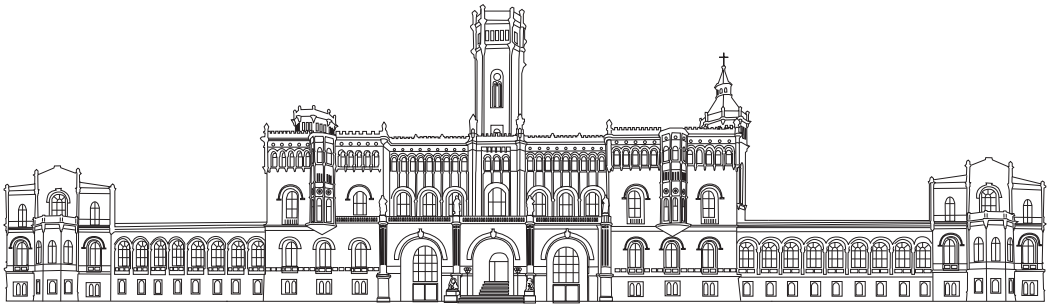

LEIBNIZ UNIVERSITÄT HANNOVER

FAKULTÄT FÜR ELEKTROTECHNIK UND INFORMATIK

INSTITUT FÜR PRAKTISCHE INFORMATIK

FACHGEBIET SOFTWARE ENGINEERING

”2D-Visualisierung der Simulation vernetzter Systeme in Scenario Tools”

Bachelorarbeit im Studiengang Informatik

eingereicht von

MINH HIEP PHAM

in Hannover, am 29. Mai 2015

Erstprüfer	:	Prof. Dr. Joel Greenyer
Zweiprüfer	:	Prof. Dr. Kurt Schneider
Betreuer	:	Prof. Dr. Joel Greenyer

Zusammenfassung

Simulation in der ScenarioTools umfassen gewisse Gemeinsamkeiten wie Z.B die aktuelle Struktur des Objektsystems, die List von erkundeten Nachrichten, usw. In der Simulation kann Benutzer die Widersprüche und die Auswirkung der Nachricht auf System zusammen mit der Umwelt in szenariobasierten Spezifikation erkennen. Um dieser Aufgaben gerecht zu werden, wird der Play - Out - Algorithmus integriert. Dieses Algorithmus bietet die Möglichkeit, um die Wirkung und die Ausführung von dem System zusammen mit der Umwelt zu analysieren. Aus diesem Grund soll ein grafische Oberfläche verlangt werden, mit der die Informationen während der Simulation in Grafik visualisiert werden können. Sobald der Benutzer simuliert, erkennt der Benutzer an welchen Zustand, welche aktivierte Nachricht und welchen Ablauf das System zusammen mit der Umwelt ausführt. Mit diesem Ergebnis von der Simulation kann er ihn ausbessern, und die Fehler zu vermeiden.

Inhaltsverzeichnis

Inhaltverzeichnis **iii**

Abbildungsverzeichnis **vi**

Tabellenverzeichnis **vii**

Quellcodelistings **viii**

Abkürzungsverzeichnis **ix**

1. Einleitung **1**
1.1. Problemstellung 1
1.2. Lösungsansatz 2
1.3. Struktur der Arbeit 3

2. Grundlagen **5**
2.1. Grundlagen der Modellierung 5
2.1.1. Beispiel 6
2.1.2. Modal Sequence Diagram (MSD) - Spezifikationen 8
2.1.2.1. Modal Sequence Diagrams (MSDs) für Anforderungen 8
2.1.2.2. MSDs für Annahme 11
2.2. ScenarioTools 15
2.2.1. Erweiterung von Play-Out-Algorithmus 16
2.2.2. Realisierung 19
2.3. Eclipse Modeling Framework 21
2.3.1. Grundlage 21
2.3.2. Persistence 24
2.4. Sirius 25
2.4.1. Sirius Grundlagen 25
2.4.2. Sirius Meta - Modelle 28

3. Anforderungsanalyse **30**
3.1. Systemsbeschreibung 30
3.2. Anforderungsspezifikation 32

3.2.1. Anforderungen 33
3.2.2. Qualitätsanforderungen 34
3.3. Anwendungsfälle 35
3.4. Zusammenfassung 37

4. Entwurf **38**
4.1. Erläuterung der Lösungsansätze 39
4.2. Struktur des Modells 41
4.2.1. Beziehung 41
4.2.2. Paketen 42
4.2.3. Klassen 43
4.3. Zusammenfassung 45

5. Projektumsetzung **46**
5.1. SimulationDiagram Editor 46
5.2. Visualisierung - Transformation 48
5.2.1. Die Bestimmung der Domänen 49
5.2.2. Ergebnis der Transformation 51
5.2.3. Umsetzung der Visualisierung - Transformation 53
5.3. NVisualisierung 54
5.3.1. Domänen und Zuordnung 55
5.3.2. Umsetzung des Algorithmus 56
5.4. Benutzeroberfläche 58

6. Verwandte Arbeiten **60**

7. Zusammenfassung und Ausblick **61**

A. Abbildung **63**

B. SimDiagram Mapping **67**

C. Beispeil **68**

Literaturverzeichnis **72**

Erklärung der Selbstständigkeit **73**

Abbildungsverzeichnis

1.1. Das Grobentwurf des Ablauf der Anwendung 3

2.1. Aufbau einer Strßenbahnsystem an [GSS14] und [GBM13]. 6
2.2. Straßenbahn vor der Endpunkt einer Laufbahn. In Anlehnung an [GSS14] und [GBM13]. 7
2.3. Kompositions-Strukturdiagramm und Klassendiagramm von Beispiel in 2.1.1. 9
2.4. MSD - Sequenzdiagramm von Anforderung 1 des Beispiels 2.1.1. In Anlehnung an [GBM13]. 10
2.5. Erläuterung des Fall „approaching a crossing“. In Anlehnung an [GBM13]. 12
2.6. MSD definiert Anforderung, um *barriers* zu schließen. In Anlehnung an [GBM13]. 13
2.7. MSDs für Annahme des Beispiel in 2.5. In Anlehnung an [GBM13]. . 15
2.8. Überblick von Funktion **performStep**. In Anlehnung an [GBM13]. . . 16
2.9. Der Screenshot von Benutzeroberfläche der Simulation in ScenarioTools. 20
2.10. Untermenge von Ecore Metamodell 22
2.11. Hauptarchitektur von Sirius Plug-in. In Anlehnung [OT13a]. 26
2.12. Die Interaktionen von Sirius Runtime. In Anlehnung [OT13a]. 27

3.1. Die Zusammenhänge der einzelnen Systemkomponenten von ScenarioTools. In Anlehnung [Gut14]. 31
3.2. Die Zusammenhang von wichtigen Use-Case inklusive ScenarioTools und Sirius. 35
3.3. Anwendungsfalldiagramm der Simulationsausführung 36

4.1. Interaktion zwischen Sirius Runtime und ScenarioTools Runtime. In Anlehnung [OT13a]. 40
4.2. Beziehungen zwischen ScenarioTools und Sirius. 41
4.3. Klassendiagramm von geänderten Klassen und neuen Klassen . . . 44

5.1. Das Aktivitäsdiagramm beschreibt den Ablauf, in den die Visualisierung - Transformation integiert wurde. 49

5.2. Zu sehen sind die Zusammenhänge von einzelnen Komponenten von ScenarioTools, die über die Grunkstruktur des Simulationsystems in Bezug gesetzt wurden. Ein Simulationsystem besteht im Wesentlichen aus ein Objectsystem von MSDRuntime. 52
5.3. Das Aktivitäsdiagramm beschreibt den Ablauf, in den die NVisualisierung integriert wurde. 55
5.4. Zu sehen sind die Zusammenhänge von einzelnen Komponenten von ScenarioTools, die über die letzte aktivierte Nachricht des Simulationsystems in Bezug gesetzt wurden. Ein Nachricht besteht im Wesentlichen aus einem sendenden und einem empfangenden Objekt von MSDRuntime. 57
5.5. Der Screenshot von Benutzeroberfläche des ScenarioTools nach der Erweiterung. 59

A.1. Die Basisklassen von Ecore Modell. 63
A.2. Die Zusammenhänge zwischen einzelnen Systemkomponenten von ScenarioTools und die Erweiterung. 64
A.3. Verlauf von neuen Lösungsansatz. 65
A.4. Struktur der Editor *SimDiagram* in EMF - Modell. 66

B.1. Die Mapping für Editor *SimDiagram* mit Hilfe von Sirius - Framework. 67

C.1. Die Simulation hat gestartet. 68
C.2. Die Nachricht RequestEnter wird aktiviert. 69
C.3. Die Nachricht EndOfTS wird aktiviert. 69
C.4. RailCabSystem in Editor SimDiagram modelliert wird. 70

Tabellenverzeichnis

2.1. Die verschiedenen Nachrichtentypen in der MSD - Spezifikation. In Anlehnung an [Gre11]. 10
2.2. List von Ecore Data Type. 23

5.1. Zuordnung der Domänen 51
5.2. Die neu Zuordnung der Domänen 56

Quellencodelistings

2.1. Beispiel von Bearbeitung mit Persistenz in Eclipse Modeling Framework (EMF) 24

Abkürzungsverzeichnis

EMF Eclipse Modeling Framework

GMF Graphical Modeling Framework

MOF Meta Object Facility

MSD Modal Sequence Diagram

MSDs Modal Sequence Diagrams

MSSs Modal State Structures

MSCs Message Sequence Charts

LSCs Live Sequence Charts

OMG Object Management Group

UML Unified Modeling Language

XMI XML Metadata Interchange

Kapitel 1.

Einleitung

1.1. Problemstellung

In unseren Lebens- und Arbeitsbereichen treffen wir oft auf viele ubiquitäre Systeme, die Z.B. „vernetze Städte“, „Car-to-Car Kommunikation“, „intelligentes Haus“, etc. sind. Ein ubiquitäres System ist ein komplexes System, das oft aus einer Kombination einer Vielzahl von Geräten und Software besteht. Die Komponenten eines ubiquitären Systems können miteinander und mit dem Menschen interagieren oder kommunizieren. Mit diesem System werden komplexe Aufgaben in unseren Lebens- und Arbeitsbereichen gelöst. Heute und in der Zukunft werden die Nachfrage von ubiquitären Systeme, die Z.B. Arbeitssysteme in Fabriken (vernetze Produktions- und Robotersysteme) oder Hilfesysteme in Krankenhäusern (Diagnose- und Überwachungsysteme) sind, sehr stark zunehmen.

Wenn Entwickler ein ubiquitäres System entwickeln, sollen sie zuerst auf die Komponenten und Steuerung solcher Systeme achten. Diese Aufgabe ist eine groše Herausforderung, weil Entwickler und Anwender teils komplexe Abläufe und die Beziehungen zwischen Komponenten verstehen möchten. Um eine hochwertige Software für diese Systeme zu entwickeln, sollen Entwickler systematisch vorgehen. Zunächst arbeiten Entwicklern meist mit Modellierung basierend auf Anwendungsfälle und Szenerien, die informell beschreiben, was das System in bestimmten Situationen machen kann. Daher wird die Werkzeugumgebung ScenarioTools am Fachgebiet Software Engineering entwickelt. In ScenarioTools können Entwickler UML-Diagramme für die gewünschten Systeme erstellen, und eine Simulation von szenariobasierten Modellen ausführen, um die Abläufe von Systemen in bestimmten Situationen zu testen. Deshalb haben Entwickler Möglichkeiten früh szenariobasierte Anforderungen zu programmieren, und auch früh Widersprüchen durch automatisierte Analysenmethoden zu finden und zu verstehen. Das Problem ist,

dass die Simulation von Komponenten des modellierten Systems und Beziehung zwischen Komponenten nicht in grafischer Form präsentiert werden, und der Benutzer Nachrichtenaustausch in jede Schritt der Simulation nicht sehen kann. Deshalb sind die Abläufe des Systems schwer verständlich. [GSS14]

Um diese Probleme zu lösen, plane ich in dieser Arbeit die Entwicklung von einer flexiblen Anwendung für die Visualisierung und Steuerung der Simulation von ScenarioTools. Mit dieser Anwendung können Entwickler ein System visualisieren, und Beziehungen zwischen Komponenten betrachten. Sie können auch die Betrachtung von der Interaktionen des Systems machen, um beispielweise die Kommunikation zwischen Robotern in einer vernetzen Produktions- und Robotersysteme zu untersuchen.

1.2. Lösungsansatz

Zur Erarbeitung der Anwendung arbeite ich mich zunächst in die Problemstellung und Technologie ein und untersuchen, welche existierenden Frameworks und Methoden ich für die Entwicklung der visuellen Anwendung verwenden kann. Dann entwickele ich ein Lösungskonzept anhand eines oder mehrerer Anwendungsbeispiele, implementiere diese prototypisch und evaluiere das Lösungskonzept am Beispiel.

Die zu erstellende Anwendung soll in zwei Teile gegliedert sein. Diese Teile sind Grafik-Editor und Event-Visualizer.

Der Grafik-Editor ist der Hauptteil der Anwendung. Er besteht aus einer grafischen Oberfläche, wo Benutzer ein Objektsystem mit aktuellen Objekten modellieren können oder neue Objekte hinzufügen können. In der grafischen Oberfläche werden alle Komponenten des modellierten Systems und die Beziehung zwischen Komponenten visualisiert. Wenn Benutzer die Modellierung simulieren möchte, werden die Nachrichten zwischen Benutzern und Systemkomponenten in diesem Teil visualisiert werden.

Event-Visualizer ist ein wichtiges Teil, um die Nachrichten zwischen dem Grafik-Editor und ScenarioTools zu tauschen. Der Ablauf soll wie folgt realisiert werden. Der Benutzer wählt ein Ereignis der Ereignisse-Liste, dann bekommt die ScenarioTools-Runtime die Informationen von dem ausgewählten Ereignis. Danach führt die ScenarioTools-Runtime das Ereignis aus. Nach dem Ablauf eines Schritts wird das ausgewählte Ereignis an den Event-Visualizer übertragen, und die Ereignisse-Liste wird gleichzeitig aktualisieren. Danach wird das ausgewählte Ereignis im Grafik-Editor visualisiert.

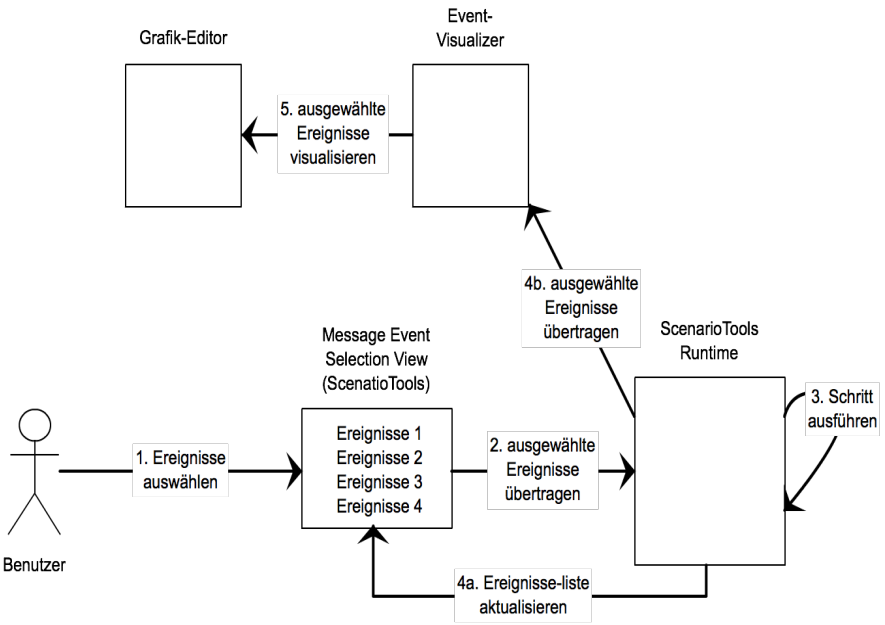

Abbildung 1.1.: Das Grobentwurf des Ablauf der Anwendung

Auf jeden Fall müssen Grafik-Editor und Event-Visualizer in der Anwendung implementiert werden, weil diese Teile die grundlegenden Bausteinarten der Anwendung sind.

1.3. Struktur der Arbeit

Hier die Struktur der Arbeit kurz zusammenfassen: Diese Arbeit ist wie folgt strukturiert.

In Kapitel **Grundlagen** gibt einen Überblick über die Grundlagen der Arbeit. Es werden die Erklärungen und Begrifflichkeit der modellgetriebener Ansätze , Live Sequence Charts (LSCs) und MSDs beschrieben. Dann wird die Gesamtüberblick über Modellierungssprache und Modellierung-Entwicklungsmethoden dargestellt, die in der Arbeit benutzt werden. Anschließend stelle ich nicht nur der Überblick über ScenarioTools, in der ich meine Lösung implementiert habe, sondern auch Sirius-Framework vor.

Das Kapitel 3 (**Anforderungsanalyse**) stellt die Spezifikation, die Anforderungen und die Designentscheidungen der Software für die Arbeit vor. Die Erläuterungen von Anwendungfälle und der wichtigen Anforderungen der Software werden im diesem Kapitel dargestellt.

Das vierte Kapitel **Entwurf** beschreibt grundlegende Design für die Anwendung

des erarbeiten Ansatzes. Zweite Möglickeiten direkt und indirekt werden in diesem Kapitel erklären. Die Erklärung des Software werden in diesem Kapitel als die Erläuterung der Beziehung zwischen Entities und Paketverteilung sein.

Im Kapitel 5 (**Projektumsetzung**) wird die Ergebnisse der Prototypenentwicklung besprochen. Anschließend wird erläutert, wie das Software der Arbeit entworfen wurde. Dann werden Struktur des Software erläutert. Die wichtigsten Teil und der dazu verwendeten Werkzeuge der Anwendung werden in diesem Kapitel erläutert. Am Ende werden einige Beispiel angezeigt, in dem die Anleitung Schritt für Schritt aufgezeigt wird. Die verschiedenen Erweiterungsmöglichkeiten werden auch dargestellt.

Verwandte Arbeiten sind in Kapitel 6 zusammengefasst. Das letzte Kapitel **Zusammenfassung und Ausblick** wird die Fazit aus der Ausarbeitungen gezogen. Weiterhin wird ein Ausblick, die die Vorschläge und die Vorstellung für die zukünftige Arbeit und auch die Erweiterungen im Bereich des Modellierung-Werkzeugs formuliert.

Kapitel 2.

Grundlagen

In den Entwicklungsprozess spielen die Modellierung eines System und die Integration von diesen Modelle eine wichtige Rolle, um die Komplexität eines System zu beherrschen. Im folgenden Kapitel wollen wir uns die Grundlagen schaffen, um grundlegende Verständnis über Arbeit zu haben. Dabei wird sowohl auf ein paare theoretischen Grundlagen eingegangen, als auch die Überblick von ScenarioTools begonnen. Im Abschnitt 2.1 erkläre ich die Grundlagen von Modellierung, MSDs-Spezifikation und ein Beispiel. Danach erläute ich ScenarioTools in 2.2, in der ich meine Lösung implementiert habe. Abschnitt 2.3 erkläre ich über EMF. Im letzten Abschnitt 2.4 wird die Funktionsweise von Sirius erklärt, bei der die Modellierungen als grafische Diagramm übernimmt werden.

2.1. Grundlagen der Modellierung

Die Beschreibung über Systemen bilden Modellen von diesem Systemen. Um die Verständnis folgen zu können, ist es sinnvoll, Definition von den Begriffen „Modelle“ und „System“ vorzustellen.

Der Begriff „System“ wird bei „IEEE 1471“ [Gro00] definiert:

„System: a collection of components organized to accomplish a specific function or set of functions.“[1]

Dies bedeutet, dass ein System durch mehreren Komponenten entsteht. Es hat eine Funktion oder eine Menge von Funktionen, um einen Zweck zu erfüllen. Die Kollektion von Komponenten ist verfügbar über eine Architektur. Es folgt, dass

[1] Mehr Detai über IEEE 1471 kann in `http://standards.ieee.org/findstds/standard/1471-2000.html` finden.

alles in unseren Lebens- und Arbeitsbereichen als System verstanden werden kann. Es kann nicht nur materielle Objekt (z.B der Komputer, der aus seinen Einzelteilen wie Bildschirm, Gehäuse, Maus, Tastatur, Eingang/Ausgangsteile usw. besteht), immaterielle Objekt wie Software (Module, Package, Klasse, EJBs, ...) sondern auch Geschäftsprozesse oder Unternehmensführung (Aktivitäten, Ressourcen, Humankapital, ...) usw. sein.

Es folgt, dass ein Modell ist die Beschreibung über einen System in passender Form. Ein Modell kann auf verschiedenste Art dargestellt werden: grafisch, textuell oder in Form von der Mischung dieser beiden. Visuelle dargestellte Modelle von ein System werden im Allgemeinen als Diagramme bezeichnet. Jede Art von Modell kann formalen oder skizzenhaften Charakter haben. Im Prozess der Modellierung kann ein System in mehreren als ein Modul zergliedert werden, und jede System besitzt die Architektur. Architektur des System entsteht durch eine Architekturbeschreibung, die von ein oder mehrere Modell zusammen fasst und mindesten durch ein Sicht organisiert. Je Modul enthält ein oder mehrere Aspekt, der in ein oder mehrere Modell beschrieben werden. Eine Sicht besteht aus eine kohärente Menge von dieser Modell. Diese Begriffen und die Zusammenhänge werden auf Buch MDA von Volker Gruhn, Daniel Pieper und Carsten Röttgers [GPR05] (Kapitel 3, Abschnitt 3.1) geklärt.

2.1.1. Beispiel

Das Beispiel behandelt eine „RailCab-System“, wie sie beispielsweise Straßenbahn in der Straßenbahnsystem kontrollieren. Die Idee des Beispiels ist den ScenarioTools Quellen [1] entnommen. In Abb. 2.1 ist ein Straßenbahnsystem zu sehen.

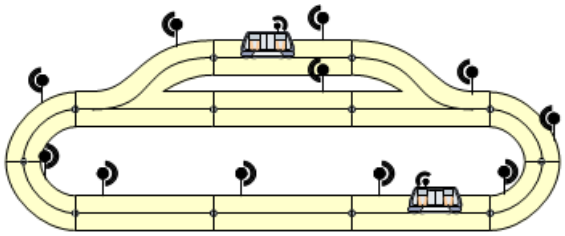

Abbildung 2.1.: Aufbau einer Strßenbahnsystem an [GSS14] und [GBM13].

Dieses System besteht aus ein oder mehrere Straßenbahnen, die autonom auf der Laufbahn des System fahren. Hinzu kommen noch mehrere

[1] Es wird von RailCab Project Paderborn (`http://www.railcab.de/index.php?id=2&L=1`) inspiriert

Abschnittskontrollen, die den Lauf dieser Straßenbahnen beaufsichtigen. Während des Laufes muss jede Straßenbahn vor der Abschnittskontrolle anmelden, und sie muss nach dem Durchlauf abmelden. Während des Laufes können die Nachrichten zwischen Komponenten des Straßenbahnsystem ändern, deshalb kann das System als dynamische System genannt. Der Lauf wird in mehrere Laufbahnen unterteilen. Zwischen zwei Laufbahnen steht ein Abschnittskontrolle. Jede Laufbahn hat ein Endpunkt, der bei einem Sensor der Straßenbahn entdeckt wird. In diesem Punkt wird sie die Ereignisnachricht bei Umwelt erhalten, danach muss sie der nächsten Abschnittskontrolle ein Nachfrage nach Erlaubnis senden, um die nächste Laufbahn zu kommen. Vor dem Eintritt der nächsten Laufbahn muss Straßenbahn einer Antwort von der nächsten Abschnittskontrolle empfangen, deshalb kann er letzte Sicherheitsbremse machen. Die Komponenten des Strßenbahnsystem werden als wie unter genannt:

- Straßenbahn als *RailCab*.
- Umwelt als *Environment*.
- Abschnittskontrolle als *TrackSectionControl*.

Die Interaktion zwischen Straßenbahn und Abschnittskontrolle sind wie in Abb. 2.2 gezeigt.

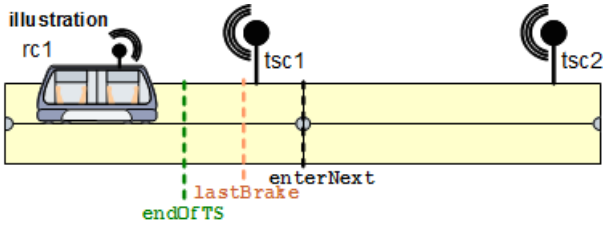

Abbildung 2.2.: Straßenbahn vor der Endpunkt einer Laufbahn. In Anlehnung an [GSS14] und [GBM13].

Im Folgende sind die Anforderungen an die Funktion des System als informale Szenarien (Nachrichtenaustausch zwischen *RailCab* rc und *TrackSectionControl* tsc1 und tsc2) beschrieben:

Anforderung 1 Nachdem *RailCab* rc den Endpunkt der Laufbahn ankommen ist, muss dem *TrackSectionControl* tsc2 die Nachfrage nach Erlaubnis der Zufahrt von *RailCab* rc nehmen und antworten. Sobald er die erlaubte Antwort von *TrackSectionControl* tsc2 bekommt, muss sie in der Richtung der nächsten Laufbahn weiterlaufen, wo er dann irgendwann ankommt.

Annahme 1 Wenn *TrackSectionControl* tsc2 ein erlaubte Antwort an *RailCab* rc senden soll, empfängt *RailCab* rc auch irgendwann.

Annahme 2 Wenn *TrackSectionControl* tsc2 ein unerlaubte Antwort an *RailCab* rc senden soll, empfängt *RailCab* rc auch irgendwann.

2.1.2. MSD - Spezifikationen

David Harel und Rami Marelly haben in Buch „Come, Let's Play“ [HM03] ein Ansatz, das als Live Sequence Charts (LSCs) genannt wird, beschrieben, um szenariobasierte Spezifikationen zu modellieren. MSD - Spezifikationen sind die eine Erweiterungen von UML Sequenzdiagramm und basiert auf Live Sequence Charts (LSCs). Sie ist eine grafische Methode, die UML-Sequenzdiagramm erweitern, um die Verhalten von System und seiner Komponenten in der im Zusammenspiel mit der Umwelt des System zu spezifizieren. Deshalb kann die Beschreibung über das System, die mit ihrer Umwelt interagieren, repräsentieren, was das System tun kann, muss bzw. nicht darf. Die Nachrichten der Komponenten von System werden bei MSD - Spezifikationen in verschiedenen Kategorien gliedert werden. Die Art von dieser MSD - Spezifikationen wird in der Dissertation von Greenyer [Gre11] und bei ihm und Christian Brenner, Valerio Panzica La Manna in den Erweiterung [GBM13] definiert.

MSD - Spezifikationen können als „existential “ oder „universal “ sein. In dieser Arbeit wird der Fokus auf „universal Modal Sequence Diagrams (MSDs)“ gelegt, bei der alle mögliche szenariobasierte Verhalten des System spezifiziert werden können. MSD - Spezifikation bestehen aus eine Menge von Modal Sequence Diagrams (MSDs), Modal State Structures (MSSs) und Message Sequence Charts (MSCs). Mit diesen Diagrammen ist es beschreibbar, welche mögliche Ereignisnachrichten vorkommen müssen, welche vorkommen mögen und welche nicht vorkommen dürfen. In [HM03] (Abschnitt 2.4) beschrieben David Harel und Rami Marrelly, dass das MSDs sich in zwei Teile unterteilt, die als Nebendiagramm und Hauptdiagramm gennant werden. Wenn Nebendiagramm erfüllt wird, dann muss System das Hauptdiagramm zu erfüllen tun. Jedes Element des Diagramm ist mit eine Temperatur definiert. Die Temperatur der lebendigen Elemente, die auftreten müssen, sind heiß („hot“). Die andere Elemente, die auftreten mögen, sind mit kalte Temperatur („cold“) definiert.

2.1.2.1. MSDs für Anforderungen

Jedes Modal Sequence Diagramm repräsentiert ein System, das als *Objekt-System* gennant wird. Alle Objekte, die zu diesem Objekt-System gehören, können zu einer von zwei Untersystem gehören, die man als *Umwelt-Objekt*

und *System-Objekt* nennt. Die erste Untersystem *Umwelt* beinhaltet alle *Umwelt-Objekte*, die zur Umwelt gehören. Die *System-Objekte* gehören zu Untersystem *System*. Objekte des Systems können miteinander Nachricht, zu der ein Name, ein sendende und ein empfangende Objekt gehören, austauschen. Die Umwelt-Objekte haben unkontrollierbare Nachrichten, aber die Nachrichten der System-Objekte sind kontrollierbar. Bezogen auf Beispiel 2.1.1 stellt das *TrackSectionControl* das System dar, die*RailCabs* werden von ihm gesteuert, ob die *RailCabs* in der nächsten Laufbahn weiterlaufen darf. Wegen dieser Nachrichten, die das *TrackSectionControl* sendet, hat es volle Kontrolle auf *RailCab*. Auf das Eintreffen von Nachrichten aus *RailCabs* kann es nicht kontrollieren, weil es auf *RailCab* warten. Diese Nachrichten können auch die asynchrone und synchrone Interaktion des System repräsentieren.

Im Beispiel von 2.1.1 beinhaltet System drei Objekte, die *Environment*, *RailCab* und *TrackSectionControl* sind. Objekt-System „RailCab-System“ kann in UML Kompositions-Strukturdiagramm und Klassendiagramm wie Abb. 2.3 und **??** modelliert werden.

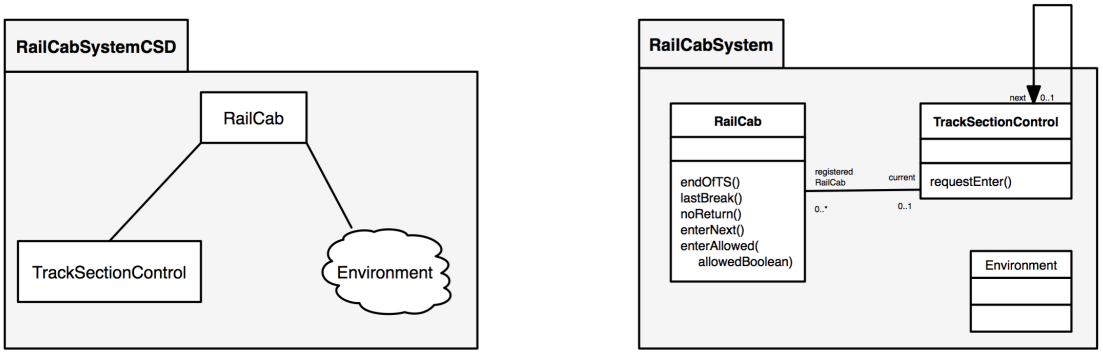

Abbildung 2.3.: Kompositions-Strukturdiagramm und Klassendiagramm von Beispiel in 2.1.1.

Im Kompositions-Strukturdiagramm werden Objekte bei Untersystem repräsentiert. Die Figur der Objekte von *Umwelt* ist eine rechteckige Form, und von *System* ist eine wolkenartige Form. Die Verbindungen erläutern, zwischen welches Objekt eine Nachricht ausgetauscht wird. Die Klassen definiert die Type des Objekt, wo die Nachrichten als Funktion der empfangenen Klasse definiert werden. Die Verbindung zwischen die Klassen sind die Zusammenhänge der Objekten des Systems.

Nachrichten zwischen Objekten mögen als heiß (*hot*) oder kalt (*cold*) sein. Eine heiße Nachricht heißt, dass sie nach der Sendung empfangen werden muss. Dieser Nachrichtentype ist als rot gezeichnet. Hingegen blaue gezeichnete

Nachricht ist kalte Nachrichten, die nach der Sendung nicht empfangen werden mögen. Wenn eine Nachricht auftreten muss, dann heißt sie *executed*, und sie wird durch einen durchgehende Pfeil repräsentiert. Eine Nachricht kann auch als *monitored* heißen, wenn sie auftreten mag, und sie wird durch gestrichelte Pfeil gezeichnet. Deshalb können die Nachrichten in MSD - Spezifikation einer von der Kombination in der Tabelle 2.1 sein.

semantische Nachrichten	*cold* (mag nicht empfangen werden)	*hot* (muss empfangen werden)
monitored event (mag auftreten)	- - - - - - - - - - - - -> c,m	- - - - - - - - - - - - -> h,m
executed event (muss auftreten)	——————————> c,e	——————————> h,e

Tabelle 2.1.: Die verschiedenen Nachrichtentypen in der MSD - Spezifikation. In Anlehnung an [Gre11].

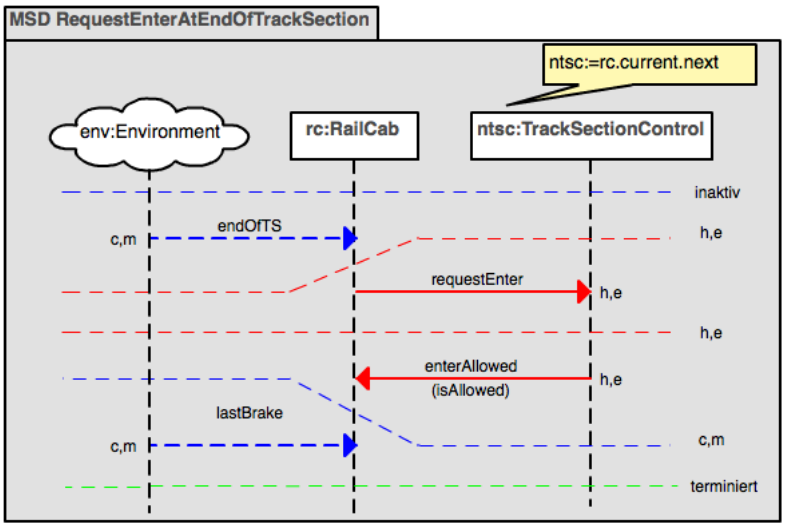

Abbildung 2.4.: MSD - Sequenzdiagramm von Anforderung 1 des Beispiels 2.1.1. In Anlehnung an [GBM13].

Anforderung 1 des Beispiels 2.1.1 wird als MSD - Spezifikation mit Sequenzdiagramm in Abb. 2.4 modelliert. Die erste Nachricht (**EndOfTS**) meint, dass *RailCab* den Endpunk der Laufbahn kommen mag. Zweite Nachricht (**requestEnter**) meint, wenn ein *RailCab* angekommen ist, muss es der nächsten *TrackSectionControl* die Nachfrage nach Erlaubnis der Zufahrt senden. Nachricht (**requestEnter)** ist *executed* und *hot*, deshalb muss sie irgendwann auftreten. Bevor dieser Nachricht gesendet wird, müssen andere Nachrichten auf seiner Sendung warten. Die näschte Nachricht (**enterAllowed(isAllowed)**) sagt, dass

TrackSectionControl eine Antwort auf die Nachfrage von *RailCab* sendet. Nachricht is *hot* und *executed*, deshalb andere Nachrichten auf die Sendung von dieser Nachricht warten müssen, wenn sie senden möchten. Sie muss irgendwann auftreten. Die letzte Nachricht ist *cold* und *monitored*, deshalb mag sie irgendwann auftreten, weil sie auf die Nachricht (**enterAllowed(isAllowed)**) wartet. Nach der Sendung dieser Nachricht mag sie nicht empfangen werden.

Wegen des Auftretens von einer Reihe der unendlichen Nachrichten wird eine Sequenz auf die Ereignisnachricht der Umwelt an die Nachrichten des Systems reagiert. Diese Nachrichten kann mit der Nachrichten in MSD vereinigen, wenn ihr Name und der Name der Nachrichten in MSD gleiche sind. Die sendende und empfangende Objekt der Nachricht werden bei der sendenden und empfangenden Lebenslinie der Nachrichten des Diagramms modelliert. Wenn erste Nachricht in MSD mit einer auftretenden Ereignisnachricht vereinigt, wird ein duplizierendes MSD erstellt, das als aktives MSD gennant wird. Die nachfolgende Nachrichten in MSD kann mit weiteren auftretenden Ereignisnachricht vereinigt, wird aktives MSD fortschreiten. Der Fortschritt wird bei den *Cut* in MSD angezeigt, wo der Zustand sich für alle Lebenslinie der sendenden und empfangenden Nachricht befinden. Der *Cut* wird durch eine gestrichelte Linie gezeichnet, die zwischen die aufgetretene Nachricht und jenen Nachrichten, die noch nicht aufgetreten sind, verläuft. Wenn die folgende Nachricht aktiviert ist, hat der *Cut* ähnliche Typen wie diese folgende Nachricht des MSD. Wenn die folgende aktive Nachricht *executed* ist, ist der *Cut executed*. Sonst ist er *monitored*. Wenn eine folgende heiße Nachricht aktiviert ist, ist der *hot*. Sonst ist er *cold*. Wenn der *Cut* der Ende des aktiven MSD, dann ist MSD terminiert.

Wenn ein auftretende Nachricht mit einer Nachricht in einem aktiven MSD, die nicht aktiviert ist, vereinigt werden kann, dann wird eine Sicherheitsverletzung (*safty violation*) auftreten. Wenn der *Cut* cold ist, wird eine kalte Verletzung (*cold violation*) auftreten. Sicherheitsverletzung ist zu auftreten verboten. Wenn eine kalte Verletzung auftreten wird, wird aktive MSD terminiert. Wenn die folgende Nachricht *executed* ist, aber kann sie nicht auftreten. Deshalb muss eine Aktion zugleich passieren, aber kann er nicht auftreten, wird eine Lebendigkeitsverletzung in dieser Situation auftreten.

2.1.2.2. MSDs für Annahme

In der Abschnitt 2.1.2.1 darf man mit MSDs für Anforderungen sowohl die Verhalten des System als auch Anforderung 1 von des Beispiel in Abschnitt 2.1.1 spezifizieren, aber es ermöglicht noch nicht die Verhalten von Umweltsystem zu

spezifizieren. Diese Beschränkung motiviert, um die Einleitung des Beispiel 2.1.1 mit einem neuen Fall zu erweitern. In diesem Fall wird *RailCab* eine Kreuzung erreichen. Diese Erweiterung ist einem Journal von Herr Joel Greenyer, Christian Brenner und Valerio Panzica La Manna [GBM13] (Abschnitt 3, S 6) entnommen.

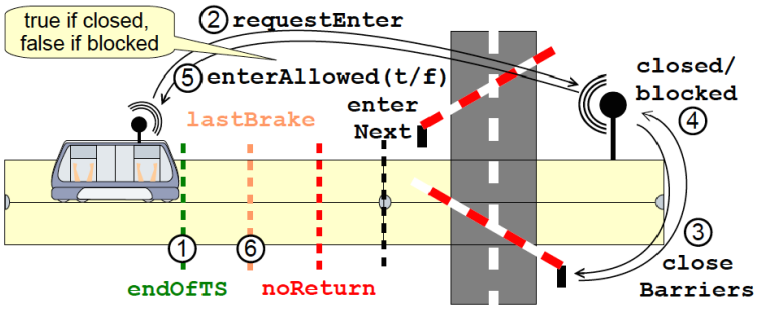

Abbildung 2.5.: Erläuterung des Fall „approaching a crossing“. In Anlehnung an [GBM13].

Abb. 2.5 erläutert diesem Fall mit mehr Detail. In diesem Fall wird *TrackSectioControl* an *CrossingControl* erweitern, bei dem Barrieren kontrolliert werden. Ein *RailCab* melden wird, dass er die Endpunkt des aktuellen Laufbahn angekommen ist (1-**endOfTS**). Dann muss er dem *CrossingControl* eine Nachfrage nach Erlaubnis senden, um die Kreuzung anzukommen (2-**EnterNext**). *CrossingControl* muss den Barrieren einen Befehl zu schließen senden (3-**closeBarriers**). Das *barriers* könn entweder geschlossen werden oder blockiert werden (4-**closed**/**blocked**). Wenn die Barrieren geschlossen werden (4-**closed**), dann soll *RailCab CrossingControl* anzukommen erlaubt werden (5-**enterAllowed(t)**). Wenn die Barrieren blockiert werden (4-**blocked**), dann darf *RailCab* nicht ankommen (5-**enterAllowed(f)**). Die Antwort muss vor dem letzten Punk gesendet, wo *RailCab* nicht mehr der letzten Sicherheitsbremse machen kann (6-**lastBrake**). Die Barrieren werden als *barriers* genannt.

Diese Fall wird als informale Szenarien bei „MSD CloseBarriers “ in der Abb. 2.6 modelliert, dieses MSD im Zusammenspiel mit „MSD RequestEnterAtEndOfTrackSection “ in Abschnitt 2.1.2.1 (Abb. 2.4) spezifiziert wird. Jedes *CrossingControl* hat ein Objekt, das als *barriers* ist. *CrossingControl* und *Barriers* sind durch Verbindung ***barriers*** verbunden. Dieses Objekt arbeitet als einen Sensor, der *Barriers* schließt und entdeckt, ob *Barriers* *closed*, *blocked* oder *opened* ist. Dieses Objekt gehört zu *Umwelt*, weil es ein Befehl empfangen werden kann. Es kann nicht kontrolliert werden, wann *Barriers* geschlossen oder blockiert werden wird.

Weil eine Benachrichtigung empfangen wird, dass *RailCab* die Endpunkt von

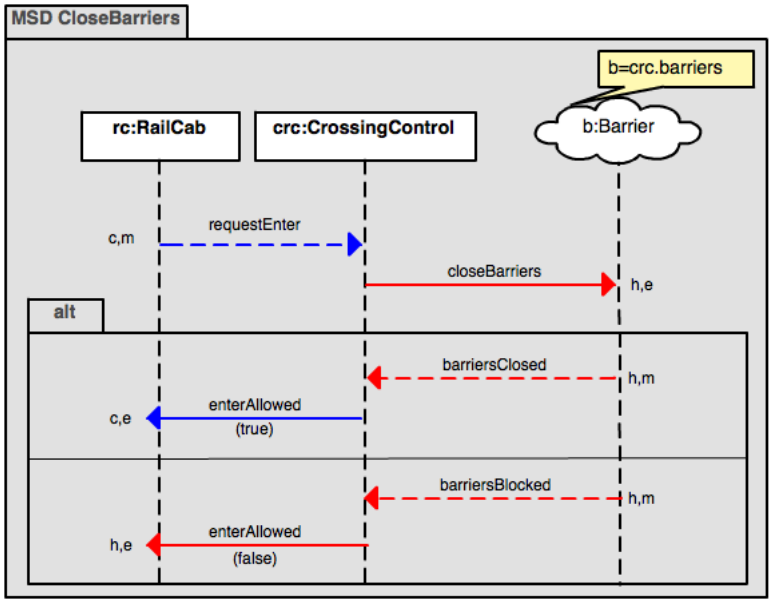

Abbildung 2.6.: MSD definiert Anforderung, um *barriers* zu schließen. In Anlehnung an [GBM13].

aktuellen Laufbahn ankommen ist. „MSD RequestEnterAtEndOfTrackSection“ erfordert, dass *RailCab* an *TrackSectionControl* eine Nachfrage senden wird. In diesem Fall wird *RailCab* der Nachfrage **requestEnter** an *CrossingControl* senden. Klasse *CrossingControl* erbt die Klasse *TrackSectionControl*. Weil die Nachfrage an *CrossingControl* gesendet wird, wird aktive „MSD CloseBarriers“ erstellt.

Nachdem *CrossingControl* der Nachricht **requestEnter** empfangen wurde, muss es eine Nachricht an *Barriers* senden, die als **closeBarriers** genannt wird. Es gibt zwei Möglichkeiten, dass die in diesem Fall geschehen werden können. Sie werden bei *alternative fragment* modelliert. Als *alternative fragment* kann es die Lebenslinien erstrecken, und zwei oder mehrere Unterinteraktionen erhalten. Die Unterinteraktionen werden durch die durchgehende Linien aufgeteilt. Eine nichtdeterminitische Wahlmöglichkeit wird modelliert, ob keine Bedienungen spezifiziert werden. Die erste Nachricht der Unterinteraktionen bestimmt das Untersystem, das die Wahl tun. *System* kann die Wahl tun, wenn die erste Nachricht die Nachricht von *System* ist. Wenn sie die Nachricht von *Umwelt* ist, dann die Wahl bei *Umwelt* getan wird. In der *Umwelt* können viele Ereignisnachrichten geschehen sein, deshalb muss *System* Reaktion auf dieser Ereignisnachrichten haben. Es gibt zwei Situationen in diesem Fall, wenn *Barriers* den Befehl zu schließen empfangen wird. *Barriers* kann entweder geschloßen werden oder blockiert werden, dann soll *CrossingControl* dem *RailCab* entweder eine Erlaubnis oder ein Verbot geben, um die nächste Laufbahn anzukommen. (**enterAllow(true)** ist als eine kalte Nachricht, weil es nicht empfangen werden

kann. Diese Fall kann durch andere MSD spezifizieren, dass wegen anderen Grund *RailCab* nicht anzukommen darf.)

Unglücklicherweise gibt es noch Probleme in diesem Fall. Nachdem *CrossingControl* schon der Nachricht **closeBarriers** gesendet hat, das System auf geschehende Ereignisnachricht von *Umwel* warten muss. Bevor *CrossingControl* diese Benachrichtigung über die Ereignisnachricht von *Umwelt*, das entweder **barriersClosed** oder **barriersBlocked** ist, empfange wird, darf *CrossingControl* nicht die Antwort **enterAllowed** an *RailCab* senden. Es gibt Möglichkeit, dass *Barriers* nicht Ereignisnachricht **barriersClosed** oder **barriersBlocked** senden wird, sodass *Environment* **lastBrake** senden soll. In „MSD RequestEnterAtEndOfTrackSection“ würde eine Sicherheitsverletzung auftreten. Die Anforderungen können erfüllt werden, wenn *Barriers* geschlossen oder blockiert zu sein gemeldet hat, bevor **lastBrake** geschehen ist.

Zweite Problem ist, dass *System* sofort aktive Nachricht von System senden wird, wartet es nicht auf *Umwelt*. Nachdem *CrossingControl* der Nachricht **closeBarriers** gesendet hat, wird er die aktive Nachricht **enterAllowed** vor geschehenden Ereignisnachricht **barriersClosed** oder **barriersBlocked** senden, dann eine Sicherheitsverletzung auftreten wird.

Um diese Probleme überzuwinden, haben Joel Greenyer, Christian Brenner, und Valerio Panzica La Manna in [GBM13] eine Erweiterung von Play-Out Algorithmus vorgeschlagen, dass der Algorithmus das *System* zulassen, um auf die Ereignisnachricht von *Umwelt* zu warten. Sie haben auch MSDs für Annahme vorgeschlagen, die deutlicher die Annahme von *Umwelt* modelliert.

MSDs für Annahme haben gleiche Syntax und Semantik wie in MSDs für Anforderungen, aber sie haben die folgende Unterschieden:

Syntax MSDs für Annahme haben zusätzliche Stereotype ≪assumption MSD≫.

Semantik Wenn eine Reihe von Nachrichten entweder nicht zur Sicherheitsverletzung oder Lebendigkeitsverletzung in jedem MSD für Anforderungen oder zur Sicherheitsverletzung oder Lebendigkeitsverletzung in nur einem MSD für Annahme führt, dann erfüllt diese Reihe den MSD - Spezifikation.

Anwendung MSDs für Anforderungen spezifiziert die Beschränkungen der Nachrichten von *System* oder die Reaktion von *System* auf der Ereignisnachrichten von *Umwelt*. Im Gegenteil spezifiziert MSDs für Annahme die Beschränkungen der Ereignisnachrichten von *Umwelt* oder die Reaktion von *Umwelt* auf die Nachrichten von *System*. Deshalb wird

das System verpflichtet den MSDs für Anforderungen zu erfüllen sein, nur wenn *Umwelt* allen MSDs für Annahme erfüllt.

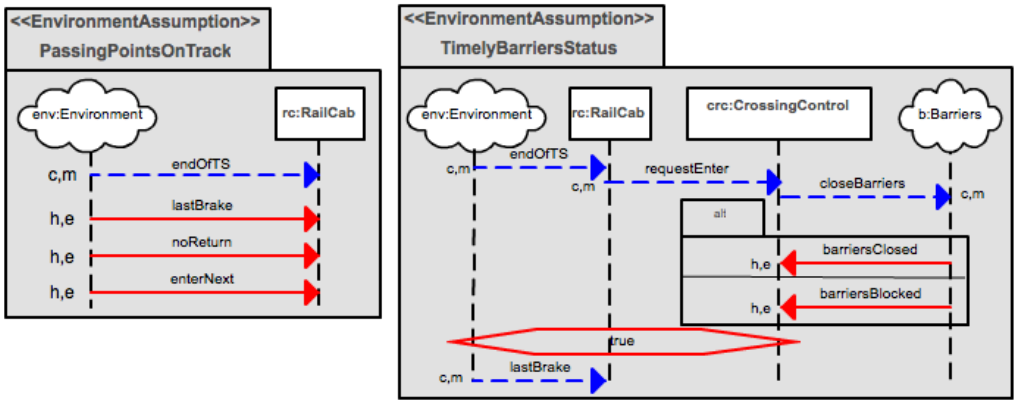

Abbildung 2.7.: MSDs für Annahme des Beispiel in 2.5. In Anlehnung an [GBM13].

In der Abb. 2.7 sind die MSDs für Annahme von Beispiel in 2.5. Die linke MSD für Annahme „PassingPointsOnTrack“ bedeutet, dass *RailCab* die Punkten von Laufbahn auf allen angegebenen Befehle durchgehen wird (*RailCab* wird nicht bremsen oder umdrehen). Die rechte MSD für Annahme „TimelyBarriersStatus“ spezifiziert die Reaktion von *Umwelt* auf Nachricht **closeBarriers**. Bevor *RailCab* dem Punkt von letzte Sicherheitsbremse durchgeht, muss die Reaktion, die als **barriersClosed** oder **barriersBlocked** ist, geschehen sein. Mit diesem MSDs für Annahme kann das System seine Anforderungen erfüllen, und die Spezifikationen von System können erfolgreich durchgeführt werden.

2.2. ScenarioTools

ScenarioTools ist ein Plug-in für Eclipse, das durch mehrere Plugins von Eclipse entsteht. Es ist eine Werkzeugumgebung, mit der Entwickler ein System modellieren und szenariobasierter Spezifikationen analysieren kann. Dafür bietet es einen grafischen Editor, um die MSD - Spezifikationen des Systems zu erstellen und arbeiten. Diese MSD - Spezifikationen sind basiert auf die Erweiterung des Play-out-Algorithmus in 2.2.1. ScenarioTools besteht aus eine Möglichkeit, um die erstellten MSD - Spezifikationen zu simulieren.

2.2.1. Erweiterung von Play-Out-Algorithmus

ScenarioTools implementiert die Erweiterung von Play-Out-Algorithmus, um die MSDs für die Annahme von *Umwelt* zu erstellen und zu simulieren. Diese Erweiterung wird bei Joel Greenyer, Christian Brenner, and Valerio Panzica La Manna in [GBM13] (Abschnitt 4) beschrieben. Sie bietet eine Methode, um Spezifikationen der Annahmen von *Umwelt* (Abschnitt 2.1.2.2) zu modellieren. Nach jeden Schritt sammelt sie die Informationen von jede Ereignisnachricht, die Auswirkung auf System im näschten Schritt haben. Dieser Algorithmus bietet Möglichkeit, um MSDs für Anforderung und Annahme zu analysieren. Deshalb kann Benutzer wissen, zum Beispiel, ob die Nachricht kalte oder heiße ist und MSD zur kalten Verletzung oder Sicherheitsverletzung geführt wird. Mit dieser Information hat dem Benutzer Fähigkeit, um die Folge von Nachrichten zu verstehen und den näschte Schritt zu wählen.

In ScenarioTools entsteht Play-Out-Algorithmus durch die wiederholte Ausführung von Nachrichten, die bei Benutzer gewählt werden. Jede Nachricht hat die Nebenwirkungen, die durch den Aufruf von Funktion **performStep** bestimmt wird. In Abb. 2.8 wird die Überblick von dieser Nebenwirkungen in einem Aktivitätsdiagramm beschrieben.

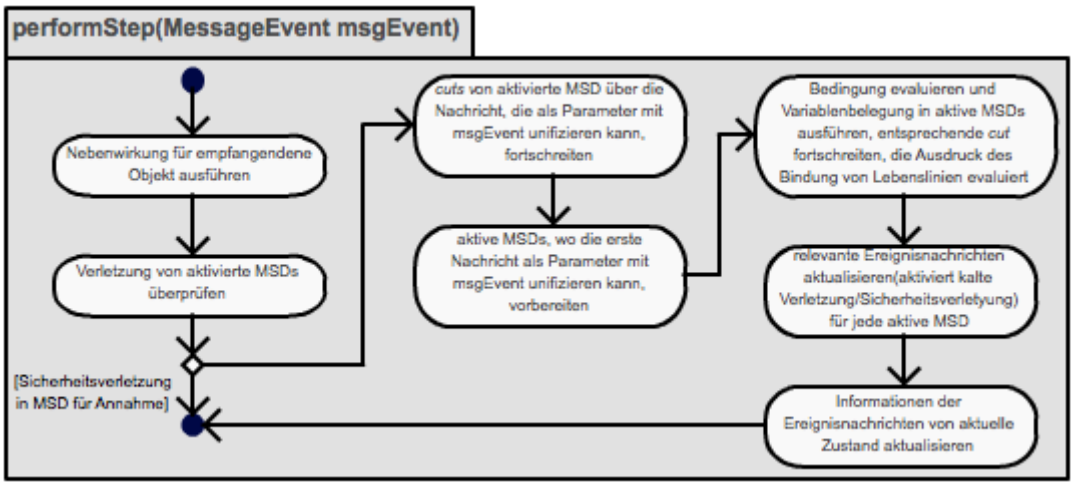

Abbildung 2.8.: Überblick von Funktion **performStep**. In Anlehnung an [GBM13].

Zuerst können die Nebenwirkungen von Ereignisnachricht für das empfangenden Objekt in Objektsystem auftreten. In ScenarioTools wird Konventionsform *set<feature-name>(<value>)* benutzt, mit dem die Attribute oder Beziehungen von Objekten festgelegt werden können. *<feature-name>* bestimmt der Name von Attribute und Beziehungen, und seinen Wert wird bei *<value>* spezifiziert. Der Typ des Parameter von Wert und der Typ von Objektmerkmal müssen immer gleichen sein.

Dann überprüft es, ob jede aktive MSD zur kalten Verletzung oder zur Sicherheitsverletzung geführt werden. Wenn ein aktive MSD zur kalten Verletzung kommt, dann wird es zu terminieren geführt. Wenn die Annahme verletzt werden, und die MSDs für Annahme zur Sicherheitsverletzung kommen, dann wird Play-Out-Algorithmus zu terminiert geführt, deshalb wird weitere Simulation sinnlos sein, dass das *System* seine Anforderungen weiter erfüllen muss. MSDs für Anforderungen kommen zur Sicherheitsverletzung, aber trotzdem führt es nicht dem Play-Out-Algorithmus zu terminieren, weil im Weiteren *Umwelt* die Annahme von Umwelt erfüllen kann. Das bedeutet, dass *Umwelt* das *System* führte, um die Anforderungen zu verletzen, aber es wird auch die Annahme von *Umwelt* zur Verletzung führen. Nachdem MSDs für Anforderung zur Sicherheitsverletzung gekommen ist, wird die Ausführung fortgesetzt, um für diesen Fall zu prüfen. Es muss daran geachtet werden, dass die Anforderung von Sicherheitsverletzung aufgetreten wird. Es gibt auch eine ähnliche Prüfung für die kalte Verletzung, die jeweilige aktive MSD zu terminieren führt.

Näschte Schritt werden aktive MSDs mit *cuts* fortgeschritten. *cuts* werden über die ermöglichte Nachricht von MSDs, wenn diese Nachricht als Parameter mit Ereignisnachricht unifizieren kann, angezeigt. Im vierten Schritt werden aktive MSDs erstellt, wenn Ereignisnachricht als Parameter mit ersten Nachricht von MSDs unifizieren kann. Danach wird jede ermöglichte Anweisungen ausgeführt, und die Bedienungen werden evaluiert. Außerdem werden die Ausdrücke des Bindung von Lebenslinien evaluiert, wodurch sie zu neuen Bindungen von Lebenslinien führen können. Wenn *cuts* noch nicht fortgeschritten wird und keine neue Lebenslinie wird gebunden, dann wird diese Schritt wiederholt. In sechsten Schritt, wird es für jede aktive MSDs determiniert, welche Ereignisnachrichten aktiviert sind, und welche kann zur Sicherheitsverletzung oder zur kalten Verletzung führen.

Zuletzt, nach der Bearbeitung sammelt die Funktion die globale Informationen über Ereignisnachricht in der Zustand. Ereignisnachricht kann wie folgende sein:

- Annahme/Anforderungen (*assumption/requirements*)
- kalte/heiße (*cold/hot*)
- *monitored/executed/aktiviert*
- Annahme mit kalten Verletzung oder Sicherheitsverletzung (*assumption safety/cold violating*), wenn es entsprechende Status in zumindest einem aktiven MSD für Annahme
- Anforderungen mit kalten Verletzung oder Sicherheitsverletzung (*requirements safety/cold violating*), wenn es entsprechende Status in zumindest einem aktiven MSD für Anforderung

Wenn eine Ereignisnachricht die Anfangsnachricht zumindest von einer MSD für Annahm/Anforderung ist, dann ist sie als *assumption/requirements initializing*. Um die globale Informationen über Ereignisnachricht zu sammeln, wird die Informationssammlung über alle aktivierte und verletzte Ereignis von allen aktiven MSDs und die Ereignisse, die aktive MSDs vorbereitet, wiederholt. Je nach der globalen Informationen über Ereignisnachricht werden diese Nachrichten und der Annotationen dieser Nachrichten zu einer Liste hinzugefügt, die als *annotation list* genannt. Wenn ein Ereignisnachricht schon in *annotation list* war, dann werden die Annotationen von Ereignisnachricht aktualisiert. Ingesamt gibt es 14 unterschiedliche Flaggen, mit den jede Ereignisnachricht in ScenarioTools dekoriert.

Es braucht einer besonders Sorgfalt, um die Informationen über Ereignisnachricht in der Anwesenheit von parametrierten und symbolischen Nachrichten zu sammeln, insbesondere wenn entsprechende symbolische und konkrete Ereignisnachricht gleichzeitig aktiviert werden können. Entsprechend (*corresponding*) bedeutet, dass diese Nachrichten ähnliche Name haben, trotzdem tragen sie unterschiedliche Wert von Parameter (oder sie haben nicht Parameter). Die aktivierte parametrierte und konkrete oder symbolische Ereignisnachricht, die zur *annotation list* hinzugefügt und aktualisiert wird, muss die folgende Regele anpassen:

1. **Aktivierte Nachricht ist parametriert und konkret:**
    a) Wenn konkrete Ereignisnachricht noch nicht in der *annotation list* ist, dann kann diese Nachricht hinzufügt werden. Die Annotationen von aktivierten Nachricht werden gesetzt und aktualisiert.
    b) Die entsprechende symbolische Ereignisnachricht wird bei einem Eintrag repräsentiert. Dieser Eintrag wird zur *annotation list* addiert, wenn er noch nicht in *annotation list* enthaltet wird. Zum Beispiel, wird **enterAllowed(?)** zur Liste addiert, wenn **enterAllowed(true)** aktiviert ist.
    c) Die konkrete Ereignisnachricht erben die existierte Annotationen in *annotation list*, die zur entsprechende symbolische Ereignisnachricht gehören. Zum Beispiel, wird **enterAllowed(?)** schon in *annotation list* enthaltet, und er ist *hot + executed* in einigen MSD für Anforderung, dann erbt die Ereignisnachricht **enterAllowed(true)** diese Annotationen, wenn er zu *annotation list* addiert wird. Der Grund für diese Regel ist, dass das Auftreten von jeder entsprechenden konkreten Ereignisnachricht die aktivierte symbolische Nachrichten fortschreiten wird.
    d) Die entsprechende symbolische Ereignisnachricht wird als kalte Verletzung oder Sicherheitsverletzung einer MSD für Annahme oder

Anforderungen gesetzt, wenn die konkrete Nachricht kalte oder heiß in einer MSD für Annahme oder Anforderungen aktiviert wird. Deshalb werden alle Kommentaren für andere entsprechende konkrete Ereignisnachricht aktualisiert.

2. **Aktivierte Nachricht ist parametriert und symbolisch:**
   a) Die symbolische Ereignisnachricht wird bei einem Eintrag repräsentiert. *annotaion list* addiert dieser Eintrag, wenn sie nicht ihn enthaltet. Die Annotationen von aktivierten Nachricht werden gesetzt und aktualisiert.
   b) Wenn *annotation list* die entsprechende konkrete Ereignisnachrichten enthaltet, dann erben diese Ereignisnachrichten die Annotationen, die für die symbolische Ereignisnachricht erstellt werden.

Diese Regel werden bei Joel Greenyer, Christian Brenner, and Valerio Panzica La Manna in [GBM13] (Abschnitt 4) beschrieben. In ScenarioTools werden diese Informationen in zwei unterschiedliche Play-Out-Modus verwendet. Mit dem ersten Modus muss *System* immer sofort die aktive Systemmeldungen senden. Zum Beispiel, in der Anwesenheit von MSDs für Annahme führt Play-Out die Systemmeldungen, wenn sie als aktive Anforderungen sind. Im zweiten Modus bietet es dem *System* eine Entscheidung, um die aktive Ereignisse von *Umwelt* zu warten, wenn die aktive Systemmeldungen existieren.

2.2.2. Realisierung

ScenarioTools enthaltet viele Plug-ins von Eclipse (z.B Eclipse Modeling Framework (EMF), Unified Modeling Language (UML), Papyrus, usw.). Das Programm basieren auf dem Eclipse Modeling Framework (EMF). Die MSDs von Modelle werden durch einen grafischen Editor bearbeitet und visualisiert, die auf

UML Editor Papyrus basiert. Das Programm bietet eine Benutzeroberfläche, die basiert auf Debug-Framework von Eclipse ist, deshalb fühlt Benutzer sich familiär. Abb. 2.9 zeigt den Screenshot von Benutzeroberfläche der Simulation in Eclipse - Perspektive ScenarioTools.

In der erste Position steht Debug - Ansicht, die aktuelle Zustand der Laufzeitumgebung von Simulation für die Spezifikationen des Beispiel 2.1.1 zeigt. Es listet sowohl die aktuelle aktive MSDs für Annahme und Anforderungen als auch die aktuelle Position von *cut* auf. Jede aktive MSD zeigt die aktuelle Lebenslinien in einer Form *[lifeline->object]*, und wird die Position von *cut* auf jede Lebenslinie mit der Form *lifeline:index* gezeigt. In der Liste lassen alle Objekten

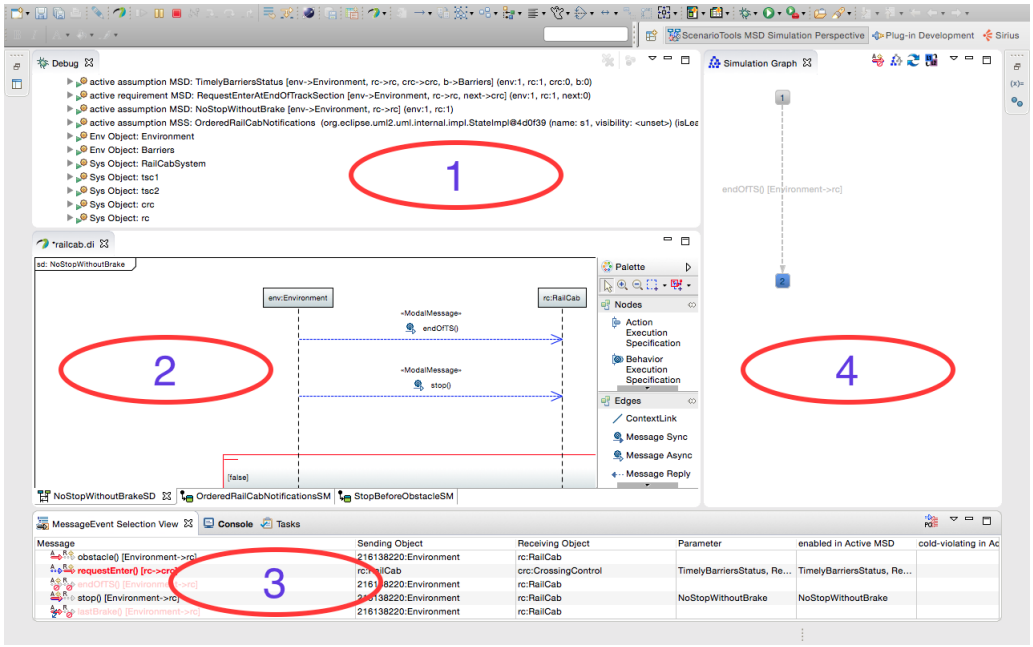

Abbildung 2.9.: Der Screenshot von Benutzeroberfläche der Simulation in ScenarioTools.

von Objektsystem, die simuliert werden, sich nach der List von MSDs zeigen.

Die zweite Position zeigt ein MSD. ScenarioTools erweitern Papyrus Plug-In, um die Temperatur (blau/rot Farbe) und ausgeführte Typ (durchgehende/gestrichelte Pfeil) von Nachrichten zu zeigen.

Position drei ist die Ereignisnachricht- Ansicht, die aktuelle aktivierte, verletzte oder initialisierende Ereignisnachrichten zeigt. In der aktuellen Zustand sind alle ausgegraute Ereignisnachrichten als Sicherheitsverletzung für Anforderungen. Die rote Ereignisnachricht ist die initialisierende für Anforderungen. Linke Seite des Name gibt es Zeichen, das Visualisierung von Flaggen für Annahme und Anforderungen ist. Die Ereignisnachrichten haben ähnliche sendende und empfangende Objekt, und die Verweisung auf ähnlichen Funktion, trotzdem haben sie unterschiedliche Wert von Parameter. Diese Nachrichten werden für parametrierte Nachrichten als Gruppe zusammengefasst.

In der letzten Position (4) werden alle ausgeführte Nachrichten gezeigt. Die Knoten sind die Ordnung in der Reihenfolge von ausgeführten Nachricht des Objektsystem. Die Verbindung zwischen Knoten zeigt die ausgeführte Nachricht, die mit Name, sendende und empfangende Objekt in einer Form *name [sending object -> receiving object]* repräsentiert wird.

2.3. Eclipse Modeling Framework

Spezifikation von Software können durch viele Mölichkeiten (z.B. Java-Interface-Klassen, UML-Diagram, XML-Schema, etc.) modellieren und speichern. Die Daten von diese Modellierung repräsentieren ähnliche Informationen. Es gibt eine Nachfrage nach eine Möglichkeit, um Java-Interface-Klassen, UML-Diagram und XML-Schema zu vereinigen. Am 25 September 2007 wird die erste Version von Eclipse Modeling Framework (EMF) bei „Eclipse Foundation“ [Fou04] ausgegeben.

EMF basiert auf Meta Object Facility (MOF) von Object Management Group (OMG). Es ist eine Modellierung-Framework und Codegenerierung für Eclipse in Java-Sprache. Mit EMF kann eine Modellierung in andere Form konvertieren, um z.B. Modellierung in Ecore Modell in Java-Interface-Klasse zu generieren. In EMF kann die Spezifikationen von Modellierung durch vielen Format importieren, dass z.B. Spezifikationen durch XML-Schema oder Java-Annotation importieren kann.

2.3.1. Grundlage

Alle Modellierung, die bei EMF modelliert und spezifiziert, werden als Ecore Modell genannt. Ecore Modell enthält die Spezifikationen der Daten von Software, die sind:

- Attributen von Objekten.
- Die Beziehungen (Verbindungen) zwischen Objekten.
- Gültige Operation von jedem Objekt.
- Die Beschränkungen des Objekt und der Beziehungen.

Ecore Modell entsteht durch die Untermenge von Ecore Metamodell, die in Buch „EMF: Eclipse Modeling Framework“ von Dave Steinber, Frank Budinsky, Marcelo Paternostro und Ed Merks [SBPM09] (Abschnitt 5.2, S. 105-106) beschrieben werden. Wegen diesem Grund wird ein Ecore Modell als auch ein Meta-Metamodell sein.

Die Abb. 2.10 [1] ist der Kern von Ecore Modell, die durch vier Basisklassen von EMF entsteht. Diese Basisklassen definiert die grundlegende Verhalten

[1] Quelle der Abb. 2.10 kann in S. 105 des Buches „EMF: Eclipse Modeling Framework“ gefunden werden.

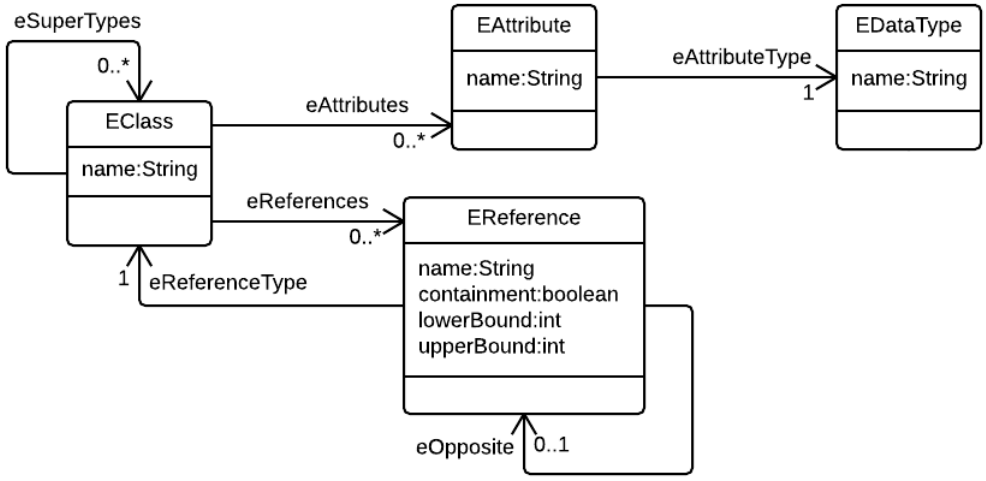

Abbildung 2.10.: Untermenge von Ecore Metamodell

des System, die als Objekte, Beziehungen zwischen Objekte, die Attribute der Objekte und Datentype sind.

Auf jeden Fall braucht ein Ecore Modell mindestens sechs Basisklassen , um ein System in EMF zu modellieren. Die Eigenschaften dieser Klassen sind wie unten:

1. EPackage: wird die Package von Modell repräsentieret. Es kann ein Name, null oder mehr EClass, null oder mehr EAnnotation, null oder mehr EENum, und null oder mehr EDataType haben.
2. EClass:wird die Klassen von Modell repräsentiert. Je Klasse kann ein Name, null oder mehr Attributen, null oder mehr Funktionen, und null oder mehr Referenz haben.
3. EAttribute:wird die Attribute von Modell repräsentiert. Attributen haben ein Name und ein Type.
4. EOperation:wird die Funktionen von Modell repräsentiert. Je Funktion hat ein Name und ein Type.
5. EReference:wird die Beziehung zwischen Klassen repräsentiert. Es hat ein Name, ein logischer Begriff für Repräsentation von Containment, und ein Referenz Type.
6. EDataType:wird das Type von Attribute repräsentiert. Datentypen können als int oder float als auch Type des Objekt wie java.util.Integer.

Ecore enthält die gemeinsame Klasse für die Klassen, die gemeinsame Aspekte haben. Zum Beispiel, haben die Klassen EClass, EDataType, EAttribute und EReference (diese Klassen sind in der Abb. 2.10) in Kern von Ecore

Modell gemeinsame Attribute, die als „name“ genannt. Diese Attribute wird in Basisklasse *ENamedElement* wird. Die EClass und EReference haben auch einige gemeinsame Aspekte, dafür Ecore ein gemeinsame Klasse für diese Klassen enthält, die als EStructuralFeature gennant. Ecore hat fünf gemeinsame Klassen, die EModeElement, ENamedElement, EClassifier, ETypeElement und EStructuralFeature sind. Es gibt noch zehn Klassen, um die Spezifikation von Objekt, System und Beziehungen zu modellieren. Diese Klassen sind EClass, EPackage, EDataTyp, EOperation, EParameter, EReference, EAttribute, EEnumLiteral, EEnum, und EObject. Die Instanzen von der Klassen des System, die in Ecore Modelle modelliert werden, werden bei EFactory erstellt.

Ecore haben insgesamt sechzehn Basisklassen, die wie Abb. A.1 [2] zusammengehängt werden. Die Zusammenhängen zwischen die Klassen des Ecore Modell wird in der Eclipse Dokumentation mit Title „The Eclipse Modeling Framework (EMF) Overview “ [Cen05] geklärt. Die Wurzel des Ecore Modell ist Objekt, das gleiche Eigenschaften als java.lang.Object hat.

Datentype repräsentiert Einzelteil von einfachen Inhalten des System und der Komponenten. In der Ecore Modell modelliert EDataType die Typen des Javasprache, die nicht nur primitive Type (int, boolean, char, usw.), sondern auch Java-Klasse, Java-Interface und auch Array sind. Es ermöglicht einfache Klassen wie String, Integer, Boolean als Einzelheit zu repräsentieren. Tabelle 2.2 enthält einige Ecore Data Type.

Type des Ecore Modell	primitive Java Type oder Java Class
EBigDecimal	java.math.BigDecimal
EBigInteger	java.math.BigInteger
EBoolean	boolean
EByte	byte
EByteArray	array[]
EChar	char
EDouble	double
EFloat	float
EInt	int
EJavaObject	java.lang.Object
EString	java.lang.String

Tabelle 2.2.: List von Ecore Data Type.

[2] Quelle der Abb. A.1 ist `http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html`

Ecore Modell kann durch ein andere Ecore Modell importieren, deshalb es kann die Ressource von anderen Ecore Modell im Arbeitsbereich oder andere Ressource URIs (wie ein Dateisystem, angemeldet Paket, usw.) laden. Zum Beispiel, wenn man Modelle für Beispiel 2.1.1 erstellt, kann man die Ressource von Ecore.ecore und die andere erstellte Modelle läden. Um ein Konzept für das entwickelte System zu erstellen, wird Ecore Meta-Modell dynamische Instanz generiert. Mit dieser Instanz kann man ein Beispiel für ein Objektssystem erstellen, die die Anzahl und die Beziehungen der Komponenten des Systems repräsentiert, deshalb kann Entwickler jetzt die Struktur des Systems analysieren und bearbeiten.

2.3.2. Persistence

Eclipse Modeling Framework (EMF) bietet eine krätige Framework, um die Persistenz während Modellierung zu bearbeiten. In Eclipse Modeling Framework (EMF) wird die grundlegende Einheit von der Persistenz als *Resource* genannt. Die Objekte von Eclipse Modeling Framework (EMF) werden durch Interface Klasse **Resource** bearbeitet, und nach der Bearbeitung werden sie zur List des Inhalt von *Resource* durch Methode **save()** hinzugefügt. Um die existierte *Resource* von einem Modell zu bearbeiten, muss die List der Inhalten zuerst durch Methode **load()** geladen. Zum Beispiel wird die Ressource von Modellierung des Objektsystem RailCab wie in der Folgende erstellt:

```
1  ResourceSet resourceSet = new ResourceSetImpl();
2  URI uri = URI.createURI("/DriveOntoCrossing/test.rc");
3  Ressouce resource = resourceSet.create(uri);
4  RailCabSystem rcs = ObjectSystemFactory
5                  .Instance.createRailCabSystem();
6  try{
7          resource.load(null);
8          resource.getContents().add(rcs);
9          resource.save(null);
10 } catch {IOException e{
11         e.printStackTrace();
12 }
```

Listing 2.1: Beispiel von Bearbeitung mit Persistenz in Eclipse Modeling Framework (EMF)

In diesem Beispiel werden die Codes von Zeile ein bis Zeile vier die Vorbereitung gewesen, die die Definition von *Resource* und die Inhalte von Objekt sind.

Von Zeile 6 bis Zeile 9 wird System zuesrt die List der Inhalten von *Resource* zu laden versucht, dann wird die Inhalte von Objekt *rcs* zur List der Inhalten hinzugefügt. Der Versuch wird mit dem Abspeichern von Resource beendet. Wenn der Versuch nicht funktioniert werden kann, dann wird das System eine Warnung ausgegeben.

2.4. Sirius

Sirius ist ein Eclipse - Projekt, das die Technologien von Eclipse - Modellierung (inklusive EMF und GMF) aufgebaut wird. Am 06.Juni.2013 präsentieren Obeo und Thales die Demo-Version von Sirius auf „EclipseCon France 2013 in Toulouse“. Das Projekt [OT13b] wird offiziell am 13.Juni.2013 erstellt. Es bietet eine Werkzeugsumgebung, mit der Entwickler einen Arbeitstisch für die grafische Modellierung erstellen kann.

Der Arbeitstisch für Modellierung wird mit einer Menge von Eclipse - Editoren (*diagrams, tables und trees*) zusammengestellt, damit Benutzer EMF Modelle erstellen, bearbeiten und visualisieren kann. Die Editoren des Arbeitstisch werden durch ein Modell definiert, mit dem die Struktur der Arbeitstisch, seine Verhalten, alle Versionen und auch Navigationswerkzeug definiert werden. Die Spezifikationen des Arbeitstischs werden bei Runtime in Eclipse - IDE interpretiert. Sirius bietet viele Möglichkeiten, um die Spezifikationen anzupassen. In Sirius kann Benutzer neue Repräsentationen, neue Abfragesprachen für seine Arbeitstisch bieten, und auch die Javacode mit Eclipse oder andere System zu interagieren aufrufen. Diese Prinzipien wird in der Website des Sirius Projekt [OT13b] erklärt.

2.4.1. Sirius Grundlagen

Sirius ist eine Werkzeugsumgebung, mit der grafische Editoren für jede Domain entwickelt und benutzt werden können. Mit Sirius kann man nur grafische Editor für jede Domain, die mit EMF beschrieben wird, erstellen. Die Benutzer des Sirius werden in zwei Kategorien unterteil:

Architects sind die Benutzer, die grafische Editoren für ihre spezifische Domains mit Sirius entwickeln. Diese Domains müssen mit *EMF's Ecore* definiert werden.

End Users sind die Benutzer, die den erstellten grafischen Editor von Architects benutzen. Sie können Modelle für ihre Domains mit diesem Editor erstellen, anzuschauen, bearbeiten und manipulieren.

Abb. 2.11 zeigt die Hauptarchitektur von Sirius. Sirius verwendet Eclipse - Plattform, deshalb kann Sirius die Menge von Framework und gemeinsame Service in Eclipse benutzen. Mit EMF kann die spezifische Domains als Ecore Meta - Modelle definiert werden, oder als Javacode mit *EMF's Ecore* exportiert werden. Das System von Sirius wird in zwei Hauptteile gliedert.

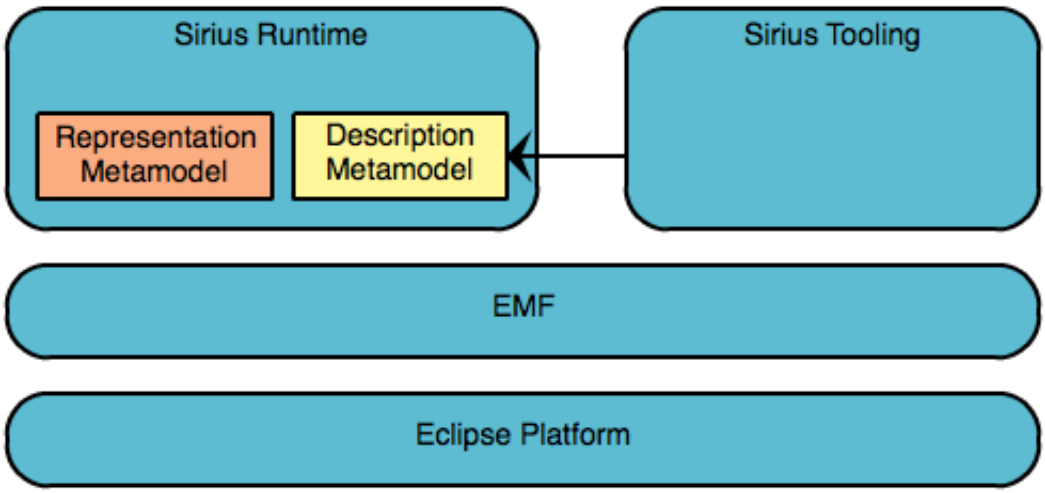

Abbildung 2.11.: Hauptarchitektur von Sirius Plug-in. In Anlehnung [OT13a].

Der erste Teil „Sirius Tooling“ wird bei **Architects** verwendet. Es bietet eine Umgebung, mit der die **Architects** die grafische Editoren für **End - Users** spezifiziert. Die Spezifikation ist geschehen, wenn eine Modellbeschreibung (*description model*) erstellt und eingerichtet wird. Für jeden grafischen Editor muss **Architect** für diese Modellbeschreibungen spezifizieren:

1. Welches Element wird im grafischen Editor repräsentiert werden? Das bedeutet, welche Elemente von Modell der Domain als einsehbar Objekt im grafischen Editor sein sollen.
2. Wie sehen diese Elemente im grafischen Editor aus? Das heißt, welche Informationen des Styling für diese Elemente im grafischen Editor visualisiert werden.
3. Wie funktionieren diese Elemente im grafischen Editor? Das bedeutet, welche Tools und Modi von Interaktion verfügbar für **End - Users** sind, um ihre Modelle zu modifizieren, und wie funktioniert jedes Tool.

„Sirius Tooling“ bietet eine vollständige Umgebung für **Architects**, die leicht zu benutzen ist, um die Spezifikationen zu erstellen. Das Ergebnis ist eine Menge

von Modellbeschreibungen. Es gibt Möglichkeit, dass das Ergebnisse mit einige zugehörige Java-code, das als Eclipse Plug-in eingesetzt werden kann.

Der zweite Hauptteil „Sirius Runtime“ wird bei **End - Users** benutzt, dieser Teil den erstellten grafischen Editor von **Architects** enthältet. Sirius Runtime werden die Modellbeschreibungen von grafischen Editor interpretieren, und präsentieren die Visualisierung und Funktion des jeweiligen grafischen Editor für Benutzer. Diese Visualisierung und Funktion wurden mit Modellbeschreibungen spezifiziert. In Abb. 2.12 zeigt die Interaktion von Sirius Runtime.

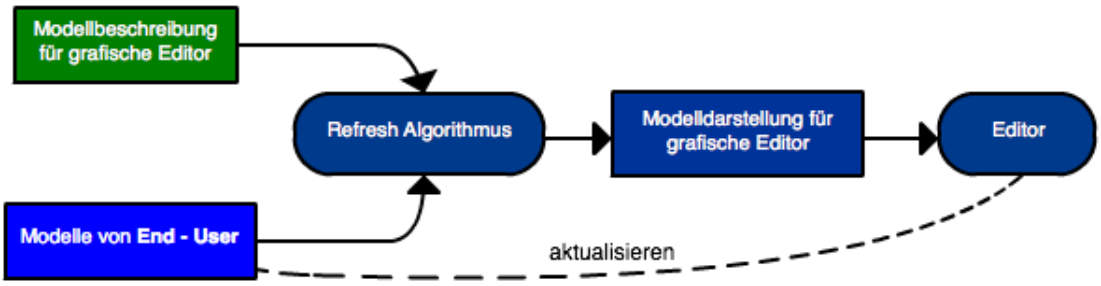

Abbildung 2.12.: Die Interaktionen von Sirius Runtime. In Anlehnung [OT13a].

Dafür benutzt Runtime eine Modelldarstellung (*representation model*), mit der die konkrete Repräsentationen für Modelle von **End - Users** nach der Einstellung der Spezifikationen für grafischen Editor beschrieben werden. Runtime erstellt diese Modelldarstellungen mit *refresh* Algorithmus, mit der die Projektion für die Daten der Modelle von **End - Users** auf Modelldarstellungen mit der Regeln, die in Modellbeschreibungen definiert werden, hergestellt wird.

Die Modelldarstellungen werden für **End - Users** mit einem grafischen Editor, der im Arbeitstisch Eclipse integriert wird. Mit diesem Editor kann **End - User** seine Modelle mit der Regeln, die bei **Architects** in Modellbeschreibungen definiert werden, visualisieren. **End - User** kann mit seinem repräsentierten Modell interagieren, trotzdem kann er nur mit der Einstellung bei **Architect** machen. Wenn Benutzer mit seinem repräsentierten Modell im Editor interagiert, steuert manche Interaktionen die definierte Verhalten des grafischen Editor an. Diese Verhalten wird das Modell von **End - User** modifizieren, trotzdem muss die Modifizierung die definierte Regeln des Modell anpassen. Nach der Modifizierung des Modell startet Sirius den *refresh* Algorithmus neu, dannn aktualisiert es das existierte Modell.

Im Standard unterstützt Sirius drei Arten für Repräsentation, die *diagrams, trees, und tables* sind. Jede Art wird bei einem bestimmten *Dialect* bearbeitet. Jedes *Dialect* bietet seine spezifische Extensionen für Architekturelementen im Kern:

1. Modelldarstellung wird mit der Konzepte von *Dialect* erweitert. Zum Beispiel, kann ein Diagramm Knoten und Verbindungen zwischen diesen Knoten haben.
2. Der Kern von Modellbeschreibung wird erweitert, um die Definitionen für die spezifische Konzepte zu unterstützen. Deshalb muss die Umgebung von *Tooling* erweitert werden, damit **Architects** diese Konzepte spezifizieren können.
3. Jedes *Dialect* hat einen *refresh* Algorithmus, diese Algorithmen unterschiedlich für alle *Dialects* sind. Sirius bietet einige gemeinsame Code, um diese Algorithmen zu unterstützen.
4. Jedes *Dialect* muss ein oder mehreren Editoren implementieren. Diese Editoren werden ihre Repräsentationen für **End - Users** presentieren, und **End - Users** können mit diesem Repräsentationen interagieren.

2.4.2. Sirius Meta - Modelle

In Sirius müssen die Modellbeschreibungen in einer Gruppe, die als *Viewpoint Specification Model* genannt wird, organisiert werden. Sirius benutzt EMF Modelle, um die Daten von *Viewpoint Specifiacations Models* und Modelldarstellung zu speichern. Die *Viewpoint Specifications Models* werden in ***.odesign** Datei gespeichert, und die Daten von Modelldarstellung werden in ***.aird** Datei gespeichert.

Die Struktur von **Viewpoint Specifications Models** wird bei verschiedene Meta - Modelle definiert. In Dokument von Sirius [OT13a] haben Obeo und Thales die erforderliche Packeten, um **Viewpoint Specifications Models** zu erstellen. Sirius enthält Ecore Packet *http://www.eclipse.org/sirius/1.1.0* und vier Unterpacketen, um die Hauptstruktur zu definieren. Jedes *Dialect* definiert eine Erweiterung von diesen Packeten für es. In dieser Arbeit wird der Fokus auf Beschreibung über Diagramm gelegt, deshalb brauchen wir vier unten Ecore Packeten:

1. *http://www.eclipse.org/sirius/diagram/1.1.0* definiert seine Unterpacketen, die die Beschreibung über Diagramm in einem *Viewpoint Specification Model* zu spezifiziert verwendet werden.
2. *http://www.eclipse.org/sirius/diagram/description/1.1.0* definiert die Gesamtstrukture. Gruppe, *Viewpoint* und der abstrakte Typ der Darstellung für die Beschreibung sind die Erweiterungen für *http://www.eclipse.org/sirius/description/1.1.0*.
3. *http://www.eclipse.org/sirius/diagram/description/style/1.1.0* definiert sowohl die abstrakte und gemeinsame Styling

für Beschreibung als auch dazugehörigre Typen, die von *http://www.eclipse.org/sirius/description/style/1.1.0* erweitert werden.

4. *http://www.eclipse.org/sirius/diagram/description/tool/1.1.0* definiert gemeinsame abstrakte Tools für Beschreibung und die gemeinsame Typen von Tool. Dieses Packet hat auch die Definitionen von der Funktionen des Modells, wie zum Beilspiel *If*, *CreateInstance*, usw. Diese Definitionen werden von *http://www.eclipse.org/sirius/diagram/description/tool/1.1.0* erweitert.

In Sirius kann Modelldarstellung Datei (***.aird**) durch eine Menge von serialisierte Version in einer Session. Diese Datei enthält sowohl die Verweise auf die semantische Modelle, die in der Session repräsentiert werden, als auch das Modell für jede jeweilige Darstellung. Diese Modelldastellungen werden in einer Gruppe, in der alle Modelldarstellung mit ähnlichen *Viewpoint* definiert werden, organisiert. Modelldastellung Datei speichert auch eine Menge von verfügbaren Viewpoints in der Session. Deshalb gibt es in der Modelldarstellung Datei die Daten von Darstellungen und Session, die Verweise auf die semantische Modelle, und auch Verweise auf die *Viewpoint Specifications Models*, die die Darstellungen in der Datei definieren. Die Meta - Modelle von Darstellung für Diagramm wird in *org.eclipse.sirius.diagram* definiert.

Jede **.aird* Datei hat die folgende Struktur: Resources von **.aird* Datei enthält ein Element, das als *DAnalysis* definiert wird. Ein *DAnalysis* wurden mit sechs EReference definiert. DAnalysis kann mit allen obersten Ebene von semantischen Resourcen in der Session verbinden, diese Verbindungen durch EReference *models* bestimmt werden. Es kann auch mit andere Untermodelldarstellungen verbinden, die mit EReference *referencedAnalysis* bestimmt werden. Wenn ein *Viewpoint* schon ein Mal in der Session aktiviert wurde, dann enthält *DAnalysis* ein *DRepresentationContainer* durch EReference *ownedViews* für jedes *Viewpoint*. *DAnalysis* zeigt durch EReference *selectedViews* eine Untermenge von *ownedViews*, diese die Menge von aktuellen aktivierten *Viewpoints* in der Session sind. Jedes *DRepresentationContainer* enthält die jeweilige Repräsentationen durch EReference *ownedRepresentations*, und verbindet mit einem *Viewpoint*, das für die Repräsentationen benutzt wird, durch EReference *viewpoint*. Weil der Fokus auf Diagramme gelegt wird, ist die Type von *ownedRepresentations* in dieser Arbeit *DSemanticDiagram*.

Jede **.aird* Datei benutzt *org.eclipse.sirius.viewpoint.DSemanticDecorator* für *ownedDiagramElements*, die eine Verbindung mit semantischen Modelle haben. Mit der Funktion *getTarget()* von *DSemanticDecorator* kann das semantische Element, das aktuell representiert wird, bekommen werden.

Kapitel 3.

Anforderungsanalyse

In diesem Kapitel möchte ich zuerst die Struktur von ein Projekt, das mit ScenarioTools erstellt wird, untersuchen. Danach führe ich die Anforderungen inklusive Qualitätsanforderungen durch. Im letzten Abschnitt 3.3 stelle ich die Möglichkeiten, die sich aus diesen vorhandenen Strukturen ergeben, für die Umsetzung der Anforderungen vor.

3.1. Systemsbeschreibung

Nach der Systemanalyse von ScenarioTools haben sich die folgende logische Zusammenhänge für die einzelne Komponenten von System ergeben, diese Zusammenhängen in Abb. 3.1, die in Anlehnung an [Gut14] ist, angezeigt wird. Hier wird die Struktur von ScenarioTools in einem Modell, das drei abstrakte Komponenten von ScenarioTools enthält, dargestellt. Bevor die Simulation druchgeführt werden kann, müssen das Verhalten und die Struktur des Objektsystems in Meta - Modelle modelliert werden. In (1) wird die UML - Diagramme (für MSD - Spezifikationen) gezeigt. Diese Komponente enthält die Beschreibungen über das Verhalten und die Struktur des Systems, die in UML spezifiziert werden. Danach werden die Teile aus dem UML - Modelle in einem dynamischen Objektsystem und einem Strukturmodell überführt (2). Das Strukturmodell definiert sowohl die Objekte, die simuliert in Simulation werden, als auch die Klassendiagramme, die diese Objekte beschreiben. Dieses Modell basiert auf Ecore - Modell. Der dynamische Objektsystem wird mit *Dynamic Instance* definiert, dieses Modell basiert auf XML Metadata Interchange (XMI) ist. Laufzeitmodell verwendet diesen dynamischen Objektsystem, um die Simulation oder Synthese aufzubauen (3). Wenn die Simulation oder Synthese ausgeführt wird, lädt Laufzeitmodell den aktuellen Zustand der Simulation und auch die ermögliche Folgezustände aus den MSD - Spezifikationen in UML - Diagramme (4). In der Synthese wird das Zwischenergebnis durch *Stategraph* bestimmt,

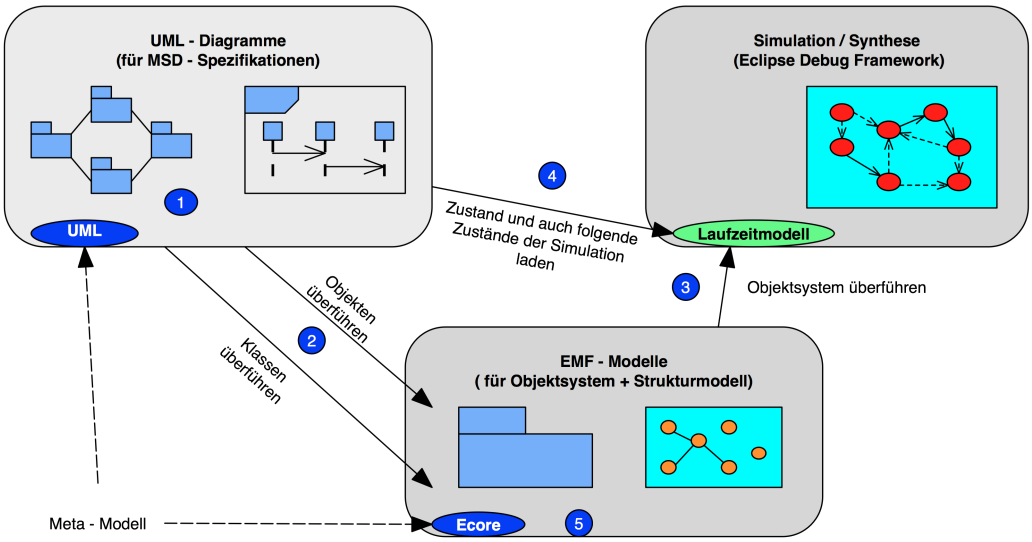

Abbildung 3.1.: Die Zusammenhänge der einzelnen Systemkomponenten von ScenarioTools. In Anlehnung [Gut14].

werden alle erkundete Zustände in diesem Zustandsgraph gespreichert. Die Synthese versucht die Strategie, die eine Kette von Ereignissen, zu finden. Der Controller bildet eine Untermenge von diesem Zustandgraph inklusive der Kette von Ereignissen. Die Strategie kann nicht ausgeführt werden, weil es noch nicht Interpreter gibt. Die Simulation und der Synthese befinden sich in verschiedenen Modell, deshalb kann sie nicht auf die Strategie zugreifen. Eine Lösung für dieses Problem wird bei Timo Gutjahr in seinem Bachelorarbeit [Gut14] geschrieben. Er bietet eine neue Transformation *strategy2mss*, um Strategie auf ein MSS abzubilden. Für die Simulation wird das Zwischenergebnis durch *MSDRuntimeStategraph* bestimmt, die eine Erweiterung von *Stategraph* ist.

Die Dateien aus der Speicherebene können auf jeden Fall der Abb. 3.1 zuordnen. Im Folgenden werden die während der Analyse erzeugten Dateien aufgezeigt:

***.di** Die Papyrus Datei beschreibt grafische Diagrame für MSD - Spezifikationen des Objektsystems. Diese Datei wird vom Benutzer erstellt und bearbeitet.

***.uml** Die UML Datei (1) in 3.1 enthält alle MSD - Spezifikationen des Objektsystems. Diese Datei wird auch vom Benutzer erstellt und bearbeitet.

Folgende Dateien können von *Create UML-to-Ecore TGG Interpreter Configuration* erzeugt werden:

***.interpreterconfiguration** Diese Datei enthält die Interpreter Konfiguration für TGG - Transformation (*uml2ecore*), damit TGG - Transformation (*uml2ecore*) durchgeführt werden kann. Dabei werden die Klassendiagramme aus der UML - Modelle abgeleitet. Das Ergebnise wird auf ***.ecore** Datei gespeichert.

***.corr.xmi** Diese Datei (2) in 3.1 bildet den Korrespondenzgraphen der TGG - Transformation (*uml2ecore*) ab, mit dem UML-Modelle in Ecore-Modelle übergeführt werden.

***.ecore** Diese Datei (5) in 3.1 ist ein Ergebnis der TGG - Transformation (*uml2ecore*), deshalb enthält sie die Struktur der Objekte von Objektsystem. Sie bildet die Beschreibung der Objekte für das Laufzeitmodell.

***.roles2eobjects** In dieser Datei (2) in 3.1 ist die Abbildung für dynamische Objekten von UML auf ECore gespeichert.

***.xmi** In dieser Datei (5) in 3.1 wird der dynamischen Objektsystem gespeichert. Die Objekte von diesem System werden aus MSD - Spezifikation abgeleitet und werden im Laufzeitmodell instanziiert.

.scenariorunconfiguration** Diese Datei fast die gebrauchte Dateien und Einstellung zusammen, um Simulation und Synthese durchzuführen. Die gebrauchte Dateien sind **(.corr.xmi, *.roles2eobjects und *.xmi).

Außerdem gibt es noch Datei, die nicht in der Speicherenben gespeichert wird. Im Folgenden wird diese Datei aufgezeigt:

***.simulation** Diese Datei enthält die Ressource von Simulation. Das Ergebnis der Simulation wird hier gespeichert. Das Ergebnis enthält den aktuelle Zustand und auch die mögliche folgende Zustände des Objektsystems.

3.2. Anforderungsspezifikation

Während der Analyse hat sich folgende Anforderungen ergeben. Diese Anforderungen werden in zwei Kategorie unterteil. Erste Teil Anforderungen wird beschreibt, welche Anforderungen es gibt, um die Erweiterung für ScenarioTools in dieser Arbeit zu folgen. Diese Anforderungen werden in 3.2.1 vorgestellt. Die zweite Teil fasst die Qualitätsanforderungen für die Erweiterung zusammen. Die Qualitätsanforderungen werden in 3.2.2 ausgelistet und erklärt.

3.2.1. Anforderungen

Ziel dieser Arbeit ist es, um eine grafische Oberfläche in ScenarioTools hinzuzufügen. Diese Oberfläche müssen die Anforderungen für jedes Projekt, das mit ScenarioTools erstellt wird, angepasst werden. Im Folgende werden diese Anforderungen, mit den die Probleme in dieser Arbeit gelöst werden, aufgezeigt:

R1 Es gibt für jedes dynamisches Objektsystem eine Repräsentation:
Jede erstellte Projekt hat ein dynamisches Objektsystem, diesen das Laufzeitmodell benutzt, um die Simulation aufzubauen. Die Repräsentation visualisiert die Struktur des Objektsystems, die die Objekte und die Beziehungen des Objektsystems sind. Diese Repräsentation wird während der Simulation gezeigt.

R2 Es kann die aktivierte Ereignisnachrichten in der Repräsentation visualisieren:
Während der Simulation können die Ereignisnachrichten aktiviert werden. Wenn eine ermöglichte Ereignisnachricht in der *MessageEvent Selection View* aktiviert wird, dann soll diese Nachricht in der Repräsentation präsentiert werden. Es wird eine Beziehung zwischen sendende und empfangende Objekt von dieser Nachricht visualisiert werden.

Um Oberfläche hinzuzufügen, wird ScenarioTools mit *Graphical Modeling Framework Sirius* verbinden. Deshalb gibt es noch Anforderungen für das System. Im Folgende werden diese Anforderungen aufgezeigt:

R3 Die Beschreibung für die Repräsentation wird ergestellt, und in der Speicherebene gespeichert:
Wenn ein Projekt für Objektsystem erstellt wird, dann soll es ein oder mehrere Modellbeschreibung für dieses dynamische Objektsystem erstellen. Diese Modellbeschreibungen werden auf ***.odesign** gespeichert.

R4 Wenn ein Simulation in *Eclipse Debug Framework* aufgeruft wird, dann wird die entsprechende Repräsentation aufgeladen:
Wenn ein Simulation aufgeruft wird, dann wird die Resourcen des Objektsystems aufgeladen. Diese Erweiterung wird die Repräsentation, die durch Modelldarstellung für das Objektsystem definiert wird, öffnen.

Die Modellbeschreibung sind unterschiedlich für jedes Projekt, deshalb wird diese ***.odesign** Datei bei Benutzer erstellt und bearbeitet. Deshalb wird diese Anforderung bei Benutzer gelöst. Wenn **R3** gelöst wird, dann kann **R1** auch gelöst werden. Daher wird der Fokus in dieser Arbeit auf Anforderungen **R2** und **R4** gelegt, um die Probleme über Visualisierung zu lösen.

3.2.2. Qualitätsanforderungen

Mit ScenarioTools können viele Projekte für Objektsystem erstellt werden, diese Projekte bei unterschiedliche Benutzer erzeugt werden. Deshalb haben die Objeksysteme verschiedene Verhalten und auch Struktur, und es braucht die unterschiedliche Beschreibung für diese Objektsysteme. Zum Beispiel, haben das Projekt „RailCab“ und das Projekt „ProductionCell“ unterschiedliche Verhalten und Struktur. Das Projekt „RailCab“ modelliert die Verhalten und Struktur von einem Railcab - System, das *RailCabs, Barriers, Evironment, usw.* besitzt, aber das Projekt „ProductionCell“ modelliert die Verhalten und Struktur von einem Produktion - System, das *Arms, Press, Controller, usw.* besitzt. Deshlab haben Sie keine ähnliche Daten. Außerdem wird ScenarioTools mit *Graphical Modeling Framework Sirius* erweitern, deshalb muss die Erweiterung die Komponenten von beide Plug-in anpassen. Im Folgenden werden die Qualitätsanforderungen für diese Aussagen aufgezeigt:

Benutzbarkeit Die Anwendung soll mit der aktuellen Java 1.8 - Software, Sirius 2.0 und ScenarioTools ohne Probleme verwendbar sein. Die Funktionen sind einfach und effizient zu benutzen und konsistent sein. Dazu sollen die folgenden Qualitätsfaktoren geachtet werden:

- Im Fall eines Fehlers soll eine Fehlermeldung angezeigt werden, damit der Benutzer mindestens die Information weiß, woher der Fehler bekommt.
- Beim Hinzufügen einer Ereignisnachricht sollen die andere Elemente in der grasichen Editor nicht verschoben oder verändert werden.

Anpassbarkeit ScenarioTools und Sirius haben verschiedene Runtime und Perspektive, deshalb muss die Verwendung die beiden Werkzeugsumgebung angepasst werden. Die Verwendung kann in sowohl Sirius - Perspektive als auch ScenarioTools - Perspektive ausgeführt werden.

Wartbarkeit ScenarioTools hat schon viele verschiedene Komponenten, deshalb gibt es eine Anforderung, um nicht viele Code von ScenarioTools für nächste Version zu verändern. Wenn die Erweiterung in ScenarioTools implementiert wird, sollen wenigstene Komponenten von ScenarioTools verändert werden.

Effizienz Erstellte Objektsystem mit ScenarioTools können viele MSD - Spezifikationen haben, deshalb gibt es viele EMF - Modell für dieses Objektsystem. Deshalb möchte Benutzer nicht Code von diesen Modelle generieren, um die Beschreibung für Repräsentation zu erstellen.

3.3. Anwendungsfälle

Während der Anforderungsanalyse haben sich die folgende gewünschte Funktionen, die zur Verwaltung der Repräsentation im Laufe der Simulation benutzt werden, in Form von Anwendungsfällen beschrieben. Die Abb. 3.2 beschreibt den Zusammenhang zwischen den wichtigsten Teilen der Erweiterung inklusive ScenarioTools und Sirius.

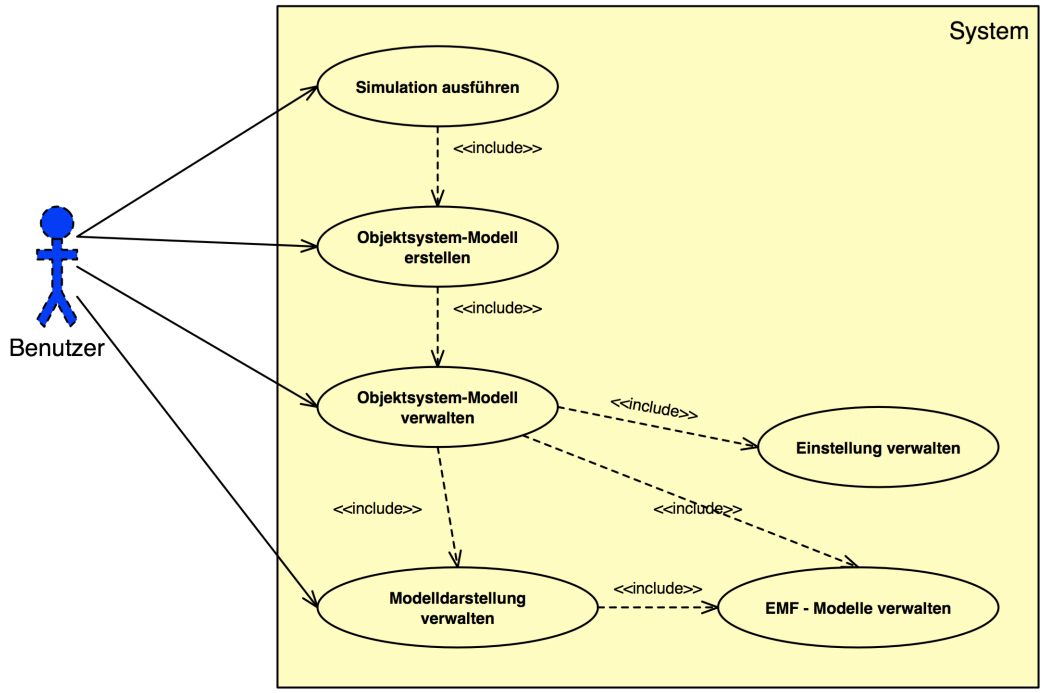

Abbildung 3.2.: Die Zusammenhang von wichtigen Use-Case inklusive ScenarioTools und Sirius.

Die Durchführung der Einstellungsverwaltung und EMF-Modelleverwaltung werden aus Funktion *Create UML-to-Ecore TGG Interpreter Configuration* abgeleitet und wieter bearbeitet. Die Modelldarstellungverwaltung kann mit *Graphical Modeling Framework Sirius* durchgeführt werden. Deswegen wird der Fokus in dieser Arbeit auf Simulationsausführung gelegt. Wenn Benutzer die *ScenarioTools MSD Simulation Perspective* öffnet, soll die Hauptsicht für Simulation geöffnet werden können. Die Anwendungsfälle für Simulation beziehen sich auf die Abb. 3.3. Die Anwendungsfälle entältet auch einigne Anwendungsfälle von ScenarioTools Runtime, mit dem können die neue Anwendungsfälle der Erweiterung zusammengearbeitet werden. Außerdem wird Simulation die Ergebnisse von Objektsystem-Modellsverwaltung aufgeladen.

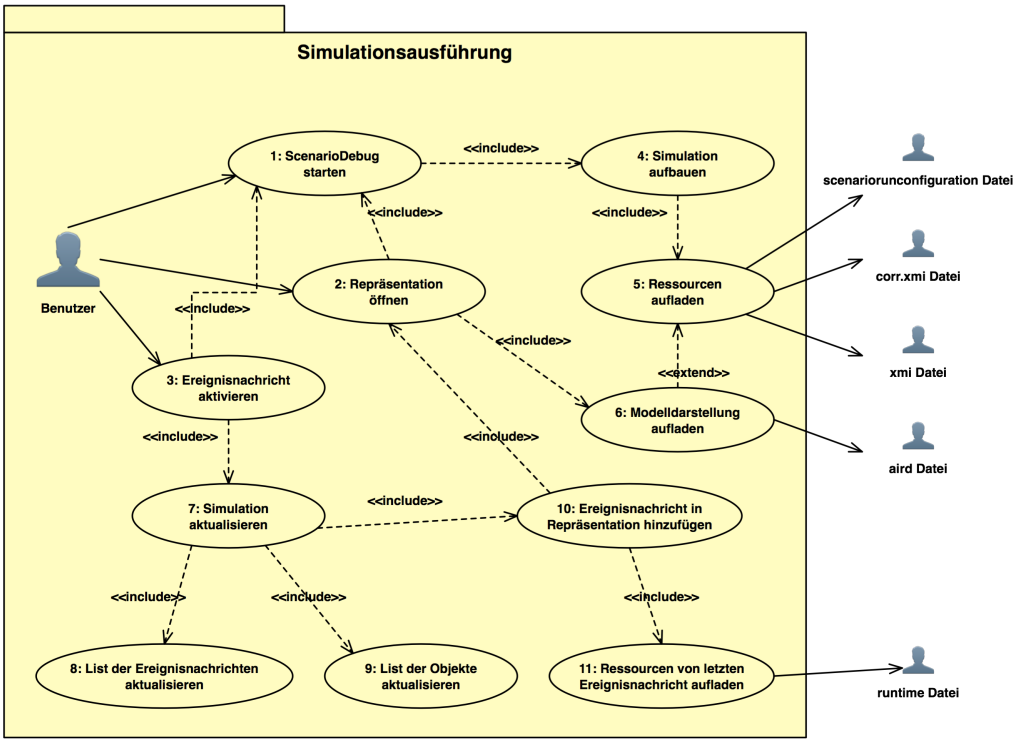

Abbildung 3.3.: Anwendungsfalldiagramm der Simulationsausführung

Die Anwendungsfälle (1), (4), (5), (8) und (9) werden von ScenarioTools entnommen. Die Anwendungsfälle (3), (7) werden die Funktionen von ScenarioTools erweitert. Die neue Anwendungsfälle (2), (6), (10) und (11) werden in dieser Arbeit implementiert. Im Folgende werden diese Anwendungsfälle aufgezeigt:

Anwendungsfall SA2: Repräsentation öffnen

In der Hauptsicht der *Scenario Tools MSD Simulation Perspective* möchte Benutzer die Repräsentation für die aktuelle dynamische Objektsystem der Simulation öffnen. Wenn diese Funktionen aktiviert wird, muss sie die Anwendungsfall SA6 aufrufen, um die Ressource von Modelldarstellung aufzuladen. Dann önnet System die Repräsentation, die Pfad als das Ergebnis aus Anwendungsfall SA6 ist.

Anwendungsfall SA6: Modelldarstellung aufladen

Diese Anwendungsfall ist die Erweiterung von Anwendungsfall SA5. Wenn diese Anwendungsfall aktiviert wird, wird sie die Ressourcespfad von Modelldarstellung in der Projektordner entnehmen. Das Ergebnis wird zu Anwendungsfall SA2 gesendet.

Anwendungsfall SA10: Ereignisnachricht in Repräsentation hinzufügen

Nach der Aktivierung von einer Ereignisnachricht soll die Visualisierung dieser Ereignisnachricht in Repräsentation gezeigt werden. Die Darstellung dieser Ereignisnachricht wird in aktuelle Modelldarstellung hinzugefügt. Die Ressourcen von dieser Ereignisnachricht wird das Ergebnis von Anwendungsfall SA11 sein.

Anwendungsfall SA11: Ressourcen von letzten Ereignisnachricht aufladen
Die Anwendungsfall wird die Ressourcen der letzten Ereignisnachricht aus ***.runtime** Datei herausnehmen. Diese Ressourcen werden die Information über den Name, die sendende und empfangende Objekt der Nachricht enthalten.

3.4. Zusammenfassung

Nach der Analysephase soll das Projekt, das mit ScenarioTools erstellt wird, geändert. Im Folgende werden die Änderungen aufgezeigt:

Jedes Objektsystem-Projekt muss mit einem *Viewpoint Specifikation Project* verbindet werden:
Das neue Projekt definiert die Modellbeschreibung für das Objektsystem, die wird auf einer ***.odesign** gespeichert. ***.odesign** kann gleichzeitig die Modellbeschreibung für mehrere Objektsysteme definiert.

In jedem Objektsystem-Ordner muss einem *.aird Datei erstellt werden:
In dieser Datei werden alle Version von Modelldarstellungen für die Objektsysteme in diesem Objektsystem-Ordner gespeichert. Diese Modelldarstellungen sind die Ergebnisse von der *refresh Algorithm* zwischen ***.odesign** Datei und ***.xmi** Datei.

Die Zusammenhänge der einzelnen Systemkomponenten werden geändert
Die Zusammenhänge soll der Zusammenhang zwischen Modelldarstellung und das System hinzufügen. Diese Änderung wird wie in Abb. A.2 aufgezeigt. Wenn die Modellbeschreibung erstellt wird, dann lädt Sirius die Informationen von der Objektsystem - Modelle (***.ecore**) auf (7). Mit *durch Algorithmus Graphical Modeling Framework Sirius* werden das Objektsystem - Modelle und die Modellbeschreibung auf einer Modelldarstellung kombiniert (8). Während der Ausführung der Simulation wird die Repräsentation für das Objektsystem in *ScenarioTools MSD Simulation Perspective* gezeigt, diese Repräsentation aus ***.aird** Datei aufgeladen wird.

Kapitel 4.

Entwurf

In der Analysephase haben sich zwei Lösungsansätze herausgestellt, um die Ereignissnachricht in Repräsentation während der Simulation hinzuzufügen. Zum einen kann die Ereignisnachticht indirekt mit Hilfe von einer Service Klasse visualisieren, mit dieser Klasse während der Simulation die Ressourcen der letzten Ereignisnachricht herausgenommen werden und dann die Repräsentation inklusive die letzte Ereignisnachricht aktualisiert wird. Zum anderen kann die letzte Ereignisnachricht direkt in die Modelldarstellung (***.aird** Datei) hinzufügen und anschließend die aktuelle Repräsentation aktualisiert wird.

Im Folgenden habe ich mich dafür entschieden die Umsetzung mit Hilfe von *Sirius Framework* durchzuführen. Die Entscheidung ist auf *Sirius* gefallen, da diese einige Vorteile mit sich bringen. ScenarioTools enthält bereits ein TGG Interpreter, damit jedes Objektssystem - Projekt in ScenarioTools mit einer ***.ecore** Datei und einer ***.xmi** Datei hat. Mit Sirius kann Nutzer einfach dieser Datei benutzen, um die Visualisierung zu definieren. Ein weiterer Grund für diese Entscheidung ist die Tatsache, dass die Erweiterung auf diese Weise als weiteres Plug-In realisiert werden kann.

Da die Visualisierung der letzten Ereignisnachricht gleichberechtigt mit der Ereignisnachricht - Ansicht von der Simulation eingelesen wird, muss keine Schnittstelle zwischen Simulation und die Visualisierung der letzten Ereignisnachricht. Zudem soll der Kern der Simulation dabei nicht erweitert werden. Bei der Integration der Visualisierung in die Simulation besteht die Schwierigkeit darin, die Visualisierung der letzten Ereignisnachricht richtig in Modelldarstellung zu definieren.

Im Folgenden ist der Entwurf zu sehen. Zuerst werde ich die erste Überblick von Möglichkeit zeigen, um die Ereignisnachricht zu visualisieren, dann werde ich den wichtigsten Teil der Anwendung mit Struktur des Modells vorstellen. Anschließend werde ich die Ergebnise von Entwurf erfassen.

4.1. Erläuterung der Lösungsansätze

Vor der Implementierung werden in diesem Abschnitt die Lösungsansätze erläutert und analysiert. Danach wird die Entscheidung für die Anwendung zusammengefasst.

Visualisierung mit Hilfe der Service Klasse:(LA01)

Sirius Framework bietet Möglichkeit, um Repräsentation mit Hilfe der Darstellung von Java Service zu aktualisieren. Mit dieser Möglichkeit wird ein Service Klasse erstellt, mit dieser Klasse die Ereignisnachricht während der Simulation visualisiert wird. Dieser Klasse definiert die sendende und empfangende Objekt von Ereignisnachricht in der Repräsentation, danach wird die Repräsentation mit letzten Ereignisnachricht aktualisiert. Mit dieser Möglichkeit wird ein Java Klasse in **Viewpoint Specifikation Project** erstellt, muss diese Klasse in **Viewpoint Specifikation Model** angemeldet werden. Die Klasse enthält alle Methoden, mit der die sendende und empfangende Objekte von Ereignisnachricht bestimmt werden.

Um diese Möglichkeit zu benutzen, muss Ecore Modell von Objektsystem anpassen. Ecore Modell muss die Ressource von *MessageEvent* laden, dann wird *MessageEvent* als ein Objekt im Objektsystem benutzt. In jedem **Viewpoint Specifikation Model** muss ein Kante hergestellt werden, diese Kante als *Element Based Edge* mit MessageEvent als *Domain Class* definiert. Die *Source Mapping* und *Target Mapping* werden durch die Methoden von Service Klassen bestimmt.

Während der Simulation wird Service Klassen die aktuelle sendende und empfangende Objekte von Ereignisnachricht aktualisieren, und sie ruft die *Refresh Algorithm*, um die Repräsentation zu aktualisieren.

Änderung der Modelldarstellung:(LA02)

Mit dieser Möglichkeit wird eine neue Interaktion zwischen Sirius Runtime und ScenarioTools Runtime hergestellt. Die Interaktion wird wie Abb. 4.1 angezeigt.

Das Visualisierung - Algorithmus wird Sirius Runtime und ScenarioTools Runtime verbinden. Während der Simulation wird ScenarioTools Runtime die letzte Ereignisnachricht bestimmt, dann wird die Ressource von letzten Ereignisnachricht durch Visualisierung - Algorithmus in die Modelldarstellung für grafische Editor hinzugefügt.

Deshalb haben Visualisierung - Algorithmus zwei Funktionen, um die Anforderung zu erfüllen. Die erste Funktion wird die Modelldarstellung für grafische Editor prüfen, ob sie schon eine letzte Ereignisnachricht enthält oder nicht. Wenn sie schon eine letzte Ereignisnachricht hat, dann löscht Visualisierung - Algorithmus dieser Ereignisnachricht. Die zweite Funktion

wird die letzte Ereignisnachricht in die Modelldarstellung für grafische Editor hinzufügen, die Ressource von dieser Ereignisnachricht von ScenarioTools Runtime herausgenommen wird.
Nach der Änderung der Modelldarstellung wird Visualisierung - Algorithmus die Editor aktualisieren.

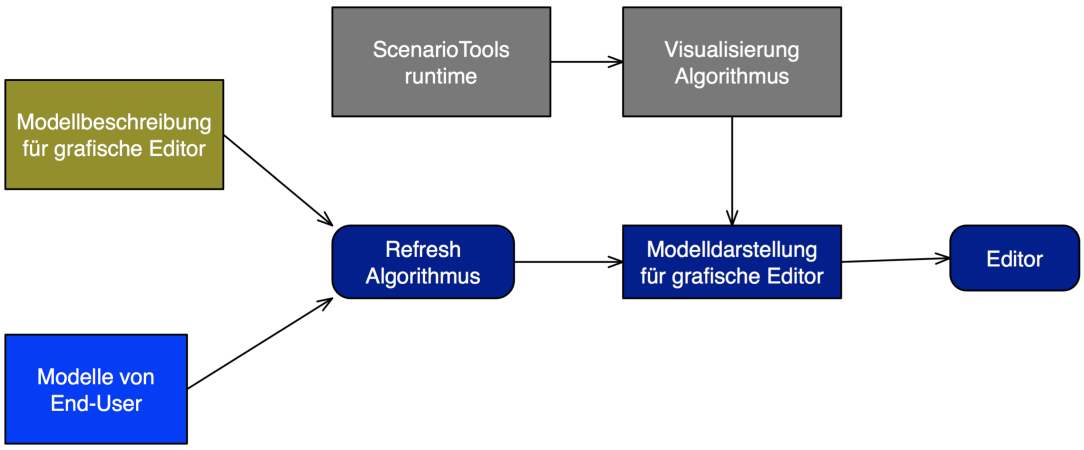

Abbildung 4.1.: Interaktion zwischen Sirius Runtime und ScenarioTools Runtime. In Anlehnung [OT13a].

Die beide Lösungsansätze haben Vorteile und Nachteil, haben sich nach der Analyse und Bearbeitung der beiden Lösungsansätze eine Zusammenfassung herausgestellt.

Visualisierung mit Hilfe der Service Klasse:

- Nachteil: Mit diesem Lösungsansatz kann nicht als Plug-Ins realisiert werden. Die Service Klasse muss immer in **Viewpoint Specifikation Project** liegen, deshalb muss Benutzer Service Klasse und Ressource von MessageEvent laden und anpassen. Das gefällt Benutzer nicht, weil er viele Anpassung machen muss. Benutzer muss ***.ecore** Datei mit MessageEvent anpassen, ***.odesign** mit Service Klasse anpassen.
- Vorteil: Lösungsansatz kann sowohl *Refresh Algorithm* als auch die Bedienungen von Sirius, um die Probleme zu lösen.

Änderung der Modelldarstellung:

Das hat sich nur Vorteil herausgestellt, dass Lösungsansatz kann als Plug-Ins realisisert werden kann. Benutzer muss nicht ***.ecore** Datei und ***.odesign** mit der Erweiterungen anpassen, um die Ereignisnachricht zu visualisieren. Das vermeidet die Fehler während der Bearbeitung von Benutzer.

4.2. Struktur des Modells

Die Umsetzung für die Erweiterung wird mit Hilfe von *Sirius Framework* durchgeführt, deshalb besteht die Zusammenhänge zwischen ScenarioTools und Sirius, die in der Abb. A.2 angezeigt wurden. Im Folgenden werden zunächst die Beziehung zwischen ScenarioTools und Sirius in der Erweiterung dargestellt, und danach werden die Änderungen von Paketen und einzelne Klassen betrachtet.

4.2.1. Beziehung

In Abb. A.2 werden die Zusammenhängen zwischen einzelnen Systemkomponenten von ScenarioTools und die Erweiterung. In diesem Abschnitt werden die Beziehung zwischen ScenarioTools und Sirius in der Erweiterung angezeigt. Um ein Objektsystem während der Simulation mit ScenarioTools visualisieren zu können, muss jedes Projekt wie im Abschnitt 4.3 zwei Projekt - Ordner erstellt werden. Zum ersten Projekt - Ordner enthält nicht nur alle Dateien **(*.uml, *.di, *.ecore, usw.)**, die in der Abbschnitt 3.1 beschrieben werden, sondern wird Datei ***.aird** hinzugefügt. Zum zweiten Projekt - Ordner enthält ein EMF - Modell ***.odesign**, das die Visualisierung des Objektsystems während der Simulation definiert. Diese Assoziationen zwischen die Dateien, die zur Visualisierung gebraucht werden, sind in dem folgenden Diagramm ersichtlich:

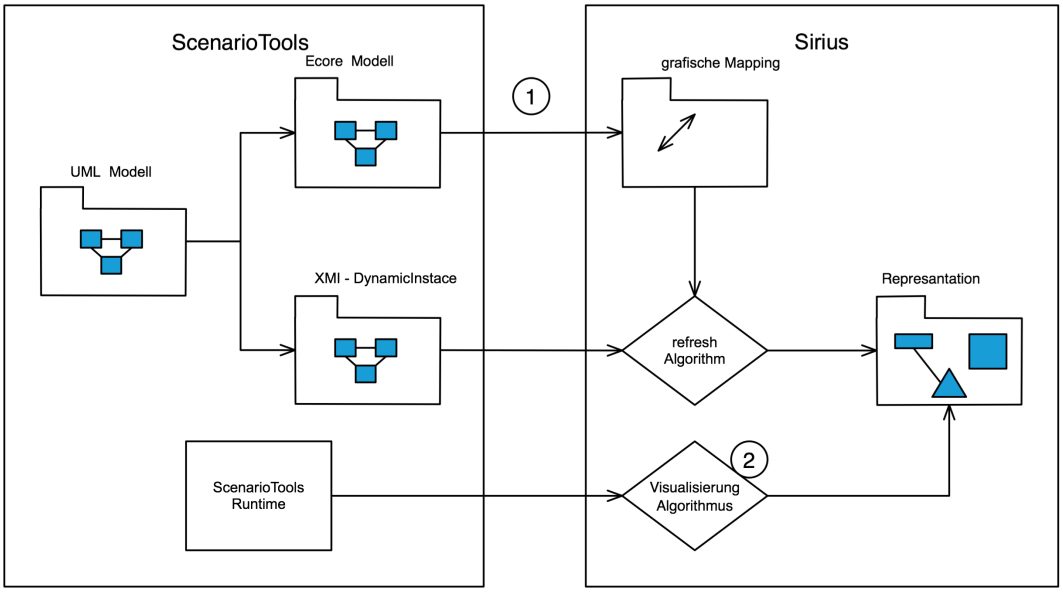

Abbildung 4.2.: Beziehungen zwischen ScenarioTools und Sirius.

Um ein Objektsystem mit Sirius zu visualisieren, muss Benutzer Ecore - Modell (***.ecore**), XMI - DynamicInstace (***.xmi**) und ein grafische Mapping (***.odesign**) erstellen. Im Kapitel 3.1 gibt es schon die Erklärung, dass nach der Erstellung von UML - Modell (***.uml**) Benutzer die TGG - Transformation (*uml2ecore*) benutzen kann, um Ecore - Modell und XMI - DynamicInstance zu erstellen. Sirius Runtime kombiniert grafische Mapping und XMI - DynamicInstance mit „*refresh Algorithm* “, um die Visualisierung für Objektsystem zu erstellen.

In Sirius kann Benutzer die Definitionen von Objektsystem im Ecore - Modell benutzen, um die grafische Mapping zu erstellen. Sirius bietet eine vollständige Umgebung, um Ecore - Modell zum grafischen Mapping zu transfomieren. [1]

Während der Simulation kann ScenarioTools - Runtime die Repräsentationen (Modelldarstellung ***.aird**) mit Hilfe von Visualisierung - Algorithmus (2) benutzen. Mit Visualisierung - Algorithmus kann ScenarioTools - Runtime die Repräsentationen von aktuellen Objektsystem öffnen und die letzte Ereignisnachricht während der Simulation visualisieren (neu erstellen/ ersetzen).

4.2.2. Paketen

Um die Visualisierung - Algorithmus in ScenarioTools zu intergrieren, werden einzelne Paketen von ScenarioTools verändert und werden neue Paketen in ScenarioTools hinzugefügt.

Weil die Komponenten von ScenarioTools durch EMF - Modell hergestellt werden, ändere ich die EMF - Modell von einzelne Projekt. Um die letzte Ereignisnachricht zu speichern, soll die Runtime von ScenarioTools verändern. Deshalb wird das Modell **runtime.ecore** im Projekt *org.scenariotools.msd.runtime* geändert. Die Paketen *org.scenariotools.msd.runtime* und *org.scenariotools.msd.runtime.impl* werden wegen dieser Information, die als „messageEventExecutedLast“ gennant wird, angepasst.

Um die Repräsentationen zu öffnen, wird ein neu Paket im Projekt *org.scenariotools.msd.simulation* hinzugefügt. Dieses Paket wird als *org.scenariotools.msd.simulation.tools* genannt. Das Paket *org.scenariotools.msd.simulation.debug* wird auch angepasst.

Die letzte Ereignisnachricht wird während Simulation visualisiert, soll das Paket *org.scenariotools.msd.simulation.ui.views* geändert werden.

[1] Die Erstellung von grafischen Mapping wird durch `http://www.eclipse.org/sirius/doc/specifier/Sirius%20Specifier%20Manual.html` deutlicher erklärt.

4.2.3. Klassen

Um die Information für „messageEventExecutedLast “hinzufügen, werden die Klasse **MSDRuntimeStateGraph.java** von *org.scenariotools.msd.runtime* und die Klassen **MSDRuntimeStateGraphImpl.java** und **MSDRuntimeStateImpl.java** von *org.scenariotools.msd.runtime.impl* verändert. Um die Klassen **MSDRuntimeStateGraph*** zu verändern, wird das EMF - Modell **runtime.ecore** eine neue Referenz „messageEventExecutedLast“ zwischen MSDRuntimeStateGraph und MessageEvent hinzugefügt, dann führt die Generierung diesem Modell aus. Für **MSDRuntimeStateImpl.java** wird die Funktion **performStep(Event)** geändert.

Weil die letzte Ereignisnachricht während der Simulation visualisiert werden wird, dann soll Methode **objectDoubleClicked(Object obj)** der Klasse **MSDModalMessageEventSelectionView.java** von Paket *org.scenariotools.msd.simulation.ui.views* angepasst, und sie wird mit der folgenden Methoden implementiert:

getCurrentRepresentation(SimulationManager sm) Diese Methode wird die aktuelle Repräsentation der Simulation ausnehmen.

displayLastMessage(MessageEvent msg) Diese Methode wird die aktuelle letzte Ereignisnachricht in der Repräsentation des Objektsystem mit msg addieren oder ersetzen.

deleteLastMessage(MessageEvent msg) Diese Methode wird die aktuelle letzte Ereignisnachricht in der Repräsentation des Objektsystem löschen.

Wenn Simulation des Objektsystemes gestartet wird, soll das Diagramm gleichzeitig geöffnet werden. Deshalb wird **ScenarioDebugTarget.java** von Paket *org.scenariotools.msd.simulation.debug* mit der folgende Änderung angepasst:

- Variable MSDSimulationDiagramEditor hinzufügen.
- Konstruktur **ScenarioDebugTarget(..)** wird anpasst. Deshalb wird das Diagram gleichzeitig geöffnet, wenn ScenarioDebug gestartet wird.

Das Paket *org.scenariotools.msd.simulation.tools* wird mit neuen Klasse **MSDSimulationDiagramEditor.java** erstellt. Die neue Klasse werden zuerst mit der folgenden Methoden implementiert:

open(IPath iPath) Diese Methode wird das Diagram mit gegebene Pfad öffnen.

openEditor(SimulationManager sm) Diese Methode wird das Diagram von aktuelle Simulation öffnen.

getInstanceModelURIString(SimulationManager sm) Diese Methode wird der Pfad von aktuelle Simulation in String - Format ausgeben.

init(IWorkbenchWindow nw) Wenn Eclipse - Workbench geschlossen wird, dann wird diese Methode alle geöffnete Diagramme sofort schließen.

In Abb. 4.3 werden die Klassendiagramm von geänderten Klassen und neuen Klassen angezeigt. Ingesamt werden drei Klassen von aktuellen ScenarioTools geändert, und ein neue Klasse wird gestellt, um die Problem über Visualisierung in ScenarioTools zu lösen.

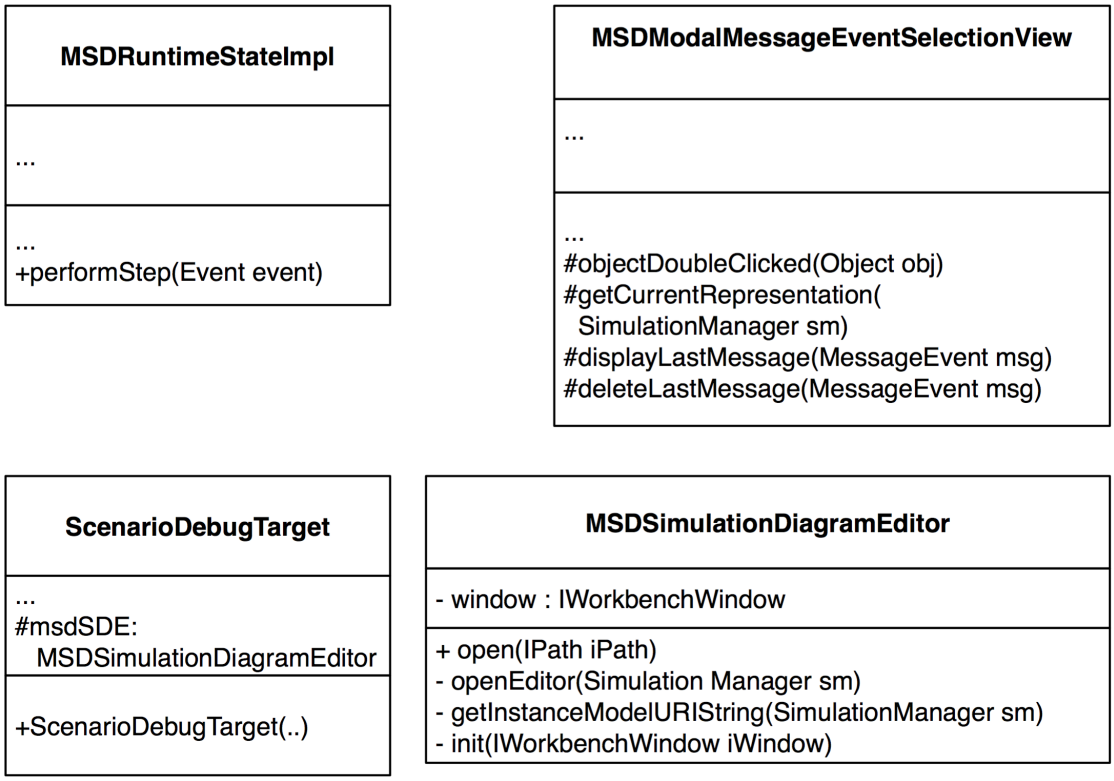

Abbildung 4.3.: Klassendiagramm von geänderten Klassen und neuen Klassen

Die Klassen in der Abb. 4.3 bildet „Visualisierung Algorithmus“.

4.3. Zusammenfassung

Nach der Bearbeitung mit Entwurf wird die Zusammenfassung von Ergebnisse herausgestellt, dass die beide Lösungsansätze in der 4.1 nicht fertig gemacht werden. In der Folgende werden die Gründe von dem Fehlschlagen:

Erste Grund Die Ressourcen von der Simulation werden während der Simulation nicht generiert, deshalb tut es eine Schwierigkeit, um die Inhalte für die Objekte von der Simulation während der Simulation mit Service Klasse oder mit Entwurf vom Visualisierung Algorithmus herauszunehmen.

Zweite Grund Es gibt sehr wenig Dokumentationen über die Funktion sowie die Strukturen von Klassen, Kommando, Aktionen, usw. von Sirius Framework. Deshalb gibt es die Schwierigkeit, um die zweite Lösungsansatze fertig einzubauen.

Mit dieser Gründe können die Probleme nicht direkt mit Entwurf von Lösungsansätze in der Abschnitt 4.1 gelöst werden, aber eine indirekt Lösungansatze wird erscheint, dass der Visualisierung - Algorithmus als eine Subsystem gebaut wird. Mit diesem Subsystem werden die Informationen von ScenarioTools Runtime in dem Format von einem Editor, der mit EMF - Modell eingebaut wird, umgewandeln. Daneben wird ein Mapping für die Objekte von diesem Editor mit Sirius Framework erstellt, um die Grafikoberfläche zu definieren, und als SimDiagram Mapping Datei (wie Abb. B.1) gespeichert wird.

Der Verlauf wird wie in der Abb. A.3 gezeigt. Zuerst wird Subsystem die Inhalte von Objektsystem herausnehmen, um das Modell im Struktur von einem neuen Editor, der als *SimulationDiagram* genannt wird, herzustellen. Dann wird das Modell mit Hilfe von Sirius Framework visualisiert. W"ahrend der Simulation muss das Modell mit neuen Zustand aktualisiert werden, deshalb wird die Grafikoberfläche auch bei refresh-Algorithmus aktualisiert. Die letzte Ereignisnachricht wird echt zeitig mit der Zustand von Simulation repräsentiert.

Der Visualisierung - Algorithmus wird in zwei Teile aufgeteilt. Erste Teil ist die Visualisierung - Transformation, die die Grundstruktur des Objektsystem während der Simulation im *SimDiagram* transformiert und dann dieses Modell visualisiert. Zweite Teil ist die NVisualisierung, die die letzte aktivierte Ereignisnachricht des Objektsystem während der Simulation visualisiert.

In der folgende Kapitel 5 wird die Erklärung von der*SimulationDiagram* und der neue Visualisierung - Algorithmus sich deutlicher vorgestellt.

Kapitel 5.

Projektumsetzung

In ScenarioTools habe ich einen neuen Editor und den Visualisierung - Algorithmus integriert und wird nun automatisch mit Ablauf von ScenarioTools ausgeführt. Die Visualisierung - Algorithmus wird ausgeführt, indem auf einer ScenarioRunConfiguration-Datei die Zustände von Simulation aufgerufen und transformiert wird. In der Folgende wird die Erkärung von der SimulationDiagram und zwei Unterteile Visualisierung - Transformation und NVisualisierung des Visualisierung - Algorithmus sich vorgestellt. Danach wird die Benutzeroberfläche von der Simulation nach der Erweiterung erklärt.

5.1. SimulationDiagram Editor

ScenarioTools besteht aus eine Möglichkeit, um die erstellten MSD - Spezifikationen zu simulieren. Deshalb enthält jede Modell während der Simulation die folgende Teile:

RootObjects sind das oberste Objekt in der Modell. Die andere Objekte gehören zu ihm. Im Beispiel 2.1.1 wird diese Objekt als *RailCabSystem* genannt.

SystemObjects sind die steuerbare Objekte in der Modell. Zum Beispiel können diese Objekte als *RailCab, CrossingControl, TrackSectionControl* im Beispiel 2.1.1 sein.

EnvironmenObjects sind die nicht kontrollierbare Objekte in der Modell. Zum Beispiel können diese Objekte als *Environment, Barriers* im Beipsiel 2.1.1 sein.

Beziehung sind die Beziehungen zwischen die Unterobjekte (Systemobjekte und Umweltobjekte) in der Modell. Zum Beispiel ist die Beziehung **current**

zwischen *RailCab* und *TrackSectionControl* im Beispiel 2.1.1.

MessageEvent sind die letzte aktivierte Nachricht in der Modell. Zum Beispiel ist die Nachricht *EndOfTS*, die die Nachricht von *Environment* nach *RailCab* im Beispiel 2.1.1 ist.

Mit der oberen Struktur wird der Editor mit Eclipse Modeling Framework (EMF) wie in der Abb. A.4 erstellt. Dieser Editor wird als SimDiagram genannt. *SimDiagram* Editor wird durch fünf EClass hergestellt. Die Informationen von dieser EClass werden in der Folgende erläutert:

SimObjectSystem sind oberste Objekte, die die Ressource von jeden RootObject des Objektsystems im *SimDiagram* Editor zu definieren benutzt wird. Jede *SimObjectSystem* enthält gleichzeitig mehrere *SimSystemObject, SimEnvironmentObject, SimReference*, aber es kann nicht mehr als ein *SimLastMessage* enthälten.

SimSystemObject sind die Unterobjekte, die die Ressource von Systemobjekte des Objektsystems im *SimDiagram* Editor zu definieren benutzt wird.

SimEnvironmentObject sind die Unterobjekte, die die Ressource von Umweltobjekte des Objektsystems im *SimDiagram* Editor zu definieren benutzt wird.

SimReference sind die Unterobjekte, die die Beziehung zwischen die Unterobjekte im *SimDiagram* Editor zu definieren benutzt wird. Jede Objekt *SimReference* enthält die Referenzen *sourceNode* und *targetNode*, die mit *EObject* gebunden werden, um die Quelle und das Ziel der Beziehung zu definieren.

SimLastMessage sind die Unterobjekte, die die Ressource von letzten aktivierten Ereignisnachricht im SimDiagram Editor zu definieren benutzt wird. Jede Objekt SimLastMessage enthält die Referenzen *source* und *target*, die mit *EObject* gebunden werden, um die Quelle und das Ziel der Nachricht zu definieren.

Alle EClass enthält EAttribute *name*, das Inhalt über den Name von Objekt definiert. Zum Beispiel wird der Anfangsbestand des Beispiel 2.1.1 im *SimDiagram* wie in Abb. C.4 modelliert.

Die Ressource des Modells, die bei dem SimulationDiagram Editor beschrieben wird, wird die Struktur des Objektsystems während der Simulation gespeichert, und als Semanticmodel Datei (*.simdiagram) genannt.

5.2. Visualisierung - Transformation

Die Transformation soll direkt im Anschluss an den Debug - Launcher erfolgen. In Abb. 5.1 ist beschreiben wie die Transformation in den bestehenden Prozess von ScenarioTools integriert werden soll.

Nachdem der Benutzer auf einer ScenarioRunConfiguration Datei das Event Debug ausgelöst hat, startet der Debugger (1) mit dieser Datei als Eingabeparameter. Der Debugger erzeugt drei Dateien. Erste Datei wird das Zwischenergebnis der Simulation, der die erkundeten Zustände des Objektsystems während der Simulation beinhaltet, gespeichert und als MSDRuntimeStateGraph genannt. Zweite Datei wird den aktuelle Zustand der Simulation gespeichert, und als MSDRuntimeState genannt. Dritte Datei wird die Ressource von MSDRuntimeStateGraph und MSDRuntimeState Dateien gespeichert, und als SimulationManager genannt.

Wenn der SimulationManager Datei nicht leer ist, dann wird die Visualisierung - Transformation (2) gestartet. Dafür benötigt der Prozess der SimulationManager Datei und der SimDiagram Mapping Datei als Eingabeparameter. Das Ergebnis der Transformation wird ebenfalls in zwei Dateien gespeichert. Erste Datei wird die Ressource von der Grundstruktur des Objektsystems gespeichert, und als SimDiagram Datei (*.simdiagram) genannt. Zweite Datei wird die Ressource von der Visualisierung des Objektsystems gespeichert, und als Representations Datei (*.aird) genannt.

Außerdem wird eine grafische Oberfläche geöffnet, mit der die Struktur des Objektsystems zusammen mit der letzten aktivierten Ereignisnachricht während der Simulation visualisiert werden kann. Um die Visualisierung mit in die Simulation integrieren zu können, muss sie der Simulation übergeben werden. Die beste Möglichkeit ist eine Erweiterung der Simulation, dass das Visualisierung - Transformation - Paket zusätzlich initialisiert wird, weil die Simulation nicht eine Paketvereinigung unterstüzen kann, und die Eingabedaten werden während der Simulation nicht verändert.

In diesem Abschnitt beschreibe ich die technische Umsetzung der Visualisierung - Transformation. Für die Transformation benutze ich den Eclipse Modeling Framework (EMF) und Sirius - Framework wie sie im Grundkapitel 2 beschrieben wurde.

Die Visualisierung - Transformation werden noch in zwei Unterteile aufgeteilt. Zum einen wird die Ressource von SimulationManager Datei in Ressource von SimDiagram Datei umgewandelt. Zum anderen wird die Ressource von SimDiagram Datei visualisiert.

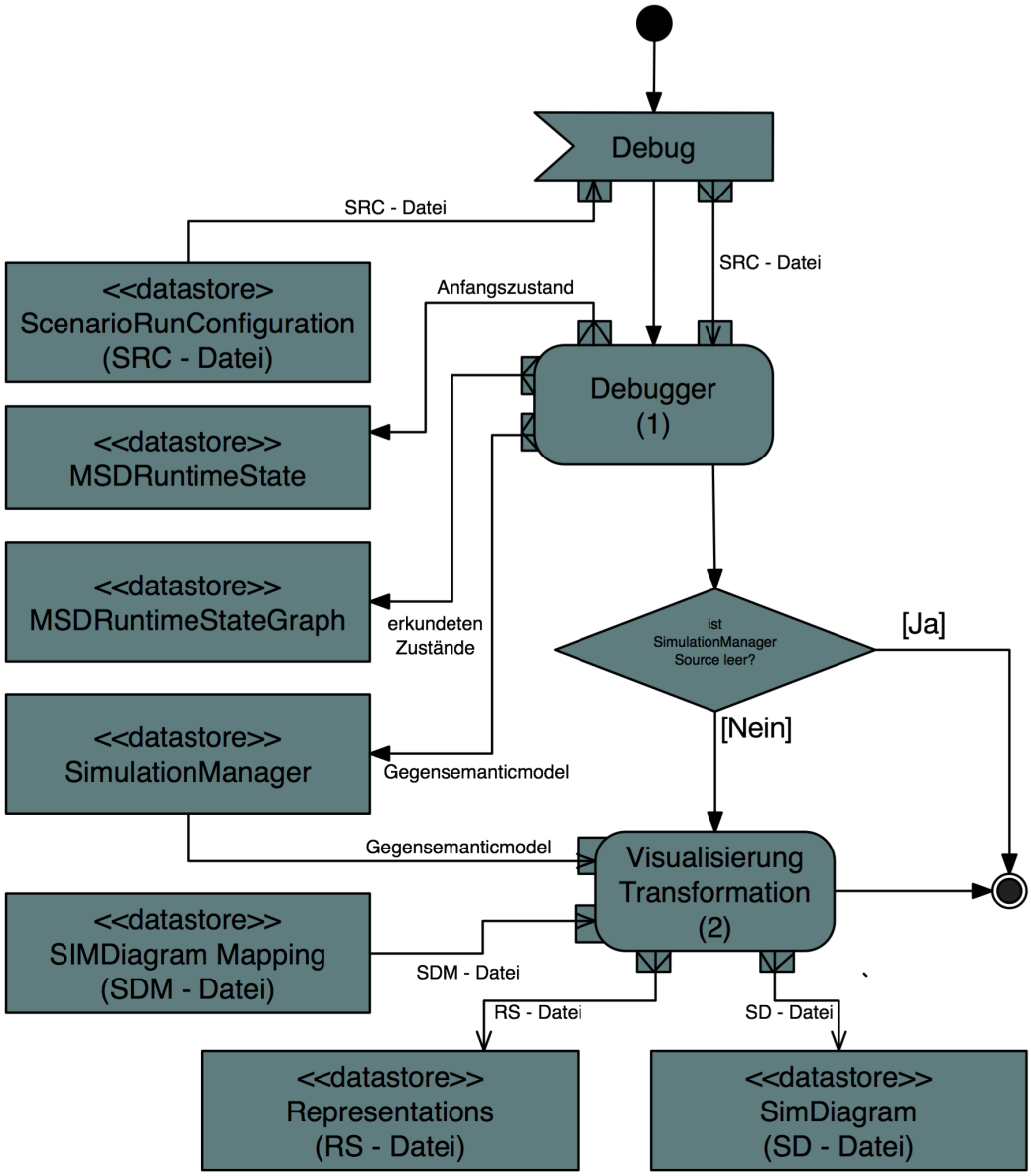

Abbildung 5.1.: Das Aktivitäsdiagramm beschreibt den Ablauf, in den die Visualisierung - Transformation integiert wurde.

5.2.1. Die Bestimmung der Domänen

Beginnen wird nun damit die Domänen der Ressourcen von Quell- und Zielmodells festzusetzen. Als Domäne verwende ich jeweils das Paket, in dem die Datenstruktur des Objektsystems, mit der die jeweilige Eigenschaften von Objekte des Objektsystems beschrieben wird, enthalten ist. Das Semanticmodel wird durch die Ressource von einen **SimulationManager** beschrieben, der sich in dem Pakete **Simulation** befindet. Die Quelldomäne erhält die Dateiendung der Datei, in der sich die Ressource für das Semanticmodel befindet, als Name

simulation. Die Quelldomäne erhält die Dateiendung der Datei, in der sich das **SimDiagram-Mapping** befindet, als Name *mapping*. Die Zielmodelle sind das Semanticmodel und das Modell für die Ressource des grafischen Editors, um das Objektsystem zu visualisieren. Deshalb trägt die Zieldomäne den Namen *visual*.

Betrachten wir den Anfangszustand, mit dem Wissen, dass dieser kaum Informationen des Objektsystems in Anfang enthält, so können wir mit dieser Zustand aus dem Semanticmodel in Anfangszustand des grafischen Editors überführen.

Schauen wird uns als zuerst die Transformation der Objekt an. Eine Objekt des Objektsystems in MSD Spezifikationen werden als EObject generalisiert. Dafür kann die Transformation auf das Ecore Meta-Modell zugegriffen werden. Damit ist der Bereich des Semanticmodel und der Simulation abgedeckt. Jede Objekt besitzt ein Name und die Beziehungen mit anderen Objekte des Objektsystems. Der Name wird durch das **EAttributte** *name* bezeichnet. Die Beziehungen werden durch die List von **EStructuralFeature** bezeichnet. Dieses **EStructuralFeature** bezeichnet den Name der Beziehung und die Zielobjekt der Beziehung. Das Semanticmodel soll in das Objektsystem der MSD Spezifikation übernommen werden und dabei die Bedeutung seine Bedeutung während der Transformation nicht verlieren.

Betrachten wir die Abbildung 5.2. Das Schaubild soll den Zusammenhang der Struktur des Objektsystems in MSD Spezifikation und der Visualisierung - Transformation verdeutlichen. Zu sehen ist die Datenstruktur von MSDRuntimeState (1). Sie bidet die Grundstruktur des Objektsystems in MSD Spezifikation. Die Datenstruktur (1) wird zusammen mit Datenstruktur von MSDRuntimeStateGraph, usw. in Datenstruktur SimulationManager (2) behielt. Dabei repräsentiert der Datenbehäter (3) .simdiagram das Semanticmodel. In der Datenstruktur (4) .odesign werden alle Mapping von der EObjekte des SimulationDiagram (.simdiagram) gespeichert. Hier werden das Semanticmodell und die verschiedene Mappings, mit der die Grundstruktur des Objektsystems visualisiert werden kann, in einer Datenstruktur (5) festhalten. Auf diese Weise soll sichergestellt werden, dass die Ressource von Objekte des Semanticmodel auf die richtigen Ressource von Objekte in der Spezifikation abgebildet werden.

Damit unsere Domäne des Quellmodells diese Zuordnung abdecken kann, müssen wir sie um die nötigen Informationen zu transformieren erweitern. Das Modell für MSDRuntimeState wird durch die Meta-Modelle runtime beschrieben, und wird das Modell für SimulationManager durch die Meta-Modelle simulation beschrieben. Die Zuordnung zwischen SimulationManager und MSDRuntimeState habe ich als eigene Domäne modelliert. Der Grund dafür ist, dass die Ressource von dieser Elemente nicht direkt zum Quellmodell gehören.

Zusätzlich wird die ScenarioRunConfiguration und SimDiagram-Mapping als Eingabe für die Transformation benötigt, damit aus ihr die Konfiguration für die Visualisierung abgeleitet werden kann. Da die ScenarioRunConfiguration die Paket Simulation enthält, dann gehört sie zu Domäne *simulation*. Weil das Paket Simulation die andere Pakete von ScenarioTools enthält, deshalb gehören die andere Meta-Modelle stategraph, events, ecore, scenariorunconfiguration, und util zu ihn. In Tabelle 5.1 ist die Zuordnung der Domänen zusammengefasst.

Domäne	Name	Meta-Modell
Quelldomäne	simulation	simulation, runtime, stategraph, events, ecore, util, scenariorunconfiguration
Quelldomäne	mapping	odesign
Zieldomäne	visual	simdiagram, aird

Tabelle 5.1.: Zuordnung der Domänen

5.2.2. Ergebnis der Transformation

Das Ergebnis der Transformation wird in mehreren Dateien abgespeichert. Dabei werden die Dateinamen und -pfad nach folgenden Muster erstellt: Alle erstellten Dateien werden im gleichen Ordner gespeichert. Es handelt sich hierbei um den Ordner, der die ScenarioRunConfiguration-Datei (*.scenariorunconfiguration) beinhaltet, auf der die Simulation ausgeführt wurde. Die Dateinamen bestehen aus dem Dateinamen der ScenarioRunConfiguration-Datei Endung und die folgende Dateiendung:

***.simdiagram** Die Datei *.simdiagram enthält ein SimulationDiagram - Modell. Hier sind die Ressource darüber gespeichert, wie die Objekte des Simulationsystem, die Beziehung zwischen der Objekte und ein letzte aktivierte Ereignisnachricht. Im Anfangszustand wird EObjekt für letzte aktivierte Ereignisnachricht leer gewesen.

***.aird** Die Datei *.aird enthält ein DAnalysis - Modell. Hier sind die Information darüber gespeichert, wie die Ressource in der Abschnitt 2.4.2 erklärt wird.

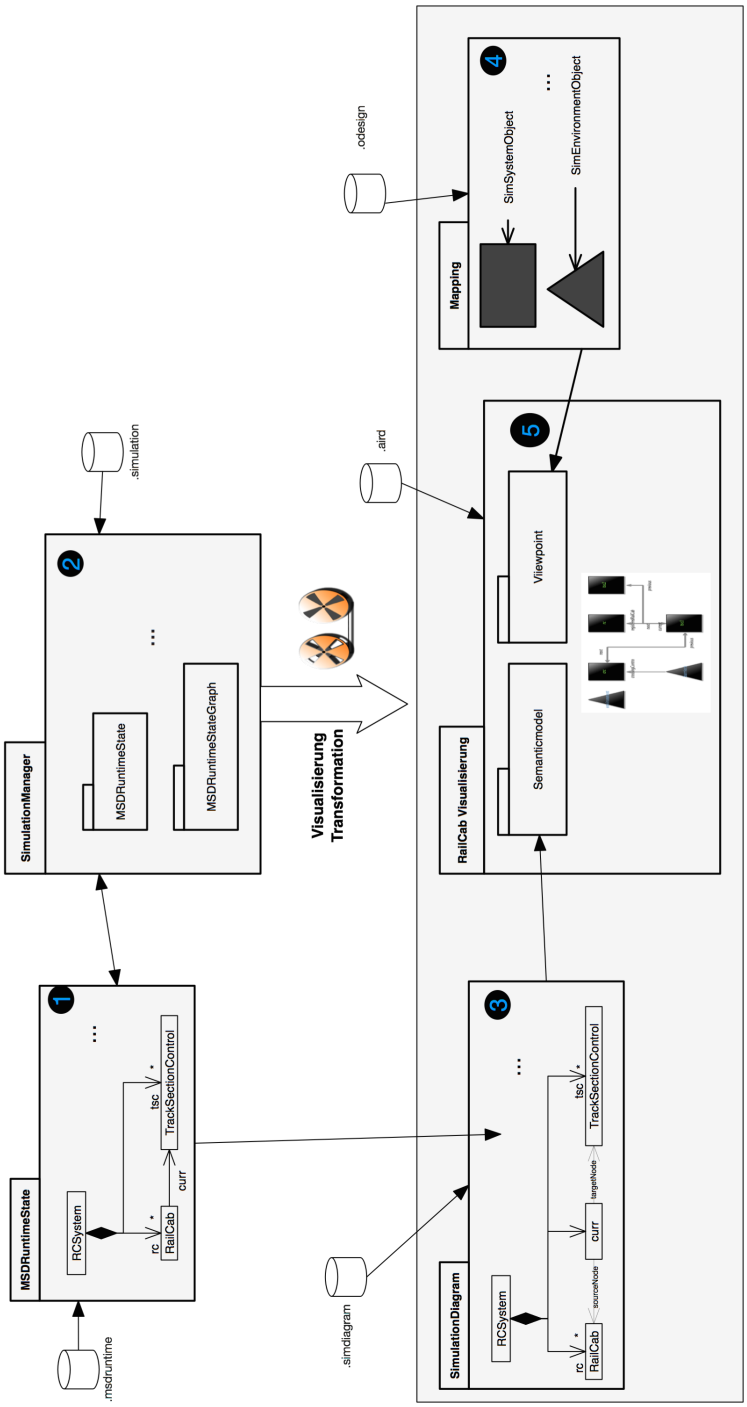

Abbildung 5.2.: Zu sehen sind die Zusammenhänge von einzelnen Komponenten von ScenarioTools, die über die Grunkstruktur des Simulationsystems in Bezug gesetzt wurden. Ein Simulationsystem besteht im Wesentlichen aus ein Objectsystem von MSDRuntime.

5.2.3. Umsetzung der Visualisierung - Transformation

Wenn die Visualisierung - Transformation ausgeführt hat, wird die alten Dateien, die mit Endung *.simdiagram und *.aird beinhaltet, gelöscht. Im Folgenden stelle ich zwei Schritte für die Umwandlung vor. Diese Schritte werden benutzt, um die Ressource der Objekte von Quelldomäne *simulation* in Zieldomäne *visual* zu transformieren. Zum einen ist die Regel, mit der die Objekte des Objektsystems von MSDRuntime in die Objekte des Objektsystems von SimulationDiagram umgewandelt werden kann. Zum anderen werden die Beziehung zwischen Objekte des Objektsystems von MSDRuntime als die Ressource benutzt, um die Objekte *SimReference* von SimulationDiagram zu erstellen. In der Abschnitt 5.1 habe ich schon vier Teile des Objektsystem erklärt. Das Ergebnis von diesem Unterteil wird als Semanticmodel Datei gespeichert. Um diese Teile umzuwandeln, werden die folgende Schritte durchgefüht:

1. Die Transformation von Objekte in der Visualisierung - Transformation muss immer die folgende Regeln anpassen:
   a) Die Ressource von SimObjectSystem muss die Ressource von RootObject übergenommen werden.
   b) Die Ressource von SimSystemObject wird die Ressource von ControllableObject, das nicht als RootObject ist, übergenommen.
   c) Die Ressource von UncontrollableObject wird in die Ressource von SimEnvironmentObject umgewandelt.
2. Die Beziehung zwischen Objekte des Objektsystem in MSDRuntime werden in EStructureFeature gespeichert, deshalb muss die Transformation von Beziehung in der Visualisierung - Transformation immer die folgende Regeln anpassen:
   a) Der Wert von EStructureFeature muss immer als ein EObject oder eine List von EObject sein. Wenn der Wert von EStructureFeature als ein String ist, dann transformiert diese Beziehung nicht.
   b) Die Transformation von Beziehung musste immer alle EStructureFeature einer Objekt fertig transformieren, bevor sie eine andere Objekt transformieren kann.
   c) Die Objekt, die EStructureFeature enthält, wird als sourceNode von SimReference definiert. Der Wert von EStructureFeature wird als targetNode von SimReference. Weil der Wert eine List von EObject sein kann, wird ein EStructureFeature in mehrere SimReference transformiert.
   d) Die EAttribute **name** von SimReference ist der Name von SimReference.

Nachdem die Transformation der Grundstruktur des Objektsystems fertig ausgeführt hat, dann wird die Semanticmodel Datei visualisiert werden. Dieser Unterteil wird durch den Funktion createRepresentation() ausgeführt. Die hergestellte Semanticmodel Datei (*.simdiagram) und die Mapping Datei (simulation.odesign) werden als Eingabeparameter für die Funktion benögtigt. Sie wird eine Session Datei erstellen, dann die Ressource von Semanticmodel Datei sowie die Ressource von Viewpoint der Mapping Datei in dieser Session Datei hinzufügen. Die Funktion wird die Repräsentationen für dieser Semanticmodel Datei erstellen. Sirius - Framework wird den refresh - Algorithmus aufrufen, um die Repräsentation zu öffnen. Die gesamte Durchführung der Visualisierung - Transformation wird mit der Klasse *CreateSimDiagram.java* abgebildet.

5.3. NVisualisierung

Die Visualisierung soll direkt im Anschluss an die aktivierte Nachricht erfolgen. In Abb. 5.3 ist beschrieben wie die NVisualisierung in den bestehenden Prozess von ScenarioTools integriert werden soll.

Nachdem der Benutzer auf einer Ereignisnachricht das Event Nachricht aktivieren ausgelöst hat, startet der NVisualisierung mit SimulationManager und MSDRuntimeState als Eingabeparameteren. Dafür benögtigt der Prozess durch den SimulationManager Datei der Semanticmodel Datei und der MSDRuntimeState Datei als Eingabeparameter. Das Ergebnis der NVisualisierung wird ebenfalls die Ressource von zwei Dateien im Ordner aktualisiert. Erste Datei wird die Ressource von der aktuellen Struktur des Objektsystems gespeichert, und als SimDiagram Datei (*.simdiagram) genannt. Zweite Datei wird die Ressource von der aktuellen Visualisierung des Objektsystems gespeichert, und als Representations Datei (*.aird) genannt.

Außerdem wird die grafische Oberfläche aktualisiert, in der die Struktur des Objektsystems zusammen mit der letzten aktivierten Ereignisnachricht visualisiert wird. Um die NVisualisierung mit in die Simulation integrieren zu können, muss sie der Benutzeroberfläche der Simulation übergeben werden. Die beste Möglichkeit ist eine Erweiterung der Oberfläche der Simulation, dass das NVisualisierung - Paket zusätzlich initialisiert wird, weil die Oberfläche der Simulation nicht eine Paketvereinigung unterstüzen kann, und die Eingabedaten werden während der Simulation nicht verändert.

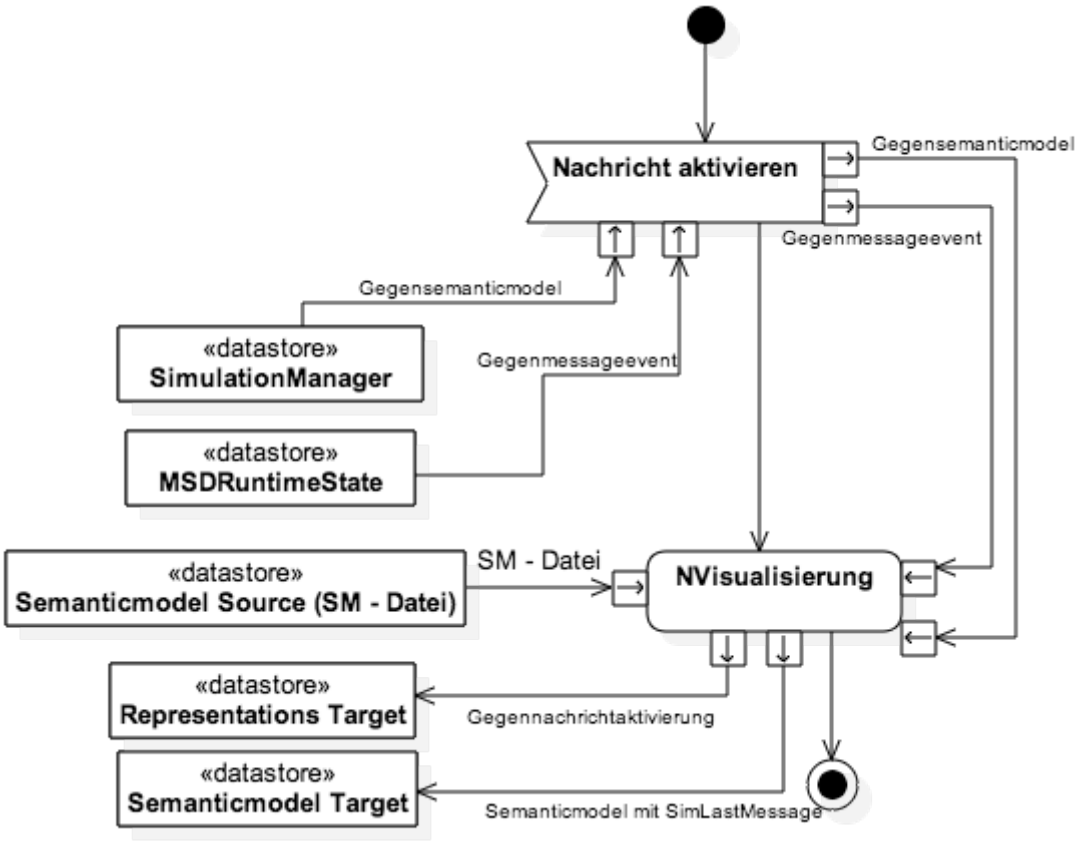

Abbildung 5.3.: Das Aktivitätsdiagramm beschreibt den Ablauf, in den die NVisualisierung integriert wurde.

5.3.1. Domänen und Zuordnung

Als Domäne verwende ich für diesen Teil das Paket, in dem die Datenstruktur des Objektsystems, mit der die jeweilige Eigenschaften von Objekten des Objektsystems beschrieben wird, enthalten ist. Das Semanticmodel wird durch die Ressource von einen SimulationManager aktualisiert, der sich in dem Pakete Simulation befindet. Die Quelldomäne enthält die Dateiendung der Datei, in der sich die Ressource von dem Semanticmodel befindet, als Name *simulation*. Die Quelldomäne erhält die Dateiendung der Datei, in der sich das **SimDiagram-Mapping** befindet, als Name *mapping*. Die Zielmodelle sind das Semanticmodel und das Modell für die Ressource des grafischen Editors, um das Objektsystem zu visualisieren. Deshalb trägt die Zieldomäne den Namen *visual*.

In diesem Algorithmus betrachten wir einen einfachen Zustand, mit dem Wissen, dass dieser kaum Informationen enthält, so können wir mit dieser Zustand zusammen mit der letzten aktivierten Ereignisnachricht visualisieren.

Als nächstes schauen wird uns die Visualisierung der Nachrichten an. In der

Kapitel 2.1.2.1 habe ich schon erklärt, dass eine Nachricht in MSD - Spezifikation ein sendendes und ein empfangendes Objekt hat. Zudem besitzt sie einen Namen. Während der Simulation ist die Nachricht einen Teil des Objektsystem, das zur Laufzeit betrachtet werden kann. Wenn eine Nachricht in der Simulation aktiviert wird, dann wird die Nachricht von der MSD - Spezifikation abgeleitet, und die Grundstruktur des Objektsystem von der SimulationDiagram abgeleitet. Nach der Erfolg von NVisualisierung wird die Struktur des Objektsystem von der SimulationDiagram aktualisiert.

Betrachten wir als nächtes die Abb. 5.4. Das Schaubild soll den Zusammenhang der Struktur des Objektsystem in Semanticmodel, die letzte aktivierte Nachricht und der NVisualisierung verdeulichen. Der Zusammenhang wird die Zusammenhang der Struktur des Objektsystem in MSD Spezifikation und der Visualisierung - Transformation abgeleitet. Zu sehen ist die Änderung des Zusammenhang, die sind die Datenstruktur von Semanticmodel (3) , Semanticmodel (5) und SimulationManager (2). Die Datenstruktur von Semanticmodel (5) wird die Grundstruktur des Objektsystems in der Datenstruktur von Semanticmodel (3) abgeleitet. Die Datenstruktur von Semanticmodel (5) wird die Ressource von letzten aktivitierten Nachricht durch den SimulationManager übergenommen.

Damit unsere Domäne des Quellmodells diese Zuordnung abdecken kann, müssen wir sie um die nötigen Information zu visualisieren erweitern. Das Grundstruktur - Modell wird durch die Meta - Modelle .simdiagram beschrieben. Deshalb habe ich die Zuordnung von 5.1 mit einige verändert, dass die Quelldomäne *mapping* mit Meta - Modell .simdiagram hinzugefügt wird, und als *source* umbenannt. In Tabelle 5.2 ist die neue Zuordnung der Domänen zusammengefasst.

Domäne	Name	Meta-Modell
Quelldomäne	simulation	simulation, runtime, stategraph, events, ecore, util, scenariorunconfiguration
Quelldomäne	source	odesign, simdiagram
Zieldomäne	visual	simdiagram, aird

Tabelle 5.2.: Die neu Zuordnung der Domänen

5.3.2. Umsetzung des Algorithmus

Im Folgenden stelle ich die Schritte für die Umwandlung vor. Diese Schritte werden benutzt, um die Ressource der Grundstruktur von Objektsystem

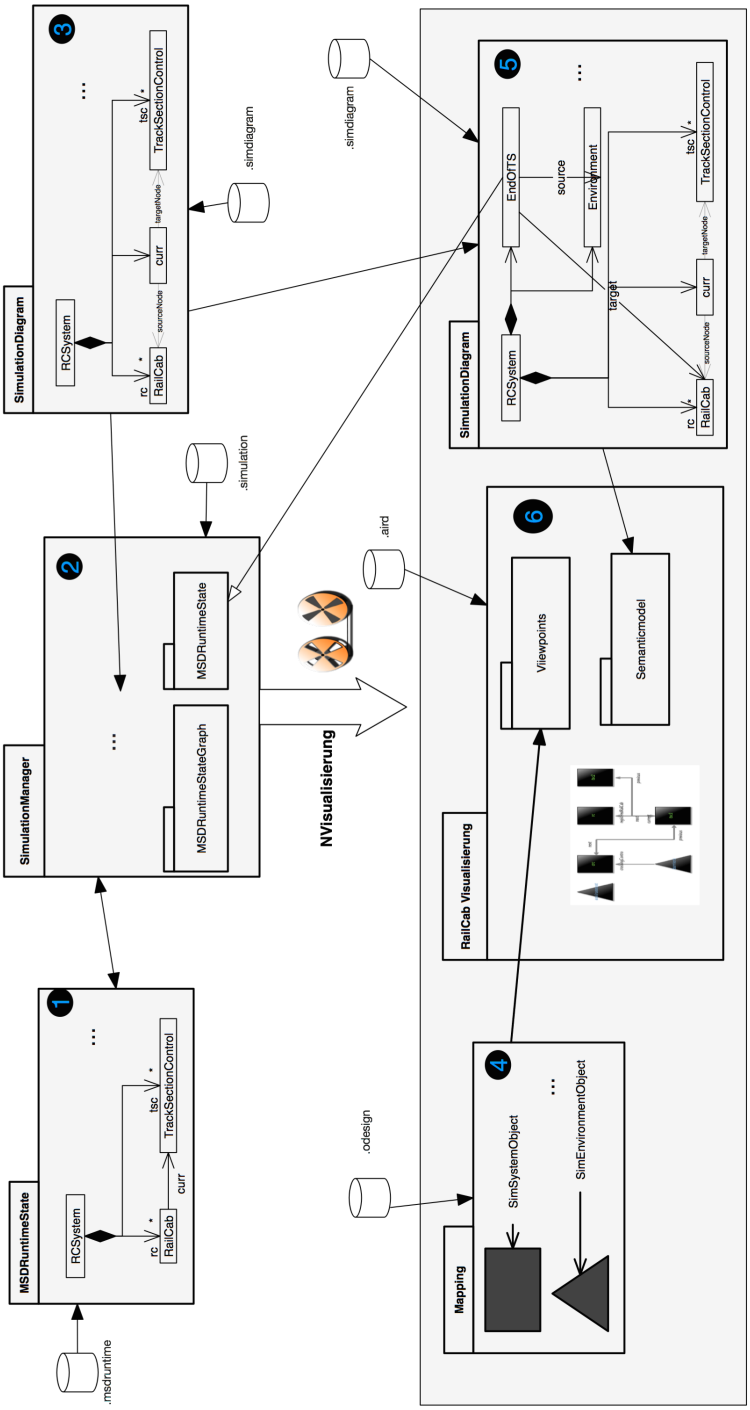

Abbildung 5.4.: Zu sehen sind die Zusammenhänge von einzelnen Komponenten von ScenarioTools, die über die letzte aktivierte Nachricht des Simulationsystems in Bezug gesetzt wurden. Ein Nachricht besteht im Wesentlichen aus einem sendenden und einem empfangenden Objekt von MSDRuntime.

von Quelldomäne *source* in Zieldomäne *visual* zu visualisieren. Die letzte aktivierte Nachricht wird die Ressource von der MSDModalMessageEvent für Objektsystem von MSDRuntime als die Ressource benutzt, um das Objekt *SimLastMessage* von SimulationDiagram zu erstellen. In der Abschnitt 5.1 habe ich schon die Eigenschaften von SimLastMessage erklärt. Das Ergebnis von diesem Unterteil wird als Semanticmodel Datei gespeichert. Die Durchführung wird mit der Klasse *DisplayLastMessageEvent.java* abgebildet. Um MSDModalMessageEvent umzuwandeln, werden die folgende Schritt durchgeführt:

1. Nachdem der Benutzer ein MSDModalMessageEvent aktiviert hat, wird die Simulation PerformStep aufgerufen. Gleichzeitig wird sie MSDModalMessageEvent als Ressource zu NVisualisierung gesendet.
2. Dann wird die Grundstruktur des Objektsystems von Semanticmodel übergenommen. Sie sammelt alle Objekte des Objektsystems von Semanticmodel in einer List.
3. Die Ressource, die sind die Ressource über sendende EObjekt, empfangende EObjekt und den Name der Nachricht, wird abgeleitet.
4. Danach wird ein Prüfung zwischen der List von Objekte in SimulationDiagram und die Ressource von MSDModalMessageEvent ausgelöst. Wenn der Name von Objekt gleich als den Name von sendenden EObjekt, dann wird diese Objekt als *source* von SimLastMessage gespeichert. Wenn der Name von Objekt gleich als den Name von empfangenden EObjekt, dann wird diese Objekt als *target* von SimLastMessage gespeichert.
5. Der Name von SimLastMessage wird immer den Name der Nachricht übergenommen.

Nach der Ausführung wird Struktur der Semanticmodel Datei zusammen mit Objekt SimLastMessage aktualisiert. Danach wird Sirius - Framework den refresh - Algorithmus aufrufen, um die Repräsentation zu aktualisieren.

5.4. Benutzeroberfläche

Nach der Erweiterung wird die Benutzeroberfläche von Simulation mit einer grafischen Oberfläche implementiert. Die Benutzeroberfläche wird wie in Abb. 5.5 angezeigt. Die Position (1) ist die grafische Oberfläche, wo die Visualisierung der Struktur von Objektsystem angezeigt wird. Wenn der Benutzer ein Ereignisnachricht in MSDModalMessageEventSelectionView aktiviert, dann wird

die Information von dieser Nachricht in der grafischen Oberfläche visualisiert. Der Ablauf der Simulation des Beispiels 2.1.1 wird genau wie im Anhang C angezeigt.

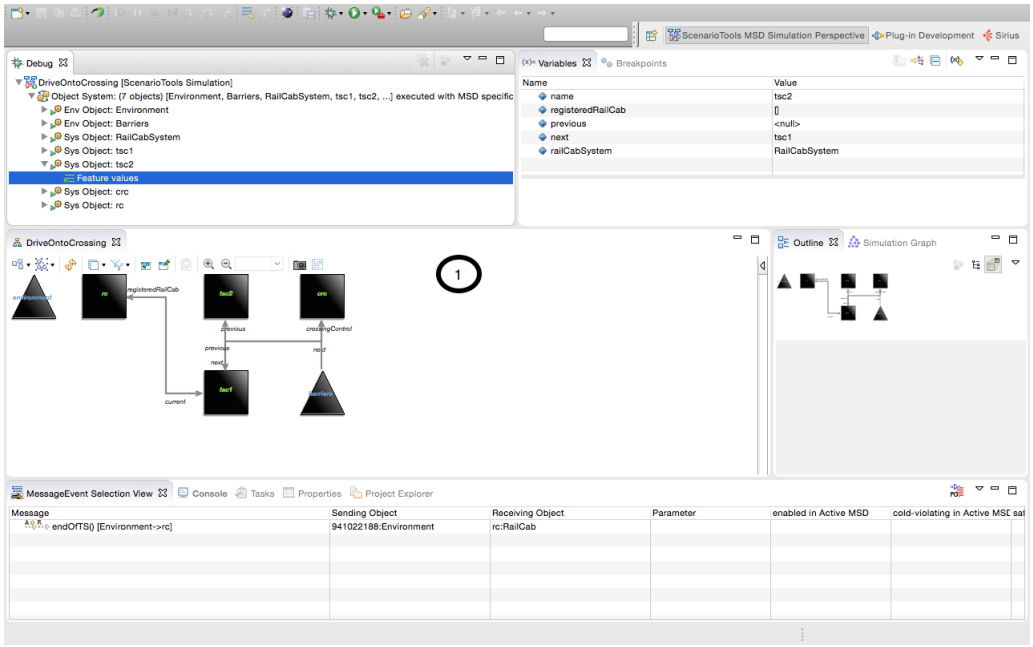

Abbildung 5.5.: Der Screenshot von Benutzeroberfläche des ScenarioTools nach der Erweiterung.

Kapitel 6.

Verwandte Arbeiten

Die vorliegende Arbeit befasste sich mit der Visualisierung für die Simulation von unrealisierbaren szenariobasierten Spezifikation, unter der Zielsetzung, den Entwickler bei der Analyse zu unterstüzen.

Eine thematisch sehr ähnliche Arbeit haben Sang Hyeok Han, Mohamed Al-Hussein, Saad Al-Jibouri und Haitao Yu in „Automated post-simulation visualization of modular building production assembly line“ [HAHAJY11] verfasst. Ziel dieser Arbeit war es den Benutzer bei der Analyse von Simulation besser zusammen mit der Visualisierung. Dabei haben sie in der Arbeit eine Methodologie, mit der den Prozess der Visualisierung als post-simulation Werkzeug automatisiert wird, vorgeschlagen. Die Methodologie wird durch die Mitteilung der Informationen von Interaktion zwischen Simulation und Visualisierung.

Ein weitere interessanter Ansatz, der sich mit der Analyse von der Veränderung in agro-ökologischen Zusammenhängen beschäftigt, ist in „An open platform to build, evaluate and simulate integrated models of farming and argo-ecosystems“ [BCG+12]. Der Ansatz konzentiert sich auf die Modellierung - Spezifikationen, um ein System für das Abhacken zu modelliern, zu simulieren und zu evaluieren. Dabei integrieren sie *RECORD platform* under Verwendung der Umgebung von VLE. Eine grafische Oberfläche wird aufgebaut, um die Kodierung zu vereinfachen. Mit diesem Werkzeug kann die Modellierung ausgeführt und die Ausgabe der Simulation analysiert werden.

Kapitel 7.

Zusammenfassung und Ausblick

Ziel dieser Arbeit was es den Benutzer während der Simulation besser bei der Analyse von unrealisierbaren szenariobasierten Spezifikation zu unterstützen. Dazu bietet ScenarioTools eine Erweiterung von Play - Out - Algorithmus, der Widersprüche und die Auswirkung der Nachricht auf System in szenariobasierten Spezifikation erkennen kann. Dieses Algorithmus bietet die Möglichkeit, um MSDs für Anforderung und Annahme zu analysieren. Es entsteht durch die wiederholte Ausführung von Nachrichten, die bei Benutzer gewählt werden. Bei umfangreichen Spezifikation ist dieser Ausführung jedoch nicht einfach zu interpretieren, weil Benutzer während der Simulation den Nachrichtenaustausch in jede Schritt nicht sehen kann. Aus diesem Grund soll die Unterstützung von dem Finden des Zustand, des Nachrichtenaustausch hin zum besseren Verstehen den Ablauf der Simulation ausgeweitet werden. Dazu sollte eine grafische Oberfläche in die Simulation integriert werden, um so ein Spiel zwischen Simulation und Benutzer zu inszenieren, bei dem die Zustände des Objektsystems während der Simulation in Grafik visualisiert werden. Sobald der Benutzer simuliert, erkennt er an welchen Zustand und welchen Ablauf das System zusammen mit der Umwelt führt und kann sie das System ausbessern.

Um die umzusetzen habe ich das Visualisierung - Algorithmus in ScenarioTools implementiert. Dabei habe ich das Ergebnis der Simulation in das grafische Format visualisiert und die Simulation so angepasst, dass beide zusammen ausgeführt werden können. Die Eigenschaft der Funktion **PerformStep**, die sowohl während der Simulation die Zustand des Objektsystems liefert, konnte bei dieser Arbeit zum Vorteil genutzt werden. So konnten, aufbauend auf dem Simulationergebnis, mit einer Visualisierung die Zustände zusammen mit der letzten aktivierten Ereignisnachricht abgedeckt werden.

Im Entwurf habe ich einen Vorschlag gemacht in welcher Weise die Benutzeroberfläche der Simulation intuitiver gestaltet werden könnte. Diesen Vorschlag konnte ich in dieser Arbeit nicht mehr umsetzen. Allerdings kann bei

der Simulation auf den SimulationDiagram Editor geachtet werden. Dieser gibt Aufschluss darüber, welcher Zustand, welche aktivierte Nachricht enthalten ist.

Anhang A.

Abbildung

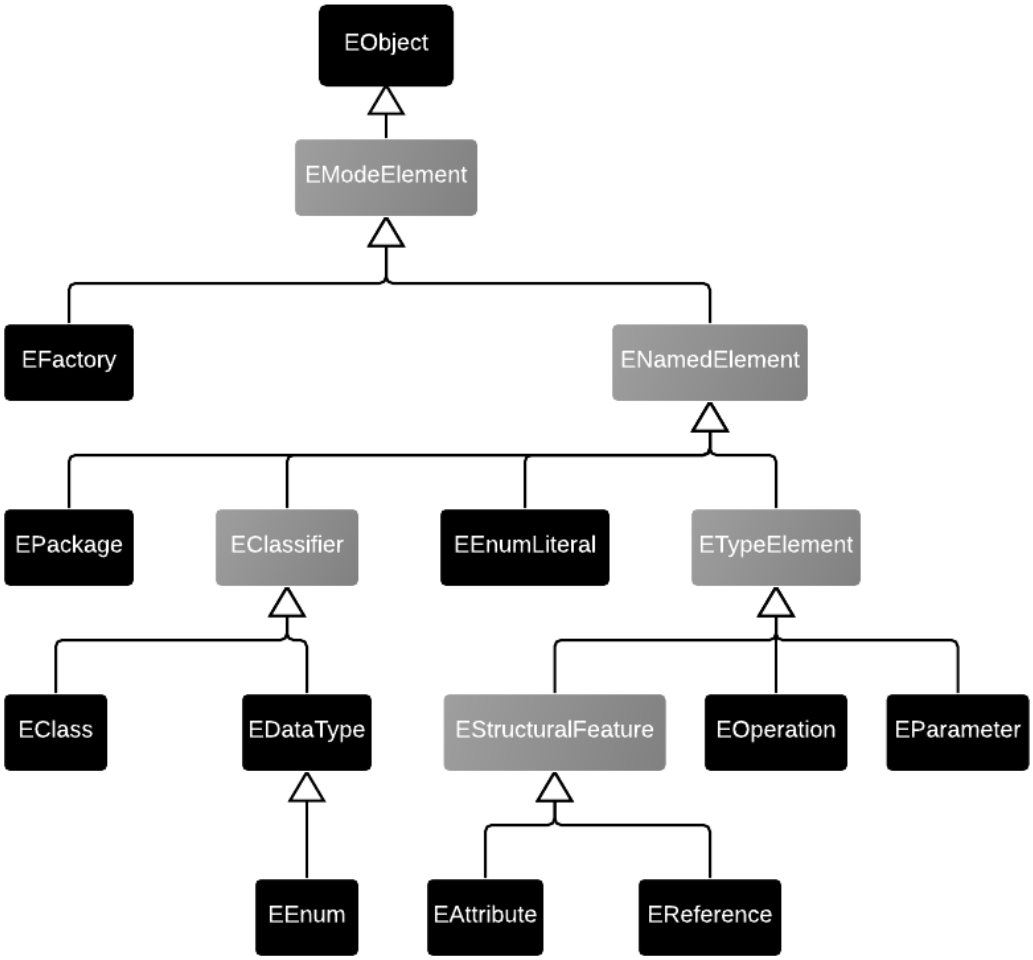

Abbildung A.1.: Die Basisklassen von Ecore Modell.

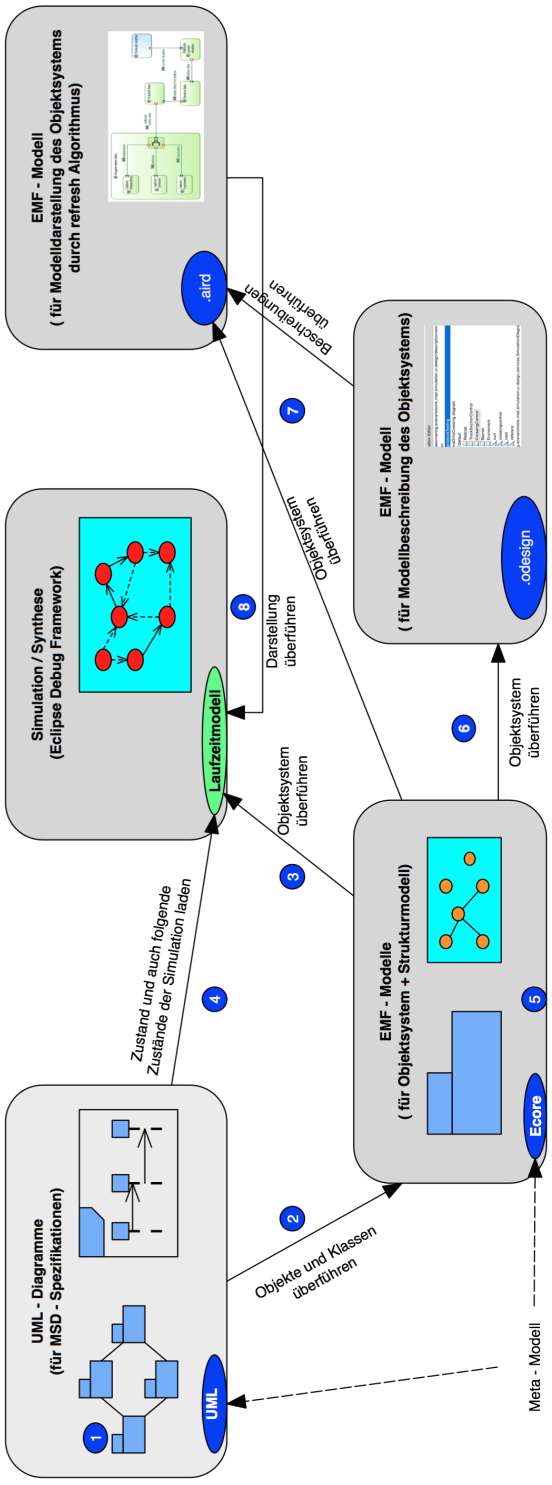

Abbildung A.2.: Die Zusammenhänge zwischen einzelnen Systemkomponenten von ScenarioTools und die Erweiterung.

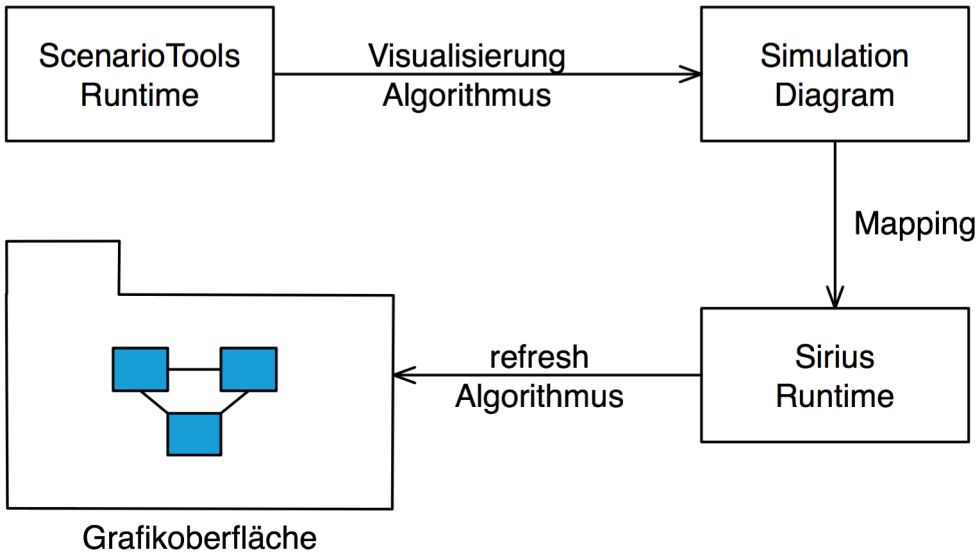

Abbildung A.3.: Verlauf von neuen Lösungsansatz.

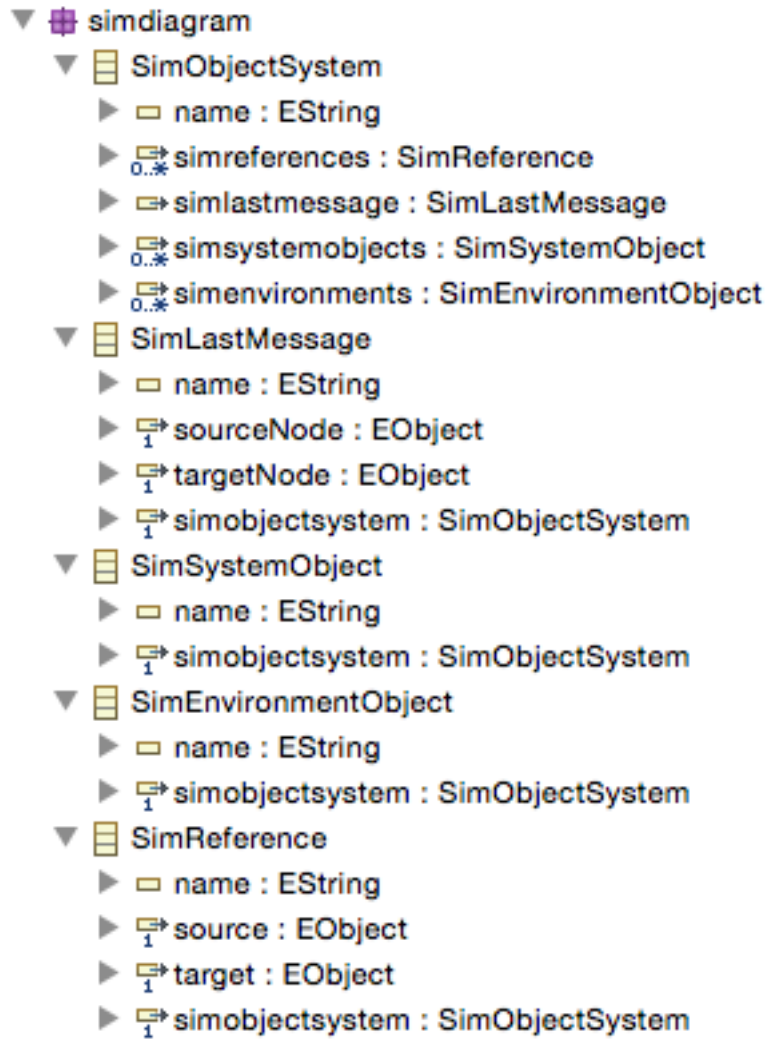

Abbildung A.4.: Struktur der Editor *SimDiagram* in EMF - Modell.

Anhang B.

SimDiagram Mapping

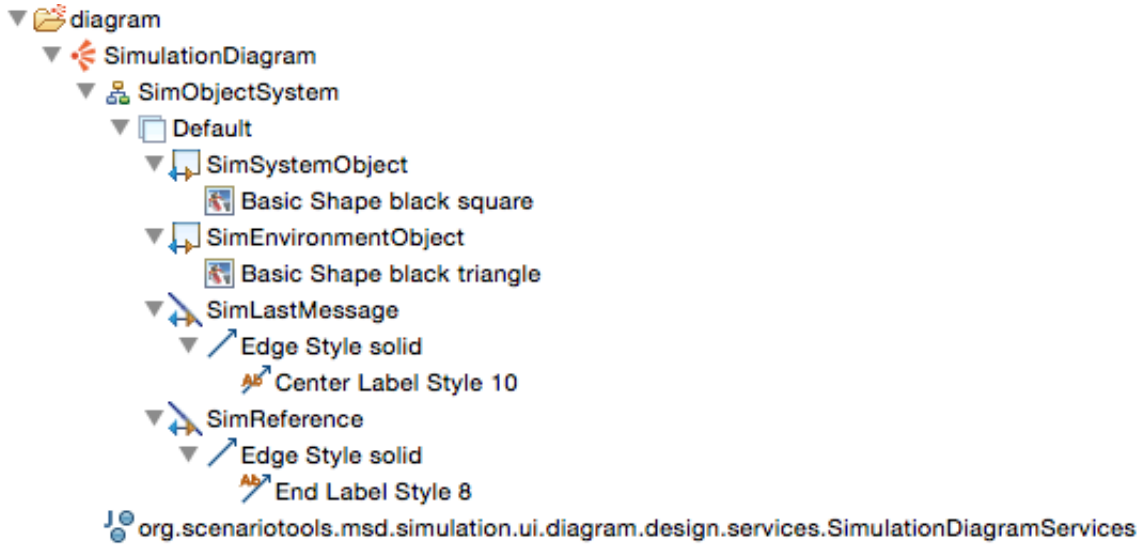

Abbildung B.1.: Die Mapping für Editor *SimDiagram* mit Hilfe von Sirius - Framework.

Anhang C.

Beispeil

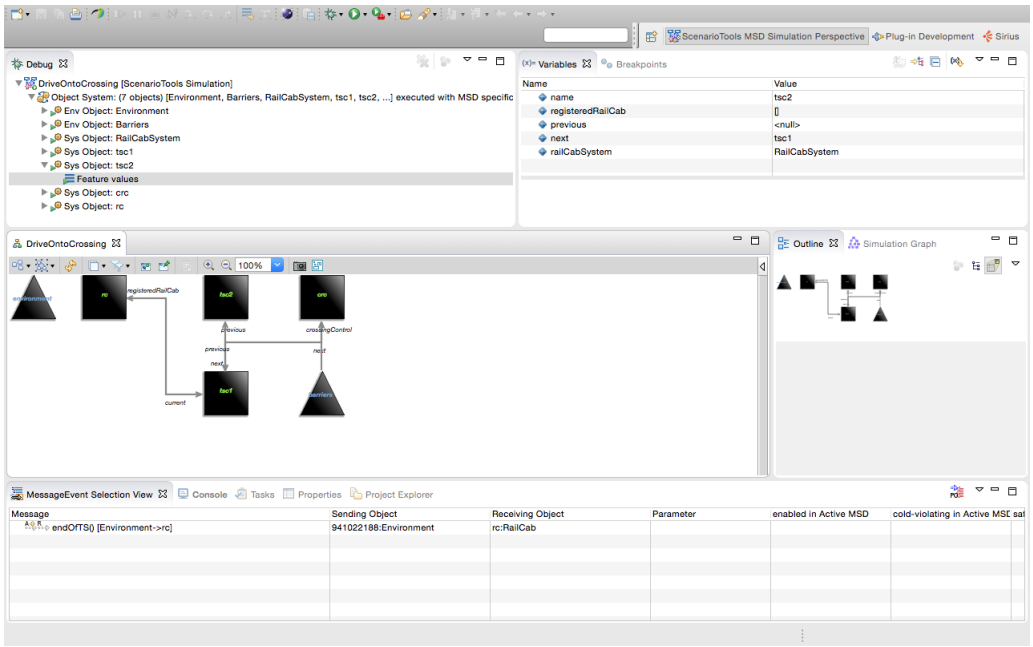

Abbildung C.1.: Die Simulation hat gestartet.

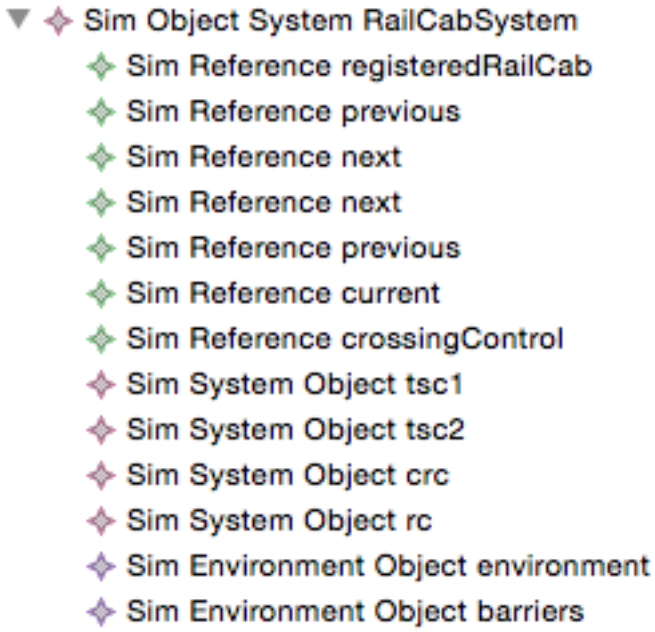

Abbildung C.2.: Grundstruktur von RailCabSystem in Editor SimDiagram modelliert wird.

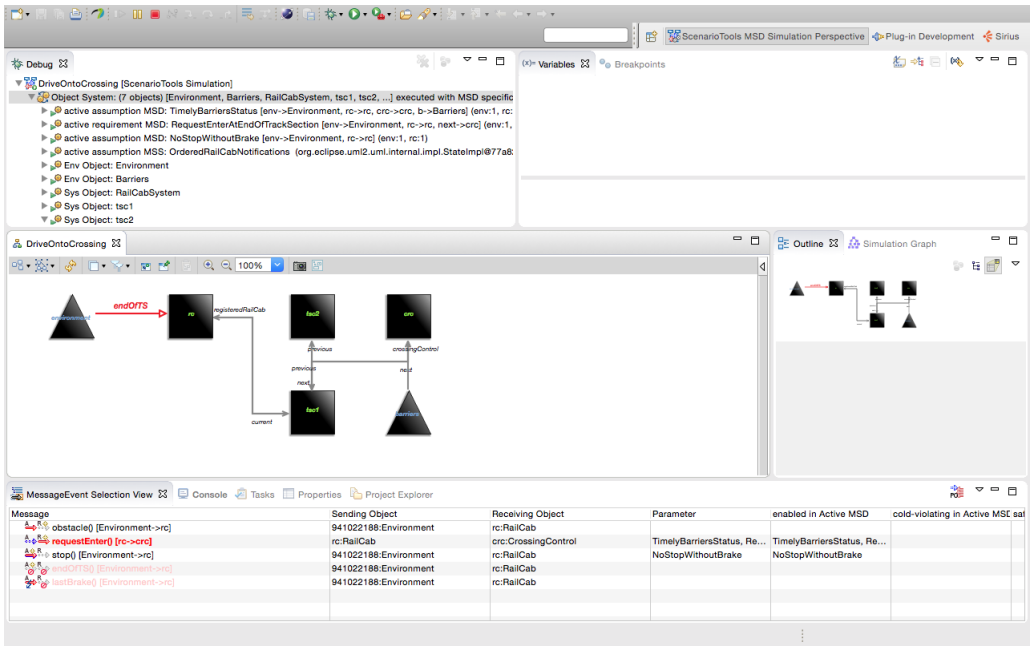

Abbildung C.3.: Die Nachricht EndOfTS wird aktiviert.

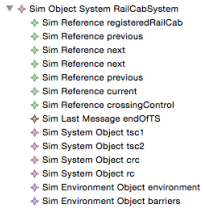

Abbildung C.4.: Das Modell mit Nachricht EndOfTS wird aktualisert.

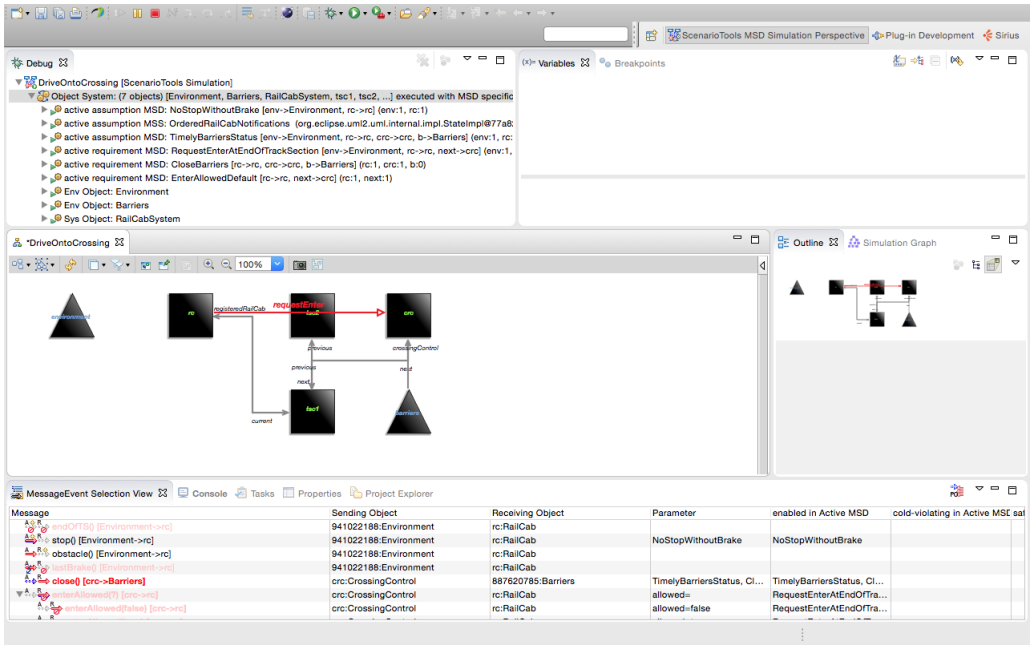

Abbildung C.5.: Die Nachricht RequestEnter wird aktiviert.

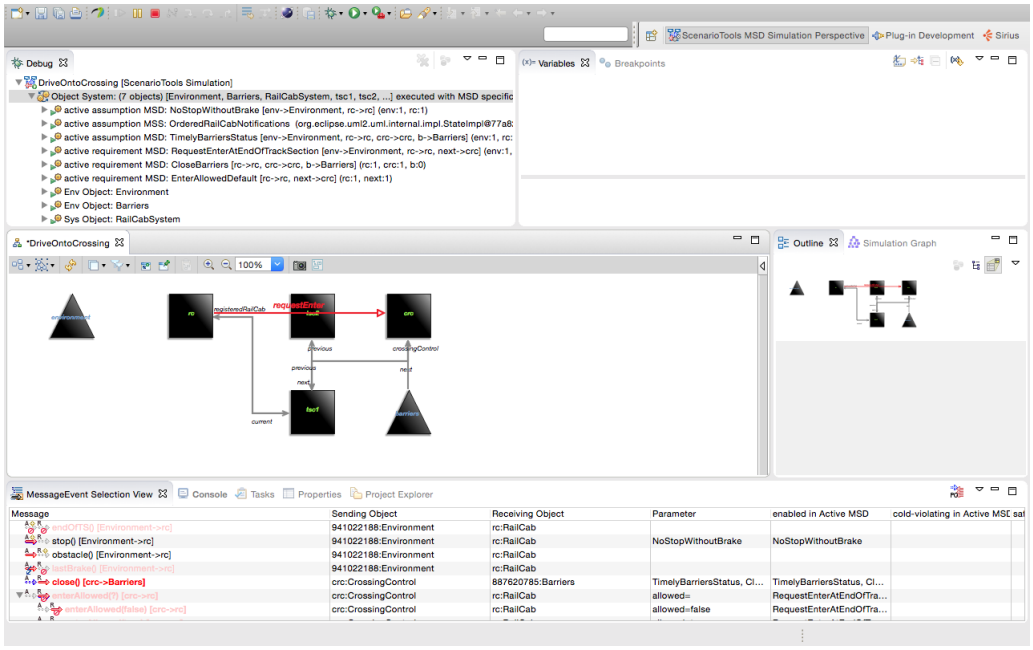

Abbildung C.6.: Das Modell mit Nachricht RequestEnter wird aktualisiert.

Literaturverzeichnis

[BCG+12] J.-E Bergez, P. Chabrier, C. Gary, M.H. Jeuffroy, D. Makowski, G. Quesnel, E. Ramat, H. Raynal, N. Rousee, D. Wallach, P. Debaeke, P. Durand, M. Duru, J. Dury, P. Faverdin, C. Gascuel-Odoux, and F. Garcia. An open platform to build, evaluate and simulate integrated models of farming and argo-ecosystem. 2012.

[Cen05] HTML Help Center. The eclipse modeling framework (emf) overview, 2005. URL: http://help.eclipse.org/indigo/index.jsp?topic=%2Forg.eclipse.emf.doc%2Freferences%2Foverview%2FEMF.html.

[Fou04] Eclipse Foundation. Eclipse modeling framework (emf), 2004. URL: http://www.eclipse.org/modeling/emf/.

[GBM13] Joel Greenyer, Christian Brenner, and Valerio Panzica La Manna. The scenariotools play-out of modal sequence diagram specifications with environment assumptions. *ECEASST*, 58, 2013. URL: http://dblp.uni-trier.de/db/journals/eceasst/eceasst58.html#GreenyerBM13.

[GPR05] Volker Gruhn, Daniel Pieper, and Carsten Röttgers. *MDA - Effektives Software-Engineering mit UML 2 und Eclipse*. Springer Verlag, Berlin, 2005.

[Gre11] Joel Greenyer. *Scenario-based Design of Mechatronic Systems*. PhD thesis, University of Paderborn, October 2011 2011. URL: http://dups.ub.uni-paderborn.de/hs/urn/urn:nbn:de:hbz:466:2-7690.

[Gro00] IEEE Architecture Working Group. IEEE Std 1471-2000, Recommended practice for architectural description of software-intensive systems. Technical report, IEEE, 2000.

[GSS14] Prof Dr. Joel Greenyer, Valerio Panzica La Manna (PhD Student), and Christian Brenner (PhD Student). Scenariotools projekt homepage, March 2014. URL: http://scenariotools.org.

[Gut14] Timo Gutjahr. Scenariotools counter-play-out simulation zur analyse von unrealisierbaren szenariobasierten spezifikationen. Master's thesis, Leibniz Universität Hannover, Fachgebiet Software Engineering, 2014.

[HAHAJY11] Sang Hyeok Han, Mohamed Al-Hussein, Saad Al-Jibouri, and Haitao Yu. Automated post-simulation visualization of modular building production assembly line. 2011.

[HM03] David Harel and Rami Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, August 2003. URL: `http://www.wisdom.weizmann.ac.il/~playbook/`.

[OT13a] Obeo and Thales. Sirius developer manual, September 2013. URL: `http://www.eclipse.org/sirius/doc/developer/Sirius%20Developer%20Manual.html`.

[OT13b] Obeo and Thales. Sirius projekt homepage, June 2013. URL: `http://www.eclipse.org/sirius/index.html`.

[SBPM09] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, Boston, MA, 2. edition, 2009.

Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit im Studiengang Informatik 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, 29. Mai 2015

_____

MINH HIEP PHAM