Universität Karlsruhe (TH)
Forschungsuniversität • gegründet 1825

Fakultät für Informatik

Institut für Angewandte Informatik und
Formale Beschreibungsverfahren
Prof. Dr. Andreas Oberweis

AIFB

Modellierung und Simulation von Maschinensteuerungen auf Basis von xUML

Diplomarbeit
von

Fabian Zentner

Verantwortlicher Betreuer:	Prof. Dr. Andreas Oberweis
Betreuender Mitarbeiter:	Dipl.-Inform. Thomas Schuster

Bearbeitungszeitraum: 16. Juni 2008 – 12. Januar 2009

Ehrenwörtliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Diplomarbeit selbständig angefertigt habe. Es wurden nur die in der Arbeit ausdrücklich benannten Quellen und Hilfsmittel benutzt. Wörtlich oder sinngemäß übernommenes Gedankengut habe ich als solches entsprechend kenntlich gemacht.

Ort, Datum

Unterschrift

Dank

Die vorliegende Diplomarbeit wurde am Institut für Angewandte Informatik und Formale Beschreibungsverfahren an der Universität Karlsruhe (TH) angefertigt.

Als erstes möchte ich mich bei Herrn Professor Dr. Oberweis bedanken, der es mir ermöglicht hat, diese Arbeit zu schreiben.

Ganz besonders möchte ich mich bei meinem Betreuer Thomas Schuster bedanken, der mich während der gesamten Diplomarbeit unterstützt hat und mir mit seinem fachlichen Rat zur Seite stand.

Ebenso möchte ich Peter Lieber, Richard Deininger und Robert Schill für ihre Unterstützung danken sowie den Firmen SparxSystems und LieberLieber für das zur Verfügung gestellte Fahrzeug und das UML Modellierungswerkzeug Enterprise Architect. (www.sparxsystems.de und www.lieberlieber.com)

Ganz herzlich bedanken möchte ich mich bei meiner Familie, insbesondere bei meinem Bruder Raphael, meiner Schwester Sarah und meiner Mutter, die mich während der gesamten Arbeit sehr unterstützt und damit das Gelingen dieser Arbeit ermöglicht haben.

Inhaltsverzeichnis

1 Einleitung

In diesem Kapitel wird zunächst eine kurze Einführung in das Themengebiet gegeben. In Abschnitt 1.2 wird auf die Problemstellung dieser Arbeit näher eingegangen. Danach wird ein Szenario vorgestellt, das als Fallstudie für die zu erarbeitende Lösung dienen soll. Das Kapitel wird mit einem Überblick der Arbeit anhand einer Gliederung abgeschlossen.

1.1 Motivation

Softwareentwickler sehen sich in der heutigen Zeit einem immer größer werdenden Wettbewerbsdruck ausgesetzt. Softwaresysteme sollen bei zunehmender Komplexität in immer kürzeren Zeitabständen entwickelt werden. Zusätzlich unterliegt der Software eine ständige Weiterentwicklung und Veränderung [ST04]. Außerdem soll die Software flexibel an Technologien und Plattformen wie z.B. Programmiersprachen, Middleware, Betriebssysteme und Hardware angepasst werden können.

Die Entwicklung von Softwaresystemen auf der Grundlage von Modellen könnte dabei helfen, mit den genannten Anforderungen besser umgehen zu können. Modelle sind schon immer ein wichtiger Bestandteil der Softwareentwicklung gewesen [WM07]. Dies lässt sich z.B. an der starken Verbreitung und Akzeptanz von Modellierungssprachen beobachten. In vielen Unternehmen wird z.B. die grafische Modellierungssprache UML zur Spezifikation, Visualisierung und Konstruktion von Softwaresystemen eingesetzt. Dabei können Modelle die Kommunikation zwischen allen am Entwicklungsprozess beteiligten Personen verbessern. Beim Übergang vom Modell zur Implementierung kann es allerdings vorkommen, dass das Modell von der weiteren Entwicklung am Quellcode entkoppelt wird, je nachdem welches Vorgehens- oder Architekturmodell für die Softwareentwicklung verwendet wird. Die Erhaltung der Konsistenz zwischen dem Modell und dem Programmcode kann einen erheblichen Aufwand erfordern und dazu führen, dass Modelle nur zur Visualisierung im Anfangsstadium der Entwicklung eingesetzt werden.

Die modellgetriebene Softwareentwicklung (MDD) (siehe Anschnitt 2.2) bietet eine Möglichkeit, mit den genannten Problemen und obigen Rahmenbedingungen umzugehen. Bei diesem Ansatz spielt das Modell eine zentrale Rolle und wird nicht nur zum Entwurf, zur Analyse und zur Dokumentation eines Systems verwendet. Das Modell wird in der MDD zur Beschreibung von Softwaresystemen benutzt, das durch Transformationen teilweise oder vollständig in ein System überführt werden kann. Damit soll eine signifikante Steigerung der Entwicklungsgeschwindigkeit erreicht werden. Zusätzlich sollen die Softwarequalität und die Effizienz bei der Entwicklung erhöht werden, was zu einer Zeit- und Kostenersparnis führen kann. Eine modellgetriebene Softwareentwicklung kann sich lohnen, wenn z.B. viele Varianten eines Systems erstellt werden müssen oder mehrere Personen verschiedene Kenntnisse zu einem Softwareprojekt bzgl. einer Anwendungsdomäne, einer spezifischen Technologie oder zu implementierungstechnischen Details beitragen.

Die Model-Driven Architechture (MDA) (siehe Abschnitt 2.3) ist eine Initiative der OMG und stellt eine konkrete Ausprägung der MDD dar. Wie bei der MDD werden Systeme durch Modelle beschrieben. Diese können schrittweise durch Transformationen in Programmcode überführt werden. Modelle werden in der MDA in unterschiedlichen Abstraktionsstufen verwendet, um eine Trennung der Spezifikation eines Systems von implementierungstechnischen Details zu erreichen. Diese Trennung erfolgt über plattformunabhängige Modelle (PIM) (siehe Abschnitt 2.3.4) und plattformspezifische Modelle (PSM) (siehe Abschnitt 2.3.5). Dadurch können Systeme mit reduziertem Aufwand für unterschiedliche Technologien und Plattformen entwickelt werden.

Executable UML (siehe Abschnitt 2.4) kann als eine Art der Implementierung der MDA verstanden werden [CG04]. Mit executable UML sollen Modelle erzeugt werden, die direkt ausgeführt werden können. Diese Modelle sollen dann getestet und für eine bestimmte Zielplattform in Programmcode transformiert werden. Ein Unterschied zur MDA besteht allerdings darin, dass der Zwischenschritt der Transformation eines PIMs in ein PSM entfällt [CG04]. Da UML in ihrer derzeitigen Version nicht direkt ausführbar ist, ist die Verwendung einer Action Language (siehe Abschnitt 2.4.1) erforderlich, die entsprechende Informationen zu den UML Modellen hinzufügt. Außerdem werden spezielle Entwicklungsumgebungen bzw. Rahmenwerke benötigt, die Modellelemente interpretieren und ausführen können.

Executable UML wird in vielen Bereichen wie der Automobilindustrie, der Telekommunikation, der Luft-und Raumfahrt und dem Verteidigungssektor eingesetzt [RCFP03]. Systeme werden dort anhand von ausführbaren Modellen entwickelt, auf denen dann spezifische Tests durchgeführt werden. Ein Modell wird bei executable UML als vollständig betrachtet, wenn es alle vorgesehenen Tests durch Simulationen erfolgreich besteht. Anschließend findet eine Implementierung durch Transformationen statt, indem das Modell durch eine Menge von Regeln in ein Zielsystem überführt wird. Das Vorgehen bei executable UML erlaubt eine iterative Entwicklung von Modellen, bei der Änderungen oder Fehler durch Simulationen sofort sichtbar werden. Damit kann das Erstellen von korrekten Modellen beschleunigt werden. Zusätzlich wird das Verständnis für ein Modell verbessert. Mit executable UML könnte z.B. die Entwicklung einer Steuerung für einen Industrieroboter erfolgen, bei dem sich der Roboter bei der Ausführung eines Modells real bewegt.

1.2 Problemstellung

In dieser Diplomarbeit soll ein Lösungsansatz entwickelt werden, mit dem UML Aktivitätsdiagramme direkt ausgeführt werden können. Mit diesen ausführbaren Aktivitätsdiagrammen soll dann ein Fahrzeug autonom gesteuert werden können. Dazu soll ein beliebiger Fahrablauf mit den Elementen des Aktivitätsdiagrammes modelliert werden können. Damit ein modellierter Fahrablauf real ausgeführt werden kann, wird von der Firma SparxSystems das Fahrzeug in Abbildung 1 zu Versuchszwecken bereitgestellt, das unter anderem zwei Servomotoren (siehe Abschnitt 4.3.4), ein Kompassmodul (siehe Abschnitt 4.3.2) und zwei Sensoren (siehe Abschnitt 4.3.3) enthält.

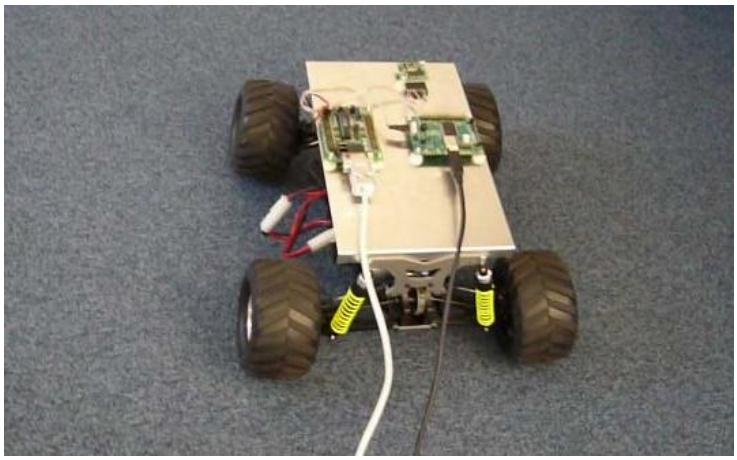


Abbildung 1: Fahrzeug mit Komponenten

Um das Fahrzeug durch die Ausführung eines Aktivitätsdiagrammes steuern zu können, ist eine entsprechende Kommunikation zwischen den einzelnen Fahrzeugkomponenten und der Ausführungseinheit, die als Erweiterung zu einem UML Modellierungswerkzeug zu entwickeln ist, zu erstellen. Diese Ausführungseinheit soll die Modellelemente eines Aktivitätsdiagrammes interpretieren und ausführen können. Zusätzlich

sollen die ausführbaren Aktivitätsdiagramme durch Transformationen der enthaltenen Modellelemente in Programmcode übersetzt werden können. Der erzeugte Code soll automatisch in eine Umgebung eingebettet werden, die alle zur Kommunikation mit dem Fahrzeug notwendigen Programmelemente enthält. Nach der Kompilierung des erzeugten Codes soll der Fahrzeugablauf erneut ausgeführt werden und dessen Performance mit der aus der Modellausführung verglichen werden.

1.3 Beschreibung einer Beispielablaufsteuerung

Die entwickelte Lösung zur Ausführung und Transformation von Aktivitätsdiagrammen sowie zur Kommunikation mit dem Fahrzeug soll an einem konkreten Fallbeispiel Anwendung finden. Bei der Fallstudie handelt es sich um ein Szenario aus der Automobilindustrie bzw. der Robotik. Dabei soll ein Fahrzeug eine ausreichend große Parklücke suchen und anschließend in diese Lücke mit einem Sicherheitsabstand rückwärts einparken. Dieser Vorgang soll wie in Abbildung 2 gezeigt ablaufen und außerdem vollständig autonom erfolgen. Dazu ist eine entsprechende Modellierung für die Suche einer geeigneten Parklücke und des darauf folgenden Einparkvorgangs zu erstellen. Mit Simulationen sollen anschließend Testläufe durchgeführt werden. Das Modell soll solange modifiziert werden, bis der reale Fahrzeugablauf den gestellten Anforderungen genügt. Danach soll Programmcode aus den Modellelementen des Fahrzeugablaufs erzeugt und anschließend ausgeführt werden.

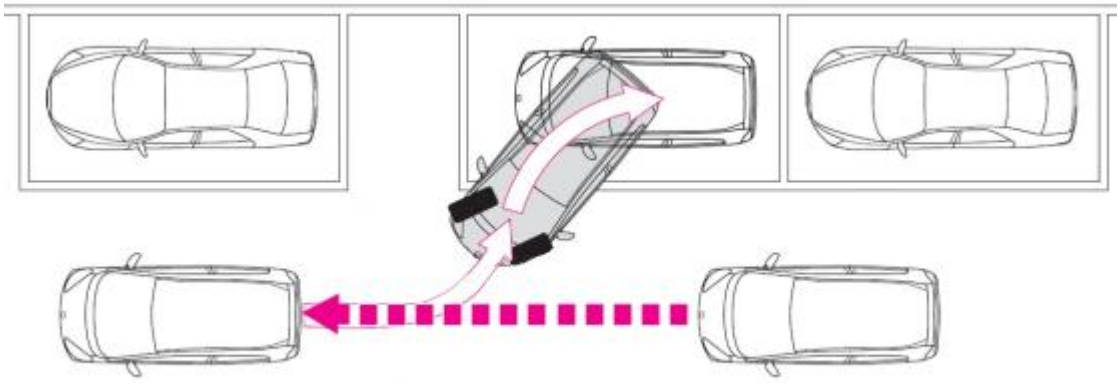


Abbildung 2: Suche einer Parklücke mit anschließendem Einparken

1.4 Gliederung der Arbeit

Nach der Einführung in die vorgestellte Problematik soll im Folgenden auf den Aufbau der Arbeit eingegangen werden.

Kapitel 2: Grundlagen

In Kapitel 2 werden zunächst die Grundlagen der UML besprochen. Insbesondere wird das UML Aktivitätsdiagramm näher behandelt. Danach wird auf die modellgetriebene Softwareentwicklung (MDD) eingegangen. Die MDA, die eine Ausprägung der MDD darstellt, wird anschließend erläutert. Das Kapitel schließt mit der executable UML ab, die einen Ansatz zur Ausführung von Modellen darstellt.

Kapitel 3: Stand der Technik

Im dritten Kapitel werden mehrere Modellausführungsrahmenwerke bzw. executable UML-fähige Werkzeuge vorgestellt, die eine Ausführung von UML Modellen unterstützen. Dabei wird genauer betrachtet, wie Modelle simuliert werden. Eine Anforderung für eine Modellausführung ist die Verwendung einer Action Language, auf die im Anschluss näher eingegangen wird.

Kapitel 4: Modellierung, Simulation und Codegenierung von Maschinensteuerungen

In Kapitel 4 wird ein eigener Lösungsansatz zur Ausführung von UML Aktivitätsdiagrammen vorgestellt. Dazu werden zunächst die Anforderungen untersucht. Mit der Simulation der Aktivitätsdiagramme soll dann ein Fahrzeug gesteuert werden. Bevor auf den genauen Ablauf einer Simulation sowie der dazu erforderlichen *Execution Engine* eingegangen wird, werden die einzelnen Komponenten des Fahrzeuges erläutert. Am Schluss des Kapitels wird gezeigt, wie Programmcode aus einem Aktivitätsdiagramm durch Transformation der Modellelemente erzeugt werden kann.

Kapitel 5: Simulation und Codegenerierung am Fallbeispiel

Um den Ansatz aus Kapitel 4 konkret anwenden zu können, wird in Kapitel 5 eine Fahrzeugablaufsteuerung zum automatischen Einparken modelliert und simuliert. Der Fahrablauf wird mit einem Aktivitätsdiagramm modelliert, das im Anschluss ausgeführt wird. Dabei ist eine Kommunikation mit den in Kapitel 4 beschriebenen Fahrzeugkomponenten notwendig. Nach der Simulation wird gezeigt, wie Quellcode aus dem Modell zur Fahrzeugsteuerung erzeugt werden kann. Der generierte Code wird automatisch in ein Visual Studio 2008 Projekt integriert, das kompiliert und ausgeführt werden kann.

Kapitel 6: Zusammenfassung und Ausblick

In Kapitel 6 werden die in dieser Arbeit erworbenen Erkenntnisse zusammengefasst. Zusätzlich wird ein Ausblick für Erweiterungen und zukünftige Arbeiten gegeben, die an das Konzept der Arbeit anknüpfen.

2 Grundlagen

Um die in der Einführung betrachtete Problemstellung lösen zu können, sind einige Grundlagen notwendig. Zuerst wird die Modellierungssprache UML näher besprochen. Danach wird auf die modellgetriebene Softwareentwicklung (MDD) und auf die MDA, die eine konkrete Ausprägung der MDD darstellt, näher eingegangen. Abschließend wird die executable UML betrachtet.

2.1 Unified Modeling Language - UML

Die Unified Modeling Language ist eine standardisierte grafische Modellierungssprache zur Visualisierung, Konstruktion, Dokumentation und Spezifikation von (Software-) Systemen. Sie findet in vielen Bereichen wie z.B. der Telekommunikation, dem Finanzwesen und der Luft- und Raumfahrt Anwendung. Mit der UML können sowohl statische als auch dynamische Aspekte eines Systems modelliert werden. Sie dient unter anderem der Beschreibung von Anwendungsfällen, Geschäftsprozessen, Echtzeitsystemen, Workflow- und Datenbankanwendungen.

Die UML ist plattform- und sprachenunabhängig, d.h. die Modellierung (Design, Entwurf) von Anwendungen kann unabhängig von Systemen bzw. Plattformen und Programmiersprachen erstellt werden. Weitere Merkmale der UML sind die Eindeutigkeit durch präzise Semantik der Notationselemente sowie die Ausdrucksstärke der verfügbaren Notationselemente. Dabei ermöglichen unterschiedliche Diagramme differenzierte Sichtweisen auf das zu modellierende System und fokussieren oder abstrahieren ihre Teilaspekte, wodurch die Kommunikation aller an der Softwareentwicklung beteiligten Personen erleichtert wird. Die Softwarequalität kann dadurch erheblich verbessert werden und reduziert so langfristig gesehen die Kosten. Durch die Standardisierung findet die UML eine breite Akzeptanz und ist weltweit in der Softwarebranche im Einsatz. Es existieren zahlreiche kommerzielle und nichtkommerzielle UML Werkzeuge mit teils sehr unterschiedlichem Funktionsumfang. In [www-05] wird ein Überblick über vorhandene Werkzeuge gegeben.

Die Geschichte der UML

Mitte der 90er Jahre wurde bei der Firma Relational mit der Entwicklung einer einheitlichen Modellierungssprache unter dem Namen Unified Method begonnen. Dabei wurde die Booch Methode mit der OMT (Object Modeling Technique) zusammengeführt. Irvan Jacobson kam 1996 mit OOSE (Object - Oriented Software Engineering) hinzu. Im selben Jahr legten sie gemeinsam eine Spezifikation der UML in der Version 0.9 vor. Im Januar 1997 wurde die UML 1.0 als erste offizielle Version verabschiedet. Zur Standardisierung, Pflege und Weiterentwicklung wurde die Sprache an die Object Management Group (OMG) übergeben, eine Vereinigung von Unternehmen zur Standardisierung in der Softwareentwicklung, die die Sprache am 19. November 1997 als Standard akzeptierte. Weitere Revisionen erfolgten schrittweise bis zur Spezifikation von UML 1.5. Danach wurde die Sprache vollständig überarbeitet und die Version 2.0 entwickelt. Die UML 2.0 enthält gegenüber der Vorgängerversion zahlreiche Neuerungen, die erforderlich wurden, um die Entwicklungen im Bereich der Programmiersprachen abzudecken und neue Technologien zu berücksichtigen. Wesentliche Änderungen der neuen Version sind unter anderem die Neugestaltung des Aufbaus des Metamodells, ein verbesserter Modellaustausch zwischen verschieden UML Werkzeugen durch XML Metadata Interchange (XMI) sowie bessere Unterstützung der Model Driven Architecture (MDA) (siehe Abschnitt 2.3), in der es unter anderem um Transformationen von Modellen und die Erzeugung von Programmcode aus diesen Modellen geht. Zusätzlich werden Echtzeitmodellierung durch neue Diagrammtypen und erweiterte Semantik sowie Geschäftsprozessmodellierung (BPM) besser unterstützt.

In der Version UML 2.1.2 sind insgesamt die 13 nachfolgenden Diagrammtypen spezifiziert (Abbildung 3).

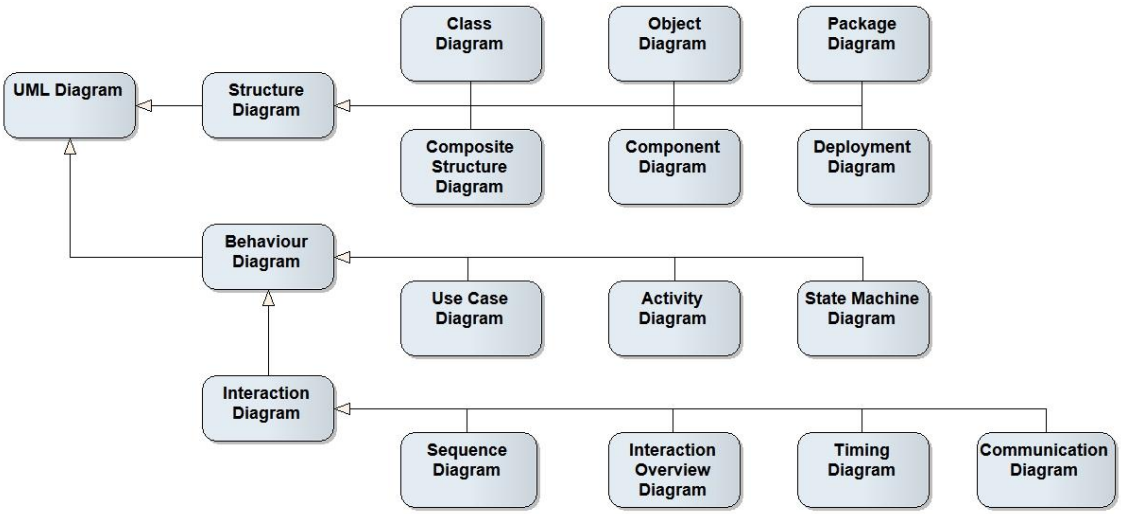


Abbildung 3: UML Diagrammtypen

Die Diagramme sind in Strukturdiagramme und Verhaltensdiagramme unterteilt. In der Praxis werden Klassendiagramme, Anwendungsfalldiagramme, Sequenzdiagramme und Aktivitätsdiagramme sehr häufig eingesetzt und beschreiben statische Strukturen und dynamische Aspekte von Systemen. Ein wichtiger Punkt ist, dass einige Diagramme hierarchisch aufgebaut und ineinander verschachtelt werden können. Die Details eines Diagramms werden dabei in einem weiteren verfeinert, wobei die Diagramme über definierte Schnittstellen miteinander verknüpft werden. Ein Diagramm kann z.B. ein Verhalten, das in einem anderen Diagramm modelliert ist, über eine definierte Schnittstelle aufrufen. Damit kann die Komplexität eines Systems aufgespalten werden. Ein Manager möchte z.B. eher von der Komplexität im Detail abstrahieren wohingegen ein Anwendungsentwickler an der gesamten Komplexität interessiert ist.

2.1.1 Aufbau von UML

Die UML basiert auf dem Konzept der Metamodellierung. Hierzu kommt in der UML Spezifikation eine 4 Ebenen Architektur (Abbildung 4) zum Einsatz, die sich wie folgt gliedert: Die primäre Aufgabe der Ebene M3 besteht darin, Metamodelle zu beschreiben. Die Meta Object Facility (MOF) [OMGM07] ist ein Meta-Metamodell und spezifiziert eine abstrakte Sprache für die Beschreibung von Modellierungssprachen und ist zugleich auch eine Instanz der Ebene M3. Sie ist reflexiv definiert und somit selbstbeschreibend. Alle Modellierungssprachen der OMG werden durch dieses gemeinsame Meta-Metamodell spezifiziert. Die MOF basiert auf der Infrastructure Library, die in der Infrastructure [OMGI07] definiert ist (siehe Abschnitt 2.1.2). Die darunterliegende Ebene M2 definiert Metamodelle, die wiederum Instanzen der MOF sind. Beispiele sind das UML Metamodell, das Common Warehouse Metamodell (CWM) sowie das Metamodell der Object Constraint Language (OCL) [OMGO06]. Jedes Element des jeweiligen Metamodells auf der Ebene M2 ist dabei eine Instanz eines Modellelements der MOF. Die UML Superstructure [OMGS07] (siehe Abschnitt 2.1.3) ist der Ebene M2 zuzuordnen, da auf dieser Ebene die eigentliche Sprachdefinition stattfindet. Die Ebene M1 beinhaltet konkrete Modelle, die Instanzen der jeweiligen Metamodelle sind. Im Falle von UML als Metamodell enthält diese Schicht z.B. UML Diagramme, Klassen, Module und Typdeklarationen. Die unterste Ebene M0 besteht schließlich aus Modellinstanzen. Diese Objekte können z.B. Datenwerte oder instantiierte Klassenobjekte sein.

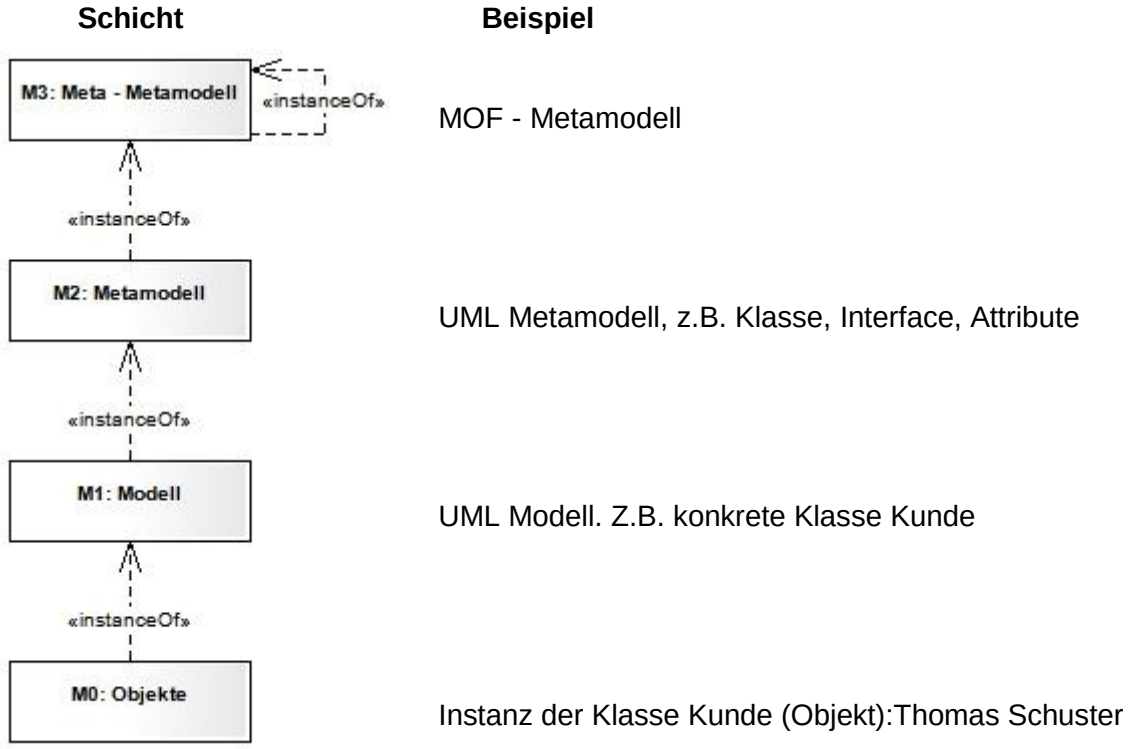


Abbildung 4: Schichten der Metamodellierung

Neben dieser Schichtenarchitektur basiert die UML 2.1.2 auf folgenden Entwurfsprinzipien:

- Modularität (starke Kohäsion und lose Kopplung bei der Bildung der Metamodellpakete und Organisation der Metaklassen)
- Schichtenarchitektur (4-Schichten-Metamodell)
- Partitionierung (Organisation von Bereichen innerhalb einer Schicht)
- Erweiterbarkeit (Profile für verschiedene Plattformen und Anwendungsbereiche, neue Sprache neben UML)
- Wiederverwendbarkeit (der Metamodelle)

In den nächsten Abschnitten wird auf die folgenden Teilspezifikationen der UML 2.1.2 näher eingegangen:

- Infrastructure: Kern der Architekur, Profile
- Superstructure: eigentliche Sprachdefinition
- Object Constraint Language: formale Sprache für Constraints

2.1.2 Infrastructure

Die UML Infrastructure [OMGI07] legt das Fundament für die UML 2.1.2 fest, indem sie den Kern der UML beschreibt. Die Infrastructure Library wird in der UML Infrastructure Spezifikation [OMGI07] definiert und bildet den Kern der grundlegenden Modellierungselemente, der sowohl in der UML Infrastructure, als auch in der UML Superstructure und in der MOF eingesetzt wird.

Die Infrastructure Library besteht aus den beiden Paketen *Core* und *Profile*. Durch das Paket *Core* wird der Kern für die UML definiert. *Core* kann außerdem dazu verwendet werden, um eine Vielzahl von weiteren Metamodellen wie MOF oder CWM zu spezifizieren und bildet somit die Grundlage für die Definition von Metamodellen. Zusätzlich

wird mit dem Paket *Profile* ein Erweiterungsmechanismus bereitgestellt, um die UML an verschiedene Plattformen und Domänen anzupassen. Die Infrastructure Library und die Struktur des Paketes *Core* mit Beziehungen zwischen den Unterpaketen sind in Abbildung 5 zu sehen.

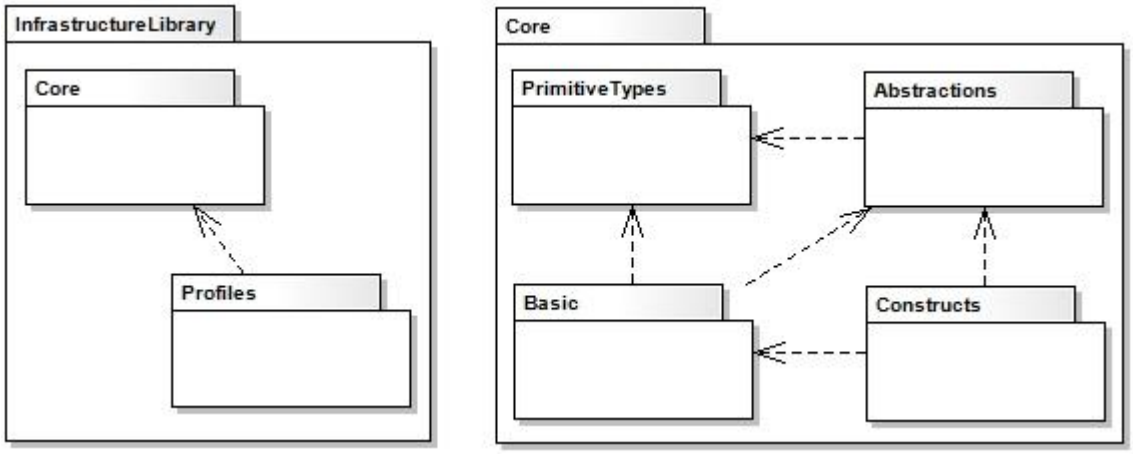


Abbildung 5: InfrastructureLibrary und Core

Das Paket *Core* besteht aus den Unterpaketen PrimitiveTypes, Abstractions, Basic und Constructs, die nun erläutert werden:

PrimitiveTypes

PrimitiveTypes enthält Definitionen von 4 primitiven Typen:

- String
- Boolean
- Integer
- UnlimitedNatural (natürliche Zahlen plus „*“ als unendlich)

Abstractions

Abstractions wird in mehrere spezialisierte Pakete unterteilt, um flexible Wiederverwendung bei der Metamodellierung zu ermöglichen. Zu diesen Paketen gehören z.B. Elements, Relationships, Multiplicities, Constraints und Expressions.

Basic

Das *Basic* Paket stellt eine minimale klassenbasierte Modellierungssprache bereit. *Basic* kann als Instanz von sich selbst angesehen werden und bildet die Grundlage für komplexere Sprachen. Die Konstrukte aus Basic werden z.B. als Basis für das Austauschformat XMI verwendet. Die Metaklassen in Basic sind durch Types, Classes, DataTypes und Packages spezifiziert.

Constructs

Das Paket *Constructs* ist abhängig von mehreren Paketen. Es importiert Modellelemente von PrimitiveTypes und enthält Metaklassen von Basic und Abstractions. Dabei erweitert *Constructs* das Paket *Basic* um komplexere Metamodellkonstrukte. *Constructs* beinhaltet z.B. Assoziationsmöglichkeiten sowie Erweiterungsmechanismen.

2.1.3 Superstructure

Die UML Superstructure Spezifikation [OMGS07] basiert auf der UML Infrastructure, benutzt und erweitert diese zur Definition der Modellierungselemente der UML. Die eigentliche Sprachdefinition der UML findet in der Superstructure statt. Dabei werden

statische und dynamische Strukturen unterschieden. Die Superstructure wird in mehrere Pakete untergliedert, wobei das *Kernel* Paket den zentralen Teil der UML bildet. Es repräsentiert die Kernmodellierungskonzepte der UML, die Klassen, Assoziationen und Pakete beinhalten. Mittels des Pakets *Kernel* werden die Pakete *Constructs* und *PrimitiveTypes* der Infrastructure Library verwendet und mit zusätzlichen Merkmalen, Assoziationen oder Superklassen erweitert. Genauso wie Constructs hat das Paket *Kernel* eine flache Struktur und enthält nur Metaklassen und keine weiteren Unterpakete.

In der Superstructure sind die Modellierungskonzepte der UML in Spracheinheiten unterteilt. Eine Spracheinheit beinhaltet zusammenhängende Modellierungselemente, mit denen ein bestimmter Aspekt eines Systems formal modelliert werden kann. Mit den Elementen der Spracheinheit Aktivitäten (Activities) lässt sich z.B. das Verhalten eines Systems beschreiben. Die meisten Spracheinheiten sind in mehrere Ebenen gegliedert. Einfache und häufig verwendete Modellierungselemente sind Bestandteil der untersten Ebene. Höhere Ebenen beinhalten Elemente mit zunehmender Komplexität. Beispielsweise enthält die Spracheinheit Aktivität als unterste Ebene *Fundamental Activities* und darauf aufbauend die Ebene *Basic Activities.*

2.1.4 Object Constraint Language

Die Object Constraint Language (OCL) [OMGO06] ist eine formale Sprache zur Formulierung von Ausdrücken in UML Modellen. Die OCL ist Teil des UML Standards seit der Version 1.1 und soll die Modellierung von Software noch präziser gestalten. Die OCL wird auch in der UML Infrastructure benutzt, um formale Bedingungen an das UML Metamodell auszudrücken. Seit der UML 2.0 können beliebige Ausdrücke über alle Elemente eines UML Diagramms formuliert werden. Während in der UML Strukturen, Beziehungen und Abläufe modelliert werden, werden in OCL zusätzlich die Randbedingungen eines Modells spezifiziert. Durch OCL Ausdrücke können z.B. Wertebereiche von Attributen eingeschränkt und einzuhaltende Restriktionen zwischen Objekten festgelegt werden. In UML Diagrammen können OCL Ausdrücke in Notizen angegeben werden, die mit Modellelementen verbunden werden.

Bedingungen können grundsätzlich in jeder beliebigen Sprache wie der Natürlichen definiert werden. Jedoch kann die Verwendung einer nicht formalen Sprache zu Uneindeutigkeiten führen. Traditionelle formale Sprachen können meistens nur von Personen mit soliden mathematischen Grundlagen benutzt werden. Die OCL wurde entwickelt, um einer breiteren Schicht von Anwendern, eine formale Sprache zu bieten, die einfacher zu verstehen ist und eine geringere Komplexität in der Anwendung aufweist.

Die OCL ist eine reine Beschreibungssprache, die unabhängig von konkreten Programmiersprachen ist. Sie gibt ihre Implementierung nicht explizit vor. Da OCL keine Programmiersprache ist, gibt es keine Möglichkeit, Programmlogik oder Kontrollflüsse zu formulieren. Aufgrund der reinen Beschreibung kann ein OCL Ausdruck bei seiner Auswertung den Zustand von Objekten nicht ändern und ist somit frei von Seiteneffekten. Jedoch können Operationen beschrieben werden, die bei ihrer Ausführung den Zustand ändern. Die OCL ist eine typsichere Sprache, in der alle Ausdrücke typkonform sein müssen. Ein *Integer* kann z.B. nicht mit einem *String* verglichen werden.

Mit der OCL können z.B. Invarianten auf Klassen und Typen, Vor- und Nachbedingungen auf Operationen und Methoden, Guards und Anfragen ausgedrückt werden. Invarianten sind Bedingungen, die von allen Instanzen einer Klasse erfüllt sein müssen. Vor- und Nachbedingungen sind *Constraints*, die die Anwendbarkeit und die Auswirkung von Operationen spezifizieren. Eine Vorbedingung (precondition) ist ein Boole' scher Ausdruck, der vor der Ausführung einer Operation als wahr ausgewertet werden muss. Eine Nachbedingung muss entsprechend erfüllt sein, unmittelbar nachdem eine Operation beendet wurde. Ein Guard gibt die Bedingung für einen Zustandsübergang an. Anfragen sind Operationen, die frei von Seiteneffekten sind, da sie den Zustand von Objekten nicht ändern. Die Ausführung einer Anfrage liefert einen Wert oder eine

Menge von Werten zurück. Die OCL ist demnach auch eine Anfragesprache, die Ähnlichkeiten mit SQL aufweist.

Ein OCL Ausdruck wird immer im Rahmen eines Kontextes durch das am Anfang stehende Wort *context* definiert. Diesem folgt ein weiteres Schlüsselwort, das die Art des Ausdrucks bestimmt, z.B. *inv*, *pre* oder *post*. Optional kann ein Name für einen Ausdruck angegeben werden.

```
context Person inv:
self.Alter > 21
```

```
context Party inv:
self.guests ->size() > 30
```

Bei obigen Beispielen handelt es sich um Invarianten (*inv*). Wenn auf die Kontextinstanz selbst verwiesen werden soll, kann dies anhand des Schlüsselworts *self* geschehen. Im ersten Beispiel ist der Kontext eine Person. *Self* referenziert dabei eine Instanz von Person mit der Bedingung, dass eine Person älter als 21 Jahre alt sein muss. Sofern der Kontext eindeutig ist, kann *self* auch weggelassen werden.

Die OCL verfügt über einfache Vergleichsoperatoren, deren Syntax sich eher an SQL als an Java oder C# orientiert. Auf Attribute, Beziehungen und Methoden einer Instanz kann über die Punktnotation („.“) zugegriffen werden. Das Navigieren auf *Collections* geschieht z.B. bei Operationen wie *size()* oder *isEmpty()* mittels der Pfeilnotation („->“).Um die gesamte Mächtigkeit und den Umfang von OCL aufzuzeigen, sei auf die OCL Spezifikation [OMGO06] verwiesen.

2.1.5 Aktivitätsdiagramm

In der UML werden Aktivitätsdiagramme [OMGS07] zur Modellierung von dynamischen Abläufen verwendet. Sie dienen der Beschreibung von Geschäftsprozessen und Workflows, der Analyse von Systemanforderungen, der Spezifikation von Algorithmen und der Visualisierung von Kontrollflüssen. Es können vielfältige Beziehungen zwischen Aktivitätsdiagrammen und anderen Diagrammen der UML bestehen. Zum Beispiel können Anwendungsfalldiagramme, die Geschäftsprozesse modellieren, in Aktivitätsdiagrammen im Detail spezifiziert werden [LM07].

Gegenüber den Vorgängerversionen hat sich beim Aktivitätsdiagramm ab der Version UML 2.0 grundlegendes geändert. War es in den Versionen UML 1.X noch eine Sonderform eines Zustandsdiagramms, ist es nun auch im Metamodell eigenständig verankert. Die Semantik der Aktivitätsdiagramme ist an die Semantik der Petri-Netze angelehnt. Die Abarbeitung eines Aktivitätsdiagramms erfolgt wie die eines Petri-Netzes Token-basiert. Mit dem Token-Konzept wurden präzise Regeln für den Kontroll- und Objektfluss geschaffen. Ein Token entspricht genau einem Ablauf-Thread und repräsentiert das Fortschreiten des Ablauf- oder Datenflusses. Anders als beim Zustandsdiagramm, bei dem der Schwerpunkt auf den statischen Zuständen, den erlaubten Zustandsübergängen und auf den entsprechenden auslösenden Ereignissen liegt, steht beim Aktivitätsdiagramm die Beschreibung einer Aktivität als der Abfolge von Aktionen, zwischen denen Kontroll- und Datenflüsse existieren (bzw. dem Aufruf weiteren Verhaltens) im Vordergrund [LM07].

Seit der UML 2.0 wird ein Aktivitätsdiagramm als Aktivität bezeichnet. Die elementaren Verhaltensbausteine heißen nicht mehr Aktivitäten, sondern Aktionen. Ein Aktivitätsdiagramm darf nun mehrere Startknoten besitzen. Neue Notationselemente wie Signale und Ablaufendknoten wurden eingeführt. Bei einem Splittingknoten muss nun eine Aufteilung nicht mehr synchronisiert werden. Eine weitere Neuerung ist, dass Aktivitäten Eingangs- und Ausgangsparameter enthalten dürfen. Dies sind nur einige Ände-

rungen. Eine vollständige Spezifikation der Aktivitätsdiagramme ist unter [OMGS07] zu finden.

In Aktivitätsdiagrammen werden Aktivitätsknoten (ActivityNode) und Aktivitätskanten zur Modellierung des Kontroll- und Datenflusses zwischen Aktionen benutzt. Ein Aktivitätsknoten ist eine abstrakte Klasse für die einzelnen Schritte einer Aktivität. Er umfasst Kontrollknoten (Control nodes), Objektknoten (Object nodes) und ausführbare Knoten (Executable nodes). Kontrollknoten koordinieren den Ablauf einer Aktivität. Zu dieser Art von Knoten zählen Startknoten, Endknoten, Splittingknoten, Synchronisationsknoten, Entscheidungsknoten und Zusammenführungsknoten. Aktivitätskanten sind Übergänge zwischen zwei Knoten und werden in zwei Arten unterteilt: Eine Kontrollflusskante dient ausschließlich der Ablaufsteuerung zwischen Aktionen wohingegen eine Objektflusskante den Fluss von Objekten von einem Objektknoten zum nächsten modelliert.

2.1.5.1 Modellelemente des Aktivitätsdiagramms

Im Folgenden werden die Modellelemente des UML Aktivitätsdiagrammes näher erläutert:

Aktion (*Action*)

Abbildung 6: Aktion ohne Pins

Abbildung 7: Aktion mit Pins

Aktionen (Abbildung 6) sind die elementaren Bausteine für die Modellierung eines Verhaltens in Aktivitätsdiagrammen, die nicht weiter zerlegt werden können. Sie rufen entweder ein Verhalten auf oder bearbeiten Daten bzw. Objekte. Obwohl eine Aktion in einer Aktivität nicht in kleinere Schritte zerlegbar ist, kann das Resultat dennoch komplex sein. Zum Beispiel ruft eine *Call Behaviour Action* eine Aktivität auf, die wieder aus einzelnen Aktionen besteht. Aktionen können mehrere eingehende und ausgehende Aktivitätskanten sowie mehrere Eingabe- und Ausgabeparameter haben. Diese Parameter modellieren Objektknoten, die Werte oder Instanzen eines bestimmten Typs repräsentieren. Passt der Typ eines Ausgangsparameters nicht mit dem Typ eines Eingangsparameters zusammen, können diese als Pins dargestellt werden (Abbildung 7). Ein Pin entspricht dabei wieder einem Objektknoten. Somit können Aktionen Eingabewerte über Eingabepins entgegennehmen und Ausgabewerte über Ausgabepins bereitstellen. Eine Aktion kann erst gestartet werden, wenn alle Eingabepins belegt bzw. sonstige Eingabebedingungen erfüllt sind und nur dann beendet werden, falls alle Ausgabepins befüllt bzw. alle Ausgabebedingungen erfüllt wurden. Außerdem besteht die Möglichkeit, Aktionen mit Vor- und Nachbedingungen zu verknüpfen, die erfüllt sein müssen, bevor die Aktion aufgerufen wird bzw. nachdem sie abgeschlossen ist. Die UML Spezifikation [OMGS07] schreibt hier keine Implementierung für die Umsetzung der Vor- und Nachbedingungen vor.

Aktion Verhaltensaufruf (*CallBehaviourAction*)

Mit dieser Aktion kann ein Verhalten direkt aufgerufen werden. Die weitere Ausführung wird erst nach vollständiger Abarbeitung des aufgerufenen Verhaltens fortgesetzt. Der Aufruf einer Aktivität ist anhand eines Harkezeichens (Abbildung 8) am rechten unteren Rand zu erkennen. Dabei wird die übergeordnete Aktivität erst dann fortgeführt, wenn die untergeordnete Aktivität vollendet wurde. Somit können Aktivitäten in weiteren Aktivitätsdiagrammen verfeinert und beliebig geschachtelt werden, wobei alle Aktivitäten

auf der untersten Ebene auf Aktionen abzubilden sind. Dieser hierarchische Aufbau von Aktivitätsdiagrammen kann die Übersichtlichkeit und Gliederung verbessern.

Abbildung 8: Aktion Verhaltensaufruf

Startknoten (*InitialNode*)

Ein Startknoten (Abbildung 9) ist ein Kontrollknoten, an dem der Ablauf einer Aktivität beginnt. Für eine Aktivität sind mehrere Startknoten erlaubt.

Abbildung 9: Startknoten

Aktivitätsendknoten (*ActivityFinalNode*)

Ein Aktivitätsendknoten (Abbildung 10) beendet alle Abläufe einer Aktivität. Eine Aktivität kann mehrere Aktivitätsendknoten haben, wobei der Erste, der erreicht wird, alle Abläufe beendet.

Abbildung 10: Aktivitätsendknoten.

Ablaufendknoten (*FlowFinalNode*)

Ein Ablaufendknoten (Abbildung 11) beendet nur einen Ablauf. Er hat keine Auswirkung auf andere Abläufe einer Aktivität.

Abbildung 11: Ablaufendknoten

Entscheidungsknoten (*DecisionNode*)

Ein Entscheidungsknoten (Abbildung 12) ist ein Kontrollknoten, der aus mehreren Abläufen einen auswählt. Die Bedingung, die für die Wahl eines Ablaufs erfüllt sein muss, wird als Ausdruck in eckigen Klammern an den ausgehenden Kanten angegeben. Ein Entscheidungsknoten hat eine eingehende Kante und mehrere ausgehende Kanten.

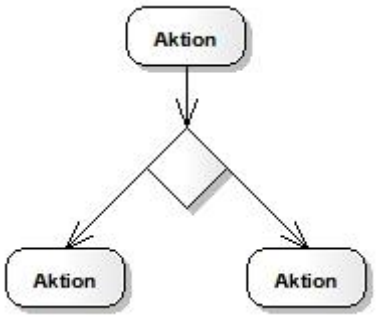


Abbildung 12: Entscheidungsknoten

Zusammenführungsknoten (*MergeNode*)

Ein Zusammenführungsknoten (Abbildung 13) ist ein Kontrollknoten, der mehrere alternative Abläufe zusammenbringt. Eine Synchronisation der nebenläufigen Abläufe findet jedoch nicht statt. Es wird lediglich ein Ablauf der verschiedenen Abläufe akzeptiert. Ein Entscheidungsknoten und ein Zusammenführungsknoten können auch kombiniert werden, um beide Funktionalitäten zu bieten. Dieser Knoten hat dann mehrere eingehende Kanten und mehrere ausgehende Kanten.

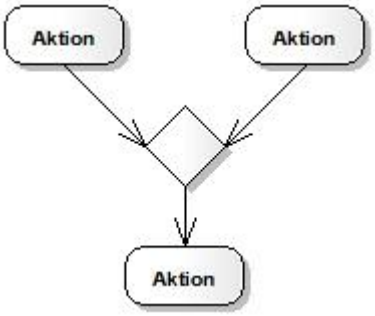


Abbildung 13: Zusammenführungsknoten

Splittingknoten (*ForkNode*)

Bei einem Splittingknoten (Abbildung 14) wird der Kontrollfluss in mehrere nebenläufige Abläufe aufgeteilt, die jeweils in einem eigenen Thread ausgeführt werden. Ein Splittingknoten besitzt eine eingehende Kante und mehrere ausgehende Kanten.

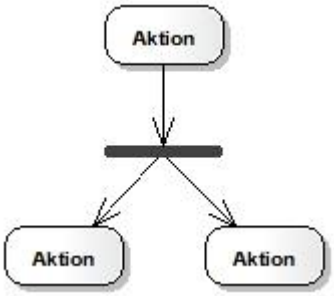


Abbildung 14: Spittingknoten

Synchronisation (*JoinNode*)

Bei der Synchronisation (Abbildung 15) werden verschiedene Abläufe zusammengeführt. Ein Synchronisierungsknoten hat mehrere eingehende Kanten und eine ausgehende Kante. Die Funktionalität eines Splittingknotens und die eines Synchronisierungsknotens können vereinigt werden. Dieser Knoten verfügt dann über mehrere eingehende und ausgehende Kanten.

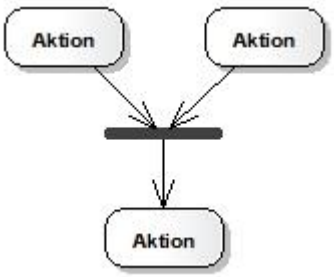


Abbildung 15: Synchronisationsknoten

Objektknoten (*ObjectNode*)

Objektknoten (Abbildung 16) repräsentieren Objekte oder Daten. Mit ihrer Hilfe werden Daten von einer Aktion zur nächsten übergeben. Sie können als unabhängige Knoten in einem Diagramm oder als Parameter für Aktionen (Input Pin, Output Pin) verwendet werden.

Abbildung 16: Objektknoten

Aktion Signal senden (*SendSignalAction*)

Mit dieser Aktion (Abbildung 17) kann ein Signal gesendet bzw. ein Ereignis ausgelöst werden.

Abbildung 17: Aktion Signal senden

Aktion Signal empfangen (*AcceptEventAction*)

Diese Aktion (Abbildung 18) empfängt ein Signal bzw. ein Ereignis, das bestimmte Bedingungen erfüllt.

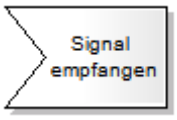


Abbildung 18: Aktion Signal empfangen

2.1.6 Erweiterungsmechanismen der UML

Die UML wurde so konzipiert, dass sie eine Vielzahl von Anwendungsdomänen unterstützt, trotz dessen ist es nicht möglich, die Bedürfnisse aller Nutzer zu erfüllen. Viele Domänen erfordern einige Anpassungen an die UML, um bestimmten Anforderungen des zu modellierenden Systems zu genügen. Aus diesem Grund spezifiziert die OMG zwei Erweiterungsmöglichkeiten, um die UML an spezielle Anforderungen anzupassen:

- Schwergewichtige Erweiterungen, bei denen das zugrundeliegende Metamodell verändert wird
- Leichtgewichtige Erweiterungen, die das bestehende UML Metamodell erweitern

Bei der schwergewichtigen Erweiterung wird ein neues Metamodell basierend auf der MOF [OMGM07] definiert. Diese Variante bietet eine sehr hohe Flexibilität durch Hinzufügen und Entfernen von Metamodellelementen, was z.B. in der Definition von neuen Diagrammen resultieren kann. Auf diese Weise können die Semantik und die Struktur der Modellelemente so definiert werden, dass sie optimal die Charakteristika der Domäne abbilden. Um schwergewichtige Erweiterungen durchführen zu können, sind sehr gute Kenntnisse der MOF und des UML Metamodells vorauszusetzen. Probleme bei der schwergewichtigen Erweiterung sind eine schlechtere Werkzeugunterstützung

bei der Modellierung, da nur wenige Werkzeuge Änderungen am UML Metamodell erlauben sowie ein generell höherer Aufwand bei den Modifikationen des UML Metamodells. Zudem werden Anwender vor die Herausforderung gestellt, eine neue Sprache zu erlernen.

Leichtgewichtige Erweiterungen erweitern das UML Metamodell. Dies kann durch Hinzufügen von Profilen geschehen. Ein Profil besteht aus einem Paket, das Stereotypes, Constraints und Tagged Values enthalten kann. Diese drei Standard-Erweiterungsmechanismen erweitern das UML Metamodell für eine spezielle Umgebung oder für einen bestimmten Anwendungsbereich. In der Praxis sind schwergewichtige Erweiterungen meist überdimensioniert, weshalb leichtgewichtige Erweiterungen eine weniger komplexe Alternative darstellen, um die UML an spezielle Bedürfnisse anzupassen.

2.1.6.1 UML - Profile

UML Profile etablieren eine leichtgewichtige Erweiterung der UML, um Modelle für spezifische Domänen zu erstellen. Ein UML Profil kann nicht eigenständig definiert werden, es muss zum UML Metamodell in Beziehung gesetzt werden, indem es das Metamodell erweitert. Profile werden verwendet, um sich an spezifische Anwendungsgebiete, technische Implementierungen und Softwareentwicklungsprozesse anzupassen. Neben der Möglichkeit Profile selbst zu definieren, existieren auch bereits vorhandene Profile wie für Enterprise Java Beans (EJB), für CORBA und für die Modellierung und Analyse von Echtzeit- und eingebetteten Systemen. Gerade in der MDA (siehe Abschnitt 2.3) sind Profile von großer Bedeutung bei der Unterstützung von Transformationen und automatisierten Codegenerierungen.

Ein UML Profil [WC06] ist eine spezielle Form des UML Paketes mit dem Schlüsselwort <<profile>> vor dem Paketnamen, das Stereotypes (Abschnitt 2.1.6.2), Tagged Values (Abschnitt 2.1.6.3) und Bedingungen (Constraints) (Abschnitt 2.1.6.4) enthalten kann. Die UML Superstructure gibt mehrere Anforderungen an ein Profil an. Es darf z.B. bei der Erweiterung von Modellelementen nicht der Semantik im UML Metamodell widersprechen. Die spezialisierten Modellelemente sollen nur wohldefinierte Regeln und zusätzliche Bedingungen einführen. Sie dürfen auch nicht mit den Constraints im UML Metamodell in Konflikt geraten. Ein Profil soll dadurch einen Modellaustausch per XMI zwischen verschiedenen UML Werkzeugen ermöglichen.

2.1.6.2 Stereotype

Ein *Stereotype* [WC06] ist ein Hilfsmittel, um Modellelemente zu erweitern und zu klassifizieren. Stereotypes erlauben UML Werkzeugen bestimmte Elemente unter anderen Elementen desselben Typs zu identifizieren.

Eine Metaklasse wird durch ein *Stereotype* erweitert, indem es ein neues Metamodellelement basierend auf einem bereits existierenden definiert. Diesem Element können neue oder zusätzliche Eigenschaften zugewiesen werden, die auch als *Tagged Values* (siehe Abschnitt 2.1.6.3) bezeichnet werden. *Stereotypes* erweitern Modellelemente und werden durch den Namen des *Stereotypes* angegeben, der von sogenannten *Guillemets* umfasst wird. Der Name des erweiterten Elements steht darunter. Die Notation sieht wie folgt aus: <<*StereotypeName*>>.

In der UML gibt es vordefinierte Standard-*Stereotypes*, die jedoch nur auf bestimmte Modellelemente angewendet werden können. Beispiele sind <<refine>>, <<derive>> und <<instanceOf>>. Durch seine leichtgewichtige Erweiterung des Metamodells bietet ein *Stereotype* einen komfortablen Mechanismus an, der durch geringe Komplexitätszunahme wünschenswerte Anpassungen und Erweiterungen ermöglicht.

2.1.6.3 Tagged Values

Tagged Values sind Eigenschaftswerte von *Stereotypes* [WC06]. Diese Name-Wert Paare sind seit der UML 2 nur noch *Stereotypes* und nicht mehr allen Modellelementen zuweisbar. Die Modellierung der *Tagged Values* kann z.B. mit Kommentaren stattfinden, die mit dem *Stereotype* verbunden werden. *Tagged Values* sind nicht gleichzusetzen mit Attributwerten einer Klasse. Sie können als Metadaten betrachtet werden, da die Wertzuweisung auf das Element selbst und nicht auf seine Instanzen angewendet wird.

2.1.6.4 Constraints

Bedingungen (*Constraints*) sind Einschränkungen bezüglich der Benutzung oder der Semantik von Modellelementen [WC06]. Beispielsweise darf der Startknoten in einem Aktivitätsdiagramm nach UML 2 keine eingehenden Kanten und nur eine ausgehende Kante aufweisen. Ein Beispiel für eine semantische Einschränkung wäre, dass eine Person älter als 30 Jahre alt sein muss: `{Alter > 30}`. Bedingungen können in natürlicher Sprache, in Programmiersprachen, in Pseudocode oder in sonstigen Sprachen ausgedrückt werden. In der UML Spezifikation werden Bedingungen an Notationselemente durch die formale und präzise Object Constraint Language (OCL) (siehe Abschnitt 2.1.4) definiert. Eine Bedingung ist als Boole' scher Ausdruck zu verstehen, der zu wahr oder falsch ausgewertet werden kann. Entweder wird eine Bedingung in einem UML Werkzeug global für alle betreffenden Modellelemente in Diagrammen definiert oder eine Bedingung wird speziell zu einem Element eines Diagramms festgelegt. Bedingungen, die auf *Stereotypes* angewendet werden, gelten allerdings für alle Modellelemente, die diesen *Stereotype* haben. Im weiteren müssen die Einschränkungen der Basisklasse der *Stereotypes* erfüllt bleiben. Die Umsetzung und Implementierung der vordefinierten UML 2 konformen Bedingungen obliegt den Herstellern von UML Werkzeugen.

2.2 Model Driven Development (MDD)

Die modellgetriebene Softwareentwicklung (Model Driven (Software) Development (MD(S)D) oder Model Driven Engineering (MDE)) verfolgt die Entwicklung von Softwaresystemen auf der Basis von Modellen [Kü07]. Anders als bei der herkömmlichen Softwareentwicklung, bei der die Modelle vorwiegend zum Entwurf, zur Analyse und zur Dokumentation verwendet werden, wird in der modellgetriebenen Entwicklung das Modell als primäre Entwicklungsgrundlage verstanden, das durch Transformationen teilweise oder vollständig in ein Softwaresystem überführt werden kann.

Die Entwicklung von Softwaresystemen auf der Grundlage von Modellen könnte dabei helfen, mit den gestiegenen Anforderungen der letzten Jahre besser umzugehen. Viele Unternehmen sehen sich einem erhöhten Kostendruck ausgesetzt. Die Software soll bei möglichst geringem Preis einen immer größeren Funktionsumfang bieten und flexibel bezüglich der Anpassung an neue Technologien sein [Kü07]. Einerseits nimmt die Komplexität von Systemen stetig zu während anderseits die Systeme in immer kürzeren Zeitabständen entwickelt werden sollen. Daher ist eine bessere Handhabbarkeit von Komplexität ein primäres Ziel modellgetriebener Entwicklungsansätze. Komplexität kann durch verschiedene Mechanismen wie z.B. *Teile und Herrsche-Prinzip*, Abstraktion und *Separation of Concerns* (*SoC*) (Trennung von Betrefflichkeiten) reduziert werden [Kü07]. Das *Teile und Herrsche-Prinzip* unterteilt Systeme in Subsysteme wohingegen mit Abstraktion die Beschränkung auf die relevanten Informationen bzw. Merkmale eines Sachverhaltes gemeint ist. *SoC* trennt eine Problemstellung in verschiedenartige Teilprobleme auf und kann in modellgetriebenen Ansätzen durch eine mögliche Aufteilung in Modellierung, Transformation und Codegenerierung unterstützt werden. Desweiteren begünstigt *SoC* die Arbeitsteilung hinsichtlich spezieller Fähigkeiten und Kenntnisse, was zu Spezialisierungen und zu einer effizienteren Ressourcenzuweisung genutzt werden kann. Diese Arbeitsteilung betrifft sowohl fachliche als auch technische

Aspekte. Die Trennung dieser Aspekte soll in der modellgetriebenen Entwicklung zu verbesserter Kommunikation zwischen Kunden, Fachexperten und Entwicklern führen.

Allgemein werden in der modellgetriebenen Softwareentwicklung folgende Ziele angestrebt:

- Höhere Effizienz bei der Entwicklung von (komplexer) Software
- Abbildung der Systemarchitektur in Modellen
- Trennung von fachlichen und technischen Aspekten
- Verbesserte Kommunikation und Verständnis zwischen Geschäfts – und Entwicklungsebene
- Höhere Produktivität durch geringere Reaktionszeit bei Designänderungen oder Change Requirements
- Reduzierung der Entwicklungskosten durch Automatisierung und Wiederverwendung
- Geringere Fehlerrate
- Verbesserte Wartbarkeit
- Höhere Qualität durch automatische Transformationen – Transformationsregeln bestimmen die Qualität des Zielmodells bzw. des Quellcodes
- Erzeugen von Software für beliebige Plattformen durch Transformation der Modelle (entsprechende Transformationsregeln für jede Plattform)

Wie am Anfang dieses Abschnitts erwähnt werden Modelle in den Mittelpunkt der Softwareentwicklung gestellt. Das Softwaresystem soll bei der Modellierung unabhängig von bestimmten Plattformen oder Standards beschrieben werden können. Zum Beispiel kann in einem Modell von implementierungsspezifischen Details wie einer konkreten Programmiersprache oder einem speziellen Betriebssystem abstrahiert werden. Fachexperten oder Entwickler können sich so zunächst auf die Modelle mit der Beschreibung der Funktionalität von Systemen oder Anwendungen konzentrieren, ohne sich mit Details des Lösungsbereiches auseinandersetzen zu müssen. Diese abstrakten Modelle können anschließend in spezifischere Modelle durch manuelle oder automatisierte Transformationen überführt werden. Dabei können Transformationen Modelle mit zusätzlichen (technischen) Informationen anreichern, indem ein Modell in ein anderes Modell nach bestimmten Vorgaben und Regeln überführt wird. Weiter ermöglichen Transformationen die Extraktion und Aggregation spezifischer Eigenschaften zur Bewertung von Modellen. Ein wesentlicher Unterschied zu älteren Ansätzen der Softwareentwicklung besteht darin, dass in der modellgetriebenen Entwicklung eine stärkere Fokussierung auf die Automatisierung des Entwicklungsprozesses stattfindet. Ein höherer Automatisierungsgrad kann sich positiv hinsichtlich der Effizienz und der Geschwindigkeit der Softwareentwicklung sowie der Softwarequalität auswirken. Zu Beachten ist jedoch, dass die Steigerung des Automatisierungsgrades auch eine Erhöhung der Formalisierung bedeutet. Des Weiteren entstehen durch die Implementierung von Transformationen zusätzliche Aufwände, die mit dem zu erwartenden Nutzen in Beziehung gesetzt werden sollten. Außerdem werden Modellierungswerkzeuge benötigt, die die Transformationen und Codegenerierung unterstützen. Zusätzlich müssen Transformationsregeln und Templates zur Codegenerierung entwickelt werden, wenn vordefinierte Regeln und Templates (sofern vorhanden) zu allgemein oder unvollständig sind. Ob sich generell der Einsatz einer modellgetriebenen Entwicklung lohnt, hängt vom konkreten Anwendungsbereich ab und kann nicht allgemein beantwortet werden [Kü07].

2.3 Model Driven Architecture (MDA)

Die Model-Driven Architechture (MDA) ist eine Initiative der OMG und stellt eine Ausprägung der MDD (siehe Abschnitt 2.2) dar. Die MDA ist nicht als konkrete Technologie sondern vielmehr als Vision der OMG zu verstehen.

"*OMG's Model Driven Architecture ® (MDA ®) provides an open, vendor-neutral approach to the challenge of business and technology change*." [www-01]

Die MDA beschreibt einen Ansatz zur modellgetriebenen Entwicklung von Software, bei der Systeme durch Modelle beschrieben werden und schrittweise durch Transformationen in Quellcode überführt werden können [ST04]. In der MDA wird die Spezifikation der Funktionalität und des Verhaltens eines Softwaresystems von deren Realisierung bzw. Implementierung auf einer spezifischen Plattform getrennt. Dieser Ansatz soll unter anderem die Portabilität, Interoperabilität und Wiederverwendbarkeit von Modellen unterstützen. Diese Eigenschaften sind insbesondere bei der heutigen schnellen Entwicklung von Technologien wichtig, von der Systemmodelle zum großen Teil unabhängig werden.

In der MDA wird das Modell in den Mittelpunkt des Softwareentwicklungsprozesses gestellt [ST04]. Ein Modell wird im Kontext der MDA als abstrakte formale Darstellung verstanden, das durch zusätzliche Informationen über Plattformen und Technologien schrittweise verfeinert und am Ende des Entwicklungsprozesses in Programmcode überführt werden kann. Die einzelnen Entwicklungsschritte sollen dabei teilweise oder sogar vollständig automatisiert erfolgen. Um diese Zielstellung erreichen zu können, verfolgt die MDA den Ansatz der Trennung der Spezifikation eines Systems von implementierungstechnischen Details.

Folgende Ziele werden von der MDA angestrebt:

- **Portabilität**: Existierende Funktionalität kann schnell in neue Umgebungen und Plattformen migriert werden.
- **Produktivität**: Indem Entwicklungsschritte automatisiert werden, können sich Softwarearchitekten und -entwickler auf die logischen Kernbereiche eines Systems konzentrieren.
- **Qualität**: Die formale Trennung von Betrefflichkeiten (*seperation of concerns*) sowie die Konsistenz und Verlässlichkeit der erzeugten Artefakte führen zu einer erhöhten Qualität des Gesamtsystems.
- **Integration**: Die Einbindung von externen Systemen und Legacy-Anwendungen wird unterstützt.
- **Wartbarkeit**: Die Verfügbarkeit von Entwürfen von Softwaresystemen in Form von formalen Modellen gibt Analysten, Entwicklern und Testern direkten Zugriff auf die Spezifikationen von Systemen und erleichtert somit die Wartung von Systemen
- **Test und Simulation**: Modelle können bezüglich Anforderungen validiert und bezüglich verschiedener Infrastrukturen getestet werden. Sie können auch in der Entwurfsphase dazu benutzt werden, das Verhalten eines Systems zu simulieren.
- **Rentabilität**: Unter der Voraussetzung, dass eine modellgetriebene Softwareentwicklung sich als lohnenswert für ein Unternehmen herausstellt (hängt vom Softwaresystem bzw. Größe und Umfang des Projekts ab), kann die Rentabilität durch Senkung der Kosten und schnellerer Entwicklungszeiten gesteigert werden.

Um diese Ziele zu erreichen wird in der MDA die Businesslogik von der Anwendungslogik getrennt [www-01]. Die hierzu von der OMG veröffentlichte Architektur der Model Driven Architecture ist in Abbildung 19 zu sehen.

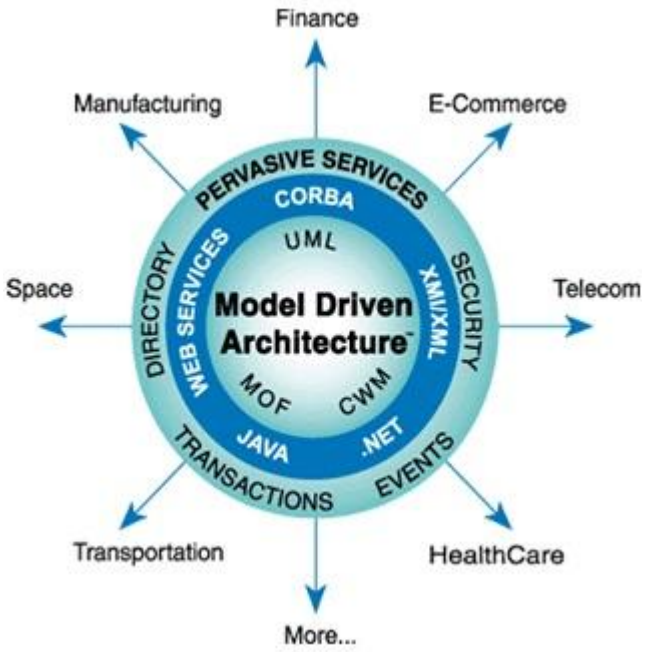


Abbildung 19: MDA Architektur [www-01]

Der Kern der Architektur basiert auf den OMG Standards UML (siehe Abschnitt 2.1), Meta Object Facility (MOF) und Common Warehouse Metamodel (CWM). Auf dem mittleren Ring werden verschiedene Zielplattformen wie Java oder .net angezeigt. Die parallele Entwicklung eines Systems für unterschiedliche Plattformen soll durch eine technologie- und plattformunabhängige Modellierung ermöglicht werden. Eine Trennung ist zwischen den Modellen und deren Implementierung auf diversen Plattformen zu erkennen. Ebenfalls auf dieser Ebene ist der XML Interchange Standard (XMI) angesiedelt, der den Modellaustausch zwischen verschiedenen Anbietern von Modellierungswerkzeugen ermöglichen soll. Der äußere Ring zeigt unter anderem transaktionsbasierte Services, Security Services und Events, die in Modellen der MDA Verwendung finden können und anschließend auf den Zielplattformen realisiert werden. Mögliche Einsatzfelder der MDA werden über die Pfeile dargestellt.

2.3.1 Modell und Metamodell

In der MDA ist das Metamodell eine Schlüsselkomponente, da es eine Voraussetzung für die Beschreibung von Modellen nach formalen Kriterien bildet [www-02]. Ein Metamodell besteht aus einer formal eindeutigen und vollständigen Definition der Syntax und Semantik. Es beschreibt wie Modelle erstellt und interpretiert werden dürfen. Das Metamodell formalisiert die Strukturen einer betrachteten Domäne und bildet die Grundlage für eine Automatisierung, die Modell-zu-Modell-Transformationen oder Modell-zu-Codegenerierung beinhalten kann. Metamodelle können in der MDA mit UML und ihren Erweiterungsmechanismen beschrieben werden, denkbar sind jedoch auch andere Sprachen, die Instanzen der Meta-Metaebene MOF sind.

Instanzen von Metamodellen sind Modelle, die Informationen eines bestimmten Teilaspektes eines Systems möglichst formal und eindeutig beschreiben sollen. Modelle können die Komplexität eines Systems reduzieren, wenn Prinzipien wie Abstraktion, Zerlegung oder Hierarchiebildung angewendet werden. Generell können Modelle in formale und nichtformale Modelle mit grafischer oder textueller Repräsentation eingestuft werden. In der MDA ist die Verwendung von formalen Modellen entscheidend. Da-

mit ein Modell formal gültig ist und für Transformationen benutzt werden kann, sollten folgende Punkte erfüllt sein:

- Einem Modell liegt eine formal eindeutige Syntax zugrunde. Im Falle von UML bedeutet dies, dass die verwendeten Notationselemente genau definiert sind.
- Die Beziehungen, die zwischen Modellelementen möglich sind, sind formal genau festgelegt
- Den einzelnen Modellelementen und deren Beziehungen zueinander ist für eine definierte Abstraktionsebene (Modellierungsebene) eine eindeutige Bedeutung (Semantik) zugeordnet

2.3.2 Sichtweisen und Sicht

Da die MDA die Trennung der Businesslogik von der Anwendungslogik vorsieht, werden verschiedene Sichtweisen (*Viewpoints*) auf ein System definiert, die den Fokus der Betrachtung auf bestimmte Konzepte und Strukturregeln setzen, um gewisse Aspekte eines Systems zu abstrahieren. Es existieren unterschiedliche Sichtweisen auf ein System, die von abstrakt bis sehr detailliert reichen:

- Die *Compuation Independent Viewpoint* ist eine abstrakte Sichtweise, die sich auf die Umgebung und die Anforderungen eines Systems konzentriert. Genauere Spezifizierungen zu Struktur und Prozessen sind nicht Bestandteil dieser Sichtweise.
- Die *Platform Independent Viewpoint* erfasst die Funktionalität und die formale Spezifikation eines Systems, ohne auf spezifische Details zu Plattformen einzugehen
- In der *Platform Specific Viewpoint* wird die *Platform Independent Viewpoint* durch Hinzufügen von Details zur Realisierung eines Systems auf einer spezifischen Plattform erweitert.

Eine Sicht hingegen ist ein Modell, das ein System aus einer bestimmten Sichtweise modelliert. Die OMG unterscheidet die Sichten *Computational Independent Model* (CIM), *Platform Independent Model* (PIM) und *Platform Specific Model* (PSM).

2.3.3 Computation Independent Model

Mithilfe des *Computational Independent Model* (CIM) werden die Anforderungen an ein Softwaresystem sowie dessen Einsatzumgebung beschrieben. Das CIM repräsentiert die Geschäfts- oder die Domänensicht des zu entwickelnden Systems und ist unabhängig von technischen Aspekten der Implementierung. Es liefert keine Beschreibung der internen Strukturen und des internen Verhaltens eines Systems. Im OMG MDA Guide [OMGM03] wird das CIM als Modelltyp mit der höchsten Abstraktionsstufe innerhalb der MDA bezeichnet. Bei der Modellierung des CIM mit UML können z.B. Anwendungsfalldiagramme, Interaktionsdiagramme und Aktivitätsdiagramme zum Einsatz kommen. Das CIM bildet die Ausgangsbasis für die weiteren Modelltypen *Platform Independent Model* (Abschnitt 2.3.4) und *Platform Specific Model* (Abschnitt 2.3.5).

2.3.4 Platform Independent Model

Im Gegensatz zum *Computational Independent Model* wird im *Platform Independent Model* (PIM) die Struktur und das Verhalten eines Softwaresystems spezifiziert. Das PIM beschreibt die Geschäftslogik und die Funktionalität eines Systems ohne Berücksichtigung spezifischer Plattformen.

Das PIM wird auf der Basis des CIMs entwickelt. Die Erstellung des PIMs aus dem CIM kann im Allgemeinen jedoch nicht automatisiert werden, da die Entscheidungen, welche Bereiche eines Geschäftsmodells softwaretechnisch unterstützt werden sollen, nicht von einer Maschine übernommen werden können. Ein wesentlicher Unterschied zum CIM besteht darin, dass beim PIM die Beschreibung von Systemen nach formalen

Kriterien erfolgt und es somit für Transformationen benutzt werden kann. Da das PIM die Ausgangsbasis für eine Automatisierung bildet, kann es als Kernstück der modellgetriebenen Architektur betrachtet werden [VJ07].

Bei der Spezifikation des PIMs spielen ausschließlich fachliche Aspekte eine Rolle, von implementierungstechnischen Details wie Software- und Hardwareplattformen wird abstrahiert. Das entstandene Modell hat auch dann Gültigkeit, wenn überhaupt keine Software entwickelt wird. Es sei hier erwähnt, dass sich die Unterscheidung, ob ein Modell als plattformabhängig oder plattformunabhängig angesehen wird, nicht immer als einfach gestaltet. Die OMG definiert zudem den Plattformbegriff als sehr allgemein. So kann die Betrachtungsweise sehr entscheidend sein, ob ein Modell als plattformunabhängig oder plattformabhängig bezeichnet wird. Im Folgenden wird auf eine Trennung der Modelltypen im Kontext von Plattformen näher eingegangen.

Plattform-Modell

Plattformunabhängige Modelle bilden die Grundlage für plattformspezifische Modelle. Um plattformspezifische Modelle automatisiert aus den Plattformunabhängigen zu erhalten, ist eine technische Spezifikation der Zielplattform notwendig [OMGM03]. Diese Spezifikation wird als Plattform-Modell (PM) oder Plattformdeskriptor bezeichnet und beschreibt die Struktur und die Dienste der Zielplattform sowie Möglichkeiten zur Verbindungen unterschiedlicher Komponenten. Es liegen jedoch häufig Spezifikationen in Form von API-Beschreibungen und Benutzerhandbücher vor. Es kann auch vorkommen, dass Spezifikationen ausschließlich in den Köpfen von Softwareentwicklern existieren. Die MDA setzt deshalb auf Standardisierung und fordert eine formale Beschreibung von Modellen, um Automatisierung in Form von Transformationen zu ermöglichen. Für diese Modelle liegt nach dem MDA Guide [OMGM03] bezüglich der Syntax zwar keine Verpflichtung vor, es kann jedoch auf vordefinierte UML Profile (siehe Abschnitt 2.1.6.1) bzw. Plattformen zurückgegriffen werden. Von der OMG herausgegebene Profile sind z.B. CORBA oder Enterprise Application Integration (EAI). Eine vollständige Liste ist unter [www-03] zu finden.

2.3.5 Platform Specific Model

Ein plattformunabhängiges Modell (PIM) kann mithilfe eines Plattformmodells in ein plattformspezifisches Modell (PSM) transformiert werden. Dazu wird ein PIM durch den Einsatz eines Plattformmodells mit Details über eine spezielle Zielplattform erweitert [VJ07]. Wenn ein PSM alle Informationen für die Erzeugung und Ausführung eines Systems innerhalb einer Plattform enthält, beschreibt ein PSM eine Implementierung. Andernfalls kann das PSM erneut als PIM betrachtet werden und durch weitere Transformationen verfeinert werden. Somit kann ein PSM in ein weiteres PSM überführt werden, da der Plattformbegriff als relativ verstanden werden kann. Im Weiteren wird auf ein PSM als Transformation eines PIMs eingegangen.

Die Beschreibung des PSMs erfolgt wie bei einem PIM wieder nach formalen Kriterien. Zur Generierung eines PSMs aus einem PIM gehören Annotationen, die für die Abbildung definierter Eigenschaften aus dem PIM in neue Eigenschaften des PSMs benötigt werden. Ein PSM kann als Erweiterung eines PIMs z.B. Stereotypen (siehe Abschnitt 2.1.6.2), spezielle Klassen oder die Definition von Typen beinhalten, die zusammen ein Profil für Plattformen wie .NET definieren. Das durch die Transformation entstandene PSM stellt das System aus der *Platform Specific Viewpoint* (PSV) (siehe Abschnitt 2.3.2) dar und ist technologieabhängig. Durch das zugrunde liegende Meta-Modell wird ein PSM auf genau eine technische Umgebung ausgerichtet. Da ein PSM eine Spezialisierung eines PIMs darstellt, dürfen die im PSM vorgenommenen Erweiterungen nicht den Definitionen im PIM widersprechen.

2.3.6 Transformationen

Die Überführung eines Modells auf eine auf der Zielplattform ausführbare Anwendung kann durch (mehrstufige) Transformationen erreicht werden. In der MDA werden diese in Modell-zu-Modell- und Modell-zu-Code-Transformationen unterschieden. Letztere werden auch als Codegenerierung bezeichnet. Bei Modell-zu-Modell-Transformationen kann ein PIM in ein weiteres PIM oder in ein PSM überführt werden. Je nach Betrachtungsweise kann ein PSM auch in ein weiteres PSM transformiert werden. Besitzt ein PSM alle relevanten plattformspezifischen Informationen, kann mithilfe eines Generierungstemplates Programmcode erzeugt werden. Kontextabhängig kann es vorkommen, dass ein PIM bereits alle Informationen für eine Codegenerierung bereitstellt [OMGM03]. In diesem Fall muss ein PIM nicht erst in ein PSM transformiert werden, bevor Quellcode erzeugt werden kann. Die dafür benötigten Architekturentscheidungen sind entweder in Werkzeugen direkt, in Templates, in Programmbibliotheken oder in Code-Generatoren implementiert. In solch einem Kontext kann ein Anwendungsentwickler ein PIM erstellen, das vollständig bezüglich der Klassifikation und der Struktur ist.

2.4 Executable UML

Bei executable UML steht das Modell wie bei der MDA (siehe Kapitel 2.3) im Vordergrund der Softwareentwicklung. Systementwickler und Benutzer können mit executable UML ausführbare Modelle eines Systems entwerfen, die getestet und für eine spezifische Zielplattform in Code transformiert werden können. Um Modelle direkt ausführen zu können, ist eine entsprechende Entwicklungsumgebung erforderlich, die Modelle interpretieren und mit eventuell vorhandenem Code verbinden kann. Zusätzlich muss ein Modell über eine detaillierte Verhaltensbeschreibung verfügen. Mithilfe einer Action Language, auf die in Abschnitt 2.4.1 näher eingegangen wird, kann dieses präzise Verhalten spezifiziert werden. Executable UML kann als eine konkrete Implementierung der MDA mit einer Ausnahme angesehen werden [GC04]. Während die MDA eine Transformation von plattformunabhängigen Modellen (PIM) (siehe Abschnitt 2.3.4) zu plattformspezifischen Modellen (PSM) (siehe Abschnitt 2.3.5) vorsieht, bevor Programmcode erzeugt wird, entfällt dieser Zwischenschritt bei executable UML. Viele UML Modellierungswerkzeuge, die executable UML unterstützen, generieren Programmcode direkt aus einem PIM ohne vorherige Transformation in ein PSM [CG04]. Die für die Codeerzeugung benötigten Entscheidungen sind in Generierungstemplates oder Codegeneratoren implementiert.

Eine Simulation mit executable UML Modellen kann entweder über einen Interpreter erfolgen, der die entsprechenden Modellelemente zur Laufzeit interpretiert und ausführt oder indem das Modell automatisch in Code übersetzt wird, der dann kompiliert und in einer Simulationsumgebung ausgeführt wird. Benutzer können auch unvollständige Modelle testen und erhalten ein schnelles Feedback darüber, wie sich UML Modelle bei ihrer Ausführung verhalten. Diese Vorgehensweise erlaubt eine iterative Entwicklung von Modellen, bei der Änderungen sofort sichtbar werden. Fehler können dadurch schon während der Designphase erkannt werden. Der Benutzer hat außerdem die Möglichkeit, das Verhalten von Modellen sowie Variationen bei Modellen in Echtzeit interaktiv zu entdecken. Diese Arbeitsweise kann zu einer innovativen und schnellen Entwicklung von Modellen führen [RDFS01], da die Zeitverzögerung zwischen einer Änderung am Modell und der Ausführung des Modells relativ gering ist. Nachdem sich ein Modell bei der Ausführung wie erwartet verhält, kann in einem zweiten Schritt Programmcode für eine spezifische Zielplattform erstellt werden.

2.4.1 Action Language

Obwohl UML verhaltensbezogene Aspekte eines Systems ausdrücken kann, sind diese jedoch entweder unvollständig (z.B. Sequenzdiagramme) oder nur auf bestimmte Typen von Systemen anwendbar (z.B. Systeme mit einer endlichen Anzahl von Zu-

ständen) und somit für eine Ausführung oder eine vollständige Codegenerierung nur eingeschränkt nutzbar [GCKP07]. Eine Möglichkeit dynamische Informationen hinzuzufügen ist über eine Action Language. Eine Action Language ist auf dem Abstraktionsniveau höher als herkömmliche objektorientierte Programmiersprachen wie Java oder C# einzuordnen. Desweiteren ist sie einfacher und weniger umfangreich. Beispielsweise müssen Threads nicht explizit ausgedrückt werden. Derzeit existiert keine standardisierte Action Language für UML. Es gibt einige proprietäre Action Languages, auf die in Abschnitt 3.4 näher eingegangen wird.

Das Verhalten von executable UML Modellen kann anhand einer Action Language genau beschrieben werden. Eine Action Language definiert Operationen, um z.B. Manipulationen auf Objekten durchführen zu können oder logische Konstrukte für Algorithmen zu spezifizieren [FLMJ07]. Mithilfe von Aktionen (siehe Abschnitt 2.1.5.1) können diese Operationen ausgeführt werden. Beispiele, die aus Aktionen im Rahmen einer Action Language resultieren, sind Objekterzeugung, Methodenaufrufe, (allgemeine) Berechnungen durchführen, Attributwerte auslesen oder schreiben. Aktionen können in Prozeduren gruppiert werden, die aus Sequenzen von Aktionen bestehen. Dabei ist unter einer Prozedur ein Aktivitätsdiagramm zu verstehen, das Aktionen enthält, die zusammen mit anderen Knoten wie Entscheidungs-, Splittings- und Synchronisationsknoten verwendet werden können.

2.4.2 Executable UML Entwicklungsprozess

Der executable UML Entwicklungsprozess ist ein Systementwicklungsverfahren, das auf der Erfahrung mit ausführbaren Modellen aus den Bereichen der Telekommunikation, der Automobilindustrie, der Luft- und Raumfahrt und des Verteidigungssektors basiert [RCFP03]. Der Prozess beinhaltet das Prinzip der Erstellung von präzisen und ausführbaren Modellen des zu entwickelten Systems. Auf diesen Modellen sollen dann spezifische Tests durchgeführt werden und eine systematische Strategie definiert werden, mit der die Modelle in Code für das gewünschte Zielsystem transformiert werden können. Der executable UML Prozess [SW02] zeichnet sich durch folgende Charakteristika aus:

- Präzise Modelle, mit denen Simulationen durchgeführt werden können: Ein Modell ist im Sinne von executable UML vollständig, wenn es alle vorgesehenen Test durch Simulationen erfolgreich besteht.
- Mithilfe der einfachen Notationen von UML können Kunden, Hardwareentwickler und Manager die entstehenden Modelle verstehen und Feedback über diese geben.
- Eine nachvollziehbare und wiederholbare Partitionsstrategie, die auf der Trennung von Betrefflichkeiten basiert.
- Analysemodelle, die sowohl während des Designs als auch bei der Codeimplementierung ohne fragwürdige Interpretation der Modelle benutzt werden können
- Implementierung durch Transformationen, in der das gesamte System automatisch aus dem Modell generiert werden kann, indem das Modell durch eine Menge von spezifizierten Regeln in ein Zielsystem überführt wird.
- Wiederverwendbarkeit: Eine Menge von Objekten kann als einzelne komplexe Komponente wiederverwendet werden.

3 Stand der Technik

Es existieren mehrere kommerzielle und nichtkommerzielle Modellausführungsrahmenwerke (*model execution frameworks*) bzw. executable UML Werkzeuge, die die Ausführung von UML Verhaltensmodellen, insbesondere von Zustands(maschinen)-diagrammen und Aktivitätsdiagrammen, unterstützen. Eines der ersten Werkzeuge, mit dem Modelle ausgeführt werden konnten, war Rational Rose Real Time, auch bekannt als Rational Rose Technical Developer. Andere Werkzeuge, die einen ähnlichen Ansatz benutzen, sind z.B. Kennedy Carter Tools for xUML, Kabira Tools (Kabira Technologies), BridgePoint Development Suite von Project Technology, I-Logix Tool Rhapsody, TAU G2 und visualState. Damit ein Werkzeug UML Modelle ausführen kann, muss es über eine Action Language (siehe Abschnitt 2.4.1) verfügen. Viele dieser Werkzeuge benutzen eine proprietäre Action Language, was zu Interoperabilitätsproblemen bei der Ausführung von Modellen mit unterschiedlichen executable UML Werkzeugen führen kann. Im Folgenden wird auf einige zum Teil in der Wissenschaft entwickelte Werkzeuge näher eingegangen. Danach wird ein Überblick über proprietäre Action Languages gegeben und anschließend eine konkrete Action Language näher besprochen.

3.1 Populo

Populo ist Werkzeug, das UML 2.0 Modelle ausführen und debuggen kann. Dabei gibt es keine Einschränkung bzgl. der Wahl eines UML Editors bei der Erstellung von UML Modellen. Entwickler können in Verbindung mit Populo weiterhin mit ihren bevorzugten UML Editoren arbeiten. Insbesondere für diese Diplomarbeit ist von Interesse, dass Populo UML 2.0 Aktivitätsdiagramme (siehe Abschnitt 2.1.5) ausführen kann. Viele momentan verfügbaren UML Werkzeuge verwenden Zustands(maschinen)-diagramme bei der Ausführung von Modellen zur Verhaltensbeschreibung. Im Folgenden wird näher auf die Funktionalität und die Eigenschaften von Populo eingegangen. Außerdem wird gezeigt, wie eine Robotersteuerung mit Populo modelliert werden kann.

Die Idee sowie die Motivation, die zur Erstellung des Populo Werkzeuges geführt hat, ist die Folgende: Durch die Konzentration auf die Modelle im Softwareentwicklungsprozess wird die Abstraktionsebene angehoben, auf relevante Details für ein bestimmtes Ziel fokussiert und irrelevante Details ausgeblendet. Diese Modelle müssen jedoch getestet werden, wenn sie als wertvoll betrachtet werden sollen. Folgendes Beispiel, das von Fuentes, Manrique und Sanchez in [FLMJ08] modelliert worden ist, soll diese Idee erläutern:

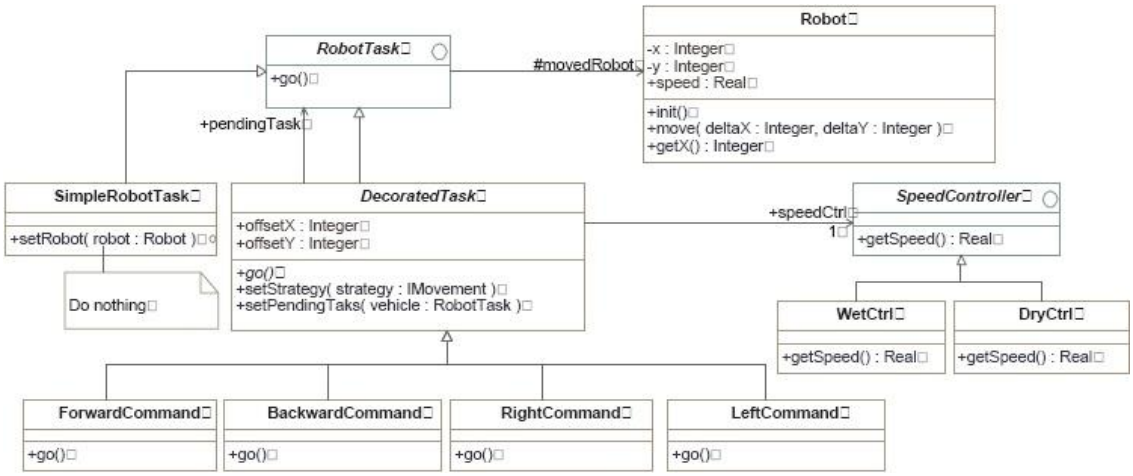


Abbildung 20: Modellierung einer Robotersteuerung [FLMJ08]

In Abbildung 20 wird die Modellierung einer Robotersteuerung dargestellt, in der verschiedene Bewegungen durch spezifische Kommandos in einer bestimmten Umgebung ausgeführt werden können. Der Roboter kann sich durch Kommandosequenzen nach vorne, hinten, rechts und links bewegen. Zusätzlich sind verschiedene Strategien bei jeder Bewegungsart definiert. Abhängig davon, ob es regnet oder die Sonne scheint, bewegt sich der Roboter mit unterschiedlicher Geschwindigkeit fort. Die Kommandos werden mit dem *Decorator-Pattern* [GEHR94] implementiert. Ein *SimpleRobotTask* macht praktisch nichts außer den Benutzer darüber zu informieren, wenn ein Task beendet wurde. Jeder Befehl wird erstellt, indem dieser Basistask *dekoriert* wird. Zusätzlich muss jeder Befehl die Geschwindigkeit der Klasse *SpeedController* erhalten, die nach dem *Strategy-Pattern* [GEHR94] erstellt wird. *SpeedController* liefert abhängig vom Wetter unterschiedliche Werte zurück. Nach Ansicht von Fuentes, Manrique und Sanchez [FLMJ08] ist es leichter, diese *Patterns* mit einem UML Modell anstatt mit deren Implementierungscode zu verstehen, da das direkte Arbeiten mit Code spezifische Programmierkenntnisse der Zielsprache voraussetzen. UML erlaubt seinen Benutzern von diesen Details zu abstrahieren und sich auf objektorientierte Elemente wie Vererbung, Beziehungen und Assoziationen zwischen Klassen zu konzentrieren. Wie bereits erwähnt, ist es wünschenswert, dass diese Modelle ausgeführt und getestet werden können. Für das Roboterbeispiel bedeutet dies, dass das erstellte Modell ausgeführt werden kann und eine Überprüfung stattfindet, ob sich der Roboter wie erwartet verhält und die Benutzung und Komposition der verwendeten *Design-Patterns* richtig ist.

Funktionalität und Eigenschaften von Populo:

- Populo verfügt über typische Debugger-Funktionalität wie Schritt-für-Schritt-Ausführung oder Breakpoint-basiertes Debugging
- Populo ist momentan (Stand 2008) als Eclipse-PlugIn verfügbar
- Populo akzeptiert UML Modelle im XMI Format
- Populo ist erweiterbar, z.B. ist es möglich, neue benutzerdefinierte Aktionen für ein ausführbares UML Profil hinzuzufügen

Die Entwurfsphase des Softwareentwicklungsprozesses sieht mit Populo folgendermaßen aus:

1. Softwareentwickler erstellen UML Modelle für Systeme, indem sie das präzise und vollständige Verhalten dieser Modelle mit Aktionen und weiteren Modellierungselementen wie Entscheidungsknoten oder Splittingknoten spezifizieren. Da spezielle UML Editoren im Gegensatz zu anderen Werkzeugen, die UML Modelle ausführen können, nicht benötigt werden, bietet Populo Kompatibilität sowie Portabilität.
2. Nach der Erstellung der ausführbaren UML Modelle werden diese in Populo geladen. Nun können die Modelle noch innerhalb der Entwurfsphase mit Debugger-Funktionalität ausgeführt werden, um mögliche Fehler oder unerwartetes Verhalten zu entdecken. Während der Implementierungsphase wäre die Behebung von Fehlern deutlich kostenintensiver [FLMJ08].

Aufbau des Populo-Werkzeugs:

- **Modell** (Abbildung 21, Label 1): Diese Sicht zeigt das auszuführende Modell an, indem das Eclipse PlugIn benutzt wird.
- **Ausführung**: (Abbildung 21, Label 2): In dieser Sicht werden Informationen über die Ausführung des Modells angezeigt. Für jede Aktion wird die zugehörige Aktivität oder Zielaktion, die dessen Output empfängt, ausgegeben. Bei Splitting-Knoten wird das Fenster automatisch in Unterfenster, welche die nebenläufigen Ausführungen beinhalten, geteilt.

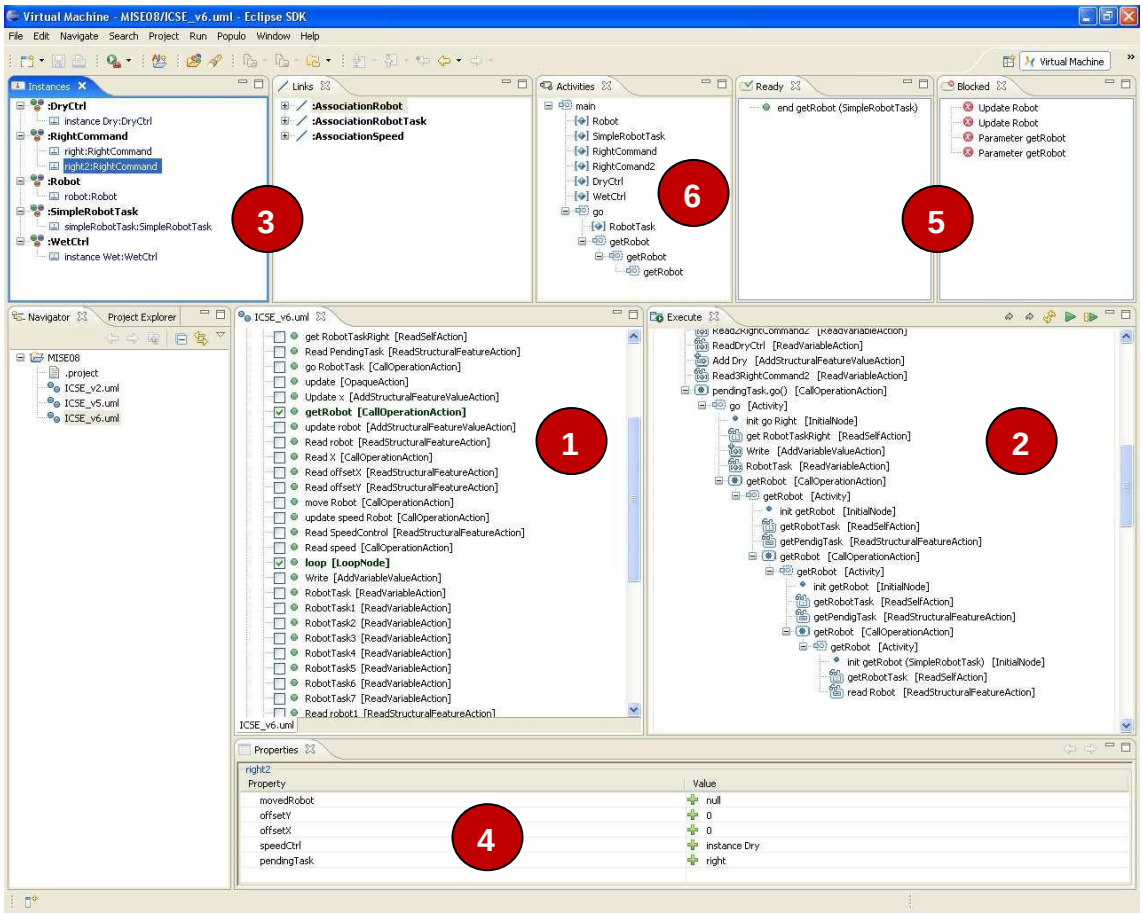


Abbildung 21: Grafische Benutzeroberfläche Populo [FLMJ08]

- **Breakpoints** (Abbildung 21, Label 1): Hier können Breakpoints für die Modellausführung definiert und gesetzt werden.
- **Instanzen und Verbindungen** (Abbildung 21, Label 3): Nach Klasse sortiert werden alle erzeugten Objekte angegeben. Diese Objekte beinhalten Klasseninstanzen, Verbindungsinstanzen und Verbindungsklasseninstanzen.
- **Objekteigenschaften** (Abbildung 21, Label 4): An dieser Stelle werden Attributwerte eines Objektes sowie Variablenwerte angezeigt.
- **Ready und Block** (Abbildung 21, Label 5): In dieser Sicht werden die zur Ausführung bereiten Aktionen (links), die alle Objekt- und Kontrollflüsse erhalten haben, in einer Warteschlange dargestellt. Der Benutzer hat die Möglichkeit, die Reihenfolge der Warteschlange zu verändern. So können z.B. schon getestete Ausführungspfade ausgelassen werden und nur die Pfade, die für den Benutzer von Interesse sind, ausgeführt werden. Alle Aktionen, die auf Kontroll- oder Objektflüsse warten, werden rechts angezeigt.
- **Call Stack** (Abbildung 21, Label 6): Der Call Stack beinhaltet aufgerufene Aktivitäten und strukturierte Knoten, die noch nicht beendet wurden.

Mit diesen Sichten kann das Modell der Robotersteuerung Schritt-für-Schritt und Breakpoint-basiert ausgeführt und ausgetestet werden.

Mit dem Werkzeug Populo ist es möglich, neben UML Zustandsmaschinendiagrammen UML Aktivitätsdiagramme direkt ausführen zu können. Eine anschließende Codegenerierung aus einem Aktivitätsdiagramm wird jedoch nicht unterstützt. Somit beschränkt sich das Werkzeug auf das Testen und Verifizieren von Modellen. Eine Stärke des

Werkzeuges besteht darin, Aktivitätsdiagramme durch benutzerspezifische Aktionen erweitern zu können, die dann zusätzlich ausgeführt werden können. Zusätzlich können UML Modellen mit einem beliebigen UML Editor erstellt werden und in Populo ausgeführt werden.

3.2 Generic Model Execution Framework

Ein weiteres Werkzeug zur Modellausführung ist das *Generic Model Execution Framework (GMEF)*, das Modelle mit einer *Execution Engine direkt* ausführen kann. Mithilfe dieser *Execution Engine* haben Kirshin, Moshkovich und Hartman [KAMD06] einen UML Modell-Simulator (Abbildung 22; *Generic model execution engine* ist als Synonym für *Execution Engine* anzusehen) implementiert, der den Rational Software Architect (RSA) durch zusätzliche Funktionen zur Ausführung von UML Modellen erweitert. Diese Funktionen umfassen:

- Schritt-für-Schritt- sowie *run to breakpoint* Ausführung
- Visualisierung des Verhaltens bei der Modellausführung
- Visualisierung des aktuellen Ausführungszustandes
- Visualisierung von Tokens und Parameterwerten
- Dynamische Objektgenerierung
- Benutzerinitiierte Verhaltensaufrufe und Operationen

Das Ziel dieses Werkzeuges ist es, durch eine Plattform für Modelluntersuchungen, die in den ersten Phasen des Softwareentwicklungsprozesses ansetzt, ein Modell besser verstehen zu können. Durch Modellausführungen kann eine frühe Simulation und das Austesten von Anwendungen stattfinden. Außerdem soll diese Vorgehensweise hilfreich bei der Kommunikation zwischen der IT- und der Geschäftswelt sein. Im Folgenden soll zunächst der Aufbau des Werkzeuges und danach die Execution Engine näher besprochen werden. Anschließend wird auf den Ablauf einer Simulation eingegangen.

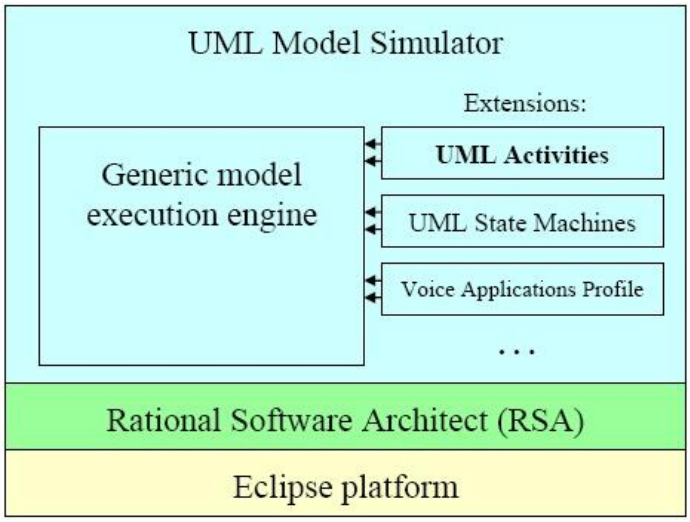


Abbildung 22: UML Simulator [KAMD06]

3.2.1 Aufbau des GMEF-Werkzeuges:

- **Model Explorer View:** Die bisherige RSA Sicht wird durch zwei Punkte im PopUp-Menü erweit*ert.* Mit *Debug Model* wird eine Simulation gestartet. *Add Breakpoint* kann auf jedes ausführbare Modellelement angewendet werden.
- **Debug View:** Diese Sicht umfasst die Steuerung des Ausführungsprozesses (Start, Stopp, schrittweise Ausführung), Objektgenerierung und -zerstörung,

Überwachung der Attributwerte von Objekten und das Aufrufen von Operationen.

- **Call Stack View:** Diese Sicht enthält alle Verhaltensaufrufe zu einer Simulation entsprechend der Aufrufhistorie
- **I/O View:** In dieser Sicht können die vom Modell generierten Outputs sowie an das Modell gesendete Signale beobachtet werden.
- **Ready View:** In dieser Sicht sind alle Elemente aufgelistet, die bereit zur Ausführung sind oder Breakpoints erreicht haben. Der Benutzer kann das nächste Element auswählen, das ausgeführt werden soll.
- **Breakpoint View:** Diese Sicht enthält alle aktiven Breakpoints.
- **Diagram Visualization:** Dieser mächtigste Teil des Werkzeuges kann die Ausführung von UML Verhaltensmodellen (siehe Abschnitt 2.1) visuell darstellen.

3.2.2 Execution Engine

Eine *Execution Engine* stellt Mechanismen zur Realsierung verhaltensbezogener Semantiken sowie zur Beobachtung des Modellverhaltens bereit. Sie wird nach Kirshin, Moshkovich und Hartman [KAMD06] als wesentliche Komponente eines Verhaltensuntersuchungswerkzeuges angesehen. Die *Execution Engine* des Werkzeuges kann die einzelnen Modellelemente von UML Aktivitäts- und Zustandsmaschinendiagrammen durch Interpretation oder Kompilierung direkt ausführen. Dazu wird eine Action Language (siehe Abschnitt 2.4.1) benötigt. Für das GMEF-Werkzeug wird Java als Action Language verwendet. Folgende Anforderungen werden an die Execution Engine des GMEF-Werkzeuges gestellt:

- Beobachtbarkeit: Bereitstellung von Schnittstellen für andere Verhaltensuntersuchungswerkzeuge, um Modellbeobachtung und -manipulation zu ermöglichen
- Steuerbarkeit: Steuerung des Ausführungsprozesses soll erlaubt werden
- Parallelisierung: Parallele Ausführung von unterschiedlichen Abläufen soll unterstützt werden
- Erweiterbarkeit: Mehrere Metamodelle sollen unterstützt werden
- Flexibilität: Unterschiedliche Implementierungen des Verhaltens für dasselbe Modellelement (*semantic variation point*) soll unterstützt werden
- Skalierbarkeit: Sehr große Modelle sollen unterstützt werden
- Einfachheit: Einfache Definition und Modifikation des Verhaltens von Modellelementen soll ermöglicht werden

Beobachtbarkeit und Steuerbarkeit sind generell nicht unbedingt Anforderungen an die *Execution Engine* des GMEF-Werkzeuges [KAMD06]. Sie werden jedoch mit einbezogen, um Interoperabilität zwischen der *Execution Engine* und anderen Entwicklungswerkzeugen, die Debugger und Testgeneratoren beinhalten, zu gewährleisten.

Die von Kirshin, Moshkovich und Hartman [KAMD06] entwickelte Execution Engine enthält alle spezifischen Informationen zur Abbildung von Modellelementen auf ihre Repräsentationen zur Laufzeit. Diese Grundlage zur Ausführung von Modellen kann auf drei verschiedene Arten erfolgen:

1. Durch Interpretation
2. Durch Kompilierung
3. Durch eine Kombination von Interpretation und Kompilierung

Der Interpreter-Ansatz (Abbildung 23) besteht aus den folgenden Schritten:

- Instanzcode wird für jedes Metamodellelement durch den Entwickler der *Execution Engine* geschrieben. Die Ausführung von Modellen erfordert keine vorherige Codegenerierung.

- Zur Laufzeit werden die entsprechenden Modellelemente analysiert. Eine Instanzimplementierung wird zur Laufzeit basierend auf dem zugehörigen Modellelementtyp gewählt.

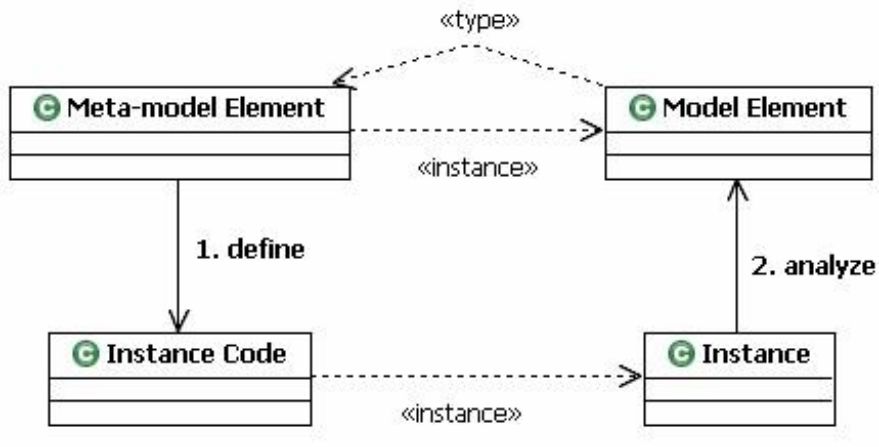


Abbildung 23: Interpreter [KAMD06]

Der Kompilierungs-Ansatz (Abbildung 24) sieht folgende Schritte vor:

- Für jedes Metamodellelement wird eine Kompilierungskomponente durch den Entwickler der *Execution Engine* geschrieben
- Eine Kompilierung wird für jedes spezifische Modell durchgeführt und damit Instancode für alle entsprechenden Modellelemente generiert
- Zur Laufzeit ist alle Logik der Modellelemente bereits im Instanzcode kompiliert. Die Verbindung zwischen einem Modellelement und seinem Instanzcode wird während des Kompilierungsschrittes hergestellt.

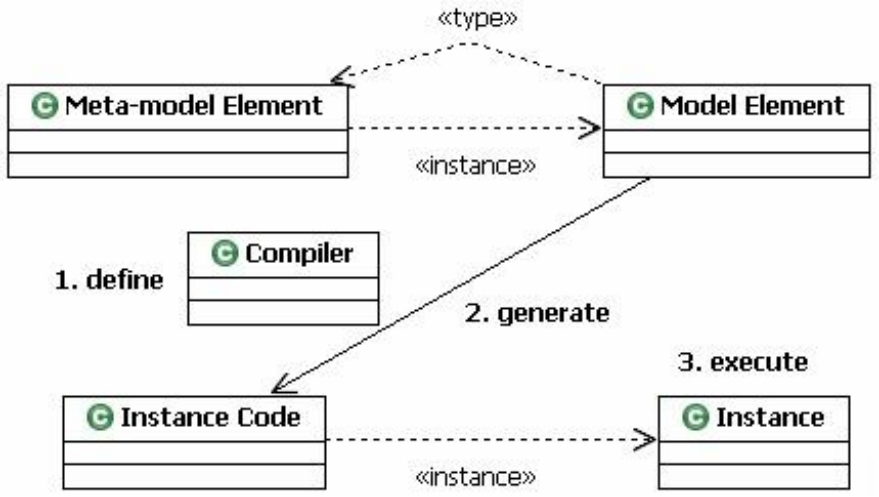


Abbildung 24: Kompilierer [KAMD06]

Eine dritte Möglichkeit besteht darin, die beiden oberen Ansätze zu kombinieren (Abbildung 25). Für einige Modellelemente wird dabei Instanzcode erzeugt, wohingegen die Übrigen zur Laufzeit generiert werden. Um solch ein Modell ausführen zu können, sind folgende Schritte notwendig:

- Für eine Teilmenge der Metamodellelemente wird Intanzcode geschrieben. Für die übrigen Elemente wird eine Kompilierungskomponente verwendet.
- Instanzcode wird für die Modellelemente erzeugt, für die der Kompilierer zuständig ist.

- Zur Laufzeit werden die Modellelemente analysiert, die im vorherigen Schritt nicht kompiliert wurden.

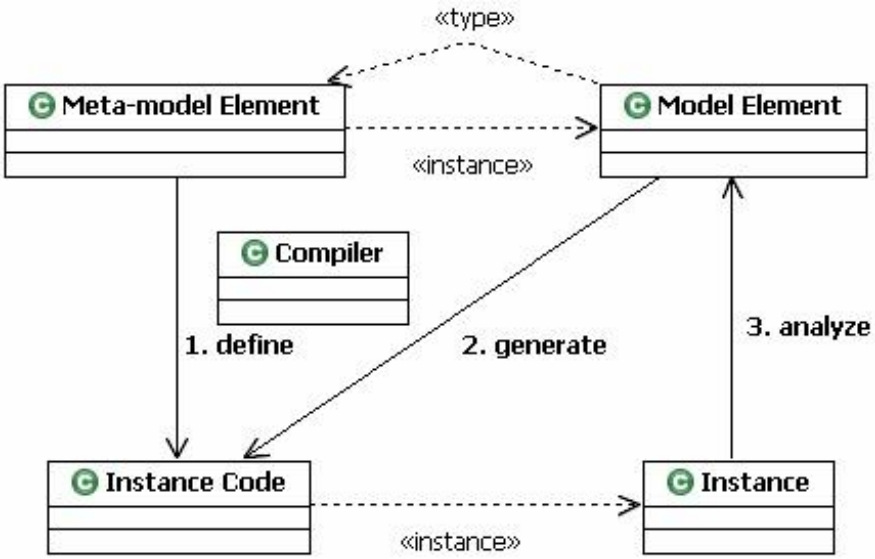


Abbildung 25: Interpreter und Kompilierer [KAMD06]

3.2.3 UML Simulation

Mit der oben beschriebenen Architektur ist es möglich, eine Umgebung für jedes Metamodell zu definieren. Dazu wird für jedes Element des UML Metamodells eine Instanz definiert, die dessen Verhalten beschreibt. Die verwendete Architektur von Kirshin, Moshkovich und Hartman [KAMD06] ist in Java implementiert, in der eine Instanz eines Metamodellelements durch eine Java Klasse abgebildet wird. Für ein einziges Metamodellelement können mehrere Instanzen definiert werden, um sogenannte *semantic variation points* zu unterstützen, die einem Metamodellelement unterschiedliche Bedeutungen zuweisen können. Durch eine Konfigurationsdatei wird die passende *semantic variation* ausgewählt. Zusätzlich können UML Profile unterstützt werden, indem Instanzen für stereotypisierte Modellelemente (siehe Abschnitt 2.1.6) definiert werden. Da Instanzen durch Java Klassen abgebildet werden, kann das Speichermanagement, die Garbage Collection und die Behandlung von primitiven Datentypen von der Java Virtual Machine übernommen werden. Diese Umsetzung erlaubt es Java als Action Language zu verwenden. Indem die Verbindung zwischen Instanzen und den Modellelementen genutzt wird, kann unter anderem eine Visualisierung bei der Ausführung von UML Modellen erfolgen. In Abbildung 26 ist die Visualisierung eines Beispielablaufes zu sehen. Ausführbarbereite Knoten sind grün, von Tokens überquerte Kanten sind blau und Knoten, die ein Token anbieten, sind magenta.

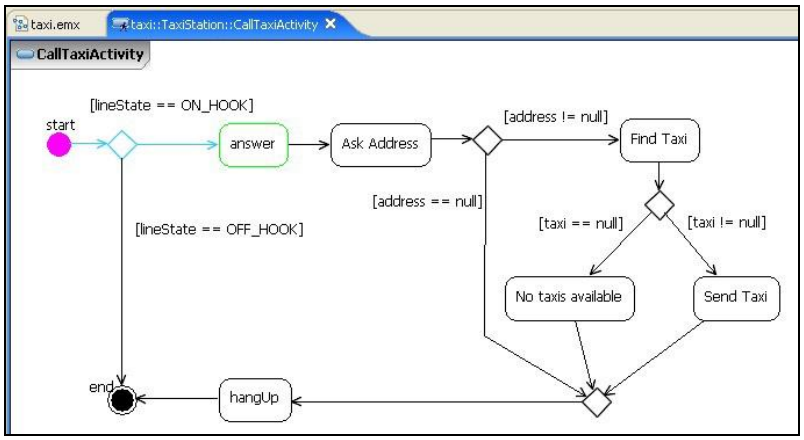


Abbildung 26: Visualisierung einer Simulation [KAMD06]

Kirshin, Moshkovich und Hartman [KAMD06] haben für die Simulationen ein UML Profil erstellt, das genutzt wird, um Folgendes zu spezifizieren:

- Elemente, die Java Code enthalten (Java wird als Action Language verwendet)
- Externe Klassenpfade, um existierenden Code zu verlinken
- Initiale Systemkonfiguration, bei der Objekte zu Beginn einer Simulation erstellt werden
- Vordefinierte Parameterwerte
- Vordefinierte Entscheidungen bei Entscheidungsknoten innerhalb von Aktivitätsdiagrammen

Das GMEF-Werkzeug eignet sich sehr gut für die Ausführung von Aktivitätsdiagrammen. Obwohl Programmcode für die Simulation selbst anhand des Kompilierungsansatzes erzeugt werden kann, ist eine Codegenerierung aus einem Aktivitätsdiagramm nicht Bestandteil des Werkzeuges. Weiter gibt es keine Möglichkeit, bereits existierenden Code für eine Simulation hinzuzufügen.

3.3 Democles

Democles ist ein UML Modellierungswerkzeug, mit dem UML Modelle ausgeführt werden können. Außerdem ist es möglich, aus den Modellen Programmcode zu erzeugen. Damit eine Ausführung eines Modells mit Democles erfolgen kann, muss die statische Struktur und das dynamische Verhalten eines Systems im Modell abgebildet werden. Die statische Struktur eines Modells kann durch Verwendung von modifizierten UML Klassendiagrammen, die Anfrageoperationen von Änderungsoperationen trennen, beschrieben werden. Für die Verhaltensmodellierung wird ein MOF-basiertes Metamodell benutzt, das Änderungsoperationen und Eigenschaften enthält und diese mit Assoziationen und OCL Ausdrücken verbinden kann [GCKP07]. Eine eigens entwickelte Verhaltensmodellierungssprache, mit der Modelle ausführbar sind, basiert auf den beiden Elementtypen Ereignis und Eigenschaften. Zusätzliche Elemente sowie *OCL Code Snippets* unterstützen die Ausführbarkeit.

Democles ist ein Eclipse-PlugIn, das Überlagerungssichten der Struktur und des Verhaltens von Systemen ermöglicht. Es unterstützt den Benutzer mit folgenden Fähigkeiten [GCKP07]:

- Eine Systemstruktursicht präsentiert UML Klassendiagramme des Systems inklusive Paketstruktur. Klassen können direkt im Diagramm editiert werden und es kann ein manuelles oder automatisches Layout ausgeführt werden.
- Eine Ereignisnavigationssicht unterstützt die schnelle Auffindung von allen externen Ereignissen, die den Zustand eines Systems beeinflussen können.
- Ein System kann ausgeführt werden, indem ein interaktiver Debugger benutzt wird, der einen Instanzgraphen anzeigt. Ereignisse können ausgelöst und anschließend die Resultate betrachtet werden.
- Das Werkzeug bietet volle Unterstützung, um OCL Ausdrücke (siehe Abschnitt 2.1.4) mit Codevervollständigung und Syntax-Hervorhebung für Struktur- und Verhaltenssichten zu schreiben.
- Änderungen des Modells werden unmittelbar durch die syntaktische Gültigkeit von Modellelementen und benutzerspezifischem *Code Snippets* sichtbar.

Obwohl der Fokus bei der Ausführung von Modellen auf Verhaltensmodellen liegt [GCKP07], werden auch strukturelle Modelle benötigt, da strukturelle Elemente in den Verhaltensmodellen wiederverwendet werden. Für die strukturelle Modellierung kommen UML Klassendiagramme mit folgenden Änderungen zum Einsatz, die formal durch ein UML Profil ausgedrückt werden können:

- Operationen dienen nur zur Abfrage und können den Zustand des Systems nicht ändern.
- Ereignisse können hinzugefügt werden, die als modifizierende Operationen verstanden werden, deren Semantiken im Verhaltensmodell detailliert beschrieben werden.
- Initiale Eigenschaftswerte werden jeweils durch das OCL Constraint *init* definiert (siehe Abschnitt 2.1.4), das einen OCL Ausdruck darstellt, der einen initialen Wert bestimmt.
- Der *Body* einer Abfrageoperation wird durch das OCL Constraint *body* definiert, das mithilfe eines OCL Ausdrucks beschreibt, was eine Anfrageoperation zurückliefern soll.

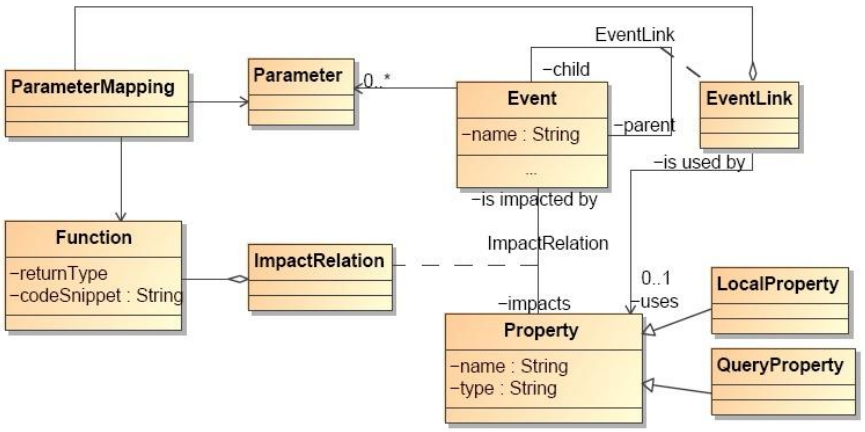


Abbildung 27: EP Modell [GCKP07]

Für die Verhaltensmodellierung wird eine neue ausführbare Modellierungssprache (executable modeling language) eingeführt. Diese neue Sprache mit Namen EP (Events und Properties) drückt das Verhalten eines Systems aus, indem sie Ereignisse und Eigenschaften der zuvor erwähnten Klassendiagramme verwendet. In Abbildung 27 wird die abstrakte Syntax der Sprache beschrieben. Die Hauptentitäten sind Ereignisse (Events), Eigenschaften (Properties) und Funktionen (Functions). Ein Ereignis kann Parameter mit Namen und Typ haben. Ein Ereignislink (Event Link) verbindet ein Ereignis mit seinem Kindereignis (Child Event) über einen Eigenschaftslink, der entweder eine Eigenschaft oder eine parameterlose Anfrageoperation ist. Ereignisse können den Systemzustand durch Änderung einer Eigenschaft in dessen Klasse modifizieren.

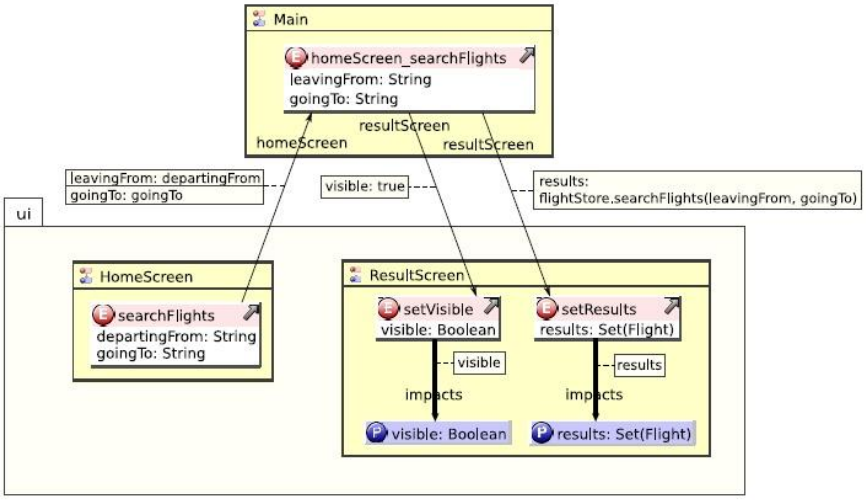


Abbildung 28: Sicht des EP Modells für das searchFlights Ereignis [GCKP07]

In Abbildung 28 ist eine Modellierung zu sehen, die beschreiben soll, wie sich der Systemzustand beim Auslösen eines Ereignisses ändert. Der alte Zustand gibt den Systemzustand direkt vor der Auslösung eines Ereignisses wieder. Entsprechend steht der neue Zustand für den Systemzustand, der direkt nach Vorkommen eines Ereignisses eintritt. Der Wert einer Anfrageoperation auf einer Instanz wird bestimmt durch die Berechnung des Wertes des OCL Ausdrucks auf dem alten Zustand. Wenn ein Ereignis ausgelöst wird, werden alle durch dieses Ereignis beeinflussten Eigenschaften auf die Werte der OCL Ausdrücke gesetzt [GCKP07].

Das Werkzeug Democles führt für die Verhaltensbeschreibung von Modellen eine neue Modellierungssprache ein. Damit wird eine Portabilität und Wiederverwendung von erzeugten Modellen auf anderen Werkzeugen schwierig. Ein Vorteil dieser neuen Sprache ist, dass durch die Verwendung von Eigenschaften und Ereignissen zur Verhaltensbeschreibung zusammen mit der statischen Strukturbeschreibung durch modifizierte UML Klassendiagramme präzise Modelle zur Ausführung erstellt werden können.

3.4 Action Languages

Mehrere Research Gruppen haben in der jüngeren Vergangenheit executable UML-Werkzeuge mit spezifischen Action Languages entwickelt. Einige Beispiele dieser Action Languages sind ASL von Kennedy Carter, Kabira AS von Kabira Company und AL von Project Technology. Die folgende Action Language OAL (Object Action Language) soll hier näher vorgestellt werden, da sie viele Gemeinsamkeiten mit der für diese Diplomarbeit entwickelte Action Language (siehe Abschnitt 4.2) aufweist:

3.4.1 OAL

Die Object Action Language (OAL) [OALM02] wird benutzt, um die Ablaufsteuerung von ausführbaren Aktionen zu definieren. Aktionen bestehen aus einer Menge von Anweisungen. Jede Anweisung kann entweder eine einfache Anweisung wie z.B. der Zugriff auf ein Klassenattribut oder eine komplexe Steuerungslogikstruktur wie z.B. ein if-Konstrukt sein. Die Ausführung einer Aktion beginnt mit der ersten Anweisung. Alle weiteren Anweisungen werden sequentiell abgearbeitet. Eine Aktion gilt als beendet, wenn die letzte Anweisung vollständig ausgeführt wurde. OAL Anweisungen setzen sich zusammen aus:

- Schlüsselwörtern (z.b. create, assign, object)
- Logischen und arithmetischen Operationen
- Den Namen der modellierten Elementen (z.B. Klassen, Attribute)
- Lokalen Variablen

Ein OAL Ausdruck hat Zugriff auf spezifische Datenelemente und kann diese auch erzeugen. Die folgenden Datenelemente können zu Beginn und während der Ausführung einer Aktion gelesen werden:

- Konstanten
- Attributwerte einer Klasse
- Zusätzliche Datenelemente eines Ereignisses, das die Aktion initialisiert hat

Folgende Datenelemente können während einer Aktion erstellt werden:

- Lokale Variablen
- Attributwerte einer Klasse
- Zusätzliche Datenelemente eines Ereignisses, das während einer Aktion ausgelöst worden ist

Alle referenzierten bzw. erstellten Datenelemente müssen einen Datentyp wie *Integer*, *String* oder *Boolean* haben. Im Folgenden sollen nun einige Programmierkonstrukte zu OAL aufgezeigt werden [OALM02].

While-Schleife:

```
// Create 20 doors with IDs 1-20
i = 1;
while (i <= 20)
create object instance d of DOOR;
d.ID = i;
i = i + 1;
end while;
```

Objekt erstellen:

```
create object instance car of C;
```

Attributwerte schreiben:

```
//First, create a new instance
create object instance my_account of ACCT;

// Now set attribute values using the returned instance handle.
my_account.branch = rcvd_evt.this_branch;
my_account.account_number = rcvd_evt.new_account;
```

Attributwerte lesen:

```
// Create new instance and get handle.
Create object instance myrobot of R;

// Use the instance handle to read attribute values.
myx = myrobot.x_position;
myy = myrobot.y_position;
```

Aufruf einer Funktion:

```
// function expression as an assignment statement read value
volume = DD::get_volume();
```

4 Modellierung, Simulation und Codegenerierung von Maschinensteuerungen

In diesem Kapitel werden zunächst die Anforderungen untersucht, um Modelle ausführen und Programmcode aus ihnen erzeugen zu können. Zusätzlich wird ein Konzept vorgestellt, mit dem ein Fahrzeug durch die Simulation von Modellen gesteuert werden kann. Bevor auf die Simulation in Abschnitt 4.4 eingegangen wird, werden in Abschnitt 4.3 die einzelnen Fahrzeugkomponenten näher besprochen. In Abschnitt 4.5 wird abschließend gezeigt, wie Code aus den Modellelementen des Aktivitätsdiagrammes generiert wird.

4.1 Ziele, Anforderungen und Konzept

Ein Ziel dieser Arbeit ist es, Modelle in Form von UML Aktivitätsdiagrammen im Sinne von executable UML (siehe Abschnitt 2.4) direkt ausführen zu können. Mit der Simulation dieser Modelle soll dann gezeigt werden, wie ein von der Firma SparxSystems zur Verfügung gestelltes Fahrzeug gesteuert werden kann. Zusätzlich soll Programmcode aus den ausführbaren Modellen erzeugt werden. Der Ansatz dieser Arbeit vereint das Konzept der executable UML durch die Simulation von Aktivitätsdiagrammen mit den Konzepten der MDD (siehe Abschnitt 2.2) und MDA (siehe Abschnitt 2.3) durch eine modellgetriebene Entwicklung mit Codegenerierung.

Das zu Testzwecken benutzte Fahrzeug ist mit 2 Servomotoren für die Lenkung und den Antrieb ausgestattet. Weiter sind ein Kompassmodul und zwei Sensoren am Fahrzeug angebracht. Mit den ausführbaren Aktivitätsdiagrammen soll ein beliebiger Fahrablauf modelliert werden können. Dazu findet eine Kommunikation zwischen dem Fahrzeug und dem ausgeführten Modell statt, wodurch das Fahrzeug indirekt gesteuert werden kann. Der Zweck der Modellierung eines Fahrablaufs mit Aktivitätsdiagrammen besteht darin, dass sich Fahrzeuge bei der Simulation sowie bei der Ausführung nach der Codegenerierung selbständig und autonom fortbewegen können. Den Mittelpunkt bildet somit die Entwicklung eines korrekten Modells zur autonomen Steuerung des Fahrzeuges. Damit wird der Abstraktionslevel bei der Entwicklung einer Fahrzeugsteuerung auf die Modellebene gesetzt. Der Vorteil dieser Vorgehensweise liegt darin, dass verschiedene Fahrabläufe relativ schnell anhand einer grafischen Modellierung mit Aktivitätsdiagrammen entwickelt werden können. Diese Fahrabläufe können dann sofort in Simulationen getestet und bei Bedarf geändert werden. Zusätzlich erfordert die Modellierung eines Fahrablaufs keine speziellen Programmierkenntnisse, womit die Erstellung einer Fahrzeugsteuerung einem breiteren Personenkreis zugänglich gemacht wird. Wenn eine Fahrzeugsteuerung ohne die Verwendung von ausführbaren Modellen entwickelt werden soll, besteht eine Möglichkeit darin, jeden gewünschten Steuerungsablauf mit entsprechendem Aufwand zu programmieren. Bei jeder Änderung des Fahrablaufs ist dann eine Änderung am Programm vorzunehmen.

Nach einer erfolgreichen Simulation bzw. auch ohne vorherige Simulation ist es möglich, Code aus den Modellen zu generieren, der im Anschluss automatisch in eine spezifische Umgebung eingefügt und kompiliert werden kann. Das Fahrzeug kann dann durch die Ausführung des kompilierten Codes autonom gesteuert werden. Der Unterschied der beiden Varianten liegt in der Ausführungsgeschwindigkeit des Fahrablaufs und der Entwicklungsgeschwindigkeit eines korrekten Modells. Änderungen am Modell können mit einer Simulation unmittelbar beobachtet werden. Eine iterative beschleunigte Entwicklung wird dadurch gefördert. Bei der zweiten Variante muss zunächst Code erzeugt werden, der dann in einem zweiten Schritt kompiliert und ausgeführt wird, bevor Modifizierungen am Modell sichtbar werden. Die Performance wird jedoch bei dieser Variante deutlich erhöht.

Zur Ausführung von Aktivitätsdiagrammen sowie zur Codegenerierung ist ein entsprechendes Framework erforderlich, das bestimmte Anforderungen erfüllen muss. Diese Anforderungen umfassen:

- eine Simulationseinheit (*Execution Engine*), die Modellelemente interpretieren und ausführen kann.
- eine Visualisierung der einzelnen Schritte einer Simulation
- eine Codegenerierungseinheit, die durch Transformation der Modellelemente Programmcode erzeugen kann
- eine Action Language, die von der Simulationseinheit interpretiert und von der Codegenerierungseinheit übersetzt werden kann. Mit dieser Action Language sollen Attributwerte von Objekten abgefragt und manipuliert werden können.

Das in dieser Arbeit verwendete UML Modellierungswerkzeug Enterprise Architect verfügt über keine Fähigkeiten zur Ausführung von Modellen und kann nur begrenzt Code aus Klassendiagrammen generieren. Da eine Action Language nicht Bestandteil des Werkzeuges ist, wird zunächst in Abschnitt 4.2 eine eigene Action Language definiert, die von der Simulations- und Codegenerierungseinheit verwendet werden. Alle Erweiterungen bzgl. der Simulation und der Codegenerierung sind in C# mit Visual Studio 2008 entwickelt worden, die als Add-Ins in Enterprise Architect integriert werden.

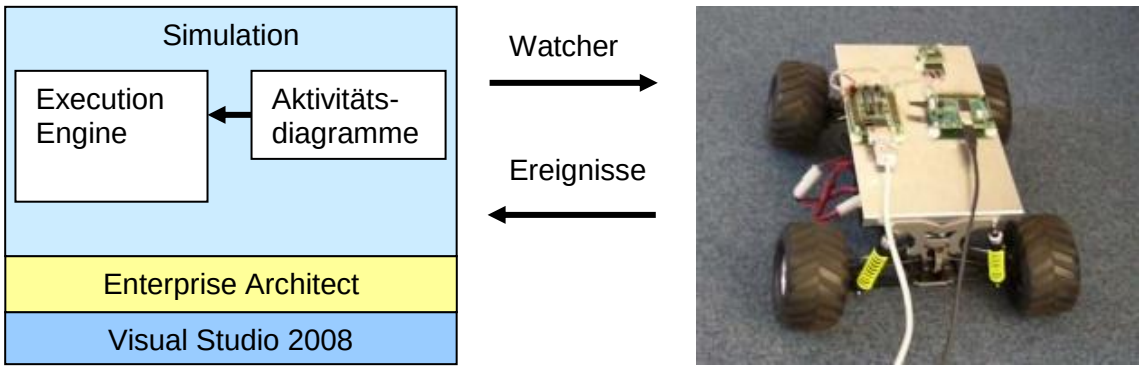


Abbildung 29: Aufbau der Simulation und Kommunikation mit dem Fahrzeug

Um ein Fahrzeug mit Hilfe von Aktivitätsdiagrammen steuern zu können, ist eine Kommunikation zwischen den Fahrzeugkomponenten (siehe Abschnitt 4.3) und Enterprise Architect erforderlich. Diese Kommunikation findet anhand von *Ereignissen* und *Watchern* statt (siehe Abbildung 29). Ein *Watcher* ist für die Kommunikation von Enterprise Architect zum Fahrzeug zuständig. Dabei untersucht der *Watcher* in periodischen Abständen die Attributwerte von bestimmten Objekten auf Veränderungen. Tritt eine Änderung eines Attributs ein, werden Signale an das Fahrzeug gesendet. Die andere Richtung der Kommunikation erfolgt über *Ereignisse*. Bei jeder Änderung von empfangenen Daten einer Fahrzeugkomponente wird ein Ereignis ausgelöst, das die Attributwerte des zur Fahrzeugkomponente zugehörigen Objektes updatet.

Bei der Codegenerierung wird der erzeugte C# Code in eine spezifische Umgebung integriert, das alle zur Kommunikation mit dem Fahrzeug notwendigen Komponenten enthält. Über Ereignisse werden weiterhin Daten empfangen, die jedoch Variablenwerte in Klassen ändern. Ein Watcher wird nicht mehr benötigt, da die Sendung von Steuerungsbefehlen von einer Funktion übernommen wird.

4.1.1 Anforderungen an die Modellierung mit Aktivitätsdiagrammen

Das mit einem Aktivitätsdiagramm erstellte Modell bildet die Grundlage für die Simulation und die Codegenerierung. Das erzeugte Modell wird bei der Simulationsausführung interpretiert und bei der Codegenerierung transformiert. Ein modelliertes Aktivi-

tätsdiagramm [siehe Abschnitt 2.1.5] sollte generell folgende Punkte erfüllen, damit eine Simulation sowie eine Codegenerierung erfolgreich durchgeführt werden können:

- Zur Modellierung sind nur die Elemente erlaubt, die von der *Execution Engine* (siehe Abschnitt 4.4.1) und der Codegenerierungseinheit (siehe Abschnitt 4.5.2) unterstützt werden.
- Es sollte darauf geachtet werden, dass ein Diagramm deadlockfrei modelliert wird. Wenn z.B. nicht alle Möglichkeiten bei einem Entscheidungsknoten berücksichtigt worden sind, kann die Ausführung eines Aktivitätsdiagramms endlos ablaufen.
- Alle Modellelemente im Aktivitätsdiagramm sollten erreichbar sein, andernfalls sind diese aus Performancegründen überflüssig und können weggelassen werden.
- Jedem Attribut eines im Modell verwendeten Objektes sollte sein korrekter Datentyp zugewiesen werden.
- Eine korrekte case-sensitive Schreibweise bei Anweisungen und Abfragen von Attributwerten konform zur Action Language ist zu beachten.
- Bei der Simulation einer Fahrzeugsteuerung ist für jede Fahrzeugkomponente, die im Modell benutzt werden soll, ein Objekt mit bestimmten Vorgaben zu erstellen (siehe Abschnitt 4.3).

Im Prinzip ist es auch möglich, Modelle auszuführen, die unvollständig im dem Sinne sind, dass sie nicht alle Details für eine komplette Ausführung beinhalten. Die Ausführung eines generierten Codes aus diesen Modellen würde jedoch zu Fehlermeldungen führen. Bei einem Entscheidungsknoten könnte es zum Beispiel vorkommen, dass nicht alle abgehenden Kontrollflusskanten mit Bedingungen versehen oder nicht alle vorkommenden Fälle abgedeckt werden. Im Falle, dass es keine Bedingung einer Kontrollflusskante gibt, die als wahr ausgewertet werden kann, würde die *Execution Engine* (siehe Abschnitt 4.4.1) an diesem Entscheidungsknoten stehen bleiben. Der Benutzer kann dann noch während der Ausführung Bedingungen zu Kontrollflusskanten hinzufügen, von denen eine als wahr ausgewertet werden kann. Danach würde die *Execution Engine* mit der Ausführung fortfahren. Diese Art der Benutzung und des Austestens von unvollständigen Modellen führt zu einer iterativen Vorgehensweise und kann den Entwicklungsprozess eines Modells beschleunigen.

4.2 Action Language

Da Enterprise Architect in der derzeitigen Version 7.1 über keine executable UML Fähigkeiten verfügt und nur begrenzt Codegenerierung in Form von Erstellung von Klassenrümpfen aus Klassendiagrammen unterstützt, wurde eine eigene Action Language entwickelt, um Simulationen anhand von Aktivitätsdiagrammen ausführen zu können. Außerdem soll mithilfe dieser Action Language Programmcode erzeugt werden können. Um diese Ziele zu erreichen, wurden folgende Anforderungen an die Action Language gesetzt:

- Zugriff auf die Attribute von Objekten
- Attributwerte von Objekten auslesen
- Attributwerte von Objekten schreiben
- Operatoren für Zuweisung, Inkrementierung und Dekrementierung
- Berechnungen von Ausdrücken durchführen
- Boole' sche Operatoren, um Ausdrücke zu verknüpfen und Bedingungen auszuwerten
- Unterstützung und Unterscheidung von unterschiedlichen Datentypen (insbesondere *double*, *boolean* und *int*)

Umsetzung der Anforderungen

Mit Hilfe der Action Language können die Attributwerte eines Objektes mit Ausdrücken der Form *Objekt.Attribut* geschrieben und gelesen werden. *Objekt* bezeichnet dabei den Namen des Objektes und *Attribut den* Namen des Attributes. Um den Wert eines Attributes zu ändern, muss ein Ausdruck mit obigem Format auf der linken Seite einer Zuweisung oder eines Inkrement- oder Dekrementoperators stehen. Zuweisungen können nur in Aktionen benutzt werden. Um einer Aktion Ausdrücke konform zur Syntax der Action Language zuzuweisen, wird zunächst in Enterprise Architect durch einen Doppelklick auf das Modellelement *Aktion* ein Fenster mit mehreren Unterpunkten geöffnet. Wie in Abbildung 30 zu sehen ist, können dann unter dem Menüpunkt *General* Anweisungen dem *Notes*-Feld hinzugefügt werden. Folgende Operatoren können derzeit bei Aktionen verwendet werden:

- =, +=, -=
- +, -, *, /
- (…)

Zusätzlich zu Aktionen können Kontrollflusskanten, die von einem Entscheidungsknoten ausgehen, unter dem Menüpunkt *Constraints* im Feld *Guard* konform zur Action Language bearbeitet werden (siehe Abbildung 31). Ein *Guard* entspricht einer normalen Bedingung (*if-else*) oder einer Schleifenbedingung. Bei Kontrollflusskanten sind nur Abfragen und keine Zuweisungen erlaubt. Weiter muss ein ausgewerteter *Guard* einen Boole'schen Wert als Rückgabe liefern, weshalb auf eine Einschränkung auf Boole'sche Operatoren bei Vergleichen zu achten ist. Folgende Operatoren können bei *Guards* benutzt werden:

- <, >, <=, >=, ==, !=
- +, -, *, /
- and, or
- (…)

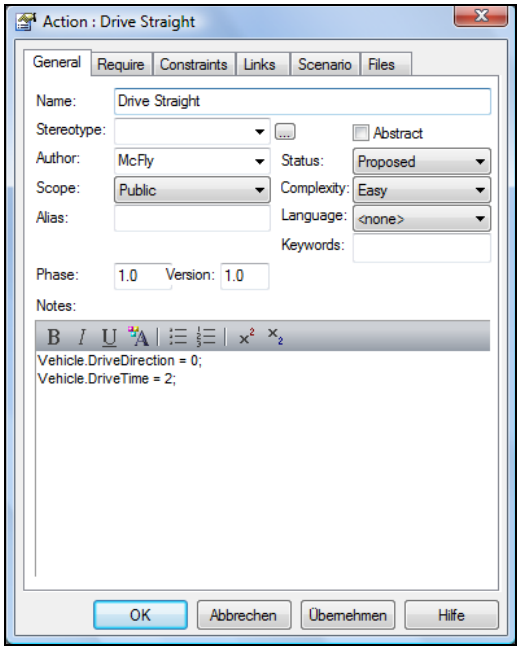


Abbildung 30: Editieren einer Aktion

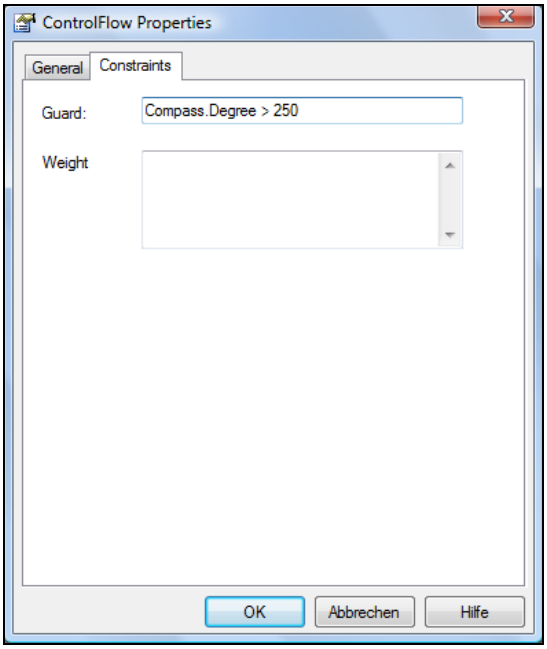


Abbildung 31: Editieren einer Kontrollflusskante

Es ist auch möglich, Modellelemente zur Laufzeit einer Simulation zu editieren. Wenn z.B. bei einem Entscheidungsknoten ein unvorhergesehener Deadlock auftritt, kann

eine Bedingung direkt geändert werden und somit der Deadlock noch während der Simulation aufgelöst werden. Bei Anweisungen bzw. Bedingungen mit ungültiger Syntax oder inkorrekten Verweisen wird eine Fehlermeldung ausgegeben und die Simulation abgebrochen.

4.3 Fahrzeug-Komponenten

In Abbildung 32 ist ein Fahrzeug zu sehen, das von der Firma SparxSystems für Simulationen bereitgestellt wurde. Das Fahrzeug ist ca. 50 cm lang, 30 cm breit und 15 cm hoch. Es entspricht im Prinzip einem modifizierten ferngesteuerten Fahrzeug, bei dem verbesserte Federungen eingebaut worden sind. Diese Federungen werden in Zukunft das Gewicht eines integrierten PCs tragen. Das Gehäuse des Fahrzeuges wurde abmontiert und durch eine Metallplattform ersetzt. Auf dieser Plattform sind ein Interface-Adapter, ein Kompassmodul, zwei Sensoren und eine Platine zur Steuerung der Servomotoren für die Lenkung und den Antrieb angebracht. Unter der Metallplattform befindet sich ein herausnehmbarer Akku, der als Stromversorgung der Servomotoren eingesetzt wird. Die Höchstgeschwindigkeit des Fahrzeuges wurde aus Sicherheitsgründen auf 20 km/h gedrosselt. Zum Zeitpunkt der durchgeführten Simulationen enthielt das Fahrzeug noch keinen integrierten PC. Die Steuerung des Fahrzeuges erfolgte deshalb über ein externes Notebook, das über USB- und COM-Kabel mit dem Fahrzeug verbunden wurde.

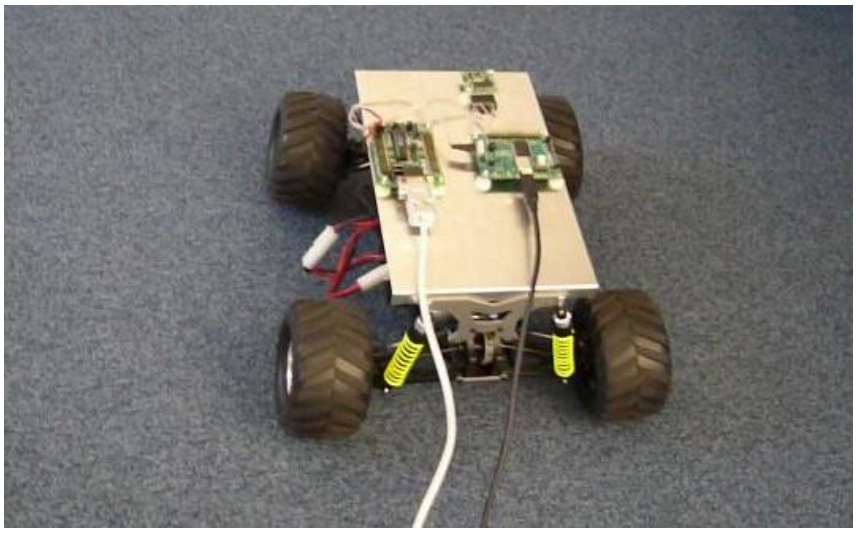
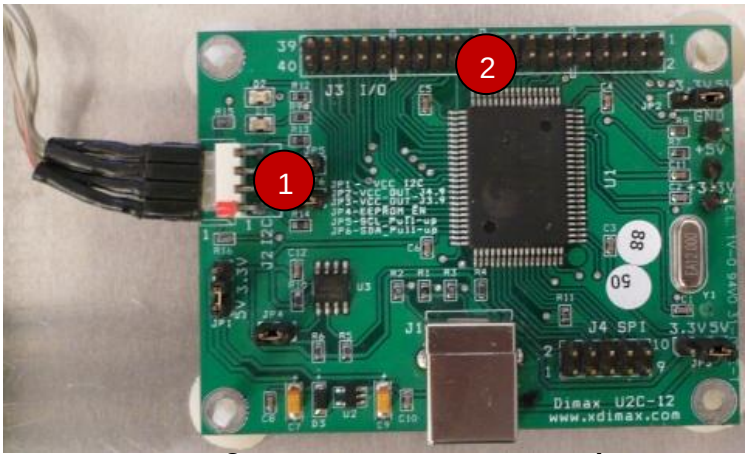


Abbildung 32: Fahrzeug mit Komponenten

Im Folgenden wird zunächst auf den Interface-Adapter eingegangen. Danach werden das Kompassmodul und die Sensoren, die mit dem Adapter verbunden sind, näher besprochen. Anschließend wird die Steuerung der Servomotoren behandelt. Die Modellierung der Sensoren, des Kompassmoduls und der Steuerung der Servomotoren erfolgt mit dem UML Modellierungswerkzeug Enterprise Architect.

4.3.1 Interface Adapter

Der U2C-12 USB-I2C/SPI/GPIO Interface Adapter in Abbildung 33 dient als Schnittstelle für das Kompassmodul (siehe Abschnitt 4.3.2) und die Sensoren (siehe Abschnitt 4.3.3). Das Kompassmodul wird an das I2C-Interface (Label 1 in Abbildung 33) und die Sensoren an die GPIO-Schnittstelle angeschlossen (Label 2 in Abbildung 33). Der Interface Adapter wird über USB mit dem Steuerungsrechner verbunden. Die in dieser Arbeit verwendeten Funktionen, um auf die Daten des I2C-Interfaces und der GPIO-Schnittstelle zugreifen zu können, sind beim Kompassmodul und bei den Sensoren aufgelistet und näher beschrieben.



4.3.2 Kompass

Das Kompassmodul Devantech CMPS03 (Abbildung 34) eignet sich für den Einsatz in mobilen Robotern und wurde deshalb in dieser Arbeit verwendet. Der Messwert wird aufgrund der horizontalen Komponente des vorhandenen Erdmagnetfeldes bestimmt. Durch eine Kalibrierungsfunktion kann das Modul an räumliche Besonderheiten und eventuelle statische Felder angepasst werden. Die Auflösung des Kompasses beträgt 0,1°, die Genauigkeit liegt bei ca. 3-4°. Für eine hohe Messgenauigkeit ist eine exakte horizontale Ausrichtung wichtig. Das Modul wird mit einer Versorgungsspannung von 5V (Pin1 - unterstes graues Kabel + Pin9 - oberstes graues Kabel) verbunden. Der Messwert wird über die I2C-Schnittstelle gelesen und steht an Pin2 (SCL) / Pin3 (SDA) zur Verfügung. Die Ausgabe des Wertes erfolgt als 16 Bit Zahl mit einem Wertebereich von 0 – 3599 (3599 entspricht 359,9°).

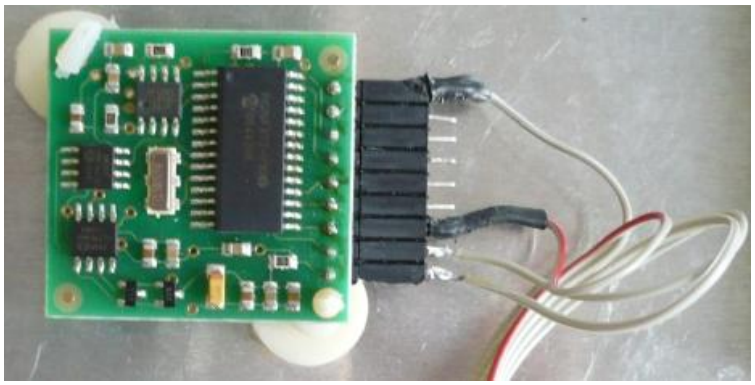


Abbildung 34: Kompassmodul Devantech CMPS03

Die Kommunikation mit dem Kompassmodul findet in UML über ein Objekt mit dem Namen *Compass* statt. Wie in Abbildung 35 zu sehen ist, enthält *Compass* nur das Attribut *Degree*:

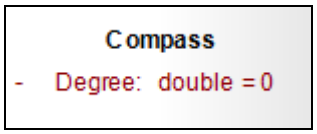


Abbildung 35: Kompass

Attribut: Degree

Typ: Double

Beschreibung:

Mit diesem Attribut wird die Gradzahl angezeigt, in der sich das Fahrzeug befindet. 0° entspricht Norden, 90° Osten, 180° Süden und 270° Westen. Auf *Degree* wird überwiegend lesend zugegriffen, um eine Steuerung des Fahrzeuges in Abhängigkeit der Himmelsrichtung zu ermöglichen. Während der Praxistests mit dem Kompassmodul wurde auf *Degree* nur bei Initialisierungsschritten schreibend zugegriffen.

Hauptfunktion zur Kommunikation mit dem Kompassmodul

In der Schnittstellenimplementierung *I2C* gibt es die Funktion *U2C_Read*, die für das Lesen der Kompassdaten über die I2C-Schnittstelle verantwortlich ist. Diese Funktion wird über einen Timer in periodischen Abständen abgerufen. Bei einem erfolgreichen Lesevorgang ist die Klasse *CompassPlugIn* für die Darstellung der gelesenen Daten in Enterprise Architect zuständig.

```
U2C_RESULT U2C_Read(int hDevice, ref U2C_TRANSACTION pTransaction);
```

Der Rückgabetyp *U2C_Result* ist als String definiert und gibt an, ob z.B. ein Lesevorgang erfolgreich war oder ob das Slave-Gerät geöffnet werden konnte. Die Funktion *U2C_Read* beinhaltet folgende Parameter:

- *hDevice*: identifiziert eindeutig einen U2C-12 Interface Adapter. Theoretisch könnten in einem Fahrzeug beliebig viele dieser Adapter verwendet werden
- *pTransaction*: ist ein Zeiger auf die *U2C_TRANSACTION*-Struktur, die während eines Lesevorganges benutzt wird. *U2C_TRANSACTION* weist folgende Struktur auf:
  - *nSlaveDeviceAddress* enthält die I2C Slave-Geräteadresse, mit der der Kompass aktiviert werden kann.
  - *nMemoryAddressLength* entält die interne Adresslänge in Bytes
  - *MemoryAddress* enthält die interne I2C Slave-Geräteadresse
  - *nBufferLength* enthält die Anzahl an Bytes, die vom I2C Slave-Gerät gelesen werden sollen. Nach erfolgreichem Lesevorgang werden die Daten des Kompasses in *Buffer* der Struktur *U2C_TRANSACTION* geschrieben. Da der Kompass eine Genauigkeit von 0,1° aufweist, gibt es 3600 verschiedene Werte. Deshalb wird *nBufferLength* beim Kompass auf 2 Bytes festgelegt.

4.3.3 Sensoren

Um eine umfassendere Modellierung zu gewährleisten, sind am Fahrzeug zwei Abstandssensoren (Abbildung 36) montiert. Diese Sensoren sind hilfreich, wenn eine Modellierung in Abhängigkeit von Abständen stattfinden soll. Zum Beispiel kann ein Fahrzeug gestoppt werden, wenn es sich in der Nähe eines Hindernisses befindet. Am Fahrzeug ist der eine Sensor vorne und der andere an der Seite rechts installiert. In Zukunft soll das Fahrzeug um zwei weitere Sensoren, hinten und seitlich links, erweitert werden. Die Messung des Abstandes erfolgt über Infrarot. Da die Sensoren digital an die GPIO-Schnittstelle des Interface-Adapters (siehe Abschnitt 4.3.1) angeschlossen werden, liefern die Sensoren je nach Abstand die digitalen Signale 0 oder 1 zurück. Der Signalwert 1 bedeutet, dass der Abstand eines Objektes zum Sensor kleiner oder gleich 15 cm beträgt, wohingegen bei 0 die Abstandsmessung über 15 cm beträgt.

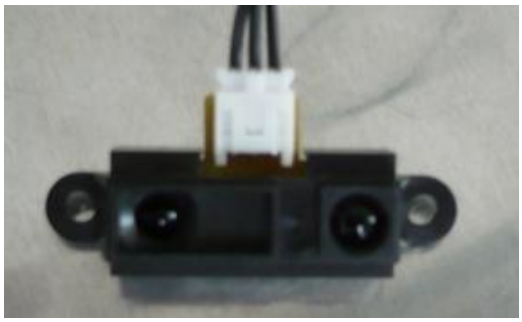


Abbildung 36: Sensor

Die Kommunikation mit den Sensoren wird wie bei den Servosteuerungen in UML über ein Objekt modelliert. Dieses Objekt mit Namen *Sensors* (Abbildung 37) beinhaltet vier Attribute, auf die im Folgenden näher eingegangen wird:

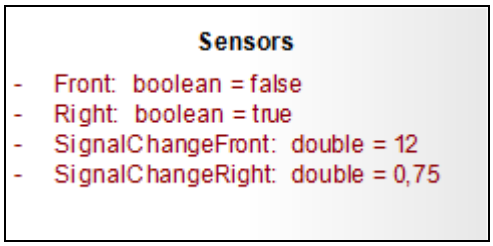


Abbildung 37: Sensoren

Attribute: Front und Right

Typ: Boolean

Beschreibung:

Mit diesen Attributen wird angezeigt, ob ein Sensor einen gewissen Abstand zu einem Objekt unterschreitet oder nicht. *Front* bezeichnet den Sensor vorne am Fahrzeug und *Right* den Sensor seitlich rechts. Der Attributwert *false* gibt an, daß sich kein Objekt vor dem Sensor mit einem Abstand kleiner gleich 15 cm befindet. Dementsprechend liefert ein Sensor den Attributwert *true* zurück, wenn ein Objekt in der Reichweite von 15 cm vorhanden ist.

Attribute: SignalChangeFront und SignalChangeRight

Typ: Double

Beschreibung:

Diese beiden Attribute geben die Dauer an, in der kein Signalwechsel mehr stattgefunden hat. Die Attributwerte werden dabei in 0,25 Sekundenschritten hochgezählt. Eine noch genauere Taktung ist aus Performancegründen nicht zu empfehlen, da das Objekt bei jeder Änderung grafisch angepasst wird. Dieser Vorgang dauert einige Millisekunden. *SignalChangeFront* entspricht der Zeit ab dem letzten Signalwechsel des Sensors vorne am Fahrzeug. Mit *SignalChangeRight* kann die Dauer ab dem letzten Signalwechsel rechts am Fahrzeug erhalten werden. *SignalChangeRight* ist insbesondere von Interesse, da z.B. der Abstand einer Lücke rechts vom Fahrzeug in Abhängigkeit der Geschwindigkeit des Fahrzeugs und der Zeit, ab dem das Signal des Sensors (*Right*) von *true* auf *false* wechselt, gemessen werden kann. Wie bereits erwähnt soll in Zukunft auch ein Sensor auf der linken Seite Abstände messen können.

Hauptfunktion zur Kommunikation mit den Sensoren

In der Schnittstellenimplementierung *U2C* gibt es die Funktion *U2C_SingleIoRead*, die für den Empfang der Sensorenwerte über die GPIO-Schnittstelle zuständig ist. Mittels eines Timers wird *U2C_SingleIoRead* in periodischen Abständen aufgerufen. Bei erfolgreichem Empfang der Sensorenwerte stellt die Klasse *SensorPlugIn* die gelesenen Daten in Enterprise Architect dar.

```
U2C_RESULT U2C_SingleIoRead(int hDevice, int IoNumber,
                            ref bool pValue);
```

Der Rückgabetyp *U2C_Result* ist wie beim Kompass als String definiert und gibt an, ob z.B. der Empfang der Sensorenwerte erfolgreich war. Die Funktion *U2C_SingleIoRead* beinhaltet folgende Parameter:

- *hDevice*: identifiziert eindeutig den U2C-12 Interface Adapter, an dem die Sensoren angeschlossen sind.
- *IoNumber* enthält die Nummer des GPIO-Pins, an dem ein Sensor angeschlossen ist. Der Sensor, der am Fahrzeug vorne befestigt ist, ist an Pin 0 angeschlossen, der seitliche rechte Sensor an Pin 1.
- *pValue* ist ein Zeiger auf einen Boole‘ schen Wert, der angibt ob sich ein Objekt im Abstand kleiner als 15 cm befindet oder nicht.

4.3.4 Servosteuerung

Die Steuerungsplatine SSC-32 wird vorwiegend zur Servosteuerung eingesetzt [www-04]. Sie wird über einen COM-Anschluss mit dem Steuerungsrechner verbunden. Insgesamt besitzt die Steuerungsplatine 32 Kanäle (Label 1 und Label 2 in Abbildung 38). Jedes Gerät wird über drei Anschlüsse mit einem Kanal verbunden (Label 1 in Abbildung 38):

- Massekabel (schwarz)
- Powerkabel (rot)
- Impulskabel (weiß)

Die Steuerungsplatine erlaubt bidirektionale Kommunikation mit Anfragebefehlen sowie synchronisierte Gruppenbewegungen.

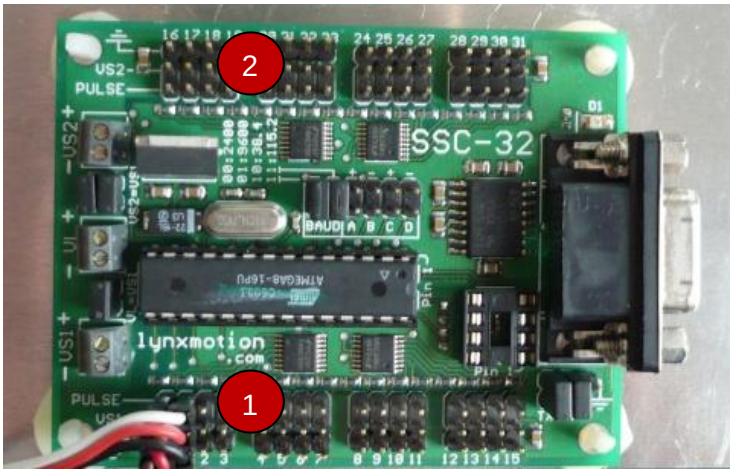


Abbildung 38: Steuerungsplatine SSC-32

4.3.4.1 Servomotoren

Die im Fahrzeug benutzten Servomotoren sind impulsproportionale Servomotoren, die in funkgesteuerten Autos, Booten und Flugzeugen eingesetzt werden. Sie erlauben z.B. eine präzise Lenkung und eine genaue Steuerung des Antriebes durch übertragene und empfangene Signale. Diese Signale bestehen aus positiven Impulsen mit einer Länge von 0,9 ms bis 2,1 ms, die 50 mal pro Sekunde wiederholt werden. Ein Servomotor positioniert seine Ausgangsachse im Verhältnis zur Breite des Impulses (Abbildung 39):

- 0,9 ms entsprechen -45°
- 1,5 ms entsprechen 0°
- 2,1 ms entsprechen +45°

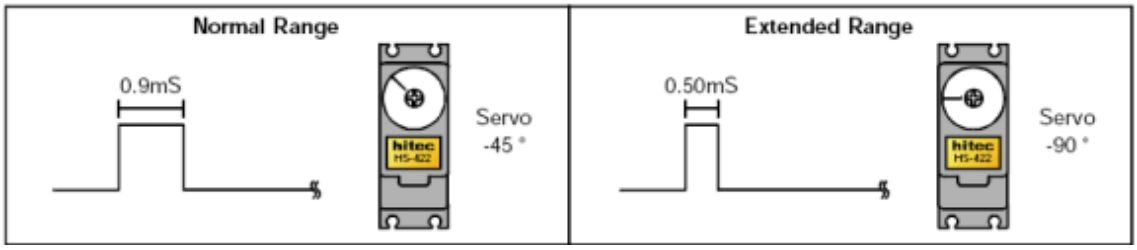


Abbildung 39: Servosteuerung

Der normale Steuerungsbereich eines Servomotors beträgt 90° [www-04] (linkes Bild in Abbildung 39). Die Steuerungsplatine SSC-32 unterstützt auch die Steuerung von Servomotoren mit einem erweiterten Bereich von bis zu 180° (rechtes Bild in Abbildung 39). In diesem Falle beträgt die Länge der Impulse zwischen 0,5 ms und 2,5 ms. Es ist jedoch darauf zu achten, dass ein Servomotor nur in seinem zulässigen Bereich gesteuert wird, ansonsten ist eine Schädigung des Servomotors nicht auszuschließen. Die Servomotoren, die im Fahrzeug eingesetzt werden, können nur in einem Bereich von bis zu 120° operieren. Aus Gründen der Sicherheit, Erweiterbarkeit und Austauschbarkeit wurde die Steuerung jedoch auf einen Bereich von 90° beschränkt. Für die Lenkung und die Antriebssteuerung des Fahrzeuges wird jeweils ein Servomotor verwendet. Sie sind auf der Steuerungsplatine SSC-32 (Abbildung 38) an Kanal 0 und Kanal 1 angeschlossen.

Die Servosteuerungen werden in UML über ein Objekt mit dem Namen *Vehicle* modelliert und indirekt angesprochen. Eine Kommunikation mit dem Fahrzeug erfolgt unter anderem über diese Instanz. In Abbildung 40 ist das Objekt *Vehicle* dargestellt. Es verfügt über drei Attribute, die im Folgenden näher erläutert werden:

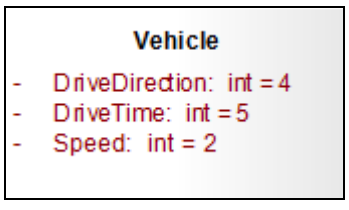


Abbildung 40: Vehicle

Attribut: DriveDirection:

Typ: Integer

Beschreibung:

Mit diesem Attribut kann die Fahrtrichtung eingegeben werden. Indirekt wird damit der Servomotor, der für die Lenkung zuständig ist, angesteuert. Die Steuerung erfolgt in Stufen, die dann in ein entsprechendes Lenksignal

übersetzt werden. Zulässig sind Attributwerte im Bereich der ganzen Zahlen von -4 bis +4. Mit negativen Zahlen erfolgt eine Steuerung nach links, wobei eine höhere negative Zahl einen stärkeren Linkseinschlag zur Folge hat. -4 entspricht z.B. einer Linkssteuerung mit einem Einschlagwinkel von -45°. Mit positiven Zahlen wird eine Steuerung nach rechts ausgelöst. Der Steuerungswinkel bei positiven Zahlen nach rechts ist äquivalent des Winkels bei negativen Zahlen. Eine gerade Fahrt mit Winkel 0° wird mit der Eingabe von 0 erreicht.

Attribut: Speed

Typ: Integer

Beschreibung:

Dieses Attribut bestimmt die Geschwindigkeit des Fahrzeuges. Dabei wird wie bei *DriveDirection* die Geschwindigkeit nicht direkt, sondern indirekt in Stufen gesteuert, die dann auf Antriebssignale transformiert werden. Die Steuerung des Antriebes liegt im Bereich der ganzen Zahlen zwischen -5 und +5. Eine Stufe entspricht dabei ca. 4 km/h bzw. ca. 1 m/s. Attributwerte mit negativem Vorzeichen kennzeichnen eine Rückwärtsfahrt. Die Eingabe von positive Attributwerten führt hingegen zu einer Vorwärtsfahrt. Bei +5 ist z.B. die max. Vorwärtsgeschwindigkeit von 20 km/h erreicht. Mit dem Attributwert 0 wird das Fahrzeug zum Stillstand gebracht.

Attribut: DriveTime

Typ: Integer

Beschreibung:

Mit *DriveTime* kann die Dauer einer Servosteuerung angegeben werden. Für dieses Attribut sind positive ganze Zahlen erlaubt. Eine Zahl entspricht der Anzahl an Sekunden, in der eine Befehlsfolge ausgeführt wird. Bei der Benutzung von *DriveTime* enthält eine Befehlsfolge mindestens eines der Attribute *Speed* und *DriveDirection* sowie *DriveTime* selbst. Das zu steuernde Fahrzeug fährt z.B. bei der unten aufgeführten Befehlsfolge für die Dauer von 5 Sekunden geradeaus, bevor weitere Befehle in einer darauffolgenden Aktion abgearbeitet werden.

```
Vehicle.DriveDirection = 0;
Vehicle.Speed = 3;
Vehicle.DriveTime = 5;
```

4.3.4.2 Befehle zur Servosteuerung

Die Steuerung der Servomotoren erfolgt mit den Befehlen der Steuerungsplatine SSC-32. Im Folgenden soll die Syntax der Befehle aufgezeigt und an Beispielen verdeutlicht werden.

Alle Befehle der Steuerungsplatine müssen mit einem *return*-Zeichen (ASCII 13) beendet werden. Mehrere Befehle können gleichzeitig in einer *Command Group* ausgeführt werden. Alle Befehle einer *Command Group* werden ausgeführt, sobald das *return*-Zeichen empfangen worden ist. So kann z.B. bei einem Fahrzeug gleichzeitig die Geschwindigkeit und die Fahrtrichtung geändert werden.

Syntax Servosteuerung und Gruppensteuerung:

```
# <ch> P <pw> S <spd>... # <ch> P <pw> S <spd> T <time><cr>
```

- <ch> = Kanalnummer, 0 - 31.

- <pw> =Impulslänge in Mikrosekunden, 900 - 2100 (erweiterter Bereich 500 - 2500)
- <spd> =Bewegungsgeschwindigkeit in uS pro Sekunde für einen Kanal (optional).
- <time> =Zeit in mS für die gesamte Bewegung, wirkt sich auf alle Kanäle aus, 65535 max (optional).
- <cr> = Return character, ASCII 13 (erforderlich, um eine Aktion zu initialisieren)
- <esc> =Abbruch des aktuellen Befehls, ASCII 27

Steuerung eines Servomotors:

```
#5 P 1600 S 750 <cr>
```

Bei diesem Beispiel wird der Servomotor auf Kanal 5 auf die Position 1600 bewegt. Von seiner derzeitigen Position wird der Servomotor mit einer Geschwindigkeit von 750uS pro Sekunde bewegt, bis er seine Zielposition erreicht hat.

Gruppensteuerung:

```
#5 P1600 #10 P750 T2500 <cr>
```

Hier wird ein Servomotor auf Kanal 5 auf die Position 1600 und ein anderer Servomotor auf Kanal 10 auf die Position 10 bewegt. Die gesamte Bewegung beider Servomotoren dauert 2,5 Sekunden. Beide starten und beenden ihre Ausführung zur gleichen Zeit. Auf diese Weise können mehrere Servomotoren synchron gesteuert werden. Dies ist dann sehr hilfreich, wenn für die Fahrzeugsteuerung ein Servomotor für das linke vordere Rad und ein Servomotor für das rechte vordere Rad benutzt werden.

4.4 Simulation

Eine Simulation erfolgt mit dem UML Modellierungswerkzeug Enterprise Architect durch die Ausführung eines Aktivitätsdiagrammes. Die Ausführung findet dabei mit der entwickelten *Execution Engine* (siehe nächster Abschnitt) statt, die die Modellelemente des Aktivitätsdiagrammes beim Durchlaufen interpretiert und ausführt. Diese *Execution Engine* wird als eine Erweiterung in Form eines Add-Ins in Enterprise Architect integriert. Damit ein Fahrzeug durch die Ausführung eines Aktivitätsdiagrammes gesteuert werden kann, sind weitere Add-Ins für die Kommunikation mit den Fahrzeugkomponenten notwendig. Die Codegenerierung, die in Abschnitt 4.5 behandelt wird, erfolgt ebenfalls über ein Add-In. Die Menüleiste von Enterprise Architect beinhaltet unter Add-Ins die in dieser Diplomarbeit entwickelte Erweiterung *UML Execution* (siehe Label 2 in Abbildung 41), unter der die folgenden Add-Ins ausgeführt werden können.

- *CarControl*: Dieses Add-In ist für die Steuerung der einzelnen Servomotoren für die Lenkung und den Antrieb zuständig. Bei Ausführung von *CarControl* wird eine Instanz der Klasse *Watcher* erzeugt, die in periodischen Abständen überprüft, ob sich die Attributwerte des Objekts *Vehicle* geändert haben. Wenn sich ein Attributwert geändert hat, werden spezifische Steuersignale (siehe Abschnitt 4.3.4.2) an die jeweiligen Servomotoren gesendet.
- *Generate Code*: Mit diesem Add-In kann Quellcode in C# aus einem zuvor selektierten Aktivitätsdiagramm generiert werden. Eine Simulation vor einer Codegenerierung ist zu empfehlen, um sicher zu stellen, dass der erstellte Programmcode bei dessen Ausführung sich so verhält wie erwartet.
- *Compass*: Durch dieses Add-In wird die Kommunikation mit dem Kompassmodul aktiviert. Es wird eine Instanz der Klasse *Compass* erzeugt, die in periodi-

schen Abständen die Daten des Kompasses empfängt. Wenn sich die Gradzahl bzw. Himmelsrichtung des Kompasses im Vergleich zum davor geprüften Wert geändert hat, wird die neue Gradzahl in das Attribut *Degree* des Objekts *Compass* geschrieben. Auf diese Weise wird in Enterprise Architect immer die aktuelle Richtung des Kompasses dargestellt.

- *Sensors*: Dieses Add-In initialisiert die Kommunikation mit den beiden Sensoren und visualisiert die empfangen Werte in Enterprise Architect. Durch Instantiierung der Klasse *Sensors* werden die Signale der Sensoren in festdefinierten Zeitabständen mithilfe eines Timers empfangen. Durch einen weiteren Timer wird wie in Abschnitt 4.3.3 beschrieben, die Zeit zwischen einem Signalwechsel eines Sensors in 0,25 Sekundenschritten hochgezählt. Alle Werte werden in die jeweiligen Attribute des Objektes *Sensors* geschrieben.
- *Execute Diagram*: Mit diesem Add-In wird eine Simulation eines ausführbaren Aktivitätsdiagrammes gestartet.

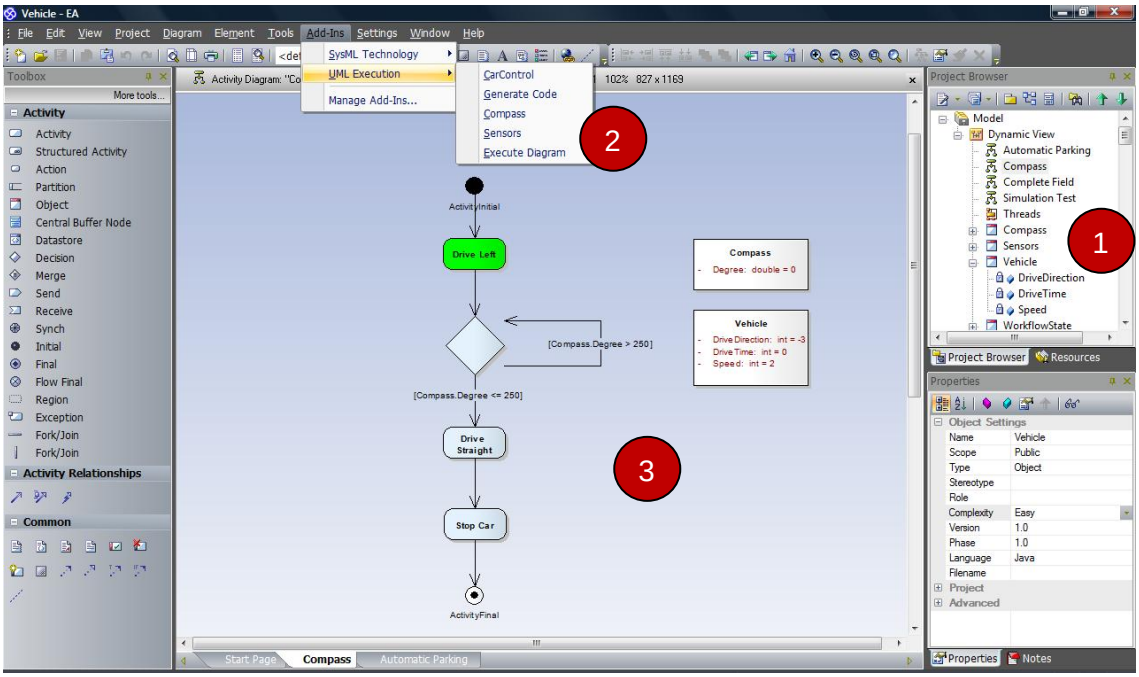


Abbildung 41: Ausführung der Add-Ins in Enterprise Architect

Um eine Simulation in Enterprise Architect auszuführen, müssen folgende Schritte im Vorfeld durchgeführt werden: Zunächst wird im *Project Browser* (Label 1 in Abbildung 41) das gewünschte Aktivitätsdiagramm, das ausgeführt werden soll, ausgewählt bzw. es wird ein neues Diagramm erstellt und modelliert. Danach kann in der Menüleiste unter Add-Ins die Erweiterung *UML Execution* ausgewählt und die für die Simulation notwendigen Add-Ins vor der eigentlichen Simulation des Aktivitätsdiagrammes ausgeführt werden. Wenn z.B. das Kompassmodul in eine Simulation miteinbezogen werden soll, muss zunächst das Add-In *Compass* ausgeführt werden, das die Kommunikation zwischen Enterprise Architect und dem Kompassmodul initialisiert. Allerdings wäre dann eine reale Fahrzeugsimulation noch nicht möglich. Zusätzlich muss das Add-In *CarControl* ausgeführt werden, um Steuerungsbefehle an das Fahrzeug senden zu können. Im Anschluss daran kann dann eine Simulation über das Add-In *Execute Diagram* gestartet werden.

Die Visualisierung der Simulation findet folgendermaßen statt: Wenn eine Simulation gestartet wird, werden die jeweils auszuführenden Modellelemente grafisch durch die Farbe Grün hervorgehoben (siehe Label 3 in Abbildung 41). Sobald ein Modellelement vollständig abgearbeitet ist, erhält es seine ursprüngliche Farbe Hellblau zurück und es wird zum nächsten Modellelement übergegangen, welches wiederum Grün eingefärbt

wird. Bei einer erfolgreichen Beendigung einer Simulation haben alle Modellelemente die Ausgangsfarbe Hellblau.

4.4.1 Execution Engine

Die *Execution Engine* umfasst die Interpretation und die Ausführung der für eine Simulation zulässigen Modellelemente. Die grafische Repräsentation sowie die Bedeutung der einzelnen Modellelemente sind in Abschnitt 2.1.5.1 näher erläutert. Im Folgenden wird im Detail darauf eingegangen, wie die jeweiligen Modellelemente während einer Simulation interpretiert und ausgeführt werden. Die vom Benutzer zugewiesenen Namen der Modellelemente spielen dabei keine Rolle, da in Enterprise Architect jedes Metamodellelement durch einen *MainType* und einen *SubType* eindeutig identifiziert wird. Eine Ausnahmeregelung gilt jedoch für die Aktionen *Signal senden* und *Signal empfangen* sowie für die Objekte, die in die Kommunikation zwischen den Fahrzeugkomponenten und Enterprise Architect involviert sind. Der Grund für diese Ausnahme wird in den Abschnitten 4.4.1.7 und 4.4.1.8 näher erklärt.

4.4.1.1 Startknoten

Wenn alle notwendigen Schritte für die Ausführung einer Simulation erfolgreich durchgeführt worden sind (siehe Abschnitt 4.4), wird beim Beginn einer Simulation nach dem Startknoten gesucht. Der Startknoten dient als Ausgangspunkt für den weiteren Simulationsverlauf. Von ihm aus wird über die ausgehende Kontrollflusskante das nächste Modellelement erhalten. Bei mehreren Startknoten wird der erste Gefundene ausgewählt. In der UML 2 sind mehrere Startknoten erlaubt, die dann jeweils in einem separaten Thread ablaufen. Sofern mehre Threads gleich zu Beginn einer Simulation gestartet werden sollen, kann dies anhand eines einzigen Startknotens und einem darauf folgenden Splitting-Knoten (siehe Abschnitt 4.4.1.5) bewerkstelligt werden.

4.4.1.2 Aktion

Mit Aktionen können Anweisungen ausgeführt werden, die z.B. die Attributwerte von Objekten verändern. Diese Anweisungen entsprechen der Syntax der Action Language (siehe Abschnitt 4.2) und können im Notes-Feld einer Aktion angegeben werden. Die einzelnen Anweisungen werden durch Strichpunkte getrennt und können somit sequentiell bearbeitet werden. Die Ausdrücke innerhalb der Anweisungen werden ersetzt und / oder berechnet. Dabei wird auf die zu ersetzenden Ausdrücke lesend zugegriffen. Bei folgendem Beispiel wird auf der rechten Seite der Ausdruck *Vehicle.DriveDirection* durch den derzeitig gültigen Wert des Attributes *DriveDirection* des Objektes *Vehicle* ersetzt. Dieser Wert wird danach mit 2 addiert und der linken Seite zugewiesen. Nach der Änderung des Attributwertes findet dann ein Update auf dem zugehörigen Objekt *Vehicle* statt. Nach Beendigung des Updates wird in Enterprise Architect der neue Attributwert angezeigt.

```
Vehicle.DriveDirection = Vehicle.DriveDirection + 2;
```

Bei der Änderung von bestimmten Attributwerten kann es erwünscht sein, dass automatisch weitere Programmabschnitte ausgeführt werden. Diese Änderungen werden mithilfe eines *Watchers* (siehe Abschnitt 4.1) überwacht. Die folgende Anweisung

```
Vehicle.Speed = 2;
```

veranlasst z.B. den zugehörigen *Watcher*, dem Fahrzeug den Steuerungsbefehl zu senden, sich mit einer Geschwindigkeit von 2 m/s fortzubewegen. Nachdem eine Aktion vollständig abgearbeitet worden ist, wird über dessen ausgehende Kante das nächste Modellelement gesucht.

Insgesamt beinhaltet die Ausführung einer Aktion folgende Aktivitäten:

- Schreiben und Lesen von Attributwerten
- Ersetzen von Ausdrücken
- Berechnung der rechten Seite einer Anweisung
- Update der geänderten Attributwerte in Enterprise Architect
- Visualisierung der Aktion

4.4.1.3 Entscheidungsknoten

Ein Entscheidungsknoten wird ausgeführt, indem die Bedingungen seiner abgehenden Kontrollflusskanten zu wahr oder falsch ausgewertet werden und anschließend zum nächsten Modellelement übergegangen wird. Die für einen Entscheidungsknoten vorgesehenen und zulässigen Operatoren sind in der Action Language in Abschnitt 4.2 zu finden. Bei einer Bedingung werden zunächst alle Ausdrücke der Form *Objekt.Attribut* durch die jeweiligen Attributwerte der Objekte ersetzt. Im Anschluss daran wird die gesamte Bedingung ausgewertet. Wenn eine Bedingung als wahr ausgewertet wird, wird zum nächsten Modellelement, das diese Bedingungskante als Eingangskante hat, übergegangen. Andernfalls wird die nächste Bedingung ausgewertet. Bei mehreren Kontrollflusskanten wird eine Überprüfung von weiteren Bedingungen abgebrochen, sobald eine Bedingung als wahr ausgewertet wird. Das hat den Vorteil, dass die Simulationsgeschwindigkeit erhöht wird. Auf der anderen Seite hat der Benutzer auf eine korrekte Modellierung zu achten, da keine Fehlermeldung ausgegeben wird, wenn mehrere Bedingungen positiv ausgewertet werden können. Bei der folgenden Bedingung wird zunächst der Ausdruck *Compass.Degree* durch den Wert des Attributes *Degree* des Objektes *Compass* ersetzt. Danach wird überprüft, ob dieser Wert größer als 250 Grad ist. Der resultierende Boole' sche Wert wird zurückgeliefert.

```
Compass.Degree > 250;
```

Bei einem Entscheidungsknoten werden insgesamt die folgenden Schritte ausgeführt:

- Lesen von Attributwerten
- Ersetzen von Ausdrücken
- Berechnung der linken und der rechten Seite von einem Boole'schen Operator
- Auswertung bzw. Vergleich der beiden Seiten anhand des Boole'schen Operators
- Visualisierung der Entscheidung

4.4.1.4 Aktion Verhaltensaufruf

Mit dieser Aktion wird ein Aktivitätsdiagramm aufgerufen und ausgeführt. In Enterprise Architect kann eine Aktion im Vorfeld als *Call Bahaviour Action* angegeben werden. Nachträglich ist dies auch möglich, indem eine Aktion selektiert wird und per Rechtsklick der Menüpunkt *Advanced -> Set Behavioral Classifier* ausgewählt wird. In dem sich öffnenden Fenster in Abbildung 42 kann die Aktivität, die von der Aktion aus aufgerufen werden soll, festgelegt werden. Diese Aktivität sollte im Vorfeld modelliert werden. Diese Aktivität wird während ihrer Ausführung fokussiert, sodass die einzelnen Simulationsschritte weiterhin beobachtet werden können. Zunächst wird wieder nach dem Startknoten gesucht, von dem aus durch das Diagramm traversiert wird. Nach erfolgreicher Ausführung wird wieder zum ursprünglichen Aktivitätsdiagramm zurückgesprungen und das nächste Modellelement gesucht. Im Beispiel von Abbildung 42 wird das Umdrehen des Fahrzeuges in eine neue Aktivität (*Turn left*) ausgelagert.

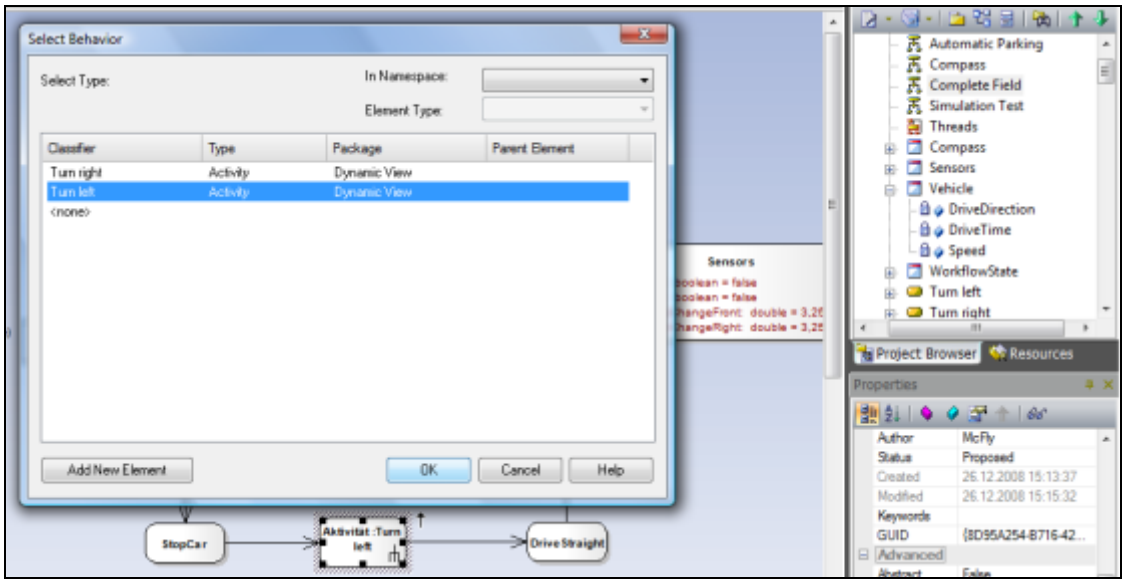


Abbildung 42: Aktion Verhaltensaufruf

4.4.1.5 Splittingknoten

Bei der Ausführung eines Splittingknotens verzweigt der Kontrollfluss in mehrere Abläufe, die jeweils in einem eigenen Thread ablaufen. Der Splittingknoten hat eine eingehende Kante, die von dem übergeordneten Thread stammt und mindestens eine Ausgangskante, über die ein neuer Thread gestartet wird. Die Anzahl der ausgehenden Kanten entspricht somit der Anzahl der zu erzeugenden Threads. Alle neuen Threads werden bei ihrer Erstellung einer Threadliste hinzugefügt. Jeder Thread wird nebenläufig bearbeitet, indem zu Beginn jeweils das nächste Modellelement jeder einzelnen abgehenden Kante gesucht und interpretiert wird. Danach wird der weitere Verlauf aller Ablaufzweige ausgeführt. Der Fortschritt eines Threads kann in der Simulation durch die Visualisierung bzw. grafische Hervorhebung beobachtet werden. Wie schnell ein Thread ausgeführt wird, hängt von der Menge der Anweisungen in den enthaltenen Aktionen sowie der generellen Ablaufmodellierung ab. Wenn ein Thread sein Ablaufende erreicht hat, wird er aus der Threadliste wieder entfernt.

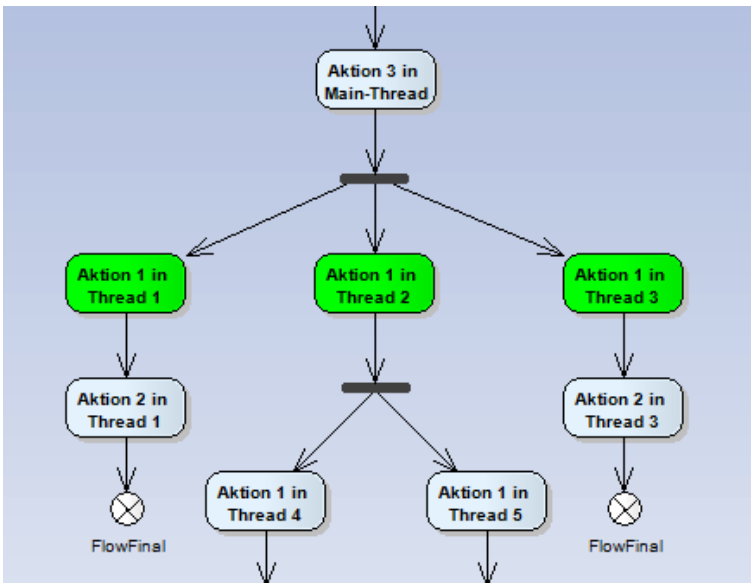


Abbildung 43: Simulation bei Splitting-Knoten

In Abbildung 43 ist ein Ausschnitt der Simulation eines Aktivitätsdiagrammes mit mehreren nebenläufigen Threads zu sehen. Der zweite Thread wird hierbei weiter aufgeteilt. Generell können Threads beliebig geschachtelt werden. In dem Beispiel aus Abbildung 43 werden nach dem ersten Splittingknoten drei Threads gestartet und der Threadliste 1 hinzugefügt. Der zweite Thread enthält nach der ersten Aktion einen weiteren Splittingknoten, der wiederum zwei neue Threads erzeugt. Diese beiden Threads werden einer neuen tiefer geschachtelten Threadliste 2 hinzugefügt. Wenn beide Threads ihr Ablaufende erreicht haben, werden sie wieder aus der Threadliste 2 entfernt. Danach wird der zweite Thread des ersten Splittingknotens weiter ausgeführt oder beendet, je nachdem, ob ein Synchronisationsknoten (siehe Abschnitt 4.4.1.6) im weiteren Verlauf dieses Ablaufzweiges enthalten ist oder nicht. Wenn der zweite Thread sein Ablaufende erreicht hat, wird er aus der Threadliste 1 gelöscht. Die Simulation gilt als beendet, wenn kein Thread mehr in der Threadliste 1 vorhanden ist und der Main-Thread ebenfalls vollständig abgearbeitet ist.

4.4.1.6 Synchronisationsknoten

Bei der Ausführung eines Synchronisationsknoten werden mehrere Abläufe, die aus der Aufspaltung eines Splitting-Knotens resultieren, wieder zusammengeführt. Dabei wird die weitere Ausführung des übergeordneten Threads verzögert. Mit dieser Modellierung können Threads auf andere Threads warten, bevor der übergeordnete Ablauf fortgesetzt wird. Damit können verschiedene Threads in Beziehung zueinander gesetzt und synchronisiert werden. Ein Splittingknoten enthält mindestens eine eingehende Kante und eine Ausgangskante, die zum übergeordneten Thread zurückführt. Die Anzahl der eingehenden Kanten ist kleiner gleich der Anzahl der untergeordneten Threads, da nicht alle Threads eines Splittingknotens synchronisiert werden müssen. Diese Threads wurden zuvor bei Erreichen eines Splittingknotens einer neuen Threadliste hinzugefügt. Sobald ein untergeordneter Thread auf einen Synchronisationsknoten trifft, wird dieser wieder aus der Threadliste entfernt. Erst wenn alle Threads, die zu einem Synchronisationsknoten führen, ihr Ablaufende erreicht haben und die untergeordnete Threadliste keinen Eintrag mehr enthält, wird über die ausgehende Kante des Synchronisationsknotens zum nächsten Modellelement übergegangen. Die Ausführung des übergeordneten Threads wird mit diesem Element fortgesetzt.

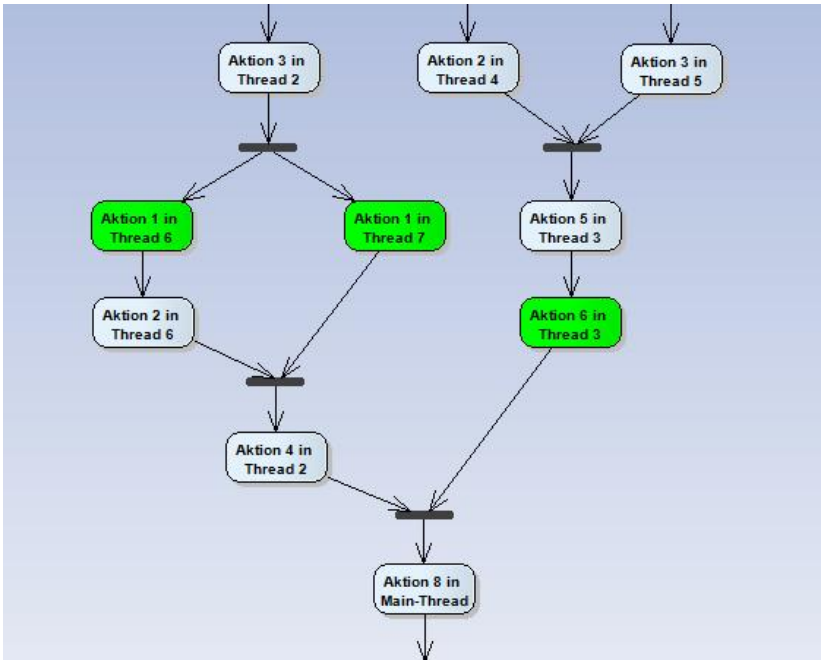


Abbildung 44: Simulation bei Synchronisationsknoten

In Abbildung 44 ist der Ausschnitt einer Simulation mit mehreren Synchronisationsknoten und einem Splittingknoten zu sehen. Auf der rechten Seite sind Thread 4 und Thread 5 aus der Threadliste 2 entfernt worden, da der erste rechte Synchronisationsknoten bereits von beiden Threads erreicht wurde. Danach wurde Thread 3 fortgesetzt, der am Ende seines Ablaufzweiges angekommen ist. Nach Beendigung der *Aktion 6 in Thread 3* wird Thread 3 aus der Threadliste 1 gelöscht und auf das Ablaufende von Thread 2 gewartet. Auf der linken Seite wurde Thread 2 durch einen Splittingknoten in Thread 6 und Thread 7 aufgespalten. Nach der Ausführung der *Aktion 1 in Thread 7* wird Thread 7 aus der Threadliste 3 entfernt und auf das Ende der Ausführung von *Aktion 2 in Thread6* gewartet. Danach kann dann Thread 2 mit *Aktion 4 in Thread 2* fortgesetzt werden. Nach Beendigung von Thread 2 beim untersten Synchronisationsknoten wird Thread 2 aus der Threadliste 1 gelöscht. Da die Threadliste 1 keinen Eintrag mehr enthält, wird zum Main-Thread zurückgesprungen und die Simulation mit *Aktion 8 in Main-Thread* weitergeführt.

4.4.1.7 Aktion Signal senden und Aktion Signal empfangen

Diese beiden Aktionen können benutzt werden, um bei einer Ausführung einen weiteren Thread zu erzeugen, der völlig unabhängig vom bisherigen Ablauf ausgeführt werden kann. Die Aktion *Signal senden* (siehe Abbildung 45) hat genau eine eingehende Kante und kann eine ausgehende Kante besitzen. Bei der Ausführung dieser Aktion wird die *Aktion Signal empfangen* (siehe Abbildung 46) gesucht, die den gleichen Namen wie die *Aktion Signal senden* hat. Die zu suchende Aktion kann im selben oder in einem anderen Aktivitätsdiagramm enthalten sein. Sobald die zugehörige Aktion gefunden wird, wird ein neuer Thread gestartet und mit der Ausführung dieses Ablaufzweiges begonnen (Aktion 1 in Abbildung 46). Die Ausführung des zur *Aktion Signal senden* zugehörenden Ablaufs wird nebenläufig zum neuen Thread durch den Übergang zum nächsten Modellelement (Aktion 3 in Abbildung 45) fortgeführt.

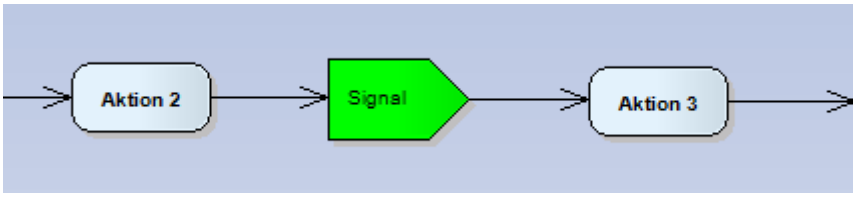


Abbildung 45: Simulation bei Aktion Signal senden

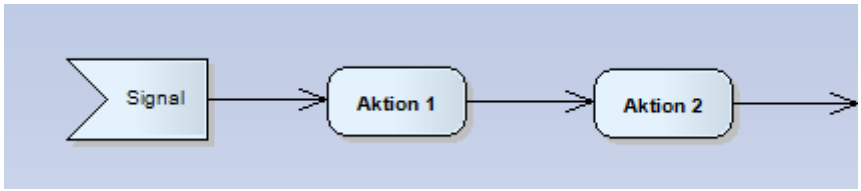


Abbildung 46: Simulation bei Aktion Signal empfangen

Der Unterschied zu einem Splittingknoten, bei dem ein Thread für den weiteren Ablauf und ein zweiter Thread für den neuen Ablaufzweig erzeugt werden, besteht darin, dass bei dieser Variante nur ein Thread erstellt wird und der Main-Thread weiter ausgeführt wird. Zusätzlich wird der neue Ablauf nicht visualisiert, wenn die zu suchende Aktion in einem anderen Diagramm enthalten ist, da es in Enterprise Architect nicht möglich ist, zwei verschiedene Aktivitätsdiagramme gleichzeitig visuell zu fokussieren. Diese Modellierungsart mit nur einem neuen Thread im Gegensatz zu zwei neu erzeugten Threads bei einem Splittingknoten wirkt sich positiv auf die Performance der Simulation aus. Zusätzlich kann dadurch ein Modell strukturiert werden, wodurch eine verbesserte Übersicht erreicht wird.

4.4.1.8 Objekte

Um bei einer Simulation mit den Fahrzeugkomponenten kommunizieren zu können, werden diese durch spezielle Objekte in Enterprise Architect repräsentiert. Die Kommunikation von einer Fahrzeugkomponente zu Enterprise Architect findet über Ereignisse statt, die bestimmte Attributwerte modifizieren. Die andere Richtung erfolgt über einen *Watcher* (siehe Abschnitt 4.1), der bei einer Änderung eines spezifischen Attributwertes, indirekt Signale an das Fahrzeug sendet.

Durch die Ausführung von Aktionen können die Attributwerte von Objekten zusätzlich verändert und ausgelesen werden. Bei einem Entscheidungsknoten wird auf die Attributwerte lesend zugegriffen. Es können drei fest vorgegebene Objekte modelliert werden, die eine Kommunikation mit den Fahrzeugkomponenten erlauben. Zusätzlich kann mit einem weiteren Objekt der momentane Ausführungszustand angezeigt werden. Dafür müssen bei einer Modellierung spezielle Namen für die Objekte sowie für die jeweiligen Attribute eingehalten werden. In Tabelle 1 sind die Namen dieser Objekte mit den jeweiligen Attributen und einer kurzen Beschreibung aufgelistet. Eine ausführliche Beschreibung bzgl. der Objekte zu den Fahrzeugkomponenten ist in Abschnitt 4.3 zu finden.

Objektname	**Attributname**	**Beschreibung**
Compass	Degree	Kompassrichtung
Vehicle	DriveDirection	Servosteuerung für den Antrieb und die Lenkung des Fahrzeuges
	Speed	
	DriveTime	
Sensors	Front	Abstandsmessung der Sensoren
	Right	
	SignalChangeFront	Zeit, die zwischen einem Signalwechsel vergeht
	SignalChangeRight	
WorkFlowState	Element	Ausführungszustand der Simulation
	Running	

Tabelle 1: Vordefinierte Objekte

4.4.1.9 Aktivitätsendknoten

Eine Simulation wird sofort beendet, wenn ein Aktivitätsendknoten erreicht wird. Andere Abläufe (Threads), die noch in der Ausführung sind, haben darauf keinen Einfluss und werden ebenfalls beendet.

4.4.1.10 Ablaufendknoten

Im Gegensatz zum Aktivitätsendknoten wird beim Ablaufendknoten nur der derzeitige Thread beendet. Die Simulation läuft solange weiter, bis alle Threads durch Ablaufendknoten oder durch einen einzigen Aktivitätsendknoten beendet werden.

4.4.2 Performance der Simulation

Um auf die Performance der Simulationsausführung näher eingehen zu können, ist es vorab notwendig, näher auf die Struktur und den Aufbau eines Enterprise Architect-Projekts einzugehen. In Abbildung 47 ist der Aufbau im Project Browser zu sehen. Das

Wurzelelement ist ein Modell, das mindestens eine *Sicht* aufweist. Zur besseren Übersicht kann eine *Sicht* in *Packages* gegliedert sein. *Sichten* und *Packages* können beliebig viele unterschiedliche UML Diagramme enthalten. Alle erzeugten Modellelemente sind unter allen Diagrammen einer *Sicht* bzw. eines *Packages* aufgelistet und können in mehreren Diagrammen verwendet werden.

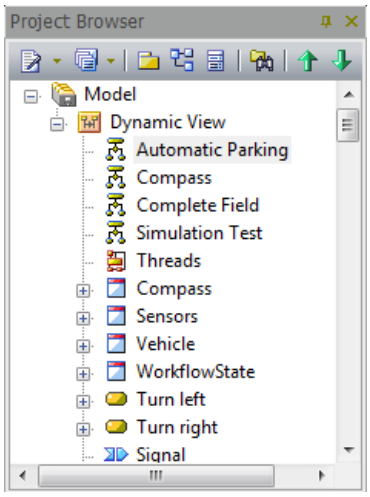


Abbildung 47: EA-Projekt

Intern wird ein Projekt in Enterprise Architect als Datenbank gespeichert, die mit Microsoft MS Access geöffnet werden kann. Diese Datenbank enthält unter anderem Tabellen für Diagramme, Modellelemente, Attribute und Kontrollflusskanten. In Abbildung 48 ist ein Ausschnitt der Datenbank zu sehen. Auf der rechten Seite ist die Tabelle *t_attribute* geöffnet, in der alle Attribute von Objekten des gesamten Projekts aufgelistet sind.

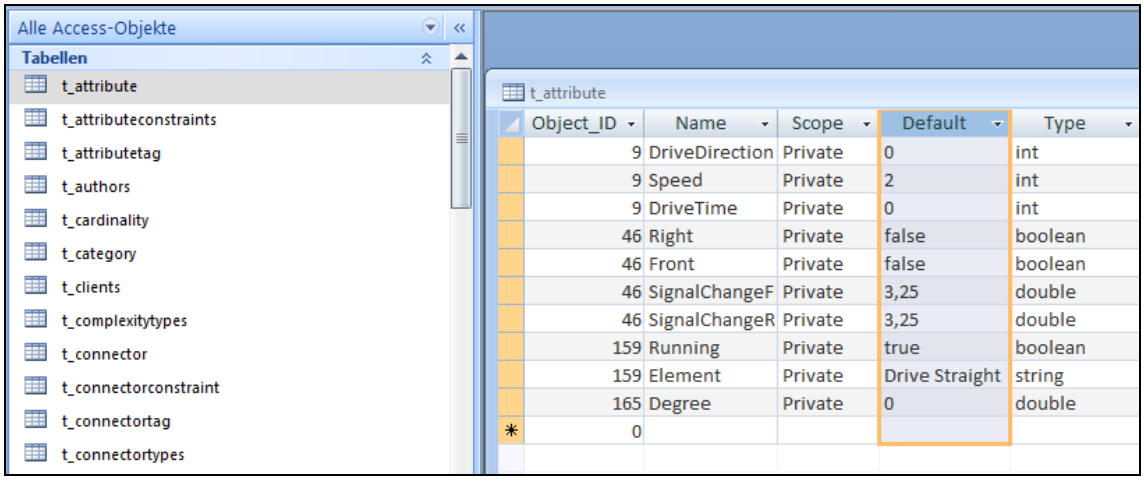


Abbildung 48: EA-Projekt als Datenbank

Im Folgenden soll nun näher darauf eingegangen werden, von welchen Faktoren die Ausführungsgeschwindigkeit einer Simulation abhängt:

- Plattformspezifische Details (CPU-Geschwindigkeit, RAM, Betriebssystem …)
- Timereinstellungen bei *Ereignissen* und bei einem *Watcher* (siehe Abschnitt 4.1): Diese Einstellungen hängen von allen anderen Faktoren ab und müssen individuell durch Ausprobieren angepasst werden. Bei einem zu niedrigen Timerwert für ein *Ereignis*, kann die Ausführungsgeschwindigkeit der gesamten Simulation erheblich verlangsamt werden, da die *Execution Engine* zu einem Großteil mit zu schnellen Änderungen des zum *Ereignis* zugehörenden Objek-

tes beschäftigt ist. Wenn der Timerwert zu hoch eingestellt ist, kann es passieren, dass eine Simulation anders als erwartet verläuft. Wenn z.B. das Fahrzeug nach rechts fährt und beim Überschreiten von 350 Grad (`Compass.Degree > 350;`) geradeaus fahren soll, ist es möglich, dass kein Update zwischen 350 und 360 Grad stattfindet (je nach Geschwindigkeit des Fahrzeuges) und sich somit das Fahrzeug im Kreis bewegt. Der *Watcher*, der das Objekt *Vehicle* mit Lenkung, Geschwindigkeit und Fahrzeit überwacht, sendet Steuersignale sowie die Fahrzeit bei der Änderung eines Attributwertes an die Servomotoren des Fahrzeuges. Mehrere Anweisungen bei der Ausführung einer Aktion, die die Attribute von *Vehicle* betreffen, werden nacheinander bearbeitet und bei einer Zuweisung die jeweiligen Attribute geändert. Erst nachdem diese Zuweisungen vollständig beendet sind, sollen die entsprechenden Befehle an das Fahrzeug übermittelt werden. Ein zu kleiner Wert für den Timer kann allerdings dazu führen, dass nicht alle Änderungen rechtzeitig ausgeführt werden und somit zum Teil veraltete Attributwerte in Befehle übersetzt werden. Wenn der Timerwert zu groß gewählt wird, werden die Signale mit einer größeren Verzögerung an das Fahrzeug übermittelt. Wenn z.B. das Fahrzeug aufgrund der Abstandsmessung des vorderen Sensors anhalten soll, wird der Steuerungsbefehl, der aus der Anweisung `Vehicle.Speed = 0;` resultiert, möglicherweise zu spät gesendet, womit das Fahrzeug mit dem Objekt zusammenstößt.

- Gesamtanzahl der Diagramme sowie der Modellelemente eines Projekts**:** Zu Beginn der Simulation wird das ausgewählte Aktivitätsdiagramm gesucht. Jedes Diagramm im EA-Projekt wird daraufhin überprüft. Die Suchzeit nimmt im Mittel proportional zur Anzahl der Diagramme eines Projekts zu. Bei der Ausführung der *Aktion Signal senden* (siehe Abschnitt 4.4.1.7) spielt die Anzahl der Diagramme auch eine Rolle, da nach der *Aktion Signal empfangen* in allen Diagrammen eines Projekts gesucht wird. Ebenso verhält es sich bei der Aktion Verhaltensaufruf, bei der das zur Aktion zugehörende Aktivitätsdiagramm ausgeführt wird. Die Performance einer Simulation hängt somit unter anderem von der Gesamtanzahl an Diagrammen ab. Zusätzlich wird die Performance von der Anzahl der enthaltenen Modellelemente eines Projekts beeinflusst. Um z.B. bei der Ausführung der Add-Ins *Compass*, *Sensors* und *CarControl* die zugehörenden Objekte mit Attributen zu erhalten, müssen alle Modellelemente auf eine Übereinstimmung überprüft werden.
- Anzahl der Add-Ins, die gleichzeitig ausgeführt werden**:** Alle Komponenten des Fahrzeuges werden durch ein eigenes Add-In repräsentiert und jeweils separat in einem eigenen Thread ausgeführt. Die Add-Ins *Compass* und *Sensors* kommunizieren über Ereignisse mit Enterprise Architect, die Änderungen auf die zu den Add-Ins zugehörenden Objekten in bestimmten Zeitabständen durchführen. Bei dem Add-In *CarControl* überwacht ein *Watcher* die Attributwerte des Objektes *Vehicle* und sendet Signale bei einer Änderung an das Fahrzeug. Die erwähnten Änderungen und die Überwachung beeinflussen die Performance der Simulation. Neben der Wahl des Zeitintervalls wie oben bereits erläutert, ist die Anzahl der ausgeführten Add-Ins entscheidend.
- Anzahl der Modellelemente des auszuführenden Aktivitätsdiagrammes: Neben der Gesamtanzahl an Modellelementen in einem Projekt hängt die Performance auch von der Menge der enthaltenen Elemente des auszuführenden Aktivitätsdiagrammes ab. Ausgehend vom Startknoten wird durch das Aktivitätsdiagramm anhand der ausgehenden Kanten der Modellelemente traversiert. Je mehr ausgehende Kanten ein Modellelement hat, desto mehr weitere Modellelemente müssen überprüft bzw. gefunden und bearbeitet werden (z.B. Entscheidungsknoten, Splittingknoten).
- Komplexität der Modellierung: Die Performance einer Simulationsausführung nimmt mit zunehmender Komplexität ab. Zum Beispiel wird durch die gleichzeitige Ausführung von mehreren Threads nach der Aufspaltung bei einem Split-

tingknoten (siehe Abschnitt 4.4.1.5) für jeden Ablaufzweig eine eigene Visualisierung gestartet. Jede zusätzliche Visualisierung ist zeitintensiv, da jedes durchlaufene Modellelement grafisch aktualisiert wird. Zusätzlich werden für die einzelnen Ablaufzweige Änderungen von Objekten bei vorkommenden Aktionen durchgeführt.

- Zugriffsart auf das EA-Projekt: Um auf ein Projekt programmiertechnisch zugreifen zu können, existieren zwei unterschiedliche Möglichkeiten:
  1. Über das COM Modell, mit dem von *repository* aus, das dem Wurzelelement eines EA-Projekts entspricht, auf Sichten, Pakete, Diagramme und Modellelemente zugegriffen werden kann. Mit *repository.getCurrentDiagram()* wird z.B. das ausgewählte Diagramm, das ausgeführt werden soll, erhalten.
  2. Direkt per SQL-Befehle auf die Datenbank. Mit folgendem SQL Statement kann z.B.der Attributwert eines Objektes ausgelesen werden:

```
"select default from t_attribute, t_object where
t_attribute.Name = '" + attributeName + "' and
t_attribute.Object_ID = t_object.Object_ID and
t_object.Name = '" + elementName + "'";
```

Der Vorteil der zweiten Methode besteht darin, dass die Performance deutlich erhöht werden kann, allerdings ist diese Zugriffsart weniger komfortabel. Außerdem besteht die Gefahr, dass bei Versionsänderungen von Enterprise Architect die unter der Objektschicht liegende Datenbankstruktur geändert werden könnte, was zur Folge hätte, dass die *Execution Engine* angepasst werden müsste. Für die Simulationsausführung werden beide Varianten in Anspruch genommen. Zeitintensive Operationen werden durch SQL Befehle auf die Datenbank optimiert. Dadurch wird eine Performanceverbesserung von bis zu Faktor 10 bei zeitkritischen Operationen erreicht.

4.5 Code-Generierung

Die Codegenerierung aus einem Aktivitätsdiagramm erfolgt durch die Transformation der enthaltenen Modellelemente. Ähnlich der Simulationsausführung (siehe Abschnitt 4.4) wird auch bei der Codegenerierung das Aktivitätsdiagramm durchlaufen. Dabei findet allerdings keine Visualisierung der Modellelemente statt. In Enterprise Architect kann ein selektiertes Aktivitätsdiagramm durch die Ausführung des Add-Ins *Generate Code* (siehe Label 2 in Abbildung 41) in C# Code transformiert werden. Ein konkreter Fahrablauf, der mit einem Aktivitätsdiagramm modelliert ist, kann auf diese Weise in Code transformiert werden. Der erzeugt Code wird in einer Datei abgespeichert, die in ein Visual Studio 2008 Projekt integriert wird. Dieses Projekt enthält alle zur Kommunikation mit dem Fahrzeug (siehe Abschnitt 4.3) notwendigen Komponenten. Der generierte Code, der den Fahrablauf des Fahrzeuges darstellt, wird zusammen mit dem Code für die Komponenten kompiliert und kann anschließend ausgeführt werden. Die nach der Ausführung erzeugte grafische Oberfläche des Visual Studio 2008 Projekts ist in Abbildung 49 zu sehen. Durch Drücken des Buttons *Start Execution* wird mit dem generierten Fahrablauf des Fahrzeuges begonnen. Oben links werden die Geschwindigkeit und oben rechts die Fahrtrichtung in Grad angezeigt. Unten rechts wird angezeigt, ob der Abstand des vorderen und rechten Sensors größer gleich oder kleiner als 15 cm beträgt. Es wäre auch möglich, ohne grafische Anzeige die Fahrzeugsteuerung auszuführen. Dies würde zu noch besseren Ergebnissen hinsichtlich der Performance führen. Allerdings würden dann keine Rückmeldungen des Fahrzeuges mehr sichtbar sein.

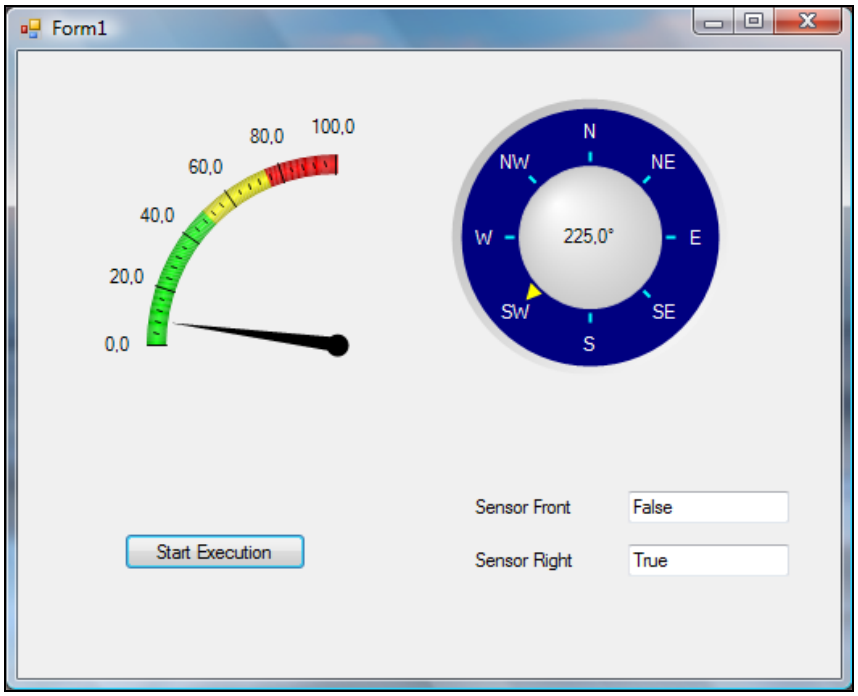


Abbildung 49: Grafische Benutzeroberfläche des Visual Studio Projekts

Die Modellierung einer konkreten Fahrzeugsteuerung kann in einem ersten Schritt mit der Simulationsumgebung in Enterprise Architect getestet werden. Wenn alle Tests erfolgreich ausgeführt worden sind, kann in einem zweiten Schritt Code generiert und ausgeführt werden. Die Performance der Fahrzeugsteuerung nach der Codegenerierung wird deutlich erhöht, erfordert jedoch einen höheren Voraufwand. Im Folgenden wird auf die Codegenerierung durch Transformation der Modellelemente, die in einem Aktivitätsdiagramm enthalten sind, näher eingegangen. Zunächst wird jedoch das .Net Code Document Object Model (CodeDom) beschrieben, mit dessen Hilfe Code erzeugt werden kann.

4.5.1 CodeDom

Das .Net Code Document Object Model (*CodeDom*) bietet eine komfortable Möglichkeit, Programmcode zu generieren. Mit *CodeDom* kann Quellcode in mehreren Programmiersprachen zur Laufzeit erstellt werden [RJSM04]. Es stellt Typen vieler und häufig verwendeter Quellcodeelemente bereit. Für nicht unterstützte Typen gibt es *CodeSnippets*, die an beliebigen Stellen im Code eingefügt werden können. Alle *CodeDom*-Elemente werden miteinander verknüpft und stellen eine Datenstruktur dar, die der des Quellcodes entspricht. *CodeDom* eignet sich besonders dann, wenn ein Programm in mehreren Sprachen geschrieben werden soll, die Zielsprache selbst jedoch noch nicht feststeht. In Tabelle 2 sind die wichtigsten *CodeDom*-Elemente aufgelistet, die bei der Codegenerierung eingesetzt werden finden:

Typ	**CodeDom Klassen**	**Beispiel nach Codegenerierung**
Using	CodeSnippetCompileUnit	using System.IO;
Namespace	CodeNamespace	namespace executableUML {….}
Klasse	CodeTypeDeclaration	class classname {…}
Methode	CodeMemberMethod	public virtual void executeProgram() { … }
Konstruktor	CodeConstructor	public classname {….}

Beliebiger Code	CodeSnippetStatement	Thread.Sleep(10);
Bedingung	CodeConditionStatement	if (Bedingung) {…} else {….}
Schleife	CodeIterationStatement	for (;Schleifenbedingung;) {….}
Code-Block	CodeStatementCollection	Statement1; Statement2; Statement3; …
Objekterzeugung	CodeVariableDeclaration-Statement	Vehicle v1 = new Vehicle();

Tabelle 2: CodeDom-Elemente

4.5.2 Transformation der Modellelemente

Der Programmcode, der durch die Transformation der Modellelemente eines Aktivitätsdiagrammes erzeugt wird, wird in einer einzigen Klasse gebündelt. Zu Beginn der Codegenerierung werden einige Initialisierungsschritte mit den *CodeDom*-Elementen des vorherigen Abschnitts durchgeführt. Für die zu erzeugende Klasse werden zunächst alle benötigten Pakete erstellt. Danach wird der Namespace und der Klassenrumpf festgelegt. Anschließend werden der Konstruktor der Klasse sowie die Einstiegsmethode in die Klasse eingefügt. Nachdem das Grundgerüst der Klasse vorhanden ist, können die Modellelemente, die in Abschnitt 2.1.5.1 näher beschrieben sind, in Code transformiert und den entsprechenden *CodeDom*-Elementen hinzugefügt werden.

4.5.2.1 Startknoten

Da auch bei der Codegenerierung die Modellelemente des Aktivitätsdiagrammes durchlaufen werden, wird zunächst wie bei der Simulation nach dem Startknoten gesucht. Der Startknoten wird nicht in Code transformiert, sondern als Ausgangspunkt für den Durchlauf des Aktivitätsdiagrammes benutzt. Von ihm ausgehend wird das nächste Modellelement über dessen ausgehende Kante erhalten.

4.5.2.2 Aktion

In Aktionen werden Anweisungen konform zur Action Language angegeben, mit denen indirekt mit dem Fahrzeug kommuniziert werden kann. Im Gegensatz zur Simulation, bei der Anweisungen Attributwerte verändern, sollen nach der Codegenerierung Variablenwerte modifiziert werden. Die Anweisungen werden in das Notes-Feld einer Aktion geschrieben. Durch Strichpunkte getrennt kann jede Anweisung einzeln bearbeitet und in Code transformiert werden. Die in einer Anweisung enthaltenen Attribute werden in Variablen übersetzt. Damit jedes Attribut eindeutig ist, besteht jede Variable aus dem Objektnamen und dem Attributnamen. Dazu werden alle vorkommenden Ausdrücke der Form *Objekt.Attribut* einer Anweisung in *ObjektAttribut* geändert. Der Ausdruck *Vehicle.Speed*, der das Attribut *Speed* des Objekts *Vehicle* darstellt, wird z.B. in eine Variable mit dem Namen *VehicleSpeed* transformiert. Alle Variablen werden als *private* deklariert. Der Typ der Variablen wird aus dem angegeben Typ des Attributes übernommen bzw. konvertiert. Danach wird die Variable mit vordefinierten Werten für einen Typ initialisiert und der gesamte Ausdruck wie im Beispiel unten direkt unterhalb des Klassenanfangs eingefügt. Um eine mehrfache Deklarierung von Variablen, die zu einer Fehlermeldung bei der Kompilierung des generierten Codes führt, auszuschließen, findet bei der Erzeugung einer Variablen eine Überprüfung anhand einer Liste statt, in die alle neu erstellten Variablen eingefügt werden.

```
private int VehicleSpeed = 0;
```

Im Anschluss daran wird die konvertierte Anweisung dem entsprechenden *CodeDom*-Element hinzugefügt. Bei der Änderung von bestimmten Attributwerten kann es erforderlich sein, dass zusätzliche Funktionen ausgeführt werden. Dazu wird der Aufruf einer Funktion direkt unterhalb den Anweisungen, die zu einer Aktion gehören, im Code integriert. Wenn z.B. das Attribut *Speed* des Objekts *Vehicle* modifiziert wird, muss die Funktion *setCarValuesAndDrive()* wie im folgenden Beispiel als Aufruf eingefügt werden. Über diese Funktion werden Steuersignale an das Fahrzeug gesendet.

```
public virtual void executeProgram() {
       VehicleSpeed = 2;
       …
      }
```

Nach vollständiger Transformation einer Aktion in Code wird das nächste Modellelement über die ausgehende Kontrollflusskante gesucht und bearbeitet.

4.5.2.3 Entscheidungsknoten

Im Gegensatz zur Simulation, bei der Entscheidungen zur Laufzeit getroffen werden, muss bei der Codegenerierung bei einem Entscheidungsknoten im Vorfeld festgestellt werden, ob es sich um eine einfache Bedingung oder eine Schleife handelt. Bei einer einfachen Bedingung hat ein Entscheidungsknoten eine einzige Eingangskante. Eine Bedingung wird in einen *if-else*-Codeblock transformiert, wobei die *if*-Abfrage der eigentlichen Bedingung entspricht. Für jede weitere Bedingung wird ein neuer *if-else*-Block dem *else*-Zweig der vorherigen Bedingung hinzugefügt. Jeder Ablaufzweig wird vollständig durchlaufen und der dabei erzeugte Code dem jeweiligen *if*-Block hinzugefügt. Die einzelnen Bedingungen werden aus dem *Guard*-Feld (siehe Abbildung 31) der abgehenden Kanten des Entscheidungsknotens erhalten und nach Konvertierung dem *CodeDom*-Element für eine Bedingungserzeugung übergeben. Dazu werden Ausdrücke der Form *Objekt.Attribut* in *ObjektAttribut* geändert. Außerdem werden die Boole' schen Operatoren *and* in *&&* und *or* in || transformiert. Wie bei einer Aktion werden alle vorkommenden Attribute innerhalb einer Bedingung in Variablen übersetzt, die aus dem Namen des Objekts und dem Namen des Attributes bestehen. Nach Abfrage des zugehörigen Typs werden die Variablen als *private* deklariert und mit vordefinierten Werten initialisiert, bevor sie unterhalb des Klassenanfangs eingefügt werden. Um eine mehrfache Erzeugung auszuschließen, werden die Variablen in eine Liste eingefügt, die vor jeder neuen Erzeugung einer Variablen überprüft wird.

Das Vorkommen einer Schleife kann durch mindestens zwei Eingangskanten bei einem Entscheidungsknoten charakterisiert werden. Für jeden Ablaufzweig einer abgehenden Kante findet dann eine Überprüfung statt, ob dieser wieder zum Entscheidungsknoten zurückführt. Wenn er nicht zurückführt, wird der Zweig als einfache Bedingung interpretiert und der entsprechende Code wie oben erzeugt, der anschließend dem übergeordneten *CodeDom*-Element hinzugefügt wird. Andernfalls wird der Code für eine Schleife erzeugt. Die gleichen Schritte bzgl. der Erzeugung von Variablen und der Konvertierung bei einer einfachen Bedingung werden auch bei einer Schleifenbedingung durchgeführt. Der Code, der sich aus dem weiteren Ablauf nach einer ausgehenden Kante bis zur Wiederkehr zum Entscheidungsknoten ergibt, wird der Schleife hinzugefügt.

Bei der Erkennung, dass ein Entscheidungsknotens über mindestens eine Schleife verfügt, wird zuerst eine leere Schleife erzeugt. Es sei hier erwähnt, dass *CodeDom* (siehe Abschnitt 4.5.1) bei einer Schleife eine spezielle *for*-Schleife generiert, die einer *while*-Schleife entspricht. Eine leere Schleife ist deshalb notwendig, da bei mehreren Schleifen, die zu einem Entscheidungsknoten zurückführen, jede Schleife nacheinan-

der im Code stehen würde und bei Beendigung einer unteren Schleife die Schleifenbedingung der Vorherigen nicht mehr abgefragt werden würde. Danach werden alle erkannten Schleifen in Code transformiert und der leeren Schleife hinzugefügt. Auch einfache Bedingungen werden in diese Schleife eingefügt. Bei der Ausführung dieser Bedingungen wird nach Ende des *if*-Blocks je nach Verschachtelungstiefe per *break*- oder *goto*-Anweisung aus der Schleife herausgesprungen. Im Anhang sind ein Aktivitätsdiagramm mit mehreren Entscheidungsknoten sowie der daraus generierte Code zu finden.

4.5.2.4 Aktion Verhaltensaufruf

Bei dieser Aktion wird ein zugehöriges Aktivitätsdiagramm in Programmcode transformiert. Als erstes wird nach dem Startknoten gesucht. Von diesem Startpunkt aus wird durch das Diagramm traversiert und Programmcode aus den durchlaufenen Modellelementen erzeugt. Am Ende wird wieder zum ursprünglichen Aktivitätsdiagramm zurückgesprungen und zum nächsten Modellelement übergegangen.

4.5.2.5 Splittingknoten

Bei einem Splittingknoten wird der Code, der für die Ausführung eines Threads notwendig ist, für jeden Ablaufzweig erstellt. Dabei ist die Anzahl der ausgehenden Kanten eines Splittingknotens mit der Anzahl der zu erzeugenden Threads gleichzusetzen. Zusätzlich wird eine Liste erstellt, zu der jeder Thread nach Beginn seiner Ausführung hinzugefügt wird. In jedem Thread wird eine Methode aufgerufen, die ebenfalls generiert wird. Dieser Methode wird der Code, der beim Durchlauf eines Threads erzeugt wird, hinzugefügt. Jeder Thread wird bis zu seinem Ablaufende durch einen Ablaufendknoten oder einen Synchronisationsknoten durchlaufen. Nachdem der Code für alle Threads erstellt worden ist, wird direkt im Anschluss eine Schleife generiert, die die Threadliste überwacht. Diese Schleife wird bei einer Ausführung solange durchlaufen, bis alle gestarteten Threads beendet sind.

4.5.2.6 Synchronisationsknoten

Mehrere Abläufe, die aus einem Splittingknoten resultieren, können in einem Synchronisationsknoten wieder zusammengeführt werden. Der weitere Ablauf wird so lange verzögert, bis alle Threads beendet sind. Dazu wird eine Schleife generiert, die nach dem Code für die Erzeugung der Threads eingefügt wird. Diese Schleife überwacht die Threadliste des zugehörigen Splittingknotens und wird solange ausgeführt, bis alle gestarteten Threads beendet und aus der Threadliste entfernt sind. Nach der Abarbeitung eines Synchronisationsknotens wird die Codegenerierung mit dem nächsten Modellelement fortgesetzt.

4.5.2.7 Aktion Signal senden und Aktion Signal empfangen

Die Transformation dieser Aktionen erzeugt den Code für einen Thread und eine Methode, die bei der Ausführung des Threads aufgerufen werden soll. Dieser Thread wird dem zugehörigen *CodeDom*-Element hinzugefügt und gestartet (siehe unteres Beispiel). Beim Durchlauf der *Aktion Signal senden* wird nach der *Aktion Signal empfangen* mit gleichem Namen wie die erstgenannte Aktion gesucht. Dabei kann die *Aktion Signal empfangen* im gleichen oder in einem anderen Aktivitätsdiagramm modelliert sein. Der Ablauf, der der *Aktion Signal empfangen* folgt, wird in Code transformiert und dem zugehörigen *CodeDom*-Element hinzugefügt. Nach Erreichen des Ablaufendknotens wird zu dem Aktivitätsdiagramm zurückgesprungen, das die *Aktion Signal senden* enthält. Danach wird mit der Bearbeitung des nächsten Modellelements begonnen.

```
Thread sendReceiveThread1;
sendReceiveThread1 = new
              Thread(newThreadStart(methodsendReceiveThread1));
sendReceiveThread1.Start();
```

4.5.2.8 Objekte

Alle in Aktionen oder Entscheidungsknoten verwendeten Objekte, die die Fahrzeugkomponenten repräsentieren, werden unterhalb des Klassenanfangs deklariert und im Konstruktor instantiiert. Diese Objekte sind gleichzeitig Instanzen der Klassen, die für die Kommunikation mit dem Fahrzeug zuständig sind. Mit ihnen können indirekt Steuerungsbefehle gesendet und Daten empfangen werden.

```
private Vehicle v1;
   //Konstruktor
   public CarControl() {
        v1 = new Vehicle();
        …
        }
```

4.5.2.9 Aktivitätsendknoten

Bei Erreichen eines Aktivitätsendknotens werden noch einige Codeblöcke der Codestruktur hinzugefügt, bevor der Code in eine Ausgabedatei geschrieben wird. Es wird abgefragt, welche Objekte bzgl. der Fahrzeugsteuerung verwendet werden. Dazu wird bei Anweisungen einer Aktion oder bei Abfragen eines Entscheidungsknotens ein *Flag* gesetzt, wenn auf ein bestimmtes Objekt zugegriffen wird. Für diese Objekte werden dann spezifische Funktionen, die für die Kommunikation mit dem Fahrzeug notwendig sind, der zu generierenden Klasse hinzugefügt. Wenn z.B. auf das Objekt *Compass* bei einem Entscheidungsknoten zugegriffen wird, soll das Kompassmodul verwendet werden. Mit der Funktion *initializeCompass()* wird unter anderem eine Instanz der Klasse *Compass* erzeugt, die für den Empfang der Kompassdaten zuständig ist. Diese Funktion ist im Anhang zu finden.

4.5.2.10 Ablaufendknoten

Im Falle dass kein Aktivitätsendknoten in der Modellierung enthalten ist, werden nach dem Durchlauf aller existierenden Ablaufzweige, die gleichen Schritte wie bei einem Aktivitätsendknoten durchgeführt. Ansonsten wird kein Code durch einen Ablaufendknoten erzeugt.

4.5.3 Performance bei der Ausführung des generierten Codes

Im Vergleich zur Simulation wird bei der Ausführung des Visual Studio-Projekts, in das der erzeugte Code integriert wird, eine deutlich höhere Performance erreicht. Das hat folgende Gründe: Bei der Simulation wird im Gegensatz zur Ausführung des VS-Projekts in jedem Schritt das zugehörige Modellelement grafisch hervorgehoben. Zusätzlich werden Objekte bei einer Änderung ihrer Attributwerte aktualisiert. Diese Vorgänge sind in Enterprise Architect sehr zeitintensiv. Beim VS-Projekt werden die Daten der Sensoren und des Kompassmoduls weiterhin über *Ereignisse* empfangen. Der Timer kann jedoch auf einen sehr viel niedrigen Wert gesetzt werden, da ein Update der Visual Studio Oberfläche im Vergleich zu einem Update der Objekte in Enterprise Architect sehr viel schneller bewerkstelligt werden kann. Der *Watcher*, der bei der Simulation das Objekt *Vehicle* überwacht hat, ist nicht mehr notwendig, da bei einer Änderung der Geschwindigkeit, der Richtung oder der Fahrzeit eine zusätzliche Funktion ausgeführt wird, die entsprechende Steuersignale ohne Verzögerung an das Fahrzeug sendet. Auch bei einer komplexen Modellierung profitiert das VS-Projekt davon, dass die Modellelemente nicht grafisch hervorgehoben werden. Bei der Ausführung von Threads wird bei der Simulation für jeden einzelnen Thread eine eigene Visualisierung gestartet. Bei der Ausführung des VS Projekts entfällt dieser Schritt.

5 Simulation und Codegenerierung am Fallbeispiel

In diesem Kapitel wird als Fallbeispiel eine konkrete Fahrzeugablaufsteuerung zum automatischen Einparken modelliert und simuliert. Die Fahrzeugsteuerung wird in Enterprise Architect mit einem Aktivitätsdiagramm modelliert. Die Simulation erfolgt durch die Ausführung des erstellten Aktivitätsdiagramms, bei der eine Kommunikation mit dem in Kapitel 4.3 beschriebenen Fahrzeug bzw. mit den enthaltenen Fahrzeugkomponenten stattfindet. Im Anschluss daran wird aufgezeigt, wie Quellcode aus dem erstellten Aktivitätsdiagramm zur Fahrzeugsteuerung erstellt werden kann. Der erzeugte Programmcode wird dabei automatisch in ein Visual Studio Projekt integriert, das alle zur Kommunikation mit dem Fahrzeug notwendigen Komponenten enthält. In einem weiteren Schritt wird das gesamte Projekt kompiliert und ausgeführt, wobei die Performance des Einparkvorgangs des VS-Projekts mit der Performance der Simulation verglichen wird.

5.1 Beschreibung des Szenarios

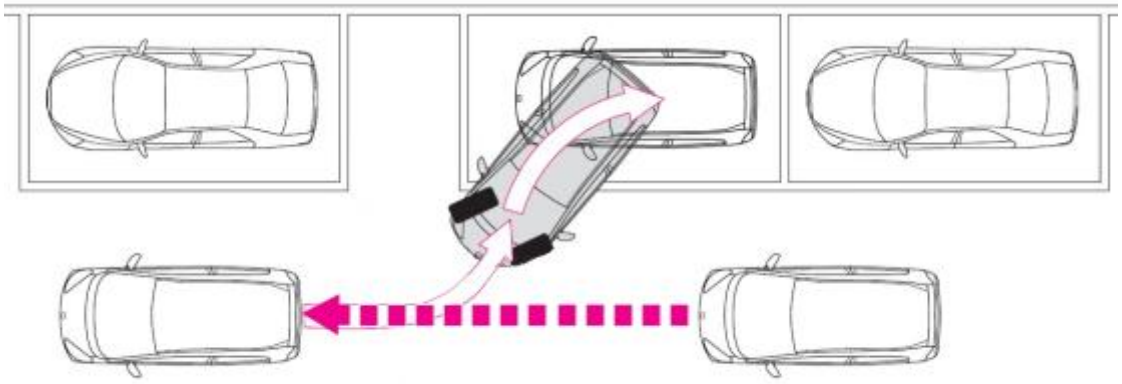


Abbildung 50: Rückwärts Einparken in eine parallel zur Fahrtrichtung befindliche Parklücke

Generell gibt es beim rückwärts Einparken zwei mögliche Fälle. Entweder wird parallel zur Fahrbahn oder in einem Winkel von 90° zur Fahrbahn eingeparkt. Die erste Art wird häufig beim Parken entlang einer Straße benutzt, wohingegen die zweite Art typisch bei Parkplätzen ist. Als Beispiel für die Modellierung wird der erste Fall betrachtet, bei dem der automatische Parkvorgang genauso wie in Abbildung 50 abläuft. Für das parallele Einparken gibt es zwei grundlegende Lösungsansätze [ARGM05]:

- Feedback-Methode: Bei diesem Ansatz fährt ein Fahrzeug autonom in eine Parklücke hinein, indem die Fahrzeugsteuerung ständig anhand von empfangenen Abstandsdaten korrigiert wird. Hierzu wird während der Bewegung des Fahrzeuges die genaue Position in Abhängigkeit der Parklücke bestimmt.
- Planning-Methode: Bei dieser Methode wird der Weg, den das Fahrzeug zur endgültigen Parkposition nehmen muss, im Vorfeld berechnet. Auf plötzliche Änderungen wie z.B. einer Verkleinerung der Parklücke kann nicht reagiert werden.

Im Fallbeispiel soll eine ausreichend große Parklücke im Vorbeifahren anhand von Abstandsmessungen der eingebauten Sensoren gesucht werden. Bei erfolgreichem Finden einer geeigneten Parklücke wird das Fahrzeug angehalten und durch eine Folge von Steuerungsbefehlen eingeparkt. Das Einparken ist geschwindigkeitsabhängig, weshalb die Modellierung entsprechend angepasst und getestet werden sollte. Durch

eine Fahrzeugsimulation kann festgestellt werden, ob eine Modellierung mit dem Aktivitätsdiagramm das gewünschte Einparkverhalten abbildet oder nicht. Sofern eine Korrektur erforderlich ist, kann das Modell relativ schnell geändert und anschließend wieder getestet werden. Diese iterative Vorgehensweise begünstigt die Beschleunigung einer korrekten Modellierung des Einparkens.

5.2 Anforderungen an die Modellierung und die Simulation des autonomen Einparkens

- Bei einer initialen Vorwärtsfahrt soll mithilfe der eingebauten Sensoren eine Parklücke parallel zur Fahrtrichtung erkannt werden, die mindestens so groß wie die Länge des Fahrzeuges plus 1 m ist.
- Wenn eine Parklücke mit obigem Kriterium gefunden wird, soll das Fahrzeug gestoppt und rückwärts eingeparkt werden.
- Das Fahrzeug soll mit einem Sicherheitsabstand in die Parklücke eingeparkt werden.
- Das Fahrzeug soll seine endgültige Parkposition durch eine Rückwärtsbewegung und eine Vorwärtsbewegung erreichen. Korrekturen während des Einparkens sollen nicht Bestandteil der Modellierung sein.
- Alle einzelnen Simulationsschritte sollen grafisch visualisiert werden.
- Jede Änderung der Fahrzeugsteuerung sowie der Sensoren soll in der Simulation sichtbar sein.
- Der Benutzer soll über eine Beendigung der Simulation informiert werden.
- Die Modellierung des Parkvorgangs soll mit einem Aktivitätsdiagramm erfolgen. Da Enterprise Architect über eine begrenzte Validierung hinsichtlich der Verknüpfung von Modellelementen verfügt, soll eine semantische Korrektheit durch den Benutzer selbst angestrebt werden.
- Syntaktische Fehler bei Anweisungen in Aktionen sowie bei Bedingungen in Entscheidungsknoten sollen angezeigt werden.

5.3 Simulation des Parkvorgangs

Das Aktivitätsdiagramm in Abbildung 51 enthält die Modellierung des autonomen Einparkvorgangs. Die Simulation erfolgt über die Ausführung des Aktivitätsdiagrammes, indem die einzelnen Modellelemente, die durch Kontrollflusskanten miteinander verbunden sind, nacheinander von der *Execution Engine* (siehe Abschnitt 4.4.1) zur Laufzeit interpretiert und ausgeführt werden. In Abbildung 51 sind neben den Modellelementen zur Ablaufsteuerung die beiden Objekte *Sensors* und *Vehicle* sowie das Objekt *WorkflowState* modelliert. Die Attributwerte von *Vehicle* werden durch die Interpretation der Aktionen direkt geändert oder gelesen. Beim Objekt *Sensors* werden die Attributwerte ausschließlich von außen durch Ereignisse modifiziert. Diese periodisch auftretenden Ereignisse werden von den optischen Abstandsmessungen der eingesetzten Sensoren ausgelöst. Das Objekt WorkflowState enthält zwei Attribute, auf die während der Simulation nur schreibend zugegriffen werden. Diese dienen dem Benutzer als Information darüber, in welchem Ausführungszusatnd sich die Simulation momentan befindet. Das Attribut *Running* ist vom Typ *Boolean*. Wenn die Simulation beginnt, wechselt der Attributwert von *false* auf *true*. Bei erfolgreicher Durchführung und Beendigung der Simulation ändert sich der Wert wieder auf *false*. Das Attribut *Element* ist vom Typ *String* und gibt das aktuelle Modellelement an, das durchlaufen, interpretiert und ausgeführt wird. Im Folgenden werden die einzelnen Simulationsschritte zusammen mit den jeweiligen Modellelementen und Anweisungen der Action Language zum autonomen Einparken des Fahrzeuges näher erläutert. Dabei sind Anweisungen konform zur Action Language (siehe Abschnitt 4.2) im *Notes*-Feld einer Aktion eingetragen (siehe Abbildung 30 in Abschnitt 4.2). Bedingungen sind jeweils im *Guard* - Feld einer

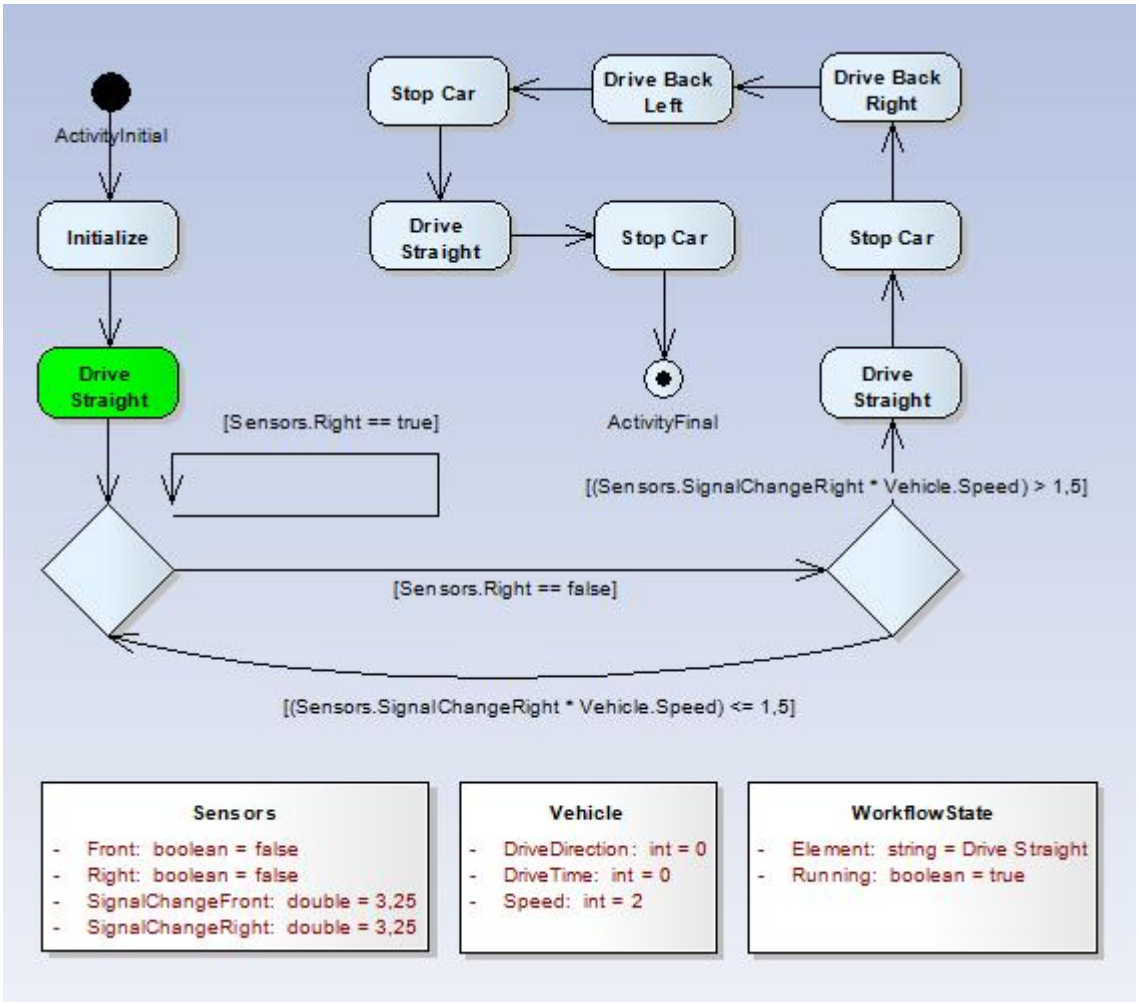


Abbildung 51: Modellierung: Autonomes Einparken

abgehenden Kontrollflusskante eines Entscheidungsknotens eingefügt (siehe Abbildung 31 in Abschnitt 4.2). Vor Beginn der Simulation müssen noch folgende Schritte durchgeführt werden: Wie in Abschnitt 4.4 beschrieben wird im *Project Browser* von Enterprise Architect das Aktivitätsdiagramm, das ausgeführt werden soll, ausgewählt. In diesem Fall wird das Aktivitätsdiagramm *Automatic Parking* selektiert. Danach werden in der Menüleiste von Enterprise Architect die Add-Ins zur Kommunikation mit den Fahrzeugkomponenten, die für die Simulation erforderlich sind, ausgeführt. Für die Modellierung des autonomen Einparkvorgangs wird das Add-In *Sensors* benötigt, das die Kommunikation zu den Sensoren herstellt und die empfangenen Sensorenwerte grafisch im Objekt *Sensors* (siehe Abbildung 51) darstellt. Zusätzlich wird das Add-In *CarControl* ausgeführt, um das Fahrzeug steuern zu können. Danach kann die Simulation durch die Auswahl des Add-Ins *Execute Diagram* gestartet werden (Abbildung 52).

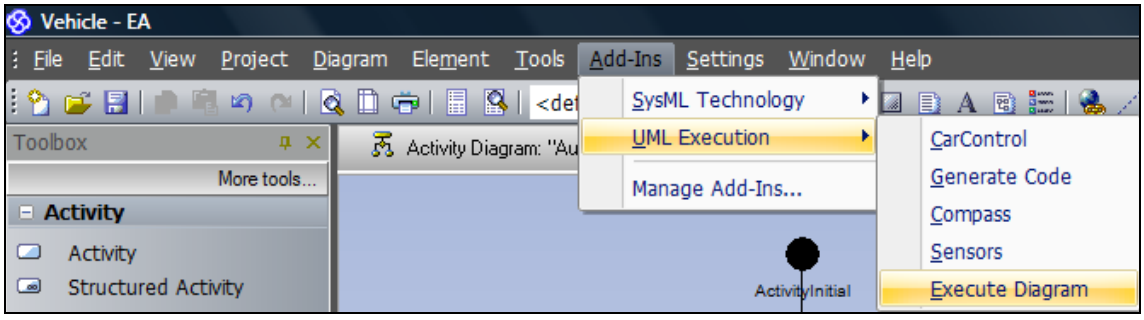


Abbildung 52: Add-In Execute Diagram

Startknoten: *Activity Initial*

Die *Execution Engine* (siehe Abschnitt 4.4.1) beginnt die Simulation, indem sie auf das zuvor ausgewählte Aktivitätsdiagramm *Automatic Parking* durch *repository.getCurrentDiagram()* zugreift (*repository* ist das Wurzelelement eines Enterprise Architect-Projekts). Im nächsten Schritt wird mit der Funktion *findInitialElement* (siehe Anhang) nach dem Startknoten *ActivityInitial* gesucht, der als Ausgangspunkt für die Traversierung durch das Diagramm dient. Der Name des Startknotens spielt dabei keine Rolle, da nur nach dem Typ gesucht wird.

Aktion: *Initialize*

Abbildung 53: Fahrzeugsimulation bei Aktion *Initialize*

Mit der Funktion *getNextElement* (siehe Anhang) wird die Aktion *Initialize* über die einzige abgehende Kontrollflusskante des Startknotens gefunden. Die *Execution Engine* erkennt, dass es sich bei diesem Modellelement um eine Aktion handelt und liest deshalb die Anweisungen im *Notes*-Feld der Aktion aus. Jede Anweisung wird mit einem Strichpunkt beendet und kann somit separat abgearbeitet werden. Durch die Funktion *DoCalculation* (siehe Anhang) wird jeder Ausdruck ausgewertet bzw. berechnet. Die Aktion *Initialize* setzt wie unten gezeigt, alle Werte der Attribute, die in der Simulation benutzt werden, auf Initialwerte wie *false* oder 0. Dieser Initialschritt ist deshalb sinnvoll, weil es vorkommen kann, dass eine Simulation mit Attributwerten beendet wird, die nicht zu Beginn vorkommen sollen. Außerdem ist es möglich, dass eine Simulation aufgrund eines Fehlers abgebrochen wird. Die Attribute wären dann mit den Werten gesetzt, die kurz vor Abbruch der Simulation gültig waren. Bei einem Neubeginn der Simulation würde dann das Fahrzeug z.B. gleich rückwärts anstatt vorwärts fahren. Mit den Anweisungen aus dieser Aktion (siehe unten) beträgt die Anfangsgeschwindigkeit 0 m/s und die Räder sind in einem Winkel von 0° zum Fahrzeug ausgerichtet. Der rechte Sensor wird auf *true* gesetzt und die Zeit, die zwischen einem Signalwechsel vergeht, auf 0. Die Initialisierung des Sensors ist deshalb notwendig, da der Abstand einer Lücke in Abhängigkeit der Zeit, die ab einem Signalwechsel vergeht, berechnet wird. Da das Add-In *Sensors* vor der Ausführung des Aktivitätsdiagrammes gestartet wird, wird die Zeit ab diesem Punkt hochgezählt (siehe Abschnitt 4.3.3). Wenn zu Beginn der Simulation auf der rechten Seite kein Gegenstand oder Objekt vorhanden wäre (`Sensors.Right = false`) und die Lücke insgesamt zu klein wäre, würde das Fahrzeug dennoch einen Parkversuch starten.

```
Sensors.Right = true;
Sensors.SignalChangeRight = 0;
Vehicle.DriveDirection = 0;
Vehicle.Speed = 0;
Vehicle.DriveTime = 0;
```

Aktion: *Drive Straight*

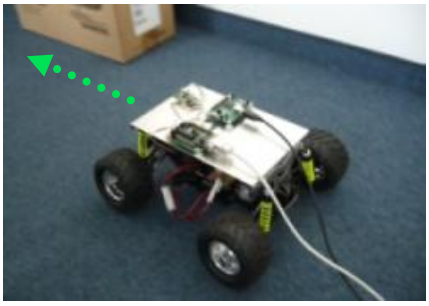


Abbildung 54: Fahrzeugsimulation bei Aktion *Drive Straight*

Nach der Ausführung der Aktion *Initialize* wird über deren Kontrollflusskante erneut mit der Funktion *getNextElement* das nächste Modellelement gefunden. Nach Auslesung der unten aufgeführten Anweisung der Aktion Drive *Straight* wird mit *DoCalculation* die Geschwindigkeit des Fahrzeuges neu gesetzt. Danach fährt es mit 2 m/s geradeaus. Da `Vehicle.DriveTime = 0;` aus der vorhergehenden Initialisierungsaktion noch gültig ist, fährt das Fahrzeug solange mit dieser Geschwindigkeit geradeaus, bis bei einer folgenden Aktion entweder eine neue Richtung angegeben oder die Geschwindigkeit geändert wird.

```
Vehicle.Speed = 2;
```

Entscheidungsknoten 1:

Als nächstes Modellelement im Aktivitätsdiagramm wird der Entscheidungsknoten 1 mit zwei Bedingungen erreicht. Die Bedingungen sind jeweils im *Guard*-Feld der abgehenden Kontrollflusskanten eingetragen. Bei diesem Entscheidungsknoten wird abgefragt, ob die Abstandsmessung des rechten Sensors 15 cm unterschreitet oder nicht. Die Auswertung der beiden Bedingungen (siehe unten) erfolgt über die Funktion *DoDecision* (siehe Anhang). Wenn eine Bedingung als wahr ausgewertet wird, wird mit der Funktion *getNextElement* das über diese Bedingungskante zu erreichende nächste Modellelement erhalten. Bei `[Sensors.Right = true]` ist das nächste Element wieder der Entscheidungsknoten 1. Die Simulation läuft solange in dieser Schleife ab, bis das Fahrzeug auf der rechten Seite eine Lücke anhand der Abstandsmessung erkennt. Wenn die Bedingung 2 mit `[Sensors.Right = false]` erfüllt ist, fährt das Fahrzeug an einer Lücke entlang und der nächste Simulationsschritt erfolgt mit der Bearbeitung des zweiten Entscheidungsknotens. Im Falle, dass die Bedingung 1 als wahr ausgewertet wird, findet eine Überprüfung der zweiten Bedingung nicht mehr statt. Diese Vorgehensweise wirkt sich positiv auf die Performance der Simulation aus.

Bedingung 1:

```
[Sensors.Right == true]
```

Bedingung 2:

```
[Sensors.Right == false]
```

Entscheidungsknoten 2

Beim Entscheidungsknoten 2, der auch zwei Bedingungen enthält, wird die Länge der Parklücke überprüft. *Sensors.SignalChangeRight* gibt die Zeit an, die seit dem Wechsel von `[Sensors.Right = true]` auf `[Sensors.Right = false]` vergangen ist. Dies entspricht der Fahrzeit ab dem Beginn der Parklücke. Mit dieser Zeitspanne und der Geschwindigkeit kann über die Formel s = v*t der zurückgelegte Weg des Fahrzeuges berechnet werden. Damit das Einparken mit Sicherheitsabstand gewährleistet werden kann, soll die Parklücke mindestens 1,5 m betragen. Wenn die berechnete Länge der Lücke kleiner als oder gleich 1,5 m beträgt, wird wieder zum Entscheidungsknoten 1 übergegangen. In dieser zweiten Schleife wird solange verblieben, bis eine Parklücke gefunden wird, die größer als 1,5 m ist. In diesem Fall (Bedingung 1) ist das nächste Element in der Simulation die Aktion *Drive Straight (2)*.

Bedingung 1

```
[(Sensors.SignalChangeRight * Vehicle.Speed) > 1,5]
```

Bedingung 2

```
[(Sensors.SignalChangeRight * Vehicle.Speed) <= 1,5]
```

Aktion: *Drive Straight (2)*

Abbildung 55: Fahrzeugsimulation bei Aktion *Drive Straight (2)*

Die Aktion *Drive Straight(2)* bildet den Ausgangspunkt für das Einparken. Wenn dieser Schritt erreicht ist, beträgt die Länge der Parklücke mehr als 1,5 m. Damit nun rückwärts eingeparkt werden kann, wird wie in Abbildung 55 für die Dauer von 1 s nach vorne gefahren, bevor zur nächsten Aktion übergegangen wird. Dieser Schritt ist deshalb notwendig, weil zum Einparken zunächst nach vorne ausgeholt werden muss, um ein korrektes Einparken mit Sicherheitsabstand durchführen zu können (siehe Abbildung 50).

```
Vehicle.Speed = 2;
Vehicle.DriveTime = 1;
```

Aktion: *Stop Car*

Mit der Aktion *Stop Car* wird das Fahrzeug zum Stillstand gebracht (Abbildung 56), indem die Geschwindigkeit auf 0 gesetzt wird. Da das Attribut *DriveTime* noch auf eine Sekunde eingestellt ist, verweilt das Fahrzeug kurz in dieser Position. Andernfalls würde das Fahrzeug vom Vorwärtsgang sofort ohne Stillstand in den Rückwärtsgang (siehe nächste Aktion) schalten, was langfristig Schäden am Antrieb verursachen kann.

Abbildung 56: Fahrzeugsimulation bei Aktion *Stop Car*

```
Vehicle.Speed = 0;
```

Aktion: *Drive Back Right*

Bei dieser Aktion wird das Fahrzeug mit dem äußersten Wert auf der Skala nach rechts gelenkt. Gleichzeitig wird die Geschwindigkeit auf den negativen Wert -2 gesetzt. Mit `Vehicle.DriveTime = 3;` beträgt die Fahrtdauer drei Sekunden. Das Fahrzeug fährt somit drei Sekunden lang rückwärts und biegt dabei rechts in die Parklücke ein (Abbildung 57).

Abbildung 57: Fahrzeugsimulation bei Aktion *Drive Back Right*

```
Vehicle.Speed = -2;
Vehicle.DriveDirection = 4;
Vehicle.DriveTime = 3;
```

Aktion Drive Back Left:

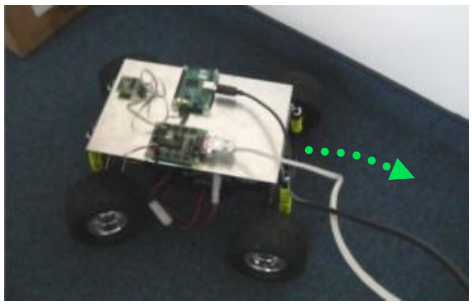


Abbildung 58: Fahrzeugsimulation bei Aktion *Drive Back Left*

Mit der Aktion *Drive Back Left* werden die Geschwindigkeit und die Fahrtdauer aus der vorherigen Aktion beibehalten. Die Lenkung ändert sich jedoch von rechts auf links (Abbildung 58). Am Ende dieser Aktion hat das Fahrzeug eine parallele Position zur

Fahrbahn eingenommen. Die Vorderreifen sind allerdings in diesem Schritt noch nicht gerade ausgerichtet.

```
Vehicle.DriveDirection = -4;
Vehicle.Speed = -2;
Vehicle.DriveTime = 3
```

Aktion: *Stop Car (2)*

Abbildung 59: Fahrzeugsimulation bei Aktion *Stop Car (2)*

Mit der Aktion *Stop Car (2)* wird das Fahrzeug erneut angehalten. Da die Vorderreifen noch nach links gerichtet sind, wird mit `Vehicle.DriveDirection = 0;` das Fahrzeug für die gerade Vorwärtsfahrt (nächste Aktion) vorbereitet (Abbildung 59). Diese Richtungsänderung kann innerhalb einer Sekunde mit `Vehicle.DriveTime = 1;` abgeschlossen werden.

```
Vehicle.DriveDirection = 0;
Vehicle.Speed = 0;
Vehicle.DriveTime = 1;
```

Aktion: *Drive Straight (3)*

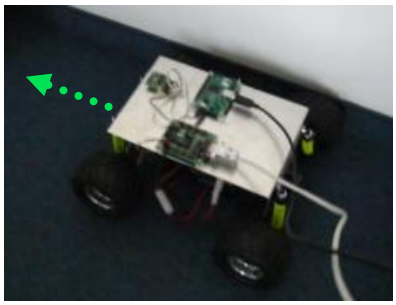


Abbildung 60: Fahrzeugsimulation bei Aktion *Drive Straight (3)*

Damit das Fahrzeug in der Mitte bzw. nicht ganz an einem Ende der Parklücke zum Stehen kommt, fährt das Fahrzeug mit der Aktion *Drive Straight (3)* ein letztes Mal für die Dauer von einer Sekunde geradeaus (Abbildung 60), bevor die letzte Aktion der Simulation ausgeführt wird.

```
Vehicle.DriveDirection = 0;
Vehicle.Speed = 2;
Vehicle.DriveTime = 1;
```

Aktion: *Stop Car (3)*

Abbildung 61: Fahrzeugsimulation bei Aktion *Stop Car (3)*

Mit der Aktion *Stop Car (3)* wird die endgültige Parkposition des Fahrzeuges erreicht, indem die Geschwindigkeit wieder auf 0 gesetzt wird. Die Simulation ist an dieser Stelle beendet.

```
Vehicle.Speed = 0;
```

5.3.1.1 Performance der Simulation

Wie in Abschnitt 4.4.2 beschrieben, hängt die Performance einer Simulation von mehreren Faktoren ab. Innerhalb dieses Fallbeispiels sollen diese nun konkret betrachtet werden. Die Simulation wurde auf einem Rechner mit einem Intel Core 2 Duo 2 GHZ Prozessor, 2 GB Ram, 256 MB Grafikkarte und Microsoft Windows Vista als Betriebssystem durchgeführt.

Für die Simulationsausführung wurden die Add-Ins *Sensors* und *CarControl* im Vorfeld ausgeführt, bevor im Anschluss der Parkvorgang durch das Add-In *ExecuteDiagram* simuliert wurde. Das Timerintervall, in dem die Sensoren ihre Abstandsmessung an Enterprise Architect senden sowie der Zeitschritt, mit dem die vergangene Zeit seit dem letzten Signalwechesel des vorderen oder rechten Sensors erhöht wird, ist auf 250 ms gesetzt worden. Der *Watcher* (siehe Abschnitt 4.1) schickt bei einer Änderung eines Attributwertes des Objekts *Vehicle* im Abstand von 50 ms Steuersignale an das Fahrzeug. Diese Timerwerte haben sich mit dem benutzten Rechner zusammen mit der speziellen Modellierung des Parkvorgangs als guter Kompromiss zwischen Performance der Simulation und der Reaktionsgeschwindigkeit auf Änderungen der Objekte in Enterprise Architect sowie der Änderung der Sensorenwerte am Fahrzeug, erwiesen.

Die Komplexität des Parkvorgangs hält sich in Grenzen. Es wurden keine rechenintensiven Threads bei der Modellierung verwendet. Außerdem wurde darauf verzichtet, bestimmte Teile in separate Aktivitätsdiagramme mit der *Aktion Verhaltensaufruf* (siehe Abschnitt 4.4.1.4) oder der *Aktion Signal senden* (siehe Abschnitt 4.4.1.7) auszulagern, was sich positiv auf die Performance der Simulation ausgewirkt hat. Da nur am Simulationsanfang nach dem ausgewählten Diagramm gesucht wurde, ist der Gesamtanzahl der im Projekt enthaltenen Diagramme keine große Beachtung geschenkt worden. Die in der Modellierung enthaltenen Entscheidungsknoten verfügen über jeweils zwei Bedingungskanten. Wenn die erste Bedingung als wahr ausgewertet wird, wird die zweite Bedingung nicht mehr überprüft. Somit finden bei einer Entscheidung im Durchschnitt 1,5 Überprüfungen statt.

Ein EA-Projekt wird intern als Datenbank gespeichert (siehe Abschnitt 4.4.2), auf die per SQL-Befehle zugegriffen werden kann. Wenn z.B. ein Attributwert eines bestimmten Objektes ausgelesen werden soll, kann dies entweder mit dem COM Modell über eine Iterierung aller Objekte mit anschließender Überprüfung der Attribute des passenden Objektes oder direkt per Datenbankzugriff erfolgen. Die erstgenannte Variante

wurde vor der Optimierung der Performance angewendet und dauerte ca. 50 ms. Nach der Optimierung mit Direktzugriff auf die Datenbank per SQL reduzierte sich das Auslesen eines Attributwertes zu einem zugehörigen Objekt auf 5 ms..

Zusammenfassend ist festzustellen, dass die Ausführungsgeschwindigkeit einer Simulation von vielen Faktoren (siehe Abschnitt 4.4.2) abhängt, die individuell angepasst und optimiert werden können.

5.4 Code-Generierung

Nach dem die Simulation des Parkvorgangs ausgeführt und getestet worden ist, wird in einem weiteren Schritt über das Add-In *Generate Code* (siehe Abbildung 41) der Programmcode für den Fahrablauf generiert. Der erzeugte Code wird in die Klasse *CarControl* geschrieben, die in ein Visual Studio 2008 Projekt integriert wird. Dieses Projekt enthält alle zur Kommunikation mit dem Fahrzeug notwendigen Klassen. Nach der Kompilierung wird eine grafische Oberfläche gestartet (siehe Abbildung 49), die die Geschwindigkeit, die Richtung und die Sensorenwerte des Fahrzeuges anzeigt. Durch Drücken des Buttons *Start Execution* wird das Projekt ausgeführt und das Fahrzeug parkt erneut autonom anhand des generierten Codes ein. Im Folgenden wird auf die Codegenerierung durch die Transformation der Modellelemente des Aktivitätsdiagrammes aus Abbildung 51, das den Parkvorgang modelliert, näher eingegangen.

5.4.1 Transformation der verwendeten Modellelemente des Parkvorgangs

Die Erzeugung von Programmcode aus den Modellelementen des Parkvorgangs findet mithilfe der *CodeDom*-Elemente statt, von denen die wichtigsten in Tabelle 2 in Abschnitt 4.5.1) aufgelistet sind. Zu Beginn der Codegenerierung werden die benötigten Pakete der Klasse *CarControl*, die am Ende den erzeugten Code enthält, erstellt. Danach wird der Namespace, der Klassenrumpf, der Kontsruktor und die Einstiegsmethode *executeProgram()* generiert. Der bis dahin erzeugte Code sieht folgendermaßen aus:

```
using System;
using System.IO;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Windows.Forms;
using LL.EA.UMLExecution.Properties;

namespace LL.EA.UMLExecution {

    public class CarControl {

        public CarControl() {}

        public virtual void executeProgram() {}
    }
}
```

Wie bei der Simulation wird das Aktivitätsdiagramm des Parkvorgangs durchlaufen und zunächst nach dem Startknoten gesucht, von dem aus zum nächsten Modellelement übergegangen wird. Die Anweisungen der Aktion *Initialize* in Abbildung 62 werden sequentiell ausgelesen und bearbeitet. Dabei werden Ausdrücke der Form *Objekt.Attribut* in *ObjektAttribut* geändert. Die Attribute des rechten Sensors sowie die Attribute des Objekts *Vehicle* werden in Variablen übersetzt und in eine Liste eingefügt. Diese Va-

riablen werden unterhalb des Klassenanfangs deklariert und mit vordefinierten Werten initialisiert.

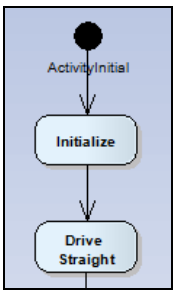


Abbildung 62: Beginn des Parkvorgangs

Anschließend werden die konvertierten Anweisungen der Methode *execute Program()* hinzugefügt. Da bei dem Visual Studio-Projekt kein *Watcher* mehr existiert, der die Änderungen des Objekts *Vehicle* überwacht, wird nach den Anweisungen einer Aktion, die die Attributwerte dieses Objekts modifiziert, die Funktion *setCarValuesAndDrive()* als Aufruf in den Code eingefügt. Diese Funktion sendet Steuerungsbefehle an das Fahrzeug und wird bei Erreichen des Aktivitätsendknotens erzeugt.

Über die Ausgangskante der Aktion *Initialize* wird die Aktion *Drive Straight* erreicht. Da das Attribut *Vehicle.Speed* schon in der Variablenliste enthalten ist, wird nur die Anweisung `Vehicle.Speed = 2;` in `VehicleSpeed = 2;` konvertiert und mit dem Aufruf der Funktion *setCarValuesAndDrive()* in die Einstiegsmethode eingefügt. Nach diesen beiden Aktionen sieht der Code wie folgt aus:

```
…
    public class CarControl {

        private bool SensorsRight = true;
        private double SensorsSignalChangeRight = 0;
        private int VehicleDriveDirection = 0;
        private int VehicleSpeed = 0;
        private int VehicleDriveTime = 0;

        public CarControl() {
        }

        public virtual void executeProgram() {
            SensorsRight = true;
            SensorsSignalChangeRight = 0;
            VehicleDriveDirection = 0;
            VehicleSpeed = 0;
            VehicleDriveTime = 0;
            setCarValuesAndDrive();
            VehicleSpeed = 2;
            setCarValuesAndDrive();
        }
    }
```

Anschließend wird der erste Entscheidungsknoten erreicht. Da er mehr als eine eingehende Kante besitzt, existiert mindestens eine Schleife. Deshalb wird zunächst eine leere Schleife erzeugt (siehe Abschnitt 4.5.2.3). Für jede abgehende Kante findet eine Überprüfung anhand eines vollständigen Durchlaufes statt, ob ein Zweig des Ablaufes wieder zurück zum Entscheidungsknoten führt. Wenn er zurückführt, wird der Code für eine Schleife erzeugt, andernfalls der Code für eine einfache Bedingung.

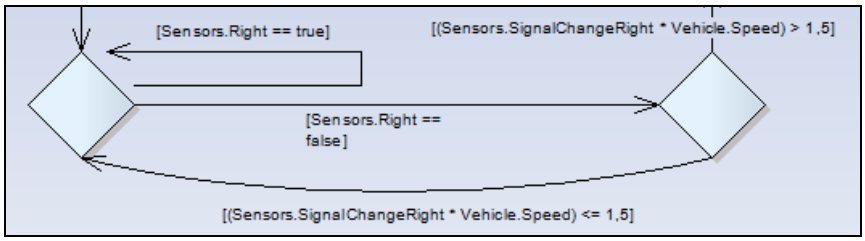


Abbildung 63: Codegenerierung bei Entscheidungsknoten

Die erste abgehende Kante mit der Bedingung `[Sensors.Right == true]` in Abbildung 63 führt direkt wieder zum Entscheidungsknoten zurück. Da die Variable *SensorsRight* bereits in der Variablenliste vorhanden ist, wird nur die Bedingung der Schleife in `[SensorsRight == true]` konvertiert und anschließend der Code für eine Schleife erzeugt. Diese Schleife wird der übergeordneten leeren Schleife hinzugefügt. Die generierte Schleife wird bei einer Ausführung solange durchlaufen, bis sich die Auswertung der Bedingung ändert, da keine Anweisungen in ihr ausgeführt werden. Für die zweite abgehende Kante mit der Bedingung `[Sensors.Right == false]` wird ebenfalls eine Schleife erzeugt, da ein Ablauf existiert, der zum Entscheidungsknoten zurückführt. Da der Code für den Ablauf der ersten Kante bereits vollständig erstellt ist, wird zum zweiten Entscheidungsknoten übergegangen. Da der Entscheidungsknoten über eine einzige Eingangskante und zwei Ausgangskanten verfügt, handelt es sich um eine einfache Bedingung. Für die Bedingung `[Sensors.SignalChangeRight * Vehicle.Speed > 1,5]` der ersten abgehenden Kante wird ein *if*-else-Codeblock nach Konvertierung der Bedingung wie oben erstellt und in die übergeordnete Schleife eingefügt. Der Code, der beim Durchlaufen des weiteren Ablaufzweiges erzeugt wird, wird dem *if*-Block hinzugefügt. Die zweite Bedingung `[Sensors.SignalChangeRight * Vehicle.Speed <= 1,5]` wird nach der Konvertierung als *if*-Codeblock in den *else*-Zweig der ersten Bedingung eingefügt. Da dieser Zweig als nächstes Modellelement wieder den ersten Entscheidungsknoten hat, wird nach der Ausführung des Ablaufzweigs wieder zum Anfang der übergeordneten Schleife zurückgesprungen. Der erzeugte Code der beiden Entscheidungsknoten ist im folgenden Codeabschnitt zu sehen:

```
for (; ;)
{
   Thread.Sleep(10);
   Application.DoEvents();
   for (
   ; SensorsRight == true;
   ) {
       Thread.Sleep(10);
       Application.DoEvents();
   }
   for (; SensorsRight == false;) {
      Thread.Sleep(10);
      Application.DoEvents();
      if ((SensorsSignalChangeRight * VehicleSpeed) > 1.5) {
     }
      else {
        if ((SensorsSignalChangeRight * VehicleSpeed) <= 1.5) {
           }
      }
   }
}
```

Als nächstes wird der weitere Ablauf der ersten Bedingung des zweiten Entscheidungsknotens bearbeitet. Alle Anweisungen, die in den Aktionen in Abbildung 64 enthalten sind, werden nach ihrer Konvertierung dem *if*-Block hinzugefügt. Zusätzlich wird der Aufruf der Funktion *setCarValuesAndDrive()* bei einer Änderung eines Attributes des Objekts *Vehicle* eingefügt.

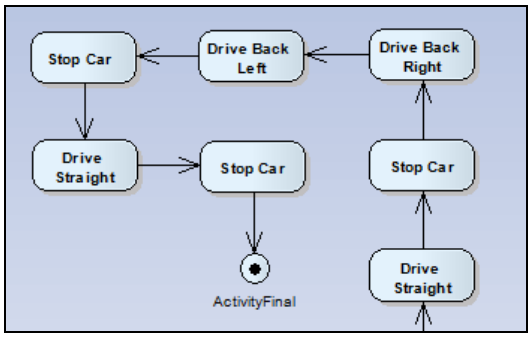


Abbildung 64: Codegenerierung bei rückwärts Einparken

Bei Erreichen des Aktivitätsendknotens werden abhängig von den verwendeten Komponenten noch einige Funktionen, Deklarationen und Zuweisungen der Codestruktur hinzugefügt. Da auf die Objekte *Vehicle* und *Sensors* zugegriffen wird, werden Instanzen der Klasse *Vehicle* und *Sensors* erzeugt. Zusätzlich wird die Funktion *setCarValuesAndDrive()* sowie ereignisbezogene Codefragmente, um auf die Sensorenwerte zugreifen zu können, erstellt. Um die Geschwindigkeit und die Sensorenwerte grafisch im VS-Projekt anzeigen zu können, werden entsprechende Funktionen in die zu erzeugende Klasse eingefügt. Der vollständig generierte Code ist im Anhang zu finden.

5.4.2 Performance bei der Ausführung des generierten Codes

Die Ausführungsgeschwindigkeit des Visual Studio-Projekts, in das der zuvor generierte Code integriert wird, ist im Vergleich zur Simulation deutlich höher. Es gibt beim VS-Projekt keine Modellelemente mehr, die in jedem Ausführungsschritt grafisch hervorgehoben werden. Es werden nur die Anzeige für die Geschwindigkeit, die Richtung und die Sensorenwerte in der grafischen Oberfläche des VS-Projekts geändert. Im Vergleich zu einem Update in Enterprise Architect kann dies jedoch viel schneller durchgeführt werden. Die Daten der Sensoren werden weiterhin über Ereignisse empfangen, der Wert für den Timer wird jedoch auf 30 ms im Gegensatz zu 250 ms bei der Simulation herabgesetzt. Mit dieser Einstellung kann das Fahrzeug schneller auf Änderungen in seiner Umgebung reagieren. Zusätzlich zu dieser Verbesserung wird kein *Watcher* (siehe Abschnitt 4.1) mehr benötigt, da an dessen Stelle eine Funktion ohne Verzögerung das Senden von Steuerungssignalen an das Fahrzeug übernimmt.

6 Zusammenfassung und Ausblick

In dieser Diplomarbeit wurde gezeigt, wie UML Aktivitätsdiagramme ausgeführt und in Code transformiert werden können. Dazu war der Einsatz einer Action Language sinnvoll, die definiert worden ist. Darauf aufbauend wurde simuliert, wie ein Fahrzeug mit ausführbaren Aktivitätsdiagrammen autonom gesteuert werden kann. Für die einzelnen Komponenten des Fahrzeuges sowie für die Simulation und die Codegenerierung wurden verschiedene Add-Ins für Enterprise Architect entwickelt. Die Kommunikation zwischen den Fahrzeugkomponenten und Enterprise Architect erfolgt über die Add-Ins *Compass*, *Sensors* und *CarControl*. Die Simulation findet dann mit dem Add-In *Execute Diagram* statt, das die *Execution Engine* (siehe Abschnitt 4.4.1) zur Interpretation und Ausführung der Modellelemente eines Aktivitätsdiagrammes enthält. Über das Add-In *Generate Code* kann Programmcode aus den Modellen generiert werden. Dieser Code wird anschließend in ein Visual Studio 2008 Projekt integriert, das kompiliert und ausgeführt werden kann. Nach der Codegenerierung ist dann theoretisch Enterprise Architect nicht mehr notwendig, um das Fahrzeug autonom steuern zu können. Als Fallbeispiel diente die Modellierung und Simulation eines Fahrablaufs, bei dem das Fahrzeug rückwärts in eine Parklücke abhängig von der Größe der Lücke einparken sollte. Zusätzlich wurde nach erfolgreichen Testläufen Programmcode aus dem Modell erstellt. Der ausgeführte Fahrablauf aus der Codegenerierung wurde zu Performancezwecken mit dem aus der Simulation verglichen.

In Zukunft ist es denkbar, dass weitere Komponenten in das Fahrzeug integriert werden. Da der modulare Ansatz der Komponentensteuerung eine einfache Erweiterbarkeit erlaubt, muss für jede neue Komponente nur ein weiteres Add-In entwickelt werden. Für die neue Komponente wird dann ein neues Objekt mit entsprechenden Attributen modelliert, auf das mit Anweisungen und Abfragen konform zur entwickelten Action Language (siehe Abschnitt 4.2) zugegriffen werden kann. Als nächste Komponente wäre ein GPS-Empfänger sinnvoll, um das Fahrzeug abhängig von Positionsdaten zu steuern. Um zu einem spezifischen GPS-Punkt zu navigieren, könnte ein grafischer Algorithmus mit den Modellelementen des Aktivitätsdiagrammes entwickelt werden. Auch denkbar wären weitere Sensoren, die am Fahrzeug befestigt werden, um noch komplexere Fahrabläufe modellieren zu können.

Mit dem Add-In *Generate Code* kann Programmcode aus Aktivitätsdiagrammen in C# erzeugt werden. Wünschenswert wären weitere Generatoren für die Erstellung von Code in anderen Programmiersprachen.

Die entwickelte *Execution Engine* umfasst die Simulation von Aktivitätsdiagrammen. Dabei können jedoch nicht alle Modellelemente, die in der UML2 spezifiziert sind, interpretiert werden. Die Execution Engine könnte erweitert werden, um Elemente wie strukturierte Schleifen zu unterstützen. Es wäre auch vorstellbar, dass in Zukunft weitere UML Diagramme wie das Sequenzdiagramm oder das Zustandsmaschinendiagramm ausgeführt werden können. Die Codegenerierungseinheit wäre bei diesen Änderungen dann ebenfalls anzupassen. Da die *Execution Engine* auch Aktivitätsdiagramme ohne Fahrzeugbezug ausführen kann, könnte diese auch für andere Szenarien benutzt werden. So könnte das Add-In *Execute Diagram*, das die *Execution Engine* enthält, z.B. zur Steuerung eines Roboters eingesetzt werden. In diesem Fall wären dann zusätzliche Add-Ins für die Kommunikation zwischen den Roboterkomponenten und Enterprise Architect zu entwickeln.

Anhang A

Abbildungsverzeichnis

Anhang B

Funktion *findInitialElement*

```
private SPXEA. Element findInitialElement(SPXEA.Diagram diagram)
{
    try
    {
        foreach (SPXEA.DiagramObject diagObj in diagram.DiagramObjects)
        {
            if (diagObj.ElementID > 0)
            {
                SPXEA.Element element = repository.GetElementByID(diagObj.ElementID);

                if (((element.Type == Settings.Default.ActivityInitial) &&
                      (element.Subtype == Settings.Default.ActivityInitialSubType)))
                {
                    return element;
                }
            }
        }
    }
    catch (Exception ex)
    {
        throw new Exception(Resources.exNoInitalElement, ex);
    }
    throw new Exception(Resources.exNoInitalElement);
}
```

Funktion *getNextElement*

```
private SPXEA.Element getNextElement(SPXEA.Element element)
{
    if (!object.Equals(element, null))
    {
        try
        {
            foreach (SPXEA. Connector con1 in element.Connectors)
            {
                if (con1.ClientID.Equals(element.ElementID))
                {
                    return repository.GetElementByID(con1.SupplierID);
                }
            }

        }
        catch (Exception ex)
        {
            throw new Exception(Resources.exNoNextElements, ex);
        }
    }
    return null;
}
```

Funktion *doCalculation*

```
private void doCalculation(string line)
{
    string leftSide = string.Empty;
    string rightSide = string.Empty;
    string[] instructions;

    List<SPXEA.Attribute> specificAttributes = new List<SPXEA.Attribute>();
    Regex regEqual = new Regex(Settings.Default.regEqual);
    Regex reg = new Regex(Settings.Default.regularexp);

    //linke Seite: Element mit Attribut
    MatchCollection matchList = reg.Matches(line);

    // wenn Liste befüllt ist...
    if (matchList.Count > 0)
```

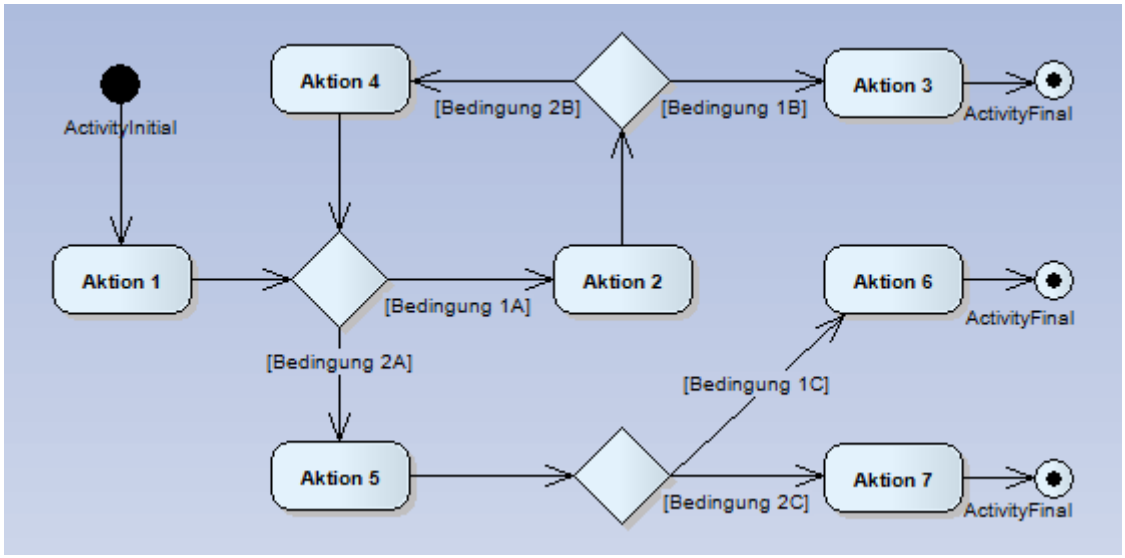
```
    {
        if ((matchList[0].Success) && (!matchList[0].Value.Length.Equals(0)))
        {
            leftSide = matchList[0].Value;
        }

        // das zu setzende Element mit Attribut
        specificAttributes = getAttributesByPath(leftSide);

        if (specificAttributes.Count > 0)
        {
            //rechte Seite:
            instructions = regEqual.Split(line);

            //rechte Seite nach '='
            if ((instructions.Length == 2) && (!string.IsNullOrEmpty(instructions[1])))
            {
                rightSide = instructions[1].Trim();
            }

            //alle Element.Attribut in rechter Seite durch aktuellen Wert ersetzen
            for (int i = 1; i < matchList.Count; i++)
            {
                string attributeValue =
                      getFirstAttributeValueByPath(matchList[i].Value);
                rightSide = rightSide.Replace(matchList[i].Value, attributeValue);

            }

            rightSide = rightSide.Replace(" False", " false");
            rightSide = rightSide.Replace(" True", " true");
            //rechte Seite an Scripting übergeben und berechnen lassen

            string returnValue = scriptingCalculation(rightSide);

            //jedem Attribut berechneten Wert zuweisen und updaten

            foreach (SPXEA.Attribute att1 in specificAttributes)
            {
                att1.Default = returnValue;
                att1.Update();
                if (!object.Equals(visualisation, null))
                {
                    repository.GetElementByID(att1.ParentID).Update();
                }

            }
        }
    }
}
```

Funktion *doDecision*

```
private bool doDecision(string line)
{

    if (!string.IsNullOrEmpty(line))
    {
        Regex reg = new Regex (Settings.Default.regularexp);
        MatchCollection matchList = reg.Matches(line);

        if (matchList.Count > 0)
        {
            if ((matchList[0].Success) && (!matchList[0].Value.Length.Equals(0)))
            {

                //alle Element.Attribut in line durch aktuellen Wert ersetzen sowie
                   double Werte umformatieren
                for (int i = 0; i < matchList.Count; i++)
                {
                    string attributeValue =
                            getFirstAttributeValueByPath(matchList[i].Value);
                    line = line.Replace(matchList[i].Value, attributeValue);
                    if (!line.Contains("."))
                    {
                        break;
                    }
```

```
                }
            }
        }

        //line an Scripting übergeben und berechnen lassen

        bool decision = false;

        line = line.Replace("False", "false");
        line = line.Replace("True", "true");
        line = line.Replace(" == ", " = ");
        line = line.Replace("false", "0 ");
        line = line.Replace("true", "1 ");

        string dec = scriptingCalculation(line);

        decision = (dec.Equals(true.ToString(),
                    StringComparison.CurrentCultureIgnoreCase));

        return decision;
    }
    throw new Exception(Resources.exErrorInDecision);
}
```

Codegenerierung bei Entscheidungsknoten




```
public virtual void executeProgram() {
    Code in Aktion 1;
    for (
    ; ;
    ) {
        Thread.Sleep(10);
        Application.DoEvents();
        for (
        ; Bedingung 1A;
        ) {
            Code in Aktion 2;
            Thread.Sleep(10);
            Application.DoEvents();
            if (Bedingung 1B) {
                Code in Aktion 3;
                goto Finish;
            }
            else {
                if (Bedingung 2B) {
                Code in Aktion 4;
                }
            }
        }
        if (Bedingung 2A) {
            break;
        }
```

```
    }
    if (Bedingung 2A) {
        Code in Aktion 5;
        if (Bedingung 1C) {
            Code in Aktion 6;

            goto Finish;
        }
        else {
            if (Bedingung 2C) {
                Code in Aktion 7;
                goto Finish;
            }
        }
        goto Finish;
    }
Finish:
;
}
```

Funktion *initializeCompass*

```
private void initializeCompass()
{
    createVariable("Compass", compassObject);
    if (!variableList.Contains("CompassDegree"))
    {
        degree = createVariable2("double", "CompassDegree");
    }
    constructor.Statements.Add(new CodeSnippetExpression(compassObject + " = new
                                                         Compass()"));
    constructor.Statements.Add(new CodeSnippetExpression(compassObject +
    ".CompassPosReceived += new CompassPosReceivedEventHandler(" + compassObject +
                                                         "_CompassPosReceived)"));
    constructor.Statements.Add(new CodeSnippetExpression(compassObject +
    ".startTimer()"));
    MethodC1CompassPosReceived();


    //Methode get_Compass_Degree
    CodeMemberMethod getCompassDegree = createMethod("getCompassDegree");
    getCompassDegree.ReturnType = new CodeTypeReference("System.Double");
    getCompassDegree.Statements.Add(new CodeMethodReturnStatement(new
    CodeArgumentReferenceExpression("CompassDegree")));
}
```

Erzeugter Code des Parkvorgangs

```
using System;
using System.IO;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Windows.Forms;
using LL.EA.UMLExecution.Properties;

namespace LL.EA.UMLExecution {


    public class CarControl {

        private bool SensorsRight = true;

        private double SensorsSignalChangeRight = 0;

        private int VehicleDriveDirection = 0;

        private int VehicleSpeed = 0;

        private int VehicleDriveTime = 0;

        private Vehicle v1;

        private Sensors s1;
```

```
private bool SensorsFront = true;

public CarControl() {
    v1 = new Vehicle();
    s1 = new Sensors();
    s1.SensorsPosReceived += new
                 SensorsPosReceivedEventHandler(s1_SensorsPosReceived);
    s1.startTimer();
}

public virtual void executeProgram() {
    SensorsRight = true;
    SensorsSignalChangeRight = 0;
    VehicleDriveDirection = 0;
    VehicleSpeed = 0;
    VehicleDriveTime = 0;
    setCarValuesAndDrive();
    VehicleSpeed = 2;
    setCarValuesAndDrive();
    for (
    ; ;
    ) {
        Thread.Sleep(10);
        Application.DoEvents();
        for (
        ; SensorsRight == true;
        ) {
            Thread.Sleep(10);
            Application.DoEvents();
        }
        for (
        ; SensorsRight == false;
        ) {
            Thread.Sleep(10);
            Application.DoEvents();
            if ((SensorsSignalChangeRight * VehicleSpeed) > 1.5) {
                VehicleSpeed = 2;
                VehicleDriveTime = 1;
                setCarValuesAndDrive();
                VehicleSpeed = 0;
                setCarValuesAndDrive();
                VehicleSpeed = -2;
                VehicleDriveDirection = 4;
                VehicleDriveTime = 3;
                setCarValuesAndDrive();
                VehicleDriveDirection = -4;
                VehicleSpeed = -2;
                VehicleDriveTime = 3;
                setCarValuesAndDrive();
                VehicleDriveDirection = 0;
                VehicleSpeed = 0;
                setCarValuesAndDrive();
                VehicleSpeed = 2;
                VehicleDriveTime = 1;
                setCarValuesAndDrive();
                VehicleSpeed = 0;
                setCarValuesAndDrive();
                goto Finish;
            }
            else {
                if ((SensorsSignalChangeRight * VehicleSpeed) <= 1.5) {
                }
            }
        }
    }
Finish:
;
}

public virtual void setCarValuesAndDrive() {
    v1.Direction = VehicleDriveDirection;
    v1.Speed = VehicleSpeed;
    v1.Time = VehicleDriveTime;
    v1.drive();
}

public virtual int getActualSpeed() {
    return VehicleSpeed;
```

```
        }

        public virtual int getActualDirection() {
            return VehicleDriveDirection;
        }

        public virtual int getActualDriveTime() {
            return VehicleDriveTime;
        }

        public virtual double getCompassDegree() {
            return 0;
        }

        public virtual void s1_SensorsPosReceived(object sender,
                                                  SensorsPosReceivedEventArgs e) {
            SensorsFront = e.FrontSensor;
            SensorsRight = e.RightSensor;
        }

        public virtual bool getSensorsFront() {
            return SensorsFront;
        }

        public virtual bool getSensorsRight() {
            return SensorsRight;
        }
    }
}
```

Stichwortverzeichnis

Literaturverzeichnis

[ARGM05] Ralf Anske, Maria Gensel: Automatisches Einparken, April 2005

[BPLS05] Paul Baker, Shiou Loh, Frank Weil: Model-Driven Engineering in a Large Industrial Context-Motorola Case Study. In Linonel C. Briand and Clay Williams, editors, Proc. Of the 8th Int. Conference on Model-Driven Engineering Languages and Systems (MoDELS), volume 3713 of LNCS, pages 476-491, Montego Bay (Jamaica), Oktober 2005

[CG04] Galvao Cavalcanti: Executable UML, Department Accounting, HANKEN-Swedish School of Economics and Business Administration, März 2004

[FLMJ08] Fuentes Lidia, Manrique Jorge, Sanchez Pablo: Execution and Si mulation of (Profiled) UML Models using Pópulo, Dpto. Lenguajes y Ciencias de la Computación, Universidad de Málaga (Spain), Mai 2008

[GCKP07] Christian Glodt, Pierre Kelsen, Elke Pulvermueller: DEMOCLES: A Tool for Executable Modeling of Platform-Independent Systems, OOPSLA ’07, Montreal, Quebec, Canada, Oktober 2007

[GEHR94] Erich Gemma, Richard Helm, Ralph Johnson and John Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, October 1994

[GV04] Volker Gruhn: Agile Model Driven Architecture, Lehrstuhl für Angewandte Telematik/e-Business, Universität Leipzig, 2004

[KAMD06] Andrei Kirshin, Dany Moshkovich, Alan Hartman: A UML Simulator Based On A Generic Model Execution Engine, IBM Haifa Research Lab, Israel, 2006

[KH02] Jernej Kovse und Theo Härder: Generic XMI-Based UML Model Transformations, University of Kaiserslautern, 2002

[Kü07] Stefan Kühne: Modellgetriebene Entwicklung im Kontext Integration Engineering, Universität Leipzig, 2007

[Kü07] Stefan Kühne: Modellgetriebene Entwicklung im Kontext Integration Engineering, Universität Leipzig, 2007

[LM07] Monica Lozano, Maria Martin: UML 2.0, Department of Computer Science and Electronics, Mälardalen University, Sweden, June 2007

[NS04] Scott Niemann: Executable Systems Design with UML 2.0, Andover, MA, 2004

[OALM02] Object Action Language Manual, Version 1.4, http://www.ooatool.com/docs/OAL02.pdf, 2002

[OMGD06] OMG Diagram Interchange, Version1.0, http://www.omg.org/docs/formal/06-04-04.pdf, April 2006

[OMGI07] OMG Unified Modeling Language (OMG UML), Infrastructure, V2.1.2 (without change bars), http://www.omg.org/docs/formal/07-11-04.pdf, November 2007

[OMGM03] Joaquin Miller and Jishnu Mukerji: MDA Guide Version 1.0.1, http://www.omg.org/docs/omg/03-06-01.pdf, June 2003

[OMGO06] OMG Object Constraint Language Specification, Version2.0, http://www.omg.org/docs/formal/06-05-01.pdf, May 2006

[OMGS07] OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2 (without change bars), http://www.omg.org/docs/formal/07-11-02.pdf, November 2007

[OMGM07] OMG MOF 2.0 / XMI Mapping, Version 2.1.1 (without change bars), http://www.omg.org/docs/formal/07-12-01.pdf, Dezember 2007

[RCFP03] Chris Raistrick, Paul Francis, John Wright, Colin Carter, Ian Wilkie: Model Driven Architecture with UML, März 2004, ISBN-10: 0521537711

[RDFS01] Dirk Riehle, Steven Fraleigh, Dirk Bucka-Lassen and Nosa Omorogbe: The Architecture of a UML Virtual Machine. In Proceedings pf the 2001 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA ’01). ACM Press, 2001

[RJSM04] Julia Reuter, Matthias Seitz: Programmiermöglichkeiten für das Microsoft .NET Framework mittels Unmanaged Code – Konkrete Lösungen am Beispiel Object REXX, Universität Augsburg, 2004

[SIVJ07] Igor Sacevski, Jadranka Veseli: Introduction to Model Driven Architecture (MDA), Juni 2007

[SN02] Selo Sulistyo, Warsun Najib: Executable UML, Dept of Information and Communications Technology Agder University College, Norway, 2002

[ST04] Thomas Springer: Ein komponentenbasiertes Meta-Modell kontextabhängiger Adaptionsgraphen für mobile und ubiquitäre Anwendungen, Technische Universität Dresden, Fakultät für Informatik, Oktober 2004

[ST07] Thomas Schuster: Modellgetriebene Entwicklung von Benutzerschnittstellen, Juni 2007

[VJ07] Johannes Vetter: Model-Driven Architecture Konzeption, Einordnung und Wirtschaftlichkeit, Oktober 2007

[WC06] Constantin Werner: UML Profile for Communicationg Systems, Mathematisch-Naturwissenschaftliche Fakultät, Georg August Universität, Göttingen, 2006

[WM07] Michael Wilk: Analyse und Bewertung der Umsetzung des Model-Driven-Development-Ansatzes durch SAP NetWeaver Visual Composer, Institut für Wirtschaftsinformatik, Universität Hamburg, November 2007

[www-01] OMG-Seite zu MDA: http://www.omg.org/mda/ , April 2008

[www-02] Modellgetriebene Softwareentwicklung: Eine Einführung, http://www.form4.de/technologie/uml-und-mda/modellgetriebene-softwareentwicklung/modelle-und-metamodelle/

[www-03] OMG-Seite: UML Profile, http://www.omg.org/technology/documents/profile_catalog.htm, September, 2008

[www-04] User Manual SSC-32, Version 2.0: http://www.lynxmotion.com/images/data/ssc-32.pdf

[www-05] UML Werkzeuge: http://www.jeckle.de/umltools.html, Juni 2004