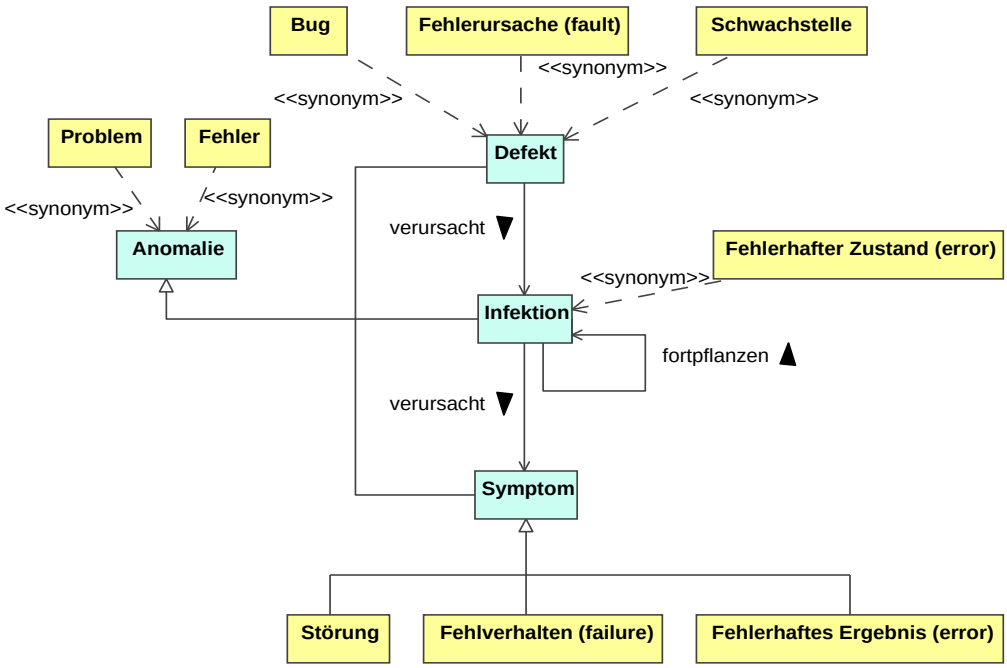

Abbildung 4.1: Klassifikation der Artefakte im Umfeld Fehlersuche

Für Verwirrung sorgt abseits der Prozesse häufig die Untergliederung der auftretenden Artefakte. Diese werden oft nur als „Fehler“ bezeichnet, beschreiben jedoch unterschiedliche Sachverhalte. Umgekehrt bezeichnen einige Begriffe in der Literatur synonyme Sachverhalte, teilweise jedoch mit unterschiedlicher Bedeutung.

Abbildung 4.1 sammelt diese Begriffe und setzt sie in Relation zueinander. Zusätzliche, hier nicht weiter betrachtete Begriffe sind *incident*, *flaw*, *problem*,

gripe und *glitch*. Im Folgenden werden zunächst die zentralen, im Weiteren genutzten Begriffe definiert und eingeordnet:

- **Defekt:** Ein *Defekt* bezeichnet allgemein ein Systemelement, spezieller ein nicht korrektes Codestück, einen Prozess oder eine Datendefinition. Gleichwertig wird im Englischen der Begriff *fault* benutzt, welcher auch als Fehlerursache übersetzt werden kann. Weitere Synonyme sind *Schwachstelle* und *Bug* (siehe [IEEE 1994]). [Zeller 2005] weist darauf hin, dass neben fehlerhafter Implementierung auch unvollständige oder fehlerhafte Nutzeranforderungen Ursache eines Defekts sein können.

- **Infektion:** Eine Infektion bezeichnet eine Abweichung des Systemzustands von dem spezifizierten oder gewünschten Zustand. Ein solcher fehlerhafter Zustand wird in [Zeller 2005] auch als *error* bezeichnet. Ein Defekt kann selbständig oder bei bestimmten Eingaben eine *Infektion* des Systems auslösen. Eine *Infektion* muss nicht nach außen sichtbar werden. Sie kann sich zeitlich weiter fortpflanzen und weitere Teile des Systemzustands erfassen, aber auch korrigiert oder durch weitere *Infektionen* überlagert werden.

- **Symptom:** Ein Symptom ist ein extern beobachtbares Fehlverhalten (engl. *failure*) des Systems. Es bezeichnet das Unvermögen des Systems die gewünschten Anforderungen zu erfüllen [IEEE 1990a]. Synonym werden die Begriffe Störung, Ausfall, Fehlverhalten und fehlerhaftes Ergebnis (*error*) [1] verwendet. Ein *Symptom* wird immer durch eine *Infektion* ausgelöst.

- **Anomalie:** Die *IEEE Standard Classification for Software Anomalies* [IEEE 1994] definiert den Begriff *anomaly* als jede Bedingung welche von Erwartungen basierend auf Spezifikation, Nutzerdokumenten, Standards oder sonstigen Erwartungen abweicht. Anomalien können während Implementierung, Test, Analyse, Übersetzung oder Nutzung von Softwareprodukten auftreten. Der Begriff dient als Überbegriff für die vorherigen Begriffe. *Problem* und *Fehler* werden synonym genutzt.

Die begriffliche Unterscheidung der unterschiedlichen Fehlerarten erzeugt eine Wirkkette, die in Abbildung 4.1 durch Assoziationen gezeigt ist und in der vorigen Aufzählung bereits erläutert wurde. Wichtig ist dabei, dass nicht jeder Defekt eine Infektion und nicht jede Infektion ein Symptom hervorruft. Es

[1] Der englische Begriff *error* wird in der Literatur widersprüchlich verwendet. Während er in [IEEE 1990a] im Sinne eines Symptoms als abgeschwächte Form des Begriffs *failure* definiert wird, bezeichnet er häufig auch eine Infektion.

liegen also mit dem Auftreten notwendige, nicht jedoch hinreichende Voraussetzungen für das Durchlaufen der gesamten Wirkkette vor. Defekte können beispielsweise versteckt bleiben, wenn die defekte Funktionalität des Systems nie genutzt wird.

[Zeller 2005] beschreibt die Wirkkette für Software von Defekten zum Fehlverhalten als einen solchen aus vier Stufen bestehenden Prozess:

1. Der Programmierer erzeugt einen Defekt.
2. Der Defekt bedingt eine Infektion.
3. Die Infektion propagiert durch das System.
4. Die Infektion erzeugt ein Symptom oder das Versagen des Systems.

Parhami erweitert diese Wirkkette und schlägt in [Parhami 1997] ein Zustandsmodell vor. Ein System kann demnach die Zustände *ideal*, *defektbehaftet*, *fehlerhaft*, *symptombehaftet*, *fehlverhaltend*, *degeneriert* und *versagend* durchlaufen. Im Lauf seiner Lebenszeit kann ein System dabei in beiden Richtungen von einem Zustand zum nächsten wechseln.

4.2 Debugging

Etymologisch ist der Ursprung des Begriffs *Debugging* nicht eindeutig geklärt. Eine weit verbreitete Geschichte ist der in Kapitel 1 beschriebene Fund und die Entfernung einer tatsächlichen Motte in den relaisbasierten Schaltkreisen eines der ersten Computer, dem Mark II. Diese dient in der Literatur meist als Gründungsmythos des Debugging. [Metzger 2004] weist darauf hin, dass unabhängig vom Wahrheitsgehalt dieser Geschichte der Begriff des *Bugs* eine tragfähige Analogie zur oft unrechtmäßigen Reputation biologischer Insekten als oft störender, aber auch zerstörende und Krankheit verursachende Wesen herstellt.

Im selben Buch definiert Metzger den Begriff **Debugging** als den Prozess der Suche nach dem Grund, *warum* eine Eingabe eines Programms eine inaktzeptable Ausgabe erzeugt und *was* geändert werden muss, um diesem Zustand abzuhelfen. In anderen Worten beschreibt [Zeller 2003] Debugging als die Verknüpfung eines Symptoms mit einer auf einen Defekt weisenden Serie von Infekten und dem folgenden Entfernen des Defekts.

[Myers 1991] betont, dass der Prozess des Debugging aus zwei Schritten besteht: Ausgehend von dem Indiz eines Symptoms, wird die Natur und der Ort des Defekts bestimmt und daraufhin dieser entfernt.

Als **Debugger** wird ein Werkzeug bezeichnet, welches den Prozess des Debugging unterstützt oder erst ermöglicht. Neben reinen Softwarewerkzeugen spielen in der vorliegenden Arbeit kombinierte Soft- und Hardwarewerkzeuge eine Rolle, wie sie für eingebettete Systeme eingesetzt werden (siehe Abschnitt 4.4).

Ein **Debuggee** ist das mit Hilfe des Debuggers untersuchte System. Beim Debugging von Software ist dieses ein Programm oder ein Prozess, bei eingebetteten Systemen kommen Hardwareaspekte hinzu.

Der Begriff des Debuggee grenzt den untersuchten Systemteil vom Gesamtsystem ab und entscheidet schon teilweise über den Blickwinkel beim Debuggen und die Art der untersuchbaren Defekte. Sucht man beispielsweise beim Absturz eines in C implementierten Programms den verursachenden Defekt, ist der Debuggee zunächst im Allgemeinen nur die Software selbst. Es ist jedoch auch möglich, dass der Compiler fehlerhaft ist, verwendete stabil geglaubte Bibliotheken Ursache des Symptoms sind oder ein Hardwaredefekt vorliegt. Solche Defekte sind dann nicht auffindbar, solange Compiler, Bibliothek oder Hardware nicht Teil des Debuggee werden.

4.2.1 Einordnung und Abgrenzung zu Testen, Monitoring und Diagnose

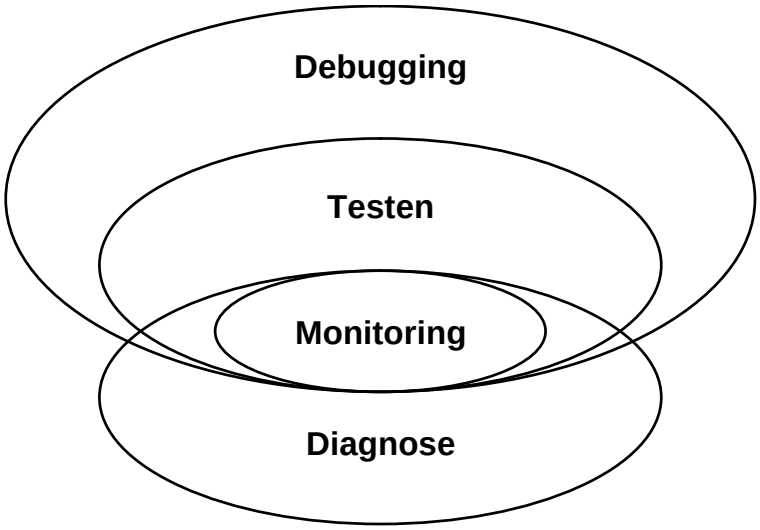

Abbildung 4.2: Kompositionsstruktur der Methoden zur Fehlersuche

Im diesem Abschnitt soll der Begriff und die Tätigkeit des Debugging von weiteren Konzepten im Bereich der Fehlersuche abgegrenzt werden. Abbildung

4.2 zeigt die Zusammenhänge zwischen Debugging, Testen, Monitoring und Diagnose zur Übersicht in einem Diagramm.

In [Myers 1991] wird **Testen** als Prozess definiert, der durch Ausführung eines Systems, im Softwarefall eines Programms, versucht Symptome als Auswirkungen von Defekten zu entdecken. Mit anderen Worten dient Testen dazu, herauszufinden *ob* eine Eingabe des Systems zu unerwünschtem oder falschem Verhalten desselben führt. Er betont dabei, dass Testen nicht dazu dienen kann, die Abwesenheit von Fehlern zu demonstrieren.

Auch heute wird in manchen Publikationen Testen und Debuggen verwechselt. Tatsächlich spielen sich diese Aktivitäten aber in unterschiedlichen Phasen der Fehlersuche ab. Folgt man der Nomenklatur aus Abschnitt 4.1, so fällt Testen in die Phase der Fehlerdetektion und Debugging in die Phasen Fehlerlokalisation und Fehleranalyse.

Unter dem Schlagwort *Testing for Debugging* (siehe [Zeller 2005]) können automatisierte Tests jedoch den Prozess der Fehlerlokalisation und –analyse unterstützen: Das Problem kann durch einen Test reproduzierbar gemacht werden. Auch kann ein Test Hinweise geben, ob der Defekt entfernt wurde oder im Lauf der Entwicklung wieder aufgetreten ist[2]. Ein mit einem Debugging von graphischen Modellen verknüpfbarer UML-basierter Testansatz ist in [Graf u. Müller-Glaser 2007] beschrieben. Automatisierte polymorphe Tests können auch automatisierte Experimente durchführen, wie sie im in Abschnitt 4.3.4 beschriebenen *Delta Debugging* benötigt werden.

Monitoring ist ein Überbegriff für Ansätze zur unmittelbaren systematischen Erfassung, Beobachtung oder Überwachung eines Vorgangs oder Prozesses mittels technischer Hilfsmittel oder anderer Beobachtungssysteme. Als deutschsprachiges Synonym kann auch der Begriff *Überwachung* verwendet werden. Ein Programm oder ein Hardwarebaustein, welcher das Monitoring ermöglicht wird als Monitor bezeichnet.

Monitoring ist mit dem Begriff des Testens insofern verwandt, als dass die Reaktion des Systems mit dem erwünschten oder tolerierbaren Verhalten verglichen wird. Im Gegensatz zu Testansätzen wird das System jedoch nicht angeregt, sondern im Regelbetrieb untersucht. Monitoring kann jedoch auch als Teilkomponente eines Testprozesses eingesetzt werden, wenn das System gesondert angeregt und das überwachte Verhalten des Systems mit einem Testfall abgeglichen wird.

Die **Diagnose** versucht die Wirkkette vom Defekt über Infektion zum Symp-

[2]Dies wird als Regressionstest bezeichnet. Siehe dazu auch [Balzert 2005].

tom zu reversieren. In technischen Systemen kann auf Basis von Monitoring-Informationen auf eine Fehlerursache geschlossen werden.

Bei den signalgestützten Verfahren wird durch Messung von relevanten Signalen des Systems auf das Vorhandensein eines Fehlerzustandes geschlossen. Dies kann durch einfache Grenzwertüberwachung geschehen, in dieser Verfahrensklasse sind aber auch anspruchsvollere Methoden wie Spektralanalyse, Mustererkennung oder statistische Verfahren enthalten. Zusätzlich stehen modellbasierte Verfahren zur Verfügung. Ein Modell des zu überwachenden Systems beschreibt das ungestörte Verhalten des Prozesses. Bei bekannten Eingangsgrößen können dann Ausgänge von Modell und Prozess verglichen werden. Bei einer Abweichung kann durch Parameterschätzverfahren darauf geschlossen werden, ob und wo ein Fehler im Prozess auftritt. Die Diagnose fokussiert auf die Untersuchung von Fehlern zur Laufzeit, für welche keine Entwurfsfehler verantwortlich sind, sondern Betriebsausfälle beispielsweise in einem regelungstechnischen System.

4.2.2 Debugging und Simulation

Im Bereich der modellbasierten Entwicklung existieren zahlreiche Verfahren, welche die Analyse des Systemverhaltens durch Simulation der Systembeschreibung durchführen. Dies ist auf Modellebene sowohl im Bereich rein funktionaler Verhaltensbeschreibungen möglich, als auch durch Simulation des im Prozess der Modell-zu-Text-Transformation entstandenen Quelltextes.

Sowohl Debugging als auch Simulation fokussieren während der Entwicklung auf dieselbe Aufgabe: Suche und Entfernung von Defekten, welche in einem Modell existieren und die zu Symptomen oder dem Versagen des Gesamtsystems führen. Die Suche nach Defekten benötigt Werkzeugunterstützung, welche sowohl simulationsbasiert, als auch auf dem Zielsystem mit Hilfe eines Debuggers durchgeführt werden kann.

Während Debugging die Erzeugung eines ausführbaren Systems oder zumindest Prototypen auf dem Zielsystem benötigt, arbeiten Simulationsansätze interpretierend, indem sie das Modell selbst zur Laufzeit der Simulation auswerten und die Reaktion von Plattform und Umwelt nachbilden. Diese Interpretation kann jedoch auch wieder Codeerzeugung beinhalten, beispielsweise um beschleunigte Simulationskerne zu erzeugen [Dreier u. a. 2003a].

Zur Simulation wird das Zielsystem nicht benötigt. Die Nutzung eines Debuggers impliziert dagegen, dass das Modell auf einem Zielsystem oder einer *Rapid-Prototyping*-Plattform ausgeführt wird, welche dem tatsächlichen Produktionssystem so ähnlich wie möglich ist.

Beim Entwurf eingebetteter Software mit Modellen werden weitere Randbedingungen an einen Simulationsansatz deutlich. Üblicherweise existiert eingebettete Software im Kontext der Peripherie und der Systemumgebung. Bei der Simulation bedeutet dies, dass Aktuatoren und Sensoren modelliert werden müssen. Weiterhin muss ein hinreichend genaues Umgebungsmodell existieren. Die Äquvalenz der simulierten Peripherie zum realen System ist jedoch nicht immer gegeben. Zusätzlich fügt die Anforderung der Modellierung der Peripherie dem System zusätzliche Komplexität und damit mögliche Defekte hinzu.

Der genannte Nachteil eines Simulationsansatzes kann sich in frühen Entwurfsphasen in das Gegenteil verkehren: Existiert keine Zielplattform ist die Simulation häufig die einzige Möglichkeiten, das Verhalten des Modells zu validieren und Defekte im Modell zu identifizieren.

Der Übergang von einem reinen Simulationsansatz zu einem Debugging im realen System kann durch Einsatz von *Software-in-the-Loop*- (SiL) und *Hardware-in-the-Loop*-Systemen (HiL) in kleinere Schritte heruntergebrochen werden [Schäuffele u. Zurawka 2004]. Dabei werden Peripherie und Umwelt des eingebetteten Systems durch eine Simulation ersetzt. Eingebettete Software wird bereits auf dem später verwendeten Prozessor oder sogar Steuergerät ausgeführt.

Eine weitere Quelle unklarer semantischer Äquvalenz zwischen Simulator und echtem System wird bei einem modellgetriebenen Ansatz hinzugefügt. Hierbei wird dem System bei der Tranformation vom Modell zum Code semantische Information hinzugefügt. Im Fall der Simulation ist es möglich, dass die Ausführungssemantik des Simulators nicht der implizit bei der Codeerzeugung hinzugefügten dynamischen Bedeutung entspricht. Beim Entwurf hierarchischer Zustandsautomaten kann die Ausführungssemantik beispielsweise zwischen Werkzeugen, Simulatoren und unterschiedlichen Codegeneratoren variieren.

Diese Problematik verschärft sich weiter beim Einsatz domänenspezifischer Modellierungssprachen (siehe Abschnitt 2.3.2). Hier ist die Ausführungssemantik einer selbst spezifizierten Modellierungssprache gewollt deutlich variabler bzw. durch den Entwickler selbst spezifizierbar und dadurch auch fehleranfällig. Fehlersuche, welche das Ergebnis der Codegenerierung beachtet ist in diesem Umfeld wertvoll.

4.3 Debugging-Methoden

Der folgende Abschnitt beschreibt wissenschaftliche und industriell eingesetzte Methoden und Werkzeuge zur Fehlerlokalisation und –analyse. Nachdem in Abschnitt 4.3.1 das grundlegende Vorgehen und ein Schema für Analysestrategien vorgestellt wird, konzentrieren sich die folgenden Abschnitte auf spezielle Ansätze und die zugehörigen unterstützenden Software-Werkzeuge.

4.3.1 Grundlegende Strategien

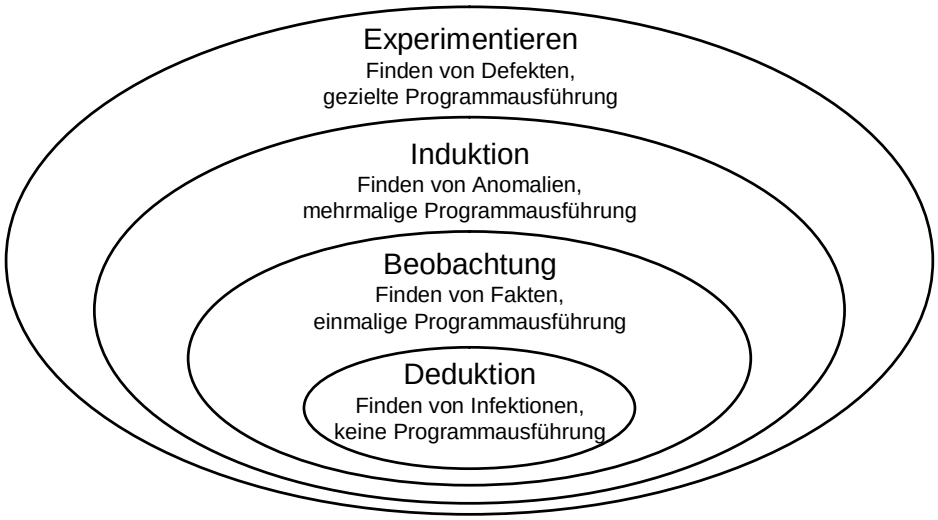

Abbildung 4.3: Hierarchie der Analysestrategien, die zum Debugging genutzt werden [Zeller 2003]

Zeller zeigt in [Zeller 2003], wie unterschiedliche logische Analysestrategien mit wachsenden Anforderungen zu einer umfassenden Debuggingsstrategie führen, welche in Analogie zu wissenschaftlicher Hypothesenbildung und -falsifizierung den Begriff des Experimentierens in ihren Mittelpunkt stellt (siehe Abbildung 4.3).

Deduktion bezeichnet das logische Folgern des Speziellen aus dem Allgemeinen. Das Allgemeine in Form des Modells oder Programmcodes wird unter Annahme der Kenntnis der Semantik der genutzten Sprache auf spezielle Abfolgen von Eingaben angewendet. Das System muss dazu nicht ausgeführt werden, da die Reaktion des Systems durch das vorhandene Wissen determiniert ist. Popper weist in [Popper 1994] darauf hin, dass positiv deduktives Vorgehen nur von axiomatischen Vorraussetzungen ausgehen kann, welche dann aber dogmatisch nicht falsifizierbar sind und damit nicht mit realen Experimenten

abgleichbar sind. Angewendet auf Fehlersuche bedeutet dies, dass zwar durch Nachverfolgung eines hypothetischen Ablaufs Defekte gefunden werden können, nicht jedoch ein Symptom erklärt werden kann, welches im realen System auftritt.

Hierzu werden dynamische Methoden benötigt, welche auf Informationen des realen Systems zurückgreifen können. **Beobachtungen** sammeln Fakten eines tatsächlichen Programm- oder Systemablaufs. Praktisch alle kommerziellen Debugging-Werkzeuge dienen direkt dazu, dem Entwickler die Beobachtung eines Systemablaufs zu erleichtern, wenn nicht erst zu ermöglichen. Eine Beobachtung allein hilft jedoch noch nicht bei der Rückführung eines Symptoms auf den auslösenden Defekt.

Dazu werden mehrere Durchläufe des Systems unter verschiedenen Randbedingungen benötigt, welche durch **Induktion** verallgemeinert werden. Induktion bezeichnet das logische Schliessen vom speziellen Beispiel auf das Allgemeine. Ein Entwickler wird während des Debugging mehrere Durchläufe mit unterschiedlichen Eingaben „durchspielen“ um in der Phase der Fehlerlokalisation die tatsächliche Ursache einzugrenzen. In [Popper 1994] wird nachgewiesen, dass der induktive Schluss aus der Erfahrung auf verifizierte Aussagen unzulässig ist. Übertragen auf die Fehlersuche besteht die Schwierigkeit schon darin, eine repräsentative Auswahl der Eingaben so zu treffen, dass das Problem in Hinblick auf den auslösenden Defekt tatsächlich eingegrenzt werden kann.

Induktion kann jedoch zur Hypothesenbildung eingesetzt werden. Hypothesen können zum gezielten Durchführen von **Experimenten** verwendet werden. Ein Experiment geht dabei von einer Hypothese aus, die durch dieses potentiell widerlegbar sein muss. Bei der Suche von Fehlern in einem Programm wird ein Entwickler durch Deduktion, Beobachtung und Induktion eine Hypothese über einen Defekt aufstellen. Durch gezielte Veränderung und erneute Ausführung des Systems kann diese Hypothese gestützt oder widerlegt werden. Aus einer gestützten Hypothese können weitere Hypothesen gewonnen werden, die an einen vermuteten Defekt heranführen. Wird der vermutete Defekt entfernt, ist mindestens eine weitere Ausführung des Systems als Experiment nötig, welche keine Symptome aufweist. Eine Garantie über die Richtigkeit der Hypothese, also eine Garantie für die nun hergestellte Korrektheit des Modells oder Programms ist jedoch nicht erreichbar.

Die aus Experimenten, also Programmabläufen gewonnenen Erkenntnisse sind nur wertvoll, wenn sie wiederholbar sind. Dies bedeutet für das Debuggen, dass Abläufe des Systems so deterministisch wie möglich sein müssen. Diese Problematik führt bei Echtzeitbedingungen, wie sie für eingebettete Systeme charakteristisch sind, zu tracegestützten Post-Mortem-Analysemethoden, die in den Abschnitten 4.3.3 und 4.4 untersucht werden.

Alle in den folgenden Abschnitten vorgestellten industriell gebräuchlichen und wissenschaftlich untersuchten Methoden dienen dazu, die beschriebenen Bausteine der auf Experimenten aufbauenden Analysestrategie zu unterstützen. Dies kann dadurch geschehen, indem Informationen für den Nutzer eines Debuggers aufbereitet werden, es kann jedoch auch die Induktion und die Durchführung von Experimenten unterstützt und automatisiert werden. In den einzelnen Abschnitten wird daher eine Einordnung in das Schema in Abbildung 4.3 gegeben.

4.3.2 Klassische Debugging-Werkzeuge

Klassische Debugging-Werkzeuge dienen der Beobachtung während dem Durchführen von Experimenten.

Als erster klassischer Debugger entstand 1961 das *DEC Debugging Tape* zur Suche von Fehlern im PDP-1 Computer der Firma *Digital Equipment Corporation* (DEC), welcher auch in neuerer Zeit als *DDT* weiterbestand [DEC Corp. 1986]. Seitdem etablierte sich das Konzept, laufende Programme *on-line* zu steuern und ihren Zustand zu untersuchen. Schon dieser Debugger, welcher ähnlich wie der im folgenden Abschnitt vorgestellte GDB eine kommandozeilengesteuerte Schnittstelle besitzt, weist die wesentlichen grundlegenden Möglichkeiten eine klassischen Debugging-Werkzeugs auf.

Ein klassischer Debugger unterstützt den Entwickler bei zwei Aufgaben: Ablaufkontrolle und Inspektion. Ablaufkontrolle bezeichnet die Steuerung des untersuchten Systems so, dass der Wechsel zwischen den Zuständen *Ausführung* und *Angehalten* beeinflusst werden kann. Im Fall eines nebenläufigen Programms besitzt jeder Prozess des Programms diese beiden Zustände, so dass moderne Debugger die Ablaufkontrolle einzelner Tasks separat steuern können. Grundlegende Aufgaben der Ablaufkontrolle sind:

- **Starten** und **Stoppen** des Programmablaufs durch direkten Eingriff des Debugger-Benutzers.
- Durchführen einzelner Schritte (***Single Step***) im Programm. Der Single Step-Modus erlaubt dem Debugger den Prozessor so zu kontrollieren, dass entweder nach der Ausführung eines Maschinenbefehl oder eines Quellcodebefehls das Programm wieder gestoppt wird. Ein Einzelschritt kann direkt nach dem nächsten ausgeführten Schritt anhalten (*Step In*) oder bei Aufruf einer Unterroutine diese komplett ausführen und erst bei Rückkehr stoppen (*Step Over*).
- Eine Haltepunkt (***Breakpoint***) ist eine Anweisung an den Debugger den

Programmablauf bei Erreichen einer bestimmten Position im Quelltext zu stoppen. Der Debugger stoppt das Programm bevor der markierte Befehl ausgeführt wird.

- Ein Überwachungspunkt (***Watchpoint***) bedeutet dem Debugger anzuhalten, wenn auf eine Variable geschrieben wurde oder diese einen bestimmten Wert annimmt. Überwachungspunkte können in einigen Debuggern mit Haltepunkten kombiniert werden um komplexe Bedingungen zu formulieren, wann der Programmablauf gestoppt werden soll.

Wurde das Programm angehalten, können verschiedene Sichten zur Inspektion des Programmzustands genutzt werden:

- In einer **Quelltextansicht** kann abgefragt werden welchen Wert der Programmzähler hat, also an welcher Stelle der Programmablauf gestoppt wurde.
- Der Zustand der verwendeten **Register** und **Variablen** kann in einem *Watch*-Fenster untersucht werden. Auch sind symbolische Debugger in Lage, den Inhalt der Attribute von Objekten strukturiert darzustellen und über Referenzen und Zeiger durch den Speicher zu navigieren. Viele Debugger ermöglichen es, beliebige Ausdrücke abzufragen, die syntaktisch an die verwendete Programmiersprache angelehnt sind und durch den Debugger ausgewertet werden.
- Der Inhalt des **Speichers** kann direkt in hexadezimaler Form ausgegeben werden.
- Beim Aufruf einer Unterroutine wird die Rücksprungaddresse mitsamt lokaler Variablen auf den Stapelspeicher des Programms geschrieben. Eine **Aufrufliste** (*Call Stack*) kann diesen auswerten und stellt die hierarchische Abfolge der Funktionsaufrufe (sogenannte *Stack Frames*) zum aktuellen Zeitpunkt dar. Viele Debugger unterstützen dabei die Auswertung von lokalen Variablen in unterliegenden Stack Frames.
- Bei einer Plattform auf der Multitasking möglich ist, können Debugger Einblick in den Zustand der laufenden **Threads/Prozesse** geben. Häufig ist diese Sicht in die Aufrufliste integriert, wobei jeder Thread eine eigene Unterliste besitzt. Ansichten für Interprozesskommunikation (Signale, Nachrichten, Semaphoren) sind möglich aber selten.

Die beschriebenen Fähigkeiten setzen in der Regel die Entwicklung mit einer imperativen Programmiersprache vorraus; für im Umfeld eingebetteter Systeme selten angewendete deklarative Sprachfamilien müssen andere Paradigmen zur Ablaufkontrolle und Inspektion gewählt werden.

Mit komplexen Abfragen auf dem Zustand des Systems beschäftigt sich der Teilbereich des *Query-Based Debugging* [Lencevicius 2000; Hobatr u. Malloy 2001]. Bei diesen Ansätzen wird der Systemzustand als Datenbank gesehen, aus welcher mit Suchausdrücken Teilmengen herausgearbeitet und mit Metriken ausgewertet werden können. Beispielsweise können alle Objekte eines bestimmten Typs erfragt werden, welche einen Zeiger auf ein anderes Objekt und gleichzeitig einen bestimmten Attributwert aufweisen.

Das typische Vorgehen bei der Nutzung eines klassischen Debug-Werkzeugs zur Hypothesenverifizierung (siehe Abschnitt 4.3.1) stellt sich wie folgt dar:

1. Aufstellen einer These über die mögliche Position des Defekts
2. Setzen eines Haltepunkts vor der vermuteten Position
3. Annäherung mit Hilfe von Breakpoints / Einzelschritt, dabei Überprüfung des Programmzustands.
4. These falsch, weiter mit 1.
5. Ansonsten: Korrigieren des Defekts

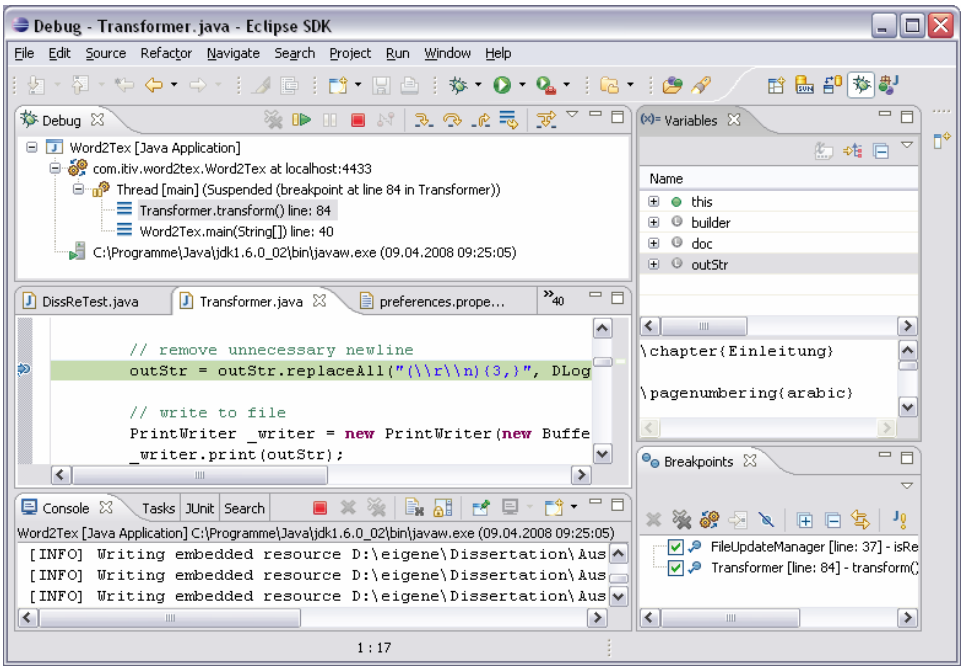

Abbildung 4.4: Graphische Benutzeroberfläche des Java-Debuggers des Eclipse SDK

Die meisten modernen Entwicklungsumgebungen beinhalten eine Umgebung zum klassischen Debuggen, welche in die Benutzeroberfläche eingebunden ist. Abbildung 4.4 zeigt beispielhaft die graphische Benutzeroberfläche der Java-Entwicklungsumgebung von Eclipse [Daum 2004]. Die Abbildung zeigt von links oben nach rechts unten: Baum der Threads und deren *Call Stacks*, Anzeige der Variablen, Quelltextansicht mit an *Breakpoint* angehaltenem Programmablauf, Konsole für Programmausgaben und eine Liste der gesetzten Haltepunkte.

Für den in dieser Arbeit entwickelten Debugger für Modelle wurden als Basistechnologie die Debugger GDB für C++-Umgebungen und JDB für virtuelle Maschinen in Java eingesetzt, welche in den folgenden Unterabschnitten genauer beschrieben werden. Ein weiterer klassischer Debugger für eingebettete Zielsysteme wird in Abschnitt 6.1.2.1 beschrieben.

4.3.2.1 GNU Debugger (GDB)

Der *GNU Debugger* (GDB)[3] ist ein umfangreiches Debugging-Paket, welches Teil der *GNU Compiler Collection*[4] ist. Die *GNU Compiler Collection* ist der de-facto Standard zur Softwareentwicklung im Linux-Umfeld und zur Softwareentwicklung für eingebettete Systeme insbesondere zur Übersetzung von C und C++ Programmen. Es existieren Portierungen des Compilers für nahezu alle wichtigen eingebetteten 16-bit und 32-bit Prozessorfamilien. Diese Portierungen beinhalten häufig auch eine für eingebettete Prozessoren nutzbare Variante des GDB.

Der GNU Debugger wurde erstmals 1986 geschrieben und ist unter der *GNU General Public License* (GPL) verfügbar.

Der GNU Debugger bietet dem Benutzer eine Textschnittstelle ähnlich einer Befehlskonsole an. Ein Ausschnitt aus einer Debugging-Sitzung ist in Abbildung 4.5 dargestellt. [Stallman u. a. 2006] beschreibt umfassend die Befehle zur Bedienung des Debuggers. Diese umfassen alle beschriebenen Möglichkeiten des klassischen Debugging und erweitern diese noch. Die in dieser Arbeit verwendeten Befehle sind aus Abbildung 6.1 in Abschnitt 6.1.1 abzulesen.

In [Gilmore u. Shebs 2006] wird auf die Interna des Debuggingpakets eingegangen. Abbildung 4.6 zeigt mögliche Ansatzpunkte zur Einbindung des GNU Debugger in eine graphische Benutzeroberfläche oder den in Kapitel 5 beschriebenen Debugger für Modelle:

[3] GNU Debugger Webseite: http://sourceware.org/gdb

[4] GNU Compiler Collection Webseite: http://gcc.gnu.org

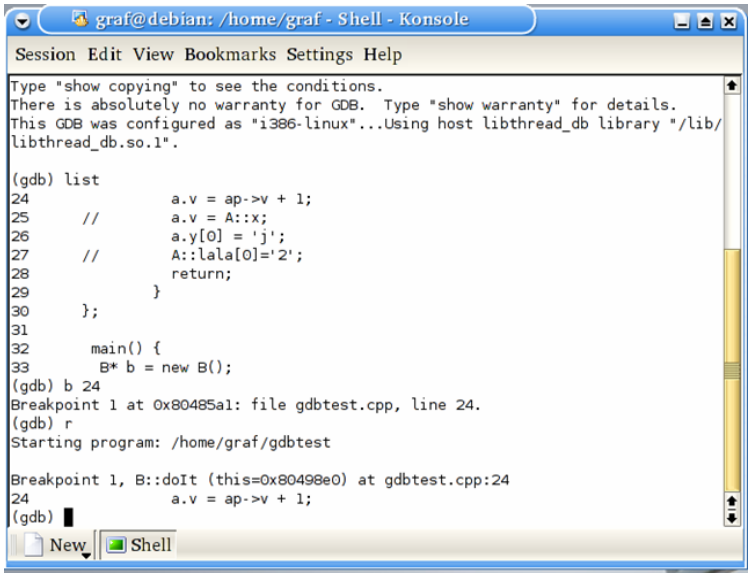

Abbildung 4.5: Textbasierte Benutzerschnittstelle des GNU Debugger

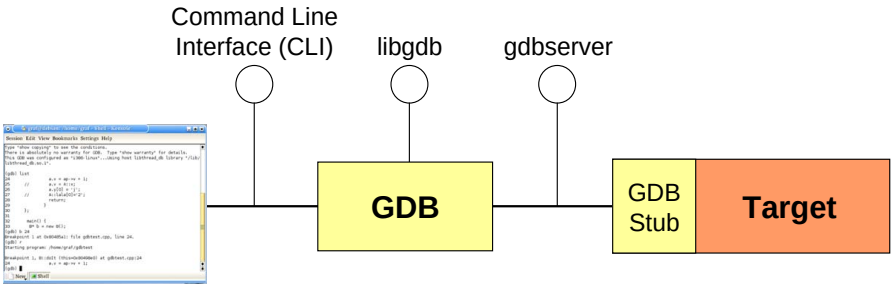

Abbildung 4.6: Schnittstellen zum programmatischen Zugriff auf GDB

- Auf unterster Ebene befindet sich die Schnittstelle *gdbserver*, welche dazu dient, ein Programm, welches auf einem anderen Prozessor als der GDB läuft, zu untersuchen. Dies ist immer bei der Entwicklung eingebetteter Software der Fall. Das Zielsystem muss dabei einen sogenannt *Stub* implementieren, über den der GDB auf niedriger Abstraktionsebene mit dem Debuggee kommuniziert.

- Um auf symbolischer Ebene den Debugger zu steuern und auszuwerten, kann die Texteingabe ferngesteuert werden, indem Befehle über die Standardeingabe der Kommandozeilenschnittstelle (CLI) an den GDB gesendet werden. Die Ausgaben können abgefangen und ausgewertet werden. Fast alle Werkzeuge, welche auf dem GDB-Debugger basieren, nutzen derzeit diese Form der Kommunikation mit dem GDB.

- Da die textuellen Ausgaben des GDB nicht in erster Linie maschinenlesbar sind und sich im Laufe der Veröffentlichung neuer Versionen ändern, ist die CLI-Schnittstelle nicht sehr stabil. Daher wurde mit *libgdb* eine Schnittstelle integriert, welche Funktionen zum Zugriff auf Befehle, Ereignisse und Abfragen des GDB bietet. Diese ist jedoch noch in der Entwicklung und in vielen Portierungen noch nicht vorhanden.

Zusammenfassend ist der Zugriff über das CLI derzeit am stabilsten und wurde daher im weiteren Verlauf der vorliegenden Arbeit weiter verfolgt.

4.3.2.2 Debugging in virtuellen Maschinen in Java

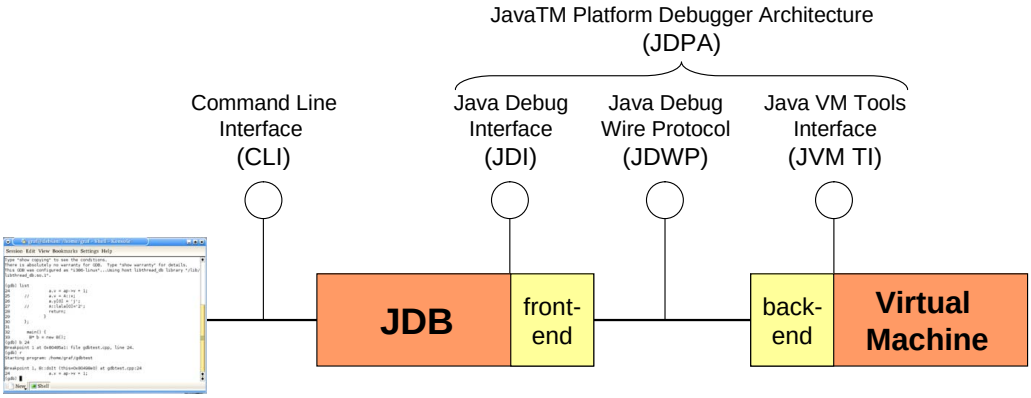

Abbildung 4.7: Schnittstellen zum programmatischen Zugriff auf virtuelle Maschinen in Java

Java Programme werden in virtuellen Maschinen ausgeführt. Zum Zugriff auf diese zur Laufzeit wurde von Sun die *Java Platform Debugger Architecture* (JPDA) definiert (siehe Abbildung 4.7). Die JDPA besteht aus drei Schnittstellen welche von Debuggern in Java-Entwicklungsumgebungen zum Zugriff auf die virtuelle Maschine genutzt werden können. Das *Java Virtual Machine Tools Interface* (JVM TI) definiert die grundlegenden Dienste, welche eine VM zum Debugging anbieten muss. Das *Java Debug Wire Protocol* (JDWP) definiert das Format der Daten und Anforderungen welche zwischen dem Debuggee und dem Debugger *Front-End* ausgetauscht werden. Das *Front-End* implementiert das *Java Debug Interface* (JDI) welche Informationen und Anforderungen auf Nutzerebene definiert [Sun Microsystems 2008].

Vergleicht man diese Architektur mit der des GDB im vorigen Abschnitt, entspricht das JVM TI dem *gdbserver* und das JDI der *libgdb*. Auch der GNU Debugger selbst besitzt eine Entsprechung in Java: den *Java Debugger* JDB

[Sun Microsystems 2002]. Dieser ist sehr ähnlich zum GDB aufgebaut und unterscheidet sich nur in wenigen Befehlen und java-spezifischen Eigenschaften. Wie der GDB besitzt auch er eine auf einer Textkonsole basierende Benutzerschnittstelle.

Um Treiber zum Debugging für Modelle möglichst einfach warten zu können, wurde in dieser Arbeit auch für die Einbindung der JDPA in die Plattform zum Debugging von Modellen das CLI des JDB genutzt. Die in dieser Arbeit verwendeten Befehle sind auch für den JDB aus Abbildung 6.1 in Abschnitt 6.1.1 ersichtlich.

4.3.3 Ablaufverfolgung (Tracing)

Im Bereich des Debugging bezeichnet **Tracing** eine spezielle Form der Aufzeichnung (*Logging*) von Information über den Ablauf eines Programms. Diese Information kann später, nach Beendigung des Programms vom Programmierer zum Debugging benutzt werden. Daher werden Debugging-Methoden, welche auf *Tracing* basieren, häufig auch als **Post-Mortem-Debugger** bezeichnet.

Eine klare Abgrenzung zwischen *Tracing* und anderen Formen von Aufzeichnungen ist nicht einfach. Eine Definition bezeichnet *Tracing* als eine Aufzeichnung des Kontrollflusses und beinhaltet damit nicht datenzentrische funktionale Anforderungen, wie Fehlerausgaben eines Programms. Bestimmte Bereiche, wie beispielsweise Zugriffsprotokolle auf Servern, bewegen sich in einem Graubereich. Einen Überblick über Methoden zur Ablaufwiederholung zum Debugging gibt [Ronsse u. a. 2000].

Der Begriff *Tracing* im engeren Sinne, wie er im Folgenden in dieser Arbeit verwendet wird, bezeichnet die Aufzeichnung folgender Artefakte:

- Programmzähler: Dies kann durch Aufzeichnung jedes ausgeführten Befehls oder durch Aufzeichnung nur von Methodenaufrufen oder Sprungbefehlen (z.B. auch wiederholtes Protokollieren von Schleifenkörpern) erfolgen.

- Zugriffe auf Daten: Hier werden primär Schreibzugriffe auf Speicher, speziell auf Variablen aufgezeichnet. Auch Lesezugriffe können protokolliert werden.

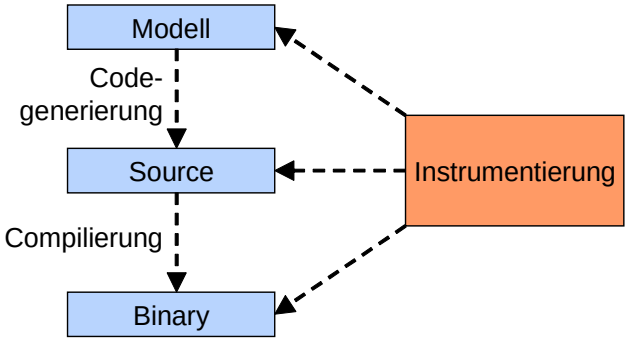

Abbildung 4.8: Mögliche Ansatzpunkte zur Instrumentierung

4.3.3.1 Datenaquisition

Es stellt sich nun zunächst die Frage, wie Trace-Daten während der Ausführung eines Programms gewonnen werden können. Ein Ansatz besteht in der Instrumentierung des Debuggee. Dabei wird das untersuchte Programm oder Modell so verändert, dass dieses selbst die gewünschten Informationen während seines Ablaufs generiert. Dies ist vergleichbar mit der manuellen Ausgabe von Logging-Informationen in Programmen.

Wie Abbildung 4.8 zeigt, sind bei modellgetriebener Entwicklung prinzipiell mehrere Ansatzpunkte möglich: Das Modell kann in jeder Stufe die es hin zum ausführbaren *Executable* durchläuft instrumentiert werden. Dabei ist diese Instrumentierung statisch, erfolgt also zum Zeitpunkt der Modell-zu-Executable-Transformation.

Im Gegensatz dazu kann der Debuggee auch dynamisch zur Laufzeit verändert werden, beispielsweise wenn der Entwickler während einer Sitzung den Blickpunkt auf einen anderen Aspekt legt.

Der Vorteil der Instrumentierung liegt darin, dass die Aufzeichnung in beliebigem Umfang und in beliebiger Granularität möglich ist. Nachteile entstehen jedoch dadurch, dass durch die Instrumentierung die Ausführungseinheit die Aufzeichnung leisten muss. Dadurch verringert sich die Ablaufgeschwindigkeit und der Umfang des Speicherverbrauchs steigt. In echtzeitfähigen Systemen verändert sich das Programm so, dass Echtzeitbedingungen verletzt werden oder das System sein Verhalten ändert[5].

Für eingebettete Prozessoren können spezielle in Hardware realisierte Trace-Mechanismen oder sogenannte externe *Emulatoren* verwendet werden (siehe

[5]Dieser Effekt wird in Anlehnung an die Unschärferelation aus der Quantentheorie als *Heisenberg-Effekt* bezeichnet und ist ein Nachteil von Debuggingmethoden, die das System zur Inspektion anhalten oder verändern.

Abschnitt 4.4.1), welche einen *Trace* erzeugen, ohne das System zu verändern. Dabei sind zur Aufzeichnung zwei grundlegende Ansätze möglich: *On-Chip Debugging* oder *Emulation* (siehe Abschnitt 4.4). Beide Ansätze sind echtzeitfähig, verändern nicht den Debuggee und umgehen damit den Heisenberg-Effekt. Nachteil dieses Vorgehens ist die begrenzte Bandbreite und die dadurch geringere Flexibilität der Instrumentierung. Ein solcher Mechanismus, der in den Modell-Debugger *ModelScope* eingebunden wurde, wird in Abschnitt 6.1.2.4 ausführlich beschrieben.

4.3.3.2 Nutzung von Trace-Daten

Trace-Daten können während des Debugging direkt zur Gewinnung zusätzlicher Informationen zum Zweck der Hypothesenbildung eingesetzt werden. Dabei können durch Rückverfolgung des Ablaufs vom Auftreten eines Symptoms aus beispielsweise folgende Fragen beantwortet werden:

- Wurde ein bestimmter Bereich des Codes bereits ausgeführt und falls ja, wann?
- Wann hat ein Datum, also eine Speicherstelle oder eine Variable den aktuellen Wert angenommen?
- Wann wurde eine Ausnahme ausgelöst?

Die Untersuchung kann dabei direkt auf der Aufzeichnung geschehen oder durch ein Debug-Werkzeug aufbereitet werden, beispielsweise in eine graphische Abbildung oder durch Rückführung von der Maschinencode- auf die Quelltextebene. Ein Beispiel für ein Werkzeug, welches Trace-Daten für Java-Programme aufzeichnen und aufbereiten kann ist JDLab der Universität Oldenbourg [Alekseev 2007].

Neben der direkten Nutzung des Trace kann versucht werden, einen Teil des Systemzustands zu einem vergangenen Zeitpunkt zu rekonstruieren. Es wird also im Gegensatz zum klassischen Debugging möglich, sich in einem virtuellen Debuggee entlang der Zeitachse auch rückwärts, also in zwei Richtungen zu bewegen [Booth u. Jones 1997].

Ein System, welches diesen Ansatz für Java-Programme realisiert, ist der in Abbildung 4.9 gezeigte *Omniscient Debugger* [Lewis 2003]. Der Omniscient Debugger zeichnet einen Trace in Form von Zeitstempeln zu "interessanten Zeitpunkten" wie Schreibzugriffen, Methodenaufrufen oder ausgelösten Ausnahmen auf. Dies geschieht durch Instrumentierung der virtuellen Maschine in Java. Der Umfang der Aufzeichnung kann beeinflusst werden. Es ist nun

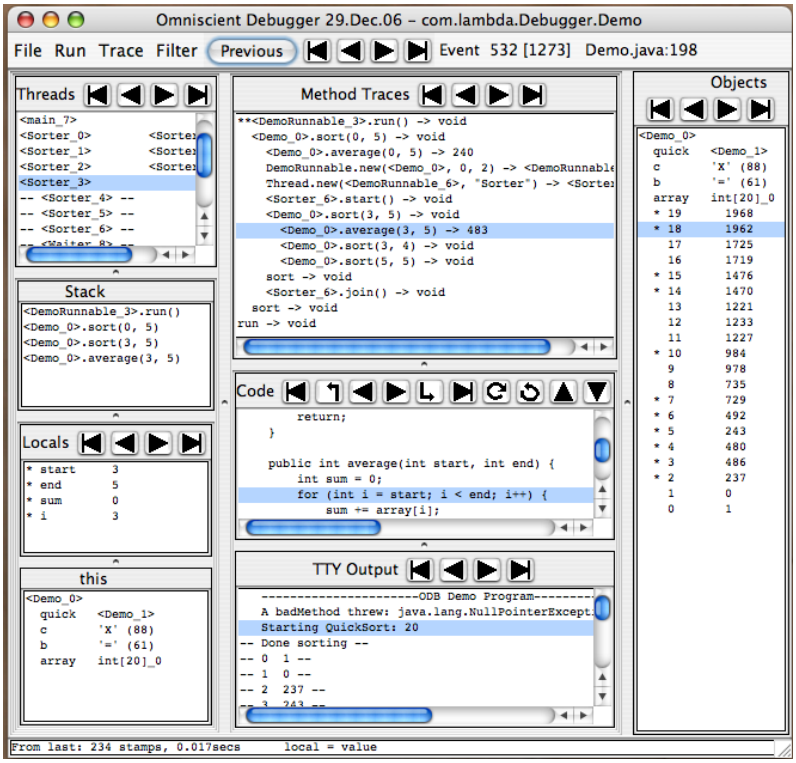

Abbildung 4.9: Benutzeroberfläche des Omniscient Debugger

möglich, *post-mortem* sich vorwärts und rückwärts in der Zeit zu bewegen und sich Variablenwerte zu diesem Zeitpunkt anzusehen. Weiterhin können für eine Variable alle Werte aufgelistet werden, welche sie angenommen hatte. Durch Auswahl eines Wertes kann der Zeitpunkt gewählt werden, an dem sie gesetzt wurde.

[Maruyama u. Terada 2003] weist darauf hin, dass durch die Nutzung von Aufzeichnungen mit Zeitstempeln beim Debugging sich das Paradigma der Stelle im Quelltext (*Location*) wie sie in klassischen Debugging-Ansätzen verbreitet ist, verschiebt zum Begriff der Position als Zeiger auf ein Element des *Trace*. Aus diesem Begriffswechsel erwachsen Möglichkeiten wie dynamische Haltepunkte, Merkerpunkte oder Halte- und Beobachtungspunkte rückwärts entlang der Zeitlinie.

Ein weiterer Schritt kann darin bestehen, das reale System in einen älteren, aus dem Trace rekonstruierten Zustand zu versetzen. Dabei entstehen aber insbesondere bei stark von Eingaben abhängigen Programmen Probleme in Hinblick auf das Ein- und Ausgabeverhalten [Somogyi 2003].

4.3.4 Weitere Methoden

4.3.4.1 Slicing

Das Konzept des *Program Slicing* entstand während der 90er Jahre ausgehend von Arbeiten von Weiser ([Weiser 1984]) und Agrawal ([Agrawal u. a. 1993]). Einen ausführlichen Überblick geben [Tip 1995] und [Lucia 2001].

Der Ausgangspunkt des *Slicing* ist die Diskrepanz zwischen dem Defekt des Systems und seiner Manifestation als Symptom oder Versagen. Der Entwickler verfolgt den Kontrollfluss rückwärts um mögliche Defekte zu finden oder vorwärts um die Auswirkungen des aktuellen Zustands zu antizipieren. Ein Slicing-fähiger Debugger kann den Entwickler bei dieser Deduktionsaufgabe unterstützen, indem die Komplexität der betrachteten Daten reduziert wird.

Basis für das *Slicing* sind Abhängigkeitsgraphen. Die Knoten der Abhängigkeitsgraphen sind in imperativen Programmen die einzelnen Anweisungen. Zwischen diesen existieren zwei Typen von Abhängigkeiten [Zeller 2005]:

- Kontrollabhängigkeiten: Eine Anweisung B ist *kontrollabhängig* von Anweisung A, wenn die Ausführung von B potentiell von A kontrolliert wird.
- Datenabhängigkeiten: Eine Anweisung B ist *datenabhängig* von Anweisung A, wenn A ein Datum schreibt, welches von B genutzt wird und ein Kontrollfluss zwischen A und B existiert, auf dem das Datum nicht überschrieben wird.

Diese Abhängigkeiten können statisch aus dem Programmcode abgeleitet werden (*statisches Slicing*) oder mit aktuellen Werten während des Debugging noch weiter eingegrenzt werden (*dynamisches Slicing*).

Durch Verfolgen der Abhängigkeiten kann bestimmt werden, welche Anweisungen weitere Anweisungen beeinflussen. Die entscheidenden Fragen die ein Slicing-Werkzeug beantworten kann sind:

- *Forward Slicing*: Wo wird dieser Wert im weiteren Programmverlauf verwendet werden?
- *Backward Slicing:* Woher kann dieser Wert kommen?

Diese grundlegenden Methoden können kombiniert werden, indem mehrere *Slices* kombiniert werden.

Obwohl diese Ansätze noch nicht umfassend kommerzialisiert sind, existiert mit *Codesurfer*[6] der Firma GrammaTech bereits ein kommerzielles Produkt. Zukünftige Arbeiten werden verstärkt versuchen *Slicing* auch auf andere Beschreibungsformen als imperative Programmiersprachen anzuwenden, beispielsweise auf graphische ausführbare Modelle, um dort die Aufmerksamkeit des Entwicklers auf relevante Modellteile zu lenken.

4.3.4.2 Delta Debugging

Delta Debugging deckt Fehler hervorrufende Umstände in einem Programm auf. Der Delta Debugging Algorithmus minimiert Testfälle, entdeckt Fehler hervorrufende Änderungen in einem Programm sowie Fehler hervorrufende Anweisungen in einem Ausführungspfad. Dies gelingt dem Algorithmus durch systematisches Testen [Cleve u. Zeller 2000].

Dabei wird ein existierender fehlerhafter Testfall so lange minimiert, bis eine weitere Minimierung das Symptom entfernen würde. In [Zeller 2005] wird als Anwendungsbeispiel die automatisierte Minimierung einer HTML-Seite angeführt, die den Browser Mozilla zum Absturz bringt. Der *Delta Debugging* Algorithmus führt demnach Experimente automatisiert durch.

4.4 Debugging eingebetteter Systeme

Alle bisher beschriebenen Ansätze sind prinzipiell auch auf eingebettete Software anwendbar. Es bestehen jedoch einige Randbedingungen beim Softwareentwurf, welche sich auf das Debugging auswirken Die Fehlersuche in eingebetteten Prozessoren weist daher besondere Schwierigkeiten auf:

- Der Debugger befindet sich nicht im Speicher des Mikrocontrollers, sondern auf einer separaten Entwicklungsumgebung. Es ist also eine Kommunikation zwischen Host-PC und Controller zum interaktiven Debuggen erforderlich. Oftmals ist kein oder nur ein minimales Betriebssystem vorhanden, welches keine Unterstützung für den Debugger bietet.

- Der ausgeführte Code ist in ein Hardwareumfeld eingebunden, welches eine komplexe Initialisierung erfordert. So muss bei marktüblichen Mikrocontrollern die Abbildung von externem und internem, ROM und

[6] Codesurfer Webseite: http://www.grammatech.com/products/codesurfer

RAM-Speicher bei der Initialisierung des Programms konfiguriert werden. In den Chip integrierte Peripherie kann unerwartete *Interrupts* auslösen oder unerwartete Werte liefern.

- Die Umgebung des eingebetteten Prozessors ist oft selbst Teil der Entwicklungsaufgabe. Es ist häufig nicht einfach, die Ursache eines Symptoms der Hardware oder der Software zuzuordnen.

- Eingebettete Systeme mit mehreren Ausführungseinheiten (beispielsweise mehreren Prozessoren und Co-Prozessoren) sind häufig de-facto asynchrone verteilte Systeme. Auch hier ist die Zuordnung eines Defekts auf Basis der Kenntnis von Symptomen erschwert.

- Ein zentrales Element eingebetteter Software ist der Aspekt der Echtzeitfähigkeit, so dass sich beim Debugging der Heisenberg-Effekt häufig auswirkt. So können sogenannte *Race Conditions* (deutsch: Wettlaufsituationen) das Ergebnis eigentlich paralleler Operationen durch Veränderung ihrer Antwortzeit verfälschen.

- Eingebettete Software wird häufig aus Kostengründen auf möglichst preiswerten und einfachen Prozessoren ausgeführt und während der Übersetzung optimiert. Die Übersetzung entfernt neben den Symbolen auch ganze Anweisungen, so dass eine eindeutige Rückabbildung in den Quelltext erschwert oder unmöglich ist. [Elms 1997] versucht Anweisungen im Quelltext so zu interpretieren, dass auch aus den beobachtbaren Daten der Wert „wegoptimierter“ Variablen gewonnen werden kann.

Im den nächsten zwei Abschnitten wird auf verfügbare Debug-Schnittstellen in eingebetteten Systemen eingegangen, zunächst für eingebettete Prozessoren, dann für komplexere zusammengesetzte Ein-Chip- oder Mehr-Chip-Systeme.

4.4.1 Debug-Schnittstellen für eingebettete Prozessoren

Es lassen sich zwei Arten von Schnittstellen zum Debugging für eingebettete Prozessoren unterscheiden: *On-Chip Debugging* und *In-Circuit Emulation* [O'Keeffe 2006].

4.4.1.1 On-Chip Debugging

Beim On-Chip Debugging wird der Mikrocontroller selbst zur Fehlersuche verwendet. Hierbei wird allgemein eine Kombination aus Hardware und Software, auf dem Chip und außerhalb verwendet. Das Verhältnis zwischen Hardware

und Software ist bei den einzelnen Controller-Familien unterschiedlich. Rahmenbedingungen wie Kosten und Chipfläche entscheiden über den Umfang der zur Verfügung gestellten Debugging-Hardware.

Bei einfachen Mikrocontrollern steht zum Debugging ein software-basierter Monitor im Speicher des Controllers zur Verfügung. Breakpoint-Register nehmen Adressen, bei deren Erreichen die Ausführung unterbrochen wird, und die notwendigen Konfigurationsbits auf. Mit der Belegung dieser Bits wird über die Art des Haltepunkts entschieden [Ball 2000]. Des Weiteren kann Hardware zum Lesen und Schreiben von Werten im Speicher oder in Registern vorhanden sein.

Die Anbindung an den Entwicklungs-PC geschieht über separate Schnittstellen. Die Bandbreite reicht hier von der seriellen Kommunikation mittels RS232, über standardisierte Test-Schnittstellen wie JTAG (siehe Abschnitt 4.4.2.1) bis hin zu dedizierten Debuginterfaces wie BDM (Abschnitt 4.4.2.2) [Haller 1997].

Im Allgemeinen werden folgende Fähigkeiten zum Debuggen realisiert:

- Lesen und Ändern von Inhalten im Speicher
- Lesen und Schreiben auf I/O-Schnittstellen (bei einfachen Monitoren für Kommunikation mit dem Host nötig)
- Neuen Code auf das Target laden
- Breakpoints setzen
- Einzelschrittausführung.

Die Verwendung eines On-Chip Debuggers beeinflusst die Ausführung der zu untersuchenden Software. So werden beim Debugging Ressourcen belegt, die sonst dem Anwenderprogramm zur Verfügung stehen. Die Ausführung eines Monitors benötigt beispielsweise CPU-Zeit. Die Verwendung des Debuggers initialisiert Teile des Mikrocontrollers. Dies führt zu veränderten Bedingungen beim Programmstart und überdeckt möglicherweise die Ursache für ein Fehlverhalten.

4.4.1.2 In-Circuit Emulation

Ein *Emulator* ist ein separates Gerät, das einen Prozessor in seinen funktionalen, elektrischen und mechanischen Eigenschaften nachbildet. Bei der *In-Circuit Emulation* werden die Aufgaben des Prozessors von einem Emulator übernommen. Hierfür wird der Mikrocontroller aus der Schaltung entfernt.

An seine Stelle wird ein Probenkopf, der sogenannte *Bond-out Chip*, eingesetzt [Haller 1997]. Dieser ist über eine vieladrige Leitung mit einem externen Emulator-System verbunden.

Neben den herkömmlichen Debugging-Funktionen wie Breakpoints, Einzelschrittausführung, Inspektion von Variablen, können weitergehende Information abgegriffen werden. Da das Emulator-System die Funktion der CPU übernimmt, kann es auch Einblick in die Zustände der internen Speicher, wie beispielsweise den Stack geben und diese auch verändern. Des Weiteren sind Funktionen wie der Echtzeitzugriff auf den Speicher, Trace-Speicher, komplexe Trigger für Breakpoints und Performance-Analyse möglich [Karcher 2008]. Diese Funktionen können unabhängig von der Debugschnittstelle des Produktivsystems realisiert werden.

4.4.2 Debug-Schnittstellen für Systems-on-Chip

Die Methoden- und Werkzeuglandschaft für das Debugging von Funktionalität, welche aus gemischten Hardware- und Softwaresubsystemen besteht, stellt sich komplizierter dar, als im Falle reiner Softwaresysteme, welche auf nur einem Prozessor ausgeführt werden.

Die langsam gewachsene Komplexität führte in der Vergangenheit zu zahlreichen Insellösungen zum Debuggen von Systemen und Subsystemen, welche über Herstellergrenzen hinweg nur schlecht kombinierbar und in gemeinsamen Werkzeugen untersuchbar waren. Dieses auch oft politisch und vermarktungstechnisch gewollte Vorgehen führt aber mit immer komplexeren auf einem physikalischen Chip integrierten Systemen in eine Sackgasse, was auch von kommerziell agierenden Herstellern von Mikrochips, IP-Lösungen und Debugwerkzeugen zunehmend erkannt wird.

[Berent 2007] benennt folgende Bereiche, welche von Standardisierungsbemühungen profitieren können:

- **Softwarewerkzeuge zum Debugging:** Die Hersteller von Debug-Lösungen verstehen sich meist in erster Linie als Hardware-Entwickler, welche zusätzlich ein festgelegtes *Front-End* zur Steuerung der Schnittstelle der eigenen Debug-Lösung anbieten. Der Nutzer kann also nicht sein bevorzugtes Softwarewerkzeug einsetzen. Daraus ergeben sich einige Probleme:

  - In den meisten Fällen ist ein Zugriff nur über die angebotene Benutzeroberfläche möglich. Ausnahmen, wie die in Abschnitt 6.1.2

beschriebene Lösung der Firma iSystem, bieten zumindest eine proprietäre Schnittstelle an.

- Jede Software besitzt ein eigenes Protokoll, um mit der angeschlossenen Hardware zu kommunizieren. Besteht ein komplexes System aus mehreren Subsystemen unterschiedlicher Hersteller, müssen gegebenenfalls nicht nur mehrere Schnittstellen, sondern auch mehrere Bedienoberflächen parallel genutzt werden.
- Werden mehrere Subsysteme von einem Softwarewerkzeug unterstützt, kann es zu Problemen bei der Beschreibung des zusammengesetzten Zielsystems kommen. Eine standardisierte Methode zur *Black-Box*-Beschreibung von zusammengesetzten Systemen würde flexible Debuggingwerkzeuge erst ermöglichen.
- Bei Nutzung mehrerer Softwarewerkzeuge sollten diese koppelbar sein, um beispielsweise die Ablaufkontrolle zu vereinheitlichen oder Systemereignisse (sogenannte *Cross-Trigger*) setzen zu können.

- **Inkompatible Schnittstellen zu Debug-Hardware:** Es existieren zahlreiche Schnittstellen zur Ansteuerung von Debug-Hardware. Dies ist eine direkte Folge der im ersten Punkt beschriebenen monolithischen Struktur zwischen Software und Hardware. Dies bedeutet hohe Entwicklungskosten für die Softwarehersteller.

- **Inkompatible Hardwareschnittstellen des *System-on-Chip*:** Der klassische Industriestandard für Hardwareschnittstellen ist die im Folgenden näher beschriebene JTAG-Schnittstelle. Diese weist jedoch für einige Anwendungen zu viele Pins auf, unterstützt kein echtzeitfähiges Tracing und besitzt nur eine stark begrenzte Geschwindigkeit. Dies führte zu einer großen Zahl besser geeigneter aber untereinander inkompatibler Systemschnittstellen der Hersteller.

- **Unterschiedliche chipinterne Debug-Architekturen:** Unterschiedliche Komponenten, speziell Rechenkerne, besitzen verschiedene Debug-Architekturen. Neben direkter Einbindung in die JTAG-Kette, existieren busbasierende Architekturen wie *Nexus 5001* oder *CoreSight* (siehe unten), welche häufig proprietär sind.

Abbildung 4.10 fasst die verschiedenen Bereiche der Debug-Infrastruktur zusammen und benennt existierende relevante Ansätze und (geplante) Standards. Diese konzentrieren sich häufig auf nur einen Bereich, überspannen und integrieren teilweise jedoch auch mehrere Ebenen.

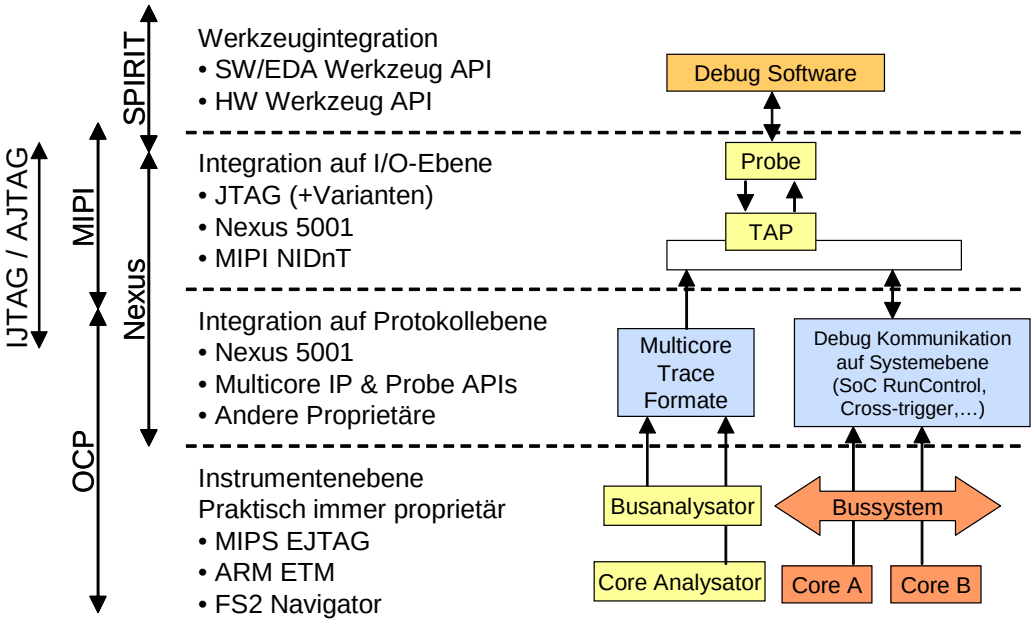

Abbildung 4.10: Bereiche in Debug-Infrastruktur und Standardisierungsansätze nach [Stollon u. a. 2007]

4.4.2.1 JTAG

Die Abkürzung JTAG bezeichnet ursprünglich die *Joint Test Action Group*, einen Zusammenschluss von Halbleiterherstellern seit dem Jahr 1985. Dieser erarbeitete einen Standardtest elektronischer Schaltungen auf der Platinenebene, welcher auch unter dem Namen *Boundary Scan*-Test bekannt ist. Der Standard wurde mehrfach erweitert und unter der Norm IEEE 1149.1-1990 festgehalten [IEEE 1990b].

Kern des Standards war die Idee, eine Schaltung ohne physischen Zugriff (so genannten In-Circuit Test) messen zu können, was durch die Miniaturisierung und die Verwendung von Mehrlagenplatinen oft unmöglich ist. Obwohl es ursprünglich für auf Platinen integrierte Schaltungen entwickelt wurde, wird JTAG heute hauptsächlich zum Zugriff auf Subsysteme in integrierten Schaltungen genutzt.

Über seine ursprüngliche Aufgabe als Testverfahren hinaus kann die JTAG-Schnittstelle in aktuellen Anwendungen für weitere Funktionen eingesetzt werden:

- Programmierung von Speicherbausteinen, z.B. Flash-Speichern

- Konfiguration von FPGAs und anderen PLDs[7].
- Debugging von Software-Anwendungen auf eingebetteten Prozessoren. Beim Debugging greift der Debugger über die JTAG-Schnittstelle mit einem proprietären Protokoll auf ein in eine CPU integriertes Debugging-Modul zu. Die Schnittstelle dient als Transportschicht.

Interessant für die Anwendung beim Debugging sind in erster Linie zwei Aspekte des Standards: Der elektrische und physikalische Anschluss *JTAG-TAP* und sein grundlegendes Protokoll im *TAP Controller*.

JTAG-TAP steht für *JTAG Test Access Port* und definiert die Pins, welche die Schnittstelle zur Verfügung stellt:

- TCK (Test Clock): Der Eingang TCK definiert den vom Systemtakt unabhängigen Takt der JTAG-Logik.
- TMS (Test Mode Select): Das Eingangssignal dient der Steuerung einer Zustandsmaschine des in jedem Gerät vorhandenen *TAP Controller* und dient somit auch der Selektion mehrerer in die JTAG-Kette eingebundener Geräte.
- TDI (Test Data In): TDI ist der serielle Eingang für Testinstruktionen und Daten.
- TDO (Test Data Out): TDO ist der serielle Ausgang für Instruktionen und Daten.
- TRST (Test Reset): Dieser optionale Ausgang erlaubt ein asynchrones Rücksetzen des *TAP Controller*.

Abbildung 4.11 zeigt die Ein- und Ausgänge (ohne Reset-Eingang) und die Verschaltung mehrerer Geräte an einem JTAG-Port. Die TDI- und TDO-Pins der einzelnen Subsysteme bilden eine Kette (*Daisy-Chain)*, durch welche die Signale seriell geschoben werden.

4.4.2.2 Weitere Schnittstellen und Standardisierungsansätze

Der folgende Abschnitt gibt einen kurzen Abriss über weitere verbreitete Schnittstellen und Standardisierungsansätze.

[7] *Programmable Logic Device*: Oberbegriff für programmierbare logische Schaltungen.

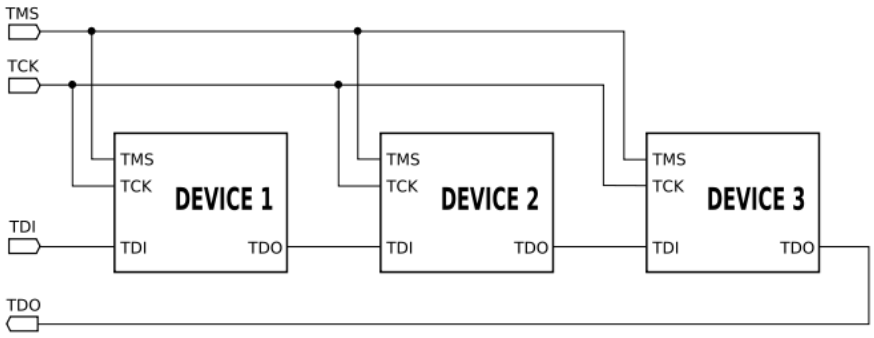

Abbildung 4.11: Serielle Anordnung mehrerer Komponenten in einer JTAG-Chain

Das SPIRIT-Konsortium besteht aus einem Zusammenschluss mehrerer Chiphersteller wie ARM, Infineon, STMicroelectronics und NXP auf der einen Seite und Entwicklern von Entwurfswerkzeugen wie Mentor Graphics und Synopsys auf der anderen Seite.

Das SPIRIT-Konsortium definierte den **IP-XACT**-Standard, welcher im Juni 2006 bei der IEEE als *IEEE P1685 Working Group Proposal* eingereicht wurde [SPIRIT Schema Working Group Membership 2005, 2006]. IP-XACT beschreibt ein XML-basiertes Datenmodell, welches dazu dient *IP-Cores* unterschiedlicher Hersteller mit einheitlichen Metadaten zu beschreiben. Ein IP-XACT-Dokument beschreibt ein Gesamtdesign, Komponenten und Bussysteme auf Basis ihrer Schnittstellen, Register, Speicherbelegungen, Signale und Debugschnittstellen. Das Verhalten der Subsysteme wird nicht beschrieben.

Für Entwickler von Debugging-Lösungen kann eine IP-XACT-Beschreibung der Konfiguration des Debuggers für unterschiedliche, auch komplexe Zielsysteme dienen. Die *Device Software Development Platform* der Eclipse Foundation[8] nutzt diese bereits für die Plattformbeschreibung angeschlossener eingebetteter Zielsysteme.

Die *Mobile Industry Processor Interface* (**MIPI**)-Allianz plant die Entwicklung eines vereinheitlichten Konnektors zum Debugging mobiler hochintegrierter Geräte mit reduzierter Pinanzahl und *Trace*-Fähigkeiten [The Mobile Industry Processor Interface (MIPI) Alliance 2008].

Nexus 5001 ist ein 2003 in Zusammenarbeit der Chiphersteller *AnalogDevices*, *Freescale*, *Infineon*, *ST* und *TexasInstruments* mit verschiedenen Werkzeugherstellern als IEEE-ISTO 5001-2003 (siehe [Nexus 5001 1999]) standardisierter Debug-Konnektor und ein Trace-fähiges hochperformantes paketbasiertes Protokoll. Das Protokoll standardisiert Ablaufkontroller, Speicherzugriff,

[8]DSDP Webseite: http://www.eclipse.org/dsdp

Haltepunkte, Instruktions- und Daten-Trace, Speichereinblendung und Portersatz sowie allgemeine Datenakquisition. Der Standard definiert unterschiedlich *Compliance Levels*, welche von Subsystemen erfüllt werden können [O'Keeffe 2008].

Der **IJTAG**-Ansatz plant die Entwicklung eines standardisierten Protokolls zum Zugriff auf Fähigkeiten von Subsystemen über eine JTAG-Schnittstelle. Dies beinhaltet neben Debug-Fähigkeiten auch Monitoring-Informationen, Testdaten oder Zuverlässigkeitsinformationen. Zentral ist dabei, dass das Verhalten der JTAG-Schnittstelle unangetastet bleibt und der *TAP Controller* sich nicht verändert. Derzeit ist dieser Ansatz noch in der Entwicklungsphase.

Die *Open Core Protocol International Partnership* (**OCP-IP**)[9] ist ein neutrales Protokoll für *Networks-on-Chips*. Die Debug-Arbeitsgruppe konzentriert sich auf die Signalisierung zwischen mehreren Kernen und anderen Subsystemen und definiert grundlegende Unterstützung von Debugsystemen durch das Bussystem. Es integriert sich in den Nexus 5001-Standard.

Neben diesen Standardisierungsansätzen existieren bereits herstellerabhängige Ansätze. Ein Vertreter ist *CoreSight*[10] der Firma ARM.

Ein weiterer Ansatz ist *ChipScope*[11] des FPGA-Herstellers Xilinx. Hier können Logikanalysatoren, Busanalysatoren und teilweise *Profiler* in ein FPGA-Design integriert und Signale auf einem angeschlossenen Entwicklungsrechner angezeigt und analysiert werden.

Der an der Universität Berkeley verfolgte Ansatz für eine integrierte Debug-Umgebung für rekonfigurierbare Komponenten in [Camera u. a. 2005] beruht auf der Nutzung eines Hardware-Betriebssystems. In einem *Stitching* genannten Prozess wird das Funktionsmodell um eine Debug-Schnittstelle erweitert. Dieses Vorgehen ist mit der in Abschnitt 6.2.2 beschriebenen generierbaren Schnittstelle vergleichbar, instrumentiert jedoch ausgehend von VHDL-Code und nicht für graphische Modelle.

4.5 Graphische Modelle und Debugging

Die bisher beschriebenen Debugging-Techniken beruhen in erster Linie auf reiner Präsentation in Textform – in manchen Fällen jedoch strukturiert als Tabelle oder Baum.

[9] OCP Webseite: http://www.ocpip.org/home

[10] CoreSight Webseite: http://www.arm.com/products/solutions/CoreSight.html

[11] ChipScope Webseite: http://www.xilinx.com/ise/optional_prod/cspro.htm

Text kann auf einfache Weise Information vermitteln, ist jedoch inhärent sequentiell, das heißt zur übersichtlichen Darstellung von Vernetzung und Zusammenhängen nur bedingt geeignet. Dies gilt insbesondere für die Darstellung von Verknüpfungen von Objekten und allen Beziehungen, welche einen Ganzes-Teil-Zusammenhang konstituieren.

Weiterhin ist eine textuelle Darstellung von Information dann nicht ausreichend, wenn die Spezifikation des Systems bereits in graphischer Form erfolgte.

Die Kombination graphischer Modelle und Debugging besitzt demnach grundsätzlich zwei Anwendungsgebiete:

- Debugging *mit* graphischen Modellen: Dies bedeutet die bereits beschriebene Aufbereitung von Laufzeitinformation in Form von Diagrammen. Eine graphische Spezifikation ist keine notwendige Vorraussetzung. Diese Ansätze werden durch den Oberbegriff *Software Visualisierung* abgedeckt.
- Debugging *von* ausführbaren graphischen Modellen: Wurde eine Modell graphisch entworfen und kann durch eine geeignete Modell-zu-Code-Transformation ausgeführt werden, sollen Laufzeitinformationen gewonnen und dem Beobachter angezeigt werden können. Eine graphische Darstellung ist hier keine notwendige Vorraussetzung.

Der Schwerpunkt dieser Arbeit liegt auf der Kombination beider Punkte, also dem Debugging von graphischen Modellen mit einer möglichst entwurfsnahen Darstellung als graphisches Modell.

Daher wird im folgenden Teilabschnitt zunächst untersucht, wie graphische Modelle zum Debugging nicht-modellbasiert entwickelter Software eingesetzt werden. Darauf aufbauend beschreibt Abschnitt 4.5.2 Stand der Forschung und kommerziell erhältliche Werkzeuge zum Debugging von Modellen. Dies führt schließlich zu einer Aufstellung von Anforderungen an den Einsatz graphischer Darstellung für modellbasiert entworfene Systeme.

4.5.1 Graphische Debugger

Der Bereich der Softwarevisualisierung fasst alle Bereiche und Techniken zusammen, welche sich mit statischer oder dynamischer Darstellung von Algorithmen, Progammen und Daten befassen. Softwarevisualisierung befasst sich dabei in erster Linie mit der Analyse von Programmen und ihrer Entwicklung.

Das Ziel ist hierbei, zu einem besseren Verständnis von unsichtbaren oder unüberschaubaren Eigenschaften von Software zu gelangen, insbesondere bei

Vorliegen großer Informationsräume in Bereichen wie Softwarewartung oder *Reverse Engineering*. Die Hauptaufgabe sieht der Bereich der Softwarevisualisierung in der Suche nach effektiven Abbildungen verschiedener Aspekte von Software in graphische Darstellungen mit passenden visuellen Metaphern [Gracanin u. a. 2005].

Softwarevisualisierung umfasst demnach auch Methoden zur Visualisierung von Software zur Laufzeit.

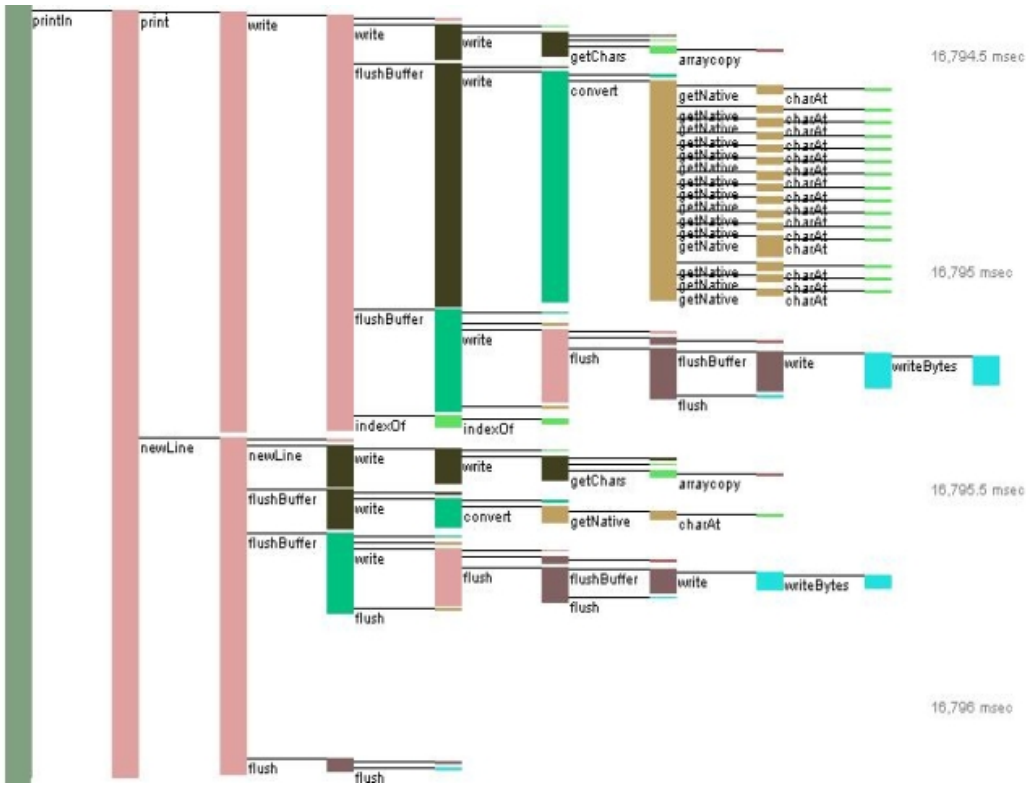

Abbildung 4.12: Visualisierung in JInsightLive

Dies geschieht beispielsweise zur Präsentation der Messergebnisse von Profilern in statistischen Diagrammen. Eine modellnahe Darstellung im Werkzeug *JInsight Live*[12] von IBM zeigt Abbildung 4.12: Die Aufzeichnung einer Abfolge von Operationsaufrufen als Trace in einem Java-basierten Webserver wird in einer Darstellung ähnlich eines Sequenzdiagramms der UML aufbereitet.

Auch zum Debugging im Sinne der in Abschnitt 4.3.2 beschriebenen klassischen Ansätze können visuelle Diagramme beitragen.

Als Lösung für das Debugging von Java-Programmen existiert das *Java In-*

[12] JInsight Live Webseite: http://www.alphaworks.ibm.com/tech/jinsightlive

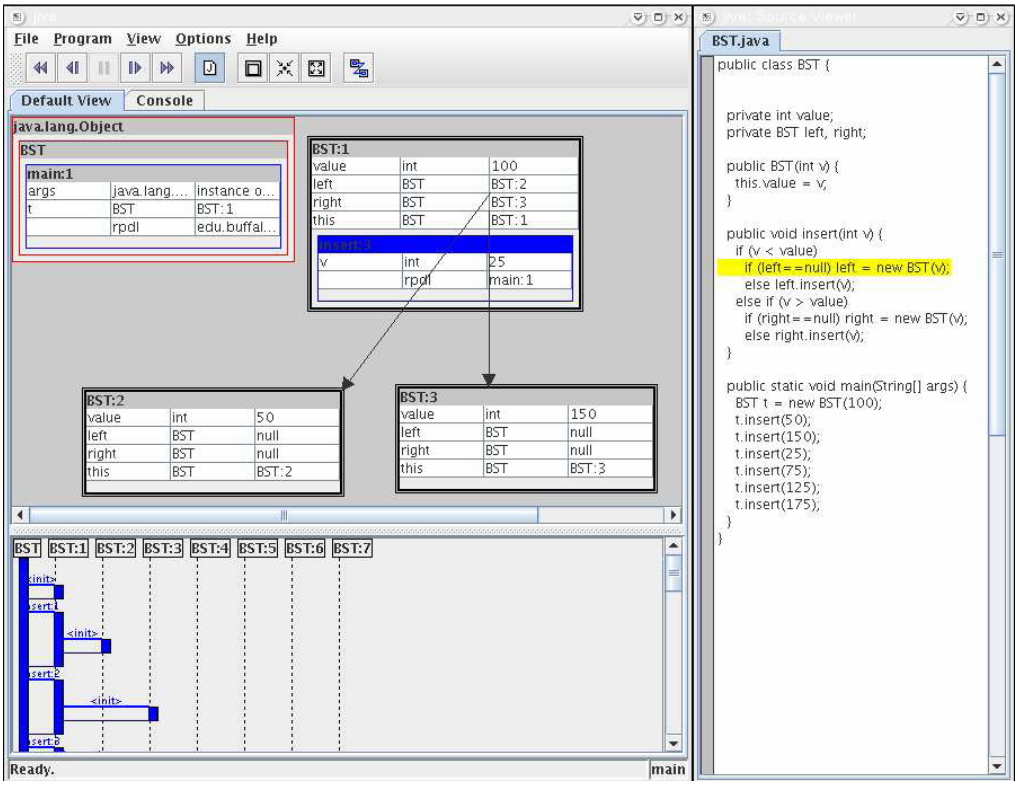

Abbildung 4.13: Benutzeroberfläche von JIVE (aus [Gestwicki 2005])

teractive Visualization Environment JIVE [Gestwicki 2005]. Das in Abbildung 4.13 gezeigte JIVE ist eine interaktive Ausführungsumgebung für Java-Programme, welche einen visuellen Ansatz zum Debuggen objektorientierter Software erlaubt und ursprünglich als Werkzeug zum Einsatz in der Lehre konzipiert wurde. JIVE integriert sich in die Entwicklungsumgebung Eclipse.

JIVE bildet sowohl den Zustand zur Laufzeit, als auch die Ausführungsgeschichte eines Programms visuell in Diagrammen auf. Der Zustand wird als modifiziertes Objektdiagramm präsentiert. Die Ausführungsgeschichte wird ähnlich wie bei *JInsight Live* als vereinfachtes Sequenzdiagramm visualisiert.

Einen ähnlichen Ansatz wie die Objektdarstellung von JIVE verfolgt das Projekt eDOBS der Universität Kassel [Geiger u. Zündorf 2006]. Hier wird eine Objektansicht in das Debugging von Java-Programmen integriert.

Für C-Programme wurde bereits 1995 die erste Version des *Data Display Debugger* (DDD) veröffentlicht [Zeller 2004]. Dieser stellt eine graphische Benutzeroberfläche für den textbasierten GNU Debugger gdb (siehe Abschnitt 4.3.2.1) dar. Der DDD besitzt eine, in Abbildung 4.14 gezeigte Ansicht mit welcher durch Zeiger verkettete Strukturen und Klassen in C- und C++-

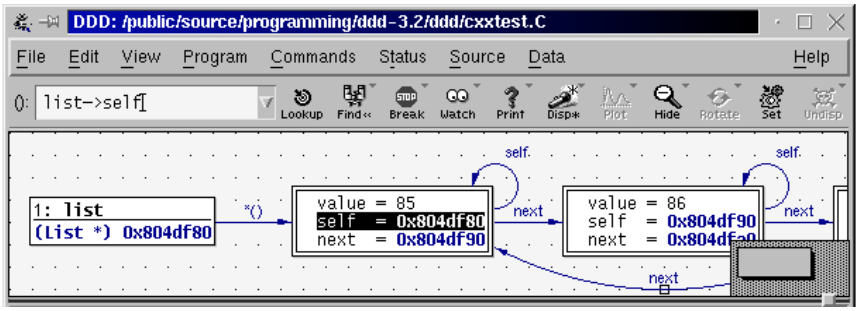

Abbildung 4.14: Objektansicht in DDD

Programmen zur Laufzeit in Diagrammen interaktiv untersucht werden können. Die Ansicht erinnert an Objektdiagramme; die Anordnung der Instanzen erfolgt jedoch nicht frei, sondern, wie in der Abbildung erkennbar, von links nach rechts. Einen vergleichbaren, in der Syntax näher an der UML orientierten Ansatz verfolgen [Malloy u. Power 2005].

[Sajaniemi u. Kuittinen 2003] beschreiben ein System zur Programmanimation *PlanAni*. *PlanAni* bereitet Inhalte von Variablen graphisch auf. Die Art der Darstellung leitet sich dabei aus der Rolle der betrachteten Variable ab, beispielsweise als Konstante, Zähler, *Flag* oder Akkumulator. Die Darstellung geht dabei weiter auf die Funktion eines Datums ein als in anderen Ansätzen, ist derzeit aber nur für Unterrichtszwecke einsetzbar. Die Visualisierung synchronisierter nebenläufiger Abläufe in Java mit Sequenz- und Kollaborationsdiagrammen beschreiben [Mehner u. Wagner 2000].

4.5.2 Graphische Debugger für Modelle

Die im vorigen Abschnitt beschriebenen Debugger beziehen sich immer auf Quelltext und seine Artefakte. Bei modellgetriebener Entwicklung bietet sich, wie schon in Kapitel 1 motiviert, eine Darstellung nahe der Entwurfsnotation an. Da die modellgetriebene Entwicklung sich derzeit noch entwickelt oder auf proprietäre Insellösungen zurückzieht, sind graphische Debugger für mit graphischen Modellen entwickelte Systeme noch selten.

Ein früher Ansatz, wie in [Schumann 2000] beschrieben, konzentrierte sich darauf, UML Sequenzdiagramme durch Umwandlung in Zustandsautomaten simulierbar zu machen und durch Simulation Fehler in der graphischen Spezifikation zu finden.

Neben wissenschaftlichen Ansätzen existieren vereinzelte proprietäre Lösungen und Ansätze für spezielle Modellierungswerkzeuge. Dabei soll im Folgenden nicht auf alle existierenden Simulationsmöglichkeiten in Modellierungswerk-

zeugen eingegangen werden. Es wurden nur Ansätze berücksichtigt, welche von einem auf einem Zielsystem ausgeführten Modell ausgehen. Auch wurden die zahlreichen Ansätze nicht beachtet, welche ein *Rapid-Prototyping* System benötigen, welches nicht der Zielplattform entspricht.

Für das UML-Werkzeug *Bridgepoint* der Firma Mentor Graphics existiert eine Erweiterung *Bridgepoint Verifier*[13], welche ausführbare UML-Modelle für eine eigene virtuelle Maschine aufbereitet und dort ausführt. Es bietet während der Ausführung Möglichkeiten, die dem klassischen Debugging entsprechen: Setzen von Haltepunkten, Einzelschritte durch Verhaltensbeschreibungen mit Actions und Anzeige von Attributen und lokalen Variablen. Die Inspektion ist dabei in das UML-Werkzeug integriert. Der Ansatz verfolgt jedoch durch die Ausführung in einer virtuellen Maschine auf dem Entwicklungsrechner im Kern einen Simulationsansatz.

Für eingebettete Systeme bietet die Firma iSystem das *winIDEA Embedded Test Integration Toolkit for LabVIEW*[14] an. Hier können spezielle Blöcke in ein Modell des Werkzeugs *LabView* integriert werden, über welche auf eingebettete Software zugegriffen werden kann. Das graphische Modell in *LabVIEW*[15] ist hier jedoch nicht Gegenstand der Ausführung auf dem eingebetteten System; es findet demnach im eigentlichen Sinne kein Debugging statt.

2008 stellte die Firma Lauterbach[16] eine prototypische Integration ihrer Debug-Lösungen für eingebettete Prozessoren in das UML-Werkzeug *Rhapsody*[17] der Firm Telelogic vor. Die Integration konzentrierte sich jedoch auf die Ausführung von Modellen. Zum Debugging steht nur das Setzen von Haltepunkten in einem Klassendiagramm in *Rhapsody* zu Verfügung.

4.5.3 Anforderungsanalyse Debugging-Plattform für Modelle

Für eine Lösung zum Debugging ausführbarer graphischer Modelle mit Darstellung in Form graphischer Modelle wurden somit folgende Anforderungen an eine Lösung identifiziert:

1. **Modellnahe Darstellung:** Wie bereits in Kapitel 1 angesprochen, verändert die Transformation die Beschreibungsartefakte bei einem modell-

[13]Bridgepoint Verifier Webseite: http://www.mentor.com/products/sm/uml_suite/bridgepoint_verifier/index.cfm

[14]Webseite: http://www.isystem.com/labview

[15]LabVIEW Webseite: http://www.ni.com/labview

[16]Lauterbach Webseite: http://www.lauterbach.com

[17]Rhapsody Webseite: http://modeling.telelogic.com/products/rhapsody/index.cfm

getriebenen Entwicklungsansatz dramatisch, speziell bei der Überführung in Code. Um den kognitiven Prozess des Beobachtens der Systemfunktion zu unterstützen ist eine Darstellung adäquat, die möglichst weit reichend der Entwurfsnotation entspricht und als bekannt vorausgesetzt werden kann.

2. **Klare Visualisierung:** Die Visualisierungsumgebung sollte Diagramme automatisch so anordnen, dass die Artefaktstruktur klar wird. Falls durch proprietäre Visualisierungen Information besser dargeboten werden kann, kann die Regel der Darstellung nahe der Entwicklungsnotation durchbrochen werden.

3. **Schlanke Benutzerschnittstelle:** Die Benutzerschnittstelle eines Debuggers für Modelle sollte möglichst schlank und intuitiv sein. Der Grundgedanke der *kognitiven Ökonomie* ist hierbei, dass die kognitive Belastung des Benutzers möglichst minimiert werden sollte [Tudoreanu 2003]. [Ungar u. a. 1997] beschreiben drei Aspekte der Unmittelbarkeit, welche ein Debugger erfüllen sollte:

   - zeitlich: Zeitliche Nähe von Auswirkungen zu Aktionen, beispielsweise automatische Aktualisierung von Anzeigen
   - räumlich: Konsistenz durch Darstellung von Information möglichst nahe an ihrem Kontext, beispielsweise bevorzugte Annotation von Daten im Diagramm anstatt in getrenntem Fenster
   - semantisch: Die konzeptionelle Distanz zwischen untersuchtem System und der gegebenen Information sollte möglichst gering sein. Dies entspricht der modellnahen Darstellung in Punkt 1.

4. **Integration in eine gemeinsame Entwicklungsumgebung:** Die Punkte 2 und 3 bedingen, dass eine Intergration in eine gemeinsame Benutzeroberfläche wünschenswert ist. Eine Kopplung von Insellösungen widerspricht dieser Anforderung und verlangt vom Benutzer des Debuggers eine Auseinandersetzung mit der Benutzerschnittstelle jedes einzelnen Werkzeugs.

5. **Flexible Anwendbarkeit auf verschiedene Zielsysteme:** Eine Debuggingplattform sollte nicht nur auf eine einzelne Familie von Ausführungseinheiten zugeschnitten sein. Dies ist insbesondere wichtig, wenn der Anspruch der Plattformunabhängigkeit bei der Modellierung ernst genommen werden soll.

6. **Integrierbarkeit von Ansichten und Modellierungsnotationen:** Die unter Punkt 4 angeführte Integrierbarkeit in ein Werkzeug bedingt

die Anforderung, dass ein Debugger über klar definierte Schnittstellen um weitere Systemsichten und Notationen erweiterbar sein soll.

7. **Gleichzeitige gemischte Ausführung:** Während einer Debugging-Sitzung sollen gemischte Systeme unterstützt werden. Dies soll über die beiden in den Punkten 5 und 6 angeführten Achsen der Flexibilität möglich sein: Für ein *Debuggee* sollten unterschiedliche Sichten möglich sein, wenn es mit mehreren Modellierungssprachen entworfen wurde. Weiterhin sollten verteilte Systeme, deren Teile auf unterschiedlichen Ausführungseinheiten ausgeführt werden integriert und synchron im Debugger visualisiert werden können.

8. **Wiederholbarkeit:** Post-Mortem Debugging auf Aufzeichnungen von Systemabläufen ermöglicht die Reproduzierbarkeit von Erkenntnissen. Dies ist umso wichtiger, als in eingebetteten Systemen Situationen und damit Symptome oder Systemversagen nicht immer einfach wiederholbar sind.

9. **Vorwärts- und Rückwärtsausführung:** Ein Modelldebugger unterstützt idealerweise die Vorwärts- und Rückwärtsnavigation in Einzelschritten entlang der Zeitachse der Modellausführung. Die Granularität der Ausführung sollte der gezeigten Modellansicht entsprechen.

Kapitel 5

Ausführung und Fehlersuche in ModelScope

Nachdem im vorhergehenden Kapitel die im Kontext der Fehlersuche existierenden Techniken zum Debugging abgegrenzt und vorgestellt wurden, wird im folgenden Kapitel die im Rahmen dieser Arbeit entstandene Debugging-Plattform *ModelScope* beschrieben [Graf u. Müller-Glaser 2008].

Abschnitt 5.1 stellt dazu zunächst die auf Eclipse basierende Plattform zur Modellverwaltung vor. Neben der Verwaltung von in Dateien oder Datenbanken vorliegenden Modellen ermöglicht diese die Erzeugung von Quelltext. Dies kann über eine auf der Programmiersprache Python basierende Skriptsprache erfolgen oder mit Hilfe einer datengetriebenen, in das Gesamtmodell integrierten graphischen Modellierung.

Zentral für das Verständnis des vorliegenden Ansatzes ist die Idee, dass die Visualisierung des Systemzustands eine Rücktransformation von Laufzeitinformationen von niedrigeren Schichten, beispielsweise des Quelltexts, hin zum Modell und eine Verknüpfung mit diesem benötigt. Dieser Ansatz wird in Abschnitt 5.2 beleuchtet.

Das Transformationskonzept wurde in einem weiteren Entwurfsschritt in eine in Abschnitt 5.3 untersuchte schichtenbasierte Softwarearchitektur integriert, welche eine modulare Verknüpfung unterschiedlicher Visualisierungen, Transformationstechnologien und Ausführungsplattformen ermöglicht und damit für die Fehlersuche in heterogenen Modellen geeignet ist.

Abschnitt 5.3 umfasst neben der Grundstruktur eine Beschreibung wichtiger Realisierungsaspekte der Softwareplattform, darunter das Plug-In-Konzept, die Visualisierung und die Benutzeroberfläche.

Spezielle in dieser Arbeit untersuchte Systemsichten und Zielarchitekturen werden in den folgenden Kapiteln 6 und 7 untersucht.

5.1 Basisplattform

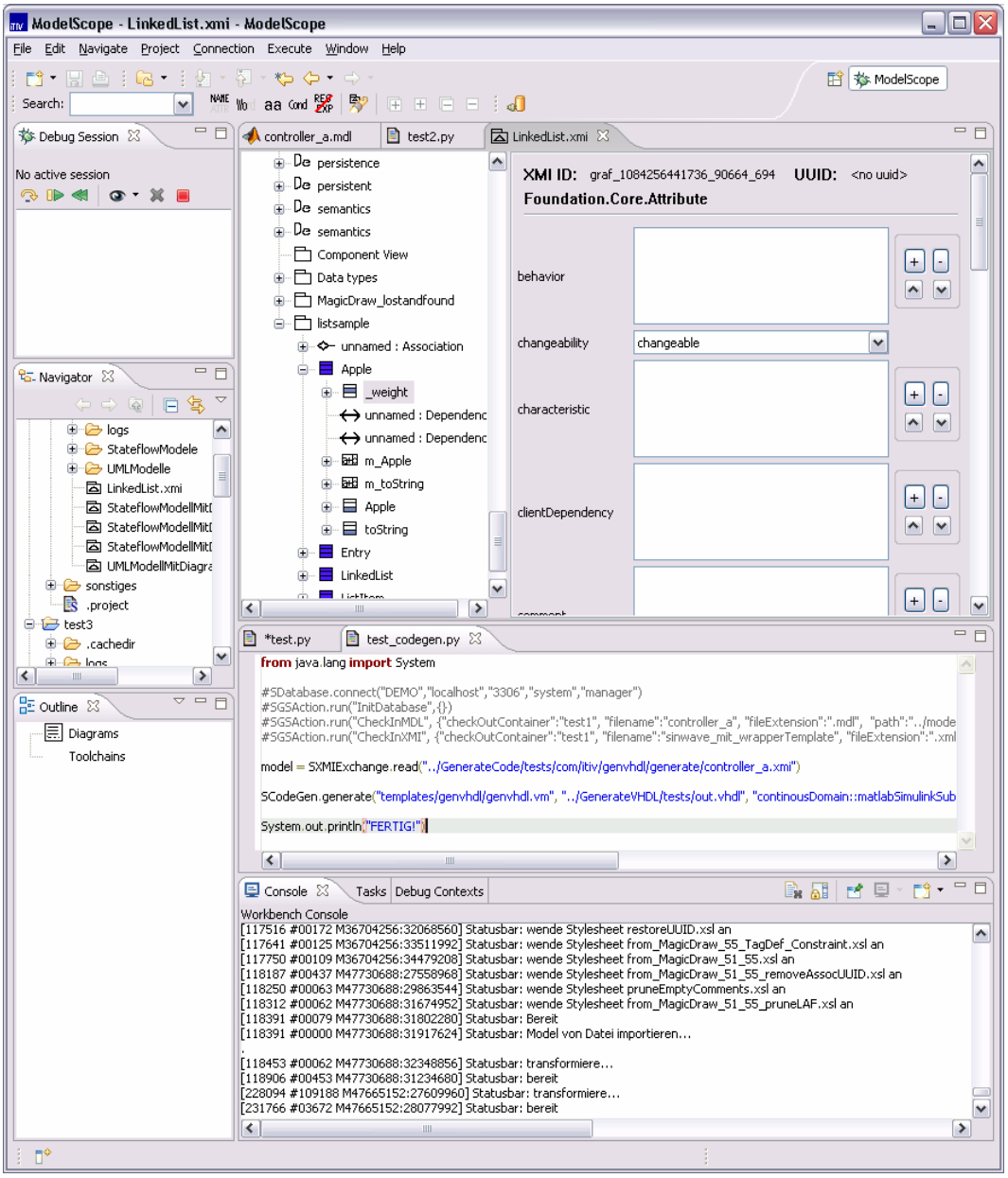

Abbildung 5.1: Benutzeroberfläche von ModelScope

Die Funktionalität zur Modellverwaltung von *ModelScope* basiert auf den in Abschnitt 3.1.5 beschriebenen Fähigkeiten der Integrationsplattform Aquintos.GS. Diese wird jedoch mit einer neuen Benutzeroberfläche angesteuert, die auf dem *Eclipse Framework* basiert[1].

Abbildung 5.1 präsentiert die Benutzeroberfläche mit der für Eclipse typischen Anordnung von *Editoren* im rechten oberen Bereich und darum gruppierten *Views*.

5.1.1 Modellverwaltung

Ausgangspunkt bei der Verarbeitung von Modellen ist der *Workspace*, welcher beliebig viele *Projekte* enthält. Eine dem Dateisystem nahe Darstellung des Workspace bietet *Navigator* im linken Bereich der Benutzeroberfläche.

Aus dem Navigator können XMI- und Simulink-Modelldateien geöffnet werden. Diese werden im Editorbereich angezeigt und können dort bearbeitet werden. Simulink-Modelle werden in ihrer Darstellung auf Basis des UML-Metamodells geöffnet.

Die auf dem Reflektionsmechanismus von JMI (siehe Abschnitt 2.2.4) basierende Darstellung ist nah am Metamodell orientiert und visualisiert alle im Modell enthaltenen Informationen. Der linke Teil des Editors zeigt den durch die Kompositionen im Metamodell aufgespannten Modellbaum. Bei Auswahl eines Elements zeigt die rechte Seite Detailinformationen zu dem Modellierungsartefakt.

Außer dem Bearbeiten von Modellen in Dateien ist der Zugriff auf eine von Aquintos.GS verwaltete Datenbank möglich. Das Modell ist dabei nicht direkt editierbar sondern dient als *Repository* zur Versionsverwaltung.

Neben einer umfangreichen Suchfunktionalität können bestimmte Modellartefakte über Filter ein- und ausgeblendet werden. Weiterhin können weitere Modelle in Unterzweige des Gesamtmodells importiert werden. Der Editor integriert sich in das bei Editoren verbreitete Paradigma, welches Öffnen, Speichern, Verwerfen der Änderungen und Speichern unter anderem Namen oder in einem anderen Format erlaubt.

Die Zusammenarbeit zwischen Aquintos.GS und Eclipse ist wie in Abbildung

[1] Eclipse ist ein auf Plug-Ins basierendes Open-Source Framework zur Erstellung von Software, insbesondere von Anwendungen, welche als Entwicklungsumgebungen agieren [Eclipse Foundation 2008a]. Durch das Plug-In-Konzept können unterschiedlichste Werkzeuge integriert wird. Für ausführliche Informationen über das Bedienkonzept und die Programmierung von Eclipse sei auf [Gamma u. Beck 2003] verwiesen.

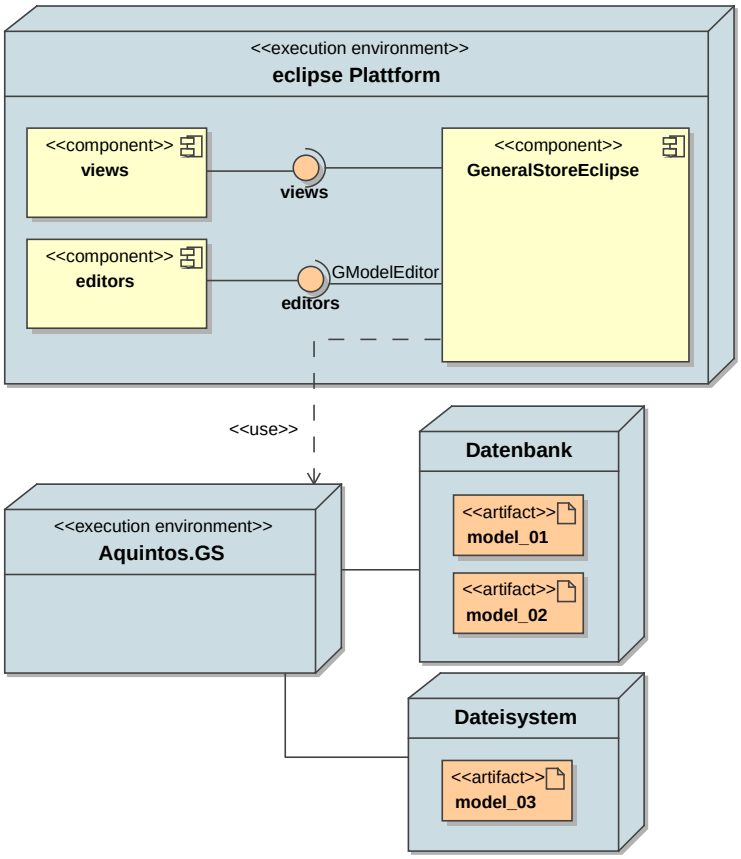

Abbildung 5.2: Verteilungsdiagramm: Die Komponente GeneralStoreEclipse verknüpft Aquintos.GS mit der Eclipse Plattform und integriert die Benutzerführung

5.2 gezeigt durch die als Plug-In realisierte Komponente *GeneralStoreEclipse* gegeben. *GeneralStoreEclipse* bietet als Teil der Eclipse Plattform Editoren und Sichten an, greift jedoch auf die Funktionalität zur Modellverwaltung von Aquintos.GS zurück.

Im linken unteren Bereich stellt der Modelleditor eine Übersicht über das gerade angewählte Modell in einer *Outline* dar, welche alle in ihm enthaltenen, auf dem *Diagram Interchange* Format basierenden Diagramme (siehe Abschnitt 2.2.6) und alle enthaltenen Werkzeugketten (siehe Abschnitt 5.1.2) auflistet.

Zur Codeerzeugung können alle an Aquintos.GS angebundenen Codegeneratoren angesprochen werden. Neben den in Abschnitt 3.1.5 beschriebenen kommerziellen Codegeneratoren und den existierenden Generatoren für Java und C++ aus UML-Modellen, kann jeder Modellteil über eine vorlagengesteuer-

Listing 5.1: Einfaches Skript zur Codeerzeugung

```
from java.lang import System

model = SXMIExchange.read("../tests/controller_a.xmi")

SCodeGen.generate("templates/genvhdl/genvhdl.vm",
        "../GenerateVHDL/tests/out.vhdl",
        "continousDomain::matlabSimulinkSubsystem::controller_a",
        "com.itiv.generalStore.xstruc.statechart.XState",
        model)

print "Fertig."
```

te Modell-zu-Text-Transformation in Quelltext oder eine Konfigurationsdatei überführt werden.

Zahlreiche Abläufe können durch Skripte automatisiert werden. Die Skripte sind in der Programmiersprache Python verfasst. Während der Ausführung können in den Interpreter durch die Plug-Ins von ModelScope Objekte für unterschiedliche Aufgaben eingeblendet werden, welche dem Skript zur Verfügung stehen. Skripte können auf das gesamte Modell zugreifen und dieses manipulieren.

Das in Listing 5.1 gezeigte Skript liest zunächst ein Modell aus einer XMI-Datei ein und legt eine Referenz in die Variable *model*. Ein weiterer Befehl erzeugt aus einem Teil des Modells mit Hilfe der Vorlage *genvhdl.vm* VHDL-Code zur Hardwarebeschreibung und legt diesen in der Datei *out.vhdl* ab..

5.1.2 Projektverwaltung

Die im vorherigen Abschnitt beschriebenen Werkzeuge erlauben bereits eine vollautomatische Generierung und Ausführung eines weitestgehend modellierten Systems. Die Automatisierung erfolgt dabei über Skripte. Diese zeigen aber eine unzureichende Flexibilität, schlechte Wartbarkeit und eine nur unzureichende Verknüpfung mit dem funktionalen Gesamtmodell. Ändert sich beispielsweise der Name eines Modellelementes oder seine ID ist es nicht mehr von dem automatisierenden Skript ansprechbar. Dieses muss manuell nachgebessert werden.

Deshalb wurden im Rahmen dieser Arbeit Erfahrungen in den Bereichen Funktionsmodellierung und Entwurfsprozess / Werkzeugkette systematisiert. Dazu wird die verwendete Werkzeugkette modellierbar gemacht und zusammen mit einem Modell der Zielplattform in das Gesamtmodell integriert.

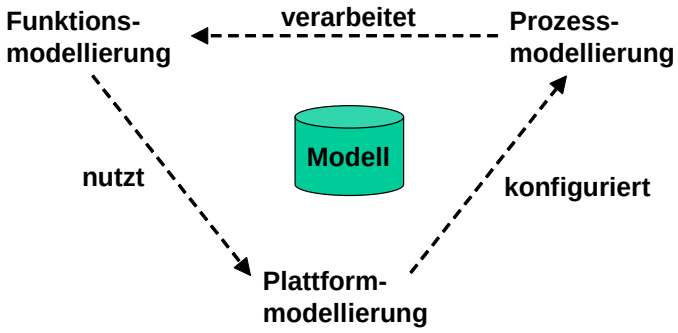

Abbildung 5.3: Schematische Darstellung der Zusammenhänge in einem Gesamtmodell, welches Funktion, Prozess und Plattform integriert

Abbildung 5.3 zeigt die Zusammenhänge zwischen den Teilmodellen. Das Funktionsmodell nutzt dabei die Services der Ausführungseinheiten, welche in einer Plattformmodellierung spezifiziert sind. Die Plattformmodellierung umfasst neben Anzahl und Parametern der Ausführungseinheiten Informationen über mögliche Datenflüsse und Schnittstellen zum Debugging.

Das Prozessmodell ordnet einzelne Arbeitschritte, wie Kommunikation mit Modellierungswerkzeugen, Codegeneratoren, Compilierung, Synthetisierung, Konfiguration von Betriebssystemen, Erzeugung von Logdateien, Anreicherung um Debug-Schnittstellen etc. zu einer gemeinsamen Werkzeugkette an. Das Prozessmodell kann sich auf Teile des Funktionsmodells beziehen und dieses weiter verarbeiten.

Das Prozessmodell und seine Ausführung werden wieder von der zur Verfügung stehenden Ausführungsplattform beeinflusst und konfiguriert.

Zentral für den Anspruch einer durchgängigen Entwurfsumgebung für ausführbare Funktionsmodelle sind insbesondere das Prozessmodell und seine Integration in das Gesamtmodell.

5.1.2.1 Konzept

Die in *ModelScope* realisierte Projektverwaltung basiert auf einem datenflussgetriebenen Prozessmodell. Das Modell besteht aus einzelnen **Aktionen**, welche bei Erfülltsein aller Vorbedingungen ausgeführt werden. Das Ergebnis einer Aktion steht an ihrem **Ausgang** zur Verfügung und kann an weitere **Eingänge** weiter gegeben werden. Die Verknüpfung der einzelnen Prozessschritte erfolgt nach einem *Fan-Out*, es darf also jeder Ausgang an beliebig viele Eingänge weitergegeben werden. Umgekehrt muss jedem Eingang genau ein Ausgang

zugeordnet sein. Neben Ein- und Ausgängen besitzt eine Aktion **Parameter**, welche ihr Verhalten konfigurieren.

Aktionen, Parameter und ihre Verknüpfung untereinander bilden gemeinsam eine **Werkzeugkette**. Eine einzelne Aktion ist dabei immer eine Instanz eines **Aktionstyps**. Die möglichen Aktionstypen, ihre typisierten Parameter, Ein- und Ausgänge sind in einer Bibliothek definiert. Diese Bibliothek beschreibt auch das Verhalten der einzelnen Aktionen. Der Ansatz folgt durch den Typ-Instanz-Zusammenhang der objektorientierten Grundidee.

Die Bibliothek basiert auf dem in der Java-Entwicklung verbreiteten Werkzeug Ant. Ant verwendet eine *Build*-Datei, die auf dem XML-Format basiert. In der *Build*-Datei werden Ziele (*targets*) beschrieben. Jedes Ziel besitzt einen Namen und kann weitere Ziele aufrufen. Die einzelnen Ziele einer *Ant*-Datei bestehen ihrerseits aus einzelnen Aufgaben (*task*s), die ausgeführt werden, um das Ziel zu erreichen. Ant kann über die Kommandozeile angesprochen werden, ist jedoch auch in zahlreiche Entwicklungsumgebungen integriert. Für den genauen Aufbau und die Verwendung von *Ant* und die bereits integrierten *Tasks* sei auf [Edlich 2002] verwiesen.

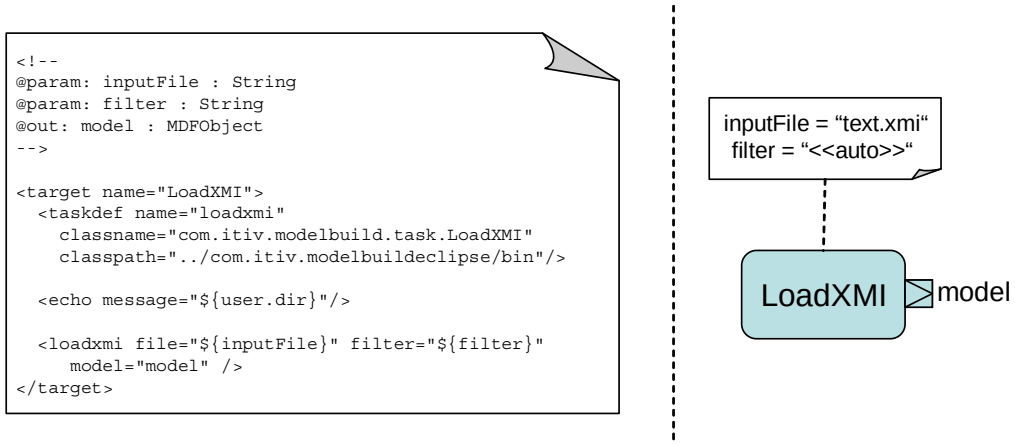

Abbildung 5.4: Ausschnitt aus der Definitionsdatei für mögliche Prozessschritte (links) und mögliche Instanziierung (rechts)

Einen Ausschnitt aus einer Ant-basierten Bibliothek zur Definition von Aktionstypen zeigt die linke Seite in Abbildung 5.4. Das *Target LoadXMI* bezeichnet eine Aktion zum Laden eines im XMI-Format (siehe Abschnitt 2.2.5) gespeicherten Modells. Im Kommentarbereich befindet sich die Signatur des Aktionstyps in Form einer Liste. Die Einträge haben immer folgenden Aufbau:

$$@\{\text{Art}\} : \{\text{Name}\} : \{\text{Datentyp}\}$$

„Art“ bezeichnet den Eintrag als Eingang (*in*), Ausgang (*out*) oder Parameter (*param*). „Name“ beschreibt den Parameternamen und „Datentyp“ einen Java-Datentyp. Bei Letzterem sind neben einfachen Datentypen auch beliebige Java-Klassen erlaubt, so dass zwischen Aktionen komplexe Objekte (beispielsweise Modelle) weitergegeben werden können.

Innerhalb des *targets* befinden sich die *Tasks*, welche die Aktion realisieren. Die Syntax und die verfügbaren Schritte entsprechen den in *Ant* vorgegebenen Möglichkeiten. *Ant* bietet jedoch die Erweiterung um eigene, in Java implementierte *Tasks* an. Im Beispiel in Abbildung 5.4 wird ein *Task loadxmi* definiert, an welchen aus der Ant-Datei weiterdelegiert wird.

Die im Kommentarbereich angegebenen Namen für Eingäng, Ausgänge und Parameter können innerhalb des *Targets* als *Ant-Property*, also als Variablen verwendet werden.

Wird der Aktionstyp *LoadXMI* realisiert, entsteht eine Aktion, welche konkrete Ein- und Ausgänge, sowie Parameterausprägungen besitzt. Abbildung 5.4 rechts zeigt eine solche Aktion in konkreter Syntax. Diese Syntax wurde an die aus Abschnitt 3.2 bekannte konkrete Syntax für Aktionen im *UML Action Metamodell* angelehnt.

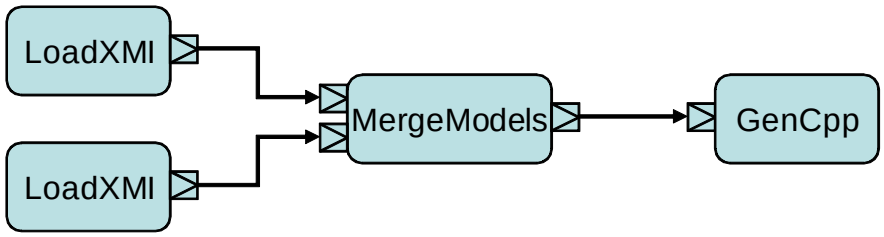

Abbildung 5.5: Einfache Kette von Abläufen

Werden mehrere Aktionen wie in Abbildung 5.5 gezeigt verkettet, wird die Abarbeitung durch Bezeichnung der letzten abzuarbeitenden Aktion gestartet. Ist diese Aktion über ihre Eingänge von weiteren Aktionen abhängig, müssen diese zuvor ihr Ergebnis am Ausgang erzeugen. Ist die Werkzeugkette zyklenfrei, können alle Vorgänger der Zielaktion in eine Ausführungsreihenfolge gebracht werden.

Im obigen Beispiel werden bei dem Ziel *GenCpp* zunächst beide Aktionen *LoadXMI* aufgerufen. Deren Ergebnis fließt in *MergeModels* ein, welches an seinem Ausgang schließlich das Modell für die durch *GenCpp* realisierte Codeerzeugung bereitstellt.

5.1.2.2 Einordnung in ein UML-basiertes Gesamtmodell

Prozesselement	Abbildung auf UML Actions
Werkzeugkette	`Procedure / GroupAction`
Aktion	`CallOperationAction`
Eingang	`InputPin`
Ausgang	`OutputPin`
Verknüpfung	`DataFlow`
Parameter	`LiteralValueAction an InputPin`
Bibliothek	`Class`
Aktionstyp	`Operation`, mögliche Parameter zur Signatur der Operation

Abbildung 5.6: Elemente des Prozessmodells und ihre Abbildung in das UML Metamodell

Die syntaktische und semantische Nähe der Werkzeugkette zu den UML Actions ermöglicht eine einfache Abbildung auf diese und damit eine Integration in das restliche UML-basierte Gesamtmodell. Abbildung 5.6 zeigt die wichtigsten Elemente des Prozessmodells und ihre Entsprechung im UML Action Metamodell. Für eine genauere Beschreibung der Modellierungsartefakte für UML Actions sei auf Abschnitt 3.2 verwiesen.

Eine Bibliothek wird in eine Klasse überführt, welche für jeden Aktionstyp eine Operation aufweist. Die möglichen Eingänge, Ausgänge und Parameter mitsamt ihrer Datentypen und Namen sind durch die Signatur, also die Parameter der Operation gegeben. Nicht überführt wird die Verhaltensbeschreibung einzelner Aktionstypen. Diese ist nur in der Ant-basierten Definitionsdatei enthalten.

Eine Werkzeugkette wird durch eine Prozedur dargestellt, welche über eine *GroupAction* die einzelnen Aktionen in Form von *CallOperationActions* enthält. Diese verweisen auf die in der Bibliothek definierte Operation. Die Eingänge und Ausgänge der Aktionen finden ihre Entsprechung in den *InputPins* und *OutputPins* des UML Metamodells. Verknüpft werden können die Aktionen über Datenflüsse (*DataFlows*). Parameter der Aktionen werden über *LiteralValueActions* bereitgestellt, welche über Datenflüsse an die Aktionen angeschlossen sind.

Abbildung 5.7 zeigt als komplexeres Beispiel die Werkzeugkette aus Abbildung 5.5 in abstrakter Syntax auf Basis des UML Metamodells und im unteren Bereich in konkreter Syntax das Klassensymbol der zugehörigen Bibliothek.

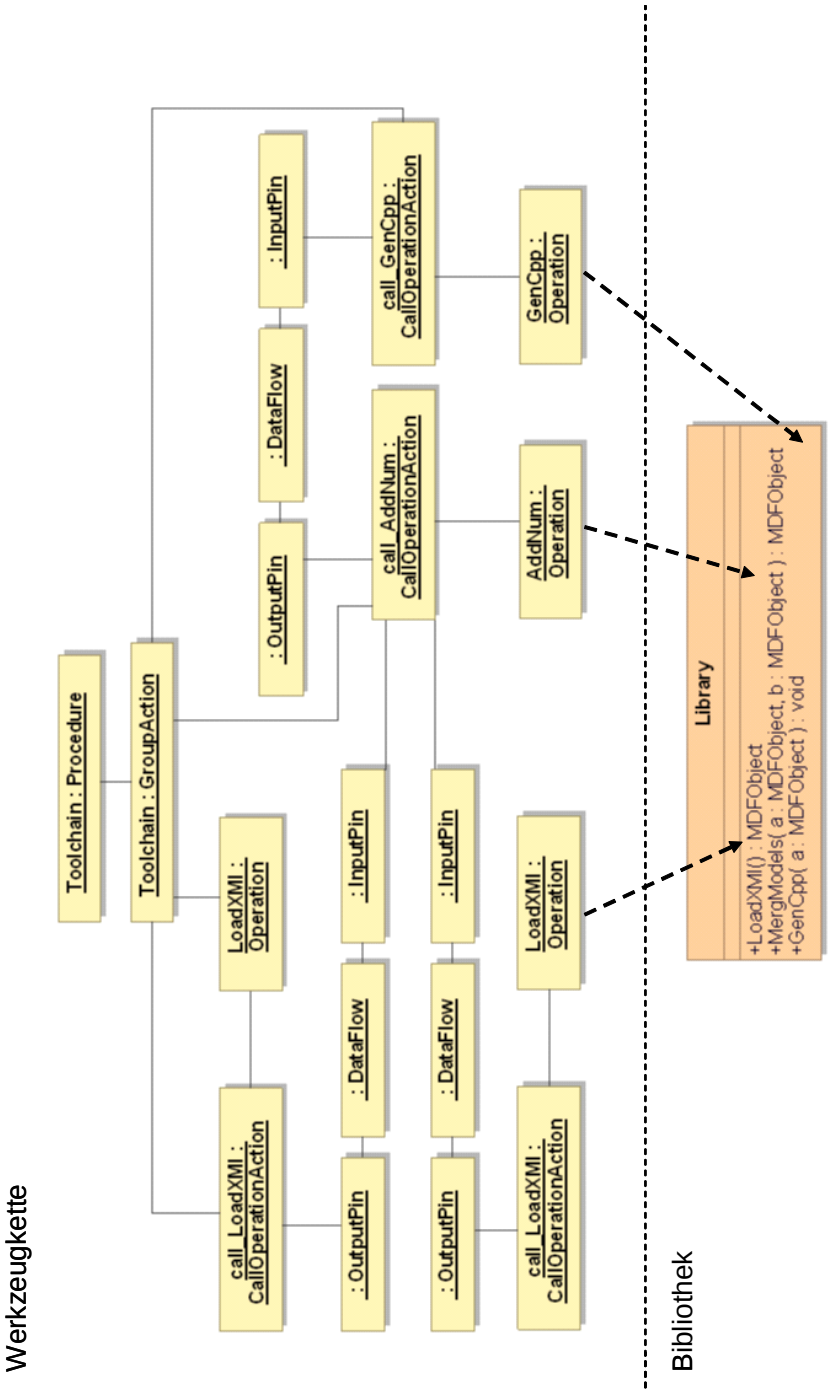

Abbildung 5.7: Werkzeugkette in abstrakter Syntax auf Basis des UML Action Metamodells (oben) und im selben Modell integrierte Signatur der Bibliothek (unten)

Die gestrichelten Pfeile verdeutlichen die Verknüpfung von Werkzeugkette und Bibliothek durch die in beiden Teilmodellen enthaltenen Operationen.

5.1.2.3 Integration in ModelScope

Die Fähigkeit zum Import, Export und zur Ausführung von Werkzeugketten ist in *ModelScope* integriert.

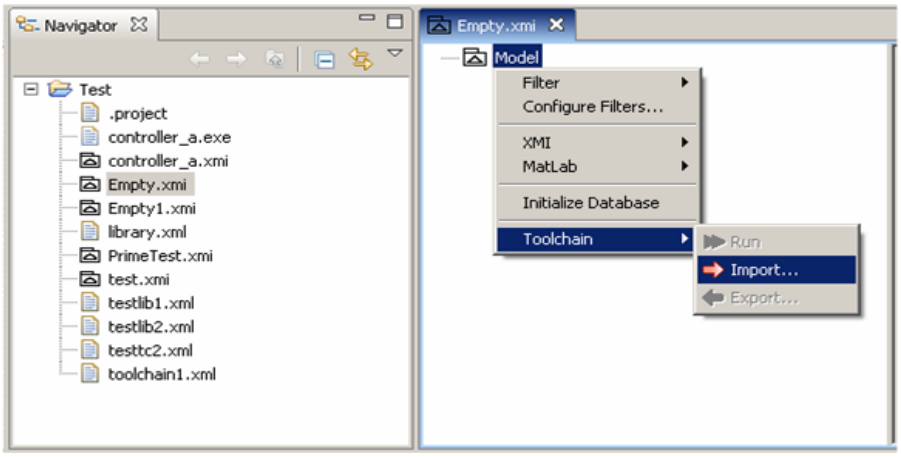

Abbildung 5.8: Kontextmenü vor Import in leeres Modell

Abbildung 5.8 zeigt ein leeres Modell und das Kontextmenü des Wurzelknotens. Da das Modell keine Werkzeugketten enthält wird nur der Eintrag zum Import angeboten. Ein Import ist jedoch nicht nur in die Modellwurzel möglich, sondern in jedes beliebige Paket des Gesamtmodells.

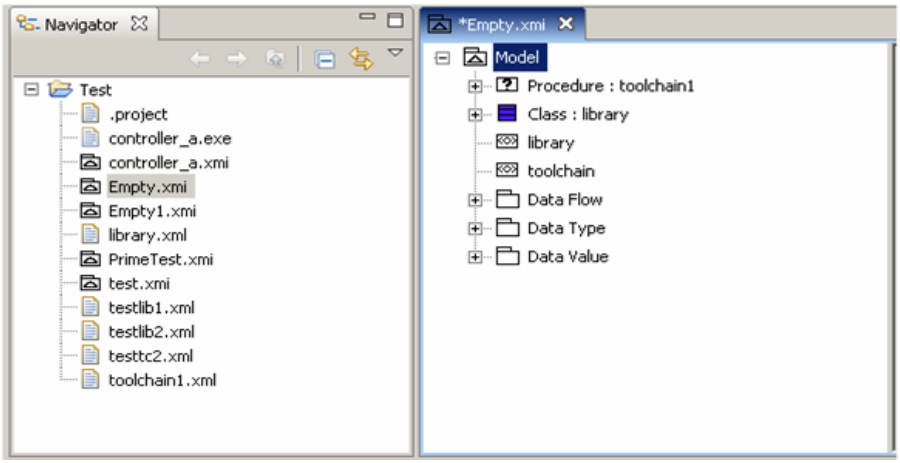

Abbildung 5.9: Baumansicht nach Import

Den Zustand des Modells nach Import einer Werkzeugkette verdeutlicht Abbildung 5.9. Neben der Klasse, welche die Bibliothek repräsentiert, ist die *Procedure*, welche die eigentliche Werkzeugkette enthält zu erkennen. Die weiteren Pakete umfassen Datentypen und -flüsse, sowie zwei Stereotypen zur Kennzeichnung von Werkzeugketten und Bibliotheken. Bibliotheken, Datentypen und Stereotypen werden bei Nutzung mehrerer Werkzeugketten in einem Modell so weit wir möglich weiterverwendet.

Bei Rechtsklick auf die Prozedur ist im Untermenü *Toolchain* der Unterpunkt *Run* auswählbar. Mit diesem kann die Werkzeugkette ausgeführt werden. Statusinformationen erhält der Nutzer in einer Fortschrittsanzeige und über die Ausgabekonsole von Eclipse.

5.2 Laufzeitmodelle und –ebenen

Zum Verständnis der in *ModelScope* realisierten flexiblen Architektur zum Debugging ausführbarer Modelle ist der Begriff von Laufzeitmodellen und –ebenen von großer Bedeutung. Zunächst wird das grundlegende Konzept vorgestellt und darauf aufbauend beschrieben, wie statisches und dynamisches Modell durch Erweiterung des UML Metamodells verknüpft werden können [Graf u. Müller-Glaser 2006b, 2007].

5.2.1 Konzept

Das Debuggen eines Systems, welches mit modellbasierten Methoden entworfen wurde, kann als Umkehrung einer Vielzahl von Transformationsschritten verstanden werden, die vom Modell bis zum ausführbaren Code durchlaufen werden.

Abbildung 5.10 verdeutlicht diesen Sachverhalt. Das als Ergebnis eines Entwurfs entstandene Modell durchläuft einen oder mehrere Transformationsschritte bis hin zum Quelltext, welcher ebenfalls als in Textform dargestelltes Modell (*Modell'*) verstanden werden kann. Die in der Abbildung gezeigte Überführung in einem einzelnen Schritt ist nicht zwingend. So kann beispielsweise im Sinne eines MDA-Ansatzes (siehe Abschnitt 3.1.2) das Modell in mehreren Transformationsschritten verfeinert, um Plattforminformationen angereichert und erst im letzten Schritt in Quelltext umgewandelt werden. Auch der Quelltext kann Transformationen durchlaufen, beispielsweise durch Werkzeuge zur Optimierung oder durch Instrumentierung zum Debugging oder zur Analyse des Laufzeitverhaltens.

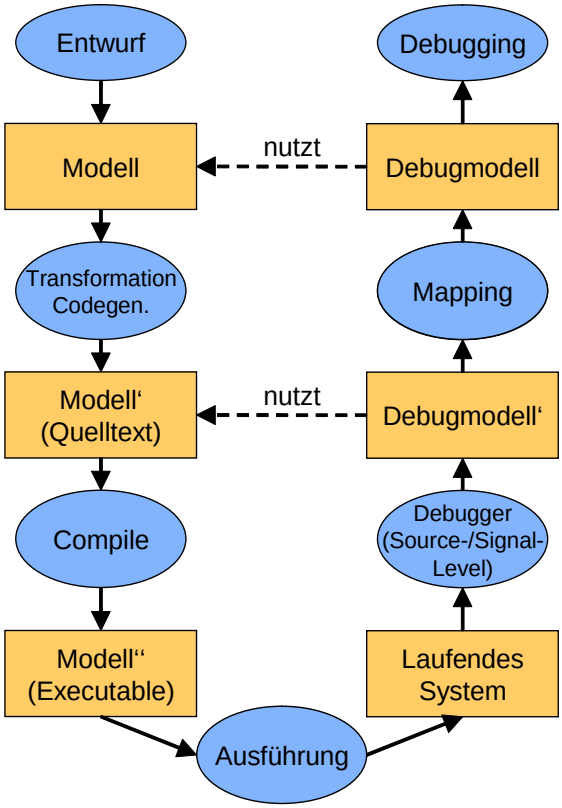

Abbildung 5.10: Abbildungen zwischen Modellebenen

In einem weiteren Schritt wird der Quelltext durch einen *Compiler* in ein weiteres Modell (*Modell”)*, das sogenannte *Executable* transformiert, welches in einer Form vorliegt, welche direkt auf einer Zielplattform ausgeführt werden kann.

Die grundlegende Aufgabe eines Debuggers ist die Steuerung des Laufzeitverhaltens des ausgeführten Modells und die Extraktion von Zustandsinformationen aus dem laufenden System. Der Begriff Zustandsinformation wird im Folgenden synonym zu Laufzeitinformation, Laufzeitmodell und Debugmodell verwendet.

Die gewonnene Information bezieht sich jedoch zunächst auf Artefakte auf Ebene des Modells *Modell”*. Ist das *Modell”* der Binärcode eines Mikroprozessors, sind Laufzeitinformationen auf dieser Ebene beispielsweise durch den Wert des Programmzählers, Registerwerte und den Speicherinhalt gegeben.

Aufgabe eines Debuggers ist die Transformation der Laufzeitinformationen so, dass durch sie Modellartefakte auf höheren Ebenen referenziert werden. Dabei wird für jeden Transformationsschritt zur Entwurfszeit hin zum spezifischen

Modell, zur Laufzeit ein Transformationsschritt zu höheren Abstraktionsebenen durchgeführt.

Glücklicherweise stehen für den ersten Schritt, die Transformation vom direkt ausführbaren *Modell"* hin zum Quelltext (*Modell'*) bereits zahlreiche Werkzeuge zur Verfügung. Alle Quelltextdebugger leisten mit den in Abschnitt 4.3.2 beschriebenen Fähigkeiten genau diese Umkehrung. Laufzeitinformation, welche dem Nutzer des Debuggers zur Verfügung steht, beinhaltet Variablenwerte, Parameterwerte, Stand des Programmzählers im Quelltext der Hochsprache und den Aufrufstapel. Somit beziehen sich Debugger auf Quelltextebene immer auf Artefakte des Modells *Modell'*.

Ein Debugger für Modelle führt diese Umkehrung weiter und abstrahiert Laufzeitinformationen so, dass sie letztlich Artefakte im ursprünglichen Entwurfsmodell referenzieren.

Die beschriebene Umkehrung darf jedoch nicht mit *Reverse-Engineering* verwechselt werden. Beim *Reverse-Engineering* wird versucht, in Abbildung 5.10 im linken Zweig Transformationsschritte umzukehren, beispielsweise das *Executable* in Quelltext umzuwandeln oder aus einem Quelltext ein grafisches Modell zu gewinnen. Beim Debugging werden zusätzliche dynamische Informationen gewonnen, welche sich jedoch auf unterschiedliche Ebenen der statischen Modelle beziehen.

Zur Transformation von Laufzeitinformationen von einer Quellebene auf eine darüberliegende Zielebene sind allgemein folgende Datenquellen verfügbar:

- Dynamische Informationen auf Quellebene
- Das statische Modell der Zielebene
- Das statische Modell der Quellebene
- Informationen aus der Transformation

Der folgende Abschnitt befasst sich jedoch zunächst mit der Frage, wie das Modell überhaupt sinnvoll um Laufzeitinformationen angereichert werden kann. Die tatsächlichen Transformationen wurden exemplarisch auf wichtige Modellierungssichten angewendet. Diese werden in Kapitel 7 näher beleuchtet.

5.2.2 Erweiterung des Metamodells um Laufzeitinformationen

Der Debugger reichert das Modell zur Laufzeit um Debuginformationen an, welche sich im Verlauf der Zeit ändern können. Zur Laufzeit unverändert blei-

ben alle Modellierungsartefakte, die aus dem Entwurf des Modells stammen. Das Gesamtmodell unterteilt sich während der Laufzeit in zwei Bereiche:

- **statisches Modell**: Das statische Modell umfasst das Modell im linken Ast in Abbildung 5.10. Auch wenn das Modell dynamische Aspekte des Systems beschreibt, bleibt es selbst während der Ausführung des Modells unverändert.
- **dynamisches Modell**: Das dynamische Modell umfasst alle Elemente des Gesamtmodells, die sich während der Laufzeit des Systems verändern.

Statisches und dynamisches Modell bilden zur Laufzeit ein Gesamtmodell. Dabei können Elemente des dynamischen Modells auf Elemente des statischen Modells verweisen. Da das dynamische Modell zur Entwurfszeit nicht existiert, ist eine umgekehrte Referenzierung jedoch nicht möglich.

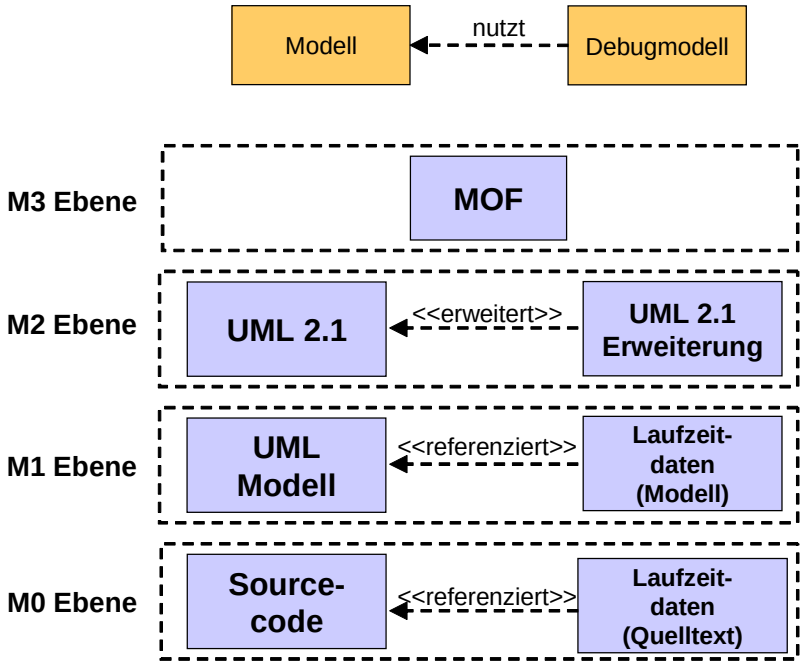

Abbildung 5.11: OMG Metabenen und Laufzeitinformation

Abbildung 5.11 spiegelt das beschriebene Konzept der Verknüpfung von dynamischem und statischem Modell am Ebenenmodell der Metamodellierung, welches von der OMG definiert und in Abschnitt 2.2.1 beschrieben wurde.

Auf Ebene M0 beziehen sich die Laufzeitdaten auf den Quelltext. Die Belegung einer Speicherstelle ist mit einer Variablen verknüpft, die mit einem

symbolischen Namen versehen ist. Nach der Abbildung auf das Modell in Ebene M1 sind die Laufzeitinformationen mit dem UML-Modell verbunden. Um diese Informationen während einer Debuggingsitzung vorhalten zu können, ist eine Beschreibung durch ein Metamodell notwendig. Hierzu wird das UML-Metamodell in Ebene M2 erweitert. Diese Beschreibung hält sich an das Meta-Metamodell MOF (Ebene M4). Die Erweiterung des UML Metamodells von Ebene M2 erlaubt es, aus dem, auf Ebene M0, laufenden Programm die für das Debugging benötigten Daten zu extrahieren und zu verwalten.

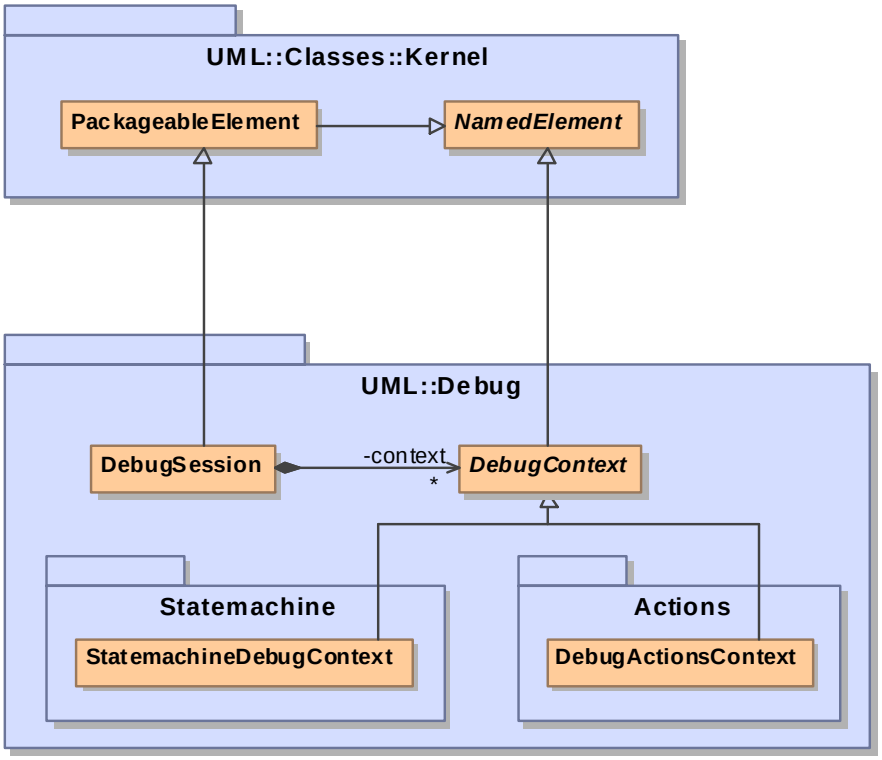

Abbildung 5.12: Erweiterung des UML Metamodells um das Paket UML::Debug

Die grundlegende Erweiterung des UML Metamodells für Laufzeitinformationen ist in Abbildung 5.12 dargestellt. Das Wurzelelement für die Speicherung der Debugdaten ist die Metaklasse *DebugSession*. Diese erbt von der aus dem *Kernel*-Paket des UML Metamodells importierten Klasse *PackageableElement*, so dass sie Teil jedes Pakets sein kann.

Die *DebugSession* enthält eine beliebige Anzahl von Artefakten des Typs *DebugContext*. Jeder *DebugContext* bezeichnet einen in sich geschlossenen autonomen Satz von Zustandsinformationen. Dieser bündelt beispielsweise alle Informationen zu einer in das Gesamtmodell eingebetteten Zustandsmaschine oder

die Debuggingdaten, welche bei der Ausführung einer auf dem UML Actions Metamodell basierenden Verhaltensbeschreibung gewonnen werden können.

In der in Abschnitt 5.3 dargestellten Architektur besitzt jeder *Viewpoint* genau einen *DebugContext*, der durch ihn verwaltet wird. Die einzelnen *Viewpoints* und ihre Erweiterungen des UML Metamodells werden in Kapitel 7 behandelt. Bei der Vorstellung der Metamodellerweiterungen wird dabei die Trennung zwischen dynamischem und statischem Modellteil markiert. In der Realisierung befinden sich alle rein dynamischen Artefakttypen im Paket *UML::Debug* des Metamodells oder einem seiner Unterpakete. Vorhandene Metamodellklassen der UML können jedoch weiter verwendet werden. Dies geschieht beispielsweise bei der Gewinnung von Objektdiagrammen aus Klassenmodellen zur Laufzeit in Abschnitt 7.3.

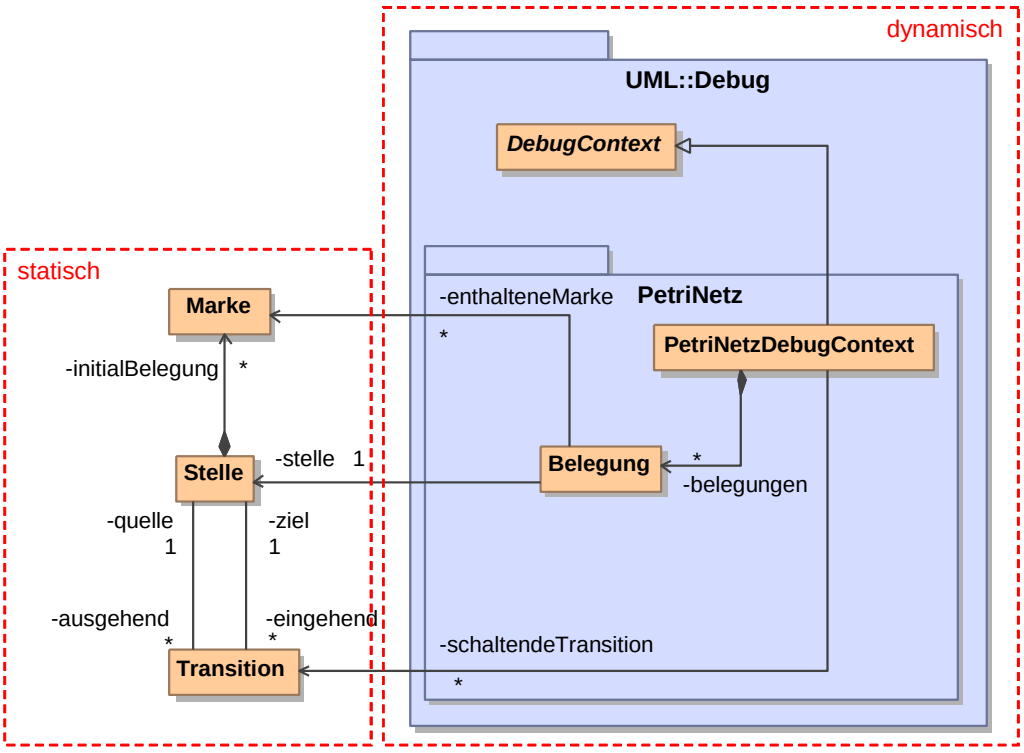

Abbildung 5.13: Erweiterung eines Metamodells für Petri-Netze um einen dynamischen Modellteil

Abbildung 5.13 verdeutlicht die Erweiterung des Metamodells um einen dynamischen Modellteil anhand eines einfachen nicht im UML Metamodell enthaltenen Beispiels. Das im linken Bereich gezeigte statische Modell beschreibt ein einfaches Petrinetz, dessen Transitionen immer nur jeweils eine Stelle im Vor- und im Nachbereich besitzen. Weiterhin ist die Kapazität der Stellen unbegrenzt. Die Anzahl der Marken im Petri-Netz ist also immer konstant.

Im Metamodell besitzt jedes Modellartefakt des Typs *Transition* genau je ein Modellelement vom Typ *Stelle* als Quelle und Ziel. Eine *Stelle* kann beliebig viele ein- und ausgehende Elemente vom Typ *Transition* aufweisen. Im Petrinetz existiert eine feste zur Entwurfszeit modellierte Anzahl an *Marken*. Jede Marke ist über eine Komposition als Anfangsbelegung einer Stelle zugeordnet. Dieser statische Modellteil bleibt zur Laufzeit unverändert.

Wird aus einem solchen Petrinetz-Modell Code generiert, interessieren zur Laufzeit folgende Information über den Systemzustand:

- Wie ist die Belegung des Petrinetzes? In welcher Stelle befinden sich wie viele Marken?
- Welche Transition ist schaltfähig? Für welche Transition sind alle Vorbedingungen erfüllt?

Im Paket *UML::Debug::PetriNetz* sind diese Informationen gebündelt. Jedes Petrinetz besitzt genau einen *PetriNetzDebugContext*, der von *DebugContext* erbt und damit Teil einer *DebugSession* sein kann.

Der *PetriNetzDebugContext* besitzt Modellobjekte des Typs *Belegung*. Jedes *Belegung*-Modellelement referenziert eine Stelle und die aktuell in ihr enthaltenen *Marken*. Weiterhin besitzt der *PetriNetzDebugContext* eine gerichtete Assoziation auf beliebig viele *Transitionen* im statischen Modellteil und kann so alle schaltfähigen Transitionen referenzieren.

Damit entsteht ein vollständiges eng gekoppeltes Modell für Laufzeitinformationen, durch welches alle interessierenden Aspekte über den Systemzustand auf Modellebene dargestellt werden können.

Die Gewinnung des dynamischen Modells zur Laufzeit und die Steuerung des Systems aus Modellsicht ist Aufgabe des Debuggers. Diese Aufgabe wird im folgenden Abschnitt in eine Gesamtarchitektur eingebettet und näher beschrieben.

5.3 Debugger für Modelle in ModelScope

5.3.1 Grundlegende Architektur

Abbildung 5.14 zeigt die Softwarearchitektur eines universellen Debuggers für Modelle [Graf u. a. 2004; Graf u. Müller-Glaser 2006a]. Sie ist das Resultat der in Abschnitt 4.5.3 vorgestellten Anforderungen an eine flexible Debugger-Architektur und dem in Abschnitt 5.2 beschriebenen Ansatz der Integration

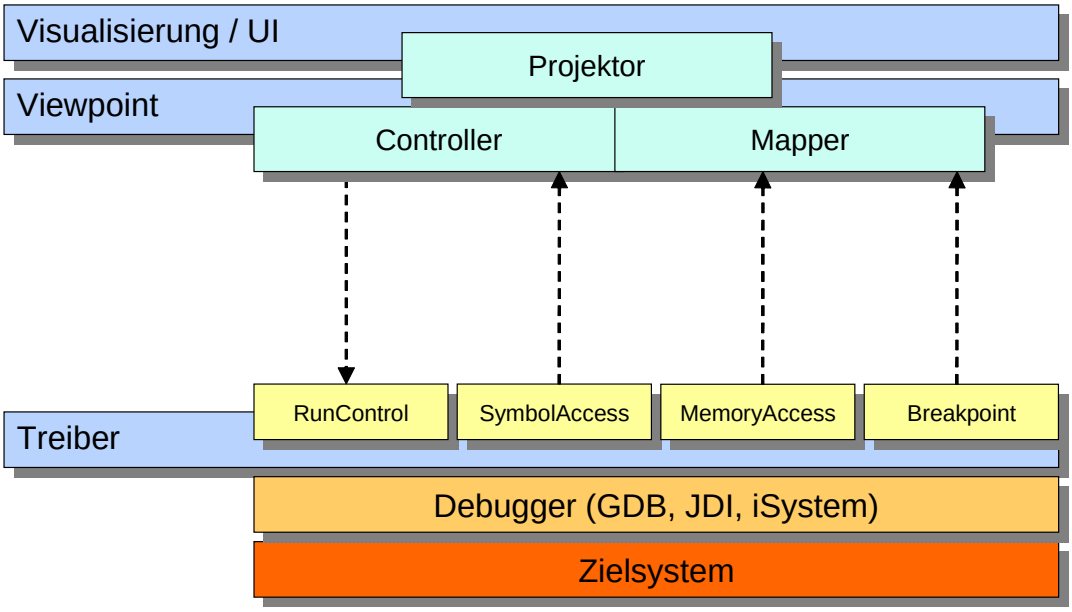

Abbildung 5.14: Schichtenarchitektur des Debugging-Frameworks

von dynamischem und statischem Modell. Sie ist in Schichten aufgebaut, welche ausgehend von der Zielarchitektur, dem Quelltextdebugger und der Programmiersprache hin zu einer Visualisierung auf Modellebene abstrahieren. In diesem Abschnitt wird zunächst auf den grundlegenden Aufbau eingegangen. Realisierungsdetails behandeln die folgenden Abschnitte 5.3.2 bis 5.3.5.

Ausgangspunkt ist hierbei die modellierte Software, welche auf einer eingebetteten Zielplattform ausgeführt wird. Verschiedene Zielarchitekturen beinhalten oder unterstützen unterschiedliche Debugger auf Quelltextebene, welche die Gewinnung von Laufzeitinformationen über proprietäre Schnittstellen erlauben. Beispiele sind das Java Debug Interface (JDI), welches in den Klassenpfad des Debugee eingeblendet wird, die textbasierte Schnittstelle des GDB oder COM-basierter Zugriff auf die Debug-Hardware eingebetteter Prozessoren.

Die **Treiber**ebene vereinheitlicht die Quelltextdebugger so weit wie möglich. Da Debugger häufig nicht dieselbe Auswahl an Fähigkeiten und Informationsquellen aufweisen, wurden Schnittstellen definiert, welche Sätze von Fähigkeiten gruppieren. Treiber implementieren im Allgemeinen nur eine Auswahl der Schnittstellen und beeinflussen damit die Fähigkeiten, welche das Gesamtsystem auf höheren Ebenen bieten kann. Geläufige Schnittstellen beinhalten Ablaufkontrolle (Starten, Stoppen, Reset, Einzelschritt), Zugriff auf Symboldaten (Speicheradressen von Operationen) und Auswertung von Ausdrücken (Lesen von Variablenwerten). Hardware-unterstützte Debugger bieten häufig

zusätzlich eine Schnittstelle zur Gewinnung und zum Auslesen von Echtzeit-Trace-Daten aus dem Zielsystem an.

Blickt man von oben auf die Architektur, so bietet sie dem Anwender des Debuggers unterschiedliche ***Viewpoints***. Ein *Viewpoint* definiert dabei einen spezifischen Blick auf das System und vermittelt zwischen Zielsystem und der Visualisierungsebene, welche über ihr liegt. Eine typische Debug-Session für heterogen modellierte Software nutzt dabei gleichzeitig verschiedene Viewpoints. Die Objektkonfiguration des mit der UML objektorientiert modellierten Systemteils, wird mit einem UML Objektdiagramm visualisiert und jedes eingebettete Statechart zeigt das Verhalten von Komponenten, Klassen oder den Zustand von Schnittstellenprotokollen. Der komplette Satz an Viewpoints erlaubt Einblick in alle Aspekte des laufenden Modells.

Die Visualisierungsschicht basiert auf der Open-Source Entwicklungsplattform Eclipse und dem dazugehörenden Diagramm-Framework GEF[2]. Das Diagrammmodell basiert dabei auf dem Diagram Interchange (DI) Metamodell, welches seit der UML 2.0 Teil des Standards ist.

Zwischen Viewpoint und Treiber befinden sich **Mapper** und **Controller**, welche die Modellebenen überbrücken. Der Viewpoint nutzt den Controller um das System aus dem Blickwinkel des Modells zu steuern. Ein Ausführungsschritt in einem Statechart resultiert beispielsweise in einer komplexen Abfolge von Befehlen auf Quelltextebene. Der Mapper aktualisiert das Laufzeitinformationen enthaltende Modell mit Informationen aus den Debugger-Treibern auf Quelltextebene. Die Visualisierung der Laufzeitdaten erfolgt durch **Projektoren**.

5.3.2 Einbettung in die Basisplattform

Die Basisfunktionalität zum Debugging von Modellen wird in der *ModelScope*-Plattform durch das Plug-In *modelDebugger.core* realisiert. Abbildung 5.15 zeigt einzelne Plug-Ins der Eclipse-Plattform als Komponenten in einem Komponentendiagramm. Der Debugkern ist die zentrale Komponente der Architektur. Der Kern besitzt zwei Erweiterungspunkte in der Eclipseplattform, über den Realisierungen für verschiedene Zielsystem und Ansichten durch weitere Plug-Ins während der Laufzeit von Eclipse hinzugefügt werden können:

- Über eine Erweiterung vom Typ **targetDriver** wird der Debugplattform ein Treiber für ein Zielsystem hinzugefügt. Neben dem Namen des Treibers wird die zentrale Java-Klasse des Treibers registriert, welche den Typ *DDebugTarget* realisiert. Das Treibermodell wird in Abschnitt

[2]Eclipse Webseite: http://www.eclipse.org

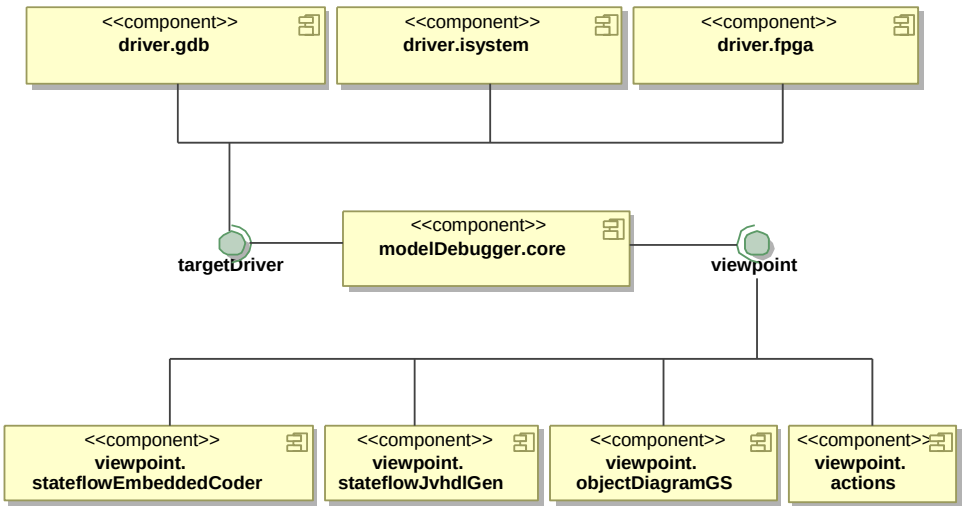

Abbildung 5.15: Kernkomponente modelDebugger.core und die Erweiterungspunkte targetDriver und viewpoint

5.3.3 genauer beschrieben. Die im oberen Bereich von Abbildung 5.15 gezeigten speziellen Treiber werden in Kapitel 6 behandelt.

- Über den Erweiterungspunkt **viewpoint** werden die verfügbaren Systemsichten registriert. Neben dem Modelltyp und dem verwendeten Codegenerator, kann hier die Hauptklasse des *Viewpoints* konfiguriert werden, welche die Schnittstelle *DIViewpoint* realisiert. Weiterhin kann festgelegt werden, welche Schnittstellen ein Treiber implementieren muss, damit der *Viewpoint* für ihn verfügbar ist. Abschnitt 5.3.4 beschreibt den Aufbau und die Funktionsweise von Viewpoints genauer. Die im Rahmen dieser Arbeit umgesetzten Systemsichten werden in Kapitel 7 behandelt.

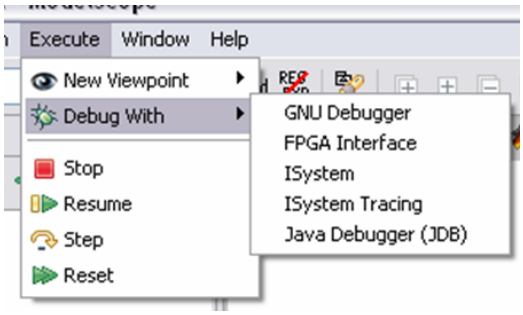

Abbildung 5.16: Menü zum Starten eine Debug-Sitzung durch Auswahl des Treibers

Abbildung 5.16 zeigt das *Execute*-Menü des Debuggers, welches alle über den Erweiterungspunkt *target* registrierten Treiber auflistet. Durch Auswahl eines Treibers wird zum Zielsystem verbunden und eine Debugging-Sitzung gestartet. Es wird immer das aktuell im Editorbereich von Eclipse aktivierte Modell ausgeführt.

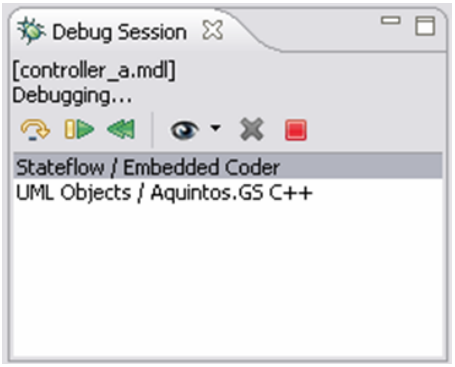

Abbildung 5.17: Die Debug Session Ansicht

Die *Debug Session* Ansicht in Abbildung 5.17 ist das zentrale Steuerelement für das Debugging von Modellen. Es zeigt in der ersten Zeile den Namen des gerade ausgeführten Modells an. Darunter befinden sich eine Statuszeile und eine Werkzeugleiste zu Steuerung des Debuggers. Im unteren Bereich befindet sich eine Liste parallel geöffneter Systemsichten. Die einzelnen Werkzeuge sind von links nach rechts:

- Einzelschritt (*Step*)
- Ausführung fortsetzen (*Resume*)
- Rücksetzen des Modelablaufs (*Reset*)
- Hinzufügen einer Systemsicht, bzw. eines *Viewpoint*
- Löschen des angewählten Viewpoint
- Beenden der Sitzung

Die *Debug Contexts* Ansicht in Abbildung 5.18 zeigt schließlich die Laufzeitinformationen aller geöffneten *Viewpoints* in einer Baumstruktur an. Diese Ansicht dient in erster Linie der Entwicklung neuer Systemsichten und sollte bei der Entwicklung nach Realisierung von *Mapper* und *Controller* durch einen geeigneten Projektor ersetzt werden.

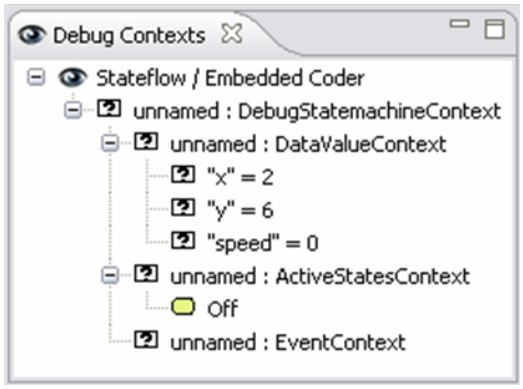

Abbildung 5.18: Darstellung aller Debug-Kontexte als Baumstruktur in der Debug Contexts Ansicht

5.3.3 Treiberschicht

Controller und Mapper greifen mittels einer Sammlung von Schnittstellen auf Treiber für unterschiedliche Quelltext-Debugger zu. Die Interfaces abstrahieren Funktionen aus dem Bereich der Ausführungskontrolle, des Speicherzugriffes und der Ablaufverfolgung.

Die Treiber realisieren diese Schnittstellen entsprechend den Fähigkeiten des Quelltext-Debuggers. Die Treiber steuern den Aufruf dieser eigenständigen Programme. Die instrumentierten Debugger kommunizieren mit der Zielplattform auf der das Modell ausgeführt wird.

Abbildung 5.19 verdeutlicht den Vorteil, welche die Vereinheitlichung der Fähigkeiten unterschiedlicher Quelltextdebugger auf eine beschränke Anzahl von Schnittstellen bietet. Die Abbildung zeigt die Verhältnisse ohne (oben) und mit (unten) Nutzung einer vereinheitlichten Schnittstelle.

Im oberen Bereich jeder Variante sind verschiedene Modellteile gezeigt, welche teilweise mit unterschiedlichen Codegeneratoren in Quelltext überführt wurden. Wird der Code auf unterschiedliche Zielplattformen gebracht und mit verschiedenen Quelltextdebuggern untersucht ergibt sich eine kombinatorisch hohe Zahl an benötigten *Viewpoints*. Diese sind in der Abbildung als schwarze Linien dargestellt. Durch Vereinheitlichung der Schnittstellen zumindest einiger Debugger kann die Anzahl der *Viewpoints* reduziert werden.

Die grundlegenden Schnittstellen der Treiberschicht sind in Abbildung 5.20 dargestellt. Jede in *DDebugSession* realisierte Debugging-Sitzung besitzt eine Verbindung zu einem *DDebugTarget*. Die Schnittstellen sind durch diesen Treiber verfügbar und werden im Folgenden näher beschrieben.

Einzelne in Kapitel 6 beschriebene Treibertypen vereinheitlichen weitere spezi-

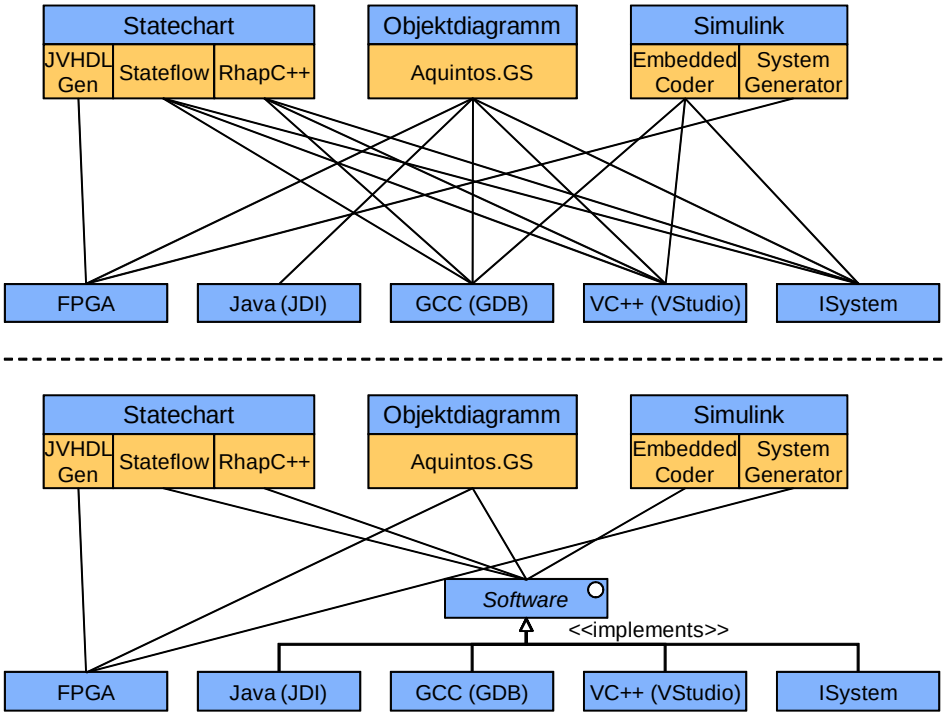

Abbildung 5.19: Verringerung der Anzahl benötigter Viewpoints (als schwarze Linien dargestellt) durch Definition standardisierter Schnittstellen

elle Schnittstellen, insbesondere Schnittstellen zum Aufzeichnen von Abläufen und zur Steuern des Zielsystems rückwärts in der Zeit (beide in Abschnitt 6.1.2.4).

5.3.3.1 DDebugTarget

Die abstrakte Klasse *DDebugTarget* definiert den Treiber gegenüber der Architektur von ModelScope. Die im Quelltext von ModelScope vorhandenen Assoziationen auf einen Treiber sind mit dieser Klasse typisiert. Alle realisierten Treiber leiten sich von dieser Klasse ab. Über das *DDebugTarget* kann die Verbindung mit dem Zielsystem aufgebaut und abgebrochen werden. Weiterhin kann überprüft werden, ob der Treiber eine spezielle Schnittstelle unterstützt. Bei Bedarf kann eine treiberspezifische Realisierung dieser Schnittstelle angefordert werden.

Die Klasse *DDebugTarget* besitzt folgende Operationen:

- **attach()** dient dem Herstellen einer Verbindung zum Zielsystem.

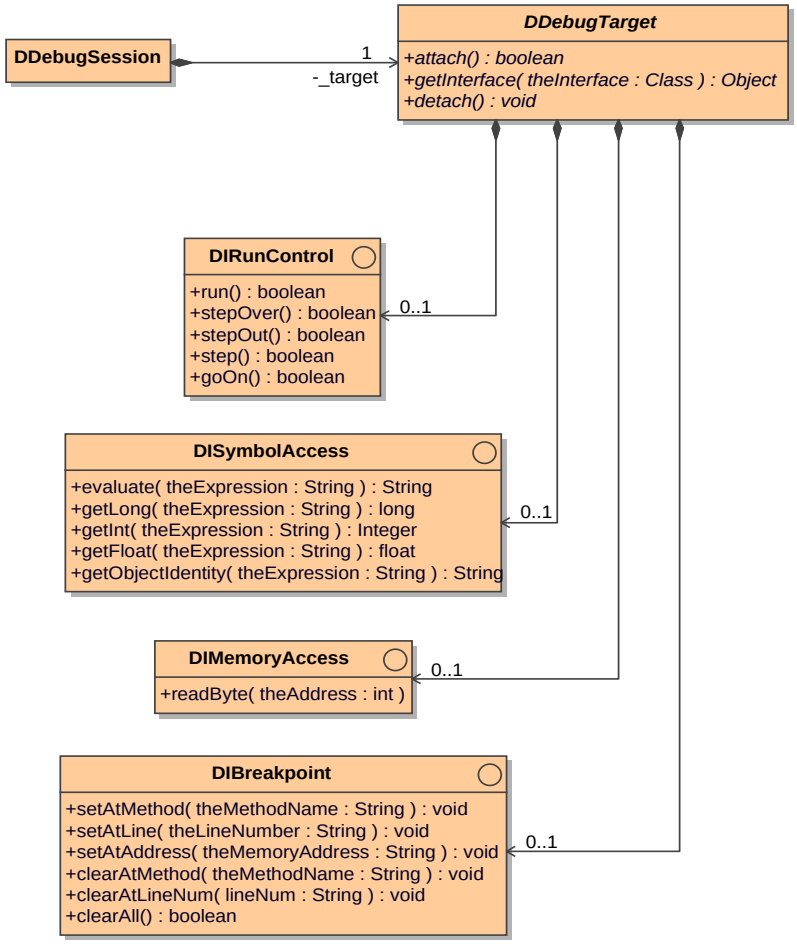

Abbildung 5.20: Softwarearchitektur der Treiberschicht

- **detach()** löst eine Verbindung zum Zielsystem auf, falls eine solche Verbindung besteht.

- **getInterface()** gibt eine Instanz der treiberspezifischen Realisierung einer Schnittstelle zurück, falls diese durch den Treiber implementiert wird. Die gewünschte Schnittstelle wird durch den Parameter übergeben. Falls keine Implementierung existiert, gibt die Operation *null* zurück[3].

[3]Das diesem Vorgehen zugrundeliegende Software-Entwurfsmuster wird als Erweiterungsobjekt (*Extension Object*) bezeichnet und ist in [Martin u. a. 1997] behandelt.

5.3.3.2 DIRunControl

Die Schnittstelle *DIRunControl* steuert den Ablauf des Debuggee. Die Operationen der Schnittstelle steuern den Ablauf auf Quelltextebene.

Die parameterlosen Operationen sind im Einzelnen:

- **run()** startet die Ausführung des Programms bis zum Erreichen eines Haltepunktes oder bis zur Beendigung des Programms.
- **goOn()** übernimmt eine ähnliche Funktion wie run(), setzt die Ausführung des Programms aber nach Erreichen eines Haltepunktes oder nach Einzelschritten fort.
- **step()** führt einen einzelnen Schritt auf Quelltextebene durch. Der Ablauf wird anschließend wieder angehalten. Stellt der Schritt den Aufruf einer Methode dar, wird in diese gesprungen und vor der ersten Anweisung gehalten.
- **stepOver()** verhält sich wie step(), führt bei einem Methodenaufruf diese jedoch komplett aus und hält den Ablauf bei der ersten Anweisung nach dem Aufruf an.
- **stepOut()** führt die Methode, die gerade ausgeführt wird so lange aus, bis durch eine Rücksprunganweisung oder eine Ausnahme ein Rücksprung in die aufrufende Methode stattfindet. Der Ablauf wird bei der ersten Anweisung nach dem Aufruf angehalten.

Alle Operationen sind blockierend und kehren nur zurück, wenn das Programm angehalten oder beendet wurde.

5.3.3.3 DISymbolAccess

Durch *DISymbolAccess* können Ausdrücke ausgewertet werden, welche Symbole enthalten, die im Debuggee auf Quelltextebene vorhanden sind. Dies kann neben den Werten von Variablen und Adressen von Objekten auch das Ergebnis komplexerer zusammengesetzter Ausdrücke, beispielsweise über mehrfache Referenzen sein. Die Syntax der Abfragen entspricht in C oder Java formulierten Ausdrücken. Die Auswertung komplizierter Ausdrücke ist nicht vereinheitlicht und stark von der tatsächlichen Realisierung des Debuggers abhängig.

Die einzelnen Operationen:

- **evaluate()** wertet den als Parameter übergebenen Ausdruck aus und bezieht dazu Werte aus dem Zielsystem. Gibt das Ergebnis als Zeichenkette oder *null* im Fehlerfall zurück.
- **getInt()** und **getLong()** rufen evaluate() auf und konvertiert das Ergebnis in einen ganzzahligen Wert.
- **getFloat()** ruft evaluate() auf und konvertiert das Ergebnis in eine Fließkommazahl.
- **getObjectIdentity()** gibt für ein durch einen Ausdruck bestimmtes Objekt eine eindeutige Zeichenkette zur Identifikation zurück. Dies kann beispielsweise in C++ die Adresse des Objekts sein. In Java werden der Typname und ein eindeutiger Index der Instanz zurückgegeben.

5.3.3.4 DIMemoryAccess

Insbesondere nicht auf Software basierende Debug-Zielsysteme besitzen eventuell keinen Zugriff über Variablennamen und andere Symbole (siehe beispielsweise die Schnittstelle für rekonfigurierbare Bausteine in Abschnitt 6.2.2). Aus dem Zielsystem können jedoch linear adressierbare Werte ausgelesen werden.

Dies geschieht über die einzige Operation der Schnittstelle:

- **readByte()** liest ein Byte aus einer durch den Parameter der Operation bezeichneten Speicherstelle.

5.3.3.5 DIBreakpoint

Die Schnittstelle *DIBreakpoint* erlaubt das Setzen und Löschen von Haltepunkten im Quelltext.

Die verfügbaren Operationen sind:

- **setAtMethod()** und **clearAtMethod()** setzen bzw. löschen einen Haltepunkt am Einsprungspunkt einer durch einen Ausdruck gegebenen Methode des Quelltexts.
- **setAtLine()** und **clearAtLine()** setzen bzw. löschen einen Haltepunkt auf einer Codezeile im Quelltext.
- **setAtAddress()** und **clearAtAddress()** setzen bzw. löschen einen Haltepunkt an einer Adresse im Instruktionsspeicher.
- **clearAll()** löscht alle zuvor gesetzten Haltepunkte.

5.3.4 Viewpoints

Viewpoints bündeln im Debugger die Ansteuerung des Zielsystems aus Modellsicht, den Aufbau eines speziellen Laufzeitmodells und die Visualisierung dieses Modells für den Benutzer.

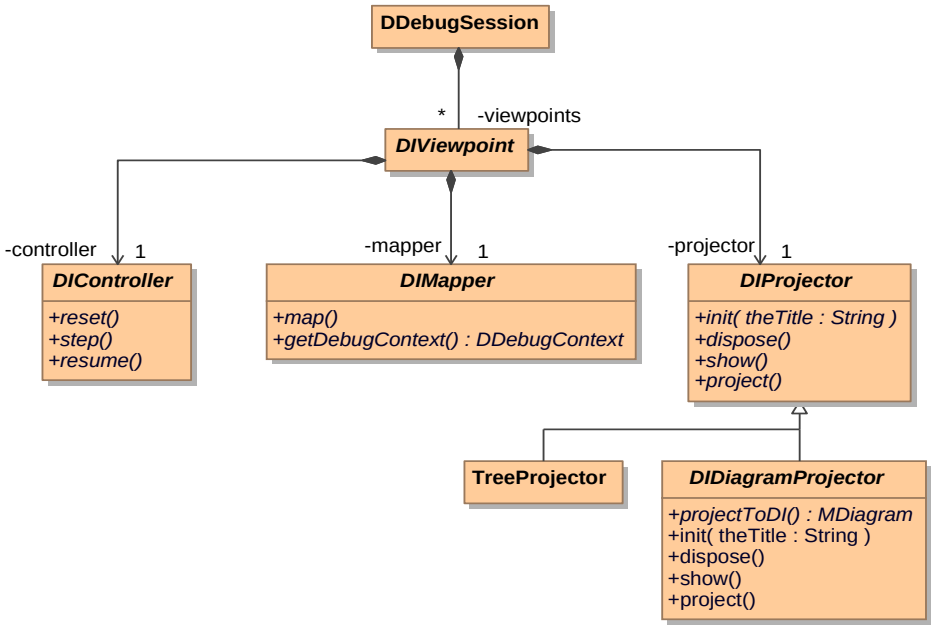

Abbildung 5.21: Grundstruktur Viewpoints, Controller, Mapper und Projector

Das in Abbildung 5.21 gezeigte Klassendiagramm verdeutlicht die logische Struktur. Im Verlauf jeder durch *DDebugSession* realisierten Sitzung können beliebig viele *DIViewpoints* aktiv sein. Ein *DIViewpoint* agiert als Mediator und kann der *DDebugSession* Viewpoint-spezifische Implementierungen von *DIController*, *DIMapper* und *DIProjector* anbieten.

5.3.4.1 Controller DIController

Der Controller steuert das Zielsystem. Jeder Viewpoint besitzt eine von *DIController* abgeleitete Klasse, welche Operationen zur Steuerung besitzt. *DIController* unterscheidet sich trotz ähnlicher Operationen fundamental von *DIRunControl* in Abschnitt 5.3.3. *DIController* steuert das Target auf Sicht des Modells und nutzt dabei die Funktionalität des Treibers, auch *DIRunControl*, zur Ansteuerung auf Quelltextebene.

Die Operationen eines *DIControllers* sind:

- **reset()** setzt die Ausführung des Modells zurück auf den Zustand nach Start des Modellausführung. Typischerweise wird dadurch auch das Zielsystem zurückgesetzt.
- **step()** führt einen Einzelschritt auf Modellebene durch. Ein Einzelschritt setzt sich im Allgemeinen aus aus mehreren Schritten bestehenden Abläufen auf Quelltextebene zusammen.
- **resume()** führt den Ablauf fort, bis das Modell beendet oder bis ein Haltepunkt oder eine Haltebedingung des Modells erreicht ist.

5.3.4.2 Mapper DIMapper

Der *DIMapper* aktualisiert das dynamische Laufzeitmodell basierend auf Informationen aus dem statischen Modell und dem Systemzustand auf Quelltext oder Signalebene. Jeder Mapper verwaltet einen *DebugContext* (siehe Abschnitt 5.2.2), der die Laufzeitdaten des *Viewpoint* enthält.

Die Schnittstelle setzt sich aus zwei Operationen zusammen:

- **map()** führt die Abbildung für diesen Viewpoint durch. Das untersuchte System darf durch die Abbildung nicht beeinflusst werden, daher wird eine Implementierung nur Operationen des Treibers nutzen, welche den Zustand des Zielsystems nicht verändern.
- **getDebugContext()** gibt den von diesem Mapper verwalteten *DebugContext* zurück.

5.3.4.3 Projektor DIProjector

Der *DIProjector* bereitet die im Laufzeitmodell gespeicherte Information visuell für den Benutzer auf. Die Darstellung kann dabei beliebig, beispielsweise in Diagrammform, als Baumstruktur oder als Textausgabe, erfolgen.

Die Schnittstelle besteht aus folgenden Operationen:

- **init()** initialisiert die Anzeige nach dem Hinzufügen des Viewpoints zur Debug-Sitzung. Die Operation öffnet beispielsweise einen graphischen Editor oder eine Komponente der Benutzerschnittstelle zur Baumanzeige.
- **dispose()** entfernt das Anzeigeelement wieder. *dispose()* wird beim Entfernen des *Viewpoints* oder der Beendigung der Debug-Sitzung aufgerufen.

- **show()** bringt die Anzeige des Projektors visuell in den Vordergrund.
- **project()** führt die visuelle Darstellung der aktuellen Laufzeitinformationen des Mappers durch. Die Operation wird von *DDebugSession* nach jedem Mapping aufgerufen.

Für den *DIProjector* wurde eine spezialisierte Variante realisiert, welche die Darstellung von Modellen in Diagrammen ermöglicht. *DIDiagramProjector* implementiert alle Operationen von *DIProjector* und ist in der Lage, auf dem in Abschnitt 2.2.6 beschriebenen *Diagram Interchange* Standard basierende Diagrammmodelle anzuzeigen und editierbar zu machen.

Zur Realisierung eines von *DIDiagramProjector* abgeleiteten Projektors muss nur eine Operation implementiert werden:

- **projectToDI()** überführt die Laufzeitinformation des Systems in ein auf dem *Diagram Interchange* Standard basierendes Diagrammmodell und gibt das Diagramm zurück.

Der *DIDiagramProjector* basiert auf dem Eclipse-Projekt *Graph Editor Framework* (GEF) [Eclipse Foundation 2008b; Moore u. a. 2004]. Beispiele für mit *DIDiagramProjector* realisierte Diagramme in ModelScope zeigen Abbildung 7.6 und Abbildung 7.12.

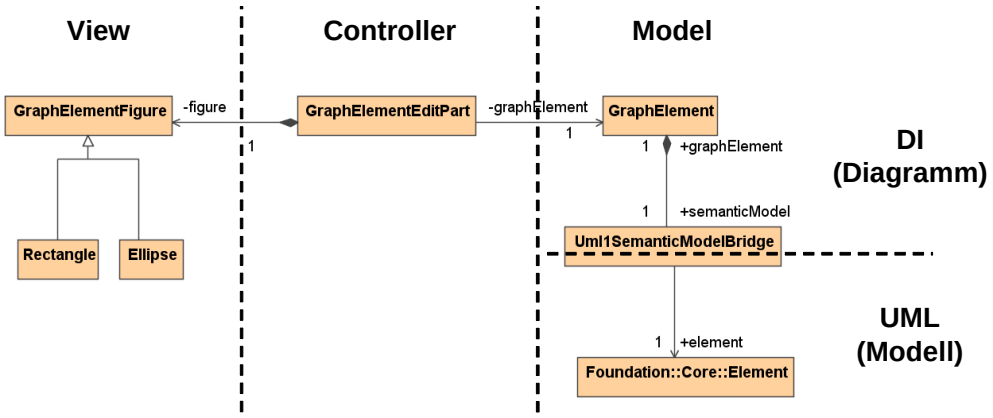

Abbildung 5.22: Model-View-Controller Ansatz zur Visualisierung

Der Projektor realisiert die in Abbildung 5.22 abgebildete *Model-View-Controller*-Architektur. Aus Sicht des graphischen Editors besteht das Diagramm aus einer graphischen Ansicht, dem *View*.

Das Modell des Diagramms basiert auf dem DI-Metamodell und referenziert Elemente im eigentlichen Datenmodel über eine *SemanticModelBridge*. Für den

Editor sind demnach primär die Diagramminformationen das Modell, welches dargestellt werden soll.

Der Controller verknüpft Ansicht und Modell. Da im *Diagram Interchange* Metamodell Diagramminformationen generisch als Menge von Knoten und Kanten abgelegt werden kann, kann ein generischer Controller diese Modellelemente für jeden Diagrammtyp mit Hilfe weniger einfacher Subklassen und einer Konfigurationsdatei verwalten.

5.3.4.4 Interaktion innerhalb des Viewpoints

Abbildung 5.23 verdeutlicht die Abläufe im Zusammenspiel der beschriebenen Komponenten des *Viewpoint* während des Debug-Ablaufs. Dies geschieht am Beispiel des in Abschnitt 7.1 beschriebenen *Viewpoint* für Stateflow-Modelle, aus welchen mit *Embedded Coder* C-Code generiert wurde. Die Debug-Sitzung besitzt in diesem Beispiel nur einen *Viewpoint*.

Löst der Benutzer des Debugger auf der Benutzeroberfläche einen Einzelschritt aus, wird die Operation *step()* auf *DDebugSession* aufgerufen. In dieser Operation wird zunächst der *DIController* des *Viewpoints* ausgewählt, der im *Debug Session* Fenster (siehe Abbildung 5.17) ausgewählt markiert ist. Der Controller dieses *Viewpoints* wird ausgewählt und dessen Operation *step()* aufgerufen.

Ist der Einzelschritt ausgeführt, müssen die Laufzeitmodelle aller Systemsichten und ihre Darstellung aktualisiert werden. Dazu wird die Operation *mapAll()* aufgerufen, in der über alle *Viewpoints* iteriert wird. Für jeden *Viewpoint* wird zunächst die Operation *map()* des zugehörigen Mappers und anschließend die Operation *project()* des zugehörigen Projektors aufgerufen. Im Beispiel in Abbildung 5.23 delegiert der *Mapper* Teile der umfangreichen Abbildung an untergeordnete spezialisierte *Mapper*, welche aktive Zustände und den Datenkontext des Stateflow-Diagramms rekonstruieren.

5.3.5 Bewertung

Die im vorigen Abschnitt beschriebene Architektur kann als Basisplattform für das Debugging von Modellen genutzt werden. Durch das flexible Erweiterungskonzept kann die Plattform um beliebige Systemsichten und Ausführungsplattformen erweitert werden. Auch *Tailoring* ist möglich. Durch Auswahl einzelner Komponenten kann *ModelScope* auf unterschiedliche Entwicklungsplattformen und Werkzeugketten zugeschnitten werden.

ModelScope kann dazu dienen, neue Ansätze zu untersuchen, wie der Entwick-

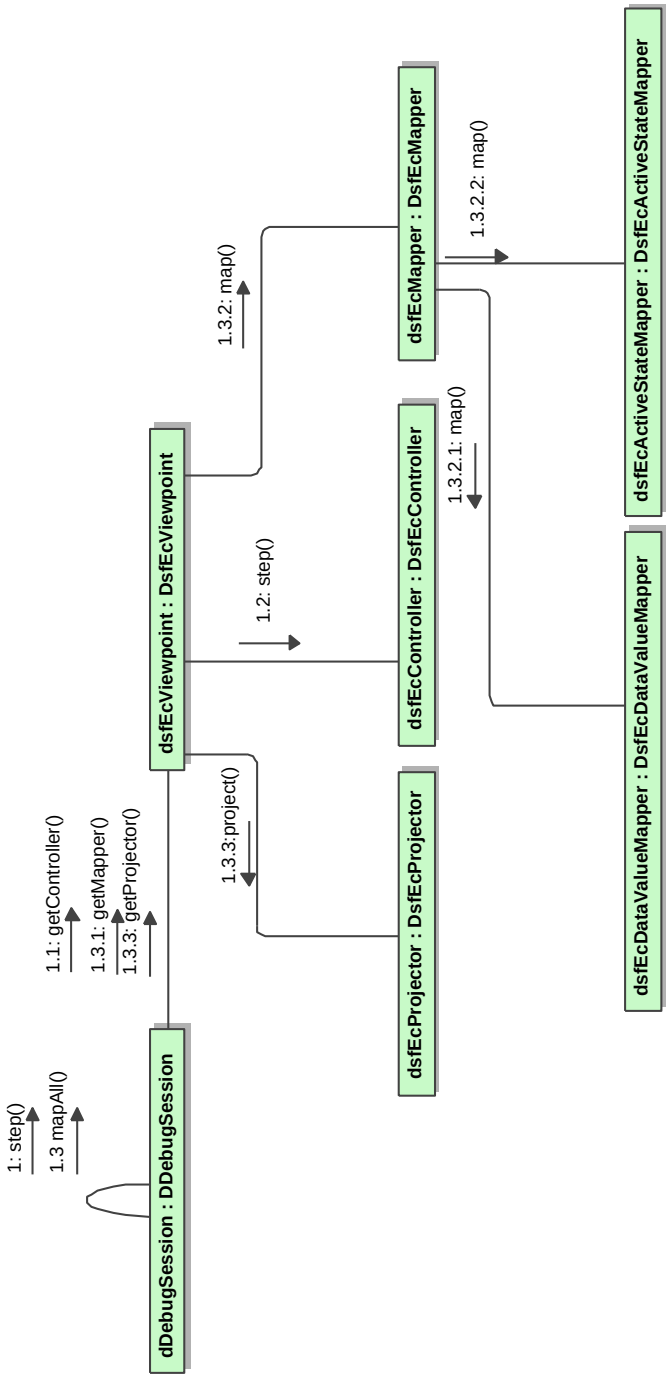

Abbildung 5.23: Beteiligte Partner bei einem Einzelschritt und ihre Interaktion im Kommunikationsdiagramm

ler eines modellbasiert entworfenen Systems dabei unterstützt werden kann, Fehler zu finden und zu analysieren.

Ein grundlegender Gewinn des Ansatzes ist, dass unterschiedliche Achsen der Heterogenität abgedeckt werden können. Dabei wird die Wiederverwendbarkeit der Komponenten innerhalb des Debuggers maximiert:

- Entwicklung mit unterschiedlichen domänenspezifischen Modellierungswerkzeugen: Durch angepasste Viewpoints kann der Nutzer/Entwickler angemessene Sichten auf unterschiedliche Subsysteme angeboten bekommen.
- Unterschiedliche Werkzeugketten und Zielsysteme bei der Entwicklung: Durch die Implementierung eines Treibers, können weitere Zielsysteme, welche mit unterschiedlichen Prozessoren, Betriebssystemen und Debug-Schnittstellen ausgestattet sind, integriert werden. Höhere Ebenen bleiben unangetastet.
- Heterogene Architekturen: Modellbasiertes Hardware-/Software-Codesign kann durch Hardware-/Software-Codebugging unterstützt werden. Mit zunehmender Wichtigkeit rekonfigurierbarer Hardware und parallelem Hardware- und Software-Entwurf, erlaubt ein modellbasierter Ansatz späte Entscheidungen über die Ausführungsplattform.

Kapitel 6

Zielarchitekturen

Die im vorigen Kapitel vorgestellte Architektur zum Debugging von Modellen erhebt den Anspruch auf unterschiedlichste Zielarchitekturen anwendbar zu sein. Zur Validierung dieser Anforderung entstanden Treiber für vier unterschiedliche Zielsysteme welche folgende Debugging-Paradigmen exemplarisch überspannen:

- Nicht-echtzeitfähiges Debuggen auf eingebetteten Mikroprozessoren (Abschnitt 6.1). Dies wurde für den Quelltextdebugger *GDB* und einen proprietäres Emulatorsystem realisiert.
- Echtzeitfähiges *Post-Mortem*-Debuggen auf eingebetteten Mikroprozessoren in Abschnitt 6.1.2.4.
- Nicht-echtzeitfähiger Treiber für eine *Java-Virtual-Machine* und damit neben der Integration von C und C++ ein Treiber für Java (Abschnitt 6.1.1).
- Generierbare Debugging-Schnittstelle und Debugger-Treiber für auf FPGAs ausgeführte Modelle [Graf u. Müller-Glaser 2007]. Diese werden neben dem Umfeld der modellbasierten Entwicklung für rekonfigurierbare Hardware und existierenden Debug-Schnittstellen in Abschnitt 6.2 behandelt.

6.1 Mikroprozessoren

Die zum Debugging von auf Mikroprozessoren ausgeführter Software relevanten Verfahren wurden bereits in Abschnitt 4.3.2 untersucht. Für solche Zielsysteme

existieren zahlreiche, oft in ihren Fähigkeiten ähnliche, Systeme zum Debugging. Sehr häufig sind diese direkt in eine Entwicklungsumgebung integriert und daher nur schlecht für einen *ModelScope*-Treiber zugänglich. Ausnahmen sind der GNU-Debugger (siehe Abschnitt 4.3.2.1) und der Java-Debugger (siehe Abschnitt 4.3.2.2). Einen weiteren Treiber, welcher über eine echtzeitfähige Schnittstelle zum Tracing verfügt, beschreibt Abschnitt 6.1.2.

6.1.1 Schnittstellen GNU Debugger und Java Debugger

Sowohl der GNU Debugger als auch der Java Debugger verfügen über textbasierte Benutzerschnittstellen, welche in ihrem Konzept und den verfügbaren Befehlen große Ähnlichkeiten aufweisen.

Eine Integration in das Treibermodell von *ModelScope* kann über eben diese Benutzerschnittstelle erfolgen: Der Debugger wird aus *ModelScope* als eigenständiger Prozess gestartet. Seine Ein- und Ausgabeströme werden so umgeleitet, dass der Treiber den Debugger steuern und die Ausgaben auswerten kann.

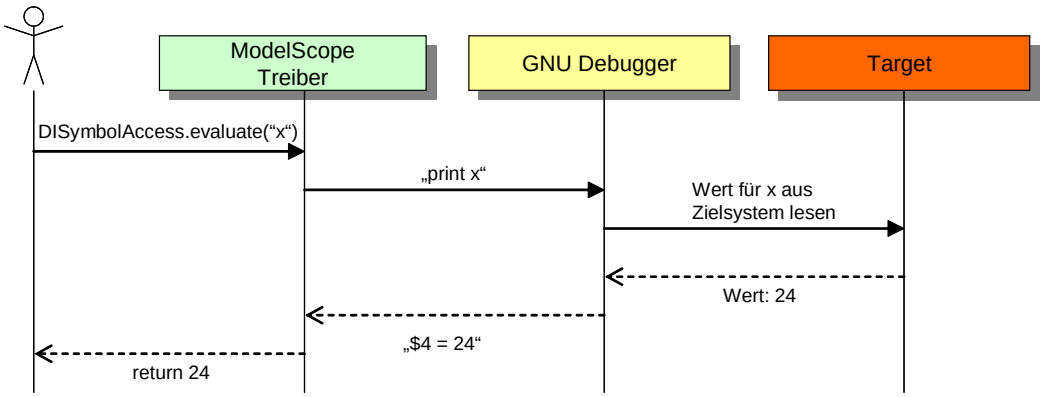

Abbildung 6.1: Steuerung des GNU Debuggers über Ein- und Ausgabeströme.

Die Befehle der Treiberschicht lassen sich auf textuelle Befehle abbilden. Das erhaltene textuelle Ergebnis kann ausgewertet und entsprechend der vereinbarten Treiberschnittstelle von *ModelScope* aufbereitet werden.

Abbildung 6.1 verdeutlicht den grundlegenden Ablauf im GNU Debugger anhand eines Sequenzdiagramms. Aus dem Aufruf eines Befehls zur Auswertung eines Ausdrucks, hier dem Lesen der Variable x, generiert der Treiber den Befehl *print x*, welcher über den Eingabestrom an den GDB gegeben wird. Dieser liest den Wert der Variablen x aus dem Zielsystem, und der GNU Debugger gibt eine menschenlesbare Version des Ergebnisses über seinen Ausgabestrom zurück. Der Treiber zerlegt die Zeichenkette und gibt den Wert zurück.

Operation in Treiber	**Realisierung GNU Debugger (GDB)**	**Realisierung Java Debugger (JDB)**
DITarget	DGdbTarget	DJdbTarget
attach()	Starten des GDB-Prozesses, `file <Dateiname>`	Starten JDB, Debuggee durch Klassenpfad und -name in Parametern
detach()	Beenden des GDB-Prozesses	Beenden des JDB-Prozesses
DIRunControl	DGdbRunControl	DJdbRunControl
run()	`run`	`cont`
goOn()	`continue`	`cont`
step()	`step`	`step`
stepOver()	`next`	`next`
stepOut()	`finish`	`step up`
DIBreakpoint	DGdbBreakpoint	DJdbBreakpoint
setAtMethod (Methodenname)	`break <Methodenname>`	`stop in <Methodenname>`
setAtLine (Zeilennummer)	`break <Zeilennummer>`	`stop at <Zeilennummer>`
setAtAddress (Adresse)	`break <Adresse>`	nicht implementiert
clearAtMethod (Methodenname)	`clear <Methodenname>`	`clear <Methodenname>`
clearAtLine (Zeilennummer)	`clear <Zeilennummer>`	`clear <Zeilennummer>`
clearAll()	`delete`	Enumeration aller Haltepunkte mit `clear`. Einzelnes Löschen mit `clear <HP>`.
DISymbolAccess	DGdbSymbolAccess	DJdbSymbolAccess
evaluate(Ausdruck)	`print <Ausdruck>`	`print <Ausdruck>`
getLong() / getInt() / getFloat()	Aufruf von evaluate()	Aufruf von evaluate()
getType(Ausdruck)	`whatis <Ausdruck>`	Typ aus evaluate()
getObjectIdentity (Ausdruck)	Adresse aus evaluate()	Identifier aus evaluate()

Abbildung 6.2: Operationen des Treibers von ModelScope und die Realisierung in GDB und JDB.

Ein solches Vorgehen bietet gegenüber der Integration auf niedrigerer Ebene (z.B GDB/MI im GNU Debugger und JDPA zum Debuggen in Java) einige Vorteile:

- Die Integration erfolgt über eine sehr einfache und gut dokumentierte Schnittstelle, welche auch von Nutzern der Quelltextdebugger täglich genutzt wird.
- Die Schnittstelle ist schlank und besteht aus nur einem Eingabe- und einem Ausgabestrom.
- Der Quelltextdebugger läuft in einem eigenen Prozess und ist damit von *ModelScope* entkoppelt. Über die Ein- und Ausgabeströme ist eine Synchronisierung trotzdem einfach realisierbar.

Es stehen dem jedoch auch Nachteile gegenüber:

- Es ist nicht garantiert, dass die Schnittstelle über mehrere Versionen des Debuggers stabil bleibt. Speziell die durch Menschen lesbare Ausgabe des Debuggers ist häufig kleineren Änderungen unterworfen. Dadurch ist die Schnittstelle nicht robust.
- Die Umwandlung von gelesenen Daten in eine Zeichenkette und die darauf folgende Aufbereitung im Treiber von *ModelScope* ist nicht sehr effizient. Dies spielt jedoch nur beim Auslesen großer Datenmengen eine Rolle.

Diese Vor- und Nachteile gelten sowohl für den GNU Debugger als auch den Java Debugger JDB.

Die Realisierung der Operationen der Treiber zum Debuggen via GDB und JDB ist in Abbildung 6.2 zusammengefasst. Neben den Treiberschnittstellen und ihren Operationen in der ersten Spalte, sind die Befehle der Debugger GDB und JDB welche ihnen grundlegend entsprechen, in den folgenden Spalten aufgeführt. Die grau hinterlegten Zeilen zeigen den Namen der Schnittstelle und die Klassen, die sie realisieren.

6.1.2 Schnittstelle In-Circuit Emulator

Neben der GNU Werkzeugkette wurde untersucht, wie die in Kapitel 5 beschriebene Plattform auf ein Emulatorsystem für eingebettete Prozessoren angewendet werden kann.

6.1.2.1 Debug-Plattform

Die Grundidee eines Emulatorsystems wurde bereits in Abschnitt 4.4 näher erläutert.

Abbildung 6.3: Physikalischer Aufbau der Emulatorsystems iC3000 ActiveEmulator (links) und ActivePOD II (oben rechts) mit Testplatine zur Lenkscheinwerferansteuerung (unten).

Für die vorliegende Arbeit kam das in Abbildung 6.3 gezeigte Emulatorsystem *iC3000 ActiveEmulator* der Firma iSYSTEM[1] aus Schwabhausen zum Einsatz. Dieses Gerät stellt ein generisches System zum echtzeitfähigen Debuggen eingebetteter Prozessoren dar. Abhängig vom verwendeten Mikroprozessor und der verwendeten Emulationsmethode *On-Chip-Debug*, *On-Chip-Trace* oder *Bond-Out Controller* (siehe Abschnitt 4.4) wird ein weiteres Gerät benötigt, das die Brücke zur Zielhardware darstellt und den eingebetteten Prozessor konnektiert oder ersetzt. Das System ist modular aufgebaut und unterstützt dadurch etliche Mikrocontroller-Familien. Für jede Typengruppe wird gesonderte Hardware benötigt. In der vorliegenden Arbeit wurde ein 68HC12-Controller der Firma Freescale[2] emuliert.

Im in Abbildung 6.3 gezeigten Fall kommt ein *ActivePOD II* Adapter für den Prozessor zum Einsatz, welcher die Funktionen der CPU nachbildet, so dass sich das Systemverhalten nicht verändert. Er enthält eine spezielle *Bond-Out*

[1] iSYSTEM Webseite: http://www.isystem.de

[2] Freescale Webseite: http://www.freescale.com

Version des 68HC12 und integriert sich in den Sockel der im unteren Teil der Platine gezeigten Zielhardware.

Die Anbindung des Emulators an einen Entwicklungsrechner kann durch USB 2.0, Ethernet oder eine serielle Schnittstelle erfolgen. Die Anbindung ermöglicht eine Busgeschwindigkeit von bis zu 100Mbit/s, ist jedoch nicht zeitkritisch. Das System enthält einen Speicher, welcher neben der ausgeführten Software einen Speicher für Haltepunkte und einen Ringpuffer für aufgezeichnete Echtzeit-Traces enthält.

Neben *Breakpoints* erlaubt die Emulationshardware durch die vollständige Kontrolle der Abläufe im Emulator komplexe Bedingungen zur Ausführungsunterbrechung. Darunter fällt das Anhalten bei bestimmten Schreibzugriffen im Speicher, jedoch auch selektives Aufzeichnen des Ablaufs und Profiling von Funktionsaufrufen.

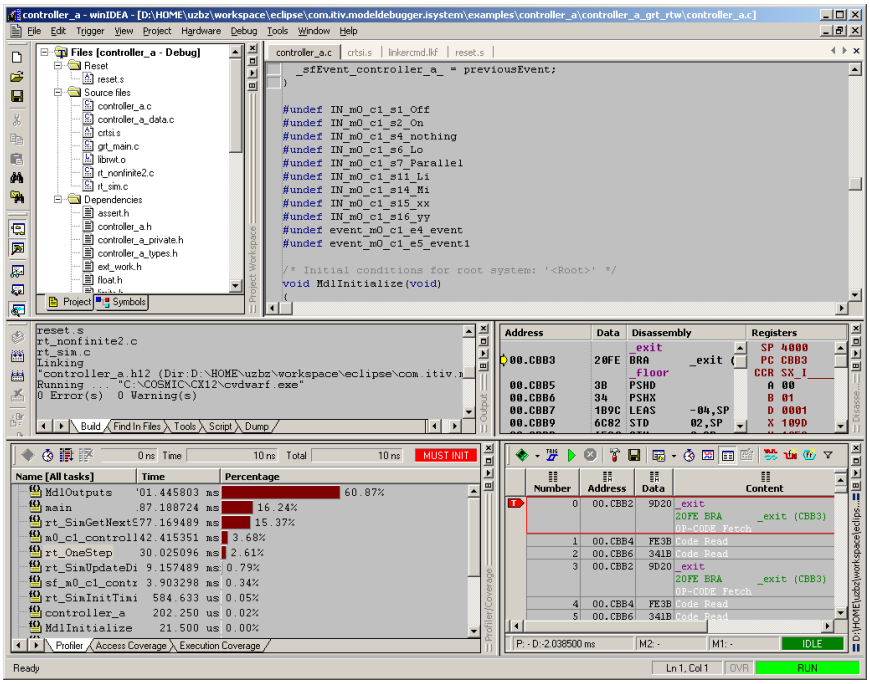

Abbildung 6.4: Grafische Benutzeroberfläche winIDEA

Die mitgelieferte Schnittstelle der Hardware zum Benutzer ist durch die Software winIDEA gegeben, die in Abbildung 6.4 dargestellt ist. Sie beinhaltet eine umfassende integrierte Entwicklungsumgebung mit Funktionen zur Projektverwaltung (links oben), einen Quelltexteditor (rechts oben) und ein automatisches *Build*-System.

Kern von winIDEA ist jedoch die Ansteuerung der iSystem Debug-Hardware. Sie bietet zum Debuggen von Software die geläufigen Ansichten wie Markierung

des Programmzählers im Quelltext, Beobachten und Manipulieren von Variablenwerten und setzen von Haltepunkten. Zusätzlich kann ein zeitliches Ausführungsprofil erstellt werden (Abbildung 6.4 links unten), sowie ein Echtzeit-Trace der unmittelbaren Vergangenheit aufgezeichnet werden (rechts unten).

6.1.2.2 Schnittstelle

winIDEA bietet Schnittstellen zur Fernsteuerung aus anderen Programmen und stellt dadurch fast die gesamten Möglichkeiten der Debug-Hardware über eine vereinheitlichte Schnittstelle zur Verfügung. Unterstützt wird eine Anbindung von *LabView*[3] der Firma National Instruments und das *RemoteSerial-Protocol* des in Abschnitt 6.1.1 vorgestellten GNU Debugger.

Die gesamten Fähigkeiten sind jedoch nur über die proprietäre Schnittstelle iSYSTEM.connect [iSystem 2008] zugänglich. Die Schnittstelle bildet die Funktionalität der Software in Methodenaufrufen einer DLL *iconnect.dll* ab. Die Fernsteuerung erfolgt als Client-Server-Architektur, d.h. zu aktiven Instanzen der Software winIDEA kann lokal oder über Netzwerk verbunden werden.

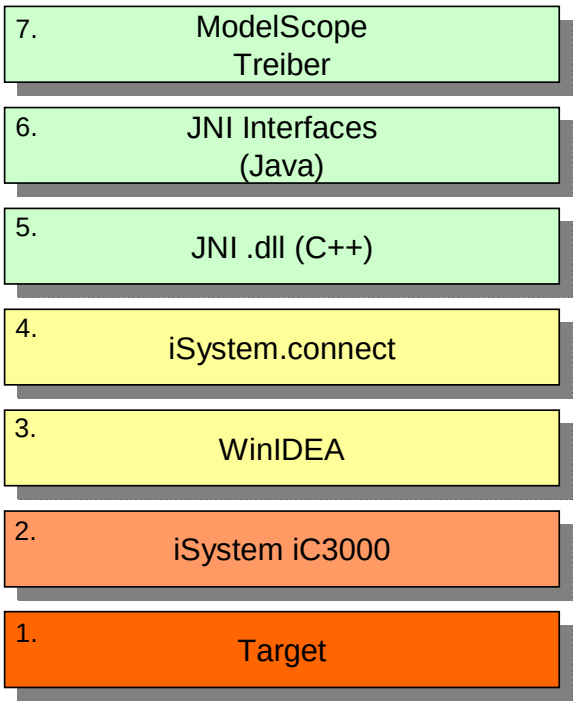

Abbildung 6.5: Schichten zwischen ModelScope und Zielsystem

Die vollständigen Zusammenhänge zwischen den Schichten, ausgehend von der

[3]LabView Webseite: http://www.ni.com/labview

Zielplattform bis zur realisierten Treiber-Schnittstelle in *ModelScope*, verdeutlicht Abbildung 6.5.

Die niedrigste Schicht 1 bezeichnet das Zielsystem, welches einen eingebetten Prozessor besitzt. Als Hardware-Adapter fungiert die *iSystem iC3000*-Hardware (Schicht 2), welche durch die Software *winIDEA* angesteuert wird (Schicht 3). *winIDEA* stellt die in Schicht 4 dargestellte Schnittstelle *iSystem.connect* zur Verfügung.

Die Nutzung unterschiedlicher Umgebungen für das nativ in Windows realisierte winIDEA und die Schnittstelle iSystem.connect auf der einen Seite und der in Java umgesetzten *ModelScope*-Plattform andererseits, erfordert einen geeigneten Übergang zwischen beiden.

Um aus der virtuellen Maschine (VM) von Java aus auf Funktionen zuzugreifen, welche nicht in Java oder der mitgelieferten Laufzeitbibliothek selbst realisiert sind, existiert das *Java Native Interface* (JNI) (siehe [Liang 1999]). JNI stellt eine einheitliche Schnittstelle zwischen der virtuellen Maschine und von der Zielplattform abhängigen Bibliotheken dar. Bei Microsoft Windows Betriebssystemen sind diese Bibliotheken *Dynamic Link Libraries* (DLL), unter Linux *Shared Libraries*.

Eine zentrale Herausforderung bei der Kopplung von Java-Code an native Bibliotheken ist die Handhabung von Datentypen. Das JNI definiert Datentypen auf welche sowohl im plattformabhängigen Quelltext, als auch in Java zugegriffen werden kann. Diese sind nur für einfache Datentypen, nicht jedoch für komplexer Datenstrukturen verfügbar.

Aus diesem Grund ist es nicht möglich, direkt auf alle Funktionen der *iconnect.dll* zuzugreifen. Lösung ist eine weitere Bibliothek, welche *iconnect.dll* umhüllt und die Datentypen der Schnittstelle auf Java-Datentypen abbildet. Dieses System bildet die Schichten 4 bis 6 in Abbildung 6.6.

Dies soll an einem Beispiel verdeutlicht werden. Im folgenden Listing ist ein Ausschnitt aus der Schnittstellenspezifikation von *iSystem.connect* gezeigt (Schicht 4):

Listing 6.1: Ausschnitt Spezifikation *iSystem.connect*

```
interface IConnectDebug: public IUnknown {
  HRESULT RunControl(DWORD runControlFlags, ADDRESS address);
}
```

Die Schnittstellenklasse *Debug* besitzt eine Operation *RunControl* zur Steuerung des Programmablaufs im Debugger. Der erwünschte Konnektor in Java in Schicht 6:

Listing 6.2: Ausschnitt *Connector*-Klasse in Java

```
package modeldebugger.driver.isystem;

public class Debug {
 public native int RunControl(int theRCFlags, int theAdress);
}
```

Verbunden werden diese Schnittstellen durch die c*onnector.dll*. Den für das Beispiel relevanten Ausschnitt zeigt Listing 6.3:

Listing 6.3: Auschnitt Realisierung *Connector* DLL

```
JNIEXPORT jint JNICALL ...
    ...Java_modeldebugger_driver_isystem_Debug_RunControl
  (JNIEnv *env, jobject obj, jint theFlags, jint theAddress) {
  HRESULT hrst = ICC.m_pIConnectDebug->RunControl(theFlags, ...
      ...theAddress);
  return hrst;
}
```

Die Realisierung eines der Anbindung des GDB entsprechenden Treibers in *ModelScope* für das soeben beschriebene Zielsystem wird im folgenden Abschnitt behandelt. Die Realisierung eines Treibers, welcher die *Trace*-Fähigkeiten zum *Post-Mortem*-Debugging unter Echtzeitrandbedingungen nutzt, wird in Abschnitt 6.1.2.4 untersucht.

6.1.2.3 Nicht-echtzeitfähiger Treiber

Der Modelldebugger *ModelScope* nutzt Treiber um den Quelltext-Debugger zu kontrollieren und um Laufzeitdaten aus ihm auszulesen. Während der Laufzeit eines Modells wird die Ausführung unterbrochen um anschließend Daten auszulesen und den Ablauf gegebenenfalls fortzusetzen. Das zeitliche Verhalten geht bei einem solchen Vorgehen verloren. Der Treiber ist, ähnlich wie der in Abschnitt 6.1.1 beschriebene Treiber für GDB, somit nicht echtzeitfähig.

Der Treiber selbst integriert sich wie in Abbildung 6.6 dargestellt in die *ModelScope*-Plattform. Die Schnittstelle für Treiber ist in Abschnitt 5.3.3 genau beschrieben. Das zentrale Element ist die abstrakte Klasse *DISysDebugTarget*, welche ein *DDebugTarget* realisiert. *DISysDebugTarget* dient als Vorlage für die konkreten Treiberklassen zum nicht-echtzeitfähigen und zum echtzeitfähigen Debugging. Sie bündelt in Komposition die im vorherigen Abschnitt vorgestellten Wrapper-Klassen *IConnect* und *Debug* zum Zugriff auf das Ziel-

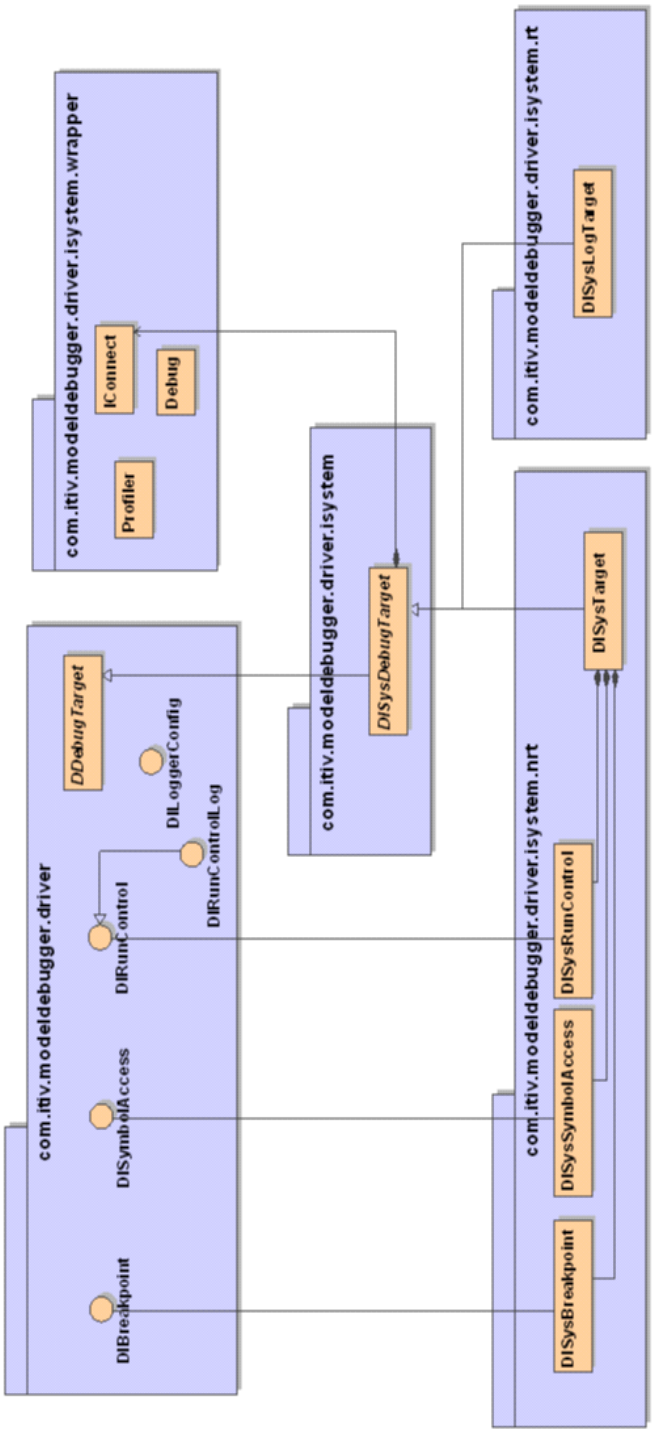

Abbildung 6.6: Einbindung Treiber in das ModelScope-Framework

system. Weiterhin wird festgelegt, dass beide Treiber die Schnittstellen *DIBreakpoint*, *DIRunControl* und *DISymbolAccess* realisieren.

Der nicht echtzeitfähige Treiber wird durch die Klasse *com.itiv.modeldebugger.driver.isystem.nrt.DISysTarget* realisiert, welcher eine Verbindung zum *Debugee* verwaltet und die in den Klassen *DISysBreakpoint*, *DISysRunControl* und *DISysSymbolAccess* implementierten Schnittstellen über die Operation *getInterface()* anbietet.

Aspekte der Realisierung der einzelnen Schnittstellen werden in den folgenden Unterabschnitten untersucht.

6.1.2.3.1 Breakpoints Die Schnittstelle *DISysBreakpoint* dient dazu, dem Quelltext-Debugger mitzuteilen an welchen Stellen Haltepunkte gesetzt oder gelöscht werden sollen. Diese Funktionalität wird von *iSystem.connect* angeboten und im Wesentlichen über Flags gesteuert. Für die Realisierung der einzelnen Methoden zum Setzen und Löschen von Haltepunkten sind die entsprechenden Flags zu setzen und als Aufrufparameter an die Methode *SetBreakpoint* weiterzuleiten. Zu Beachten ist, dass *iSystem.connect* keine Möglichkeit bietet, Informationen über gesetzte Haltepunkte auszulesen, so dass der *ModelScope*-Treiber diese Information selbst verwaltet.

6.1.2.3.2 Ablaufkontrolle Über *DIRunControl* kann die Ausführung gesteuert werden. Zum Starten der CPU dienen die Befehle *run* und *goOn*. *run* setzt den Emulator zurück, bevor die Ausführung startet. Die Bibliothek von *iSystem.connect* bietet zur Ablaufkontrolle die bereits in Listing 6.1 vorgestellte Methode *RunControl*. Auch diese wird über Flags gesteuert.

Die Aufrufe der Treiberklasse *DIRunControl* müssen blockierend sein, bis das Programm beendet ist oder durch einen Haltepunkt oder nach einem Einzelschritt wieder angehalten ist. Die Funktionen von *iSystem.connect* stoßen die Ausführung jedoch nur an und kehren sofort zurück. Daher warten alle Methoden der Klasse *DISysRunControl* nach Starten der Ausführung über regelmäßiges Abfragen des Ausführungszustands ab, bis der Emulator nicht mehr im Zustand *Running* ist.

6.1.2.3.3 Symbolzugriff Die Schnittstelle *DISymbolAccess* erlaubt das Auslesen von Variablen und die Evaluation komplexer Ausdrücke, welche Laufzeitwerte enthalten. *iSystem.connect* besitzt eine Methode *Evaluate()* welche genau dies ermöglicht. Die in *DISysSymbolAccess* enthaltenen Operationen *evaluate*, *getInt*, *getFloat* und *getLong* werden auf diese abgebildet.

Die Operation *getObjectIdentity()* wird durch das Auslesen der Objektadresse mitsamt dem Speicherbereich über *GetAddress()* realisiert.

Auch die Typisierung des Ergebnisses von Ausdrücken, welche für *getType()* benötigt wird, kann aus dem von *Evaluate()* erhaltenen Rückgabewert abgelesen werden.

6.1.2.4 Echtzeitfähiger Treiber zum Post-Mortem Debugging

6.1.2.4.1 Grundidee und -ablauf Beim Debuggen von Modellen, welche in eingebetteter Software ausgeführt werden, ist es wünschenswert das Modell zur Inspektion seines Verhaltens zur Laufzeit nicht anhalten zu müssen. Der Mikroprozessor ist mit seiner Umgebung durch Sensoren und Aktoren oder über ein Bussystem mit weiteren Teilsystemen verbunden.

So ist es beispielsweise vorstellbar, dass Nachrichten über einen Bus versendet werden, der gestoppte Mikrocontroller die Nachrichten jedoch nicht empfangen kann und das Modell damit in einen inkonsistenten, dem Echtzeitverhalten nicht entsprechenden Zustand geraten kann. Eine vergleichbare Problematik besteht auch bei der Verarbeitung von Sensorsignalen und auf ihnen basierenden modellbasierten Regelalgorithmen mit integrierendem oder differenzierendem Verhalten.

Mit Hilfe des Tracing-Mechanismus eines Emulatorsystems für eingebettete Zielsysteme kann dieses Problem umgangen werden. Die Grundidee besteht darin, den Ablauf einer Ausführung des Modells in Echtzeit so aufzuzeichnen, dass der Ablauf im Anschluss an die Ausführung wieder rekonstruiert werden kann. Dazu soll durch geeignete Ansteuerung des Emulatorsystems der Programmfluss aufgezeichnet und diese Information dem *ModelScope*-Framework so zur Verfügung gestellt werden, dass dieser auf ihm über die bereits beschriebenen Debugger-Schnittstellen auf dieser Aufzeichnung navigieren und Werte zu einem virtuellen Zeitpunkt auslesen kann.

Neben der klassischen Trace-Funktionalität können die benötigten Daten über die Profiling-Funktion von *iSystem.connect* gewonnen werden. Dessen Rohdaten umfassen Einsprung- und Aussprungereignisse in Methoden im Quelltext und Ereignisse zu Schreibzugriffen auf Variablen. Jedes Ereignis ist mit einem Zeitstempel versehen.

Da der Emulator nur eine begrenzte Bandbreite und begrenzten Speicher aufweist, müssen die zu protokollierenden Ereignisse vor der Aufzeichnung festgelegt werden. Während der Aufzeichnung selbst, also während der Modellausführung, können keine weiteren Überwachungsaufgaben hinzugefügt werden. Daher müssen *ModelScope* alle zur Laufzeit interessierenden Variablenzugriffe

und Methodenaufrufe vor der Ausführung bekannt sein, für welche Ereignisse aufgezeichnet werden. Es wird also ein Beobachter-Entwurfsmuster angewandt. Dies steht im Gegensatz zu dem in den vorigen Abschnitten beschriebenen Vorgehen, bei dem ein *Polling*, also ein aktives Abfragen zur Laufzeit implementiert wird.

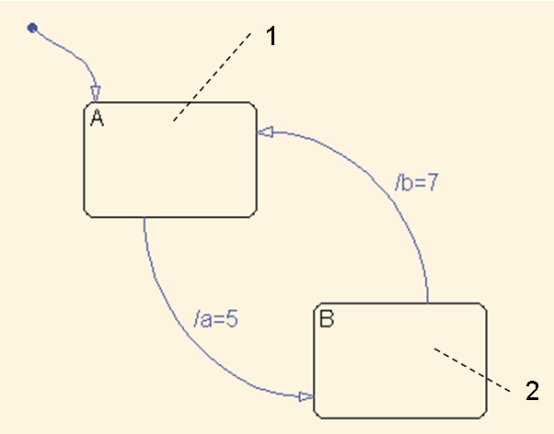

Abbildung 6.7: Ein einfaches StateFlow-Diagramm zur Erläuterung des Konzepts für den echtzeitfähigen Treiber

Abbildung 6.7 zeigt ein einfaches StateFlow-Diagramm, an welchem das Konzept des echtzeitfähigen Treibers erläutert werden soll. Die Funktionalität das *Mapping* für diese Modellierungsdomäne ist in Abschnitt 7.1 ausführlich dargestellt. Entscheidend für den echtzeitfähigen Treiber ist, dass die Zustandsinformationen in einem Satz globaler Variablen abgelegt sind und in jedem Zeitschritt die Methode *MdlOutputs()* die Auswertung desselben durchführt. Im Beispiel ist der Zustand durch eine Variable codiert, welche für den Zustand *A* den Wert 1 und für den Zustand B den Wert 2 annehmen kann.

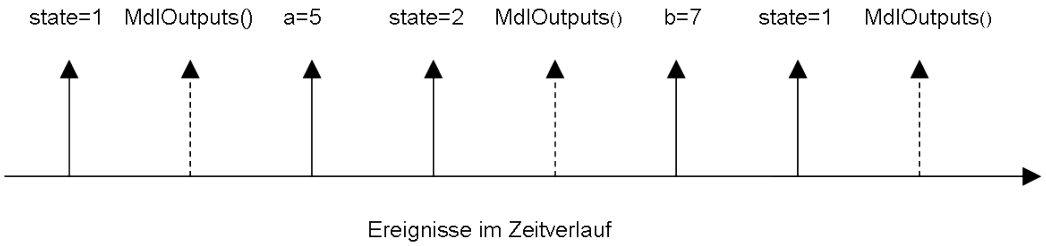

Abbildung 6.8: Abfolge von Schreibzugriffen auf Variablen (durchgehende Pfeile) und Methodenaufrufen (gestrichelte Pfeile) entlang der Zeitlinie

Instrumentiert man den Profiler so, dass er Schreibzugriffe auf die Zustandsvariablen des Diagramms und die Aufrufe der Methode *MdlOutputs()* auf-

zeichnet, erhält man eine Abfolge von Ereignissen. Abbildung 6.8 zeigt eine Aufzeichnung für das Diagramm in Abbildung 6.7. Hierbei sind die Schreibzugriffe auf Variablen als durchgezogene Pfeile dargestellt, das Betreten einer Methode ist als gestrichelter Pfeil notiert. Die absoluten Zeitpunkte der Ereignisse entsprechen nicht dem tatsächlichen Auftreten, verdeutlichen jedoch das grundlegende Vorgehen.

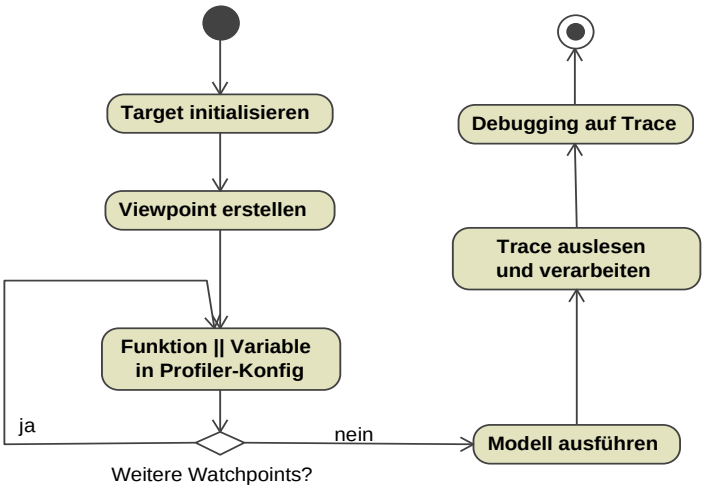

Abbildung 6.9: Aktivitäten bei Post-Mortem Debugging

Abbildung 6.9 zeigt das sich aus diesen Überlegungen ergebende grundlegende Vorgehen in einer Debugging-Sitzung. Der Nutzer lädt ein Modell in *ModelScope* und verbindet sich mit einem *DebugTarget*, hier über den echtzeitfähigen Treiber. Der Treiber baut die Verbindung auf und erzeugt eine Konfiguration für den Profiler.

Zu dieser müssen die Variablen und Methoden hinzugefügt werden, deren Ereignisse protokolliert werden sollen. Diese Daten sind jedoch vom verwendeten *Viewpoint* abhängig. Ein geeigneter *Viewpoint* wird vom Benutzer, wie in Kapitel 5 beschrieben hinzugefügt. Ist dieser für echtzeitfähiges *Post-Mortem*-Debugging geeignet, fügt er bei seiner Initialisierung die von ihm benötigten Variablen und Methoden über eine spezielle Treiberschnittstelle zur Profiler-Konfiguration hinzu. Dies kann für beliebig viele *Viewpoints* geschehen, solange die technische Bandbreite und der verfügbare Speicherplatz nicht überschritten werden.

Der Benutzer stößt nun die Ausführung des Modells im Emulator an. Anhand einer definierten Startbedingung nimmt der Profiler seine Arbeit auf und stoppt den Emulator bei Erreichen einer Zielbedingung. Da die Möglichkeiten für diese Bedingungen stark von den Fähigkeiten des verwendeten Emulator-

Listing 6.4: Ausschnitt aus Ablaufaufzeichnung

```
name ; time ; value ; type
rtDWork.SFunction_ChartInstance.State.is_m0_c1_controller_a...
    ...;215883;0;VARIABLE
rtDWork.SFunction_ChartInstance.State.is_m0_c1_s5_On1;1157383;0;...
    ...VARIABLE
rtDWork.SFunction_ChartInstance.State.is_m0_c1_s10_On2...
    ...;1159883;0;VARIABLE
rtDWork.SFunction_ChartInstance.State.is_0_c1_s14_Mi;1161766;0;...
    ...VARIABLE
MdlOutputs;1344383;-2147483648;FUNCTION
rtDWork.SFunction_ChartInstance.State.is_m0_c1_controller_a...
    ...;1847016;1;VARIABLE
MdlOutputs;2196633;-2147483648;FUNCTION
rtDWork.SFunction_ChartInstance.State.is_m0_c1_controller_a...
    ...;2794766;0;VARIABLE
rtDWork.SFunction_ChartInstance.State.is_m0_c1_controller_a...
    ...;2806266;2;VARIABLE
```

systems abhängen, wurde in der Betrachtung für diese Arbeit keine Verbindung zwischen den *Triggern* zum Starten und Stoppen und dem eigentlichen Modell hergestellt. Stattdessen werden diese textuell und systemabhängig definiert. Die Aufzeichnung stellt den Teil des Ablaufs dar, welcher harten Echtzeitrandbedingungen unterliegt. Diese werden vom Emulatorsystem garantiert.

Anschließend liest der Treiber die aufgetretenen Ereignisse aus dem Speicher des Emulators und bereitet die aufgezeichneten Spuren für die Verwendung in *ModelScope* auf. Der Trace kann dabei in eine Datei serialisiert werden. Listing 6.4 zeigt einen Ausschnitt aus einer solchen Datei. Jede Zeile entspricht einem Ereignis, welches in vier durch Semikolons getrennte Spalten unterteilt ist. Die erste Spalte benennt die instrumentierte Variable oder Methode. Darauf folgt der Zeitstempel und bei Variablen der geschriebene Wert. Die letzte Spalte gibt an, ob die Ursache des Ereignisses ein Variablenzugriff oder ein Methodenaufruf war.

6.1.2.4.2 Debugging auf der Aufzeichnung Auf den aufgezeichneten Ereignissen kann nun die eigentliche in Abschnitt 5.3.3 beschriebene Treiberschnittstelle navigieren und den Systemablauf nachträglich simulieren.

Bei Verwendung eines nicht-echtzeitfähigen Treibers wird ein Haltepunkt auf die Methode *MdlOutputs()* gesetzt, die in jedem Modellschritt ausgeführt wird. Mit jedem Erreichen des Methodenaufrufs ist ein Zeitschritt vergangen und der Zustand kann durch einen *Mapper* ausgelesen werden (vgl. Abschnitt 7.1.3).

Bei Verwendung des echtzeitfähigen Treibers wird der Zeitpunkt, an dem die Methode aufgerufen wird, als Ereignis protokolliert. Der aktuelle, virtuelle Zeitpunkt der simulierten Ausführung kann beliebig auf der Zeitleiste gesetzt werden.

Um einen Zeitschritt im Modell durchzuführen, wird auf der Aufzeichnung von einem Aufruf von *MdlOutputs()* zum nächsten Aufruf navigiert, wobei Ereignisse für Schreibzugriffe zunächst übersprungen werden. In Abbildung 6.8 entspricht dies einem Sprung von einem gestrichelten Pfeil zum nächsten.

Wenn ein Zeitschritt einfach der Navigation zum nächsten Haltepunkt entspricht, kann der aktuelle virtuelle Zeitpunkt auch wieder beliebig zurück verschoben werden. Daraus ergibt sich als Seiteneffekt dieses Vorgehens, dass ein Debugging rückwärts in der (virtuellen) Zeit problemlos realisierbar ist.

Ist im virtuellen Ablauf ein solcher „Haltepunkt“ erreicht, wird die Belegung von Variablen zu diesem Zeitpunkt abgefragt. Im Beispiel sei der letzte Aufruf von *MdlOutputs()* erreicht. Die Belegung der Variablen ist durch den letzten Schreibzugriff bestimmt. Die Variable a besitzt den Wert 5, Variable b den Wert 7.

Um die Belegung der Variablen zu ermitteln, sind prinzipiell zwei Ansätze möglich:

1. Schon bei Sprung auf den nächsten Haltepunkt werden die aktuellen Werte der Variablen zwischengespeichert.
2. Bei jedem Auslesen einer Variablen wird die Aufzeichnung ausgehend vom aktuellen Zeitpunkt rückwärts bis zum letzten Schreibzugriff auf die auszulesende Variable durchsucht und der geschriebene Wert zurückgegeben.

In der realisierten Variante wurde der zweite Ansatz gewählt, welcher bei häufig geschriebenen Variablen kaum Einbußen in der Ablaufgeschwindigkeit bedeutet, jedoch eine deutlich einfachere Realisierung, speziell bei mehreren *Viewpoints* bedeutet.

6.1.2.4.3 Realisierung Die Komponente des Treibers, welche für die Wiedergabe zuständig ist, realisiert mehrere Schnittstellen, welche den Ablauf nach Beendigung der Aufzeichnung simulieren.

Um die zusätzlichen Funktionalitäten zum umkehrbaren Navigieren entlang der virtuellen Zeitachse und zur Konfiguration des *Trace*-Mechanismus anzubieten, wurde das Treibermodell um zwei weitere Schnittstellen erweitert. Diese sind in Abbildung 6.10 dargestellt.

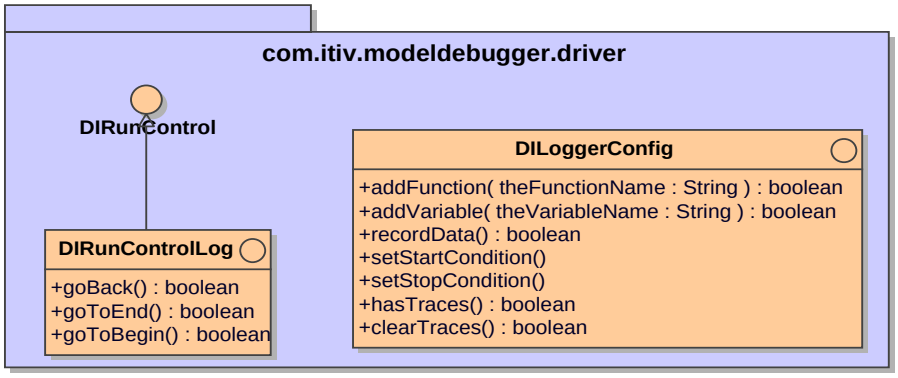

Abbildung 6.10: Zusätzliche Schnittstellen der Treiberschicht von ModelScope für echtzeitfähiges Debugging.

Durch den Treiber selbst werden folgende Schnittstellen implementiert:

- *DIBreakpoint*: Diese Schnittstelle wird durch die Klasse *DISysLogBreakpoint* umgesetzt. Für die Aufzeichnung wird der an *setAtMethod()* übergebene Methodenname in die Liste der zu protokollierenden Funktionsaufrufe eingefügt. Die Methoden zum Löschen von Haltepunkten entfernen diese wieder aus der Konfiguration des Profilers.

- *DIRunControl*: Die Liste der Ereignisse erlaubt neben Einzelschritten vorwärts in der Zeit auch Schritte rückwärts. Um dies der Benutzerschnittstelle von *ModelScope* anzubieten, erweitert eine Schnittstelle *DIRunControlLog*, welche *DIRunControl* um die Methoden *goBack()*, *goToBegin()*, *goToEnd()* erweitert. Die Schnittstelle wird durch *DISysRunControl* realisiert. Die Operationen *step(), stepOut()* und *stepOver()* sind auf Quelltextebene nicht realisiert, da der Profiler nur das Betreten und Verlassen von Methoden aufzeichnen kann und somit einzelne Schritte nicht verfügbar sind.

- *DISymbolAccess*: Der Treiber zum Symbolzugriff und seine zentrale Operation *evaluate()* sind durch *DISysLogSymbolAccess* realisiert, bieten aber nur vereinfachte Funktionalität. So können nur Zahlenwerte gelesen werden. Der Zugriff auf zusammengesetzte Datentypen, Objekte und über allgemeine zusammengesetzte Ausdrücke ist nicht möglich.

- *DILoggerConfig*: Die Schnittstelle *DILoggerConfig* abstrahiert die spezifische Ansteuerung tracefähiger Treiber. Dem Trace-Mechanismus werden vor Modellausführung die zu überwachenden Methoden und Variablen bekannt gemacht. Dies geschieht durch die Operationen *addFunc-*

tion() und *addVariable()*. Über *setStartCondition()* und *setStopCondition()* können die zielsystemspezifischen Start- und Endbedingungen gesetzt werden. Über *recordData()* wird eine Aufzeichnung gestartet und über *hasTraces()* kann geprüft werden ob Ereignisse aufgezeichnet wurden. *clearTrace()* löscht schließlich die Aufzeichnung. *DISysLoggerConfig* realisiert diese Schnittstelle vollständig für *iSystem.connect*.

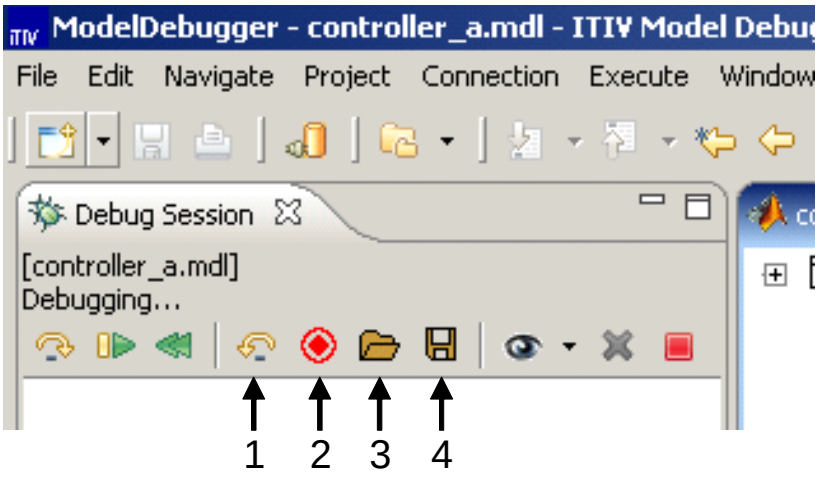

Abbildung 6.11: Erweiterung der Benutzeroberfläche durch Trace-Mechanismus

Der erweiterte Funktionsumfang dieses Treibers und der komplexere in Abbildung 6.9 gezeigte Arbeitsfluss führen dazu, dass auch die Oberfläche, speziell das *Debug Session*-Fenster neu hinzugekommene Schaltflächen aufweist.

Diese sind in Abbildung 6.11 dargestellt. Zusätzlich zu den geläufigen Operationen *Einzelschritt*, *Start, Reset* auf der linken Seite und den Operationen zum Hinzufügen und Löschen von *Viewpoints,* sowie dem Abbrechen des Debug-Vorgangs auf der rechten Seite, befinden sich in der Mitte vier neue Aktionen (von links nach rechts):

1. Einzelschritt rückwärts
2. Aufzeichnung starten
3. Aufzeichnung aus Datei laden
4. Aufzeichnung in Datei speichern

Die Aktionen 1 und 4 sind nur anwählbar, wenn bereits eine Aufzeichnung vorliegt.

6.2 Rekonfigurierbare Logikbausteine

Neben reinen prozessorartigen von-Neumann- oder Harvard-Architekturen, wie sie eingebettete Mikrocontroller oder Digitale Signalprozessoren aufweisen, können Modelle auch auf programmierbaren Bausteinen abweichender Architektur ausgeführt werden.

Durch Konfiguration primär während der Initialisierungsphase, jedoch auch durch Rekonfiguration während der Laufzeit, kann die grundlegende Topologie rekonfigurierbarer Bausteine so konfiguriert und verknüpft werden, dass verschiedenartigste Schaltungen realisierbar sind.

Im Gegensatz zur sequentiellen kontrollflussgetriebenen Verarbeitung von Instruktionen fokussieren rekonfigurierbare Architekturen auf den Datenfluss. Dieser Wechsel des Paradigmas ermöglicht Parallelisierung und bietet speziell für datenintensive Anwendungen eine erhöhte Bandbreite bei konstanter Taktfrequenz und gleicher Fläche.

Gegenüber anwendungsspezifischen integrierten Schaltungen (engl. *application specific integrated circuit*, ASIC) ermöglichen rekonfigurierbare Bausteine eine schnellere und preisgünstigere Erstellung von Prototypen und werden speziell bei kleineren und mittleren Stückzahlen in Produkten verbaut. Nachteilig gegenüber ASICs sind jedoch die geringere Logikdichte und die deutlich geringeren maximalen Taktraten, welche durch die zusätzliche Konfigurationslogik bedingt sind.

Eine weit verbreitete Klasse solcher Bausteine stellen *Field Programmable Gate Arrays* (FPGA) dar. FPGAs sind feingranular konfigurierbare Bausteine, welche als datengetriebene Co-Prozessoren in zahlreichen eingebetteten Systemen unter anderem in der Avionik, im Mobilfunk, im Automobilbau und für medizintechnische Anwendungen eingesetzt werden.

Ihre programmierbaren Funktionseinheiten enthalten logische Komponenten auf Basis einer Pfadbreite von einem Bit, welche grundlegende logische boolesche Operationen abbilden können. Diese Funktionen werden meist durch *Look-up-Tables* (LUT) und zusätzliche Speicherelemente (Flip-Flops) realisiert und können durch ebenfalls konfigurierbare Verbindungsstrukturen miteinander verknüpft und zu größeren Funktionseinheiten verschaltet werden. Spezielle Bausteine vereinigen diese Architektur mit eingebetteten Prozessoren, RAM und weiterer Peripherie.

Eine mögliche und weit verbreitete Beschreibungssprache zur Spezifikation von Funktionalität stellt die *Very High Speed Integrated Circuit Hardware Description Language* (VHDL) dar. VHDL ermöglicht die Beschreibung von Archi-

tektur und Verhalten einer logischen Schaltung auf der Abstraktionsebene von Modulen, Signalen und arithmetischen Operationen.

In den 80er Jahren wurde VHDL zunächst als Sprache entwickelt und später standardisiert (siehe [Institute of Electrical and Electronics Engineers 1988]), um durch funktionale textliche Modellierung die Simulation und formale Verifikation von komplexen Logikschaltkreisen zu ermöglichen. Im weiteren Verlauf wurden synthesefähige Sprachteile identifiziert, so dass aus einem Subset von VHDL automatisiert Netzlisten und im weiteren Verlauf Konfigurationsbitströme für FPGAs oder Masken zur Produktion von ASICs erzeugt werden können. VHDL kann also, ähnlich wie die UML ausführbar gemacht und als Programmiersprache genutzt werden.

Aus Sicht des Anwendungs-/Funktionsentwicklers stehen sich die Beschreibungssprachen und -paradigmen für den Entwurf von FPGAs und ASICs wie beschrieben zunächst sehr nahe, es findet also *Hardware*-Entwurf statt. Andererseits ist das Produkt dieses Vorgehens ein Bitstrom oder eine andere geeignete Konfiguration, welche auf einer existierenden Hardware (dem konfigurierbaren Baustein) ausgeführt wird. Zudem sind Iterationen sehr kurz, wodurch agile Entwurfsprozesse aus dem *Software*-Entwurf angewandt werden können. Aufgrund dieser dualistischen Natur wurde für Prozesse und Ergebnis konfigurierbaren Entwurfs der Begriff *Configware* geprägt [Hartenstein 2007].

6.2.1 Graphische Modellierung für konfigurierbare Bausteine

Folgt man der Interpretation von Configware als Software für eine spezielle Ausführungseinheit, bietet es sich an, auch hier die in Kapitel 2 diskutierten Technologien graphischer Modellierung anzuwenden.

Dabei sind prinzipiell zwei Vorgehensweisen möglich:

1. Von existierenden Hardware-Beschreibungssprachen (z.B. VHDL) und den beschriebenen Problemdomänen wird auf eine eigene domänenspezifische Modellierungssprache abstrahiert.

2. Existierende graphische Modellierungssprachen, welche sich für den Entwurf von Software bewährt haben, werden auf Configware angewendet.

Unter die erste Kategorie fallen Ansätze einer graphischen Darstellung von VHDL-Konstrukten. Dabei wird unter Beibehaltung der Abstraktionsebene eine graphische Syntax eingeführt, welche die Verständlichkeit und Übersichtlichkeit verbessern soll. Am geläufigsten sind dabei modulare Darstellungen

für komplexe, aus mehreren VHDL-Modulen zusammengesetzte Gesamtsysteme. Zahlreiche moderne Entwicklungsumgebungen bieten diese Möglichkeit zur Anordnung von Modulen. Andere akademische Ansätze erweitern die Modellierung um Zustände und Abläufe (siehe z.B. [Hadlich 1997]).

Im Rahmen des von der Deutschen Forschungsgemeinschaft im Schwerpunktprogramm „Rekonfigurierbare Rechensysteme“ geförderten Projekts „Aladyn“ wurde für eine dynamisch zur Laufzeit rekonfigurierbare Systemarchitektur untersucht, wie unter Nutzung existierender Modellierungssprachen und -werkzeuge eine modellbasierte Entwurfskette umgesetzt werden kann (siehe auch [Graf u. a. 2005]). Vorteil dieses unter die zweite Kategorie fallenden Vorgehens ist, dass existierende Modellierungsparadigmen verwendet werden können. Weiterhin kann sich das Modell auf einer beliebig höheren Abstraktionsebene als die textuelle Darstellung befinden und erst später im Sinne der in Abschnitt 3.1.2 vorgestellten MDA auf synthetisierbare Artefakte abgebildet werden. Im Rahmen von „Aladyn“ wurde gezeigt, dass Stateflow genutzt werden kann, um Teile des Verhaltens von Soft- und Configware-Implementierung in einem Modell zu bündeln.

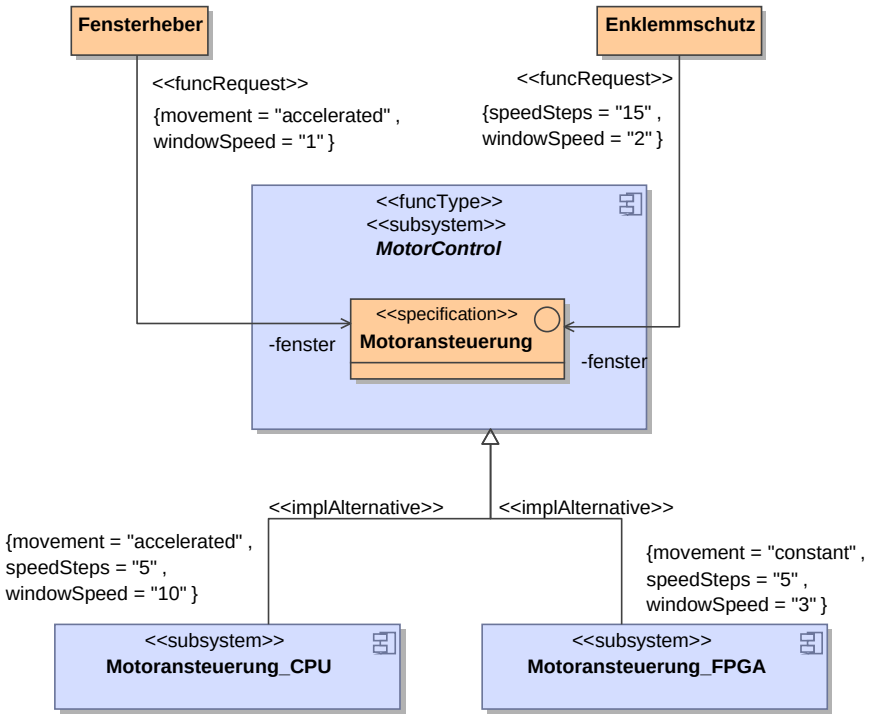

Abbildung 6.12: Top-Level Modell eines hardwarebeschleunigten Fensterhebersystems

Abbildung 6.12 zeigt das UML-Systemmodell einer Gesamtapplikation für die

dynamisch rekonfigurierbare Architektur. Der FPGA agiert dabei als Baustein zur Beschleunigung einzelner Funktionen. Die Abbildung zeigt oben beispielhaft zwei Klassen, welche unterschiedliche Anwender der Funktion „Fensterheber“ repräsentieren. Dabei enthält das mit dem *«funcType»*-Stereotyp versehene Subsystem Fensterheber nur Schnittstellen. Diese können durch die Realisierungsalternativen *Fensterheber_CPU* und *Fensterheber_FPGA* im unteren Bereich der Abbildung für einen Prozessor und einen FPGA implementiert werden. Das Verhalten der Realisierungsalternativen kann mit Hilfe von Verhaltensmodellen der UML (Aktivitätsdiagramme, Aktionsmodelle, Zustandsmaschinen) oder Simulink/Stateflow-Diagrammen spezifiziert werden. Welches Modellierungswerkzeug dabei eingesetzt wird, ist auch davon abhängig, ob geeignete Codegeneratoren für die Ziel-Hardware-Architektur verfügbar sind, also ob das Teilmodell auf der angepeilten Zielplattform *ausführbar* ist.

Das in Abbildung 6.12 gezeigte Klassendiagramm ist dabei um ein UML-Profil erweitert, welches Stereotypen und TaggedValues definiert, mit welchen der dynamisch rekonfigurierbare Mechanismus konfiguriert werden kann. Neben funcType für abstrakte Funktionstypen, können Implementierungsalternativen (Stereotyp <<implAlternative>> auf Subsystemen) und ihre Qualitätseigenschaften als TaggedValues definiert werden. Einer Klasse aus der softwarebasierten Applikation wird der Zugriff auf die Funktion ermöglicht, indem eine mit dem Stereotyp <<funcRequest>> versehene Assoziation zwischen Nutzerklasse und einer als Service-Anbieter agierenden Schnittstelle des abstrakten Subsystems modelliert wird. Die benötigten Dienstgütemerkmale werden als TaggedValues annotiert.

Abbildung 6.13 zeigt ein detailliertes systemweites Profil. Das Diagramm zeigt die schon beschriebenen Stereotypen und ihre Verknüpfung mit den erweiterten Metaklassen. Weiterhin zeigt das Diagramm einen abstrakten Stereotypen <<attributeTypes>> welcher die Güteattribute bündelt, die zur Beschreibung der Gütemerkmale in den Realisierungsalternativen und zur Beschreibung der gewünschten Ausprägungen in den Funktionsanforderungen dienen.

Ausgehend von der Modellierung muss der Schritt vom Modell zu einer synthesefähigen Beschreibung (z.B. VHDL) automatisiert werden, will man ein ausführbares Modell erhalten.

Abbildung 6.14 zeigt die Werkzeugkette, welche zur Erzeugung von Prototypen genutzt wurde. Zur Funktionsdefinition wird Matlab Simulink/Stateflow gewählt. Diese Entscheidung wurde von den verfügbaren Generatoren getrieben, welche die Erzeugung von C-Code und synthetisierbarem VHDL ermöglichen. Statecharts können mittels des Werkzeugs VHDLGen (siehe [Dreier u. a. 2003b]) übersetzt werden.

Die Ablaufsteuerung und dazugehörige Scheduling-Strategien werden mit der

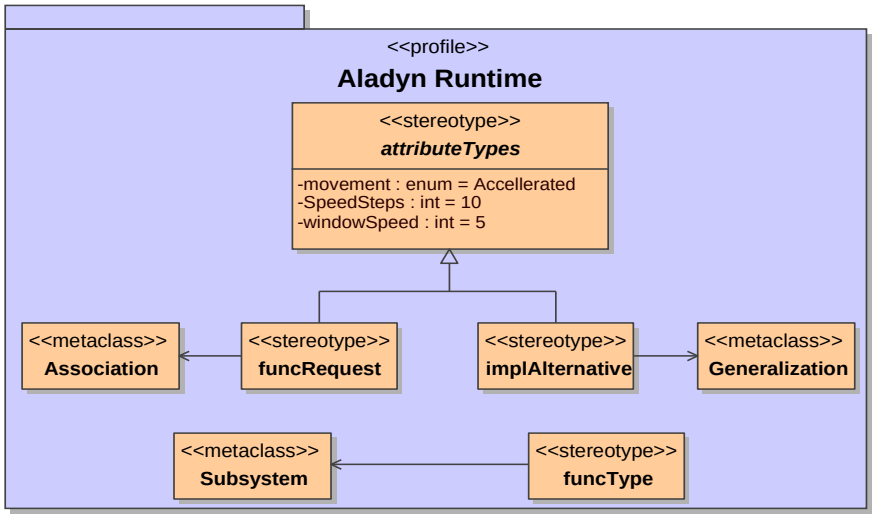

Abbildung 6.13: UML Profil zur Annotation des Systemmodells im Projekt Aladyn [Graf u. a. 2005]

Unified Modeling Language (UML) in Klassendiagrammen und mit Hilfe einer Aktionssprache spezifiziert und mit Hilfe des Aquintos.GS Codegenerators erzeugt. VHDL-Modelle werden um die in Abschnitt 6.2.2 eingehend beschriebene Debugging-Schnittstelle erweitert.

Im Backend wurde die Xilinx ISE Werkzeugkette[4] zur Erzeugung von Bitströmen genutzt. Die frei verfügbaren GNU Werkzeuge erzeugen Binärcode für die eingesetzten Microblaze Softcore-Prozessoren[5]. Aus den erhaltenen Bitströmen werden Slots extrahiert und komprimiert und ein Bitstream für die komplette Konfiguration erzeugt. Dieser wird auf unserem FPGA-basierenden Zielsystem ausgeführt.

Der beschriebene Ansatz erlaubt derzeit eine vollautomatische Ausführung eines weitestgehend modellierten Systems. Die Automatisierung erfolgt über Skripte oder mit Hilfe der in Abschnitt 5.1.2 beschriebenen Projektverwaltung.

6.2.2 Generierbare Schnittstelle für Modelle

Aus graphischen Modellen erzeugter VHDL-Code ist so optimiert, dass er keine gesonderten Ausgabemöglichkeiten für die Werte interner Register besitzt.

[4]Xilinx ISE Webseite:
http://www.xilinx.com/products/design_resources/design_tool/index.htm

[5]Xilinx Microblaze Webseite:
http://www.xilinx.com/products/design_resources/proc_central/microblaze.htm

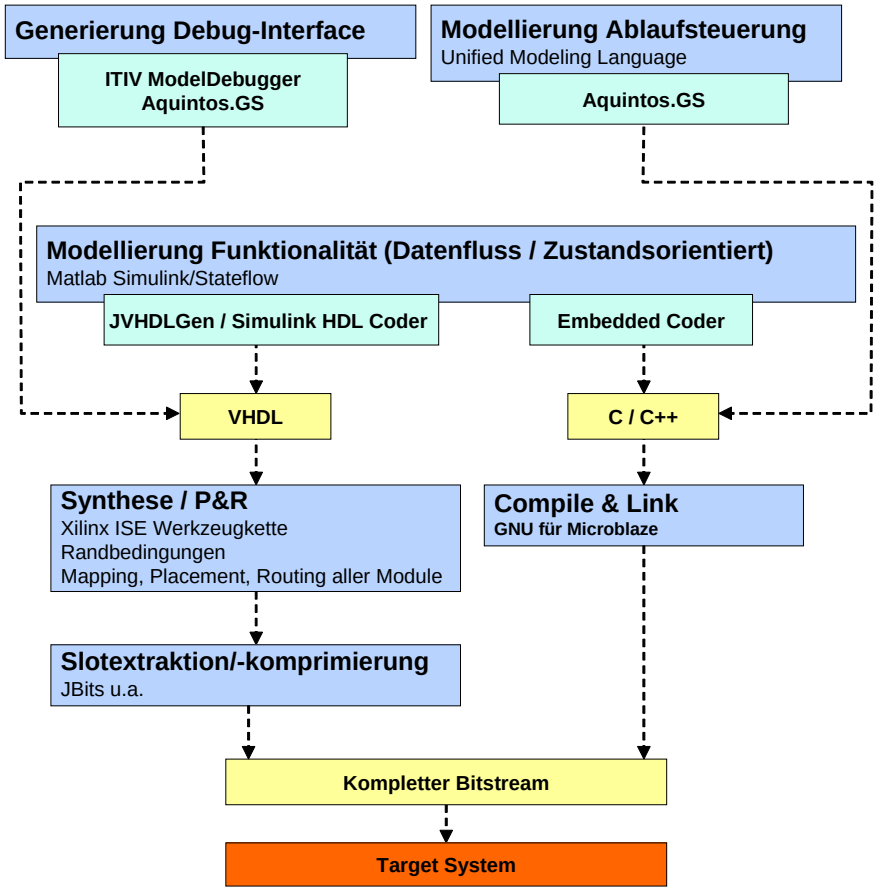

Abbildung 6.14: Modellgetriebene Werkzeugkette im Hardware/Software-Codesign

Instrumentiert man diesen VHDL-Code also so, dass er eine Schnittstelle besitzt, über welche zur Laufzeit Informationen gelesen werden können, ist nur eine Inspektion der Ein- und Ausgabeports möglich. Dies entspräche nur einem *Black-Box* Ansatz, wünschenswert ist jedoch der Zugriff auf zusätzliche Interna.

Abbildung 6.15 stellt das Gesamtsystem dar. Im oberen Bereich ist der Entwicklungsrechner skizziert, welcher ein Debugging für auf einem FPGA ausgeführte Modelle unterstützen soll.

In der vorliegenden Arbeit wurde die Gangbarkeit eines solchen Ansatzes für hierarchische Zustandsautomaten, genauer *StateFlow*-Diagramme untersucht. Ein solches Modell kann über die in Abschnitt 2.3.3.1 vorgestellten Codegeneratoren in compilierbaren C-Code oder synthetisierbaren VHDL-Code umgewandelt werden. Der VHDL-Code erzeugende Generator wurde so erweitert, dass das erzeugte Funktionsmodul auch alle internen Register, welche

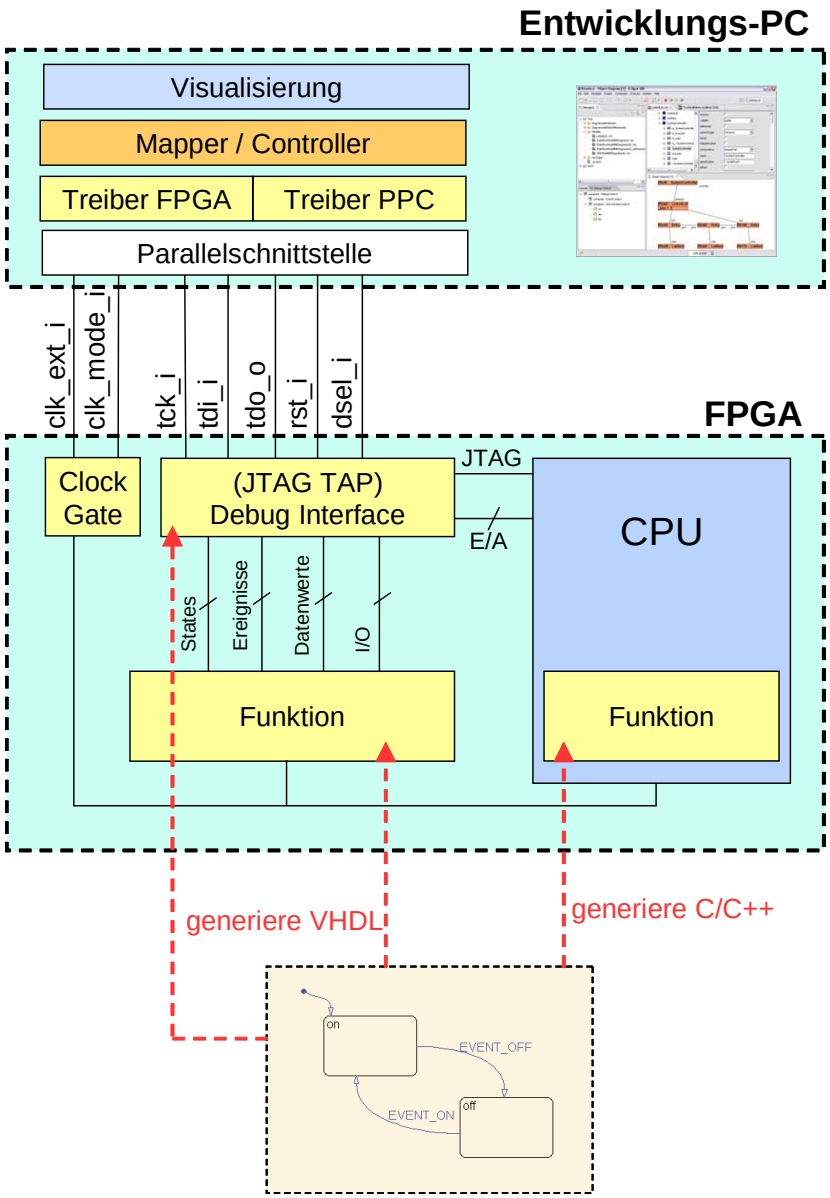

Abbildung 6.15: Blockdiagramm der Schnittstelle im FPGA, der Verbindung mit dem Entwicklungsrechner und der aus dem Modell generierten Elemente.

den Zustand, den Datenkontext und Ereignisse repräsentieren über eine breite Schnittstelle nach außen zur Verfügung stellt.

Die Verbindung zwischen Entwicklungsrechner und FPGA geschieht in erster Linie über ein serielles Schieberegister. Da die im vorigen Absatz vorgestellten Konzepte oft komplex sind und ein verbreiteter Standard für das *System-on-Chip*-Debugging derzeit nicht abzusehen ist, fiel die Wahl für diese Arbeit

auf eine einfache, auf Schieberegistern basierende Lösung. Das Konzept lehnt sich an die Funktionsweise einer JTAG-Schnittstelle an, implementiert jedoch nicht die Zustandsmaschine des *Test Access Port* (TAP). Sie unterstützt *Daisy Chaining*, also das Hintereinanderschalten mehrerer Komponenten und ist voraussichtlich einfach in ein komplexeres Protokoll integrierbar.

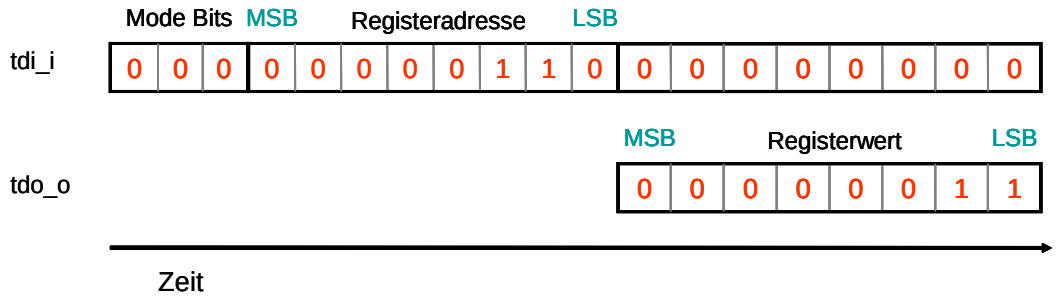

Abbildung 6.16: Mode Bits, Registeradressen und Registerwert als Schieberegister

Der Schnittstelle kann über ein Eingangssignal *tdi_i* eine Registeradresse in das FPGA geschrieben werden. Das Lesen der Schnittstelle geschieht über das Ausgangssignal *tdo_o*. Wie in Abbildung 6.16 dargestellt, werden die Registeradressen in Form eines Schieberegisters über *tdi_i* in die Debugging-Schnittstelle geschrieben. Der Registerwert wird ebenfalls als Schieberegister über *tdo_o* ausgelesen. Der über *tdo_o* erhaltene 8-bittige Wert entspricht dem Wert des internen Registers des Hardware Modells.

Zunächst werden 3 *Mode*-Bits (jedes Bit hat den Wert 0) gesetzt, welches jeweils 3 Taktzyklen beträgt. Dabei besteht ein Taktzyklus aus dem einmaligen Setzen und einmaligen Rücksetzen von *tck_i*, einem zusätzlichen Taktsignal für das Debugging. Die *Mode*-Bits sollen in Zukunft das Protokoll erweiterbar machen. So könnten damit beispielsweise unterschiedliche Modelle selektiert werden oder in Register der Funktion geschrieben werden. Derzeit ermöglichen die Bits nur das Verknüpfen mehrerer Debug-Schnittstellen.

Nach Setzen der Mode Bits, wird in 8 Taktzyklen die zu schreibende Registeradresse als Byte in die Zielarchitektur geschrieben. Gefolgt werden sie von weiteren 8 Taktzyklen, nach welchen das vom FPGA zurückgegebene Byte gelesen und zu einer Zahl rekonstruiert ist.

Ein weiteres Modul umhüllt die generierte Hardware-Funktion und übernimmt dabei die Aufbereitung der Daten aus dem generierten Modell für die serielle Schnittstelle zu *ModelScope*. Dies geschieht über Multiplexen und anschließendem Serialisieren des über *tdi_i* selektierten Registers.

Zum Auslesen des Zustands des Systems muss sein Ablauf angehalten werden können. Weiterhin soll der Debugger auch den Ablauf in Einzelschritten unterstützen. Daher wurde in die Debug-Schnittstelle eine Funktionalität zur Auswahl unterschiedlicher Taktquellen für das Funktionsmodell integriert. Über *clk_mode_i* kann vom Entwicklungsrechner aus zwischen dem internen Taktgeber des FPGAs und einer Taktung über das externe Signal *clk_ext_i* gewählt werden.

Als weitere Aufgabe übernimmt die Schnittstelle bei zukünftiger vollständiger Integration in ein JTAG-Interface das Verteilen von Anfragen auch an andere Teilsysteme, wie in den FPGA eingebettete Mikroprozessoren um so ein gemischtes Debugging von Hardware- und Software-Funktionalität zu ermöglichen.

Pin Nr. Parallel Port	Signal nach IEEE 1284	Richtung aus Sicht PC	Signal FPGA
2	Data0	Out	tdi_i
3	Data1	Out	tck_i
4	Data2	Out	clk_ext_i
5	Data3	Out	clk_mode_i
6	Data4	Out	rst_i
7	Data5	Out	dsel_i
17	Select-In	In	tdo_o

Abbildung 6.17: Abbildung der parallelen Schnittstelle auf Signale des Debug-Ports im FPGA

Abbildung 6.17 zeigt die Verknüpfung der Signale zwischen der parallelen Schnittstelle auf Seite des PC und den Signalen der Schnittstelle im FPGA. Alle Signale, welche im FPGA Eingangssignale darstellen, werden über den parallelen Datenbus der IEEE 1284-Schnittstelle realisiert. Als Eingang auf Seiten des Entwicklungs-PC fungiert das Statusbit *Select-In*.

Das Debug-Interface ist vom verwendeten Modell abhängig. So unterscheiden sich abhängig vom Gesamtmodell:

- Zustandsadressierung und -codierung in der Funktion
- Datenwertadressierung und -codierung in der Funktion
- Ereignisadressierung und -codierung in der Funktion
- Angeschlossene Funktionsmodule, für welche eine externe Taktquelle wählbar sein soll.

- Angeschlossene eigenständige Subsysteme wie eingebettete Prozessoren die in die Schnittstelle integriert werden sollen.

Um eine aufwendige und fehlerträchtige händische Erstellung und Anpassung zu vermeiden, wird auch die Debug-Schnittstelle aus dem UML-basierten Gesamtmodell generiert. Dies geschieht auf Basis der Plattform *Aquintos.GS* (siehe Abschnitt 3.1.5). Die Daten werden über den integrierten Template-Mechanismus erzeugt, so dass der Ansatz gut wartbar und erweiterbar ist.

6.2.3 Treiber in ModelScope

Der Treiber für *ModelScope* greift vom PC aus über die in Abschnitt 6.2.2 beschriebene Schnittstelle auf das FPGA zu und stellt einfache, vereinheitlichte Methoden bereit, die der Ansteuerung des FPGA dienen. Insgesamt verfügt die Treiberschicht über sämtliche Funktionen, mit denen das FPGA und seine Debugging Schnittstelle vom PC aus bedient werden können. Die Komponenten des Treibers basieren jedoch teilweise auf anderen Schnittstellenklassen als die Realisierungen für Software-Debugger und erfordern daher eigene *Mapper*. Die Unterschiede in den Schichten oberhalb des Treibers werden beispielsweise durch die unterschiedlichen *Viewpoints* für die Animation von *Stateflow*-Diagrammen deutlich (siehe Abschnitte 7.1.3 und 7.1.4).

Der Treiber selbst kennt keine Modellartefakte und bietet nur grundlegenden Möglichkeiten zur Taktsteuerung und zum Lesen von Zahlenwerten aus linear adressierbaren Registern.

Kern des Treibers ist das *Debug Target*. Die in Abbildung 6.18 gezeigte Realisierung für das FPGA-Zielsystem *FPGATarget* definiert die Operationen, die ein Verbinden des PC's zum FPGA über die parallele Schnittstelle erlauben. Dabei wird immer überprüft, ob bereits eine Verbindung zum FPGA besteht oder bereits abgebrochen ist. Das Verbinden und das Abbrechen der Verbindung werden in den Methoden *attach()* und *detach()* realisiert.

Zudem ermöglicht diese Klasse über *getInterface()* den Zugriff auf zwei weitere Klassen: *FPGARunControl* und *FPGAMemoryAccess*.

Die Treiberklasse *FPGAMemoryAccess* erfüllt den Zweck, einen Speicherzugriff auf das FPGA durchzuführen. Dies geschieht durch Schreiben einer Registeradresse in das FPGA und dem Auslesen des Ausgabewertes.

Die Treiberklasse implementiert die Schnittstelle *DIMemoryAccess*, welche nur eine Methode *readByte()* definiert und im Gegensatz zu *DISymbolAccess* keine Ausdrücke oder Symbole auswerten kann. Das Auslesen über die Schnittstelle geschieht über den zu *FPGATarget* gehörenden *DebugPortHandler*.

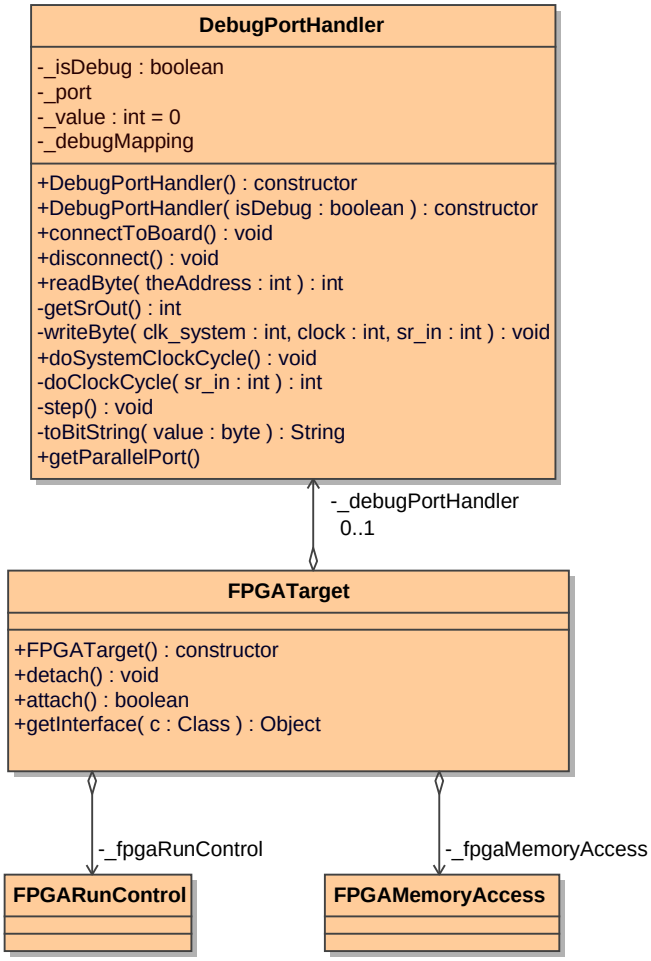

Abbildung 6.18: Grundstruktur des Treibers

FPGARunControl erlaubt es, das FPGA durch Adressieren des FPGA Clock Eingangs *clk_mode_i* zu kontrollieren. Die Treiberklasse *FPGARunControl* realisiert *DIRunControl* so, dass sie das im Softwarefall bekannte Verhalten der Schnittstelle auf die Steuerung mit Takten als atomarer Einheit der Ausführung abbildet.

Mit *clk_ext_i* wird ein einzelner Takt in das FPGA eingegeben, das FPGA reagiert auf die positive Flanke des Takts und führt eine Zustandsänderung durch. Dieses Vorgehen entspricht einem Einzelschritt über die Operation *step()*.

Abbildung 6.20 zeigt die Implementierung *FPGARunControl*, welche *DIRunControl* realisiert.

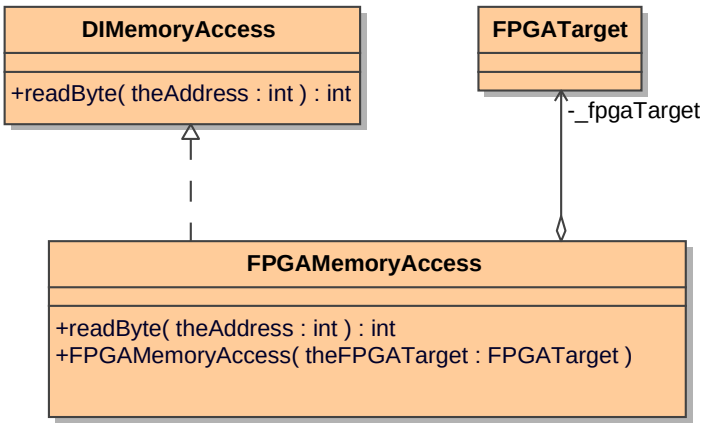

Abbildung 6.19: Klassendiagramm FPGAMemoryAccess

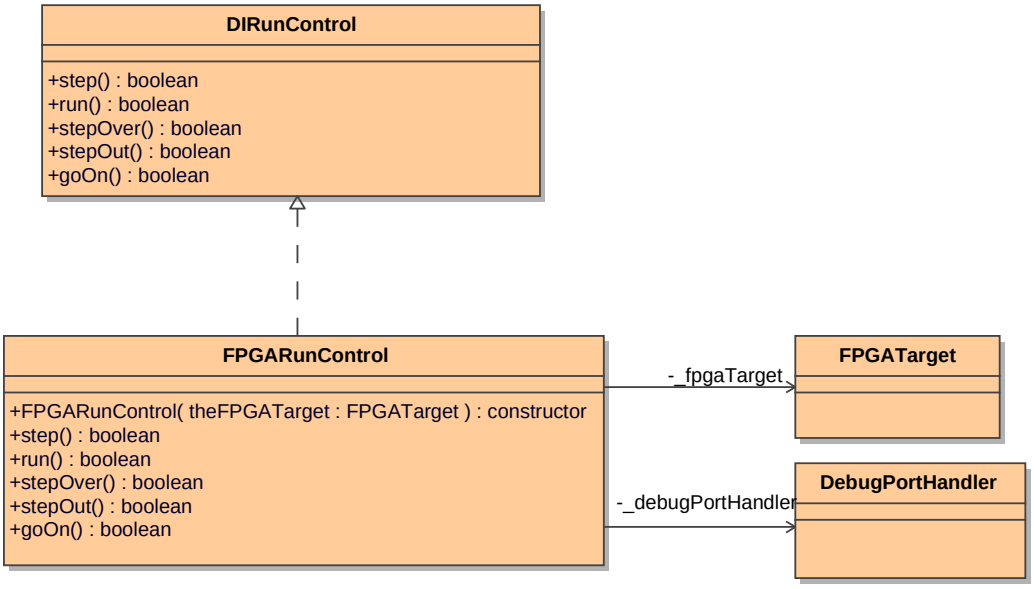

Abbildung 6.20: Klassendiagramm FPGARunControl

Kapitel 7

Systemsichten

Systemsichten stellen in der in Kapitel 5 vorgestellten Plattform zum Debuggen von Modellen die durch den Nutzer erfahrbaren Einblicke in den Zustand aber auch in vergangene und zukünftige Abläufe des Systems visuell dar. Sie kombinieren die Transformation von Daten auf die Modellebene mit Steuerung der Ausführungsplattform aus Sicht des Modells und sorgen für eine modellnahe graphische Anzeige.

In dieser Arbeit wurden drei Ansichten tiefergehend betrachtet, die unterschiedlich Modellierungsnotationen abdecken und gemeinsam sowohl Struktur- als auch Verhaltensaspekte eines ausführbaren heterogenen Modells zur Laufzeit visualisieren.

Für jede dieser Ansichten werden zunächst die Anforderungen des Benutzers des Debuggers beleuchtet, also die Frage untersucht: „Was interessiert den Entwickler eines Modells zu Laufzeit?“. Der folgende Schritt präsentiert einen Algorithmus, welcher das Laufzeitmodell erstellt und eine Ablaufsteuerung realisiert. Abschließend werden Aspekte der Visualisierung und Realisierung behandelt.

Abschnitt 7.1 befasst sich mit der Animation reaktiver hierarchischer Zustandsautomaten. Dies kann sowohl für Zustandsautomaten, aus welchen direkt Code erzeugt wurde, geschehen. Es ist jedoch auch möglich, dass ein Statechart ein übergeordnetes Verhalten modelliert, beispielsweise das Verhalten eines Kommunikationsprotokolls. Dies erfordert eine andere Art der Informationsgewinnung, welche in der vorliegenden Arbeit nicht untersucht wird.

Neben der Verhaltensbeschreibung mit Zustandsautomaten wurde in Abschnitt 3.2 das UML Action Metamodell vorgestellt. Abschnitt 7.2 beschreibt das Debugging von UML Actions auf niedriger Ebene, also ohne Abstrakti-

on zu einer *Surface Language*, und präsentiert die erhaltenen Daten in einer gemischten Baum- und HTML-Struktur.

In einer objektorientierten Software entstehen zur Laufzeit zahlreiche Objekte. Mit Hilfe des Klassenmodells, welches eine „Vorlage“ für die möglichen Objektkonfigurationen darstellt, kann ein Objektdiagramm extrahiert werden. Abschnitt 7.3 beschreibt das Vorgehen dabei und behandelt neben der Gewinnung des Datenmodells Aspekte der übersichtlichen graphischen Präsentation wie Layout und *Slicing*.

7.1 Hierarchische Zustandsautomaten

Die in Abschnitt 2.3.3 beschriebene Verwendung von Statecharts zur Verhaltensbeschreibung eingebetteter Software führt bei Nutzung dieser Beschreibungsform zur Erzeugung umfangreicher Mengen an Quelltext. Diese zeichnet sich beispielsweise durch eine komplexe Codierung des Modellzustands aus, welche teilweise aus Gründen der Speicheroptimierung auch die Hierarchisierung in zusammengesetzte Zustände aufhebt. Weiterhin werden Ereignisse und Datenwerte auf Bitfelder abgebildet, so dass das Verhalten des ausführbaren Modells zur Laufzeit nicht oder nur mühsam nachvollziehbar ist.

Der folgende Abschnitt analysiert die von einem modellbasiert entwickelnden Nutzer benötigten Informationen. Darauf aufbauend wird das Metamodell zur Speicherung von Modellinformationen so erweitert, dass durch einen geeigneten Mapper die benötigten dynamischen Informationen dargestellt werden. Das Mapping für in Software realisierte Statecharts beschreibt dann Abschnitt 7.1.3, während sich der darauf folgende Abschnitt mit der Realisierung in Hardware befasst. Abschließend wird die visuelle Realisierung und die Schnittstelle für den Nutzer des Debuggers präsentiert.

7.1.1 Anforderungsanalyse

Die bei Inspektion einer mit Stateflow modellierten hierarchischen Zustandsmaschine potentiell interessanten Daten sind:

- **Aktiver Zustand** („Welche einfachen Zustände sind gerade aktiv?“): Die primär interessierende Frage beim Debugging ist die Beobachtung des Systemzustands. Zentraler Aspekt sind bei Stateflow die jeweiligen *BasicStates*, von welchen bei einem nebenläufigen Statechart mehrere parallel aktiv sein können.

- **Ausgelöste Ereignisse** („Welche Ereignisse wurden im letzten Zeitschritt ausgelöst?“): Um das weitere Verhalten antizipieren zu können, werden Daten über ausgelöste Ereignisse benötigt, welche im kommenden Zeitschritt verarbeitet werden. Dabei können die Ereignisse sowohl extrinsische von außen kommende Signale sein, jedoch auch innerhalb der Zustandsmaschine selbst entstanden sein.

- **Datenwerte** („Welche Werte hat der Datenkontext in Stateflow?“): Neben den visuell sichtbaren Zuständen besitzen Stateflow-Diagramme einen den Systemzustand mitprägenden Datenkontext, welcher *Data Values* kapselt. Diese entsprechen semantisch Variablen einer Programmiersprache, welche über Aktionen des Diagramms manipuliert werden können. Die Inhalte dieser Datenwerte zur Laufzeit tragen zum Verständnis des Systemzustands bei.

- **Transitionen** („Welche Transitionen haben geschaltet?“): Häufig ist auch bei Kenntnis der aktiven Zustände im vorherigen Zeitschritt nicht eindeutig erkennbar, über welche Transition(en) die aktuelle Konfiguration erreicht wurde. Da Transitionen Aktionen auslösen, ist diese Information für den Beobachter des Systemablaufs jedoch wichtig. Ein Debugger sollte also rekonstruieren können, welche Transitionen im vergangenen Zeitschritt schalteten.

- ***Guards*** („Zu welchem Wahrheitswert werden die auf Transitionen notierten Bedingungen ausgewertet?“): Neben der Reaktion auf Ereignisse, entscheiden an den Transitionen notierte Ausdrücke, sogenannte *Guards* über das Schalten von Transitionen. Es ist hilfreich zu wissen, ob ein Guard im kommenden Zeitschritt wahr oder falsch sein wird.

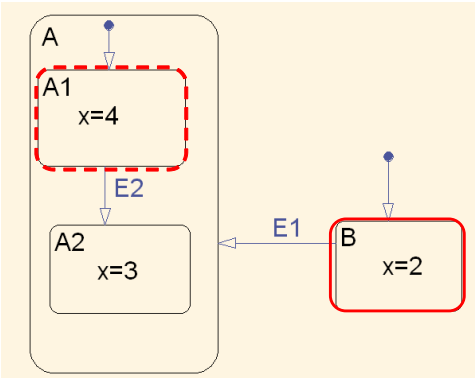

Abbildung 7.1: Beispiel für ein mit Stateflow modelliertes Statechart; durchgehend umrandet der Zustand zu t=0, gestrichtelt zu t=1.

Die interessierenden Laufzeitaspekte sollen an dem in Abbildung 7.1 gezeigten Beispiel verdeutlicht werden. Diese in Stateflow modellierte Zustandsmaschine besitzt die drei einfachen Zustände *A1*, *A2* und *B*. Zum Zeitpunkt t=0 befindet sich das modellierte reaktive System im aktiven Zustand *B*. Dem Datenwert x ist zu diesem Zeitpunkt der Wert 2 zugewiesen. Weiterhin tritt vor Ausführung des nächsten Zeitschritts das Ereignis *E1* auf. Zum Zeitpunkt t=1 wird als Reaktion auf das Ereignis *E1* der Zustandsübergang von *B* nach *A* und dort direkt in den Unterzustand *A1* ausgeführt. Der Datenwert x hat nun den Wert 4.

Die beschriebenen Abläufe können anhand von Informationen über aktiven Zustand, aktive Ereignisse, Werte von *DataValues* und geschaltete Transitionen nachvollzogen werden.

7.1.2 Metamodell

Um die erforderlichen Laufzeitdaten in das statische Modell zu integrieren, wird das Metamodell um einige Artefakte erweitert, so dass das Modell um die gewünschten Informationen angereichert werden kann. Dabei ist zu beachten, dass das gezeigte Metamodell nicht nur zur Repräsentation von Laufzeitinformationen in UML Zustandsmaschinen genutzt werden kann. Da Stateflow-Modelle, wie in Abschnitt 2.3.3.1 beschrieben, auch in einer UML-Darstellung vorliegen, ist das folgende Metamodell auch für diese geeignet.

Die in Abbildung 7.2 gezeigte Erweiterung des UML Metamodells für Zustandsmaschinen erlaubt die Erfüllung der im vorigen Absatz beschriebenen Anforderungen.

Der obere gestrichelt umrandete Bildteil umfasst die für den Debugger wichtigsten Artefakte des Metamodells der UML 2.1 (siehe auch Abbildung 2.21). Dieser Teil des Modells ist zur Laufzeit statisch.

Im unteren Teil ist der zur Laufzeit dynamische *StatemachineDebugContext* definiert, welcher Mediator für alle Informationen über eine *StateMachine* zur Laufzeit ist.

Zur Darstellung des aktiven Zustands besitzt der Kontext eine Assoziation zur Metaklasse *State*. Zu einem Zeitpunkt wird dem Kontext eine beliebige Anzahl von Zuständen, also Instanzen der Metaklasse *State*, zugeordnet, die in ihrer Gesamtheit die aktuell aktiven Zustände umfassen.

Aktuell ausgelöste Ereignisse können analog dazu über eine Assoziation zu den an Transitionen annotierten *Triggern* repräsentiert werden.

Zur Darstellung der Transitionen, die im vergangenen Zeitschritt schalteten,

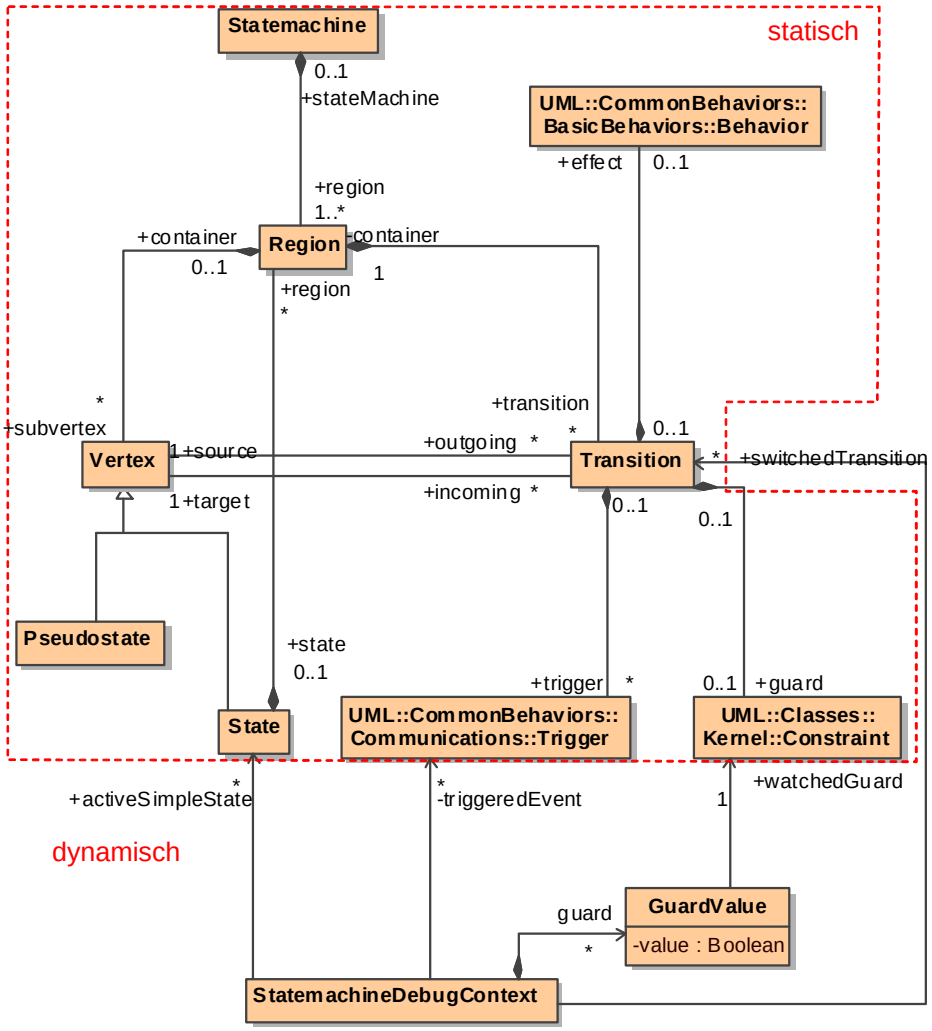

Abbildung 7.2: Erweiterung des UML Metamodells für Zustandsmaschinen um einen dynamischen Modellteil

dient eine Assoziation zur Metaklasse *Transition*. Zur Laufzeit werden dynamisch wechselnd immer genau die Transitionen durch den *StatemachineDebugContext* referenziert, welche gerade geschaltet hatten.

Zur Darstellung von Wahrheitswerten von *Guards* wird eine weitere Metaklasse *GuardValue* definiert. Diese referenziert den beobachteten *Constraint* über eine Assoziation und trägt als Attribut den derzeitigen Wahrheitswert.

Nicht im obigen Diagramm gezeigt ist die Darstellung von *DataValues*. Diese werden ausserhalb des Metamodells für Zustandsmaschinen über Instanzen der

Metaklasse *Variable* definiert und können zur Laufzeit über eine Metaklasse *DataValueValue* Werte zugewiesen bekommen.

7.1.3 Abbildung für Softwareplattform

Die Softwarerealisierung baut auf dem durch Matlab *EmbeddedCoder* und *RealTimeWorkshop* erzeugten C-Code auf, ist jedoch auch auf andere Modell-zu-Code-Transformatoren anwendbar.

Die Abbildung basiert auf folgenden Grundprinzipien:

- Die Berechnung eines einzelnen Zeitschritts im Modell geschieht durch Aufruf einer Methode, welche die Semantik des StateCharts auswertet.
- Zustandsinformation wird in einer Struktur von Variablen abgelegt. Die Variablen tragen Informationen über die Aktivität ihrer Unterzustände.
- Ereignisse und Datenwerten sind in eigenen Strukturen von Variablen abgelegt.

7.1.3.1 Controller

Aufbauend auf der Berechnung eines einzelnen Zeitschritts im Modell durch eine Methode im erzeugten Quelltext, instrumentiert der *Controller* den *Debuggee* durch einen Breakpoint zu Beginn der entsprechenden Methode.

Die Fähigkeiten des Controllers umfassen:

- Herstellen und Verwalten einer Verbindung zwischen PC und Debug-Zielsystem
- Reset: Zurücksetzen der Modellausführung in den Initialzustand.
- Starten und Anhalten des Ablaufs: Der Controller nutzt den Treiber auf Quelltextebene, um das Modell zu starten und zu stoppen.
- Einzelner Zeitschritt: Ein einzelner Zeitschritt im ereignisdiskreten Modell entspricht dann genau dem Setzen des Haltepunkts und dem Fortsetzen bis zu diesem. Weitere Einzelschritte können durch einfaches Fortsetzen des Programmablaufs bis zum erneuten Erreichen des Breakpoints durchgeführt werden.

Der Controller ist weiterhin für die Nutzung mit einem Trace-fähigen Zielsystem vorbereitet. Dies geschieht wie in Abschnitt 6.1.2.4 beschrieben durch einen speziellen Treiber welcher Instrumentierung, Aufzeichnung und Post-Mortem-Analyse anbietet.

7.1.3.2 Mapper

Der Mapper synchronisiert durch Iteration durch das statische Modell und unter Zuhilfenahme dynamischer Laufzeitinformationen das in Abschnitt 7.1.2 gezeigte Metamodell mit dem tatsächlichen Zielsystem. Dabei müssen die in Abschnitt 7.1.1 vorgestellten Informationen gewonnen werden.

Im Folgenden wird beispielhaft die Gewinnung des aktiven Zustands untersucht. Dazu soll zunächst der Zusammenhang zwischen Modell und Quelltext anhand eines einfachen Beispiels verdeutlicht werden.

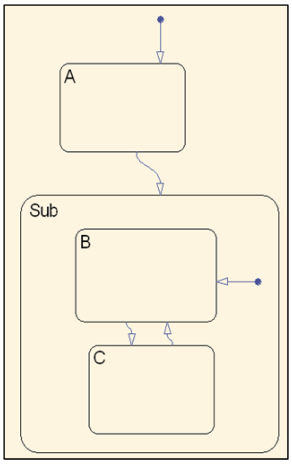

Abbildung 7.3: Exemplarisches Stateflow-Diagramm

Abbildung 7.3 zeigt ein einfaches Stateflow-Diagramm. Die von Real-Time-Workshop erzeugten Variablen im C-Quelltext zeigt das folgende Listing:

Listing 7.1: Von Real-Time-Workshop erzeugte Variablen

```
unsigned int is_m0_c1_beispiel_stateMapper : 2;
unsigned int is_m0_c1_s2_Sub : 2;
```

Jedem zusammengesetzten XOR-Zustand wird eine Variable zugewiesen, die beschreibt, welcher untergeordnete Zustand aktiv ist. Die Codierung der Variablennamen folgt einem festen Schema. Sie setzt sich aus dem Präfix *is_m0_*, der Nummerierung der in ein Simulink-Modell eingebetteten

Stateflow-Diagramme und dem Namen des Zustands im Modell zusammen und kann so aus Informationen aus dem Modell rekonstruiert werden.

Im Beispiel existieren zwei Variablen: Eine für den unsichtbaren XOR-State, welcher das gesamte Diagramm repräsentiert und eine weitere für den Zustand *Sub*.

Listing 7.2: Konstanten zur Zuordnung Variablenwert zu aktivem Zustand

```
#define IN_m0_c1_s1_A      1
#define IN_m0_c1_s2_Sub    2
#define IN_m0_c1_s3_B      1
#define IN_m0_c1_s4_C      2
```

Listing 7.2 zeigt die Konstanten im erzeugten Quelltext, welche einen numerischen Variablenwert einem Zustand im Modell zuordnen können. Jeder Zustand erhält eine eigene Enumeration seiner Unterzustände, welche alphabetisch sortiert aufgezählt werden.

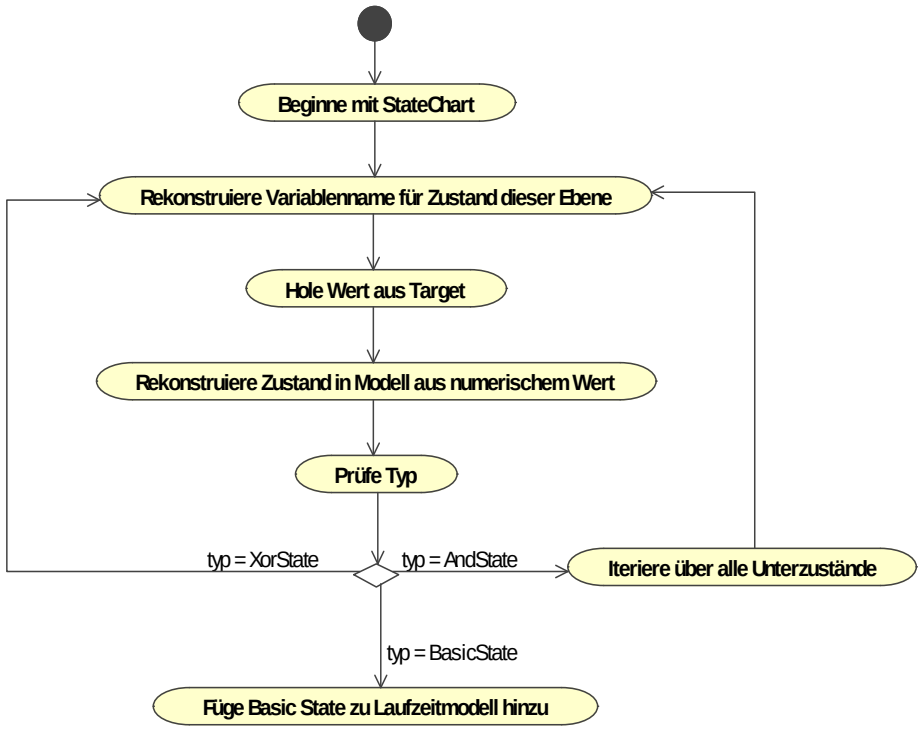

Abbildung 7.4: Algorithmus Mapping des aktiven Zustands im Aktivitätsdiagramm

Mit diesem Wissen über diese Zusammenhänge zwischen Modell und Quelltext,

kann ein Algorithmus zur Gewinnung von Laufzeitdaten über aktive Zustände gewonnen werden. Dieser ist in Abbildung 7.4 dargestellt.

Ausgegangen wird vom kompletten Statechart als XOR-State. Für diesen Zustand wird, wie beschrieben, der Variablenname im Code rekonstruiert und mit Hilfe des Debug-Treibers der zugehörige numerische Wert der Variable aus dem Zielsystem gelesen. Aus dem numerischen Wert kann nun eine Referenz auf ein Artefakt im UML-Modell gewonnen werden. Dieses Modellelement wird auf seinen Typ geprüft. Falls ein einfacher Zustand vorliegt, ist der aktive einfache Zustand gefunden und wird dem *StatemachineDebugContext* hinzugefügt. Im Falle eine XOR-States wird rekursiv weiter in der Hierarchie hinabgestiegen bis ein einfacher Zustand gefunden wird. Ein AND-State iteriert über alle sein Unterzustände und behandelt jeden wie einen XOR-Zustand. Nur durch AND-States kann mehr als ein aktiver einfacher Zustand auftreten.

7.1.4 Abbildung für Hardwareplattform

Die Hardwareimplementierung basiert auf der in Abschnitt 6.2 beschriebenen Schnittstelle für mit Hilfe von JVHDLGen in synthetisierbares VHDL umgewandelte Stateflow-Diagramme. Mittels des in Abschnitt 6.2.3 vorgestellten Treibers für die Schnittstelle können die instrumentierten Register gelesen und der Ablauf über eine Taktsteuerung beeinflusst werden. Die unterstützten Schnittstellen dieser Treiber entsprechen nicht denen der Software-Debugger, so dass sich die Abbildung in das Laufzeitmodell signifikant unterschiedlich gestaltet.

7.1.4.1 Controller

Der Controller steuert den FPGA auf Modellebene. Seine Schnittstellen erlauben das Starten und Stoppen sowie das Durchlaufen in Einzelschritten. Betrachtet man die Struktur des FPGA und den Stand der realisierten Debug-Schnittstelle, sind der Steuerung des FPGA enge Grenzen gesetzt.

Der minimale Satz an Fähigkeiten des umgesetzten Controllers umfasst:

- Herstellen und Verwalten einer Verbindung zwischen PC und FPGA
- Reset: Zurücksetzen der Modellausführung in den Initialzustand.
- Starten des Ablaufs: Der Controller setzt über die Debug-Schnittstelle den FPGA auf seinen internen Takt. Das Modell wird in Echtzeit ausgeführt.

- Anhalten des Ablaufs: Über die Debug-Schnittstelle wird der FPGA auf externe Taktung konfiguriert und damit angehalten.
- Einzelner Zeitschritt: Einem einzelnen Zeitschritt auf Modellebene entspricht das Schreiben eines einzelnen Taktes am auf externe Taktung konfigurierten FPGA. Im ausgeführten Modell entspricht dies genau einem Modellschritt.

Der Controller ist einfacher aufgebaut als in der Softwarerealisierung, was in erster Linie durch die einfache Abbildung zwischen Modellschritt und Takt bedingt ist.

Ein Treiber, welcher komplexe Echtzeitereignisse oder Tracing ermöglicht, würde es in Zukunft erlauben, komplexe Stopp-Bedingungen (Breakpoints) im Modell zu definieren und ähnlich dem in Abschnitt 6.1.2.4 beschriebenen Echtzeittreiber eine Post-Mortem-Analyse ermöglichen.

7.1.4.2 Mapper

Analog zum Mapper in Abschnitt 7.1.3.2 iteriert auch die Hardwareimplementierung durch das statische Modell.

Zur Rekonstruktion der Menge der aktuell aktiven einfachen Zustände des Modells sind folgende Fragestellungen zu beachten:

1. Für welche zusammengesetzten Zustände existieren in der Realisierung Datenwerte, hier Register, welche Auskunft über aktuell aktive Unterzustände enthalten?
2. Wie kann aus der Anordnung im Modell auf die Adresse im linearen Adressraum der Debug-Schnittstelle geschlossen werden?
3. Wie werden ausgelesene Werte interpretiert? Die ganzzahligen Registerwerte müssen auf Modellartefakte, hier Unterzustände, abgebildet werden.

Frage 1 und 2 können anhand von Abbildung 7.5 beantwortet werden. JVHDLGen bildet genau jedes parallel ablaufende Kind eines AND-States, welches nicht ausschließlich weitere AND-States enthält auf genau ein Register ab. Dagegen werden XOR-Hierarchien, die das Modell strukturieren, bei der Codeerzeugung auf eine einzelne Ebene reduziert. Das Beispiel in der Abbildung enthält sechs parallele Bereiche. Hinzu kommt ein Register für die oberste Modellebene.

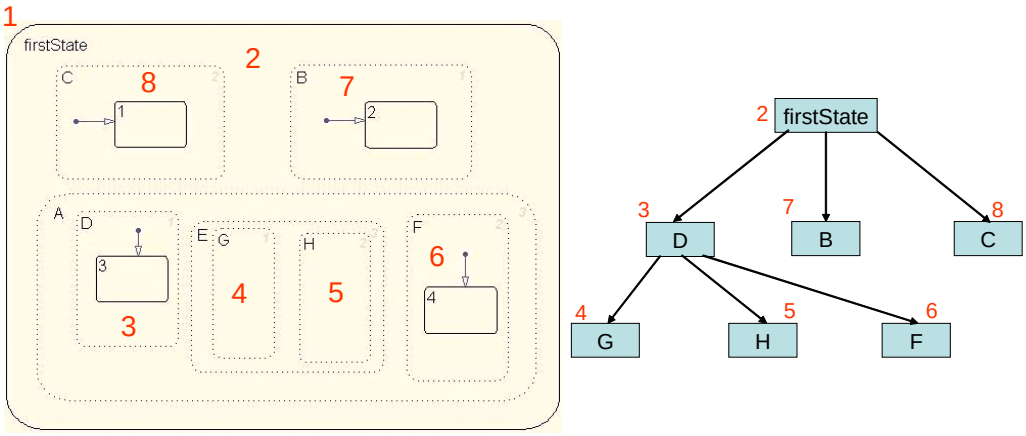

Abbildung 7.5: Parallele Zustände in Stateflow und ihre Zuordnung zu Adressen (links); Die aufgespannte hierarchische Baumstruktur in VHDL (rechts)

Die Nummerierung der Adressen folgt der Anordnung im Modell: Beginnend mit dem unsichtbaren Wurzelzustand der Zustandsmaschine werden alle direkten Unterzustände eingesammelt. Die Unterzustände werden alphabetische aufsteigend ihrem Namen nach sortiert. Anschließend werden rekursiv von jedem Unterzustand dessen Unterzustände gesammelt und sortiert usw. Dabei entsteht die in Abbildung 7.5 rechts gezeigte Baumstruktur. Die Vergabe der Adressen folgt den Strängen der entstehenden Baumstruktur. Dabei wird von der Wurzel ausgehend ein Strang bis zu seinen Blättern verfolgt und inkrementell jedem relevanten Knoten die Registeradresse vergeben. Die Iteration folgt dabei dem Muster einer Tiefensuche.

Ein einzelnes Register, welches den Zustand eines parallelen Prozesses verwaltet, wird in VHDL wie folgt erzeugt:

Listing 7.3: Von JVHDLGen erzeugtes Register

```
controller_c_StateMachine_State1_DBG   : out std_logic_vector(7 ...
    ... downto 0);
```

Das Register codiert mit seinem Wert, welcher seiner Unterzustände aktiv ist. Auch die enumerierende Codierung der Werte folgt einem festen Muster. Die Unterzustände werden dabei anhand ihrer graphischen Position im Zustandsdiagramm in eine Reihenfolge gebracht. Ausschlaggebend ist hier die vertikale Y-Position und bei Gleichheit die horizontale X-Position der linken oberen Ecke des Zustandssymbols.

Listing 7.4 zeigt die Definition des aufzählenden Datentyps im VHDL-Code:

Listing 7.4: Typisierung des von JVHDLGen erzeugten Registers

```
type controller_c_StateMachine_State1_TYPE is (virtual,
  controller_c_StateMachine_State1_Active,
  controller_c_StateMachine_State1_Idle);
```

Der eigentliche Mapping-Algorithmus entspricht dem Ablauf bei der Softwarerealisierung.

Neben der Rekonstruktion des Zustands können Datenwerte ausgelesen werden. Jedem Datenwert entspricht ein Register im FPGA, welches über die Treiberschicht adressiert und ausgelesen werden kann.

7.1.5 Realisierung

Die beschriebene Sicht wurde in mehreren *Viewpoints* realisiert, welche sich ihre Projektion und teilweise weitere Komponenten teilen. Es existieren jedoch Unterschiede in der Ansteuerung einzelner Zielsysteme, so dass unterschieden wird zwischen:

- Debugging eines mit Real-Time-Workshop generierten und mit GDB ausgeführten Stateflow-Chart
- Debugging eines mit Real-Time-Workshop generierten und mittels des iSystem-Treibers ausgeführten Stateflow-Chart. Es wurden zwei Treiber für nichtechtzeitfähiges und echtzeitfähiges Debugging realisiert.
- Debugging eines mit JVHDLGen erzeugten und auf einem FPGA ausgeführten Stateflow-Modell.

In Hinblick auf die Handhabung und die gewonnene Visualisierung der Debugging-Informationen ist der Debugger für den Benutzer jedoch entsprechend der an ihn gestellten Anforderungen völlig einheitlich.

Abbildung 7.6 zeigt einen Ausschnitt aus einer Debug-Session mit dem Viewpoint in der Variante für ein in Hardware ausgeführtes Stateflow-Modell. Die Funktionalität des *Debug Session*-Views, über die der Ablauf gesteuert werden kann, wird in Abschnitt 5.3.2 beleuchtet.

Im rechten oberen Bereich befindet sich die graphische Darstellung des modellierten Zustandsautomaten. Grün eingefärbt sind die aktuell aktiven einfachen Zustände des im FPGA laufenden Modells. Neben der direkt im Modell dargestellten Information über aktive Zustände zeigt der *Debug Contexts*-View rechts unten weitere gewonnene Daten in einer Baumstruktur. Eine Annotation direkt im Diagramm wäre künftig wünschenswert.

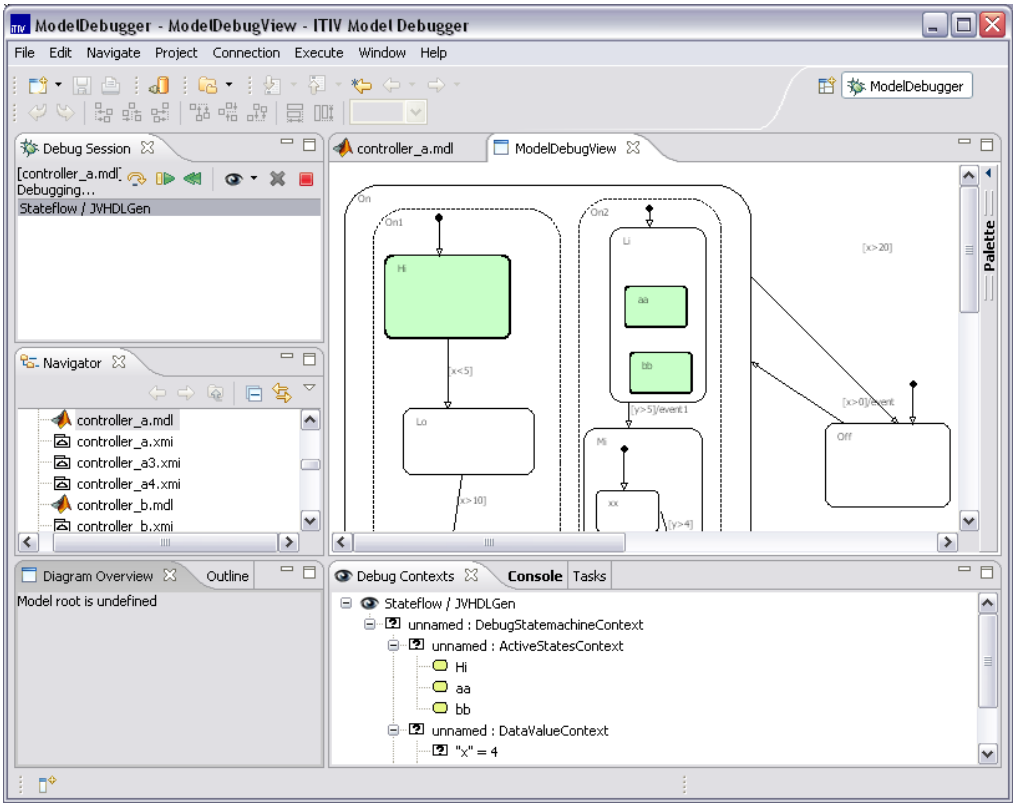

Abbildung 7.6: Visualisierung der gewonnen Laufzeitinformationen über Statecharts im Debugger

7.2 Verhaltensmodellierung mit Actions

Neben reaktivem Verhalten, wurden in Abschnitt 3.2 die UML *Action Semantics* als zentrale interne Darstellung von Verhalten in der UML beschrieben. Der folgende Abschnitt beschreibt die Visualisierung und Steuerung solcher Modelle zur Laufzeit durch einen spezialisierten *Viewpoint*.

7.2.1 Anforderungsanalyse

Der Codegenerator geht von einem UML-Modell aus, das die Action Semantics zur Beschreibung des Verhaltens von Operationen nutzt und erstellt daraus den ausführbaren Code in einer Zielsprache. Der generierte Code für sich allein ist nicht ohne weiteres mit dem Modell in Verbindung zu bringen und eine Fehlersuche auf Quellcodeebene wäre äußerst aufwendig durchzuführen. Durch die Intergration in den ITIV Modelldebugger, werden Laufzeitinformationen

des generierten Codes gewonnen und zurück auf die Modellebene abgebildet, so dass eine Fehlersuche im Action Semantics Modell möglich wird. Wichtig ist dabei, welche Funktionen das Modelldebugging ermöglichen soll, welche Informationen dafür benötigt werden, wie die Informationen gesammelt und wie sie für den Benutzer aufbereitet werden können.

Der *Viewpoint* für Action Semantics bietet eine Reihe von Funktionen. Die Actions können in Einzelschritten abgearbeitet werden oder es können Haltepunkte bei einzelnen Actions gesetzt werden, an denen die Ausführung dann anhält. Ist die Ausführung angehalten, können die Werte der Pins der einzelnen Actions und die Werte der Variablen abgelesen werden. Außerdem wird der Status im Lebenzyklus für jede Action ermittelt. Die Informationen zu den Actions werden in einer hierarchischen Baumstruktur widergegeben, die sowohl die Verschachtelung als auch die Ausführungsreihenfolge der Actions berücksichtigt.

Wie in Abschnitt 3.2.1 beschrieben, ist eine Modellierung direkt mit Action Semantics ohne eine darüberliegende Surface Langugage aus Gründen der Effektivität nicht sinnvoll. Daraus folgt, dass eine Fehlersuche im Modell auf Ebene der Action Semantics ebenfalls nicht sinnvoll ist. Wird demnach in zukünftiger Anwendung mit einer Surface Langugage gearbeitet, sollte auch der *Viewpoint* für Action Semantics um eine Sicht erweitert werden, die das Debugging auf diese Ebene hebt.

7.2.2 Metamodell

Im Folgenden sind die Informationen, die für das Debugging auf Action-Ebene relevant sind, aufgelistet. Es wird dabei zwischen dynamischen Informationen, die zur Laufzeit feststehen, und statischen Informationen, die aus dem Modell ableitbar sind, unterschieden. Anschließend wird das Informationsmodell oder Debug-Metamodell des Action Semantics *Viewpoint* vorgestellt.

1. Statische Informationen:

   (a) Ausführungsreihenfolge der Actions: In den Zielsprachen werden die Actions streng sequentiell abgearbeitet. Eine Darstellung sortiert anhand der Ausführungsreihenfolge ist deshalb sinnvoll.

   (b) Vorgänger und Nachfolger im Datenfluss: Nötig um zu verfolgen, welche Actions Werte untereinander austauschen.

   (c) Attribute und Assoziationen einer Action: Viele Actions besitzen Attribute und Assoziationen, die bekannt sein müssen, um ihre

Funktion zu verstehen. Bei einer *VariableAction* muss zum Beispiel abgelesen werden können, auf welche Variable sie sich bezieht.

2. Dynamische Informationen

   (a) Aktuell ausgeführte Action: Gibt an, bei welcher Action die Ausführung angehalten wurde.

   (b) Status im Lebenszyklus einer Action: Diese Information gibt Aufschluss über den Zustand einer Action. Außerdem kann festgestellt werden, welche Actions parallel ausgeführt werden könnten.

3. Werte an den Pins

4. Werte der Variablen

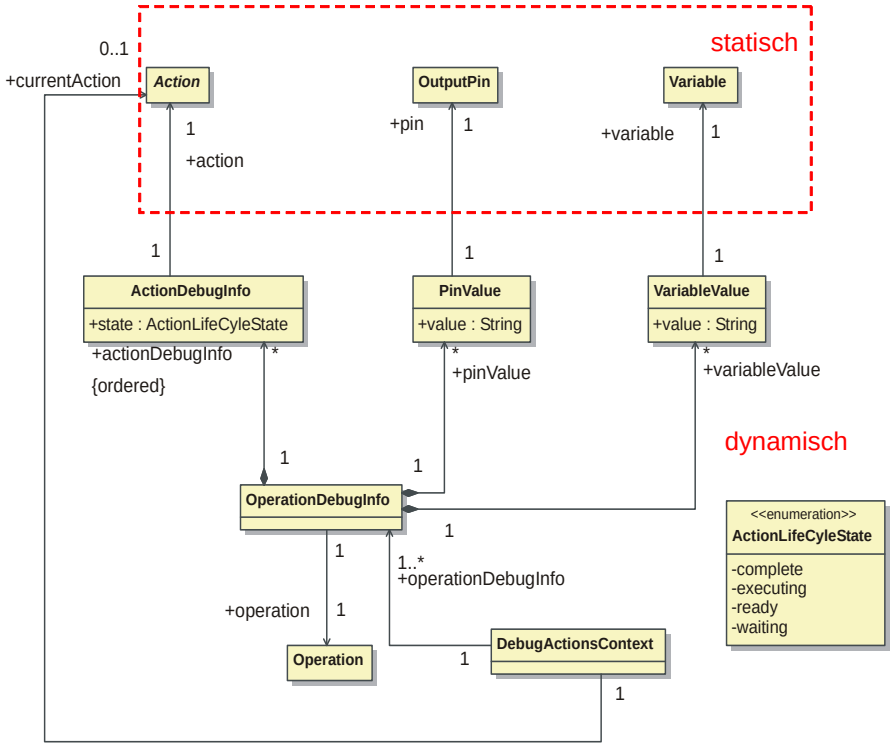

Abbildung 7.7: Erweiterung des UML Metamodells um Laufzeitinformationen für Actions

Im Debugmetamodell für Action Semantics (siehe Abbildung 7.7) sind die Informationen, die für das Debugging auf Action-Ebene relevant sind, zusammengefasst. Die Klasse *DebugActionsContext* stellt den Einstiegspunkt für den

Kontext eines Modells, das mit Action Semantics arbeitet, dar. Die Assoziation *currentAction* gibt die aktuell ausgeführte Action an. Bei ihr wurde die Ausführung angehalten. Für jede Operation steht eine Instanz der Klasse *OperationDebugInfo* zur Verfügung. Mit ihr sind alle Actions der Operation sortiert nach ihrer Ausführungsreihenfolge (*actionDebugInfo*) verbunden. Für jede Action wird der Status im Lebenszyklus in Form des Aufzählungstyps *LifeCycleState* angegeben. Die Klassen *PinValue* und *VariableValue* ordnen jedem Ausgabe-Pin und jeder Variablen ihren aktuellen Wert zu. Den Eingabe-Pins muss kein Wert zugeordnet werden, da ihr Wert über einen Datenfluss zum zugehörigen Ausgabe-Pin ermittelt werden kann.

7.2.3 Mapping im Debugger

Das in Abschnitt 7.2.2 beschriebene Metamodell wird benötigt, um die Funktionen des Debuggers zu realisieren. In diesem Abschnitt werden die Verfahren und Algorithmen zur Gewinnung der Informationen beschrieben.

7.2.3.1 Ausführungsreihenfolge der Actions

Der die Ausführungsreihenfolge bestimmende Algorithmus ist in Abschnitt 3.2.3.1 beschrieben. Die Ausführungsreihenfolge wird im Debugger für die sortierte Darstellung der Actions und zur Bestimmung des Status im Lebenzyklus einer Action eingesetzt.

Eine prinzipielle Möglichkeit besteht darin, den Algorithmus, welcher die Ausführungsreihenfolge bei der Codeerzeugung bestimmt während des Debugging noch einmal durchlaufen zu lassen. Die vorliegende Realisierung protokolliert die Abfolge jedoch während der Codeerzeugung mit. Durch Nutzung dieser Datei wird der Debugger robuster gegenüber verschiedenen Varianten bei der Bestimmung der Reihenfolge von Actions.

7.2.3.2 Aktuell ausgeführte Action

Um das Action Modell ähnlich wie den Quelltext eines klassischen Debuggers in einzelnen Schritten durchlaufen zu können, beziehungsweise Haltepunkte setzen zu können, ist eine eineindeutige Zuordnung zwischen Quelltextzeile und zugehöriger Action notwendig. Dabei durchläuft der Quelltext-Debugger den Code in Einzelschritten und überprüft nach jedem Schritt, ob eine neue Action erreicht werde und gegebenenfalls angehalten. Somit wird ein Schritt auf Modellebene durchgeführt.

Die aktuelle Action ist nicht mit der Action identisch, deren Lebenszyklus den Status *executing* aufweist. Nicht nur die gerade ausgeführte Action hat den Status *executing*, sondern auch alle Actions in der sie enthalten ist.

Um die aktuelle Action bestimmen zu können, wird im generierten Quelltext am Anfang des Codes jeder Action ein Kommentar mit folgendem Aufbau eingefügt:

//§[Action UUID]

Der Kommentar markiert die Codezeile, bei der eine neue Action beginnt durch die eindeutige UUID[1] der zugehörigen Action im Modell. Stoppt die Ausführung an einer bestimmten Zeile des Quelltextes, kann durch aufsteigendes Absuchen des Quellcodes der Kommentar der zugehörigen Action und damit die Action selbst gesucht werden. Die Quelltextzeilen können anders nicht zugeordnet werden, da die Codeerzeugung der Formatierung durch Templates unterliegt.

7.2.3.3 Status im Lebenszyklus einer Action

Der Debugger soll eine Darstellung des Status im Lebenszyklus für jede einzelne Action erlauben. Dazu werden nur die aktuell ausführte Action sowie die Ausführungsreihenfolge aller Actions benötigt. Die aktuell ausgeführte Action repräsentiert dabei den Zeitpunkt, zu dem die Bestimmung stattfindet.

Der Algorithmus geht wie folgt vor:

- Alle Actions welche in der Ausführungsreihenfolge vor der aktuellen Action liegen erhalten den Zustand *complete*.
- Actions, welche die aktuelle Action enthalten, erhalten den Zustand *executing*.
- Für Actions im Zustand *ready* wurden bereits alle Daten- und Kontrollflussvorgänger abgearbeitet, d.h. alle Vorgänger besitzen bereits den Zustand *complete*.
- Alle weiteren Actions, welche Vorgänger im Daten- oder Kontrollfluss aufweisen, die nicht im Zustand *complete* sind, befinden sich selbst im Zustand *waiting*.

[1] Ein *Universally Unique Identifier* (UUID) bezeichnet einen Standard für Identifikatoren, welcher aus einer 16-Byte Zahl besteht, die als Zeichenkette dargestellt wird.

Listing 7.5: Pseudocode des Algorithmus zur Bestimmung des Lebenszyklus einer Action

```
currentAction.state = "executing";
for(int i=0; i < currentActionIndex; i++) {
  sortedActions.get(i).state = "complete";
}

List currentParents = getParents(currentAction);
for(int i=0; i < parents.size(); i++) {
  currentParents.get(i).state = "executing";
}

for(int i=currentActionIndex+1, i<sortedActions.size(); i++) {
  allPredecessorsComplete = true;

  List predecessors = getPredecessors( sortedActions.get(i) );
  for(Action action : predecessors) {
    if( action.state != "complete" ) {
      allPredecessorsComplete = false;
      break;
    }
  }

  List parents = getParents( sortedActions.get(i) );
  for(Action action : parents) {
    if( action.state == "waiting" ) {
      allPredecessorsComplete = false;
      break;
    }
  }

  if( allPredecessorsComplete ) {
    sortedActions.get(i).state = "ready";
  } else {
    sortedActions.get(i).state = "waiting";
  }
}
```

Listing 7.5 zeigt den Pseudocode des beschriebenen Algorithmus in einer Übersicht. Dabei sind:

- *currentAction*: die aktuell ausgeführte Action
- *sortedActions*: eine Liste aus Actions nach Ausführungsreihenfolge sortiert
- *currentActionIndex*: der Index der aktuellen Action in der Liste sortedActions
- *getParents(Action)*: Methode, die alle Eltern der übergebenen Action im Modellbaum findet
- *getPredecessors(Action)*: Methode, die alle Vorgänger im Daten-und Kontrollfluss einer Action liefert

7.2.3.4 Werte an Pins und Variablen

Über die Symboltabelle ist jedem Pin und jeder Variablen im Modell ein Variablenname im Quelltext zugeordnet. Der Debugger kann die Werte der Quelltextvariablen zur Laufzeit liefern. Diese werden dann den entsprechenden Pins und Variablen im Modell zugeordnet. Die Abbildung erfolgt dabei im Allgemeinen aus Richtung des Modells in den Variablennamen auf Quelltextebene.

7.2.4 Realisierung

Um die durch den Mapper erhaltenen Laufzeitdaten zu visualisieren, wurde im Rahmen von [Gutstein 2007] ein Projektor für *ModelScope* erarbeitet. Abbildung 7.8 zeigt den Viewpoint zum Debugging von UML Actions. Die linke Seite der Oberfläche zeigt einen Modellbaum, welcher auf Pakete, Klassen, Operationen und Actions reduziert ist. Dieser gibt die Kompositionshierarchie des Modells wieder. Die Elemente sind jeweils durch ihren Namen im Modell gekennzeichnet. Actions auf der gleichen Hierarchieebene sind in ihrer Ausführungsreihenfolge sortiert dargestellt.

Bei Doppelklick auf eine Action wird ein Haltepunkt gesetzt und in der zweiten Spalte des Baums markiert. Die dritte Spalte zeigt den Status im Lebenszyklus der jeweiligen Action und in der letzten Spalte befindet sich der Typ des Modellelements.

Im rechten Bereich des Projektors befindet sich eine Detailansicht für die aktuell im Baum angewählte Action. In dieser sind Werte von Variablen, Attributen

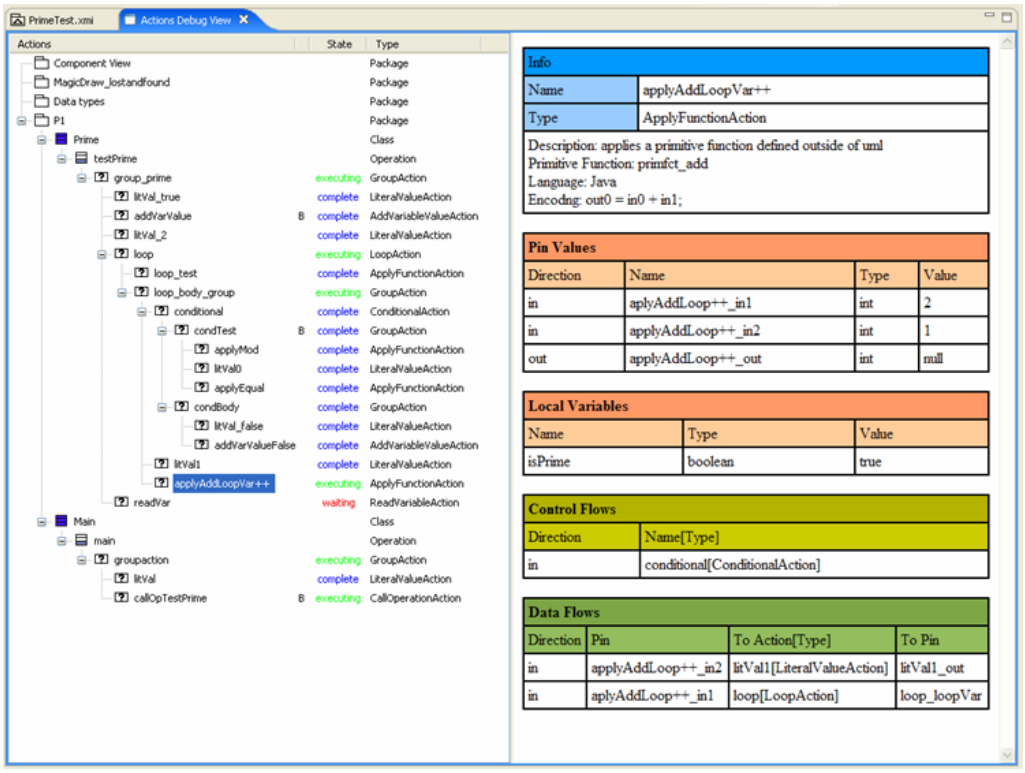

Abbildung 7.8: Die durch den Projektor erzeugte Visualisierung und Benutzeroberfläche zum Debugging von Actions

und Assoziationen angegeben. Weiterhin können hier Vorgänger und Nachfolger im Daten- und Kontrollfluss der Action abgelesen werden.

Die realisierte Darstellung ist für das Debugging auf Ebene der Actions geeignet. Datenflussbeziehungen zwischen den Actions sind jedoch nicht sehr gut ablesbar. Hier wäre eine diagrammatische Darstellung in einem Datenflussgraphen deutlich besser geeignet. Ein prinzipielles Problem wäre aber auch hier wegen des Detaillierungsgrades des Actions-Modell die Unübersichtlichkeit. Sollte wie bereits erwähnt ein Entwurf auf Ebene einer Action Language möglich sein, ist die hier gezeigte Darstellung nicht mehr von Nutzen. Die ihr zu Grunde liegenden Debug-Informationen können jedoch als Grundlage einer neuen Darstellung auf höherer Ebene dienen.

7.3 Darstellung von Klassenmodellen als Objektdiagramme zur Laufzeit

Objektdiagramme dienen, wie in Abschnitt 2.3.1.3.2 gezeigt dazu, einen Teil der Instanziierung eines Klassenmodells zu einem bestimmten Zeitpunkt zur Laufzeit darzustellen.

Wie aus den in Abschnitt 4.5 vorgestellten Werkzeugen ersichtlich ist, existieren bereits mehrere Ansätze welche versuchen, Instanzen zur Laufzeit als Graphen darzustellen, welche sich implizit oder explizit an eine Notation als Objektdiagramm anlehnen.

Auch in *ModelScope* wurde eine solche Sicht integriert. Neben den Anforderungen, welche in Abschnitt 7.3.1 beleuchtet werden, legt der folgende Abschnitt den Schwerpunkt der Betrachtung auf die Gewinnung des Modells aus Laufzeitinformationen auf Quelltextebene.

Abschließend wird auch hier die Darstellung im Debugger präsentiert und auf die Problematik eines durch den Menschen erfassbaren Layouts eingegangen.

7.3.1 Anforderungsanalyse

Ausgangspunkt eines erzeugbaren Objektmodells ist die vollständige Strukturbeschreibung durch ein in einem Klassendiagramm notierten Klassenmodell. Dieses Klassenmodell ist vollständig generierbar, muss jedoch keine den Operationen zugeordneten Verhaltensspezifikation beinhalten.

In den für diese Arbeit untersuchten Prototypen wurde jedoch für alle Operationen Verhalten sowohl in textueller, als auch in Form von Actions (siehe Abschnitt 3.2) spezifiziert. Als Codegenerator wurde der vorlagengesteuerte Codegenerator der Integrationsplattform Aquintos.GS (siehe Abschnitt 3.1.5) eingesetzt, so dass ein Klassenmodell die Software komplett beschreibt.

Abbildung 7.9 zeigt als Beispiel eine Applikation *SystemController*, welche eine doppelt verkettete Liste[2] nutzt, die in der Klasse *LinkedList* realisiert wurde. Zur Laufzeit ist es von Interesse eine Überblick zu erhalten, welche Objektinstanzen als Listeneinträge existieren und wie diese angeordnet sind.

Die Hauptapplikation besitzt als Ausgangspunkt einen klassenweiten Zeiger

[2]Eine doppelt verkettete Liste ist eine elementare Datenstruktur zur Sammlung mehrerer Elemente, welche schnelles Einfügen und Löschen von Einträgen begünstigt. Die Elemente sind durch Zeiger jeweils mit ihrem Vorgänger und Nachfolger verknüpft. Eine ausführliche Beschreibung bieten [Cormen u. a. 2001], S. 205ff.

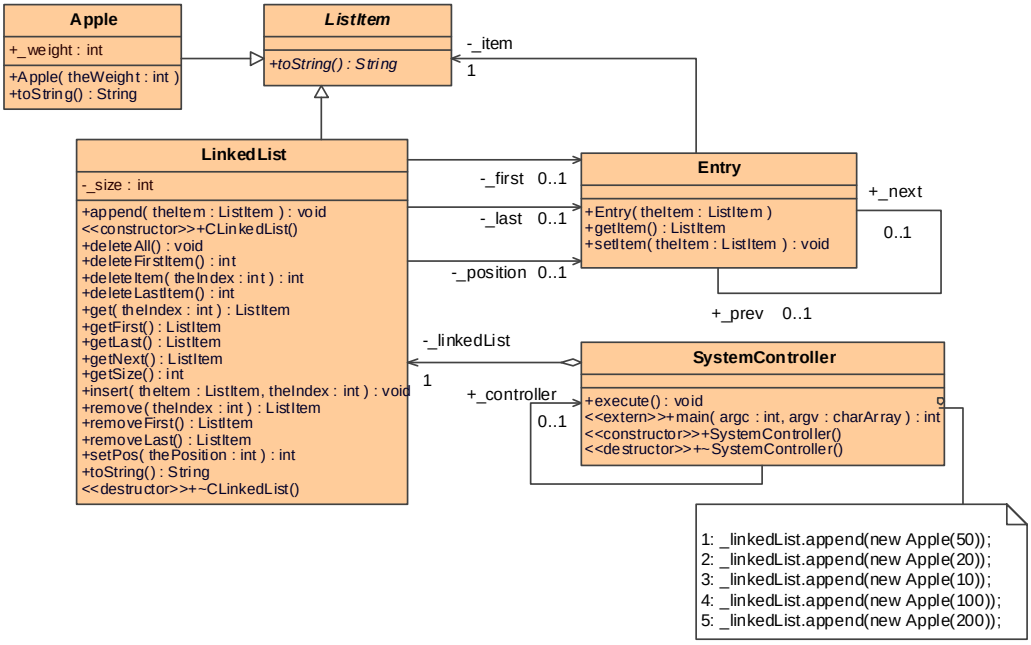

Abbildung 7.9: Beispiel eines auführbaren UML Klassenmodells: Doppelt verkettete Liste LinkedList

_controller auf die einzige Instanz von *SystemController*. Dieser erzeugt genau eine Instanz der Klasse *LinkedList*. Im Verlauf der Operation *execute()* fügt der *SystemController* durch Aufruf der Operation *append()* aus *LinkedList* der Liste zusätzliche Elemente hinzu. Die Klasse *LinkedList* verwaltet dabei die Elemente der Liste über die Zeiger *_first* auf das erste und *_last* auf das letzte Element. Als Elemente der Liste kommen alle Objekte (hier *Apple*) in Frage, deren Klasse von der abstrakten Klasse *ListItem* erben. Beim Hinzufügen dieser Elemente zur Liste, werden sie mit einer Instanz der Klasse *Entry* gekapselt, welche über die auf sich selbst verweisenden Assoziation *_prev* - *_next* die Verknüpfung der Listenelemente verwaltet.

Die Objektansicht erfüllt folgende Anforderungen:

- Darstellung aller **Instanzen** im Speicher zur Laufzeit. Für das Objekt wird ein eindeutiger *Identifier* und seine Klasse dargestellt.
- Die **Werte aller Attribute** der instanziierten Klassen werden als Slots mitsamt ihrer Werte in Klartext angezeigt.
- Die **Verknüpfung** der Objekte untereinander durch *Links* wird visualisiert. Eine Zuordnung zur instanziierten Assoziation ist durch optional an die *Link*-Enden annotierte Rollennamen möglich.

- **Polymorphie**: Unterstützen die Laufzeitumgebung und der Debugger-Treiber Laufzeittypinformation[3], so wird nicht die über die Assoziation verknüpfte Klassifizierung einer Instanz dargestellt sondern der tatsächliche Typ, welcher das Assoziationsende erbt. Abbildung 7.10 verdeutlicht dies am Beispiel der Assoziation zwischen *Entry* und *ListItem* in Abbildung 7.9: Ist es möglich, über den Treiber Typinformationen über ein referenziertes Objekt zu erhalten, können dadurch der tatsächliche Typ aller Objekte und zusätzliche Attribute gewonnen werden. Andernfalls steht nur die ungenauere Information aus dem statischen Modell zur Verfügung.

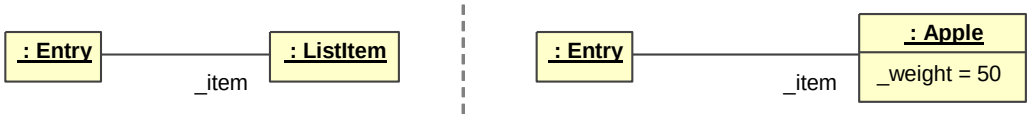

Abbildung 7.10: Instanziierung der Assoziation zwischen Entry und ListItem ohne (links) und mit (rechts) Beachtung der Polymorphie durch Laufzeittypinformation.

Die Sicht zur Darstellung der Objektkonfiguration besitzt keinen eigenen *Controller* und damit keine Möglichkeit den Ablauf selbst zu steuern. Dies geschieht mit Hilfe der Ablaufsteuerung anderer *Viewpoints*, beispielsweise der *Actions* und Zustandsautomaten.

7.3.2 Mapping

Ziel des Mapping ist der Aufbau eines Netzes aus Objekten, Links und Attributwerten zur Laufzeit, welches die im vorigen Abschnitt ausgearbeiteten Eigenschaften aufweist.

Im Gegensatz zu den bisher beschriebenen *Viewpoints* wird keine gesonderte Erweiterung des UML-Metamodells benötigt, um die Laufzeitinformationen im Datenmodell darzustellen. Die dynamisch erzeugten Modelldaten werden auf Basis der Beschreibung von Objektmodellen im Metamodell der UML abgelegt (siehe Abschnitt 2.3.1.3.2).

[3]Laufzeittypinformationen (engl. *Run-Time Type Information*, RTTI) werden in Java über den Reflektionsmechanismus unterstützt. In C und C++ ist sie optional (siehe [Stroustrup 2000], S. 434ff). Sie wird jedoch durch zahlreiche Compiler, darunter die GNU Werkzeugkette unterstützt. Da Laufzeittypinformation das ausführbare Programm und den benötigten Arbeitsspeicher vergrößert, muss sie explizit beim Übersetzen des Programms angegeben werden. Aus demselben Grund steht RTTI in eingebetteter Software meist nicht zur Verfügung.

Ausgangspunkt zur Gewinnung eines solchen Netzes muss immer eine statische Referenz auf eine Instanz einer Klasse im Speicher der Ausführungsplattform sein.

Abbildung 7.11 verdeutlicht die im Folgenden beschriebenen Schritte zum rekursiven Aufbau des Objektnetzes:

Ausgegangen wird von der Behandlung einer konkreten Klasse, auf welche die statische Referenz verweist. Zunächst wird, wie in Abbildung 7.11 oben gezeigt, eine *InstanceSpecification* erzeugt, welche das Objekt selbst darstellt. Die *InstanceSpecification* verweist auf die Klasse im statischen Modellteil, welche durch sie instanziiert wird.

Im folgenden Schritt wird über alle eigenen und ererbten Attribute (Metaklasse *Property*) der instanziierten Klasse iteriert, eine Instanz der Metaklasse *Slot* erzeugt und der jeweiligen *Property* zugewiesen (siehe Abbildung 7.11 mitte). Der Wert der Attributrealisierung wird aus dem Zielsystem über den Treiber ausgelesen und über eine weitere typisierte *InstanceSpecification* dem *Slot* zugewiesen.

Die Abbildung von Attributen auf Artefakte im Quelltext geschieht durch den Codegenerator von Aquintos.GS. Da dieser über Vorlagen angepasst werden kann und die erzeugten Daten im Quelltext flexibel veränderbar sind, wird der Ausdruck, welcher zur Abfrage eines Attributwerts dient, mittels UML Actions spezifiziert und durch den Codegenerator auf Basis der auch bei der Codegenerierung genutzten *Templates* nach C++ und damit in die Sprache des Debuggers umgewandelt.

Nach Behandlung eines Objekts wird über alle Assoziationen, die mit der betrachteten Klasse verknüpft sind, iteriert. Besitzt die Assoziation eine Multiplizität größer als 1, wird über alle Einträge der verknüpften Datenstruktur iteriert. Dabei wird zunächst das Ziel der jeweiligen Assoziation durch Evaluation des entsprechenden Ausdrucks analog zur Auswertung von Attributen aus dem Zielsystem gelesen.

Falls für das referenzierte Objekt im Zielsystem noch keine *InstanceSpecification* im dynamischen Modellteil existiert, wird diese rekursiv mitsamt ihrer Attributwerte und Verknüpfungen erzeugt. Die schon behandelten Objekte und dabei erzeugten *InstanceSpecifications* werden intern in einem Verzeichnis gehalten, so dass nach Durchlaufen der Rekursion genau jedes über eine Assoziation ereichbare Objekt im Zielsystem eine *InstanceSpecification* erhält.

Für das aktuell behandelte und das referenzierte Objekt wird eine *InstanceSpecification* der Assoziation erzeugt und über *Slots* mit den zur Assoziation gehörenden *Properties* verknüpft. Dies ist in Abbildung 7.11 unten verdeutlicht.

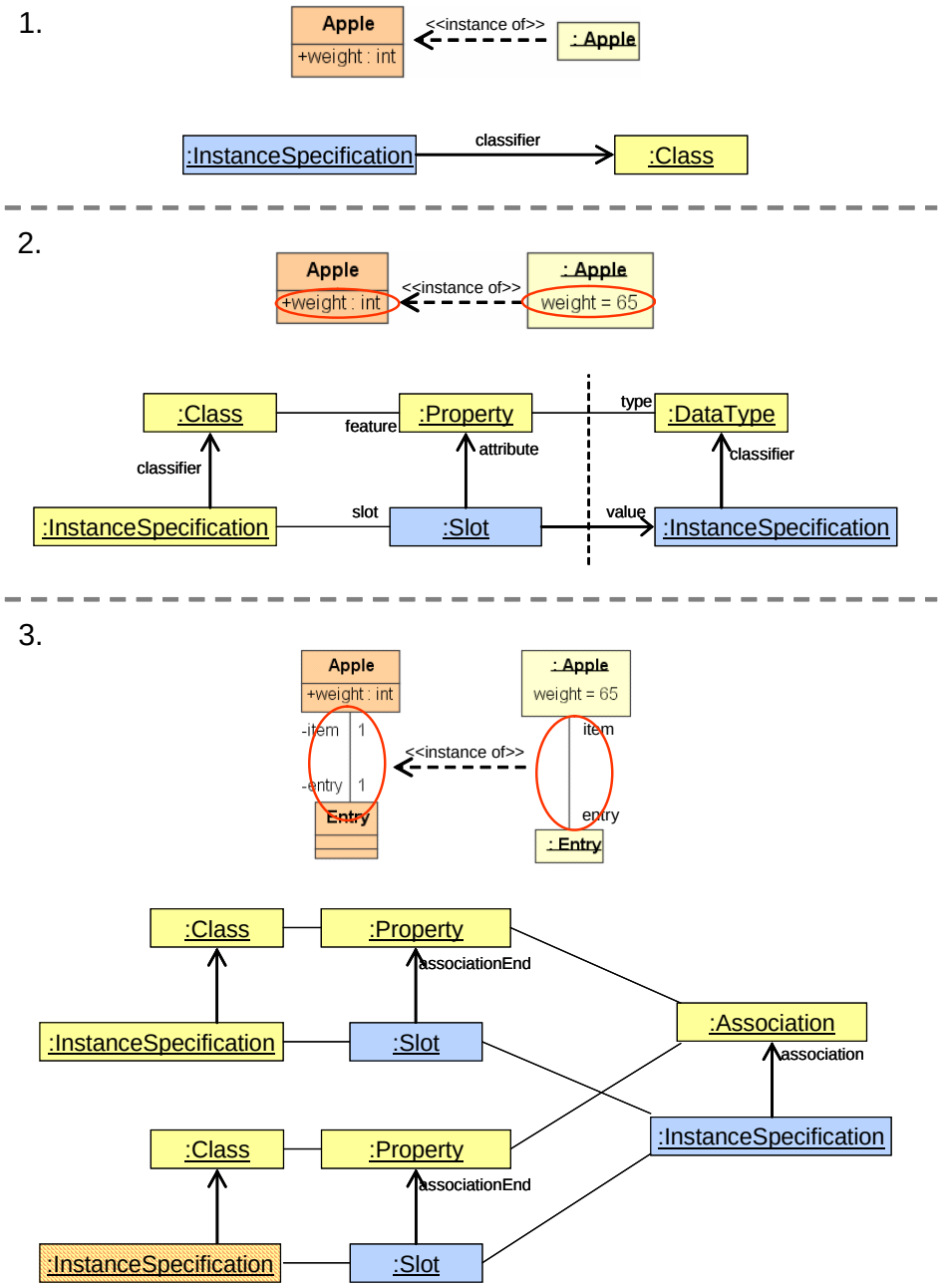

Abbildung 7.11: Schritte beim Aufbau eines Objektmodells durch Erzeugen von Objekten (Schritt 1), Attributwerten (Schritt 2) und Links (Schritt 3). Oben jeweils das Objektdiagramm als Instanz des Klassenmodells in konkreter Syntax, darunter die Darstellung in abstrakter Syntax

7.3.3 Realisierung

Zur Darstellung von Objektdiagrammen zur Laufzeit wurde eine Visualisierung von Objektmodellen auf der Basis des in Abschnitt 5.3.4.3 beschriebenen Frameworks realisiert.

Neben dem Mapping des Objektmodells muss hier jedoch auch das von der Visualisierungsschicht als *DiagramInterchange*-Modell erwartete Diagramm neu erzeugt werden. Dies kann entweder zu jedem Zeitpunkt neu oder inkrementell erfolgen. Der folgende Unterabschnitt 7.3.3.1 beschreibt Randbedingungen, welche sich beim Layout eines solchen Diagramms ergeben.

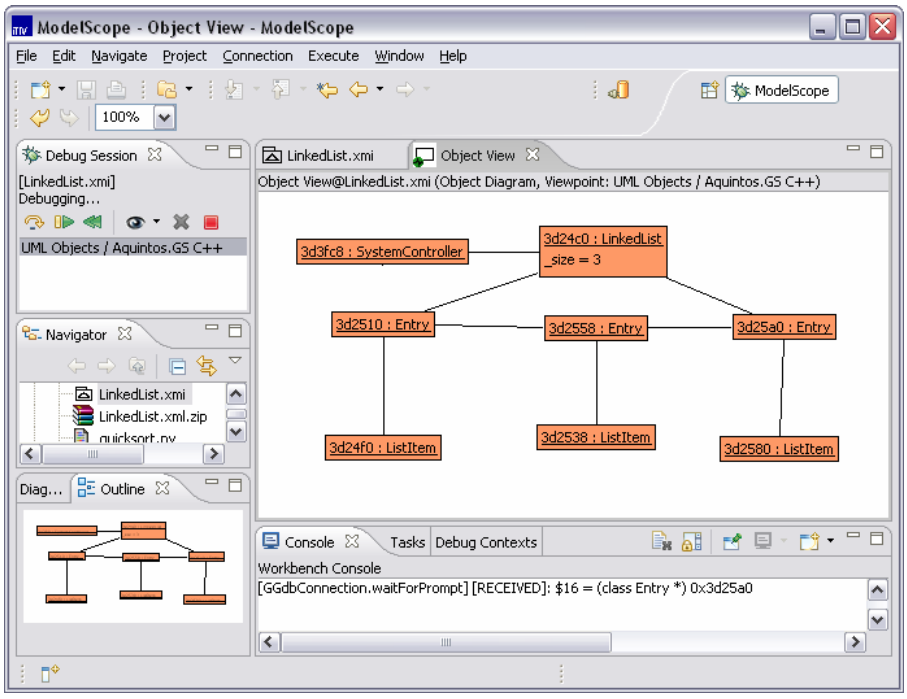

Abbildung 7.12: Darstellung der Objektkonfiguration in ModelScope

In Abbildung 7.12 ist der *Viewpoint* im zentralen Bereich der Abbildung dargestellt. Das untersuchte Modell ist hier das Beispiel der doppelt verketteten Liste aus Abbildung 7.9.

Die Modellausführung wurde unterbrochen, nachdem drei Elemente hinzugefügt wurden. Aus der Darstellung ist ersichtlich, dass das hier verwendete Compiler-Debugger-Paar keine Laufzeittypinformation unterstützt.

7.3.3.1 Layout-Aspekte

Bei Erzeugung einer großen Menge an Knoten und Kanten muss der menschliche Nutzer eines erzeugten Diagramms in seiner Rezeption durch eine geeignetes Layout der gezeigten Information unterstützt werden.

Ziel eines Layoutalgorithmus ist dabei, überschaubare Zeichnungen zu erzeugen, die ästhetischen Anforderungen genügen. Graphen besitzen jedoch abhängig von ihrem Diagrammtyp und der dargestellten Information variierende Ästhetikkriterien. [Coleman u. Parker 1996; Tamassia u. a. 1988] untersuchen zahlreiche allgemeine Kriterien. Dies wurden von [Purchase u. a. 2001] auf die Darstellung von Klassendiagrammen angewandt und durch Befragung von Studenten, welche Diagramme bewerten sollten, in ihrer Wichtigkeit evaluiert. Die wichtigsten Kriterien waren:

- Minimale Anzahl von Linienkreuzungen
- Minimale Anzahl von Linienknicken
- Nur horizontale Beschriftungen

Diese Kriterien können aufgrund der Ähnlichkeit auf Objektdiagramme übertragen werden.

In [Fan 2007] wurden verschiedene Layout-Algorithmen auf ihre Anwendbarkeit auf Objektdiagramme in *ModelScope* untersucht. Abbildung 7.13 zeigt drei Möglichkeiten für automatisches Layout.

Punkt 1 zeigt das Diagramm ohne jegliches Layout. Der Benutzer kann das Diagramm im Editor manuell arrangieren; dies bedeutet jedoch speziell bei größeren Modellen, einen unvertretbaren Aufwand.

Eine Verbesserung stellt ein hierarchisches Layout mit Hilfe des Sugiyama-Algorithmus dar (siehe dazu [Battista u. a. 1998; Bastert u. Matuszewski 2001]). Der Sugiyama-Algorithmus ist der bekannteste Vertreter von Algorithmen, die hierarchische Graphen layouten. Als Eingabe erhält er einen gerichteten Graphen, der im Laufe des Algorithmus hierarchisiert wird. Als Ausgabe entsteht eine Zeichnung, die die im Graph vorherrschende Hierarchie herausstellt. Der Algorithmus besteht aus drei Phasen: Ebeneneinteilung, Kreuzungsreduzierung und Koordinatenzuweisung.

Bei der Anwendung auf Objektnetze (Abbildung 7.13, Punkt 2) erweist sich der Algorithmus nur für stark hierarchisierte Objektnetze als geeignet, im Beispiel der doppelt verketteten Liste suggeriert er sogar nicht existierende Hierarchien.

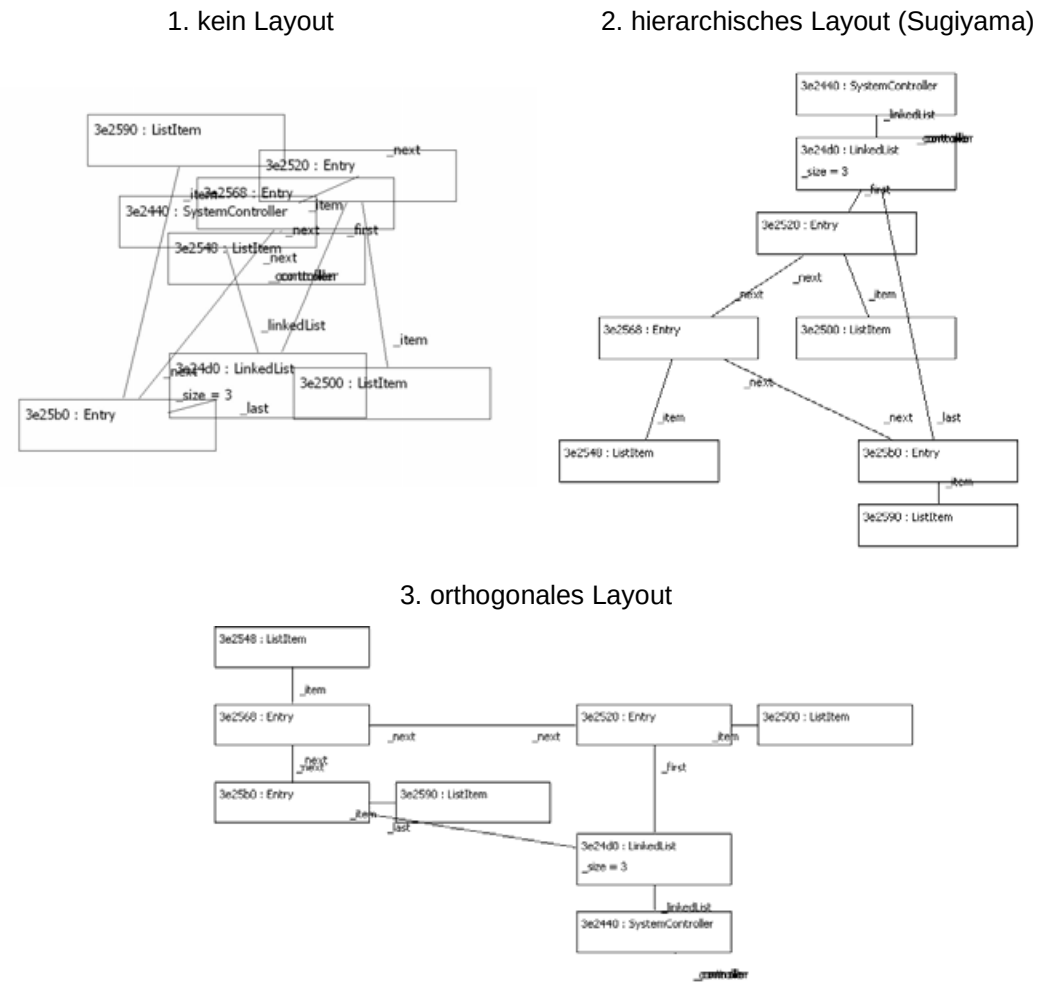

Abbildung 7.13: Layout für das LinkedList-Beispiel mit unterschiedlichen Verfahren

Geeigneter sind orthogonale Layoutalgorithmen, welche Knoten und Kanten so anordnen, dass eine möglichst optimale Darstellung ohne diagonale Kanten möglich wird. Zur Evaluation dieses Verfahrens wurde der Kandinsky-Algorithmus evaluiert (siehe dazu [Tamassia 1987; Eiglsperger u.a. 2001]). Dieser Algorithmus ist für Objektnetze besser geeignet und erzeugt ein akzeptables Layout.

Sinnvoll ist jedoch im Allgemeinen ein halbautomatisches Vorgehen, bei dem ein grundlegendes Layout mit Hilfe eines automatischen Algorithmus erzeugt wird. Die Anordnung wird jedoch anschließend manuell durch den Benutzer verbessert.

Dabei ist eine inkrementelle Veränderung des Diagramms nach Starten und Stoppen des Systems wünschenswert, damit sich das Diagramm möglichst nur geringfügig verändert. Weiterhin wäre eine farbliche Markierung der Veränderungen hilfreich für die Übersicht.

Ein Diagramm, welches alle Objekte in einer komplexen Software darstellt, gerät sehr schnell an die Grenze des menschlich Erfassbaren. Hier ist zukünftig ein erweitertes Konzept nötig, welches erlaubt, den Fokus auf bestimmte Bereiche des Objektmodells zu legen. [Jacobs u. Musial 2003] schlagen hierzu die sukzessive Reduktion von Information in den Knoten und auf den Kanten (*Level-of-Detail*) vor, welche das Umfeld eines fokussierten Objekts entsprechend der Entfernung reduzieren oder ganz ausblenden. Es sind jedoch auch interaktive Konzepte vorstellbar, welche Instanzen bestimmter Klassen filterbar oder manuell Assoziationen verfolgbar machen.

Kapitel 8

Fazit

Modellgetriebene Entwicklung gewinnt in den letzten Jahren, auch durch kommerzielle Initiativen großer Softwarehersteller immer mehr an Bedeutung und führt die Tradition der Entwicklung mächtigerer und stärker abstrahierender Beschreibungsmittel im System- und Softwareentwurf fort. Dies gilt insbesondere auch für eingebettete Software – einerseits da in verschiedenen Ingenieurdisziplinen graphische Modelle bereits verbreitet sind, andererseits da die Nutzung von Modellen als primäres Artefakt in der Entwicklung neue Wege in der Validierung und Verifizierung sicherheitskritischer Systeme eröffnet.

Die Nutzung ausführbarer Modelle verringert den Abstand zwischen der Erfahrungswelt des Systemarchitekten und, im Falle domänenspezifischer Sprachen des Domänenexperten auf der einen Seite und den Artefakten des Zielsystems in Form von Quellcode auf der anderen Seite. Verschiedene Ansätze zur Nutzung ausführbarer Modelle in eingebetteten Systemen, wie CASE-Werkzeuge, die *Unified Modeling Language*, das Subset der *Executable UML*, die *Model Driven Architecture* und domänenspezifische Sprachen konkurrieren miteinander.

8.1 Zusammenfassung

Inhalt der vorliegenden Arbeit war der Entwurf einer Entwicklungsumgebung zur Arbeit mit ausführbaren graphischen Modellen. Das Augenmerk lag insbesondere auf dem Entwurf und der Realisierung einer Plattform zum Debugging ausführbarer Modelle. Um eine solche Plattform realisieren zu können, wurde jedoch in einigen Fällen, speziell bei der Nutzung von UML Actions zur Verhaltensbeschreibung und beim Aufbau einer in das Modell integrierten Pro-

jektverwaltung, Arbeiten durchgeführt, welche erst eine Nutzung heterogener Modelle als ausführbare Programmiersprachen ermöglichen.

Dazu wurden in einem Grundlagenkapitel zunächst Begriffe geklärt und Technologien vorgestellt, welche für die Arbeit mit Modellen essentiell sind. Darunter fallen insbesondere Konzepte der Metamodellierung und die Beschreibung wichtiger Spezifikationen der *Object Management Group* in diesem Umfeld. Neben dem Vier-Schichten-Modell der Metamodellierung wurde auf die Metametasprache MOF eingegangen und Abbildungen nach Java durch JMI und XML durch XMI diskutiert. Weiterhin wurde die Spezifikation DI zum Diagrammaustausch zwischen Modellierungssprachen beschrieben. Die Modellierungssprachen und –konzepte die heute in der Entwicklung eingebetteter Software eine wichtige Rolle spielen sind die UML, auf OMG-Technologie beruhende domänenspezifische Sprachen und vor allem nach wie vor proprietäre CASE-Werkzeuge.

Die Modellierungssprachen können als Programmiersprachen genutzt werden, wenn mit ihnen beschriebene Modelle ausführbar sind. Hier ist als derzeit wichtiges Paradigma die *Model Driven Architecture* (MDA) zentral, welche Begriffe und Grundkonzepte auf dem Weg vom Modell zum ausführbaren Code spezifiziert. Weitere Konzepte und Werkzeuge lassen sich in diesen Entwurfsfluss integrieren. Die Modellverwaltung und –integration, sowie die Codeerzeugung aus statischen Teilen des UML-Modells geschah in dieser Arbeit mit Hilfe des Werkzeugs Aquintos.GS.

Damit auch UML Modelle vollständig in Code generierbar und damit ausführbar werden, wurde im Rahmen dieser Arbeit eine fast vollständige Abbildung von UML Actions auf Java- und C++-Code durchgeführt und realisiert. Da UML Actions keine spezifizierte Syntax besitzen, wurde auch der umgekehrte Weg, die Überführung von Prozedurbeschreibungen in der Programmiersprache Java in ein Modell auf Basis des UML Actions Pakets entworfen und realisiert.

Kern der Arbeit war jedoch die Untersuchung von Fehlersuche in eingebetteter Software und die Entwicklung einer Debugging-Methodik, welche für einen modellgestützten Entwicklungsprozess adäquat ist. Dies geschah ausgehend von der Erkenntnis, dass die Erweiterung der Kette von der Systembeschreibung bis zur ausführbaren Software um Modelle neue Werkzeuge nötig macht. Diese müssen es ermöglichen, das laufende System so zu beobachten, dass sich Informationen auf Entitäten aus der semantischen Welt des Modells beziehen.

Zunächst wurden dazu Begriffe und grundlegende Konzepte bei der Fehlersuche vorgestellt. Der Begriff *Debugging* wurde definiert und in Relation zu anderen Themen aus dem Bereich Fehlersuche gesetzt. Darauf aufbauend wurden grundlegende Strategien und Methoden für das Debugging von Software

dargestellt und die für die vorliegende Arbeit wichtigen Werkzeuge beschrieben. Insbesondere wurde auch auf die Besonderheiten beim Debugging von eingebetteten Systemen eingegangen. Diese zeichnen sich durch oft harte Echtzeitbedingungen aus und integrieren Prozessoren mit Peripherie und Ausführungseinheiten in Form konfigurierbarer Hardware. Sie weisen spezielle Schnittstellen und Werkzeuge auf Quelltextebene auf, welche das Tracing in Echtzeit mit begrenzter Bandbreite erlauben. Im der Evaluation existierender Ansätze kristallisierte sich heraus, dass Methoden zum Debugging von Modellen bislang nur vereinzelt vorhanden und für eingebettete Software nur bedingt nutzbar sind.

Daher wurde *ModelScope* entwickelt, eine Plattform zur Ausführung von und zum Debugging in graphischen Modellen. Die Basisplattform bietet die Arbeit mit in das UML Metamodell transformierten Modellen und eine Projektverwaltung, welche den Weg vom Modell zum ausführbaren System als ausführbaren Teil des Modells selbst beschreibt.

Entscheidend für das Debugging ist das Konzept der Erweiterung des statischen Entwurfsmodells um eine zur Laufzeit dynamische Komponente, welche in das verwendete Metamodell, hier die UML, integriert wird. Diese ist Basis für den Debugger, der in die Basisplattform von *ModelScope* integriert wurde. Der Debugger besitzt eine in Schichten aufgebaute Struktur; diese ermöglicht die Erweiterung um und die Neukombination einzelne Teile. Die Brücke zur Ausführungsplattform bildet dabei die Treiberschicht, die die Fähigkeiten von Debuggern auf Quelltextebene in wohldefinierte Schnittstellen bündelt. Der Debugger ermöglicht dem Nutzer die Instanziierung verschiedener Sichten auf das ausgeführte Modell. Die Sichttypen sind in *Viewpoints* zusammengestellt, welche wieder aus Komponenten zur Steuerung, zur Gewinnung des Laufzeitmodells und zur Präsentation der Informationen aufgeteilt sind.

Für die Treiberarchitektur wurden mehrere heterogene, für eingebettete Software relevante Zielsysteme integriert. Für eingebettete Prozessoren wurden Schnittstellen zum GNU Debugger, zum Java Debugger und zu einem proprietären Emulatorsystem realisiert. Diese setzen das klassische Debugging-Paradigma um. Für das Emulatorsystem wurde ein echtzeitfähiger Treiber erstellt, welcher eine bidirektionale Navigation auf der virtuellen Zeitachse des Trace ermöglicht. Schliesslich wurden rekonfigurierbare Bausteine, hier FPGAs in das Treiberkonzept integriert, deren Funktion modellgetrieben spezifiziert wurde. Aus dem Modell kann eine Schnittstelle generiert werden, über die ein angepasster Treiber Debugging von in FPGAs realisierten Modellen ermöglicht.

Zur Exploration einer zweiten Achse der Heterogenität wurden mehrere Systemsichten untersucht und realisiert. Hier wurde die Animation reaktiver hier-

archischer Zustandsautomaten (*Statecharts*) umgesetzt, für die Quelltext generiert wurde. Die Inspektion des Zustands einer mit UML Actions realisierten Verhaltensbeschreibungen wurde in die Plattform integriert, sowie die Visualisierung der Konfiguration von Klasseninstanzen zur Laufzeit mit einem Objektdiagramm.

Die Plattform zeichnet demnach durch die Definition von Treibern und Viewpoints die zwei Dimensionen des Begriffs Heterogenität nach, welcher die Vereinigung unterschiedlicher Notationen und unterschiedlicher Plattformen in einem System bezeichnet.

ModelScope erfüllt weitgehend die in Abschnitt 4.5.3 beschriebenen Anforderungen an eine Plattform zum Debugging ausführbarer graphischer Modelle und kann als Basis für die Integration weiterer Zielsysteme und Sichten dienen.

8.2 Ausblick

Ein weites Feld eröffnet die Erweiterung von *ModelScope* um zusätzliche Systemsichten in Form von *Viewpoints*. Hier ist insbesondere die Integration von ausführbaren Signalflussmodellen der Signalverarbeitung und Regelungstechnik zu nennen, wie sie in Matlab Simulink und integrierten Codegeneratoren realisiert sind. Weiterhin wäre eine Darstellung von Klassendiagrammen hilfreich, da sie zwar keine dynamischen Informationen darstellen können, jedoch zur Steuerung des Systems aus Sicht eines objektorientierten Modells einsetzbar sind.

Zusätzliche dynamische Information über den Systemzustand kann die Visualisierung der Aufrufliste (*Call-Stack*) als Sequenzdiagramm bieten, wobei die beteiligten Objekte klar ersichtlich wären. Die Visualisierung von hierarchischen Zustandsautomaten könnte auf implizite, das Gesamtsystem überspannende Zustandsmodelle und das Verhalten von Klassen beschreibende Zustandsmaschinen erweitert werden.

Bei der Nutzung der Visualisierungen für umfangreichere industrielle Projekte stellt sich schnell die Frage der Übersichtlichkeit einzelner Diagramme. Hier wäre eine Untersuchung der Anwendbarkeit von *Slicing*-Methoden (siehe Abschnitt 4.3.4.1) interessant, um den Benutzer kognitiv zu entlasten. Weiterhin können die implementierten Algorithmen zum Layout von Graphen so erweitert werden, dass eine inkrementelle minimale Veränderung der Darstellung über die Zeit angestrebt wird.

Als spezielle Sicht ist die Integration von Quelltext in das Debugging-Umfeld

interessant, da auch in Zukunft reale Projekte immer Quelltextanteile aufweisen werden und bei Nutzung von domänenspezifischen Sprachen häufig auch die Funktion der Modell-zu-Text-Transformation validiert werden muss.

Im Bereich der Zielsysteme ist eine größere Anzahl Trace-fähiger Treiber für CPUs und rekonfigurierbare Hardware wünschenswert, da sie ein bidirektionales Debugging ermöglichen. Neben der Nutzung spezieller Hardware, ist ein generischer, das Modell instrumentierender Treiber denkbar.

Anhang A

Beispiele UML Actions

A.1 Codegenerierung nach Java

Der hier aufgeführte Quellcode ist mit dem in Abschnitt 3.2.3 beschriebenen Codegenerator für Action Semantics entstanden. Das Beispiel besteht aus der Klasse *Prime* in Listing A.1, die die Funktionalität bereitstellt und der Klasse *Main* in Listing A.2, die die Main-Methode enthält.

Der in der Methode *testPrime* implementierte Algorithmus kann überprüfen, ob es sich bei einer eingegebenen Zahl um eine Primzahl handelt oder nicht. Das Action-Modell umfasst 70 Metaobjekte, welche miteinander verknüpft sind (siehe auch [Gutstein 2007]).

Listing A.1: Die Klasse Prime

```
package P1;

class Prime {

  // Methods
  static boolean testPrime(int numberToTest) {
    //$[bfa80164111f9a625c29] Operation:testPrime
    int PIN0;
    PIN0 = numberToTest;
    //$[bfa80164111f9f37f261] GroupAction:group_prime
    boolean PIN1;
    {
     boolean isPrime;
     //$[bfa80164111f9f37f263] LiteralValueAction:litVal_true
     boolean PIN2;
     PIN2 = true;
```

```
//§[bfa80164111f9f37f267] AddVariableValueAction:...
    ...addVarValue
isPrime = PIN2;
//§[bfa80164111fa0a51dd7] LiteralValueAction:litVal_2
int PIN3;
PIN3 = 2;
//§[bfa80164111fa0a51dd1] LoopAction:loop
int PIN4;
PIN4 = PIN3;
//§[bfa80164111fa0a51dd9] ApplyFunctionAction:loop_test
boolean PIN5;
PIN5 = PIN4 < PIN0;
while (PIN5)
{
  int PIN6;
  //§[bfa80164111fa0a51ddd] GroupAction:loop_body_group
  {
    //§[bfa80164111fa0a51dd11] ConditionalAction:...
        ...conditional
    boolean PIN7;
    //§[bfa80164111fa0a51dd13] GroupAction:condTest
    {
      //§[bfa80164111fa0a51dd18] ApplyFunctionAction:...
          ...applyMod
      int PIN8;
      PIN8 = PIN0 % PIN4;
      //§[bfa80164111fb13df20d] LiteralValueAction:litVal0
      int PIN9;
      PIN9 = 0;
      //§[bfa80164111fa0a51dd19] ApplyFunctionAction:...
          ...applyEqual
      PIN7 = PIN8 == PIN9;
    }
    if (PIN7)
    {
      //§[bfa80164111fa0a51dd14] GroupAction:condBody
      {
        //§[bfa80164111fa0a51dd1a] LiteralValueAction:...
            ...litVal_false
        boolean PIN10;
        PIN10 = false;
        //§[bfa80164111fa0a51dd1b] AddVariableValueAction:...
            ...addVarValueFalse
        isPrime = PIN10;
      }
    }
    //§[bfa80164111fb13df2011] LiteralValueAction:litVal1
    int PIN11;
    PIN11 = 1;
```

```
            //§[bfa80164111fb13df2010] ApplyFunctionAction:...
                ...applyAddLoopVar++
            PIN6 = PIN4 + PIN11;
          }
          PIN4 = PIN6;
          //§[bfa80164111fa0a51dd9] ApplyFunctionAction:loop_test
          PIN5 = PIN4 < PIN0;
        }
        //§[bfa80164111f9f37f26a] ReadVariableAction:readVar
        PIN1 = isPrime;
      }
      return PIN1;
    }
}
```

Listing A.2: Die Klasse Main

```
package P1;

public class Main {

  // Methods
  public static void main(String[] arguments) {
    //§[bfa8016411128531103d1] Operation:main
    String[] PIN0;
    PIN0 = arguments;
    //§[ac158d1810f95bd1ceb3] GroupAction:groupaction
    {
      //§[bfa80164111f9a625c210] LiteralValueAction:litVal
      int PIN1;
      PIN1 = 7;
      //§[bfa80164111f9a625c211] CallOperationAction:...
          ...callOpTestPrime
      boolean PIN2;
      PIN2 = Prime.testPrime(PIN1);
    }
  }
}
```

A.2 Reversierung einer Java-Methode

Der in Abschnitt 3.2.4 beschriebene Ansatz ist in der Lage, ein Action-Modell aus einer Prozedurbeschreibung in Java-Syntax zu erzeugen.

Am Beispiel einer If-Anweisung sollen im Folgenden die im Abschnitt 3.2.3 entwickelten Transformationsvorschriften angewendet werden:

Listing A.3: Verhaltensbeschreibung in Java

```
if (a > 0) {
    a := 10 - a*2 ;
    a := -a ;
} else {
    a ++ ;
}
```

Am in Abbildung A.1 bis Abbildung A.4 gezeigten Objektdiagramm lässt sich nachvollziehen, wie die Strukturen zur Beschreibung des gleichen Verhaltens transformiert wurden. Aus den Abbildungen ist auch ersichtlich, dass die abstrakte Syntax selbst keine geeignete Notation zur Verhaltensbeschreibung. Das Beispiel wurde ausführlich in [Koschuhar 2007] dokumentiert.

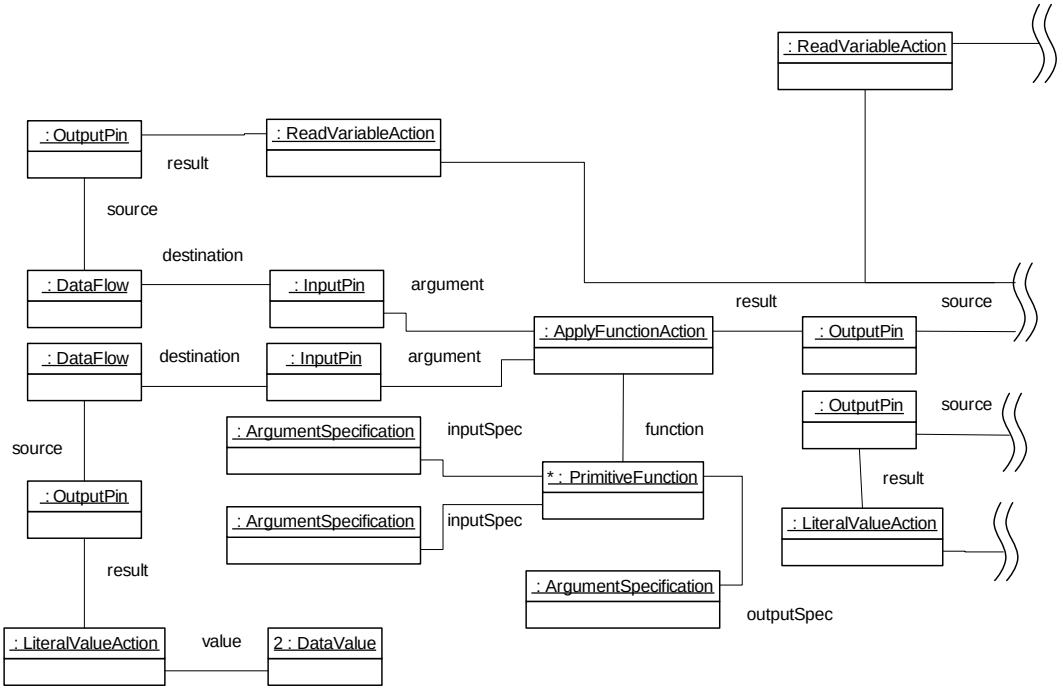

Abbildung A.1: Darstellung der If-Anweisung in Action Semantics – Teil 1

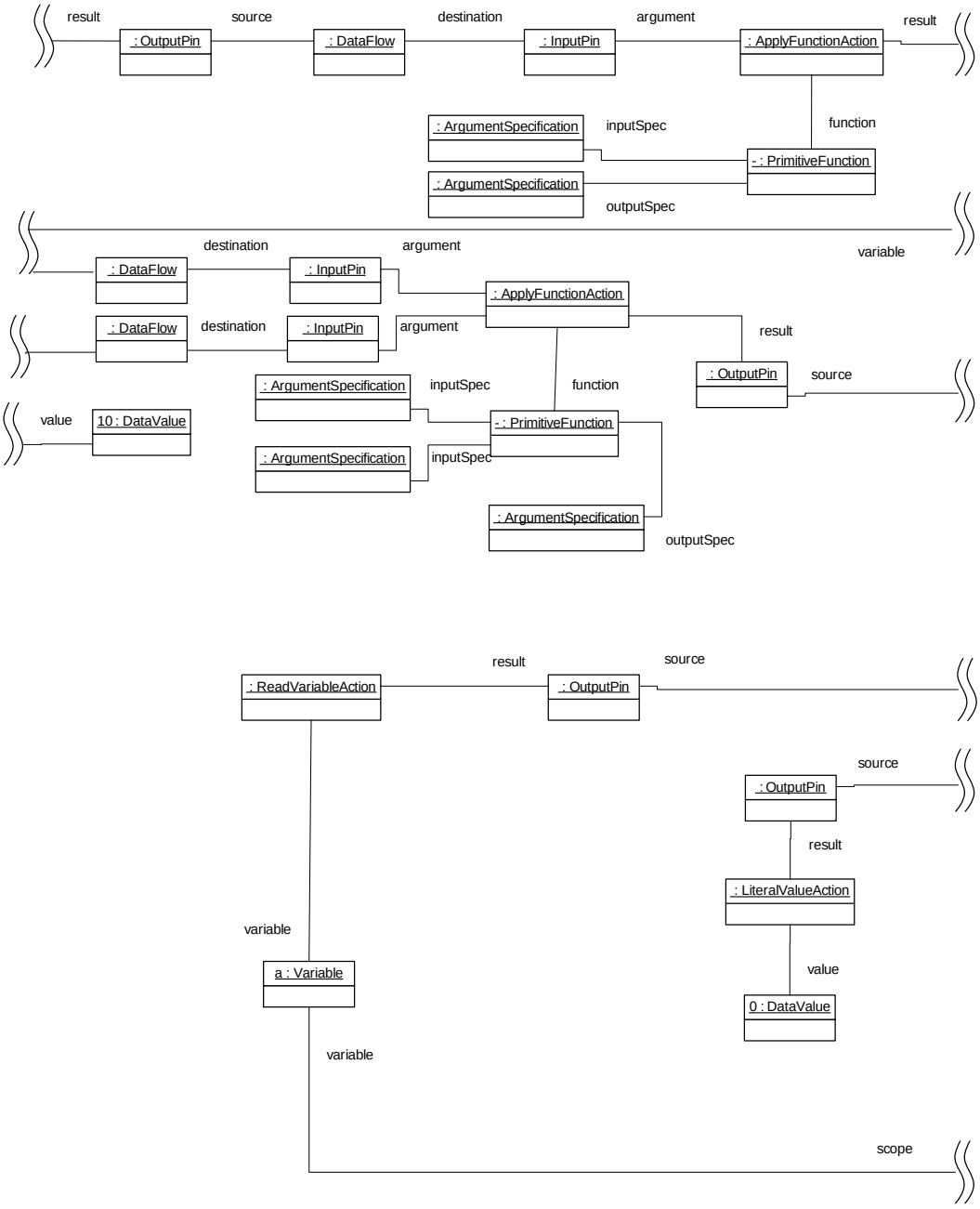

Abbildung A.2: Darstellung der If-Anweisung in Action Semantics – Teil 2

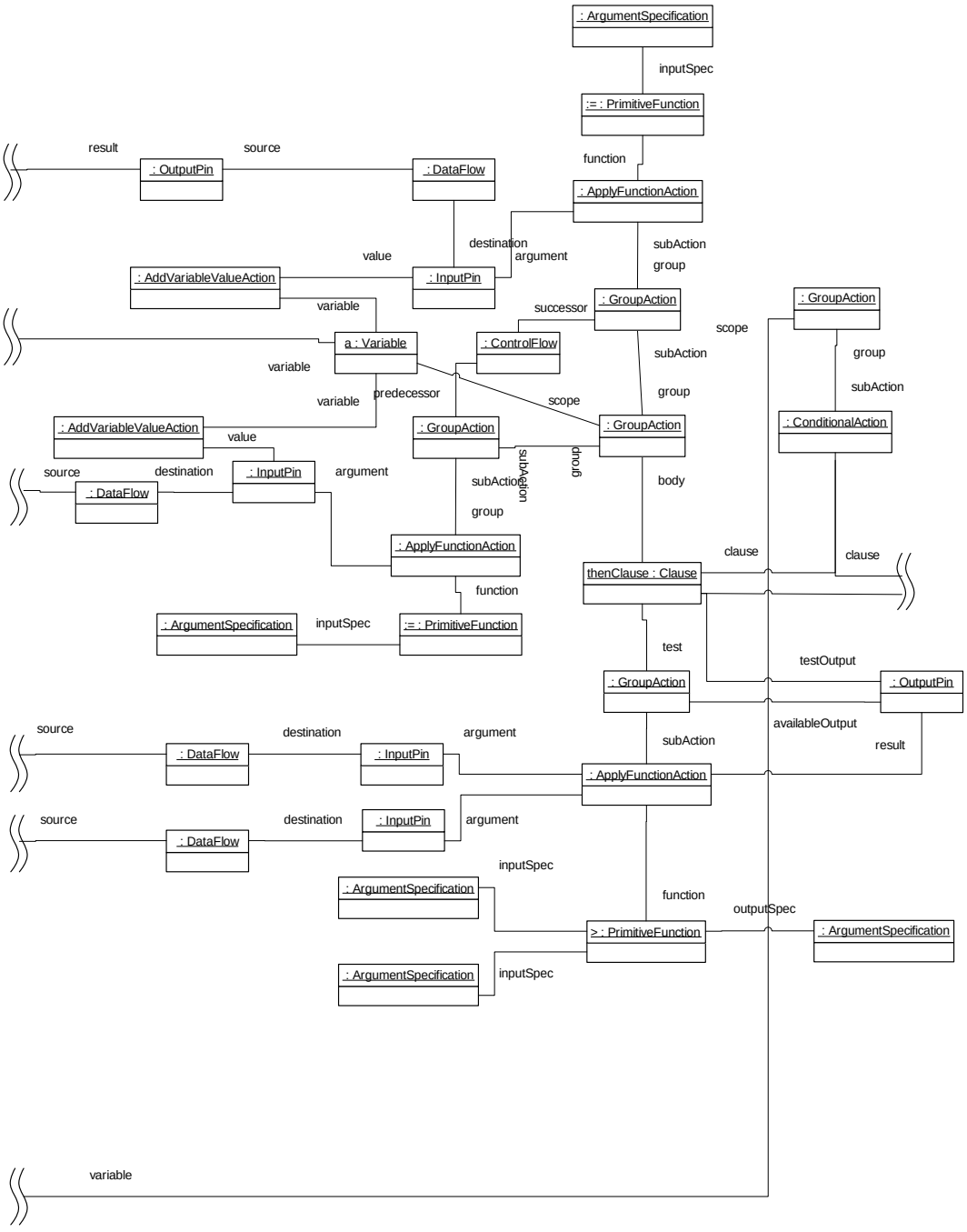

Abbildung A.3: Darstellung der If-Anweisung in Action Semantics – Teil 3

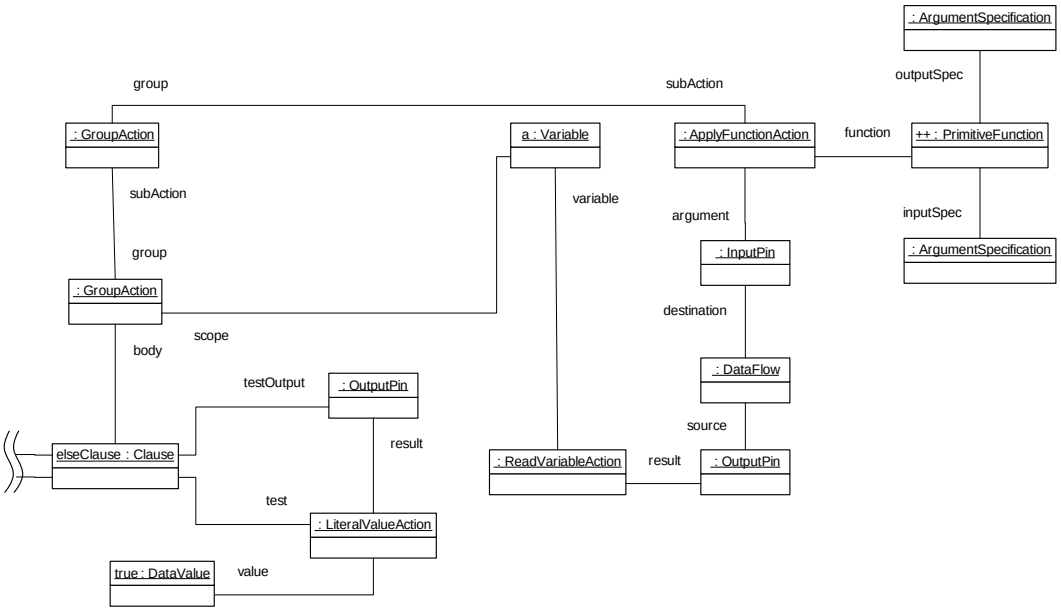

Abbildung A.4: Darstellung der If-Anweisung in Action Semantics – Teil 4

Literaturverzeichnis

[Agans 2002] Agans, David J.: *Debugging : The Nine Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems.* AMACOM, 2002

[Agrawal u. a. 1993] Agrawal, H. ; Demillo, R. ; Spafford, E.: *Debugging with Dynamic Slicing and Backtracking.* `citeseer.comp.nus.edu.sg/agrawal93debugging.html`. Version: 1993

[Alekseev 2007] Alekseev, Sergej: Java debugging laboratory for automatic generation and analysis of trace data. In: *SE'07: Proceedings of the 25th conference on IASTED International Multi-Conference.* Anaheim, CA, USA : ACTA Press, 2007, S. 177–182

[Ball 2000] Ball, Stuart R.: *Embedded Microprocessor Systems: Real World Design.* Newton, MA, USA : Butterworth-Heinemann, 2000

[Balzert 2005] Balzert, Helmut: *Lehrbuch Grundlagen der Informatik.* 2. Heidelberg : Spektrum Akademischer Verlag, 2005

[Bastert u. Matuszewski 2001] Bastert, Oliver ; Matuszewski, Christian: Layered drawings of digraphs. (2001), S. 87–120

[Battista u. a. 1998] Battista, Giuseppe D. ; Eades, Peter ; Tamassia, Roberto ; Tollis, Ioannis G.: *Graph Drawing: Algorithms for the Visualization of Graphs.* Upper Saddle River, NJ, USA : Prentice Hall PTR, 1998

[Berent 2007] Berent, Anthony: Should Embedded Debug be Standardized? In: *ECSI Workshop on SoC Debug Standards*, 2007

[Booth u. Jones 1997] Booth, Simon P. ; Jones, Simon B.: Walk Backwards to Happiness - Debugging by Time Travel. In: *Automated and Algorithmic Debugging*, 1997, 171-183

[Bray u. a. 1998] Bray, Tim ; Paoli, Jean ; Sperberg-McQueen, C.M.: *Extensible Markup Language (XML) 1.0.* W3C Recommendation 10 February 1998, http://www.w3.org/TR/1998/REC-xml-19980210, 1998

[Camera u. a. 2005] Camera, Kevin ; So, Hayden Kwok-Hay ; Brodersen, Robert W.: An integrated debugging environment for reprogrammble hardware systems. In: *AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging.* New York, NY, USA : ACM Press, 2005, S. 111–116

[Cleve u. Zeller 2000] Cleve, Holger ; Zeller, Andreas: *Finding Failure Causes through Automated Testing.* 2000

[Coleman u. Parker 1996] Coleman, Michael K. ; Parker, D. S.: Aesthetics-based graph layout for human consumption. In: *Softw. Pract. Exper.* 26 (1996), Nr. 12, S. 1415–1438

[Cormen u. a. 2001] Cormen, Thomas H. ; Leiserson, Charles E. ; Rivest, Ronald L. ; Stein, Clifford: *Introduction to Algorithms, Second Edition.* The MIT Press, 2001

[Danis 1997] Danis, Sharron A.: *Rear Admiral Grace Murray Hopper.* http://ei.cs.vt.edu/ history/Hopper.Danis.html, Februar 1997

[Daum 2004] Daum, Berthold: *Java-Entwicklung mit Eclipse 3: Anwendungen, Plugins und Rich Clients.* 2., überarb. und erw. Aufl. Heidelberg : dpunkt-Verlag, 2004

[DEC Corp. 1986] DEC Corp.: *AA-BH82B-TB TOPS-10 DDT Manual.* Marlboro, 1986. `http://www.bitsavers.org/pdf/dec/pdp10/TOPS10_softwareNotebooks/vol13/AA-BH82B-TB_ddt.pdf`

[Dirckze 2002] Dirckze, Ravi: *Java™ Metadata Interface(JMI) Specification.* June 2002

[Dodani 2006] Dodani, Mahesh H.: *A Picture is Worth a 1000 Words?* http://www.jot.fm/issues/issue_2006_03/column4/, 2006

[Dreier u. a. 2003a] Dreier, Rico ; Dummer, Georg ; Zhang, Guoxing ; Müller-Glaser, Klaus D.: Partitioning and FPGA-Based Co-Simulation of Statecharts. In: *15th Annual European Simulation Symposium (ESS'2003).* Delft, The Netherlands, October 2003

[Dreier u. a. 2003b] Dreier, Rico ; Dummer, Georg ; Zhang, Guoxing ; Müller-Glaser, Klaus D.: Partitioning and FPGA-Based Co-Simulation

of Statecharts. In: *Proceedings of the 15th Annual European Simulation Symposium (ESS'2003)*. Delft, The Netherlands, October 2003

[Eclipse Foundation 2008a] Eclipse Foundation: *Eclipse*. http://www.eclipse.org, 2008

[Eclipse Foundation 2008b] Eclipse Foundation: *Graphical Editor Framework*. http://www.eclipse.org/gef, 2008

[Edlich 2002] Edlich, Stefan: *Ant kurz & gut*. 1. Köln : O'Reilly Verlag, 2002

[Eiglsperger u. a. 2001] Eiglsperger, Markus ; Fekete, Sándor P. ; Klau, Gunnar W.: Orthogonal graph drawing. (2001), S. 121–171

[Elms 1997] Elms, Kim: Debugging Optimised Code Using Function Interpretation. In: *Automated and Algorithmic Debugging*, 1997, 27-36

[Fowler 2003] Fowler, Martin: *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2003

[Frank 1994] Frank, P. M.: Diagnoseverfahren in der Automatisierungstechnik. In: *at - Automatisierungstechnik* 42 (1994), S. 47–64

[Gamma u. Beck 2003] Gamma, Erich ; Beck, Kent: *Contributing to Eclipse: Principles, Patterns, and Plug-Ins*. 1st. Addison Wesley Professional., 2003 (The Eclipse Series.)

[Gamma u. a. 2004] Gamma, Erich ; Helm, Richard ; Johnson, Ralph E.: *Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software*. München : Addison-Wesley, 2004

[Geiger u. Zündorf 2006] Geiger, L. ; Zündorf, A.: eDOBS - Graphical Debugging for eclipse. In: *3rd International Workshop on Graph-Based Tools (GraBaTs)*. Natal, Brasil, September 2006

[Gestwicki 2005] Gestwicki, Paul V.: *Interactive Visualization of Object-Oriented Programs*, Faculty of the Graduate School of University at Buffalo The State University of New York, Dissertation, Juni 2005

[Gilmore u. Shebs 2006] Gilmore, John ; Shebs, Stan: *GDB Internals, A guide to the internals of the GNU debugger*. http://www.gnu.org/software/gdb/documentation, 2006

[Glass 2006] Glass, Robert L.: The Standish report: does it really describe a software crisis? In: *Commun. ACM* 49 (2006), Nr. 8, S. 15–16

[Gracanin u. a. 2005] Gracanin, Denis ; Matkovic, Kresimir ; Eltoweissy, Mohamed: Software visualization. In: *ISSE* 1 (2005), Nr. 2, S. 221–230

[Graf u. Müller-Glaser 2006a] Graf, Philipp ; Müller-Glaser, Klaus D.: Eine Architektur für die modellbasierte Fehlersuche in der Software eingebetteter Systeme. In: *Modellierung 2006, Workshop: Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen.* Innsbruck, Österreich, März 2006

[Graf u. Müller-Glaser 2006b] Graf, Philipp ; Müller-Glaser, Klaus D.: Dynamic Mapping of Runtime Information Models for Debugging Embedded Software. In: *17th IEEE International Workshop on Rapid System Prototyping.* Chania, Griechenland, Juni 2006

[Graf u. Müller-Glaser 2007] Graf, Philipp ; Müller-Glaser, Klaus D.: Gaining Insight into Executable Models during Runtime: Architecture and Mappings. In: *IEEE Distributed Systems Online* 8 (2007), Nr. 3. – art. no. 0703-o3001

[Graf u. Müller-Glaser 2008] Graf, Philipp ; Müller-Glaser, Klaus D.: ModelScope – Inspecting Executable Models during Run-time. In: *Proceedings of 30th International Conference on Software Engineering*, 2008

[Graf u. a. 2004] Graf, Philipp ; Reichmann, Clemens ; Müller-Glaser, Klaus D.: Towards a Platform for Debugging Executed UML-Models in Embedded Systems. In: *UML Modelling Languages and applications: UML 2004.* Lissabon, Portugal : Springer, Oktober 2004

[Graf u. a. 2005] Graf, Philipp ; Reichmann, Clemens ; Müller-Glaser, Klaus D.: Model-Based Design of Functionality and QoS-Properties for a Dynamically Reconfigurable Architecture. In: , Oktober 2. Paderborn (Hrsg.): *3rd Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER 3)*, 2005

[Grönniger u. a. 2006] Grönniger, Hans ; Krahn, Holger ; Rumpe, Bernhard ; Schindler, Martin: Integration von Modellen in einen codebasierten Softwareentwicklungsprozess. In: Mayr, Heinrich C. (Hrsg.) ; Breu, Ruth (Hrsg.): *Modellierung* Bd. 82, GI, 2006 (LNI), S. 67–81

[Grose u. a. 2001] Grose, Timothy J. ; Doney, Gary C. ; Brodsky, Stephen A.: *Mastering XMI: Java Programming with XMI, XML and UML.* New York, NY, USA : John Wiley & Sons, Inc., 2001

[Hadlich 1997] Hadlich, T.: Proposing Graphic Extensions to VHDL. In: *VIUF '97: Proceedings of the 1997 VHDL International User's Forum (VIUF '97).* Washington, DC, USA : IEEE Computer Society, 1997, S. 109

[Haller 1997] Haller, Craig A.: *The Zen of BDM*. http://www.macraigor.com/downloads/zenofbdm.pdf, 1997

[Harel 1987] Harel, David: Statecharts: A Visual Formalism for Complex Systems. In: *Science of Computer Programming* 8 (1987), S. 231–274

[Hartenstein 2007] Hartenstein, Rainer: Basics of Reconfigurable Computing. In: Henkel, Joerg (Hrsg.) ; Parameswaran, Sridevan (Hrsg.): *Designing Embedded Processors. A Low Power Perspective*. Heidelberg : Springer, 2007

[Hobatr u. Malloy 2001] Hobatr, Chanika ; Malloy, Brian A.: The design of an OCL query-based debugger for C++. In: *SAC '01: Proceedings of the 2001 ACM symposium on Applied computing*. New York, NY, USA : ACM, 2001, S. 658–662

[Hooman u. Hendriks 2007] Hooman, Jozef ; Hendriks, Teun: Model-Based Run-Time Error Detection. In: *Proceedings of the Models Workshop on Models@Runtime*. Nashville, USA, Oktober 2007

[Hudak 1998] Hudak, Paul: Modular Domain Specific Languages and Tools. In: Devanbu, P. (Hrsg.) ; Poulin, J. (Hrsg.): *Proceedings: Fifth International Conference on Software Reuse*, IEEE Computer Society Press, 1998, 134–142

[IEEE 1990a] IEEE: IEEE standard glossary of software engineering terminology. Version: 1990. `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=159342`. 1990. – Forschungsbericht

[IEEE 1990b] IEEE: IEEE Standard Test Access Port and Boundary - Scan Architecture. Version: 1990. `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=211226`. 1990. – Forschungsbericht

[IEEE 1994] IEEE: *IEEE Standard Classification for Software Anomalies*. IEEE Std. 1044-1993, IEEE Computer Society, 1994

[Institute of Electrical and Electronics Engineers 1988] Institute of Electrical and Electronics Engineers: IEEE standard VHDL language reference manual. Version: 1988. `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=26487`. 1988. – Forschungsbericht

[iSystem 2008] iSystem: *iSystem.connect*. http://www.isystem.com/content/91/44/, 2008

[Jacobs u. Musial 2003] Jacobs, Timothy ; Musial, Benjamin: Interactive visual debugging with UML. In: *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*. New York, NY, USA : ACM Press, 2003, S. 115–122

[Jeckle u. a. 2003] Jeckle, Mario ; Rupp, Chris ; Hahn, Jürgen ; Zengler, Barbara ; Queins, Stefan: *UML 2 glasklar*. Hanser Fachbuchverlag, 2003

[Jones 2006] Jones, Capers: *Programming Languages Table, Version 8.2*. Software Productivity Research Inc., available on-line at http://www.spr.com/library/langtbl.htm, 2006

[Karagiannis u. Kühn 2002] Karagiannis, Dimitris ; Kühn, Harald: Metamodelling Platforms. In: *EC-WEB '02: Proceedings of the Third International Conference on E-Commerce and Web Technologies*. London, UK : Springer-Verlag, 2002, S. 182

[Karcher 2008] Karcher, Matthias: *On-Chip Debugging versus In-Circuit-Emulation*. http://www.hitex.de/news/articles/con_article_on-chip-debugging-versus-in-circuit-emulation.html, 2008

[Krüger 2002] Krüger, Guido: *Handbuch der Java-Programmierung*. 3. München; Boston [u.a.] : Addison-Wesley, 2002

[Lencevicius 2000] Lencevicius, Raimondas: *On-the-fly Query-Based Debugging with Examples*. 2000

[Lettrari u. Klose 2001] Lettrari, Marc ; Klose, Jochen: Scenario-Based Monitoring and Testing of Real-Time UML Models. In: *«UML» '01: Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*. London, UK : Springer-Verlag, 2001, S. 317–328

[Lewis 2003] Lewis, Bil: *Debugging Backwards in Time*. 2003

[Liang 1999] Liang, Sheng: *Java Native Interface: Programmer's Guide and Reference*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1999

[Lieberman 1997] Lieberman, Henry: Introduction. In: *Commun. ACM* 40 (1997), Nr. 4, S. 26–29

[Lucia 2001] Lucia, Andrea D.: Program slicing: methods and applications. In: *Source Code Analysis and Manipulation, 2001. Proceedings. First IEEE International Workshop on* (2001), S. 142–149

[Malloy u. Power 2005] Malloy, Brian A. ; Power, James F.: Exploiting UML dynamic object modeling for the visualization of C++ programs. In: *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*. New York, NY, USA : ACM Press, 2005, S. 105–114

[Martin u. a. 1997] Martin, Robert C. ; Riehle, Dirk ; Buschmann, Frank: *Pattern languages of program design 3*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1997

[Maruyama u. Terada 2003] Maruyama, Kazutaka ; Terada, Minoru: *Timestamp Based Execution Control for C and Java Programs*. 2003

[Mehner u. Wagner 2000] Mehner, Katharina ; Wagner, Annika: Visualisierung der Synchronisation von Java-Threads mit UML. In: Ebert, Jürgen (Hrsg.) ; Frank, Ulrich (Hrsg.): *Modelle und Modellierungssprachen in Informatik und Wirtschaftsinformatik. Beiträge des Workshops Modellierung 2000', St. Goar, 5. - 7. April 2000.*, Koblenz: Fölbach, 2000 (Koblenzer Schriften zur Informatik, Band 15)

[Mellor u. Balcer 2002] Mellor, Stephen J. ; Balcer, Marc: *Executable UML: A Foundation for Model-Driven Architectures*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2002

[Mellor u. a. 1998] Mellor, Stephen J. ; Tockey, Steve ; Arthaud, Rodolphe ; LeBlanc, Philippe: Software-platform-independent, Precise Action Specifications for UML. In: Bézivin, Jean (Hrsg.) ; Muller, Pierre-Alain (Hrsg.): *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998*, 1998, S. 281–286

[Mernik u. a. 2005] Mernik, Marjan ; Heering, Jan ; Sloane, Anthony M.: When and how to develop domain-specific languages. In: *ACM Comput. Surv.* 37 (2005), Nr. 4, S. 316–344

[Metzger 2004] Metzger, Robert C.: *Debugging by Thinking : A Multidisciplinary Approach*. Amsterdam : Elsevier Digital Press, 2004

[Müller-Glaser u. a. 2004] Müller-Glaser, K.D. ; Reichmann, C. ; Graf, P. ; Kühl, M. ; Ritter, K.: Heterogenous Modeling for Automotive Electronic Control Units using a CASE-Tool Integration Platform. In: *Conference on Computer Aided Control Systems Design*. Taipei, Taiwan, September 2004

[Moore u. a. 2004] Moore, Bill ; Dean, David ; Gerber, Anna ; Wagenknecht, Gunnar ; Vanderheyden, Philippe: *Eclipse Development*

using the Graphical Editing Framework and the Eclipse Modeling Framework. ibm.com/redbooks, February 2004

[Muller u. Barais 2007] Muller, Pierre-Alain ; Barais, Olivier: Control-theory and models at runtime. In: *Proceedings of the Models Workshop on Models@Runtime*. Nashville, USA, Oktober 2007

[Myers 1991] Myers, Glenford J.: *Methodisches Testen von Programmen*. 4. München [u.a.] : Oldenbourg, 1991 (Reihe Datenverarbeitung)

[Naval Historical Centre 1988] Naval Historical Centre: *Online Library of Selected Images, Photo: NH 96566-KN (Color), The First C̈omputer Bug¨*. http://www.history.navy.mil/photos/images/h96000/h96566kc.htm, 1988

[Nexus 5001 1999] Nexus 5001: *The Nexus 5001 Forum Standard for a Global Embedded Processor Debug Interface*. December 1999

[Object Management Group (OMG) 2001] Object Management Group (OMG): *Meta Object Facility (MOF) Specification, Version 1.4*. 2001

[Object Management Group (OMG) 2003a] Object Management Group (OMG): *MDA Guide Version 1.0.1*. ftp://ftp.omg.org/pub/docs/omg/03-06-01.pdf, Juni 2003

[Object Management Group (OMG) 2003b] Object Management Group (OMG): *Unified Modeling Language (UML) Specification, Version 1.5*. 2003

[Object Management Group (OMG) 2004] Object Management Group (OMG): *Meta Object Facility (MOF) 2.0 Core Specification*. Oktorber 2004

[Object Management Group (OMG) 2005a] Object Management Group (OMG): *MOF 2.0/XMI Mapping Specification, v2.1*. September 2005

[Object Management Group (OMG) 2005b] Object Management Group (OMG): *OCL 2.0 Specification Version 2.0 ptc/2005-06-06*. June 2005

[Object Management Group (OMG) 2005c] Object Management Group (OMG): *Unified Modeling Language: Diagram Interchange*. June 2005

[Object Management Group (OMG) 2005d] Object Management Group (OMG): *Unified Modeling Language: Superstructure*. August 2005

[Object Management Group (OMG) 2007] Object Management Group (OMG): *Unified Modeling Language (UML) Specification: Infrastructure version 2.1.1*. Februar 2007

[O'Keeffe 2006] O'Keeffe, Hugh: *Embedded Debugging, A White Paper.* http://www.ashling.com/images/stories/pdfs/technicalarticles/apb179-nexusbooklet.pdf, 2006

[O'Keeffe 2008] O'Keeffe, Hugh: *The NEXUS Debug Standard: Gateway to the Embedded Systems of the Future.* http://www.ashling.com/images/stories/pdfs/technicalarticles/apb179-nexusbooklet.pdf, 2008

[Parhami 1997] Parhami, B.: Defect, Fault, Error,..., or Failure? In: *IEEE Transactions on Reliability* 46 (1997), December, Nr. 4, S. 450–451

[Pomberger u. Rechenberg 1997] Pomberger, G. ; Rechenberg, P.: *Handbuch der Informatik.* München : Carl Hanser Verlag, 1997

[Popper 1994] Popper, Karl R.: *Logik der Forschung.* 10. Auflage. Mohr, 1994

[Purchase u. a. 2001] Purchase, Helen C. ; Allder, Jo-Anne ; Carrington, David A.: User Preference of Graph Layout Aesthetics: A UML Study. In: *GD '00: Proceedings of the 8th International Symposium on Graph Drawing.* London, UK : Springer-Verlag, 2001, S. 5–18

[Ray 2001] Ray, Eric T.: *Einführung in XML.* O'Reilly, Köln, 2001

[Reichmann u. a. 2003a] Reichmann, C. ; Graf, P. ; Kühl, M. ; Müller-Glaser, K.D.: GeneralStore – Eine CASE-Tool Integrationsplattform für den durchgängigen Entwurf eingebetteter Systeme. In: Spath, K. D.; H. D.; Haasis (Hrsg.): *Aktuelle Trends in der Softwareforschung, Tagungsband zum doIT Software-Forschungstag am 18. November 2003*, Fraunhofer IRB-Verlag, 2003

[Reichmann u. a. 2003b] Reichmann, C. ; Graf, P. ; Müller-Glaser, K.D.: Ein durchgängiger Ansatz vom Systementwurf zu Codegeneration auf Basis eines objektorientierten Modells für eingebettete heterogene Systeme. In: *Tagungsband EKA 2003 - Entwurf komplexer Automatisierungssysteme.* Braunschweig, Juni 2003

[Reichmann u. a. 2003c] Reichmann, C. ; Graf, P. ; Müller-Glaser, K.D.: Ein durchgängiger Ansatz vom Systementwurf zur Codegeneration auf Basis eines objektorientierten Modells für eingebettete heterogene Systeme. In: *atp, Automatisierungstechnische Praxis* 45 (2003), Juni, Nr. 6, S. 45–52

[Reichmann 2005] Reichmann, Clemens: *Grafisch notierte Modell-zu-Modell-Transformationen für den Entwurf eingebetteter elektronischer Systeme.* Aachen : Shaker, 2005

[Reichmann u. a. 2005] Reichmann, Clemens ; Graf, Philipp ; Gebauer, Daniel ; Müller-Glaser, Klaus D.: Heterogene Software Modellierung für ein chemisches Analysegerät mit GeneralStore – eine Fallstudie. In: *5. GMM/ITG/GI-Workshop Multi-Nature Systems*. Dresden, Februar 2005

[Reichmann u. a. 2004a] Reichmann, Clemens ; Graf, Philipp ; Kühl, Markus ; Müller-Glaser, Klaus D.: Automatisierte Modellkopplung heterogener eingebetteter Systeme. In: *GI PEARL 2004, Eingebettete Systeme*, 2004

[Reichmann u. a. 2004b] Reichmann, Clemens ; Kühl, Markus ; Graf, Philipp ; Müller-Glaser, Klaus D.: GeneralStore - A CASE-tool integration platform enabling model level coupling of heterogeneous designs for embedded electronic systems. In: *Proceedings of the 11th IEEE International Conference on the Engineering of Computer-Based Systems*. Brno, Tschechische Republik : Springer, 2004

[Ronsse u. a. 2000] Ronsse, Michiel ; De Bosschere, Koen ; de Kergommeaux, Jacques C.: *Execution replay and debugging*. 2000

[Rosenberg u. a. 1998] Rosenberg, Linda ; Hammer, Ted ; Shaw, Jack: Software Metrics and Reliability. In: *Proceedings of the 9th International Symposium on Software Reliability Engineering*, 1998

[Sajaniemi u. Kuittinen 2003] Sajaniemi, Jorma ; Kuittinen, Marja: Program animation based on the roles of variables. In: *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*. New York, NY, USA : ACM Press, 2003, S. 7–ff

[Schäuffele u. Zurawka 2004] Schäuffele, Jörg ; Zurawka, Thomas: *Automotive Software Engineering*. Vieweg, 2004

[Schumann 2000] Schumann, Johann: *Automatic Debugging Support for UML Designs*. 2000

[Somogyi 2003] Somogyi, Zoltan: *Idempotent I/O for safe time travel*. 2003

[SPIRIT Schema Working Group Membership 2005] SPIRIT Schema Working Group Membership: *SPIRIT User Guide v1.1*. http://www.spiritconsortium.org, Juni 2005

[SPIRIT Schema Working Group Membership 2006] SPIRIT Schema Working Group Membership: *IP-XACT User Guide v1.2*. http://www.spiritconsortium.org, Juli 2006

[Stahl u. Völter 2005] Stahl, Thomas ; Völter, Markus: *Modellgetriebene Softwareentwicklung.* dpunkt.verlag, 2005

[Stallman u. a. 2006] Stallman, Richard ; Pesch, Roland ; Shebs, Stan: *Debugging with GDB, The GNU Source-Level Debugger, Ninth Edition, for GDB version 6.8.50.20080407.* http://www.gnu.org/software/gdb/documentation, 2006

[Standish Group International 2004] Standish Group International: *The Chaos Report 2004.* http://www.standishgroup.com/sample_research/PDFpages/q3-spotlight.pdf, 2004

[Stollon u. a. 2007] Stollon, Neal ; Uvacek, Bob ; Laurenti, Gilbert: Defining standard Debug Interface Socket requirements for OCP-Compliant multicore SoCs: Part 1. In: *http://www.embedded.com/design/multicore/201000620* (2007)

[Stroustrup 2000] Stroustrup, Bjarne: *Die C++ Programmiersprache.* 4. München : Addison-Wesley Verlag, 2000

[Sun Microsystems 2002] Sun Microsystems: *jdb - The Java Debugger.* http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/jdb.html, 2002

[Sun Microsystems 2008] Sun Microsystems: *Java Platform Debugger Architecture Overview.* http://java.sun.com/javase/6/docs/technotes/guides/jpda/jpda.html, 2008

[Sunye u. a. 2001] Sunye, Gerson ; Pennaneac'h, Francois ; Ho, Wai-Ming ; Guennec, Alain L. ; Jquel, Jean-Marc: Using UML Action Semantics for Executable Modeling and Beyond. In: *Conference on Advanced Information Systems Engineering*, 2001, 433-447

[Tamassia 1987] Tamassia, Roberto: On embedding a graph in the grid with the minimum number of bends. In: *SIAM J. Comput.* 16 (1987), Nr. 3, S. 421–444

[Tamassia u. a. 1988] Tamassia, Roberto ; Battista, Giuseppe D. ; Batini, Carlo: Automatic graph drawing and readability of diagrams. In: *IEEE Trans. Syst. Man Cybern.* 18 (1988), Nr. 1, S. 61–79

[The Mobile Industry Processor Interface (MIPI) Alliance 2008] MIPI2008

[Tip 1995] Tip, Frank: A Survey of Program Slicing Techniques. In: *Journal of Programming Languages* 3 (1995), S. 121–189

[Tolvanen u. a. 2002] Tolvanen, Juha-Pekka ; Gray, Jeff ; Rossi, Matti: 2nd workshop on domain-specific visual languages. In: *OOPSLA '02: Companion of the 17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA : ACM Press, 2002, S. 94–95

[Tudoreanu 2003] Tudoreanu, M. E.: Designing effective program visualization tools for reducing user's cognitive effort. In: *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*. New York, NY, USA : ACM Press, 2003, S. 105–ff

[Ungar u. a. 1997] Ungar, David ; Lieberman, Henry ; Fry, Christopher: Debugging and the experience of immediacy. In: *Commun. ACM* 40 (1997), Nr. 4, S. 38–43

[Weiser 1984] Weiser, Mark: Program Slicing. In: *IEEE Transactions on Software Engineering* SE-10 (1984), July, Nr. 4

[Zeller 2003] Zeller, Andreas: *Causes and Effects in Computer Programs*. AADEBUG'03: Proceedings of the sixth international symposium on Automated analysis-driven debugging, 2003

[Zeller 2004] Zeller, Andreas: *Debugging with DDD, User's Guide and Reference Manual First Edition, for DDD Version 3.3.9*. http://www.gnu.org/manual/ddd/pdf/ddd.pdf, Januar 2004

[Zeller 2005] Zeller, Andreas: *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2005

Betreute Studien- und Diplomarbeiten

[Branchat Freixa 2006] Branchat Freixa, Robert: *Modellnahe Visualisierung und Steuerung für das Debugging hierarchischer Zustandsautomaten.* Studienarbeit, 2006

[Bukhari 2006] Bukhari, Amir: *Software-Framework für ein elektronisches Steuergerät auf Basis Freescale MPC5200.* Masterarbeit, 2006

[Fan 2007] Fan, Yijun: *Optimierte Visualisierung inkrementell erzeugter UML Objektdiagramme.* Studienarbeit, 2007

[Fu 2007] Fu, Lei: *Erweiterung und Integration einer flexiblen Projektverwaltung für ausführbare Modelle.* Studienarbeit, 2007

[Genchev 2008] Genchev, Hristo: *Ermittlung der Basiskenngrößen von Rechnerarchitekturen für den Einsatz in eingebetten Systemen im Automobilbereich.* Diplomarbeit, 2008

[Gutstein 2007] Gutstein, Helge: *Entwicklung einer umkehrbaren Abbildung von Verhaltensbeschreibungen im UML Action Metamodell in ausführbaren Code.* Diplomarbeit, 2007

[Koschuhar 2007] Koschuhar, Vadim: *Entwicklung einer Abbildung textbasierter Verhaltensbeschreibungen in das UML Action Metamodell.* Studienarbeit, 2007

[Long 2007] Long, Jun: *Entwicklung einer flexiblen Projektverwaltung für ausführbare Modelle.* Studienarbeit, 2007

[Ma 2006] Ma, Bin: *Entwicklung einer Schnittstelle für das Debugging von durch Zustandsautomaten spezifizierten Funktionen einer rekonfigurierbaren Architektur.* Studienarbeit, 2006

[Schober 2007] Schober, Jürgen: *Entwicklung eines echtzeitfähigen Messsystems für Automotive-Anwendungen mit Multi-Bus-Architektur.* Studienarbeit, 2007

[Tadenfok Nguetse 2008] Tadenfok Nguetse, Gilbert: *Analyse der effizienten Umsetzung großer Datenmengen von externen Datenquellen auf fahrzeuginterne Bussysteme zum Flashenvon Steuergeräten.* Diplomarbeit, 2008

[Wang 2007] Wang, Willy L.: *Entwicklung eines Modell-Debuggers für in konfigurierbarer Hardware ausgeführte hierarchische Zustandsautomaten.* Studienarbeit, 2007

[Zhang 2005] Zhang, Liang: *Entwicklung einer Methode zur Modellsynchronisierung für das Debugging modellbasiert entworfener Systeme.* Diplomarbeit, 2005

[Zhang 2006] Zhang, Wuxue: *Automatisiertes echtzeitfähiges Testen auf Basis von UML-Sequenzdiagrammen.* Studienarbeit, 2006

[Zumbülte 2007] Zumbülte, Michael: *Entwicklung einer Komponente zum echtzeitfähigen Debugging von Modellen eingebetteter Software.* Diplomarbeit, 2007

Abbildungsverzeichnis

Listings

Verwendete Abkürzungen

API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
AVOPT	Analyse, Verifikation, Optimierung, Parallelisierung und Transformation
BDM	Background Debug Mode
CASE	Computer Aided Software/Systems Engineering
CIM	Computation Independent Model
CLI	Command Line Interface
CMOF	Complete Meta-Object Facility
COM	Component Object Model
CPU	Central Processing Unit
DI	Diagram Interchange
DSL	Domain Specific Language
DSML	Domain Specific Modeling Language
DSP	Digitaler Signalprozessor
DTD	Dokumententypdefinition
EMF	Eclipse Modeling Framework
EMOF	Essential Meta-Object Facility
FPGA	Field Programmable Gate Array
GDB	GNU Debugger
GEF	Graph Editor Framework
GPL	General Purpose Language
JDB	Java Debugger
JDI	Java Debug Interface
JDPA	Java Debug Platform Architecture
JDWP	Java Debug Wire Protocol

JMI	Java Metadata Interface
JNI	Java Native Interface
JTAG	Joint Test Action Group
MDA	Model Driven Architecture
MDSD	Model Driven Software Development
MDSE	Model Driven Software Engineering
MOF	Meta Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
PIM	Platform Independent Model
PSM	Platform Specific Model
RTTI	Run-time Type Information
SoC	System on Chip
SQL	Structured Query Language
SysML	Systems Modeling Language
TAP	Test Access Port
UML	Unified Modeling Language
URI	Uniform Resource Identifier
VM	Virtuelle Maschine
VHDL	Very High Speed Integrated Circuit Hardware Description Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Lebenslauf

Name	Philipp Michael Graf
Geburtsdatum	06.08.1974
Geburtsort	Freiburg im Breisgau
Staatsangehörigkeit	deutsch
Familienstand	ledig
August 1981 – Juli 1985	Grundschule in Freiburg
August 1985 – Juli 1987	Wentzinger Gymnasium Freiburg
August 1987 – Juli 1994	Goethe-Gymnasium Bensheim
Juni 1994	Abitur
Oktober 1995 – Juli 1996	Studium der Mathematik an der Universität Freiburg
Oktober 1996 – Mai 2002	Studium der Elektrotechnik an der Universität Karlsruhe (TH)
Mai 2002	Diplom-Ingenieur Elektrotechnik
August 2002 – Juli 2008	Wissenschaftlicher Mitarbeiter am Institut für Technik der Informationsverarbeitung (ITIV), Universität Karlsruhe
16. Juli 2008	Promotionsprüfung Doktor-Ingenieur Elektrotechnik
seit Juli 2008	am Forschungszentrum Informatik (FZI) in Karlsruhe

Universität Karlsruhe (TH)
Research University · founded 1825