Diplomarbeit

Simulationssystem zur Analyse von Fehlertoleranzverfahren in physiknahen service-orientierten Architekturen am Beispiel einer virtuellen Förderanlage

Jan Krüger

Diplomarbeit
am Fachbereich Informatik
der Universität Dortmund
Lehrstuhl IV

Betreuer:
Prof. Dr. Heiko Krumm
Dipl.-Inform. Andre Pohl

11. Dezember 2007

Inhaltsverzeichnis

1 Einleitung **1**

2 Fehlertoleranz **5**
2.1 Definition und Ziel . . . 5
2.2 Redundanz . . . 6
2.2.1 Merkmale . . . 6
2.2.2 Aktivierung . . . 7
2.3 Abgrenzung . . . 9
2.4 Teilbereiche . . . 10
2.4.1 Fehlerdiagnose . . . 10
2.4.2 Fehlerbehandlung . . . 11

3 Service-orientierte Architekturen **13**
3.1 Grundlagen . . . 13
3.1.1 Dienste und Dienstbeschreibungen . . . 14
3.1.2 Rollen . . . 14
3.2 Web Services . . . 15
3.2.1 XML . . . 16
3.2.2 SOAP . . . 16
3.2.3 WSDL . . . 17
3.2.4 UDDI . . . 18
3.2.5 Weitere Kernkomponenten . . . 19
3.3 Devices Profile for Web Services . . . 20
3.3.1 WS-Eventing . . . 20
3.3.2 WS-Discovery . . . 20

4 Simulation **23**
4.1 Grundlagen . . . 23
4.2 Systeme . . . 24
4.2.1 Systemverhalten . . . 25
4.2.2 System-Ausprägungen . . . 25
4.3 Modelle . . . 26
4.3.1 Klassifikationen . . . 26
4.3.2 Modell-Sichtweisen . . . 27
4.3.3 Vor- und Nachteile . . . 28
4.4 Verfahren . . . 28
4.4.1 Zeitkontinuierliches Simulieren . . . 28
4.4.2 Zeitdiskretes Simulieren . . . 29
4.5 Zusammenfassung . . . 30

5 Technisches System **31**
5.1 Vorbemerkung . . . 31
5.2 Identifizierung . . . 32
5.2.1 Systemzweck . . . 32
5.2.2 Systemkomponenten . . . 32
5.2.3 Zusatz-Komponenten . . . 35
5.2.4 Service-orientierung der Komponenten . . . 35
5.3 Simulations-Modell . . . 35
5.3.1 Anforderungen an die Modell-Komponenten . . . 35
5.3.2 Anforderungen an die Modell-Umgebung . . . 36
5.4 Gewähltes Simulationsverfahren . . . 37

6 Anforderungen an die Simulations-Software **39**
6.1 Beliebige Modelle . . . 39
6.2 Dateioperationen und Dateiformate . . . 39
6.3 Zeit . . . 40
6.4 Externe Steuerung und Kontrolle . . . 40
6.4.1 Anforderungen an die Simulations-Umgebung . . . 40
6.4.2 Anforderungen an Komponenten des Modell-Frameworks . . . 40
6.5 Devices Profile for Web Services - Technologie . . . 41
6.6 Benutzerschnittstelle . . . 41
6.7 Protokollierung . . . 41
6.8 Modifizierbarkeit . . . 42
6.8.1 Modifizierbarkeit der Simulations-Umgebung . . . 42
6.8.2 Modifizierbarkeit des Modell-Rahmens . . . 42

7 Grundlagen der Implementierung **43**
7.1 Unterteilung der implementierten Software . . . 43
7.2 Programmiersprache Java . . . 43
7.3 Benutzte Bibliotheken . . . 45
7.3.1 SWT . . . 45
7.3.2 JDOM . . . 46
7.3.3 WS4D . . . 46

8 Modell-Framework **49**
8.1 Modell-Strukturen . . . 49
8.2 Verbindungen zwischen Komponenten . . . 50
8.2.1 Theoretischer Ansatz . . . 50
8.2.2 Implementierung . . . 51
8.3 Grundlagen der Komponenten . . . 53
8.3.1 Arten von Komponenten . . . 53
8.3.2 Einteilung der Komponenten . . . 54
8.3.3 Gemeinsame Eigenschaften aller Komponenten . . . 54
8.3.4 Gemeinsame Eigenschaften innerhalb der Komponenten-Gruppen . . . 55
8.3.5 Resultierende Struktur der Komponenten . . . 55
8.3.6 Aufteilung der Simulations-Schritte . . . 56
8.4 Implementierung der Komponenten . . . 57
8.4.1 Klassenhierarchie der Implementierung . . . 57
8.4.2 BaseSimElement . . . 58
8.4.3 EntryElement . . . 61
8.4.4 ExitElement . . . 61
8.4.5 InsideElement . . . 62
8.4.6 InPort . . . 62

8.4.7 OutPort 63
8.4.8 Conveyer 64
8.4.9 Gate 65
8.4.10 RobotGrabber 66
8.4.11 WorkStation 67
8.5 Web Services der Simulations-Elemente 68
8.5.1 Grundlagen 69
8.5.2 Services der Komponenten 69
8.5.3 Schnittstelle zwischen Service und Komponente 70
8.6 Auxiliary Devices 70
8.6.1 Grundlagen der Implementierung 71
8.6.2 Beispiel eines Auxiliary Devices 71

9 Simulations-Umgebung **73**
9.1 Architektur und Konzept 73
9.2 Details der Module 75
9.2.1 SimulatorRoof - Modul 75
9.2.2 SimControl - Modul 79
9.2.3 SimLogger - Modul 80
9.2.4 SimTimer - Modul 81
9.2.5 SimulatorRoofService - Modul 82
9.2.6 SimulationToXmlSaver - Modul 83
9.2.7 SimLoader - Modul 83
9.2.8 MainWin - Modul 84
9.2.9 NewModelCreator - Modul 87
9.2.10 Hilfs-Module 89
9.3 Threads 92
9.4 Web Service für die Simulations-Umgebung 93

10 Anwendungsbeispiel **97**
10.1 Benutztes Modell 97
10.1.1 Modell-Teile und Redundanzen 99
10.2 Externes Kontrollprogramm 100
10.2.1 Grundlagen des Kontrollprogrammes 100
10.2.2 Umsetzung der Fehlertoleranzverfahren 101
10.3 Durchgeführte Simulations-Läufe 106
10.4 Auswertung 107

11 Zusammenfassung und Fazit **111**
11.1 Erweiterungsmöglichkeiten 113
11.2 Fazit 113

Abbildungsverzeichnis **115**

Tabellenverzeichnis **117**

Literaturverzeichnis **118**

A SWT-unterstützte Betriebssysteme **121**

B XML-strukturiertes Simulations-Modell **122**

Kapitel 1

Einleitung

Betrachtet man die aktuellen Entwicklungen in der Informatik- und IT-Landschaft, so fällt ein bereits länger andauernder Trend hin zu dem Konzept der sogenannten *service-orientierten Architekturen*, kurz *SOA*, auf. Im Zentrum dieses Konzeptes stehen dabei die sogenannten *Dienste*, die in einem Netzwerk *angeboten*, *gesucht* und *genutzt* werden können. Die Definition einheitlicher *Standards*, z. B. zur Beschreibung der bereitgestellten Dienste, ermöglicht dabei das flexible Suchen und Finden eines Dienstes. Nach erfolgreicher Suche eines Dienstes und der Aushandlung der Interaktions-Bedingungen, gehen der *Dienstnutzer* und der Dienst in diesem Konzept dynamisch eine *lose Bindung* ein. Ferner ist ein weiterer zentraler Aspekt in diesem Konzept, dass die verwendeten Arten von Hard- oder Software – wie z. B. die Betriebssysteme oder die Rechner-Plattformen – für die Interaktion der einzelnen Beteiligten nicht von Belang sind.

Parallel zu dem Trend hin zu service-orientierten Architekturen ermöglicht der technologische Fortschritt, mit dem beispielsweise Prozessoren, die bei immer geringerer Größe immer mehr Leistungsfähigkeit gewinnen, die Entwicklung sogenannter *eingebetteter Systeme* (*embedded systems*) für den Einsatz in mehr und mehr Bereichen. Ein eingebettetes System ist dabei oft ein Klein- oder Kleinst-Rechner, der ganz allgemein in ein technisches Gerät eingebunden (oder eben *eingebettet*) ist. Dabei sind diese Rechner oftmals speziell für die Funktionen entworfen bzw. angepasst, für die sie in diesem technischen Gerät vorgesehen ist. Sehr oft besitzen sie die Fähigkeit, z. B. mittels Ethernet oder WLAN in ein Rechnernetzwerk eingebunden zu werden. Derartige eingebettete Systeme finden sich sowohl im häuslichen Bereich, wie auch insbesondere im *industriellen Umfeld*. In letzterem werden diese Systeme häufig bei der *Automatisierung*, also bei der Verlagerung von Aufgaben weg vom Menschen, hin zum Automaten, eingesetzt, um z. B. Maschinen zu kontrollieren und zu steuern.

Dieser Einsatz von netzwerkfähigen, eingebetteten Systemen im Rahmen der Automatisierung eröffnet nun die auch zunehmend genutzte Möglichkeit, das Konzept der service-orientierten Architekturen auf diesen Bereich zu übertragen. Daraus ergeben sich zahlreiche neue Möglichkeiten, die es ohne den Einsatz des SOA-Konzeptes in diesem Bereich so nicht geben würde. So ist es möglich, dass die einzelnen Komponenten eines automatisierten Systems Dienste zur Verfügung stellen, mit denen sehr einfach aktuelle Informationen von diesen Komponenten abgefragt werden können. Dieses wiederum ermöglicht es, diese Informationen praktisch ohne Zeitverlust in die Kontrolle der Geschäftsprozesse eines Unternehmens mit einzubeziehen. Denkbar wäre hier z. B. das Abfragen der aktuellen Tages-Produktion einer Maschine. Selbstverständlich kann der Informationsfluß an dieser Stelle auch in Richtung der Komponenten eines automatisierten Systemes gehen. Sofern diese die entsprechenden Dienste bereitstellen ist es z. B. möglich, diesen Komponenten im Rahmen dieser Dienste definierte Steuerbefehle zukommen zu lassen. Ferner wird hier mit der Umsetzung des SOA-Konzeptes auch eine sehr viel höhere Flexibilität bezüglich der Steuerung oder Kontrolle dieser Systeme erreicht. Werden z. B. neue System-Komponenten installiert, melden sich die von diesen Komponenten bereitgestellten Dienste entsprechend im Netzwerk an, und können direkt benutzt werden. Weitere, möglicherweise sehr aufwändige Maßnahmen, um die Kompatibilität bezüglich

der Kommunikation dieser neuen Komponenten mit dem bestehenden System zu gewährleisten, entfallen hier. Als Beispiel könnte man sich ein automatisiertes Waren-Beförderungssystem vorstellen, bei dem Waren mittels ferngesteuerter Transportwagen befördert werden, welche über eine WLAN-Schnittstelle verfügen. Soll nun ein neuer solcher Wagen mit in das System eingebunden werden, gibt dieser die Anwesenheit seines Dienstes in dem entsprechenden WLAN bekannt, wo die für die Steuerung aller Wagen zuständige Kontrollinstanz diesen neuen Dienst entdeckt. Da dieser Kontrollinstanz die Struktur dieses Dienstes bekannt ist und sie diesen Dienst daher nutzen kann, kann dieser neue Transportwagen sofort und ohne weitere externe Konfigurations-Maßnahmen mit in dieses automatisierte Waren-Beförderungssystem eingebunden werden.

Ein Problem, das mit der zunehmenden Automatisierung von Systemen einhergeht, sind Ausfälle von Komponenten innerhalb dieser Systeme. Derartige Fehler können bis hin zum Stillstand des gesamten Systems führen, und somit große *Folgekosten* (z. B. wegen eines Produktions-Ausfalls) verursachen. Deswegen ist es erstrebenswert, zur Kontrolle und Steuerung eines solchen Systems zusätzliche Verfahren einzusetzen, die auch bei Ausfällen von bzw. Fehlern an einigen der Komponenten des Systems die Funktion des Gesamt-Systems aufrechterhalten. Bei derartigen Verfahren handelt es sich um die sogenannten *Fehlertoleranzverfahren*. Die hinter diesen Verfahren stehenden Ideen und Ansätze sind nicht unbedingt neu. Bisherige Umsetzungen dieser Verfahren waren aber zumeist wegen der statischen Anbindung der Verfahren an die zu kontrollierenden Systeme nur wenig dynamisch, d. h. bei strukturellen Veränderungen im System mussten zumeist auch große Änderungen bei der Anbindung des Fehlertoleranzverfahrens an das System vorgenommen werden. Der Einsatz von SOA-basierten Konzepten ermöglicht hier im Gegensatz dazu nun eine wesentlich flexiblere und somit einfachere Anbindung dieser Kontroll- und Steuerungsverfahren an die jeweiligen Systeme.

Praktische Erfahrungen bezüglich der Anbindung von Fehlertoleranzverfahren an ein nach dem SOA-Konzept aufgebautes automatisiertes System existieren bisher allerdings kaum. Somit lassen sich keine verlässlichen Aussagen darüber treffen, wie zuverlässig eine derartige Anbindung bzw. ein derartig angebundenes Fehlertoleranzverfahren arbeitet. Daher ist es notwendig, in diesem Bereich entsprechende *Tests* durchzuführen und auszuwerten. Diese Tests ließen sich dabei z. B. an einem realen, dem SOA-Konzept entsprechenden automatisierten System testen, für das ein passendes, die Dienste dieses Systems nutzendes Fehlertoleranzverfahren implementiert wird. Dieses Vorgehen wäre allerdings mit zahlreichen Nachteilen wie z. B. möglicherweise großen Kosten oder Risiken verbunden. Stattdessen bietet es sich eher an, hier eine rechnergestützte *Simulation* eines derartigen Systems zu verwenden, die an die Stelle des physisch vorhandenen Systems tritt, und mit der dann das entsprechende Fehlertoleranzverfahren interagiert. Sofern dabei die Simulation das zugrundeliegende automatisierte System hinreichend genau nachbildet ist es möglich, das anhand dieser Simulation getestete Fehlertoleranzverfahren direkt auf das reale System zu übertragen.

Im Rahmen dieser Diplomarbeit soll ein Werkzeug entwickelt und implementiert werden, dass die Simulation derartiger, dem Konzept der service-orientierten Architekturen entsprechender, automatisierter Systeme ermöglicht. Dieses Werkzeug selbst soll dabei ebenfalls unter Berücksichtigung des SOA-Konzeptes entwickelt werden.

Zur Realisierung dieses Ziels ist es zunächst notwendig, die hier verwendeten Konzepte zu untersuchen bzw. eine Einarbeitung in diese durchzuführen. Dieses umfasst insbesondere eine Einarbeitung in die Verfahren und Vorgehensweisen zur Umsetzung bzw. Gewährleistung von Fehlertoleranz, sowie in das Konzept der service-orientierten Architekturen, der hinter diesen stehenden Grundlagen, und von konkreten Umsetzungen des SOA-Konzeptes. Außerdem ist es notwendig, verschiedene Verfahren zur Durchführung von Simulationen zu untersuchen, um das für die hier gestellten Anforderungen am besten geeignete Verfahren zu identifizieren und anschließend bei dem Entwurf und der Implementierung des Simulations-Werkzeuges entsprechend umzusetzen. Nach Erarbeitung dieser Grundlagen erfolgt der Entwurf und die Implementierung des Simulations-Werkzeuges, wobei insbesondere dem genauen Entwurf des Modells für das zu simulierende automatisierte System große Bedeutung beigemessen wird. Abschließend wird das Simulations-Werkzeug getestet, indem mittels diesem Werkzeug ein Modell eines entsprechenden Systems erzeugt wird, und Simulationen dieses Modells in Verbindung mit einem dafür passenden Fehlertoleranz-

verfahren durchgeführt werden.

Der Entwurf und die Implementierung des Simulations-Werkzeuges waren erfolgreich. Mit dem Werkzeug ist es möglich, Modelle in weiten Grenzen beliebiger Automatisierungssysteme zu erstellen, zu konfigurieren, und diese daran anschließend *mit oder ohne* Verwendung eines zusätzlichen, externen Fehlertoleranzverfahrens zu simulieren. Sowohl bei den einzelnen möglichen *Komponenten* der Simulations-Modelle, als auch bei der entwickelten *Simulations-Umgebung* wurde das Konzept der *service-orientierten Architekturen* umgesetzt. Die Anforderung, ein externes Fehlertoleranzverfahren mittels der somit bereitgestellten *Dienste* an das Simulations-Werkzeug anbinden und somit testen zu können, konnte ebenfalls vollständig umgesetzt werden, wie nicht zuletzt zahlreiche erfolgreich durchgeführte Test-Simulationen zeigten.

Die ersten nun folgenden Kapitel dieser Diplomarbeit befassen sich mit den zuvor genannten Grundlagen. So werden in Kapitel 2 zunächst die Grundlagen von Fehlertoleranzverfahren erarbeitet. Dabei werden zahlreiche, im weiteren Verlauf dieser Diplomarbeit benutzte Begriffe eingeführt und erläutert. Im darauffolgenden Kapitel 3 wird näher auf das Konzept der service-orientierten Architekturen und deren Details eingegangen. Außerdem werden mit den sogenannten *Web Services* und dem *Devices Profile for Web Services* zwei konkrete Ansätze zur Implementierung dieses Konzeptes vorgestellt. Kapitel 4 untersucht zum Abschluß der theoretischen Grundlagen dieser Diplomarbeit die verschiedenen Ansätze und Konzepte bei dem Entwurf von Simulationen, sowie die Strukturen, die Simulationen und Simulations-Modellen zugrundeliegen.

In Kapitel 5 wird das zu simulierende automatisierte System einer genaueren Betrachtung unterzogen. Dabei wird insbesondere der Systemzweck untersucht, sowie die einzelnen Komponenten identifiziert, aus denen ein derartiges System und somit ein Modell für dieses System bestehen kann. Daraus werden die Anforderungen an dieses Simulations-Modell abgeleitet, was schließlich die Auswahl eines für ein solches Modell am besten geeignetem Simulationsverfahren begründet.

Das darauffolgende Kapitel 6 gibt einen Überblick über alle Anforderungen, die explizit an die zu entwickelnde Simulations-Software gestellt wurden, und somit beim Entwurf und der Implementierung zu berücksichtigen waren. Auf die Implementierung wird genauer in Kapitel 7 eingegangen. Dort wird beispielsweise erläutert, in welche logischen Teile sich die entwickelte Software einteilen lässt, oder aber auch, welche externen Bibliotheken bei der Implementierung der Software benutzt wurden.

Die Kapitel 8 und 9 erläutern daran anschließend den Entwurf und die Implementierung des entstandenen *Modell-Frameworks* und der entstandenen *Simulations-Umgebung* im Detail. Dabei wird sowohl auf die beim Entwurf getroffenen Design-Entscheidungen eingegangen und Begründungen für diese Entscheidungen gegeben, als auch die konkreten Implementierungs-Details erläutert.

In Kapitel 10 wird schließlich eine konkrete Anwendung des Simulations-Werkzeuges vorgeführt. Dazu wird zunächst ein Simulations-Modell entwickelt und vorgestellt. Dieses wird im Anschluß mehrfach simuliert, und die Ergebnisse dieser Simulations-Läufe werden ausgewertet. Bei diesen Simulations-Läufen wird die Ausprägung der eingesetzten Fehlertoleranz-Maßnahmen bewußt variiert; zur besseren Einschätzung der eingesetzten Verfahren erfolgen auch einige Simulations-Läufe ohne jeglichen Einsatz eines Fehlertoleranz-Verfahrens. Kapitel 11 dient dann zum Abschluß dieser Diplomarbeit dazu, ein Fazit zu ziehen, und mögliche Erweiterungs- oder Ergänzungsmöglichkeiten aufzuzeigen.

Kapitel 2

Fehlertoleranz

Dieses Kapitel erläutert zunächst, was sich hinter dem Begriff der Fehlertoleranz in dem hier benutzten Sinne verbirgt. Anschließend wird der im Rahmen der Fehlertoleranz verwendete Begriff der *Redundanz* eingeführt, und eine Abgrenzung der Fehlertoleranz von der *Fehlervermeidung* vorgenommen. Abschließend wird dargestellt, wie Verfahren zur Gewährleistung von Fehlertoleranz funktionieren bzw. in welche Teilbereiche man diese Verfahren unterteilen kann, und welche Funktionen die einzelnen Teilbereiche darstellen.

2.1 Definition und Ziel

Vor der Definition des Begriffes Fehlertoleranz sei hier nun zunächst der Begriff des *Systems* kurz erläutert: unter diesem Begriff ist in diesem Zusammenhang die Kombination von einzelnen (System-) Komponenten mit jeweils eigenem, komponentenspezifischem Verhalten und eigener, komponentenspezifischer Teilaufgabe zu verstehen, die in ihrem aufeinander abgestimmten Zusammenspiel einen spezifizierten Systemzweck erfüllen sollen. Aufsetzend auf diesem Begriff des Systems definiert Echtle [Ech90] den Begriff der *Fehlertoleranz* wie folgt:

> „Fehlertoleranz (fault tolerance) bezeichnet die Fähigkeit eines Systems, auch mit einer begrenzten Anzahl fehlerhafter Komponenten seine spezifizierte Funktion zu erfüllen.“

Das Ziel von Verfahren zur Gewährleistung von fehlertolerantem Systemverhalten liegt also darin, dass ein durch ein solches Verfahren gesteuertes System auch beim Auftreten von Fehlern in einzelnen Komponenten dieses Systems (wie z. B. Ausfall) seinen Systemzweck als solchen weiterhin erfüllt, d. h. dass dieses System „funktionsbereit“ gehalten wird. Anders formuliert: eine Eingabe an das System, die bei fehlerfreiem Arbeiten des Systems zu einem bestimmten Ergebnis führt, soll auch bei Auftreten von tolerierbaren Fehlern an einer oder mehreren Komponenten innerhalb des Systems zum gleichen Ergebnis führen.

Es lässt sich also sagen, dass die Verwendung von Fehlertoleranzverfahren zur Kontrolle bzw. Steuerung eines Systems mit potentiell fehleranfälligen Teilkomponenten dazu dient, die *Zuverlässigkeit* dieses Systems nach außen hin zu erhöhen. Zuverlässigkeit bezeichnet hier gemäß [Ech90] allgemein „die Fähigkeit eines Systems, während einer vorgegebenen Zeitdauer bei zulässigen Betriebsbedingungen die spezifierte Funktion zu erbringen“.

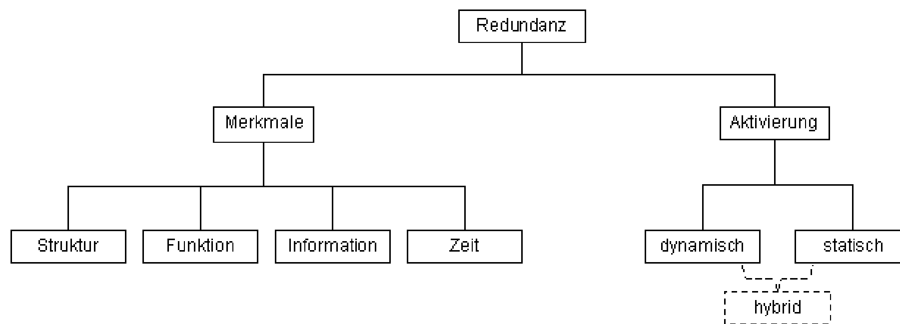


Abbildung 2.1: Unterteilung der Redundanz nach [Ech90]

2.2 Redundanz

Ein im Rahmen der Fehlertoleranz wichtiger Begriff ist der Begriff der *Redundanz*. Dieser Begriff soll hier zunächst näher erläutert und unterteilt werden. Gemäß [Arb82] ist Redundanz wie folgt definiert:

> „Redundanz bezeichnet das funktionsbereite Vorhandensein von mehr technischen Mitteln, als für die spezifizierte Nutzfunktion eines Systems benötigt werden (die Fehlertoleranz-Fähigkeit selbst wird in diesem Zusammenhang nicht als eigentliche Nutzfunktion eines Systems angesehen)."

Redundante Mittel sind gemäß dieser Definition alle Mittel, die sich im nicht-fehlerbehafteten Systembetrieb aus der Gesamtmenge aller in einem System zur Verfügung stehenden Mittel entfernen lassen, ohne dass der eigentliche Systemzweck beeinträchtigt wird.

Eine erste Unterteilung des Begriffes der Redundanz lässt sich vornehmen in die *Merkmale* der Redundanz, und in die Art der *Aktivierung* der Redundanz. Dieses und weitergehende Unterteilungen der Merkmale und der Aktivierung wird dargestellt in Abbildung 2.1.

2.2.1 Merkmale

Die redundanten Mittel lassen sich gemäß [Ech90] durch die vier hier folgenden Merkmale charakterisieren und einteilen. Hierbei ist zu beachten, dass im Prinzip jede Realisierung von redundanten Mitteln alle diese vier Merkmale aufweist, so dass eine Einteilung nach dem am stärksten ausgeprägtem Merkmal zu erfolgen hat.

Strukturelle Redundanz

Bei struktureller Redundanz sind Komponenten innerhalb des Systems mehrfach vorhanden, obwohl sie in dieser mehrfachen Ausführung im nicht-fehlerbehafteten Systembetrieb nicht benötigt werden. Fällt nun eine benötigte Komponente aus, so kann dieser Ausfall durch eine der zuvor noch nicht benötigten Komponenten ausgeglichen werden, indem diese Aufgaben der ausgefallen Komponente übernimmt.

Als Beispiel könnte man sich hier ein Computer-Netzwerk mit zwei redundanten SMTP-Servern vorstellen, in dem automatisch der sekundäre Server aktiviert und auf diesen umgeschaltet wird (z. B. mittels Änderung des entsprechenden DNS-Eintrages), wenn der primäre Server fehlerhaft zu arbeiten scheint (wenn er beispielsweise nicht mehr auf einen regelmäßig ausgeführten Test in Form eines „Ping" antwortet). Dieser sekundäre Server wäre dann eine strukturelle Redundanz.

Funktionelle Redundanz

Funktionelle Redundanz bezeichnet die Erweiterung des Systems um zusätzliche Funktionen, die nicht dem eigentlichen Systemzweck, sondern ausschließlich dem Gewährleisten der Fehlertoleranz dienen. Derartige funktionelle Redundanzen werden dementsprechend auch von jedem fehlertoleranten System benötigt, da ohne diese das Erkennen und Lokalisieren von Fehlern (und somit die Reaktion auf diese) nicht möglich wäre. Eine solche zusätzliche Funktion ist z. B. das im zuvor angeführten Beispiel benutzte permanente „Pingen“ des primären SMTP-Servers, oder auch das beschriebene Umschalten auf den sekundären SMTP-Server mittels der dafür vorgesehenen, redundanten Funktion.

Informationsredundanz

Wird informationsredundant gearbeitet, so werden neben den minimalen, für den regulären Systembetrieb nötigen Informationen (den sog. Nutzinformationen, d. h. den Informationen, mit denen die einzelnen Systemkomponenten arbeiten) zusätzlich weitere Informationen gehalten. Diese gestatten es dann z. B., durch Abgleich mit den Nutzinformationen zu erkennen, ob letztere korrekt sind. Ein Beispiel hierfür sind Prüfsummen. Bei informationsredundanter Vorgehensweise wird immer auch zusätzlich die zuvor genannte funktionelle Redundanz gebraucht, eben um z. B. derartige Abgleiche vornehmen zu können.

Zeitredundanz

Zeitredundanz ist dadurch gekennzeichnet, dass zusätzliche Zeit innerhalb des Systems bzw. innerhalb der Ausführung des Systemzwecks vorgesehen ist, die über den Zeitbedarf des Systems im regulären, d. h. nicht-fehlerbehafteten Betrieb hinausgeht. Diese Zeitredundanz ermöglicht es im Fehlerfall, z. B. eine auf einer fehlerbehafteten Komponente abgebrochene Aufgabe auf einer zweiten, strukturell redundanten Komponente erneut zu starten, und auf die Fertigstellung dieser Aufgabe zu warten. Wäre hier keine Zeitredundanz vorgesehen, so wäre dieses so nicht möglich, da dann die zu erledigende Aufgabe nicht mehr rechtzeitig abzuschließen wäre.

2.2.2 Aktivierung

Ein weiteres Kriterium, nach dem Redundanzen eingeteilt bzw. mit dem Redundanzen beschrieben werden können, ist gemäß [Ech90] der Zeitpunkt ihrer Aktivierung.

Statische Redundanz

Statische Redundanz bezeichnet die Existenz von redundanten Mitteln, die während des gesamten Zeitraumes aktiv sind, in dem das zugrundeliegende System aktiv ist. Durch ihre Aktivitäten unterstützen sie die Gesamtfunktion des Systems. Daher werden sie auch als *funktionsbeteiligte Redundanzen* bezeichnet. Eine derartige Redundanz wäre das im obigen Beispiel genannte ständige „Pingen“ des primären Servers; ganz offensichtlich muss dieses während der gesamten Zeit durchgeführt werden, in der das System aktiv ist. So wird dann eine systemunterstützende Zusatzfunktion, nämlich das Erkennen eines Fehlers am primären Server, geleistet. Dementsprechend handelt es sich hier um eine *statische funktionelle Redundanz in Form von Zusatzinformationen*. Eine andere Ausprägungen der statischen Redundanz ist z. B. die *statische strukturelle Redundanz*, bei der es mehrere gleichartige Komponenten $K^1, K^2, \ldots, K^m$ zur Erbringung der gleichen Funktion f gibt. Diese bilden typischerweise ein sogenanntes *n-von-m-System*, bei dem die Realisierung eines Prozesses P mittels einer Mehrheitsentscheidung getroffen wird, d. h. die Ergebnisse der einzelnen Komponenten werden verglichen, und das von der Mehrheit der Komponenten bestimmte Ergebnis wird als fehlerfrei angenommen. Ein Beispiel für ein n-von-m-System in Form eines 2-von-3-Systems findet sich in Abbildung 2.2. Weitere Ausprägungen statischer Redundanz sind (vgl.

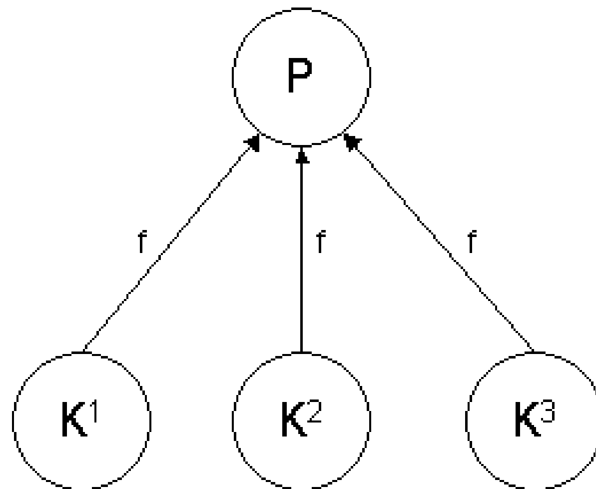


Abbildung 2.2: Statische Redundanz als 2-von-3-System nach [Ech90]

[Ech90]) die *statische funktionelle Redundanz in Form von Diversität*, die *statische Informationsredundanz*, und die *statische Zeitredundanz.*

Dynamische Redundanz

Dynamische Redundanzen sind im Gegensatz zu statischen Redundanzen nicht ständig aktiv, sondern werden erst im Falle des Auftretens eines Fehlers im System aktiviert. Daher werden sie auch als *Reserveredundanzen* bezeichnet. Der im obigen Beispiel genannte sekundäre SMTP-Server stellt somit eine dynamische strukturelle Redundanz dar, und die Funktionen, die diesen sekundären Server aktivieren und das System auf die Verwendung dieses Servers umstellen sind dynamisch funktionelle Redundanzen.

Ferner lassen sich insbesondere in Systemen mit dynamisch strukturellen Redundanzen die Komponenten unterscheiden in *Primär- und Ersatzkomponenten.* Letztere werden auch als Reserve- oder Sekundärkomponenten bezeichnet. Im fehlerfreien Betrieb laufen zwecks Erfüllung des Systemzwecks nur die Primärkomponenten; erst, wenn ein Fehler auftritt, werden die nötigen Ersatzkomponenten aktiviert. Je nachdem, ob in einem System ausschließlich Primär-, oder auch Ersatzkomponenten aktiv sind, spricht man hier von einem *Primärsystem* oder aber einem *Ersatzsystem.*

Das Umschalten auf ein Ersatzsystem erfordert unter Umständen gewisse Vorbereitungen der Ersatzkomponenten, und somit eine gewisse Zeit. Diese ist abhängig davon, wie das Ersatzsystem auf die zu erfüllende Aufgabe vorbereitet werden muss. Hier kann unterschieden werden in die sogenannte *heiße Reserve*, bei der alle nötigen strukturellen Ersatzkomponenten in gewissen Zeitabständen vorsorglich vorbereitet werden, und in die sogenannte *kalte Reserve*, bei der diese Vorbereitung erst erfolgt, wenn ein Fehler im System aufgetreten ist. Der Vorteil der heißen Reserve liegt darin, dass im Fehlerfall schneller umgeschaltet werden kann. Andererseits unterliegen Komponenten in einer kalten Reserve einer geringeren Beanspruchung, da diese nicht dauerhaft eingeschaltet sein müssen. Dieses kann zu einer längeren Lebenserwartung bei den Komponenten einer kalten Reserve führen.

In dem hier wiederholt angeführten Beispiel würde der sekundäre Server eine heiße Reserve darstellen, wenn er dauerhaft eingeschaltet wäre und in regelmäßigen Abständen mittels einer Speicherabbildung auf den aktuellen Zustand des primären Servers aktualisiert werden würde. Würde er andererseits erst eingeschaltet und auf den aktuellen Zustand des primären Servers aktualisiert werden, wenn an diesem ein Fehler aufgetreten ist (mögliche praktische Probleme dabei, aus einem z. B. nicht mehr reagierenden Server den aktuellen Zustand auszulesen seien hier jetzt vernachlässigt), so würde er eine kalte Reserve darstellen.

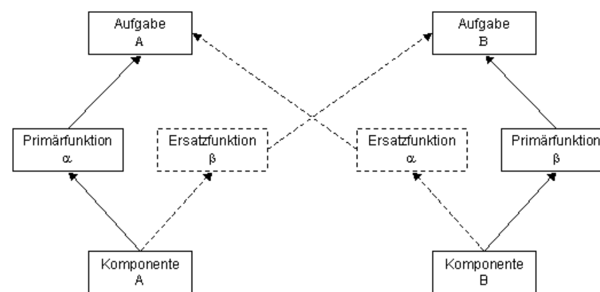


Abbildung 2.3: Gegenseitige Redundanz der beiden Systemkomponenten A und B nach [Ech90]

Bei dynamischer Redundanz dürfen Ersatzkomponenten im fehlerfreien Systembetrieb nicht aktiviert werden, um die Funktionen wahrzunehmen, für die sie redundant sind. Sehr wohl dürfen diese Ersatzkomponenten im Normalbetrieb aber *andere* als diese redundanten Funktionen erbringen. Dieses führt gemäß [BEG86] nun zu einer weiteren Unterteilung der dynamisch strukturellen Redundanz in

- gegenseitige Redundanz, bei der sich Systemkomponenten gegenseitig als Ersatzkomponenten zur Verfügung stehen, d. h. im Fehlerfall eine Komponente zusätzlich zu ihren eigenen Funktionen auch noch redundant zu Funktionen der fehlerhaften Komponente ist und diese übernimmt,
- fremdgenutzte Redundanz, bei der Ersatzkomponenten andere als ihre Redundanzfunktionen bereitstellen, die im Fehlerfall allerdings evtl. storniert werden (müssen), und schließlich
- ungenutzte Redundanz, bei der Ersatzkomponenten keinerlei sonstige Funktionen bereitstellen und rein passiv auf Aktivierung im Fehlerfall warten.

Abbildung 2.3 verdeutlicht noch einmal das Prinzip der gegenseitigen Redundanz: neben ihrer Primärfunktion α stellt Komponente A auch noch eine Ersatzfunktion β für die Primärfunktion β von Komponente B zur Verfügung. Sollte an Komponente B an Fehler auftreten, so übernimmt Komponente A mittels dieser Ersatzfunktion die Aufgabe B (analog für Komponente B).

Hybridredundanz

Hybridredundanz kann vorliegen, wenn mit den Komponenten eines Systems zugleich statische und dynamische Redundanzen realisiert werden können. Bei dieser Form der Redundanz wird ein statisch redundantes Mittel dynamisch während des Systemausführung verändert. Beispielsweise wäre dieses der Fall, wenn zur Lösung eines Problems mehr statische Redundanzen vorliegen, als zur Lösung benötigt werden, und dann dynamisch aus der Menge dieser statischen Redundanzen eine Teilmenge ausgewählt wird, mit der das Problem gelöst werden kann.

2.3 Abgrenzung

Wie in Abschnitt 2.1 herausgestellt liegt das Hauptziel bei der Verwendung von Fehlertoleranzverfahren zur Kontrolle von Systemen mit fehleranfälligen Komponenten darin, die Zuverlässigkeit dieser Systeme nach außen hin zu erhöhen. Eine andere, entgegengesetzte Vorgehensweise zur Erhöhung der Zuverlässigkeit liegt in der *Fehlervermeidung*. Gemäß [Ech90] bezeichnet „Fehlervermeidung (fault avoidance) (...) die Verbesserung der Zuverlässigkeit durch Perfektionierung der konstruktiven Maßnahmen, um das Auftreten von Fehlern von vornherein zu vermeiden. (...)“.

Zu erreichen ist dieses, indem z. B. vor der Inbetriebnahme des Systems eine hohe Anzahl von Tests mit anschließenden Verbesserungen durchgeführt wird, indem geeignetere Materialen für die Komponenten des Systems verwendet werden, oder auch, indem die Herstellungstechniken für diese einzelnen Komponenten perfektioniert werden. Ganz offensichtlich sind diese Vorgehensweisen zur Fehlervermeidung allerdings

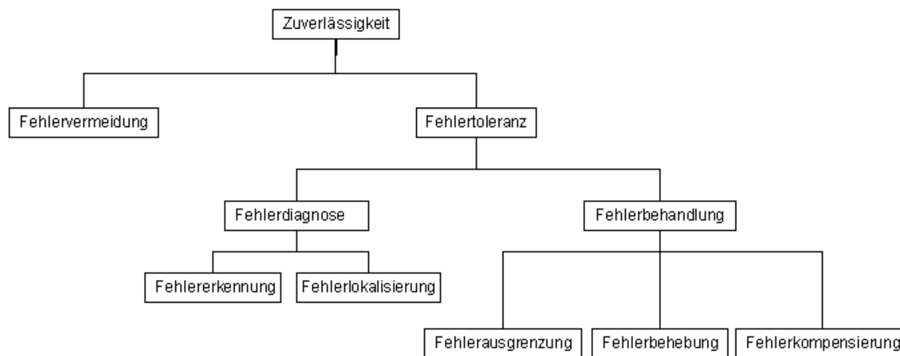


Abbildung 2.4: Unterteilung von Zuverlässigkeit und Fehlertoleranz nach [Ech90]

mit einem gewissen Zeit- und Kostenaufwand verbunden, so dass es auch nicht sinnvoll erscheint, diese Vorgehensweise bis in das kleinste Detail des Systems zu verfolgen. Hier ist es dann im Allgemeinen das günstigste, einen Kompromiss aus Fehlervermeidung und Fehlertoleranz zu finden, der das System bis zu einer gewissen, festgelegten Aufwandsschwelle perfektioniert. Wird diese Aufwandsschwelle erreicht und ist die geforderte System-Zuverlässigkeit noch nicht gewährleistet, so werden dann noch Verfahren zur Fehlertoleranz mit einbezogen. Da Fehlervermeidung nicht Bestandteil dieser Arbeit ist, soll hier nicht weiter auf diesen Begriff eingegangen werden.

2.4 Teilbereiche

Verfahren zur Gewährleistung von Fehlertoleranz lassen sich nach [Ech90] grob in zwei Teilbereiche bzw. Teilschritte aufteilen: zum einen in den Bereich der Fehlerdiagnose, und zum anderen in den Bereich der Fehlerbehandlung. Diese beiden Teilbereiche lassen sich feiner unterteilen, was in Abbildung 2.4 dargestellt wird. Ausgangspunkt ist dabei der bereits in Kapitel 2.1 angesprochene Begriff der Zuverlässigkeit.

2.4.1 Fehlerdiagnose

Die Diagnose eines an einer inneren Komponente des Sytems aufgetretenen Fehlers ist der erste Schritt, der im Rahmen der Fehlertoleranz durchzuführen ist. Erst nachdem der Fehler diagnostiziert worden ist, lässt sich entscheiden, wie (und ob überhaupt) dieser Fehler zu behandeln ist. Dabei besteht die Fehlerdiagnose aus zwei Schritten.

Erkennung

Die Fehlererkennung dient dazu, generell zu registrieren, dass ein Fehler aufgetreten ist. Das Vorgehen dabei ist, dass die Fehlererkennung ganz allgemein eine Aussage darüber liefert, ob alle Komponenten fehlerfrei arbeiten. Ist dieses nicht der Fall, so liegt offensichtlich ein Fehler vor. Allerdings lässt die Fehlererkennung insbesondere die Frage offen, in welcher (oder welchen) der Komponenten des Systems Fehler vorliegen.

Lokalisierung

Die Lokalisierung eines erkannten Fehlers ist der auf die Erkennung eines Fehlers folgende Schritt, in dem versucht wird, diesen erkannten Fehler innerhalb des Systems auf eine oder mehrere Komponenten einzugrenzen, d. h. den sog. *Fehlerort* zu bestimmen. Dazu gibt die Fehlerlokalisierung an, welche der

Komponenten des Systems fehlerfrei funktionieren. Komponenten, die von der Fehlerlokalisierung nicht als fehlerfrei eingestuft wurden, können fehlerbehaftet sein, *müssen* es aber nicht sein. Beispielweise ist der Fall denkbar, dass an einer Komponente lediglich nach außen hin nicht ordnungsgemäß arbeitet, weil eine weitere, mit dieser Komponente zusammenarbeitende Komponente fehlerhaft ist, und an dieser Komponente deswegen ein Folgefehler auftritt. Die Fehlerlokalisierung versucht, Fehlerorte „so gut wie möglich" zu benennen.

2.4.2 Fehlerbehandlung

Unter Fehlerbehandlung ist der Vorgang zu verstehen, der sich an die Diagnostizierung eines aufgetretenen Fehlers anschließt. Wie aus Abb. 2.4 ersichtlich gibt es drei verschiedene Arten von Maßnahmen, mit denen auf einen diagnostizierten Fehler reagiert werden kann. Meistens sind diese Maßnahmen miteinander zu kombinieren, damit der Fehler erfolgreich behandelt werden kann. Allerdings kann grundsätzlich auch die Möglichkeit bestehen, dass bereits die Anwendung einer einzelnen Maßnahme zum erfolgreichen Behandeln des Fehlers führt.

Fehlerausgrenzung

Im Rahmen der Fehlerausgrenzung werden Komponenten, an denen ein Fehler aufgetreten ist, aus dem System entfernt. Die bloße Entfernung dieser fehlerbehafteten Komponenten kann an dieser Stelle ausreichen, um den diagnostizierten Fehler zu behandeln (beispielsweise wäre dieses in dem in Abschnitt 2.2.2 vorgestellten n-von-m-System so möglich). Allerdings ist es auch denkbar, dass die zu entfernenden Komponenten Funktionalitäten bereitstellen, die für die Erfüllung des Systemzwecks elementar sind. Würden diese Komponenten dann lediglich entfernt, und ansonsten keine weiteren Maßnahmen durchgeführt, so würde das Gesamtsystem seinen Zweck nicht mehr erfüllen können.

Die Lösung dieses Problems liegt nun darin, bei der sog. *Ausgliederung* fehlerhafter Komponenten andere, für diesen Fall vorgesehene Komponenten *einzugliedern*. Diese müssen die von den ausgegliederten Komponenten bereitgestellte Funktionalität zur Verfügung stellen, so dass diese Funktionen auf diese Komponenten verlagert werden können. Bei diesem Vorgang handelt es sich um eine sog. *Rekonfiguration*, und bei den ersatzweise eingegliederten Komponenten spricht man von *redundanten* Komponenten (vgl. Abschnitt 2.2). Als Beispiel kann hier auch wieder das in 2.2.1 angeführte Netzwerk-System mit zwei redundanten SMTP-Servern dienen. Dabei entspricht die redundante Komponente dem sekundären SMTP-Server, und der Rekonfiguration entspricht das Umschalten von dem primären auf den sekundären Server.

Fehlerbehebung

Bei der Fehlerbehebung werden fehlerbehaftete Komponenten nicht, wie bei der Fehlerausgrenzung, aus dem System entfernt. Stattdessen wird versucht, die fehlerbehafteten Komponenten wieder in einen fehlerfreien Zustand zu überführen. Hierbei wird unterschieden zwischen der sog. *Vorwärtsbehebung* und der sog. *Rückwärtsbehebung*; das Unterscheidungskriterium ist dabei der Informationsbestand, auf dem die entsprechende Fehlerbehebung aufsetzt.

Bei der Rückwärtsbehebung werden zur Widerherstellung eines fehlerfreien Systemzustandes Informationen über Systemzustände in der Vergangenheit benutzt. Dazu werden im Verlauf der Systemaktivität immer wieder „Systemwiederherstellungspunkte" gespeichert, die eine Abbildung eines konsistenten, d. h. fehlerfreien, Zustandes darstellen. Tritt nun an einer Komponente ein Fehler auf, so kann diese Komponente mittels dieser gespeicherten Informationen wieder in einen für sie konsistenten Zustand zurückgesetzt werden. Unter Umständen reicht es allerdings nicht aus, nur die fehlerbehaftete Komponente wieder zurückzusetzen: wenn diese direkt mit anderen Komponenten (die nicht notwendigerweise auch fehlerbehaftet sind) interagiert, müssen auch diese Komponenten wieder auf einen gemeinsamen Zustand

zurückgesetzt werden. Die Gesamtmenge aller Komponenten in einem System, die durch Rückwärtsbehebung zurücksetzbar sind, wird dabei als *Rückwärtsbehebungsbereich* bezeichnet. Diese Notwendigkeit, nicht nur eine fehlerbehaftete Komponente, sondern eventuell auch mit dieser interagierende, fehlerfreie Komponenten auf einen gemeinsamen konsistenten Zustand zurückzusetzen, lässt erkennen, dass mit der Verwendung der Rückwärtsbehebung ein (je nach Größe und Komplexitiät des Systems u. U. sehr großer) Mehraufwand verbunden ist. Dieses Effekt wird dabei noch verstärkt durch den Aufwand der anfällt, um regelmäßig Rücksetzpunkte zu speichern.

Ein sehr einfaches Beispiel für eine Rückwärtsbehebung angewendet auf das hier mehrfach benutzte Beispiel eines Mailservers wäre das regelmäßige Erstellen einer Sicherheitskopie des Festplatten- und Speicher-Inhaltes. Im Falle eines Fehlers würde bei der Vorgehenweise der Rückwärtsbehebung ein für den Server konsistenter Zustand dadurch wieder hergestellt, dass die aktuellste, als konsistent klassifizierte Sicherheitskopie wieder in den Speicher bzw. auf die Festplatte eingespielt wird.

Im Gegensatz zur Rückwärtsbehebung benötigt die Vorwärtsbehebung keine gespeicherten Informationen über vergangene Zustände. Die Fehlerbehebung mittels Vorwärtsbehebung erfolgt rein auf Basis des aktuellen Zustandes des Systems und evtl. unter Verwendung von Redundanzen (vgl. 2.2) innerhalb des Systems. Beispielsweise kann eine Aufgabe, die auf der jetzt fehlerbehafteten Komponente nicht mehr zu Ende geführt wurde, auf einer anderen, redundanten Komponente noch einmal ausgeführt werden. Während diese redundante Komponente diese Aufgabe ausführt, kann die fehlerbehaftete Komponente wieder in einen fehlerfreien Zustand überführt werden, was beispielsweise mittels eines Neustarts dieser Komponente geschehen kann. Nach einem erfolgreichen Neustart wäre sie dann wieder für zukünftige Aufgaben einsetzbar. Dieses Vorgehen setzt allerdings eine zeitliche Redundanz (vgl. 2.2.1) innerhalb des Systems voraus, da sich die Fertigstellung dieser Aufgabe verzögert.

Fehlerkompensierung

Die Vorgehensweise der Fehlerkompensierung setzt nicht bei der Korrektur von fehlerbehafteten Zuständen von Komponenten an, sondern korrigiert lediglich die möglicherweise falschen Ergebnisse fehlerbehafteter Komponenten. Dazu werden zusätzliche Komponenten zur Kontrolle der Ergebnisse vorgesehen, die an den Schnittstellen zwischen potentiell fehlerbehafteten Komponenten und deren Nachfolgekomponenten, die mit den Ergebnissen dieser weiterarbeiten, eingesetzt werden. Bei diesen Kontroll-Komponenten kann man zwei verschiedene Vorgehensweisen unterscheiden: zum einen die sog. *Fehlerkorrektur*, bei der ein verfälschtes Ergebnis von der Kontroll-Komponente korrigiert wird, bevor es an die nachfolgende Komponente weitergegeben wird, und zum anderen die sog. *Fehlermaskierung*. Letztere setzt voraus, dass mehrere Komponenten die an dieser Stelle benötigte Aufgabenlösung bestimmen, und dass nur eine begrenzte Anzahl dieser Lösungen falsch ist. Dann ergibt sich die korrekte Lösung, die an die nachfogende Komponente weitergegeben werden muss, beispielsweise wie in dem in Abschnitt 2.2.2 eingeführten *n-von-m-System* als Mehrheitsentscheidung unter allen bestimmten Lösungen

Kapitel 3

Service-orientierte Architekturen

In diesem Kapitel wird dargestellt, was unter dem Konzept der *service-orientierten Architektur* (kurz: *SOA*) zu verstehen ist bzw. in welchem Zusammenhang dieser Begriff hier verwendet wird. Anschließend wird mit *Web Services* die Implementierung einer solchen Archtitektur vorgestellt, die zusammen mit einigen Erweiterungen das abschließend behandelte Konzept des *Devices Profile for Web Services (DPWS)* darstellt.

3.1 Grundlagen

Gemäß [Mel07] sind service-orientierte Architekturen das abstrakte Konzept einer Software-Architektur, die in erster Linie das Anbieten, Suchen und Nutzen von *Diensten* in einem Netzwerk gewährleisten soll. Diese Dienste werden von *Clients* genutzt, dieses können ganz allgemein Software-Applikationen oder andere Diensten sein. Dabei soll es unerheblich sein, welche möglicherweise verschiedenen Arten von Hardware, Programmiersprachen oder Betriebssystemen die einzelnen beteiligten Komponenten benutzen.

Ein zentrales Merkmal ist dabei die *lose Kupplung (loose coupling)* der Dienste. Dieses bedeutet, dass ein Dienst von einem Client, der diesen gerade benutzen will, erst im konkreten Bedarfsfall gesucht und dynamisch eingebunden wird. Hierbei spricht man von dem sogenannten *dynamischen Binden*. Um dieses zu ermöglichen, muss der gesuchte Dienst zunächst von dem suchenden Client gefunden werden. Dafür gibt es innerhalb der SOA ein sogennantes *Dienstverzeichnis* (auch *Repository* genannt), in dem sich die einzelnen Dienste registrieren müssen, und in dem ein Client folglich nach einem bestimmten Dienst suchen kann. Dabei basiert die geforderte plattformunabhängige Kommunikation und Interaktion von Komponenten einer SOA auf *offenen Standards*, wodurch auf einfache Weise neue Dienste in das System aufgenommen und verwendet werden können.

Aus den zuvor aufgestellten Anforderungen und Merkmalen ergibt sich nach [Mel07] folgende allgemeine Definition einer SOA:

> „Unter einer SOA versteht man eine Systemarchitektur, die vielfältige, verschiedene und eventuell inkompatible Methoden oder Applikationen als wiederverwendbare und offen zugreifbare Dienste repräsentiert und dadurch eine plattform- und sprachenunabhängige Nutzung und Wiederverwendung ermöglicht.“

Im folgenden sollen nun die wichtigeBegriffe im Zusammenhang mit einer SOA näher erläutert werden.

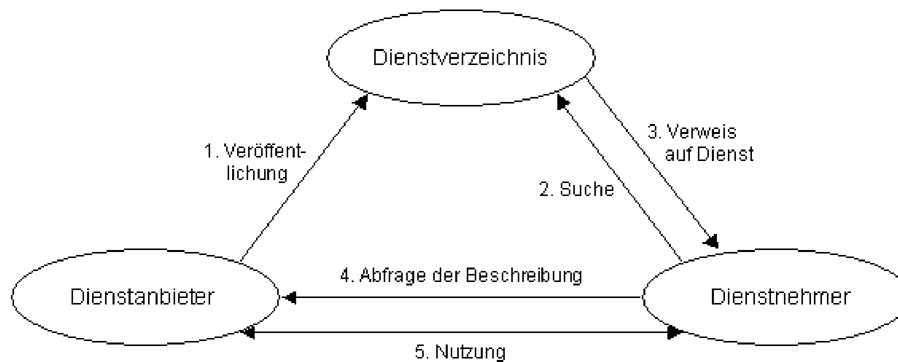


Abbildung 3.1: Rollen und Aktionen in einer SOA nach [Mel07]

3.1.1 Dienste und Dienstbeschreibungen

Ein *Dienst* (oder auch *Service*) in einer SOA ist, wie bereits beschrieben, eine Softwarekomponente, die über eine Schnittstelle eine gewisse Funktionalität bereitstellt. Zugriffe auf diesen Dienst sind nur über diese Schnittstelle zulässig, d. h. Details der Implementierung des Dienstes sind für die Nutzer des Dienstes nicht sichtbar. Damit potentielle Nutzer auf diese Schnittstelle zugreifen können ist es nötig, dass diese öffentlich beschrieben wird. Hierbei spricht man von der sogenannten *Service Description*. Diese sollte unabhängig von der Dienst-Implementierung, der benutzten Programmiersprache und der ausführenden Plattform sein. Die Inhalte einer solchen Beschreibung lassen sich unterteilen in *funktionale* und *nicht-funktionale* Inhalte. Erstere wären z. B. Informationen über Parameter des Dienstes (z. B. ein Ergebnis einer im Dienst berechneten Funktion), letztere z. B. Informationen über Antwortzeiten des Dienstes, dessen Verfügbarkeit, oder aber auch die Kosten für die Nutzung des Dienstes.

3.1.2 Rollen

Aus der zuvor erfolgten Beschreibung der Grundlagen einer SOA lassen sich bereits drei verschiedene *Rollen* innerhalb des Systems erkennen. Diese Rollen sind zusammen mit ihren ausgeführten Aktionen in Abbildung 3.1 dargestellt.

Dienstverzeichnis

Das Ziel bzw. die Aufgabe des *Dienstverzeichnis* innerhalb einer SOA liegt darin, an der Nutzung eines Dienstes interessierten Komponenten das Auffinden dieses Dienstes zu ermöglichen. Dazu müssen diese Dienste mit den entsprechenden Information im Dienstverzeichnis registriert sein. In einer einfach entworfenen SOA kann diese Registrierung beispielsweise direkt durch jeden Anbieter eines oder mehrerer Dienste geschehen, d. h. die Anbieter müssen ihre Dienste selbst aktiv registrieren. Alternativ kann es in einer etwas komplexer gestalteten SOA auch z. B. einen dedizierten Dienst geben, der wie eine „Suchmaschine“ arbeitet, d. h. selbstständig nach Diensten innerhalb des Netzwerkes sucht und diese in das Dienstverzeichnis einträgt.

Innerhalb einer SOA muss es nicht zwangsläufig nur ein Dienstverzeichnis geben. In der Regel ist es so, dass es innerhalb eines Systems mehrere verschiedene Dienstverzeichnisse gibt, um beispielweise die Verwaltung von Diensten für unterschiedliche Zielgruppen zu trennen. Als Beispiel kann man sich hier ein Firmennetzwerk vorstellen, in dem Dienste für interne (z. B. Verwaltungsfunktionen) und externe Funktionen (z. B. für Anfragen von Kunden) bereitgestellt werden. Hier würde es sich dann anbieten, dass ein Dienstvereichnis die internen, und ein weiteres Dienstverzeichnis die externen Dienste verwaltet.

Dienstanbieter

Bei einem *Dienstanbieter* handelt es sich um eine Komponente, die eine Plattform bereitstellt, mittels der andere Komponenten auf mindestens einen Dienst dieses Dienstanbieters zugreifen können.

Hier bedeutet „Bereitstellen einer Plattform“ nicht nur, dass der Anbieter einmalig die benötigte Infrastruktur des Dienstes bereitstellt, sondern auch zu gewährleisten, dass die bereitgestellten Dienste konsistent und verfügbar bleiben. Beispielsweise könnte man sich hier vorstellen, dass der Anbieter Datensicherungen durchführt, die bereitgestellten Dienste wartet, usw. Ferner muss der Dienstanbieter die Sicherheit der von ihm betriebenen Plattform gewährleisten. Dieses umfasst z. B. Aufgaben wie Authentifizierung (ist der Dienstaufrufer auch in der Tat der, der er zu sein vorgibt?) und Authorisierung (darf der Dienstaufrufer die von ihm angeforderte Funktionalität überhaupt nutzen?).

Damit ein von einem Dienstanbieter angebotener Dienst verfügbar ist, muss dieser zunächst *veröffentlicht* werden. Dieser Prozess beginnt bereits mit der Installation des entsprechenden Dienstes in der Umgebung, dem sogenannten *Deployment*. Nachdem diese Installation erfolgreich durch den Dienstanbieter abgeschlossen ist, muss dieser dann noch die eigentliche *Veröffentlichung* in Form der Registrierung des Dienstes im *Dienstverzeichnis* vornehmen (vgl. Abbildung 3.1).

Dienstnutzer

Ein *Dienstnutzer* ist daran interessiert, die Funktionalität eines bestimmten Dienstes innerhalb des Systems zu benutzen. Dabei ist es für einen Dienstnutzer oft unerheblich, welcher Dienstanbieter diesen Dienst bereitstellt.

Damit ein Dienst benutzt werden kann, muss dieser zunächst lokalisiert werden. Dieses geschieht, indem ein potentieller Dienstnutzers eine *Suche* nach dem gewünschten Dienst im entsprechenden Dienstverzeichnis durchführt. Ist in diesem ein Dienst registriert, der der Suchanfrage entspricht, so wird dem anfragenden Dienstnutzer mit einem *Verweis auf diesen Dienst* geantwortet. In diesem wird dem Dienstnutzer insbesondere mitgeteilt, wo der Dienst bzw. dessen Dienstschnittstelle lokalisiert ist. Anschließend findet dann die Kommunikation zum Aufruf des Dienstes direkt zwischen dem Dienstnutzer und dem Dienstanbieter statt, es beginnt die sogenannte *Dienst-Interaktion*. Von diesem Punkt an entspricht das Verhalten zwischen dem Dienstnutzer und dem Dienstanbieter grundsätzlich dem klassischen Client/-Server - Paradigma (vgl. [Com01]), wobei der Dienstnutzer dem Client entspricht. Durch diesen erfolgt zunächst eine *Abfrage der Beschreibung* des Dienstes, gefolgt von möglicherweise noch auszutauschenden Informationen über die Nutzung des Dienstes. Dabei kann es sich z. B. um die Authentifizierung und Authorisierung des Clients handeln. Akzeptiert der Dienstanbieter den Client, d. h. kann zwischen diesen beiden Seiten eine *Einigung* ausgehandelt werden, so erfolgt die *Bindung* des Dienstnutzers an den Dienst. Von diesem Punkt an ist es dem Client möglich, die Funktionen des Dienstes zu nutzen.

3.2 Web Services

Bei der Technologie der *Web Services* (WS) handelt es sich um einen Implementierungsansatz einer serviceorientierten Architektur. Dabei lässt sich ein konkreter Web Service beschreiben als eine eigenständige Software-Komponente, die mittels beliebiger Netzwerktechnologien mit anderen Software-Komponenten kommuniziert und iteragiert. Für diese Kommunikation stellt ein Web Service standardisierte Schnittstellen bereit. Die in einer SOA geforderte Beschreibung der Schnittstellen (vgl. 3.1.1) erfolgt bei Web Services mittels der *Web Services Description Language* (WSDL). Die Übertragung von Daten wie z. B. der Schnittstellenbeschreibungen oder der Nutzdaten erfolgt mittels *SOAP*. Als standardisiertes Datenformat wird *XML* gewählt, was der in den den Grundlagen für SOA formulierten Forderung nach offenen Standards für die Kommunikation der Komponenten Rechnung trägt. Die Funktion des Dienstverzeichnisses wird bei den Web Services mittels des *Universal Description, Discovery and Integration protocol*

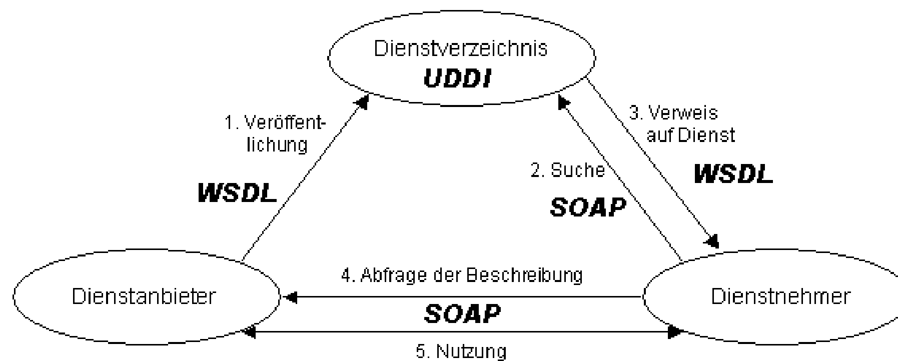


Abbildung 3.2: Web Services - System nach [Mel07]

(UDDI) bereitgestellt.

Eine entsprechende Definition der Web Services liefert das World Wide Web Consortium (vgl. [Wor04]):

> „A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.“

Neben dieser gibt es noch weitere Definitionen für Web Services. Im weiteren Verlauf wird allerdings diese Definition zugrunde gelegt, weswegen hier auf die weiteren nicht näher eingegangen wird. Die Struktur eines Web Service - Systems gemäß obiger Einführung und Definition ist dargestellt in Abbildung 3.2.

3.2.1 XML

Die *eXtensible Markup Language*, oder kurz XML, ist das Basisformat, für alle Daten, die bei Web Services ausgetauscht werden. Dieses sind z. B. sowohl die eigentlichen Nutzdaten, die zwischen den einzelnen Komponenten ausgetauscht werden, aber auch z. B. die Schnittstellenbeschreibungen der WSDL. Neben dem Kernkonzept von XML gemäß [Wora] sind die Spezifikationen *XML Infoset* (vgl. [Wor01]), *XML Schema* und *XML Namespaces* wichtige Komponenten, die im Konzept der Web Services benutzt werden.

3.2.2 SOAP

SOAP (vgl. [Worb]) dient als Übertragungsprotokoll für Nachrichten, die zwischen den Kommunikationspartnern ausgetauscht werden, z. B. einem Web Service und einem Client. Ursprünglich stand die Abkürzung SOAP für *Simple Object Access Protocol*. Aufgrund der zunehmenden Komplexität des Protokolls und der Erweiterung um die Möglichkeit, auf mehr als nur Objekte zugreifen zu können, ist dieser vollständig ausgeschriebene Name seit der Version 1.2 jedoch nicht mehr gebräuchlich. Stattdessen wird nur noch die Bezeichnung SOAP benutzt.

SOAP basiert ebenfalls auf XML. Das XML-Wurzelelement ist der sogenannte *Envelope*. In diesem wird mittels eines Namespace-Attributes festgelegt, welche Version der SOAP-Spezifikation für diese Nachricht verwendet wird. Optional kann das erste folgende Kind des Wurzelelementes der *SOAP-Header* sein, in dem z. B. sicherheitsrelevante Informationen wie eine Absender-Identifikation enthalten sein können. Das nächste Kind des Wurzelelementes ist der obligatorische *SOAP-Body*. Dieser dient zum Transport der eigentlichen Nutzdaten (payload) der Nachricht, z. B. also bei einer Anfrage an einen Web Service die

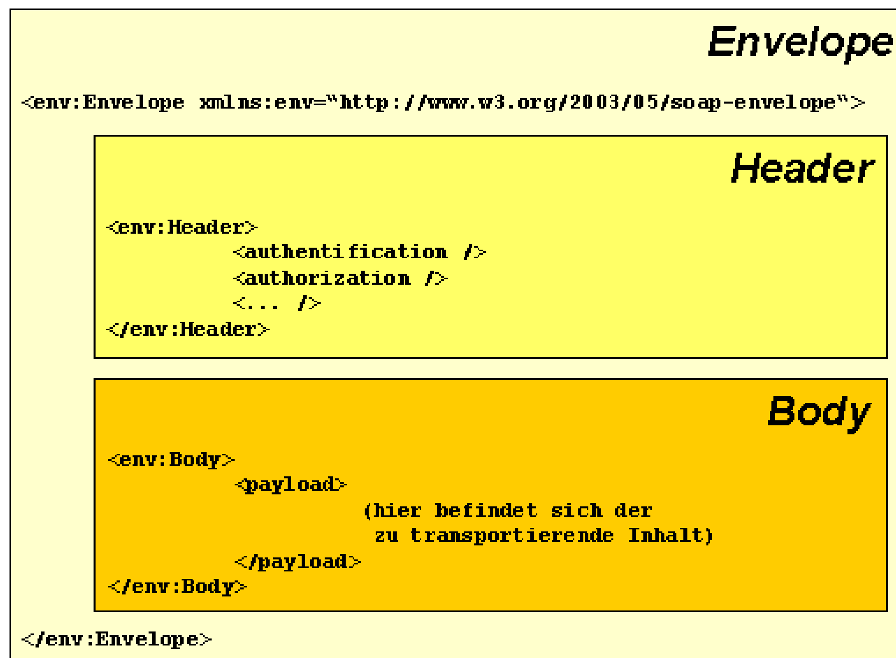


Abbildung 3.3: Beispiel eines SOAP-Envelopes mit Header und Body (nach [Mel07])

Eingabewerte für diesen Dienst. Die Struktur eines solchen SOAP-Envelopes mit Header und Body ist in Abbildung 3.3 dargestellt.

Grundsätzlich ist in keiner der SOAP-Spezifikationen vorgeschrieben, wie, d. h. mit welchen darunterliegenden Protokoll die SOAP-Nachrichten auszutauschen sind. Es wird lediglich (vgl. [Worb], Kapitel 4) mittels des *SOAP Protocol Binding Framework* beschrieben, wie eine SOAP-Nachricht mittels eines Protokolls übertragen werden muss. Ein Beispiel für ein solches Binding findet sich in [Worc], Kapitel 7. In diesem wird HTTP benutzt, was sich wegen der weiten Verbreitung und allgemeinen Akzeptanz dieses Protokolls auch für die generelle Benutzung anbietet. Aber auch z. B. SMTP oder FTP sind möglich.

3.2.3 WSDL

Die *Web Service Description Language* (WSDL) dient der Beschreibung der Schnittstellen von Web Services, und beinhaltet somit Methodennamen, Parameter- und Typinformationen, sowie die benutzten Transportprotokolle. Das WSDL-Format ist ebenfalls XML-basiert.

Eine WSDL-Beschreibung, die von einem `<description>`-Element umschlossen ist (bei älteren Versionen als 2.0 einem `<definitions>`-Element), ist unterteilt in einen *abstrakten* und in einen *konkreten* Teil. Im abstrakten Teil werden werden zunächst allgemein und insbesondere protokoll-unabhängig die Operationen des Web Service und die von diesen Operationen verwendeten Datentypen beschrieben. Dazu werden zunächst mittels `<types>`-Elementen die Datentypen mit XML-Schema definiert, die in den darauffolgenden `<message>`-Elementen verwendet werden können. Anschließend können diese `<message>`-Elemente wiederum als `<input>`- oder `<output>`-Nachrichten den entsprechenden Operationen des Web Service zugeordnet werden, wobei eine Operation mit sowohl Eingabe- wie auch Rückgabe-Wert dann entsprechend ein `<input>`- und ein `<output>`-Element besitzt. Die Kombination dieser sogenannten `<operation>`-Elemente ergibt dann die eigentliche Schnittstellenbeschreibung des Web Service. Hier ist es wichtig zu beachten, dass das XML-Element, unter dem die `<operation>`-Elemente in der WSDL angeordnet sind (d. h. das Element, dass die eigentliche Schnittstelle definiert), bei dem Übergang von WSDL 1.1 nach

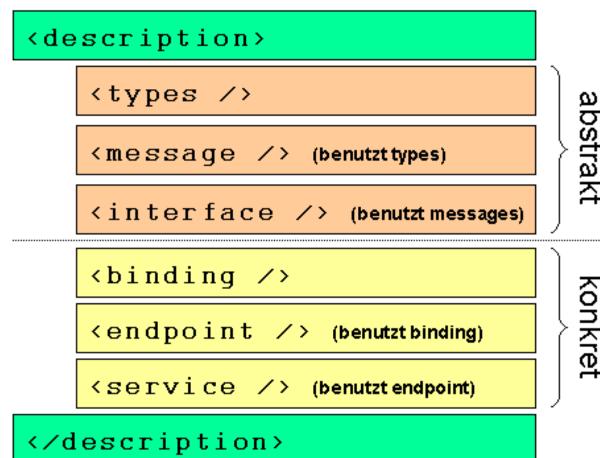


Abbildung 3.4: Überblick über die WSDL-Komponenten

WSDL 2.0 umbenannt wurde: in der Version 1.1 ist der Bezeichner dieses Elements `<portType>`, während er in der Version 2.0 `<interface>` lautet.

Der konkrete Teil der WSDL-Beschreibung legt explizit fest, über welche Protokolle der Web Service anzusprechen ist (ausdrücklich können dieses auch mehrere Protokolle wie z. B. HTTP und HTTPS sein), und unter welchem *Uniform Resource Identifier* (URI) der Web Service zu finden ist. Dazu gibt es zunächst das `<binding>`-Element, mit welchem das benutzte Transportprotokoll (z. B. HTTP) festgelegt wird (bei mehreren zugelassenen Transportprotokollen existiert für jedes Protokoll ein solches `<binding>`-Element). Anschließend beschreibt dann das `<service>`-Element, welche der Operationen mit welchem der zuvor definierten Bindings und über welchen URI zu erreichen ist. Dazu kombiniert ein `<endpoint>`-Element (bzw. bei älteren WSDL-Versionen als 2.0 ein `<port>`-Element) ein Binding-Element mit einer konkreten Adresse in Form eines URI.

Ein optionales WSDL-Element, das hier der Vollständigkeit halber kurz angesprochen werden soll, ist das `<documentation>`-Element. Mit diesem lassen sich zusätzlich textuelle Beschreibungen zu anderen WSDL-Elementen hinzufügen. Eine schematischer Überblick über die WSDL-Komponenten findet sich in Abbildung 3.4. Das optionale `<documentation>`-Element findet dort keine Anwendung; es wäre optional zu jeder der Komponenten hinzufügbar.

3.2.4 UDDI

Das *Universal Description, Discovery and Integration Protocol* (UDDI) stellt im Konzept der Web Services die Funktionalität des in 3.1 angesprochenen *Dienstverzeichnisses* innerhalb der SOA bereit. Anbieter von Web Services tragen die Daten ihrer Dienste in dieses Verzeichnis ein, und Clients (d. h. die potentiellen Nutzer eines Dienstes) fragen diese Daten ab. Bei diesen hinterlegten Daten handelt es sich nicht unbedingt nur um abgelegte WSDL-Informationen zur Beschreibung der einzelnen Service-Schnittstellen.

Neben diesen können innerhalb einer *UDDI Registry* von den Dienstanbietern auch noch weitere Arten von Informationen angelegt werden. Gemäß [Lan03] lassen sich hier drei verschiedene Informationstypen unterscheiden bzw. lässt sich die UDDI Registry dementsprechend in drei Teile unterteilen:

- *White Pages* mit sogenannten Business Informationen, d. h. Informationen über das Unternehmen, das den entsprechenden Web Service zur Verfügung stellt. Dieses können z. B. Adress- oder Kontaktdaten des Unternehmens sein,
- *Yellow Pages* mit sogenannten Service Informationen, d. h. Informationen über die bereitgestellten Dienste. Hier lässt sich eine Kategorisierung der Dienste vornehmen, so dass ein Dienstnutzer, der

nur weiss, was für eine Art von Dienst er nutzen will, mit Hilfe des Suchens in der entsprechenden Kategorie einen Anbieter eines solchen Dienstes finden kann,

- *Green Pages* mit Technik-Informationen, d. h. den technischen Spezifikationen der einzelnen Web Services. In diesem Verzeichnis kann z. B. gesucht werden, wenn weder der Anbieter noch die Kategorie des benötigten Dienstes bekannt ist, aber bekannt ist, welche Funktion der gesuchte Dienst genau erfüllen soll. Dabei ist die hier angesprochene WSDL eine Ausprägung einer derartigen technischen Spezifikation. Es wären aber auch andere Arten von Spezifikationen möglich.

In [Mel07] werden die *Green Pages* etwas anders definiert. Hier dienen diese eher zum allgemeinen Beschreiben der Funktionen von Diensten in für den Menschen lesbarer Form, während es für die eigentliche technische Beschreibung noch als vierten Teil die sogenannte *Service Type Registration* gibt, in der die Informationen in maschinenlesbarer Form abgelegt sind. Diese beiden Teile verweisen dabei ausdrücklich aufeinander.

3.2.5 Weitere Kernkomponenten

Neben den bisher beschriebenen, direkt auf das Konzept einer SOA übertragbaren Web Service - Kernkomponenten gibt es noch einige weitere wichtige Komponenten, die der Web Service - Architektur zugrunde liegen. Diese werden hier nun lediglich kurz erläutert. Für weitergehende Details sei jeweils auf die entsprechende Spezifikation verwiesen.

WS-Addressing

Mittels WS-Addressing werden die Daten, die zur Adressierung von auszutauschenden Nachrichten notwendig sind, konzentriert in den Header-Teil eines SOAP-Envelops (vgl. 3.2.2) geschrieben. Dieses ermöglicht die transportneutrale Adressierung, d. h. die protokollunabhängige Defintion von Service-Endpunkten. Weitere Details bezüglich WS-Addressing finden sich in [Wore].

WS-Policy

WS-Policy wird eingesetzt, um Verfahrensweisen und Richtlinien festzulegen, die bei der Interaktion zweier Web Services Anwendung finden sollen. Dabei bietet WS-Policy eine Syntax und Grammatik zur Formulierung dieser Verfahrensweisen und Richtlinien, und ist zugleich auch für die Interpretation und Durchsetzung dieser Policies verantwortlich. Typische Beispiele, bei denen WS-Policy Anwendung findet, sind die Wahl von Transportprotokollen oder Verfahren zur Authentifikation (vgl. [Mel07]). Für weitere Details sei auch hier wieder auf die Spezifikation von WS-Policy ([Worf]) verwiesen.

WS-MetadataExchange / WS-Transfer

Mittels WS-MetadataExchange bzw. gemäß der neuesten Spezifikation unter Verwendung von WS-Transfer/GET (vgl. [ABe06]) können dynamisch Metadaten eines Web Services (wie z. B. die Beschreibung des Services, die Policies des Services, ...) abgefragt werden. Die z. Zt. aktuelle Spezifikation von WS-MetadataExchange findet sich unter [BBe06].

WS-Security

Mittels WS-Security werden Methoden und Protokolle definiert, die die Grundlage darstellen, um die Sicherheit (z. B. bezüglich der Integrität, Vertraulichkeit oder Echtheit) beim Austausch von Nachrichten zu gewährleisten. Dabei wird die Integrität von Nachrichten z. B. mittels *XML Signature* (vgl. [ERe02]) sichergestellt, und die Vertraulichkeit mittels *XML Encryption* (vgl. [IDS02]). Die Standardisierungen von WS-Security und weitere Details sind zu finden unter [Org].

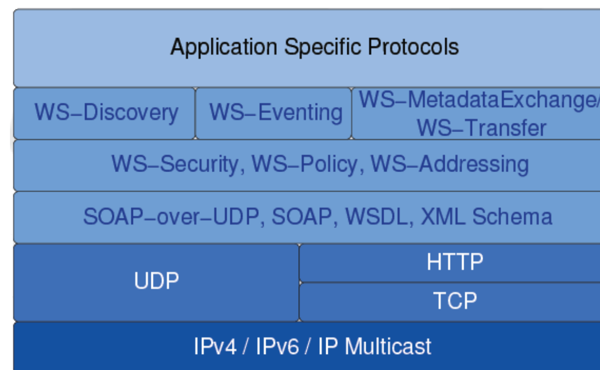


Abbildung 3.5: DPWS-Stack nach [ZBB+07]

3.3 Devices Profile for Web Services

Das *Devices Profile for Web Services* (DPWS) ist ein ursprünglich von Microsoft vorgeschlagener Standard für den Einsatz von Web Services auf Hardware mit eingeschränkten Ressourcen. Mittels DPWS werden Richtlinien zum Einsatz von Web Service - Funktionen, wie dem Austausch von Nachrichten oder dem Auffinden von Diensten, festgelegt (vgl. ([Mic06])). Es setzt auf dem zuvor beschriebenen Konzept der Web Services bzw. einem Teil der WS-Spezifikationen auf und erweitert diese um die nachfolgend beschriebenen Komponenten *WS-Eventing* und *WS-Discovery* (vgl. Abbildung 3.5). Mittels dieser vorgenommenen Erweiterungen soll es explizit ermöglicht werden, dass netzwerkfähige eingebettete Systeme leicht miteinander kommunizieren und interagieren können. Diese Kommunikation zwischen Client, Device und bereitgestellten Diensten (Services) ist dargestellt in Abbildung 3.6.

Hier sei aber ebenfalls noch einmal ausdrücklich darauf hingewiesen, dass auch in der DPWS-Spezifikation (ebenso, wie in der zugrundeliegenden Web Service - Technologie) lediglich der Rahmen für die Kommunikation und Interaktion der einzelnen Komponenten vorgegeben wird. Welche Nachrichten die Komponenten untereinander zur eigentlichen Interaktion austauschen ist der jeweiligen Implementierung überlassen.

3.3.1 WS-Eventing

Mittels *WS-Eventing* wird ein Protokoll beschrieben, das es Web Services erlaubt, auf von anderen Web Services ausgelöste Events zu lauschen. In diesem Zusammenhang wird von *Event Sinks* und *Event Sources* gesprochen. Ein an Events von einem anderen Web Service interessierter Service nimmt dabei die Rolle einer Event Sink an, welche sich mittels einer sogenannten *Subscription* bei der Event Source, d. h. dem das Event auslösenden Web Service, registriert. Die zeitliche Dauer dieser Subscriptions ist dabei nicht durch WS-Eventing vorgegeben. Gegebenenfalls muss eine Subscription regelmäßig erneuert werden.

3.3.2 WS-Discovery

WS-Discovery dient der Suche und Lokalisierung von im jeweiligen Subnetz verfügbaren Web Services. Ausgelöst wird eine solche Suche von einem an der Nutzung eines Services interessierten Client. Dazu sendet dieser per Multicast eine sogenannte *Probe Message* in sein Netzwerk, welche von eventuell vorhandenen passenden Services beantwortet wird. Diese Antwort erfolgt dann per Unicast direkt an den suchenden Client. Eine Suchanfrage kann dabei sowohl allgemein nach einem Service-Typ wie auch konkret nach einer bestimmten Service-Instanz suchen. Ferner sendet gemäß WS-Discovery ein Web Service, der neu einem Subnetz beitritt, eine sogenannte *Announcement Message* (in Form eines *„Hello“*) mit seinen Metadaten an die Multicast-Adresse dieses Subnetzes, um auf dieser Multicast-Adresse lauschende

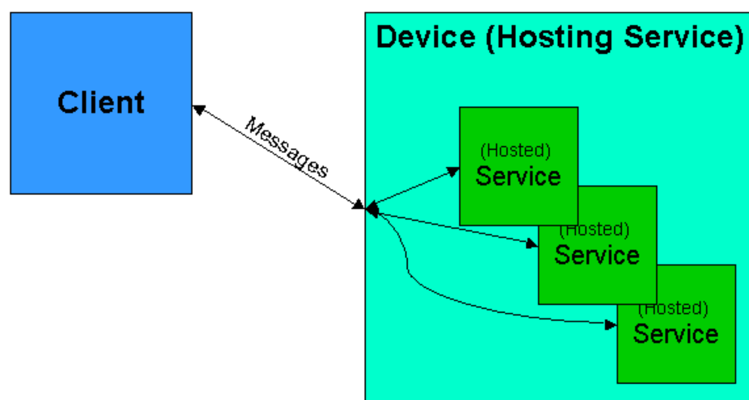


Abbildung 3.6: Nachrichtenaustausch in DPWS nach [Mic06]

Clients über seine Anwesenheit zu informieren. Verlässt ein Service ein Subnetz, meldet er sich äquivalent dazu mittels eines „*Bye*“ ab.

Kapitel 4

Simulation

Das Ziel dieser Diplomarbeit liegt in dem Entwurf und der Implementierung eines Simulationssystemes, mit dem die in Kapitel 2 vorgestellten Ansätze von Fehlertoleranzverfahren im Bereich von serviceorientierten Architekturen (vgl. Kapitel 3) simuliert werden können. In diesem Kapitel soll nun näher auf den Begriff der *Simulation* eingegangen werden. Dazu wird dieser Begriff zunächst definiert, und es werden Gründe für die Durchführung von Simulationen erläutert. Im Rahmen dessen werden bereits Begriffe wie z. B. *System* oder *Modell* benutzt, die in den darauffolgenden Teilen noch näher erläutert werden. Anschließend werden verschiedene Vorgehenweisen bei Simulations-Verfahren vorgestellt. Am Ende des Kapitels erfolgt schließlich aufbauend auf dem zuvor Beschriebenen eine Zusammenfassung mit einem Überblick über das Vorgehen bei der Erstellung von Simulationen.

4.1 Grundlagen

Ganz allgemein wird von einer *Simulation* gesprochen, wenn bei der Analyse eines *Systems* ein *Modell* an die Stelle des Originalsystems tritt, und an diesem Modell Experimente durchgeführt werden. Ist dabei die Abbildung des Systems in dem Modell hinreichend korrekt, so lassen sich in diesem die dynamischen Abläufe innerhalb des Systems nachvollziehen. Sammelt man nun die Erkenntnisse, die das Beobachten des Modellverhaltens bei der Simulation liefert, und wertet diese aus, so lassen sich damit (in gewissen Grenzen) Rückschlüsse auf das Verhalten des Originalsystems ziehen (vgl. [Krü74]).

Eine formale Defintion für die zuvor erfolgte allgemeine Beschreibung des Begriffes der Simulation findet sich in [Möl92], S. 119:

> „Wir definieren die Simulation als die Reproduktion des statischen und/oder dynamischen Verhaltens eines realen Systems, basierend auf einem materiellen oder immateriellen Abbild der Realität, dem Modell, welches diejenigen Aspekte des realen Systems beschreibt, die für den angestrebten Erkenntnisgewinn von Bedeutung sind, um aus den Simulationsergebnissen auf die Eigenschaften des realen Systems rückschließen zu können.“

Ergänzend zu dieser Definition sei hier noch hinzugefügt, dass im weiteren Verlauf der Begriff Simulation immer im Sinne der Reproduktion eines realen System auf einer *Rechenanlage*, d. h. auf einem Computer, benutzt wird (vgl. dazu auch die Definition in [Lek93], S. 648). Andere Darstellungsformen wie z. B. hydraulische, elektrische oder mechanische Analogien werden hier nicht berücksichtigt.

Die Definitionen und die Erklärung zuvor haben gezeigt, was unter dem Begriff der Simulation zu verstehen ist. Was allerdings nun noch nicht genannt wurde sind die *Gründe*, warum es überhaupt sinnvoll sein kann, bestimmte Vorgänge oder Systeme zu simulieren, anstatt die Versuche am entsprechenden

Originalsystem durchzuführen. Einige der wichtigsten Gründe werden hier nun noch kurz aufgeführt und erläutert:

- *Zeit*: der zeitliche Ablauf eines simulierten Vorgangs kann (erheblich) beschleunigt werden, was im Vergleich mit dem einfachen Beobachten des realen Vorganges dazu führt, dass das Ergebnis sehr viel schneller vorliegt. Alternativ kann auch die umgekehrte Vorgehensweise angewandt werden: ein Vorgang, der in einem realen System sehr schnell abläuft, kann mit Hilfe einer geeigneten Simulation ausgedehnt werden, so dass ein genaueres Beobachten und Analysieren der einzelnen Schritte möglich wird.
- *Kosten*: im allgemeinen sind die Kosten, die für das Erstellen und Simulieren eines Modells mittels Rechnerunterstützung anfallen, deutlich geringer als die Kosten, die anfallen würden, wenn man ein reales System ähnlich umfassend beobachten und analysieren würde. Ebenso fallen z. B. im Zerstörungsfall des simulierten Modells natürlich keine Kosten an.
- *Risiken*: nicht zuletzt bestehen bei einer rechnergestützten Simulation – verglichen mit dem „Ausprobieren“ am realen System – deutlich weniger bzw. gar keine Risiken gleich welcher Art. Diese Risiken können z. B. wieder wie zuvor angesprochen finanzieller Natur sein, wenn z. B. eine Dynamik getestet werden soll, die eventuell bis hin zu einer Systemzerstörung führt, aber z. B. auch durchaus wirkliche physische Risiken für das Versuchsumfeld. Ein Extrembeispiel hierfür wären Versuche bis an und über die Belastungsgrenze in z. B. einem Kernreaktor.

Weitere Gründe für die Durchführung von Simulationen sind z. B. in [Krü74] oder [Bos92] nachzulesen.

4.2 Systeme

Hier soll nun zunächst festgelegt werden, was im folgenden mit dem Begriff des Systems (oder auch mit dem Begriff des *Real-Systems*) bezeichnet wird. Nach [Bos92] gilt ganz allgemein ein Objekt als System, wenn es die folgenden Merkmale aufweist:

- Es wird eine bestimmte Funktion, der sogenannte *Systemzweck* erfüllt. Diese Funktion ist für einen Beobachter erkennbar.
- Innerhalb des Systems existiert eine bestimmte Konstellation von *Systemelementen*, die untereinander *wirkungsverknüpft* sind, d. h. in *Relationen* zueinander stehen. Dabei ist die *Komplexität* des Systems bestimmt durch die Anzahl der Elemente und Anzahl der Relationen dieser Elemente untereinander. Bei letzterem spricht man auch von dem sogenannten *Verflechtungsgrad* der Elemente.
- Das System ist nicht teilbar, ohne seine *Systemidentität* zu verlieren. Dieses bedeutet, dass innerhalb des Systems Elemente und Relationen existieren, deren Herauslösung oder Zerstörung es dem System unmöglich machen würde, seine vorher festgelegte Systemfunktion weiterhin zu erfüllen. Mit dem Herauslösen oder Zerstören von Elementen würde sich also die Systemidentität verändern bzw. diese würde zerstört werden.

Ein Element ist dabei eine nicht weiter unterteilbare oder aus Systemsicht als atomar angenommene Komponente des Systems (vgl. [Nie77]). Diese Elemente wiederum lassen sich als Systeme einer niedrigeren Ebene betrachten bzw. umgekehrt kann ein System ebenfalls wieder ein Element eines übergeordneten Systems (in dem es dann wiederum als unteilbar angenommen wird) darstellen. Mit dieser Herangehensweise ergibt sich die sogenannte *Hierarchie der Systeme* (vgl. [Krü74]). Systemelementen können ferner Eigenschaften, sogenannte *Attribute*, anhaften.

Ein einfaches Beispiel (vgl. [Bos92]) für ein System gemäß obigen Merkmalen wäre dementsprechend bereits ein Stuhl. Dieser erfüllt der einen klaren Systemzweck, besteht aus einer bestimmten Konstellation von Elementen (Beine, Lehne, . . .) besteht, und das Entfernen einzelner Elemente (z. B. eines Beines) führt zu einer Zerstörung der Systemintegrität, wonach das System seinen Systemzweck nicht mehr erfüllen könnte.

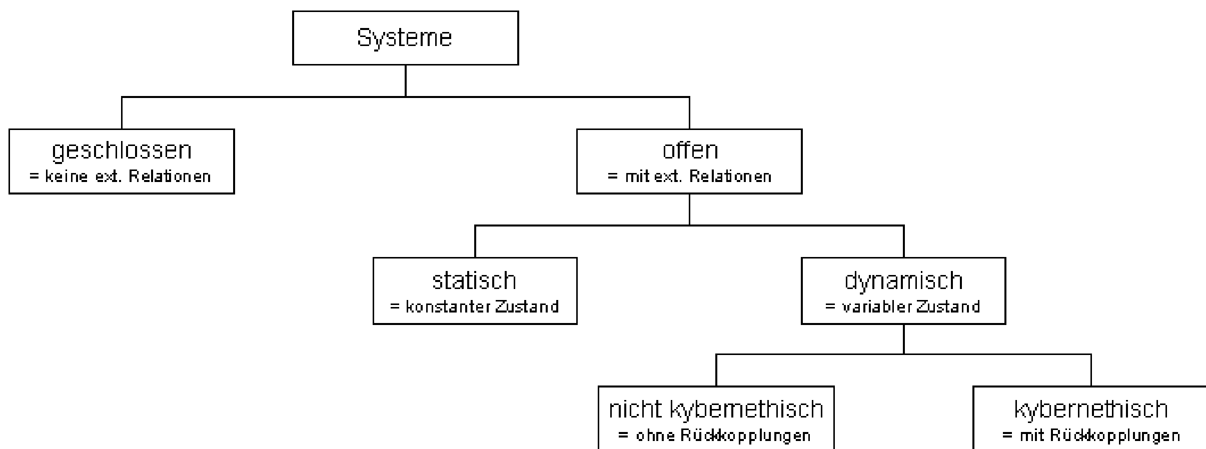


Abbildung 4.1: Ausprägungen von Systemen nach [Pag91]

4.2.1 Systemverhalten

Wie zuvor beschrieben setzt sich ein System zusammen aus seinen einzelnen Elementen, denen Attribute anhaften. Diese Attribute nehmen verschiedene Werte an, die den Zustand der einzelnen Elemente charakterisieren. Sind diese Werte veränderlich, so werden diese Attribute als *Zustandvariablen* oder auch *Zustandsgrößen* bezeichnet. Der *Systemzustand* des Gesamtsystems ist die Menge aller dieser Zustandsvariablen mit ihren Werten. Verändern sich nun einzelne dieser Zustandsvariablen im Verlauf der Zeit, so ergibt eine *Zustandsfolge* für das System, welche das *Systemverhalten* darstellt.

4.2.2 System-Ausprägungen

Systeme lassen sich gemäß [Pag91] anhand ihrer Ausprägungen in verschiedene Kategorien einteilen. Dieses ist dargestellt in 4.1.

Ein erstes Kriterium ist die Art der Beziehung des Systems zu seiner Umgebung: interagiert ein System mittels eines Systemein- oder -ausganges mit seiner Umgebung, so spricht man von einem *offenen System*. Dabei existiert dann eine sogenannte *Interaktionsbeziehung*. Findet hingegen keine Interaktion des Systems mit seiner Umgebung statt, so handelt es sich um ein *geschlossenes System*. Das Verhalten eines geschlossenen Systems ist unabhängig von äusseren Einflüssen, während ein offenes System sehr wohl von außen beeinflusst werden kann. Hierbei kann einschränkend erwähnt werden, dass real existierende System praktisch immer offene Systeme sind, da praktisch immer in irgendeiner Weise ein äusserer Einfluss erfolgen kann. Um sie trotzdem als geschlossene Systeme betrachten zu können, müssen gewisse Vereinfachungen vorgenommen werden.

Ein weiteres Kriterium zur Unterscheidung von Systemen ist das Verhalten der Systeme in der Zeit. Dabei sind *statische Systeme* solche, in denen sich der Systemzustand im Zeitablauf nicht verändert. Dementsprechend spricht man von einem *dynamischen System*, wenn derartige Veränderungen des Systemzustandes eintreten. Schließlich lassen sich die dynamischen Systeme noch weiter unterteilen in sogenannten *kybernetische* und *nicht-kybernetische* Systeme. Dabei ist ein kybernetisches System dadurch gekennzeichnet, dass in dem Graph der Relationen der Elemente des Systems Zyklen (sogenannte *Rückkopplungsschleifen*) vorhanden sind. In einem nicht-kybernetischen System existieren derartige Rückkopplungen nicht.

4.3 Modelle

Ein *Modell* (auch bezeichnet als *Simulations-Modell*) dient dazu, ein Real-System derart abzubilden, dass es mit Hilfe dieses Modells möglich ist, das System zu imitieren. Gemäß der Beschreibung des System-Begriffes ist auch ein Modell wieder ein System, das aber die Elemente und Relationen des zugrundeliegenden Real-Systems in (möglicherweise stark) veränderter Weise darstellt. Ganz ausdrücklich handelt es sich dabei dann nicht mehr um ein Real-System.

Eine formale Definition des Modell-Begriffes liefert [Nie77]:

> „Modelle sind materielle oder immaterielle Strukturen, die andere Systeme so darstellen, daß eine experimentelle Manipulation der abgebildeten Strukturen und Zustände möglich ist."

Der Übergang vom realen System hin zum Modell wird dabei als sogenannte *Modellbildung* bezeichnet. Im Rahmen dieser Modellbildung findet eine Abstraktion vom zugrundeliegenden Real-System und eine Idealisierung statt, d. h. dass das Modell das Real-System also nur in vereinfachter Weise darstellt. Dieses erst macht die Untersuchung sehr komplexer Real-Systeme mittels einer Simulation überhaupt erst möglich. Da die mit der Simulation des Modells erzielten Ergebnisse allerdings wieder auf das Real-System übertragen werden sollen gilt es hier zu beachten, dass die Modellbildung ausreichend genau geschieht. Wird im Rahmen dieses Prozesses zu sehr abstrahiert bzw. vereinfacht, so lassen sich die Simulations-Ergebnisse u. U. nicht mehr hinreichend präzise auf das Real-System übertragen. Wird andersherum nicht genügend abstrahiert, so bleibt auch das Modell sehr komplex, was das effiziente Simulieren dieses Modells erschweren oder sogar praktisch unmöglich machen kann.

4.3.1 Klassifikationen

Modelle für Real-Systeme lassen sich nach verschiedenen Kriterien klassifizieren. Bei diesen Kriterien handelt es sich nach [Pag91] um die Art der *Untersuchungsmethode*, des *Abbildungsmediums*, der *Zustandsübergänge*, und des *Verwendungszweckes*.

Klassifizierung nach der Untersuchungsmethode

Wird ein Modell untersucht, so können Erkenntnisse aufgrund von analytischen Berechnungen oder aufgrund von Simulation gewonnen werden. Dementsprechend unterscheidet man hier die Modelle in *analytische Modelle* und *Simulationsmodelle*. Erstere erlauben es, mittels einem die Systembeziehungen darstellenden Gleichungssystems die Systemzustände direkt zu bestimmen bzw. zu berechnen, bei letzteren wird der Zustand des Modells Schritt für Schritt entwickelt, ausgehend von einem Startzustand.

Klassifizierung nach dem Abbildungsmedium

Modelle müssen nicht zwangsläufig formal dargestellt werden. Es sind auch andere Darstellungs- bzw. Abbildungsformen möglich. Hier wird auf einer ersten Stufe unterschieden zwischen *materiellen* und *immateriellen* Modellen. Ein materielles Modell ist dabei ein wirklicher „Nachbau" des Real-Systems wie z. B. ein maßstabsgetreuer Nachbau eines Flugzeuges für Simulationen in einem Windkanal. Immaterielle Modelle lassen sich noch weiter unterteilen in *informale* Modelle, wie eine verbale Beschreibung oder ein grafisch-deskriptives Modell, und in *formale* Modelle, wie z. B. ein mathematisches Gleichungssystem.

Klassifizierung nach Art der Zustandübergänge

Hier erfolgt die Klassifizierung der Modelle nach der Art der Zustandveränderungen im Modell: auf einer ersten Stufe wird zwischen *statischen* und *dynamischen* Modellen unterschieden. Bei einem statischen Mo-

dell treten im Verlauf der Zeit keinerlei Veränderungen des Modellzustandes auf, bei einem dynamischen Modell hingegen ist der Zustand abhängig von der verstrichenen Zeit. Dynamische Modelle wiederum lassen sich weiter einteilen in *kontinuierliche* Modelle, die sich durch stetige Funktionen beschreiben lassen, d. h. die zu jedem beliebigen Zeitpunkt definiert sind, und in *diskrete* Modelle, bei denen sich der Zustand nur sprunghaft eben an diskreten Zeitpunkten bestimmen lässt. Diese beiden Modellformen lassen sich jeweils weiter unterteilen in *deterministische* Modelle, bei denen der gleiche Startzustand mit der gleichen Eingabe immer zu demselben eindeutigen Ergebnis führt, und in *stochastische* Modelle, bei denen Wahrscheinlichkeitswerte die Zustandübergänge beeinflussen.

Klassifizierung nach dem Verwendungszweck

Aufgrund der unterschiedlichen Anforderungen, die bei verschiedenen *Verwendungszwecken* an Modelle gestellt werden, kann eine Modell-Klassifizierung auch nach diesem Verwendungszweck erfolgen. Dabei wird nach [Pag91] unterschieden zwischen vier deratigen Zwecken. Es existieren *Erklärungsmodelle*, die zur Erklärungs des beobachteten Systemverhaltens herangezogen werden sollen und deswegen die Real-System-Struktur sehr detailliert abbilden müssen. Daneben gibt es *Prognosemodelle*, mit denen künftige Entwicklungen unter verschiedenen Voraussetzungen simuliert werden sollen, und *Gestaltungsmodelle*, die bei der Planung bzw. dem Entwurf von Projekten helfen sollen. Als vierte Modellart existieren *Optimierungsmodelle*, die zur Ermittlung und Herbeiführung möglichst optimaler Systemzustände dienen.

4.3.2 Modell-Sichtweisen

Ist das Real-System identifiziert und sind die Anforderungen aufgestellt, die an die Simulation des zu erstellenden Modells gestellt werden, so muss entschieden werden, in welcher Form die Abbildung des Real-Systems auf das Modell erfolgt. Hier gibt es grundsätzlich zwei verschiedene Vorgehensweisen, die sich in gewisser Weise auch zu einer dritten Vorgehensweise verbinden lassen (vgl. [Bos92]). Grundsätzlich erfolgt die Unterscheidung danach, ob das *Systemverhalten* oder die *Systemstruktur* in dem zu erstellenden Modell nachgebildet wird.

„Black Box“-Modell

Von einem *Black Box*-Modell wird gesprochen, wenn mittels des Modells lediglich das *Systemverhalten* nachgebildet wird. Dabei ist es unerheblich, wie die interne Struktur, d. h. die Wirkungsstruktur des Modells aufgebaut ist; wichtig ist nur, dass das Verhalten des Modells nach außen hin identisch ist. Für die Konstruktion eines solchen Modells ist es nötig, das Verhalten des zugrundeliegenden Real-Systems zu beobachten, und das Modell auf Grundlage dieser Beobachtungen zu konstruieren.

„White Box“-Modell

In einem *White Box*-Modell wird im Gegensatz zu einem Black Box-Modell das Real-System in seiner für die Simulation wesentlichen Struktur nachgebildet (eventuelle für den Simulations- / Modellzweck irrelevante Teile können dabei selbstverständlich ausgelassen werden). Dementsprechend sollte das Modell, bezogen auf den Modellzweck, das gleiche Verhalten wie das Real-System zeigen. Offensichtlich muss für eine derartige *Strukturnachbildung* die Wirkungsstruktur des Real-Systems genau analysiert und erkannt werden, um ein hinreichend strukturtreues, akzeptables Modell konstruieren zu können. Wegen der genauen Nachbildung der Wirkungsstruktur des Real-Systems wird diese Art von Modell auch als *Glass Box*-Modell bezeichnet.

„Grey Box“ - Modell

Bei *Grey Box*-Modellen handelt es sich um die oben bereits erwähnte Mischform der zuvor beschriebenen Modell-Arten. Diese wird in der Praxis häufig angewendet, wenn die Wirkungsstruktur eines Real-Systems nur teilweise (aber nicht ausreichend genug für ein White Box-Modell) ermittelt werden kann. Hier wird versucht, diese Wirkungsstruktur „so gut wie möglich“ nachzubilden, und fehlende, d. h. unbekannte, Real-System-Eigenschaften so im Modell nachzubilden, dass das Modellverhalten nach außen dem Verhalten des Real-Systems entspricht. Daher müssen hier sowohl Analysen über die Wirkungszusammenhänge im Real-System wie auch Verhaltensbeobachtungen des Real-Systems durchgeführt werden, um ein solches auch als „opaque“ (halb-durchsichtig) bezeichnetes Modell erstellen zu können.

4.3.3 Vor- und Nachteile

Vor- und Nachteile bei der Verwendung von Modellen für die Simulation eines Real-Systems sollen hier nur kurz angesprochen werden. Für weitere Details sei auf die entsprechende Literatur verwiesen.

Ganz offensichtlich wäre der einfachste und präziseste Weg, um Informationen über das Verhalten eines Real-Systems zu erhalten, das Beobachten des Real-Systems selbst. Dieses ist aber u. U. sehr aufwändig oder kann sehr langwierig sein. Hier lässt sich offensichtlich mittels der Simulation eines geeigneten Modells eine Vereinfachung bzw. Beschleunigung erreichen. Ferner müssen keine Experimente am möglicherweise fragilen Real-System durchgeführt werden, d. h. dieses wird nicht gefährdet. Somit lässt sich an einem Modell auch das Systemverhalten in einem wesentlich breiteren Parameterbereich testen, z. B. können auch ganz bewusst Parametereinstellungen vorgenommen werden, die in dieser Form in einem Real-System nicht möglich wären. Nicht zuletzt lassen sich verschiedene, alternative Entwicklungen gleichzeitig simulieren, indem das Real-System in mehreren Modellen nachgebildet wird, und diese unabhängig voneinander mit verschiedenen Parametereinstellungen simuliert werden.

Dem gegenüber stehen die Nachteile, die mit der Verwendung eines explizit erstellten Modells anstelle des originalen Real-Systems einhergehen. Zunächst einmal muss aus dem Real-System das geeignete Modell entwickelt werden. Dieses ist mit einem gewissen z. B. zeitlichen oder finanziellen Aufwand verbunden, wohingegen das einfache Beobachten des Real-Systems ohne derartigen Aufwand sofort beginnen könnte. Außerdem ist ein Modell gerade ausdrücklich nicht das Original, sondern lediglich eine Nachbildung mit gewissen Abstraktionen. Somit bleibt prinzipiell immer die Unsicherheit bestehen, ob das Modell tatsächlich das Systemverhalten in allen Aspekten und mit der nötigen Detailtreue wiedergeben kann. Dieses ist nicht zuletzt auch mit Risiken verbunden; man stelle sich hier z. B. ein nicht korrektes Simulationsmodell eines Kernkraftwerkes vor, dessen fehlerhafte Ergebnisse dann auf den Betrieb des realen Kraftwerks übertragen werden.

4.4 Verfahren

Nachdem das entsprechende, zu simulierende Modell für das zuvor identifizierte Real-System erstellt worden ist, muss noch das für dieses Modell am besten geeignete *Simulationsverfahren* ausgewählt werden. Dabei lassen sich Simulationsverfahren in verschiedene Kategorien einteilen, ausgehend von der Art, wie sie das Systemverhalten über die Zeit simulieren. Diese Einteilung ist in Abbildung 4.2 dargestellt.

4.4.1 Zeitkontinuierliches Simulieren

Bei dieser Art der Simulation verläuft die der Simulation zugrundeliegende Zeit bzw. verlaufen die Zustandsänderungen des Modells über diese Zeit – wie schon der Name dieser Simulationsart erkennen lässt – *kontinuierlich*, d. h. hier gibt es keinerlei Zeit- bzw. Modellzustands-Sprünge. Die Grundlage für zeitkontinuierliche Simulation bilden oftmals mathematische Modelle, z. B. in Form von Differentialgleichungen.

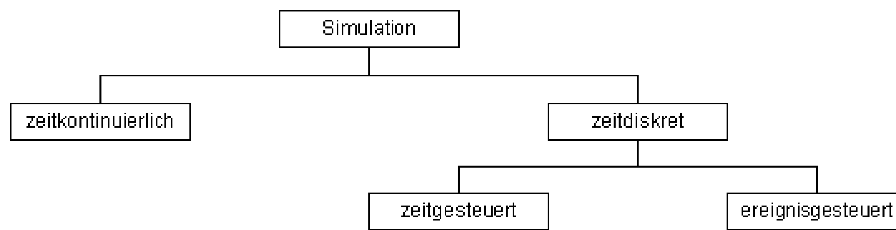


Abbildung 4.2: Einteilung der Simulations-Verfahren nach [FPW90]

Daher ist ein zeitkontinuierlich simuliertes Modell zu jedem beliebigen Zeitpunkt zwischen Start und Beenden der Simulation in einem exakt definierten, berechenbaren Zustand. Anwendung finden derartige Simulationsverfahren oft bei der Simulation von physikalischen, biologischen oder auch medizinischen Systemen. Als Beispiel sei hier die Berechnung von Strömungsdynamiken an Flugzeugflügeln genannt.

Zu beachten ist, dass zeitkontinuierliche Simulation auf den heute in aller Regel verwendeten Digitalrechnern nur annäherungsweise möglich ist. Aufgrund der sequentiellen Arbeitsweise dieser Rechner ist ein wirklich echtes kontinuierliches Vorgehen nicht möglich. Statt dessen ist es nur möglich, dieses Verhalten durch das sukzessive Berechnen von sehr sehr kleinen Zeitschritten zu imitieren bzw. sich diesem Verhalten zu weit wie möglich anzunähern. Hierbei spricht man dann von *quasi-kontinuierlicher Simulation* (vgl. [Kli81]).

4.4.2 Zeitdiskretes Simulieren

Die zweite Kategorie, in die sich Simulationsverfahren einteilen lassen, ist das sogenannte *zeitdiskrete Simulieren*. Bei diesen Verfahren verändert sich der Zustand des simulierten Modells nicht kontinuierlich, sondern *diskret* bzw. *sprunghaft*. Daher lässt sich hier für einen Zustand s_i zu einem Simulations-Zeitpunkt t_i mit einem direkten Nachfolgezustand s_{i+1} am Zeitpunkt t_{i+1} kein Zwischenzustand s_{i+k} zu einem Zeitpunkt $t_{i+k}, 0 < k < 1$, bestimmen. Bei zeitkontinuierlichem Vorgehen hingegen wäre dieses möglich bzw. berechenbar. Zugrunde liegen hier also Modelle, deren Zustände sich über die Zeit aus ihren Vorgängerzuständen entwickeln, und die auch nur an diesen diskreten Punkten an der Zeitachse betrachtet werden können.

Zeitdiskrete Simulationsverfahren lassen sich, wie in Abbildung 4.2 dargestellt, ebenfalls noch genauer unterteilen. Hierbei erfolgt die Unterteilung gemäß der Art, wie die diskreten Zeitpunkte, an denen sich das Simulationsmodell in einem definierten Zustand befindet, bestimmt werden (vgl. auch [FPW90]).

Zeitgesteuert

Von einem *zeitgesteuerten* Simulationsverfahren wird gesprochen, wenn die Zeit im Simulationsmodell in festen Zeitschritten Δt voranschreitet. Dabei werden Vorgänge innerhalb dieser Zeitspanne und somit eventuelle Zustandsveränderungen im Modell erst am Ende dieser Zeitspanne durchgeführt. Ganz offensichtlich kann es dabei auch so sein, dass sich der Modellzustand nicht nach jedem Zeitschritt ändert, d. h., dass nicht in jeden Zeitschritt auch mindestens ein Ereignis im Modell auftritt, das eine Zustandsänderung auslöst. Diese Zeit zwischen zwei Ereignissen im Modell wird – da in dieser Zeit gewissermaßen nichts passiert – als *Totzeit* bezeichnet. Totzeiten verursachen bei der Simulation einen überflüssigen Aufwand, der, wenn möglich, minimiert werden sollte. Dieses geschieht, indem die Zeitspanne Δt im Optimalfall weder zu groß, noch zu klein gewählt wird. Ein Extremfall bei dieser Vorgehensweise ist das zuvor bereits erwähnte *quasi-kontinuierliche* Simulieren: dabei handelt es sich um eine zeitdiskrete, zeitgesteuerte Vorgehensweise, bei der das Zeitintervall Δt derart klein gewählt wird, dass der Eindruck einer kontinuierlichen Vorgehensweise entsteht.

Ereignisgesteuert

Bei *ereignisgesteuerten* Simulationsverfahren schließlich werden nur noch genau die Zeitpunkte betrachtet, an denen sich der Zustand des Simulationsmodells verändert. Dieses sind die Zeitpunkte, an denen Ereignisse innerhalb des Modells auftreten. Dafür werden diese Ereignisse, sortiert nach den Zeitpunkten ihres Eintretens, in den sogenannten *Ereigniskalender* eingetragen. In diesem wird dann von Ereignis zu Ereignis übergegangen, unabhängig davon, wieviel Zeit zwischen einen Ereignis und dessen Nachfolgeereignis in der Praxis verstreichen würde. Totzeiten, wie sie zuvor in der zeitgesteuerten Simulation vorkommen konnten, werden hier also nun ausdrücklich vermieden. Neue zukünftige Ereignisse können offensichtlich auch nur entstehen, wenn in der Simulation gerade ein anderes, früher eintretendes Ereignis für bestimmte Modellkomponenten abgearbeitet wird. Werden derartige zukünftige Ereignisse erkannt, so werden sie im Ereigniskalender an die chronologisch richtige Stelle geschrieben, und entsprechend ausgeführt.

4.5 Zusammenfassung

Abschließend soll hier nun noch ein kurzer Überblick über das Vorgehens bei der Erstellung von Simulationen, ausgehend von den zuvor angesprochenen Begriffen und Verfahren, gegeben werden.

Wie beschrieben ist der erste, für die Erstellung einer Simulation notwendige Schritt das Erkennen und Analysieren des Real-Systems, das simuliert werden soll. Dazu sind insbesondere die einzelnen Komponenten des Systems und ihre Verknüpfungen untereinander zu identifizieren. Mittels dieser Verknüpfungen ergibt sich die Wirkungsstruktur des Systems, aus der sich das Systemverhalten ableitet. Ist es nicht möglich, die einzelnen Komponenten genau zu identifizieren, so muss das Systemverhalten beobachtet werden, um daraus für die Simulation notwendige Rückschlüsse ziehen zu können.

Nach erfolgter Analyse des Real-Systems bzw. des Verhaltens des Real-Systems muss ein für die Simulation möglichst gut geeignetes Simulations-Modell konstruiert werden. Dazu sind Abstraktionen und Idealisierungen des Real-Systems notwendig. Bei dem Modell handelt es sich also um eine vereinfachte Darstellung dieses Systems. Um einerseits das Real-System mit diesem Modell möglichst originalgetreu simulieren zu können, und andererseits unnötige, die Simulation erschwerende Modell-Komplexität zu vermeiden gilt es, das Ausmaß dieser Vereinfachungen gut zu wählen. Außerdem muss dabei entschieden werden, welche Art von Simulationsmodell konstruiert werden soll: war es z. B. nicht möglich, die interne Wirkungsstruktur des Real-Systems hinreichend genau zu untersuchen, so wird sich lediglich das Systemverhalten mittels eines Black Box-Modells nachbilden lassen. Waren andererseits genaue Untersuchungen möglich, so kann das Modell auch als z. B. White Box- oder Grey Box-Modell konstruiert werden.

Der letzte Schritt zur Erstellung der Simulation ist dann die Auswahl eines für das zuvor konstruierte Modell am besten geeigneten Simulationsverfahrens. Welche Art von Verfahren hier gewählt wird hängt dabei wieder davon ab, was für ein Real-System bei der Konstruktion des Simulationsmodells zugrunde lag: war dieses z. B. ein System, in dem physikalische Vorgänge simuliert werden sollen, so bietet sich (da derartige Vorgänge kontinuierlich ablaufen) ein zeitkontinuierliches oder ein quasi-zeitkontinuierliches Verfahren an. Soll hingegen z. B. eine Warteschlange simuliert werden, in der es nur wenige einfache Ereignisse wie z. B. „Kunde reiht sich ein“, „Bedienung beginnt“, und „Bedienung ist abgeschlossen“ gibt, so bietet sich eine zeitdiskrete, ereignisgesteuerte Vorgehensweise an.

Kapitel 5

Technisches System

In diesem Kapitel werden nun die zuvor vorgestellten Erkenntnisse für die und Vorgehensweisen bei der Erstellung einer Simulation auf das Konzept des zu simulierenden technischen Real-Systems, d. h. einer *virtuellen Förderanlage* angewendet. Dazu wird diese zunächst einer genaueren Betrachtung unterzogen, bei der insbesondere die einzelnen Komponenten des Systems mit ihrem jeweiligen Verhalten erläutert werden. Anschließend wird näher auf geeignete Simulations-Modelle eingegangen, und schließlich anhand dieser Modelle das für die Simulation am besten geeignete Simulations-Verfahren ausgewählt und die Entscheidung begründet.

5.1 Vorbemerkung

Bevor mit der Identifizierung des Real-Systems und daran anschließend der Modellbildung begonnen wird ist es notwendig, hier eine Vorbemerkung vorzunehmen. Gemäß der gestellten Anforderungen (vgl. Kapitel 6.1) soll es mittels der im Rahmen dieser Diplomarbeit zu erstellenden Software möglich sein, *beliebige* Modelle *virtueller* Förderanlagen zu simulieren. Dieses steht in gewisser Weise im Konflikt zu der in Kapitel 4.3 genannten Vorgehensweise im Rahmen der Modellbildung, bei der explizit von einem konkret existierenden, realen System ausgegangen wird. Ein derartiges System besteht dann wie dort erläutert aus Systemkomponenten, die untereinander wirkungsverknüpft sind. Da es hier aber nun ausdrücklich nicht *ein* solches System gibt, sondern die Simulation *beliebiger derartiger* Systeme möglich sein soll, ist es hier nicht möglich, bereits konkrete Wirkungsbeziehungen von real in Relation stehenden Systemkomponenten zu untersuchen und nachzubilden.

Statt dessen wird ganz allgemein das *Systemkonzept* des zugrundeliegenden Systems untersucht, und dabei werden die Komponenten identifiziert, aus denen ein derartiges System grundsätzlich bestehen *kann*. Dabei kann es selbstverständlich sein, dass bestimmte Komponenten, die hier als potentielle Systemkomponenten identifiziert und erläutert werden, bei der Erstellung eines konkreten Simulationsmodelles nicht benutzt werden. Beispiele für Förderanlagen und deren Komponenten, wie sie hier bei der Analyse des Systemkonzeptes zugrunde gelegt wurden, finden sich in [Jün88] und [tSN07].

Eine weitere Schwierigkeit liegt darin, dass es an dieser Stelle noch nicht möglich ist, *konkrete Wirkungsbeziehungen* zwischen Systemkomponenten zu untersuchen. Dieses ist so, da nicht bekannt ist, welche Komponente mit welchen anderen in Relation steht. Deswegen müssen die Komponenten bei der später folgenden Implementierung derart angelegt werden, dass sie „erkennen" können, mit welcher Art von weiteren Komponenten sie in Relation stehen. Abhängig davon müssen sie ihr Komponentenverhalten *selbst* gemäß dieser Relationen bestimmen bzw. steuern können. Alternativ muss es möglich sein, die Komponenten während der Zusammenstellung eines dann konkreten Simulations-Modells entsprechend manuell zu konfigurieren.

5.2 Identifizierung

Hier sollen nun in einem ersten Schritt die Funktionen bzw. die Aufgaben erläutert werden, die eine zu simulierende virtuelle Förderanlage erfüllen soll. Dieses entspricht dem in Abschnitt 4.2 genannten *Systemzweck*. Daraus wird dann entsprechend abgeleitet, welche Arten von *Systemkomponenten* mit welchem *Komponentenverhalten* und welchen *Attributen* es geben muss, um den zuvor genannten Systemzweck erfüllen zu können.

5.2.1 Systemzweck

Der Systemzweck einer der hier zu simulierenden Förderanlagen liegt darin, Elemente, die diese Anlage an einer Stelle (d. h. einer Systemkomponente) *betreten* durch diese Anlage hindurchzuschleusen, bis sie die Anlage schließlich wieder *verlassen*. Ein derartiges Element wird hier im weiteren auch mit *Payload* bezeichnet. Der *Weg*, den diese Payloads innerhalb der Förderanlage zurücklegen, ist dabei abhängig davon, wie die verschiedenartigen, zur Beförderung dieser Payloads vorgesehenen Systemkomponenten kombiniert sind, und wie diese möglicherweise gesteuert werden. Ferner soll es möglich sein, Payloads innerhalb einer solchen Anlage virtuell zu *bearbeiten*. Was genau unter einer derartigen Bearbeitung zu verstehen ist, ist für die Erfüllung des Systemzwecks uninteressant – deswegen wird dieses hier nicht näher betrachtet. Es sei lediglich festgehalten, dass es dafür innerhalb eines Fördersystem *Bearbeitungsstationen* geben kann. Ein Payload wird durch die Förderanlage zu einer solchen transportiert, dort bearbeitet, und nach abgeschlossener Bearbeitung wieder von der Förderanlage weitertransportiert. Wichtig dabei ist, dass in derartigen Bearbeitungsstationen *Defekte* auftreten können, was dann dazu führt, dass eine solche Station für eine gewisse Zeit ausfallen kann. Ein solcher Defekt stellt dann einen *Fehler* innerhalb der Förderanlage dar. Andere Fehler, die innerhalb einer solchen Förderanlage auftreten können, sind die Überlastung von einzelnen Systemkomponenten, oder aber, dass ein Systemelement ein zu lieferndes Paylaod nicht an seinen Nachfolger innerhalb der *Transportkette* liefern kann, da dieser z. B. gerade angehalten ist.

Als ein Beispiel für ein solches Förder- und Bearbeitungssystem kann man sich eine Fabrikhalle vorstellen, in der es ein Tor für den Wareneingang gibt. An diesem Tor kommen Werkstücke (hier eben als Payloads bezeichnet) an. Diese werden zunächst auf ein Förderband gelegt, welches sie dann zu einer Weiche transportiert, die diese Werkstücke auf verschiedene Maschinen verteilt, wo sie z. B. einen Farbanstrich erhalten. Nach erfolgter Bearbeitung an einer solchen Maschine werden die Werkstücke mittels eines Greifarms von der jeweiligen Maschine abgeholt, und auf ein weiteres Förderband gelegt. Dieses transportiert sie schließlich zu einem anderen Tor der Fabrikhalle, wo sie dann auf einen Lastwagen verladen werden und das Förder- und Bearbeitungssystem somit verlassen haben.

5.2.2 Systemkomponenten

In dem zuvor gegebenen Beispiel lassen sich bereits einige Elemente erkennen, die es in einem Modell zur Simulation einer derartigen Förder- und Bearbeitungsanlage offensichtlich geben muss. Ebenfalls erkennen lässt sich bereits eine erste Einteilung der Komponenten in drei verschiedene Klassen: so gibt es einerseits Komponenten, die am *Beginn* einer jeden Transport- und Bearbeitungskette in einem solchen System stehen, dann Komponenten, die an beliebigen Positionen *innerhalb* einer solchen Bearbeitungskette stehen, und schließlich noch Komponenten, die am *Ende* einer solchen Kette stehen. Hier werden nun alle diese Komponenten aufgeführt und erläutert.

InPort

Mit *InPort* wird hier die Art von Systemkomponente bezeichnet, über die Payloads ein Fördersystem *betreten* (somit entspricht ein InPort dem, was in obigem Beispiel das Wareneingangs-Tor ist). Eine der-

artige Komponente steht am Beginn einer jeden Transportkette des Systems, und stellt eine eingehende Schnittstelle dar, mit der dieses System mit anderen, externen Systemen verbunden ist. Diese externen Systeme und die Art, wie von diesen möglicherweise die Payloads geliefert werden, werden hier nicht berücksichtigt, da sie außerhalb der Systemgrenzen liegen. Statt dessen wird die Funktionsweise eines InPorts derart definiert, dass durch ihn in regelmäßigen zeitlichen Abständen Payloads in das System eintreten, und von dem in der Transportkette folgenden Element übernommen werden müssen. Dabei gilt, dass ein InPort erst dann ein weiteres Payload in das System geben kann, wenn das zuvor produzierte Payload von der Nachfolgekomponente im System übernommen worden ist. Ist dieses noch nicht geschehen, obwohl der InPort das nächste Payload in das System geben müsste, verursacht dieses einen Fehler im System.

OutPort

Ein *OutPort* stellt das Gegenstück zu der zuvor genannten InPort-Komponente dar. Statt am Anfang einer jeden Transportkette steht ein OutPort am Ende einer jeden solchen Kette. Dieses Element dient dazu, Payloads wieder aus dem System zu *„entlassen“*. Dieser Vorgang dauert für jedes Payload eine gewisse, festzulegende Abarbeitungszeit. Da möglicherweise während der Abarbeitung eines Payloads weitere ankommen, die daran anschließend ebenfalls abgearbeitet werden müssen, besitzt ein solches OutPort-Element eine gewisse einstellbare Kapazität, um Payloads zu „lagern“. Sollte diese Lagerkapazität überschritten werden, würde dieses einen Fehler im System verursachen. Wohin Payloads mittels eines OutPorts entlassen werden ist für das System unerheblich, da dieses wiederum außerhalb der Systemgrenzen liegt. Dementsprechend endet mit dem Ablauf der Abarbeitungszeit eines Payloads dessen Betrachtung. Im obigen Beispiel entspricht ein OutPort dem Tor, an dem die durch die Förderanlage beförderten Werkstücke wieder verladen werden.

Conveyer

Der *Conveyer* stellt die einfachste *innere* Systemkomponente in einem Fördersystem dar. Grundsätzlich handelt es sich hierbei um eine Art *Förderband* (vgl. wiederum obiges Beispiel), das dazu dient, Payloads von genau einer am Beginn des Conveyers angeschlossenen Vorgänger-Systemkomponente zu genau einer am Ende des Conveyer angeschlossenen Nachfolger-Systemkomponente zu transportieren. Ein Conveyer hat, ebenso wie z. B. der OutPort, eine bestimmte einstellbare Kapazität, d. h. eine maximale Anzahl von Payloads, die er gleichzeitig befördern kann. Wird diese maximal zugelassene Kapazität überschritten, so verursacht dieses einen Fehler im System. Ferner können die Längen verschiedener Conveyer variieren, ebenso wie die Transportgeschwindigkeiten. Schließlich existieren noch zwei verschiedene Modi, in denen ein Conveyer betrieben werden kann. Im einfachen Modus läuft ein Conveyer konstant mit seiner eingestellten Geschwindigkeit und versucht, Payloads, die an seinem Ende angekommen sind, unmittelbar an seine Nachfolge-Komponente weiterzureichen. Akzeptiert diese ein solches Payload gerade nicht, so würde auch hier ein Fehler im System auftreten (sehr einfach und bildhaft formuliert würde das Payload somit dann vom Transportsystem herunterfallen). Demgegenüber steht der zweite mögliche Modus, in dem ein Conveyer laufen kann: in diesem stoppt er vollautomatisch, wenn ein transportiertes Payload an seinem Ende ankommt, und läuft erst wieder an, wenn die Nachfolgekomponente dieses Payload angenommen hat. In diesem Moment kann allerdings ganz offensichtlich ein Fehler im System auftreten, wenn während dieses Stillstandes mehr als ein neues Payload von der Vorgänger-Komponente in der Transportkette an den Conveyer übergeben werden soll. Der Grund dafür liegt darin, dass sich zwei Payloads nicht gewissermaßen einen Platz auf dem Conveyer „teilen“ können.

Gate

Mittels eines *Gate* werden entweder von *einem* Vorgänger ankommende, zu transportierende Payloads auf *mehrere* Nachfolge-Komponenten aufgeteilt, oder aber von *mehreren* Vorgänger-Komponenten an-

kommende Payloads auf eine *gemeinsame* Nachfolger-Komponente gebündelt. Ein Gate entspricht somit einer *Weiche* mit entweder genau einem Ein- und mindestens zwei Ausgängen, oder alternativ mit mindestens zwei Ein- und genau einem Ausgang. Ähnlich wie schon zuvor bei dem OutPort dauert es auch bei einem Gate eine gewisse, einstellbare Zeit, bis ein zuvor angekommenes Payload abgearbeitet, d. h. zu dem entsprechenden Nachfolger des Gate weitergeleitet ist. Deswegen besitzt auch ein Gate eine gewisse, ebenfalls einstellbare Kapazität an Payloads, die es gleichzeitig aufnehmen kann. Auch hier entsteht ein Systemfehler, wenn diese Kapazität überschritten wird. Ein Gate, das Payloads von einem Eingang auf mehrere Ausgänge verteilt, kann in verschiedenen Modi laufen; der Modus bestimmt, an welches der verbundenen Nachfolge-Elemente das aktuell bearbeitete Payload übertragen wird. Im ersten der möglichen Modi wird dieses Element alternierend bestimmt, d. h. die Nachfolge-Elemente werden der Reihe nach durchgegangen. Alternativ ist es möglich, den aktuell zu benutzenden Nachfolger für jedes Payload per Zufall unter allen verbundenen Nachfolge-Elementen auszuwählen. Bei beiden dieser Modi kann es sein, dass der bestimmte Nachfolger gerade kein Payload akzeptiert, was dann ebenfalls wieder zu einem Systemfehler führt. Daneben gibt es noch die Möglichkeit, dass ein Gate den Nachfolger, an den das aktuell bearbeitete Payload geliefert werden muss, „intelligent“ bestimmt. Dabei werden dann alle Nachfolge-Komponenten durchlaufen, und das Payload an die erste Komponente übergeben, die dieses akzeptiert. Sollte keine der Nachfolge-Komponenten aktuell dieses Payload akzeptieren, so würde dieses auch hier einen Fehler im System verursachen. Im vierten möglichen Modus schließlich verhält sich das Gate derart, dass es solange alle Payloads an einen fix eingestellten Nachfolger liefert, bis dieser Nachfolger manuell auf eine andere der angeschlossenen Nachfolge-Komponenten umgestellt wird. Hierbei kann festgelegt werden, ob nur genau ein Payload an diesen neuen fix gesetzten Nachfolger geliefert werden soll, und danach wieder automatisch auf den vorherigen umgeschaltet wird, oder aber ob dieser neue Nachfolger solange als fix eingestellt bleibt, bis wieder manuell eine neue Änderung erfolgt. Auch hier entsteht ein Fehler im System, wenn ein eingestellter Nachfolger ein Payload nicht akzeptiert.

RobotGrabber

Der *RobotGrabber* entspricht dem in obigem Beispiel genannten *Greifarm*. Dieser dient ebenfalls dazu, Payloads von einem bestimmten Vorgänger- zu einem bestimmten Nachfolger-Element innerhalb der Transportkette zu befördern. Dabei ist eine wichtige Eigenschaft, dass der RobotGrabber eine gewisse *Positionierungszeit* benötigt, wenn der Greifarm gerade nicht an dem Anschluss steht, an dem das aktuell zu befördernde Payload aufzunehmen ist. Neben dieser Positionierungszeit braucht der Robotgrabber ebenfalls noch eine gewisse *Lieferzeit*, wenn er ein Payload aufgenommen hat, und sich mit diesem zu der Nachfolge-Komponente bewegt, an die dieses Payload zu liefern ist. Ferner kann immer nur ein Payload zur Zeit befördert werden.

WorkStation

Eine *WorkStation* realisiert die in 5.2.1 angesprochene *Bearbeitungsstation*. Was genau innerhalb einer solchen Bearbeitungsstation an Funktionalität vorhanden ist, ist, wie zuvor bereits erläutert, für den eigentlichen Systemzweck uninteressant und wird nicht näher betrachtet. Wichtig ist es lediglich festzuhalten, dass eine solche WorkStation zu einem Zeitpunkt immer nur ein Payload bearbeiten kann und eine derartige Bearbeitung eine gewisse, einstellbare Zeit in Anspruch nimmt. Ferner kann an einer WorkStation mit einer gewissen, ebenfalls einstellbaren Wahrscheinlichkeit ein Defekt auftreten, wenn sie ein Payload bearbeitet. Ein solcher Defekt dauert einen (festzulegenden) Zeitraum an, während dem die WorkStation stillsteht und (bildlich gesprochen) repariert wird. Desweiteren gilt noch, dass sich in dem betrachteten System per Definition eine WorkStation nur mittels eines RobotGrabbers an eine Transportkette anschließen lässt. Dieser dient dazu, Payloads sowohl zur WorkStation hin- als auch von ihr wegzubefördern. Eine Bearbeitungseinheit besteht also immer aus einem RobotGrabber und einer WorkStation.

5.2.3 Zusatz-Komponenten

Neben den aufgeführten und erläuterten *Systemkomponenten*, die eigenständig agieren können (solange sie z. B. im Fall einer inneren Komponente die Möglichkeit besitzen, Payloads zu erhalten und nach Erfüllung ihrer Teilaufgabe weiter zu geben), gibt es in dem zugrundegelegten Real-System noch eine weitere Klasse von Komponenten, die sich von diesem eigentlichen Systemkomponenten unterscheidet. Bei diesen handelt es sich um Komponenten, die zwecks Gewährleistung einer gewissen Zusatz-Funktionalität mit jeweils einer der Systemkomponenten kombiniert werden können. Eigenständig benutzbar, d. h. nicht in Kombination mit einer der Systemkomponenten, sind diese Zusatzkomponenten nicht. Beispielhaft kann man sich vorstellen, dass eine solche Zusatz-Komponente an einer der eigentlichen Systemkomponenten „montiert“ wird. Bezeichnet werden derartige Zusatz-Komponenten im weiteren Verlauf mit dem Oberbegriff *Auxiliary Devices*.

LightBarrier

Die *LightBarrier* stellt ein solches Auxiliary Device dar. Wie sich schon anhand des Namens erkennen lässt handelt es sich dabei um eine *Lichtschranke*. Diese kann mit einer der Systemkomponenten wie z. B. an einem Conveyer kombiniert werden, und löst ganz allgemein eine Aktion (wie z. B. ein Event) aus, wenn eines der mittels dieser Systemkomponente beförderten Payloads die Befestigungsposition der LightBarrier auf dieser Systemkomponente erreicht.

5.2.4 Service-orientierung der Komponenten

Jede der Komponenten des Systems („normale“ Systemkomponenten ebenso wie die zuvor angesprochenen Zusatz-Komponenten) stellt in diesem System einen *Web Service* bzw. ein *Device* mit bereitgestellten *Services* (vgl. Abschnitt 3.3 bzw. Abbildung 3.6) zur Verfügung. Dieses ermöglicht es, bestimmte komponenten-spezifische Informationen abzufragen, oder Einfluss auf Parameter dieser Komponenten zu nehmen, d. h. dieser in gewisser Weise steuern zu können.

5.3 Simulations-Modell

Gemäß dem in Kapitel 4 vorgestellten Vorgehen bei der Erstellung einer Simulation wäre der nächste, auf die Identifizierung des Real-Systems folgende Schritt das Erstellen eines Simulations-Modells für dieses Real-System. Wie aber bereits in Kapitel 5.1 erwähnt kann in diesem speziellen Fall hier ein solches Simulations-Modell nicht konkret erstellt werden. Statt dessen konnte nur das einem solchen Real-System zugrundeliegende *Systemkonzept* untersucht werden, wonach nun zwar alle für ein zu erstellendes Simulationsmodell möglichen Arten von Modell-Komponenten mit ihren zu berücksichtigen Eigenschaften bekannt sind, aber nicht, welche konkrete Komponente mit welchen anderen Komponenten verknüpft ist, und wie diese in Relation stehen. Dementsprechend ist es noch nicht möglich, ein festes Simulationsmodell zu entwerfen, sondern es können lediglich Voraussetzungen bzw. Anforderungen aufgestellt werden, die bei der Konzeptionierung und Implementierung der einzelnen Modell-Komponenten und für die Modell-Kontrolle bzw. Modell-Umgebung selbst zu beachten sind.

5.3.1 Anforderungen an die Modell-Komponenten

Aufgrund des angesprochenen Problems, dass an dieser Stelle des Entwurfs-Prozesses noch keine Informationen über die Relationen der einzelnen Komponenten untereinander bekannt ist liegt es nahe, dass das Verwalten der Relationen für eine bestimmte konkrete Komponente in einem Modell nicht von einer zentralen „Relations-Verwaltung“ für dieses Modell durchgeführt wird, sondern dass jede Komponente

eines Simulations-Modells dieses für sich selbst leistet. Dieses sind insbesondere die reinen „Verbindungsdaten“ einer solchen Komponente, d. h. jede Komponente muss eigenständig verwalten, welche anderen Komponenten ihre Vorgänger bzw. ihre Nachfolger im Simulationsmodell sind. Daneben muss jede Komponente selbstständig die Aktionen durchführen können, die notwendig werden, wenn an einer derartigen Verbindung eine Interaktion mit der entsprechend verbundenen Komponente nötig wird. Diese Aktionen lassen sich recht stark eingrenzen, da die einzige echte Interaktion der einzelnen Modell-Komponenten untereinander das Austauschen der angesprochenen Payloads ist. Demzufolge muss eine Komponente mit verbundenen „Vorgänger“-Komponenten erkennen, ob von einem solchen Vorgänger ein Payload zu übernehmen ist. Wenn dem so ist, muss dieses Payload entsprechend übernommen werden. Sollte dieses nicht möglich sein, weil z. B. die maximale Kapazität damit überschritten werden würde, so muss die betroffene Modell-Komponente selbst (und nicht etwa eine zentrale Modell-Kontrolle) einen Fehler auslösen und weitermelden. Andersherum muss eine Modell-Komponente, die Payloads zu „Nachfolge“-Komponenten weiterreicht, neben der Verwaltung der Zeitpunkte, zu denen dieses Weiterreichen durchzuführen ist, auch einen solchen Fehler einer Nachfolgekomponente erkennen können, um dann gegebenenfalls selbst einen Fehler auszulösen. Unabhängig davon, ob eine solche Interaktion erfolgreich war oder ein Fehler auftrat muss jede Komponente nach einer Interaktion selbstständig ihren Status aktualisieren; beispielsweise die Anzahl der aktuell auf einer Komponente befindlichen Payloads erhöhen oder verringern, usw. Das zuvor Beschriebene setzt ferner bereits implizit voraus, dass es ebenfalls möglich sein muss, alle Modellkomponenten darüber zu informieren, dass die Simulation fortschreitet, damit diese erkennen können, dass sie eventuell eine Aktion ausführen müssen. Daher muss in diesen Komponenten ebenfalls eine Funktionalität für das Empfangen derartiger Benachrichtigungen berücksichtigt werden.

Neben den bisher beschriebenen, eher auf die Interaktion von Modellkomponenten bezogenen Anforderungen gibt es einige weitere, die zu berücksichtigen sind. Bei diesen handelt es sich um Anforderungen bezüglich der „Verwaltung“ der Daten und Einstellungen von einzelnen Modellkomponenten. So muss jede Komponente selbstständig ihre Attribut-Einstellungen (z. B. benötigte Bearbeitungs- oder Transportzeiten, Kapazitäten, ...) verwalten können, und die Funktionalität bereitstellen, diese beim Erstellen oder Bearbeiten eines konkreten Simulations-Modells einstellen zu können. Außerdem muss jede Komponente zu protokollierende Daten (z. B. ihren aktuellen Zustand, ...) zusammenstellen können, und die Protokollierung dieser sofern gewünscht veranlassen. Schließlich ist es, da sich die Komponenten selbst verwalten, sinnvoll, diesen die Verwaltung von den für die Darstellung der Simulation benötigten Daten zu übergeben. Dieses sind z. B. die Daten bezüglich der Position der Modellkomponente innerhalb der Simulations-Visualisierung, ihre Grösse, usw. Derartige Daten sind für die eigentliche Simulation bzw. für den eigentlichen Zweck des Simulations-Modells zwar irrelevant, müssen aber ebenfalls verwaltet werden.

5.3.2 Anforderungen an die Modell-Umgebung

Die Anforderungen an die Umgebung eines Modells, d. h. an den Rahmen, in den ein jedes Simulations-Modell eingebettet wird, lassen sich aus den zuvor formulierten Anforderungen an die einzelnen Modell-Komponenten ableiten. So ist es nötig, dass aus der Umgebung des Simulations-Modells Informationen über das Fortschreiten der Simulation an jede der Komponenten gegeben werden, damit diese ihre zeitlich entsprechenden Aktionen durchführen können. Das Fortschreiten der Simulation muss in der Modell-Umgebung verwaltet werden. Außerdem muss diese Umgebung Kenntnis von jeder der im Modell existierenden Komponenten haben. Da im Rahmen der angesprochenen Aktionen möglicherweise Fehler durch die einzelnen Komponenten ausgelöst werden, muss die Umgebung neben diesem Zeitgeber auch eine Funktionalität bereitstellen, mit der derartige Fehler angenommen werden können, um sie z. B. dem Benutzer mitzuteilen, und dann auf eine Reaktion von diesem zu warten. Schließlich muss es in der Umgebung eines Modells noch die Möglichkeit für die Modellkomponenten geben, die zusammengestellten Protokolldaten einer jeden Komponente zusammenzuführen und zu sichern.

5.4 Gewähltes Simulationsverfahren

Nach den zuvor angestellten Überlegungen bezüglich der Art des Simulations-Modells steht nun noch die Auswahl eines geeigneten Simulations-Verfahrens aus. Dafür sei zunächst noch einmal erinnert an die in Kapitel 4.4 erläuterte und in Abbildung 4.2 dargestellte Einteilung der verschiedenen Simulationsverfahren. Aus dieser ist ersichtlich, dass eine erste Einteilung gemäß der Übergänge der Systemzustände erfolgt, abhängig davon, ob diese Übergänge kontinuierlich oder diskret stattfinden. Bei dem hier zugrundeliegenden Real-System erfolgen diese Zustandsübergänge offensichtlich diskret; nämlich dann, wenn Payloads innerhalb des Systems weitergereicht werden, oder wenn ein Fehler innerhalb des Systems auftritt. Der eigentliche Vorgang des Weiterreichens von Payloads, der als kontinuierlicher Vorgang angesehen werden könnte, wird hier nicht berücksichtigt, jedes Payload innerhalb des Systems ist immer genau einer Komponente zugeordnet. Ähnlich verhält es sich beim Auftreten von Fehlern: entweder das System ist fehlerfrei, oder aber es existiert mindestens ein Fehler. Der eigentliche Prozess, in dem dieser Fehler entsteht (welcher ebenfalls wieder als kontinuierlich angesehen werden könnte), wird nicht betrachtet. Dementsprechend kann ein Modell für ein solches Real-System am sinnvollsten diskret simuliert werden.

Gemäß Abbildung 4.2 lassen sich diskrete Verfahren genauer einteilen, abhängig davon, ob ein zeit- oder ein ereignisgesteuertes Verfahren benutzt wird. Da einer der aufgestellten Anforderungen gerade die Forderung nach Unterstützung von Echtzeit und simulierter Zeit ist liegt es hier nahe, ein zeitgesteuertes Simulationsverfahren auszuwählen. Dieses bietet sich auch an, da bei bestimmten System-Elementen Wahrscheinlichkeitswerte eine Rolle spielen (z. B. bei einer Bearbeitungsstation, an der pro Schritt mit einer gewissen Wahrscheinlichkeit ein Defekt auftreten kann). Derartige Wahrscheinlichkeitswerte ließen sich grundsätzlich natürlich auch für zukünftige Schritte berechnen, und entsprechend eintretende Ereignisse in der Ereignisliste (vgl. Kapitel 4.4.2) verwalten; allerdings ist es deutlich weniger aufwändig, derartige Ereignisse bzw. Wahrscheinlichkeiten direkt in jedem Schritt zu berechnen. Insgesamt wird hier also ein zeitdiskretes, zeitgesteuertes Simulationsverfahren benutzt. Um Missverständnisse zu vermeiden sei bereits an dieser Stelle darauf hingewiesen, dass es in der späteren Simulations-Software die Möglichkeit geben wird, gewissermaßen „manuell“ Ereignisse in den Ablauf der Simulation hineinzugeben (beispielsweise indem „von Hand“ ein Defekt in einer Bearbeitungsstation erzeugt wird). Derartige Ereignisse werden immer erst mit Ablauf des nächsten Zeit-Schrittes ausgeführt, d. h. dieses stellt keine Einschränkung der zeitgesteuerten Vorgehensweise dar.

Kapitel 6

Anforderungen an die Simulations-Software

Bevor auf den Entwurf und die eigentliche Implementierung der Simulations-Software eingegangen wird, sollen in diesem Kapitel die an diese Software gestellten Anforderungen aufgeführt und erläutert werden.

6.1 Beliebige Modelle

Eine zentrale, an die Simulations-Software gestellte Anforderung ist, wie bereits an verschiedenen Stellen angeführt, die Berücksichtigung der Möglichkeit, *beliebige Modelle* derartiger Förderanlagen zu benutzen. Diese Anforderung lässt sich aufteilen in zwei Teilbereiche: zum einen muss es mittels der Simulations-Software möglich sein, beliebige Konstellationen der identifizierten und implementierten Modell-Komponenten *verarbeiten* zu können. Ferner sollte es mittels der Software möglich sein, beliebige Modelle entsprechend *entwerfen* zu können. Die Software sollte dem Benutzer dafür Funktionalität zur Verfügung stellen, mit der dieser möglichst einfach zulässige Simulations-Modelle zusammenstellen kann. Dieses schließt das möglichst einfache Konfigurieren der einzelnen Modell-Komponenten mittels dieser Funktionalität ein.

6.2 Dateioperationen und Dateiformate

Die zuvor formulierte Forderung nach der Möglichkeit, Modelle mittels der Software erstellen zu können führt zu einer weiteren Anforderung. Dieses ist die, erstellte Modelle abspeichern und wieder laden zu können.

Damit verbunden ist die Forderung nach der Verwendung einer *leicht verständlichen Struktur* für die Modell-Dateien, damit diese Dateien z. B. auch bei der direkten Betrachtung mittels eines Editor leicht zu erfassen sind. Optimalerweise sollte das Dateiformat so gewählt werden, dass eine möglichst große Kompatibilität zu anderen Software-Applikationen gewährleistet ist. Wenn möglich sollen sich erstellte Modelle auch ohne großen Aufwand in andere Anwendungen laden und dort sinnvoll darstellen oder benutzen lassen. Dieses führt insgesamt zu der Forderung nach der Verwendung von *XML* als standardisiertes Format für Dateien von aus der Software heraus abzuspeichernden Simulations-Modellen.

6.3 Zeit

Eine wichtige Forderung, die an die Software gestellt wird, ist die Forderung nach Unterstützung von Simulationen in *Echtzeit* und in *simulierter Zeit*. Dieses bedeutet, dass es mittels der zu erstellenden Software möglich sein soll, eine Simulation sowohl in realer Zeit arbeiten zu lassen, als auch in beschleunigter Zeit. Letzteres ermöglicht es dann beispielsweise, schneller an Ergebnisse zu kommen, für die ein System eine gewisse (längere) Zeit betrieben werden müsste. Da Umschalten zwischen einer Simulation, die in Echtzeit läuft, und einer Simulation, die beschleunigt läuft, soll dabei zu jedem Zeitpunkt möglich sein. Ferner soll das Einstellen der beschleunigten Zeit bzw. des Faktors, um den die Zeit beschleunigt wird, ohne Eingriffe in eine laufende Simulation möglich sein, d. h., dass eine Simulation nicht z. B. erst angehalten werden muss, um den Faktor der Zeitbeschleunigung verändern zu können.

6.4 Externe Steuerung und Kontrolle

Anforderungen bezüglich der *externen Steuerung und Kontrolle* werden einerseits an die Simulations-Umgebung, d. h. an die Software, in der ein simuliertes Modell ausgeführt wird, andererseits aber ebenfalls auch an ein derartiges Modell bzw. jede einzelne Komponente eines solchen Modells gestellt. Daher werden die Anforderungen an die Simulations-Umgebung und die Anforderungen an die einzelnen Komponenten des Modell-Frameworks gesondert betrachtet.

6.4.1 Anforderungen an die Simulations-Umgebung

Steuerungs- und Kontrollanforderungen an die Simulations-Umgebung beziehen sich in erster Linie darauf, mit welchen Mitteln es einem Benutzer der Simulations-Software z. B. möglich sein soll, Simulations-Läufe zu starten, anzuhalten, Informationen über den aktuellen Status einer Simulation zu erhalten, oder auch den Beschleunigungsfaktor (vgl. 6.3) der Simulations-Zeit einzustellen. Dieses soll zum einen natürlich direkt über eine entsprechende graphische Benutzerschnittstelle ermöglicht werden, mittels der dem Benutzer der Zugriff auf derartige Informationen und Funktionen gegeben wird. Zum anderen soll es aber auch möglich sein, die Software „fernbedient“ zu benutzen. Dazu ist es notwendig, von außerhalb des eigentlichen Simulations-Programmes aus auf Programm-Funktionen und Informationen zugreifen zu können. Dabei handelt es sich beispielsweise um Anweisungen zum Starten oder Beenden eines Simulations-Laufes, um das Einstellen des zuvor beschriebenen Zeitbeschleunigungs-Faktors, die Möglichkeit, das Laden eines neuen Simulations-Modells zu veranlassen, usw. Daneben sollen Informationen bezüglich z. B. dem Status der aktuellen Information abgefragt werden können, wie weit die Simulation bereits fortgeschritten ist, usw. Die Realisierung dieser „Fernbedienbarkeit“ soll erfolgen, indem die Simulations-Software *selbst* einen Web Service bereitstellt, der als Schnittstelle derartige Zugriffe fungiert. Dieses ermöglicht es dann, solche Zugriffe z. B. auch von einem entfernten Rechner aus vorzunehmen.

6.4.2 Anforderungen an Komponenten des Modell-Frameworks

Eine der Hauptaufgaben der mittels der zu erstellenden Software zu simulierenden Modelle liegt darin, anhand dieser Modelle Verfahren zur Gewährleistung von fehlertolerantem Verhalten in service-orientierten Architekturen (vgl. Kapitel 2 und 3) zu testen. Dazu ist es notwendig, dass im Fall eines aufgetretenen Fehlers Modifikationen (abhängig von der Art des Fehlertoleranzverfahrens) an mindestens einer der Modell-Komponenten vorgenommen werden können. Da die zu testenden Fehlertoleranzverfahren unabhängig von der Simulations-Software zu implementieren sind, d. h. eigenständige Programme darstellen, muss jede einzelne der vom Modell-Framework bereitgestellten Komponenten eine *Schnittstelle* nach außen zur Verfügung stellen. Mittels dieser soll es möglich sein, bestimmte Parameter der Modell-Komponente zur Laufzeit der Simulation flexibel anzupassen. Die Realisierung dieser Schnittstellen soll erfolgen, indem

jede in einem Simulations-Modell vorhandene Komponente einen *Dienstanbieter* im Sinne der Web Services darstellt (vgl. Abschnitt 3.1.2), welcher entsprechende *Dienste* bzw. *Services* (vgl. Abschnitt 3.1.1) zur Parameter-Anpassungen oder Rekonfiguration der zugrundeliegenden Modell-Komponente anbietet. Daneben muss jeder dieser Dienste der Modell-Komponenten noch die Funktionalitäten bereitstellen, mit denen es dem externen Fehlertoleranzprogramm möglich ist zu erkennen, ob eine Komponente fehlerfrei arbeitet, oder ob möglicherweise ein Fehler aufgetreten ist und somit ein Eingreifen notwendig wird.

6.5 Devices Profile for Web Services - Technologie

Die in Kapitel 6.4 aufgestellte Forderung nach Bereitstellung von externen Steuerungs- und Kontrolmöglichkeiten durch Anbieten von entsprechenden Web Services soll hier mittels Verwendung von *Devices Profile for Web Services*-konformer Implementierung realisiert werden (vgl. Abschnitt 3.3). Benutzt werden soll der *WS4D J2ME-Stacks*, der eben eine konkrete DPWS-Implementierung (vgl. [WS4b]) darstellt. Für weitere Details bezüglich dieses zu verwendenden Protokollstapels sei verwiesen auf 7.3.3.

6.6 Benutzerschnittstelle

Grundsätzlich ist bei dem Entwurf der Simulations-Software eine graphische Benutzerschnittstelle mit zu entwerfen und zu implementieren. Diese hat zunächst die Funktion, dem Benutzer die Interaktion mit dem Programm zu ermöglichen. Das umfasst den Entwurf einer geeigneten Menü-Struktur, mittels der es dem Benutzer ermöglicht wird, das Programmfunktionen wie Laden, Speichern usw. zu bedienen. Ferner soll es mittels dieser GUI einfach und intuitiv möglich sein, einen Simulations-Lauf zu kontrollieren und zu steuern. Dieses bedeutet insbesondere das Starten, Pausieren und Stoppen von Simulations-Läufen, aber auch das flexible Einstellen der anzuwendenden Zeitbeschleunigung in der Simulation. Schließlich soll die GUI dazu dienen, dem Benutzer Informationen über eine laufende Simulation anzuzeigen. Dieses umfasst einerseits das bloße Anzeigen von Status-Informationen der Simulation, andererseits aber insbesondere auch das Visualisieren von Simulations-Modellen unter Berücksichtigung des aktuellen Zustandes der Simulation bzw. jeder einzelnen Komponente des Simulations-Modells.

Dem gegenüber steht eine weitere zu berücksichtigende Anforderung an die Simulations-Software. Dieses ist die Forderung, dass bei dem Gesamtentwurf der Software die Möglichkeit erhalten bleibt, diese Software ohne übermäßig großen Aufwand so zu modifizieren, dass sie komplett ohne jede graphische Benutzeroberfläche betrieben werden kann. Daher gilt es zu beachten, dass an keiner Stelle der Implementierung eine graphische Oberfläche derart tief mit einbezogen wird, dass sich diese nicht wieder aus dem Gesamtkonzept herauslösen lässt. In dieser Forderung liegt auch mit einer der Gründe für die in Abschnitt 6.4.1 geforderte Berücksichtigung einer externen Programm-Steuerung. Diese soll in einer eventuellen nicht-graphischen Version der Software zur Steuerung dieser dienen.

6.7 Protokollierung

Eine weitere an die Simulations-Software gestellte Forderung ist die nach der Möglichkeit zur Protokollierung einer Simulation bzw. eines Simulations-Ablaufes. Hier lassen sich die zu protokollierenden Daten einteilen in einerseits die Daten, die im Rahmen der Simulations-Verwaltung in ein Protokoll geschrieben werden müssen (wie z. B. die Daten, wann eine Simulation gestartet wurde, in welchem Simulationsschritt sich die Simulation befindet bzw. wieviel Simulationszeit bereits vergangen ist, usw.), und andererseits die Daten, die für jede einzelne Simulations-Komponente zu protokollieren ist. Die Funktionalität, erstere Daten zu schreiben, ist dabei in der die Simulations-Umgebung realisierenden Software zu berücksichtigen (da über diese z. B. die Anweisungen zum Starten oder Anhalten eines Simulations-Laufes gegeben werden, oder auch, da von dieser der Zeittakt gegeben wird). Die Daten, die für die Protokollierung jeder

einzelnen Modell-Komponente nötig sind können hingegen – wiederum begründet in der Forderung nach beliebigen verwendbaren Modellen – nur direkt in jeder dieser von Modell-Framework bereitgestellten Komponenten zusammengestellt werden, so dass diese Funktionalität in diesen Komponenten berücksichtigt werden muss. Das Zusammenführen und Sichern dieser Protokoll-Informationen der einzelnen Komponenten wiederum ist eine Aufgabe, die durch die Simulations-Umgebung zu leisten ist.

6.8 Modifizierbarkeit

Hinter der Forderung nach einfacher Modifizierbarkeit verbergen sich ebenfalls wieder Anforderungen an einerseits die Simulations-Umgebung, und andererseits an das Framework für Simulationsmodelle, d. h. an die zu erstellenden Software-Komponenten, mittels denen sich innerhalb der Simulations-Umgebung ein konkretes Simulations-Modell zusammenstellen lässt. Daher sollen diese verschiedenen Anforderungen hier auch wieder seperat betrachtet werden.

6.8.1 Modifizierbarkeit der Simulations-Umgebung

Die Simulations-Umgebung soll derart entworfen und implementiert werden, dass es ohne übermäßig großen Aufwand möglich ist, bestimmte Software-Teile durch anders aufgebaute Teile zu ersetzen, die eine vergleichbare Funktionalität bereitstellen, diese aber anders realisieren. Dafür ist es notwendig, die Simulations-Umgebung *modular* zu entwerfen und zu implementieren. Dieses bedeutet, dass eine bestimmte Funktionalität der Simulations-Verwaltung durch einen exakt abgrenzbaren Teil der Software (eben ein *Modul*) bereitgestellt wird. Soll die Funktionalität nun auf andere Art und Weise bereitsgestellt werden, so muss lediglich dieses spezielle Modul der Simulations-Umgebung entsprechend geändert oder ersetzt werden.

Ein Beispiel hierfür ist das Modul, dass das Fortschreiten der Simulationszeitpunkte für jede Simulation vorgibt. Wie beschrieben wird hier ein zeitgesteuertes Simulationsverfahren eingesetzt, d. h. dieses Modul muss entsprechend zu festen Zeitintervallen ein Fortschreiten der Simulation veranlassen. Sollte nun anstelle dieses Simulationsverfahrens möglicherweise doch einmal ein ereignisgesteuertes Simulationsverfahren ermöglicht werden sollen, so muss lediglich dieses Modul gegen ein entsprechend anders aufgebautes Modul ausgetauscht werden. Dieses „neue“ Modul müsste dann anstelle der festen Zeitintervalle den in 4.4.2 beschriebenen Ereigniskalender verwalten, und die zu simulierenden Zeitpunkte, d. h. die Zeitpunkte, zu denen ein neuer Zustand für das Simulations-Modell berechnet werden müsste, anhand dieser Liste bestimmen.

6.8.2 Modifizierbarkeit des Modell-Rahmens

Auch der Entwurf und die Implementierung des Modell-Rahmens soll *modular* erfolgen. So soll es zu späteren Zeitpunkten unkompliziert möglich ist, weitere (neue) Modell-Komponenten zu diesem Modell-Rahmen hinzuzufügen oder „alte“ Komponenten aus dem Rahmen zu entfernen. Dazu ist es einerseits notwendig, dass alle Modell-Komponenten die gleiche einheitliche Struktur aufweisen, und diese auch für neue Komponenten vorgegeben ist. Anderseits müssen dafür auch die einzelnen Komponenten voneinander abgegrenzt entworfen und implementiert werden, so dass jede Komponente für sich ein einzelnes Modul darstellt, dass unabhängig von den anderen modifiziert oder entfernt werden kann.

Daneben soll auch die *innere Struktur* einer jeden Komponente modular aufgebaut sein, d. h. dass für eine bestimmte Funktion ein bestimmtes Modul *innerhalb* der Komponente zuständig ist. Dieses hat den Vorteil, dass bei Änderung einer bestimmten Funktionalität in einer Komponente keine „Seiteneffekte“ auf andere Funktionalitäten auftreten können.

Kapitel 7

Grundlagen der Implementierung

Bevor in den Kapiteln 8 und 9 auf die im Rahmen dieser Diplomarbeit entworfene und erstellte Software in ihren Einzelheiten eingegangen wird, sollen in diesem Kapitel einleitend dazu noch einige Grundlagen erläutert werden.

7.1 Unterteilung der implementierten Software

Die erstellte Software lässt sich logisch, wie dieses bereits implizit in den vorangegangenen Kapiteln getan wurde, in zwei Teile trennen. Der erste dieser beiden Teile gibt das Umfeld vor, in dem ein konkretes, zu simulierendes Modell ausgeführt wird. Dazu gehört es dann z. B., dass aus diesem Umfeld die Informationen bzw. Anweisungen zum Starten oder Anhalten des Simulationsvorganges kommen, dass die Anweisungen bezüglich der Ausführung der Simulationsschritte an das simulierte Modell gegeben werden, dass dieser die Visualisierung einer laufenden Simulation ermöglicht, usw. Bezeichnet wird dieser Teil der Software mit *Simulations-Umgebung*.

Der zweite logische Teil ist derjenige, mittels dem dann entsprechend die beliebigen Simulations-Modelle der zu simulierenden Förderanlagen erstellt werden können. Die verwendete Bezeichnung für diesen Teil lautet *Modell-Framework*. Der Grund für diese Bezeichnung liegt darin, dass mittels des Modell-Frameworks keine konkreten Simulations-Modelle vorgegeben werden, sondern dieses Framework lediglich den (Software-) Rahmen bereitstellt, mit bzw. in dem beliebige Simulations-Modelle zusammengestellt werden können. Solche Simulations-Modelle werden dann in die Simulations-Umgebung eingebettet und interagieren mit dieser. Die Unterteilung und einige beispielhafte Interaktionen sind darstellt in Abbildung 7.1.

7.2 Programmiersprache Java

Die Implementierung der Simulations-Umgebung und des Modell-Frameworks erfolgte mit der von Sun Microsystems seit Anfang der 90er Jahre entwickelten objektorientierten Programmiersprache *Java*. Benutzt wurde dabei die Version 1.5 (vgl. [Sun04]). Aufgrund der inzwischen sehr weiten Verbreitung von Java und dem damit verbundenem hohen Bekanntheitsgrad der Details dieser Programmiersprache wird hier nicht in aller Ausführlichkeit auf diese eingegangen. Es sollen lediglich einige wenige bei der Implementierung der Software sehr wichtige (weil benutzte) Eigenschaften von Java aufgezählt und kurz erläutert werden.

- *Multithread-Fähigkeit*: in Java war es von vornherein vorgesehen, innerhalb eines Programmes mit mehreren parallelen Threads zu arbeiten (aufgrund der Art der Ausführung eines Programmes

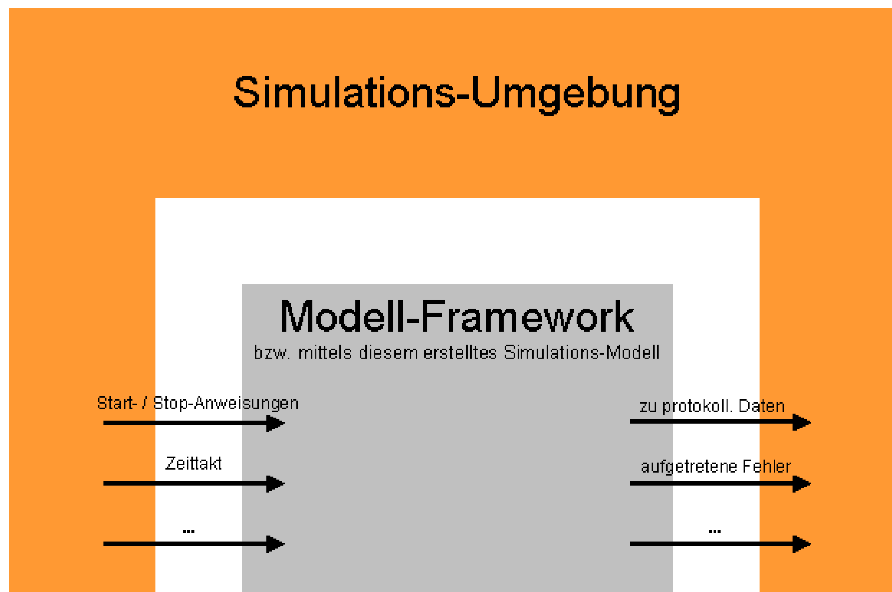
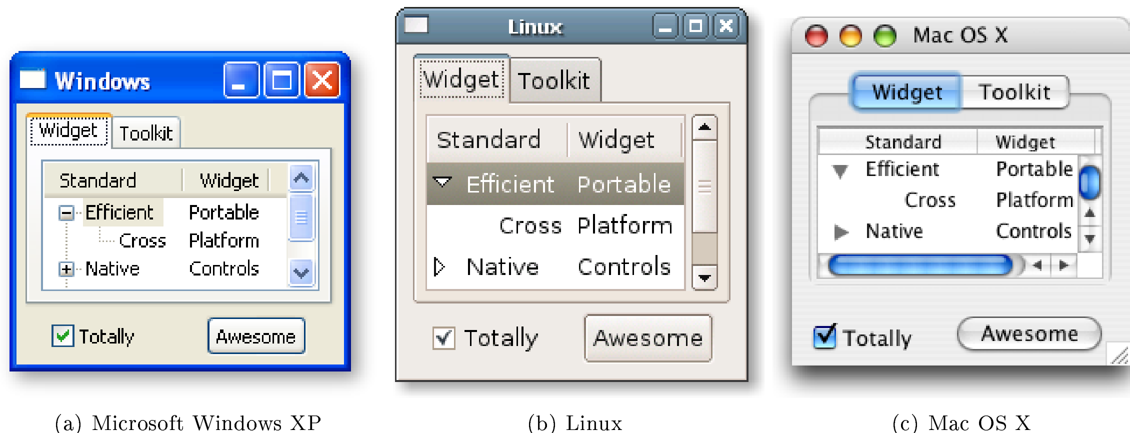


Abbildung 7.1: Unterteilung der Software mit beispielhaften Interaktionen

innerhalb der virtuellen Maschine ist dieses mittels Java ebenfalls unter Betriebssystemen, die Multithreading an sich nicht unterstützen, möglich). Betrachtet man die Anforderungen, die an die zu erstellende Software gestellt sind, so ist klar erkennbar, dass diese Multithread-Fähigkeit hier z. B. benötigt wird, um den Zeittakt-Geber für eine laufende Simulation nebenläufig zu betreiben, oder um die nebenläufige Kommunikation via der Web Services bereitstellen zu können.

- *Objektorientierung*: eines der Kernprinzipien der objektorientierten Programmierung (und somit auch von Java) ist die *Vererbung*. Mittels dieser ist es möglich, ähnlich aufgebaute Software-Komponenten (die *Klassen*), die teilweise die gleichen Funktionalitäten bereitstellen, auf eine gemeinsame *Basisklasse* zurückzuführen. Diese stellt dann diese gemeinsamen Funktionalitäten bereit. In den von dieser Basisklasse erbenden Komponenten ist es dann nur noch nötig, die für diese Komponenten spezifischen Funktionalitäten hinzuzufügen, was eine Vereinfachung in der Programmierung darstellt. Angewandt wurde diese Vorgehensweise hier insbesondere bei der Erstellung der Komponenten, die das Modell-Framework für Simulations-Modelle bereitstellt (vgl. Kapitel 8).
- *Plattformunabhängigkeit*: ein weiterer großer Vorteil, der für die Verwendung von Java als Programmiersprache bei der Implementierung der Simulations-Umgebung und des Modell-Frameworks spricht, ist die grundsätzliche Plattformunabhängigkeit von Java-Programmen. Existiert für eine bestimmte Plattform die entsprechende Laufzeitumgebung, so kann auch ein Java-Programm, das nicht explizit unter dieser Plattform entwickelt wurde, auf dieser betrieben werden. Dieses ermöglicht es in diesem speziellen Fall hier beispielsweise, die Performance der Kommunikation via der gleichen bereitsgestellten Web Services grundsätzlich unter verschiedenen Plattformen zu testen. Für gewisse Einschränkungen, denen dieses hier unterworfen ist, sei hier bereits verwiesen auf Abschnitt 7.3.1.

Neben diesen gibt es selbstverständlich noch zahlreiche weitere Eigenschaften, die der Programmiersprache Java zugrundeliegen. Wie einleitend bereits geschrieben würde es an dieser Stelle zu weit führen, auf all diese Eigenschaften und Details einzugehen. Statt dessen sei hier auf weiterführende Literatur wie z. B. [DD02] oder die Java-Dokumentation von Sun Microsystems, Inc. ([Sun]) verwiesen.



(a) Microsoft Windows XP (b) Linux (c) Mac OS X

Abbildung 7.2: Darstellung des gleichen SWT-Fensters unter verschiedenen Betriebssystemen (Bilderquelle: [Eclb])

7.3 Benutzte Bibliotheken

Bei der Realisierung einiger Teile der Software (sowohl bei der Implementierung der Simulation-Umgebung als auch bei der Implementierung des Modell-Frameworks) wurde auf verschiedene fertige, frei zur Verfügung stehende Bibliotheken zurückgegriffen. Hier erfolgt nun noch eine kurze Übersicht, welche dieses im einzelnen waren, und wofür diese verwendet wurden.

7.3.1 SWT

Für die Gestaltung der graphischen Benutzerschnittstelle der Software wurde das von IBM im Jahre 2001 vorgestellte, ursprünglich für die Entwicklungsumgebung Eclipse (s. [Ecla]) entwickelte *Standard Widget Toolkit* (oder kurz *SWT*) eingesetzt. Hierbei handelt es sich um Open-Source-Software, die unter der *Eclipse Public License* (vgl. [Ecl07b]) steht. Im Gegensatz zu dem beispielsweise ebenfalls für die GUI-Gestaltung denkbaren, von Sun Microsystems, Inc. entwickelten und standardmäßig zur Java-Laufzeitumgebung zugehörigem Swing nutzt das Standard Widget Toolkit die nativen Oberflächenkomponenten des Betriebssystems, das der Programmausführung zugrunde liegt (bzw. bedient sich dieser vom Betriebssystem bereitgestellten Komponenten). Dabei werden Aufrufe von graphischen Komponenten, die von der SWT-API aus erfolgen, so direkt wie möglich an das Betriebssystem weitergegeben, was sich verglichen mit Swing positiv auf die Geschwindigkeit von mit SWT erstellten Oberflächen auswirkt. Außerdem sorgt dieses Vorgehen dafür, dass diese Oberflächen dem „normalen“ Erscheinungsbild der graphischen Komponenten des verwendeten Betriebssystems entsprechen. Dieses ist für ein sehr einfaches Fenster exemplarisch in Abbildung 7.2 dargestellt.

Aufsetzend auf SWT exisitiert noch die hier ebenfalls verwendete Klassenbibliothek *JFace*, welche aus SWT-Basiskomponenten komplexere Komponeten, sogenannte *Widgets*, zusammenstellt und den Zugriff auf diese Basiskomponenten somit vereinfacht. Während das SWT selbst eigenständig unter [Eclb] zu beziehen ist, ist die Klassenbibliothek JFace lediglich als Bestandteil eines Eclipse-Paketes (s. [Ecla]) zu beziehen. Konkret verwendet wurden für die Erstellung der Simulations-Umgebung und des Modell-Frameworks die SWT-Version 3.320 sowie die JFace-Version Version 3.2.0 (welches dann auch die Version der verwendeten Eclipse-Entwicklungsumgebung war).

Die nahe Anbindung des SWT an die nativen Komponenten des Betriebssystemes stellt allerdings in gewisser Weise einen Nachteil bezüglich der zuvor angesprochenen Plattformunabhängigkeit von Java dar. So wird für jede Ausführung eines mit dem Standard Widget Toolkit erstellten Programmes ei-

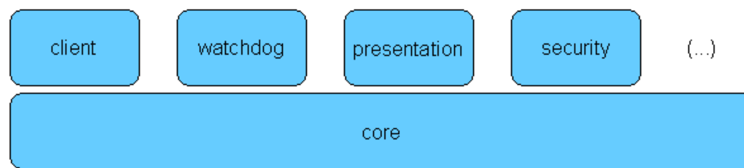


Abbildung 7.3: Module des WS4D-J2ME - Stacks nach [ZBB+07]

ne weitere plattformspezifische SWT-Systembibliothek benötigt, die die Kommunikation zwischen dem Java-Programm bzw. der virtuellen Maschine einerseits und dem zugrundeliegenden Betriebssystem andererseits steuert. Diese Bibliothek ist der VM entsprechend zur Verfügung zu stellen. Existiert eine solche SWT-Systembibliothek für ein Betriebssystem nicht, so kann das SWT und unter Verwendung dieses erstellte Programme – auch bei Existenz einer Java-Laufzeitumgebung für dieses Betriebssystem – nicht genutzt werden. In der Praxis stellt dieses allerdings kein Problem dar, da es für alle gängigen Betriebssysteme eine Implementierung dieser benötigten Systembibliothek gibt. Eine Auflistung der zum Zeitpunkt der Fertigstellung dieser Diplomarbeit für die verschiedenen Betriebssysteme und Plattformen verfügbaren SWT-Implementationen und Systembibliotheken findet sich im Anhang A.

7.3.2 JDOM

An mehreren Stellen innerhalb der erstellten Software war es notwendig, XML-formatierte Dokumente zu lesen oder zu schreiben. Die Verwaltung bzw. Verarbeitung dieser Dokumente wurde dabei vorgenommen mittels der Open-Source Java - Bibliothek *JDOM* (vgl. [HM]). Dabei steht JDOM aus markenrechtlichen Gründen nicht für ein Akronym, sondern stellt den Namen dieser Bibliothek dar.

Die von JDOM benutzte Modellstruktur für die Verwaltung des XML-Dokumentes ist – wie sich anhand des Namens bereits vermuten lässt – ähnlich der Struktur, die auch dem Document Object Model (DOM) des World Wide Web Consortiums zugrunde liegt (für dessen genaue Struktur vergleiche [Word]). Das Modell für das XML-Dokument beispielsweise wird auch bei JDOM in Baumform dargestellt. Dennoch unterscheidet diese sich in einigen wesentlichen Punkten. So wurde das Document Object Model plattform- und programmiersprachenneutral entwickelt, während JDOM speziell für Java und unter Ausnutzung von Java-spezifischen Vorgehensweisen und Möglichkeiten entwickelt wurde. Ein Hauptaugenmerk lag dabei darauf, die komplexen XML-Strukturen der verarbeiteten Dokumente wo möglich zu verbergen bzw. auf einfachere Strukturen abzubilden. So ist es z. B. möglich, XML-Elemente durch eine entsprechende, von der JDOM-Bibliothek bereitgestellte Klasse zu repräsentieren, und Operationen wie z. B. das Hinzufügen eines neuen Child-Elemente für ein XML-Element über einen einfachen Methodenaufruf am Parent-Element vorzunehmen. Eine tiefergehende Einführung in die Arbeitsweise mit JDOM bzw. in die Unterschiede und Gemeinsamkeiten zwischen JDOM und dem Document Object Model würde hier zu weit führen, statt dessen sei verwiesen auf [Hun02].

7.3.3 WS4D

Wie bereits in Abschnitt 6.5 kurz angesprochen sollte die Realisierung der von der Software bereitgestellten Web Services mittels des *Web Services for Devices (WS4D) J2ME Stacks* erfolgen. Entwickelt wurde dieser Stack im Rahmen einer Kooperation zwischen Partnern aus der Wissenschaft, unterstützt durch Partner aus der Industrie, der *Web Services for Devices - Initiative*. Im einzelnen waren dieses die Universität Rostock und die Universität Dortmund (seit 1. November 2007 die Technische Universität Dortmund), sowie die Firma Materna. Hervorgegangen ist diese Kooperationsinitiative aus dem im Frühjahr 2006 ausgelaufenem europäischen Forschungsprojekt *SIRENA*, dessen Ziel es unter anderem war, eine Software-Infrastruktur für die Kommunikation und Interaktion eingebetteter Systeme zu schaffen. Dafür wurden im Rahmen dieses Projektes einige Implementierungen von frühen DPWS-Spezifikationen (vgl.

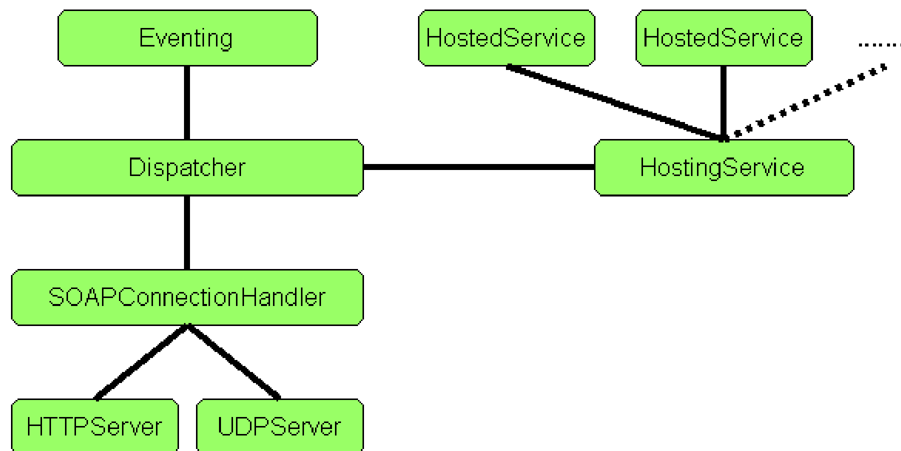


Abbildung 7.4: Software-Architektur des WS4D-J2ME - Stacks nach [ZBB+07]

3.3) entwickelt. Darauf aufsetzend formulierte die WS4D-Initiative das Ziel, eine Open-Source-Plattform zu entwickeln, die die DPWS-Verwendung in verschiedensten Umgebungen ermöglichen soll. Grundlegend dafür erfolgte die Entwicklung von drei Stacks: neben dem hier verwendeten und gleich noch näher betrachteten *WS4D-J2ME*-Stack waren dieses der auf gSOAP (vgl. [van07]) und somit C/C++ basiernde *WS4D-gSOAP*-Stack, und der auf dem Java-basierten SOAP-Werkzeug Axis2 (vgl. [Axi07]) aufbauende *WS4D-Axis2*-Stack. Da die beiden letzteren Stacks an dieser Stelle nicht benutzt wurden wird hier nicht näher auf sie eingegangen. Für Details sei verwiesen auf [WS4a] oder [ZBB+07].

WS4D-J2ME - Stack

Der *WS4D-J2ME*-Stack basiert auf der sogenannten *Connected Limited Device Configuration (CLDC)*. Bei dieser handelt es sich um die kleinstmögliche Konfiguration der Laufzeitumgebung für die *Java 2 Micro Edition (J2ME)*. Diese wurde speziell für den Einsatz auf Geräten mit eingeschränkten Ressourcen wie z. B. Mobiltelefonen oder PDAs entwickelt (vgl. [Sun07]). Durch die Benutzung der CLDC als Basis ist gewährleistet, dass dieser Stack mit allen anderen J2ME-Konfigurationen kompatibel ist.

Die Module des Stacks sind teilweise dargestellt in Abbildung 7.3. Wie dort zu erkennen bildet das `core`-Modul die Basis. Dieses stellt alle minimal notwendigen Funktionalitäten zur Verfügung, um ein DPWS-Device zu erstellen, das mittels *WS-Discovery* (vgl. Abschnitt 3.3.2) entdeckt und als *event source* (vgl. Abschnitt 3.3.1) benutzt werden kann. Die weiteren, in der Abbildung über dem `core`-Modul abgebildeten Module stellen *zusätzliche* Funktionalitäten zur Verfügung (neben den dort abgebildeten Modulen existieren noch weitere). So dient z. B. das `client`-Paket dazu, es zusätzlich zu ermöglichen, andere Devices oder Services im Subnetz zu finden und zu benutzen. Bei Verwendung dieses Paket verhält sich ein Service also wie ein *Client* (vgl. Kapitel 3.1). Auf die weiteren Module soll hier nicht explizit eingegangen werden, statt dessen sei verwiesen auf [ZBB+07].

Die Software-Architektur des WS4D-J2ME - Stacks ist dargestellt in Abbildung 7.4. Wie dort zu erkennen ist wird die Netzwerk-Kommunikation abgewickelt über die beiden Klassen `HTTPServer` und `UDPServer`. In diesen Klassen eingehende Nachrichten werden weitergegebenan die Klasse `SOAPConnectionHandler`, in welcher die Nachrichten verarbeitet werden. Ist diese Verarbeitung erfolgreich gewesen, so werden die im Rahmen der Verarbeitung zusammengestellten Daten an die Klasse `Dispatcher` weitergereicht, in welcher ermittelt wird, an welches Device oder an welchen Service die Nachricht weiterzuleiten ist. Ein konkretes Device ist dabei eine Instanz der Klasse `HostingService`, und ein konkreter Service eine Instanz der Klasse `HostedService`. Angelegt werden die Devices hier, indem zunächst eine (eigene) Klasse von der Klasse `HostingService` erbt, diese dabei gegebenenfalls um eigene Funktionen ergänzt,

und dann eine Instanz dieser Klasse erzeugt wird. Um einen Service zu diesem Device hinzuzufügen, muss zunächst eine (wiederum eigene) Klasse von der Klasse `HostedService` erben bzw. diese erweitern. Anschließend kann der Instanz der zuvor erstellten, von `HostingService` erbenden Klasse eine Instanz der von `HostedService` erbenden Klasse hinzugefügt werden. Das *WSDL* (vgl. 3.2.3) für derartige Services und die *Metadaten* der Devices werden dabei erst dann erzeugt, wenn ein *Client* eine Anfrage bezüglich dieser sendet.

Kapitel 8

Modell-Framework

In diesem Kapitel soll das entstandene *Modell-Framework* (vgl. Kapitel 7.1) erläutert werden. Dieses stellt die Elemente bereit, mittels denen die Simulations-Modelle erstellt werden. Dazu wird zunächst untersucht, was für ein Aufbau derartigen Modellen generell zugrunde liegt. Dieses umfasst insbesondere auch die benötigte Struktur der Modell-Komponenten und die Art, wie diese Komponenten untereinander in Verbindung stehen können bzw. was zu berücksichtigen ist, um entsprechende Verbindungen flexibel realisieren zu können. Aus diesen Erkenntnissen werden dann die für die Implementierung des Modell-Frameworks zu treffenden Überlegungen abgeleitet, und die konkrete Umsetzung dieser vorgestellt. Grundlage für alle diese hier angestellten Überlegungen ist dabei das in Kapitel 5 identifizierte System.

8.1 Modell-Strukturen

Hier soll zunächst der *Aufbau* bzw. die *Struktur* eines mittels des Modell-Frameworks erstellbaren Simulations-Modells näher betrachtet und unter Berücksichtigung der in Kapitel 5 gewonnenen Erkenntnisse erläutert werden. Dafür ist in Abbildung 8.1 die allgemeine Struktur eines (potentiellen) Simulations-Modells dargestellt. Wie dort ersichtlich ist, existiert ein Simulations-Modell innerhalb fest abgeschlossender Grenzen. Simulierte Einwirkungen auf dieses Simulations-Modell von außen treten nur an bestimmten, dedizierten *Modell-Komponenten* auf. Ebenso treten (simulierte) Auswirkungen des Modells auf seine Umwelt nur an bestimmten, dedizierten Komponenten auf. In *jeder* Modell-Komponente laufen außerdem ganz allgemein betrachtet bestimmte, komponentenspezifische Funktionen ab, die an dieser Stelle jedoch noch nicht näher betrachtet werden sollen. Ferner sind die Modell-Kompoenten unter Verwendung von *Verbindungen* miteinander kombiniert. Diese Verbindungen geben – bezogen auf das Real-System – an, welche Komponente die dort angesprochenen *Payloads* an welche Nachfolgekomponente weiterreicht. Die Richtung, in der Payloads dabei weitergereicht werden, ist bei einer einmal eingerichteten Verbindung unidirektional und unumkehrbar.

Ein solches Simulations-Modell lässt sich beschreiben als einen *Graph*, bei dem die *Knoten* die Rollen der Modell-Komponenten einnehmen (und somit in den Knoten die oben angesprochenen Prozesse ablaufen), und die *Kanten* die Verbindungen zwischen diesen Komponenten darstellen. Aufgrund der oben angesprochenen Voraussetzung, dass sich die „Transportrichtung“ in einer Verbindung (d. h. einer Kante) niemals umkehrt, kann hier sogar von einem *gerichteten Graphen* ausgegangen werden. Es ist dabei allerdings zu beachten, dass dieser gerichtete Graph nicht notwendigerweise kreisfrei sein muss. Betrachtet man das reale System, so ist durchaus vorstellbar, dass *Zyklen* existieren können, um z. B. eine Art von „Warteschleife“ zu realisieren, in der Payloads „kreisen“ können.

Nach diesen Überlegungen lassen sich nun zwei Teilaufgaben erkennen, die es für die Bereitstellung eines

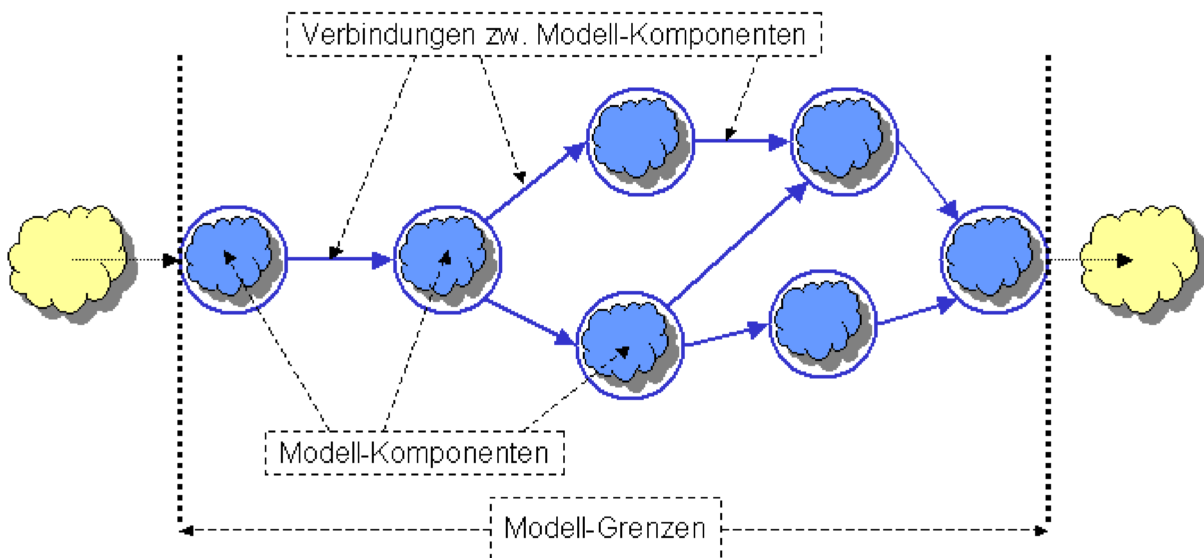


Abbildung 8.1: Allgemeine Struktur eines potentiellen Simulations-Modells

Modell-Frameworks für beliebig zusammenstellbare Simulations-Modelle zu lösen gilt. Dabei handelt es sich zum einen um das Bereitstellen der eigentlichen *Modell-Komponenten* mit jeweils ihren individuellen Eigenschaften (wie z. B. den zuvor angesprochenen Prozessen, die in ihnen ablaufen), und zum anderen um die Realisierung der *Verbindungen* dieser Komponenten untereinander. Letzteres schließt insbesondere die Frage mit ein, wie die Verbindungen zwischen den Komponenten zu verwalten sind, und wie das Weitergeben von Payloads mittels dieser Verbindungen dargestellt werden soll.

8.2 Verbindungen zwischen Komponenten

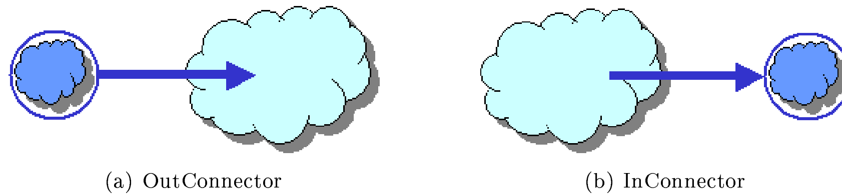
Hier wird nun auf die Realisierung der Verbindungen zwischen den einzelnen Modell-Komponenten eingegangen. Dazu sei in einem ersten Schritt daran erinnert, was eine Verbindung zweier Komponenten im Modell bezogen auf das zugrundeliegende Real-System bedeutet: eine solche Verbindung stellt dar, dass *Payloads* von der einen Komponente zu der anderen Komponente *weitergegeben* werden. Daraus folgt, dass die hier entwickelte Darstellungs-Struktur von Verbindungen zweierlei Funktion wird realisieren müssen: zum einen die bloße Indikation einer Verbindung (d. h. eine Modell-Komponente muss mittels einer Verbindung erkennen können, mit welcher anderen Komponente sie verbunden ist), und zum anderen den Übergang eines Payloads, das von einer Komponente an die nächste Komponente übergeben werden soll.

8.2.1 Theoretischer Ansatz

Die Grundidee bei der Realisierung der Verbindungen liegt darin, spezielle Elemente einzuführen, mittels denen die Verbindungen zwischen den einzelnen Komponenten vorgenommen werden, und die die Funktionalitäten bereitstellen, um Payloads weiterzureichen. Diese Verbindungs-Elemente (im weiteren mit *Connectoren* bezeichnet) sind den Modell-Komponenten, die sie miteinander verbinden, bekannt, und jede Komponente muss für sich selbst ihre Verbindungen bzw. ihre *Connectoren* verwalten.

In einem ersten Schritt soll nun eine weitere Aufteilung der Connectoren geschehen. Diese geschieht anhand der *Richtung*, in die eine Verbindung geht: diese kann *in die Komponente hinein* gehen, oder aber *von der Komponente ausgehen*. Dieses ist dargestellt in Abbildung 8.2. Dort wird die für die Connectoren zukünftig benutzte Terminologie benutzt: als *OutConnector* wird hier eine ausgehende Verbindung

bzw. das Element, das eine solche Verbindung realisiert, bezeichnet; *InConnector* bezeichnet äquivalent dazu eine eingehende Verbindung. Bezüglich des Transports von Payloads ist hier erkennbar, dass diese beiden Arten von Connectoren komplementäre Funktionen ausüben: über einen OutConnector *verlassen* Payloads ein Modell-Element, während sie über einen InConnector bei einem Modell-Element *ankommen*.



(a) OutConnector (b) InConnector

Abbildung 8.2: Unterscheidung von Verbindungen anhand ihrer Richtung

Jede Verbindung zweier Elemente lässt sich als Kombination eines OutConnectors mit einem InConnector beschreiben. Dabei „verlässt" der OutConnector das Element, das Payloads an ein Folgeelement weiterreichen muss, und trifft in einer Schnittstelle auf einen korrespondierenden InConnector, der auf das Element zeigt, an das das Payload weiterzureichen ist. In dieser Schnittstelle erfolgt die „Übergabe" des Payloads an das Nachfolge-Element. Dieses Prinzip ist dargestellt in Abbildung 8.3.

(a) Connector (b) In- und OutConnector

Abbildung 8.3: Transformation eines Connectors in je einen In- und OutConnector

Auf Grundlage der zuvor angeführten Überlegungen lässt sich nun das Prinzip erläutern, das hier der Realisierung der Verbindungen zugrundegelegt wurde. Dabei verwaltet jede der Modell-Komponenten des Modell-Frameworks eine Liste seiner InConnectoren und eine Liste seiner OutConnectoren. Jedem dieser Connectoren wiederum ist bekannt, mit welchem korrespondierenden Connector er verbunden ist. Außerdem ist jedem Connector bekannt, welchem Modell-Element er zugeordnet ist. Somit kann jede der Modell-Komponenten zu jeder Zeit ermitteln, wer seine Vorgänger und Nachfolger im Simulations-Modell sind.

Neben der Verwaltung der Verbindungs-Struktur dienen die Connectoren auch dem Weiterreichen von Payloads. Die Funktionen, die dafür von den Connectoren-Arten zur Verfügung gestellt werden müssen, ergänzen sich. So muss ein OutConnector die Möglichkeit bereitstellen, ein Payload von der angeschlossenen Modell-Komponente zu empfangen, und dann dem verbundenen InConnector die Information zur Verfügung stellen, dass an der Schnittstelle dieser beiden Connectoren ein Payload vorhanden ist. Versucht nun der InConnector dieses Payload zu übernehmen, so muss der OutConnector dieses übergeben, und wäre somit wieder bereit, ein neues Payload anzunehmen. Der InConnector wiederum muss seinem Besitzer (d. h. der entsprechenden Modell-Komponente) die Information zur Verfügung stellen, dass ein neues Payload eingetroffen ist, was entsprechene Reaktionen der Modell-Komponente auslösen kann.

8.2.2 Implementierung

Nun soll darauf eingegangen werden, wie die oben aufgestellten Anforderungen an die Connectoren bei der Implementierung umgesetzt wurden. Dazu werden die beiden Klassen, die die OutConnectoren und InConnectoren realisieren, betrachtet, und ihre Methoden erläutert.

OutConnector

Die Klasse `OutConnector` realisiert das oben beschriebene Element zum Darstellen einer ausgehenden Verbindung. Bei der Instanziierung dieser Klasse muss das Simulations-Element übergeben werden, das diesen Connector „besitzt“. Dieses Element wird in der lokalen Variablen `owner` verwaltet.

Bei den Methoden, die von diesem Element bereitgestellt werden, handelt es sich im einzelnen um:

- `getOwner()`: liefert das Simulations-Element, von dem aus dieser OutConnector ausgeht.
- `sendPayloadIn()`: mittels dieser Methode muss das verbundene Simulations-Element versuchen, ein Payload an diesen OutConnector zu übergeben. Dieser akzeptiert dieses Payload, wenn im Augenblick des Aufrufes dieser Methode kein Payload in diesem OutConnector vorhanden ist (Rückgabewert: `true`), oder akzeptiert es nicht, wenn noch ein Payload aus einem vorherigen Schritt vorhanden ist, dass erst an den verbundenen InConnector weiterzureichen ist. Dann wird entsprechend `false` zurückgegeben.
- `hasPayload()`: liefert `true`, wenn an diesem OutConnector aktuell ein Payload anliegt, oder `false`, wenn dieser OutConnector „leer“ ist.
- `getPayloadDelivery()`: diese Methode dient der Übernahme eines vorhandenen Payloads durch den verbundenen InConnector. Sollte die Methode `getPayloadDelivery()` aufgerufen werden, ohne dass an diesem OutConnector ein Payload anliegt, so liefert sie `false` zurück. Für den Fall, dass ein Payload anliegt und dieses korrekt übergeben werden konnte ist der Rückgabewert `true`.
- `connectTo(...)`: diese Methode dient dem Verbinden dieses OutConnectors mit seinem korrespondierendem InConnector. Dieser wird bei dem Methodenaufruf übergeben, und in der Variablen `connectedInConnector` hinterlegt. Ferner wird noch überprüft, ob diesem nun verbundenen InConnector bereits dieser OutConnector als Partner bekannt ist. Sollte dem nicht so sein wird die korrespondierende Methode `connectTo(...)` des InConnectors mit `this` als Parameter aufgerufen.
- `getConnectedInConnector()`: liefert den mit diesem OutConnector verbundenen InConnector.
- `isConnected()`: dient der Abfrage, ob dieser OutConnector bereits mit einem InConnector verbunden ist.
- `resetConnectorVariables()`: setzt den Wert, ob an diesem OutConnector gerade ein Payload anliegt, zurück, d. h. löscht ein eventuell vorhandenes Payload.
- `willAcceptPayloadDelievery()`: mittels eines Aufrufes dieser Methode kann festgestellt werden, ob der korrespondierende InConnector dieses OutConnectors aktuell ein Payload akzeptieren würde. Der OutConnector ruft dafür die entsprechende Methode des verbundenen InConnectors auf, und liefert den von dort erhaltenen Wert (`true` oder `false`) zurück.

Mittels dieser zuvor erläuterten Methoden sind Instanzen der Klasse `OutConnector` nun in der Lage, die oben an OutConnectoren aufgestellten Anforderungen zu erfüllen.

InConnector

Die Klasse `InConnector` ist ähnlich aufgebaut, wie die zuvor erläuterte Klasse `OutConnector`. Auch bei dieser Klasse hier muss bei der Instanziierung das Simulations-Element übergeben werden, mit dem dieser InConnector assoziiert ist.

Diese Klasse stellt die folgenden Methoden zur Verfügung:

- `getOwner()`: liefert das verbundene Simulations-Element zurück.

- `connectTo(...)`: verbindet diesen InConnector mit dem in dem Methodenaufruf übergebenen OutConnector. Diese Methode stellt das Gegenstück zu der namensgleichen Methode des OutConnectors dar, und arbeitet auf identische Art und Weise.
- `getConnectedOutConnector()`: liefert den verbundenen OutConnector zurück, bzw. `null`, falls noch kein verbundener OutConnector existiert.
- `isConnected()`: `true`, wenn dieser InConnector bereits mit einem OutConnector verbunden ist, oder aber `false`, wenn dem nicht so ist.
- `willAcceptPayloadDelievery()`: diese Methode bewirkt eine Anfrage an den `owner` dieses InConnectors, ob dieser aktuell ein Payload akzeptiert. Dessen Antwort wird an die aufrufende Instanz zurückgegeben. Die für diese Anfragen vorgesehene Methode des `owner` muss entsprechend von jedem Simulations-Element implementiert werden.

Weitere Methoden muss ein InConnector nicht zur Verfügung stellen, um seine geforderte Funktion zu gewährleisten. In erster Linie stellt ein derartiger InConnector für das ihn besitzende Simulations-Element eine Schnittstelle zum Zugriff auf den verbundenen OutConnector, und somit auf das verbundene Vorgänger-Element dar.

8.3 Grundlagen der Komponenten

In diesem Abschnitt sollen nun die *Modell-Komponenten* selbst näher betrachtet werden. Dazu wird zunächst eine Einteilung der Modell-Komponenten in verschiedene Gruppen vorgenommen, und anschließend dargestellt, inwieweit es trotz dieser Differenzierung der Komponenten gemeinsame Eigenschaften oder Funktionen gibt, die allen Elementen oder zumindestens den Elementen der gleichen Gruppe anhaften. Dieses wird dann genutzt, um eine geeignete Struktur für die konkrete Implementierung zu entwickeln.

8.3.1 Arten von Komponenten

Hier sei zunächst noch einmal erinnert an Abbildung 8.1. In dieser ist zu erkennen, dass es Modell-Komponenten gibt, von denen innerhalb des Modells nur Verbindungen ausgehen, andere, in die nur Verbindungen hineingehen, und schließlich solche, die sowohl ein- wie auch ausgehende Verbindungen haben. Dieses wird hier nun als Unterscheidungskriterium für die verschiedenen Arten von Modell-Komponenten bzw. als Kriterium für die Einteilung dieser in verschiedene Gruppen genutzt. Diese Einteilung wurde bereits bei der Vorstellung des Real-Systems in Abschnitt 5.2.2 angesprochen.

Komponenten mit ausgehenden Verbindungen

Komponenten mit ausgehenden Verbindungen stehen am Beginn einer jeden *Transportkette* eines Simulations-Modells. Ihre Aufgabe besteht darin, eine Schnittstelle nach außen zu simulieren, über die *Payloads* in das Simulationsmodell hereingegeben werden. Da von diesen Elementen aus nur Verbindungen ausgehen, aber keine in diese hineingehen, müssen derartige Elemente entsprechend nur ihre *OutConnectoren* verwalten. Bezeichnet werden die Elemente dieser Gruppe im weiteren Verlauf mit *Entry-Elemente*. Diese Bezeichnung wurde deswegen so gewählt, weil Payloads ein Simulations-System über diese Elemente gewissermaßen „betreten".

Komponenten mit eingehenden Verbindungen

Diese Gruppe von Komponenten stellt das Gegenstück zu der zuvor beschriebenen Gruppe dar. Ein Element mit nur eingehenden Verbindungen steht im Simulations-Modell am Ende einer jeden Trans-

portkette. Es dient zur Simulation einer Schnittstelle nach außen, d. h. Payloads, die zu diesem Element geliefert werden, verlassen über dieses Element das Simulations-Modell (wonach sie nicht mehr betrachtet werden müssen). Dementsprechend wird als Oberbegriff für derartige Elemente im Folgenden die Bezeichnung *Exit-Elemente* benutzt. Da diese Exit-Elemente lediglich eingehende Verbindungen haben, müssen sie auch nur eine Liste ihrer *InConnectoren* verwalten.

Komponenten mit ein- und ausgehenden Verbindungen

Komponenten aus dieser Gruppe stehen innerhalb einer Transportkette im Simulations-Modell. Dementsprechend benötigen sie sowohl eingehende *InConnectoren*, mittels denen sie Payloads „empfangen“ können, als auch ausgehende *OutConnectoren*, um die Payloads wieder weiterzureichen. Diese Elemente müssen also zwei Listen mit diesen verschiedenen Connectoren verwalten. Der hier für diese Elemente benutzte Sammelbegriff ist – da diese Elemente eben *innerhalb* einer Transportkette benutzt werden – *Inside-Elemente*.

8.3.2 Einteilung der Komponenten

Hier sollen die in Abschnitt 5.2.2 identifizierten Komponenten in die zuvor erläuterten Gruppen eingeteilt werden. Diese Einteilung ist dargestellt in Tabelle 8.1. Warum die Komponenten derart eingeteilt wurden, leitet sich aus der in Abschnitt 5.2.2 erfolgten Beschreibung der Funktionen der Komponenten ab.

Komponente	Gruppe
InPort	Entry-Element
OutPort	Exit-Element
Conveyer	Inside-Element
Gate	Inside-Element
RobotGrabber	Inside-Element
WorkStation	Inside-Element

Tabelle 8.1: Einteilung der Komponenten des Real-Systems in Entry-, Exit- und Inside-Elements

Aus Tabelle 8.1 ist ersichtlich, dass es bei dem zugrundegelegten Realsystem nur jeweils ein Element in der Gruppe Entry-Element bzw. Exit-Element gibt. Dennoch ist es sinnvoll, diese Einteilung vorzunehmen, da dieses das spätere Erweitern des Modell-Frameworks um weitere Entry- oder Exit-Elemente vereinfacht bzw. für diese den Rahmen definiert, der bei der Implementierung solcher Elemente zu beachten ist.

8.3.3 Gemeinsame Eigenschaften aller Komponenten

Auch, wenn sich die Komponenten gemäß ihrer Verbindungen in drei disjunkte Gruppen von Komponenten unterteilen lassen, haben alle Komponenten dennoch gewisse *gemeinsame* Attribute bzw. Funktionalitäten. Dabei können sich die Vorgehensweisen bei der Bereitstellung dieser Funktionen sehr wohl von Element zu Element unterscheiden. Diese von allen Simulations-Elementen zu leistenden Funktionalitäten bzw. allen Simulations-Elementen anhaftenden Attribute sind im folgenden aufgelistet:

- *Verwaltungs-Daten und Funktionen*: dieses umfasst z. B. Variablen zur Verwaltung oder Methoden zum Setzen des Element-Namens, des Zustandes des Elementes, der Position, an der das Element bei der Visualisierung dargestellt werden soll, usw.
- *Visualisierung*: jedes der Simulations-Elemente muss in der Lage sein, sich selbst entsprechend graphisch darzustellen. Dieses schließt auch das Animieren der Elemente mit ein.

- *Durchführung von Simulations-Schritten*: jedes Element muss selbstständig seine lokalen Simulations-Schritte durchführen können, und für die externe Simulations-Kontrolle die Funktionen bereitstellen, um diese Schritte zu veranlassen.
- *Protokoll-Daten*: jedes Element muss in der Lage sein, selbst seine zu protollierenden Daten zusammenzustellen, und diese auf Anfrage hin nach außen zur Verfügung zu stellen. Ferner muss jedes Element selbst verwalten, ob überhaupt Protokolldaten für dieses Element zu schreiben sind.
- *Benutzer-Interaktion*: jedes der Elemente muss dem Benutzer die Funktionen zum Konfigurieren dieses Elements und zum Interagieren mit diesem Element zur Verfügung stellen. Praktisch gesprochen bedeutet dieses, dass jedes der Elemente dafür geeignete Dialoge bereitstellen muss.
- *Speichern und Laden*: jedes Element muss selbstständig seine zu speichernden Konfigurations-Daten zusammenzustellen, und sich umgekehrt beim Laden dieser Daten selbst wieder entsprechend konfigurieren.
- *Fehler-Verwaltung*: auftretende Fehler müssen vom jedem Simulations-Element erkannt und weitergegeben werden können.
- *Web Service - Verwaltung*: jedes der Elemente muss selbstständig seine Web Services verwalten können. Dieses schließt das Ein- oder Ausschalten dieser Services mit ein.

Für die Implementierung bedeutet dieses, dass derartige Funktionen grundsätzlich bereits in einer allen Elementen zugrundeliegenden Basisklasse realisiert werden können. Ebenso können Variablen zur Verwaltung von Attributen bereits dort angelegt werden. Sollte absehbar sein, dass die konkrete Realisierung dieser Funktionalität stark von dem konkreten Simulations-Element abhängt, so kann die entsprechende Funktion zumindest bereits *abstrakt* in dieser Basisklasse vorgegeben werden.

8.3.4 Gemeinsame Eigenschaften innerhalb der Komponenten-Gruppen

Gemeinsame Eigenschaften von Komponenten aus einer der oben angeführten Gruppen beziehen sich auf die Verwaltung und Bereitstellung der Connectoren, die von dem jeweiligen Element benutzt werden. So müssen Komponenten aus der Gruppe der Entry-Elemente Methoden bereitstellen, um sich selbst mittels OutConnectoren mit Nachfolge-Elementen zu verbinden, und diese OutConnectoren verwalten. Dieses Verwalten umfasst dabei auch z. B. das Löschen von OutConnectoren, d. h. das Entfernen von Verbindungen zu ehemaligen Nachfolgern. Exit-Elemente müssen ähnliche Funktionalitäten bereitstellen, um ihre InConnectoren zu verwalten, und sich über diese InConnectoren mit Vorgängern innerhalb des Simulations-Graphen verbinden zu können. Inside-Elemente schließlich müssen sowohl Funktionen zur Verwaltung von bzw. zum Arbeiten mit In- wie auch mit OutConnectoren bereitstellen.

Weitere gemeinsame Eigenschaften bzw. gemeinsame Funktionen, die allen Elementen einer Komponenten-Gruppe anhaften, sind nicht explizit vorhanden.

8.3.5 Resultierende Struktur der Komponenten

Aus den oben beschriebenen gemeinsamen Eigenschaften aller Komponenten bzw. den gemeinsamen Eigenschaften innerhalb der Gruppen *Entry-Element*, *Exit-Element* und *Inside-Element* lässt sich nun eine Struktur ableiten. Diese ist dargestellt in Abbildung 8.4. Dort ist ersichtlich, wie sich die konkreten Modell-Komponenten auf die Gruppen, in die sie eingeteilt sind, und auf eine gemeinsame Basis zurückführen lassen. Auf dieser identifizierten Struktur setzt die Implementierung der Simulations-Elemente auf. Dabei ist es zu beachten, dass die konkreten Modell-Komponenten die sind, die in Tabelle 8.1 genannt wurden, bzw. die, die in Abbildung 8.4 nicht noch weiter verfeinert werden. In einem Simulations-Modell darf es beispielsweise keine Komponente des Typs „Inside-Element“ geben können.

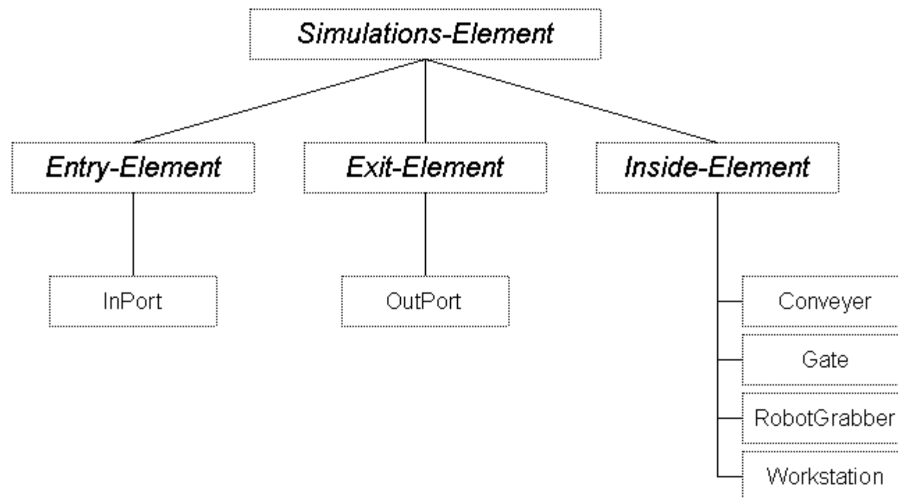


Abbildung 8.4: Struktur der Modell-Komponenten

8.3.6 Aufteilung der Simulations-Schritte

Vorbereitend auf die konkrete Implementierung der Komponenten soll hier nun noch einmal ein durchzuführender *Simulations-Schritt* bzw. die im Rahmen eines solchen durchzuführenden Aktionen betrachtet werden. Dabei fällt auf, dass es bezogen auf die zu transportierenden *Payloads* grundsätzlich drei verschiedene, auszuführende Aktionen gibt:

- *Annehmen von Payloads*: hier ist zu überprüfen, ob ein Element im aktuellen Simulations-Schritt von einem Vorgänger-Element in der Transportkette ein Payloads zu übernehmen hat. Sollte dem so sein ist zu überprüfen, ob dieses angenommen werden kann. Beispielsweise wäre eine Annahme nicht möglich, wenn damit die Kapazitätsgrenze des Elements überschritten werden würde. Ist die Annahme des Payloads hingegen zulässig, so muss das Element dieses von seinem Vorgänger-Element übernehmen.

- *Abarbeiten von Payloads*: dieses umfasst Aktionen, die von Komponente zu Komponente verschieden sind, und die von diesen ausgeführt werden müssen, um ihr komponentenspezifisches Verhalten bzw. ihre komponentenspezifische Funktion bereitstellen zu können. Bei einem Förderband wäre dieses z. B. das Weiterbefördern von Payloads um eine bestimmte Strecke in jedem Simulations-Schritt.

- *Weitergeben von Payloads*: nachdem eine Komponente für ein Payload alle für dieses Payload auszuführenden komponentenspezifischen Aktionen durchgeführt hat, d. h. dieses Payload gewissermaßen „abgearbeitet“ ist, muss dieses Payload an die Nachfolge-Komponente in der Transportkette weitergereicht werden.

Bei den oben beschriebenen Aktionen ist zu beachten, dass nicht notwendigerweise jede Modell-Komponente alle diese Aktionen durchführen können muss. Beispielsweise müssen *EntryElements* keine Payloads von Vorgänger-Elmenten annehmen können, oder *ExitElements* keine Payloads an Nachfolger-Elemente weiterreichen können.

Ferner lassen sich die beschriebenen Aktionen noch zeitlich betrachten. Dabei entspricht der zeitliche Ablauf dieser Aktionen innerhalb eines Simulations-Schrittes der Reihenfolge, in der diese Aktionen oben aufgeführt wurden: betrachtet man ein konkretes Payload, so muss dieses zunächst von einer Modell-Komponente *angenommen* werden, bevor es von dieser *abgearbeitet* werden kann. Erst nachdem dieser Vorgang abgeschlossen ist, muss es möglicherweise *weitergegeben* werden. Ein Simulations-Schritt, den eine

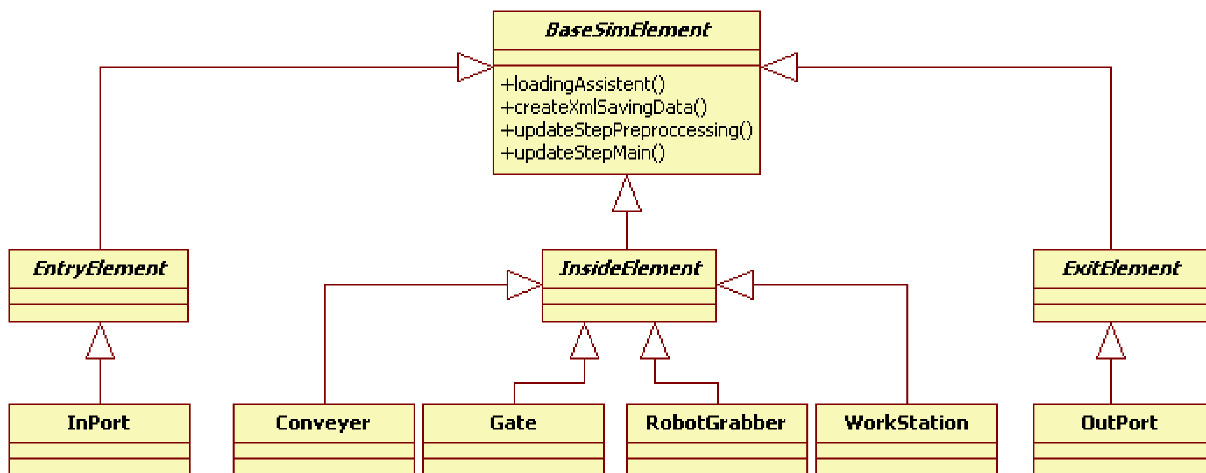


Abbildung 8.5: Klassenhierarchie der Simulations-Elemente

Komponente durchzuführen hat, lässt sich also in diese drei Teil-Phasen aufteilen. Desweiteren kollidieren diese Teil-Phasen auch nicht miteinander. Es ist möglich, in einem Simulations-Modell zur Einleitung eines Simulations-Schrittes zunächst alle vorhandenen Elemente überprüfen zu lassen, ob neue Payloads anzunehmen sind, und diese gegebenenfalls anzunehmen. Daran anschließend können alle Komponenten in einer Art Haupt-Schritt die notwendigen Aktionen durchführen, die ihre Komponentenfunktion ausmachen, und schließlich zur Fertigstellung des Simulations-Schrittes überprüfen, ob es nun nach der Abarbeitung des Simulations-Schrittes Payloads gibt, die an eine Nachfolge-Komponente weiterzuleiten sind, was dann noch durchzuführen wäre. Dieses würde den Schritt abschließen, und der nächste Schritt könnte eingeleitet werden.

8.4 Implementierung der Komponenten

Hier soll nun, nachdem zuvor die Grundlagen hinter den zu implementierenden Modell-Komponenten erläutert worden sind, auf die Implementierung dieser eingegangen werden. Dazu wird zunächst ein Überblick über die dafür benutzte Klassenhierarchie gegeben, um bei den folgenden Beschreibungen der weiteren Elemente einordnen zu können, welche Funktion die jeweils implementierten Klassen bereitstellen bzw. welche Funktionen sie zu ihren geerbten Funktionen hinzufügen. Daran anschließend wird das *Basis-Element* ausführlich dargestellt, da dieses die Grundlage für alle weiteren Simulations-Elemente darstellt. Deswegen stellt dieses bereits viele der allen Elementen anhaftenden bzw. von allen Elementen benutzten Funktionen bereit. Nachdem dieses Basis-Element erläutert worden ist wird auf die weiteren, aus der Klasse dieses Elementes abgeleiteten Simulations-Elemente eingegangen, wobei bei diesen lediglich kurz zusätzlich hinzugefügte oder durch diese modifizierte Funktionalitäten erläutert werden sollen.

8.4.1 Klassenhierarchie der Implementierung

Die den Simulations-Elementen zugrundeliegende Hierarchie ist dargestellt in Abbildung 8.5. Wie dort zu erkennen ist stellt die abstrakte Klasse `BaseSimElement` die Grundlage für alle weiteren Simulations-Elemente dar. In dieser werden die Methoden und Attribute angelegt, die alle Simulations-Elemente besitzen.

Aus dieser Super-Klasse werden auf einer ersten Stufe die (ebenfalls noch abstrakten) Klassen `EntryElement`, `ExitElement` und `InsideElement` abgeleitet, welche weitere spezifische Eigenschaften gemäß der in Abschnitt 8.3.4 vorgestellten Gruppen von Simulations-Elementen hinzufügen. Auf diesen Klassen

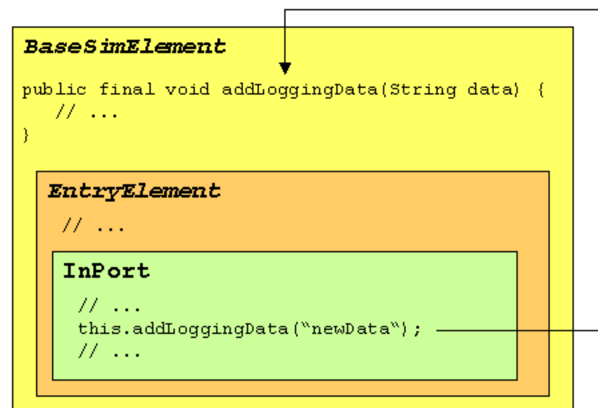


Abbildung 8.6: Hinzufügen von zu protokollierenden Daten

aufbauend werden schließlich in einer letzten Stufe die konkreten Klassen `InPort`, `OutPort`, `RobotGrabber`, `WorkStation`, `Gate` und `Conveyer` für die instanziierbaren Simulations-Elemente implementiert.

8.4.2 BaseSimElement

Die abstrakte Klasse `BaseSimElement` stellt die Basis für alle weiteren Klassen dar, mittels denen die Simulations-Elemente implementiert werden. Dazu stellt diese Klasse eine Vielzahl von bereits fertig implementierten Methoden zur Verfügung, die in dieser Form auch in den von ihr erbenden Klassen Verwendung finden, und gibt außerdem zahlreiche abstrakte Methoden vor, die von den von ihr erbenden konkreten Klassen implementiert werden müssen.

Die Funktionen, die dabei in dieser Klasse entweder konkret implementiert oder abstrakt zur Implementierung vorgegeben werden lassen sich dabei in verschiedene Gruppen unterteilen.

Verwaltung des Simulations-Elementes

Bei diesem Teilbereich handelt es sich um denjenigen, der den überwiegenden Anteil der hier zur Verfügung gestellten Funktionalitäten ausmacht, da diese Funktionen bei allen Simulations-Elementen auf die gleiche Art genutzt werden werden. Die zur Verfügung gestellten Funktionen umfassen konkret implementierte Methoden zum Setzen oder Abfragen von Verwaltungsdaten wie z. B. den Namen des Elementes.

Außerdem umfasst dieses auch die Verwaltung von Variablen, mit denen zur Laufzeit einer Simulation in diesem Element gearbeitet wird.

Visualisierung

Bei den Funktionen, die bezüglich der Visualisierung von dieser Klasse zur Verfügung gestellt bzw. vorgegeben werden sind zwei verschiedene Teil zu unterscheiden. Einerseits wird in dieser Klasse bereits konkret eine *grafische Repräsentierung* für das entsprechende Element in Form eines SWT-`CLabel` vorgegeben, welches über die öffentliche Methode `getGraphicalRepresentation()` zu referenzieren ist. Anderseits fallen hierunter die Vorbereitungen der Animationen, die jedes der konkreten Simulations-Elemente für sich selbst durchführen muss. Dafür ist bereits in dieser Klasse die abstrakte Methode `animateYourself(...)` vorgesehen, die jedes der von dieser Klasse erbenden Simulations-Elemente implementieren muss.

Durchführung von Simulations-Schritten

Die Art der Durchführung der Simulations-Schritte wird bereits in dieser Klasse festgelegt. Diese Festlegung erfolgt gemäß der in Abschnitt 8.3.6 vorgenommenen Dreiteilung eines jeden Simulations-Schrittes. Dafür werden hier die Methode `updateStepPreprocessing()`, `updateStepMain()` und `updateStepPostprocessing()` eingeführt, welche jedes der von dieser Klasse erbenden Elemente um seine eigene in dem jeweiligen Teilschritt benötigte Funktionalität *erweitern* kann. Da ein `BaseSimElement` noch keine wirklichen element-spezifischen Aktionen im Rahmen eines Simulations-Schrittes durchführen muss, werden auf dieser Ebene lediglich einige Aktualisierungen von Verwaltungs-Variablen wie z. B. der Anzahl der absolvierten Simulations-Schritte durchgeführt. Ferner werden einige bereits auf dieser Ebene bekannte Informationen (wie eben z. B. die Anzahl von Simulations-Schritten, während der dieses Element aktiv war) zu den zu protokollierenden Daten hinzugefügt.

Protokollierungs-Funktionen

Protokollierungs-Daten zusammenzustellen und nach außen hin bereitzustellen ist eine der zentralen Anforderungen, die jedes konkrete Simulations-Element erfüllen muss. Dieses ist daher bereits in dieser Klasse implementiert: die für einen Schritt eines Elements zu protokollierenden Daten werden in einem `Vector` verwaltet, der mittels einer Abfrage der öffentlichen, finalen Methode `getLoggingData()` ausgelesen werden kann. Ferner existiert in dieser Klasse die finale Methode `addLoggingData(...)`, die von jedem von dieser Klasse erbenden Element benutzt werden kann, um auf seiner Ebene angefallene, zu protokollierende Daten mit hinzuzufügen. Diese Struktur ist in Abbildung 8.6 dargestellt.

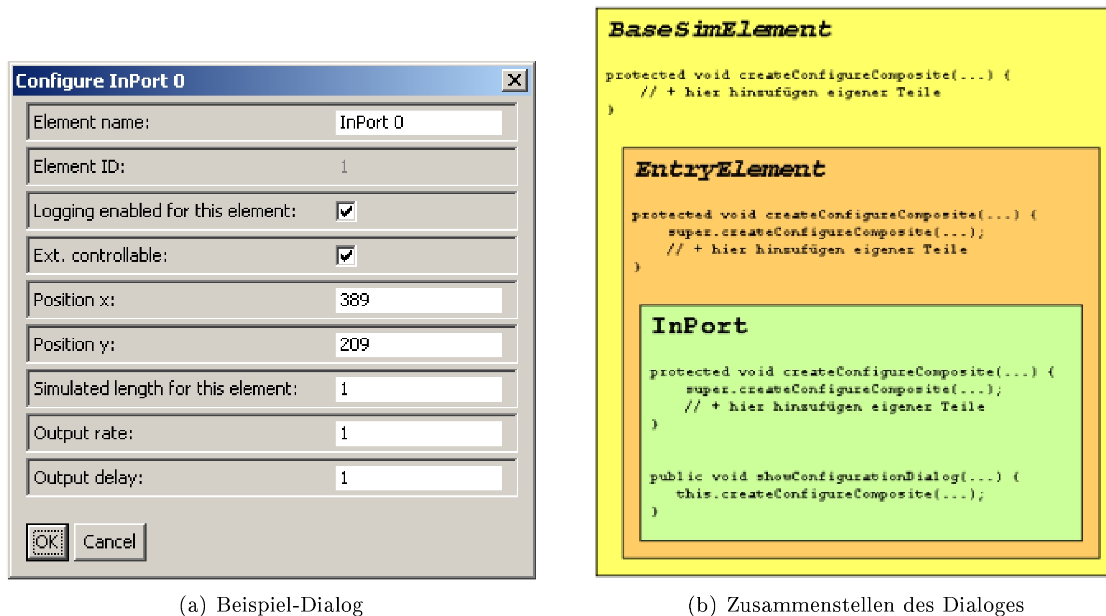
Interaktion mit dem Benutzer

Interaktion mit dem Benutzer kann für ein Simulations-Element an mehreren Stellen bzw. in mehreren Situationen notwendig sein. So soll es möglich sein, mittels eines Rechtsklicks auf ein Simulations-Element einer laufenden Simulation für dieses ein Kontext-Menü zu öffnen, mittels dem verschiedene Einstellungen verändert werden können. Dafür wird zunächst in dieser Basis-Klasse die Instanz dieses zu öffnenden Popup-Menüs angelegt, und für erbende Klassen über die geschützte Methode `getPopupMenuInstance()` zugreifbar gemacht. Desweiteren exisitiert in dieser Klasse die Methode `createPopupMenuItems()`, mit der bereits hier erste Menüpunkte zu diesem Menü hinzugefügt werden, und die von den erbenden Sub-Klassen entsprechend um eigene Menüpunkte ergänzt werden kann. Für die Anzeige dieses Menüs wird bereits hier die abstrakte Methode `showPopupMenu()` eingeführt, die in konkreten Klassen implementiert werden muss, und aus welcher dann der Aufruf der Methode zum Erstellen des Menüs erfolgen muss.

Ferner muss jedes Simulations-Element einen Konfigurationsdialog bereitstellen, mittels dem der Benutzer die die Funktion des Simulations-Element bestimmenden Parameter setzen kann. Da auf jeder Vererbungs-Stufe potentiell weitere einstellbare Parameter hinzukommen, muss bereits auf dieser Ebene mit dem Zusammenstellen dieses Dialoges begonnen werden, und erbende Klassen müssen diesen Dialog entsprechend erweitern. Dafür stellt die Klasse `BaseSimElement` die geschützte Methode `createConfigureComposite` zur Verfügung. Sub-Klassen des BaseSimElement können diese durch Hinzufügen von Komponenten zur Konfiguration eigener Parameter erweitern. Außerdem wird dafür bereits in dieser Klasse die abstrakte Methode `showConfigurationDialog(...)` definiert, die von den konkreten Sub-Klassen implementiert werden muss, und aus welcher dann ein Aufruf der zuvor beschriebenen Methode zum Zusammenstellen des Dialoges erfolgen muss. Eine beispielhafte Konfigurations-Oberfläche und die dazugehörige Aufruf-Sequenz beim Zusammenstellen der Konfigurations-Komponente sind dargestellt in Abbildung 8.7.

Laden und Speichern

Durch die Vererbungs-Struktur werden auf jeder Implementierungs-Ebene weitere Attribute hinzugefügt, die beim Laden eines Elements gesetzt bzw. beim Speichern gesichert werden müssen. Die Funktionen



(a) Beispiel-Dialog

(b) Zusammenstellen des Dialoges

Abbildung 8.7: Beispiel: Konfigurations-Dialog und Ablauf beim Zusammenstellen dieses Dialoges

zur Verwaltung dieser Daten müssen daher bereits im `BaseSimElement` bereitgestellt werden. Dafür exisitiert die Methode `addToXmlSavingData(...)`, mittels welcher zu speichernde Daten einer Sub-Klasse der in dieser Super-Klasse verwalteten Datenstruktur hinzugefügt werden können. Dabei besteht ein Datensatz immer aus einem identifizierenden *Schlüssel*, und dem eigentlichen *Wert*, der gespeichert werden soll. Ferner existiert die Methode `createXmlSavingData()`, die beim Anfordern der zu speichernden Daten aus der entsprechenden Sub-Klasse aufgerufen wird, und mittels der dann die zu speichernden Datensätz zusammengestellt werden. Für den Abruf der zu speichernden Daten von außen wird bereits in dieser Klasse die abstrakte Methode `getSavingData()` vorgegeben, die konkrete Klassen entsprechend zu implementieren haben.

Das Vorgehen zum Laden eines Simulations-Elements funktioniert ähnlich: bereits in der Klasse `BaseSimElement` existiert dafür die Methode `loadingAssistent(...)`, die einen String bestehend aus einem Daten-Schlüssel und dem eigentlichen Wert als Eingabe bekommt, dieses auswertet, und den entsprechenden Parameter dieser Klasse auf diesen Wert setzt. Von `BaseSimElement` erbende Klassen müssen diese Methode übernehmen und erweitern, um eigene Schlüssel und dazugehörige Werte auslesen zu können.

Fehler-Verwaltung

Im Rahmen eines Simulations-Ablaufes können innerhalb eines Simulations-Elementes an den verschiedensten Stellen *Fehler* entstehen. Dafür ist es nötig, dass jedem dieser Elemente die Möglichkeit zur Verfügung steht, diese Fehler zu verwalten und weiterzumelden. Beides sind Funktionalitäten, die bereits in dieser Basis-Klasse fertig implementiert sind, und somit von allen Sub-Klassen benutzt werden können. Die Verwaltung von Fehlern erfolgt dabei mittels sogenannter *ErrorInformationObjects* (vgl. Abschnitt 9.2.10). Ein derartiges Fehler-Objekt wird aus einer Sub-Klasse mittels der hier bereitgestellten finalen Methode `createError(...)` erzeugt. Nach außen verfügbar gemacht wird dieses Objekt – z. B., um es beim Schreiben des Protokolls mit auswerten zu können – über die hier bereitgestellte Methode `getErrorObject()`.

Verwaltung der Web Services

Jedes der Simulations-Elemente soll einen *Web Service* bereitstellen. Dafür wird in dieser Basis-Klasse eine Datenstruktur verwaltet, in der alle Devices, die von einem konkreten, von dieser Klasse abgeleiteten Simulations-Element zur Verfügung gestellt werden, verzeichnet sind. Daneben wird eine abstrakte Methode mit der Bezeichnung `startDpwsServices()` vorgegeben, die von erbenden Klassen mit der entsprechenden Funktionalität zum Erstellen und Starten eines entsprechenden Web Service implementiert werden muss. Für die Details bezüglich der hier angelegten und bereitgestellten Web Services sei verwiesen auf Kapitel 8.5. Konkrete implementiert ist im `BaseSimElement` die Methode `stopDpwsServices()`, mittels der jedes einzelne Device des Simulations-Elementes gestoppt wird.

8.4.3 EntryElement

Die abstrakte Klasse `EntryElement` dient der Erweiterung des `BaseSimElement` um die Funktionalitäten, die von Simulations-Elementen benötigt werden, die am Anfang einer Transportkette innerhalb eines Simulations-Modells stehen (vgl. Abschnitt 8.3.1). Dafür hinzuzufügende Funktionalitäten umfassen Strukturen zur Verwaltung von *ausgehenden Verbindungen*, d. h. von Instanzen der Klasse `OutConnector`. Um dieses bereitzustellen wird hier eine Datenstruktur eingeführt, in der die hier erzeugten OutConnectoren eingetragen werden. Ferner werden Methoden bereitgestellt, mit denen es möglich ist, *ausgehende* Verbindungen zu anderen Simulations-Elementen hinzuzufügen oder bestehende ausgehende Verbindungen so wieder abzubauen, dass das zuvor verbundene Element über diesen Vorgang informiert wird. Diese Methoden erstellen dabei die benötigten neuen Instanzen des Typs `OutConnector` und fügen diese mit in die zuvor genannte Datenstruktur ein, bzw. entfernen beim Abbauen einer Verbindung entsprechend den OutConnector aus dieser. Die Bezeichner dieser Methoden lauten dabei `connectElementToNewSuccessor(...)` bzw. `removeConnectionToSuccessor(...)`.

Außerdem werden die im `BaseSimElement` eingeführten Methode zum Speichern der Daten eines Simulations-Elementes (`createXmlSavingData()`) bzw. umgekehrt zum Laden (`loadingAssistent(...)`) so erweitert, dass die auf dieser Ebene mit hinzugekommenen Informationen über verbundene Nachfolge-Elemente dort berücksichtigt werden. Desweiteren wird eine Methode hinzugefügt, mittels der beim Laden eines Simulations-Elementes aus den gespeicherten Daten die konkreten Verbindungen erzeugt werden. Schließlich werden noch die von der Basis-Klasse abstrakt vorgegebenen Methoden zum Abfragen aller In- bzw. OutConnectoren eines Simulations-Elementes konkret implementiert; bei der Abfrage aller OutConnectoren werden die hier nun verwalteten OutConnectoren zurückgegeben, die Abfrage der vorhandenen InConnectoren liefert (da für ein EntryElement keine Vorgänger im Simulations-Modell existieren können) konstant `null`.

8.4.4 ExitElement

Die abstrakte Klasse `ExitElement` stellt das Gegenstück zu der zuvor beschriebenen Klasse `EntryElement` dar. Diese Klasse erweitert das `BaseSimElement` um die Funktionalitäten, die notwendig sind, um ein Simulations-Element am Ende einer Transportkette im Simulations-System positionieren zu können. Dementsprechend wird dieses `BaseSimelement` hier um Funktionalitäten zum Verwalten und Verarbeiten von Instanzen der Klasse `InConnector` erweitert. Die Vorgehensweisen bzw. Methoden zum Verwalten der Connectoren entsprechen dabei grundsätzlich denen, die schon zuvor bei bei der Klasse `EntryElement` beschrieben wurden. Allerdings ist der Umfang der hier zusätzlich hinzugefügten Methoden etwas geringer, da z. B. darauf verzichtet wurde, eine explizite Methode zum Initiieren des Verbindens dieses Elementes mit einem *Vorgänger*-Element hinzuzufügen. Das Verbinden zweier Elemente wird stattdessen *immer* von einem Vorgänger-Element initiiert, welches ein Element (das dabei keine Instanz einer Sub-Klasse des `EntryElement` sein darf) zu einem seiner Nachfolger macht. Dafür wiederum wird hier im `ExitElement` die Methode `addInConnector(...)` zur Verfügung gestellt. Diese erhält als Eingabe eine Instanz des Typs `OutConnector`, instanziiert einen korrespondierenden InConnector, verbindet diese beiden Connectoren,

und fügt der lokalen Datenstruktur zur Verwaltung der benutzten InConnectoren diesen neuen hinzu. Ebenfalls nicht notwendig wäre prinzipiell das *Abspeichern* der existierenden InConnectoren (da für das Erstellen aller Verbindungen beim Laden eines Modells Kenntnis über alle OutConnectoren ausreicht), was hier dennoch in Form einer Erweiterung der geerbten Methode `createXmlSavingData()` mit hinzugefügt wird. Dieses wurde implementiert, da dadurch die entstehenden XML-Dateien der Simulations-Modelle für den Benutzer beim direkten Betrachten verständlicher sind. Beim Laden werden diese abgespeicherten Informationen über die existierenden InConnectoren nicht verwendet. Eine letzte Funktion, die von der Klasse `ExitElement` noch explizit zur Verügung gestellt wird, dient zum kontrollierten *Abbauen* einer zuvor aufgebauten Verbindung zu einem Vorgänger-Element. Dazu wird die Methode `removeConnectionToPredecessor(...)` bereitgestellt. Bei einem Aufruf wird die Liste der InConnectoren durchlaufen, und dabei nach dem Element gesucht, was die Eingabe darstellte. Wird dieses gefunden wird die eingehende Verbindung von diesem kontrolliert abgebaut (d. h., dass dieses Element über den Abbau der Verbindung unterrichtet wird bzw. aufgefordert wird, die entsprechende ausgehende Verbindung ebenfalls zu löschen).

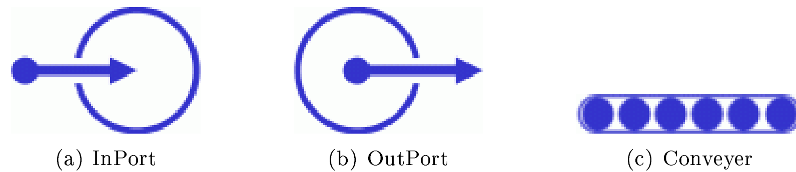
8.4.5 InsideElement

Die Klasse `InsideElement` stellt als Erweiterung der Klasse `BaseSimElement` die abstrakte Basis-Klasse für alle konkreten Klassen dar, die Elemente *innerhalb* einer jeden Transportkette eines Simulations-Modells (vgl. Abschnitt 8.3.1) implementieren. Dementsprechend muss mittels dieser Klasse die Funktionalität zum Verwalten von InConnectoren wie auch OutConnectoren bereitgestellt werden. Die Methoden und Funktionen, die dafür hier implementiert werden, entsprechen denen, die zuvor separat für die beiden Klassen `EntryElement` und `ExitElement` beschrieben wurden. Hier existieren Methoden zum expliziten Verbinden von Simulations-Elementen in Form konkreter Sub-Klassen des `InsideElement` mit dafür zulässigen Nachfolgern in einer Transportkette. Ebenso existieren derartige Methoden zum kontrollierten Abbauen von Verbindungen. Ferner wird auch hier wieder das Speichern und Laden der Connector-Instanzen (bzw. im Fall von InConnectoren wiederum nur das Speichern; vgl. 8.4.4) mittels Erweiterungen der Methoden `createXmlSavingData()` und `loadingAssistent(...)` implementiert.

8.4.6 InPort

Die Klasse `InPort` stellt die erste der hier nun folgenden konkreten Klassen dar, deren Instanzen ein Simulations-Modell bilden. Die Beschreibung der Funktion, die ein derartiges Element simulieren soll, findet sich in Abschnitt 5.2.2. Da es sich hier um ein Element handelt, dass am Anfang einer Transportkette eines Simulations-Modells steht, ist diese Klasse eine Sub-Klasse von `EntryElement`.

Die von Instanzen dieser Klasse geleistete Funktionalität ist, in regelmäßigen, festen Abständen ein neues *Payload* simuliert zu *produzieren*, und dieses in die Simulation bzw. an das angeschlossene Simulations-Element weiter zu geben. Dafür wird in dieser Klasse eine Variable benutzt, in der verwaltet wird, wie viele Simulations-Schritte zwischen dem Hineingeben von zwei Payloads in die Simulation vergehen müssen. Gesetzt wird diese Variable entweder dann, wenn der Benutzer dieses Element über den von diesem bereitgestellten Konfigurations-Dialog konfiguriert, oder aber im Rahmen des Ladevorganges aus der von dieser Klasse entsprechend erweiterten Methode `loadingAssistent(...)` heraus. An diesen beiden Stellen besteht ferner die Möglichkeit, die *Verzögerung* einzustellen, die bis zum Hineingeben des ersten Payloads nach dem Start eines Simulations-Laufes zu verstreichen hat. Weitere Konfigurationen der Instanzen dieser Klasse erfolgen im *Konstruktor*; in diesem wird z. B. die maximal gestattete Anzahl von ausgehenden Verbindungen eines solchen Elementes auf 1 gesetzt (da ein InPort nur genau einen Nachfolger haben darf). Außerdem wird dort, sofern die Simulation mit einer grafischen Oberfläche betrieben wird (vgl. Kapitel 6.6), unter Verwendung der von der Super-Klasse `BaseSimElement` bereitgestellten Visualisierungs-Komponente die *Symbol-Abbildung* für Instanzen dieser Klasse gesetzt. Diese ist in Abbildung 8.8 dargestellt.



(a) InPort (b) OutPort (c) Conveyer

Abbildung 8.8: Symbolabbildungen der Simulations-Elemente (Teil 1)

Die Realisierung der von den Instanzen dieser Klasse zu leistenden Aufgabe (d. h. das regelmäßige Hineingeben neuer Payloads in die Simulation) erfolgt mittels einer Erweiterung der geerbten Methode `updateStepMain()` (vgl. Abschnitt 8.4.2). In dieser wird überprüft, ob es gemäß der Anzahl der absolvierten Simulations-Schritte seit dem letzten Hineingeben eines Payloads in die Simulation wieder notwendig ist, ein neues Payload zu erzeugen. Sollte dieses so sein, und sollte die eine zulässige Instanz eines `OutConnector` existieren und mit einer korrespondierenden `InConnector`-Instanz verbunden sein, wird mittels eines Aufrufes der Methode `sendPayloadIn()` des OutConnectors versucht, den Payload-Transfer durchzuführen. Sollte dieser Transfer erfolgreich sein, erfolgen diverse Aktualisierungen von Verwaltungs-Variablen (wie z. B. der Anzahl transferierter Payloads oder des Zeitpunktes des letzten Transfers), sowie das Hinzufügen von Informationen über diesen Transfer zu den Protokoll-Daten dieses Elements. Sollte hingegen bei einer der Überprüfungen zuvor oder bei dem Versuch, das Payload zu transferieren ein Fehler aufgetreten sein, so wird eine diesem Fehler entsprechende Instanz eines `ErrorObjects` angelegt, und die Methode `createError(...)` der Super-Klasse `BaseSimElement` aufgerufen (vgl. 8.4.2). Schließlich wird an dieser Stelle noch der Status dieser Simulations-Element-Instanz entsprechend der zuletzt ausgeführten Aktion und entsprechend des Erfolges oder Misserfolges dieser Aktion aktualisiert.

8.4.7 OutPort

Instanzen der konkreten Klasse `OutPort` realisieren die Funktionen des System-Elementes, über das Payloads aus dem Simulations-Modell entfernt werden, d. h. über das diese Payloads das Modell simuliert verlassen. Die Beschreibung des zugrundeliegenden Elementes des Real-Systems findet sich in Abschnitt 5.2.2. Wie dort beschrieben ist, dauert das „Entlassen“ von Payloads aus dem System eine gewisse Bearbeitungszeit. Somit muss dieses auch in der Simulation eine entsprechende Zeitspanne in Anspruch nehmen. Diese Zeit, die für ein Payload ab dem Zeitpunkt des Eintreffens an diesem Simulations-Element benötigt wird, bis dieses Payload von diesem Element und somit aus dem System entfernt werden kann, wird in dieser Klasse mittels einer dafür vorgesehenen Variable verwaltet. Das Setzen dieser erfolgt hier wiederum entweder über den Konfigurations-Dialog, d. h. mittels einer direkten Benutzer-Interaktion, oder beim Laden dieses Elements aus der entsprechend erweiterten Methode `loadingAssistent(...)` heraus. Eine weitere, für die Gewährleistung der korrekten Funktion dieses Elementes nötige Variable, die an diesen Stellen gesetzt werden kann, ist die *Kapazität* dieses Elementes, d. h. die Anzahl der Payloads, die maximal zu einem Zeitpunkt t „in“ diesem Element vorhanden sein dürfen. Weitere Konfigurationen der Instanzen dieser Klasse erfolgen auch hier im *Konstruktor* der Klasse. Dort vorgenommene Konfigurationen umfassen ebenfalls wieder (im Falle des Betriebs der Simulation in einer grafischen Umgebung) das Setzen der Symbol-Abbildung (vgl. Abbildung 8.8) für diese Art von Simulations-Element, sowie das Festlegen der zulässigen Anzahl von eingehenden Verbindungen.

Die Durchführung des Hauptteils der von dieser Klasse bereitgestellten Funktionalität erfolgt hier, indem die geerbte Methode `updateStepPreproccessing()` erweitert wird. Generell ist dieses der Schritt, in dem Simulations-Elemente überprüfen, ob an ihren eingehenden Verbindungen neue, zu übernehmende Payloads anliegen. Dieses wird auch hier in einem ersten Schritt zunächst überprüft, indem die Methode `hasPayload()` des mit dem einen hier zulässigen InConnector verbundenen OutConnector aufgerufen wird. Liefert diese `true` zurück, so liegt dort ein Payload an, das übernommen werden muss. Dazu wird nun zunächst überprüft, ob die Anzahl der Payloads auf dieser `OutPort`-Instanz kleiner der festgelegten

Payload-Kapazität ist. Sollte dieses so sein, wird das an dem InConnector anliegende Payload mittels eines Aufrufes der Methode `getPayloadDelivery()`, die von dem dort verbundenen OutConnector bereitgestellt wird, übernommen. Daran anschließend werden die Variablen, die den aktuellen Zustand dieses OutPorts darstellen, entsprechend aktualisiert. Dabei erfolgt insbesondere ein Eintrag in eine verwaltete Liste mit Zeitpunkten, zu denen Payloads von diesem OutPort entfernt werden müssen. Dieser neue Eintrag stellt den Zeitpunkt dar, zu dem das gerade übernommene Payload „gelöscht" werden kann. Sollte hingegen die Payload-Kapazität des OutPorts im Moment dieser Überprüfung ausgelastet sein, wird von diesem OutPort ein entsprechender Fehler erzeugt, und mittels der von der Super-Klasse `BaseSimelement` bereitgestellten Methode `createError(...)` weitergegeben.

Das Entfernen von Payloads von einem OutPort ist realisiert in einer Erweiterung der geerbten Methode `updateStepMain()`. Bei jedem Aufruf dieser Methode wird überprüft, ob die Liste mit Zeitpunkten, zu denen ein Payload von diesem OutPort entfernt werden muss, den aktuellen Simulations-Zeitpunkt (d. h. die Nummer des aktuellen Simulations-Schrittes) enthält. Sollte dieser Zeitpunkt in der Liste enthalten sein, wird ein Payload simuliert von diesem OutPort entfernt.

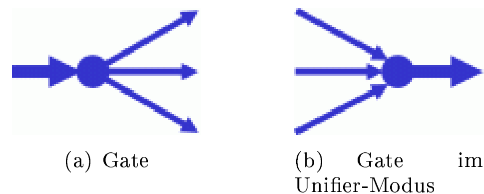
8.4.8 Conveyer

Die konkrete Klasse `Conveyer` realisiert die in Abschnitt 5.2.2 beschriebene, gleichnamige Komponente des Real-Systems. Die Aufgabe von Instanzen dieser Klasse ist es, von genau einem Vorgänger übernommene Payloads zu genau einem Nachfolger zu transportieren. Dabei kann ein solcher Conveyer in zwei verschiedenen Modi arbeiten: entweder, er arbeitet durchgängig, unabhängig von den Positionen der gerade beförderten Payloads, oder aber er läuft solange, bis ein Payload an seinem (simulierten) Ende eintrifft, und stoppt dann, bis dieses Payload an die Nachfolge-Komponente übergeben werden kann. Dieser Modus wird in der Klasse `Conveyer` verwaltet, und lässt sich nur aus der Klasse oder gegebenenfalls aus Sub-Klassen heraus mittels eines Aufrufes der Methode `setConveyerMode(...)` setzen. Ähnlich angelegt ist das Setzen weiterer Parameter wie der Kapazität des Conveyers (d. h. die Anzahl von Payloads, die zu einem Zeitpunkt t befördert werden kann), und die benötigte Beförderungszeit, d. h. die Anzahl der Simulations-Schritte, die der Conveyer aktiv sein muss, um ein an seinem InConnector eingetroffenes Payload zu seinem OutConnector zu befördern. Weitere Konfigurationen der Instanzen dieser Klasse erfolgen auch hier wieder im Konstruktur. Das dort beispielsweise festgelegte Symbol für einen Conveyer findet sich in Abbildung 8.8. Ferner wird dort wiederum die zulässigen Anzahl von ein- und ausgehenden Verbindungen gesetzt.

Wird nun für einen Conveyer die erste Phase eines Simulations-Schrittes eingeleitet (d. h. die in der Klasse `Conveyer` erweiterte Methode `updateStepPreproccessing()` aufgerufen, so wird zunächst überprüft, ob das Element im automatisch stoppenden Modus arbeitet, und deswegen gegenbenenfalls gerade angehalten ist. Sollte dem nicht so sein bzw. sollte der Conveyer im permanent laufenden Modus arbeiten wird überrpüft, ob am InConnector des Conveyer ein zu übernehmendes Payload anliegt. Sollte dem so sein, und sollte mit der Annahme dieses Payloads die Kapazität des Conveyers nicht überschritten werden, wird dieses Payload angenommen. Andernfalls, d. h. sofern das Payload nicht angenommen werden konnte, erfolgt mittels der bereits zuvor erläuterten Vorgehensweise das Auslösen eines Fehlers. Gelangt der Conveyer in die zweite Phase eines Simulations-Schrittes, d. h. wird die Methode `updateStepMain()` aufgerufen, wird zunächst überprüft, ob der Conveyer gerade gestoppt ist (im Falle eines konstant laufenden Conveyers liefert diese Überprüfung immer, dass der Conveyer nicht gestoppt ist). Daran anschließend wird der aktuelle Simulations-Zeitpunkt mit der Liste der verwalteten Zeitpunkte verglichen, zu denen Payloads den Conveyer wieder „verlassen" müssen. Sollte der aktuelle Zeitpunkt in dieser Liste als Eintrag existieren wird versucht, den Transfer eines Payloads an den angeschlossenen OutConnector einzuleiten. Im Erfolgsfall werden Verwaltungs-Parameter wie z. B. die Anzahl weitergeleiteter Payloads aktualisiert, Protokolldaten für diesen Schritt zusammengestellt, und der Schritt abgeschlossen. War das Weiterleiten des Payloads hingegen fehlerhaft, weil z. B. an dem entsprechenden OutConnector noch ein Payload eines früheren Transfers anlag, wird ein zu der Art des Fehlers passendes `ErrorObject` erzeugt, und als Fehlermeldung weitergegeben.

8.4.9 Gate

Eine genaue Beschreibung der möglichen Funktionsweisen eines Gates erfolgte ebenfalls in Abschnitt 5.2.2. Aus dieser gingen die verschiedenen Modi hervor, in denen diese Komponente arbeiten kann. Die praktischen Umsetzungen dieser alle im Detail zu beschreiben würde hier zu weit führen, statt dessen sei für diese auf den Quelltext und die dort erfolgte Kommentierung verwiesen. Hier soll nun lediglich ein kurzer Überblick über den Ablauf innerhalb einer Instanz der Klasse `Gate` gegeben werden. Die für Instanzen dieser Klasse benutzten Symbole sind dargestellt in Abbildung 8.9.



(a) Gate (b) Gate im Unifier-Modus

Abbildung 8.9: Symbolabbildungen der Simulations-Elemente (Teil 2)

Der wichtigste für die korrekte Funktionsweise eines Gates zu verwaltende Parameter ist die Zeitspanne, die zwischen der Ankunft eines Payloads an den Gate und dem Weiterreichen dieses an die dafür vorgesehene Nachfolge-Komponente liegt. Außerdem besitzt auch ein Gate wieder eine Kapazität bezüglich gleichzeitig zu verarbeitender Payloads. Ebenfalls verwaltet werden muss der Modus, in dem die entsprechende Instanz eines `Gate` zu arbeitet, sowie in diesem Zusammenhang modus-spezifische Daten wie z. B. der präferierte Nachfolger, an den Payloads wenn irgend möglich weitergegeben werden sollen. Das Setzen dieser Parameter erfolgt wie schon zuvor bei den anderen konkreten Klassen erläutert.

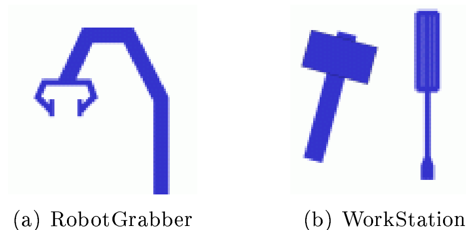
In der von der Klasse `Gate` vorgenommenen Erweiterung der Methode `updateStepPreproccessing()` unterscheidet ein Gate zunächst, ob es im *Unifier*-Modus läuft, d. h. mehrere Eingänge auf einen Ausgang bündeln muss, oder ob von von einem Eingang ankommende Payloads auf mehrere Nachfolge-Komponenten verteilen muss. Befindet es sich in ersterem Modus, so werden die vorhandenen InConnectoren der Reihe nach auf das Vorliegen eines neuen Payloads überprüft. Liegt dabei an einem InConnector ein Payload vor, so wird dieses Payload angenommen, sofern damit nicht die Kapazität des Gate überschritten wird. In diesem Fall würde ein entsprechender Fehler ausgelöst. Ein weiterer Fehler könnte entstehen, wenn zu einem Zeitpunkt t an mehreren InConnectoren zugleich neue Payloads anliegen würden. Auch ein derartiger Fehler würde entsprechend mittels einer neu erzeugten Instanz eines `ErrorObject` weitergegeben werden. Arbeitet die `Gate`-Instanz hingegen nicht im Unifier-Modus, so kann maximal der Fehler entstehen, dass ein Payload nicht angenommen werden kann, da dieses die Kapazität des Gates überschreiten würde. Auch dieser Fehler würde mit dem entsprechenden Aufruf der `createError(...)`-Methode der Super-Klasse `BaseSimElement` weitergegeben werden. Kann hingegen – unabhängig vom Modus – ein Payload erfolgreich übernommen werden, wird für dieses der Zeitpunkt in eine Liste eingetragen, zu dem dieses Payload dieses Gate wieder „verlassen“ muss. Ferner werden hier diverse Zustands-Variablen wie z. B. die Anzahl angenommener Payloads aktualisiert, und die für diesen Teilschritt zu protokollierenden Daten zusammengestellt.

In der nächsten Phase eines Simulationsschrittes, d. h. in der von der Klasse `Gate` vorgenommenen Erweiterung der Methode `updateStepMain()`, wird zunächst überprüft, ob die Liste mit den Zeitpunkten, zu denen ein Payload an eine Nachfolge-Komponente weiterzuleiten ist, den aktuellen Simulations-Zeitpunkt enthält. Ist dieser dort enthalten, so wird unter den verwalteten `OutConnector`-Instanzen diejenige gesucht, an die dieses Payload weiterzugeben ist. Wie genau der entsprechende Nachfolger bestimmt bzw. ausgewählt wird, hängt dabei von dem Modus ab, in dem die `Gate`-Instanz arbeitet; beispielsweise kann dieser in dem entsprechenden Modus randomisiert bestimmt werden. Die verschiedenen Vorgehensweise zur Bestimmung sollen hier nun nicht betrachtet werden, da dieses zu weit führen würde. Statt dessen wird nun angenommen, dass der entsprechende Nachfolger bestimmt sei. Dann wird generell, d. h. unabhängig von dem Modus mittels eines Aufrufes der Methode `sendPayloadIn()` des entsprechenden

OutConnectors versucht, das Payload zu diesem weiterzureichen. Ist dieser Versuch erfolgreich werden ebenfalls wieder unabhängig vom Modus des Gate Aktualisierungen der Zustands-Variablen der `Gate`-Instanz vorgenommen. War der Versuch der Weiterleitung hingegen nicht erfolgreich wird ein entsprechendes ErrorObject erstellt und weitergeleitet. Weitere Schritte, die in einem solchen Fehlerfall folgen, hängen vom Betriebs-Modus der `Gate`-Instanz ab. Beispielsweise könnte im entsprechenden Modus die Auslieferung dieses Payloads zunächst einfach um einen Zeittakt weiter in die Zukunft verschoben werden. Dieses würde realisiert mittels einer Änderung des entsprechenden Eintrages in der verwalteten Liste der Zeitpunkte, zu denen Payloads von dieser `Gate`-Instanz aus weitergegeben werden müssen.

8.4.10 RobotGrabber

Instanzen der Klasse `RobotGrabber` stellen die Funktionalität bereit, Payloads von einer ihrer eingehenden Verbindungen zu einer ihrer ausgehenden Verbindungen zu „transportieren". Dementsprechend ist die Klasse `RobotGrabber` eine Sub-Klasse von `InsideElement`. Bildlich gesprochen erkennt ein solcher RobotGrabber, dass er an einer seiner Schnittstellen ein Payload abholen und dieser zu einer anderen seiner Schnittstellen befördern muss, was eine gewisse Zeit dauert. In der konkreten Implementierung wird explizit zwischen ein- und ausgehenden Schnittstellen unterschieden; eine Beförderung eines Payloads ist nur von einer eingehenden zu einer ausgehenden Schnittstelle möglich. Ferner ist hier zu beachten, dass ein solcher RobotGrabber möglicherweise vor Beginn des Transports eines Payloads eine gewisse Zeit braucht, um sich passend zu positionieren. Bei der hier vorgenommenen Implementierung kann der RobotGrabber außerdem in zwei verschiedenen Modi arbeiten: zum einen kann er eingesetzt werden, um Payloads von beliebig vielen Vorgänger-Elementen an beliebig viele Nachfolger-Komponenten zu verteilen, und zum zweiten kann er in einem Modus arbeiten, in dem er eine *WorkStation* (vgl. Abschnitt 8.4.11) mit zu bearbeitenden Payloads „beliefert", und von dieser WorkStation bearbeitete Payloads wieder dort „abholt". In diesem Fall ist diese WorkStation also sowohl über eine eingehende als auch über eine ausgehende Verbindung mit der entsprechenden `RobotGrabber`-Instanz verbunden.



(a) RobotGrabber (b) WorkStation

Abbildung 8.10: Symbolabbildungen der Simulations-Elemente (Teil 3)

Die wichtigen, bei der Erstellung oder dem Laden einer `RobotGrabber`-Instanz zu setzenden und von dieser zu verwaltenden Parameter sind bereits angesprochen worden. Dieses ist zunächst die Zeit (in Form von Simulations-Schritten), die ein solcher RobotGrabber für den Transport eines Payloads benötigt. Außerdem wird unter Umständen eine gewisse Positionierungszeit benötigt, die verstreicht, bevor ein RobotGrabber ein an einem InConnector angekommenes Payload aufnehmen kann. Diese muss ebenfalls entsprechend gesetzt werden. Schließlich muss noch der Modus, in dem der RobotGrabber arbeitet, gesetzt und verwaltet werden. Diese Einstellungen werden, wie bei den zuvor beschriebenen Komponenten, mittels des bereitgestellten Benutzer-Dialoges oder beim Laden einer Instanz aus der Methode `loadingAssistent(...)` heraus vorgenommen.

In der ersten Phase eines Simulations-Schrittes, d. h. in der in dieser Klasse vorgenommenen Erweiterung der Methode `updateStepPreproccessing()`, überprüft ein RobotGrabber, ob an einer seiner eingehenden Verbindungen ein zu beförderndes Payload anliegt. Ist dieses der Fall, so wird zunächst bestimmt, an welche der ausgehenden Verbindungen dieses Payload zu liefern ist. Dieses ist u. a. abhängig von dem Modus, in dem diese `RobotGrabber`-Instanz arbeitet. Ist dieses Ziel für das neue Payload bestimmt,

so wird ein sogenannter *Delievery Job* erstellt. Dieser geschieht in Form einer Instanz der innerhalb der Klasse `RobotGrabber` implementierten Klasse `DelieveryJob`, in welcher alle wichtigen, einen solchen „Liefer-Auftrag" definierenden Parameter (wie z. B. das Ziel dieser „Lieferung") verwaltet werden. Diese neu erzeugte `DelieveryJob`-Instanz wird in eine innerhalb des RobotGrabbers verwaltete *Queue* (vom Typ `java.util.Queue`) eingereiht. Hier ist es auch möglich, dass in einem Schritt an mehreren InConnectoren gleichzeitig neue Payloads anliegen; diese werden dann in der Reihenfolge in die Warteschlange einsortiert, in der sie beim Iterieren über die existierenden InConnectoren gefunden wurden (aufgrund dieses Vorgehens ist es im Übrigen ausgeschlossen, dass bei der Abarbeitung der Methode `updateStepPreproccessing()` Fehler auftreten können).

In der Erweiterung der Methode `updateStepMain()`, also in der zweiten Phase eines Simulations-Schrittes, wird nun zunächst überprüft, ob der RobotGrabber gerade aktiv ist, d. h. ob dieser entweder ein Payload von einem InConnector zu einem OutConnector befördert, oder aber ob dieser sich gerade positioniert, um ein neu zu beförderndes Payload aufnehmen zu können. In beiden Fällen wird dieser Vorgang zunächst fortgesetzt, d. h. in diesem Aufruf dieses Schrittes wird keine weitere Aktion durchgeführt. Ist der RobotGrabber hingegen nicht aktiv (weil er z. B. in dem vorangegangenen Simulations-Schritt eine Payload-Lieferung abgeschlossen hat), so wird die erste `DelieveryJob`-Instanz aus der zuvor genannten Queue entfernt, und die Abarbeitung dieses *Delievery Jobs* initiiert. Dazu wird zunächst ausgewertet, an welchem der InConnectoren das entsprechende Payload zu übernehmen ist. Dieses wird mit der aktuellen Position des RobotGrabbers verglichen (dieser ist positioniert an dem verbundenen Vorgänger oder Nachfolger, mit dem er zuletzt interagiert hat, d. h. an den er z. B. zuletzt ein Payload geliefert hat). Entspricht diese Postion der Position, an der das nächste Payload gemäß der ausgewerteten `DelieveryJob`-Instanz abzuholen ist (dieses wäre z. B. der Fall, wenn zu einer WorkStation geliefert wurde, und dort wiederum gerade ein Payload fertig bearbeitet worden ist), kann direkt die Beförderung dieses Payloads initiiert werden, d. h. der RobotGrabber ist nun so lange aktiv, wie dieses die benötigte Zeit zur Lieferung eines Payloads vorgibt. Andernfalls muss sich der RobotGrabber positionieren. Dieses wird simuliert, indem er so lange aktiv gesetzt wird, wie dieses durch den Parameter vorgegeben wird, der die Positionierungszeit (s. o.) definiert. Gleichzeitig wird hier eine Variable gesetzt, die anzeigt, dass der RobotGrabber in einer Positionierungs-Phase ist. Ist diese Phase beendet kann die Beförderung des Payloads dieses aktuellen Delievery Jobs initiiert werden. Bei der Initiierung dieses Delievery Jobs wird der RobotGrabber wieder auf aktiv gesetzt. Der Delievery Job ist abgeschlossen, wenn die für das Liefern benötigte Zeit verstrichen ist, und der entsprechende OutConnector, zu dem das Payload zu liefern war, dieses angenommen hat. Sollte diese das Payload nicht akzeptieren wird hier ein entsprechender Fehler in Form einer `ErrorObject`-Instanz erzeugt und weitergeleitet.

8.4.11 WorkStation

Die konkrete Klasse `WorkStation`, deren Symbolisierung in Abbildung 8.10 dargestellt ist, stellt verglichen mit den zuvor beschriebenen Klassen eine gewisse Sonderform dar. Während diese die hauptsächliche Aufgabe hatten, Payloads simuliert zu *befördern*, hat diese Komponente die Aufgabe, Payloads simuliert zu *bearbeiten*. Die Vorgänger-Komponente, von der diese zu bearbeitenden Payloads geliefert werden, ist dabei die gleiche Komponente, an die fertig bearbeitete Payloads „zurückgegeben" werden. Dabei wurde festgelegt (vgl. Abschnitt 5.2.2), dass eine WorkStation immer von einem RobotGrabber „bedient" wird, d. h. *eine* Instanz der Klasse `RobotGrabber` befördert Payloads zu einer WorkStation hin und wieder von dieser weg. Ferner können beim „Bearbeiten" eines Payloads Fehler innerhalb einer WorkStation auftreten, d. h. an dieser kann ein Defekt auftreten. Tritt ein Defekt auf, so dauert es eine gewisse Zeit, bis dieser wieder behoben ist. Aus dieser Erläuterung der Funktion einer WorkStation lassen sich nun die von dieser Klasse zu verwaltenden Parameter ableiten: dieses sind neben der Zeit, die für die simulierte Bearbeitung eines Payloads benötigt wird die Wahrscheinlichkeit, dass bei einer derartigen Bearbeitung ein Defekt an der WorkStation auftritt, sowie noch die Zeit, die im Falle eines solchen Defektes verstreichen muss, bis dieser wieder behoben ist. Das Setzen dieser Parameter erfolgt wie schon bei den Klassen zuvor entweder über den von der Klasse `WorkStation` bereitgestellten Konfigurations-Dialog, oder beim Laden

von Instanzen dieser Klasse aus der vorgenommenen Erweiterung der Methode `loadingAssistent(...)` heraus.

Der erste Schritt, den Instanzen der Klasse `WorkStation` in der ersten Phase eines Simulations-Schrittes durchführen, ist zu überprüfen, ob an dieser WorkStation aktuell ein Defekt vorliegt. Ist dieses der Fall, so können hier keine weiteren Aktionen ausgeführt werden. Es wird lediglich eine Fehlermeldung erzeugt, in der unter anderem explizit die verbleibende Zeit vermerkt ist, bis die WorkStation wieder einsatzbereit ist, und diese Phase des Simulations-Schrittes wird verlassen. Liegt hingegen kein Defekt vor, so wird in einem nächsten Schritt überprüft, ob an dem einen verwalteten InConnector ein Payload anliegt, das bearbeitet werden muss. Ist dieses der Fall, und ist die WorkStation aktuell nicht mit der Beabeitung eines anderen Payloads befasst, so wird die Bearbeitung dieses Payloads initiiert. Dazu wird z. B. der Status der WorkStation von zuvor *idle* auf *working* gesetzt, die Anzahl von angenommenen Payloads heraufgesetzt, und insbesondere die *verbleibende Bearbeitungszeit* für den aktuellen Bearbeitungsvorgang zurückgesetzt auf den Wert, der für die gesamte Bearbeitung eines Payloads benötigt wird. Ist hingegen die WorkStation noch mit der Bearbeitung eines Payloads befasst, so wird dieses neue Payload noch nicht von dem InConnector übernommen, sondern in dieser eingehenden Schnittstelle belassen. Ferner wird eine Information erzeugt, dass bereits ein neues Payload zur Bearbeitung vorliegt, obwohl die Bearbeitung des vorherigen Payloads noch nicht abgeschlossen war. Diese Warnung wird erzeugt in Form einer `ErrorObject`-Instanz mit der *Severity* `Information`, und propagiert mit der Methode `createError()`[1]. Dieses schließt die erste Phase des Simulations-Schrittes ab.

In der zweiten Phase des Simulations-Schrittes, also in der Methode `updateStepMain()`, wird ebenfalls zunächst überprüft, ob die WorkStation aktuell defekt ist. Ist dieses der Fall, so wird die *remaining repair time*, also die Zeit, bis der Defekt behoben ist, um einen Zeitschritt verringert. Anschließend wird überprüft, ob durch diese Verringerung möglicherweise die Zeit, die für die Behebung eines Defekts notwendig ist, verstrichen ist. Ist dieses der Fall, so wäre die WorkStation im nächsten Simulations-Schritt wieder einsatzbereit. Dementsprechend würde der Status der WorkStation gesetzt, und es würde mittels einer geeigneten `ErrorObject`-Instanz weitergemeldet, dass der Defekt der WorkStation aufgelöst wurde. Befindet sich die WorkStation gemäß des Ergebnisses der Überprüfung zu Anfang dieses Teilschrittes nicht im Zustand eines Defekts, so wird zunächst überprüft, ob sie aktuell mit der Bearbeitung eines Payloads befasst ist. Sollte dem nicht so sein müssen in diesem Schritt keine weiteren Aktionen durchgeführt werden. Andernfalls wird die verbleibende Zeit, die zur Bearbeitung dieses Payloads notwendig ist, um einen Zeitschritt verringert. Dieses könnte dazu führen, dass dieses Payload fertig bearbeitet ist, was im Folgenden überprüft wird. Sollte die Bearbeitungszeit abgelaufen sein wird versucht, dieses nun bearbeitete Payload an die verwaltete `OutConnector`-Instanz zu übergeben. Ist dieses erfolgreich werden noch diverse Zustands-Variablen der `WorkStation`-Instanz aktualisiert (wie z. B. die Anzahl fertig-bearbeiteter Payloads oder der *Zustand* dieses Elementes). War dieses hingegen nicht erfolgreich wird ein entsprechender Fehler erzeugt und weitergegeben.

Sollte es in diesem Schritt notwendig gewesen sein, eine Bearbeitung eines Payloads durchzuführen, so erfolgt zuletzt noch die Berechnung, ob diese `WorkStation`-Instanz dabei möglicherweise „beschädigt“ wurde. Dieses wird berechnet anhand der eingestellten *Defekt-Wahrscheinlichkeit* in der Methode `calculateBreakdownHappened()`. Das Ergebnis dieser Berechnung wird weitergegeben an die Methode `setIsBrokenDown(...)`, welche die dann notwendigen Aktualisierungen der Status-Variablen vornimmt.

8.5 Web Services der Simulations-Elemente

Hier soll nun darauf eingegangen werden, wie die in Abschnitt 6.4.2 geforderte Bereitstellung von *Web Services* für die einzelnen Modell-Komponenten realisiert wurde. Die Implementierung dieser erfolgte dabei unter Verwendung des *WS4D J2ME*-Stacks.

[1] Hier mag die Bezeichnung dieser Methode etwas verwirren. Deswegen sei noch einmal ausdrücklich herausgestellt, dass diese Methode, ebenso wie das `ErrorObject`, trotz der Bezeichnung nicht nur zum Propagieren von Fehlern, sondern auch von *Warnungen* und *Informationen* dient.

8.5.1 Grundlagen

Grundlegend für die Realisierung ist die Klasse `SimulationElementDevice`. Diese stellt ganz allgemein ein *Device* dar, dem *beliebige* Services für *beliebige* Modell-Komponenten hinzugefügt werden können. Die Services, die einer konkreten Instanz dieser Klasse hinzugefügt werden bestimmen dann, an welche Art von Modell-Komponente (*Gate*, *Conveyer*, ...) diese Instanz gebunden ist. Dazu erweitert die Klasse `SimulationElementDevice` die von dem WS4D J2ME - Stack vorgegebene Klasse `HostingService` (vgl. Abschnitt 7.3.3). Wichtige Parameter wie z. B. der Name oder die Adresse für dieses *Device* werden dabei im Konstruktor dieser Klasse übergeben, und entsprechend gesetzt. Weiterhin grundlegend ist die Verwaltung der Instanzen der Klasse `SimulationElementDevice`, die eine Instanz einer Modell-Komponente erzeugt hat und bereitstellt. Die für diese Verwaltung nötigen Funktionen werden dabei größtenteils bereits in der abstrakten Super-Klasse `BaseSimElement` implementiert, von der alle konkreten, die Modell-Komponenten darstellenden Klassen, abgeleitet sind. Verwaltet bzw. bereitgestellt werden dort insbesondere:

- ein `Vector`, in dem alle in den entsprechenden Sub-Klassen erzeugten Instanzen der Klasse `SimulationElementDevice` verwaltet werden.
- eine bool'sche Variable, die verwaltet, ob für diese Instanz der Sub-Klasse des `BaseSimElement` zur Zeit die Web Service - Funktionalitäten aktiviert oder deaktiviert sind; sowie die Methoden zum Abfragen (`getDpwsIsEnabled()`) oder Setzen (`setDpwsIsEnabled(...)`) dieser Variablen.
- die Methode `stopDpwsServices()`, welche alle zu dieser Instanz gehörigen Instanzen der Klasse `SimulationElementDevice` durchläuft und stoppt.

Außerdem werden in dieser Super-Klasse die abstrakten, von den konkreten Klassen zu implementierenden Methoden `startDpwsServices()` und `getDpwsService()` vorgegeben. Diese sollen dazu dienen, die Web Service - Funktionalitäten in den konkreten Sub-Klassen anzulegen und zu starten, bzw. nachdem dieses erfolgt ist den Zugriff auf diesen Web Service ermöglichen.

Grundlegend für die implementierten *Services* der konkreten Modell-Komponenten, die dem oben erläuterten `SimulationElementDevice` hinzugefügt werden und dann dessen Funktionen definieren, ist die Klasse `BaseSimElementService`. Diese realisiert einen Basis-Service für das BaseSimElement, der Zugriff auf Grundfunktionen und Grundinformationen des BaseSimElement ermöglicht. Dieses umfasst beispielsweise das Abfragen oder Setzen des Bezeichners des Simulations-Elementes, oder auch das Stoppen des Web Services dieses Elementes. Insbesondere umfasst dieses die Möglichkeit, dass sich ein Client auf von diesem Service ausgehende Events subscribed, die im Fehlerfalle ausgelöst werden. Um als Service dem `SimulationElementDevice` hinzugefügt werden zu können, erweitert die Klasse `BaseSimElementService` die von dem *WS4D*-Stack vorgegebene Klasse `HostedService` (vgl. 7.3.3).

8.5.2 Services der Komponenten

Die *Services*, die Zugriff auf spezielle von den Komponenten bereitgestellte Funktionen ermöglichen, erweitern alle wie in Abbildung 8.11 dargestellt die Klasse `BaseSimElementService`. Diese stellt, wie oben beschrieben, Zugriffsmöglichkeiten auf die Basis-Funktionen eines jeden Simulations-Elementes zur Verfügung. Für die speziellen Zugriffmöglichkeiten, die die Services der konkreten Modell-Komponenten gestatten, sei auf den Quelltext dieser Services verwiesen.

Erstellt werden müssen diese Services der Komponenten in der von diesen konkreten Klassen zu implementierenden, vom `BaseSimElement` abstrakt vorgegebenen Methode `startDpwsService()`. In dieser erzeugt eine konkrete Klasse einer Modell-Komponente zunächst eine neue Instanz der Klasse `SimulationElementDevice`. Im nächsten Schritt wird eine neue Instanz der Service-Klasse dieses Simulations-Elementes angelegt (beispielsweise erzeugt ein *InPort* hier eine Instanz der Klasse `InPortService`). Diese Instanz

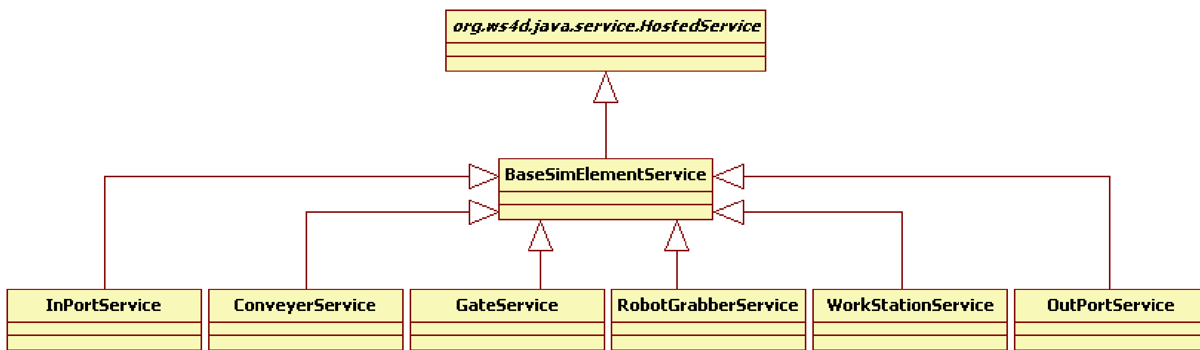


Abbildung 8.11: Klassen-Struktur der Services der Modell-Komponenten

des Services wird der Instanz des `SimulationElementDevice` über die dort bereitgestellte Methode `add-HostedService(...)` hinzugefügt. Anschließend wird dieses Device der von `BaseSimElement` geerbten Datenstruktur hinzugefügt, die diese Devices verwaltet. In einem letzten Schritt wird das Device gestartet.

8.5.3 Schnittstelle zwischen Service und Komponente

Ein Problem bei der Implementierung der von den Modell-Komponenten bereitgestellten Web Services lag darin, dass alle Methoden zum Manipulieren von für die Funktion der entsprechenden Modell-Komponente wichtigen Parameter zur Sicherung der Integrität dieser Parameter grundsätzlich als `private` oder `protected` implementiert wurden. Daher sind diese Methoden und Parameter von außen nicht zugreifbar; folglich auch nicht aus den die Web Services realisierenden Klassen. Um dennoch die Kommunikation zwischen dem bzw. die Kontrolle durch den entsprechenden Service und der dazugehörigen Modell-Komponente zu realisieren, und dabei dennoch die Integrität der Parameter so weit wie möglich zu gewährleisten, wurde hierfür eine spezielle *Schnittstelle* geschaffen. Diese Schnittstelle ist implementiert in Form einer inneren Klasse mit der Bezeichnung `ExternalControl`. In dieser Klasse befinden sich Methoden, mittels denen auf die privaten oder geschützten Methoden der äusseren Klasse zugegriffen werden kann. Diese Klasse ist implementiert in der Super-Klasse `BaseSimElement`, und wird in den Sub-Klassen sukzessive um die Funktionen erweitert, die für den Zugriff auf die speziellen Parameter dieser Sub-Klassen notwendig sind. Ferner wird in der Super-Klasse `BaseSimElement` die abstrakte, von den konkreten Sub-Klassen zu implementierende Methode `getExternalControl()` vorgegeben. In den Implementierungen dieser Methode in den Sub-Klassen wird nun zunächst bei einem Aufruf dieser Methode überprüft, ob das konkrete Element durch externe Kontrollinstanzen steuerbar sein soll. Dieses wird mittels einer entsprechenden, vom Benutzer zu setzenden Variablen verwaltet. Sollte dem so sein, wird eine Instanz der internen Klasse `ExternalControl` zurückgeliefert, mittels der dann z. B. der diese Methode aufrufende Service Zugriff auf die benötigten Methoden erhält.

8.6 Auxiliary Devices

Bei den sogenannten *Auxiliary Devices* handelt es sich um zusätzliche Komponenten, die in einem Simulations-Modell nicht eigenständig existieren können, sondern immer an eine der zuvor vorgestellten Modell-Komponenten gebunden sind. Dabei liefern diese Auxiliary Devices dann eine gewisse zusätzliche Funktionalität. Denkbar wären Auxiliary Devices z. B. in Form von Wiege- oder Vermessungs-Geräten, als Sensoren zur Verfolgung dedizierter Payloads, usw. Vorgreifend kann hier bereits erwähnt werden, dass es augenblicklich lediglich ein beispielhaftes Auxiliary Device zur Verdeutlichung des dahinterstehenden Prinzips gibt. Dabei handelt es sich um eine *Lichtschranke*, die „an" einer bestimmten Position „auf" einer Modell-Komponenten angebracht werden kann, und eine ein Event auslösende Methode ihres

Web Services aufruft, wenn sie von einem Payload passiert wird.

8.6.1 Grundlagen der Implementierung

Grundlegend für die Implementierung und Einbindung von *Auxiliary Devices* sind die Klasse `BaseAuxiliaryDevice` und das Interface `IAuxiliaryDevice`. Letzteres definiert einige wenige Methoden, die von Klassen implementiert werden müssen, zu denen Auxiliary Devices hinzufügbar sein sollen. Diese Methoden dienen dabei den angeschlossenen Auxiliary Devices, um für ihren Betrieb wichtige Daten der Modell-Komponente abzufragen, „an" der sie „befestigt" sind. Dieses umfasst z. B. die Möglichkeit, die Ankunftszeiten von auf dieser Komponente aktuell vorhandenen Payloads abzufragen (`getPayloadArrivalTimes()`), oder die Geschwindigkeit, mit der Payloads von dieser Komponente simuliert „befördert" werden (`getSpeedOfElement()`). Da es grundsätzlich möglich sein soll, zu *jeder* Modell-Komponente Auxiliary Devices hinzuzufügen, ist hier dieses Interface direkt von der Klasse `BaseSimElement` implementiert. Von dieser Klasse werden auch die zur Verwaltung von Auxiliary Devices nötigen Funktionen bereitgestellt, die somit auch allem Sub-Klassen des `BaseSimElement` zur Verfügung stehen. Diese Funktionen umfassen neben einer Datenstruktur zur Verwaltung der zu dieser Instanz einer Modell-Komponente gehörigen Auxiliary Devices eine Methode zum Hinzufügen derartiger Devices. Hinzugefügt werden derartige Auxiliary Devices entweder mittels einer Benutzer-Interaktion, oder aber beim Laden einer Instanz einer Modell-Komponente.

Die grundsätzliche Struktur hinter den Auxiliary Devices ist ähnlich wie die hinter den normalen Modell-Komponenten. Auch hier existiert eine abstrakte Basis-Klasse (in Form der o. g. Klasse `BaseAuxiliaryDevice`), in der alle Grundfunktionen aller denkbaren Auxiliary Devices implementiert oder abstrakt vorgesehen sind. Dieses umfasst insbesondere die beiden abstrakt vorgesehenen, von den konkreten Auxiliary Device - Klassen zu implementierenden Methoden `payloadAtThisElement(...)` und `noPayloadAtThisElement(...)`, und die bereits im BaseAuxiliaryDevice konkret implementierte Methode `performStep(...)`. Diese muss von der das konkrete Auxiliary Device „besitzenden" Modell-Komponente bei jedem durchgeführten Simulations-Schritt aufgerufen werden. Diese Methode überprüft, ob in diesem Schritt ein Payload an der Position dieser Auxiliary Device - Instanz vorhanden ist, und ruft abhängig davon eine der beiden Methoden `payloadAtThisElement(...)` oder `noPayloadAtThisElement(...)` auf, in welcher dann die konkrete Auxiliary Device - Klasse ihre auszuführenden Aktionen implementiert hat. Daneben stellt diese Auxiliary Device - Basisklasse ebenfalls die grundlegenden Funktionen zum Bereitstellen eines Konfigurations-Dialoges für Instanzen von Sub-Klassen des `BaseAuxiliaryDevice` zur Verfügung, ebenso, wie die Funktionen zur Verwaltung und Bereitstellung von expliziten *Web Services* für die Auxiliary Devices.

8.6.2 Beispiel eines Auxiliary Devices

Als Beispiel für ein konkretes Auxiliary Device soll hier die Lichtschranke dienen. Diese stellt zugleich auch das zur Zeit einzige implementierte Auxiliary Device dar.

Wie in den Grundlagen erläutert erweitert die Klasse `LightBarrier`, die die Lichtschranke realisiert, die Klasse `BaseAuxiliaryDevice`. Dabei implementiert sie insbesondere die von dieser Klasse abstrakt vorgegebene Methode `startDpwsServices()`. Die Vorgehensweise entspricht dabei grundsätzlich der, wie sie auch von regulären Modell-Komponenten durchgeführt wird: zunächst wird ein spezielles *Device* im Sinne der Verwendung dieses Begriffes in Abschnitt 3.3 erzeugt, welchem Auxiliary Device - spezifische *Services* hinzugefügt werden können. Anschließend wird eine Instanz der Klasse `LightBarrierService` erzeugt, welche einen Web Service für diese LightBarrier darstellt. Dieser Service wird dem zuvor erzeugten Device hinzugefügt. Desweiteren implementiert die Klasse `LightBarrier` die beiden Methoden `payloadAtThisElement(...)` bzw. `noPayloadAtThisElement(...)`, und füllt diese mit der LightBarrier-spezifischen Funktionalität. Diese Funktionalität ist das Informieren des bereitgestellten Web Services, wenn ein Payload sich an der Position der LightBarrier befindet, bzw. wenn umgekehrt ein Payload diese

Position wieder verlassen hat. Da es sich bei der LightBarrier um ein recht einfaches Auxiliary Device handelt, ist dieses auch die einzige Aktion, die hier ausgelöst wird.

Kapitel 9

Simulations-Umgebung

In diesem Kapitel wird nun auf die in Kapitel 7.1 angeführte *Simulations-Umgebung* eingegangen, die den Rahmen für die Durchführung von Simulationen darstellt.

Dazu wird zunächst ein Überblick über die vorhandenen Module der Simulations-Umgebung gegeben. Ferner wird kurz dargestellt, wie diese Module untereinander in Relation stehen, d. h. wie die Software-Architektur angelegt ist. Daran anschließend wird auf die einzelnen Module eingegangen und erläutert, welches Modul welche Funktionen realisiert. Daraus ergibt sich, wie die Module genau miteinander interagieren. Daran schließt sich eine kurze Erläuterung der dem Programm zugrundeliegenden Threads an. Abschließend wird explizit herausgestellt, wie der für die Simulations-Umgebung geforderte Web Service (vgl. Abschnitt 6.4.1) realisiert wurde, und welche Funktionalitäten dieser bereitstellt.

9.1 Architektur und Konzept

In Anlehnung an Abbildung 7.1 zeigt Abbildung 9.1, in welche Hauptmodule sich die Simulations-Umgebung unterteilen lässt. Dabei entsprechen die dort verwendeten Bezeichnungen für die Module grundsätzlich den Bezeichnungen, die auch für die implementierenden Klassen gewählt wurden.

Die Module lassen sich entsprechend ihrer Funktion in 4 Gruppen unterteilen: zum einen existieren die in der Abbildung in der linken Spalte dargestellten *graphischen Module*, die ganz allgemein GUI-Funktionen bereitstellen. Mittels dieser Funktionen wird es dem Benutzer beispielsweise ermöglicht, mit dem Programm zu interagieren. Daneben gibt es die Gruppe der *administrativen Module*, die die von der Simulations-Umgebung benötigten Funktionen zum Steuern von Simulations-Abläufen und Funktionen zur Verwaltung von Simulationsmodellen realisieren. Diese sind hier in der rechten Spalte der Abbildung dargestellt. In dieser Spalte befindet sich ebenfalls noch eine Symbolisierung für verschiedene, in dieser Abbildung nicht explizit dargestellte *Hilfs-Module*. Diese realisieren teils graphische und teils administrative Hilfs-Funktionalitäten von für die Modul-Struktur geringerer Bedeutung. In dieser Abbildung werden diese aus Gründen der Übersichtlichkeit nicht einzeln dargestellt. Eine Übersicht über diese Hilfs-Module findet sich in Abschnitt 9.2.10, deswegen soll hier zunächst nicht näher auf diese eingegangen werden.

Schließlich existiert in Abbildung 9.1 noch das *Haupt-Modul* der gesamten Simulations-Umgebung, welches eine Art „Dach" darstellt, unter dem alle anderen Module zusammengeführt werden (daher auch die Bezeichnung *SimulatorRoof*). Eine der Hauptaufgaben dieses Moduls ist es, eine Schnittstelle bereitzustellen, über die die Kommunikation zwischen den verschiedenen weiteren Modulen zu erfolgen hat. Dafür stellt dieses Modul eine Vielzahl von Methoden zur Verfügung, die die anderen Module aufrufen können, und die dann die Informationen entsprechend weiterleiten. Eine genaue Beschreibung dieses Moduls findet sich in Abschnitt 9.2.1.

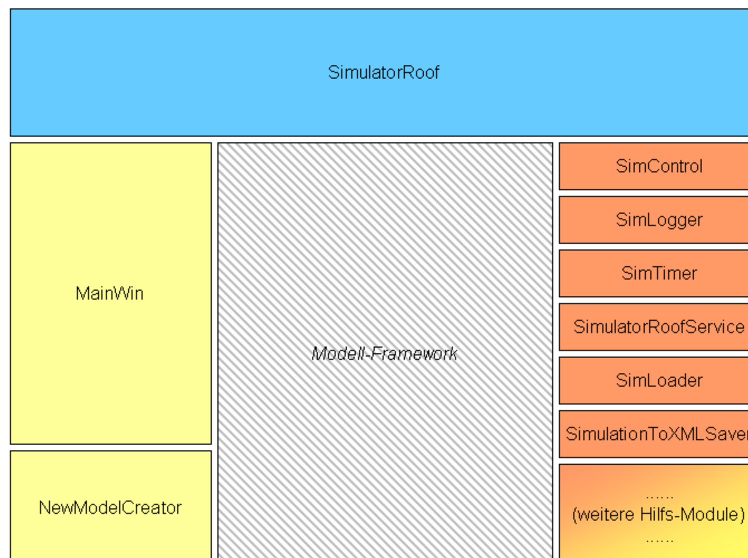


Abbildung 9.1: Module der Simulations-Umgebung

Abbildung 9.2 gibt einen Überblick über das Zusammenwirken dieser Module. Die genauen Interaktionen dieser Module werden näher im folgenden Kapitel 9.2 im Rahmen der Beschreibungen der einzelnen Module herausgestellt. Hier kann bereits festgehalten werden, dass sich die Aufgaben der Module größtenteils direkt aus den hier verwendeten Bezeichnungen der Module ableiten lassen. So dient das Modul *MainWin* dazu, die graphische Benutzerschnittstelle zur Verfügung zu stellen. Diese dient hier neben dem Entgegennehmen von Anweisungen durch den Benutzer auch zum Anzeigen von beispielsweise dem Fortschreiten einer Simulation, oder dem Darstellen von Informationen über die Simulation. Die Funktion des weiteren in den Abbildungen 9.1 und 9.2 erkennbaren graphischen Moduls *NewModelCreator* liegt in der Bereitstellung einer graphischen Schnittstelle, mit der es dem Benutzer ermöglicht wird, neue Simulations-Modelle zusammenzustellen und zu konfigurieren. Dieses geschieht losgelöst von dem durch das MainWin-Modul dargestellten Hauptfenster.

Die administrative Klasse *SimControl* stellt die Schnittstelle zwischen der Simulations-Umgebung und dem eingebetten Simulations-Modell dar, welches aus vom Modell-Framework bereitgestellten Komponenten besteht (vgl. Kapitel 8). Dazu werden die einzelnen Komponenten eines Modells in diesem Modul verwaltet, und anderen Modulen der Zugriff auf diese Komponenten ermöglicht. Ferner sind in diesem Modul bestimmte Funktionalitäten vorgesehen, die von anderen Modulen aufgerufen werden können, wenn bestimmte Aktionen für alle Komponenten des aktuellen Simulations-Modells durchzuführen sind.

In dem Modul *SimLogger* wird die geforderte Protokollierungs-Funktion realisiert. Da das Zusammenstellen der zu protokollierenden Daten der einzelnen Simulations-Elemente in bzw. von diesen geleistet wird, müssen in diesem Modul in erster Linie die Operationen zur Verfügung gestellt werden, mit denen diese zusammengestellten Daten von den Elementen geholt und in eine Datei geschrieben werden. Ferner müssen hier noch gewisse Rahmendaten in diese Datei geschrieben werden, wie z. B. auf welchen Simulations-Schritt sich die geschriebenen Daten beziehen, oder wann eine Simulation gestartet bzw. beendet wurde.

Mittels dem *SimTimer*-Modul wird der Simulations-Umgebung der Zeittakt einer laufenden Simulation vorgegeben. Um dieses leisten zu können läuft dieses Modul bzw. die Implementierung dieses Moduls in einem eigenen Thread, überprüft dort regelmäßig, ob es notwendig ist, einen neuen Simulations-Schritt anzustoßen, und initiiert diesen gegebenenfalls. Ferner wird in diesem Modul die aufgestellte Anforderung nach Beschleunigung der Simulations-Zeit realisiert.

Das Modul *SimulatorRoofService* (nicht zu verwechseln mit dem Modul SimulatorRoof) dient der Rea-

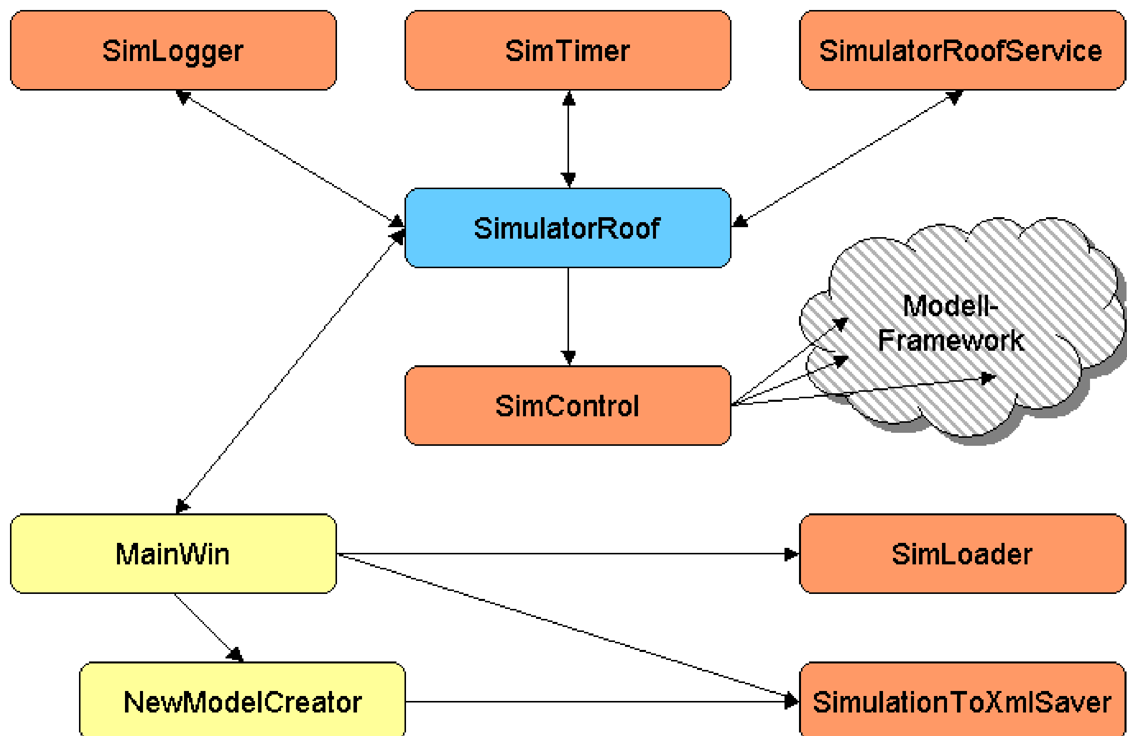


Abbildung 9.2: Interaktion der Module

lisierung des Web Services, der von der Simulations-Umgebung bereitgestellt werden soll. Dieser Service gestattet es, die Simulations-Umgebung „fernzubedienen", sowie Informationen zu dem aktuellen Simulations-Lauf abzufragen. Dazu werden mittels diesen Moduls sowohl das bereitzustellende Device, als auch der auf diesem Device bereitgestellte Service realisiert.

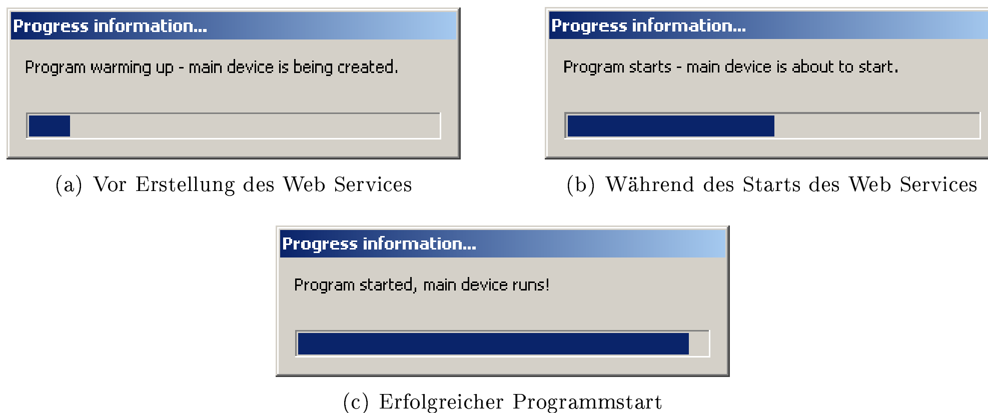
Die beiden Module *SimulationToXMLSaver* und *SimLoader* dienen der Realisierung der für das Speichern und Laden notwendigen Dateioperationen. Da Simulations-Modelle XML-basiert abgespeichert werden, befinden sich in diesen Modulen ebenfalls noch Funktionalitäten, die zum korrekten Erstellen von XML-Schema-konformen Dokumenten dienen, bzw. beim Laden XML-Dokumente korrekt auswerten können. Für diese Verarbeitung der XML-Daten wurde dabei überwiegend eine externe Bibliothek verwendet.

9.2 Details der Module

Hier sollen nun die Aufgaben und Details der einzelnen Module erläutert werden. Ebenfalls wird dabei auf ihr Zusammenspiel eingegangen. Der Übersichtlichkeit halber wird bei Methodenaufrufen größtenteils auf die explizite Nennung von vorhandenen Übergabe-Parametern verzichtet; sind bei einer Methode Parameter zu übergeben, so wird dieses durch das Hinzufügen von „. . ." im Methodenaufruf angedeutet (z. B. `methodenName(...)`).

9.2.1 SimulatorRoof - Modul

Wie zuvor erwähnt stellt dieses Modul bzw. die Java-Klasse `SimulatorRoof` das *Dach* der Simulations-Umgebung dar. Über dieses Modul wird der Großteil der Kommunikation und Interaktion zwischen den Modulen abgewickelt (siehe Abbildung 9.2). Ferner werden aus dieser Klasse heraus die Instanzen



(a) Vor Erstellung des Web Services

(b) Während des Starts des Web Services

(c) Erfolgreicher Programmstart

Abbildung 9.3: Startbildschirm mit Fortschrittsanzeige

der übrigen Modul-Klassen erzeugt, und dabei insbesondere verwaltet, dass von bestimmten Modulen (wie z. B. dem *SimTimer*) immer nur genau eine Instanz exisitiert. Die Klasse `SimulatorRoof` selbst ist ebenfalls ein *Singleton*, d. h. wird bei einer Ausführung des Programms in einer virtuellen Maschine genau einmal instanziiert. Diese Instanz stellt diese Klasse für andere Klassen über ihre öffentliche Methode `SimulatorRoof.getInstance()` zur Verfügung.

Neben dieser Methode werden von dieser Klasse noch zahlreiche weitere Funktionen bzw. Methoden bereitgestellt. Diese sollen hier nun der Übersichtlichkeit halber kurz in Form einer Aufzählung dargestellt und jeweils kurz erläutert werden.

- `startSimulation(...)`: dient dem starten eines neuen Laufes der aktuell geladenen Simulation. Dabei ist es möglich, dieser Methode ganz allgemein gehalten ein `java.lang.Object` mit zu übergeben um z. B. anzuzeigen, von wo der Aufruf erfolgte. Bei einem Aufruf der Methode werden zunächst alle Variablen dieser Klasse, die Informationen über einen Simulations-Lauf verwalten, mittels eines Aufrufes der Methode `resetVariables()` zurückgesetzt. Anschließend werden die bool'schen Variablen, die den aktuellen Laufzustand verwalten (wie z. B. `simulationWasStarted` oder `simulationIsPaused`), mittels der dafür vorgesehenen Methoden gesetzt. Ferner wird ein neuer *SimTimer* angelegt und gestartet (vgl. Abschnitt 9.2.4). Als letzter Schritt wird die Methode `fireSimulationEvent(...)` aufgerufen, um ein *Simulations-Event* des Typs `SIMULATION_STARTED` auszulösen.

- `stopSimulation(...)`: mittels eines Aufrufes dieser Methode kann ein laufender Simulations-Lauf beendet werden. Dafür wird zunächst der laufende *SimTimer* mittels der dort bereitgestellten Methode `stop()` angehalten, und ein eventuell ebenfalls laufender *SimLogger* wird mittels seiner Methode `closeWriter()` angewiesen, das Protokollieren abzuschließen. Daran anschließend werden die bereits erwähnten bool'schen Variablen zur Verwaltung des Laufzustandes auf die entsprechenden Werte gesetzt, und abschließend mittels `fireSimulationsEvent(...)` ein `SIMULATION_STOPPED`-Event ausgelöst.

- `pauseSimulation(...)` / `resumeSimulation(...)`: diese Methoden ermöglichen es, eine *laufende* Simulation zu pausieren, oder eine angehaltene Simulation wieder fortzusetzen. Die einzigen Aktionen, die in diesen Methoden durchgeführt werden, sind dabei das Setzen einer bool'schen Variable mittels `setSimulationIsPaused(...)` auf entsprechend `true` oder `false`, und das Auslösen eines Simulations-Events vom Typ `SIMULATION_PAUSED` oder `SIMULATION_RESUMED`. Das Auswerten, ob ein Simulations-Lauf unterbrochen ist, geschieht anhand des hier gesetzten bool'schen Parameters in der Methode `timestepOccurred()`.

- `setTimeaccelerationFactor(...)` / `getTimeaccelerationFactor()`: hiermit kann der Zeitbe-

schleunigungs-Faktor, der auf die laufende Simulation angewendet werden soll, gesetzt oder abgefragt werden. Abgefragt wird er z. B. von dem Modul *SimTimer*, wenn dieses ermitteln muss, wie viele Simulationsschritte pro Zeitschritt auszuführen sind (vgl. Abschnitt 9.2.4), gesetzt wird er entweder aus dem *MainWin* oder dem *SimulatorRoofService* heraus. In der Methode zum Setzen dieses Faktors wird ferner noch ein Simulations-Event des Typs `SIMULATION_TIMEACCELERATIONFACTOR_CHANGED` ausgelöst.

- `getPerformedSimulationsSteps()` / `getSimulationRunningTime()` / `getSimulatedTime()`: im Modul *SimulatorRoof* werden u. a. Variablen benutzt, die z. B. die Anzahl der im aktuellen Simulations-Lauf absolvierten Simulations-Schritte oder die vergangene simulierte Zeit (ganzzahlig als Millisekunden) verwalten. Die Werte dieser Variablen lassen sich mittels dieser Methode aus anderen Modulen heraus abfragen (z. B. benutzt das *MainWin* diese Methode, um die Anzahl der Simulations-Schritte anzeigen zu können).

- `getSimulationWasStarted()` / `getSimulationWasStopped()` / `getSimulationIsRunning()`: diese Methoden dienen dazu, die im SimulatorRoof zum Erkennen des Zustandes eines Simulations-Laufes verwalteten bool'schen Variablen direkt abfragen zu können. Beispielsweise liefert die Methode `getSimulationWasStarted()` den Wert `true` zurück, wenn die aktive Simulation bereits einmal gestartet wurde. Dafür ist es unerheblich, ob dieser Lauf inzwischen möglicherweise unter- oder abgebrochen wurde. Wurde eine Simulation gestartet und anschließend wieder gestoppt, so würden sowohl `getSimulationWasStarted()` als auch `getSimulationWasStopped()` den Wert `true` liefern. Lediglich `getSimulationIsRunning()` würde `false` liefern. Aus diesem *3-Tupel* lässt sich somit dann der Zustand einer Simulation präzise ableiten.

- `getErrorMode()` / `setErrorMode(...)`: bei dem *ErrorMode* handelt es sich um einen bestimmten Zustand, in den ein Simulations-Lauf versetzt wird, wenn an einem oder mehreren Elementen des Simulations-Modells Fehler aufgetreten sind. Befindet sich der Simulations-Lauf in diesem ErrorMode, so werden keine weiteren Simulations-Schritte durchgeführt, bis diese Fehler entweder (z. B. durch ein extern laufendes *Fehlertoleranzverfahren*) behandelt sind, oder aber der Benutzer die Anweisung gibt, diese Fehler zu ignorieren und mit der Simulation fortzufahren.

- `loadingFromFileSuccessful(...)`: ein Aufruf dieser Methode stellt den Abschluss eines jeden Ladevorganges (vgl. Abschnitt 9.2.7) dar, um dem Modul SimulatorRoof anzuzeigen, dass dieser Ladevorgang entweder erfolgreich war (Übergabeparamter `true`), oder aber fehlgeschlagen ist (`false`). War der Ladevorgang erfolgreich, so werden zunächst die zur Verwaltung des Simulations-Zustandes benutzten bool'schen Variablen auf ihre aktuellen Werte gesetzt, und eine neue Instanz des Moduls *SimLogger* für diese Simulation erstellt (vgl. Abschnitt 9.2.3). Im Anschluss wird überprüft, ob in dem neu geladenen Simulations-Modell das *Property* zum Ein- / Ausschalten der Web Services für alle Modell-Elemente exisitiert. Existiert dieses, wird es ausgewertet, und die den Status der Web Services aller Modell-Komponenten verwaltende Variable im SimulatorRoof auf den entsprechenden Wert (Services aktiviert oder deaktiviert) gesetzt. In einem letzten Schritt wird ein *Simulations-Event* des Typs `LOADING_SUCCESSFUL` ausgelöst; dieses bekommt als *Payload* den Namen des erfolgreich geladenen Modells. War das Laden hingegen nicht erfolgreich, d. h. war der Eingabeparameter dieser Methode `false`, so werden die Zustand-Verwaltungsvariablen auf die dafür passenden Werte gesetzt, und ein *Simulations-Event* des Typs `LOADING_FAILED` ausgelöst (dieses veranlasst z. B. die Klasse `MainWin`, eine entsprechende Fehlermeldung auszugeben). Außerdem wird die *SimControl* explizit mittels der von ihr bereitgestellten Methode `resetSimControl()` zurückgesetzt. Dieses geschieht, da nicht auszuschließen ist, dass im Rahmen des fehlgeschlagenen Ladevorgangs doch bestimmte Elemente in der SimControl gesetzt wurden.

- `setWriteLogfile(...)` / `getWriteLogfile()`: mittels dieser Methoden wird der Parameter gesetzt bzw. abgefragt, der anzeigt, ob aktuell ein Protokoll für die aktive Simulation geschrieben werden soll. Aufgerufen werden können beide Methoden dabei zu beliebigen Zeitpunkten, unabhängig davon, ob gerade eine Simulation durchgeführt wird.

- `setExecutionControlMode(...)` / `getExecutionControlMode()`: der *ExecutionControlMode* gibt an, wie, d. h. über welches Benutzer-Interface, ein Simulations-Lauf gestartet wurde. Bei dem derzeitigen Stand der Implementierung gibt es zwei Möglichkeiten, einen Simulations-Lauf zu starten: zum einen über die graphische Benutzerschnittstelle (vgl. Abschnitt 9.2.8), und zum anderen über den von der Simulations-Umgebung bereitgestellten *Web Service*. Mittels der Methode `setExecutionControlMode(...)` wird dieser Modus im Rahmen des Aufrufes der Methode `startSimulation(...)` mit gesetzt. Ausgelesen wird dieser Modus mittels `getExecutionControlMode()` in dem Fall, dass an einem Element des Simulations-Modells ein Fehler auftritt. Dann ist zu unterscheiden, ob die Simulation über das graphische Benutzerinterface gesteuert und kontrolliert wird, oder aber, ob die Kontrolle mittels des Web Services realisiert wird. In diesem Fall *darf* ausdrücklich keine blockierende Fehlermeldung mittels der graphischen Benutzerschnittstelle erfolgen, sondern es ist auf eine Reaktion auf diesen Fehler aus den Web Services heraus zu warten.

- `createSimulatorRoofDpwsDevice(...)` / `startSimulatorRoofDpwsDevice()`: diese Methoden dienen dem Erstellen und dem Starten des *Web Services* der Simulations-Umgebung. Dazu muss zunächst die Methode `createSimulatorRoofDpwsDevice(...)` aufgerufen werden. Diese Methode erzeugt eine neue Instanz der Klasse `SimulatorRoofDevice`, welche ein *Device* im Sinne der *DPWS-Spezifikation* (vgl. Abschnitt 3.3) darstellt. Bei der Instanziierung dieser Klasse wird diesem Device direkt die benötigte Instanz der Klasse `SimulatorRoofService` als bereitgestellter *Service* hinzugefügt. Der Aufruf dieser Methode erfolgt direkt beim Start des Programmes. Daran anschließend erfolgt auch der Aufruf der Methode `startSimulatorRoofDpwsDevice()`, welcher das Device dann startet.

- `fireSimulationEvent(...)` / `addSimulationListener(...)`: die hier schon mehrfach referenzierte Methode `fireSimulationEvent(...)` dient dazu, *Events* auszulösen. Diese Events sind vom Typ `simulator.util.eventing.SimulationEvent`, und werden von allen Klassen empfangen, die das *Interface* `ISimulationListener` implementieren, und sich mittels der Methode `addSimulationListener(...)` selbst zur Liste der Listener hinzugefügt haben (vgl. hierzu auch Abbildung 9.7). Das Auslösen von Events, die ein beliebiges `java.lang.Object` als *Payload* transportieren können, ist für das SimulatorRoof-Modul die einzige Möglichkeit, Daten an Module zu schicken, auf die aus Gründen der Modularität nicht direkt zugegriffen werden kann. Dieses sind z. B. die Module *MainWin* und *SimulatorRoofService*.

- `timestepOccurred()`: bei dieser Methode handelt es sich um die Schnittstelle zwischen dem *SimTimer*-Modul und der Simulations-Umgebung. Wie in Abschnitt 9.2.4 beschrieben läuft der SimTimer in einem *seperaten Thread*, und ruft `timestepOccurred()` zu jedem Zeitpunkt auf, zu dem ein Simulations-Schritt auszulösen ist. In dieser Methode werden nun alle für die Durchführung eines Simulations-Schrittes notwendigen Teilschritte sequentiell durchgeführt. Dieser Ablauf ist stark vereinfacht und mit Pseudo-Code in Abbildung 9.4 dargestellt. Zunächst wird dabei überprüft,ob sich die Simulation im *ErrorMode* (s. o.) oder im *PopupMode* befindet (letzteres bedeutet, dass wegen eines Rechtsklicks auf ein Simulations-Element dort ein Kontext-Menü geöffnet ist). In diesem Fall dürfte kein Simulations-Schritt durchgeführt werden, und der Methodenaufruf würde abgebrochen werden. Andernfalls darf der Simulations-Schritt initiiert werden. Dazu wird zunächst ein *Simulations-Event* des Typs `TIMESTEP_INITIATED` ausgelöst. Daran anschließend wird für die Elemente des Simulations-Modells der eigentliche Simulations-Schritt durchgeführt. Dieses geschieht mittels eines Aufrufs der Methode `nextStep()` des Moduls *SimControl*. Nach Abarbeitung dieses Aufrufes haben alle Simulations-Elemente genau einen weiteren Simulations-Schritt abgearbeitet, und es erfolgen zunächst Aktualisierungen der im Modul SimulatorRoof verwalteten Variablen wie z. B. die Anzahl der durchgeführten Simulations-Schritte oder der simulierten Zeit. Darauf folgend wird überprüft, ob für den aktuellen Schritt eine Protokollierung erfolgen soll. Ist diese Option aktiv wird diese Protokollierung mittels eines Aufrufes der Methode `writeLog(...)` des Moduls *SimLogger* durchgeführt (vgl. Abschnitt 9.2.3). Dieser Methode werden dabei eventuell in diesem Schritt angefallene Fehler übergeben, die diese entsprechend verarbeitet. Sollte die Protokollierung für

diesen Schritt deaktiviert gewesen sein werden an dieser Stelle stattdessen die von den Simulations-Elementen dennoch zusammengestellten Protokoll-Informationen gelöscht. Dieses geschieht, indem die von der *SimControl* bereitgestellte Liste der Simulations-Elemente durchlaufen wird, und bei jedem dieser Elemente ein Aufruf der Methode `clearLoggingData()` erfolgt. Dieses schließt die für die Protokollierung dieses Schrittes nötigen Aktionen ab. An dieser Stelle wird nun ein *Simulations-Event* des Typs `TIMESTEP_OCCURRED` ausgelöst. Dieses bewirkt z. B. im Modul *MainWin*, dass die Visualisierung des zuletzt durchgeführten Schrittes ausgelöst wird (vgl. Abschnitt 9.2.8). Daran anschließend wird überprüft, ob in dem zuletzt durchgeführten Schritt möglicherweise *Fehler* an einem oder mehreren Simulations-Elementen aufgetreten sind. War dieses der Fall, so wird ein weiteres *Simulations-Event* ausgelöst, welches den Typ `ERROR_OCCURRED` hat. Dieses Event bzw. die damit verbundenen Fehler werden abhängig von der Art des *ExecutionControlMode* entweder durch das Modul *MainWin* mittels eines Benutzer-Dialoges oder durch das Modul *SimulatorRoofService* durch die Anwendung einer Fehlerbehandlung verarbeitet. Sollte dabei festgestellt werden, dass die aufgetretenen Fehler derart schwerwiegend sind, dass sie nicht zu reparieren waren, muss der Simulations-Lauf abgebrochen werden. Abhängig davon, ob die Fehler aufgelöst werden konnten, wird dann im nächsten Schritt ein `ERRORS_ALL_SOLVED` oder ein `ERROR_UNSOLVABLE` - *Simulations-Event* ausgelöst. Letzteres würde zur Beendigung des Simulations-Laufes führen. Konnten hingegen die Fehler behandelt werden, oder traten in diesem Schritt keine Fehler auf, so wird als letzte für einen Simulations-Schritt notwendige Aktion ein *Simulations-Event* des Typs `TIMESTEP_COMPLETED` ausgelöst. Dieses schließt den Simulations-Schritt ab.

Neben diesen gibt es noch zahlreiche weitere Methoden im Modul *SimulatorRoof*. Da es sich bei diesen Methoden zum überwiegenden Teil um Methoden mit einer sehr einfachen Funktion handelt, sollen diese hier nicht explizit aufgeüfhrt und erläutert werden. Statt dessen sei hier verwiesen auf die Kommentare bzw. die Dokumentation im Quelltext.

9.2.2 SimControl - Modul

Eine Instanz des Moduls *SimControl* dient als Schnittstelle zwischen der Simulations-Umgebung und den konkreten Instanzen der vom Modell-Framework bereitgestellten Simulations-Elemente (vgl. Kapitel 8.3). Grundlegend dafür wird in diesem Modul eine Liste verwaltet, in der z. B. beim Laden eines Modells jede Instanz eines dort existierenden Simulations-Elements angemeldet wird. Dieses geschieht mittels der von der SimControl bereitgestellten Methode `addSimElement(...)`. Sind alle Elemente eines Modells zu der entsprechenden SimControl hinzugefügt, so wird diesen mit einem Aufruf der Methode `createAllConnectionsBetweenElements()` mitgeteilt, dass sie mit dem Erstellen ihrer Verbindungen untereinander beginnen können. Anschließend dient die SimControl während der laufenden Simulation primär dazu, eine Schnittstelle bereitzustellen, mittels der den Elementen der Simulations-Fortschritt angezeigt wird. Dieses geschieht, indem das Modul, das für das Verwalten des Simulations-Fortschrittes zuständig ist, die Methode `nextStep()` aufruft, welche dann entsprechend alle Elemente in dieser Sim-Control veranlasst, genau einen Simulationsschritt durchzuführen. Dabei liefert dann ein Element, bei dem in diesem Schritt möglicherweise ein Fehler aufgetreten ist, eine entsprechende Meldung zurück. Diese Meldungen werden von der SimControl gesammelt, und an das diese Methode aufrufende Modul zurückgegeben.

Verwaltet werden die Instanzen der Simulations-Elemente, die in ihrer Gesamtheit das aktuelle Simulations-Modell ausmachen, ausschließlich in der Liste in diesem Modul. Daher ist es naheliegend, diesem Modul auch die Verwaltung von Daten zu übergeben, die auf eine konkrete Simulation bezogen sind. Dieses sind z. B. Simulations-Eigenschaften wie der Name des Modell-Autoren oder die Modell-Version. Daneben sind in dieses Modul noch diverse Verwaltungsfunktionen ausgegliedert, die, wenn sie ausgeführt werden, immer für alle Elemente des aktuellen Simulations-Modells durchgeführt werden müssen. Beispiele dafür sind das Ein- oder Ausschalten der Web Services für alle Elemente mittels der Methoden `createDpwsServices(...)` bzw. `stopAllDpwsServices(...)`, oder z. B. die Überprüfung, ob eine Änderung eines Bezeichners eines der Modell-Elemente zulässig ist. Schließlich stellt dieses Modul ebenfalls

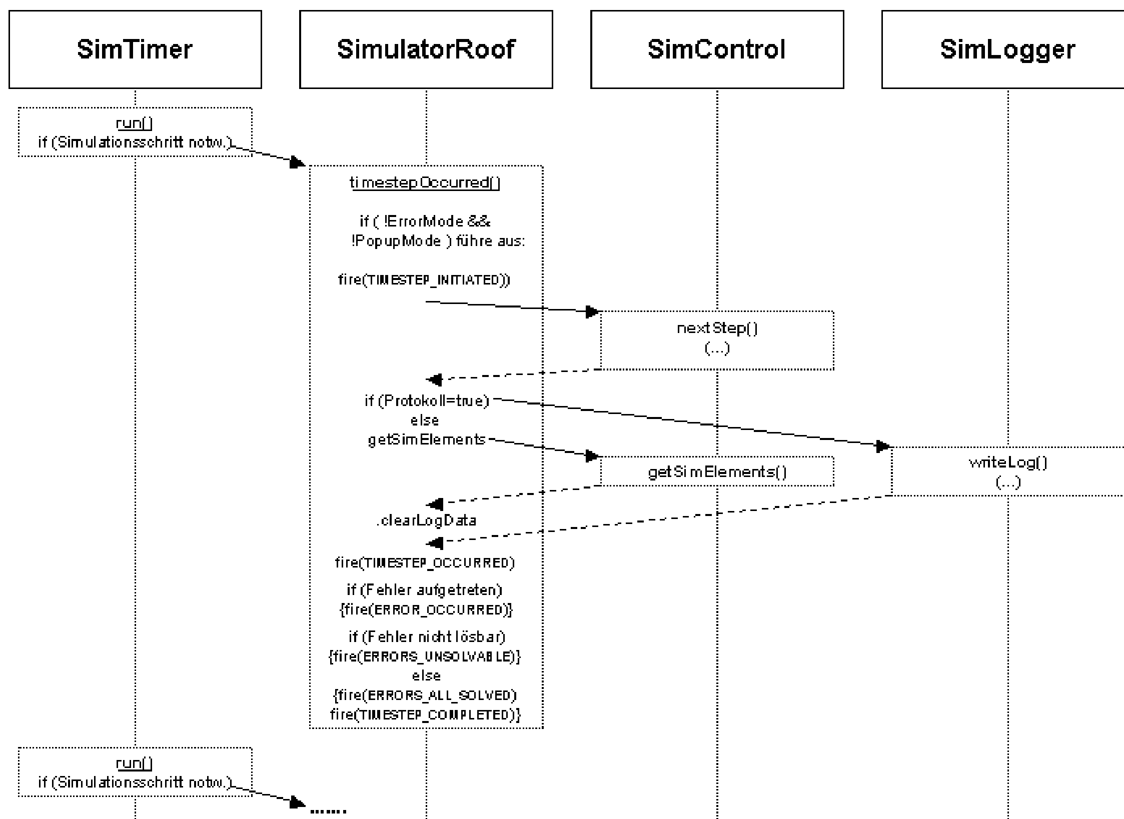


Abbildung 9.4: Ablauf eines Simulations-Schrittes

noch die Methode `getSimElements()` zur Verfügung, mittels der anderen Modulen Zugriff auf die Liste der Instanzen der Simulations-Elemente gewährt werden kann.

9.2.3 SimLogger - Modul

Das Modul *SimLogger* hat die Aufgabe, die während eines Simulations-Schrittes anfallenden, zu protokollierenden Daten zu erfassen und sukzessive in eine Datei zu schreiben. Dabei wird für jeden neu gestarteten Simulations-Lauf eine neue Instanz des SimLoggers angelegt, welche dann für das Schreiben der Simulations-Protokolldaten genutzt wird. Im Rahmen dieser Instanziierung wird dann bereits der Name der zu schreibenden Protokolldatei festgelegt.

Die zentrale, zum Schreiben der Protokolldaten dienende Methode, die ein SimLogger bereitstellt, ist die Methode `writeLog(...)`. Diese muss nach jedem absolvierten Simulations-Schritt von dem diese Schritte steuernden Modul aufgerufen werden. In einem ersten Schritt nach dem Aufruf dieser Methode überprüft die Instanz des SimLoggers, ob die zu schreibende Protokolldatei bereits initialisiert wurde. Ist dieses nicht der Fall, so erfolgt die Initialisierung, indem ein `BufferedWriter` für die zu schreibende Datei angelegt wird. Ferner werden dabei grundlegende *Header-Daten* wie z. B. der Zeitpunkt des Starts des Simulations-Laufes oder der Name der benutzten Simulations-Datei in dieser BufferedWriter geschrieben. Ist die Protokolldatei entsprechend initialisiert, so wird in der weiteren Abarbeitung der `writeLog(...)`-Methode zunächst die Anzahl der absolvierten Simulationsschritte in die Protokolldatei geschrieben. Danach wird überprüft, ob bei dem Aufruf der Methode ganz allgemein Informationen über eventuell aufgetretene Fehler im aktuellen Simulationsschritt übergeben wurden. War dieses der Fall, so wird eine entsprechende allgemeine Information eingefügt. Anschließend wird von der Instanz der *Sim-*

Control die aktuelle Liste der Simulationselementen angefordert. Von dieser wird eine Kopie angelegt, welche dann gemäß der Namen der Elemente sortiert wird. Dieses stellt sicher, dass die Protokollinformationen der einzelnen Elemente immer in der gleichen Reihenfolge geschrieben werden. Danach iteriert der SimLogger über diese sortierte Liste, und fragt bei jedem Element an, ob für dieses Protokolldaten für den aktuellen Schritt geschrieben werden sollen. Ist dieses der Fall werden Informationen bezüglich des Namens und des Typs dieses Simulations-Elementes an die Protokolldatei angehängt, und anschließend die von dem Element selbst für diesen Schritt zusammengestellten Protokolldaten abgefragt und dem Protokoll hinzugefügt. Dieses schließt Informationen über Fehler in diesem Element mit ein. Abschließend wird dem Element mitgeteilt, dass es die gerade abgefragten Protokolldaten löschen kann, und zum nächsten Element übergegangen.

Jede SimLogger-Instanz wird nach Abschluss des Simulations-Laufes, für den sie erzeugt worden ist, beendet. Dieses geschieht mittels der bereitgestellten Methode `closeWriter()`. Dabei werden zunächst noch einige Informationen über die Simulation selbst dem Protokoll hinzugefügt (z. B. die Gesamtanzahl der absolvierten Simulations-Schritte oder die Uhrzeit, zu der der Simulations-Lauf gestoppt wurde), eventuell noch nicht in die Datei geschriebene Daten gespeichert, und der benutzte `BufferedWriter` geschlossen.

9.2.4 SimTimer - Modul

Das Modul *SimTimer* hat die Funktion, dem Modul SimulatorRoof im Falle einer laufenden Simulation den Zeittakt vorzugeben. Um dieses gewährleisten zu können, erweitert die Klasse SimTimer zunächst `java.util.TimerTask`. Wird nun eine neue Simulation gestartet, so erzeugt das Modul SimulatorRoof eine neue Instanz dieser Klasse, und ruft anschließend explizit die bereitgestellte Methode `start()` auf. Bei diesem Aufruf speichert sich die Instanz des SimTimers zunächst die aktuelle Systemzeit, und erstellt anschließend eine Instanz eines `java.util.Timer`. Diesen konfiguriert die SimTimer-Instanz derart, dass er sie selbst mit einer bestimmten, fest eingestellten Rate pro Zeiteinheit aufruft, und startet ihn. Dabei läuft dieser `Timer` und somit auch die Instanz des Moduls SimTimer in einem eigenen, vom Haupt-Thread der Simulations-Umgebung unabhängigen Thread (vgl. Kapitel 9.3).

Nach Abarbeitung der `start()`-Methode wird der SimTimer nun in einem festen Zeitintervall aufgerufen. Ein solcher Aufruf geschieht durch Anstossen der `run()`-Methode, die der SimTimer von `TimerTask` geerbt hat. In einem ersten Schritt wird dort überprüft, ob der Haupt-Thread der Simulations-Umgebung und somit die Instanz des Moduls SimulatorRoof noch ordnungsgemäß arbeitet (wegen der Nebenläufigkeit der beiden Threads ist es theoretisch möglich, dass dieser Thread bereits beendet wäre - vgl. Kapitel 9.3). Arbeitet die Instanz des SimulatorRoofs ordnungsgemäß, so kann mit dieser interagiert werden, ohne dass kritische Probleme wie z. B. ein Nullpointer beim Zugriff auftreten. In einem nächsten Schritt wird überprüft, ob es notwendig ist, in der laufenden Simulation mindestens einen Simulations-Schritt auszulösen. Diese Überprüfung erfolgt, indem die beim letzten Auslösen von mindestens einem Simulations-Schritt gespeicherte Systemzeit mit der aktuellen Systemzeit verglichen wird. Dieses ist notwendig, da mittels Konstanten innerhalb des Programms (vgl. `SimConstants` in Abschnitt 9.2.10) die Aufruf-Frequenz des SimTimers unabhängig von der Dauer eines Echtzeit-Simulationsschrittes eingestellt werden kann. Eventuell muss dementsprechend gar nicht bei jedem Aufruf der `run()`-Methode des SimTimers auch ein weiterer Simulations-Schritt ausgelöst werden. Dieses führt zu einer großen Flexibilität in der Zeitsteuerung.

Wurde festgestellt, dass das Auslösen von mindestens einem neuen Simulations-Schritt nötig ist, so wird zunächst beim SimulatorRoof-Modul angefragt, ob die Simulation auch wirklich noch läuft. Dieses muss geschehen, da nicht auszuschließen ist, dass der Simulations-Lauf in der Zeitspanne seit dem letzten Auslösen eines Simulations-Schrittes angehalten worden ist. Nur wenn auch diese Bedingung erfüllt ist beginnt der SimTimer damit, mittels eines entsprechenden Methodenaufrufs im Modul SimulatorRoof einen oder mehrere Simulationsschritte auszulösen. Dabei erfolgt an dieser Stelle die Realisierung der Forderung nach *Beschleunigung der Simulationszeit* (vgl. Kapitel 6.3): ist hier keine Beschleunigung der Simulationszeit eingestellt, so wird hier genau ein Schritt ausgelöst. Ist hingegen der *Zeitbeschleunigungsfaktor*

auf einen Wert > 1 gesetzt, so werden an dieser Stelle sequentiell entsprechend viele Simulations-Schritte angestoßen.

Nachdem hier die entsprechende Anzahl von Simulations-Schritten abgearbeitet worden ist, ist der aktuelle Durchlauf der `run()`-Methode des SimTimers weitestgehend abgeschlossen. Es erfolgt noch eine Aktualisierung des abgespeicherten Zeitpunktes der letzten Ausführung von Simulations-Schritten.

Beendet wird eine SimTimer-Instanz mittels des Aufrufes der Methode `stop()`. Dieser Aufruf, der beim Beenden eines Simulations-Laufes erfolgt, sorgt dafür, dass die angelegte Instanz des `java.util.Timer` gestoppt wird. Somit erfolgen keine weiteren Aufrufe der `run()`-Methode dieses Moduls mehr, d. h. der SimTimer ist angehalten.

9.2.5 SimulatorRoofService - Modul

Das *SimulatorRoofService*-Modul dient der Bereitstellung des in Abschnitt 6.4.1 geforderten Web Service für die Steuerung und Kontrolle der Simulations-Umgebung. Die Realisierung dieses Web Services erfolgt mittels dem in Abschnitt 7.3.3 vorgestellten *WS4D-J2ME*-Stacks.

Dazu ist das Modul SimulatorRoofService zunächst in zwei Teile zu unterteilen: zum einen in die Klasse `SimulatorRoofDevice`, und zum anderen in die Klasse `SimulatorRoofService.java`. Die erste Klasse realisiert dabei das *Device* im Sinne der DPWS-Spezifikation, und die zweite Klasse realisiert den *Service*, der von diesem Device zur Verfügung gestellt wird. Gestartet wird das Device und somit der Service automatisch bei jedem Start des Programmes.

Erstellt und gestartet wird das Device dabei mittels Methodenaufrufen in dem Modul SimulatorRoof. Die Methode `createSimulatorRoofDpwsDevice(...)` dieses Moduls erstellt zunächst das Device, und ein anschließend erfolgender Aufruf der Methode `startSimulatorRoofDpwsDevice()` startet dieses. Die beiden Klassen `SimulatorRoofDevice` und `SimulatorRoofService` sollen hier nun noch kurz getrennt voneinander betrachtet und erläutert werden.

SimulatorRoofService

Diese Klasse realisiert wie beschrieben den *Service*, der auf dem *Device* bereitgestellt wird. Dazu erweitert die Klasse `SimulatorRoofService` die Klasse `org.ws4d.java.service.HostedService`, und implementiert gleichzeitig das Interface `ISimulationListener.java` (vgl. Abschnitt 9.2.10). Ferner fügt sich der SimulatorRoofService der Liste der Simulation-Listeners des SimulatorRoof-Moduls hinzu. Dieses Anlegen des SimulatorRoofService als Listener auf das SimulatorRoof-Modul ist notwendig, da dieses die einzige Möglichkeit der Kommunikation aus dem SimulatorRoof in Richtung seines Services ist. Direkte Aufrufe von Methoden o. ä. des SimulatorRoofService aus dem SimulatorRoof- oder anderen Modulen heraus sind wegen der Wahrung der Modularität nicht vorgesehen. Nach dem Einrichten des SimulatorRoofService als Listener wird für jede der von dem SimulatorRoofService-Modul bereitgestellten Funktionalitäten eine entsprechende `org.ws4d.java.service.Action` erstellt, und die notwendigen Ein- oder Ausgabe-Parameter (als `org.ws4d.java.service.Parameter`) werden zu diesen Actions hinzugefügt.

Auf die einzelnen Actions soll an dieser Stelle noch nicht näher eingegangen werden, da es sich bei diesen Actions um die Funktionalitäten handelt, die dieser Service zur Verfügung stellt. Stattdessen sei hier verwiesen auf Abschnitt 9.4, in welchem explizit auf den für die Simulations-Umgebung bereitgestellten Web Service eingegangen wird.

SimulatorRoofDevice

Wie beschrieben realisiert die Klasse `SimulatorRoofDevice` das *Gerät* oder *Device*, welches den Web Service zur Kontrolle der Simulations-Umgebung bereitstellt (vgl. dazu die in Abbildung 3.6 dargestellte Struktur eines Devices mit Services und einem kommunizierenden Client). Dazu erweitert diese Klasse

die vom *WS4D-J2ME*-Stack vorgegebene Klasse `org.ws4d.java.service.HostingService`, erstellt eine neue Instanz des `SimulatorRoofService`, und fügt diese Instanz dem *HostingService* hinzu.

9.2.6 SimulationToXmlSaver - Modul

Die Aufgabe des *SimulationToXmlSaver-Moduls* liegt darin, ein Simulations-Modell aus der Simulations-Umgebung heraus in eine *XML-basierte* Datei zu speichern. Dafür muss zunächst eine Instanz dieses Moduls erzeugt werden, bei der direkt der Name der anzulegenden Datei mit zu übergeben ist. Neben diesem Dateinamen kann auch die Instanz des Moduls SimControl mit übergeben werden, mittels der das zu speichernde Simulations-Modell verwaltet wird. Dieses wird benutzt, wenn ein neues, mittels dem NewModelCreator-Modul erstelltes Modell gespeichert werden soll (vgl. 9.2.9). Wird keine explizite SimControl mit übergeben, so benutzt dieses Modul die SimControl, auf die mittels der `getSimControl()`-Methode des SimulatorRoof-Moduls Zugriff zu erhalten ist.

Ist eine Instanz dieses Moduls ordnungsgemäß erzeugt worden, so gibt es dort genau eine bereitgestellte Methode, die zum Starten des Speichervorganges aufzurufen ist, und die dann alle weiteren notwendigen Aktionen ausführt: dieses ist die Methode `save()`. Die Vorgehensweise dieser Methode lässt sich grob in zwei Teile aufteilen: im ersten Teil wird im Speicher ein XML-konformes Dokument erzeugt, welches im zweiten Teil in eine Datei geschrieben wird. Sämtliche Funktionen bezüglich der Verarbeitung des XML-Dokumentes erfolgen dabei mittels der externen Bibliothek *JDOM* (vgl. 7.3.2). Dabei wird zunächst ein neues `org.jdom.Document` mit einem XML-Wurzelelement erzeugt. Diesem Wurzelelement werden mögliche *Properties* des zu speichernden Simulations-Modells als Attribute hinzugefügt. Die Elemente des Modells werden als unmittelbare Kinder des Wurzelelementes eingefügt. Dazu wird über die Liste der Modell-Elemente iteriert, welche von der SimControl-Instanz bereitgestellt wird. Für jedes Element wird ein neues XML-Element angelegt; der Typ des Simulations-Elementes wird dabei als Attribut dieses XML-Elementes gesetzt. Die Details des aktuellen Modell-Elementes werden über dessen Methode `getSavingData()` angefordert, und als Kinder zu diesem Element hinzugefügt. Nachdem dieses für alle Elemente durchgeführt wurde, repräsentiert das XML-Dokument das komplette Simulations-Modell. Ein Beispiel für ein sehr einfaches solches als XML-Dokument organisiertes Simulations-Modell findet sich in Anhang B.

Abschließend wird dieses im Speicher abgebildete XML-Dokument mittels eines `org.jdom.output.XMLOutputter` in eine Datei geschrieben. Der Dateiname ist dabei der Name, der bei der Instanziierung des *SimulationToXmlSavers* festgelegt worden war.

9.2.7 SimLoader - Modul

Das *SimLoader-Modul* stellt das Gegenstück zu dem zuvor erläuterten SimulationToXmlSaver-Modul dar. Entsprechend dient dieses Modul dazu, zuvor in eine Datei gespeicherte Simulations-Modelle wieder zu laden. Dabei besteht das eigentliche Modul nicht nur aus der gleichnamigen Klasse `SimLoader`. Die Klasse `OpenSimulationAction` stellt ebenfalls mittels ihrer Methode `performLoading(...)` einen wichtigen Teil der in diesem Modul bereitgestellten Funktionalität zur Verfügung. Ein Aufruf dieser Methode erfolgt unter Übergabe des Namens der zu ladenden Datei. Dort wird eine Instanz der Klasse `SimLoader` erzeugt, und die Methode `loadFromXmlFormatFile(...)` dieser Instanz aufgerufen. Diese Methode hat nun die Aufgabe, die Datei auszulesen, und die XML-Daten zu verarbeiten.

Dazu wird dort wiederum die schon zuvor angesprochene Bibliothek *JDOM* benutzt. In einem ersten Schritt wird ein `org.jdom.input.SAXBuilder` erzeugt, und mittels diesem die übergebene Datei ausgelesen. Dabei wird automatisch das XML geparsed und auf mögliche Fehler in der Struktur untersucht. Existieren keine Fehler, so ist nach diesem Schritt ein `org.jdom.Document` mit den XML-Daten verfügbar. Von diesem wird nun zunächst das Wurzel-Element betrachtet und auf eventuell vorhandene *Attribute* untersucht, welche Properties des Modells darstellen und entsrpechend gesetzt werden. Anschließend werden die direkten Kinder-Elemente vom Typ `org.jdom.Element` des XML-Wurzelelementes, welche die

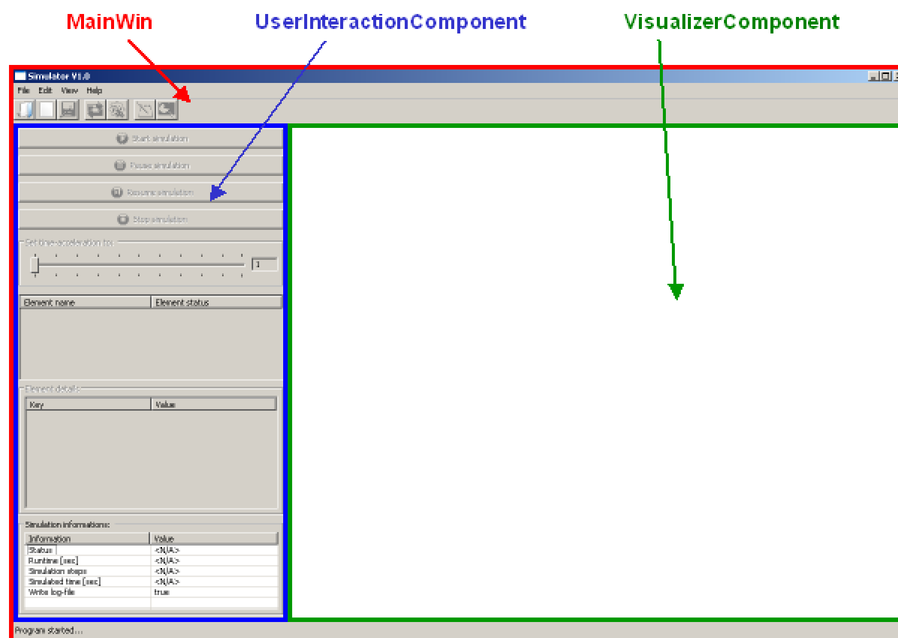


Abbildung 9.5: Module der graphischen Benutzerschnittstelle

einzelnen Simulations-Elemente repräsentieren ausgelesen. Über diese wird iteriert und dabei werden diese auf Korrektheit überprüft. Liegt ein gültiges Element vor, so wird dessen Attribut `TYPE` ausgewertet, mit welchem der *Typ* des Simulations-Elementes spezifiziert ist. Wird Typ des Simulations-Elementes nicht erkannt, oder existiert kein Attribut mit der Bezeichnung `TYPE`, so stellt dieses XML-Element kein gültiges Simulations-Element dar und wird ignoriert. Kann der Typ erkannt werden wird eine Instanz eines entsprechenden Simulations-Elementes erzeugt, und es werden die Kinder des XML-Elementes ausgelesen. Diese stellen die Parameter-Werte des Simulations-Elementes dar, und werden dem `loadingAssistent()` des zuvor erzeugten Simulations-Elementes übergeben (vgl. Abschnitt 8.4.2). Nachdem dieses mit allen Kinder-Elementen dieses XML-Elementes geschehen ist, ist das Simulations-Element fertig konfiguriert, und wird der SimControl mittels ihrer Methode `addSimElement(...)` hinzugefügt. Nachdem auf diese Weise alle Kinder-Elemente des XML-Wurzelelementes abgearbeitet sind, sind alle Simulations-Elemente in der SimControl verzeichnet.

Die weiteren Aktionen, die im Rahmen des Ladens eines Simulations-Modells nun noch ausgeführt werden müssen, gehen nun wieder von der Klasse `OpenSimulationAction` aus. Diese erkennt, ob ein Modell korrekt in die SimControl geladen werden konnte. War dieses der Fall initiiert diese noch das Erstellen der Verbindungen der Simulations-Elemente mittels eines Aufrufes der Methode `createAllConnections-BetweenElements()` der SimControl, aktiviert gegebenenfalls die Web Services der einzelnen Simulations-Elemente und veranlasst das Setzen von Verwaltungs-Variablen. Dieses schließt das Laden eines Simulations-Modells ab.

9.2.8 MainWin - Modul

Das MainWin-Modul dient der Bereitstellung der *graphischen Benutzerschnittstelle*. Um auch hier einen modularen Aufbau der Software zu gewährleisten, bei dem sich einzelne Komponenten ohne großen Aufwand durch andere Komponenten ersetzen lassen, ist dieses Modul intern aus mehreren Modulen aufgebaut, von denen jedes eine bestimmte Funktionalität bereitstellt. Abbildung 9.5 zeigt einen Überblick über die gesamte, von der Kombination dieser Unter-Module bereitgestellte graphische Benutzerschnittstelle. Ferner wird dort dargestellt, aus welchen Teil-Modulen das Gesamtmodul besteht. Diese werden

im Folgenden gesondert betrachtet.

MainWin

Diese Klasse stellt die *Basis* dar, auf der die gesamte graphische Benutzerschnittstelle aufsetzt. Wie in Abbildung 9.5 ersichtlich realisiert diese Klasse das eigentliche Fenster, in dem die weiteren graphischen Komponenten angesiedelt sind. Ferner werden die Menü-Leiste, die Toolbar zum schnellen Zugriff auf bestimmte Programmfunktionen (wie z. B. Laden oder Speichern), und die Statuszeile, in der Informationen zu der zuletzt ausgeführten Aktion angezeigt werden können, von dieser Klasse bereitgestellt.

Generell erweitert diese Klasse das von der SWT-Bibliothek (vgl. Abschnitt 7.3.1) bereitgestellte `org.eclipse.jface.window.ApplicationWindow`. Dieses stellt ein bereits recht weitentwickeltes Fenster zur Realisierung einer Anwendung mittels SWT dar. Dieses Fenster wird unterteilt mittels eines `org.eclipse.swt.custom.SashForm`, was es ermöglicht, das Größenverhältnis der beiden Komponenten in diesem Fenster zu verändern. Diese beiden Komponenten sind Instanzen der Klassen `UserInteractionComponent`, mittels welcher der Benutzer das Programm steuern kann, und `VisualizerComponent`, in welcher eine Simulation angezeigt wird. Letztere wird zusätzlich in eine `org.eclipse.swt.custom.ScrolledComposite` eingebettet, was wenn notwendig das Scrollen dieses Teils des Fensters ermöglicht.

Sehr wichtig ist es hier ferner noch, dass die Klasse `MainWin` einen `ISimulationListener` implementiert. Die Klasse MainWin „lauscht" also auf Events, die in dem Haupt-Modul SimulatorRoof ausgelöst werden, um auch hier die Modularität zu gewährleisten. Bei dieser Implementierung bleibt die Möglichkeit erhalten, die GUI-Komponente möglichst einfach gegen eine andere auszutauschen (oder komplett auf diese zu verzichten - vgl. Kapitel 6.6). Dafür war es notwendig, Methodenaufrufe aus dem SimulatorRoof-Modul direkt an das MainWin-Modul zu vermeiden. Statt dessen wurde dieses eben mittels eines entsprechenden Listeners gelöst, der auf für das MainWin „interessante" Ereignisse entsprechend reagiert. Diese Reaktion erfolgt in der Methode `simulationEventOccurred(...)` (vgl. auch Abschnitt 9.2.10 und Abbildung 9.7). Dort wird das mit übergebene `SimulationEvent` auf seinen Typ hin untersucht, und entsprechend diesem Typ die notwendigen Aktionen ausgelöst. Eine solche Aktion kann z. B. das Aktivieren oder Deaktivieren von Knöpfen zur Steuerung eines Simulations-Laufes sein, oder das Aktualisieren der Simulations-Visualisierung und der Simulations-Informationen nach einem erfolgten Simulations-Schritt.

UserInteractionComponent

Die Klasse `UserInteractionComponent` dient der Bereitstellung der zur Bedienung eines Simulations-Laufes notwendigen Komponenten und zum Anzeigen von Informationen über diesen Simulations-Lauf. Als Basis für diese Klasse dient eine `org.eclipse.swt.widgets.Composite`.

Bei den *Bedienelementen*, die von dieser Komponente bereitgestellt werden, handelt es sich im wesentlichen um Schalter zum Starten, Stoppen, Pausieren und Fortsetzen eines Simulations-Laufes. Diese sind implementiert als `org.eclipse.swt.widgets.Button`, und rufen bei erfolgter Auswahl die entsprechenden Methoden zum z. B. Starten oder Stoppen eines Simulations-Laufes im SimulatorRoof-Modul auf. Ferner wird nach jeder Auswahl eines solchen `Buttons` direkt die von dieser Klasse bereitgestellte Methode `enableOrDisableButtonsAndTimeScale()` aufgerufen. Diese überprüft, ob nach dieser ausgeführten Aktion Bedienelemente dieser Komponente aktiviert oder deaktiviert werden müssen. Zu diesen Elemente, die gegebenenfalls deaktiviert werden müssen, gehört ein weiteres Bedienelement, dass an dieser Stelle dem Benutzer zu Verfügung gestellt wird: ein Schieberegler zum Einstellen des gewünschten Zeitbeschleunigungs-Faktors. Dieser ist implementiert als `org.eclipse.swt.widgets.Scale`. Der minimale bzw. maximale Wert wird über Konstanten aus dem Hilfs-Modul *SimConstants* festgelegt.

Neben diesen Bedienelementen stellt die UserInteractionComponent auch Komponenten bereit, die der Anzeige von Informationen dienen. Die erste dieser Komponenten dient der Anzeige des aktuellen Status einer jeden im aktiven Simulations-Modell existierenden Komponente. Realisiert ist dieses mittels einer zweispaltigen Tabelle in Form eines `org.eclipse.swt.widgets.Table`. Der Status wird bei je-

der Aktualisierung der Tabelle (z. B. nach jeden absolvierten Simulations-Schritt) mittels der Methode `getElementStatus()` bei jedem Simulations-Element abgefragt. Neben dieser Tabelle für „allgemeine“ Informationen über alle an der Simulation beteiligten Elemente existiert eine weitere, die der Visualisierung von Detail-Informationen eines Simulations-Elementes dient. Von der Struktur her entspricht diese der vorherigen Tabelle, lediglich stehen in den einzelnen Zeilen nicht die gesamten Simulations-Elemente, sondern die exakten Zustands-Informationen des von Benutzer ausgewählten Elementes. Diese werden bei jeder Aktualisierung von dem entsprechend selektierten Element mittels dessen Methode `getElementDetails()` abgefragt. Schließlich werden noch mittels einer weiteren Tabelle Informationen über den aktuellen Simulations-Lauf wie z. B. die absolvierten Simulations-Schritte oder die verstrichene, simulierte Zeit angezeigt.

Die *Aktualisierung* der zuvor genannten Informations-Komponenten erfolgt nach *jedem* absolvierten Simulations-Schritt. Dafür stellt die Klasse `UserInteractionComponent` die Methode `updateAllData()` zur Verfügung, welche alle notwendigen Schritte zur Aktualisierung der einzelnen Komponenten anstößt.

VisualizerComponent

Die Klasse `VisualizerComponent` dient der Bereitstellung des Teils der graphischen Benutzerschnittstelle, in dem die Visualisierung eines Simulations-Modells angezeigt wird. Das „Visualisieren eines Simulations-Modells“ lässt sich aufteilen in zwei Teilbereiche: zum einen müssen die Symbole der Simulations-Elemente angezeigt werden (diese sind statisch, d. h. verändern sich nicht). Zum anderen ist es auch notwendig, die Verbindungen zwischen den einzelnen Elementen sowie die Simulations-Animation anzuzeigen, was beides dynamisch, d. h. Veränderungen unterworfen ist.

Grundlegend dafür stellt diese Klasse die *Parent-Komponente*[1] dar, auf der alle Symbole von Simulations-Element angezeigt werden. Um diese Funktion zu gewährleisten, und gleichzeitig das „Zeichnen“ innerhalb dieser Komponente zu ermöglichen (z. B. für die Animation), erweitert diese Klasse die SWT-Klasse `org.eclipse.swt.widgets.Canvas`. Im Gegensatz dazu ist es für die Darstellung der Animation der Simulations-Elemente und für die Darstellung der Verbindungen zwischen diesen Elementen sehr wohl nötig, dass die VisualizerComponent noch eine gewisse Funktionalität bereitstellt. So ist die Grundvoraussetzung, um die Verbindungen zwischen den Simulations-Elementen anzeigen zu können, die *Berechnung* der graphischen Komponenten, mittels denen diese Verbindungen symbolisiert werden. Dafür existiert die Methode `calculateConnectionPoints()`, welche nach jeder Änderung der darzustellenden Modell-Struktur aufgerufen werden muss. Die Methode berechnet bei einem Aufruf für jedes Paar von in Verbindung stehenden Simulations-Elementen die Koordinaten-Paare, zwischen denen die die Verbindung symbolisierende Linie gezogen werden muss.

Das eigentliche Animieren eines Simulations-Modelles ist gelöst mittels eines `org.eclipse.swt.events.PaintListener`. Um einen Fortschritt in der Animation anzustoßen wird die von der VisualizerComponent bereitgestellte, öffentliche Methode `animateSimulation()` aufgerufen. Dieses geschieht nach jedem Simulations-Schritt. In dieser Methode wird zunächst überprüft, ob gemäß der Einstellungen der Software die Simulation zu animieren ist. Sollte diese Option aktiv sein, so wird die `redraw()`-Methode aufgerufen. Dieses bewirkt, dass der dem Canvas hinzugefügte `PaintListener` ein entsprechendes Event empfängt, und infolgedessen die Methode `drawConnectionsAndAnimateElements(...)` der VisualizerComponent aufruft. Diese Methode entfernt alle im vorherigen Schritt gezeichneten Elemente. Im nächsten Schritt werden nun die Verbindungs-Symbole zwischen den Simulations-Elementen erzeugt. Dazu wird über die zuvor mittels `calculateConnectionPoints()` (s. o.) erzeugte Liste der Koordinaten-Paare dieser Verbindungs-Symbole iteriert und überprüft, in welcher Farbe das aktuelle Verbindungs-Symbol darzustellen ist. Diese ist abhängig davon, ob an der aktuellen Verbindung gerade eine *Payload* übergeben wird. Nachdem diese ermittelt wurde, wird das Verbindungssymbol in Form einer Linie eingezeichnet, sowie Symbole für den Beginn und das Ende dieser Verbindung. In einem letzten Schritt wird nun noch die Animation der Simulations-Elemente veranlasst. Diese Funktionalität ist in die einzelnen Siulations-Elemente ausgeglie-

[1] Graphische Elemente in SWT benötigen einen sogenannten *Parent*, vgl. [Eclb]

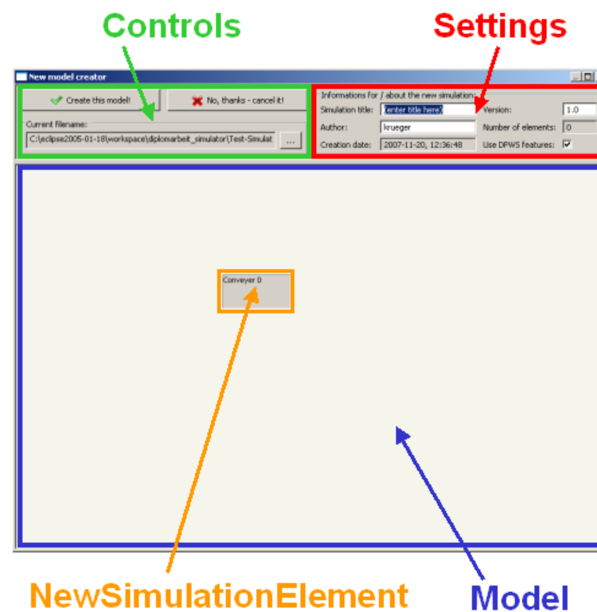


Abbildung 9.6: Struktur des NewModelCreator - Moduls

dert. Deswegen beschränkt sich die hier durchgeführte Aktion darauf, zunächst zu überprüfen, ob gemäß der gesetzten Optionen eine Animation der Elemente durchgeführt werden muss. Soll diese durchgeführt werden wird über die Liste der Simulations-Elemente iteriert, und die von jedem dieser Elemente bereitgestellte Methode `animateYourself(...)` aufgerufen. Alle weiteren für die Animation auszuführenden Aktionen erfolgen dann von jedem einzelnen Element aus.

9.2.9 NewModelCreator - Modul

Das Modul *NewModelCreator* dient dem Erstellen komplett neuer oder dem Editieren bereits bestehender Simulations-Modelle. Dieses beinhaltet insbesondere auch das Editieren der Eigenschaften bzw. Parameter-Werte der an einem Modell beteiligten Instanzen von Simulations-Elementen. Die Struktur des NewModelCreator ist auf Grundlage seiner graphischen Oberfläche in Abbildung 9.6 dargestellt. Basis des Fensters ist eine `org.eclipse.swt.widgets.Shell`. Bei der Instanziierung des NewModelCreator wird festgelegt bzw. erkannt, ob ein bestehendes Simulations-Modell bearbeitet oder ein komplett neues Modell erstellt werden soll. Dieses geschieht, indem dem Konstruktor dieser Klasse eine Instanz des Moduls `SimControl` übergeben wird. Wird hier eine Instanz übergeben, in der kein Simulations-Element angemeldet ist, so soll ein neues Simulations-Modell erstellt werden (dieses wäre der Fall, wenn der Benutzer den Menüpunkt „New simulation model“ bzw. den entsprechenden Toolbar-Schalter ausgewählt hätte). Soll hingegen ein bestehendes Modell bearbeitet werden, wird das in der SimControl dargestellte Modell mit seinen Elementen und Parametern in den NewModelCreator übernommen.

Eine zentrale Komponente, die für das Erstellen eines neuen oder Bearbeiten eines bestehenden Modells benötigt wird, ist das sogenannte `NewSimulationElement`. Dieses ist als innere Klasse implementiert, und stellt alle Funktionalitäten bereit, die im Rahmen des NewModelCreators benötigt werden, um ein neu zu erstellendes Simulations-Element verarbeiten zu können. Da hier mehr bzw. andere Funktionalität benötigt wird, als diese von den „normalen“ Simulations-Elementen des Modell-Frameworks bereitgestellt wird, reicht es an dieser Stelle nicht aus, diese Elemente zu benutzen. Insbesondere werden dabei die folgenden Funktionen von der Klasse `NewSimulationElement` zur Verfügung gestellt:

- `getVisualizer()`: liefert das hier verwendete Visualisierungs-Element des neuen Simulations-Ele-

ments. Zur Visualisierung werden hier – damit Drag'n'Drop mit diesen Elementen ermöglicht werden kann – `org.eclipse.swt.widgets.Label` verwendet.

- `getSimElement()`: mittels dieser Methode kann von außen auf die Instanz des Simulations-Elementes (und somit dessen Methoden etc.) zugegriffen werden, die das entsprechende `NewSimulationElement` repräsentiert.
- `isAlreadyConfigured()`: hiermit ist es möglich zu überprüfen, ob das repräsentierte Simulations-Element bereits (mittels des von jedem dieser Elemente zur Verfügung gestellten Konfigurations-Dialoges) konfiguriert worden ist.
- `deleteYourself()`: diese Methode entfernt sowohl das entsprechende `NewSimulationElement` aus dem NewModelCreator, als auch das dazugehörige Simulations-Element aus der entsprechenden *SimControl*-Instanz.
- `getItemContextMenu()`: hier handelt es sich um eine sehr umfangreiche Methode, in der zahlreiche Sonderfälle getrennt betrachtet und behandelt werden müssen. Grundsätzlich dient diese Methode dazu, ein Kontext-Menü für ein `NewSimulationElement` anzuzeigen. Dafür besitzt jedes `NewSimulationElement` einen `org.eclipse.swt.events.MouseListener`, der diese Methode entsprechend aufruft. Mittels dem Menü kann beispielsweise der Konfigurations-Dialog des zugrundeliegenden Simulations-Elementes aufgerufen werden. Außerdem gibt es in diesem Menü die Möglichkeit, die ausgewählte Komponente zu löschen, oder ein sogenanntes *Auxiliary Device* zu diesem Simulations-Element hinzuzufügen (vgl. Kapitel 8.6). Schließlich gibt es in diesem Menü noch die Möglichkeit, in dem bearbeiteten Modell eine Verbindung von diesem Simulations-Element zu einem Nachfolge-Element anzulegen, d. h. eine sogenannte *outgoing connection*.

Benutzt wird die zuvor beschriebene innere Klasse `NewSimulationElement` innerhalb der in Abbildung 9.6 mit „Model“ bezeichneten Komponente. Bei dieser Komponente handelt es sich um einen `org.eclipse.swt.widgets.Canvas`, der dazu dient, die Visualisierungen der neu erstellten Simulations-Elemente und der Verbindungen dieser untereinander anzuzeigen. Gleichzeitig ist dieser `Canvas` als `org.eclipse.swt.dnd.DropTarget` angelegt, was es ermöglicht, auf ihm die Instanzen der Klasse `NewSimulationElement` mittels Drag'n'Drop zu positionieren. Mittels eines Kontext-Menüs wird es dem Benutzer ermöglicht, ein neues Element zu dem Modell hinzuzufügen. Dafür sind in diesem Menü alle verfügbaren Typen von Simulations-Elementen aufgelistet. Das neue Element wird der Datenstruktur `newSimulationElements` hinzugefügt, in der alle Instanzen verwaltet werden, die für neue Simulations-Elemente angelegt wurden. Ferner wird das Visualisierungs-Element für das neu erzeugte Simulations-Element bzw. die neue Instanz der Klasse `NewSimulationElement` an der Position angezeigt, an der der Rechtsklick und die anschließende Auswahl des Element-Typs erfolgte. Mittels Drag'n'Drop ist dieses in der Komponente „Model“ (vgl. Abbildung 9.6) beliebig positionierbar. Außerdem ist nun mittels eines Rechtsklicks auf diese Visualisierung des Elements das `NewSimulationElement`-Kontext-Menü aufrufbar.

Ein weiterer Teil des NewModelCreators, der in Abbildung 9.6 zu erkennen ist, ist die „Settings“-Komponente. Die Elemente in dieser Komponente dienen zur Eingabe von Details (wie z. B. der Name des Autoren oder die Version) für das neu zu erstellende Simulations-Modell. Ferner kann in diesem Feld eingestellt werden, ob alle Elemente des Simulations-Modells per Default ihre Web Services aktiviert oder deaktiviert haben. Die letzte Komponente des NewModelCreator ist schließlich die „Controls“-Komponente. In dieser befinden sich neber der Möglichkeit zur Angabe der Datei, in dem das neue Modell gespeichert werden soll, auch die Schalter zum Übernehmen des neu erstellten Modells in die Simuloations-Umgebung bzw. alternativ zum Abbrechen des Vorganges. Eine Betätigung des Schalters zum Übernehmen des Models bewirkt zunächst eine kurze Validierung des Modells, in der z. B. die Namen der Modell-Elemente überprüft werden (diese müssen eindeutig sein). Ist das Modell gültig, wird es anschließend in die Simulations-Umgebung übernommen. Dazu wird die Liste der Instanzen von `newSimulationElement` durchlaufen, bei jedem dieser Elemente über die Methode `getSimElement()` das Modell-Element ausgelesen, und in die Instanz der `SimControl` eingefügt, die das jetzt aktuelle Modell verwaltet. Diese Instanz wird abschließend noch mittels einer Instanz der Klasse `SimulationToXmlSaver` gespeichert.

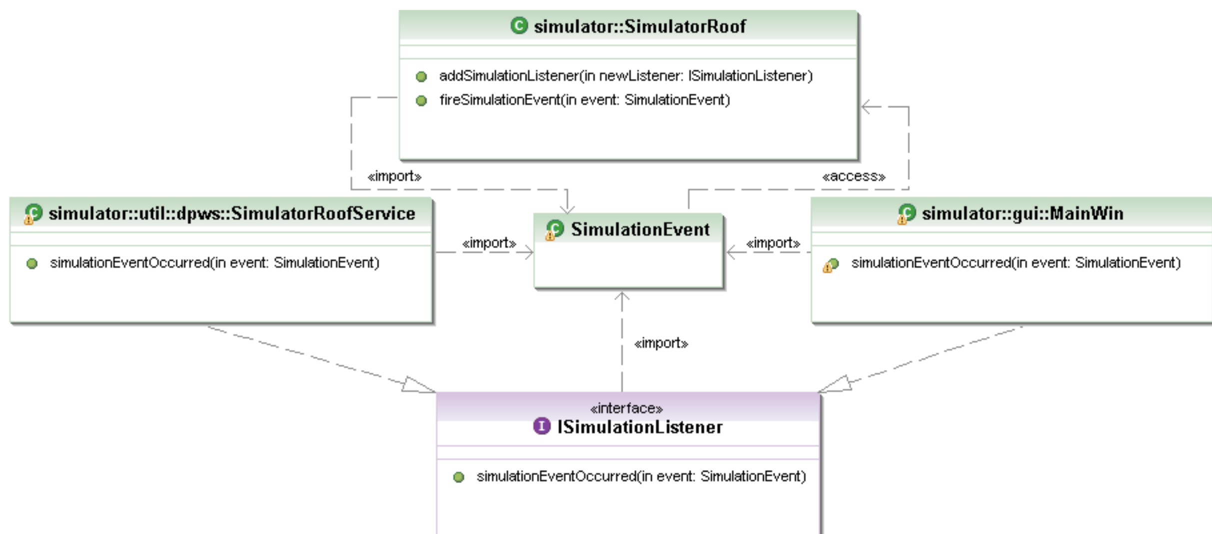


Abbildung 9.7: Das Interface ISimulationListener

9.2.10 Hilfs-Module

SimulationEvent / ISimulationListener

Mittels der Klasse `SimulationEvent` und dem Interface `ISimulationListener` werden die Grundlagen für das in der Simulations-Umgebung benutzte *Eventing* bereitgestellt. Das *Interface* `ISimulationListener` muss bei diesem Eventing von den Klassen implementiert werden, die Events empfangen möchten, die von dem Modul *SimulatorRoof* ausgelöst werden. Ferner müssen sich diese Klassen selbst über die Methode `addSimulationListener(...)`, die von dem SimulatorRoof-Modul bereitgestellt wird, zu dessen Liste von Listenern hinzufügen. Wird in dem Modul SimulatorRoof ein Event (mittels der dortigen Methode `fireSimulationEvent(...)`) ausgelöst, empfangen diese Listener dieses Event in der Methode `simulationEventOccurred(...)`.

Die Events, die dabei übergeben werden, sind Instanzen der Klasse `SimulationEvent`. In dieser sind die notwendigen Funktionen und Parameter implementiert, die für ein derartiges Event benötigt werden. Dieses sind im einzelnen:

- Der *Sender* eines Events. Dieses ist immer die Instanz des SimulatorRoof-Moduls. Da es von diesem Modul nur eine einzige Instanz innerhalb des Programmes geben darf, und diese Instanz allen anderen Komponenten bekannt ist, wäre diese Information prinzipiell nicht notwendig. Sie wurde dennoch vorgesehen, um eventuelle zukünftige Erweiterungsmöglichkeiten offen zu halten. Nach außen zur Verfügung gestellt wird der Sender über die Methode `getSource()`.
- Der *Typ* des Events. Mögliche Typen, die ein *SimulationEvent* annehmen kann, sind innerhalb der Klasse `SimulationEvent` in einer öffentlichen, statischen Enumeration mit der Bezeichnung `SimulationEventType` definiert. Beispiele für solche Typen sind `SIMULATION_STARTED` oder `TIMESTEP_OCCURRED`. Bereitgestellt wird der Typ eines SimulationEvents mittels der Methode `getEventType()`.
- Die *Nutzlast* bzw. der *Payload* des Events. Events werden u. a. auch benutzt, um Daten vom SimulatorRoof-Modul aus an Module zu schicken, die vom SimulatorRoof aus wegen der geforderten Modularität (vgl. Abschnitt 6.8.1) nicht direkt adressiert werden dürfen. Dafür ist in einem SimulationEvent ein spezieller Daten-Teil vorgesehen, der *beliebige* Daten zum Inhalt haben kann. Dieses ist die sogenannte *Nutzlast* bzw. das *Payload* dieses Events. Dieses ist, um beliebige Inhal-

te darstellen zu können, implementiert als `java.lang.Object`, und wird nach außen bereitgestellt über die Methode `getPayload()`. Ein Payload darf mit `null` belegt sein.

Gesetzt werden diese Felder alle beim Erstellen einer Instanz der Klasse `SimulationEvent.java`. Nach erfolgter Erstellung dieser Instanz können die Felder nur noch mittels oben genannter Methode abgefragt, aber nicht mehr geändert werden. In Abbildung 9.7 wird dargestellt, welche Klassen das Interface `ISimulationListener` implementieren, und somit Listener auf im SimulatorRoof-Modul ausgelöste Simulations-Events darstellen.

ErrorInformationObject

Instanzen der Klasse `ErrorInformationObject` werden benutzt, um während eines Simulations-Laufes aufgetretene Fehler an oder Informationen von einzelnen Simulations-Elementen weitergeben zu können. Dieses geschieht beispielsweise, indem eine Instanz eines *ErrorInformationObjects* als *Payload* eines zuvor beschriebenen *Simulations-Events* benutzt wird. Der Aufbau eines solchen ErrorInformationObjects ist dabei dem Aufbau eines SimuationEvents ähnlich. Jedes dieser Objekte besitzt einen *Owner*, der das Simulations-Element darstellt, an dem der von diesem Objekt repräsentierte Fehler aufgetreten ist bzw. der die Information gesendet hat. Um diesen Owner zu erhalten wird die Methode `getOwner()` bereitgestellt. Daneben muss für jedes ErrorInformationElement die sogenannte *ErrorSeverity* angegeben werden, d. h. der Grad der Schwere des Fehlers. Diese ErrorSeveritys sind festgelegt in der öffentlichen, statischen Enumeration `ErrorSeverity`, und können mit zunehmender „Schwere“ des Fehlers (in dieser Reihenfolge) die Werte `INFORMATION`, `WARNING`, `CRITICAL` oder `ERROR` annehmen. Abgefragt wird diese ErrorSeverity über die Methode `getSeverity()`. Schließlich kann auch einem ErrorInformationObject ein *Payload* mit hinzugefügt werden. Dieses ist wiederum ein `java.lang.Object`, und kann über die Methode `getPayload()` des ErrorInformationObjects abgefragt werden.

HttpReader

Der *HttpReader* dient dazu, eine Simulations-Datei zu laden, die nicht auf dem lokalen Computer, sondern auf einem Webserver abgelegt ist. Dafür stellt eine Instanz der Klasse `HttpReader` die Funktionen bereit, mittels denen eine solche Datei, die über eine *URL* spezifiziert wird, eingelesen und in eine temporäre Datei geschrieben werden kann. Diese kann dann wiederum mittels bereits existierender Module wie dem *SimLoader* geladen werden. Bei der Instanziierung der Klasse `HttpReader` muss bereits die URL als `String` übergeben werden, von der die Simulations-Datei geladen werden soll. Wahlweise kann – falls die erstellte temporäre Datei nach dem Laden nicht automatisch gelöscht werden soll – auch ein Name für eine lokale Datei mit übergeben werden, unter dem dieses Simulations-Modell lokal gespeichert werden soll. Nach erfolgter Instanziierung wird mittels eines Aufrufes der bereitgestellten Methode `buildTmpFile()` die Datei unter der spezifizierten URL gesucht, und wenn sie dort existiert mittels eines `java.io.BufferedReader` und eines `java.io.InputStreamReader` eingelesen. Daran anschließend wird sie mittels eines `BufferedWriters` lokal in eine Datei geschrieben, welche nun der den HttpReader aufrufenden Instanz (z. B. der entsprechenden `Action` des Menüs) zum Laden zur Verfügung steht. Hat diese die erstellte Datei verarbeitet (d. h. mittels eines *SimLoaders* geladen), kann diese noch die vom HttpReader bereitgestellte Methode `clean()` aufrufen, welche diese Datei wieder löscht.

SimPreferencesFileManager

Der *SimPreferencesFileManager* dient dazu, die für das Simulations-Programm vom Benutzer gesetzten *Einstellungen* beim Programmstart zu laden und wieder zu setzen, bzw. beim Beenden des Programmes auszulesen und zu speichern. Dazu stellt eine Instanz der Klasse `SimPreferencesFileManager` jeweils eine Methode zur Verfügung, die alle entsprechend notwendigen Aktionen durchführt. Als zugrundeliegendes

Daten-Format wird hier wieder *XML* benutzt, und zum Aufbauen, Laden und Speichern dieser XML-Dokumente wird die in Abschnitt 7.3.2 vorgestellte Bibliothek *JDOM* eingesetzt.

- `loadSimulatorPreferences()`: dient dem Laden der Preferences beim Programmstart. Zunächst wird mittels eines `org.jdom.input.SAXBuilder` aus der Preferences-Datei, deren Name in der Hilfsklasse `SimConstants.java` spezifiziert ist, ein `org.jdom.Document` aufgebaut. In diesem repräsentieren die einzelnen Kinder des *Wurzel-Elementes* die verschiedenen Parameter, die die Programm-Preferences darstellen. Diese XML-Elemente werden durchlaufen, und bei jedem dieser Elemente wird anhand des über `element.getName()` bereitgestellten Namens überprüft, welcher Parameter in diesem XML-Element verwaltet wird. Anschließend wird die für das Setzen speziell dieses Parameters notwendige `set...(...)`-Methode des Moduls *SimulatorRoof* mit dem *Inhalt* dieses XML-Elements, welcher über `element.getText()` verfügbar ist, aufgerufen, und dieser Parameter somit gesetzt.
- `saveSimulatorPreferences()`: mittels dieser Methode, welche beim kontrollierten Beenden des Programms aufgerufen wird, werden die Preferences gespeichert. Dazu wird zunächst ein `org.jdom.Document` erstellt. In diesem werden anschließend die verschiedenen Parameter für die Preferences als direkte Kindern des XML-Wurzelelementes gesetzt. Dafür werden die vom Modul *SimulatorRoof* bereitgestellten `get`-Methoden für die verschiedenen Parameter durchlaufen. Für jeden dieser Parameter wird ein entsprechendes `org.jdom.Element` angelegt, wobei der *Name* dieser Elemente den *Bezeichnern* der Parameter der Preferences entspricht. Als Inhalt dieser XML-Elemente wird die jeweilige Belegung des Parameters gesetzt, und die dann fertig erstellten Elemente werden mittels der Methode `getChildren().add(...)` dem XML-Wurzelelement als Kinder hinzugefügt. Nachdem dieses für alle Parameter der Preferences durchgeführt wurde, ist dieses XML-Dokument eine korrekte XML-Darstellung der Preferences, und wird mittels eines `org.jdom.output.XMLOutputter` in die entsprechende Datei geschrieben.

SimUtilities

In der Klasse `SimUtilities.java` befinden sich verschiedene kleine statische Hilfsmethoden, die mit gleicher Funktion an mehreren Stellen im übrigen Programmcode benötigt werden. Hier sollen nur einige wenige Beispiele für diese Funktionen gegeben werden. Für den kompletten Umfang der bereitgestellten Funktionen sei auf den Quellcode verwiesen.

- `getTimestamp()`: liefert einen String zuürck, der das aktuelle Datum und die aktuelle Uhrzeit darstellt. Benötigt wird diese Funktion z. B. bei der Protokollierung von Simulations-Läufen.
- `calculateBestConnectionsPoints(...)`: mittels dieser Methode lassen sich die Anfangs- und Endpunkte von Verbindungs-Visualisierungen zweier Simulations-Elemente berechnen. Dieses ist relativ aufwändig, und wird z. B. sowohl im Model *NewModelCreator* als auch in der normalen Simulations-Visualisierung im *MainWin*-Modul benötigt.
- `drawOutconnSymbol(...)`: dient dem „Malen“ eines Symbols für eine ausgehende Verbindung eines Simulations-Elementes. Die Eingabeparamter sind der *Grafik Kontext*, in dem das entsprechende Symbol zu erstellen ist, sowie die Koordinaten und die Farbe des Symbols. Benutzt wird diese Methode – ähnlich wie die zuvor angesprochene Methode `calculateBestConnectionsPoints(...)` – sowohl im *MainWin*- als auch im *NewModelCreator*-Modul.

SimConstants

In der Klasse `SimConstants.java` befinden sich Konstanten, die in der erstellten Software benutzt werden. Insbesondere wird oft die *gleiche* Konstante an verschiedenen Stellen innerhalb des Programmcode

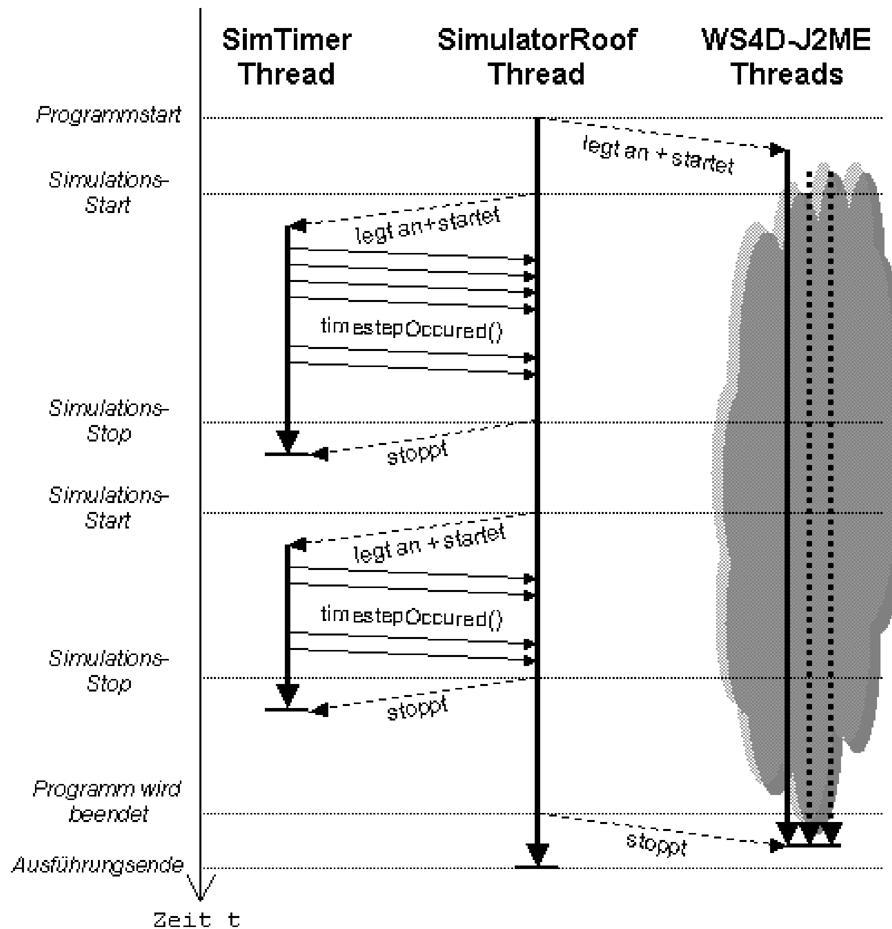


Abbildung 9.8: Threads der Simulations-Umgebung

benutzt. Dieses legt es nahe, diese statischen Konstanten in einer gemeinsamen Klasse zu verwalten. Einerseits vereinfacht dieses die Implementierung, da mehrfach benötigte Werte nicht mehrfach angelegt werden müssen, vor allem aber sorgt dieses für Sicherheit bei der *Änderung* von Konstanten, da diese Änderung nur an eben einer zentralen Stelle durchgeführt werden muss.

9.3 Threads

An dieser Stelle soll noch einmal kurz auf die *Threads* eingegangen werden, in denen das Programm ausgeführt wird. Vereinfacht dargestellt sind diese Threads und die Aufruf- bzw. Erzeugungs-Struktur, die diesen zugrunde liegt, in Abbildung 9.8. Wie dort ersichtlich ist gibt es einen *Haupt-Thread*, der während der gesamten Programmausführung aktiv ist. Von diesem Thread ausgehend werden alle möglicherweise benötigten weiteren Threads erzeugt, gestartet, und schließlich wieder beendet. Ferner finden neben dem Starten und Beenden weitere Interaktionen zwischen den Threads statt. Diese sind in Abbildung 9.8 nur teilweise symbolisiert.

Die Threads, die aus dem Haupt-Thread heraus gestartet werden, lassen sich dabei in zwei Typen klassifizieren. Einerseits sind dieses die Threads, die für die Bereitstellung und Funktion der diversen *Web Services* der verschiedenen Programm-Teile (wie der Simulations-Umgebung selbst und der Komponenten der Simulations-Modelle) benötigt werden. Andererseits ist dieses der Thread, mittels dem der *Timer* der

Simulation nebenläufig betrieben werden kann (vgl. Abschnitt 9.2.4).

Wie in Abbildung 9.8 dargestellt werden die Threads für die Bereitstellung der Web Services unmittelbar nach den Start des Hauptprogrammes automatisch erzeugt und gestartet. Dieses geschieht ohne explizite, durch den Benutzer auszuführende Aktionen. Der Grund dafür liegt darin, dass sofort beim Start des Programmes das *Device* und der von diesem Device bereitgestellte *Service* für die *externe Kontrolle* des Simulations-Umfeldes erstellt und gestartet werden. Die Erzeugung der für für diese Komponenten benötigten Threads wird von der für diese Web Services benutzten Bibliothek, d. h. dem *WS4D-J2ME*-Stack bereitgestellt, ebenso, wie die Verwaltung dieser Threads. Deswegen soll hier nicht näher auf diese dort benutzten Threads eingegangen werden, statt dessen sei verwiesen auf die Dokumentation dieses verwendeten Protokoll-Stacks (vgl. [WS4b]). Es sei lediglich darauf hingewiesen, dass für diese Web Services *mehrere* nebenläufige Threads existieren. Ferner sei noch erwähnt, dass diese Threads beim kontrollierten Beenden des Haupt-Programmes in einem der letzten ausgeführten Schritte (vgl. die Beschreibung der Methode „programWillTerminate()“ in Abschnitt 9.2.1) explizit beendet werden, wie dieses auch in Abbildung 9.8 dargestellt ist.

Neben diesen Threads existiert – wie oben beschrieben – neben dem Haupt-Thread gegebenenfalls noch ein weiterer Thread, in dem der *Timer* der Simulation nebenläufig betrieben wird. Wie in Abbildung 9.8 dargestellt ist dieser Thread (im Gegensatz zu den Threads der Web Services) nicht während der gesamten Laufzeit des Haupt-Threads parallel zu diesem aktiv. Erzeugt und gestartet wird dieser Thread (und somit der Timer), wenn der Timer benötigt wird. Dieses ist in dem Moment der Fall, in dem ein Simulations-Lauf gestartet wird. Während der Timer läuft interagiert er Thread-übergreifend – wie in obiger Abbildung symbolisiert – mit der im Haupt-Thread arbeitenden Instanz des Haupt-Programmes. Wird der Simulations-Lauf gestoppt, so wird auch der Thread des Timers gestoppt und die Instanz somit verworfen.

9.4 Web Service für die Simulations-Umgebung

Abschließend soll hier nun noch der von der Simulations-Umgebung bereitgestellte *Web Service* näher betrachtet werden. Die „technische“ Seite der konkreten Implementierung des bereitgestellten Devices und des dazugehörigen Services wurde dabei bereits in Abschnitt 9.2.5 erläutert. Hier sollen nun noch konkret die Funktionen erläutert werden, die der bereitgestellte Web Service zur externen Steuerung und Kontrolle der Simulations-Umgebung anbietet. Diese bereitgestellten Funktionen lassen sich unterteilen in zwei Arten: zum einen Funktionen, die explizit aufgerufen werden müssen, und zum anderen um Funktionen, die *Events* bereitstellen, die ein Client empfangen kann. Letztere befinden sich am Ende der folgenden Auflistung.

Im einzelnen werden von diesem Web Service die folgenden Funktionen bzw. mittels dieser Funktionen der Zugriff auf die entsprechenden Funktionen der Simulations-Umgebung bereitgestellt:

- `loadSimulation`: mittels dieser Funktion kann eine Simulations-Datei, deren Dateiname einen `InputParameter` der `Action` darstellt, in die Simulations-Umgebung geladen werden. Dabei überprüft die Funktion selbst zunächst, ob die angegebene Datei existiert, und wenn dem so ist wird das Laden der Datei in die Simulations-Umgebung durch Auslösen eines entsprechenden *Events* eingeleitet.

- `loadFromHttpLocation`: hier wird eine ähnliche Funktion wie bei der zuvor beschriebenen realisiert, nur dass hier nicht eine lokale Simulations-Datei geladen wird, eine Datei von einem entfernten Web-Server. Dafür besitzt diese Funktion einen `InputParameter`, mittels dem die *URL* dieser Datei spezifiziert werden muss. Diese URL wird in einem ersten Schritt überprüft, und sollte sie korrekt sein (d. h. sollte die angegebene Datei unter dieser URL existieren), wird eine Anweisung zum Laden dieser URL an die Simulations-Umgebung weitergegeben. Dieses geschieht wiederum durch auslösen eines dafür vorgesehenen *Events*. Bei dieser Funktion exisitiert ferner noch ein `OutputParamter`,

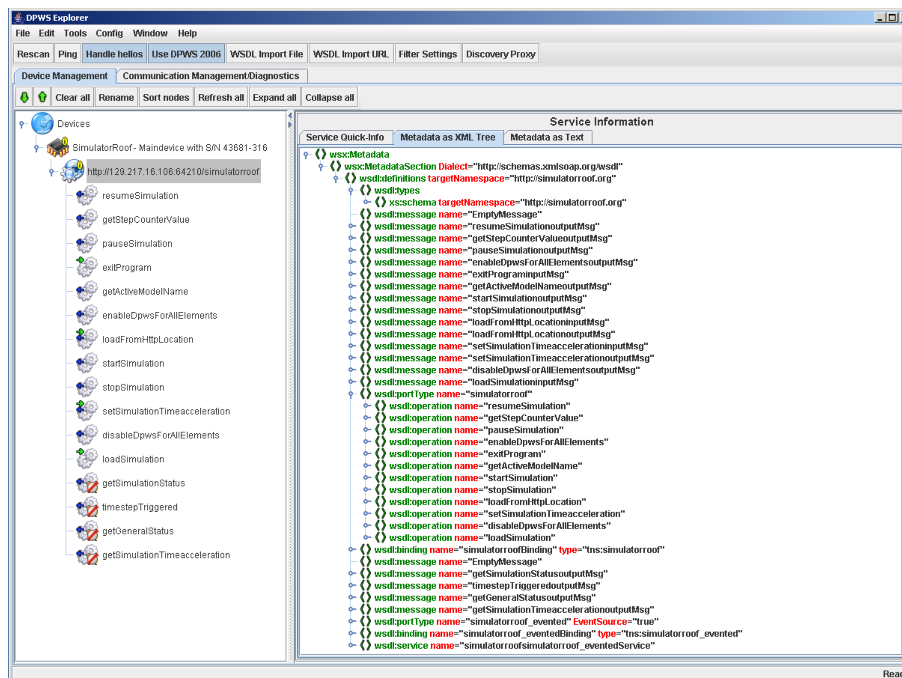


Abbildung 9.9: Darstellung des Web Services der Simulations-Umgebung im DPWS-Explorer von [WS4a]

der dem *Client* zurückgegeben wird, von dem dieser Aufruf erfolgte. Mittels dieser Rückgabe wird der Client über das erfolgreiche oder fehlgeschlagene Laden der URL informiert.

- `setSimulationTimeacceleration`: diese Funktion dient dem Setzen des Zeitbeschleunigungs-Faktors der Simulation mittels eines externen Eingriffs. Dafür muss als `InputParameter` der neu zu setzende Wert für diesen Faktor übergeben werden. Dieser wird zunächst auf Gültigkeit überprüft (d. h., ob er in den angegebenen Grenzen für erlaubte Faktoren liegt), und wird im Falle der Gültigkeit in der Simulations-Umgebung gesetzt. Auch hier exisitiert ein `OutputParamter`, der dem aufrufenden Client zurückgegeben wird. Dieser gibt entweder eine Bestätigung zurück, dass der Faktor erfolgreich geändert wurde, oder aber eine Fehlermeldung, warum eine Änderung nicht möglich war.

- `startSimulation`: ein Aufruf dieser Funktion, die keinen Eingabeparameter besitzt, bewirkt das Starten eines Simulations-Laufes einer geladenen Simulation. Ferner wird der *ExecutionControlMode* der Simulations-Umgebung bei diesem Aufruf auf `EXECUTION_CONTROL_MODE_REMOTE_CONTROLLED` gesetzt (vgl. Abschnitt 9.2.1). Diese Funktion besitzt einen `OutputParameter`, mittels dem der Aufrufer dieser Funktion über einen erfolgreichen Start des Simulations-Laufes informiert wird, oder aber eine Fehlermeldung bekommt, wenn die Simulation nicht gestartet werden konnte.

- `pauseSimulation` / `resumeSimulation`: hiermit kann eine laufende Simulation pausiert werden, bzw. eine pausierte Simulation wieder fortgesetzt werden. Eingabeparameter für diese Funktionen existieren keine, mittels eines `OutputParameter` wird der aufrufende Client wiederum über Erfolg oder Misserfolg seines Aufrufes informiert.

- `stopSimulation`: dient dem Beenden eines Simulations-Laufes. Wie bei den zuvor beschriebenen Funktionen zum Pausieren oder wieder Fortsetzen eines Simulations-Laufes existieren auch hier keine Eingabeparameter, aber ein `OutputParamter`, der über Erfolg oder Misserfolg des Aufrufes informiert.

- `enableDpwsForAllElements` / `disableDpwsForAllElements`: diese Funktionen diesen dem Aktivieren bzw. Deaktivieren der *Web Services* der einzelnen Simulations-Elemente eines geladenen Simulations-Modells. Diese Services sind dabei nicht zu verwechseln mit dem von der Simulations-Umgebung bereitgestellten Service, welcher *hier* gerade erläutert wird. Eingabeparameter für diese beiden Funktionen sind nicht vorgesehen, `OutputParamter` beider Funktionen, die dem aufrufenden Client zurückgeliefert werden, geben Informationen darüber, ob das (De-) Aktivieren der Services durchgeführt werden konnte.

- `getStepCounterValue`: mittels dieser Funktion kann ein Client die im Moment des Funktionsaufrufes aktuelle Anzahl von absolvierten Simulations-Schritten des aktuellen Simulations-Laufes abfragen. Ein Eingabeparameter existiert hier ebenfalls nicht, bei einem Aufruf der Funktion wird als einzige durchgeführte Aktion lediglich der aktuelle Wert des Simulations-Schritt-Zählers der Simulations-Umgebung ausgelesen, und der `OutputParamter` dieser Funktion entsprechend gesetzt.

- `getActiveModelName`: entspricht von Struktur und Vorgehen her der zuvor beschriebenen Funktion `getStepCounterValue`, mit dem Unterschied, dass mittels dieser Funktion der Name des aktiven Simulations-Modells abgefragt werden kann.

- `exitProgram`: dient dem kontrollierten Beenden des Simulations-Programmes. Um ein versehentliches Beenden zu vermeiden besitzt diese Funktion einen `InputParamter`. Dieser muss von dem aufrufenden Client mit „*y*" oder „*yes*" belegt werden, damit die Funktion das Simulations-Programm beendet. Das eigentliche Beenden erfolgt dann mittels eines Aufrufes der von der Simulations-Umgebung bereitgestellten Methode `exitProgram()`, welche alle notwendigen Schritte zum kontrollierten Beenden des Programms (inklusive Beenden der hier bereitgestellten Web Services) ausführt.

- `getGeneralStatus` / `getSimulationStatus` / `getSimulationTimeacceleration`: diese Methoden ermöglichen es Clients, die zuvor eine entsprechende *Subscription* getätigt haben (vgl. Abschnitt 3.3), mittels Events über Veränderungen bzgl. des allgemeinen Programm-Zustands, des Zustands der aktiven Simulation, oder der eingestellten Zeitbeschleunigung informiert zu werden. Jede dieser Methoden ist auch direkt, d. h. ohne Subscription, aufrufbar. In diesem Fall liefert sie als Rückgabe einmal den aktuellen Wert des entsprechenden Parameters.

- `timestepTriggered`: diese Funktion dient Clients, die eine entsprechende *Subscription* vornehmen, um konstant über das Fortschreiten der Simulation informiert zu werden. Dafür besitzt diese Funktion einen `OutputParamter`, der mit `true` belegt wird, wenn ein Simulations-Schritt eingeleitet wird, und wieder auf `false` gesetzt wird, wenn der Simulations-Schritt abgeschlossen ist. Jede Veränderung dieses Parameters bewirkt dabei das Auslösen eines *Events*. Somit erkennen als *Event Sinks* registrierte Clients exakt, wann ein neuer Simulations-Schritt initiiert wird, und wann dieser Schritt abgeschlossen ist. Ferner besitzt diese Funktion einen weiteren `OutputParamter`, der die laufende Nummer des zuletzt initiierten bzw. abgeschlossenen Simulations-Schrittes enthält. Somit kann ein Client verifizieren, dass er die Events in der richtigen Reihenfolge erhalten hat.

Abbildung 9.9 zeigt in der linken der in diesem Fenster erkennbaren Spalten die Liste aller von diesem Service bereitgestellten Funktionen, wie sie der *DPWS-Explorer* der *Web Services for Devices* (WS4D) - Initiative (vgl. [WS4a]) findet und darstellt.

Kapitel 10

Anwendungsbeispiel

In diesem Kapitel wird als Beispiel für die Anwendung des erstellten Simulations-Systems ein mit diesem System entwickeltes Simulations-Modell vorgestellt und simuliert. Bei einigen der durchgeführten Simulations-Läufe wird ein externes Kontroll-Programm zur Gewährleistung von fehlertolerantem Verhalten benutzt werden, bei anderen hingegen, um den Unterschied zu demonstrieren, nicht. Wird es verwendet, so hat das externe Kontrollprogramm die Aufgabe, im Fehlerfall einen Teil der Modell-Komponenten von außen über die von diesen Komponenten bereitgestellten Web Services steuern bzw. beeinflussen, und mittels dieser Eingriffe das fehlertolerante Verhalten des Gesamt-Systems zu gewährleisten.

Zunächst wird nun das benutzte Modell einschließlich der Konfigurations-Details der in diesem Modell benutzten Komponenten vorgestellt werden. Daran anschließend wird gezeigt, wie sich Fehlertoleranzverfahren in dieses Modell einbringen lassen. Dieses führt zur Vorstellung des benutzten externen Kontroll-Programmes, dass diese Verfahren implementiert. Nachdem diese Grundlagen dargestellt worden sind werden abschließend die Ergebnisse verschiedener Simulations-Läufe dieses Modells mit verschieden starken Ausprägungen der benutzten Fehlertoleranz-Verfahren vorgestellt und ausgewertet.

10.1 Benutztes Modell

Eine aus der Simulations-Software heraus gespeicherte Darstellung des benutzten Simulationsmodells findet sich in Abbildung 10.1. Wie aus dieser Abbildung erkennbar ist, handelt es sich bei dem hier benutzten Modell um die Nachbildung einer *Produktions-Straße*, bei der *Payloads* (hier die *Werkstücke*) in einer Anfangskomponente diese Produktions-Straße betreten, über bzw. durch verschiedene Transport-Komponenten und Bearbeitungsstationen laufen, um schließlich die Produktions-Straße an einer Endkomponente wieder verlassen. Hier folgt nun zunächst eine Auflistung der Komponenten mit jeweils einer kurzen Beschreibung und der Konfigurations-Einstellungen. Die Bezeichnungen der Komponenten entsprechen dabei den in Abbildung 10.1 erkennbaren Bezeichnungen. In dieser Abbildung befinden sich ferner einige weitere Ergänzungen, die im Folgenden noch erläutert werden.

- *InPort*: dient als einzige Komponente dem Hereingeben von Payloads auf die Produktions-Straße. Ein Payload wird alle 10 Zeiteinheiten hereingegeben; die Hereingabe des 1. Payloads erfolgt im 1. Simulations-Schritt.
- *Conveyer 0*: transportiert die vom InPort hereingegebenen Payloads weiter. Die simulierte Länge dieses Conveyers beträgt 100 Längeneinheiten, ein Payload wird binnen 15 Zeiteinheiten über diese Komponente befördert, und die Förder-Kapazität beträgt 20 Payloads. Ferner läuft dieser Conveyer im *AutoHalted*-Modus, d. h. wenn ein Payload das Ende dieser Komponente erreicht stoppt dieser Conveyer so lange, bis dieses Payload von der Nachfolge-Komponente übernommen wurde.

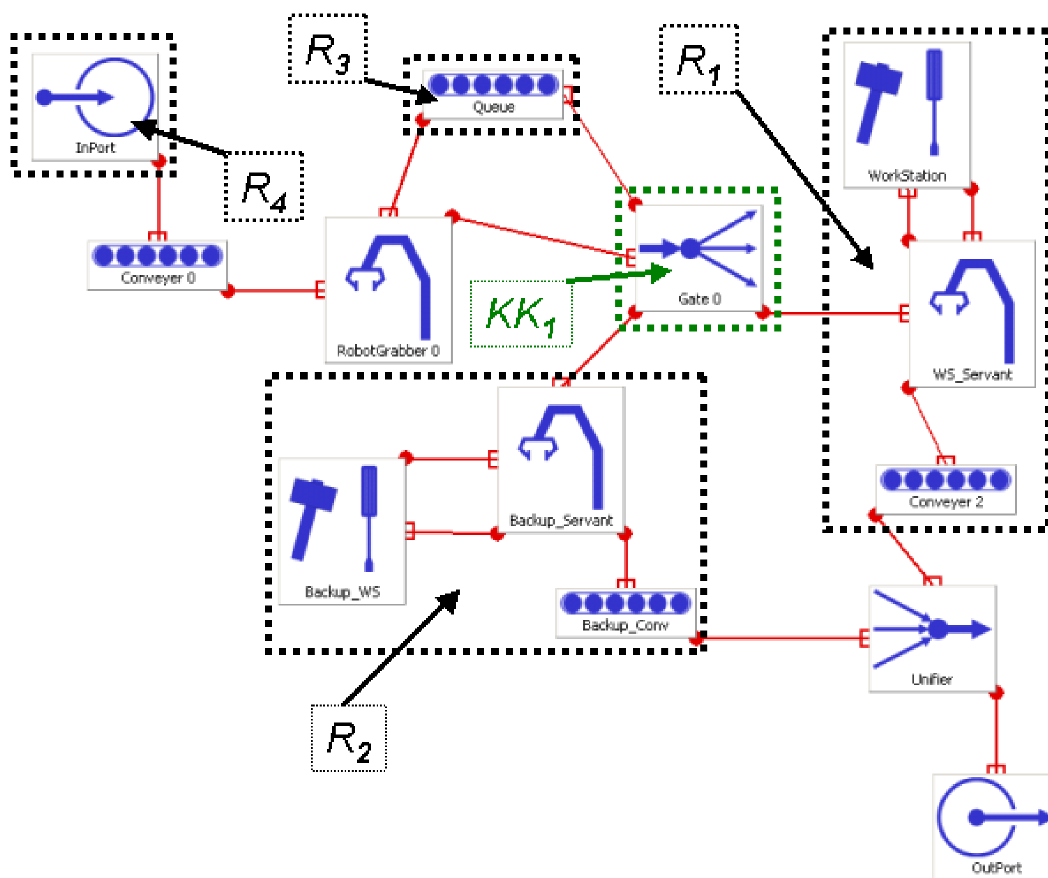


Abbildung 10.1: Wichtige Teile des Simulations-Modells

- *RobotGrabber 0*: holt Payloads von den angeschlossenen Komponenten Conveyer 0 und Queue ab, und liefert diese zu der Komponente Gate 0. Die Lieferzeit beträgt dabei 4 Zeiteinheiten, die Positionierungs-Zeit beträgt 2 Zeiteinheiten.

- *Gate 0*: dient dem Verteilen von ankommenden Payloads auf die Nachfolge-Komponenten Queue, WS_Servant oder Backup_Servant. Dabei liefert dieses Gate per Voreinstellung an den WS_Servant, bis es eine Anweisung zum Liefern an eine der anderen Komponenten erhält. Die Kapazität dieses Gate liegt bei 1; die benötigte Anzahl von Zeiteinheiten zum Weiterreichen eines Payloads liegt bei 5.

- *Queue*: transportiert von Gate 0 ankommende Payloads zurück zu RobotGrabber 0. Die simulierte Länge dieses Conveyers beträgt 100 Längeneinheiten, ein Payload benötigt 100 Zeiteinheiten, um über diesen Conveyer befördert zu werden. Ferner besitzt dieser eine Kapazität von 25 Payloads. Außerdem gehören zu diesem Conveyer zwei in Abbildung 10.1 nicht erkennbare Instanzen der *LightBarrier* (vgl. Abschnitt 8.6) mit den Bezeichnungen *QueueEntryLB* und *QueueExitLB*.

- *QueueEntryLB / QueueExitLB*: wie zuvor beschrieben handelt es sich hierbei um zwei Lichtschranken-Instanzen, die sich „auf“ der Queue befinden. Die QueueEntryLB befindet sich dabei an der Längen-Position 1; die QueueExitLB an der Position 99.

- *WS_Servant*: befördert von Gate 0 ankommende Payloads zur Bearbeitung zu dem Element WorkStation, und nach dort erfolgter Bearbeitung weiter zu Conveyer 2. Die benötigte Positionierungs-Zeit beträgt 2, die benötigte Liefer-Zeit beträgt 3 Zeiteinheiten.

- *WorkStation*: bearbeitet die von WS_Servant angelieferten Payloads. Die Bearbeitungs-Zeit pro Payload beträgt 25 Zeiteinheiten, dabei gibt es in jedem Bearbeitungs-Schritt eine Wahrscheinlichkeit von 3%, dass an der WorkStation ein Defekt auftritt. Tritt ein solcher Defekt auf dauert es 25 Zeiteinheiten, bis dieser wieder behoben ist.
- *Conveyer 2*: transportiert von WS_Servant ankommende Payloads weiter. Die Kapazität dieses Conveyers beträgt 1 Payload, die simulierte Länge beträgt 5 Längeneinheiten, und die Transportzeit für ein Payload 5 Zeiteinheiten. Ferner läuft der Conveyer durchgängig, d. h. ohne zu stoppen.
- *Backup_Servant*: befördert von Gate 0 gelieferte Payloads zur Backup_WS, bzw. von dieser zurückgelieferte Payloads weiter zum Backup_Conv. Die Positionierungs-Zeit beträgt 2, die Liefer-Zeit beträgt 3 Zeiteinheiten.
- *Backup_WS*: bearbeitet die von der Komponente Backup_Servant angelieferten Payloads. Die Bearbeitungs-Zeit beträgt dabei 10 Zeiteinheiten, die benötigte Reparatur-Zeit im Falle eines Defektes (welcher mit einer Wahrscheinlichkeit von 2% pro Arbeits-Schritt eintritt) beträgt 25 Zeiteinheiten.
- *Backup_Conv*: transportiert vom Backup_Servant angelieferte Payloads weiter zum Unifier. Die simulierte Länge beträgt 10 Längeneinheiten, die Payload-Kapazität 10 Payloads, und die benötigte Transportzeit für ein Payload liegt bei 10 Zeiteinheiten. Ferner läuft dieser Conveyer dauerhaft.
- *Unifier*: hierbei handelt es sich um ein Gate im Unifier-Mode. Dieses bündelt die von Conveyer 2 und Backup_Conv angelieferten Payloads wieder auf einen Ausgang. Dabei benötigt diese Komponente für die Verarbeitung eines Payloads 4 Zeiteinheiten, und es können gleichzeitig maximal 2 Payloads verarbeitet werden.
- *OutPort*: diese Komponente beschließt die Produktions-Straße, d. h. über diese Komponente verlassen die Payloads das Simulations-System. Um ein Payload aus dem System zu entlassen werden hier 50 Zeiteinheiten benötigt; gleichzeitig kann diese Komponente 10 Payloads verarbeiten.

10.1.1 Modell-Teile und Redundanzen

Hier soll nun das zuvor beschriebene Gesamt-Modell unterteilt werden. Bei dieser Unterteilung wird direkt darauf eingegangen, welche (Teil-) Aufgabe bzw. Funktion der jeweilige Teil bei dem später folgenden, implementierten Fehlertoleranzverfahren darstellen wird. Die im Folgenden wichtigen Komponenten bzw. Komponten-Gruppen sind in Abbildung 10.1 markiert und mit R_1 bis R_4 bzw. mit KK_1 bezeichnet. Diese Bezeichnungen werden hier im weiteren Verlauf für diese Komponenten oder Komponenten-Gruppen benutzt.

Die zentrale Komponente, die von einem externen Programm gesteuert werden muss, um fehlertolerantes Verhalten des Gesamt-Systems zu gewährleisten, ist die Kontroll-Komponente KK_1 bzw. das *Gate*, welches die Payloads entsprechend auf die drei hier anschließend erläuterten Nachfolge-Komponenten verteilt. Auf die genaue Funktion bzw. Arbeitsweise von KK_1 soll hier noch nicht näher eingegangen werden, da dieses in Abschnitt 10.2 folgt. Es sei lediglich bereits erwähnt, dass diese Komponente primär versucht, Payloads an die Komponente R_1 zu liefern. Ist dieses aus welchen Gründen auch immer nicht möglich, tritt ein Fehler auf, und ein externes Kontroll-Programm sollte eingreifen, und Komponente KK_1 entsprechend *rekonfigurieren*. Findet ein solcher Eingriff nicht statt, wird der Fehler nicht aufgelöst und würde sich potentiell auf das Gesamt-System auswirken.

Die Komponente R_1 stellt in diesem Modell den primären Weg dar, den die transportierten Payloads zurücklegen sollten. Gewissermaßen handelt es sich dort um die primäre Fertigungs-Straße, die benutzt wird, solange sie korrekt arbeitet. Wie oben beschrieben existiert allerdings eine Defekt-Wahrscheinlichkeit von 3%, dass an der Komponente *WorkStation* innerhalb von R_1 beim Bearbeiten eines Payloads ein Defekt auftritt, d. h. dementsprechend die gesamte Komponente R_1 nicht mehr arbeiten kann. Dafür ist

hier Komponenten R_2 vorgesehen, die die gleiche Funktionalität aufweist, wie die zuvor beschriebene Komponente. Dementsprechend muss durch ein entsprechendes Kontroll-Programm zum Gewährleisten des fehlertoleranten Verhaltens auf R_2 umgeschaltet werden, wenn R_1 fehlerhaft arbeitet (vgl. Abschnitt 10.2). Das gleiche muss geschehen, wenn an Komponente R_1 ein Payload zu liefern wäre, diese aber dieses Payload nicht akzeptiert, da sie das vorherige Payload noch nicht abgearbeitet hat. Hier lässt sich bereits feststellen, dass es sich bei Komponente R_2 also um eine *dynamisch-strukturelle Redundanz* in Sinne von Kapitel 2.2 handelt.

Eine weitere der markierten Komponenten ist die *Queue*, welche im Folgenden mit R_3 bezeichnet wird. Diese Komponente ist für den Fall vorgesehen, dass sowohl die Komponenten R_1 wie auch R_2 zu einem Zeitpunkt t_x nicht in der Lage sind, ihre Funktion auszuüben bzw. ein Payload anzunehmen (entweder, weil entsprechende Defekte vorliegen, oder weil z. B. ein weiteres System die jeweilige Komponente überlasten würde, und deswegen nicht angenommen wird). Dann ist es möglich, Payloads auf die Komponente R_3 „umzuleiten", auf welcher diese gewissermaßen in einer „Warteschleife" kreisen, um später erneut zur Kontroll-Komponente KK_1 zu gelangen, welche sie dann möglicherweise zu einer der Fertigungs-Straßen R_1 oder R_2 weiterleitet. Mittels Benutzung der Komponente R_3 wird also für die Bearbeitung eines konkreten Payloads eine *dynamische Zeitredundanz* (vgl. Kapitel 2.2) realisiert.

Die letzte der explizit herauszuhebenden Komponenten ist der *InPort*, also Komponente R_4. Dieser dient wie beschrieben dazu, neue Payloads in die Simulation hineinzugeben. Ferner ist diese Komponente so angelegt, dass sie in ein Fehlertolerenzverfahren mit einbezogen werden kann. Dieses geschieht, indem die *Rate*, mit der dort Payloads in die Simulation hineingegeben werden, abhängig von der Anzahl der konkret vorliegenden Fehler abgesenkt bzw. bei abnehmender Anzahl von Fehlern wieder erhöht wird. Dieses realisiert eine weitere *dynamische Zeitredundanz*, und wird ebenfalls näher erläutert in Abschnitt 10.2.

10.2 Externes Kontrollprogramm

Das externe Kontrollprogramm für die Simulation des zuvor vorgestellten Modells hat mehrere Funktionen bzw. Aufgaben. Es dient einerseits zur Kontrolle der Simulation, d. h. um Starten und gegebenenfalls Stoppen von Simulations-Läufen, und während einer laufenden Simulation zur Darstellung der mit dem Simulationsprogramm ausgetauschten Daten. Auf der anderen Seite wird von diesem Kontrollprogramm die fehlertolerante Steuerung des Simulations-Modells gewährleistet. Die Betrachtung bzw. Erläuterung dieses Kontrollprogrammes erfolgt hier deswegen zweigeteilt. Zunächst wird kurz auf die Implementierung des Programmes eingegangen, bevor daran anschließend explizit auf die Programmteile eingegangen wird, die für die Gewährleistung der Fehlertoleranz eingesetzt werden. Bei der Erklärung dieser Programmteile wird dann herausgestellt, wie die Verfahren zur Gewährleistung des fehlertolerantem Verhaltens umgesetzt wurden. Hier sei bereits festgehalten, dass die gesamte Kommunikation zwischen dem Kontrollprogramm, der Simulations-Umgebung und jeder einzelnen Komponente des Simulations-Modells über die von der Umgebung bzw. den Modell-Komponenten bereitgestellten Web Services realisiert wird. Dieses ist vereinfacht (z. B. ohne explizite Abbildung der *Devices*) dargestellt in Abbildung 10.2.

10.2.1 Grundlagen des Kontrollprogrammes

Die grafische Benutzerschnittstelle des Kontrollprogrammes ist dargestellt in Abbildung 10.3. In dieser Abbildung sind die zentralen Komponenten dieses Programmes erkennbar: zum einen sind dieses die Komponenten zur Interaktion des Benutzers mit dem Programm in der oberen Zeile, und zum zweiten eine Komponente zum Anzeigen von Programm-Informationen und mit dem Simulations-Programm ausgetauschten Daten. Bereitgestellt wird diese GUI von der Klasse `TestScenarioMain`, worauf hier allerdings nicht näher eingegangen werden soll. Abbildung 10.3 zeigt das Kontrollprogramm nach bereits erfolgter Initialisierung, welche durch Betätigung des Schalter *Prepare* erfolgt. Erst nach erfolgter Initialisierung sind die in dieser Abbildung bereits als aktiv angezeigten Komponenten zur Steuerung der Simulation

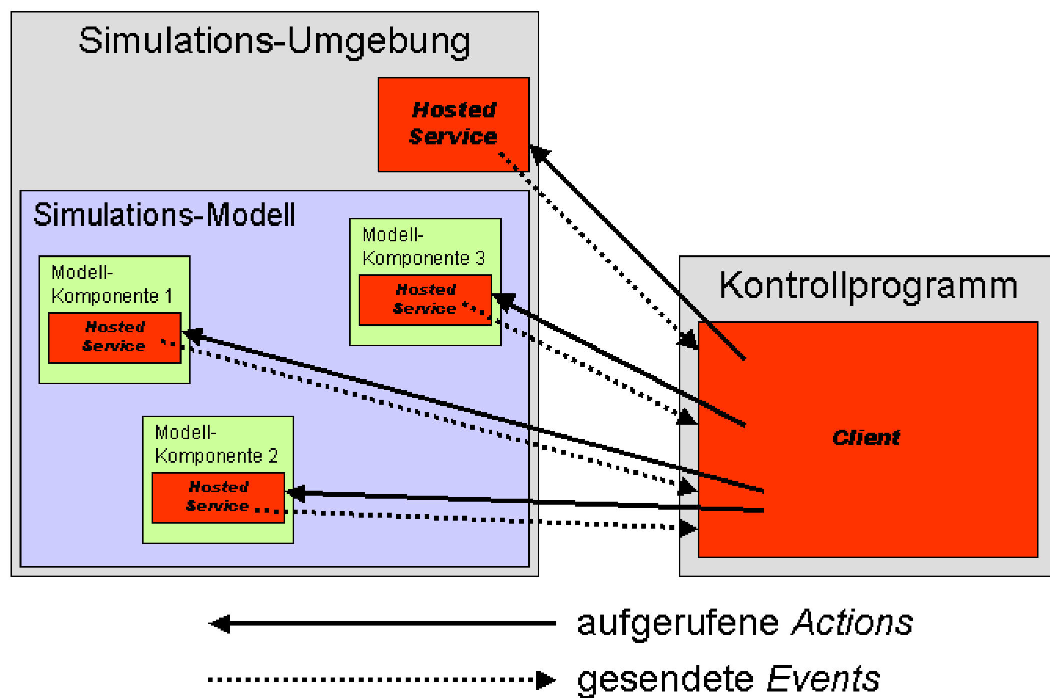


Abbildung 10.2: Kommunikationsstruktur des Anwendungsbeispiels

(wie *Start simulation* oder *Stop simulation*) aktiv; zuvor sind sie deaktiviert.

Im Rahmen dieser *Initialisierung* des Kontrollprogrammes wird eine Instanz der Klasse `TestScenarioDpwsClient` erzeugt und gestartet. Diese Klasse realisiert die Kommunikation des Kontrollprogrammes mit der Simulations-Umgebung und den Komponenten des Simulations-Modells. Dazu erweitert diese Klasse `org.ws4d.java.client.Client`, welche Funktionen wie z. B. das Suchen nach Services bereitstellt. Diese Funktion wird im Rahmen der Initialisierung benutzt, um nach allen im lokalen Netzwerk verfügbaren Services zu suchen (vgl. Abschnitt 3.3.2). Wird dabei ein Service gefunden, so wird mittels der Methode `addService(...)` überprüft, ob es sich bei diesem Service um einen der hier benötigten Services der Simulations-Elemente oder der Simulations-Umgebung handelt. Dafür wird eine Liste verwaltet, in der alle benötigten Services verzeichnet sind. Handelt es sich um einen dieser benötigten Services, so wird dieser hier zu den verwalteten Services hinzugefügt, und der Benutzer mittels einer entsprechenden Ausgabe in der grafischen Benutzerschnittstelle darüber informiert (dieses ist ebenfalls in Abbildung 10.3 dargestellt). Wurde ein Service eines Simulations-Elementes gefunden, wird außerdem eine *Subscription* auf die von jedem dieser Services bereitgestellte *Action* `ErrorsWarningsInformations` vorgenommen, um zukünftig über von diesem Simulations-Element ausgelöste Informations-, Warnungs-, oder Fehler-Meldungen informiert zu werden (vgl. Abschnitt 3.3.1). Außerdem wird nach jedem Finden eines benötigten Service überprüft, ob mit diesem Service alle für die Kontrolle bzw. Steuerung des Modells und der Simulation benötigten Services gefunden wurden. Wenn dem so ist wird der Benutzer darüber informiert, und die Simulation kann mittels Betätigung des *Start simulation*-Schalters gestartet werden.

10.2.2 Umsetzung der Fehlertoleranzverfahren

Nachdem der Simulations-Lauf des Beispiel-Modells gestartet wurde, empfängt das Kontrollprogramm nun die von den verschiedenen Services ausgelösten Events, auf die zuvor Subscriptions getätigt wurden. Dabei führen nicht alle diese empfangenen Events notwendigerweise zur Einleitung von Fehlertoleranz-

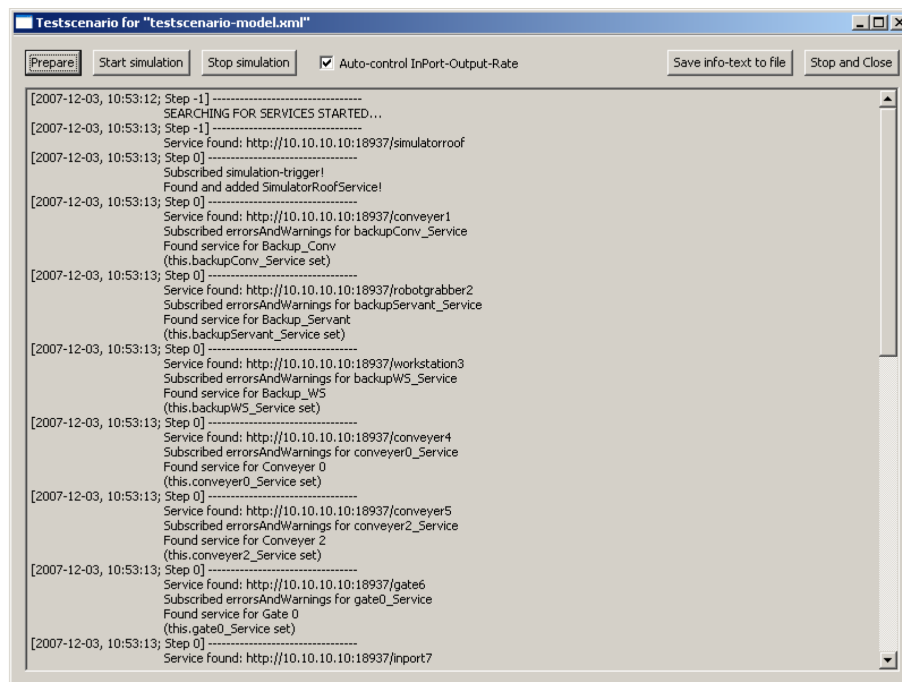


Abbildung 10.3: Grafische Benutzerschnittstelle des Kontrollprogrammes

Maßnahmen. Fehler an bestimmten Komponenten des Modells sind nicht kompensierbar oder tolerierbar, und können somit nicht durch das Kontrollprogramm behoben werden. Im Falle eines Fehlers an einer solchen Komponente muss die Simulation mit einer entsprechenden Mitteilung abgebrochen werden. Bei den Komponenten, auf deren Events hin nichts unternommen werden kann, handelt es sich um die Komponenten *InPort*, *Conveyer 0*, *RobotGrabber 0*, *Conveyer 2*, *Backup_ Conv*, *Unifier* und *OutPort*. Desweiteren wird auf mögliche von der Komponente *Queue* ausgehende Events ebenfalls nicht reagiert (abgesehen gegebenenfalls vom Abbruch der Simulation), allerdings wird auf Events der beiden „an“ der Queue angebrachten Lichtschranken reagiert.

Im Folgenden werden nun die eingeleiteten Aktionen zur Gewährleistung der Fehlertoleranz betrachtet. Dieses geschieht in Abhängigkeit von den Komponenten, die das entsprechende eine solche Aktion auslösende Event gesendet haben. Empfangen werden diese Events in der Methode `receiveEvent(...)`. Dort wird zunächst mittels eines Vergleichs der *Subscription ID* festgestellt, von welchem der Services und somit von welcher der Komponenten der Simulation dieses Event gesendet wurde. Daran anschließend wird die entsprechende Methode zur Verarbeitung eines von diesem Sender empfangenen Events aufgerufen; z. B. also `gate0FiredEvent(...)`, sofern die Komponente *Gate 0* dieses Event gesendet hat.

WorkStation

Von der *WorkStation* gesendete Events können zwei verschiedene Inhalte haben. Zum einen kann die WorkStation darüber informieren, dass im Rahmen der Bearbeitung eines Payloads ein *Defekt* aufgetreten ist. Dieses hat zur Folge, dass die gesamte Komponente R_1 ab dieser Mitteilung zunächst nicht mehr benutzt werden kann. Dementsprechend überprüft das Kontrollprogramm nun, ob eine Umleitung der Payloads auf die zu R_1 redundante Komponente R_2 möglich ist, d. h. es wird überprüft, ob an Komponente aktuell kein Defekt vorliegt. Sollte dem so sein, wird die Komponente KK_1 unter Verwendung ihres Services angewiesen, bis auf weiteres alle Payloads an Komponente R_2 zu liefern. Sollte R_2 hingegen auch defekt sein, wird Komponente KK_1 angewiesen, bis auf weiteres alle Payloads an Komponente R_3 zu liefern. Das System wird hier also *rekonfiguriert*, wobei Komponente R_1 *ausgegliedert*, und eine

der beiden Ersatzkomponente R_2 oder R_3 *eingegliedert* wird (vgl. Abschnitt 2.4.2). Die Eingliederung von Komponente R_2 entspricht dabei der Aktivierung einer vorhandenen *strukturellen Redundanz*, die Aktivierung von Komponente R_3 entspricht der Aktivierung einer *zeitlichen Redundanz* (vgl. Abschnitt 2.2.1).

Der andere mögliche Inhalt eines Events von der WorkStation kann die Mitteilung sein, dass ein zuvor aufgetretener Defekt wieder behoben wurde. Dieses führt dazu, dass das Kontrollprogramm die Komponente KK_1 anweist, ab diesem Zeitpunkt wieder alle auszuliefernden Payloads an Komponente R_1 zu liefern. Diese Komponente wird also *wiedereingegliedert*.

WS_Servant

Mitteilungen, die von der Komponente *WS_Servant* empfangen werden, können z. Zt. nur den Inhalt haben, dass es diesem nicht möglich war, ein Payload an die von ihm zu beliefernde WorkStation zu übergeben. Dieses hat zur Folge, dass der WS_Servant solange keine weiteren Payloads von KK_1 annehmen kann, bis er das aktuell zu liefernde Payload an die WorkStation geliefert hat. Also muss hier Komponente R_1 wieder ausgegliedert werden. Dazu überprüft das Kontrollprogramm, ob an Komponente R_2 aktuell ein Defekt vorliegt, und wenn dem nicht so sein sollte wird diese Komponente mittels einer entsprechenden Anweisung an KK_1 *eingegliedert*. Andernfalls wird KK_1 angewiesen, an Komponente R_3 zu liefern. Diese Anweisung erfolgt dabei derart, dass sie zunächst nur für den nächsten Simulations-Schritt gilt. Kann der WS_Servant auch in diesem Schritt nicht an die WorkStation liefern, so erfolgt von diesem erneut die entsprechende Mitteilung, und das Kontrollprogramm führt erneut die Ausgliederung von R_1 für den dann nächsten Schritt durch. Andernfalls würde R_1 durch das Ausbleiben dieser erneuten Anweisung automatisch wieder eingegliedert. Dieses wurde in dieser Form gelöst, da eine *RobotGrabber*-Instanz, wie es die Komponente WS_Servant ist, keine explizite Mitteilung aussendet, wenn eine zuvor fehlgeschlagene Lieferung in einem späteren Schritt erfolgreich war. Dementsprechend kann das Kontrollprogramm hier nicht erkennen, wann R_1 wieder eingegliedert werden könnte.

Wichtig ist es hier außerdem, dass die sukzessiv wiederholte Ausgliederung von Komponente R_1 unabhängig davon erfolgt, ob Komponente KK_1 in dem betreffenden Simulations-Schritt ein Payload auszuliefern hat; d. h. auch, wenn dort aktuell kein Payload vorliegt, wird diese Ausgliederung durchgeführt.

Backup_WS

Die von dem *Backup_WS* empfangenen Mitteilungen entsprechen – da beides Instanzen einer `WorkStation` sind – denen der Komponente WorkStation. Auch hier müssen Mitteilungen bezüglich eines aufgetretenen oder wieder behobenen Defektes verarbeitet werden. Diese Verarbeitung unterscheidet sich allerdings etwas von der zuvor erläuterten Verarbeitung der Komponente WorkStation.

Empfängt das Kontrollprogramm eine Mitteilung bezüglich eines neu aufgetretenen Defektes an der Backup_WS, wird zunächst mittels einer Anfrage an den *Service* der Komponente KK_1 überprüft, ob dieser aktuell an Komponente R_2 liefert. Sollte dieses nicht so sein, muss hier (abgesehen von einer entsprechenden Ausgabe für den Benutzer) nichts unternommen werden. Sollte KK_1 hingegen gerade so konfiguriert sein, dass an R_2 geliefert wird, muss ein Eingriff durch das Kontrollprogramm erfolgen. Dazu wird in einem ersten Schritt geprüft, ob an Komponente R_1 zur Zeit ein Defekt vorliegt. Ist dieses nicht der Fall, so wird Komponente R_1 *ein-*, und dafür R_2 *ausgegliedert*[1]. Sollte an R_1 hingegen ebenfalls ein Defekt vorliegen, erfolgt die *Eingliederung* von R_3.

Wird hingegen eine Mitteilung empfangen, derzufolge ein zuvor an der Backup_WS aufgetretener Defekt wieder behoben ist, so wird vom Kontrollprogramm in einem ersten Schritt überprüft, ob an Komponente R_1 aktuell ein Defekt vorliegt. Sollte R_1 korrekt arbeiten, sind hier keine weiteren Maßnahmen notwendig, da in diesem Fall Payloads ohnehin an R_1 geliefert werden. Liegt an R_1 andererseits ein Defekt vor, so

[1] Dieser Fall sollte sehr selten sein, da R_2 ohnehin nur eingegliedert wäre, wenn an R_1 ein Defekt vorlag. Damit dieser Fall hier eintritt müsste dieser Defekt gerade behoben sein, aber noch keine Mitteilung darüber als Event erfolgt sein.

kann nun die wieder funktionstüchtige Komponente R_2 als Ersatz für R_1 *eingegliedert* werden. Dieses geschieht mittels einer Anweisung an den Service der Komponente KK_1, bis auf weiteres auszuliefernde Payloads an die Komponente R_2 zu liefern.

Backup_Servant

Ähnlich wie schon zuvor bei der Komponente Backup_WS beschrieben gibt es auch bei der Komponente *Backup_Servant* gewisse Parallelen zu der korrespondierenden Komponente WS_Servant in R_1. So ist auch hier die einzige Mitteilung, die das Kontrollprogramm von der Komponente Backup_Servant empfängt die, derzufolge diese Komponente nicht in der Lage war, ein Payload an die angeschlossene Backup_WS zu liefern. Demzufolge kann R_2 keine weiteren Payloads mehr annehmen. Daher überprüft das Kontrollprogramm nun, ob an Komponente R_1 ein Defekt vorliegt. Dieses ist mit sehr großer Wahrscheinlichkeit der Fall, da ansonsten keine Payloads an R_2 geliefert worden wären. In diesem Fall würde für Komponente R_2 Komponente R_3 eingegliedert. Sollte hingegen an R_1 kein Defekt vorliegen (weil dieser z. B. unmittelbar zuvor behoben wurde), so wird R_1 *eingegliedert*. Diese Eingliederung erfolgt wiederum mittels einer Anweisung an den Service der Komponente KK_1, weitere Payloads entsprechend zu liefern. Im Gegensatz zu dem Vorgehen bei der Komponente WS_Servant muss diese Anweisung nicht nur für den nächsten Schritt erfolgen, sondern derart, dass KK_1 bis auf weiteres an den entsprechenden Nachfolger liefert.

QueueEntryLB / QueueExitLB

Die Überwachung der Mitteilungen, die als Events von den beiden Komponenten *QueueEntryLB* und *QueueExitLB* empfangen werden, dienen der Umsetzung der in Abschnitt 10.1.1 beschriebenen, zusätzlichen *zeitlichen Redundanz*, die mittels Komponente R_4 realisiert werden kann. Wirkliche „Mitteilungen“ werden dabei von diesen beiden Instanzen der `LightBarrier` nicht verschickt, d. h. die von dem Kontrollprogramm empfangenen Events enthalten keine auszuwertenden Daten als Inhalt. Die Events informieren lediglich darüber, dass die entsprechende Lichtschranke „ausgelöst“ wurde, d. h. dass ein Payload die Position der Lichtschranke passiert hat. Aufgrund der Positionen, an denen die beiden Lichtschranken „an“ der Komponente R_3 „angebracht“ sind, bedeutet dieses, dass sich die Anzahl der Payloads, die aktuell von Komponente R_3 befördert werden, verändert hat. Die Anzahl der Payloads auf Komponente R_3 bzw. präziser die relative Auslastung von Komponente R_3 wird nun als Indikator dafür herangezogen, wie hoch das aktuelle *Fehler-Niveau* in dem System ist. Um dieses zu ermitteln wird bei dem von Komponente R_3 bereitgestellten Service die aktuelle Anzahl der dort vorhandenen Payloads und die maximale Kapazität dieser Komponente abgefragt. Die relative Auslastung ergibt sich als der Wert des Quotienten dieser beiden zurückgelieferten Werte. Abhängig davon, ob sich dieser Wert im Vergleich mit dem letzten der zuvor ermittelten Werte verändert hat, und ob dabei zuvor festgelegte *Grenzwerte* über- oder unterschritten wurden, erfolgt dann eine Anweisung an Komponente R_4, die Rate, mit der neue Payloads in das Modell gegeben werden, entweder zu senken oder zu erhöhen. Senkungen erfolgen dabei immer durch Verdoppelung, Erhöhungen durch Halbierung der Zeitrate, mit der neue Payloads in das Modell gegeben werden. Ferner sind dieser Rate gewisse Grenzwerte gesetzt, die niemals unter- oder überschritten werden. Beispielsweise ist das kleinstmögliche Intervall, dass zwischen der Hereingabe zweier Payloads in die Simulation liegen kann, auf 10 Zeiteinheiten gesetzt. Häufigeres Hereingeben von Payloads in die Simulation ist nicht vorgesehen.

Hier bleibt außerdem noch festzuhalten, dass vor Beginn eines Simulations-Laufes die Möglichkeit besteht, speziell diese Redundanz explizit auszuschließen, d. h. nicht zu benutzen. Dafür existiert in der grafischen Benutzerschnittstelle eine *Checkbox* mit der Bezeichnung *Auto-control InPort-Output-Rate* (vgl. Abbildung 10.3). Ist diese ausgewählt, so wird die Steuerung der Rate, mit der Payloads in die Simulation gegeben werden (und somit die durch R_4 realisierte zeitliche Redundanz), mit eingebunden.

Gate 0

Fehler in der Komponente *Gate 0*, die mittels eines Events weitergemeldet und dann entsprechend von dem Kontrollprogramm empfangen werden können nur von der Art sein, dass die aktuell als Nachfolge-Komponente eingestellte Modell-Komponente ein zu lieferndes Payload nicht akzeptiert hat. Dieses ist der Fall, wenn in dem entsprechenden *OutConnector*, an den das Payload hätte geliefert werden müssen, noch ein Payload einer vorherigen Lieferung existiert, das bisher nicht von der über diesen Connector angeschlossenen Komponente abgerufen wurde (vgl. Kapitel 8.2).

Hier sind auf einer ersten Ebene zwei Unterarten dieses Fehlers zu unterscheiden: zum einen existieren Fehler, bei denen eine Lieferung eines Payloads zwar nicht erfolgreich war, aber die Möglichkeit besteht, dieses Payload im nächsten Schritt erneut zu liefern. Dieses kann dann möglicherweise an einen anderen, vom Kontrollprogramm neu gesetzten Nachfolger geschehen. Dem gegenüber stehen Fehler, bei denen ein erneuter Liefer-Versuch im nächsten Simulations-Schritt nicht möglich ist. Dieses ist z. B. der Fall, wenn im nächsten Simulations-Schritt bereits auch das nächste Payload weitergegeben werden muss. Hier müssten dann also in einem Schritt zwei Payloads weitergegeben werden, was nicht möglich ist. Derartige Fehler sind auch im Rahmen der gewährleisteten Fehlertoleranz nicht kompensierbar, und führen daher zwangsläufig zum Abbruch der Simulation. Daher kann die Betrachtung hier auf die Fehler beschränkt werden, bei denen eine Verzögerung der Weitergabe des aktuellen Payloads möglich ist. In einem solchen Fall prüft das Kontrollprogramm nun zunächst, welche der für KK_1 möglichen Nachfolgekomponenten das Ziel des erfolglosen Lieferversuches war. Abhängig davon existieren verschiedene Möglichkeiten, welche der anderen mit KK_1 verbundenen Komponenten als mögliche Ziele eines erneuten Lieferversuches in Frage kommen.

Erfolgte der fehlgeschlagene Lieferversuch an den regulären Nachfolger von KK_1, d. h. an R_1, so sind sowohl R_2 als auch R_3 potentielle neue Lieferziele für dieses Payload. Die Präferenz liegt hier auf R_2, da diese Komponente wie bekannt die Funktionalität R_1 vollständig ersetzt. Daher wird hier überprüft, ob an R_2 aktuell ein Defekt vorliegt. Liegt dort kein Defekt vor, so wird R_2 *eingegliedert*, indem R_2 bei KK_1 als neues Ziel für *dieses* Payload gesetzt wird. KK_1 wird also nicht dauerhaft, sondern nur für den nächsten Schritt auf R_2 umgestellt. Sollte dort hingegen ein Defekt vorliegen, so erfolgt (wiederum nur für den nächsten Simulation-Schritt) mittels der entsprechenden Anweisung an KK_1 die Eingliederung von Komponente R_3. Der Grund für die Umstellung von KK_1 nur für den nächsten Schritt bzw. somit nur für dieses eine Payload liegt darin, dass der bei der Lieferung an R_1 aufgetretene Fehler nicht darin begründet lag, dass an der Komponente WorkStation ein Defekt vorliegt, sondern daran, dass aus nicht näher spezifizierten Gründen die Komponente WS_Servant in diesem Schritt die Annahme dieses Payloads verweigerte. Später folgende Payloads werden hingegen möglicherweise wieder von dieser Komponente akzeptiert. Dieses kann allerdings nur festgestellt werden, indem diese Payloads an diese Komponente geliefert werden. Daher darf lediglich das Lieferziel für das aktuelle Payload umgestellt werden.

War das Ziel des fehlgeschlagenen Liefervorganges hingegen nicht R_1, sondern R_2, so ist nur möglich den Versuch zu unternehmen, dieses Payload im nächsten Schritt auf Komponente R_3 zu liefern. Der Grund dafür liegt darin, dass R_2 nur dann als Nachfolger für Komponente KK_1 gesetzt wird, wenn an R_1 ein Defekt vorliegt, oder diese Komponente zuvor dieses Payload nicht akzeptiert. In beiden Fällen kann dieses Payload nicht an diese Komponente geliefert werden. Falls dieses Payload im Schritt zuvor nicht an R_1 geliefert werden konnte, da dort ein Defekt vorlag, welcher inzwischen wieder behoben ist, so wäre ein entsprechendes Event von dieser Komponente empfangen worden, und KK_1 wäre wieder angewiesen worden, an R_1 zu liefern. Dieser Fall ist hier also ebenfalls auszuschließen.

Das letzte mögliche Ziel dieser fehlgeschlagenen Lieferung kann Komponente R_3 gewesen sein. Wie zuvor erläutert wird KK_1 ausschließlich dann angewiesen, dorthin zu liefern, wenn Lieferungen sowohl an R_1, als auch an R_2 nicht möglich sind. In diesem Fall kann also keine dieser beiden Komponenten als Ersatz für R_3 eingegliedert werden, d. h. es existiert für KK_1 keine Möglichkeit, dieses Payload weiterzureichen. Dieses ist ein nicht zu kompensierender Fehler, der dazu führt, dass die Simulation mit einer entsprechenden Fehlermeldung abgebrochen wird.

10.3 Durchgeführte Simulations-Läufe

Bei der zuvor erfolgten Vorstellung des Kontrollprogrammes für die Gewährleistung von fehlertolerantem Verhalten des in Kapitel 10.1 erläuterten Simulations-Modells wurde herausgestellt, dass dieses Programm zwei Stufen einer Fehlertoleranz-Kontrolle bereitstellen kann. Das Unterscheidungskriterium dieser beiden Stufen ist, ob neben der fehlertoleranten Steuerung der inneren Komponenten des Modells auch (abhängig von dem aktuellen *Fehler-Niveau* innerhalb des Simulations-Modells) die Rate angepasst wird, mit der neue Payloads in die Simulation hineingegeben werden. Hier werden nun die Ergebnisse von Simulations-Läufen mit aktivierter Kontrolle dieser Rate, Simulations-Läufen ohne diese Kontrolle, sowie von Simulations-Läufen gänzlich ohne Gewährleistung von fehlertolerantem Verhalten dargestellt. Die Auswertung dieser Ergebnisse folgt in Kapitel 10.4. Da es theoretisch möglich ist, dass ein solcher Simulations-Lauf *nicht* wegen eines aufgetretenen, schwerwiegenden Fehlers abbricht, wurde dabei festgelegt, dass ein Lauf nach 3000 absolvierten Simulations-Schritten (dieses entspricht einer simulierten Laufzeit von 50 Minuten) beendet und als *erfolgreicher Durchlauf* bewertet wird. Ferner wurden alle Simulations-Läufe mit dem identischen *Zeitbeschleunigungs-Faktor* von 2 (doppelte Echtzeit) durchgeführt. Dieses demonstriert einerseits die Fähigkeit der Simulations-Software, eine Simulation zeitbeschleunigt ablaufen zu lassen, ermöglicht aber dennoch ein angemessenes Beobachten des visualisierten Simulations-Ablaufes. Eine Übersicht über diese Läufe ist dargestellt in Tabelle 10.1. Außerdem sind die Ergebnisse der Läufe, d. h. die Zahl der erfolgreich zurückgelegten Simulations-Schritte, noch zur Veranschaulichung in Abbildung 10.4 in Diagrammform dargestellt.

Lauf Nr.	Fehlertoleranz-verfahren	Steuerung InPort	Simulations-schritte	Ergebnis
01	aktiv	ein	834	Fehler bei Gate 0
02	aktiv	ein	1742	Fehler bei InPort
03	aktiv	ein	> 3000	erfolgreicher Durchlauf
04	aktiv	ein	552	Fehler bei InPort
05	aktiv	ein	> 3000	erfolgreicher Durchlauf
06	aktiv	ein	1184	Fehler bei Gate 0
07	aktiv	ein	786	Fehler bei Gate 0
08	aktiv	ein	1564	Fehler bei Gate 0
09	aktiv	aus	188	Fehler bei Gate 0
10	aktiv	aus	291	Fehler bei InPort
11	aktiv	aus	381	Fehler bei InPort
12	aktiv	aus	482	Fehler bei InPort
13	aktiv	aus	402	Fehler bei InPort
14	aktiv	aus	492	Fehler bei InPort
15	aktiv	aus	432	Fehler bei InPort
16	aktiv	aus	521	Fehler bei InPort
17	inaktiv	entfällt	65	Fehler bei Gate 0
18	inaktiv	entfällt	65	Fehler bei Gate 0
19	inaktiv	entfällt	65	Fehler bei Gate 0
20	inaktiv	entfällt	65	Fehler bei Gate 0
21	inaktiv	entfällt	65	Fehler bei Gate 0
22	inaktiv	entfällt	65	Fehler bei Gate 0
23	inaktiv	entfällt	65	Fehler bei Gate 0
24	inaktiv	entfällt	65	Fehler bei Gate 0

Tabelle 10.1: Übersicht über die verschiedenen Simulations-Läufe

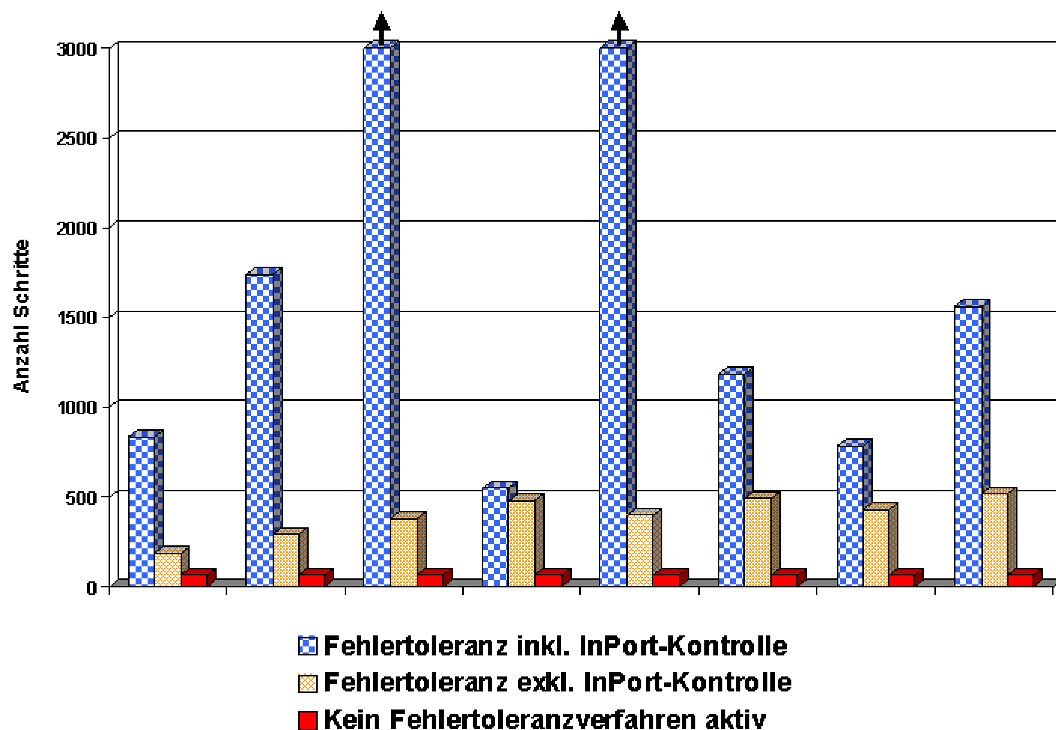


Abbildung 10.4: Gegenüberstellung der Anzahl der erfolgreichen Simulations-Schritte

Wie aus dieser Tabelle hervorgeht, wurden jeweils acht Simulations-Läufe mit jeder der beiden Ausprägungsformen des Fehlertoleranzverfahren sowie acht Durchläufe ohne Anwendung eines Fehlertoleranzverfahrens durchgeführt. Dabei geben die Spalten *Fehlertoleranzverfahren* und *Steuerung InPort* an, ob das Kontrollprogramm zur Gewährleistung der Fehlertoleranz den Lauf kontrollierte, und in welcher Ausprägung dieses gegebenenfalls arbeitete. In den weiteren Spalten ist dargestellt, wie viele Simulations-Schritte bis zum Beenden der Simulation ausgeführt wurden, und ob die Simulation erfolgreich war (d. h. manuell beendet wurde). War der Simulations-Lauf nicht erfolgreich, wird dort die Komponente genannt, in der der zum Abbruch führende Fehler auftrat.

10.4 Auswertung

Hier soll nun eine kurze Betrachtung und Auswertung der in Tabelle 10.1 dargestellten Ergebnisse der Simulations-Läufe durchgeführt werden. Dafür seien zunächst die Läufe 17 bis 24 betrachtet; dieses sind die Läufe, bei denen keinerlei fehlertolerante Steuerung eingesetzt wurde. Wie in Tabelle 10.1 eindeutig abzulesen ist, führt dieses reproduzierbar zu dem Ergebnis, dass die Simulation nach 65 simulierten Zeiteinheiten wegen eines Fehlers an der Komponente *Gate 0* abgebrochen wird. Der Grund dafür ist ersichtlich aus den in Kapitel 10.1 erläuterten Details der einzelnen Komponenten des Simulations-Modells. Dort ist zu erkennen, dass die Komponente *InPort* zu Beginn eines Simulations-Laufes so konfiguriert ist, dass von dieser Komponente alle 10 Zeiteinheiten ein neues Payload in die Simulation gegeben wird. Diese Rate wird bei nicht vorhandener externer Kontrolle während eines Simulations-Laufes nicht verändert. Ferner ist die Komponente *WorkStation* so konfiguriert, dass sie für die Bearbeitung eines jeden Payloads 25 Zeiteinheiten benötigt. Sollte die Komponente *Gate 0* also keiner externen Steuerung unterworfen sein, ist es unausweichlich, dass es für diese Komponente nach einer gewissen Anzahl von Simulations-Schritten (eben offensichtlich nach 65 Zeiteinheiten) nicht mehr möglich ist, das aktuell verarbeitete Payload an die fix eingestellte Nachfolge-Komponente *WS_Servant* bzw. R_1 weiterzuleiten, da diese kein weiteres

Payload akzeptiert. Ohne die Existenz einer externen Kontrollinstanz kann dieser Fehler nicht behoben werden, da KK_1 somit nicht auf einen anderen Nachfolger umgeschaltet werden kann. Dieses führt zum Abbruch der Simulation. Im weiteren Verlauf werden diese Simulations-Läufe wegen ihres reproduzierbaren Ergebnisses nicht mehr berücksichtigt. Es wird festgehalten, dass das Modell in der vorliegenden Konfiguration ohne Anwendung einer fehlertoleranten Steuerung nicht länger als 65 Zeiteinheiten simuliert werden kann.

Betrachtet werden nun die Simulations-Läufe *mit* aktivem Kontrollprogramm zur Gewährleistung der fehlertoleranten Steuerung. Dort ist zu erkennen, dass sämtliche Läufe, bei denen neben der Steuerung der Komponente KK_1 auch noch die Komponente R_4 (d. h. der InPort) gesteuert wurde, bis zum Abbruch wegen eines nicht-tolerierbaren Fehlers länger liefen als die, bei denen die Steuerung von R_4 deaktiviert war. Tabelle 10.2 stellt die bei diesen Läufen durchschnittlich bis zum Abbruch wegen eines Fehlers absolvierten Simulations-Schritte dar.

Art der genutzten Fehlerkontrolle	Durchschnittliche Schrittzahl bis Abbruch	Geglättete Schrittzahl
mit Steuerung des InPort	1110 Schritte	1092 Schritte
ohne Steuerung des InPort	399 Schritte	413 Schritte

Tabelle 10.2: Vergleich der erfolgreichen Simulations-Schritte

Bei der Berechnung der Durchschnitts-Werte wurden die Läufe 03 und 05 nicht mit einbezogen, da diese nach Erreichen der als *Erfolg* definierten Anzahl von 3000 Schritten manuell beendet wurden. Die *geglättete Schrittzahl* wurde ferner ermittelt, indem der jeweils maximale und minimale Wert bei der Berechnung des Durchschnitts nicht berücksichtigt wurden.

Vergleicht man nun diese Durchschnittswerte miteinander, so fällt auf, dass sich bei Steuerung der Rate, mit der neue Payloads in die Simulation gegeben werden, die Anzahl der Simulations-Schritte bis zum Abbruch mehr als verdoppelt. Das Verhältnis liegt bei ungefähr 2,6 : 1 im Falle des geglätteten Durchschnitts, bzw. sogar bei ca. 2,8 : 1 bei dem nicht-geglätteten Durchschnitt. Betrachtet man außerdem noch einmal Tabelle 10.1, so erkennt man dort, dass unter allen durchgeführten Simulations-Läufen lediglich zwei Läufe existieren, bei denen die als *Erfolg* definierte Anzahl von absolvierten Schrittzahlen erreicht wurde. Diese beiden Erfolge wurden unter Verwendung des Verfahrens erzielt, das die Rate der von Komponente R_4 ausgestoßenen Payloads abhängig vom *Fehler-Niveau* im System steuert. Dieses legt die Vermutung nahe, dass es für den erfolgreichen Betrieb dieses Systems unter Verwendung der in Kapitel 10.1 beschriebenen Konfigurationen der einzelnen Komponenten notwendig ist, diese Rate abhängig vom Fehler-Niveau zu steuern. Diese Vermutung liegt bei ausschließlicher Betrachtung der Konfiguration nicht unbedingt nahe: dort lässt sich zwar erkennen, dass die Payload-Rate von einem Payload alle 10 Zeiteinheiten die Kapazität der Komponente R_1 definitiv überlastet (da die Komponente WorkStation pro Payload-Bearbeitung 25 Zeiteinheiten benötigt). Allerdings wäre es dort ebenfalls naheliegend zu vermuten, dass dieses durch die zu R_1 redundante Komponente R_2, in der ein Payload in 10 Zeiteinheiten bearbeitet wird, zumindestens kompensiert wird. Die durchgeführten Simulationen haben nun aufgezeigt, dass dieses bereits bei den Defekt-Wahrscheinlichkeiten von 3% in R_1 bzw. 2% in R_2 nicht mehr so ist.

Um dieses Verhalten weiter zu untersuchen wurden noch einige Simulations-Läufe durchgeführt, bei denen die Defekt-Wahrscheinlichkeit der Komponente R_2 von 2% auf nur noch 1% herabgesetzt wurde. Bei diesen Läufen stieg die Anzahl der vor einem Abbruch absolvierten Simulations-Schritte bei Verwendung der Kontroll-Variante *ohne* Steuerung der Payload-Rate des InPort auf im geglätteten Durchschnitt 708 Schritte, was einem Zugewinn von über 75% entspricht. Die als Kriterium für einen *erfolgreichen* Simulations-Lauf festgelegten 3000 absolvierten Simulations-Schritte konnten hier allerdings ebenfalls in keinem der durchgeführten Simulations-Läufe erreicht werden. Dennoch zeigt dieses auf, welche Auswirkungen hier die Veränderung von nur einem Konfigurations-Parameter eines Simulations-Elementes haben kann.

Was bei der Betrachtung von Tabelle 10.1 ebenfalls noch auffällt ist, dass sich die Fehler, wegen denen die Läufe abgebrochen wurden, abhängig von der Ausprägung des verwendeten Fehlertoleranzverfahrens an einer Komponenten häufen. Bei eingebundener Steuerung der Payload-Rate des *InPort* liegt der Grund für den Abbruch zu über 66% in der Komponente *Gate 0*; ist diese Steuerung nicht eingebunden liegt der Grund sogar zu 87,5% in der Komponente *InPort* selbst. Der Vollständigkeit halber soll der Grund dafür hier noch kurz und informell dargestellt werden. Dafür muss zunächst betrachtet werden, was für Fehler an diesen beiden Komponenten auftreten:

- *Fehler an InPort*: hier treten Fehler ausschließlich dann auf, wenn ein vom InPort in die Simulation zu gebendes Payload nicht an die Nachfolge-Komponente *Conveyer 0* weitergereicht werden kann. Dieses ist der Fall, wenn *Conveyer 0* gestoppt ist, da am Ende dieser Komponente ein Payload durch die Komponente *RobotGrabber 0* übernommen werden muss.
- *Fehler an Gate 0*: die Fehler, die in dieser Komponente auftreten, sind derart, dass es dieser Komponente nicht möglich ist, ein Payload an eine ihrer drei Nachfolge-Komponenten weiterzuleiten. Dieses ist dann der Fall, wenn die Komponenten R_1 und R_2 defekt sind, und Komponente R_3 gestoppt ist, da am Ende dieser Queue gerade ein Payload an die Komponente *RobotGrabber 0* weitergereicht werden muss.

Der Grund für die Häufung der Fehler am *InPort* bei der Kontroll-Variante ohne Steuerung der Payload-Rate dieses InPort liegt nun genau darin, dass diese Rate mit zunehmendem Fortschreiten der Simulation nicht angepasst wird. Je länger die Simulation andauert, umso größer wird die Anzahl der auf Komponente R_3 transportierten Payloads. Demzufolge steigt die Frequenz an, mit der Payloads durch die Komponente *RobotGrabber 0* von R_3 angenommen werden müssen. Gleichermaßen erhöht sich die Zeit, während der Komponente *Conveyer 0* angehalten ist, da dort ebenfalls ein Payload von *RobotGrabber 0* angenommen werden muss, und somit die Wahrscheinlichkeit, dass die Komponente *InPort* während einer solchen Phase vergeblich versucht, ein Payload an Komponente it Conveyer 0 zu übergeben. Dort tritt dann der zuvor beschriebene Fehler auf. Wird hingegen die Payload-Rate des InPort in Abhängigkeit von der Auslastung der Komponente R_3 angepasst, so verringert sich die Wahrscheinlichkeit dieses Fehlers, und der wahrscheinlichere Fehlerort verschiebt sich hin zu KK_1. Dieses ist dadurch zu erklären, dass KK_1 hier verglichen mit R_4 in deutlich mehr Schritten aktiv ist, bzw. dadurch, dass die inaktiven Phasen von R_4, in denen an dieser Komponente kein Fehler auftreten kann, zunehmen.

Kapitel 11

Zusammenfassung und Fazit

Ziel dieser Diplomarbeit war die Entwicklung eines Werkzeuges zur *Simulation automatisierter Systeme*, die unter Berücksichtigung des *service-orientierten Konzeptes* entwickelt wurden. Mittels dieses Werkzeugs sollte es ermöglicht werden, *Fehlertoleranzverfahren* zu testen, die die simulierten Systeme über deren Services kontrollieren und steuern. Das Simulations-Werkzeug sollte dabei selbst gemäß dem SOA-Konzept entwickelt werden, und außerdem derart modular aufgebaut sein, dass es möglich ist, wenn nötig bestimmte Teile der Software gegen neue Teile auszutauschen.

Dieses Ziel konnte erfolgreich erreicht werden. Dazu war es zunächst notwendig, die Themengebiete der *Fehlertoleranz*, der *service-orientierten Architekturen* und der *Simulation* näher zu betrachten, und dabei die für die Erstellung dieses Simulation-Werkzeuges wichtigen Aspekte herauszuarbeiten. So wurde im Rahmen der Betrachtung der Fehlertoleranz der Begriff der *Redundanz* angeführt, und die verschiedenen möglichen Ausprägungen von Redundanz vorgestellt. Außerdem wurden die Teilbereiche bzw. Teilschritte der Fehlertoleranz, nämlich die *Fehlerdiagnose* und die *Fehlerbehandlung*, erläutert.

Bei der Vorstellung des Konzeptes der *service-orientierten Architekturen* wurden in einem ersten Schritt in die Grundlagen hinter diesem Konzept, wie z. B. der Begriff des *Dienstes* oder das Prinzip der *losen Kupplung*, eingeführt. Daran anschließend wurde mit den *Web Services* ein Implementierungsansatz einer service-orientierten Architektur vorgestellt. Dieses umfasste für Web Services wichtige Begriffe und Komponenten wie z. B. das *Universal Description, Discovery and Integration*-Protokoll, welches bei den Web Services die Funktion des Dienstverzeichnisses bereitstellt, oder auch die *Web Services Description Language*, die in diesem Ansatz die Beschreibung der Dienst-Schnittstellen leistet. Abschließend wurde das *Devices Profile for Web Services* vorgestellt, welches das Konzept der Web Services um Funktionalitäten bezüglich des *Eventing* und des *Auffindens von Diensten* erweitert.

Im Rahmen der anschließenden Vorstellung der verschiedenen Ansätze zur Simulations-Durchführung wurden zunächst einige Grundlagen wie z. B. eine formale *Definition* des Begriffes *Simulation* eingeführt. Ferner wurde der Begriff des *Systems* eingeführt, und erläutert, was mit diesem Begriff bezeichnet wird, und wie sich derartige Systeme anhand ihrer Ausprägungen *unterteilen* lassen. Danach war es möglich, von dem Begriff des Systems ausgehend den Begriff des *Modells* einzuführen. Es wurde erläutert, wie sich Modelle *klassifizieren* lassen, und welche Arten von *Modell-Sichtweisen*, wie z. B. das *Black Box*- oder *White Box*-Modell, existieren. Abgeschlossen wurde diese Vorstellung der Simulation mit einer Erläuterung der verschiedenen *Simulations-Verfahren*, die benutzt werden können. Hier existieren neben den *zeitkontinuierlichen Verfahren* die *zeitdiskreten Verfahren*, wobei sich letztere noch weiter nach der Art, wie die diskreten Simulations-Zeitpunkte gewählt werden, unterteilen ließen. Hier wurden sowohl die *zeitgesteuerten* als auch die *ereignisgesteuerten* Verfahren vorgestellt.

Nachdem mit der Vorstellung der Simulations-Konzepte die Erarbeitung der theoretischen Grundlagen, die die Voraussetzung für die erfolgreiche Entwicklung des Simulations-Werkzeuges waren, abgeschlossen waren, wurde daran anschließend das zu simulierende *technische System* analysiert. Dazu wurden zunächst

der *Systemzweck* und die einzelnen *Systemkomponenten* identifiziert und erläutert. Daraus ergaben sich die Anforderungen, die an das Simulations-Modell und an die Umgebung, in der ein solches Modell simuliert wird, gestellt werden. Aus diesen identifizierten Anforderungen folgte dann das gewählte und in dem Simulations-Werkzeug umgesetzte *Simulationsverfahren*. Gewählt wurde hier ein *zeitdiskretes* und dabei *zeitgesteuertes* Verfahren.

In einem letzten Schritt vor dem eigentlichen Entwurf und der eigentlichen Implementierung wurden noch explizit die an das Werkzeug gestellten Anforderungen aufgeführt und erläutert. Zentrale Anforderungen waren dabei die Möglichkeit, *beliebige Modelle* des zugrundeliegenden technischen Systems zusammenstellen und simulieren zu können, sowie die Umsetzung des *SOA-Konzeptes*. Außerdem wurden die Grundlagen der Implementierung vorgestellt, wie z. B. die gewählte logische Zweiteilung des Werkzeuges in die *Simulations-Umgebung* und das *Modell-Framework*, oder die bei der Implementierung verwendeten Bibliotheken wie z. B. der *WS4D J2ME*-Stack.

Der Entwurf und die Implementierung des Simulations-Werkzeuges erfolgte daran anschließend zweigeteilt. Zunächst wurde das entwickelte *Modell-Framework* vorgestellt, das die Komponenten bereitstellt, aus denen sich ein Simulations-Modell zusammensetzt. Dazu wurde in einem ersten Schritt auf die *Strukturen* eingegangen, die hinter einem hier verwendeten Modell stehen. Dabei wurde festgehalten, dass sich jedes Modell innerhalb genau definierbarer Grenzen bezüglich seiner Umwelt befindet, und dass jedes Modell ganz allgemein aus *Modell-Komponenten* besteht, die miteinander *verbunden* sind. Auf diese *Verbindungen* wurde anschließend näher eingegangen, und es wurden sowohl die theoretischen Überlegungen hinter als auch die praktische Implementierung der diese Verbindungen realisierenden *Connectoren* vorgestellt. Im weiteren Verlauf wurden dann die Grundlagen der einzelnen Modell-Komponenten vorgestellt, und diese in die drei Gruppen *Entry-Elemente*, *Exit-Elemente* und *Inside-Elemente* eingeteilt. Daraus ergaben sich gemeinsame Eigenschaften aller Komponenten, oder zumindestens der Komponenten innerhalb einer solchen Gruppe, was zu der verwendeten *Strukturierung* der Modell-Komponenten führte. Nachdem mit dieser Feststellung der Komponenten-Struktur die Grundlagen abgeschlossen waren, wurde die konkrete Implementierung der Modell-Komponenten vorgestellt. Ausgangspunkt war dabei das sogenannte *BaseSimElement*, welches die Basis aller Modell-Komponenten darstellt. Dieses wurde verfeinert durch die Implementierungen der Elemente *EntryElement*, *ExitElement* und *InsideElement*, welche wiederum auf einer letzten Stufe von den eigentlichen, konkreten Modell-Komponenten wie z. B. *InPort*, *Gate* oder *Conveyer* spezialisiert wurden. Die Vorstellung dieser Komponenten wurde abgeschlossen durch die Erläuterung der *Web Services* dieser Komponenten, sowie einer kurzen Vorstellung der Hilfs-Komponenten, d. h. den sogenannten *AuxiliaryDevices*.

Der zweite Teil der Vorstellung des Simulations-Werkzeuges umfasste die *Simulations-Umgebung*, also den Teil des Simulations-Werkzeuges, in den ein zu simulierendes Modell *eingebunden* wird. Dazu wurde zunächst ein Überblick über den modularen Aufbau der Simulations-Umgebung gegeben, und dabei aufgezeigt, welche Module innerhalb der Implementierung existieren. Diese wurden gemäß ihrer Aufgaben in die *graphischen* und die *administrativen* Module unterteilt. Außerdem wurde das „Dach"-Modul *SimulatorRoof* vorgestellt, unter welchem alle Module vereint werden. Im Anschluß daran erfolgte die Vorstellung der Details der einzelnen Module, bzw. der Vorgehensweisen, mittels der diese Module die ihnen übertragenen Aufgaben erledigen. Die Vorstellung der Simulations-Umgebung wurde vervollständigt durch eine kurze Erläuterung der einzelnen *Threads*, in denen das Programm innerhalb der virtuellen Java-Maschine läuft, sowie durch eine explizite Vorstellung des von der Simulations-Umgebung bereitgestellten *Web Services*. Letzteres umfasste insbesondere die Auflistung der Funktionen der Simulations-Umgebung, die mittels dieses Services genutzt werden können.

Den Abschluss dieser Diplomarbeit bildete ein *Anwendungsbeispiel* für die Verwendung des Simulations-Werkzeuges. Dazu wurde zunächst das verwendete, mit Hilfe der Software erstellte Simulations-Modell vorgestellt, und dabei insbesondere die in diesem Modell vorgesehenen *Redundanzen* erläutert. Ferner wurde das *externe Kontrollprogramm* vorgestellt, das der fehlertoleranten Steuerung des Modells dient. Die Umsetzung der Fehlertoleranz für das Anwendungsbeispiel wurde ebenfalls explizit herausgearbeitet und vorgestellt. Daran anschließend erfolgte die Beschreibung einiger Simulations-Durchläufe mit diesem Modell und dem Fehlertoleranz-Kontrollprogramm. Die Ergebnisse dieser Durchläufe wurden dargestellt,

und in einem letzten Schritt ausgewertet.

11.1 Erweiterungsmöglichkeiten

Die im Rahmen dieser Diplomarbeit entstandene Software ist – sowohl, was die Simulations-Umgebung als auch das Modell-Framework betrifft – fertig und in ihrer derzeitigen Form voll nutzbar. Dennoch bleiben einige Punkte, an denen Erweiterungen dieses Werkzeuges denkbar wären.

Eine naheliegende Erweiterung wäre das Hinzufügen weiterer Komponenten zu dem Modell-Framework. Durch die Art der Implementierung dieses Frameworks ist dieses sehr einfach zu realisieren: in einem ersten Schritt müsste festgestellt werden, ob die zu implementierende neue Komponente an Anfang, in der Mitte oder am Ende einer Transportkette in einem Modell steht. Ist dieses erfolgt, müsste die neue Komponente entsprechend entweder eine der Klassen `EntryElement`, `InsideElement` oder `ExitElement` erweitern, und die von dieser Klasse vorgegebenen Methoden mit ihrem komponentenspezifischen Verhalten füllen. Veränderungen an der Simulations-Umgebung wären bei einer solchen Erweiterung nicht notwendig.

Ebenfalls denkbar wären Erweiterungen bezüglich der Visualisierung einer Simulation. Diese ist bisher bewusst recht einfach gehalten, um hier Ressourcen (z. B. bezüglich der Rechenleistung) einzusparen, die möglicherweise besser an anderen Stellen des Programmes zur Verfügung stehen sollten. Dennoch wäre es denkbar, dass auch hier eine Erweiterung des Werkzeuges vorgenommen werden könnte. Die Visualisierung einer Animation ist, wie in Kapitel 8 erläutert, weitestgehend in die Modell-Komponenten ausgelagert. Eine Erweiterung dieser müsste also dort in diesen Komponenten ansetzen.

Eine weitere sinnvolle Erweiterung könnte in der stärkeren Kontrolle der mittels der Web Services durchgeführten Kommunikation liegen. Gerade, wenn eine Simulation mit einem größeren Zeitbeschleunigungs-Faktor betrieben wird, ist es vorstellbar, dass die Nachrichtenlaufzeiten dazu führen, dass Nachrichten möglicherweise erst bei einer Fehlertoleranz-Kontrollinstanz eintreffen, wenn der Simulations-Schritt, in dem diese Nachricht erzeugt wurde, bereits abgeschlossen ist. Hierzu müsste eine Synchronisation der Nachrichten mit dem Takt der Simulation vorgenommen werden. Dieses führte bei den bisherigen Tests und Auswertungen zwar nie zu Problemen, stellt aber dennoch eine sinnvolle Erweiterung zur Erhöhung der Funktions-Sicherheit des Werkzeuges dar.

11.2 Fazit

Als abschließendes Fazit lässt sich hier festhalten, dass die gestellten Ziele vollständig erreicht wurden. Mittels dem erstellten Simulations-Werkzeug ist es möglich, Simulationen von *beliebigen* Modellen aus den vom Modell-Framework zur Verfügung gestellten Komponenten zusammenzustellen und durchzuführen. Dabei ist die Kontrolle dieser Simulationen und der einzelnen Komponenten eines Simulations-Modells über die von diesen bereitgestellten *Web Services* voll gegeben, so dass entsprechend angebundene Fehlertoleranzverfahren ebenso mit einer solchen Simulation interagieren können, wie sie es auch mit einem real vorhandenem, automatisierten System tun würden. Somit ist also die Analyse dieser Verfahren anhand derartiger Simulationen möglich. Sofern ein erstelltes Simulations-Modell ein automatisiertes System präzise genug nachbildet, sind die Ergebnisse dieser Simulations-Läufe im Bezug auf die Anwendung des Fehlertoleranzverfahrens auf das reale System übertragbar.

Ebenfalls erreicht wurde das geforderte Ziel des *modularen Software-Aufbaus*. Dabei bewirkt insbesondere die strikte Trennung der Simulations-Umgebung von dem Modell-Framework, dass diese beiden Teile komplett unabhängig voneinander modifiziert werden können. Prinzipiell wäre es sogar denkbar, dass ein völlig neues Modell-Framework für eine völlig andere Art von zu simulierendem Real-System entwickelt werden könnte, welches sich bei Beachtung der Struktur des jetzigen Frameworks problemlos in die Simulations-Umgebung einbinden lassen würde. Außerdem ist es durch den ebenfalls modularen Aufbau der Simulations-Umgebung möglich, bestimmte Module dieser Umgebung gegen anders arbeiten-

de Module auszutauschen. Beispielsweise wäre es denkbar, die Steuerung einer Simulation weg von dem verwendeten zeitgesteuerten, hin zu einem ereignisgesteuerten Verfahren umzustellen. Neben minimaler Änderungen im Modul `SimulatorRoof` wäre es hierfür lediglich notwendig, das Modul `SimTimer` gegen ein entsprechend anders arbeitendes Modul auszutauschen. Insbesondere berücksichtigt wurde in diesem Zusammenhang auch die Möglichkeit, das Werkzeug ohne jede grafische Oberfläche zu starten.

Abbildungsverzeichnis

2.1 Unterteilung der Redundanz nach [Ech90] 6
2.2 Statische Redundanz als 2-von-3-System nach [Ech90] 8
2.3 Gegenseitige Redundanz der beiden Systemkomponenten A und B nach [Ech90] 9
2.4 Unterteilung von Zuverlässigkeit und Fehlertoleranz nach [Ech90] 10

3.1 Rollen und Aktionen in einer SOA nach [Mel07] 14
3.2 Web Services - System nach [Mel07] 16
3.3 Beispiel eines SOAP-Envelopes mit Header und Body (nach [Mel07]) 17
3.4 Überblick über die WSDL-Komponenten 18
3.5 DPWS-Stack nach [ZBB+07] 20
3.6 Nachrichtenaustausch in DPWS nach [Mic06] 21

4.1 Ausprägungen von Systemen nach [Pag91] 25
4.2 Einteilung der Simulations-Verfahren nach [FPW90] 29

7.1 Unterteilung der Software mit beispielhaften Interaktionen 44
7.2 Darstellung des gleichen SWT-Fensters unter verschiedenen Betriebssystemen (Bilderquelle: [Eclb]) 45
7.3 Module des WS4D-J2ME - Stacks nach [ZBB+07] 46
7.4 Software-Architektur des WS4D-J2ME - Stacks nach [ZBB+07] 47

8.1 Allgemeine Struktur eines potentiellen Simulations-Modells 50
8.2 Unterscheidung von Verbindungen anhand ihrer Richtung 51
8.3 Transformation eines Connectors in je einen In- und OutConnector 51
8.4 Struktur der Modell-Komponenten 56
8.5 Klassenhierarchie der Simulations-Elemente 57
8.6 Hinzufügen von zu protokollierenden Daten 58
8.7 Beispiel: Konfigurations-Dialog und Ablauf beim Zusammenstellen dieses Dialoges 60
8.8 Symbolabbildungen der Simulations-Elemente (Teil 1) 63
8.9 Symbolabbildungen der Simulations-Elemente (Teil 2) 65

8.10 Symbolabbildungen der Simulations-Elemente (Teil 3) 66
8.11 Klassen-Struktur der Services der Modell-Komponenten 70

9.1 Module der Simulations-Umgebung 74
9.2 Interaktion der Module 75
9.3 Startbildschirm mit Fortschrittsanzeige 76
9.4 Ablauf eines Simulations-Schrittes 80
9.5 Module der graphischen Benutzerschnittstelle 84
9.6 Struktur des NewModelCreator - Moduls 87
9.7 Das Interface ISimulationListener 89
9.8 Threads der Simulations-Umgebung 92
9.9 Darstellung des Web Services der Simulations-Umgebung im DPWS-Explorer von [WS4a] 94

10.1 Wichtige Teile des Simulations-Modells 98
10.2 Kommunikationsstruktur des Anwendungsbeispiels 101
10.3 Grafische Benutzerschnittstelle des Kontrollprogrammes 102
10.4 Gegenüberstellung der Anzahl der erfolgreichen Simulations-Schritte 107

B.1 Ein einfaches Beispiel-Modell 123

Tabellenverzeichnis

8.1 Einteilung der Komponenten des Real-Systems in Entry-, Exit- und Inside-Elements . . . 54

10.1 Übersicht über die verschiedenen Simulations-Läufe 106
10.2 Vergleich der erfolgreichen Simulations-Schritte 108

A.1 Existierende SWT-Implementierungen für verschiedene Plattformen 121

Literaturverzeichnis

[ABe06] Alexander, J., Box, Don und Cabrera, L. et al.: *Web Services Transfer (WS-Transfer).* http://www.w3.org/Submission/2006/SUBM-WS-Transfer-20060315/, 2006.

[Arb82] Arbeitsgruppe der NTG-Fachausschüsse 6 und 30: *Zuverlässigkeitsbegriffe im Hinblick auf komplexe Systeme und Hardware.* Nachrichtentechnische Zeitschrift, Band 35, Heft 5, Seiten 327 – 333, 1982.

[Axi07] *Apache Axis2.* http://ws.apache.org/axis2/, 2007.

[BBe06] Ballinger, K., Bissett, B. und Box, D., et al.: *Web Services Metadata Exchange (WS-MetadataExchange).* http://specs.xmlsoap.org/ws/2004/09/mex/WS-MetadataExchange.pdf, 2006.

[BEG86] Belli, F., Echtle, K. und Görke, W.: *Methoden und Modelle der Fehlertoleranz.* Informatik-Spektrum, Band 9, Heft 2, Seiten 68 – 81, 1986.

[Bos92] Bossel, H.: *Modellbildung und Simulation.* Friedr. Vieweg & Sohn Verlagsgesellschaft mbH, 1992.

[Com01] Comer, D.: *Computer Networks and Internets.* Prentice-Hall, Inc., 3. Auflage, 2001.

[DD02] Doberkat, E. und Dissmann, S.: *Einführung in die objektorientierte Programmierung mit Java.* Oldenbourg, 2. Auflage, 2002.

[Ech90] Echtle, K.: *Fehlertoleranzverfahren.* Springer-Verlag, 1990.

[Ecla] Eclipse Foundation: *Eclipse - an open development platform.* http://www.eclipse.org/.

[Eclb] Eclipse Foundation: *SWT: The Standard Widget Toolkit.* http://www.eclipse.org/swt/.

[Ecl07a] Eclipse Foundation: *Eclipse Project - Release Build: 3.3.* http://download.eclipse.org/eclipse/downloads/drops/R-3.3-200706251500/index.php#SWT, 2007.

[Ecl07b] Eclipse Foundation: *Eclipse Public License - v 1.0.* http://www.eclipse.org/org/documents/epl-v10.php, 2007.

[ERe02] Eastlake, D., Reagle, J. und Solo, D. et al.: *XML-Signature Syntax and Processing.* http://www.w3.org/TR/xmldsig-core/, 2002.

[FPW90] Frauenstein, T., Pape, U. und Wagner, O.: *Objektorientierte Sprachkonzepte und Diskrete Simulation.* Springer-Verlag, 1990.

[HM] Hunter, J. und McLaughlin, B.: *JDOM.* http://www.jdom.org.

[Hun02] Hunter, J.: *JDOM and XML Parsing, Part 1 - 3.* Oracle Magazine, 2002.

[IDS02] Imamura, T., Dillaway, B. und Simon, E.: *XML Encryption Syntax and Processing.* http://www.w3.org/TR/xmlenc-core/, 2002.

[Jün88] Jünemann, R.: *Integrierte Materialflußsysteme.* Verlag TÜV Rheinland GmbH, 1988.

[Kli81] Klingenberg, G.: *Zeitkontinuierliche Simulation im fertigungstechnischen Bereich*, 1981.

[KPH+07] Krumm, H., Pohl, A., Holland, F., Lück, I. und Stewing, F.-J.: *Service-orientation and Flexible Service Binding in Distributed Automation and Control Systems*, 2007.

[Krü74] Krüger, S.: *Simulation. Grundlagen, Techniken, Anwendungen.* De Gruyter, 1974.

[Lan03] Langner, T.: *Web Services mit Java.* Markt + Technik Verlag, 2003.

[Lek93] Lektorat des B.I.-Wissenschaftsverlags: *Duden Informatik: Ein Sachlexikon für Studium und Praxis.* Bibliographisches Institus & F.A. Brockhaus AG, 2. vollst. überarb. und erw. Aufl. Auflage, 1993.

[Mel07] Melzer, I.: *Service-orientierte Architekturen mit Web Services.* Spektrum Akademischer Verlag, 2007.

[Mic06] Microsoft Corporation: *The Devices Profile for Web Service specification.* http://specs.xmlsoap.org/ws/2006/02/devprof/devicesprofile.pdf, 2006.

[Möl92] Möller, D.: *Modellbildung, Simulation und Identifikation dynamischer Systeme.* Springer-Verlag, 1992.

[Nec98] Necker, T.: *Entwicklung eines objektorientierten Werkzeugs für verschiedene Verfahren der parallelen ereignisgesteuerten Simulation*, 1998.

[Nie77] Niemeyer, G.: *Kybernetische System- und Modelltheorie.* Vahlen, 1977.

[Org] Organization for the Advancement of Structured Information Standards: *OASIS Web Services Security (WSS) TC.* http://www.oasis-open.org/committees/wss/.

[Pag91] Page, B.: *Diskrete Simulation. Eine Einführung mit Modula-2.* Springer-Verlag, 1991.

[Sun] Sun Microsystems, Inc.: *Java Reference Documentation.* http://java.sun.com/reference/docs/.

[Sun04] Sun Microsystems, Inc.: *JDK 5.0 Documentation.* http://java.sun.com/j2se/1.5.0/docs/index.html, 2004.

[Sun07] Sun Microsystems, Inc.: *The Java ME Platform: the Most Ubiquitous Application Platform for Mobile Devices.* http://java.sun.com/javame/, 2007.

[tSN07] ten Hompel, M., Schmidt, T. und Nagel, L.: *Materialflusssysteme.* Springer-Verlag, 2007.

[van07] van Emgelen, R.: *gSOAP: C/C++ Web Services.* http://www.cs.fsu.edu/~engelen/soap.html, 2007.

[Wora] World Wide Web Consortium / W3C: *Extensible Markup Language (XML).* http://www.w3.org/XML/.

[Worb] World Wide Web Consortium / W3C: *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition).* http://www.w3.org/TR/soap12/.

[Worc] World Wide Web Consortium / W3C: *SOAP Version 1.2 Part 2: Adjuncts (Second Edition).* http://www.w3.org/TR/soap12-part2/.

[Word] World Wide Web Consortium / W3C: *W3C Document Object Model (DOM)*. http://www.w3.org/DOM/.

[Wore] World Wide Web Consortium / W3C: *Web Services Addressing (WS-Addressing)*. http://www.w3.org/Submission/ws-addressing/.

[Worf] World Wide Web Consortium / W3C: *Web Services Policy 1.5 - Framework*. http://www.w3.org/TR/2007/REC-ws-policy-20070904.

[Wor01] World Wide Web Consortium / W3C: *XML Information Set*. http://www.w3.org/TR/2001/REC-xml-infoset-20011024/, 2001.

[Wor04] World Wide Web Consortium / W3C: *Web Services Architecture*. http://www.w3.org/TR/ws-arch/, 2004.

[WS4a] *WS4D - Web Services for Devices*. http://www.ws4d.org.

[WS4b] *WS4D J2ME-Stack*. http://www.ws4d.org/j2me-overview.html.

[ZBB+07] Zeeb, E., Bobek, A., Bohn, H., Prüter, S., Pohl, A., Krumm, H., Lück, I., Golatowski, F. und Timmermann, D.: *WS4D: SOA-Toolkits making embedded systems ready for Web Services*. http://www4.cs.uni-dortmund.de/ Pohl/publications/osspl07.pdf, 2007.

Anhang A

SWT-unterstützte Betriebssysteme

Wie in Abschnitt 7.3.1 beschrieben ist für für die Benutzung der SWT-Bibliothek notwendig, dass eine entsprechende Implementierung dieser für das verwendete Betriebssystem existiert. Zum Zeitpunkt der Fertigstellung dieser Diplomarbeit existierten Implementierungen der *Standard Widget Toolkit* Bibliothek und der dazugehörigen Systembibliothek in der Version 3.320 für die folgenden Betriebssysteme bzw. Plattformen:

Plattform	Dateiname	Größe
Windows (w32)	swt-3.3-win32-win32-x86.zip	4,6 MB
Windows (x86/WPF)	swt-3.3-wpf-win32-x86.zip	2,9 MB
Windows CE (ARM PocketPC)	swt-3.3-win32-wce_ppc-arm-j2se.zip	1,7 MB
Windows CE (ARM PocketPC, J2ME profile)	swt-3.3-win32-wce_ppc-arm-j2me.zip	1,7 MB
Linux (x86/GTK 2)	swt-3.3-gtk-linux-x86.zip	4,2 MB
Linux (x86_64/GTK 2)	swt-3.3-gtk-linux-x86_64.zip	4,4 MB
Linux (PPC/GTK 2)	swt-3.3-gtk-linux-ppc.zip	4,3 MB
Linux (x86/Motif)	swt-3.3-motif-linux-x86.zip	5,6 MB
Solaris 8 (SPARC/GTK 2)	swt-3.3-gtk-solaris-sparc.zip	4,2 MB
Solaris 8 (SPARC/Motif)	swt-3.3-motif-solaris-sparc.zip	4,0 MB
QNX (x86/Photon)	swt-3.3-photon-qnx-x86.zip	2,8 MB
AIX (PPC/Motif)	swt-3.3-motif-aix-ppc.zip	4,1 MB
Mac OSX (Mac/Carbon)	swt-3.3-carbon-macosx.zip	4,0 MB

Tabelle A.1: Existierende SWT-Implementierungen für verschiedene Plattformen

Zu erhalten sind diese unter den in der Tabelle angegebenen Dateinamen bei [Ecl07a]. Aufgrund der stetigen Weiterentwicklung des Eclipse Project ist davon auszugehen, dass es zumindestens für einige der genannten Plattformen in absehbarer Zeit neuere Versionen dieser Bibliotheken gibt. Relevant für die im Rahmen dieser Diplomarbeit erstellte Software und mit dieser getestet sind aber nur die oben genannten Bibliotheken in der Version 3.320, da nicht sichergestellt werden kann, dass sich die Struktur der Bibliotheken im Rahmen ihrer fortschreitenden Entwicklung verändert. Dieses führt unter Umständen dazu, dass derartige neuere Versionen nicht mehr vollständig kompatibel zu dem hier erstellten Code wären, was bereits bei einigen SWT-Versionssprüngen der Fall war.

Anhang B

XML-strukturiertes Simulations-Modell

Dieses hier ist ein sehr einfaches Beispiel für ein Simulations-Modell, das aus einem `InPort` besteht, an den sich ein `Conveyer` anschließt, der die zu transportierenden *Payloads* wiederum an einen `OutPort` weiterreicht. Dort verlassen diese das System wieder. Hier folgt nun zunächst der XML-Quellcode für das Modell, welches unter diesem Code noch einmal als Abbildung zu finden ist.

```
<?xml version="1.0" encoding="UTF-8"?>
<Simulation-File LAST-CHANGED="2007-11-25,␣15:39:20" AUTHOR="krueger"
DPWS-ENABLED="true" ELEMENT-COUNT="3" CREATION-DATE="2007-11-25,
15:38:22" VERSION="1.0" NAME="Einfaches␣Beispiel-Modell">
  <ELEMENT TYPE="InPort">
    <DPWS-STATUS-ENABLED>false</DPWS-STATUS-ENABLED>
    <ELEMENT-LENGTH>1</ELEMENT-LENGTH>
    <EXTERNAL-CONTROLLABLE>true</EXTERNAL-CONTROLLABLE>
    <ID>1</ID>
    <LOGGING-ENABLED-FOR-ELEMENT>true</LOGGING-ENABLED-FOR-ELEMENT>
    <NAME>InPort 0</NAME>
    <OP_DELAY>10</OP_DELAY>
    <OP_RATE>5</OP_RATE>
    <OUTCONN-TO>Conveyer 0</OUTCONN-TO>
    <POSITION_X>106</POSITION_X>
    <POSITION_Y>77</POSITION_Y>
  </ELEMENT>
  <ELEMENT TYPE="Conveyer">
    <CAPACITY>20</CAPACITY>
    <DPWS-STATUS-ENABLED>false</DPWS-STATUS-ENABLED>
    <ELEMENT-LENGTH>100</ELEMENT-LENGTH>
    <EXTERNAL-CONTROLLABLE>true</EXTERNAL-CONTROLLABLE>
    <ID>2</ID>
    <INCONN-FROM>InPort 0</INCONN-FROM>
    <LOGGING-ENABLED-FOR-ELEMENT>true</LOGGING-ENABLED-FOR-ELEMENT>
    <MODE>AutoHalted</MODE>
    <NAME>Conveyer 0</NAME>
```

```
    <OUTCONN-TO>OutPort 0</OUTCONN-TO>                                              28
    <POSITION_X>279</POSITION_X>                                                    29
    <POSITION_Y>166</POSITION_Y>                                                    30
    <THROUGHPUT-TIME>25</THROUGHPUT-TIME>                                           31
  </ELEMENT>                                                                        32
  <ELEMENT TYPE="OutPort">                                                          33
    <DISPOSAL-TIME>12</DISPOSAL-TIME>                                               34
    <DPWS-STATUS-ENABLED>false</DPWS-STATUS-ENABLED>                                35
    <ELEMENT-LENGTH>0</ELEMENT-LENGTH>                                              36
    <EXTERNAL-CONTROLLABLE>true</EXTERNAL-CONTROLLABLE>                             37
    <ID>3</ID>                                                                      38
    <INCONN-FROM>Conveyer 0</INCONN-FROM>                                           39
    <LOGGING-ENABLED-FOR-ELEMENT>true</LOGGING-ENABLED-FOR-ELEMENT>                 40
    <NAME>OutPort 0</NAME>                                                          41
    <PAYLOAD-STORE-CAPACITY>8</PAYLOAD-STORE-CAPACITY>                              42
    <POSITION_X>534</POSITION_X>                                                    43
    <POSITION_Y>269</POSITION_Y>                                                    44
  </ELEMENT>                                                                        45
</Simulation-File>                                                                  46
```

Dieser Code stellt das folgende Modell dar (hier abgebildet zum Simulationszeitpunkt 0, d. h. vor der Durchführung des 1. Schrittes):

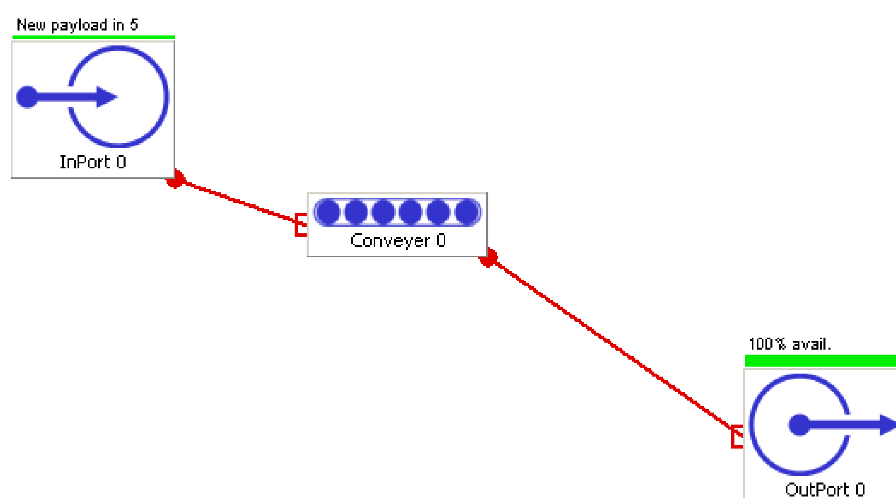


Abbildung B.1: Ein einfaches Beispiel-Modell