Gottfried Wilhelm
Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Praktische Informatik
Fachgebiet Software Engineering

Ein Framework für die Definition von 3D-Editoren für Visualisierungen modellbasierter Simulationen

Bachelorarbeit

im Studiengang Informatik

von

Joey-Timon Schnurre

Prüfer: Prof. Dr. Joel Greenyer
Zweitprüfer: Prof. Dr. Kurt Schneider
Betreuer: Prof. Dr. Joel Greenyer,
Dipl.-Inform. Daniel Gritzner

Hannover, 28.09.2015

Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 28.09.2015

Joey-Timon Schnurre

Zusammenfassung

Modellbasierte Simulationen können in vielen Bereichen eingesetzt werden, um das Verhalten dynamischer Systeme wie etwa automatisierten Fabriksystemen zu untersuchen. Um solche Simulationen zu veranschaulichen und mit den Objekten während einer Simulation zu interagieren, ist eine interaktive 3D-Darstellung dieser Simulation hilfreich. Für jede Domäne einen eigenen Editor für solche 3D-Darstellungen zu entwickeln ist jedoch sehr kostspielig und oft unwirtschaftlich.

Aus diesem Grund entwerfe ich in dieser Arbeit ein Framework, mit dem zu beliebigen domänenspezifischen Modellen Editoren definiert werden können. Diese ermöglichen es, Instanzen dieser Modelle in einer 3D-Umgebung zu visualisieren und diese Visualisuerungen anzupassen. Außerdem implementiere ich einen Prototypen dieses Frameworks, in dem die wichtigsten Features umgesetzt sind.

Abstract

Model-based simulations can be used to examine the behavior of dynamic systems (e.g. automated factories) in many different fields. Interactive 3D visualizations are a good way to illustrate these simulations and interact with objects during a simulation. However, developing an editor for 3D visualizations for each domain is expensive and often not economical.

Therefore, I design a framework for defining editors for arbitrary domain-specific models in this thesis. These editors can be used to create and customize visualizations of instances of those models in a 3D environment. Furthermore, I implement a prototype of this framework where the most important features are realized.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Motivation . . . 1
1.2 Ziel der Arbeit . . . 1
1.3 Struktur der Arbeit . . . 2

2 Grundlagen **3**
2.1 EMF und ECore . . . 3
2.2 Konzepte der 3D-Computergrafik . . . 4
2.2.1 Weltkoordinatensystem und Objektkoordinatensystem 4
2.2.2 Meshes . . . 5
2.2.3 Transformationen . . . 5
2.2.4 Scene Graph . . . 5
2.3 Verwandte Arbeiten . . . 6
2.3.1 Sirius . . . 6

3 Problemanalyse **9**
3.1 Beispiele . . . 9
3.1.1 Fabrikroboter . . . 9
3.1.2 Autoturm . . . 10
3.2 Anwendungsmöglichkeiten . . . 11
3.2.1 Erstellung eines Instanzmodells mit 3D-Darstellung . . 11
3.2.2 Erstellung einer 3D-Darstellung zu einem existierenden Instanzmodell . . . 11
3.2.3 Speichern von Instanzmodellen und ihrer 3D-Repräsentation . . . 12
3.3 Analyse verschiedener räumlicher Relationen . . . 12
3.3.1 Ankerpunktrelationen . . . 12
3.3.2 Relationen mit komplexeren Ankerobjekten . . . 14
3.3.3 Magnet-Relationen . . . 14
3.3.4 Weitere Relationen . . . 15
3.4 Abgrenzung . . . 15

4 Lösungskonzept **17**
4.1 Allgemeine Beschreibung 17
4.2 Mapping-Editor 18
4.3 3D-Editor 21
4.3.1 Erstellen, Laden und Speichern von Modellen 21
4.3.2 3D-Ansicht 22

5 Technische Details **25**
5.1 Struktur der Software 25
5.2 Mapping-, Representation- und Editor-Modell 26
5.2.1 Representation-Modell 26
5.2.2 Mapping-Modell und Zusammenhang zu Representation-Modell 27
5.3 Verwaltungsklassen 27
5.4 Interne Klassen 28
5.5 Swing-GUI 29

6 Evaluation **31**

7 Zusammenfassung und Ausblick **33**
7.1 Zusammenfassung 33
7.2 Ausblick 33
7.2.1 Verbesserungen am Editor 33
7.2.2 Erweiterungen des Frameworks 35

A Klassendiagramme **37**

Kapitel 1

Einleitung

1.1 Motivation

In vielen Bereichen, wie zum Beispiel der Industrie, sind dynamische Systeme zu finden. Ein solches System kann beispielsweise eine Fabrik sein, in der Roboter Gegenstände zwischen Lagerregalen und Fertigungsstationen transportieren. Zur Untersuchung und Veranschaulichung des Verhaltens dynamischer Systeme eignet sich der Einsatz modellbasierter Simulationen. In einer modellbasierten Simulation wird jeder Teil des Systems durch ein Objekt modelliert. Interaktionen zwischen verschiedenen Teilen des Systems werden durch Interaktionen zwischen diesen Objekten modelliert. Eine modellbasierte Simulation kann unter anderem eingesetzt werden, um verschiedene Anordnungen von Robotern, Fertigungsstationen und Lagerregalen zu untersuchen oder um verschiedene Verhaltensweisen der Roboter zu testen.

Zur Veranschaulichung modellbasierter Simulationen ist es oft sinnvoll, eine interaktive 3D-Darstellung zu erstellen. Diese Darstellung kann es dem Anwender erleichtern, die Simulationsobjekte zu erstellen, zu beobachten und sie während der Simulation zu beeinflussen. Zudem eignen sich 3D-Darstellungen gut zur Präsentation von Ergebnissen wie einem neuen Algorithmus für Fabrikroboter.

Zur Zeit müssen zur Erstellung derartiger 3D-Darstellungen Frameworks benutzt werden, die für jeden Anwendungsbereich einzeln entwickelt werden. Da die Entwicklung solcher Frameworks kostspielig ist und nicht zu jedem Anwendungsbereich bereits ein solches Framework existiert, ist die Erstellung interaktiver 3D-Darstellungen oftmals wirtschaftlich nicht realisierbar.

1.2 Ziel der Arbeit

Um das Problem der Wirtschaftlichkeit von 3D-Darstellungen für modellbasierte Simulationen zu lösen, konzipiere ich in dieser Arbeit ein domänen-

unabhängiges Framework, mit dem ohne Programmieraufwand zu beliebigen existierenden Modellen domänenspezifische Editoren erstellt werden können. Diese können dann zur Erstellung von 3D-Visualisierungen von Simulationen genutzt werden. Dabei soll es unter anderem die Möglichkeit geben, jeder Klasse ein entsprechendes 3D-Modell und andere visuelle Eigenschaften, wie Lichteffekte, zuzuordnen ("graphical mapping"). Für Assoziationen sollen Visualisierungen (z.B. eine gepunktete Linie, die zwischen den involvierten Objekten gezeichnet wird) und räumliche Bedingungen (z.B. ist-auf-Beziehung) festgelegt werden können. Die so spezifizierten Editoren sollen Anwendern die Möglichkeit geben, Objekte des jeweiligen Modells zu erstellen, im Raum anzuordnen und zu verändern. Ein solches Framework ermöglicht es die Benutzung von 3D-Darstellungen in Szenarien, in denen ein speziell entwickeltes 3D-Framework nicht wirtschaftlich wäre.

Des Weiteren implementiere ich einen Prototypen dieses Frameworks, in dem die wichtigsten Features vereinfacht umgesetzt sind.

Nicht Teil dieser Arbeit ist die Visualisierung der Objekte und die Interaktion mit den Objekten während der Simulation – in dieser Arbeit beschränke ich mich auf die Spezifikation der Editoren und die Erstellung der 3D-Visualisierungen mit diesen Editoren.

1.3 Struktur der Arbeit

Diese Arbeit ist wie folgt strukturiert:

Grundlegende Konzepte und Technologien, auf denen diese Arbeit aufbaut, erläutere ich in Kapitel 2. Außerdem wird in diesem Kapitel ein kurzer Überblick über verwandte Arbeiten gegeben.

In Kapitel 3 analysiere ich anhand von Beispielen die Probleme, die das Framework lösen können soll. Insbesondere werden hier verschiedene räumliche Beziehungen zwischen Objekten und mögliche Implementierungen dieser Beziehungen betrachtet.

Das Framework und den implementierten Prototypen stelle ich in Kapitel 4 als Lösungskonzept vor. Die technischen Details des Frameworks und der prototypischen Implementierung werden in Kapitel 5 näher beleuchtet.

In Kapitel 6 reflektiere ich darüber, wie gut das in dieser Arbeit konzipierte Framework die Problemstellung löst und wie gut die Anforderungen durch das Framework erfüllt werden.

Kapitel 7 enthält eine kurze Zusammenfassung der Arbeit und einen Ausblick auf künftige Arbeiten, die auf dieser Arbeit aufbauen können.

Kapitel 2

Grundlagen

In diesem Kapitel erläutere ich grundlegende Konzepte und Begriffe, deren Kenntnis für die Verständlichkeit dieser Arbeit erforderlich ist. Dabei stelle ich zunächst kurz das Eclipse Modeling Framework (EMF) vor, wobei ich mich hauptsächlich auf ECore konzentriere. Im nächsten Abschnitt erläutere ich einige wichtige Konzepte der 3D-Computergrafik. Zum Schluss gebe ich einen Überblick über verwandte Arbeiten.

2.1 EMF und ECore

EMF (Eclipse Modeling Framework) ist ein Framework für objektorientierte Modellierung. Wichtigster Teil von EMF ist ECore, ein Metamodell zur Spezifikation von Modellen. Ähnlich wie UML enthält es unter anderem Definitionen für Klassen, Methoden, Attribute, Objekte.

Mit diesem Framework lassen sich ECore-Klassenmodelle erstellen, in denen eigene Klassen definiert werden können. Solche Modelle sind dabei Instanzen des ECore-Metamodells.

Zum Beispiel wird bei der Definition einer Klasse *Robot* mit dem Attribut *name* tatsächlich ein Objekt vom Typ *EClass*, welches ein Objekt vom Typ *EAttribute* enthält, erstellt.

Von diesen selbst definierten Klassen lassen sich außerdem Instanzen erstellen, die dann vom Typ *EObject* sind. *EClass*, *EAttribute* und *EObject* sind Klassen, die im ECore-Metamodell definiert sind. Abbildung 2.1 zeigt einen stark vereinfachten Auszug des ECore-Metamodells, ein mit diesem Metamodell definiertes ECore-Klassenmodell und ein Instanzmodell dieses ECore-Klassenmodells.

EMF stellt die Tools zur Verfügung, mit denen ECore-Klassenmodelle erstellt und editiert werden können. Außerdem bietet EMF die Möglichkeit, aus benutzerdefinierten ECore-Klassenmodellen Code und einfache Editoren für Instanzen dieser Modelle zu generieren. Der generierte Code erlaubt es, die in benutzerdefinierten ECore-Klassenmodellen spezifizierten Klassen

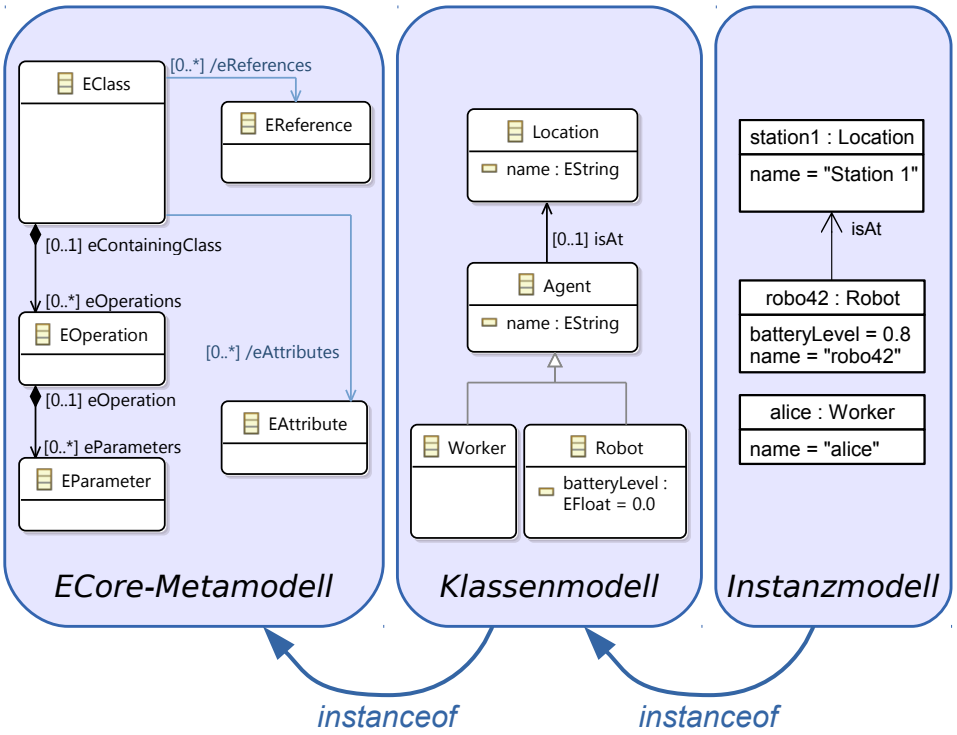

Abbildung 2.1: Auszug aus dem ECore-Metamodell, einem ECore-Klassenmodell und einem Instanzmodell dieses ECore-Klassenmodells.

direkt zu benutzen und Instanzen der Klassen zu erstellen, ohne den Code für diese Klassen selbst zu schreiben.

2.2 Konzepte der 3D-Computergrafik

2.2.1 Weltkoordinatensystem und Objektkoordinatensystem

Um die Positionen der Objekte in der Welt definieren zu können, wird üblicherweise ein dreidimensionales kartesisches Weltkoordinatensystem benutzt. So kann jeder Punkt im dreidimensionalen Raum durch einen Koordinatenvektor repräsentiert werden.

Unabhängig vom Weltkoordinatensystem wird außerdem für jedes Objekt ein eigenes Objektkoordinatensystem definiert, welches seinen Ursprung üblicherweise innerhalb des Objekts hat. Das Objektkoordinatensystem eines Objekts kann gegenüber dem Weltkoordinatensystem beliebig verschoben, rotiert und skaliert sein.

2.2.2 Meshes

Dreidimensionale Objekte werden in der Computergrafik meistens als *Mesh* dargestellt. Ein Mesh besteht aus einer Menge von Polygonen, also abgegrenzten zweidimensionalen Flächen wie z.B. Dreiecken. Definiert sind diese Polygone durch die Koordinaten ihrer Eckpunkte im Objektkoordinatensystem des Meshes. Diese Polygone sind im Raum so angeordnet, dass sie zusammen die Oberfläche des dreidimensionalen Objekts bilden, welches angezeigt werden soll.

2.2.3 Transformationen

Transformationen sind Funktionen, die jeden Vektor und Punkt im dreidimensionalen Raum auf einen anderen Vektor/Punkt abbilden. Die in dieser Arbeit relevanten Transformationen sind affine Transformationen – also solche, mit denen beliebige Kombinationen aus Drehungen, Skalierungen und Translationen durchgeführt werden können. Diese Transformationen können auf Vektoren sowie auf komplexere Objekte wie Meshes angewendet werden.

Derartige Transformationen werden benötigt, um Objekte innerhalb des dreidimensionalen Raumes (des Weltkoordinatensystems) zu bewegen und anzuordnen. Dazu wird die Transformation auf das Objektkoordinatensystem angewendet. Das Objektkoordinatensystem wird durch die Transformation also verschoben, rotiert und/oder skaliert. Da die Bestandteile des Objekts in diesem Objektkoordinatensystem definiert sind, wird das Objekt ebenfalls verschoben, rotiert bzw. skaliert.

2.2.4 Scene Graph

Der Scene Graph ist ein gerichteter Baum, der eine hierarchische Anordnung aller in der 3D-Ansicht angezeigten Objekte definiert. Jedes angezeigte Objekt ist ein Knoten im Scene Graph. Es können jedoch auch Knoten existieren, die keine sichtbaren Objekte sind. Jedem Knoten ist außerdem eine affine Transformation zugeordnet, welche die Position, Rotation und Skalierung des Objektkoordinatensystems des Knotens in Bezug zum Elternknoten angibt. Es existiert immer ein Wurzelknoten, der den Nullpunkt des Weltkoordinatensystems repräsentiert. Dieser wird benötigt, damit die Objektkoordinatensysteme aller Knoten im Scene Graph eine definierte Position im Weltkoordinatensystem haben.

Abbildung 2.2 zeigt ein Beispiel für einen solchen Scene Graph. Das Objekt kPferd wird durch die Transformation tPferd nach links verschoben; das Objekt kHindernis wird nach rechts verschoben. Da beide Knoten Kinder des Wurzelknotens sind, wird diese Verschiebung vom Nullpunkt des Weltkoordinatensystems aus ausgeführt – das Pferd ist demzufolge an Position (-5, 0, 0) und das Hindernis ist an Position (5, 0, 0).

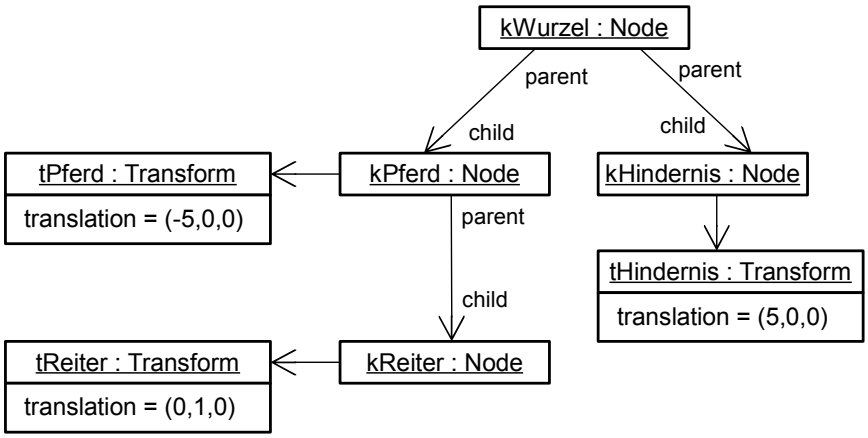

Abbildung 2.2: Beispiel für einen Scene Graph

Der Knoten kReiter ist Kind des Knotens kPferd – die Transformation von kReiter wird daher im Objektkoordinatensystem des Knotens kPferd ausgeführt. Die Position des Reiters ist somit (-5, 1, 0). Um die Transformation des Reiters zu ermitteln, werden die Transformationen aller Knoten im Zweig von kReiter bis zum Wurzelknoten hintereinander ausgeführt – also erst die Transformation von kReiter und dann die Transformation von kPferd. Dies hat zur Folge, dass die Position des Reiters in Relation zum Objektkoordinatensystem des Pferdes definiert ist anstatt in Relation zum Weltkoordinatensystem. Will man das Pferd mitsamt dem Reiter rotieren oder skalieren, muss lediglich die Transformation des Knotens kPferd geändert werden und die Position, Rotation und Skalierung des Reiters wird automatisch so angepasst, dass der Reiter immer auf dem Pferd sitzt.

2.3 Verwandte Arbeiten

2.3.1 Sirius

Sirius [4] ist ein auf Eclipse aufbauendes Framework zur Erstellung eigener Modellierungswerkzeuge und zweidimensionalen Visualisierungen von Modellen. Mit einem Baum-Editor können zu einem beliebigen ECore-Klassenmodell Visualisierungen und Werkzeuge spezifiziert werden. Diese können anschließend genutzt werden, um in einer grafischen Umgebung Instanzen des ECore-Klassenmodells zu erstellen und die Visualisierung anzupassen.

Damit löst Sirius das Problem, dass die Entwicklung domänenspezifischer Modellierungstools aufwändig und damit oft unwirtschaftlich ist. Diese Problemstellung ist sehr ähnlich zur Problemstellung dieser Arbeit, jedoch ist Sirius nicht auf die Visualisierung von Simulationen und die

Interaktion mit Objekten während einer Simulation ausgelegt. Bei Sirius liegt außerdem neben der Visualisierung ein Schwerpunkt bei der Spezifikation von Werkzeugen, während in dieser Arbeit der Fokus ausschließlich auf der Visualisierung liegt. Des Weiteren arbeitet Sirius im Gegensatz zu dieser Arbeit mit zweidimensionalen Visualisierungen.

Kapitel 3

Problemanalyse

In diesem Kapitel erläutere ich zunächst zwei Modelle, die ich in dieser Arbeit als Beispielmodelle benutze. Anschließend werden kurz die wichtigsten Anwendungsfälle für das Framework beschrieben. Außerdem werden verschiedene Arten räumlicher Zusammenhänge zwischen Objekten untersucht.

3.1 Beispiele

Im Folgenden definiere ich die Beispielmodelle, anhand derer ich in dieser Arbeit verschiedene Problemstellungen erläutern werde.

3.1.1 Fabrikroboter

Dieses Modell ist angelehnt an das *FactorySystem*-Modell aus [7]. Modelliert ist eine Fabrik *(Factory)* mit Lagerregalen *(Warehouse)* und Fertigungsstationen *(AssemblyStation)*, in der Roboter *(Robot)* und menschliche Arbeiter *(Worker)* Gegenstände *(Item)* zwischen Lagerregalen und Fertigungsstationen transportieren.

Um beispielsweise ein Gehäuse und ein LCD-Panel zu einem fertigen Monitor zu montieren, muss zunächst ein Roboter oder ein Arbeiter aus dem ersten Lager ein Gehäuse und danach aus dem zweiten Lager ein LCD-Panel zur Fertigungsstation bringen. Anschließend setzt die Fertigungsstation die beiden Komponenten zu einem Monitor zusammen. Dieser kann nun von einem weiteren Roboter oder Arbeiter in ein Lager gebracht werden.

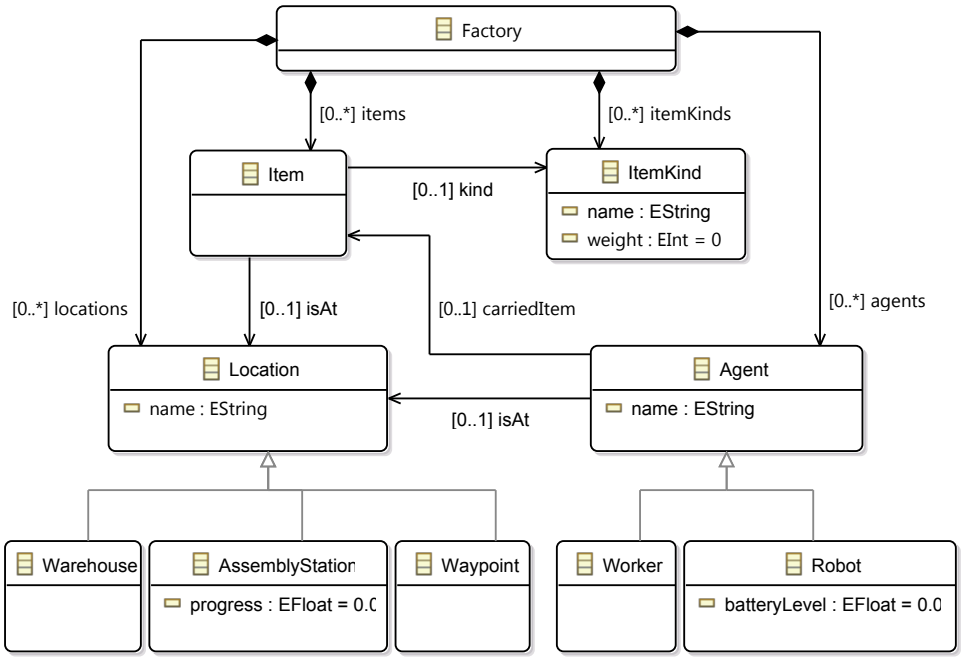

Abbildung 3.1: Fabrikroboter-Beispielmodell

3.1.2 Autoturm

Dieses Beispiel ist ein Modell eines Autoturms ähnlich der VW-Autotürme in Wolfsburg [1]. Ein Autoturm *(CarTower)* kann dabei mehrere Aufzüge *(Elevator)* haben, die jeweils eine Menge von Plattformen *(ElevatorPlatform)* haben. Durch Hoch- und Runterfahren können diese Plattformen verschiedene Sektionen *(ElevatorSection)* ansteuern. Jede dieser Sektionen besteht aus nebeneinander gelegenen Stellplätzen *(CarSlot)*, von denen jeder ein Auto *(Car)* aufnehmen kann. Die Plattformen können diese Stellplätze durch eine Drehbewegung erreichen.

Derartige Autotürme werden in der Regel zur Zwischenlagerung frisch produzierter Autos, die später vom Kunden vor Ort abgeholt werden, genutzt. Ein neu produziertes Auto wird über ein (hier der Einfachheit halber nicht modelliertes) Förderbandsystem von der Fabrik direkt auf eine Plattform im Erdgeschoss eines Autoturms befördert. Diese Plattform fährt nun zu einem noch freien Stellplatz, um das Auto dort hineinzuschieben. Möchte ein Kunde sein Auto abholen, fährt die Plattform zum Stellplatz, in dem sich das Auto befindet, zieht das Auto auf die Plattform und fährt ins Erdgeschoss. Von dort aus transportiert ein weiteres Förderband das Auto zum Abholbereich.

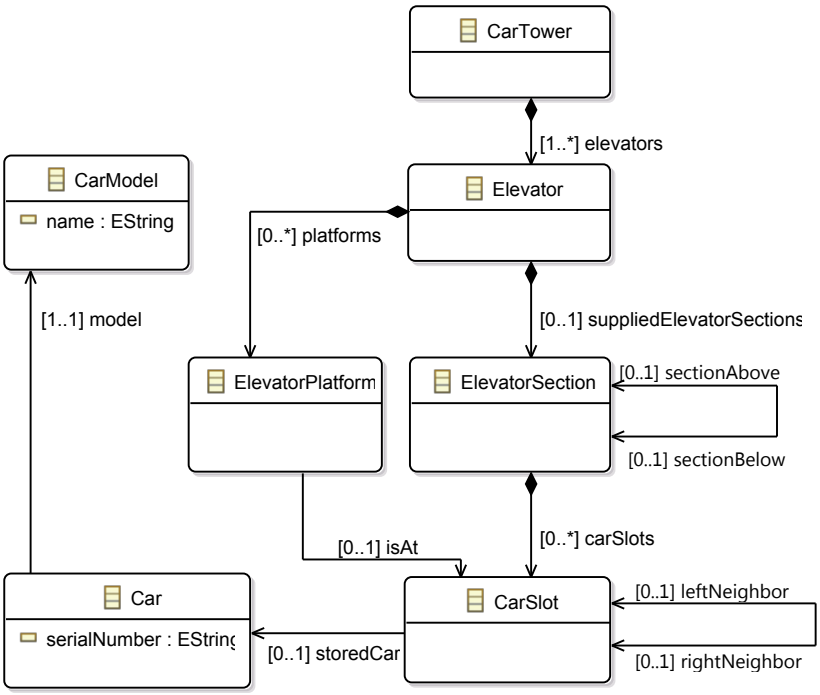

Abbildung 3.2: Autoturm-Beispielmodell

3.2 Anwendungsmöglichkeiten

3.2.1 Erstellung eines Instanzmodells mit 3D-Darstellung

Einer der zentralen Anwendungsfälle für das 3D-Framework ist die Erstellung eines Instanzmodells zu einem gegebenen ECore-Klassenmodell. Dabei spezifiziert der Anwender zunächst das Mapping-Modell. Im Mapping-Modell wird festgelegt, wie Objekte jeder Klasse im ECore-Klassenmodell dargestellt werden sollen. Dazu zählt auch, welche Relationen in der 3D-Darstellung durch welche räumlichen Relationen abgebildet werden sollen.

Nachdem das Mapping-Modell spezifiziert ist, kann der Anwender Objekte des ECore-Klassenmodells, zu denen ein Mapping-Modell definiert wurde, in einem 3D-Editor erstellen und im Raum anordnen. Außerdem soll es dem Anwender möglich sein, die Eigenschaften der Objekte zu verändern, Relationen zwischen den erstellten Objekten zu anzulegen und Objekte zu löschen.

3.2.2 Erstellung einer 3D-Darstellung zu einem existierenden Instanzmodell

Oft ist bereits ein Instanzmodell vorhanden, zu dem man eine 3D-Darstellung entwerfen möchte. Ein weiterer wichtiger Anwendungsfall ist daher das

Laden eines existierenden Instanzmodells, welches dann wie ein neu erstelltes Instanzmodell bearbeitet werden kann. Um ein Instanzmodell zu laden, muss zunächst das Mapping-Modell definiert sein. Die geladenen Objekte werden in der 3D-Ansicht automatisch gemäß der im Mapping-Modell definierten räumlichen Relationen angeordnet. Lädt man beispielsweise ein Objekt vom Typ *AssemblyStation* und ein Objekt vom Typ *Item*, welches über die *isAt*-Relation mit der *AssemblyStation* verbunden ist, kann das *Item* in der 3D-Darstellung automatisch auf die *AssemblyStation* gelegt werden.

Das so geladene Instanzmodell kann anschließend wie ein neu erstelltes Instanzmodell im 3D-Editor bearbeitet werden.

3.2.3 Speichern von Instanzmodellen und ihrer 3D-Repräsentation

Um das mit dem 3D-Editor erstellte bzw. bearbeitete Instanzmodell später weiter benutzen zu können, muss sowohl das Instanzmodell als auch die 3D-Repräsentation gespeichert werden. Instanzmodell und 3D-Repräsentation werden dabei zusammen in eine Datei gespeichert, die später wieder mit dem 3D-Editor geöffnet werden kann. Damit das Instanzmodell auch von anderer Software unabhängig vom Framework genutzt werden kann, soll es außerdem die Möglichkeit geben, das Instanzmodell in eine separate Datei zu exportieren.

3.3 Analyse verschiedener räumlicher Relationen

Räumliche Relationen sind Relationen zwischen zwei Objekten, die einen Raumbezug haben, wie zum Beispiel die Relation *isAt* der Klasse *Agent* im *FactoryRobots*-Beispielmodell (siehe Abschnitt 3.1.1). Im 3D-Editor können solche Relationen dargestellt werden, indem die involvierten Objekte im Raum automatisch entsprechend angeordnet werden.

3.3.1 Ankerpunktrelationen

Eine Art von räumlichen Relationen sind Beziehungen zwischen zwei Objekten, die einfach die Position und Lage des referenzierten Objekts in Relation zum referenzierenden Objekt definieren. Eine solche Beziehung besteht beispielsweise im *FactoryRobots*-Beispielmodell aus Abschnitt 3.1.2 zwischen einer *AssemblyStation* und einem *Item*, das auf der *AssemblyStation* steht.

Zur Darstellung derartiger Beziehungen habe ich die *Ankerpunktrelation* entwickelt. Zu jedem Objekt können im jeweiligen Objektkoordinatensystem Ankerpunkte definiert werden. Ein Ankerpunkt ist dabei eine affine Transformation, daher kann neben der Position des Ankerpunkts auch eine Rotation und eine Skalierung definiert werden. Für jede Relation zwischen

zwei Objekten ist außerdem für jedes der beiden Objekte festgelegt, welcher Ankerpunkt für diese Relation benutzt wird.

Um nun eine solche Relation darzustellen, werden die Objekte im Raum so angeordnet, dass die Ankerpunkte beider Objekte im Weltkoordinatensystem übereinstimmen. Das führt dazu, dass der Ankerpunkt des ersten Objekts genau auf dem Ankerpunkt des zweiten Objekts liegt und die Rotationen und Skalierungen der beiden Ankerpunkte ebenfalls übereinstimmen.

Im *FactoryRobots*-Modell kann beispielsweise Objekten der Klasse *AssemblyStation* ein Ankerpunkt *agentPosition* zugeordnet werden. Dieser Ankerpunkt spezifiziert die Position des Arbeiters, der momentan an der Fertigungsstation arbeitet. Außerdem kann Objekten der Klasse *Robot* ein Ankerpunkt *standingOn* zugewiesen werden, der sich an der Position des Bodens befindet, auf dem der Roboter steht. Wird nun ein *Robot* über die *isAt*-Relation mit einer *AssemblyStation* verknüpft, ordnen sich die beiden Objekte so an, dass der Roboter an der Fertigungsstation steht.

Die Ankerpunktrelation eignet sich gut, um verschiedene einfache räumliche Relationen darzustellen, bei denen die relative Anordnung der Objekte zueinander genau definiert werden kann. Das ist oft bei Relationen gegeben, bei denen ein Objekt auf oder neben einem anderen Objekt steht. Auch für Beziehungen, bei denen ein Objekt sich in einem anderen Objekt befindet – wie z.B. ein Fahrer, der sich in einem Auto befindet – kann die Ankerpunktrelation ein geeignetes Modell sein.

Weniger geeignet ist die Ankerpunktrelation, wenn die räumliche Beziehung nicht genau definiert werden kann, wie zum Beispiel, wenn ein Objekt einfach nur irgendwo in der Nähe eines anderen Objekts sein soll.

Mehrwertige Relationen lassen sich auf diese Weise ebenfalls nicht gut darstellen, da alle referenzierten Objekte an der gleichen Position angezeigt werden. Dieses Problem tritt etwa bei der *isAt*-Relation zwischen Objekten der Klassen *Item* und *Warehouse* auf. Will man mehrere Items in einem Warehouse abstellen, werden alle Items an der gleichen Position angezeigt. Für mehrwertige Relationen mit nur wenigen Objekten kann das Problem gelöst werden, indem ermöglicht wird, für jede Relation mehrere verschiedene Ankerpunkte zu spezifizieren.

Problematisch ist auch, wenn zwei oder mehr Objekte zyklisch über mehr als eine Relation miteinander in Verbindung stehen. Dies ist unter anderem der Fall, wenn im *CarTower*-Modell ein Ring aus Stellplätzen modelliert wird. In so einer Situation kann es passieren, dass es nicht möglich ist, die Objekte so zu platzieren, dass alle räumlichen Bedingungen erfüllt werden. Dieses Problem kann man minimieren, indem man jeder Ankerpunktrelation ein Gewicht zuordnet und im Falle eines Konflikts die Relation mit dem geringsten Gewicht nicht beachtet.

3.3.2 Relationen mit komplexeren Ankerobjekten

Das Konzept der Ankerpunktrelation kann erweitert werden, indem neben Ankerpunkten auch Strecken, Flächen und Körper als Anker benutzt werden. Wie bei der Ankerpunktrelation können bei der *Ankerobjektrelation* zu jedem Objekt mehrere Strecken, Flächen und Körper definiert werden. Für jede Relation wählt man dann von einem Objekt einen Ankerpunkt und vom anderen Objekt eine oder mehrere Ankerstrecken, Ankerflächen oder Ankerkörper. Um eine solche Relation darzustellen, verschiebt der Editor die Objekte so, dass der Ankerpunkt des einen Objekts in der Ankerstrecke, der Ankerfläche oder dem Ankerkörper des anderen Objekts liegt. Dabei wird versucht, die Objekte so zu platzieren, dass sich diese nicht überschneiden.

Verglichen mit der Ankerpunktrelation bietet diese Art von Relation mehr Flexibilität, da man hier auch räumliche Zusammenhänge beschreiben kann, die nicht punktgenau definiert sind.

Mehrwertige Relationen lassen sich mit Ankerobjektrelationen im Gegensatz zu Ankerpunktrelationen gut darstellen, da nicht mehr alle referenzierten Objekte an einen Ankerpunkt gebunden werden müssen. Bei der oben angesprochenen *isAt*-Relation zwischen *Item* und *Warehouse* kann man für jedes Regalbrett der Lagerregale eine Ankerfläche definieren. Die eingelagerten Gegenstände werden dann vom Editor auf die Regalbretter verteilt und nicht wie bei der Ankerpunktrelation alle auf den gleichen Punkt gestellt.

Die Implementierung von Ankerobjektrelationen ist sehr viel komplizierter als bei Ankerpunktrelationen. Bei Ankerpunktrelationen müssen lediglich die Transformationen der Ankerpunkte kombiniert werden. Wenn komplexere Ankerobjekte involviert sind, gibt es in der Regel viele Lösungen und der Algorithmus muss heuristisch eine gute Lösung finden.

Wie mit Ankerpunktrelationen lassen sich auch mit Ankerobjektrelationen Modelle konstruieren, bei denen nicht alle räumlichen Bedingungen erfüllt werden können. In solchen Fällen kann man wie bei der Ankerpunktrelation vorgeschlagen ebenfalls für jede Relation ein Gewicht definieren und im Konfliktfall Relationen mit niedrigem Gewicht ignorieren.

Bei Konstruktionen aus vielen Objekten und zyklischen Ankerobjektrelationen kann es außerdem schwierig sein, eine Lösung zu finden, auch wenn eine Lösung existiert. In einer derartigen Situation kann es daher sinnvoller zu sein, ebenfalls einige Ankerobjektrelationen zu ignorieren, anstatt zu versuchen, eine Lösung zu finden.

3.3.3 Magnet-Relationen

Eine weitere Variante der Ankerpunktrelation ist die *Magnetrelation*. Ähnlich wie bei der Ankerpunktrelation wird hier für die beiden beteiligten Objekte je eine Menge von Ankerpunkten definiert, die zusammengeführt werden

sollen. Anders als bei den oben genannten Relationen wird hier jedoch nicht versucht, die Ankerpunkte perfekt aufeinanderzulegen. Stattdessen wird innerhalb einer Relation für jeden Ankerpunkt eine Anziehungskraft auf jeden Ankerpunkt des anderen Objekts simuliert. Die Anziehungskraft zwischen zwei Ankerpunkten wird dabei durch eine benutzerdefinierte Funktion in Abhängigkeit zur Entfernung ermittelt. Unter der Annahme, dass jeder Ankerpunkt starr mit dem Objekt verbunden ist, kann nun anhand der auf das Objekt wirkenden Kräfte iterativ eine Bewegung des Objekts simuliert werden. Auf diese Weise richten sich die Objekte zueinander aus.

Die Magnetrelation kann ähnlich wie die Ankerobjektrelation gut bei Relationen angewandt werden, bei denen der Raumbezug nicht punktgenau definiert ist. Autokonvois stellen ein gutes Anwendungsbeispiel für Magnetrelationen dar: Jedes Auto im Konvoi hat zwei Ankerpunkte *front* und *back*. Wenn nun ein Auto dem anderen folgt, kann dies durch eine Magnetrelation zwischen dem hinteren Ankerpunkt des vorderen Autos und dem vorderen Ankerpunkt des hinteren Autos dargestellt werden. Die Anziehungsfunktion sollte dabei so gewählt werden, dass die Anziehungskraft erst ab einer bestimmten Entfernung wirkt.

Denkbar ist ebenfalls, die Magnetrelation mit der Ankerobjektrelation zu kombinieren. In diesem Fall gibt die Ankerobjektrelation vor, welche Anordnungen der beiden involvierten Objekte erlaubt sind und die Magnetrelation sorgt für die genaue Positionierung der beiden Objekte. So kann zum Beispiel beim Autokonvoi sichergestellt werden, dass eine bestimmte Entfernung nicht überschritten wird.

3.3.4 Weitere Relationen

Die oben erläuterten Arten von räumlichen Relationen decken nicht alle Anwendungsfälle ab. Objekte, die sich an anderen Objekten ausrichten, sich dabei aber nicht bewegen, lassen sich beispielsweise nicht mit diesen Relationen modellieren. Dazu zählt zum Beispiel ein Kompass, der immer nach Norden zeigt oder die Augen einer Person, die ein bestimmtes Objekt anschaut. Für solche Fälle ist es erforderlich, weitere Arten von Relationen zu entwerfen und zu implementieren.

3.4 Abgrenzung

In dieser Arbeit beschränke ich mich auf Ankerpunktrelationen, da diese einfach zu implementieren sind und bereits viel Flexibilität bieten. Komplexere Konzepte wie animierte oder dynamisch generierte Meshes (wie z.B. automatisch generierte Rohre in einem Fabrikmodell) sind ebenfalls nicht Teil dieser Arbeit.

Kapitel 4

Lösungskonzept

In diesem Kapitel stelle ich als Lösungskonzept den von mir implementierten Prototypen des Frameworks vor. Auf technische Details gehe ich dabei in diesem Kapitel nicht ein. Zunächst wird ein grober Überblick über das Framework gegeben. Anschließend wird das Mapping und der Mapping-Editor vorgestellt. Zum Schluss wird der 3D-Editor vorgestellt, in dem Instanzmodelle erstellt werden können.

4.1 Allgemeine Beschreibung

In diesem Abschnitt wird zunächst ein Überblick über die Editoren und Modelle gegeben, mit denen das Framework arbeitet.

Das Framework besteht aus zwei Komponenten: dem Mapping-Editor und dem 3D-Editor für Instanzmodelle. Der Mapping-Editor ermöglicht die Erstellung eines *Mapping-Modells* zu einem beliebigen bereits gegebenen ECore-Klassenmodell, wie zum Beispiel dem in Abschnitt 3.1.2 vorgestellten *CarTower*-Beispielmodell. Dieses Mapping-Modell definiert, wie Instanzen von Klassen des gegebenen ECore-Klassenmodells später im 3D-Editor dargestellt werden.

Wenn das Mapping-Modell fertig definiert ist, können mit dem 3D-Editor Instanzmodelle des im Mapping definierten ECore-Klassenmodells erstellt und bearbeitet werden. Mit einem Mapping-Modell zum *CarTower*-Beispielmodell kann man folglich Autotürme erstellen. Informationen über die räumliche Lage und der Visualisierung der so erstellten Objekte werden in einem separaten Modell, dem *Representation-Modell*, gespeichert. Erstellt man zum Beispiel im Editor zwei Autos und platziert diese nebeneinander, werden im Representation-Modell unter anderem die Koordinaten beider Objekte sowie die Darstellung der Objekte als Auto-Mesh gespeichert.

In Abbildung 4.1 ist der Zusammenhang zwischen den vier involvierten Modellen noch einmal graphisch dargestellt. Details zum Mapping-Modell

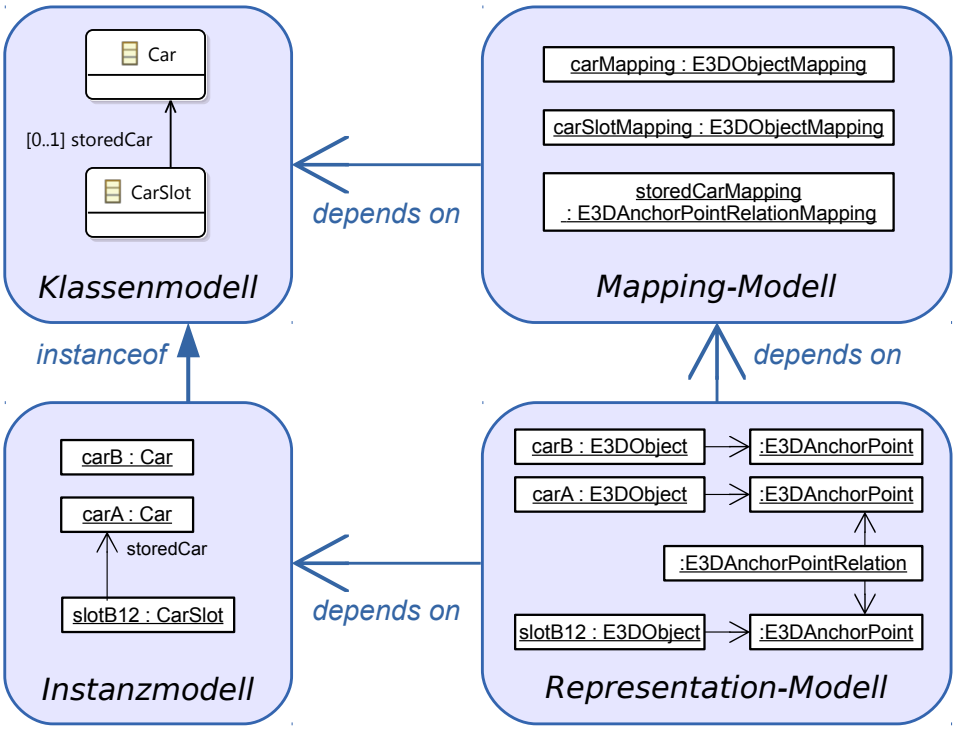

Abbildung 4.1: Übersicht über die vier Modelle, mit denen das Framework arbeitet. Die mit "depends on" beschrifteten Pfeile stellen Abhängigkeitsbeziehungen zwischen den Modellen dar. Das Mapping-Modell hängt beispielsweise vom Klassenmodell ab.

und zum Representation-Modell werden in den folgenden Abschnitten erläutert.

4.2 Mapping-Editor

Der Mapping-Editor dient der Definition von Mapping-Modellen zu einem benutzerdefinierten ECore-Klassenmodell. Das Mapping-Modell legt fest, wie Objekte des Instanzmodells dargestellt werden sollen, es spezifiziert also ein Mapping von Instanzmodell auf Representation-Modell.

Implementiert ist der Mapping-Editor in diesem Prototypen als einfacher, von EMF generierter Baum-Editor, in dem das Mapping-Modell manuell zusammengebaut werden kann. Dieser Baum-Editor zeigt pro Ressource jeweils ein Wurzelobjekt und alle Objekte, die Teil dieses Wurzelobjekts sind, an. Objekte, die Teil eines übergeordneten Objekts sind, werden dabei im Baum als Kinder des übergeordneten Objekts angezeigt. Ein Objekt ist genau dann Teil eines übergeordneten Objekts, wenn eine Kompositionsbeziehung vom übergeordneten Objekt zum betreffenden Objekt besteht.

Abbildung 4.2 zeigt einen Screenshot, in dem gerade ein Mapping-Modell zum *FactoryRobots*-Beispielmodell erstellt wird. Auf der obersten Ebene des Baumes sieht man das Mapping-Modell *(FactoryRobots.e3ddesc)* und das referenzierte ECore-Klassenmodell *(FactoryRobots.ecore)*. Das Klassenmodell wurde dabei aus einer bereits existierenden .ecore-Datei geladen.

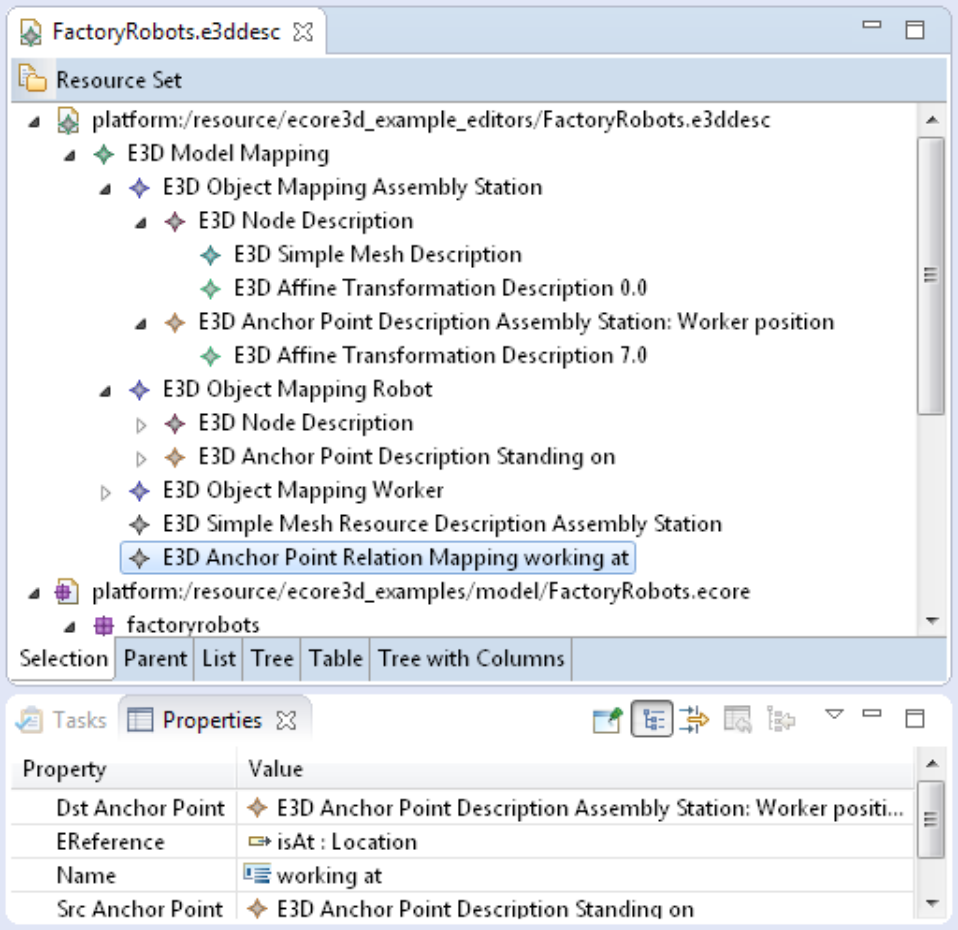

Abbildung 4.2: Screenshot des Mapping-Editors

Das Mapping-Modell, welches in diesem Editor erstellt wird, ist eine Instanz des Mapping-Metamodells. Eine Übersicht über dieses Metamodell findet sich im Anhang A.

Das Wurzelobjekt des Mapping-Modells ist immer vom Typ *E3DModelMapping*. Für jede Klasse, die später im 3D-Editor dargestellt werden soll, enthält dieses Wurzelobjekt ein *E3DObjectMapping*. Im Screenshot sind E3DObjectMappings zu den Klassen *AssemblyStation*, *Robot* und *Worker* enthalten.

Ein *E3DObjectMapping*-Objekt definiert, wie Objekte einer bestimmten Klasse später im 3D-Editor dargestellt werden. Zu diesem Zweck kann man zu jedem *E3DObjectMapping* einen eigenen Scene Graph erstellen, in dem man verschiedene Visualisierungen kombinieren kann. In diesem Beispiel wurde für Objekte vom Typ *AssemblyStation* ein einfacher Scene Graph mit einem Knoten *(E3DNodeDescription)* und einem diesem Knoten zugeordneten Mesh *(E3DSimpleMeshDescription)* konstruiert. Es wäre hier auch möglich, noch weitere Knoten und Meshes zu definieren. Zu jedem Knoten muss dabei eine affine Transformation *(E3DAffineTransformationDescription)* angegeben werden.

Neben dem Scene Graph kann man für jedes *E3DObjectMapping* beliebig viele Ankerpunkte *(E3DAnchorPointMapping)* definieren. Wie bei den Knoten muss man dabei für jeden Ankerpunkt eine affine Transformation angeben.

Zusätzlich zu den *E3DObjectMappings* können im Mapping-Modell Ressourcen *(E3DResourceDescription)* und räumliche Relationen *(E3DSpatialRelationMapping)* spezifiziert werden.

Mithilfe von Ressourcen werden externe Daten wie z.B. Meshes eingebunden. Im Mapping-Modell wird mit einem *E3DResourceDescription*-Objekt angegeben, wie diese Ressourcen vom 3D-Editor geladen werden können. Im Prototypen ist nur eine Art von Ressourcen implementiert: einfache 3D-Modelle, zu denen im Mapping-Modell in Objekten vom Typ *E3DSimpleMeshDescription* der Pfad der Datei, aus der das 3D-Modell geladen werden soll, angegeben wird. Dieser Dateipfad ist dabei als relativer Pfad ausgehend von der Datei, in der das Mapping-Modell gespeichert ist, anzugeben.

Im Mapping-Modell in Abbildung 4.2 ist eine *E3DSimpleMeshDescription* definiert, in der der Dateipfad des 3D-Modells einer Fertigungsstation angegeben ist. Die *SimpleMeshDescription*, die im *E3DObjectMapping* der AssemblyStation definiert ist, verweist auf diese *E3DSimpleMeshDescription*. Das stellt sicher, dass im 3D-Editor Objekte vom Typ *AssemblyStation* tatsächlich als Fertigungsstation angezeigt werden.

Neben den Ressourcen werden im Mapping-Modell Mappings für räumliche Relationen *(E3DSpatialRelationMapping)* angegeben. Jedes Mapping definiert für eine Relation im referenzierten Klassenmodell, wie Instanzen dieser Relation sich auf die räumliche Anordnung der involvierten Objekte auswirken sollen.

Das im Screenshot ausgewählte *E3DAnchorPointRelationMapping*-Objekt spezifiziert, dass die *isAt*-Relation zwischen Objekten der Klassen *AssemblyStation* und *Robot* durch eine Ankerpunktrelation dargestellt werden soll. Die *isAt*-Relation aus dem *FactoryRobots*-Klassenmodell wurde in der *Properties*-Ansicht direkt angegeben. Zusätzlich zu der Relation sind noch die Ankerpunkte anzugeben, die durch die Ankerpunktrelation verbunden werden sollen. Dazu wählt man

einfach jeweils eine *AnchorPointDescription* aus, die bereits in einem *E3DObjectMapping* definiert ist. Als *SrcAnchorPoint* ist dabei der zur referenzierenden Klasse (hier *Robot*) gehörende Ankerpunkt anzugeben, als *DstAnchorPoint* muss der zur referenzierten Klasse (hier *AssemblyStation*) gehörende Ankerpunkt angegeben werden.

4.3 3D-Editor

Im 3D-Editor kann man ECore-Instanzmodelle in einer 3D-Ansicht erstellen und editieren. Zunächst erkläre ich, wie man Modelle erstellen, laden und speichern kann. Anschließend erläutere ich die 3D-Ansicht und veranschauliche, wie man diese benutzt.

4.3.1 Erstellen, Laden und Speichern von Modellen

Um mit einem Instanzmodell arbeiten zu können, muss zunächst ein bereits existierendes Mapping-Modell geladen werden, aus dem der Editor ein Representation-Modell erstellt.

Über den Menüeintrag *File/New* kann man ein leeres Representation-Modell erstellen. Dabei wird man aufgefordert, die Datei mit dem zu benutzenden Mapping-Modell zu öffnen.

Hat man ein Representation-Modell erstellt, kann man über das Menü neue Objekte erstellen und existierende Instanzmodelle importieren. Es öffnet sich ein Dialogfenster, in dem alle Objekt-Mappings des geladenen Mapping-Modells aufgelistet sind (siehe Abbildung 4.3). Mithilfe dieses Dialogs kann zu jedem Mapping ein neues Objekt erstellt werden, welches dann in der 3D-Ansicht angezeigt wird.

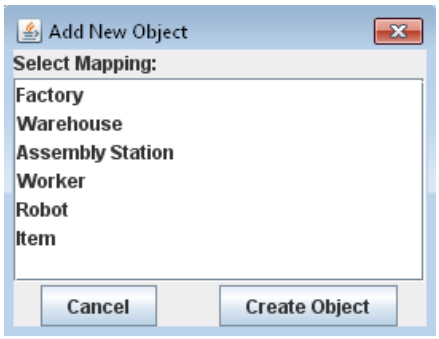

Abbildung 4.3: Screenshot des Auswahldialogs zur Erstellung eines neuen Objekts

Um ein existierendes Instanzmodell zu importieren, muss man die Datei auswählen, die das Instanzmodell enthält. Alle sich in dieser Datei befindenden Objekte, zu denen im aktuell geladenen Mapping-Modell ein

Mapping existiert, werden importiert. Die Kindobjekte der importierten Objekte werden ebenfalls rekursiv importiert. Existiert zu einem Objekt kein Mapping, wird weder das Objekt noch die Kinder dieses Objekts importiert. Relationen zwischen den importierten Objekten, zu denen im Mapping-Modell ein *SpatialRelationMapping* definiert ist, werden automatisch als räumliche Relation dargestellt.

Nachdem man im Editor ein Modell erstellt hat, kann man das Representation-Modell inklusive der Objekte des Instanzmodells in eine Datei speichern. Gespeichert werden alle Objekte mitsamt ihrer Darstellung und ihrer Position im Raum. Die Kameraposition wird nicht gespeichert, da sie nicht Teil des Representation-Modells ist.

Zu beachten ist, dass die gespeicherte Datei von der Datei, die das Mapping-Modell enthält, abhängt. Man sollte die Datei daher so speichern, dass sich der relative Pfad der Mapping-Datei von der gespeicherten Datei aus gesehen nicht verändert – also beispielsweise direkt neben der Mapping-Datei oder in einem Unterordner des Ordners, in dem sich die Mapping-Datei befindet. Eine auf diese Weise gespeicherte Datei kann zu einem beliebigen Zeitpunkt wieder geladen werden.

4.3.2 3D-Ansicht

Die 3D-Ansicht dient dazu, Objekte im Raum anzuordnen und darzustellen und mit den Objekten zu interagieren. Zur besseren Orientierung wird standardmäßig ein Gitter aus Linien dargestellt. Abbildung 4.4 zeigt einen Screenshot des 3D-Editors, in dem ein *FactoryRobots*-Instanzmodell geladen ist.

Die Navigation in der 3D-Ansicht erfolgt größtenteils mit der Maus. Mit der Maus kann die Kamera um einen Fixpunkt rotiert werden. Dieser Fixpunkt kann außerdem mitsamt der Kamera verschoben werden. Der Abstand der Kamera zum anvisierten Fixpunkt kann ebenfalls vergrößert und verkleinert werden. Auf diese Weise ist es möglich, die Szene von jedem Punkt im dreidimensionalen Raum aus zu betrachten.

Objekte, die in der 3D-Ansicht dargestellt werden, können durch Anklicken selektiert werden. Das selektierte Objekt wird durch einen pulsierenden Leuchteffekt hervorgehoben. Das ausgewählte Objekt kann verschoben, rotiert und skaliert werden. Mithilfe der drei Knöpfe in der oberen Toolbar kann dabei zwischen diesen Aktionen umgeschaltet werden.

Wird ein Objekt auf diese Weise bewegt, werden alle Objekte, die direkt oder indirekt über eine räumliche Relation mit dem bewegten Objekt verbunden sind, automatisch mit bewegt. Verschiebt, rotiert oder skaliert man beispielsweise die Platine, die im Screenshot auf der linken

[2] Die im Screenshot dargestellten Roboterarme und der am Regal stehende Mann sind Abwandlungen der 3D-Modelle aus [5] und [6]

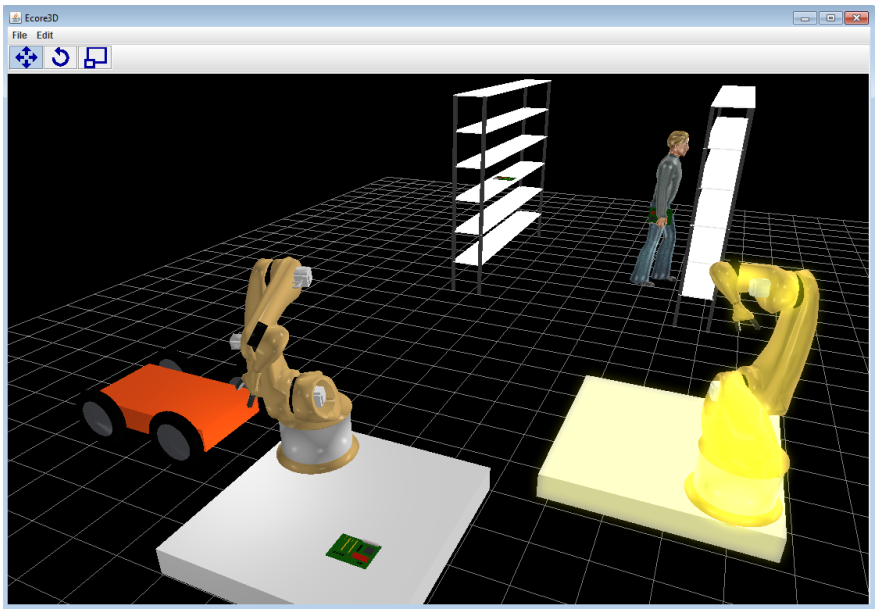

Abbildung 4.4: Screenshot des 3D-Editors mit einer selektierten Fertigungsstation [2]

Fertigungsstation liegt, wird die Fertigungsstation und der rote Roboter, der an der Fertigungsstation steht, ebenfalls verschoben, rotiert bzw. skaliert.

Kapitel 5

Technische Details

In diesem Kapitel erläutere ich die technischen Details der implementierten Software. Da der Mapping-Editor automatisch von EMF generiert ist, konzentriere ich mich hierbei auf den 3D-Editor. Zunächst gebe ich einen Überblick über den Aufbau des 3D-Editors. Danach erläutere ich jeden Teil der Software im Detail.

5.1 Struktur der Software

Der 3D-Editor ist in Java programmiert und benutzt für die 3D-Darstellung die JMonkeyEngine 3.0 (JME 3.0) [3]. Er ist kompatibel mit allen gängigen Versionen von Windows, Linux und OS X. Voraussetzung ist, dass eine Java7-Laufzeitumgebung (JRE) installiert ist. Getestet wurde der Editor unter Windows 7 und Xubuntu 14.04.3 LTS.

Der 3D-Editor ist in verschiedene Komponenten aufgeteilt. Von zentraler Bedeutung sind dabei zunächst einmal die Modelle selbst, also das Mapping-, das Representation- und das Editor-Metamodell. Die Implementierung der Modell-Klassen befindet sich im Package `ecore3d.model` und ist größtenteils automatisch von EMF generiert.

Das Package `ecore.model.description.impl` enthält die Implementierung des Mappings. Dieses ist dafür verantwortlich, Objekte des Representation-Modells zu Objekten des Instanzmodells zu erzeugen.

Direkt im `ecore`-Package befindet sich die Hauptklasse (*Application*), sowie verschiedene Klassen, die für die Verwaltung des Modells zuständig sind, aber nicht mit JME-spezifischen Klassen interagieren.

Im Package `ecore.jme` sind zwei Arten von Klassen enthalten, die ich in den folgenden Abschnitten getrennt betrachte: Einerseits die `JmeE3D*`-Klassen, die zu jeder Klasse des Representation-Modells und des Editor-Modells eine JME-spezifische Implementierung bereitstellen. Andererseits sind auch Klassen enthalten, die keinen direkten Bezug zu einer spezifischen

Modellklasse haben. Diese Klassen dienen zur Verwaltung JME-spezifischer Datenstrukturen und der Behandlung von Benutzereingaben.

Das `ecore3d.gui`-Package enthält die Swing-GUI, welche die gesamte Benutzeroberfläche außer der 3D-Ansicht bereitstellt. Die zur GUI zugehörigen Controller befinden sich im Package `ecore3d.controller`.

In den folgenden Abschnitten erläutere ich die oben aufgezählten Kategorien genauer und erkläre die Funktionsweise der enthaltenen Klassen.

5.2 Mapping-, Representation- und Editor-Modell

Im Framework sind drei verschiedene Metamodelle implementiert, die jeweils eine eigene Aufgabe haben.

Das Mapping-Metamodell dient zur Spezifikation von Mappings, die aus Objekten eines Instanzmodells die entsprechenden Objekte des Representation-Modells generieren. Der Aufbau und die Anwendung des Mapping-Metamodells ist in Abschnitt 4.2 genauer erläutert. Instanzen des Representation-Metamodells legen fest, wie Objekte im Raum dargestellt werden; sie definieren unter anderem die Position und das Aussehen der Objekte. Das Editor-Modell enthält Informationen über den Zustand des Editors selbst, wie z.B. die Position der Kamera und das momentan ausgewählte Objekt. Dieses Modell wird im Gegensatz zu den anderen Modellen nicht gespeichert, sondern lediglich intern genutzt. In Anhang A befindet sich zu jedem der drei Modelle ein Klassendiagramm.

5.2.1 Representation-Modell

Representation-Modelle bestehen immer aus einem Objekt vom Typ *E3DModelRepresentation*. Dieses Objekt enthält Instanzen der Klassen *E3DObject*, *E3DSpatialRelation* und *E3DResource*.

Ein *E3DObject* repräsentiert die Darstellung des Objekts, welches über die Eigenschaft `eObject` mit dem E3DObject verknüpft ist. Es enthält einen Scene Graph, der aus *E3DNode*-Objekten aufgebaut ist. Jeder Knoten kann ein *E3DNodeVisualization*-Objekt enthalten, welches festlegt, wie der Knoten angezeigt werden soll. Ein Knoten, der ein *E3DSimpleMesh*-Objekt enthält, wird beispielsweise als Mesh angezeigt. Neben dem Scene Graph enthält das *E3DObject* eine Transformation, welche die Position, Lage und Skalierung des Objekts in Weltkoordinaten angibt. Des Weiteren beinhaltet das *E3DObject* noch eine Menge von Ankerpunkten *(E3DAnchorPoint)*, die in Ankerpunktrelationen zu anderen Objekten genutzt werden können.

Jede räumliche Relation zwischen zwei Objekten wird durch eine *E3DSpatialRelation* abgebildet. Derzeit ist nur die Ankerpunktrelation *(E3DAnchorPointRelation)* implementiert, in zukünftigen Versionen können jedoch weitere räumliche Relationen als Unterklasse von *E3DSpatialRelation* implementiert werden. Ein Objekt vom Typ *E3DAnchorPointRelation*

enthält Verweise auf die beiden Ankerpunkte, die verknüpft werden sollen. Da jeder Ankerpunkt Teil eines *E3DObjects* ist, ist dadurch auch definiert, welche Objekte miteinander verknüpft werden sollen.

Neben *E3DObject*- und *E3DSpatialRelation*-Objekten enthält die *E3DModelRepresentation* Ressourcen, die durch *E3DResource*-Objekte repräsentiert werden. Ressourcen sind in diesem Zusammenhang Daten, die nicht direkt einem Objekt oder einer Relation zugeordnet sind, aber für die Darstellung benötigt werden. Die Klasse *E3DSimpleMeshResource* repräsentiert einen Mesh, der aus einer externen Datei geladen wird. Zusätzlich zu Meshes können in Zukunft weitere Arten von Ressourcen wie Shader oder Partikelsysteme implementiert werden.

5.2.2 Mapping-Modell und Zusammenhang zu Representation-Modell

Das Representation-Metamodell hängt eng mit dem Mapping-Metamodell zusammen. Jeder Klasse des Representation-Metamodells ist eine Klasse im Mapping-Metamodell zugeordnet. Jede Klasse des Mapping-Metamodells hat eine `createInstance`-Methode, die anhand der Informationen im Mapping-Objekt eine Instanz der jeweiligen Representation-Klasse erstellen kann. Dabei werden auch die `createInstance`-Methoden aller relevanten Kinder des Mapping-Objekts aufgerufen.

Auf diese Weise kann unter anderem mit einem einzigen Aufruf der `createInstance(E3DModelRepresentation, EObject)`-Methode der Klasse *E3DObjectMapping* das gesamte Representation-Modell zu einem Objekt des Instanzmodells erzeugen. Dabei werden zu allen im *E3DObjectMapping* spezifizierten Knoten, Visualisierungen und Ankerpunkten automatisch die jeweiligen Objekte des Representation-Modells erstellt.

Die auf diese Weise erstellten Representation-Objekte werden über das `mapping`- oder `description`-Attribut mit dem Mapping-Objekt, welches sie erstellt hat, verknüpft.

Implementiert ist die Erstellung von Representation-Objekten im Package `ecore3d.model.description.impl`.

5.3 Verwaltungsklassen

In diesem Abschnitt beschreibe ich die Funktionsweise der Klassen, die für die Verwaltung der Modelle zuständig sind.

Die Klasse *ResourceManager* ist für das Laden und Speichern von Modellen aus Dateien verantwortlich. Das Wort "resource" bezieht sich in diesem Kontext auf ECore-Ressourcen und hat nichts mit den Ressourcen zu tun, die im Mapping-Modell und im Representation-Modell vorkommen. ECore-Ressourcen sind hier die Dateien, in denen die Modelle gespeichert sind.

Das Speichern und Laden ist dabei mithilfe der in ECore enthaltenen Funktionalität zum Speichern und Laden von ECore-Modellen aus XMI-Dateien umgesetzt. XMI ist eine Erweiterung von XML und eignet sich besonders zum Speichern von Modellen und ihren Metamodellen.

Die Klasse *ObjectListener* beobachtet das Representation-Modell und verwaltet die räumlichen Relationen.

Der *ObjectListener* stellt sicher, dass zu jeder räumlichen Relation zwischen zwei Objekten des Instanzmodells die passenden *E3DSpatialRelation*-Objekte im Representation-Modell erzeugt werden. Berücksichtigt werden dabei nur Objekte, zu denen im Representation-Modell ein *E3DObject* existiert. Welche *E3DSpatialRelations* dabei zu welchen Relationen erzeugt werden, wird durch das Mapping-Modell definiert. Der *ObjectListener* sorgt ebenfalls dafür, dass die *E3DSpatialRelations* nur solange existieren, wie die dazugehörige Relation im Instanzmodell existiert.

Um dies zu bewerkstelligen, observiert der *ObjectListener* sowohl das Instanzmodell als auch das Representation-Modell. Entsteht im Instanzmodell eine neue Relation zwischen zwei Objekten oder wird im Representation-Modell ein neues *E3DObject* hinzugefügt, wird zu allen neuen Relationen eine *E3DSpatialRelation* erzeugt. Zu diesem Zweck wird für jede neue hinzukommende Relation zu jedem Mapping versucht, eine *E3DSpatialRelation* hinzuzufügen. Wird ein *E3DObject* entfernt oder wird eine Relation zwischen zwei Objekten aufgelöst, werden alle *E3DSpatialRelation*-Objekte, die zur entfernten Relation oder zum entfernten Objekt gehören, aus dem Representation-Modell entfernt.

5.4 Interne Klassen

Das Representation-Modell ist lediglich ein Modell der Darstellung. Die Objekte des Representation-Modells können nicht direkt dargestellt werden, da die verwendete 3D-Engine in der Regel eine eigene Datenstruktur benutzt. Aus diesem Grund kann jedes Objekt des Representation-Modells über ein `internalObject`-Attribut mit einem internen Objekt verknüpft werden, welches die Datenstrukturen der 3D-Engine für das jeweilige Objekt des Representation-Modells verwaltet. Neben dem Representation-Modell verfügen auch Objekte des Editor-Modells über solche internen Objekte.

Diese internen Objekte werden beim Speichern eines Representation-Modells nicht mit gespeichert. Sie müssen deshalb für jedes neu erstellte oder neu geladene Objekt des Representation-Modells neu generiert werden. Das hat den Vorteil, dass das Representation-Modell selbst nicht an eine spezifische Implementierung oder 3D-Engine gebunden ist. Dies ermöglicht die Entwicklung alternativer 3D-Editoren, die trotz verschiedener 3D-Engines oder Programmiersprachen zueinander kompatibel sind.

Alle internen Objekte sind im Package `ecore3d.jme` implementiert. Mit der Ausnahme von *E3DAnchorPoint* ist für jede Klasse des Representation-Metamodells eine interne Klasse mit dem Präfix `JmeE3D` implementiert.

Zur Klasse *E3DAnchorPointRelation* ist beispielsweise die Klasse *JmeE3DAnchorPointRelation* implementiert.

Ein *JmeE3DAnchorPointRelation*-Objekt ist immer einer *E3DAnchorPointRelation* zugeordnet. Es sorgt dafür, dass die vom *E3DAnchorPointRelation*-Objekt definierte Ankerpunktrelation in der JMonkeyEngine zu jedem Zeitpunkt umgesetzt wird.

5.5 Swing-GUI

Die Benutzeroberfläche des 3D-Editors basiert auf Swing. Die 3D-Ansicht ist dabei eine Swing-Komponente, die von JMonkeyEngine zur Verfügung gestellt und in das Hauptfenster eingebunden wird.

Die Swing-GUI ist nach dem MVC-Entwurfsmuster aufgebaut. Die Views, also die einzelnen Swing-Fenster, sind im Package `ecore3d.gui` implementiert. Das Package `ecore3d.controller` enthält den dazugehörigen Controller. Als Model wird das Editor-Modell verwendet, welches in ECore spezifiziert wurde und dessen Implementierung wie bei den anderen Modellen von EMF automatisch generiert wurde. Die Klassen zum Editor-Modell befinden sich im Package `ecore3d.model.editor`.

Kapitel 6

Evaluation

In diesem Kapitel evaluiere ich, inwiefern das hier entworfene Framework und der implementierte Prototyp zur Lösung des in Kapitel 1 beschriebenen Problems beiträgt.

Zur Erstellung neuer Modelle für Simulationen kann der Prototyp im derzeitigen Zustand nicht genutzt werden, da keine Funktionen zur Verfügung stehen, um das geladene Instanzmodell zu bearbeiten. Es ist zwar möglich, neue Objekte des Instanzmodells zu erstellen, jedoch können diese nicht im Editor mit anderen Objekten verknüpft oder gelöscht werden.

Das Erstellen von 3D-Darstellungen zu existierenden Instanzmodellen ist mit dem Prototypen bereits möglich. Hat man etwa bereits ein Modell einer Fabrik zum *FactoryRobots*-Modell erstellt, ist es möglich, dieses im 3D-Editor zu importieren und eine 3D-Darstellung dieser Fabrik zu erstellen.

Im derzeitigen Zustand kann der Prototyp jedoch noch nicht produktiv eingesetzt werden, da es mit den aktuell implementierten Tools sehr schwierig ist, Objekte genau anzuordnen. Objekte so zu verschieben, dass sie an einem bestimmten Punkt stehen und sie so zu rotieren, dass sie in eine bestimmte Richtung zeigen ist mühsam.

Der Mapping-Editor ist ebenfalls schwierig zu benutzen, da alle Daten manuell eingegeben werden müssen. Insbesondere die Definition der affinen Transformationen ist kompliziert, da sowohl die Positionskoordinaten als auch die Komponenten des Rotationsquaternions manuell angegeben werden müssen.

Diese Probleme sind allerdings keine systematischen Probleme, die durch die Struktur des Frameworks entstehen. Sie lassen sich durch die Implementierung weiterer Features gut lösen. In Abschnitt 7.2.1 schlage ich einige Ansätze zur Lösung dieser Probleme vor.

Das in dieser Arbeit implementierte Framework ist daher eine gute Grundlage, die in zukünftigen Arbeiten genutzt und erweitert werden kann, um das Ziel der domänenunabhängigen Erstellung interaktiver 3D-Editoren für modellbasierte Simulationen umzusetzen.

Kapitel 7

Zusammenfassung und Ausblick

7.1 Zusammenfassung

Das Ziel dieser Arbeit ist der Entwurf und die Implementierung eines Prototypen eines domänenunabhängigen Frameworks zur Definition von 3D-Editoren für modellbasierte Simulationen. Dabei habe ich mich auf die Definition der Editoren mit dem Mapping-Editor und der interaktiven Erstellung von 3D-Darstellungen zu benutzerdefinierten Modellen beschränkt. Der Prototyp ist aus verschiedenen Gründen, die ich in Kapitel 6 näher erläutere, noch nicht für den produktiven Einsatz geeignet. Er bietet jedoch eine gute Grundlage, auf der in zukünftigen Arbeiten aufgebaut werden kann um das Framework zu erweitern und so das Ziel des universell einsetzbaren 3D-Frameworks für Simulationen zu erreichen.

7.2 Ausblick

In diesem Abschnitt beschreibe ich zunächst einige Ansätze zur Lösung der in Kapitel 6 angesprochenen Probleme der derzeitigen Implementierung. Anschließend schlage ich Erweiterungen des momentanen Frameworks vor, die in zukünftigen Arbeiten umgesetzt werden können.

7.2.1 Verbesserungen am Editor

Eine wichtige im Prototypen nicht implementierte Funktion ist das Exportieren von Instanzmodellen. Damit die mit dem Framework erstellten Instanzmodelle auch von Drittsoftware genutzt werden können, muss es möglich sein, Instanzmodelle getrennt vom Representation-Modell zu exportieren. Das Exportieren von Instanzmodellen kann dabei ähnlich wie das Speichern des Representation-Modells über einen Menüpunkt umgesetzt werden.

Um Objekte besser im Raum platzieren zu können, werden außerdem vielfältigere und genauere Tools zum Verschieben, Rotieren und Skalieren der Objekte benötigt. Mit den momentan implementierten Tools ist es zum Beispiel schwierig, ein Objekt so zu verschieben, dass die Höhe gleich bleibt oder es so zu drehen, dass es nicht schief steht. Es ist daher sinnvoll, dem Nutzer zu ermöglichen, die Verschiebung, Rotation und Skalierung wie in der 3D-Modellierungssoftware *Blender* [2] auf eine oder zwei Achsen zu beschränken. Dies kann unter anderem mithilfe zusätzlicher Buttons in der Toolbar (einem für jede Achse) und Tastenkürzeln umgesetzt werden.

Damit man die Eigenschaften und Relationen zwischen Objekten auch tatsächlich im 3D-Editor bearbeiten kann, sollte zudem ein Editor für Objekteigenschaften implementiert werden. Zur besseren Übersicht ist außerdem eine Baumansicht sinnvoll, in der wie in den von EMF generierten Editoren alle Objekte in einem Baum aufgelistet sind. Abbildung 7.1 zeigt, wie so ein Editor aussehen könnte. Neben Eigenschaften des Instanzobjekts können im Eigenschafts-Editor auch Eigenschaften der Repräsentation wie z.B. Position, Skalierung und Drehung bearbeitet werden.

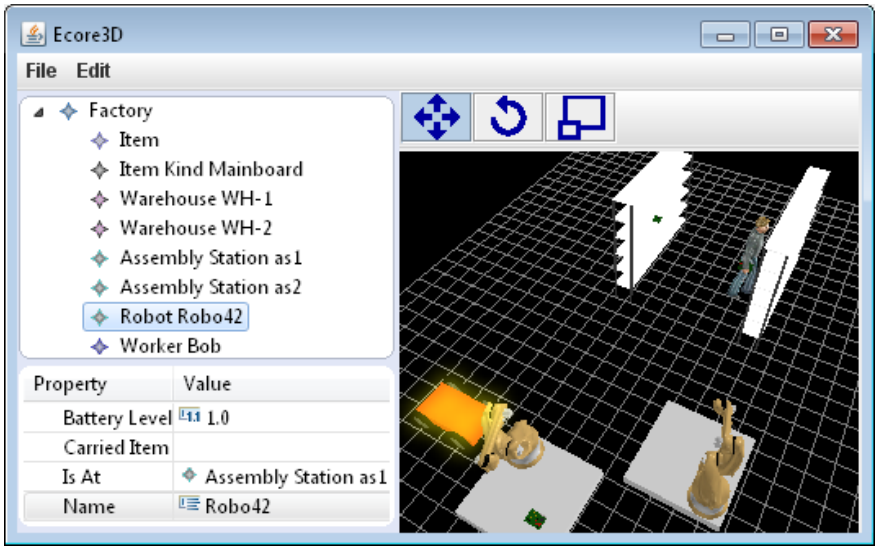

Abbildung 7.1: Mockup des 3D-Editors mit integriertem Baum-Editor und Eigenschaften-Editor

Sinnvoll ist ebenfalls, die 3D-Ansicht auch bei der Erstellung des Mapping-Modells einzusetzen. Dort könnte sie beispielsweise zur Vorschau von Objekten dienen, die mit dem ausgewählten Mapping-Modell erstellt werden. Zudem kann man mit einer 3D-Ansicht viel einfacher und intuitiver Ankerpunkte und Transformationen für Knoten definieren, als nur mit einem Baum-Editor.

7.2.2 Erweiterungen des Frameworks

Wie in Kapitel 1 beschrieben, soll das Framework nicht nur dazu dienen, 3D-Darstellungen zu Modellen anzufertigen. Es soll ebenfalls ermöglichen, interaktive 3D-Darstellungen zu Simulationen zu erstellen. Dabei sollen die Visualisierungen der Objekte während der Simulation automatisch angepasst werden. Dazu ist es erforderlich, dass der 3D-Editor auf einfache Weise in existierende Software, die Simulationen mit ECore-Modellen durchführt, integriert werden kann. Dieser Simulationsmodus und die dafür erforderliche Integrationsmöglichkeit kann in einer zukünftigen Arbeit erarbeitet und implementiert werden.

Da das Framework domänenunabhängig funktionieren soll, ist es ebenfalls wichtig, dass die Funktionalität des Frameworks bei Bedarf mit Plugins erweiterbar ist. Auf diese Weise kann es für Domänen angepasst werden, für die der Funktionsumfang des Frameworks nicht ausreicht. Durch Plugins können in solchen Fällen etwa komplizierte räumliche Relationen oder dynamisch generierte 3D-Meshes zur Verfügung gestellt werden. Die Umsetzung einer solchen Plugin-Struktur kann ebenfalls in einer zukünftigen Arbeit vorgenommen werden.

Anhang A

Klassendiagramme

Dieser Anhang enthält vereinfachte Versionen des Mapping-Metamodells, des Representation-Metamodells und des Editor-Metamodells. Einige Attribute und Methoden der Klassen wurden aus Platzgründen weggelassen. Dazu zählen insbesondere die `createInstance`-Methoden der Klassen des Mapping-Metamodells und die `mapping`- und `description`-Attribute der Klassen des Representation-Metamodells, welche auf das Mapping-Modell verweisen.

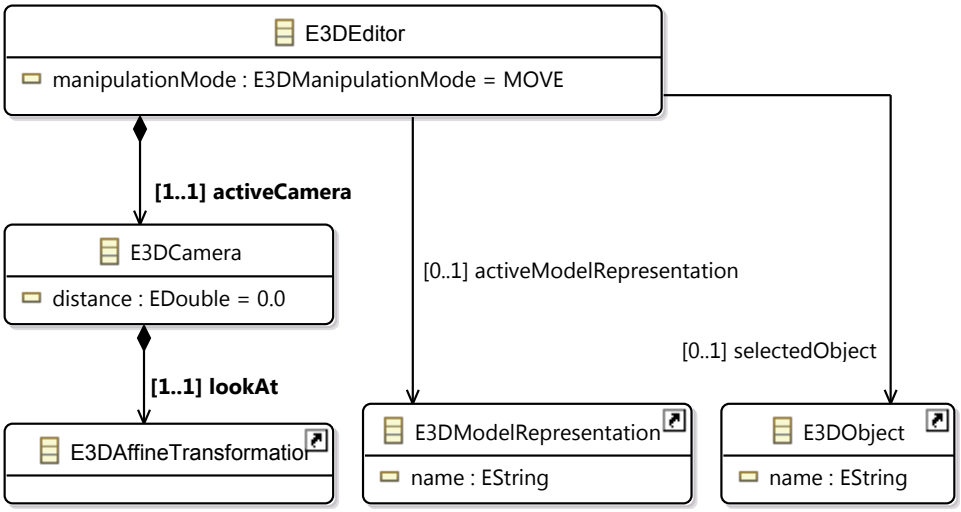

Abbildung A.1: Vereinfachte Version des Metamodells für Editor-Modelle.

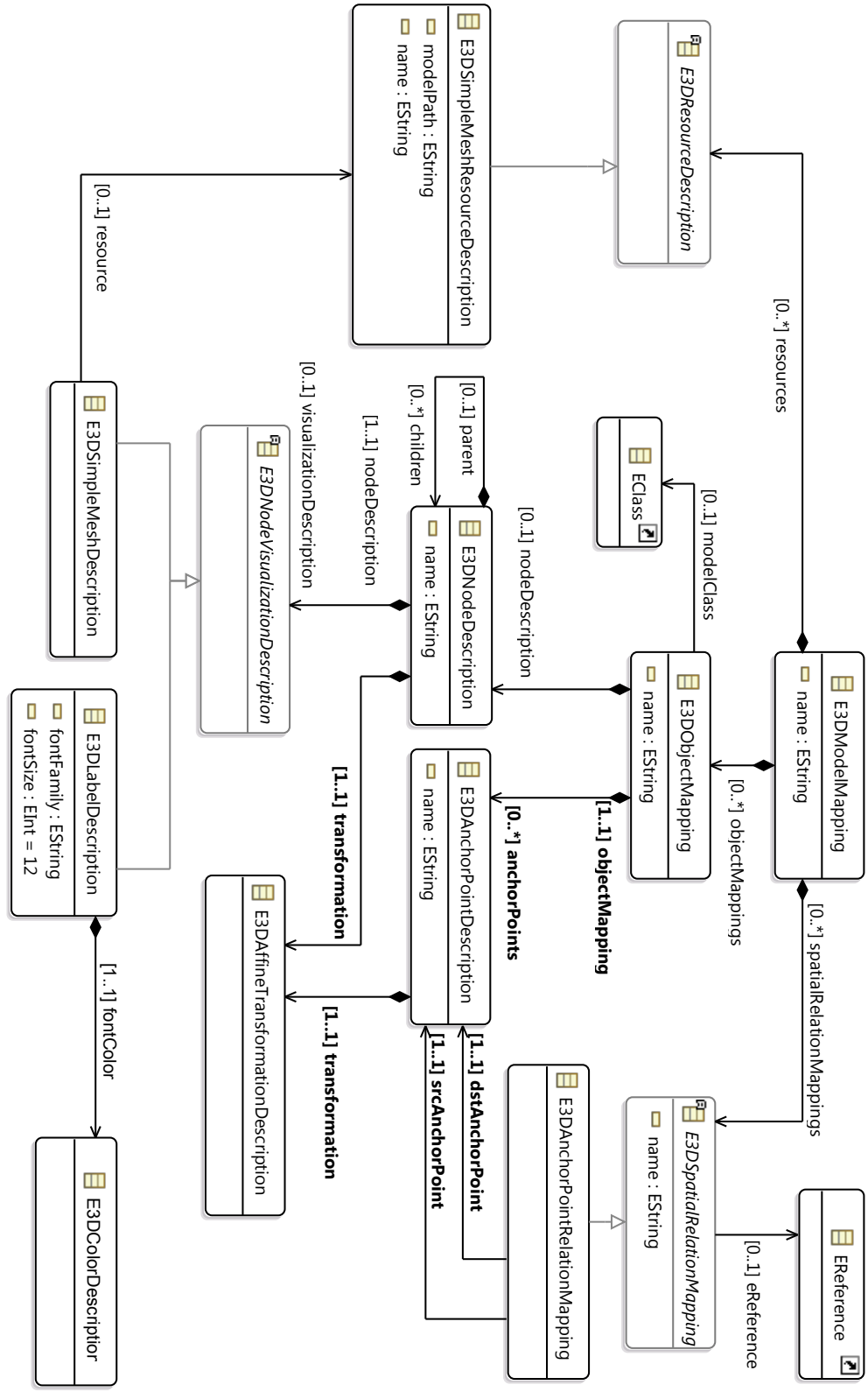

Abbildung A.2: Vereinfachte Version des Metamodells für Mapping-Modelle.

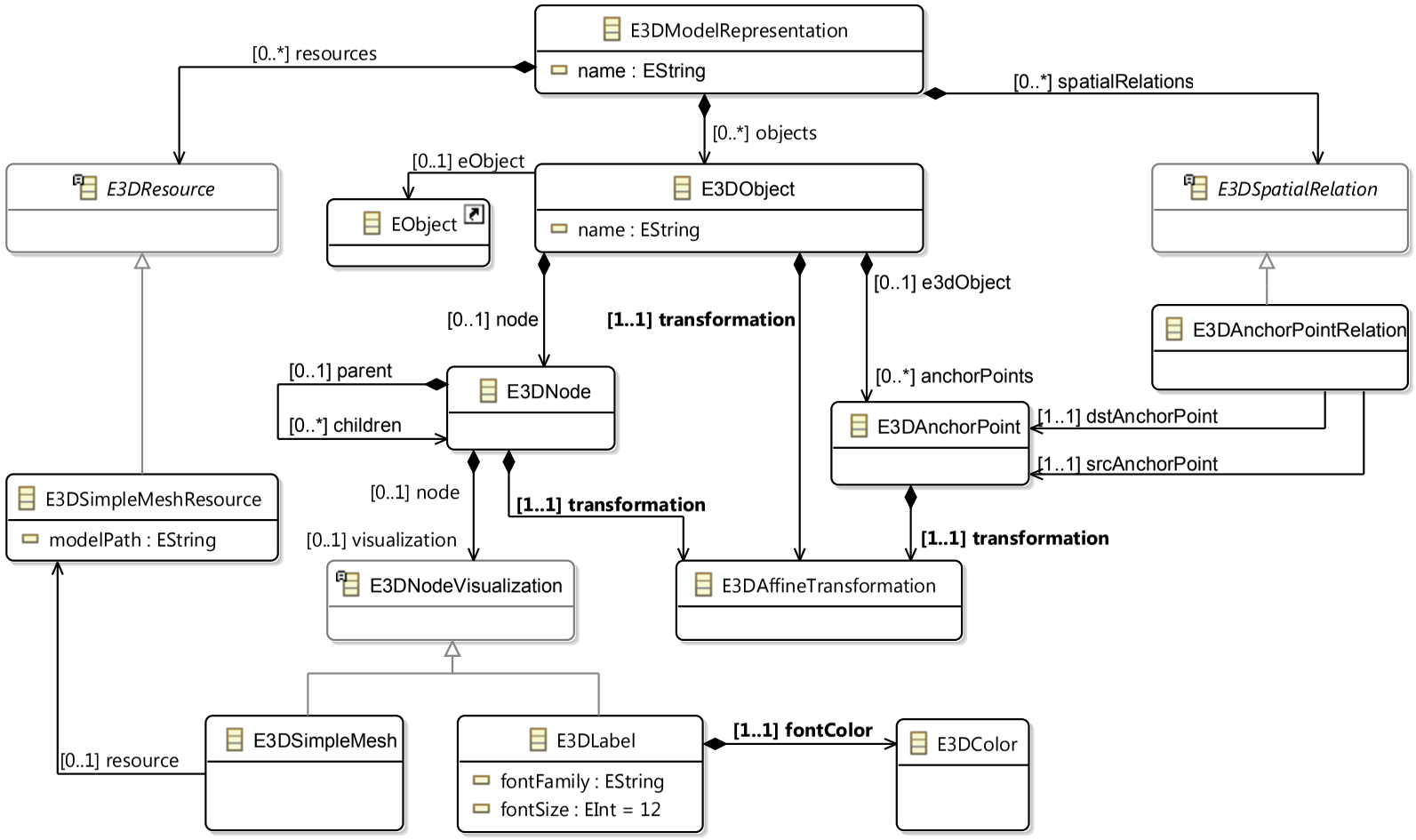

Abbildung A.3: Vereinfachte Version des Metamodells für Representation-Modelle.

Literaturverzeichnis

[1] Autotürme: www.autostadt.de. http://www.autostadt.de/de/autostadt-erkunden/autotuerme/ (Stand: 13.09.2015).

[2] Introduction – blender reference manual. https://www.blender.org/manual/editors/3dview/transform/introduction.html (Stand: 22.09.2015).

[3] jmonkeyengine - 3d game development for java developers. http://jmonkeyengine.org/ (Stand: 25.09.2015).

[4] Sirius - the easiest way to get your own modeling tool. https://eclipse.org/sirius/ (Stand: 27.09.2015).

[5] blendswap.com Benutzer "raovq". Rigged robotic arm. http://www.blendswap.com/blends/view/67329 (Stand: 14.09.2015).

[6] T. Loni. Lowpoly man. http://www.blendswap.com/blends/view/66412 (Stand: 20.09.2015).

[7] S. Winetzhammer, J. Greenyer, and M. Tichy. Integrating graph transformations and modal sequence diagrams for specifying structurally dynamic reactive systems. In D. Amyot, P. Fonseca i Casas, and G. Mussbacher, editors, *System Analysis and Modeling: Models and Reusability*, volume 8769 of *Lecture Notes in Computer Science*, pages 126–141. Springer International Publishing, 2014.