Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 270

Testwerkzeug zur Simulation dynamischer Datenströme

Janis Schöck

Studiengang: Softwaretechnik

Prüfer/in: Prof. Dr.-Ing. habil. Bernhard Mitschang

Betreuer/in: Dipl.-Inf. Pascal Hirmer

Beginn am: 15. Oktober 2015

Beendet am: 21. April 2016

CR-Nummer: B.8.1, D.3.4, I.6.0

Kurzfassung

Eingebetteten Systemen sind in der heutigen Zeit in einer großen Bandbreite von Produkten zu finden. Immer häufiger werden dabei auch Netzwerkschnittstellen mit verbaut, die eine Verbindung mit den immer flächendeckender verfügbaren Netzwerken ermöglichen. Die Kommunikation dieser Systeme untereinander, sowie deren Anbindung an Serversysteme und Endgeräte wird über die fortlaufende Entwicklung von Schnittstellen immer weiter vorangetrieben. Um verarbeitende Systeme wie CEP-Engines zu testen, und Operatoren zur Auswertung der kommunizierten Informationen besser entwickeln und validieren zu können, werden Daten zur Simulation einer komplexen Umgebung benötigt.

Diese Arbeit befasst sich daher mit der Entwicklung einer Plattform zur dynamischen Generierung logisch zusammenhängender Werte mit Hilfe von modellbasierten Simulationen. Dabei wird unter anderem auch eine Programmier- und Scriptsprachen übersetzende und ausführende Komponente ausgearbeitet, welche dynamische Änderungen am Modellzustand berechnet, und auf Konzepten kontextfreien Grammatiken aufbaut. Ein Konzept zur Filterung und Übertragung der simulierten Werte wird ebenfalls entwickelt. Mit einer Weboberfläche wird schlussendlich das Konfigurieren von Modellen und Testfällen, sowie der Schnittstellen zu getesteten Systemen ermöglicht.

Inhaltsverzeichnis

1 Einführung und Motivation **11**

2 Grundlagen **13**
2.1 Internet of Things und Industrie 4.0 . . . 13
2.2 Komplexe Ereignisverarbeitung . . . 14
2.2.1 Ereignisströme und Ereignisstromverarbeitung . . . 14
2.2.2 Komplexe Ereignisse . . . 15
2.3 Kontextfreie Grammatiken . . . 16

3 Anforderungen **19**
3.1 Funktionale Anforderungen . . . 19
3.2 Nicht-Funktionale Anforderungen . . . 20

4 Konzeptionelle Lösung **23**
4.1 Modellbasierte Simulation . . . 23
4.1.1 Modellbeschreibung . . . 24
4.2 Simulationsdurchführung . . . 25
4.2.1 Zugriffe auf Objektattribute . . . 25
4.2.2 Ausführbare Programme . . . 27
4.2.3 Virtuelle Prozessoren . . . 28
4.3 Programm-Kompilation . . . 29
4.3.1 Kompilationskette . . . 29
4.3.2 Sprachdefinition . . . 30
4.4 Filterkonzept . . . 31
4.5 Netzwerkschnittstellen . . . 32
4.6 Unit-Tests . . . 32

5 Umsetzung und Implementierung **35**
5.1 Implementierung der Programm-API . . . 35
5.1.1 Ausführbare Programme . . . 36
5.1.2 Prozessor . . . 37
5.1.3 Sprache . . . 40
5.2 Implementierung der Serveranwendung . . . 44
5.3 Umsetzung der Weboberfläche . . . 44

6 Zusammenfassung und Ausblick **47**
6.1 Ausblick 48

Abbildungsverzeichnis

2.1 Komplexe Ereignisverarbeitung 16

4.1 Veranschaulichung des Metamodells als Klassendiagramm 24
4.2 Beispiel für Zugriff auf Objektattribute 26
4.3 Ablauf der Ausführung eines Beispielprogramms 28
4.4 veranschaulichung der Kompilationsabfolge 29

5.1 Veranschaulichung der Zusammenhänge und Abhängigkeiten der Pakete in der API-Komponente 36
5.2 Veranschaulichung der Durchreichung von Anfragen 39
5.3 Reduziertes Klassendiagramm des Syntax-Paketes 43

Verzeichnis der Listings

5.1 Auszug der abstrakten Klasse ExecutableProgramItem 36
5.2 Auszüge der Schnittstellenklasse ProgramAccessibleComponent 38
5.3 Auszug der Klasse StackLevel 39
5.4 Auszug aus der Methode execute der Klasse Processor 40
5.5 Methode executeStep der Klasse Processor 41
5.6 Beispiel der Definition von Token-Typen 42
5.7 Beispiel einer Regel-Definition 43

1 Einführung und Motivation

Seit der breiten Etablierung von *Wireless LAN* in Firmen und Privathaushalten, sowie der weitreichenden Abdeckung öffentlicher Orte durch Mobilfunknetze, wird die Vernetzung *eingebetteter Systeme* (engl. Embedded Systems - kurz ES) immer relevanter. Diese kommen heute üblicherweise in den meisten Haushaltsgegenständen, Werkzeugmaschinen, Fabrikanlagen, Gebäuden und Fahrzeugen, sowie in einer Bandbreite weiterer Geräte zum Einsatz. Vergleichsweise neu ist aber, dass *ES* Informationen bereitstellen, die nicht mehr ausschließlich der Funktion des einbettenden Gerätes dienen, und in der Lage sind, Daten über Netzwerke auszutauschen. Solche Daten können von anderen Geräten und Servern genutzt werden, um Ausfälle zu kompensieren, Messwerte abzugleichen, Systeme zu regeln oder sonstige Mehrgewinne zu erzielen. Diese Infrastrukturentwicklung wird mit dem Begriff *Internet of Things (kurz IoT)* [Fri10] bezeichnet, welcher erstmals 1999 von Kevin Ashton im Kontext von RFID[Ash09] geprägt wurde. Durch zunehmend erschwinglichere Hardware wird die Verbreitung des *IoT* immer schneller vorangetrieben, weshalb in modernen Anwendungsszenarien oft tausende Geräte und Sensoren existieren, welche in ständigem Austausch stehen.

Zur Interpretation konfluierter Datenpakete werden Systeme eingesetzt, welche Datenströme aus diskreten Datensätzen ableiten, um diese systematisch zu verarbeiten. Dadurch können Informationen gewonnen werden, welche den Informationsgehalt der einzelnen Teilnachrichten übersteigen. Werden darüber hinaus mehrere Datenströme gleichzeitig betrachtet, wird von *Complex-Event-Processing* (CEP)[Adi07][Gar13] gesprochen. Hieraus lassen sich Informationen ableiten, die keiner Einzelkomponente zur Verfügung stehen, und sich ausschließlich aus der gesamtheitlichen Betrachtung mehrerer Datenquellen ergeben. Die dafür eingesetzten Verarbeitungsplattformen werden unter dem Begriff *Streaming-Engines*[15] oder *Complex-Event-Processing-Engines*[Adi07] (kurz CEP-Engines) zusammengefasst. Sie stellen häufig mächtige Anfragesprachen zur Verfügung, welche in ihrer Syntax und Komplexität Datenbank-Anfragesprachen wie SQL[Gen06] [15] nahe stehen, und mit diesen häufig verwandt, oder von ihnen abgeleitet sind[15]. Überdies bieten derartige Plattformen oft eine Vielzahl verschiedener Anbindungsschnittstellen für verschiedene Protokolltypen.

Aufgrund der Komplexität dieser Plattformen, Sprachen und Systeme, sind menschliche Fehler in der Entwicklung von Anwendungsfällen alltäglich. Wird nicht nach ihnen gesucht, bleiben sie häufig unentdeckt, und werden erst auffällig, wenn das System deutliches Fehlverhalten ausweist. Sicherheitskritische Anwendungsfälle können somit nicht zuverlässig auf den Technologien des *IoT* aufgebaut werden, ohne dieses Problem organisiert und nachhaltig zu lösen. Außerdem lassen sich selbst unkritische Szenarien wenn überhaupt nur mit hohen

Kosten in einer realen Umgebung verifizieren. Wie auch in klassischen Softwareprojekten sind daher systematische und wiederholbare Tests notwendig, um Fehler frühzeitig ausfindig zu machen, und so ein hohes Maß an Zuverlässigkeit zu gewährleisten. Dies ist, für die meisten Anwendungsszenarien, nur durch Simulation kosteneffizient lösbar.

In Anlehnung an das Forschungsprojekt *SitOPT* des Fachbereiches Anwendungssoftware aus dem Institut für Parallele und Verteilte Systeme der Universität Stuttgart, befasst sich diese Arbeit daher mit der Konzeption und Entwicklung einer Plattform, mit der sich *Streaming-Engines* durch Anwendungsfälle mit möglichst geringem Aufwand und Fehlerpotential testen lassen. Dabei sollen Testdaten aus *Modellen* abgeleitet werden können, um so logisch konsistente Datenpakete an die zu testenden Systeme abzusetzen. Die angelegten *Modelle* sollen von Unit-Test-nahen Szenarios verwendet werden, um konkrete Situationen zu simulieren, und für diese qualifizierte Resultate zu protokollieren. Werte sollen darüber hinaus durch Filter verzerrt oder geglättet werden können, um realistischere Messdaten zu generieren, welche von Idealwerten abzuweichen. Die Plattform wird unter dem Namen *SimPl* (Simulationsbasierte Test-Plattform) bereitgestellt, vorausgesetzt dieser ist zum Zeitpunkt der Veröffentlichung noch verfügbar.

Die entwickelte Plattform bietet darüber hinaus Schnittstellen, die auch zum Testen von Anwendungsfällen anderer *Streaming-Engines* genutzt werden können. Nach Fertigstellung soll *SimPl* als *Open-Source-Projekt* unter der GnUPL zur Verfügung gestellt werden.

2 Grundlagen

Die in Kapitel 2 - Grundlagen beschriebenen Grundlagen erläutern die für die übrige Arbeit zugrunde gelegten Annahmen. Die in 2.1 - Internet of Things und Industrie 4.0 beschriebenen Konzepte dienen dem grundlegenden Verständnis der Motivation der Arbeit, sowie der Einordnung der in 2.2 - Komplexe Ereignisverarbeitung beschriebenen Grundlagen. Diese werden wiederum für die Konzeptionelle Lösung, sowie die Implementierung vorausgesetzt. 2.3 erläutert die in dieser Arbeit zugrunde gelegten Konzepte kontextfreier Grammatiken, welche für die Übersetzung von Quelltexten von Programmier- und Scriptsprache genutzt wird.

2.1 Internet of Things und Industrie 4.0

Der Begriff *Internet of Things*[Ash09], beschreibt allgemein das Netzwerk all jener Objekte, welche über eine Schnittstelle zu bestehenden Netzwerkinfrastrukturen verfügen und über diese kommunizieren. Im Gegensatz zum herkömmlichen Internet bestehend aus Servern, Personal Computern, Terminals und Mobilen Geräten, schließt dies auch solche Objekte mit ein, welche nicht direkt oder indirekt vom Nutzer dazu verwendet werden auf das Internet zuzugreifen, beispielsweise Gebäude und Fahrzeuge, aber auch Haushaltsgeräte und Werkzeugmaschinen. Diese *Smart Objects*[Ale10] zeichnen sich im allgemeinen durch eine elektronische Komponente aus, welche über eine Netzwerkschnittstelle verfügt. Infrage kommen beispielsweise *eingebettete Systeme*[Hea03] (engl. Embedded Systems - kurz ES), aber auch *Mikrosysteme*[Nad04] (engl. Micro-Electromechanical Systems - kurz MEMS). Das IoT ist heute bereits allgegenwärtig und schließt fast alle neueren Endgeräte mit ein. Daher ist eine losgelöste Sichtweise auf einzelne Teilausschnitte fast immer unvollständig, es sei denn, es handelt sich um ein vom Internet entkoppeltes Intranet.

Neben der oben beschriebenen Bedeutung, werden in dieser Arbeit mit dem Begriff IoT auch solche Infrastruktur-relevanten Hard- und Softwaretechnologien und Komponenten zusammengefasst, die zur Vernetzung und Kommunikation der beteiligten Geräte genutzt werden, insbesondere die dafür eingesetzten Netzwerke.

Im Zusammenhang mit *IoT* steht auch der Begriff *Industrie 4.0*[Mar15], welcher eine angestrebte vierte industrielle Revolution postuliert. Zentrale Bedeutung kommt dabei vielfältigen Optimierungen der Wertschöpfungskette zu - allen voran die Etablierung von *Smart-Factories*[Mar15] (kurz SF), welchen *cyber-physische Systeme*[Dan16] (kurz CPS) zugrunde liegen, die direkt

auf Entwicklungen und Technologien aufbauen, die mit denen des IoT überlappen. Je nach Definition sind CPS daher entweder als Teilbereich des IoT, mindestens aber als eng verwandt zu sehen. CPS nutzen die Vernetzung mechatronischer Komponenten, um dezentrale Entscheidungen zu treffen. Sie unterscheiden sich von herkömmlichen ES hauptsächlich durch ihre Integration mit physischen Komponenten, und in vernetzte Umgebungen.

Die Vision von SF sieht CPS als Standbein für optimierte dynamische Wertschöpfungsketten, und die daraus resultierenden Gewinne in der Fertigung, wie beispielsweise die Reduzierung von Rüstzeiten im Kontext kleiner Losgrößen. Zentrale Rolle spielen hier neben Arbeitssicherheit vor allem Stabilität und Zuverlässigkeit. Obwohl die Integrationskosten von CPS oft sekundär sind, da ihnen meist großen Gesamtfertigungsmengen gegenüberstehen, ist hier der Einsatz simulierter Umgebungen zur Beschleunigung der Integration und Installation von großer Bedeutung.

Diese Arbeit bedient sich Konzepten und Technologien, welche im Kontext mit dem IoT und Industrie 4.0 stehen, beispielsweise der Schnittstellen und Protokolle, die in virtuellen Umgebungen zum Einsatz kommen. Die in dieser Arbeit entwickelte Plattform dient wiederum der Verbesserung der Zuverlässigkeit der jeweiligen Applikationen. Dies soll die Entwicklung und Integration von CPS und IoT-Komponenten erleichtern.

2.2 Komplexe Ereignisverarbeitung

In 2.2 werden vor allem die zugrunde liegenden Zielplattformen erklärt, an deren Operationsweise die entwickelte Plattform anknüpft.

2.2.1 Ereignisströme und Ereignisstromverarbeitung

Unter *Datenströmen* versteht man einen kontinuierlichen Fluss diskreter Datensätze, welche fortlaufend von einem Sender an einen oder mehrere Empfänger übermittelt werden. Sie sind innerhalb eines Datenstroms von gleichem Datentyp, und können einzelne Werte, oder aus Werten zusammengesetzte Tupel sein. Der Vorteil von Datenströmen gegenüber einzelnen Nachrichten, ist die Möglichkeit, diskrete Datensätze verkettet zu betrachten, und diese in den Kontext bereits erhaltener Nachrichten zu setzen. Außerdem wird immer auch Lebendigkeit des Senders, und der Zustand des Übertragungskanals implizit mitgeteilt, da der Empfänger beides aus dem Erhalt der Nachricht schließen kann. Selbst zeitweise ungenutzte Datenströme werden daher meist durch sogenannte Keepalive Signale, auch Heartbeats oder Pings genannt, aufrecht erhalten, da der Datenstrom sonst in aller Regel nach einer Zeitüberschreitung (engl. Timeout) vom nicht erreichten Empfänger als geschlossen betrachtet wird. Datenströme werden üblicherweise in der *Interprozesskommunikation* eingesetzt, um stetige Informationsflüsse zwischen Prozessen zu ermöglichen.

Bei der *Ereignisverarbeitung*[Adi07] spricht man stattdessen von *Ereignisströmen*. Der Begriff *Ereignis* leitet sich von dem Paradigma der ereignisorientierten Programmierung ab, bei der die Steuerung des Programmflusses im Vordergrund steht. Im Kontext der *Ereignisverarbeitung* ist jedoch meist der Informationsgehalt vordergründig, da eingehende Nachrichten den Ablauf der verarbeitenden Plattform in der Regel nur soweit beeinflussen, dass sie korrekt verarbeitet werden. Sie dienen aber nicht im klassischen Sinn ihrer Steuerung. Ausgehende Ereignisse können hingegen durchaus Prozesse steuern. *Ereignisse* im Sinne der *Ereignisverarbeitung* können aber auch Vorgänge in der realen Welt sein, wie beispielsweise das Drücken eines Knopfes oder das Bremsen eines Fahrzeuges. *Ereignisströme* sind demnach auch aufeinanderfolgende Ereignisse, welche nicht zwingend für die Weiterverarbeitung ausgelegt sind, aber dennoch zu diesem Zweck genutzt werden. Bedeutsam für die Verarbeitung ist lediglich, dass *reale Ereignisse* von einem Sensor erfasst, und zur Verarbeitung als *virtuelle Ereignisse* bereitgestellt werden. Solche *Ereignisse* und jene, welche nicht selbst das Resultat von *Ereignisverarbeitung* sind, werden als *Elementarereignisse* bezeichnet. Im Folgenden beschränkt sich der Begriff *Ereignis* lediglich auf solche in der virtuellen Welt, selbst wenn diese Ereignisse in der Realität repräsentieren.

Ein Datenstrom im klassischen Sinn geht üblicherweise von jeweils einem Sender und Empfänger einer Nachricht aus. Werden Daten an mehrere Empfänger übermittelt, geschieht dies in der Regel durch das Senden mehrerer Einzelnachrichten an verschiedene Empfänger. Ausnahmen hierfür können beispielsweise Broad- oder Anycasts sein. Ein unidirektionaler *Ereignisstrom* kann aber auch an keinen spezifischen Empfänger gerichtet sein. Ereignisse können beispielsweise empfängerlos an einen Speicher übergeben werden, um dort von potentiellen Interessenten abgeholt zu werden. Beispiel für derartige Informationsflüsse sind Nachrichten-Ticker und Blogs.

Unter *Ereignisstromverarbeitung* versteht man die Interpretation und Nutzbarmachung der in Ereignissen enthaltenen Informationen, sowie der aus der gesamtheitlichen Betrachtung dieser gewonnenen Mehrinformationen. Diese können wiederum für eine Bandbreite unterschiedlicher Zwecke eingesetzt werden, wie beispielsweise der Steuerung von Werkzeugmaschinen, oder der Initiierung von Aktienanan- oder Verkäufen.

Ereignisse, welche bei Verarbeitungsplattformen eingehen, werden üblicherweise zunächst in einem Speicher abgelegt. Sie werden dann durch die Plattform meist in *First-In-First-Out*-Reihenfolge (kurz FIFO), also sequenziell abgearbeitet. Anschließend können sie, je nach Anwendungsfall, für spätere Zugriffe durch sogenannte *Fenstern (engl. Windows)* zeitweise zwischengespeichert bleiben.

2.2.2 Komplexe Ereignisse

Bei der *Komplexen Ereignisverarbeitung* (kurz CEP) werden mehrere Ereignisströme gesamtheitlich betrachtet[Adi07], und die so gewonnenen Mehrinformationen als *komplexe Ereignisse*

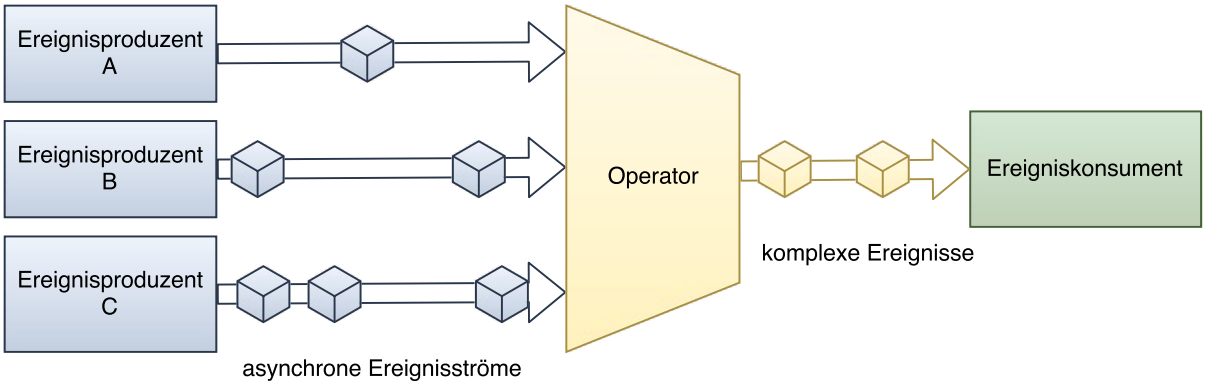

Abbildung 2.1: Komplexe Ereignisverarbeitung

weiter genutzt. *Komplexe Ereignisse* bezeichnen also solche Ereignisse, welche von erhaltenen Ereignissen aus unterschiedlichen Quellen abgeleitet sind.

Zur *komplexen Ereignisverarbeitung* werden *Komplexe Ereignisverarbeitungsplattformen* (engl. kurz CEP-Engines) eingesetzt. Diese verwenden zur Generierung von *Ereignisströmen* sogenannte *Operatoren*, welche einen oder mehrere *Eingangsströme* kontinuierlich betrachten. *Operatoren* werden üblicherweise als kontinuierliche Anfragen in einer Engine-abhängigen *Anfragesprache* verfasst, oder durch Graphen definiert. Je nach Semantik des *Operators* werden dann zu bestimmten Zeitpunkten Ereignisse aus den Informationen der *Eingangsereignisse* abgeleitet. Für *komplexe Ereignisse* betrachtete *Eingangsereignisse* können selbst komplex sein. Man spricht dann von *verketteten Operatoren*.

In der CEP wird zwischen *Ereignisproduzenten* und *Ereigniskonsumenten* unterschieden. Erstere stellen *Ereignisse* in Form von *Ereignisströmen* zur Verfügung, wohingegen Letztere sie empfangen, und deren Informationen nutzen. *Ereignisproduzenten* sind beispielsweise Sensoren, aber auch *Operatoren* von *CEP-Engines*, welche neben Aktoren auch zu den *Ereigniskonsumenten* zählen. Die zu entwickelnde Plattform soll die Umgebung von Operatoren simulieren. Sie fungiert also in erster Linie als Ereignisproduzent. Allerdings werden zur Bewertung der zu testenden Operatoren auch die ausgegebenen komplexen Ereignisse ausgewertet, weshalb sie auch als Ereigniskonsument agiert.

2.3 Kontextfreie Grammatiken

Mit *kontextfreien Grammatiken* lassen sich durch *Ableitung kontextfreie Sprachen* erzeugen[Mar11]. Sie sind Teilmenge der *formalen Grammatiken*, welche ihrerseits die Menge der *formale Sprachen* erzeugen. Diese werden hauptsächlich für mathematische Beweisführungen eingesetzt. *kontextfreie Grammatiken* sind 4-Tupel und bestehen aus jeweils einem *Alphabet* von Terminalenerminalen T und Nichtterminalen N, sowie einer Menge an Produktionsregeln P und einem Startsymbol S, welches Teil von N ist. Unter einem *Alphabet* versteht man eine

endliche Menge von *Symbolen*. Die in der Menge der Terminale enthaltenen Symbole müssen verschieden von den Symbolen in der Menge der Nichtterminale sein. Symbolfolgen die in der durch die Grammatik beschriebenen Sprache liegen nennt man *Wörter*. Um die Menge aller *Wörter* zu bilden, die durch eine *formale Grammatik* erzeugt werden, müssen, ausgehend vom Startsymbol, die Produktionsregeln angewendet werden, um Nichtterminale durch eine Verkettung von Terminalen und Nichtterminalen zu ersetzen. Sind keine Nichtterminale mehr in einer Symbolfolge, ist diese fertig abgeleitet, und es handelt sich um ein Wort in der durch die Grammatik erzeugten Sprache. Dieser Vorgang wird als *Ableitung* bezeichnet. Die Produktionsregeln von *kontextfreien Grammatiken* sind dadurch beschränkt, dass jede Produktionsregel nur genau einem Nichtterminal Symbolfolgen zuweist[Mar11]. Dies bedeutet, das ein Nichtterminal unabhängig von benachbarten Symbolen ist, und stets nach der gleichen Regel abgeleitet wird.

Die meisten Programmiersprachen zählen ebenfalls zu den *formalen Sprachen*, weshalb sich ihre jeweilige Syntax durch Grammatiken ausdrücken lässt. Die meisten relevanten Programmiersprachen sind überdies auch kontextfrei. Somit lässt sich die Syntax einer kontextfreien Programmiersprache auch durch eine kontextfreie Grammatik beschreiben. In den Sprachspezifikationen mancher Programmiersprachen, wie beispielsweise Java 8[Jam15], sind explizite Produktionsregeln angegeben, nach denen die Syntax aufgebaut ist. Ist dies der Fall, so ist es möglich, für eine entsprechende Sprache Vorschriften festzulegen, nach denn eine abgeleitete Zeichenfolge eingelesen werden kann, um auf die Baumstruktur der Ableitungsschritte zu schließen, durch die sie diese erzeugt. Es ist nicht auszuschließen, dass verschiedene Ableitungsfolgen das selbe Wort erzeugen. Ist dies der Fall, ist das Wort mehrdeutig.

Ein auf seine Ableitungsabfolge zurückführbarer Quelltext kann von einem Compiler übersetzt werden. Dieser Eigenschaft wird sich im Kapitel 4 - Konzeptionelle Lösung und bedient.

3 Anforderungen

Im Zusammenhang mit dem Projekt SitOPT des Institutes für Parallele und Verteilte Systeme der Universität Stuttgart wird für diese Arbeit die Konzeption und Implementierung einer Testplattform (im Folgenden mit Plattform bezeichnet) gefordert. Die Anforderungen an die Plattform werden in die gängigen Kategorien der *funktionalen* und *nicht-funktionalen Anforderungen* unterteilt. *Funktionale Anforderungen* verlangen durch Tests nachweisbare Funktionalitäten von der Plattform, wohingegen *nicht-funktionale Anforderungen* die Ausprägung der Implementierung, sowie die Laufzeiteigenschaften und Dokumentation beschreiben, ohne jedoch eindeutig nachgewiesen werden zu können.

Über die im Folgenden gelisteten Anforderungen existieren noch implizite Anforderungen, welche direkt oder indirekt aus der beschriebenen Anforderungen folgen. Dies beinhaltet unter anderem verständlich kommentierten Quelltext, eine nachvollziehbare Dokumentation, Standard-konforme Entwicklungsmuster, eine zweckorientierte Paketunterteilung und Modularisierung sowie konsequente Objektorientierung.

3.1 Funktionale Anforderungen

Die im Folgenden beschriebenen funktionalen Anforderungen definieren maßgeblich das Verhalten, und den Funktionsumfang der Plattform.

Anforderung F1 - Dynamische Ausprägung von Datensätzen Um robuste Test durchführen zu können, muss die Plattform in der Lage sein, dynamische Datensätze abzusetzen. Dies erlaubt, im Gegensatz zu statischen Datensätzen, die flexible Anpassung von Tests an geänderte Bedingungen oder Testanforderungen, sowie das einfache Adaptieren von Werten durch das Modifizieren weniger Variablen. Die Kosten für die Entwicklung und Definition von Testdaten werden dadurch erheblich reduziert.

Anforderung F2 - Flexible Ausprägung von Datenströmen Um verschiedene Szenarien und Geräte abdecken zu können, ist es erforderlich, dass die Datentypen der von den Datenströmen übermittelten Datenpakete flexibel ausgeprägt sein können. Dadurch wird erreicht, dass nicht nur wenige vorgegebene Datenströme angebunden werden können.

Anforderung F3 - Simultane Datenströme Um ganzheitliche Szenarien abdecken zu können, ist es erforderlich, dass die Plattform das Übermitteln simultaner Datenströme erlaubt. Außerdem müssen Datenströme voneinander unabhängige Datentypen unterstützen, und mit verschiedenen Empfängern in Verbindung stehen können.

Anforderung F4 - Logisch korrelierte Werte Um Szenarien plausibel simulieren zu können, muss die Plattform in der Lage sein Werte absetzen, welche logisch zusammenpassen. Auch wenn einzelne Szenarien gezielt abweichende oder widersprüchliche Werte absetzen können, sollte es die Plattform erlauben, plausible Werte zu erzeugen, um ein repräsentatives Testergebnis zu erhalten.

Anforderung F5 - Filter und Rückkopplungsfilter Um generierte Werte an reale und errechnete Werte anzunähern, soll die Plattform Filter und Rückkopplungsfilter bereitstellen, welche es dem Anwender erlauben Werte vor der Übermittlung zu verzerren. Dadurch soll es möglich sein, plausible Abweichungen von tatsächlichen Soll-Werten zu erreichen. Durch diese Funktionalität können zu testende Plattformen auch auf den Umgang mit inakkuraten Eingangsdaten geprüft werden.

Anforderung F6 - Vorimplementierte Netzwerkschnittstellen Um sofort einsetzbar zu sein, soll die Plattform Netzwerkschnittstellen für die Interaktion mit der im Projekt SitOPT verwendeten Streaming-Engine bieten.

Anforderung F7 - Unit-Test-Architektur Um wiederholbare und nachvollziehbare Tests zu ermöglichen, soll es möglich sein, Unit-Tests anzulegen, und diese auf Anforderung oder automatisiert auszuführen. Diese Tests sollen auf Seite der Plattform über einer Weboberfläche verwaltet werden.

Anforderung F8 - Schnittstellen zur Kommunikation mit der zu testenden Plattform Um das Verhalten der zu testenden Streaming-Engine permanent verfolgen zu können, ist es erforderlich, dass die Plattform Schnittstellen bietet, um mit diesen zu kommunizieren, sodass eine Qualifizierung der Tests möglich wird.

3.2 Nicht-Funktionale Anforderungen

Die nicht-funktionalen Anforderungen an die Plattform bestimmen maßgeblich die konzeptions- und implementierungsrelevanten Entscheidungen.

Anforderung NF1 - Erweiterbarkeit Die zentrale Anforderung an die Erweiterbarkeit der entwickelten Plattform erleichtert Wartung und Entwicklung existierender und zukünftig erforderlicher Funktionalitäten. Die Implementierung der Plattform soll möglichst so gestaltet sein, dass sie im Rahmen der verwendeten Programmiersprache neue Ausprägungen existierender, sowie gänzlich neue Funktionalitäten zulässt.

Anforderung NF2 - Unit-Test-Abdeckung Um die Robustheit und Funktionalitäten der Plattform nachzuweisen, soll durch eine Reihe von Unit-Tests angelegt werden. Diese sollen sowohl Teil-, wie auch Gesamtfunktionen der funktionalen Anforderungen wo es möglich ist überdecken.

4 Konzeptionelle Lösung

Kapitel 4 - Konzeptionelle Lösung erfüllt den Konzeptionsteil der Arbeit, und beschreibt die Konzepte der entwickelten Plattform, deren Umsetzung später in Kapitel 5 - Implementierung weiter ausgeführt wird.

Die Anforderungen NF1 und NF2 an die entwickelte Plattform im Zusammenhang mit dem Forschungsprojekt SitOPT legen die Programmiersprache Java nahe, da diese hier vorrangig eingesetzt wird, und für die Umsetzung geeignet ist. Alle Komponenten der Serverapplikation wurden daher in Java, sowie in Javascript und html entwickelt.

Die Plattform wird in zwei Teile unterteilt. Hauptfunktionalität liefert der Server, welcher Testfälle auf Anfrage, und zu konfigurierbaren Zeitpunkten durchführt. Die Resultate von Testläufen werden in Protokollen festgehalten und im Dateisystem abgelegt. Außerdem werden Ausprägungen von Interfaces geladen und instantiiert, über die sich Funktionen an Plattformen sowie veränderte Anforderungen anpassen lassen.

Die zweite Komponente ist die Weboberfläche, welche als Grafische Schnittstelle zum Server, und damit indirekt zu den Testfällen und Modellen dient. So lassen sich Modelle komfortabel anlegen, anpassen, einsehen und löschen. Außerdem zeigt sie Testprotokolle in übersichtlicher Weise an, und beschleunigt deren Auswertung. Die Weboberfläche verbindet sich mithilfe von Websockets mit dem Server, und muss daher nicht zwingend von der selben Origin stammen.

4.1 Modellbasierte Simulation

Die in Kapitel 3 - Anforderungen dargelegten Anforderungen F1 und F3 fordern dynamisch ausgeprägte und logisch zusammenpassende Werte für die von den Events generierten Daten. Diese lassen sich nur durch *Simulation* sinnvoll erfüllen. Damit ist für die Konzeption implizit eine eindeutige Richtung vorgegeben.

Die für die Plattform definierbaren Modelle entsprechen einem im Abschnitt 4.1 beschriebenen Metamodell, welches Grundlage für die Entsprechung in der Implementierung ist. Die aufgeführten Zusammenhänge zwischen den Modellkomponenten werden in Folgeabschnitten 4.2, 4.4 und 4.6 wieder aufgegriffen, und weiter vertieft.

4.1.1 Modellbeschreibung

Bei den zur Beschreibung des Metamodells verwendeten Begriffen handelt es sich im Sinne der Programmiersprache Java stets um Objekte, beziehungsweise deren Klassen, auch wenn es sich um Begriffe handelt, die im Normalfall im Zusammenhang mit objektorientierter Programmierung bereits belegt sind. Grund für diese Überschneidungen ist, dass die im Metamodell beschriebenen Objekte Abbildungen von Komponenten aus der Programmierung sind. Ausnahme hiervon sind die alleinstehenden Begriffe Objekt und Modell.

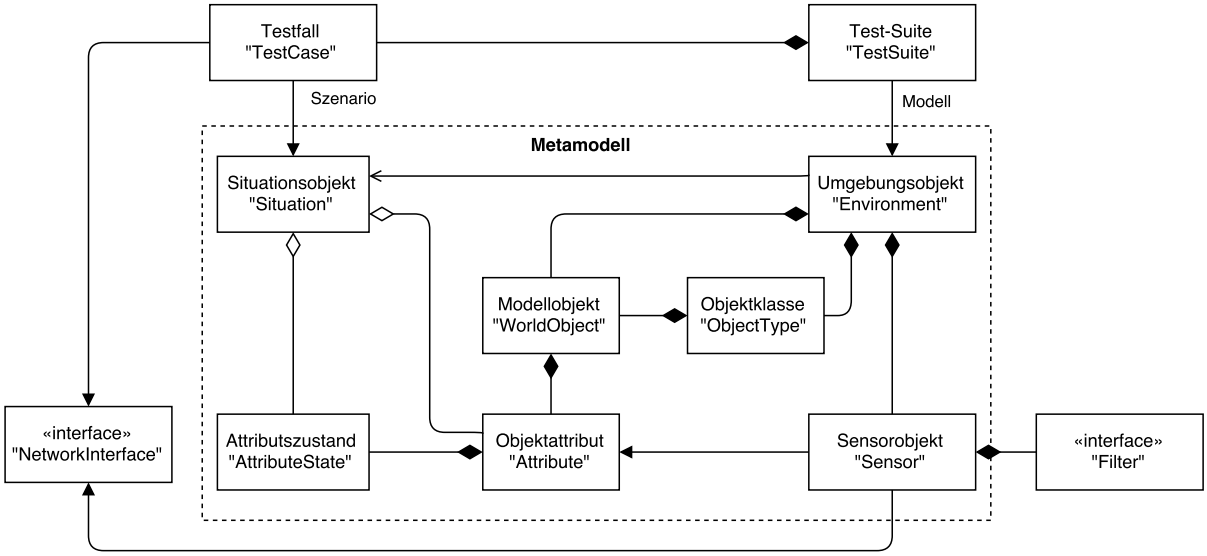

Abbildung 4.1: Veranschaulichung des Metamodells als Klassendiagramm

Herzstück des Metamodells sind die Komponenten Objektattribut und Attributszustand. Ihre Funktionsweise ähnelt denen von Attributen und Werten in Programmiersprachen. Im Unterschied dazu kapseln Objektattribute aber mehrere Attributszustände in der Reihenfolge ihrer Zuweisung. Die Anzahl gespeicherter Zustände kann für jedes Objektattribut in der sogenannten Objektklasse definiert werden. Objektklassen sind, entsprechend Klassen in Objektorientierten Programmiersprachen, selbst wiederum Modelle für Modellobjekte. Modellobjekte entsprechen Instanzen von Objektklassen. Anders als bei Objektorientierten Programmiersprachen existieren aber keine von Objektattributen losgelöste Methoden. Das Verhalten von Modellobjekten ergibt sich aus dem Verhalten der Objektattribute. Hierauf wird in Abschnitt 4.2 weiter eingegangen.

Objektklassen und Modellobjekte sind Umgebungsobjekten zugeordnet, die Modellen entsprechen. Sie sind aus Modellsicht die Wurzel Datenstruktur. Umgebungsobjekte beinhalten die Objektklassen und Modellobjekte. Die Startwerte der Objektattribute sind in Situationsobjekten abgelegt, welche mit jeweils einem Testfall, oder einem Umgebungsobjekt assoziiert sind. Umgebungsobjekte beinhalten das Situationsobjekt, welches denn Normalzustand festlegt. Testfälle beinhalten jeweils das Situationsobjekt, welches das Testszenario beschreibt, und

die Startwerte von Objektattributen überschreibt.. So lassen sich unterschiedliche Szenarien innerhalb des selben Modells definieren.

Modellobjekte können mit Sensorobjekten assoziiert werden. Bei diesen handelt es sich um virtuelle Sensoren, welche Attributszustände aus Objektattributen extrahieren, aus denen wiederum die gekapselten Objekte als Events an eine Filterpipeline, oder direkt an eine Netzwerkschnittstelle weitergereicht werden. Von dort aus werden diese an die Zielplattform abgesetzt. Es ist Aufgabe der Netzwerkschnittstelle Events in eine für die Übermittlung geeignete Form zu bringen. Die Abläufe davon werden in den Abschnitten 4.4 und 4.5 näher erklärt.

In Abbildung 4.1 werden die Zusammenhänge des Metamodells veranschaulicht. Die hier gewählten Bezeichner sind nicht konform mit den Klassennamen der Implementierung.

4.2 Simulationsdurchführung

Um Zustandsänderungen im in Abschnitt 4.1 beschriebenen Modell zu ermöglichen, müssen aktive Komponenten definierbar sein. Dies ist zur Durchführung der Simulation unbedingt erforderlich, da es sich sonst um eine statische Definition handelt, die im Widerspruch zur Anforderung F1 steht.

Um dynamische Änderungen des Modellzustandes zu ermöglichen, bietet die Plattform die Möglichkeit zur Definition von Programmen, über die sich Objektattributen neue Attributszustände zuweisen lassen. Das konzipierte Metamodell ermöglicht hier, für jedes Objektattribut ein eigenes Programm zu beschreiben, welches zur Laufzeit kompiliert und ausgeführt wird. Auf die Kompilation, sowie die Sprache wird in Abschnitt 4.3 näher eingegangen. Liegt für einen Objektattribut ein kompiliertes Programm vor, so kann dieses von einem virtuellen Prozessor ausgeführt werden, und der zurückgegebene Attributszustand dem Objektattribut zugewiesen werden. Der Zustand von Objektattributen ändert sich also potenziell mit der Frequenz, mit welcher ein Prozessor dessen assoziiertes Programm ausführt. Diese ist innerhalb eines Modells einheitlich definiert, und wird im Folgenden mit Iterationsfrequenz und Modellfrequenz bezeichnet. Sie ist von der in Abschnitt 4.5 definierten Schnittstellenfrequenz zu unterscheiden.

4.2.1 Zugriffe auf Objektattribute

Ein kompiliertes Programm hat Zugriff auf vergangene Attributszustände von Objektattributen, nicht aber auf Attributszustände, welche im gleichen Zyklus berechnet werden. Grund hierfür ist, dass die Reihenfolge der Berechnung von Attributszuständen innerhalb eines Zyklus undefiniert ist, und parallel abläuft. Der jüngste verfügbare Attributszustand entspricht somit dem Berechnungsergebnis des jeweils vorangegangenen Zyklus, oder im Fall des ersten Zyklus dem Startwert. Um in den ersten Zyklen keine Zugriffsverletzungen beim Lesen von Zuständen

zu erzeugen, welche weiter als einen Interationsschritt zurückliegen, wird für diese Zugriffe der jeweilige Startwert verwendet, es sei denn, es sind explizite Initialwerte für Altzustände definiert. Attributszustände, die im aktuellen Zyklus berechnet wurden, werden in einen Zwischenspeicher kopiert, bis der Berechnungszyklus abgeschlossen ist. Anschließend werden sie in den Speicher des Objektzustandes geschoben, welcher die alten Attributszustände beinhaltet. Dabei wird der jeweils älteste Wert aus dem Speicher verdrängt. Die Anzahl der für jedes Objektattribut gespeicherten Attributszustände wird für jedes Objektattribut einer Objektklasse spezifisch definiert. Wird kein Wert gewählt, wird ein einzelner Wert gespeichert. Im Berechnungszyklus steht in diesem Fall lediglich der jeweils jüngste Attributszustand zur Verfügung. Um auf einen Attributszustand zuzugreifen wird dem Identifikator des Objektattributes der Index des Attributszustandes mit übergeben. Dieser zählt vom jüngsten Feld aufwärts. Somit entspricht der jüngste Zustand dem Index '0' und der älteste dem Index 'length - 1', wobei 'length' für die Anzahl der gespeicherten Attributszustände steht. Wenn beim Zugriff auf dieses kein Index übergeben wird, wird 0 angenommen. Die Syntax der jeweiligen Übergabe richtet sich nach der Syntax der Sprache, auf welche in Kapitel 5 - Umsetzung und Implementierung näher eingegangen wird.

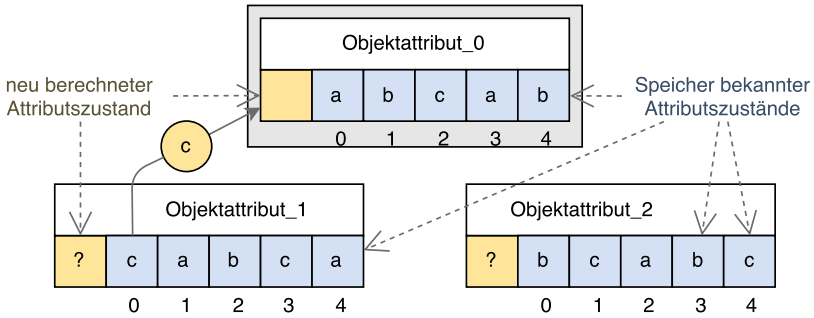

Abbildung 4.2: Beispiel für Zugriff auf Objektattribute

Programme können über den eindeutigen Identifikator von Objekten auch auf Objektzustände zugreifen, die nicht zum gleichen Objekt gehören. Auf diese Weise lassen sich diese untereinander verknüpfen, und Werte zwischen ihnen austauschen. Da Programme den jeweiligen Objektattributen eines Objektes Attributszustände flexibel zuweisen können müssen, sie selbst aber objektklassenspezifisch sind, ist es möglich, Objektreferenzen als Objektattribute zu definieren. Somit ist es möglich, dass verschiedene Objekte der selben Objektklasse über das gleiche Programm auf unterschiedliche Attributszustände zugreifen. In Abbildung 4.2 wird dargestellt, wie drei Objekte der selben Objektklasse mithilfe unterschiedlicher Objektreferenzen Attributszustände zirkulieren. Objektreferenzen sind ebenfalls Attributszustände. Objektattribute welche diese speichern können folglich ebenfalls dynamisch verändert werden, was es möglich macht Programme in verschiedenen Berechnungszyklen auf die Attributszustände jeweils unterschiedlicher Objekte zugreifen zu lassen. Damit sind hochflexible Modelle mit dynamischen Relationsdiagrammen möglich.

4.2.2 Ausführbare Programme

Ein kompiliertes Programm im oben beschriebenen Sinne ist im Speicher als Baumstruktur abgelegt. Sie werden im Folgenden ausführbare Programme genannt. Dabei repräsentiert jeder Knoten zusammen mit seinen Kindknoten ein kompiliertes Quelltextsegment. Die Unterteilung der Segmente ist so gewählt, dass sie eine möglichst kleine Berechnung auf der Ebene einer Hochsprache, oder wichtige semantische Gruppierungen widerspiegeln. Beispiele hierfür sind For-Schleifen, Ausdrücke, Blöcke und Methodenaufrufe. Die Abhängigkeiten und Beziehungen der Berechnungen sind dabei durch die Baumstruktur repräsentiert. Ein Methodenaufruf kann beispielsweise alleine als Ausdruck in einem Block stehen, oder an einer Stelle, an der ein Rückgabewert erwartet wird.

Die Baumstruktur wird vom Prozessor in einer Reihenfolge durchlaufen, welche der der Tiefensuche ähnelt. Es gibt aber Ausnahmen in denen sie davon abweicht. Die Repräsentationen von Kontrollstrukturen werden unter Umständen nicht immer vollständig durchlaufen, sondern nur dann, wenn diese den jeweiligen Durchlauf der entsprechenden Kindknoten in Abhängigkeit von Variablen oder Attributszuständen vorsehen. Dies ist im Normalfall vom Modellzustand abhängig. Außerdem können bestimmte Knoten dynamisch viele Kindknoten enthalten. Anweisungsblöcke können beispielsweise beliebig viele Kindknoten sequenziell ausführen. Abbildung 4.3 stellt ein Beispielprogramm dar, welches einen Knoten enthält, der eine If-Anweisung abbildet. Seine Kindknoten repräsentieren die Teilschritte der Ausführung. Klar erkennbar ist, wie der Programmblock des else-Blocks, und damit auch seine Kindknoten nicht ausgeführt werden. Das Verhalten von Schleifen ist äquivalent, wobei hier Kindknoten potentiell mehrfach ausgeführt werden. Die Wurzel der Baumstruktur bildet immer ein Anweisungsblock. Es wird erwartet, dass dieser in jedem Durchlauf mindestens eine Wertrückgabe ausführt.

Eine Berechnung ist weiter in - aus Sicht der Programmbeschreibung - elementare Berechnungseinheiten unterteilt. Diese können nicht mehr durch andere Berechnungen oder Berechnungseinheiten unterbrochen werden. Sie repräsentieren die größtmöglichen isolierten Operationen die möglich sind, ohne dass währenddessen weitere Berechnungseinheiten notwendig werden. Eine Berechnungseinheit kann zum Beispiel die Initiierung der Durchführung eines Kindknotens sein. Für ihre Berechnung sind unter Umständen Informationen aus der Modellumgebung, oder vorangegangenen Berechnungseinheiten erforderlich. Diese werden entweder direkt aus dem sie umgebenden Modell bezogen, oder liegen auf dem durch die Prozessorinstanz verwalteten Stapel.

Zur Ausführung von Methodenaufrufen wird die Reflections-Bibliothek von Java verwendet. Hierzu mehr in Kapitel 5 - Umsetzung und Implementierung.

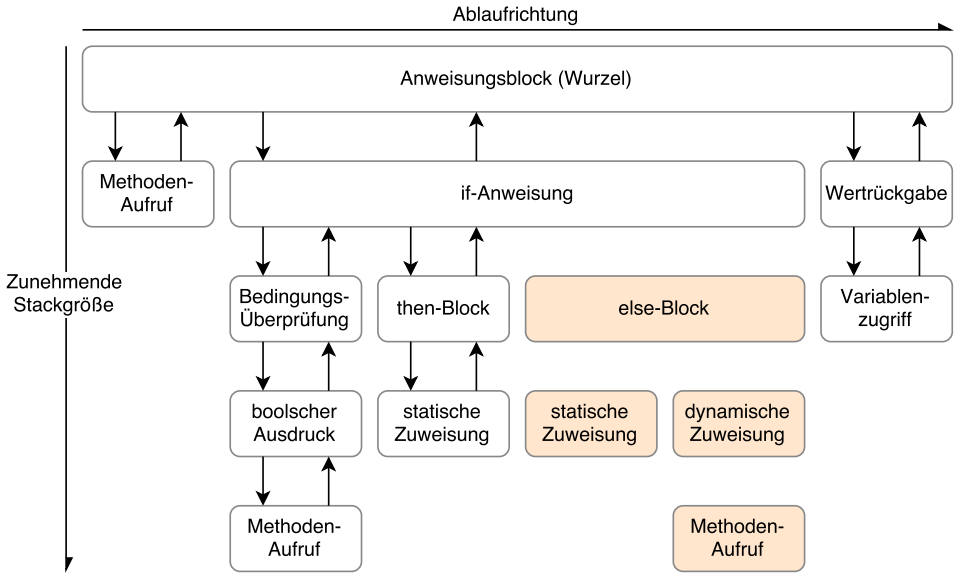

Abbildung 4.3: Ablauf der Ausführung eines Beispielprogramms

4.2.3 Virtuelle Prozessoren

Innerhalb eines Berechnungszyklus wird eine für die Plattform konfigurierbare Zahl an Instanzen virtueller Prozessoren verwendet. Diese arbeiten zu berechnende Programme eines Modells nach einem Ticket-Schema ab. Dabei handelt es sich um ein Schema der Abarbeitung von Arbeitsaufträgen, das wie folgt umgesetzt wurde. Beendet eine Prozessorinstanz eine Berechnung, wird ein neues Ticket gelöst, und eine weitere Berechnung begonnen. Für jeden Testfall werden neue Prozessorinstanzen erzeugt, um Übertragungsfehler von vorangegangenen Testfällen auszuschließen.

Eine Instanz eines virtuellen Prozessors organisiert zur Ausführung von Programmknoten relevante Informationen überwiegend in einem Stapelobjekt. Hier werden unter anderem die Fortschritte einzelner Berechnungen zwischengespeichert, die erforderlich sind, um nach Ausführung von Kindknoten die Berechnung mit der richtigen Berechnungseinheit fortzusetzen. Für jeden Programmknoten wird eine neue Stapelebene angelegt. In dieser wird unter anderem ein Zeigerobjekt (Cursor) für den assoziierten Knoten abgelegt. Dieses wird verwendet, um Berechnungseinheiten zu verfolgen. Damit verhält sich der Stapel ähnlich denen von Assemblersprachen für Stapelmaschinen??. Bei der Abarbeitung von Programmknoten führt ein Prozessor immer den obersten auf dem Stapel liegende aus. Er bearbeitet dabei die jeweilige Berechnungseinheit, auf die das dort ebenfalls abgelegte Zeigerobjekt referenziert. Dieser isolierbare, elementare Teilschritt wird in der Implementierung als 'Step' bezeichnet.

Auf dem Stapel werden auch Sichtbarkeitsbereiche verwaltet. Diese werden von bestimmten Berechnungen erzeugt und speichern Variablen für den Zugriff durch sie selbst sowie ihren Kindknoten. Eine deklarierte Variable wird stets auf der höchstmöglichen Stapel-Ebene ab-

gelegt, und mit dieser gelöscht, wenn die Berechnung abgeschlossen ist. Beim Zugriff auf Variablen wird der Stapel ebenenweise von oben herunter nach dem Identifikator durchsucht. Dies übersteigt prinzipiell die Möglichkeiten herkömmlicher Stapel, wird hier aber durch eine Rückwärtsverkettung ermöglicht.

4.3 Programm-Kompilation

Um der Anforderung NF1 zu genügen, soll die Plattform die Möglichkeit bieten, möglichst viele Hochsprachen als Definitionssprachen von Objektattributsverhalten zu erlauben. Vereinfachte Programmiersprachen können über Sprachdefinitionsdateien in die Plattform eingebettet werden. Verschiedene Sprachen weisen syntaktische und funktionale Merkmale auf, die in Java nur schwer emulierbar sind. Aus diesem Grund können nicht alle Aspekte aller Sprachen abgebildet werden. Von zentraler Wichtigkeit ist vor allem die Möglichkeit prozedurale Abläufe definieren zu können. In Kapitel 5 - 5 wird näher auf die exemplarisch eingebettete reduzierte Abwandlung von Java, sowie auf die für die Sprachdefinition notwendige Syntax eingegangen.

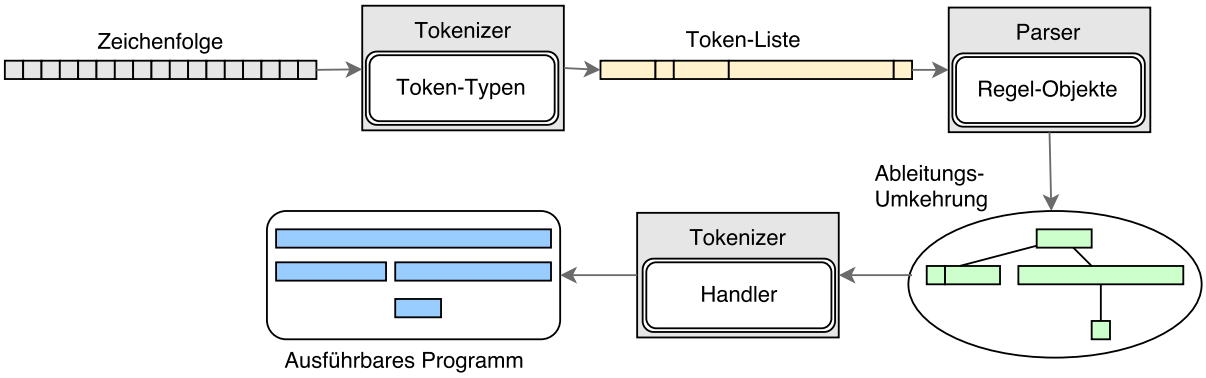

Abbildung 4.4: veranschaulichung der Kompilationsabfolge

4.3.1 Kompilationskette

Um Programme zu kompilieren werden diese nacheinander drei Komponenten zugeführt, was in Abbildung 4.4 veranschaulicht wird. Dabei wird der jeweils weitergereichten Datenstruktur die in jedem Schritt die gewonnene Mehrinformation mitgegeben. Dabei sind die jeweiligen Eingabeelemente in der neuen Datenstruktur vollständig enthalten. Eine Ausnahme bildet das am Ende ausgegebene ausführbare Programm.

Ein Programm liegt ursprünglich als Quelltext vor und wird als solcher eingelesen. Er wird zunächst der Tokenizer-Komponente zugeführt. Diese identifiziert atomare Quelltextelemente, wie beispielsweise Schlüsselwörter, Sonderzeichen, Identifikatoren und Literale, verpackt

diese in kapselnde Token-Objekte, und gibt sie als Liste zurück. Token-Objekte kennen außerdem ihren Token-Typ. Dieser entspricht einem Terminal in der Grammatik der definierten Sprache.

Anschließend wird die Token-Liste an ein Parser-Komponente übergeben, welches diese mithilfe von Regel-Objekten auf das Startsymbol zurückzuführen. Regel-Objekte sind dabei die Entsprechung der Produktionsregeln der Grammatik. Bei diesem Prozess handelt es sich um die Umkehrung der Ableitung. Die Repräsentation dieser Rückführung ist eine Baumstruktur bestehend aus Ableitungsknoten. Ableitungsknoten sind aus angewandten Regel-Objekten, der konkret angewendeten Variante der Auflösung, genannt Alternative, des Nichtterminals welches die Regel ableitet, sowie der Repräsentation der weiteren Ableitung durch Tokens und Ableitungsknoten. Die Blätter dieser Baumstruktur bilden immer Tokens. Ist dies nicht der Fall, so konnte die Token-Liste nicht erfolgreich eingelesen werden, beziehungsweise lag nicht in der durch die Sprachdefinition definierten Sprache. Um die Definition von Sprachen zu erleichtern, werden außerdem Gruppierungen innerhalb von Produktionsregeln zugelassen. Eine mögliche Gruppierungsform ist die einfache Gruppe, mit deren Hilfe sich Token-Typen einer Alternative innerhalb einer Produktionsregel genauer zuordnen Lassen. Eine zweite Form ist die optionale Gruppe. Sie verhält sich wie eine einfache Gruppe, kann in der Ableitung aber auch kein mal vorkommen. Die letzte mögliche Gruppierungsform ist die Wiederholbare Gruppe. Sie kann beliebig oft in der Ableitung vorkommen, also auch ein- oder kein mal.

Die resultierende Baumstruktur wird anschließend dem Compiler-Komponente übergeben. Dieses löst sie rekursiv auf, und bedient sich dabei Handler-Objekten, welche jeweils für eine Teilmenge von Nichtterminalen zuständig sind. Sie geben jeweils eine Baumstruktur aus Ausführbaren Programmknoten zurück. Handler-Objekte können Kindknoten der ihnen übergebenen Baumstruktur an andere Handler-Objekte delegieren. Auf diese Weise entsteht eine neue Baumstruktur aus Ausführbaren Programmknoten.

4.3.2 Sprachdefinition

Eine vollständige Sprachdefinitionsdatei wird zur Laufzeit der Plattform eingelesen. Sie liegt als XML-Datei vor, und wird von einer eigens dafür verantwortlichen Parser-Komponente interpretiert. Sie enthält alle Informationen über die Syntax der Sprache, die für den Einsatz in der Plattform relevant sind. Dies beschränkt sich meist auf prozedurale Aufrufe. Außerdem können Methoden, die in der definierten Sprache üblich sind, und häufig eingesetzt werden, auf äquivalente Java-eigene Methoden gemappt werden. Dies ist aber nur dann sinnvoll, wenn die Signaturen der Methoden vergleichbare Datentypen entgegennehmen. Mapperklassen erweitern ein bereitgestelltes Interface, und können im Sinne der Erweiterbarkeit beliebig ausgeprägt werden. Sie können dann beispielsweise nicht-eigene Java-Funktionen auf bereitgestellte Utility-Klassen umleiten. Dadurch lässt sich der Funktionsumfang der Programme dem einer Bandbreite von Sprachen nachahmen.

4.4 Filterkonzept

Um die durch die in Abschnitt 4.1 beschriebenen virtuellen Sensoren ausgelesenen Daten vor der Übermittlung manipulieren zu können, stellt die Plattform Filter bereit. Diese werden als class-Dateien im Filterverzeichnis abgelegt und implementieren ein generisches Java-Interface. Sie können dann in der Weboberfläche an virtuelle Sensoren angehängt werden, und werden in entsprechender Reihenfolge sequenziell durchlaufen. Durch diese Auslegung lassen sich einmal definierte Filter Test-Suite-übergreifend wiederverwenden, und sind überaus flexibel. Sie können beispielsweise eingesetzt werden, um Werte zu verzerren, um Abweichungen in Messwerten zu imitieren. Dies erlaubt es die getesteten Operatoren auf Toleranz gegenüber Ausreißenden Werten zu testen.

Eine Verkettung von Filtern wird mit dem Begriff Filter-Pipeline bezeichnet, auch wenn es sich nicht im strikten Sinne um eine Pipeline handelt, da die gefilterten Objekte nicht direkt weitergereicht werden Stattdessen werden sie als Rückgabewert an die aufrufende Methode zurückgegeben, welche anschließend die entsprechende Methode des nächsten Filters aufruft. Ein verwendeter Filter muss das vorgegebene Interface so implementieren, dass der vom Sensor oder vorangegangenen Filtern übergebenen Datentyp dem generischen Datentyp entspricht. Es ist auch möglich, dass ein generischer Datentyp eine Superklasse oder Superinterface des übergebenen Datentyps ist. Dadurch kann der Datentyp des jeweils übergebenen Objektes entlang der Filter-Pipeline zunehmend degeneriert werden. Ein Filter muss stets ein neues Objekt instanziieren und zurückgeben. Dies ist erforderlich, damit Filter, die Werte zwischenspeichern, auf diesen operieren können, ohne andere Filter zu beeinflussen. Die Verletzung dieser Eigenschaft kann nur zur Laufzeit geprüft werden, und resultiert bei Verletzung in einem Abbruch des Durchlaufs. Am Ende der Pipeline muss der Objekttyp der verwendeten Schnittstelle übergeben werden können, also einem von ihr erwarteten Objekttyp entsprechen. Der sequenzielle Durchlauf der Filterpipeline wird nicht seitens des Modells, sondern seitens der übermittelnden Schnittstelle initiiert. Dies bedingt sich durch die in Abschnitt 4.5 erläuterte Loslösung der Iterationsfrequenz der Schnittstelle von der des Modells.

Bei der Implementierung von Filtern hat der Entwickler in der Regel keine Information darüber, in welchen Zeitabständen die Filtermethode aufgerufen wird. Dies wird bei zyklischer Datengegerierung erst zur Laufzeit über eine im Interface definierten Methode bekannt. Handelt es sich bei der absetzenden Schnittstelle um eine Pull-Schnittstelle, so gibt es keine Information über die Iterationsfrequenz, es sei denn, der Entwickler berechnet diese zur Laufzeit selbst aus der Systemzeit.

Es ist mit diesem Konzept möglich Filter zu definieren, welche durchgereichte Objekte zwischenspeichern, um später ausgegebene Objekte davon abhängig auszuprägen. Ist dies der Fall, so spricht man von einem Rückkopplungsfilter. Rückkopplungsfilter können beispielsweise eingesetzt werden, um Werte zu glätten oder verzögert auszugeben. In jedem Fall muss aber dennoch bei jedem Aufruf ein valides Objekt zurückgegeben werden, auch wenn noch nicht die erwartete Objektzahl vorhanden ist. Das resultierende Objekt wird durch den Sensor an den nächsten Filter, oder gegebenenfalls an die Netzwerkschnittstelle weitergegeben.

Nach jedem Testfall wird eine clear-Methode des Interfaces aufgerufen. Der Entwickler ist selbst dafür verantwortlich, ob er dies nutzt, um Altwerte zu bereinigen. Wird darauf verzichtet, wird ein Folgetestfall unter Umständen mit fehlerhaften Werten im Speicher begonnen.

4.5 Netzwerkschnittstellen

Um Ereignisse an eine Zielplattform übertragen und von ihr empfangen zu können, sind Netzwerkschnittstellen erforderlich. Sie werden als Klassen implementiert und an einer konfigurierbaren URL gesucht. Diese implementieren ein vorgegebenes generisches Interface, und werden an einer konfigurierbaren Stelle erwartet. Es werden grundsätzlich zwei Arten von Schnittstellen unterschieden, „push“ und „pull“. Push-Schnittstellen setzen in definierten Zeitabständen Events ab, wohingegen Pull-Schnittstellen von sich aus keine Events absetzen, sondern auf Anfragen des Empfängers warten. Dies knüpft an die in Abschnitt 4.1 beschriebenen Paradigmen der zyklischen und synchronen Datengenerierung an. Schnittstellenimplementationen entscheiden selbst, welchem Schema sie dabei entsprechen, und können gegebenenfalls auch beides anbieten.

Gibt das Schnittstellenobjekt nach der Instantiierung in einer dafür zuständigen Methode einen Wert größer ’0’ zurück, wird für dieses seitens der Plattform der Aufruf der Objektabfrage zyklisch initiiert. Andernfalls wird diese ausschließlich durch Anfragen des Empfängers ausgelöst. Es sind Schnittstellen möglich, für die beides zutrifft, obwohl dies lediglich der Erweiterbarkeit dient.

Die Plattform teilt jedem Schnittstellenobjekt einen sogenannten Adapter zu, der Aufrufe an die in Abschnitt 4.4 beschriebene Filter-Pipeline in Abhängigkeit des Schnittstellentyps weiterleitet, oder initiiert.

4.6 Unit-Tests

Um Operatoren in CEP-Engines zuverlässig und wiederholbar zu testen, soll es die Plattform ermöglichen automatische Testabläufe zu definieren, welche in Anlehnung an Software-Modultests als Unit-Tests oder Testfälle bezeichnet werden. Diese sollen in voneinander unabhängigen Abläufen qualifizierende Aussagen über die Korrektheit der zu testenden Operatoren liefern. Im Unterschied zu Software-Testfällen laufen diese Testfälle aber nicht prozedural ab, sondern übergeben dem Modell vorher definierte Randbedingungen, welche das jeweilige Testszenario ausprägen. Bestimmte Startwerte in konkreten ObjectStates erlauben dabei individuelles Verhalten, bei gleichem zugrundeliegendem Modell. Somit muss für eine Umgebung jeweils nur ein Modell angelegt werden, um sämtliche Operatoren prüfen zu können.

Für jeden Testfall wird mindestens ein komplexes Antwortereignis erwartet. Die Schnittstelle hierfür ist in Abschnitt 4.5 beschrieben. Testfälle erlauben außerdem die Spezifikation einer

absoluten Laufzeit für einen Test. Hierdurch wird es ermöglicht zeitabhängige Operatoren zu prüfen, oder eine größere Zahl an Zeiteinheiten im Modell zu simulieren. Dies beeinflusst in aller Regel auch die Menge an Informationen, welche den getesteten Operatoren übermittelt werden, bevor ein Testfall ausgewertet wird.

Testfälle werden in Test-Suites zusammengefasst, welchen jeweils ein Modell zugeordnet ist. Eine Test-Suite liefert ein gesamtheitliches Ergebnis über alle enthaltenen Testfälle. Diese Ergebnisse werden in Protokollen festgehalten, welche an vorher konfiguriertem Ort und Namen im Dateisystem abgelegt werden. Schlägt ein Testfall fehl, so werden die übrigen Testfälle dennoch ausgeführt. In diesem Fall wird die Ausführung der Test-Suite insgesamt als fehlgeschlagen qualifiziert. Es ist dennoch möglich, dass andere Testfälle als erfolgreich qualifiziert worden sind. Testfälle sind aus der Weboberfläche heraus auch individuell ausführbar. Dies erlaubt das schnelle Testen einzelner oder verketteter Operatoren, ohne die Gesamtlaufzeit der kompletten Test-Suite abwarten zu müssen.

5 Umsetzung und Implementierung

Die Implementierung der in Kapitel 4 - Konzeptionelle Lösung beschrieben Konzepte wird mit Java 8, sowie HTML und Javascript umgesetzt. Dafür wurden drei Projekte erstellt, deren Umsetzung in den folgenden Abschnitten näher beschrieben wird. Als übergeordnete Bezeichnung für die in dieser Arbeit entstandene Plattform wurde 'SimPl' gewählt, was Simulations-Plattform abkürzt, und sich an die Bezeichnung 'SitOPT' angelehnt ist.

Das erste Projekt 'SimPlScript' kapselt dabei die Klassen und Funktionalitäten der Prozessoren, Programmstruktur und Programmübersetzung. Es ist als API ausgelegt und bietet entsprechende Schnittstellen. Es ist für sich funktional und hat keine Abhängigkeiten zu den beiden anderen Projekten, ist aber alleine nicht ausführbar. Das Projekt 'SimPlServer' implementiert die in Kapitel 4 beschriebenen Funktionalitäten der Serveranwendung, und führt sie mit denen des ersten Projektes zusammen. Die Anbindung der Weboberfläche ist über Websockets[Ian11] ermöglicht. Hauptaufgabe ist die Verwaltung und Organisation der Objekte des Modells, sowie deren, im Dateisystem abgelegte Repräsentationen. Außerdem bindet sie dynamisch die Ausprägungen der Interfaces ein, die der Erweiterung von Funktionalitäten, oder zur Bereitstellung der Kompatibilität zu Eventprotokollen dienen. Es besteht die Möglichkeit abstrakte Klassen zu implementieren, um die Strukturellen Möglichkeiten der ausführbaren Programme zu erweitern, und so neue Sprachkonstrukte zu ermöglichen. Diese können aber nicht dynamisch geladen werden, sondern erfordern einen Neustart des Servers. In 'SimPlWeb' ist die Weboberfläche als Webseite in HTML und Javascript umgesetzt. Sie ermöglicht Zugriff auf die verwaltenden Operationen der Serveranwendung, und zeigt Testprotokolle in überschaubarer Form an.

Zu den mit Java umgesetzten Projekten sind jeweils JUnit-Test-Cases implementiert, welche das Verhalten der implementierten Klassen validieren. Dies wird von NF2 verlangt. Um aber nicht in jeder Methode Ausnahmen behandeln zu müssen, werden diese von 'RuntimeException' abgeleitet, und müssen so nicht explizit bei der Deklaration einer Methode angegeben werden. Außerdem werden bei Fehlbedienungen entsprechende Ausnahmezustände geworfen, die im Folgenden nicht explizit beschrieben sind.

5.1 Implementierung der Programm-API

Die Klassen und Interfaces der Programm-API sind auf vier übergeordnete Pakete aufgeteilt. Im Paket 'processor' sind die operativen Klassen zur Ausführung von Programmen zusam-

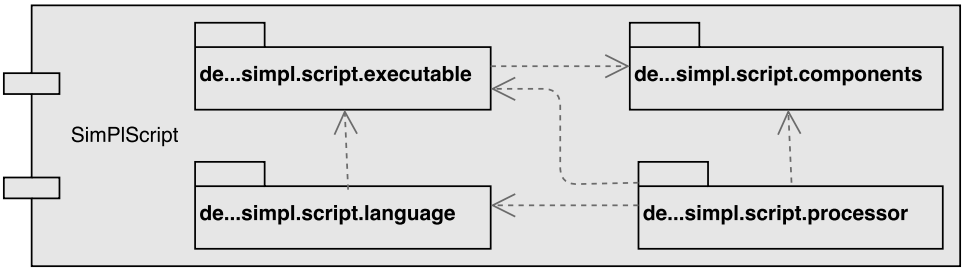

Abbildung 5.1: Veranschaulichung der Zusammenhänge und Abhängigkeiten der Pakete in der API-Komponente

mengefasst. Ausführbare Programme werden hingegen in einer Datenstruktur abgebildet, die im Paket 'executable' implementiert ist. In 'language' sind die für die Sprachdefinition und Übersetzung relevanten Klassen organisiert. Schnittstellenklassen und -Interfaces, auf denen Objekte basieren, deren Zustände und Beziehungen durch die definierten Programme manipulierbar sind, sind im Paket 'components' abgelegt. Die Pakete 'language' und 'processor' sind funktional weitestmöglich voneinander getrennt. Beide nutzen aber die in 'executable' definierten und implementierten Interfaces und Klassen.

5.1.1 Ausführbare Programme

Ein Ausführbares Programm wird im Speicher durch eine Baumartige Objektstruktur repräsentiert, deren Wurzel eine Instanz der Klasse 'ExecutapleProgram' ist. Diese ist weiter in der Klasse 'Program' gekapselt, welche Metainformationen, wie den Quelltext, die kompilierte Datei, das verwendete 'Language'-Objekt, und den verwendeten Mapper enthält.

Listing 5.1 Auszug der abstrakten Klasse ExecutableProgramItem

```
/** .. */
public abstract StackOperation executeState(Processor processor,
   int executedState);

/** .. */
public abstract int getNextState(Processor processor, int previousState,
   boolean conditionState);
```

Klassen von Programmelementen erweitern die abstrakte Klasse 'ExecutalbleProgramItem', welche das grundlegende Verhalten von Knoten innerhalb der Baumstruktur vereinheitlicht. Listing 5.1 zeigt die beiden zentralen Methoden, die eine Programmeinheit ausmachen. An der Signatur der Methode 'getNextState' wird deutlich, dass ein Programmelement den Zustand des Stapels, und damit seinen eigenen Ausführungsstand, nicht kennt. Es kann nur implizit unter Zuhilfenahme der Parameter 'previousState' und 'conditionState' den nächsten auszuführenden Programmschritt bestimmen. 'previousState' ist der Index der zuletzt für dieses

Programmelement durchgeführten Berechnungseinheit. Wird als Ergebnis des Aufrufes '-1' zurückgegeben, signalisiert dies dem Prozessor, dass die Berechnung des Programmelementes beendet ist, und die für dieses angelegte Stapelebene gelöscht werden kann. In diesem Fall wird iterativ weitergeprüft, ob auch das jeweils nächste Programmelement beendet ist. Dies ist notwendig, damit der Aufruf 'executeState' wirklich immer nur einen Programmschritt ausführt. Der Parameter 'conditionState' wird verwendet, um die Resultate zuvor ausgeführter Bedingungen in Kontrollstrukturen weiterzureichen, und ist in den meisten Fällen 'null'.

Die Methode 'executeState' ist für das executive Verhalten eines Programmelementes zuständig. Je nach übergebenem Wert des Parameters 'executedState', führt kann das Objekt eine andere Berechnungseinheit durchführen. Eine Berechnungseinheit kann beispielsweise sein, den Prozessor anzuweisen, eine neue Stapelebene für einen Kindknoten zu erzeugen. Ist dies der Fall, wird 'StackOperation.PUSHED' zurückgegeben, was dem Prozessor signalisiert, mit der Berechnung des neuen Elementes zu beginnen.

Manche Erweiterungen der Klasse 'ExecutableProgramState' müssen in der Lage sein, Methoden von Objekten und Klassen aufzurufen, die sie erst zur Laufzeit kennen. Beispiel hierfür ist die Klasse 'ExecutableInvokation', welche Methodenaufrufe im Programm repräsentiert. Hier muss die Objektorientierung zugunsten von Funktionalität zurückgestellt, und die Java-eigene Reflections-API genutzt werden. Diese erlaubt es auf Attribute und Methoden von Objekten und Klassen zuzugreifen, die erst zur Laufzeit bekannt sind. Auf diese Weise sind Methodenaufrufe in Programmen möglich. Da zur Kompilierzeit eines Programmes nicht immer bekannt ist, ob eine gesuchte Objekteigenschaft existiert, oder ob übergebene Parameter den erwarteten Datentypen entsprechen, sind hier Laufzeitausnahmen möglich. Diese beenden den Prozessoraufruf. Im Fall eines Testlaufes in der entwickelten Plattform führt dies zum Abbruch des Testfalles, welcher mit fehlgeschlagen bewertet wird. Eine detaillierte Fehlerausgabe wird in der Log-Datei festgehalten.

5.1.2 Prozessor

Programmobjekte werden Instanzen der Prozessorklasse 'Processor' in einer Aufrufmethode übergeben, welche die Ausführbare Objektstruktur ausführt. Als weitere Parameter nimmt sie ein Umgebungsobjekt und ein Bezugsobjekt entgegen.

Umgebungsobjekte, deren Klassen das Interface 'ProgramAccessibleEnvironment' implementieren, referenzierten die durch das Programm einlesbaren und modifizierbaren Objekte. Diese sind Instanzen von Klassen, welche das in Listing 5.2 aufgeführte Interface implementieren. Im Sinne der Schnittstelle hat der Prozessor auf sie nur lesenden Zugriff. Es kann aber nicht verhindert werden, dass ein Methodenaufruf eines so verwendeten Objektes seinen Zustand, oder den anderer Objekte verändert. Von dieser Praxis wird abgeraten. Das Bezugsobjekt kann hingegen manipuliert werden, und wird verwendet, um Bezugslose Aufrufe auszuführen. Es verhält sich also ähnlich dem, in Java durch das Schlüsselwort 'this' referenzierten Objekt. Der Prozessor geht dabei davon aus, dass ihm das zu beschreibende Objekt alleine zur Verfügung

Listing 5.2 Auszüge der Schnittstellenklasse ProgramAccessibleComponent

```
/** ... */
public interface ProgramAccessibleComponent
{
  /** ... */
  public void insertAccessibleValue(String identifier, Object value);

  /** ... */
  public Object getAccessibleValue(String identifier);

  /** ... */
  public void reserve(Processor reservingProcessor);

  /** ... */
  public void release();
}
```

steht. Um dies zu garantieren, reserviert er ein Objekt, indem er vor dem eigentlichen Zugriff eine in der entsprechenden Schnittstelle definierte Methode aufruft, welche der Entwickler nutzen kann, um Zugriffe durch andere Threads zeitweise zurückzustellen. Dabei wird der Prozessor als Argument mitgegeben, sodass bei einem weiteren Aufruf der Methode durch den selben Thread nachvollzogen werden kann, dass der Zugriff erlaubt ist. Nachdem ein Zugriff beendet wird, explizit nach der Entfernung der Referenz aus dem Stapel, wird eine weitere Methode aufgerufen, welche die Freigabe des Objektes signalisiert. Für jedes Zugriffs-Signal muss ein entsprechendes Freigabe-Signal erteilt werden, bevor das Objekt von einem anderen Thread reserviert werden kann. Wird die API außerhalb dieser Implementierung wiederverwendet, ohne dass parallele Zugriffe bedacht werden müssen, können diese Methoden leer implementiert werden.

Stapel

Der Prozessor verwaltet zur Laufzeit von ausführbaren Programmen ein Stapelobjekt. Dieses ist verwaltet Stapelebenen in Instanzen der 'StackLevel'-Klasse. Eine Stapelebene speichert Informationen, die der Prozessor benötigt, um das ausführbare Programmelement, für welches es angelegt wurde, und dessen Unterprogrammelemente (Kindknoten im Graphen), korrekt auszuführen.

Dieses wird in Listing 5.3 als 'executedItem' aufgeführt. Das in 'definedScope' referenzierte Scope-Objekt speichert die Variablen, die auf dieser Stapelebene deklariert wurden, sowie die aller Unterprogrammknoten, für deren Ausführung kein eigenes Scope-Objekt angelegt wurde. Scope-Objekte werden nur dann angelegt, wenn der Typ des ausgeführten Programmelements dies verlangt. Dies ist bei Block-Elementen der Fall. Innerhalb eines Blocks deklarierte

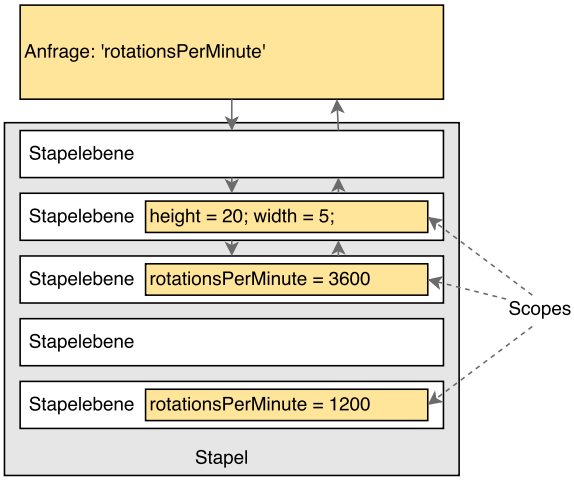

Abbildung 5.2: Veranschaulichung der Durchreichung von Anfragen

Listing 5.3 Auszug der Klasse StackLevel

```
/** ... */
public class StackLevel
{
  /**
   * Collection object for all variables declared between this
   * and the next upper ScopeLevel with a definedScope object != null.
   */
  private final Scope definedScope;

  /** .. */
  private final Cursor executionPosition;

  /** .. */
  private final ExecutableProgramItem executedItem;

  /** .. */
  private Object result = null;
```

Variablen werden erst gelöscht, wenn die Ausführung des Blocks abgeschlossen ist. Zugriffe auf Variablen werden von oben herab durch die Stabelebenen gereicht, bis der gesuchte Identifikator gefunden wurde. Somit ist es möglich, Variablen lokal zu überdecken, bis die Ausführung des überschreibenden Programmblocks abgeschlossen ist. Dies wird in Abbildung 5.2 veranschaulicht. Stapelebenen werden nach Beenden des jeweiligen Programmobjekts für das sie erzeugt wurden vom Stapel gelöscht.

Programmelemente teilen dem Prozessor nach der Durchführung des jeweiligen Aufrufes mit, wie anschließend mit dem Stack verfahren wird. Die definierte Enumeration 'StackOperation'

Listing 5.4 Auszug aus der Methode execute der Klasse Processor

```
public void execute()
{
  try
  {
    // while the stack is not empty, execute next item.
    while (!this.stack.isEmpty())
    {
      this.executeStep();
    }
  }
  catch (ProcessorRuntimeException e)
  {
    //[...]
  }
}
```

kennt die Zustände 'Pushed' und 'Advanced', welche implizieren, ob es sich bei der letzten durchgeführten Berechnungseinheit eine neue Stapelebene auf den Stapel gelegt oder ein interner Berechnungsschritt durchgeführt wurde. Ersteres veranlasst das Prozessorobjekt auf die neue oberste Stapelebene zu wechseln und diese abzuarbeiten. Der in Listing 5.5 aufgeführte Quelltextauszug zeigt die Methode 'executeStep' des Prozessors. Diese wird im normalen Anwendungsfall von der 'execute'-Methode des Prozessors iteriert, bis der Stapel leer ist. Dies ist in Listing 5.4 dargestellt. Ein leeres Stapelobjekt impliziert das Programmende. Zu Testzwecken oder zum Debugging können auch einzelne Prozessorschritte ausgeführt werden.

5.1.3 Sprache

Die Übersetzung von Programmsegmenten wird in der API durch das 'language'-Paket abgedeckt. Dieses ist, abgesehen von der Kernklasse 'Language', welche die zur Übersetzung verwendeten Komponenten und Informationen speichert, in zwei Unterpakete unterteilt. Die Definition von vereinfachten Programmier- und Skriptsprachen wird durch das Paket 'definition' abgedeckt. In ihm sind Funktionalitäten zum Einlesen von Sprachdefinitionsdateien, Komponenten für die Modellierung der Syntax, und sowohl die Schnittstelle, wie auch die Standardausprägung eines Mappers organisiert. Zentral für die Implementierung ist dabei die Umsetzung der Sprachdefinition, sowie der Modellierung von Sprachen aus Klassensicht.

Listing 5.5 Methode executeStep der Klasse Processor

```
public void executeStep()
{
  // get top stack level.
  StackLevel top = this.stack.peek();
  // execute top level item.
  StackOperation prevStackOp = top.getExecutedItem().executeState(this,
     top.getCursorPosition().getProcessingState(),
     top.getCursorPosition().getConditionState());
  // if the stack was increased, execute new top item.
  if (prevStackOp != StackOperation.PUSHED)
  {
    // otherwise advance top cursor.
    int nextState;
    do
    {
      // get state to advance to.
      nextState = top.getExecutedItem().getNextState(this,
         top.getCursorPosition().getProcessingState());
      // if the item was completed, pop the top item.
      if (nextState == -1)
      {
        this.tempResult = this.stack.finalizeItem();
        if (this.stack.isEmpty())
        {
          // end the execution, if the stack is empty.
          break;
        }
        else
        {
          // get next top level.
          top = this.stack.peek();
        }
      }
      else
      {
        // advance to the next execution state.
        top.advanceCursorPosition(nextState);
      }
    }
    while (nextState == -1);
  }
}
```

Sprachdefinition und Übersetzung

Sprachdefinitionsdateien, aus denen SimPl Language-Objekte erzeugt, mit denen Quelltext übersetzt werden kann, sind in XML formuliert. Dabei sind die definierten Eigenschaften zwei Elementknoten zugeordnet, 'tokenization' und 'parsing'.

In 'tokenization' werden alle Token-Typen einer Sprache definiert. Diese sind aus einem Identifikator und einem regulären Ausdruck zusammengesetzt. Der Reguläre Ausdruck wird vom Tokenizer verwendet, um Tokens in der übergebenen Zeichenfolge zu identifizieren. Optional können Token-Typen als 'omitted' markiert werden, um für die Kompilierung nicht weiter berücksichtigt zu werden.

Listing 5.6 Beispiel der Definition von Token-Typen

```
<tokenization>
  <!-- omitted tokens -->
  <token id="INLINE_COMMENT" omitted="true" start_regex="//" regex="//.*?[\n$]"/>
  <token id="STAR_COMMENT" omitted="true" start_regex="/\*" regex="/\*.*?\*/"/>
  <!-- some keywords -->
  <token id="WHILE" regex="while(?!\w)"/>
  <token id="DO" regex="do(?!\w)"/>
  <token id="CONTINUE" regex="continue(?!\w)"/>
  <token id="BREAK" regex="break(?!\w)"/>
  <!-- operators -->
  <token id="AND" regex="&amp;"/>
  <token id="ASTERISK" regex="\*"/>
  <token id="CIRCUMFLEX" regex="\^"/>
```

Die Reihenfolge der Definition entscheidet darüber, nach welchen Token-Typen zuerst gesucht wird. In Listing 5.6 werden zunächst Kommentare aus dem Quelltext ausgeschnitten, bevor nach weiteren Token-Typen gesucht wird. Es wird immer zunächst die vollständige Eingabe nach einem Token-Typ durchsucht, bevor die Suche nach einem Weiteren begonnen wird. Wird ein Token in der Zeichenfolge erkannt, so wird diese um das Token herum geteilt. Das Token wird dann in die Zeichenfolge eingefügt, was möglich ist, weil die Liste zur Verwaltung auch Tokens als Zeichen akzeptiert. Anschließend wird die Suche hinter dem Token fortgesetzt. Diese wird bis zum Ende der Zeichenfolge fortgesetzt. Die Umwandlung in eine Token-Liste war erfolgreich, wenn am Ende keine mehr in der Zeichenfolge stehen, sondern ausschließlich Tokens.

Der 'parsing'-Knoten ist in die Knoten 'symbols' und 'productionrules' unterteilt. 'symbols' listet alle verwendeten Nichtterminale der Grammatik als 'symbol'-Knoten. Sie besitzen lediglich einen Identifikator. Das erste aufgeführte Symbol wird als Startsymbol definiert. Die Reihenfolge der Definition der übrigen Symbole hat keine Auswirkungen. der Knoten 'productionrules' definiert die Produktionsregeln der Grammatik. Die hier definierten 'rule'-Knoten kapseln beliebig tief verschachtelte Gruppen von Ableitungsartefakten, wie Terminal- und

Listing 5.7 Beispiel einer Regel-Definition

```
<!-- IF_STATEMENT ::= "if" "(" EXPRESSION ")" BLOCK
                { "else if" "(" EXPRESSION ")" BLOCK }
                [ "else" BLOCK ] -->
<rule symbol="IF_STATEMENT">
  <item id="IF"/>
  <item id="BRACE_O"/>
  <item id="EXPRESSION"/>
  <item id="BRACE_C"/>
  <item id="BLOCK"/>
  <repeatable>
        <item id="ELSEIF"/>
        <item id="BRACE_O"/>
        <item id="EXPRESSION"/>
        <item id="BRACE_C"/>
        <item id="BLOCK"/>
  </repeatable>
  <optional>
    <item id="ELSE"/>
    <item id="BLOCK"/>
  </optional>
</rule>
```

Nichtterminalsymbolen, sowie Gruppierungsknoten. Die Definition von Alternativen in der Produktionsregel ist abgekürzt. Hier wird nach jedem gefundenen Knoten eine neue Alternative begonnen. In der resultierenden Objektstruktur sind diese Aber den unter ihnen folgenden Ableitungsartefakten übergeordnet.

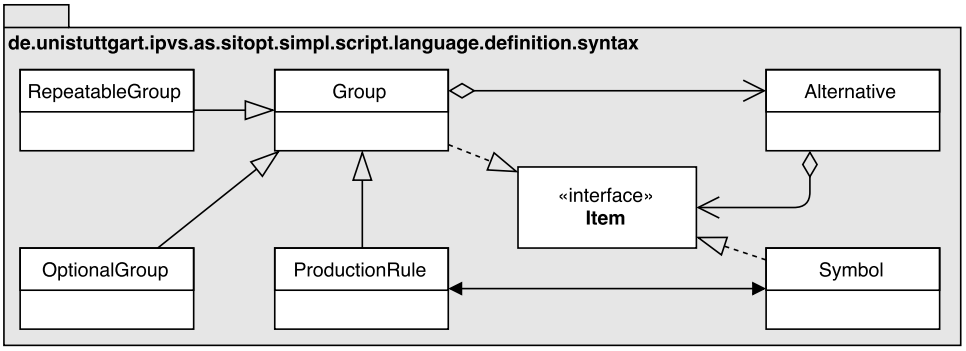

Abbildung 5.3: Reduziertes Klassendiagramm des Syntax-Paketes

5.2 Implementierung der Serveranwendung

Die Serveranwendung besteht, wie in Kapitel 4 - Konzeptionelle Lösung teilweise dargestellt, hauptsächlich aus Teilpaketen. Diese werden von einer Kernkomponente 'core' verwaltet. Dabei steht vor allem Modell im Vordergrund, dessen Objekte über die Klassen 'TestSuite' und 'TestCase' referenzierbar sind. Die Kernkomponente kennt die Objekte des Modells also nur indirekt. Das Modell wurde gemäß des in Kapitel 4 beschriebenen Metamodells implementiert. 'TestSuite'-Klassen verwalten außerdem selbst die Menge ihrer Prozessoren.

Auf die gleiche Art sind die Objekte der Filter und Netzwerkschnittstellen lediglich über den Umweg über ihren zugehörigen Testfall, oder zugehöriges Sensorobjekt referenzierbar. Durch die Kernkomponente direkt referenzierbar sind hingegen 'Language'-Objekte, welche wie oben beschrieben aus einer entsprechenden Definition ausgelesen werden. Sie kennen zur Laufzeit die jeweiligen Komponenten zur Übersetzung von Quelltext-Zeichenfolgen ihrer jeweils modellierten Sprache. Dabei ist zu beachten, dass der Prozess der Übersetzung angestoßen wird, sobald der Server gestartet, ein neues Programm definiert, oder ein altes Programm überschrieben wird. Das resultierende Programm wird dem entsprechenden ÖbjectStateObjekt zur Ablage übergeben, und vom Prozessor bei der Abarbeitung des Objektes ausgelesen.

Abgesehen von der Organisation des Modells, ist die Kernkomponente auch für die Kommunikation mit der Weboberfläche per WebSockets verantwortlich. Auf die zur Verfügung stehenden Operationen wird im Abschnitt 5.3 näher eingegangen.

5.3 Umsetzung der Weboberfläche

In der Konzeption wurde die Weboberfläche zunächst zurückgestellt, da sie sich nach der Serverimplementierung richten sollte. Sie ist in HTML und Javascript implementiert und bedient sich Websockets zur Kommunikation mit dem Server. Zur Gestaltung und Adaption werden *Cascading Style Sheets* verwendet.

Die Weboberfläche stellt eine dauerhafte Verbindung zum Server her und legt dazu eine Session an. Grund hierfür ist, dass Änderungen am Informationsstand der Servers nicht immer unmittelbar auf eine Eingabe des Benutzers folgen. Um neue Informationen zu erhalten müsste der Benutzer die Seite also regelmäßig neu aufbauen. Da dies unkomfortabel ist, und das Ziel einer Webanwendung verfehlt, wurde die Lösung so ausgelegt. Die Auswahl des zu bearbeitenden Elements wird aus einer von zwei Hierarchien ausgewählt. Die Bearbeitung findet in einem dynamisch angepassten Bearbeitungsbereich statt, welcher je nach ausgewähltem Elementtyp entsprechende Modifikationen zulässt.

Ist die Test-Suite selbst ausgewählt, lassen sich Einstellungen für die periodische Ausführung durchführen. Diese werden als 'Chron-jobs' Server-seitig installiert. Dies ist einfacher wartbar als eine integrierte Lösung. Außerdem gibt es eine Schaltfläche zum Anlegen eines neuen Testfalls. Der übrige Bearbeitungsbereich dient zur Anzeige von Log-Dateien. Hierbei handelt

es sich um eine Tabellen-ähnliche Ansicht, bei der Test-Suite-Durchläuft, und einzelne Testfälle auf- und zuklappbar sind. Ein fehlgeschlagener Testfall ist bei der ersten Anzeige zunächst geöffnet, ein erfolgreich durchgeführter nicht.

Modell-Verwaltung

Die Weboberfläche erlaubt die Auswahl einer Test-Suite über das Menü der Seite. Ist eine Test-Suite geladen, wird eine hierarchische Listen-Repräsentation des Modells in einem im Folgenden Modellansicht genannten Teilbereich angezeigt. In ihm können Objektklassen definiert, und zusammengestellt werden. Sind Objektklassen in einem Modell definiert, können Modellobjekte angelegt werden, die diesen zugehörig sind. Ein Modellobjekt ist - entsprechend Kapitel 4 - immer eine Repräsentation seiner Definierten Objektklasse.

Die Modellansicht erlaubt auch die Definition von Sensorobjekten. Diesen wird bei der Erstellung jeweils ein Modellobjekt, ein Objektattribut, und eine Netzwerkschnittstelle zugewiesen. Für letztere können optional eine URL, und ein String definiert werden, die bei der Serverseitigen Initialisierung des Modells dem Schnittstellenobjekt übergeben werden. Wie diese genutzt werden, bleibt der implementierenden Klasse überlassen, es liegt jedoch nahe, diese zur Verbindung mit dem Zielsystem, und zur Spezifizierung von Ereignisstrom-Eigenschaften zu nutzen. Es lassen sich außerdem Filterobjekte in die dem Sensor zugehörige Filterpipeline einfügen. Diese werden, wie in Kapitel 4 beschrieben sequenziell durchlaufen, wenn ein Wert aus dem Objektattribut ausgelesen wird.

Testfall-Verwaltung

Die zweite Hierarchie erlaubt den Zugriff auf Testfälle, die für diese definierten Szenarien, und damit auf die Startzustände der Objektattribute. Um für ein Szenario für den Normalfall, oder einen Testfall zu modifizieren, wird der entsprechende Testfall, oder die entsprechende Test-Suite in der Hierarchie ausgewählt, und anschließend für Modellobjekte Attributszustände überschrieben.

Um das korrekte Verhalten der Zielplattform zu überprüfen, können eine, oder mehrere Netzwerkschnittstellen definiert werden, über den die Zielplattform Ereignisse an den Server zurücksenden kann. Für jede definierte Schnittstelle werden die jeweils eingehenden Ereignisse in String-Form mit einem regulären Ausdruck verglichen. Wird für einen dieser Ausdrücke eine passende Zeichenfolge empfangen, so wird der Test als erfolgreich gewertet. 'Und'-Verknüpfungen lassen sich erreichen, indem mehrere Testfälle angelegt werden. verglichen, um festzustellen, Außerdem ist die Einstellung einer Zeitspanne möglich, über welche ein Testfall durchgeführt wird. Während dieser Zeitspanne wird weiter simuliert und Ereignisse abgesetzt, auch wenn das erwartete Antwortsignal von der Zielplattform bereits eingetroffen ist.

Weitere Funktionen

Um Funktionen auf dem Server nutzen zu können, muss sich der Benutzer anmelden. Das Anlegen neuer Benutzer ist nicht über die Weboberfläche möglich.

Um es zu ermöglichen neue Filter- und Schnittstellenklassen in Modellen einzusetzen, bietet die Weboberfläche eine über das Menü erreichbare Funktion, um neue Klassen zu übergeben. Werden diese erfolgreich geladen, können sie in Modellen verwendet werden. Es ist gegenwärtig nicht möglich über die Weboberfläche Klassen wieder zu löschen, da es sich nur schwer feststellen lässt, welche Test-Suites davon betroffen sind. Um eine existierende Klasse zu löschen oder zu ersetzen, ist also ein Zugriff auf anderem Weg erforderlich.

6 Zusammenfassung und Ausblick

In dieser Arbeit wurde die Konzipierung und Implementierung einer Serverplattform zur Simulation von dynamischen Ereignisströmen durchgeführt. Dabei wurde der Fokus zunächst auf die zu definierenden Anforderungen gerichtet, um danach ein entsprechendes Konzept auszuarbeiten. In diesem wurde vor allem die Fragestellung zur Dynamik im System hervorgehoben. Die anschließende Implementierung richtete sich zunächst auf die Ausführung von Programmen und deren Übersetzung aus, und konzentrierte sich anschließend auf die Ankopplung dieser Funktionalitäten an das Modell. Anschließend wurden Schnittstellen zwischen einer Weboberfläche und der Serverapplikation hergestellt, um diese komfortabler bedienen zu können.

Zu Beginn wurde in dieser Ausarbeitung wurde das Thema in den Kontext von IoT und von Industrie 4.0 gesetzt. Außerdem wurden die grundlegenden Konzepte der komplexen Ereignissverarbeitung beschrieben. Darüber hinaus wurde das Modell der kontextfreien Grammatiken erläutert und ihr Zusammenhang mit der Syntax von Programmiersprachen aufgezeigt.

Es wurden die funktionalen und nicht-funktionalen Anforderungen an die entwickelte Plattform erfasst. Dabei ergab sich, dass die modellbasierte Simulation für die Umsetzung der Plattform gewählt werden sollte. Ein Metamodell, dessen Aufbau synchrone und asynchrone Lesezugriffe zum Erzeugen von Ereignissen zulässt wurde ausgelegt. Der Zugriff auf modelleigene Objekte wurde organisiert. Eine Datenstruktur zum Anlegen baumartig strukturierter Programm-Artefakte wurde ausgelegt und es wurde ein Konzept zu ihrer Ausführung erarbeitet. Kernstück davon ist ein virtueller Prozessor mit Stapel-organisiertem Ablaufverhalten.

Die Notwendigkeit für die Konzipierung einer Übersetzungskomponente von Quelltexten wurde deutlich. Es wurde eine Kette von Teilübersetzern ausgearbeitet und die von ihnen durchgereichten Datenstrukturen erfasst. Ein Metamodell zur Definition von Sprachen wurde ausgelegt. Ein Konzept für die Anwendung von Filtern für simulierte Daten wurde zusammengefasst. Schlussendlich wurden diese Konzepte in einer Form zusammengefasst, die sie für eine Unit-Test-artige Plattform-Konzeption geeignet macht.

Die konkrete Umsetzung und Implementierung wurde durchgeführt. Dabei wurde Wert auf die Erweiterbarkeit gelegt, weshalb, wo immer sinnvoll möglich, Interfaces in gekapselte Pakete mit eingeplant wurden, um die Anbindung neuer Funktionalitäten so leicht wie möglich zu gestalten. Die Organisation von Kernklassen, wie dem Prozessor, oder des Stapels, wurde in Auszügen veranschaulicht. Es wurde gezeigt, wie die Definition einer Sprache in ein für die Plattform verwertbares Modell gebracht werden kann.

Die Weboberfläche wurde an die Funktionalitäten des Servers angepasst, und es wurden Benutzerschnittstellen geschaffen, welche die Verwaltung der auf dem Server organisierten Testfälle und Test-Suites ermöglichen. Es wurden JUnit-Tests zur Validierung der Komponenten angelegt. Mit ihrer Hilfe ließ sich die funktionale Erfüllung der Anforderungen zeigen.

Die Script-API kann aber aufgrund ihrer Kapselung bei Bedarf in anderen Projekten wiederverwendet werden, was den hohen Entwicklungsaufwand rechtfertigt.

6.1 Ausblick

Die entwickelte Plattform SimPl ist für die angedachte Nutzung geeignet, und erfüllt die in Kapitel 3 - Anforderungen aufgeführten Anforderungen. Sie kann zum Testen und zur Fehlersuche in Ereignisverarbeitungsplattformen genutzt werden. Es ist auch denkbar, die Plattform über ihren eigentlichen Zweck hinaus in anderen Bereichen einzusetzen, beispielsweise als Werkzeug zur Modell-gesteuerten Regelung.

Es ist anzumerken, dass die gegenwärtige Umsetzung der Plattform Optimierungen im Bereich der Nutzerschnittstellen zulässt. Sie bedarf außerdem einer organisierten Benutzerverwaltung mit verschlüsselten Zugangsdaten und einer Rechteverwaltung. Darüber hinaus gibt es momentan keine Möglichkeit zur Laufzeit, oder auf dem Web über die Benutzeroberfläche, eine Implementierung des Mapper-Interfaces an den Server zu übergeben. Diese Hürden, sowie alle weiteren, welche in der Langzeitnutzung auffallen, können zu einem späteren Zeitpunkt in einer neuen Arbeit weiter vergeben werden.

Es ist vorgesehen die Plattform als Open-Source-Projekt freizugeben. Dies erfordert allerdings zunächst noch eine Nacharbeit im Bereich der Quellcode-Organisation und der Dokumentation der Klassen und Interfaces.

Literaturverzeichnis

[15] *Esper Reference*. 5.3.0. EsperTech Inc. 2015. URL: http://www.espertech.com.

[Adi07] A. Adi. *Complex Event Processing*. Englisch. IBM Corporation. Jan. 2007. URL: https://www.research.ibm.com/haifa/dept/services/papers/cep_Jan07_nc.pdf.

[Ale10] M. K. Alecander Pflaum. „Das intelligente Parfüm kennt seinen Nachbarn - Warensicherung mit Smart Object Technologie". In: *RFID special* (2010).

[Ash09] K. Ashton. „That 'Internet of Things' Thing". In: *RFID Journal* (Juni 2009), http://www.rfidjournal.com/articles/view?4986.

[Dan16] I. S. Danda B. Rawat Joel J.P.C. Rodrigues. *Cyber-Physical Systems - From Theory to Practice*. ISBN-13: 978-1-4822-6333-6. CRC Press - Taylor & Francis Group, 2016.

[Fri10] C. F. Friedemann Mattern. „From the Internet of Computers to the Internet of Things". In: *Kai Sachs, Ilia Petrov, Pablo Guerrero (Eds.): From Active Data Management to Event-Based Systems and More. LNCS, Vol. 6462, Springer, pp. 242-259* (2010).

[Gar13] Gartner. *CEP*. 2013. URL: http://www.gartner.com/it-glossary/complex-event-processing.

[Gen06] J. Gennick. *SQL - kurz & gut*. Hrsg. von S. Aping. 2. Aufl. O'Reilly's Taschenbibliothek, 2006.

[Hea03] S. Heath. *Embedded Systems Design*. 2. Aufl. ISBN 0-7506-5546-1. Newnes, 2003.

[Ian11] A. M. Ian Fette. *The Websocket Protocol*. Internet Engineering Task Force. https://tools.ietf.org/html/rfc6455.pdf, Dez. 2011.

[Jam15] G. S. e. a. James Gosling Bill Joy. *The Java Language Specification*. Feb. 2015.

[Mar11] A. Maruoka. *Concise Guide to Computation Theory*. Springer Verlag London Limited, 2011.

[Mar15] B. O. Mario Hermann Tobias Pentek. *Design Principles for Industrie 4.0 Scenarios: A Literature Rewiev*. Techn. Ber. http://www.snom.mb.tu-dortmund.de/cms/de/forschung/Arbeitsberichte/Design-Principles-for-Industrie-$4_0 - Scenarios.pdf$: Technische Universität Dortmund, Fakultät Maschinenbau - Audi Stiftungslehrstuhl Supply Net Order management, 2015.

[Nad04] K. W. Nadim Maluf. *An Introduction to Microelectromechanical Systems Engineering*. 2. Aufl. Microelectromechanical Systems Series ISBN 1-58053-590-9. Artech House, Inc., 2004.

[Smo01] G. Smolka. *Programmierung - eine Einführung in die Informatik mit Standard ML.* 2. Aufl. ISBN: 978-3-486-70517-1. Oldenbourg Wissenschaftsverlag GmbH, 2001.

Alle URLs wurden zuletzt am 20. 04. 2016 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift