Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

Daniel Steiman

Kontexterkennung in Multiagentensimulationen auf Basis von Reactive Computing

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Daniel Steiman

Kontexterkennung in Multiagentensimulationen auf Basis von Reactive Computing

Masterarbeit eingereicht im Rahmen der Masterprüfung

im Studiengang Master of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Thomas Thiel-Clemen
Zweitgutachter: Prof. Dr. Stefan Sarstedt

Eingereicht am: 1. Oktober 2014

Daniel Steiman

Thema der Arbeit

Kontexterkennung in Multiagentensimulationen auf Basis von Reactive Computing

Stichworte

Context-Awareness, Reactive Computing, Multiagentensimulationen, Event-Driven Architecture, Complex Event Processing

Kurzzusammenfassung

Kontextbewusste Anwendungen bieten die Möglichkeit, ihr Verhalten ohne die aktive Einflussnahme von Anwendern selbstständig an vorherrschende Situationen anpassen. Aus diesem Grund wurden bereits Toolkits, Frameworks und Ansätze entwickelt, die eine einfache Entwicklung von kontextbewussten Anwendungen ermöglichen sollen. Die bisher auf dem Gebiet der *Context-Awareness* vorgestellten Arbeiten weisen jedoch Nachteile auf, die einen Einsatz in bestimmten Anwendungsbereichen stark einschränken oder sogar völlig ausschließen. Diese Arbeit zeigt die Nachteile bestehender Arbeiten auf und befasst sich mit der Entwicklung und Realisierung eines neuen Ansatzes auf Basis des *Reactive Computing* Paradigmas.

Daniel Steiman

Title of the paper

Context-Awareness in Multi-agent Simulations Based on Reactive Computing

Keywords

Context-Awareness, Reactive Computing, Multi-agent Simulations, Event-Driven Architecture, Complex Event Processing

Abstract

Context-aware applicatons offer the possibility to adapt their behavior autonomously to prevailing situations without the active influence of users. For this reason, toolkits, frameworks and approaches have been developed to allow the easy development of context-aware applications. However, former introduced approaches in the area of context-awareness have disadvantages which strongly limit or even completely exclude their use in certain application areas. This work shows the drawbacks of existing approaches and deals with the development and realization of a new concept based on the *reactive computing* paradigm.

Inhaltsverzeichnis

1. Einführung 1

1.1. Ziel der Arbeit . . . 2

1.1.1. Hypothesen . . . 2

1.2. Abgrenzung . . . 3

1.3. Aufbau der Arbeit . . . 4

2. Grundlagen 6

2.1. Kontext und Context-Awareness . . . 6

2.1.1. Definition von Kontext und Context-Awareness . . . 7

2.1.2. Ansätze für Context-Awareness . . . 8

2.1.3. Ansätze zur Kontextmodellierung . . . 10

2.2. Multiagentensysteme und Multiagentensimulationen . . . 12

2.2.1. Beispiele für Multiagentensysteme . . . 13

2.2.2. Beispiele von Anwendungsszenarien . . . 14

2.3. Context Reasoning . . . 14

2.4. Event-Driven Architectures und Complex Event Processing . . . 15

2.5. Grundlagen Event-Driven Architectures . . . 15

2.5.1. Was sind Ereignisse . . . 16

2.5.2. Vergleich mit herkömmlichen Architekturen . . . 17

2.6. Complex Event Processing . . . 18

2.6.1. Definition des Begriffs . . . 18

2.6.2. Grundlegende Konzepte und Eigenschaften . . . 19

2.6.3. Komplexe Ereignisse . . . 21

2.6.4. Ereignismuster . . . 22

2.6.5. Zentrale Bestandteile und weitere Begriffe . . . 23

2.7. Paradigmen . . . 24

2.7.1. Responsive Computing . . . 24

2.7.2. Reactive Computing . . . 25

2.7.3. Zusammenfassung . . . 26

3. Verwandte Arbeiten 27

3.1. Ontologiebasierter Ansatz - Gaia . . . 27

3.1.1. Kontextmodell . . . 27

3.1.2. Kontextverarbeitung . . . 28

3.1.3. Kontextzugriff . . . 28

3.1.4. Infrastruktur und Beispielanwendung 29
3.2. Objektorientierter Ansatz - JCAF 30
3.2.1. Kontextmodell 30
3.2.2. Kontextverarbeitung 31
3.2.3. Kontextzugriff 31
3.2.4. Runtime Architektur 32
3.3. Eignung als Kontext-Komponente für Multiagentensysteme 33
3.4. Diskussion von Gaia und JCAF 35
3.4.1. Fazit 36

4. Analyse **37**
4.1. Grundlegende Anforderungen an die Simulationsumgebung 37
4.2. Grundlegende Anforderungen an die Context-Service Komponente 38
4.2.1. Funktionale Anforderungen 39
4.2.2. Nichtfunktionale Anforderungen 40
4.3. Beispielszenario 41
4.4. Räumliche Darstellung und Umwelt 42
4.4.1. Sektoren 43
4.4.2. Aussichtspunkte 43
4.4.3. Wasserstellen 44
4.5. Agenten 44
4.5.1. Geparden 44
4.5.2. Gazellen 44
4.6. Verschiedene Aspekte von Kontext 45
4.7. Relevante Kontexte 46
4.7.1. Beispiele 46
4.8. Zusammenfassung 47

5. Design **48**
5.1. Simulationsumgebung 48
5.1.1. Gesamtbild der MARS-Architektur 49
5.1.2. Layerstruktur 51
5.1.3. Agenten 52
5.1.4. Integration in MARS 53
5.2. MARS Surveyor 55
5.2.1. Systemarchitektur 55
5.2.2. Kontextmodellierung 57
5.2.3. Kontextverarbeitung 58
5.2.4. Abfrage von Kontext 60
5.3. Ereignistypen 61
5.3.1. Geparden 61
5.3.2. Gazellen 62
5.3.3. Kill Events 63

6. Realisierung **64**
6.1. Context-Service Client 64
6.1.1. ContextServiceClient 65
6.1.2. EventProducer 66
6.1.3. ContextListener 67
6.2. Testumgebung 67
6.2.1. Simulation 68
6.2.2. Nachrichtenkommunikation 71
6.2.3. Context-Service 73

7. Experimente **77**
7.1. Messgrößen 77
7.2. Testszenarien 78
7.2.1. Kontexterkennung 78
7.2.2. Performance 81
7.2.3. Integrierbarkeit 87
7.2.4. Flexibilität 89
7.3. Nicht Teil der Tests 92
7.3.1. Partitionierung 92
7.3.2. Synchronisation 93
7.4. Weitere Anwendungsszenarien 94
7.4.1. Kollisionserkennung 94
7.4.2. Kontext-Dashboard 95

8. Ergebnisse und Diskussion **96**
8.1. Vorteile des Reactive Computing Paradigma 97
8.2. Performance 99
8.3. Skalierbarkeit 100
8.4. Kontexterkennung 101
8.5. Flexibilität 102
8.6. Eignung für Multiagentensimulationen 103

9. Fazit und Ausblick **105**
9.1. Ausblick 106

A. Inhalt der DVD **107**

B. Danksagung **108**

Listings

3.1. Beispiel eines Kontextmusters und der aufzurufenden Methode in Gaia 29
3.2. Beispiel eines EntityListener in JCAF 32

6.1. Verbindungsaufbau zum Context-Service 65
6.2. Registrieren neuer Ereignistypen beim Context-Service 65
6.3. Registrieren neuer Kontextregeln beim Context-Service 65
6.4. Delegate-Methode die aufgerufen wird wenn die zugehörige Kontextregel vom Context-Service erkannt wurde 66
6.5. Konstruktor der *EventProducer*-Klasse 66
6.6. Erzeugen einer neuen *EventProducer*-Instanz und Versenden eines *GazelleEvent* 66
6.7. Empfang einer Trigger-Nachricht vom Context-Service und Aufruf des zugehörigen Delegates als Reaktion 67
6.8. Ausschnitt aus der Implementierung des Gepard-Agenten 69
6.9. Codieren und Senden von *GepardEvent*-Ereignisnachrichten 71
6.10. Empfang und Decodierung von Ereignisnachrichten 72
6.11. Initialisierung der *ContextRuleBase* und Registrieren von Kontextregeln 75
6.12. Versenden einer Trigger-Nachricht an die Simulation 76

7.1. Empfangen und Verarbeiten einer Context-Service Benachrichtigung 88
7.2. Implementierung des Ereignistypen *KillEvent* als *Plain Old Java Object* (POJO) 89
7.3. Registrierung des *KillEvent*-Ereignistypen bei der CEP-Engine 90
7.4. Verarbeitung einer *KillEvent*-Ereignisnachricht 91
7.5. Registrieren einer Kontextregel zur Kollisionserkennung 94
7.6. Delegate-Methode die aufgerufen wird wenn eine Kollision vom Context-Service erkannt wurde 94

1. Einführung

Eine Anwendung die kontextbewusst ist, erfasst Informationen aus ihrer Umgebung und wertet sie aus, um sich automatisch den aktuellen Gegebenheiten und Ereignissen anzupassen [Abowd u. a. (1999)]. Jeder Vorgang in der realen Welt, als auch jede Zustandsänderung in einem System, kann als ein Ereignis angesehen werden. All diese Ereignisse können einen Einfluss auf den Ablauf eines Prozesses einer Anwendung nehmen und sich somit auf dessen weiteren Gesamtverlauf auswirken. In komplexen Simulationen von Modellen, können dabei je nach Simulation extrem große Anzahlen von Agenten zum Einsatz kommen. Jede Veränderung des Simulationszustands bedeutet eine Kontextveränderung und somit eine riesige Menge von Ereignissen, die kontinuierlich und möglichst performant verarbeitet werden muss.

Die Herausforderung besteht inzwischen darin, aus der Informationsflut die wichtigen Erkenntnisse herauszuziehen und darauf entsprechend zu reagieren. Bei bisherigen Ansätzen ist es so, dass alle Informationen, welche beispielsweise durch das Auftreten von Ereignissen entstehen, zunächst gespeichert werden. Danach können entweder manuelle Abfragen von Kontextänderungen gemacht oder automatische Benachrichtigungen über entsprechende Kontextänderungen veranlasst werden. Die Arbeitsweise sowie der architektonische Aufbau dieser Ansätze verfolgen das Paradigma des *Responsive Computing* [Engel und Etzion (2011)]. Diese Vorgehensweise ist jedoch für eine Auswertung, der in einer Simulation kontinuierlich auftretenden Ereignisse, zu unperformant. Zudem sind die analysierbaren Daten, durch den Zwischenschritt des Abspeicherns und Aufbereitens der Informationen, im Normalfall bereits veraltet.

Das Paradigma des *Reactive Computing* [Engel und Etzion (2011)] verfolgt dagegen einen anderen Architekturstil, durch den Vorteile in mehreren Bereichen entstehen können. Dazu zählen zum Beispiel die Bereiche Performance, Modularität, Skalierbarkeit und Integrierbarkeit. Die durch einen Wechsel vom Responsive Computing zum Reactive Computing Paradigma entstehenden Vorteile im Anwendungsbereich der Kontexterkennung in Multiagentensimulationen, werden im Verlauf dieser Arbeit aufgezeigt.

1.1. Ziel der Arbeit

Motivation ist der Ausgangspunkt, dass es unter den bekannten Ansätzen zur Kontexterkennung [Kumar und Xie (2012); Baldauf u. a. (2007)] keine Lösung gibt, die sich auf Kontexterkennung in komplexen Multiagentensimulationen fokussiert. Die vorhandenen Lösungen haben einen anderen Anwendungsfokus oder weisen generelle Nachteile auf, weshalb sie für einen sinnvollen Einsatz in dem hier behandelten Anwendungsgebiet der Multiagentensimulation nicht zu empfehlen sind. Die verwandten Arbeiten nennen außerdem offene Punkte, wie zum Beispiel die Erweiterung um die Möglichkeit der Erkennung von temporalen Korrelationen zwischen Kontexten, sowie die Möglichkeit der Formulierung von Kontextregeln mit temporalen Constraints [Ranganathan und Campbell (2003)].

Ziel dieser Arbeit ist es nun, ein eigenes Konzept zur Realisierung eines Context-Service auszuarbeiten, welches primär für einen Einsatz in komplexen Multiagentensimulationen zugeschnitten ist. Das entwickelte Konzept soll sich außerdem paradigmatisch nicht an Responsive Computing, sondern an Reactive Computing orientieren. Dabei wird angestrebt, die in den verwandten Arbeiten identifizierten Nachteile zu beheben sowie die darin noch offenen Punkte zu lösen. Dadurch soll vor allem das Potenzial des Reactive Computing Paradigmas gegenüber dem Paradigma des Responsive Computing für die Anwendungsdomäne der Multiagentensimulation aufgezeigt werden.

1.1.1. Hypothesen

Mit dem in dieser Arbeit vorgestellten Konzept eines Context-Service für Multiagentensimulationen, sollen daher die folgenden Hypothesen untersucht werden:

1. Durch einen Wechsel von einem Responsive Computing zum Paradigma des Reactive Computing, können bisherige Ansätze auf dem Gebiet der Context-Awareness unter anderem in den Bereichen Performance, Modularität, Skalierbarkeit und Integrierbarkeit verbessert werden.

2. Durch die Kombination einer ereignisgesteuerten Architektur mit der Technik des Complex Event Processing, kann ein Context-Service realisiert werden, der die performante Auswertung von Kontextinformationen und Erkennung von Kontextmustern in komplexen Multiagentensimulationen in Echtzeit gewährleisten kann.

3. Die Verarbeitungsgeschwindigkeit des Context-Service skaliert mindestens linear. Das heißt, dass der Skalierungsfaktor des Systems mit steigendem Volumen an auszuwer-

tenden Kontextdaten gleich bleibt oder sogar kleiner wird. Dies gilt ebenfalls für eine steigende Anzahl von definierten Kontextregeln.

4. Der entwickelte Context-Service gewährleistet eine zuverlässige Erkennung von definierten Kontexten und Teilkontexten in dem jeweils simulierten Szenario. Dazu gehört unter anderem die Erkennung von zeitlichen (temporalen), örtlichen (spatialen) sowie zeitlich und örtlich (spatiotemporal) zusammenhängenden Kontexten.

5. Der Context-Service ist gegenüber konventionellen Ansätzen flexibler in der Anpassung an sich ändernde Simulations- und Anwendungsszenarien. Das bedeutet, dass sich der Context-Service mit wenig Implementierungsaufwand für neue Simulationsszenarien konfigurieren lässt.

Um die aufgestellten Hypothesen verifizieren bzw. widerlegen zu können, wird ein Konzept eines Context-Service auf Basis von Reactive Computing entwickelt. Zusätzlich wird eine Simulationsumgebung mit einem Testszenario entwickelt, mit dem der Context-Service unter realen Bedingungen getestet werden kann.

1.2. Abgrenzung

Der Schwerpunkt dieser Arbeit, ist die Entwicklung eines Konzepts für einen Context-Service auf Basis des Reactive Computing Paradigmas. Mit dem entwickelten Konzept sollen Erkenntnisse darüber gewonnen werden, welche Vorteile sich durch diesen Paradigmenwechsel gegenüber Konzepten auf Basis des Responsive Computing Paradigmas ergeben. Der Anwendungsfokus des in dieser Arbeit entwickelten Context-Service, liegt bei Multiagentensimulationen mit hohem Volumen an Kontextinformationen, die performant ausgewertet werden müssen. Bei der Entwicklung der Testumgebung und der Untersuchung des Context-Service Konzepts sind die zwei folgenden Aspekte nicht Teil dieser Arbeit:

- **Synchronisation und Zeitmanagement**: Die Berücksichtigung eines Zeitmanagements zur Synchronisation der Multiagentensimulation und des entwickelten Context-Service, ist für die Untersuchung der aufgestellten Hypothesen nicht zwingend notwendig. Die Architektur des Context-Service basiert auf dem Reactive Computing Paradigma und setzt daher eine asynchrone Kommunikation zwischen Komponenten voraus. Eine gegebenenfalls asynchrone Übertragung von Kontextinformationen im Bezug auf Simulationsiterationen (Ticks), beeinträchtigt die Funktionsweise des Context-Service daher nicht.

- **Validierung des Simulationsmodells**: Die exakte und detailgetreue Simulation von Agenten und Umwelt in dem simulierten Testszenario ist nicht Teil dieser Arbeit, da dies keinen Einfluss auf die Untersuchung der aufgestellten Hypothesen hat. Das Simulationsmodell ist lediglich ein Werkzeug zur Überprüfung der Einsatzfähigkeit und der Funktionsweise des entwickelten Context-Service Konzepts, in Bezug auf die aufgestellten Hypothesen. Eine detailgetreue Implementierung eines validierten Simulationsmodells, hätte keinen Einfluss auf die Ergebnisse der Auswertung und ist daher nicht Teil dieser Arbeit.

1.3. Aufbau der Arbeit

Zu Beginn werden in Kapitel 2 zunächst die essentiellen Grundlagen für das weitere Verständnis dieser Arbeit erläutert und die entsprechende Terminologie eingeführt. Zunächst erfolgt die Definition und Erläuterung der Begriffe Kontext und Context-Awareness. Anschließend werden verschiedene Ansätze für Context-Awareness von Anwendungen sowie Ansätze zur Kontextmodellierung eingeführt. Danach folgt eine Einführung in die Thematik und Konzepte der Multiagentensysteme- und simulationen, der Event-Driven Architectures und des Complex Event Processing, sowie eine Einführung in die Paradigmen des Responsive Computing und des Reactive Computing.

Aufbauend auf die in Kapitel 2 eingeführten Ansätze für Context-Awareness, werden in Kapitel 3 die zwei Referenzmodelle *Gaia* [Ranganathan und Campbell (2003)] und *JCAF* [Bardram (2005b)] detaillierter betrachtet sowie deren Nachteile und Probleme identifiziert. Diese werden ausführlich analysiert und diskutiert, um den Rahmen und die Problemstellung dieser Arbeit genauer zu erläutern.

Kapitel 4 befasst sich mit der Analyse für die Entwicklung des Context-Service Konzepts und die Testumgebung, in der das Konzept evaluiert werden soll. Dazu werden zunächst die Annahmen und Rahmenbedingungen festgehalten, auf denen der Entwurf des Konzepts aufbaut. Danach werden die konkreten Anforderungen an den Context-Service bezüglich der Kriterien Architektur, Kontextmodellierung, Context Reasoning, Kontextabfrage und Performance betrachtet. Den zweiten Teil des Kapitels bildet die Analyse der Multiagentensimulation, mit der der Context-Service getestet werden soll. Es wird ein Beispielszenario sowie die darin zu modellierenden Umwelteigenschaften, Ereignistypen und relevanten Kontexte erläutert.

In Kapitel 5 wird das Design des Context-Service (*MARS-Surveyor*) und dessen Integration in das MARS-Framework beschrieben. Außerdem wird das Design der Agenten in der Multiagentensimulation sowie die Simulationsarchitektur selbst erläutert. Im weiteren Verlauf des Kapitels, werden die geplante Visualisierung der Simulation und die Designentscheidungen für wichtige Faktoren, wie zum Beispiel die Modularität des Systems beleuchtet. Anschließend folgt eine genauere Darstellung der Kontextmodellierung und eines Ereignismodells für das prototypische Simulationsszenario.

In Kapitel 6 wird die Realisierung der Testumgebung und seiner wesentlichen Bestandteile dokumentiert. Zu diesen wesentlichen Bestandteilen gehört die Multiagentensimulation, bei der auf die Implementierung der Agenten und der Umwelt sowie die Visualisierung eingegangen wird. Danach wird der Aspekt der Kommunikation der Simulation mit dem Context-Service mit Hilfe einer Message Oriented Middleware erläutert. Abschließend wird die Realisierung des Context-Service und seiner Teilkomponenten beschrieben.

Kapitel 7 beschreibt die Experimente, die mit der realisierten Testumgebung durchgeführt werden sollen. Es werden zunächst die Messgrößen erläutert, mit denen die durchgeführten Experimente dokumentiert und bewertet werden können. Danach folgt eine Beschreibung verschiedener Testszenarien, die sich aus unterschiedlichen Konfigurationen der Multiagentensimulation und des Context-Service ergeben. Zum Beispiel kann die Anzahl der simulierten Agenten oder der definierten Kontextregeln variieren. Es werden außerdem Aspekte genannt, die bei den Experimenten nicht behandelt werden.

In Kapitel 8 werden die Ergebnisse, der im vorangegangenen Kapitel durchgeführten Experimente, dokumentiert und diskutiert. Es werden die Erkenntnisse zur Performance, Kontexterkennung, Integrierbarkeit und Flexibilität des Context-Service festgehalten. Anhand der dokumentierten Ergebnisse werden dann Aussagen zu den aufgestellten Hypothesen getroffen.

Zum Schluss folgt in Kapitel 9 eine Zusammenfassung der Ergebnisse dieser Arbeit. Im Ausblick werden außerdem noch mögliche offene Punkte und Verbesserungsmöglichkeiten genannt.

2. Grundlagen

Das nun folgende Kapitel geht auf die für das weitere Verständnis dieser Arbeit notwendigen Grundlagen ein und gibt eine Einführung in die entsprechende Terminologie. Zu Beginn des Kapitels, wird auf die Definition und Bedeutung von Kontext sowie Context-Awareness im informationstechnischen Sinne eingegangen. Anschließend folgt eine Einführung in Multiagentensysteme und Multiagentensimulationen. Danach werden die Konzepte der Event-Driven Architectures und des Complex Event Processing, sowie die Verbindung zwischen diesen Konzepten erläutert. Abschließend werden die Paradigmen des Responsive Computing und des Reactive Computing betrachtet und der Unterschied zwischen diesen Paradigmen aufgezeigt.

2.1. Kontext und Context-Awareness

Zunächst geht es darum, einen grundlegenden Einstieg in die Thematik Kontext zu bekommen und die folgenden Fragen zu klären:

1. Was ist Kontext und Context-Awareness im informationstechnischen Sinne?
2. Wie werden Kontextinformationen modelliert und welche Ansätze dafür gibt es?
3. Wie werden die Kontextinformationen in den vorhandenen Ansätzen verwaltet und verarbeitet?
4. Welche Ansätze zeigen das größte Potenzial auf und sind vorwiegend im Einsatz?
5. Eignen sich diese Ansätze für einen konkreten Einsatz in komplexen Multiagentensimulationen?

Auf diese Fragen wird nun im weiteren Verlauf dieses Kapitels, als auch in Kapitel 3 genauer eingegangen.

2.1.1. Definition von Kontext und Context-Awareness

Die in der Literatur am häufigsten verwendete Definition von Kontext im informationstechnischen Sinne stammt von Anind K. Dey und Gregory D. Abowd [Abowd u. a. (1999)] und lautet wie folgt:

> "Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves."

Diese Definition von Kontext deckt damit das Verständnis von physischem als auch emotionalem Kontext ab. Wichtig ist, dass die Definition eine Beschränkung der Kontextinformationen, auf ausschließlich für die Anwendung bzw. den Benutzer relevante Kontextinformationen, vorsieht. Dies erscheint äußerst sinnvoll, da sonst irrelevante Informationen ebenfalls ausgewertet werden und unnötig Ressourcen verbrauchen würden. Neben dem Begriff Kontext definieren Dey und Abowd außerdem den Begriff Context-Awareness folgendermaßen:

> "A system is context-aware if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task." [Abowd u. a. (1999)]

Dies ist eine sinnvolle Definition von Context-Awareness, die den grundlegenden Aspekt einer kontextbewussten Anwendung darstellt und verschiedene Anwendungsdomänen abdeckt. In dieser Arbeit handelt es sich bei der Anwendung um eine Multiagentensimulation. Nach der Definition wäre die Simulation kontextbewusst, wenn sie die verfügbaren Kontextinformationen verwendet, um relevante Informationen und Dienste bereitzustellen. In diesem Fall wäre dies die Information über in der Simulation erkannte Kontextmuster und das Anstoßen von entsprechenden Prozeduren als Reaktion. Die folgende Abbildung 2.1 veranschaulicht ein simples Simulationsbeispiel eines Gebäudeinneren mit drei Sektoren, in denen sich Personen aufhalten. Die Sektoren sind an den farblichen Unterteilungen zu erkennen. Die Simulation könnte nun zum Beispiel als Kontextinformation auswerten, wie viele Personen sich auf einmal in jedem Sektor aufhalten und dies gegen einen definierten Schwellenwert und ein Zeitfenster prüfen. Wird der Schwellenwert länger als das vorgegebene Zeitfenster überschritten, kann eine automatische Benachrichtigung über diese Kontextsituation erfolgen. Falls nötig kann dann eine entsprechende Reaktion angestoßen werden. Die Simulation würde sich somit kontextbewusst verhalten. Sie verwendet zur Verfügung stehende Kontextinformationen, um relevante Informationen und Dienste bereitzustellen.

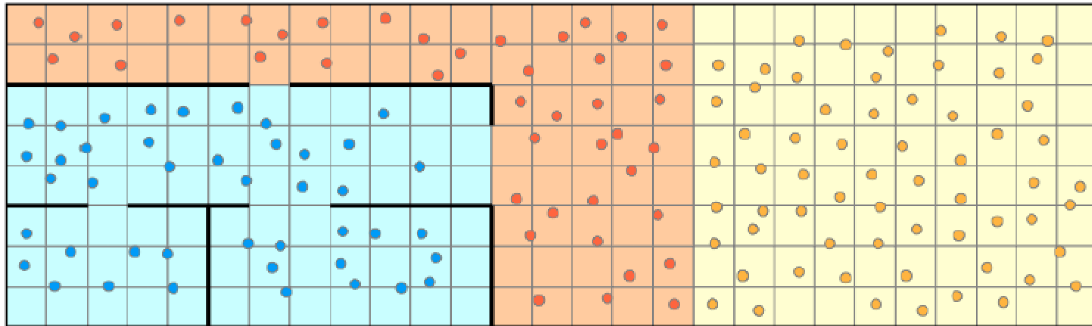


Abbildung 2.1.: Multiagentensystem mit mehreren Sektoren [Thiel (2013)]

2.1.2. Ansätze für Context-Awareness

Mittlerweile gibt es neben dem *Context Toolkit* [Dey u. a. (2001)] von Dey und Abowd einige weitere Arbeiten, die sich mit Konzepten zur Entwicklung kontextbewusster Anwendungen beschäftigen. Zu den bekanntesten gehören unter anderem:

- **SOCAM** [Gu u. a. (2004)]: SOCAM ist eine serviceorientierte Middleware, die die Entwicklung kontextbewusster Anwendungen unterstützt. Die Architektur der *Service-Oriented Context-Aware Middleware* (SOCAM) ist in drei Layer unterteilt: einem Sensor-Layer mit verschiedenen Sensoren, einem Middleware-Layer mit Kontextprovidern, -interpretern und -datenbanken in der Mitte und einem Anwendungs-Layer ganz oben (vgl. Abbildung 2.2). Für die verteilte Kommunikation zwischen den Komponenten verwendet SOCAM Java RMI. Kontextinformationen werden in SOCAM mit Hilfe eines ontologiebasierten Kontextmodells repräsentiert.

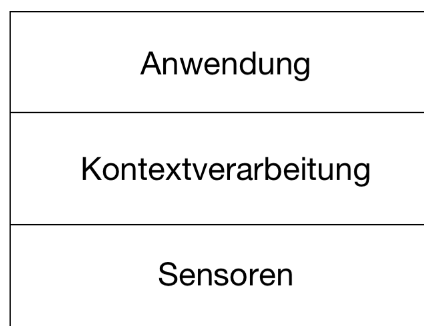


Abbildung 2.2.: Layer Architektur Konzept für kontextbewusste Anwendungen

- **Hydrogen** [Hofer u. a. (2003)]: Ein weiterer Ansatz der auf einer Layer Architektur basiert ist das Hydrogen Projekt, welches auf Context-Awareness für mobile Geräte spezialisiert ist. Auch hier ist die Gesamtarchitektur in drei Layer unterteilt, die in Abbildung 2.3 dargestellt sind. Der *Adaptor Layer* ist für die Abfrage der unverarbeiteten Kontextdaten von den verschiedenen Sensoren zuständig. Der zweite Layer ist der *Management Layer*. Dieser nutzt den Adaptor Layer, um Kontextinformationen zu erhalten und diese mit Hilfe des *Context Server* für die darüber liegenden Client Anwendungen zur Verfügung zu stellen. Im dritten Layer (*Application Layer*) sind die Client Anwendungen repräsentiert, die auf die vom Management Layer zur Verfügung gestellten Kontextinformationen reagieren können.

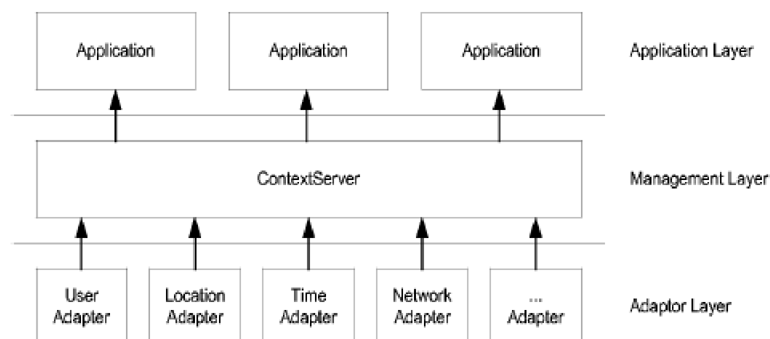


Abbildung 2.3.: Layer Architektur des Hydrogen Projekt [Baldauf u. a. (2007)]

- **Gaia** [Ranganathan und Campbell (2003)]: Gaia besteht ebenfalls aus einer Infrastruktur, die ähnlich wie in Abbildung 2.2 zu sehen, in drei Layer unterteilt werden kann. Auf dem untersten Layer befinden sich auch hier verschiedene Sensoren für die Erfassung von Kontextinformationen (*Context Provider*). Der mittlere Layer ist zuständig für das Management und die Verarbeitung der Informationen (*Context Synthesizer*), die letztendlich der Anwendungsschicht bereitgestellt werden (*Context Consumer*). Der Aufbau der Gaia-Infrastruktur ist in Abbildung 3.1 dargestellt. Gaia nutzt ebenfalls wie SOCAM ein ontologiebasiertes Kontextmodell zur Repräsentation von Kontextdaten. Zudem unterstützt Gaia das Reasoning (siehe Abschnitt 2.3) von Kontextinformationen auf Basis von Prädikatenlogik. In Gaia ist es außerdem möglich, Kontextmuster mit Hilfe von Regeln auf Basis von Aussagenlogik zu formulieren. Dadurch können automatisch spezifische Kontextsituationen erkannt werden, auf die die entsprechende Anwendung dann reagieren kann. Auf Gaia wird in Kapitel 3 ausführlicher eingegangen.

- **JCAF** [Bardram (2005b)]: Das *Java Context-Awareness Framework* (JCAF) stellt eine Java basierte Infrastruktur und einige APIs zur Entwicklung von kontextbewussten Anwendungen zur Verfügung. Es unterstützt die Möglichkeit sich per Callback über Kontextänderungen von Objekten informieren zu lassen, bietet jedoch keinerlei Reasoningfunktionen an. JCAF wird ebenfalls in Kapitel 3 genauer erläutert.

Eine Übersicht über weitere bestehende Arbeiten auf dem Gebiet der Context-Awareness bieten [Baldauf u. a. (2007)] und [Kumar und Xie (2012)].

Gaia und JCAF gehören dabei bereits zu den Ansätzen, die positive Eigenschaften und Funktionalitäten vorangegangener Arbeiten aufgreifen und diese zu einem eigenen verbesserten Ansatz weiterentwickeln. Aus diesem Grund werden primär diese beiden Arbeiten als Referenz für einen eigenes Context-Service Konzept herangezogen (siehe Kapitel 3). Die Konzepte von Gaia als auch von JCAF, verfügen jedoch trotzdem noch über einige Nachteile und ungelöste Probleme, die in 3.3 aufgezeigt und genauer erläutert werden.

2.1.3. Ansätze zur Kontextmodellierung

Für die Modellierung von Kontext wurden mittlerweile mehrere verschiedene Ansätze entwickelt, die jeweils ihre eigenen Konzepte für das Speichern, Verarbeiten und Managen von Kontextinformationen aufweisen. Sechs dieser Ansätze wurden von Thomas Strang und Claudia Linnhoff-Popien in ihrem Paper „*A Context Modeling Survey*“ [Strang und Linnhoff-Popien (2004)] anhand von sechs Anforderungskriterien evaluiert. Diese werden im Folgenden genannt:

- **Distributed Composition**: Betrachtet die Möglichkeit der Verteilung von Kontextinformationen in einem verteilten bzw. ubiquitären System.
- **Partial Validation**: Beschreibt die Anforderung jede Kontextinformation mit Hilfe des verwendeten Kontextmodells zu bewerten bzw. die Informationen gegen das Kontextmodell zu validieren.
- **Richness and Quality of Information**: Hier geht es um die Qualität und Fülle der Kontextinformationen von Sensoren, die das verwendete Kontextmodell bieten kann.
- **Incompleteness and Ambiguity**: Diese Anforderung untersucht, wie das Kontextmodell mit unvollständigen Informationen umgeht, falls zum Beispiel einmal ein Sensor keine Informationen mehr liefert. Das Modell könnte diese Information dann zum Beispiel aus älteren Daten interpolieren oder eine andere Art der Fehlerbehandlung durchführen.

- **Level of Formality**: Befasst sich mit der formalen Repräsentation von Kontextinformationen in dem jeweiligen Kontextmodell. Dies ist zum Beispiel wichtig, damit alle Parteien des Systems ein einheitliches Verständnis der Informationen haben und diese entsprechend interpretieren können.

- **Applicability to Existing Environments**: Hier wird untersucht wie integrationsfähig das jeweilige Kontextmodell ist bzw. wie einfach es sich in bestehende Systeme und Umgebungen integrieren lässt.

Das Ergebnis der Analyse und Bewertung von Strang und Linnhoff-Popien, für die allgemeinen Ansätze zur Modellierung von Kontext, ist in Abbildung 2.4 zu sehen.

Ansatz | Anforderung	dc	pv	qua	inc	for	app
Key-Value Modelle	-	-	--	--	--	+
Markup Schema Modelle	+	++	-	-	+	++
Graphische Modelle	--	-	+	-	+	+
Objektorientierte Modelle	++	+	+	+	+	+
Logikbasierte Modelle	++	-	-	-	++	--
Ontologiebasierte Modelle	++	++	+	+	++	+

Abbildung 2.4.: Ergebnis der Analyse von [Strang und Linnhoff-Popien (2004)]. Abkürzungen: dc = Distributed Composition, pv = Partial Validation, qua = Richness and Quality of Information, inc = Incompleteness and Ambiguity, for = Level of Formality, app = Applicability to Existing Environments

Laut des Ergebnisses sind also die ontologiebasierten sowie die objektorientierten Ansätze zur Modellierung von Kontext am besten mit den Anforderungen zu vereinbaren. Aus diesem Grund wurde jeweils ein konkretes Beispiel dieser Ansätze näher untersucht (siehe Abschnitt 3.1 und 3.2). Strang und Linnhoff-Popien betonen jedoch, dass die anderen Ansätze deshalb aber nicht ungeeignet für die Modellierung von Kontext seien und es auf die jeweilige Zielanwendung ankomme, welches Modell Verwendung findet.

2.2. Multiagentensysteme und Multiagentensimulationen

Jennings und Wooldridge definieren den Begriff des Agenten wie folgt:

> "An agent is an encapsulated computer system that is situated in some environment and that is capable of flexible, autonomous action in that environment in order to meet its design objectives." [Jennings und Wooldridge (2000)]

Ein Agent ist also ein gekapseltes Computersystem, welches in einer beliebigen Umgebung arbeitet und in der Lage ist darin flexibel und autonom zu agieren, um seine angestrebten Ziele zu erreichen. Die innere Struktur von Agenten, kann zum Beispiel durch das BDI-Modell [Schönmann (2012)] dargestellt werden, wobei BDI für Beliefs-Desires-Intentions steht. *Beliefs* ist die Wahrnehmung der Welt durch den Agenten oder auch seine Vorstellung wie seine Welt zu sein scheint. Die *Desires* beschreiben die Wünsche und Ziele des Agenten und die *Intentions* sind die momentanen Absichten und Pläne des Agenten. Das BDI-Modell ist in Abbildung 2.5 dargestellt. Agenten können als Softwareagenten oder als Roboter zum Einsatz kommen, wie später folgende Beispiele zeigen.

Eine weitere und aktuellere Möglichkeit für die Realisierung von Agentenlogik ist das Goal-Oriented Action Planning (*GOAP*) [Orkin (2002)]. GOAP basiert auf einer von Jeff Orkin modifizierten Form des Stanford Research Institute Problem Solver (*STRIPS*) [Russell und Norvig (2010)] und wird unter anderem für künstliche Intelligenz in Computerspielen eingesetzt. Ein GOAP-Agent agiert zielorientiert (goal-oriented) und plant seine Aktionen anhand seiner jeweiligen Ziele (action planning). Wichtige Begriffe bei GOAP sind *Goal*, *Action* und *Plan*. Ein *Goal* ist eine Bedingung bzw. ein Zustand, den der Agent erfüllen möchte. Zu jedem Zeitpunkt verfolgt der Agent ein bestimmtes Ziel, welches sein Verhalten steuert. Eine *Action* ist ein einzelner atomarer Schritt innerhalb eines Plans, den ein Agent ausführen kann. Ein *Plan* ist wiederum eine Sequenz von Aktionen, die den Agenten von einem Startzustand in einen Zustand der das Goal erfüllt überführt.

Ein Zusammenschluss von Agenten bzw. ein System in dem mehrere Agenten agieren und interagieren stellt ein sogenanntes *Multiagentensystem* dar. Das Ziel eines Multiagentensystems ist das verteilte lösen eines Problems. Dabei können die beteiligten Agenten gemeinsam auf ein Ziel hinarbeiten aber auch gegeneinander. Multiagentensysteme kommen hauptsächlich in Simulationen und KI-Systemen zum Einsatz. Der Zusammenhang zur Multiagentensimulation entsteht dadurch, dass jede Person, jedes Tier oder auch jede Pflanze in einer Simulation, durch einen Softwareagenten eines Multiagentensystems simuliert werden kann.

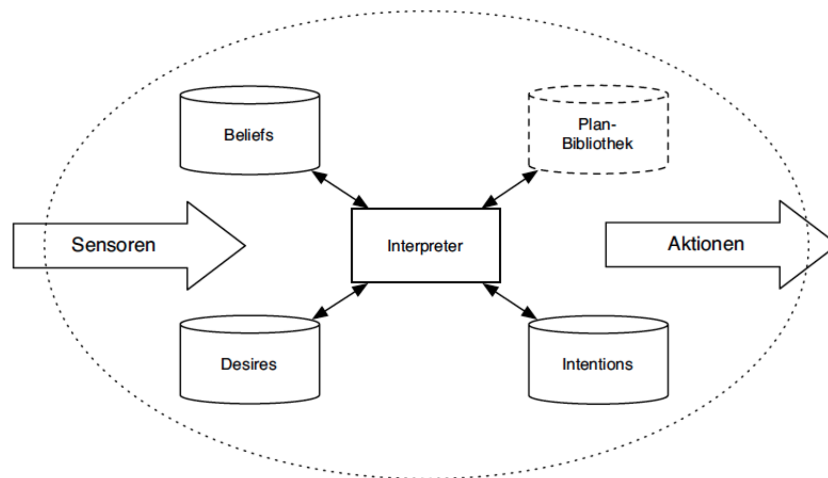


Abbildung 2.5.: BDI-Modell eines Softwareagenten [Schönmann (2012)]

2.2.1. Beispiele für Multiagentensysteme

Im Folgenden werden einige Beispiele für Multiagentensysteme genannt, die aktuell relevant sind. Dazu gehören unter anderem:

- **MARS**[1]: MARS [Hüning u. a. (2014)] ist ein Projekt der *Multi-Agent Research & Simulation* Forschungsgruppe[2] an der Hochschule für angewandte Wissenschaften Hamburg
- **Repast Simphony**[3]: eine umfangreiche Modellierungs- und Simulationsplattform für Multiagentensysteme. Repast Simphony ist außerdem ein Open Source Projekt und steht frei zur Verfügung
- **WALK**: WALK ist ein skalierbares Multiagentensystem, welches sich stark auf Evakuierungsszenarios fokussiert und ebenfalls an der Hochschule für angewandte Wissenschaften Hamburg entwicklet wurde [Thiel (2013); Thiel-Clemen u. a. (2011)]
- **SeSAm (Shell for Simulated Agent Systems)**[4]: ein freies Open Source Tool zur Erstellung komplexer agentenbasierter Modelle und Simulationen

[1] http://mars-group.org/
[2] http://mars-group.org/index.php/members
[3] http://repast.sourceforge.net/repast_simphony.php
[4] http://130.243.124.21/sesam/

2.2.2. Beispiele von Anwendungsszenarien

Bekannte Anwendungsszenarien von Multiagentensystemen sind zum Beispiel:

- **Ausbreitung von Krankheiten**: Mit Hilfe von Multiagentensystemen ist es möglich, die temporale und spatiale Ausbreitung von Krankheiten zu simulieren. Es können zum Beispiel epidemiologische Gefahren, bei der Ausbreitung von Krankheiten in öffentlichen Verkehrsmitteln, anhand von entsprechenden Simulationsmodellen untersucht werden. Dabei werden die reisenden Personen durch Agenten simuliert, die ein möglichst realitätsgetreues Verhalten von Personen in öffentlichen Verkehrsmitteln abbilden. Dadurch kann die Ausbreitung einer Krankheit in Metropolregionen dargestellt und ihr Gefahrenpotenzial eingeschätzt werden. [Noetzel u. a. (2013)]

- **Entfluchtung von Fußgängern**: Ein weiteres Anwendungsszenario ist, die Simulation einer Entfluchtung von Fußgängern aus Gebäuden oder Gefahrengebieten. Dazu können Simulationsmodelle entwickelt werden, welche das Fluchtverhalten von Fußgängern in Paniksituationen abbilden. Mit Hilfe einer solchen Simulation ist es dann zum Beispiel möglich, Schwachstellen in Fluchtplänen zu identifizieren oder gefährliche Areale und Hindernisse in Gebäuden ausmachen zu können. [Thiel-Clemen u. a. (2011)]

- **Entwicklung von Biomasse**: Die Simulation der Entwicklung von Waldbiomasse über lange Zeiträume, ist ebenfalls ein Anwendungsszenario für Multiagentensysteme. Unter Berücksichtigung von verschiedenen Parametern aus der realen Welt, die Veränderungen in Waldgebieten verursachen, kann die Entwicklung der Biomasse in den entsprechenden Wäldern untersucht werden. Dies könnte zum Beispiel Aufschluss darüber geben, ob und wie stark sich bestimmte Eingriffe des Menschen in Waldgebiete auf die Bilanz der Biomasse in diesen Gebieten auswirken. [Pereki u. a. (2013); Baldowski u. a. (2014)]

2.3. Context Reasoning

Context Reasoning beschreibt den Vorgang der Deduktion, also der Schlussfolgerung, von für die Anwendung relevanten Informationen aus verschiedenen Kontextinformationen. Dabei werden sogenannte *low-level* Informationen, zum Beispiel durch Aggregation, auf eine höhere Hierarchiebene (*high-level*) abgebildet. Beispielsweise können GPS-Signale als Längen- und Breitengrad vorliegen und dann durch Reasoning auf Orte wie Zuhause oder Arbeitsplatz abgebildet werden (siehe Abbildung 2.6).

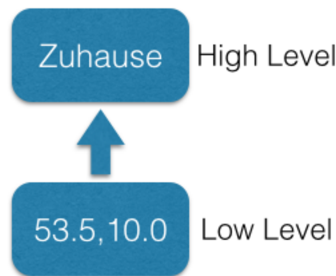


Abbildung 2.6.: Low- und High-Level Kontexte

2.4. Event-Driven Architectures und Complex Event Processing

Event-Driven Architectures und Complex Event Processing hängen grundsätzlich miteinander zusammen und bilden ein eigenes und relativ junges Fachgebiet aus dem Bereich des *Event Processing*, also der Ereignisverarbeitung. Der Begriff Event-Driven Architecture (EDA) beschreibt so gesehen das softwaretechnische Konzept, bei dem Ereignisse bzw. sogenannte *Events* im Mittelpunkt der Informationsverarbeitung stehen.

Complex Event Processing (CEP) stellt die Technologie dar, die in einer EDA-Anwendung zum Einsatz kommen kann, um jegliche Art von auftretenden Ereignissen dynamisch und in Echtzeit zu erkennen, zu verarbeiten und auf diese zu reagieren. Im Jahr 2008 wurde die *Event Processing Technical Society* (EPTS) [EPTS (2013)] gegründet, die sich thematisch auf das Gebiet der Ereignisverarbeitung spezialisiert hat. Die EPTS hat es sich unter anderem zur Aufgabe gemacht, Begriffe aus dem Gebiet der EDA und des CEP zu definieren und in einem Glossar zusammenzufassen, welches aktuell in der Version 2.0 vorliegt [Luckham und Schulte (2013)]. Dies soll vor allem der Standardisierung der Begriffe dienen.

2.5. Grundlagen Event-Driven Architectures

Ein großes Einsatzgebiet ereignisgesteuerter Architekturen für Anwendungen liegt heute vor allem in dem Bereich der Unternehmensanwendungen. Typische Domänen sind hier zum Beispiel Finanzunternehmen oder Logistikunternehmen, da die Geschäftsprozesse dieser Branchen sehr stark von internen und externen Ereignissen abhängig sind. Zum Beispiel sind folgende Ereignisse für ein Logistikunternehmen von besonderem Interesse:

- Auf der Route von Lastwagen a ist aktuell ein Verkehrsstau.
- Artikel b ist nicht mehr ausreichend im Lager vorrätig.

- Kunde xy hat seine Bestellung storniert.

Das Unternehmen möchte nun in der Lage sein, zeitnah auf diese Ereignisse zu reagieren und entsprechende Aktionen automatisch ausführen:

- Lastwagen a wird über den Verkehrsstau informiert und bekommt eine neue Route vorgeschlagen.
- Artikel b wird in ausreichender Stückzahl nachbestellt.
- Die Bestellung von Kunde xy wird nicht weiter bearbeitet und gegebenenfalls bereits abgebuchtes Geld wird zurücküberwiesen.

Diese Beispiele von Ereignissen und Reaktionen zeigen den Bedarf und den möglichen Nutzen von ereignisorientierten Anwendungsarchitekturen für entsprechende Domänen. Auch in Multiagentensimulationen finden in jedem Simulationstick Ereignisse statt, die erkannt werden müssen, um entsprechende Reaktionen anzustoßen.

2.5.1. Was sind Ereignisse

Prinzipiell kann jeder Vorgang in der realen Welt oder jede Zustandsänderung in einem System als ein Ereignis (*Event*) angesehen werden. Weitere explizite Beispiele für Ereignisse neben den eben angeführten wären folgende: Ein Flugzeug landet am Hamburger Flughafen, ein Wettersensor meldet eine Temperaturveränderung, es werden mit einer Kreditkarte 500 Euro von einem Konto abgebucht, ein Musikalbum wird online heruntergeladen. All diese Ereignisse können einen Einfluss auf den Ablauf eines Prozesses nehmen und sich somit auf den weiteren Verlauf auswirken. Es ist jedoch nicht immer vorausgesetzt, dass dies auch tatsächlich passiert. Ein eintretendes Ereignis kann auch ignoriert werden, wenn es momentan als nicht relevant angesehen wird. Im EPTS *Event Processing Glossary 2.0* [Luckham und Schulte (2013)] von David Luckham und W. Roy Schulte wird ein *Event* definiert als:

> „Anything that happens, or is contemplated as happening.“

Alles was passiert oder eine Zustandsänderung impliziert, kann also als ein Ereignis angesehen werden. Diese Definition von Ereignissen ist gerade hinsichtlich einer Simulation komplexer Abläufe in der realen Welt sinnvoll. Beispielsweise ist bei einer Simulation jede Positionsveränderung eines Agenten eine relevante Information und sollte deshalb als Ereignis behandelt werden.

2.5.2. Vergleich mit herkömmlichen Architekturen

Dieser Abschnitt soll einige Abgrenzungen zu herkömmlichen Softwarearchitekturen aufzeigen die Ereignisse verarbeiten. Die folgenden Punkte beschreiben Eigenschaften in denen sich EDA-Anwendungen von herkömmlichen Anwendungen unterscheiden:

- **Ereignisorientierung als Basis der Softwarearchitektur**: Beispiele hierfür bei herkömmlichen Architekturen sind vor allem Anwendungen aus der Netzwerktechnik oder grafische Benutzeroberflächen. Diese registrieren Ereignisse und verarbeiten diese zum Beispiel, wenn ein Benutzer auf eine Schaltfläche klickt. Jedoch stellt die Ereignisorientierung bei diesen Anwendungen nicht den Kern der Softwarearchitektur dar, wie es bei EDA-Anwendungen der Fall ist.

- **Echtzeitfähigkeit**: Informationen des Systems werden bei herkömmlichen Anwendungen zunächst gespeichert und können nur nachträglich abgefragt und analysiert werden. Zum Beispiel „Wie viele Bestellungen hat Kunde xy im letzten Monat getätigt?“. Eine EDA-Anwendung ermöglicht hingegen eine aktuelle Betrachtung der zur Verfügung stehenden Informationen, wie zum Beispiel „Wo befindet sich das Paket der Bestellung 42 von Kunde xy genau in diesem Moment?“. Diese Eigenschaft ist für eine automatische Steuerung eines Prozesses bzw. einer Anwendung (z. B. einer Simulation) in Echtzeit essentiell. Eine nachträgliche Analyse der Daten wird von EDA-Anwendungen natürlich ebenfalls unterstützt.

- **Prozesse und Prozeduren**: Bei einer herkömmlichen Anwendung sind Prozesse und Prozeduren ablauforientiert. Das heißt, dass sie nach festen Regeln und im Grunde auch immer in der gleichen Reihenfolge ablaufen und dabei nacheinander einzelne Informationen im Sinne von Ereignissen verarbeiten. Die spontane Reaktion auf ein eintretendes Ereignis oder sogar mehrere Ereignisse als Ereignismuster liegt hier nicht im Fokus. Daher sind auch keine Erkennung von Abhängigkeiten zwischen Ereignissen, sowie eine Abstraktion mehrerer in Zusammenhang stehender Ereignisse zu einem komplexen Ereignis möglich. Bei EDA-Anwendungen sind die Prozesse und Prozeduren hingegen ereignisorientiert. Es kann also auf spontan eintretende Ereignisse reagiert werden, woraus ein veränderter Ablauf des Prozesses resultieren kann. Dadurch können reale Situationen und Simulationsszenarios deutlich besser modelliert und abgebildet werden. Zusätzlich können Informationen von einzelnen Ereignissen aus ggf. heterogenen Quellen, auf mehrere Ebenen mit unterschiedlich hohem Detailgrad abstrahiert werden.

- **Grad der Kopplung**: Bei herkömmlichen Architekturen werden Dienste meist über ihre Signatur und mit den entsprechenden Parametern aufgerufen. Oft sind darüber hinaus noch weitere Kenntnisse über den Dienst nötig, um ihn korrekt nutzen zu können. Die Kommunikation zwischen der aufrufenden und der aufgerufenen Komponente findet dabei normalerweise synchron per Request-Response statt. Diese Faktoren verursachen eine eher starke Kopplung der Komponenten. In EDA-Anwendungen kommunizieren die Komponenten dagegen meist über Nachrichten, deren Struktur allen Parteien bekannt ist. Diese Nachrichten werden dann von einer Middleware weiterverarbeitet. Es wird also lediglich die Kenntnis der Nachrichtenstruktur für eine Kommunikation vorausgesetzt. Die Nachrichten werden zudem in der Regel asynchron und per Publish-Subscribe verarbeitet. Insgesamt entsteht dadurch eine losere Kopplung der Komponenten.

In Anwendungen mit einer Event-Driven Architecture ist daher die Ereignisverarbeitung ein essentieller Grundstein, der meist von einer speziell dafür konzipierten Komponente des Systems übernommen wird. Eine zur Verarbeitung von hohen Volumen an gleichzeitig eintretenden Ereignissen entwickelte Technologie, ist das sogenannte Complex Event Processing, welches im folgenden Abschnitt erläutert wird.

2.6. Complex Event Processing

Complex Event Processing beschreibt die Technologie, die verwendet wird, um eine Event-Driven Architecture zu realisieren. Der Begriff des Complex Event Processing, sowie das dahinterstehende Konzept wurde maßgeblich durch David Luckham geprägt. Mit seinem Buch *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems* von 2001 [Luckham (2001)] hat Luckham den Grundstein für Ereignisorientierung als neues Forschungsgebiet gelegt. Er ist außerdem aktuell an der Erstellung des *Event Processing Glossary* der EPTS beteiligt[5].

2.6.1. Definition des Begriffs

Auf der Complex Event Processing Webseite[6] von Luckham wird CEP wie folgt definiert:

> “Complex Event Processing (CEP) is a technology for building and managing information systems including:
>
> - Business Activity Monitoring

[5]http://www.complexevents.com/2011/08/23/event-processing-glossary-version-2-0/
[6]http://www.complexevents.com/event-processing/

- Business Process Management
- Enterprise Application Integration
- Event-Driven Architectures
- Network and business level Security
- Real time conformance to regulations and policies.

> [...] CEP employs techniques such as detection of complex patterns of many events, event streams processing, event correlation and abstraction, event hierarchies, and relationships between events such as causality, membership, and timing. CEP can complement and contribute to technologies such as service oriented architecture (SOA), event driven architecture (EDA) and business process management (BPM)."
> [ComplexEvents (2013)]

Complex Event Processing stellt also Techniken, wie die Erkennung komplexer Muster mehrerer Ereignisse, Event Stream Processing, Ereigniskorrelation und -abstraktion, Ereignishierarchien und Beziehungen zwischen Ereignissen, wie zum Beispiel kausale oder zeitliche Beziehungen bereit. Damit stellt Complex Event Processing eine scheinbar optimale Technologie, für die Verarbeitung und das Erkennen von Zusammenhängen zwischen Ereignissen in Multiagentensimulationen dar.

2.6.2. Grundlegende Konzepte und Eigenschaften

Complex Event Processing ist ein zentraler Bestandteil einer Event-Driven Architecture und dreht die konventionelle Art der Datenverarbeitung, wie man sie sonst von gewöhnlichen Datenbankanwendungen kennt, um. Dort sind die Daten in einer Datenbank gespeichert und können mit Hilfe von Skriptsprachen wie SQL abgefragt werden. Die dynamische Anfrage wird gegen die Daten geschickt. Bei Complex Event Processing sind dagegen die Anfragen persistent. Sie bestehen aus vordefinierten Regeln, über die kontinuierlich alle eintreffenden Daten geschickt werden [GICEP (2013)] (siehe Abbildung 2.7). Die performante kontinuierliche Prüfung der Datenströme gegen die gespeicherten Regeln, wird durch eine sogenannte *In-Memory Computing*-Technik[7] ermöglicht. Dabei werden die zu analysierenden Daten im Hauptspeicher anstatt in langsameren Datenbanken oder Festplatten gehalten, wodurch die Zugriffs- und Verarbeitungsgeschwindigkeit signifikant verbessert wird. Diese Technik wird mittlerweile in sehr großen Big-Data-Anwendungen, wie dem SAP HANA Projekt [8] eingesetzt.

[7] http://www.techopedia.com/definition/28539/in-memory-computing
[8] http://www.sap.com/germany/pc/tech/in-memory-computing-hana.html

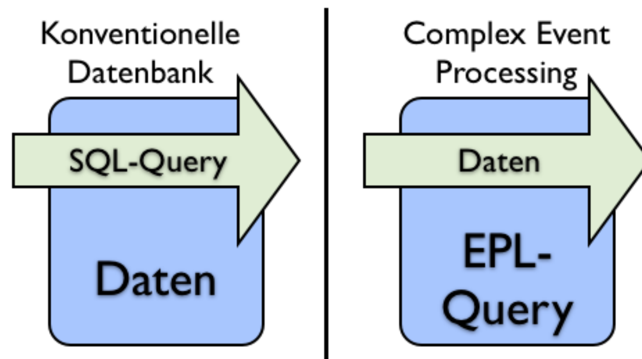


Abbildung 2.7.: Konventionelle Abfrage von persistenten Daten gegenüber kontinuierlichem Event Processing von Datenströmen gegen persistente Ereignismuster

Das Complex Event Processing ist letztendlich vor allem für die folgenden Eigenschaften und Aufgaben einer Event-Driven Architecture maßgeblich:

- Echtzeitfähigkeit durch die kontinuierliche Analyse der eingehenden Datenströme aus ggf. mehreren heterogenen Datenquellen und ihrer Verarbeitung. Das gleichzeitig zu verarbeitende Volumen an Daten kann dabei durchaus sehr groß werden.
- Erkennung von Mustern im Ereignisstrom (*Event pattern matching*) und Abstraktion der Muster zu komplexen Ereignissen. Dadurch können Kausalitäten und Zusammenhänge (siehe Beispiele aus Abschnitt 3.1.2 und 2.6.3) zwischen den Ereignissen festgestellt und ggf. weitere Informationen deduziert werden. Die Kausalitäten und Zusammenhänge zwischen Ereignissen können unter anderem zum Beispiel spatialer oder temporaler Natur sein. Mögliche Konstellationen temporaler Korrelationen zwischen Ereignissen sind in Abbildung 2.8 dargestellt.
- Deklaration und Persistierung von Regeln mit denen gesuchte bzw. relevante Ereignismuster beschrieben werden können. Damit verbunden ist das Festlegen der entsprechenden Aktion die ausgeführt werden soll, wenn das Ereignismuster erkannt wurde.

Diese Eigenschaften wären für die Verarbeitung von Kontextdaten einer Multiagentensimulation durchaus Vorteilhaft. Echtzeitfähigkeit und die Möglichkeit der Erkennung von Ereignismustern und Abstraktion dieser Muster zu komplexeren Ereignissen, sowie die einfache Deklaration von Kontextregeln sind wichtige Punkte, die eine Context-Service Komponente erfüllen sollte.

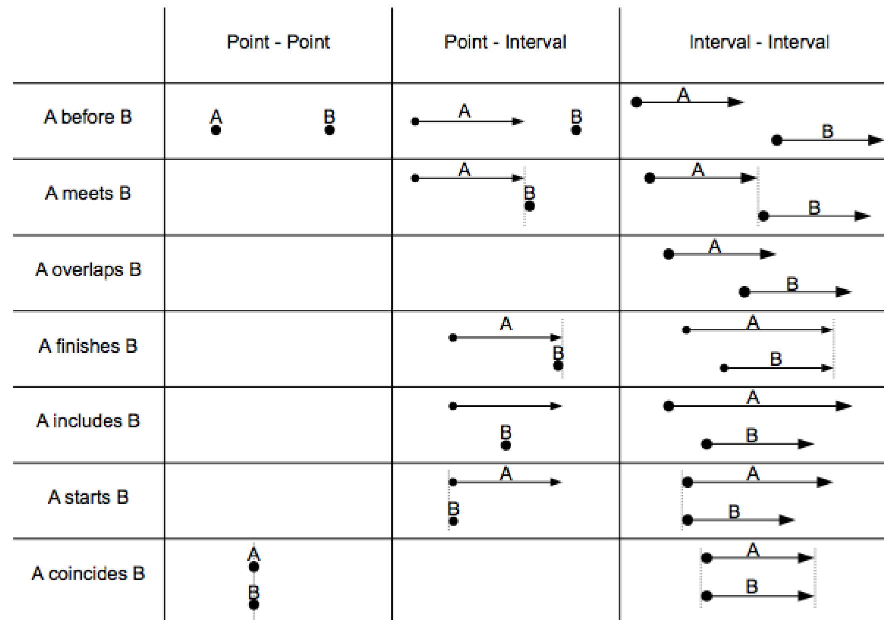


Abbildung 2.8.: Mögliche temporale Korrelationen und Beziehungen zwischen Ereignissen nach [DroolsFusion (2013)]

2.6.3. Komplexe Ereignisse

In Luckham und Schulte (2013) wird ein komplexes Ereignis wie folgt definiert:

> „An event that summarizes, represents, or denotes a set of other events."

Die Ereignisse werden also in einen gemeinsamen *Kontext* gebracht. Ein einzelnes Ereignis für sich hat in den meisten Fällen keinen ausreichenden Informationsgehalt, um daraus einen Kontext bzw. eine bestimmte Situation ableiten zu können. Wenn zum Beispiel ein vermeintlicher Kunde versucht, sich in sein Kundenkonto einzuloggen und dies fehlschlägt, hat man ein Ereignis das für sich alleine betrachtet noch keine auffällige Situation darstellt. Werden aber mehrere fehlerhafte Loginversuche hintereinander registriert und diese miteinander in *temporale* (also zeitliche) Korrelation gebracht, dann könnte man auf einen möglichen Missbrauchsversuch schließen. Wird versucht sich mit den Kundendaten an mehreren verschiedenen Standorten gleichzeitig anzumelden, lässt jedes Loginereignis einzelnd betrachtet keine weiteren Rückschlüsse zu. Bringt man all diese Ereignisse jedoch wieder in eine temporale und zusätzlich in eine *spatiale* (also räumliche) Korrelation, lässt sich wieder ein Missbrauchsversuch vermuten. Durch die Korrelation zwischen entsprechenden singulären Ereignissen, entsteht dann ein sogenanntes *komplexes Ereignis* (*complex event*).

2.6.4. Ereignismuster

Das Erkennen von Abhängigkeiten oder Beziehungen zwischen einzelnen Ereignissen spiegelt ein sogenanntes Ereignismuster (*event pattern*) wieder. Um diese relevanten Muster aus mehreren einzelnen Ereignissen erkennen zu können, ist es notwendig den Ereignisfluss aus ggf. heterogenen Datenquellen über einen bestimmten Zeitraum hinweg zu überwachen und zu untersuchen. Wird ein Ereignismuster erkannt, wird es zu einem komplexen Ereignis abstrahiert. Viele Ereignisse mit relativ feingranularen Informationen können dadurch auf einen höheren Abstraktionsgrad gebracht werden (vgl. 3.1.2).

Abbildung 2.9 zeigt einen Informationsfluss der zu analysierenden Daten. Darin können Ereignismuster erkannt und auf einer höheren Abstraktionsebene als komplexe Ereignisse zusammengefasst werden. Diese Abstraktionsebenen fassen einfache Informationen der

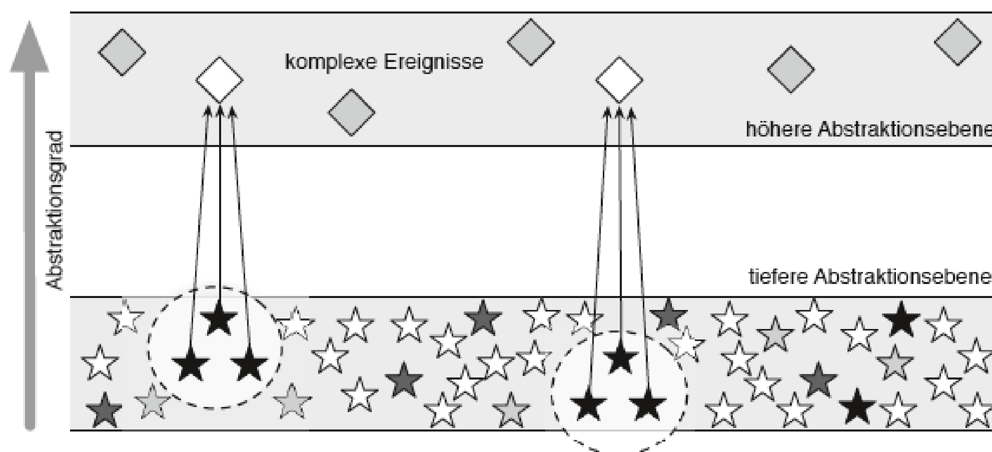


Abbildung 2.9.: Mustererkennung im Ereignisstrom (Event Stream) und Abstraktion zu komplexen Ereignissen auf einer höheren Abstraktionsebene [Bruns und Dunkel (2010)]

untersten Ebene von *low level* Ereignissen auf jeder höheren Ebene immer weiter zusammen. Dadurch entstehen dann *high level* Ereignisse sowie eine Abstraktionshierarchie, in der die Ereignisse umso aussagekräftiger werden, je höher sie sich in der Hierarchie befinden. So könnten die drei Ereignisse Flugzeug ist vollgetankt, Passagiere sind an Board und Startbahn ist frei zu einem komplexen Ereignis Flugzeug kann starten aggregiert werden.

Mit Hilfe der Ereignismuster ist darüber hinaus auch die Möglichkeit gegeben, kausale Beziehungen in die umgekehrte Richtung zu analysieren. So kann zum Beispiel der Grund des

Ausbleibens eines komplexen Ereignisses nachvollzogen werden. Im vorangegangenen Flugzeug kann starten Ereignismuster, kann das Flugzeug schon vollgetankt und die Startbahn frei aber die Passagiere noch nicht alle an Board sein. Bei einer Analyse kann das Ausbleiben des komplexen Ereignisses darauf zurückgeführt werden. Letztendlich ermöglicht das Erkennen eines Ereignismusters die automatische Benachrichtigung eines Benutzers, der dann weitere Aktionen einleitet oder die automatische Ausführung eines Prozeduraufrufs im System.

2.6.5. Zentrale Bestandteile und weitere Begriffe

Eine Complex Event Processing-Komponente besteht aus mehreren Bestandteilen, die im EPTS Glossary zu einem *Event Processing Agent* (EPA) [Luckham und Schulte (2013)] zusammengefasst und wie folgt definiert werden:

1. "**Event type (event class, event definition, or event schema)**: An event type is a class of event objects."
2. "**Rule (in event processing)**: A prescribed method for processing events."
3. "**Complex-event processing (CEP)**: Computing that performs operations on complex events, including reading, creating, transforming or abstracting them."

Der Ereignistyp (*Event Type*) bzw. das Ereignisschema, beschreibt den Inhalt oder auch die Struktur der Ereignisse, die als Nachricht zwischen Ereignisquellen und -senken ausgetauscht werden. Darin können zum Beispiel unter anderem die Quelle, eine Nachrichten-ID und ein Zeitstempel angegeben werden.

Die Ereignisregeln (*Rule*) können mit Hilfe sogenannter *Event Pattern Languages* (EPL) definiert werden, wobei so gut wie jedes auf dem Markt verfügbare CEP-Produkt seine eigene EPL besitzt. David Luckham definiert den Begriff der Event Pattern Language wie folgt:

> "An event pattern language is a computer language in which we can precisely describe patterns of events. It is similar to mathematical language for logical expressions or a Web search language with more than the usual options and some other bells and whistles. It lets us describe, without any ambiguity, exactly the patterns in which we are interested." [Luckham (2001)]

Die Regeln können dann zum Beispiel in Form einer "Wenn → Dann"-Semantik repräsentiert werden. Die Prozedur die ausgeführt werden soll, wenn ein Ereignis oder ein komplexes

Ereignis eintritt und erkannt wird, kann also jeweils direkt festgelegt werden. Das folgende Beispiel soll dies anhand von Pseudocode nach [DroolsFusion (2013)] verdeutlichen:

```
1 rule "Fire sprinkler"
2 when
3 fireDetected( ) AND peopleInBuilding( ) > 10
4 then
5 activateSprinklers( )
6 activateAlarmSound( )
```

Das Complex Event Processing, welches die definierten Regeln dann mit den eingehenden Datenströmen abgleicht, wird von einer CEP-Engine - der dritten Teilkomponente eines Event Processing Agents - übernommen. Sie verwendet die gespeicherten Regeln, um Ereignismuster zu erkennen und dann den Aktionsteil derjenigen Regel auszuführen, dessen Ereignismuster erkannt wurde. Mittlerweile sind mehrere CEP-Engines mit einem unterschiedlichen Spektrum an Features und mit unterschiedlichem Fokus auf dem Markt erhältlich. Zu den wenigen frei verfügbaren Open Source Lösungen gehört zum Beispiel die CEP-Engine *Esper*[9].

2.7. Paradigmen

Dieser Abschnitt soll nun abschließend für dieses Kapitel die wichtigsten Merkmale der zwei Computing Paradigmen erläutern, die in dieser Arbeit gegenüber gestellt werden. Dabei werden für jedes Paradigma die grundlegenden Eigenschaften und ihre jeweiligen Einsatzgebiete genannt, für die diese Paradigmen empfehlenswert oder gar prädestiniert sind. Für beide Paradigmen werden jeweils ein paar Beispielanwendungen angeführt.

2.7.1. Responsive Computing

Das Paradigma des Responsive Computing repräsentiert die bekannteste Art der Anwendungsarchitektur, in der traditionell synchrone Request-Response Interaktionen verwendet werden. Typische Anwendungen, die das Responsive Computing Paradigma verfolgen, sind zum Beispiel:

- **Graphical User Interface (GUI)**: Die grafische Benutzeroberfläche von Programmen, bei der ein Benutzer auf eine Schaltfläche klicken oder Benutzereingaben durchführen kann, ist das deutlichste Beispiel für Responsive Computing. Der Benutzer sendet in diesem Fall durch einen Klick auf eine Schaltfläche eine explizite Anfrage (Request) und bekommt meist auch eine explizite Antwort (Response) von der Anwendung.

[9] http://esper.codehaus.org/

- **Webserver**: Ein Webserver verarbeitet ebenfalls explizite Anfragen, die primär von Benutzern stammen und reagiert mit einer Antwort. Zum Beispiel fragt ein Benutzer eine Webseite über seinen Browser an. Der Webserver antwortet auf diese Anfrage mit dem entsprechenden Webinhalt.
- **Datenbanken**: Ein typischer Anwendungsfall für Datenbanken ist das speichern von Informationen, die zu bestimmten Zeitpunkten durch Anfragen von Benutzern abgerufen werden. Auch hier gibt es dann wieder jeweils explizite Anfragen die angestoßen werden und explizite Antworten als Ergebnis.

Weitere Eigenschaften von Responsive Computing Systemen sind meist:

- **Starke Kopplung**: Die Systemkomponenten sind stark voneinander abhängig. Dadurch lässt sich das System nur aufwändig Erweitern und Warten, da die Änderungen in einer Komponente meist starke Auswirkungen auf die wiederum von ihr anhängigen Systemkomponenten haben.
- **Synchrone Kommunikation**: Das System arbeitet nach dem Request-Response Prinzip. Dabei synchronisieren die Komponenten und Prozesse beim Senden und Empfangen von Daten. Bis ein Prozess eine Antwort (Response) auf seine Anfrage (Request) erhalten hat, blockiert er. Das bedeutet, dass der Prozess so lange nicht weiterarbeitet, bis er eine Antwort erhalten hat und die Kommunikation abgeschlossen ist. Durch synchrone Kommunikation ist die Latenzzeit höher und der Durchsatz des Systems niedriger, als bei nicht blockierender asynchroner Kommunikation.
- **Niedrige Modularität**: Ein Austausch einer Systemkomponente kann nicht ohne weiteres durchgeführt werden. Meist entsteht dadurch ein hoher Aufwand die anderen Systemkomponenten an den Austausch anzupassen, damit das System wieder voll funktionsfähig ist. Das System ist dadurch insgesamt relativ unflexibel gegenüber Änderungen.

2.7.2. Reactive Computing

Das Paradigma des Reactive Computing, baut auf das Konzept der Event-Driven Architecture auf und unterscheidet sich daher in den im Abschnitt 2.5.2 beschriebenen Punkten, von Responsive Computing Anwendungen. Eine Event-Driven Architecture kann also als Vehikel für Reactive Computing angesehen werden. Ein System, welches das Reactive Computing Paradigma verfolgt, kann durch die folgenden vier Eigenschaften charakterisiert werden [EPTS (2013)]:

- **Event-Driven**: Das System arbeitet ereignisgesteuert. Es basiert auf einer asynchronen Kommunikation und implementiert ein lose gekoppeltes Design. Die enthaltenen Komponenten interagieren miteinander über das Generieren und Konsumieren von Nachrichten.
- **Scalable**: Das System ist skalierbar, d. h. dass es sich einfach erweitern und aufrüsten lässt. Dies kann zum Beispiel durch horizontale Skalierung erfolgen bei dem je nach Belastung des Systems mehr Instanzen einer Komponente hinzugefügt werden, um so den Durchsatz bzw. die allgemeine Performance zu erhöhen.
- **Resilient**: Das System sollte belastbar sein, um Fehlverhalten und Ausfälle des Systems zu vermeiden.
- **Responsive**: An dieser Stelle steht *Responsive* für schnelles Antwortverhalten. Das System sollte möglichst echtzeitfähig sein und schnelle Reaktionszeiten ermöglichen. Dies wird unter anderem durch die Verwendung von Event Streams (siehe Abbildung 2.9) und asynchrone Kommunikation erreicht.

2.7.3. Zusammenfassung

Yagil Engel und Opher Etzion beschreiben den Unterschied zwischen den beiden Paradigmen zusammenfassend wie folgt:

> „Event driven architectures and conceptual models that support them have evolved in the last several years, departing from the traditional computing architectures which employ synchronous, request-response interactions between client and servers. This is a paradigm shift in two senses: first, event driven architectures support applications that are reactive in nature, in which processing is triggered in response to events, contrary to traditional responsive applications, in which processing is done in response to an explicit request. Second, event driven architecture adhere to the decoupling principle, in which there are event producers, event consumers and event processing agents that are mutually independent.“ [Engel und Etzion (2011)]

Ein System nach dem Responsive Computing Paradigma reagiert also jeweils auf eine explizite Anfrage (Request) mit einer expliziten Antwort (Response). In einem System nach dem Reactive Computing Paradigma lösen hingegen Ereignisse Prozesse aus, ohne Antwort auf eine explizite Anfrage zu sein, was zu einem Paradigmenwechsel führt. Die Ereignisverarbeitung (*Event Processing*) ist dabei das Vehikel für das reaktive Computing.

3. Verwandte Arbeiten

Dieses Kapitel betrachtet und diskutiert verwandte Arbeiten auf dem Gebiet der Kontexterkennung und der Context-Awareness von Anwendungen. Es wird jeweils ein Referenzbeispiel für einen ontologiebasierten und einen objektorientierten Context-Awareness Ansatz genauer betrachtet und erläutert. Abschließend werden die beiden Ansätze diskutiert und eine Bewertung bezüglich ihrer Einsatzmöglichkeit in Multiagentensimulationen gegeben.

3.1. Ontologiebasierter Ansatz - Gaia

Als Grundlage für das Beispiel eines ontologiebasierten Ansatz für Context-Awareness von Anwendungen, dient *Gaia* aus dem Paper *„An infrastructure for context-awareness based on first order logic"* von Ranganathan und Campbell (2003). Gaia beschreibt eine komplette Infrastruktur (siehe Abbildung 3.1) von der Erfassung von Kontextinformationen von verschiedenen Sensoren (Context Provider), über das Management und die Verarbeitung der Informationen (Context Synthesizer), bis hin zu ihrer Bereitstellung für die entsprechende Endanwendung (Context Consumer).

3.1.1. Kontextmodell

Das Kontextmodell von Gaia basiert auf Ontologien in denen die Kontextinformationen in Form von Kontextprädikaten abgelegt werden. Die geschieht meistens als ein Tripel in dem allgemeinen Format (<Subjekt> , <Verb> , <Objekt>). Beispiele solcher Kontextprädikate können wie folgt aussehen:

- Location (chris, entering, room 3231): Das Verb *entering* beschreibt die Beziehung des Subjekts *chris* zum Objekt *room 3231* und gibt damit die Information der entsprechenden Ontologie wieder, hier eine Location. Chris betritt also gerade den Raum 3231 bzw. befindet sich nun in ihm.

- Sister (venus, serena): Dieses Beispiel zeigt, dass das oben erwähnte Tripelformat nicht immer Anwendung finden muss sondern die Beziehung zwischen den Daten auch über die Ontologie selbst beschrieben werden kann. Venus und Serena sind also Schwestern.

- StockQuote (msft, “>“, $60): Es ist außerdem möglich als Verb eines Tripels einen Operator zu verwenden, wie in diesem Beispiel zu sehen ist. Dies ist hilfreich für weitere wichtige Konzepte von Gaia zur Verarbeitung von Kontextinformationen, wie zum Beispiel Kontextaggregation und das Evaluieren von vordefinierten Regeln (wie im folgenden Abschnitt 3.1.2 näher erläutert).

3.1.2. Kontextverarbeitung

Durch die Verknüpfung mehrerer Kontextinformationen mit Hilfe logischer Operatoren wie Konjunktion, Disjunktion und Negation, können neue Kontextinformationen aggregiert werden. Dazu können Kontextregeln definiert werden, die kontinuierlich darauf überprüft werden, ob sie in dem aktuellen Moment zutreffen oder nicht. Es kann also ein sogenanntes komplexes Ereignis erkannt werden (vgl. Abschnitt 2.6.3), welches sich aus mehreren einzelnen Ereignissen zusammensetzt:

Sound (Room 3234, “>“, 40 dB) ∧ Lightning (Room 3234, Stroboscopic)
∧ # People (Room 3234, “>“, 6) => Social Activity (Room 3234, Party)

Eine solche Regel und das Ergebnis ihrer Evaluation, kann dann verwendet werden, um die deduzierte Kontextinformation in der *Context History* zu speichern (siehe Abschnitt 3.1.4). Optional kann als Reaktion auf das erkannte komplexe Ereignis eine entsprechende Prozedur in der Anwendung angestoßen werden (siehe Listing in Abschnitt 3.1.4). Die Anwendung ist also fähig auf Situationen bzw. Zustände der Umwelt zu reagieren und ist somit Context-Aware.

3.1.3. Kontextzugriff

Der Zugriff auf Kontextinformationen der Sensoren (Context Provider aus Abbildung 3.1) in Gaia kann auf zwei Arten erfolgen:

1. Request - Response: Die Anwendung kann bei einem entsprechenden Context Provider per Request die aktuellen Informationen abfragen und bekommt diese per Response übermittelt.

2. Subscribe - Notify: Dies ist die interessantere Zugriffsmöglichkeit. Hier wird die Anwendung automatisch über den aktuellen Kontext informiert, falls eine Änderung eintritt. Die Anwendung kann sich dafür per Subscribe bei einem oder mehreren Context Providern registrieren und bekommt dann immer automatisch eine Benachrichtigung (Notify), wenn den jeweiligen Sensoren aktualisierte Daten vorliegen. Dadurch kann die Anwendung sofort auf eintretende Situationen reagieren. Die Subscribe - Notify Methode hat

den Vorteil, dass die Anwendung nur die für sie relevanten Informationen erhält und diese nicht unnötig dauerhaft bei den Context Providern abgefragt werden müssen.

3.1.4. Infrastruktur und Beispielanwendung

Abbildung 3.1 zeigt die Kontext-Infrastruktur von Gaia und die beteiligten Komponenten die im Folgenden aufgeführt sind:

- **Context Provider**: Beschreiben die Sensoren, die verschiedene Arten von Kontextinformationen erfassen und der Anwendung zur Verfügung stellen können. Ein Context Provider könnte zum Beispiel ein Temperatur- oder GPS-Sensor sein.
- **Context Synthesizer**: Sie können wie in 3.1.2 beschrieben mit Hilfe vordefinierter Regeln aus einfachen Kontextinformationen höhere Informationen aggregieren bzw. erschließen. Diese Informationen werden dann wieder zur Verfügung gestellt. Context Synthesizer sind also Context Provider aber auch Context Consumer.
- **Context Consumers**: Stellt die kontextsensitiven Endanwendungen dar, die die zur Verfügung gestellten Kontextinformationen verwenden, um ihr Verhalten der aktuellen Situation anzupassen.
- **Context Provider Lookup Service**: Hier können Anwendungen den entsprechenden Context Provider erfragen, der die für sie relevanten Kontextinformationen bereitstellt. Die Anwendung bekommt dann eine Referenz auf den jeweiligen Context Provider. Die Context Provider wiederum können hier die Art der Kontextinformationen registrieren, die sie zur Verfügung stellen.
- **Context History**: In der Context History werden bereits vergangene Kontextinformationen persistiert. Alle Informationen werden mit einem Zeitstempel in einer Datenbank gesichert, damit die Anwendungen bei Bedarf auch später noch auf die Informationen zurückgreifen kann.

Das folgende Listing 3.1 zeigt einen kleinen Ausschnitt aus einer Beispielanwendung mit Gaia. Es beinhaltet eine vordefinierte Regel und die dazugehörende Methode die aufgerufen wird, wenn die Regel zutrifft bzw. wenn die durch die Regel definierte Situation eintritt.

```
1 Location(Bhaskar, In, 2401) AND StockPrice(MSFT, >, 50)
2 AND StockPrice(SUNW, >, 10)
3 PlayVeryHappyMusic()
```

Listing 3.1: Beispiel eines Kontextmusters und der aufzurufenden Methode in Gaia

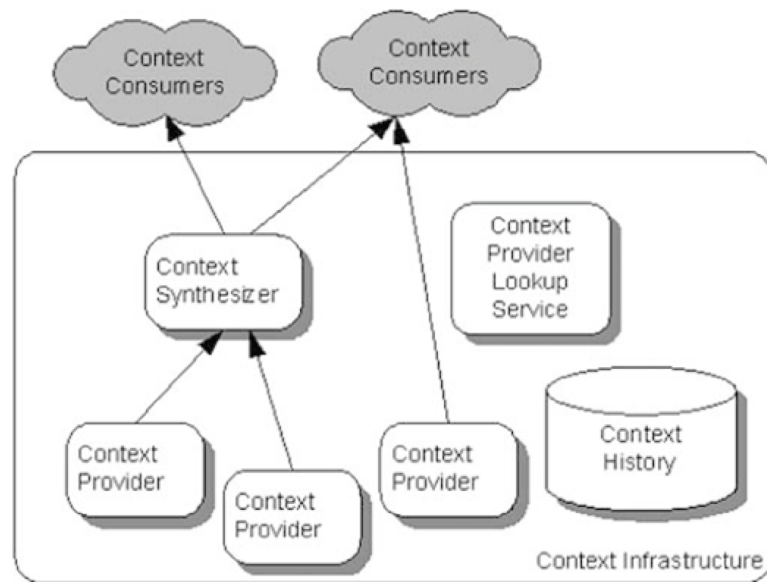


Abbildung 3.1.: Gaia Kontext-Infrastruktur [Ranganathan und Campbell (2003)]

3.2. Objektorientierter Ansatz - JCAF

Als Beispiel für den objektorientierten Ansatz wird oft das Paper „*The Java Context Awareness Framework (JCAF) - A Service Infrastructure and Programming Framework for Context-Aware Applications*“ von Bardram (2005b) zitiert, weshalb es nun im Folgenden genauer erläutert wird.

3.2.1. Kontextmodell

Das Kontextmodell von JCAF ist eine objektorientierte Abstraktion von Kontext und verwendet ebenfalls eine Tripelstruktur, um Kontextinformationen zu speichern und zu verwalten. Im Gegenteil zu Gaia, muss bei JCAF jedoch die Konvention einer Tripelstruktur immer eingehalten werden. Um Kontextinformationen ablegen zu können, gibt es in JCAF die Interfaces *Entity*, *Relationship* und *ContextItem*. Jede Entity-Instanz verfügt außerdem über einen *Context*-Container, in dem beliebig viele Relationships und damit verbundene Context Items gespeichert und der Entity-Instanz zugeordnet werden können. Das Kontextmodell von JCAF ist in Abbildung 3.2 dargestellt.

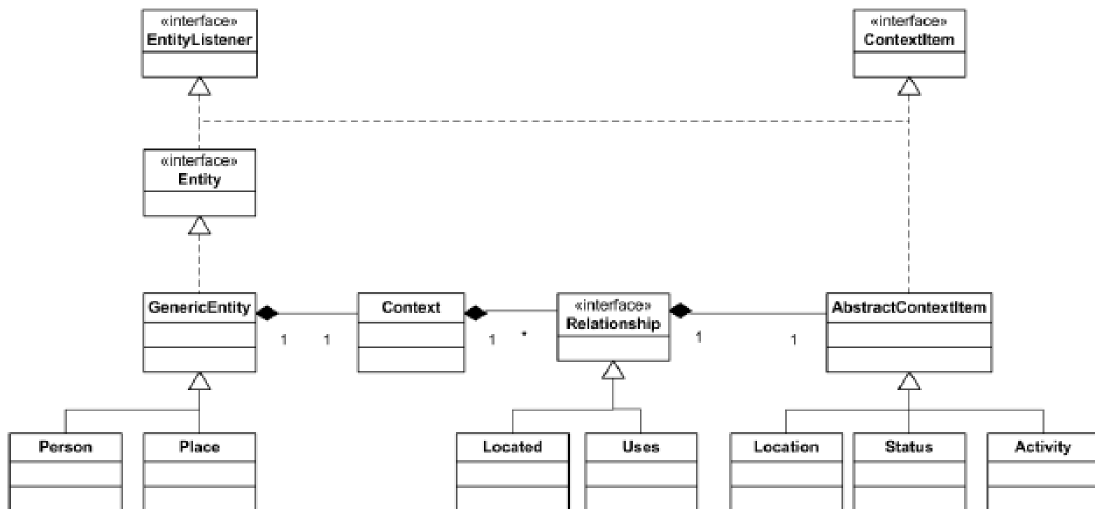


Abbildung 3.2.: JCAF Kontextmodell [Bardram (2005a)]

3.2.2. Kontextverarbeitung

Eine Möglichkeit einfache *low level* Kontextinformationen auf eine höhere Abstraktionsebene zu aggregieren oder mit Hilfe von Reasoning höherwertige Informationen aus Kontextdaten abzuleiten, ist in JCAF nicht gegeben. JCAF bietet außerdem keine Event Pattern Language an, mit der Auswahlkriterien oder Kontextmuster definiert werden können (siehe Abschnitt 2.6.5).

3.2.3. Kontextzugriff

Der Kontextzugriff in JCAF gestaltet sich von den Zugriffsmöglichkeiten analog zu Gaia. JCAF bietet ebenfalls ein einfaches Request - Response an. Ein Sensor, der durch einen *Context-Monitor* (Abbildung 3.3) implementiert wird, kann dann nach seinen aktuellen Daten gefragt werden kann. Es ist jedoch ebenfalls möglich per Subscribe - Notify automatisch über Kontextänderungen einer Entity oder einer bestimmten Entity-Instanz informiert zu werden.

JCAF bietet APIs an, mit denen sich der gesamte Kontext von Objekten abfragen lässt. Man kann sich dabei für Kontextänderungen einer Klasse oder einer spezifischen Instanz einer Klasse registrieren. Wenn man sich also beispielsweise für die Instanz mit der ID *2* der Klasse Gepard registriert, wird man über jegliche Kontextänderungen dieses Objekts benachrichtigt. Bei jeder Kontextänderung des *Entity*-Objekts, wird dann die Methode *contextChanged()* (siehe Listing 3.2) in dem Objekt oder zusätzlich in einem auf das Objekt registrierten *EntityListener* aufgerufen.

```java
public class ContextTester implements EntityListener {
  ...
  private void test() {
    getContextService().addEntityListener(listener, Gepard.class);
  }

  public void contextChanged(ContextEvent event) {
    // Check context changes
  }
}
```

Listing 3.2: Beispiel eines EntityListener in JCAF

Was sich an dem Kontext des Objekts geändert hat, muss dann allerdings selbstständig nachträglich herausgefunden werden. Dies hat den Nachteil, dass die Information über die Kontextänderung oft überflüssig sind, wenn es für den jeweiligen Fall unrelevante Kontextänderungen sind. In JCAF lassen sich damit keine Auswahlkriterien zur Eingrenzung der Ergebnismenge, wie zum Beispiel bei einer SQL-Abfrage von Datensätzen, formulieren.

3.2.4. Runtime Architektur

Die folgende Abbildung 3.3 zeigt die Runtime Architektur von JCAF, die Infrastruktur der Komponenten und ihre Abstraktion in drei verschiedene Layer.

- **Context Client Layer**: Der Context Client Layer repräsentiert die kontextbewussten Anwendungen, die die darunter liegenden Context Services benutzen, um Kontextinformationen abzufragen. Ebenfalls in diesem Layer befinden sich auch die *EntityListener*, die Kontextänderungen spezifischer Instanzen oder ganzen Klassentypen überwachen.

- **Context Service Layer**: Der Context Service Layer verwaltet die Kontextinformationen aller *Entity*-Objekte in Entity Containern. Diese kümmern sich um die Subscriber von Kontextereignissen und Benachrichtigen die entsprechenden Clients über Veränderungen des Kontexts von Objekten.

- **Context Sensor and Actuator Layer**: Der Context Sensor and Actuator Layer besteht aus den Kontextsensoren, die den aktuellen Kontext aus der Umwelt wahrnehmen und ihn den entsprechenden Entity-Objekten zuordnen sowie den Aktuatoren, die den Kontext der Umwelt aktiv verändern können.

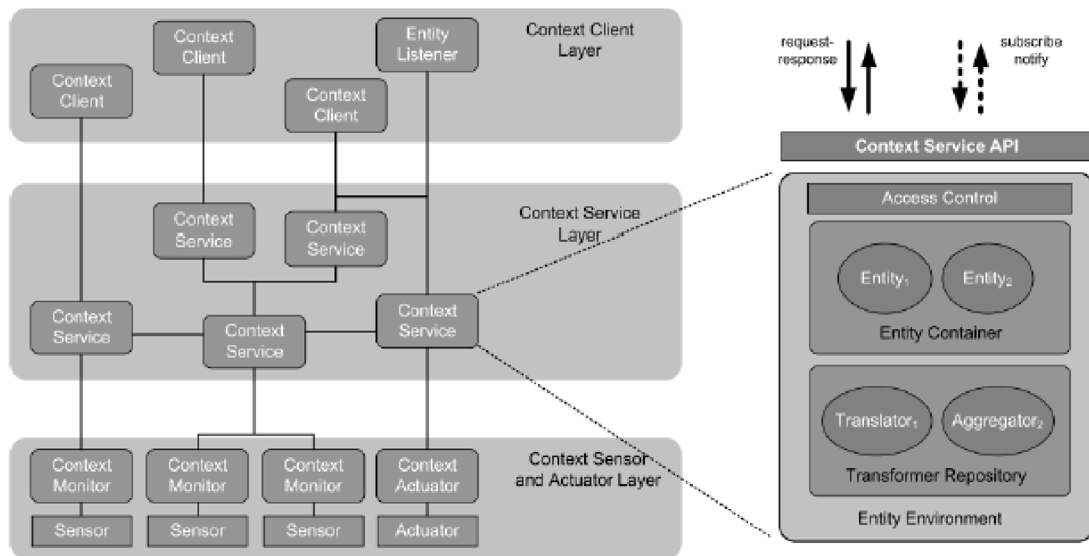


Abbildung 3.3.: Beispiel einer Deployment Situation und Details des Context Service von JCAF [Bardram (2005a)]

3.3. Eignung als Kontext-Komponente für Multiagentensysteme

Da das JCAF Framework für Forschungszwecke als Open Source frei zur Verfügung gestellt wird, wurde getestet inwieweit sich JCAF für eine Verwendung als eine Art *Plug & Play* Kontext-Komponente, in Verbindung mit einem Multiagentensystem eignet. Dabei wurden einige negative Aspekte von JCAF für den entsprechenden Anwendungsfall deutlich, die im Folgenden erläutert werden:

- **Abhängigkeit von Interfaces**: JCAF ist anscheinend nicht dafür vorgesehen, als eine Komponente in eine bestehende Anwendung integriert zu werden. Vielmehr muss die geplante Anwendung in dem Framework gegen zahlreiche Interfaces implementiert werden. Alle Abstraktionen von Kontextinformationen, die in der Anwendung verwendet werden sollen, müssen entsprechend als *Entity*, *Relationship* oder *ContextItem* realisiert werden. Dies gilt auch für die jeweiligen Sensoren und Listener, für die ebenfalls Interfaces vorgesehen sind. Andernfalls kann JCAF bzw. der bereitgestellte *ContextService* (Abbildung 3.3) die Kontextinformationen weder verwalten noch verarbeiten. Diese Umstände verhindern zudem, dass JCAF in Simulationen integriert und verwendet werden kann, die nicht in Java implementiert wurden.

- **Java RMI**: JCAF verwendet Java RMI zur Verteilung und Kommunikation zwischen der Anwendung (Client) und dem Kontext-Service (Server). Dies hat zur Folge, dass jede Instanz die zur Laufzeit in der Anwendung verwendet wird, als Java RMI Stub auf der Serverseite generiert werden muss. Dadurch wird ein entsprechend hoher Ressourcenverbrauch beim Arbeitsspeicher verursacht. Die Performance von JCAF war hinsichtlich einer Anforderung ggf. mehrere hunderttausend Agenten zu simulieren ebenfalls nicht ausreichend. Schon bei einer Menge zwischen zehn- und zwanzigtausend Kontextänderungen pro Simulationstick, wurden keine akzeptablen Zeitspannen erzielt, womit JCAF zumindest für möglichst echtzeitfähige Simulationen nicht in Betracht gezogen werden kann.

- **Context Reasoning**: JCAF stellt keinerlei Mechanismen für die Schlussfolgerung (*Reasoning*) [Bettini u. a. (2010)], von für die Anwendung relevanten Informationen aus verschiedenen Kontextinformationen bereit. Die Erkennung von sogenannten komplexen Ereignissen wird von JCAF nicht unterstützt. Es gibt zwar einen *Aggregator*, der aber über keinerlei direkt verwendbare Funktionen verfügt. Gaia unterstützt Context Reasoning hingegen zumindest für aussagenlogische Ausdrücke mit den Operatoren UND, ODER und NICHT (siehe Abschnitt 3.1.2).

- **Context Query Language**: Neben der Möglichkeit des Context Reasoning, fehlt JCAF auch eine umfangreiche Context Query Language (CQL) zur spezifischen Abfrage von Kontext. In JCAF werden APIs angeboten mit denen sich der Kontext von Objekten abfragen lässt. Jedoch lassen sich damit keine Auswahlkriterien zur Eingrenzung der Ergebnismenge, wie zum Beispiel bei einer SQL-Abfrage von Datensätzen, formulieren. Gaia dagegen unterstützt etwas spezifischere Abfragen von Kontextinformationen, welche die in [Reichle u. a. (2008)] genannten Anforderungen an eine CQL jedoch nur zu einem Bruchteil erfüllen.

- **Modularität und Flexibilität**: Die starke Kopplung und tiefe Verwobenheit der Gaia Infrastruktur und vor allem der JCAF Infrastruktur mit einer darin realisierten Anwendung, macht das gesamte System unmodular und unflexibel gegenüber Änderungen. Die für die Kontexterkennung in der Simulation zuständige Komponente, könnte nicht ohne schwerwiegende Auswirkungen auf das Gesamtsystem und hohen Aufwand gegen eine andere ausgetauscht werden. Dies liegt daran, dass alle Kontextproduzierenden und -konsumierenden Komponenten auf die Verwendung der entsprechenden Interfaces von Gaia bzw. JCAF abgestimmt sind.

3.4. Diskussion von Gaia und JCAF

Bei den Überlegungen zur effektiven Integration eines Context-Service in eine Multiagentensimulation, wurde zunächst ein möglicher Einsatz des *JCAF*-Frameworks in Betracht gezogen. Dies wurde jedoch aufgrund der im vorherigen Abschnitt 3.3 identifizierten Nachteile von JCAF verworfen. Statt dessen wurden die starken Vorzüge aus den Architekturen und Konzepten von JCAF und Gaia herausgearbeitet, die einen äußerst hohen Mehrwert für die Kontexterkennung in einer Multiagentensimulation haben könnten. Die Vorzüge waren vor allem die Folgenden:

1. **Producer Consumer Architecture Ansätze**: Die Architektur bzw. Infrastruktur von Gaia und vor allem JCAF, hat ein hohes Potenzial die fachlichen Situationen und Ereignisse einer Simulation verschiedenster Szenarien technisch angemessen zu realisieren und zu verarbeiten. Die logische Unterteilung der Anwendung in Kontext produzierende, verarbeitende und konsumierende Layer bzw. Teilkomponenten, eignet sich hervorragend für einen Einsatz in einer Multiagentensimulation.

   Die Agenten sowie die Umwelt der Simulation würden Kontextinformationen produzieren, die zur Verarbeitung an den Context-Service übertragen werden. Dieser würde die Kontextinformationen dann verarbeiten und die entsprechende Auswertung an die Simulation zurückführen. Dies wäre ein grundlegender Schritt in Richtung einer Event-Driven Architecture, bei dem die Ereignisverarbeitung im Mittelpunkt der Anwendung steht und die sich logisch aus produzierenden, verarbeitenden und konsumierenden Komponenten zusammensetzt (siehe Abbildung 2.7).

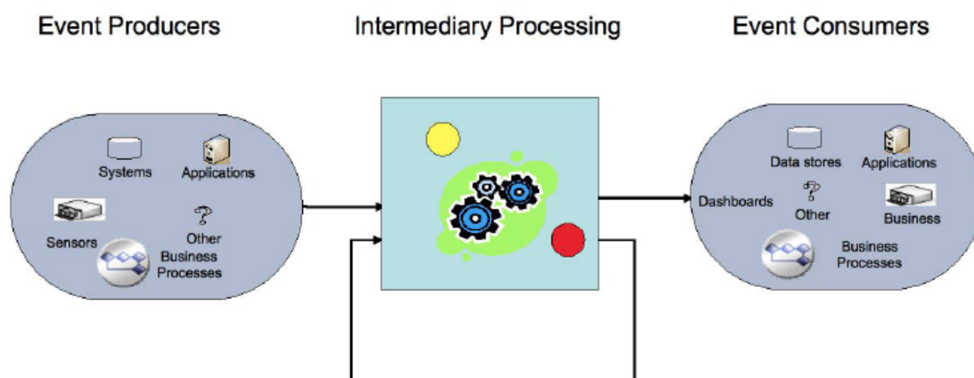


Abbildung 3.4.: Komponenten einer Event-Driven Architecture [Engel und Etzion (2011)]

2. **Kommunikationsverfahren**: Neben dem regulären *Request-Response*-Verfahren, sind in Gaia und JCAF auch *Publish-Subscribe* und *Push* ähnliche Kommunikations- und Benachrichtigungsmethoden zum Einsatz gekommen. Diese hätten den gravierenden Vorteil, dass ein Agent oder die gesamte Simulation nicht ununterbrochen bei dem Context-Service anfragen muss, ob ein relevantes Ereignis oder eine bestimmte Situation vorliegt. Statt dessen teilt der Context-Service diese Informationen vorgefiltert und vollautomatisch den entsprechenden Agenten oder Simulationskomponenten mit. Somit bekommen die Agenten ausschließlich die für sie relevanten Informationen direkt, wenn die entsprechenden Ereignisse oder Situationen eingetreten sind. Anschließend kann sofort eine Reaktion in Form eines Prozeduraufrufs oder ähnlichem eingeleitet werden.

3. **Automatisches Erkennen von definierten Kontextmustern**: JCAF erlaubt die automatische Benachrichtigung über Kontextänderungen eines Objekts, jedoch nicht über spezielle Kontextmuster oder Situationen, die zum Beispiel mit einer Kontextregel definiert werden können. Gaia hingegen bietet diese - wenn auch nur auf aussagenlogische Operatoren beschränkte - Möglichkeit bereits an. Die grundsätzlichen Ideen der beiden Ansätze, Änderungen und Kontextmuster automatisch zu erkennen ist daher ein nennenswerter Vorzug.

3.4.1. Fazit

In den Ansätzen von Gaia und JCFA wurden also Nachteile und Probleme in den folgenden Teilbereichen identifiziert:

- Systemarchitektur
- Kontextmodellierung
- Kontextverarbeitung
- Abfrage von Kontext
- Performance

Grundsätzlich zeigen beide Ansätze aber auch bereits einige Vorzüge und gute Ideen, um kontextbewusste Anwendungen entwickeln zu können. Diese sollen deshalb beim Design und der Realisierung eines eigenen Context-Service Konzepts aufgegriffen werden.

4. Analyse

In diesem Kapitel erfolgt die Analyse für den Entwurf der Testumgebung und das Konzept des Context-Service, der entwickelt werden soll. Der Context-Service soll dann in der prototypischen Testumgebung zum Einsatz kommen, um die zu Beginn dieser Arbeit aufgestellten Hypothesen untersuchen zu können. Dazu werden zunächst die Anforderungen an die Architektur der Testumgebung sowie des Context-Service identifiziert. Anschließend folgt die Analyse für das prototypische Simulationsszenario mit dem der Context-Service getestet werden soll. Dabei werden unter anderem die simulierten Agenten, die Ereignistypen, die Umwelteigenschaften sowie die relevanten Kontexte erläutert.

4.1. Grundlegende Anforderungen an die Simulationsumgebung

Im Folgenden werden nun die Anforderungen an die Simulationsumgebung analysiert und erläutert, mit der der Context-Service evaluiert werden soll. Die Anforderungen an die Simulationsumgebung für das Testsystem orientieren sich dabei zum Teil auch an den Anforderungen an die Simulationsumgebung *MARS*[1] [Hüning u. a. (2014); Baldowski u. a. (2014)], die die folgenden Möglichkeiten bieten soll:

- **Simulationsmodelle**: Eine grundlegende Anforderung an die Simulationsumgebung ist die Unterstützung verschiedenster fachlicher Simulationsmodelle aus unterschiedlichen Domänen. MARS bzw. die für diese Arbeit zu entwickelnde Testumgebung soll also nicht ausschließlich nur für ein bestimmtes Simulationsszenario genutzt werden können, sondern zum Beispiel für die Modellbildung in der Ökologie (Movement Ecology), Fußgängersimulationen, Krankheitsverbreitungen (Pandemien) und weitere Gebiete [Macal und North (2011)]. Die Modellierung und Implementierung von Simulationsmodellen sollte also möglichst modular sein, um Simulationsszenarien mit relativ wenig Aufwand realisieren und falls nötig anpassen zu können.

[1] http://mars-group.org/index.php/mars-system

- **Generische Struktur**: Die Verwendung einer generischen Struktur, für die jeweils in einem Szenario simulierten Instanzen ist erforderlich, um einen modularen Aufbau der Simulationsmodelle zu ermöglichen. Dazu gehören die Umwelt, sowie die darin agierenden Agenten und Objekte eines Szenarios. Dafür sollten alle in einem Simulationsszenario zum Einsatz kommenden Agenten und Objekte jeweils als eine eigene Komponente realisiert werden, in der dann die entsprechenden Implementierungsdetails enthalten sind.
- **Geoinformationssystem**: Geoinformationen, wie die Position von Pflanzen, Flüssen, Straßen und weitere, sollen mit Hilfe eines Geoinformationssystems (GIS) verwaltet werden. Die für ein Simulationsszenario benötigten Daten können dann vom GIS zur Verfügung gestellt und in die Simulation geladen werden, um zum Beispiel die Vegetation und weitere Umgebungsdaten entsprechend abbilden zu können.
- **Context-Awareness**: Die Umwelt, als auch die in ihr agierenden Agenten einer Simulation, sollen kontextbewusst sein. Dafür müssen aktuelle Ereignisse und Zustandsänderungen in der Simulation verarbeitet und ggf. entsprechende Aktionen als Reaktion ausgeführt werden. Die Umwelt sowie die Agenten, sollen sich ihrer aktuellen Situation bewusst sein und dadurch ein realistischeres Verhalten der gesamten Simulation ermöglichen.
- **Ereignisse**: Jede Zustandsänderung der Umwelt oder eines Agenten, soll als Ereignis wahrgenommen werden. Dazu gehört zum Beispiel die Positionsänderung eines Agenten, die Veränderung von Objekten in der Umwelt sowie jede anderweitige Aktion, die eine Veränderung der aktuellen Simulationssituation bewirkt. Jedes an einem Simulationsszenario beteiligte Objekt und jeder beteiligte Agent, soll seine Aktionen bzw. Veränderungen als Ereignis an den Context-Service mitteilen können.

4.2. Grundlegende Anforderungen an die Context-Service Komponente

Neben den vorangegangenen wünschenswerten Kriterien für die Simulationsumgebung, werden nun im weiteren Verlauf die Anforderungen an die Context-Service Komponente herausgearbeitet, die unter anderem von den allgemeinen Anforderungen an die Simulationsumgebung herrühren bzw. mit diesen einhergehen. Im Folgenden werden die Anforderungen an die Context-Service Komponente genannt, welche in funktionale und nichtfunktionale Anforderungen unterteilt sind.

4.2.1. Funktionale Anforderungen

- **Kontextinformationen**: Der Context-Service sollte eine einfache Abbildung von Kontextinformationen unterstützten und auf Strukturvorgaben, wie bei JCAF oder Gaia (siehe Abschnitt 3.1.1 und 3.2.1) weitestgehend verzichten. Dadurch soll eine leichtere Modellierung unterschiedlichster Kontextinformationen ermöglicht werden. Der Context-Service muss Kontextinformationen verschiedener fachlicher Domänen verwalten und verarbeiten können.

- **Reasoning**: Der Context-Service sollte das Reasoning von Kontextinformationen [Bettini u. a. (2010)] unterstützen
  - um Veränderungen in der Anwendung, wie zum Beispiel die Positionsveränderung eines Agenten in der Simulation, wahrnehmen und bewerten zu können.
  - um entscheiden zu können, ob Anpassungen an die durch die Veränderungen entstehenden neuen Kontextsituationen erforderlich sind.
  - um *High Level* Kontext aus *Low Level* Kontextinformationen ableiten zu können (siehe Abschnitt 2.3).

- **Abfrage von Kontext**: Der Context-Service sollte eine einfache Möglichkeit bieten, die für das jeweilige Simulationsszenario relevanten Kontextmuster, die während der Simulation erkannt werden sollen, zu definieren. Dabei sollte es auch möglich sein, komplexe Ereignismuster (siehe Abschnitt 2.6.3) sowie Zusammenhänge und Abhängigkeiten zwischen Kontextinformationen festlegen zu können. Es müssen Eingrenzungen der Ergebnismenge bei einer Abfrage, ähnlich wie bei einer SQL-Query, gemacht werden können. Ein Beispiel wäre die Filterung nach Objekten die einen bestimmten Attributwert haben.

- **Temporallogik**: Der Context-Service sollte zeitliche Abhängigkeiten und Zusammenhänge zwischen Kontexten erkennen können. Das bedeutet, dass eine zeitliche Dimension bei der Verarbeitung der Kontextinformationen unterstützt werden muss. Dies wird unter anderem benötigt, um Reasoning von Kontextinformationen über einen längeren Zeitabschnitt Δt, statt einem singulären Zeitpunkt t wie bei JCAF und Gaia durchführen zu können.

  Aufbauend auf die Definition temporallogischer Aussagen, wäre eine Möglichkeit zur Verbindung temporaler Merkmale mit spatialen Merkmalen wünschenswert, um spatiotemporale Kontextmuster formulieren zu können (siehe Abschnitt 2.6.3).

- **Automatische Benachrichtigungen**: Der Context-Service sollte die Simulation automatisch über das Erkennen bzw. Eintreten eines vordefinierten Kontextmusters informieren können. Die Benachrichtigung muss es der Simulation außerdem ermöglichen zu identifizieren, welches Kontextmuster erkannt wurde, um die jeweils passende Reaktion anstoßen zu können.

4.2.2. Nichtfunktionale Anforderungen

- **Hochsprachenunabhängigkeit**: Das Kontextmodell sollte auf Strukturvorgaben, wie bei JCAF oder Gaia (siehe Abschnitt 3.1.1), sowie Vorgaben zur Verwendung hochsprachenabhängiger Interfaces verzichten. Dadurch soll eine leichtere Modellierung unterschiedlichster Kontextinformationen und eine hochsprachenunabhängige Verwendung des Context-Service ermöglicht werden.

- **Lose Kopplung der Komponente**: Die hinter der Context-Service Komponente stehende technische Umsetzung, sollte für die anderen Systemkomponenten transparent bleiben. Die Komponente sollte sich auf den Austausch von Kontextdaten in Form eines Nachrichtenformats und der Definition von Regeln zur Erkennung von Ereignismustern beschränken. Dadurch bleibt das System leichter wartbar und erweiterbar.

- **Leichte Integration**: Die Context-Service Komponente sollte möglichst leicht integriert werden können. Das heißt, dass eine Simulation mit verhältnismäßig wenig Aufwand kontextbewusst gemacht werden können soll. Dabei soll außerdem auch der Use Case in Betracht gezogen werden, dass eine bereits bestehende Simulation nachträglich kontextbewusst gemacht werden kann. Dies soll ebenfalls mit einem guten Verhältnis aus Aufwand und Nutzen möglich sein.

- **Skalierbarkeit**: Die Context-Service Komponente sollte nach Möglichkeit eine Form der Skalierbarkeit unterstützen, um bei entsprechender Last trotzdem eine akzeptable Performance bieten zu können.

- **Kommunikation über Nachrichten**: Die Kommunikation mit der Context-Service Komponente, sollte über Nachrichten bzw. Nachrichtenströme (Message Queues) erfolgen können. Dies hätte den Mehrwert, dass die technische Umsetzung der Komponente möglichst transparent bleibt und somit nur das Nachrichtenformat bzw. die inhaltliche Struktur der Nachrichten bekannt sein und eingehalten werden muss.

- **Flexibilität**: Die Context-Service Komponente sollte eine einfache Anpassung an das jeweils gewünschte Szenario, welches simuliert werden soll ermöglichen. Hier ist vor allem die flexible Anpassung von bestehenden, sowie die neue Definition von Regeln zur Erkennung von wichtigen Ereignissen und speziellen Situationen in der Simulation wünschenswert.

- **Performance**: Damit in der Simulation umgehend auf aktuelle Ereignisse reagiert und der weitere Simulationsverlauf entsprechend gesteuert werden kann, wäre es von Vorteil, wenn eine Betrachtung des aktuellen Simulationszustands, anstatt einer retrospektiven Betrachtung durchgeführt werden kann. Das bedeutet, dass der Context-Service eine möglichst nahezu echtzeitfähige Verarbeitung von Kontextdaten bewerkstelligen können sollte.

Die hier identifizierten nichtfunktionalen Anforderungen an die Context-Service Komponente sind gleichzeitig Anforderungen an ein System, welches das Reactive Computing Paradigma verfolgt. Viele Anforderungen an ein Reactive Computing System, die in Abschnitt 2.7.2 genannt wurden, scheinen von den Konzepten einer Event-Driven Architecture und der Technik des Complex Event Processing ebenfalls erfüllt werden zu können. Durch die Verwendung der Konzepte und Technologien einer Event-Driven Architecture und des Complex Event Processing zur Kontexterkennung in Multiagentensimulationen, sollen die Vorteile des Reactive Computing gegenüber des Responsive Computing aufgezeigt werden.

4.3. Beispielszenario

Das entwickelte Konzept des Context-Service, wurde primär mit einem Fokus auf den Einsatz in Multiagentensimulationen entworfen, in denen komplexe Abläufe der realen Welt mit Hilfe von autonomen Agenten simuliert werden. Um den Context-Service und seine Einsatzfähigkeit in den in Abschnitt 3.4.1 genannten Teilbereichen zu evaluieren, muss daher eine Beispielsimulation entworfen und implementiert werden, die entsprechende Testbedingungen ermöglicht. Simulationstools wie zum Beispiel *MARS*, in denen der Context-Service Verwendung finden können soll, sollten neben weiteren Simulationsdomänen zum Beispiel ökologische Modelle simulieren können. Aus diesem Grund wurde für das hier verwendete Beispielszenario, das in der Testumgebung simuliert werden soll, die Bachelorarbeit von Malte Eckhoff [Eckhoff (2013)] als Informationsgrundlage herangezogen. Die Arbeit befasst sich mit der Simulation des Jagdverhaltens von Geparden.

Das Beispielszenario basiert auf einem ökologischen Modell, in dem das Verhalten von Geparden und Gazellen in freier Natur simuliert wird. Die Geparden werden in ihrem Jagdverhalten simuliert und verfolgen dabei Gazellen. Die Gazellen fliehen wiederum vor den Geparden, solange sie genügend Energie haben. Eine Gazelle kann dem Geparden dann entweder entkommen oder sie wird von dem Geparden erlegt. Während der gesamten Simulationszeit verändern sich nun in jeder Simulationsiteration die Zustände der Gazellen und Geparden, wie zum Beispiel ihre Position, ihre Energie usw. und damit ihr derzeitiger Kontext.

Für die Steuerung des weiteren Simulationsablaufs könnte es zum Beispiel wichtig sein Kontexte zu erkennen, die über den normalen Sichtbereich eines Agenten hinaus gehen. Ein denkbares Beispiel wäre, dass erkannt wird wenn ein Gepard hungrig ist, sich auf einem Aussichtspunkt befindet und sich mehr als fünf Gazellen in einem Radius von eintausend Metern zur Position des Geparden aufhalten. Dieser Kontext ist als Beispiel in Abbildung 4.1 im oberen rechten Sektor des Teilausschnitts dargestellt. Falls dieses *komplexe Ereignis* während der Simulation eintritt, könnte der Gepard beginnen die Gazellen zu jagen. Die Kontextmuster, die während der Simulation automatisch erkannt werden sollen, können mit Hilfe von Regeln im Context-Service definiert werden.

Die im weiteren Verlauf dieses Abschnitts beschriebene Analyse des Testszenarios und seiner Anforderungen, beschränkt sich auf eine grobe Modellierung der Simulation, da eine detailgetreue Nachstellung des echten Jagdverhaltens von Geparden nicht Ziel dieser Arbeit ist.

4.4. Räumliche Darstellung und Umwelt

Die räumliche Ausdehnung der Simulation benötigt mindestens eine zweidimensionale Darstellung des Testszenarios aus der Vogelperspektive, da dies eine Voraussetzung ist, um räumlichen bzw. spatialen Kontext und Kontextmuster in der Simulation abbilden zu können. Die Betrachtung des Szenarios auf einer 2D-Ebene reicht aus, um Bewegung von Agenten, Sektoren und Flächen wie zum Beispiel Wälder, Grasflächen, Seen oder Flüsse zu simulieren und zu visualisieren. Eine Simulation im 3D-Raum ist daher nicht zwingend notwendig. In dem simulierten 2D-Raum, soll ein x-y-Koordinatensystem verwendet werden. Innerhalb dieses Koordinatensystems sollen sich dann die verschiedenen Objekte und Agententypen anordnen bzw. frei bewegen.

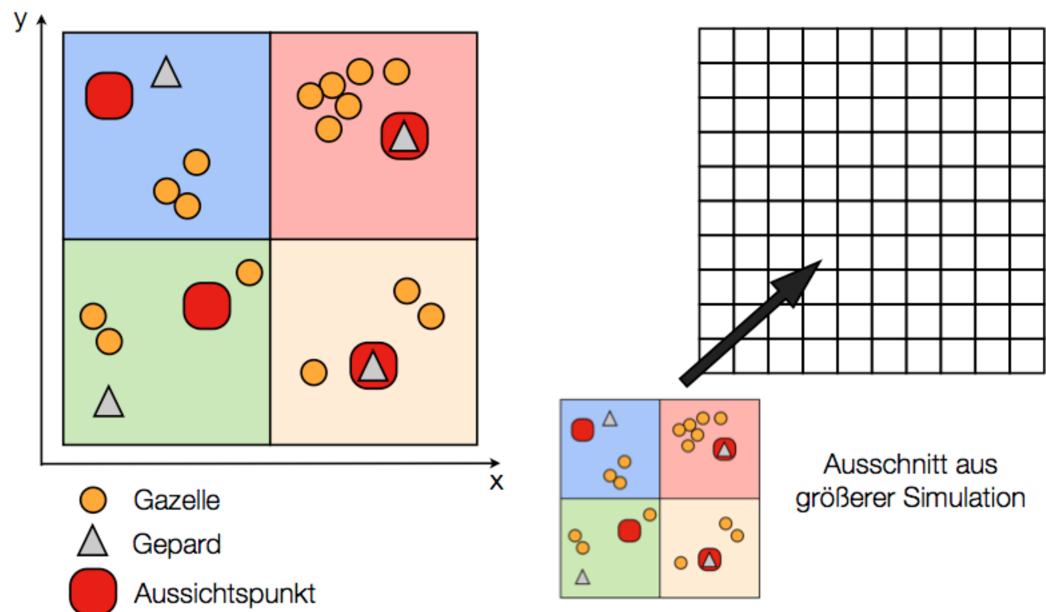


Abbildung 4.1.: Beispiel der Geparden- und Gazellen-Simulation zur Auswertung von Kontexten

4.4.1. Sektoren

Die Umwelt der Simulation, kann aus mehreren räumlichen Sektoren bestehen bzw. logisch in diese unterteilt werden. Alle Objekte werden in bestimmten Sektoren abgebildet. Die Agenten können die Sektoren betreten und in ihnen verbleiben oder sie durchqueren und auch wieder verlassen. Dadurch können über einen zeitlichen Verlauf, verschiedene spatiale Kontexte und Kontextmuster entstehen, die dann vom Context-Service erkannt werden sollen. Die Aufteilung des Simulationsgebiets in mehrere unterschiedliche Sektoren ist in Abbildung 4.1 veranschaulicht. Innerhalb der Sektoren herrschen verschiedene Teilkontexte vor, die gegebenenfalls ein vordefiniertes Kontextmuster darstellen.

4.4.2. Aussichtspunkte

Die Aussichtspunkte dienen den Geparden zur besseren Sichtung ihrer potentiellen Beutetiere. Befinden sich die Geparden auf so einem Aussichtspunkt, können sie über einen größeren Radius nach Gazellen Ausschau halten und sich gegebenenfalls auf die Jagd begeben. Der Aussichtspunkt stellt ein logisches Teilgebiet des Simulationsraums dar.

4.4.3. Wasserstellen

In dem Beispielszenario werden zudem auch Wasserstellen dargestellt. Diese werden ebenfalls wie die Aussichtspunkte, als Teilgebiete innerhalb des simulierten Gebiets abgebildet. Die Geparden und Gazellen können die Wasserstellen aufsuchen aber nicht betreten oder überqueren.

4.5. Agenten

Für das Testszenario, welches das Jagdverhalten von Geparden simulieren soll, werden entsprechende Agenten benötigt, mit deren Hilfe die Geparden und ihre Beutetiere in der Simulation abgebildet werden können. Bei den Beutetieren beschränkt sich das Testszenario auf die Simulation von Gazellen.

4.5.1. Geparden

Die simulierten Geparden bewegen sich zufällig durch das Simulationsgebiet und halten Ausschau nach Beutetieren. Wenn sie Beutetiere in ihrer unmittelbaren Nähe wahrnehmen, beginnen sie mit der Jagd auf diese. Der Jagderfolg des Geparden richtet sich dann nach verschiedenen Faktoren [Hilborn u. a. (2012)]. Die Geparden sollen in der Simulation mit verschiedenen Attributen abgebildet werden, wie zum Beispiel Geschlecht, Position und Alter. Bestimmte Attribute können sich über den gesamten Simulationsverlauf verändern. Eine kontinuierliche Auswertung und Untersuchung dieser Informationen auf Kontextmuster, soll dann vom Context-Service durchgeführt werden.

4.5.2. Gazellen

Die Gazellen in dem Beispielszenario dienen als Beutetiere für die Geparden. Diese bewegen sich ebenfalls wie die Geparden zufällig innerhalb des simulierten Gebiets. Nehmen die Gazellen einen Geparden in ihrer Nähe wahr oder werden sie von einem Geparden gejagt, dann fliehen sie. Die Chance einer Gazelle, einem Geparden der sie verfolgt zu entkommen, richtet sich sich wie bei dem Geparden nach verschiedenen Faktoren. Hat die Gazelle zum Beispiel mehr Energie als der Gepard, der sie verfolgt, dann kann sie entkommen. Die Gazellen sollen mit einigen Eigenschaften, wie zum Beispiel Energie und Position in der Simulation abgebildet werden. Wenn nicht mehr genügend Gazellen in der Simulation vorhanden sind, können sie sich auch vermehren. So wird sichergestellt, dass es immer Beutetiere gibt.

4.6. Verschiedene Aspekte von Kontext

Das in dieser Arbeit entwickelte Context-Service Konzept, soll unabhängig von der fachlichen Domäne der Simulation die den Context-Service nutzt, eingesetzt werden können. Daher müssen verschiedene Aspekte bzw. mögliche Eigenschaften von Kontext [Reichle u. a. (2008)] betrachtet werden, die im Folgenden aufgeführt sind:

- **Dynamisch oder statisch**: Kontextinformationen können dynamisch oder statisch sein. Ein Beispiel für dynamischen Kontext, wäre die Position eines Geparden während des Simulationsverlaufs, da diese sich laufend ändert. Die *ID* eines Geparden ist dagegen eine statische Kontextinformation, die sich während der gesamten Simulation nicht verändert.

- **Kontinuierliche Datenströme**: Kontextinformationen können kontinuierliche Datenströme bilden, wenn zum Beispiel in regelmäßigen Abständen Kontextinformationen ausgelesen bzw. empfangen werden. Diese Datenströme stellen dann ein großes Volumen an auszuwertenden Kontextdaten dar.

- **Metadaten und Qualität**: Kontextinformationen können mit Metadaten, wie zum Beispiel der Angabe der Kontextquelle versehen werden. Die Kontextdaten können temporale Informationen beinhalten [van Bunningen u. a. (2005)], wie zum Beispiel die Angabe eines Zeitstempels. Außerdem kann Kontext starke Qualitätsschwankungen aufweisen. Die Informationen können fehlerhaft, nicht eindeutig, nicht verfügbar oder unvollständig sein.

- **Räumliche Abhängigkeiten**: Kontextdaten können spatiale Informationen enthalten [van Bunningen u. a. (2005)], wie zum Beispiel die Angabe der aktuellen Position. Die Kontextdaten können aber auch Abstände bzw. Entfernungen oder Dimensionen, wie Länge, Breite und Höhe enthalten.

- **Situationen**: Kontextdaten können Situationen ergeben, die aus anderen Kontextdaten abgeleitet wurden (Reasoning), wie zum Beispiel die Information, dass sich ein Gepard auf einem Aussichtspunkt befindet. Dies kann aus seinen Positionsdaten abgeleitet werden.

All diese verschiedenen Aspekte von Kontext sollte der Context-Service berücksichtigen und verarbeiten können, wenn er universell und unabhängig von der fachlichen Domäne der Simulation eingesetzt werden können soll.

4.7. Relevante Kontexte

Die hier relevanten Kontexte setzen sich aus mehreren Kontextdimensionen zusammen. Dazu gehören die *räumliche*, *zeitliche* und *soziale* Dimension.

- **Räumliche Dimension**: Hier werden spatiale Kontextinformationen abgebildet. Die Position eines Agenten ist für fast jede Kontextsituation relevant, da sich aus seiner Position ergibt, wie sich seine direkte Umgebung zusammensetzt bzw. wie diese beschaffen ist.

- **Zeitliche Dimension**: Die zeitliche Dimension ist ebenfalls sehr wichtig, da sich aus aus dem Zeitpunkt unterschiedlicher Kontext ergeben kann. Außerdem gibt die Zeitdimension die Möglichkeit, temporale Korrelationen bzw. Zusammenhänge zwischen Ereignissen herzustellen und diese in einem zeitlichen Verlauf zu ordnen. Zum Beispiel kann Kontext A vor, während oder nach Kontext B eingetreten sein. Dies lässt sich nur feststellen, wenn eine zeitliche Dimension vorhanden ist.

- **Soziale Dimension**: Der sozialen Dimension sollen in diesem Fall zum Beispiel Eigenschaften wie Hunger, Alter oder das Geschlecht eines Geparden zugeordnet werden. Dadurch kann dann die Eigenschaft, dass ein Gepard gerade Hunger hat mit dem Zeitpunkt dieses Zustands und seiner Position kombiniert und gegen etwaige relevante Kontextregeln geprüft werden.

4.7.1. Beispiele

Die Aufgabe des Context-Service, ist das Erkennen von komplexen Mustern zwischen Ereignissen in der Simulation und die Initiierung von Reaktionen, wenn diese komplexen Muster erkannt werden. Um die Funktionsfähigkeit des Context-Service hinsichtlich der Erkennung von Kontexten zu testen, werden im Folgenden ein paar Beispiele für Kontextmuster mit unterschiedlicher Komplexität genannt.

- Ein simples Beispiel, wäre die Benachrichtigung bei jedem Gepard-, Gazellen- oder Kill-Event. Dies könnte zum Beispiel aus Logging-Gründen interessant sein, um ein Bewegungsprofil eines Geparden oder einer Gazelle für retrospektive Betrachtungen des Simulationsverlaufs zu erstellen.

- Ein komplexeres Beispiel, wäre die Benachrichtigung, wenn ein Gepard innerhalb einer festgelegten Zeitspanne bestimmte Teilzonen des simulierten Gebiets betreten hat. Hier

würden dann bereits die zeitliche (*temporale)* und die örtliche (*spatiale*) Dimension der auftretenden Ereignisse betrachtet und miteinander in Beziehung gesetzt werden.

- Ein weiteres interessantes Beispiel ist die Benachrichtigung über das Ausbleiben, also das nicht auftreten, von bestimmten Ereignissen und Ereignismustern. Ein solches Muster könnte das Ausbleiben des Jagderfolges von Geparden innerhalb eines festgelegten Zeitraums und Teilgebiets der Simulation sein. Wenn ein Gepard also innerhalb des festgelegten Zeitraums keinen Jagderfolg in seinem Gebiet hatte, d. h. er hat keine Gazelle erlegen können, dann verhungert er. Erkennt der Context-Service das Ausbleiben des Kontextmusters, kann er die Simulation darüber informieren und der entsprechende Gepard wird inaktiv. Alternativ könnten neue Gazellen zur Simulation hinzugefügt werden, damit mehr Beutetiere zur Verfügung stehen.

Diese Beispiele können als Kontextregel in der *ContextRuleBase*, also der Regelbasis des Context-Service (siehe Abschnitt 6.2.3), registriert werden.

4.8. Zusammenfassung

In der Analyse wurden einige anspruchsvolle Anforderungen an die Simulationsumgebung und den Context-Service festgehalten. Die Simulationsumgebung sollte die realen Einsatzbedingungen für die Context-Service Komponente abbilden können. Dazu gehört vor allem die Möglichkeit verschiedenste fachliche Modelle simulieren zu können, sowie Ereignisse mit Kontextinformationen zu produzieren und diese an den Context-Service zu übermitteln.

Die Anforderungen an den Context-Service bestehen gleichermaßen aus funktionalen sowie nicht funktionalen Kriterien, die so gut wie möglich erfüllt werden sollen, um die in Kapitel 3 diskutierten Arbeiten zu verbessern. Die analysierten nichtfunktionalen Anforderungen, sind unter anderem wichtige Bausteine für eine Realisierung eines Systems nach dem Reactive Computing Paradigma.

Um das Zusammenspiel der Simulationsumgebung und der Context-Service Komponente letztendlich untersuchen zu können, wurden Kriterien für ein Testszenario festgehalten, mit dem unterschiedliche Testkonfigurationen realisiert und die verschiedenen Aspekte von Kontext während der Simulation abgebildet werden können. Es wurden außerdem für die Testsimulation relevante Kontextdimensionen genannt und Beispiele für unterschiedliche komplexe Kontextmuster gegeben, die vom Context-Service erkannt werden können sollen.

5. Design

Das nun folgende Kapitel befasst sich mit dem Design des *MARS*-Systems und der Integration des Context-Service in das MARS-Framework, sowie dem Design des Testszenarios welches simuliert werden soll. Dazu werden die im vorangegangenen Kapitel ermittelten Anforderungen aufgegriffen und entsprechende Designentscheidungen getroffen, um die jeweiligen Anforderungen so gut wie möglich erfüllen zu können. Bei dem Design der Context-Service Komponente, werden zudem die maßgeblichen Elemente für eine Reactive Computing Architektur berücksichtigt. Zunächst wird die Architektur des MARS-Systems betrachtet. Anschließend wird auf die Layerstruktur der Simulation, den Aufbau der Agenten und die Schnittstellen zum Context-Service eingegangen. Danach folgt eine Erläuterung der Teilkomponenten des Context-Service und ihrer jeweiligen Aufgabe, sowie eine Darstellung der Funktionsweise des Gesamtsystems. Abschließend wird die Modellierung des Testszenarios und der darin simulierten Ereignisse behandelt, für die jeweils ein entsprechender Ereignistyp vorgestellt wird.

5.1. Simulationsumgebung

Die Simulationsumgebung stellt in dem Gesamtsystem, welches aus der Simulationsumgebung selbst und dem Context-Service besteht, einen *Event Producer* und gleichzeitig einen *Event Consumer* dar. Sie produziert Ereignisse und übermittelt sie an den Context-Service zur Auswertung. Wenn ein definiertes Kontextmuster erkannt wird, sendet der Context-Service ein Ereignis an die Simulationsumgebung, welche das Ereignis konsumiert. Dieser Aufbau ist exemplarisch in Abbildung 5.1 veranschaulicht. An dieser Stelle ist bereits eine grundlegende Designentscheidung ersichtlich, nämlich die vollständige Entkopplung der Context-Service Komponente vom Rest des Systems. Die Gründe für diese Designentscheidung, werden im Verlauf dieses Kapitels erläutert. Zunächst wird nun aber ein Überblick über die MARS-Architektur gegeben und ihre wesentlichen Teilkomponenten betrachtet.

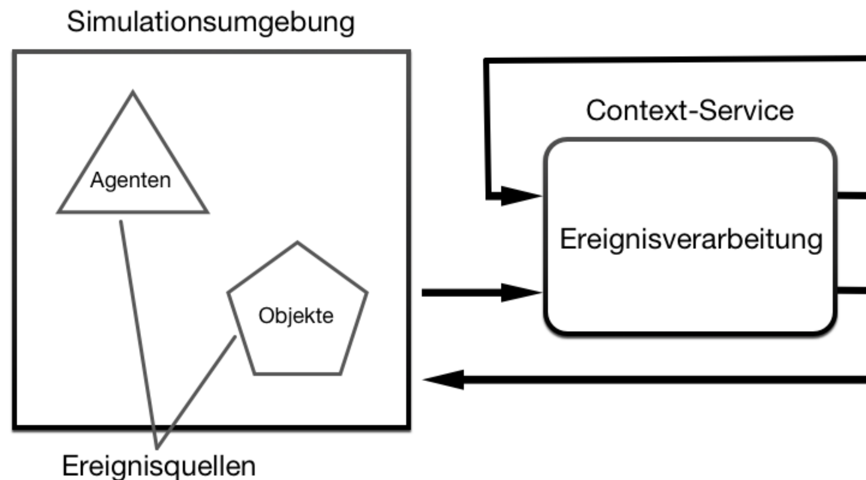


Abbildung 5.1.: Einteilung in die wesentlichen Komponenten einer Event-Driven Architecture. Die Simulationsumgebung als *Event Producer* und *Event Consumer* ist logisch getrennt von der Ereignis verarbeitenden Context-Service Komponente.

5.1.1. Gesamtbild der MARS-Architektur

MARS ist ein interdisziplinäres Projekt der *Multi-Agent Research and Simulation* (MARS) Forschungsgruppe am Department für Informatik der Hochschule für angewandte Wissenschaften in Hamburg. Das MARS-System ist ein verteilbares und hoch skalierbares Multiagenten-Simulationsframework [Hüning u. a. (2014)], mit dem sich komplexe Real-World Szenarios [Thiel-Clemen u. a. (2011); Baldowski u. a. (2014)] aus unterschiedlichsten fachlichen Domänen simulieren lassen.

Das MARS-System besteht aus einer Reihe von zusammenhängenden Komponenten, die unterschiedliche Funktionen wahrnehmen und gemeinsam das Gesamtsystem bilden. Die Gesamtarchitektur lässt sich in die folgenden drei wesentlichen Aufgabenbereiche unterteilen, denen die Teilkomponenten entsprechend zugeordnet sind:

- **Datenintegration**: Dazu gehören *GROUND*, *ROCK*, *SHUTTLE* und *DEIMOS*
- **Simulation**: Wird von der *LIFE* Komponente durchgeführt
- **Visualisierung & Analyse**: Hier kommen die Komponenten *VIEW* und *SURVEYOR* zum Einsatz

Das Gesamtbild der MARS-Architektur ist in Abbildung 5.2 dargestellt. Der Context-Service wird in der MARS-Architektur durch die *SURVEYOR*-Komponente dargestellt. Die Namensgebung beruht auf einem NASA Projekt[1] mit der gleichnamigen Raumsonde zur Erforschung des Mars.

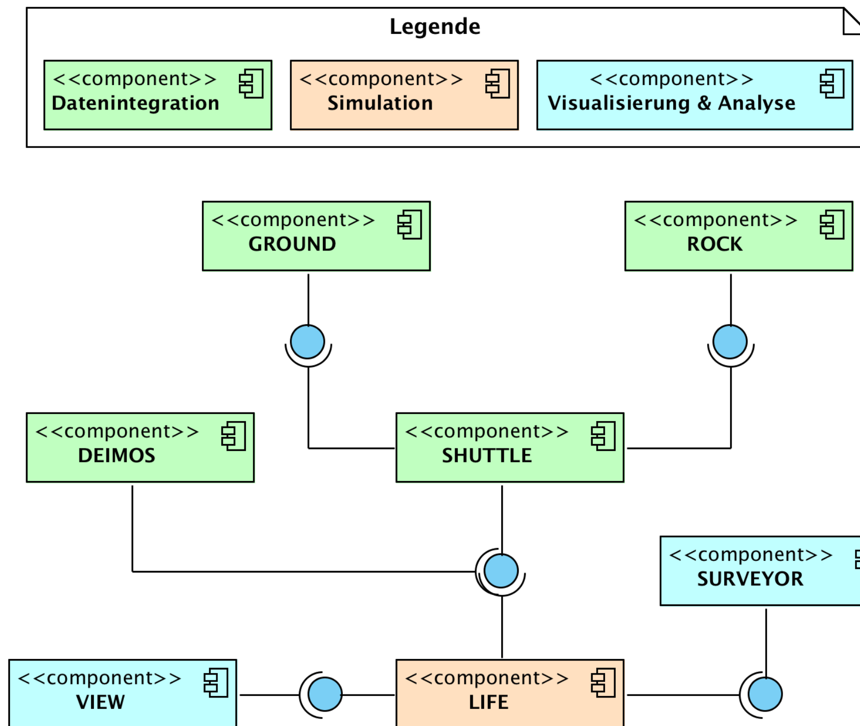


Abbildung 5.2.: MARS Deployment Architektur

Die SURVEYOR-Komponente gehört zu den MARS LIFE-Services und kann von der LIFE-Komponente mit Hilfe eines Adapters angesprochen werden, der entsprechende Interfaces zur Verfügung stellt. Die verwendbaren Interfaces werden in Kapitel 6 anhand von Beispielen genauer beschrieben. Der Context-Service läuft dabei als eigenständige und vollständig entkoppelte Server Anwendung und kommuniziert über Nachrichten mit der LIFE-Komponente (siehe Abbildung 5.5). LIFE stellt die grundlegende Komponente von MARS dar, in der letztendlich die Simulation eines Szenarios durchgeführt wird. Ein wesentlicher Bestandteil der LIFE-Komponente sind *Layer Container*, mit deren Hilfe der zentrale Ansatz einer Layerstruktur umgesetzt wird [Hüning u. a. (2014)]. Dieser wird im folgenden Abschnitt ausführlicher erläutert.

[1] http://de.wikipedia.org/wiki/Mars_Global_Surveyor

5.1.2. Layerstruktur

Durch die Verwendung einer Layerstruktur, soll eine logische Unterteilung der einzelnen Simulationsbestandteile erreicht werden. Das bedeutet, dass jedes in der Simulation vorkommende Objekt und jeder Agent, einen eigenen Layer implementiert. Jeder Layer kapselt seine entsprechenden Attribute und seine Geschäftslogik. Alle Layer zusammen bilden dann das Simulationsszenario ab. Ein Beispiel für die Darstellung dieser Layerstruktur ist in Abbildung 5.3 zu sehen. Diese Art der Erzeugung von Simulationsszenarien, ermöglicht eine Unterstützung verschiedenster fachlicher Simulationsmodelle aus unterschiedlichen Domänen. Die Modellierung und Implementierung von Simulationsszenarien kann also modular erfolgen. Durch den modularen Aufbau können die Szenarien außerdem leichter gewartet oder angepasst werden. Neue Komponenten bzw. Layer lassen sich flexibel hinzufügen. Die Layerarchitektur bietet damit eine gute Grundlage für die in Abschnitt 4.1 genannten entsprechenden Anforderungen an die Simulationsumgebung.

Jeder einzelne Layer eines Simulationsszenarios, kann durch den modularen Aufbau als eine eigene Ereignisquelle (*Event Producer*) angesehen werden und agieren (siehe Abbildung 5.4). Während des Simulationsverlaufs, kann jeder Layer relevante Ereignisse und Veränderungen als Nachricht an den Context-Service übermitteln.

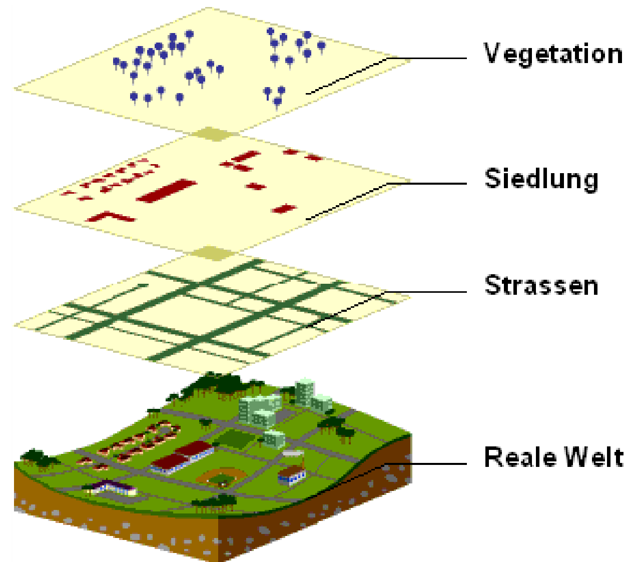


Abbildung 5.3.: Aufbau eines Simulationsszenarios aus mehreren Layern[2]

[2]http://www.gis-projektmanager.de/frames/gis.htm

5.1.3. Agenten

Die Agenten, die in der Simulation zum Einsatz kommen, werden fachlich getrennt als Layer modelliert. Die Geparden und Gazellen (siehe Abschnitt 4.5) bilden also jeweils eine eigene Komponente in der Layerstruktur [Hüning u. a. (2014)]. Das für das Testszenario verwendete Model der Agenten besteht aus drei wesentlichen Elementen [Macal und North (2011)]:

- **Attribute und Methoden**: Die Agenten besitzen jeweils statische und dynamische *Resource*-Attribute [Macal und North (2009)], die sie beschreiben und über die sich ihr Zustand definieren lässt. Ein Geparden-Agent besitzt zum Beispiel eine feste *ID* als statisches Attribut und eine *Position* als dynamisches Attribut, das sich während der Simulation verändert. Die Agenten verfügen außerdem über Methoden die ihr Verhalten abbilden und Methoden, die ihre dynamischen *Resource*-Attribute aktualisieren.

- **Relationen**: Die Agenten und Objekte stehen miteinander und mit ihrer simulierten Umwelt in einer topologischen Relation. Agenten und Objekte werden in einem *Grid*, also einem Raster welches wiederum aus Zellen besteht positioniert. Die Zellen, die einen Agenten oder ein Objekt direkt umgeben, stellen seine wahrnehmbare Umgebung dar. Über die aktuelle Position eines Agenten, ergeben sich die jeweils entsprechenden Relationen zu anderen Agenten und Objekten in der Umwelt.

- **Umwelt**: Die Agenten leben und interagieren mit anderen Agenten und Objekten in ihrer Umwelt. Geparden-Agenten können zum Beispiel Gazellen-Agenten jagen und diese ggf. auch erlegen. Wenn ein Gepard eine Gazelle erlegt hat, wird der entsprechende Gazellen-Agent aus der Umwelt entfernt.

Die Wahrnehmung der Umwelt und das Verhalten eines Agententyps, kann modular im Layer implementiert werden. Jeder Agent kann dann während der Simulation individuell seine Umwelt wahrnehmen und die entsprechenden Umweltinformationen auswerten, um darauf basierend seine Aktionen durchzuführen. Dabei gibt es zwei Kategorien von Aktionen. Zum einen gibt es Aktionen, die der Agent jedes mal ausführt, wenn er aktiv wird. In dem für diese Arbeit modellierten Testszenario wäre das zum Beispiel die Bewegungsaktion der Geparden und der Gazellen, mit der sie ihre Position ändern und sich durch das Simulationsgebiet bewegen. Zum anderen gibt es Aktionen, die der Agent nur ausführt, wenn die entsprechenden Bedingungen dafür gegeben sind, also ein spezieller Kontext vorherrscht. Ein Beispiel für eine Aktion eines Geparden-Agenten wäre das Verfolgen einer Gazelle, da diese nur ausgeführt wird, wenn sich eine Gazelle in der wahrnehmbaren Umgebung des Geparden befindet.

Die Ausführung der Agentenlogik, wird durch einen *Tick* angestoßen. Ein Tick ist ein externer Impuls, der in festen regelmäßigen Abständen bzw. einem getakteten Intervall ausgelöst wird. Alternativ können Aktionen der Agenten auch ereignisgesteuert über ihre Interfaces angestoßen werden. Dies geschieht, wenn eine durch ein Kontextmuster beschriebene Situation in der Simulation vom Context-Service erkannt wird. Bei jeder Simulationsiteration, also jedem Tick, können die Agenten und Objekte Ereignisse produzieren und diese als Nachrichten an den Context-Service übermitteln. Die Modellierung der versendeten Ereignisse und des Nachrichtenformats, werden im Verlauf dieses Kapitels noch näher erläutert.

Durch das in diesem Abschnitt erläuterte Design der Agenten und Objekte der Testsimulation, werden die Punkte *Context-Awareness* und *Ereignisse* der Anforderungen an die Simulationsumgebung aus Abschnitt 4.1 adressiert.

5.1.4. Integration in MARS

Ein Beispiel eines Runtime Deployments der MARS-Simulationsumgebung unter Verwendung der *SURVEYOR*-Komponente (Context-Service), ist in Abbildung 5.4 dargestellt. Hier ist noch einmal die Layerstruktur aus Abschnitt 5.1.2 veranschaulicht. Es zeigt zudem die in jedem Layer enthaltenen *Event Producer*, die für das Generieren von Ereignissen aus Kontextinformationen und den Versand der Ereignisse an den Context-Service zuständig sind.

Ebenfalls in Abbildung 5.4 zu sehen, ist der *Context Service Client*. Diese Teilkomponente von MARS-Surveyor stellt die Schnittstelle der Simulationsumgebung für die Kommunikation mit dem Context-Service Server dar und ist für die folgenden Aufgaben zuständig:

- die Initialisierung und den Aufbau der Nachrichtenverbindungen zur Message Oriented Middleware bzw. zum Context-Service Server
- die Registrierung und Bereitstellung der Nachrichtenverbindungen für die *Event Producer* in den einzelnen Layern, zur Kommunikation mit dem Context-Service Server
- die Registrierung neuer Ereignistypen beim Context-Service Server
- die Initialisierung und Bereitstellung des *Context Listener* (siehe Abbildung 5.4)
- die Registrierung neuer Kontextregeln beim Context-Service Server und die Verwaltung der zugehörigen Listener-Methoden im *Context Listener*

Jeder Layer in der Simulation verwendet einen eigenen *Event Producer*. Jeder Event Producer der durch den *Context Service Client* instanziiert wird, bekommt eine eigene Message Queue beim Message Broker zugewiesen, über die er mit dem Context-Service kommuniziert und Kontextereignisse senden kann. Dies dient der Parallelisierung des Versands von Ereignisnachrichten und somit der Skalierbarkeit.

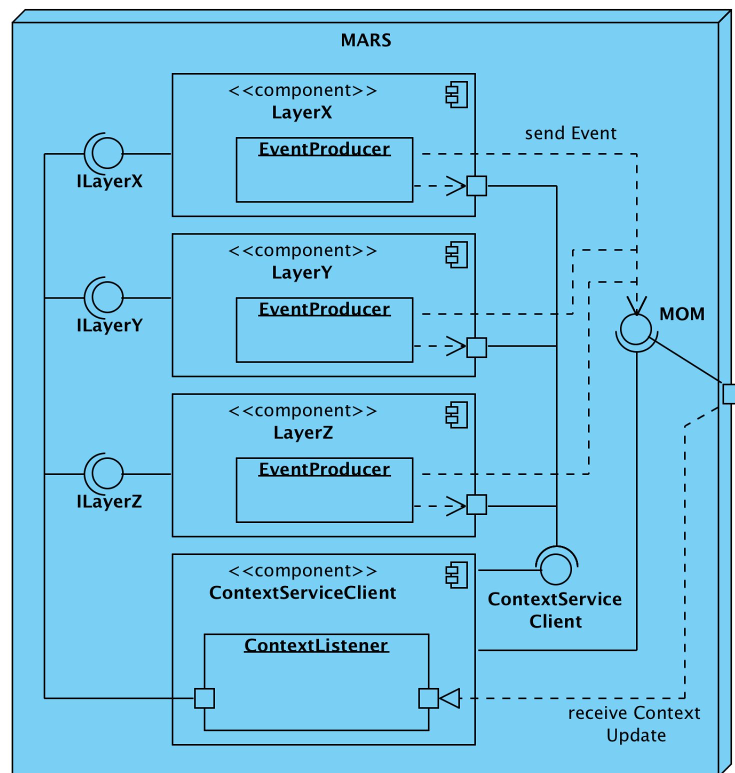


Abbildung 5.4.: Beispiel eines Runtime Deployment in MARS. Jeder Layer enthält einen eigenen *Event Producer*, mit dem Kontextereignisse an den Context-Service (MARS-Surveyor) gesendet werden können.

Der *Context Listener* der vom *Context Service Client* bereit gestellt wird, verarbeitet eingehende Nachrichten vom Context-Service Server. Die empfangenen Nachrichten des Context-Service zeigen an, dass ein Kontextmuster erkannt wurde. Die Nachrichten werden als Ereignisse wahrgenommen, auf die die Simulation dann mit dem anstoßen entsprechender Aktionen reagiert. Das System wird dadurch reaktiv. Es antwortet nicht auf explizite Anfragen, sondern es stößt Prozesse durch Reaktion auf Ereignisse an [Engel und Etzion (2011)].

5.2. MARS Surveyor

Das vorgestellte Konzept zur Entwicklung eines Context-Service für Multiagentensimulationen, soll die in den bereits bestehenden Arbeiten identifizierten Nachteile aus Kapitel 3 lösen. Aus diesem Grund, wurden bei der Konzipierung des Context-Service wichtige Designentscheidungen getroffen. Wie diese Designentscheidungen zur Lösung der in Kapitel 3 genannten Probleme beitragen, wird in den folgenden Abschnitten näher erläutert.

5.2.1. Systemarchitektur

Eine für das in dieser Arbeit entwickelte Context-Service Konzept sehr wichtige Designentscheidung, ist die vollständige Entkopplung des Context-Service von der Endanwendung (siehe Abbildung 5.5), mit Hilfe einer Message Oriented Middleware (MOM). Dadurch können gleich mehrere, der in Abschnitt 4.2 genannten Anforderungen an den Context-Service, erfüllt werden. Darüber hinaus, werden durch die Entkopplung außerdem auch mehrere Probleme von Gaia und JCAF aus Kapitel 3 gelöst und Vorteile gegenüber den Systemarchitekturen der beiden Ansätze erreicht.

Da die Kommunikation der Simulation mit dem Context-Service nun ausschließlich über Nachrichten stattfindet (siehe Abschnitt 6.2.2), bleibt die Endanwendung modular und die technische Umsetzung des Context-Service vollkommen transparent. Die bei JCAF verwendete Kommunikation per Java RMI und der dadurch entstehende Overhead beim Ressourcenverbrauch von Arbeitsspeicher entfällt, da auf der Seite des Context-Service ausschließlich die Verarbeitung der Kontextinformationen aus der Simulation stattfindet. Eine redundante Persistierung und Verwaltung der Agenten- und Objektinstanzen aus der Simulation und ihrer Daten im Context-Service ist dafür nicht nötig. Der Ressourcenverbrauch wird dadurch stark minimiert und es wird eine sehr lose Kopplung erreicht.

Ein weiterer Vorteil der dadurch entsteht ist, dass die Simulation nun in einer beliebigen Programmiersprache entwickelt werden kann. Um sich mit der Message Oriented Middleware und letztendlich mit dem Context-Service verbinden zu können, muss lediglich ein Message Client für das Versenden und Empfangen von Nachrichten in der Simulation vorhanden sein. Die Kontextinformationen können in der Simulation in einer beliebigen Sprache und einer beliebigen Struktur vorliegen, bevor sie mit Hilfe eines Adapters in übertragbare Ereignisnachrichten überführt werden. Somit müssen auf der Anwendungsseite auch keine Interfaces wie bei Gaia und JCAF mehr verwendet werden, um Kontextinformationen verarbeiten zu können.

Durch die Entkopplung des Context-Service über eine Message Oriented Middleware, wird die räumliche Trennung (*Space Decoupling*) [Eugster u. a. (2003)] von Ereignis produzierenden und Ereignis konsumierenden Anwendungen möglich. Die verschiedenen Anwendungen müssen dafür nicht von der Existenz der anderen Anwendungen wissen. Dadurch können mehrere Anwendungen auf Ereignisse reagieren, die von einer einzelnen Anwendung produziert wurden und umgekehrt. Der Context-Service kann somit von verschiedenen Anwendungen gleichzeitig angesprochen werden, Ereignisse empfangen und diese verarbeiten.

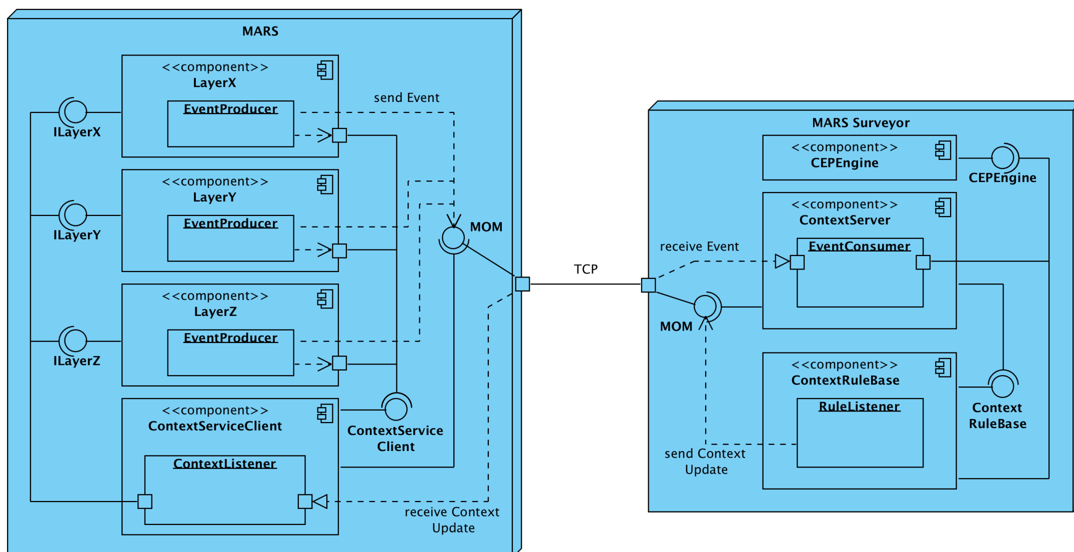


Abbildung 5.5.: Runtime Deployment von MARS und MARS-Surveyor (Context-Service)

Ein sehr wichtiger Grund für die Entkopplung ist letztendlich auch, dass dadurch wichtige Faktoren des Reactive Computing Paradigma (siehe Abschnitt 2.7.2) adressiert werden. Die Simulation kommuniziert mit dem Context-Service ereignisgetrieben (Event-Driven), asynchron und nicht-blockierend. Die Ereignisdaten werden von der Simulation an den Context-Service gesendet, sobald sie verfügbar sind (*Push*). Der Context-Service muss also nicht ständig bei der Simulation die Daten manuell abfragen (*Polling*). Dies gilt ebenso umgekehrt vom Context-Service in Richtung der Simulation. Dadurch wird verhindert, dass Ressourcen durch Polling oder das Warten auf Antworten verschwendet werden. Durch die nicht-blockierende Kommunikation, können kontinuierlich Kontextinformationen mit niedrigerer Latenz (Responsive) und höherem Durchsatz verarbeitet werden, als bei synchroner blockierender Kommunikation. Dadurch ist das System auch belastbarer und fehlertoleranter (Resilient). Wenn zum Beispiel Ereignisse verloren gehen, bleibt das System trotzdem funktionstüchtig.

5.2.2. Kontextmodellierung

Die Kontextmodellierung in diesem Ansatz basiert auf einem objektorientierten Kontextmodell [Bettini u. a. (2010); Strang und Linnhoff-Popien (2004)]. Die Kontextdaten der Simulation werden dabei als Ereignistypen verwaltet und verarbeitet. Ein Ereignistyp ist vergleichbar mit einer Klasse. Er spezifiziert also die Eigenschaften und die Struktur aller gleichartigen Ereignisse. Ein allgemeines Beispiel für einen Ereignistyp, der sechs Kontextattribute speichert, ist in Abbildung 5.6 dargestellt. Jede Ereignisinstanz dieses Ereignistyps hat dieselbe Struktur und Bedeutung. Die Ereignisse können in der Simulation in einem beliebigen Format vorliegen und als Nachrichten an den Context-Service übertragen werden. Die Übersetzung der Nachrichten, in ein für die CEP-Engine verständliches Ereignisformat, kann dann im Context-Service mit Hilfe eines Adapters (siehe Abschnitt 6.2.2) erfolgen. Dem Entwickler steht also völlig frei, in was für einer Struktur und in welchem Format die relevanten Kontextinformationen in der Simulation gespeichert und verwaltet werden. Zusätzlich entfällt die Notwendigkeit vorgegebene Interfaces und die damit implizit vorgegebene Programmiersprache zu verwenden. Insgesamt entsteht der Vorteil, dass auch eine bereits bestehende Simulation nachträglich ohne Probleme und ohne aufwändige Anpassungen Context-Aware gemacht werden kann.

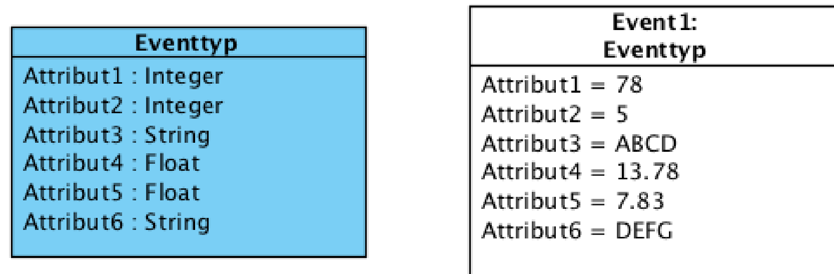


Abbildung 5.6.: Beispiel eines Ereignistyps und einer entsprechender Ereignisinstanz

Die Ereignistypen können unabhängig von der Implementierung der Simulation modelliert werden. Um eine Ereignisnachricht zu generieren, müssen die Informationen der in einem Ereignistyp spezifizierten Attribute lediglich an einer geeigneten Stelle in der Anwendung abgerufen werden können. Bei der in dieser Arbeit verwendeten Testsimulation wird zum Beispiel jedes mal, wenn ein Gepard seine Position ändert, ein Ereignis generiert. Dafür werden einfach die aktualisierten Attributwerte des Geparden-Agenten abgerufen und eine Ereignisnachricht erstellt. Diese Ereignisnachricht wird dann an den Context-Service gesendet. Diese Art der Kontextmodellierung erlaubt also eine leichte Integration bzw. Verwendung des Context-Service in neuen als auch in bereits bestehenden Simulationen.

5.2.3. Kontextverarbeitung

Der Context-Service stellt in der logischen Darstellung der Architektur des Gesamtsystems aus Abbildung 2.2 bzw. Abbildung 3.4, die kontextverarbeitende Komponente dar. Die Verarbeitung der als Ereignis modellierten Kontextinformationen aus der Simulation, ist die primäre und somit wichtigste Aufgabe des Context-Service. In diesem Schritt der Informationsverarbeitung, findet unter anderem das Context Reasoning statt (siehe Abschnitt 2.3), welches eine wichtige funktionale Anforderung an den Context-Service ist.

Sobald die Simulation die feingranularen und noch unverarbeiteten Kontextdaten erfasst und in Form von Ereignissen an den Context-Service gesendet hat, müssen diese ausgewertet werden. Um der Simulation bzw. der kontextkonsumierenden Anwendung danach verarbeitete und aggregierte Informationen bereitstellen zu können, kommt an dieser Stelle Complex Event Processing zum Einsatz (siehe Abschnitt 2.6). Durch die Verwendung der Complex Event Processing Technologie in der Context-Service Komponente, können die folgenden Operationen zur Kontextverarbeitung [Etzion und Niblett (2010)] angewendet werden:

- **Filterung**: Durch die Anwendung von Filtern, kann eine riesige Menge an Kontextinformationen auf eine kleinere, nach bestimmten Kriterien eingeschränkte Menge an Kontextinformationen reduziert werden. Überflüssige Informationen werden dadurch aussortiert.

- **Aggregation**: Durch Aggregation kann zum Beispiel die Summe, der Durchschnitt, das Maximum oder Minimum von Kontextinformationen ermittelt werden. Aggregation kann zusätzlich mit dem Einsatz von Filteroperationen kombiniert werden.

- **Matching**: Durch Matching können Kontextinformationen anhand von spezifischen Kriterien anderen Informationen zugeordnet werden, wie dies zum Beispiel auch in klassischen Datenbanken mit Hilfe von Fremdschlüsseln geschehen kann.

- **Merging**: Durch die Verwendung von Mergingoperationen können Kontextinformationen mit anderen Informationen zu einer einzelnen Informationsmenge zusammengeführt bzw. um diese erweitert werden.

- **Reasoning**: Mit Hilfe der anderen Operationen kann letztendlich Context Reasoning durchgeführt werden. Durch Reasoningoperationen, können aus den feingranularen unverarbeiteten Kontextinformationen neue Informationen mit höherer Aussagekraft hergeleitet werden.

Mit dieser Auswahl an anwendbaren Operationen auf Kontextinformationen, werden die bei JCAF fehlenden und bei Gaia eingeschränkten Möglichkeiten der Kontextverarbeitung bereits signifikant verbessert. Ein weiterer Grund für die Entscheidung Complex Event Processing für die Verarbeitung der Simulationsereignisse zu verwenden, ist die dadurch entstehende Möglichkeit, temporale Zusammenhänge zwischen den Kontextinformationen erkennen zu können (siehe Abschnitt 4.2.1). Die Einführung von Temporallogik ermöglicht somit eine Analyse von Kontextinformationen über einen längeren Zeitabschnitt Δt, statt nur zu einem singulären Zeitpunkt t, wie dies bei JCAF und Gaia der Fall ist. Der Zeitabschnitt, in dem die Analyse zusammenhängender Ereignismengen stattfindet, wird als *Sliding Window* bezeichnet [(Wu u. a., 2006)]. Die folgende Abbildung 5.7 stellt diesen Unterschied dar.

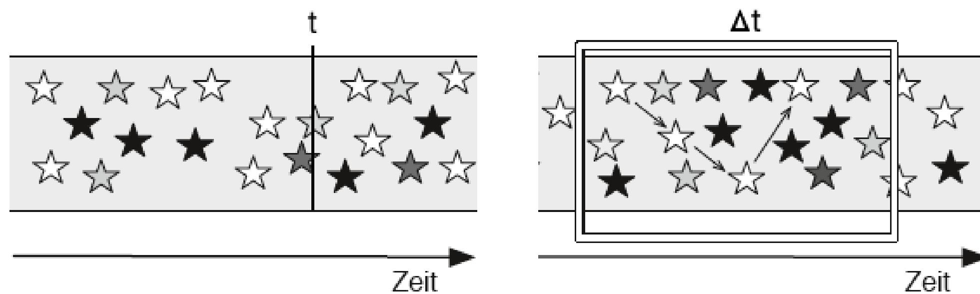


Abbildung 5.7.: Analyse der Ereignisse zu einem singulären Zeitpunkt t wie bei JCAF und Gaia (links) und über einen längeren Zeitabschnitt Δt mit CEP (rechts)

Ein zusätzliches ausschlaggebendes Argument für die Verwendung von Complex Event Processing sind die Punkte Skalierbarkeit und Performance der nichtfunktionalen Anforderungen an den Context-Service aus Abschnitt 4.2.2. Die Performance, um die möglicherweise sehr großen Ereignismengen einer Simulation in Echtzeit verarbeiten zu können, könnte durch den Einsatz von Complex Event Processing erreicht werden. Der Zwischenschritt, die gegebenenfalls riesigen Volumen an zu verarbeitenden Ereignissen zunächst zu speichern entfällt bei Complex Event Processing. Die Ereignisdaten werden on-the-fly gegen die in der Regelbasis des Context-Service persistierten Ereignisregeln geschickt (siehe Abbildung 2.7). Dadurch kann die Verarbeitungsgeschwindigkeit stark gesteigert werden, sodass große Mengen an Ereignissen pro Sekunde verarbeitet werden können. Dies muss jedoch noch mit Hilfe der Testumgebung unter verschiedenen Konfigurationen überprüft werden. Bei den Performancetests sollen zum Beispiel Eigenschaften, wie die Verarbeitungsgeschwindigkeit, der Durchsatz und die Skalierbarkeit des Context-Service genauer untersucht werden.

5.2.4. Abfrage von Kontext

Um eine Simulation in ihrem Verhalten an den aktuellen Kontext anpassen zu können, muss vom Context-Service eine Möglichkeit zur Abfrage des Kontexts und zur Definition von Kontextregeln zur Verfügung gestellt werden. Dies wird durch die Verwendung einer speziell für diese Aufgabe entwickelte Art von Abfragesprachen für Kontext, sogenannte *Context Query Languages* (CQLs) erreicht. Es wurden bereits mehrere verschiedene Arten von Context Query Languages, wie zum Beispiel XML-basierte, RDF-basierte, Graph-basierte oder SQL-basierte CQLs entworfen. Dabei sollte die verwendete Query Language einige wichtige Kriterien und Anforderungen erfüllen [Reichle u. a. (2008)]:

- Die Abfrage mehrerer Kontextdaten mit einer einzelnen Query sollte möglich sein. Zum Beispiel ID, Position und Energie aller simulierten Geparden.
- Es sollte möglich sein Filter in Abfragen spezifizieren zu können, um nur die Kontextdaten zu erhalten, die die entsprechenden Merkmale aufweisen. Zum Beispiel alle Geparden, die männlich sind.
- Die Angabe von temporalen und spatialen Bedingungen. Zum Beispiel alle Geparden, die in den letzten zehn Minuten in einem Gebiet zwischen $x1$, $y1$ und $x2$, $y2$ eine Gazelle gerissen haben.

Eine Übersicht und Bewertung der verschiedenen CQL Typen in Haghighi u. a. (2006) ergibt, dass sich neben den RDF-basierten Ansätzen die SQL-basierten Ansätze besonders gut eignen. In dem in dieser Arbeit entwickelten Context-Service Konzept, wird deshalb eine SQL-basierte Context Query Language verwendet. Dabei orientieren sich die Abfragen an der Basisstruktur gewöhnlicher SQL Querys, weshalb diese Art von Context Query Languages als SQL-basiert bezeichnet werden. Das folgende Beispiel zeigt wie eine Abfrage aussehen könnte, mit der die IDs aller männlichen Geparden die sich in einem Bereich zwischen 50 und 250 auf der x-Achse aufhalten, abgefragt werden können:

SELECT id **as** gepardId **FROM** GepardEvent **as** gepard **WHERE** gepard.gender='Male' **and** gepard.xCoordinate **between** 50 **and** 250

Mit der im Context-Service verwendeten SQL-basierten Context Query Language, lassen sich viele der in Haghighi u. a. (2006) genannten Aspekte von Kontext und der in Reichle u. a. (2008) genannten Anforderungen an eine gute CQL bedienen. Die verwendete Context Query Language, bietet damit ein deutlich umfangreicheres Spektrum an möglichen Abfragen von Kontext und Operationen auf Kontextdaten, als die bisherigen Ansätze aus Kapitel 3.

5.3. Ereignistypen

Die in der Simulation verwendeten Ereignistypen [Luckham (2001); Luckham und Schulte (2013)], bilden die Informationen der Agenten und Objekte ab, die über einen Zeitverlauf simuliert werden. In dem für die Testumgebung modellierten Beispielszenario sind dies unter anderem die Geparden, die Gazellen und die *KillEvents* (siehe Abschnitt 5.3.3) der Geparden. Man könnte zum Beispiel auch Veränderungen von simulierten Wasserstellen, kleineren Pflanzen, Bäumen, Parkrangern oder weiteren Tierarten auf entsprechende Ereignistypen abbilden.

Ziel dieser Abbildung von Simulationsereignissen, ist der Austausch von Kontextinformationen der jeweiligen Ereignistypen mit dem Context-Service. Jede Instanz eines Ereignistypen stellt dabei ein Ereignis, also eine Veränderung in der Simulation dar. Dies kann zum Beispiel die Positionsänderung eines Geparden von Position x nach Position $x + 5$ sein. Die während der Simulation an den Context-Service übertragenen Ereignisse, werden dann in einem Ereignisstrom (siehe Abbildung 6.3) aufbereitet, um Ereignismuster erkennen zu können. Im Folgenden werden die in dem Beispielszenario verwendeten Ereignistypen näher erläutert.

5.3.1. Geparden

Der Ereignistyp *GepardEvent* der Geparden, die in diesem Beispielszenario simuliert werden, verfügt über die folgenden Informationen, die vom Context-Service ausgewertet werden können:

- **Eindeutige ID**: Eine ID für die eindeutige Identifikation der simulierten Geparden. Die ID ist im *ID*-Feld jeder GepardEvent Instanz enthalten.
- **Position**: Die Position des Geparden während des Simulationsverlaufs ist als *xCoordinate* und *yCoordinate* Angabe gespeichert. Anhand der Position eines Geparden, kann dann zum Beispiel festgestellt werden, ob sich dieser auf einem Aussichtspunkt befindet und wie weit er von potenziellen Beutetieren entfernt ist.
- **Gerissene Gazellen**: Die Anzahl der Gazellen, die ein Gepard während der Simulation gerissen hat zeigt, wie erfolgreich der Gepard bei der Jagd ist. Die Anzahl der gerissenen Gazellen könnte verwendet werden um zu überprüfen, ob sich ein weiblicher Gepard innerhalb eines bestimmten Zeitraums in der Nähe eines männlichen Geparden aufgehalten hat und beide Geparden eine bestimmte Anzahl von Gazellen erlegt haben. Wenn dies zutrifft, könnte zum Beispiel als Reaktion ein weiterer Gepard als Nachwuchs zur Simulation hinzugefügt werden. Diese Information wird im *count*-Feld angegeben.

- **Geschlecht**: Das Geschlecht des Geparden ist im *gender*-Feld enthalten und gibt Aufschluss, ob ein Gepard männlich oder weiblich ist. Das Geschlecht kann Einfluss auf das Verhalten des Geparden haben. Die Gepardenweibchen Leben zum Beispiel eher alleine, während Gepardenmännchen meist Gruppen von zwei bis drei Tieren bilden. Dadurch haben weibliche Geparden einen kleineren Aktionsraum als männliche Geparden [Houser (2008)].
- **Alter**: Das Alter eines simulierten Geparden wird im *age*-Feld gespeichert. Anhand des Alters könnte zum Beispiel die maximale Energie eines Geparden variieren und mit einem hohen Alter langsam wieder abnehmen.

Abbildung 5.8 zeigt den Ereignistyp *GepardEvent* und zwei konkrete Ereignisinstanzen eines GepardEvents als Beispiel. Der aktuelle Zustand jedes Geparden, wird dann während der gesamten Simulation kontinuierlich in Form solcher Ereignisse an den Context-Service zur Auswertung übertragen und auf Kontextmuster untersucht.

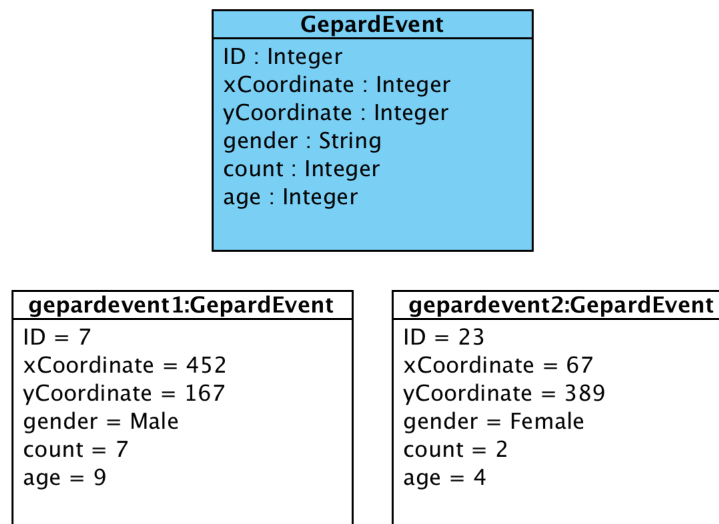


Abbildung 5.8.: *GepardEvent* Ereignistyp und -instanzen

5.3.2. Gazellen

Als Beutetiere für die Geparden, deren Jagdverhalten in diesem Beispielszenario simuliert werden soll, dienen Gazellen. Diese passen in das Beuteschema von Geparden. Größere Tiere, wie zum Beispiel Zebras und Gnus werden daher in der Beispielsimulation nicht betrachtet, da diese nicht als Beutetiere für die Geparden in Frage kommen. Damit der Context-Service

den Kontext der Gazellen auswerten kann, wurde der Ereignistyp *GazelleEvent* modelliert. Ein Ereignis dieses Ereignistyps enthält die folgenden Informationen:

- **Eindeutige ID**: Eine ID für jede der simulierten Gazellen, um diese eindeutig identifizieren zu können. Die ID ist im *ID*-Feld jeder GazelleEvent Instanz enthalten.
- **Position**: Die Position der Gazellen während des Simulationsverlaufs ist als *xCoordinate* und *yCoordinate* Angabe gespeichert. Anhand der Position einer Gazelle, kann dann zum Beispiel festgestellt werden, ob sich diese in Sichtweite oder sogar unmittelbarer Nähe eines Geparden aufhält.
- **Energie**: Die Energie der Gazelle beeinflusst, wie lange sie vor einem Geparden fliehen kann, wenn dieser sie mit einer hohen Geschwindigkeit verfolgt. Sie nimmt während der Flucht ab und kann sich in einer Ruhephase wieder regenerieren. Die aktuelle Energie wird im *energy*-Feld gespeichert. Wenn die Energie einer Gazelle niedriger ist als die Energie des sie jagenden Geparden, ist die Wahrscheinlichkeit geringer, dass sie dem Geparden entkommt.

Weitere Eigenschaften, wie zum Beispiel das Geschlecht und das Alter der Gazellen, werden in dieser Beispielsimulation nicht berücksichtigt.

5.3.3. Kill Events

Ein weiteres relevantes Ereignis in der Simulation wird auf den Ereignistyp *KillEvent* abgebildet. Dabei handelt es sich um das Ereignis, wenn ein Gepard eine Gazelle erlegt. Jedes mal wenn in der Simulation ein Gepard eine Gazelle gerissen hat, wird ein KillEvent an den Context-Service übertragen. Ein KillEvent beinhaltet folgende Informationen:

- **Gepard ID**: Die ID des simulierten Geparden, der die Gazelle erlegt hat, um diesen eindeutig identifizieren zu können.
- **Gazelle ID**: Die ID der simulierten Gazelle, die erlegt wurde, um diese eindeutig identifizieren zu können.
- **Position**: Die Position, an der der Gepard die Gazelle während des Simulationsverlaufs erlegt hat, ist als x und y Koordinate gespeichert. Anhand dieser Position, kann dann zum Beispiel festgestellt werden, in welchem Sektor des simulierten Gebiets die Gazelle erlegt worden ist.

6. Realisierung

Dieses Kapitel beschreibt den Aufbau und die Realisierung der *SURVEYOR*-Komponente des MARS-Frameworks, sowie die Realisierung der Testumgebung zur Durchführung der in Kapitel 7 beschriebenen Experimente. Zu Beginn wird ein Überblick über die Interfaces des MARS-Surveyor Clients gegeben, die zur Integration in eine Simulation mit MARS-Life verwendet werden können. Anschließend wird die konkrete Umsetzung der Testumgebung beschrieben, in der das in dieser Arbeit vorgestellte Context-Service Konzept zum Einsatz kommt. Es werden die grundlegenden Teilkomponenten genannt, aus denen sich die Testumgebung zusammensetzt und ihre jeweiligen Aufgaben betrachtet. Anschließend wird der genaue Aufbau des Context-Service und seiner Teilkomponenten beleuchtet. Grundlegende Funktionsweisen der erläuterten Teilkomponenten, werden anhand von Ausschnitten aus der Implementierung in entsprechenden Code-Listings veranschaulicht.

6.1. Context-Service Client

Um die *SURVEYOR*-Komponente des MARS-Frameworks in einer Simulation verwenden zu können, stehen die Interfaces des Context-Service Client zur Verfügung, mit denen der Context-Service (siehe Abschnitt 6.2.3) angesprochen werden kann. Der Context-Service Client besteht im wesentlichen aus den folgenden drei Klassen:

- ***ContextServiceClient***
- ***EventProducer***
- ***ContextListener***

Die Klassen sind in Abbildung 5.4 dargestellt und werden in den folgenden Abschnitten genauer erläutert. Die folgenden Listings zeigen außerdem die Instanziierung des *ContextServiceClient* und die Verwendung der grundlegenden Interfaces, um sich mit dem Context-Service zu verbinden und neue Ereignistypen und Kontextregeln registrieren zu können.

6.1.1. ContextServiceClient

Der *ContextServiceClient* ist primär für die folgenden Aufgaben zuständig:

- **Verbindungsaufbau**: Der Verbindungsaufbau zum Context-Service kann mit Hilfe der *ConnectTo*-Methode durchgeführt werden. Die *ConnectTo*-Methode erhält als Parameter den Host und den Port des Message Brokers, über den die Ereignisnachrichten an den Context-Service gesendet werden.

```
1 // Get ContextServiceClient instance
2 ContextServiceClient csc = ContextServiceClient.Instance;
3 // Connect to the Context-Service
4 csc.ConnectTo ("127.0.0.1", 5672);
```

Listing 6.1: Verbindungsaufbau zum Context-Service

- **Registrierung von Ereignistypen**: Nach dem Verbindungsaufbau können mit der *RegisterNewEventType*-Methode Ereignistypen beim Context-Service registriert werden. Dazu wird der Methode ein Objekt der jeweiligen Ereignisklasse, die registriert werden soll, als Parameter übergeben.

```
1 // Register new event types
2 csc.RegisterNewEventType(new GazelleEvent());
3 csc.RegisterNewEventType(new GepardEvent());
```

Listing 6.2: Registrieren neuer Ereignistypen beim Context-Service

- **Registrierung von Kontextregeln**: Nachdem der neue Ereignistyp registriert wurde, kann er in Kontextregeln verwendet werden. Kontextregeln können mit Hilfe der *RegisterNewContextRule*-Methode beim Context-Service angemeldet werden. Die Methode erhält neben der Kontextregel eine Delegate-Methode, die aufgerufen werden soll, wenn die Kontextregel erkannt wurde.

```
1 // Register a new context rule
2 csc.RegisterNewContextRule("select␣e1,␣e2␣from␣pattern␣[every
3 (e1=KillEvent(x␣<␣30,␣y␣<␣30)␣->␣e2=KillEvent(x␣>␣30,␣y␣>␣30))
4 where␣timer:within(10␣sec)]", Listener.Method);
```

Listing 6.3: Registrieren neuer Kontextregeln beim Context-Service

In der Delegate-Methode, kann dann der Handler-Code implementiert werden, der als Reaktion auf die erkannte Kontextregel ausgeführt werden soll (siehe Listing 6.4).

```
public static void Method(Dictionary<string, object> results) {
  // Do something
}
```

Listing 6.4: Delegate-Methode die aufgerufen wird wenn die zugehörige Kontextregel vom Context-Service erkannt wurde

6.1.2. EventProducer

Jeder fachlich getrennte Layer einer Simulation in MARS (siehe Abbildung 5.4), kann mittels einer *EventProducer*-Instanz Ereignisnachrichten an den Context-Service senden, in denen entsprechende Kontextinformationen enthalten sind. Jede *EventProducer*-Instanz bekommt eine eigene Message Queue zugewiesen, über die er Ereignisse an den Context-Service senden kann, sodass der parallele Versand aus mehreren Layern gleichzeitig möglich ist.

```
public EventProducer() {
  // Connect to message broker
  var factory = new ConnectionFactory () {
    HostName = ContextServiceClient.Instance.Host,
    Port = ContextServiceClient.Instance.Port
  };
  connection = factory.CreateConnection();

  // Declare new message queue for outgoing Events
  eventChannel = connection.CreateModel();
  queueName = string.Format ("queue{0}", ContextServiceClient
    .Instance.RegisterNewEventProducer());
  eventChannel.QueueDeclare(queueName);
}
```

Listing 6.5: Konstruktor der *EventProducer*-Klasse

Eine *EventProducer*-Instanz kann wie folgt erzeugt und zum Versand von Ereignisnachrichten verwendet werden:

```
EventProducer eventProducer = new EventProducer();

GazelleEvent gazelleEvent = new GazelleEvent(1, 5, 78, 11);
eventProducer.SendEvent(gazelleEvent);
```

Listing 6.6: Erzeugen einer neuen *EventProducer*-Instanz und Versenden eines *GazelleEvent*

6.1.3. ContextListener

Wenn der Context-Service eine registrierte Kontextregel erkennt, versendet er eine Trigger-Nachricht an die Simulation. Der *ContextListener* empfängt die Trigger-Nachrichten des Context-Service und ruft mittels eines Delegates die entsprechende Methode auf, die für die jeweils erkannte Kontextregel registriert wurde (siehe Listing 6.7). Die Zuordnung der Methode zur entsprechenden Kontextregel geschieht mit Hilfe eines Dictionarys.

```
public class ContextListener {
  ...
  public Dictionary<string, MethodDelegate> contextRuleDictionary;
  ...
  public void run() {
    while (true)
    {
      var ea = (BasicDeliverEventArgs)consumer.Queue.Dequeue();
      var body = ea.Body;
      string message = Encoding.UTF8.GetString(body);
      string[] result = message.Split (';');

      if(contextRuleDictionary.ContainsKey(result[0]))
      {
        Dictionary<string, object> results = JsonConvert
          .DeserializeObject<Dictionary<string, object>>(result[1]);
        contextRuleDictionary[result[0]].Invoke(results);
      }
    ...
}
```

Listing 6.7: Empfang einer Trigger-Nachricht vom Context-Service und Aufruf des zugehörigen Delegates als Reaktion

6.2. Testumgebung

In diesem Abschnitt wird nun die konkrete Umsetzung der Testumgebung beschrieben. Die Testumgebung besteht im wesentlichen aus einer Multiagentensimulation des Modells aus Abschnitt 4.3, einer *Message Oriented Middleware* zur Übertragung von Nachrichten und dem Context-Service. Mit der Testumgebung sollen letztendlich verschiedene Experimente mit unterschiedlichen Konfigurationen durchgeführt werden. Die entsprechenden Experimente werden in Kapitel 7 ausführlicher erläutert.

6.2.1. Simulation

Zur Implementierung und Visualisierung des in Abschnitt 4.3 beschriebenen Beispielszenarios, wurde das Simulationstool *Repast Simphony*[1] verwendet. Laut einer Evaluation von Railsback u. a. (2006), ist *Repast* eine gute Wahl für eine Realisierung und performante Simulation von komplexen Modellen. Repast Simphony ist ein Java basiertes und kostenlos erhältliches Open Source Simulationstool, mit dem Multiagentensimulationen modelliert und implementiert werden können. Es ist für alle gängigen Betriebssysteme wie Microsoft Windows, Mac OS und Linux verfügbar und unterstützt verschiedene Sprachen und Möglichkeiten um Simulationsmodelle zu entwickeln. Dazu gehören ReLogo, NetLogo, Java, Groovy sowie die Möglichkeit per Statecharts zu entwickeln. Dadurch kann die Modellierung von Simulationsszenarien, je nach Präferenz der Entwicklungssprache, durchgeführt werden. Als Beispiele für potenzielle Einsatzgebiete von Repast Simphony wird folgendes genannt:

> "Repast Simphony has been successfully used in many application domains including social science, consumer products, supply chains, possible future hydrogen infrastructures, and ancient pedestrian traffic to name a few."[2]

Neben der Unterstützung verschiedener Plattformen und Programmiersprachen, verfügt Repast Simphony über weitere nützliche Features. Dazu gehören zum Beispiel integrierte Möglichkeiten zur Visualisierung der entwickelten Simulationsmodelle (siehe Abbildung 6.1), zum automatischen Export von Simulationsdaten und zur Erstellung von Grafikcharts aus den Simulationsdaten. Als Entwicklungsumgebung kommt dabei Eclipse[3] zum Einsatz. Zu den grundlegenden Aufgaben der Simulationsumgebung zählt:

- die Simulation der Geparden- und Gazellen-Agenten
- das Produzieren und Versenden entsprechender Ereignisse (GepardEvent, GazelleEvent, KillEvent) als Nachrichten an den Context-Service
- die Visualisierung des Simulationsszenarios und die Reaktionen auf die vom Context-Service erkannten Kontexte

Die mit Repast Simphony realisierte Simulation des Beispielszenarios, wird mittels der Message Oriented Middleware *RabbitMQ* mit dem Context-Service verbunden (vgl. Abbildung 5.5).

[1] http://repast.sourceforge.net/repast_simphony.php
[2] http://repast.sourceforge.net/repast_simphony.php
[3] http://www.eclipse.org/downloads/

Agenten

Dieser Abschnitt erläutert die Implementierung des Geparden-Agenten und seiner Steuerungslogik als Beispiel für einen in der Simulation verwendeten Layer (siehe Abschnitt 5.1.2). Als Grundlage dafür dient eine generische Java Klasse (*Java Bean*). Das folgende Listing zeigt einen Ausschnitt aus der Implementierung, der wichtige Stellen der Geparden-Klasse enthält.

```
public class Gepard {
    ...
    private ContextServiceClient contextClient =
    ContextServiceClient.getInstance();

    public Gepard(ContinuousSpace<Object> space, Grid<Object> grid)
    {
        this.space = space;
        this.grid = grid;
    }

    @ScheduledMethod(start = 1, interval = 1)
    public void step() {
        // Send GepardEvent Message to Context-Service
        ...
        if (allowMove) {
            moveTowards(pointWithMostGazelles);
        }
        kill();
    }

    public void kill() { /* Send KillEvent if a kill happened*/ }
```

Listing 6.8: Ausschnitt aus der Implementierung des Gepard-Agenten

Die Geparden-Agenten bilden einen Ereignisse produzierenden Layer der Simulation ab. Daher besitzt jeder Gepard-Agent neben den in 5.3 spezifizierten Attributen eine *ContextServiceClient*-Referenz, damit er Ereignisse als Nachrichten an den Context-Service übermitteln kann (siehe Abschnitt 6.2.2). Dies geschieht unter anderem in der *step()*-Methode, die im Abstand eines festgelegten Intervalls aufgerufen wird. Neben dem Versenden eines *GepardEvent* an den Context-Service, wird dort auch die Bewegung des Geparden durch das simulierte Gebiet berechnet und die *kill()-Methode* aufgerufen. In der *kill()-Methode* wird geprüft, ob der Gepard eine Gazelle erlegt hat. Ist dies der Fall, dann wird dort ein *KillEvent* generiert und an den Context-Service gesendet.

Visualisierung

Das User Interface der Simulationsoberfläche von Repast Simphony und die Visualisierung des Beispielszenarios, ist in Abbildung 6.1 zu sehen. In der Visualisierung der Simulation, werden die folgenden Agenten und Objekte dargestellt:

- **Gazellen**: Die Gazellen sind als hellbraune Kreise dargestellt.
- **Geparden**: Die Geparden werden durch rote Quadrate gekennzeichnet.
- **Aussichtspunkte**: Aussichtspunkte werden als X-Markierungen visualisiert.
- **Wasserstellen**: Die Wasserstellen werden als blaue Flächen angezeigt.

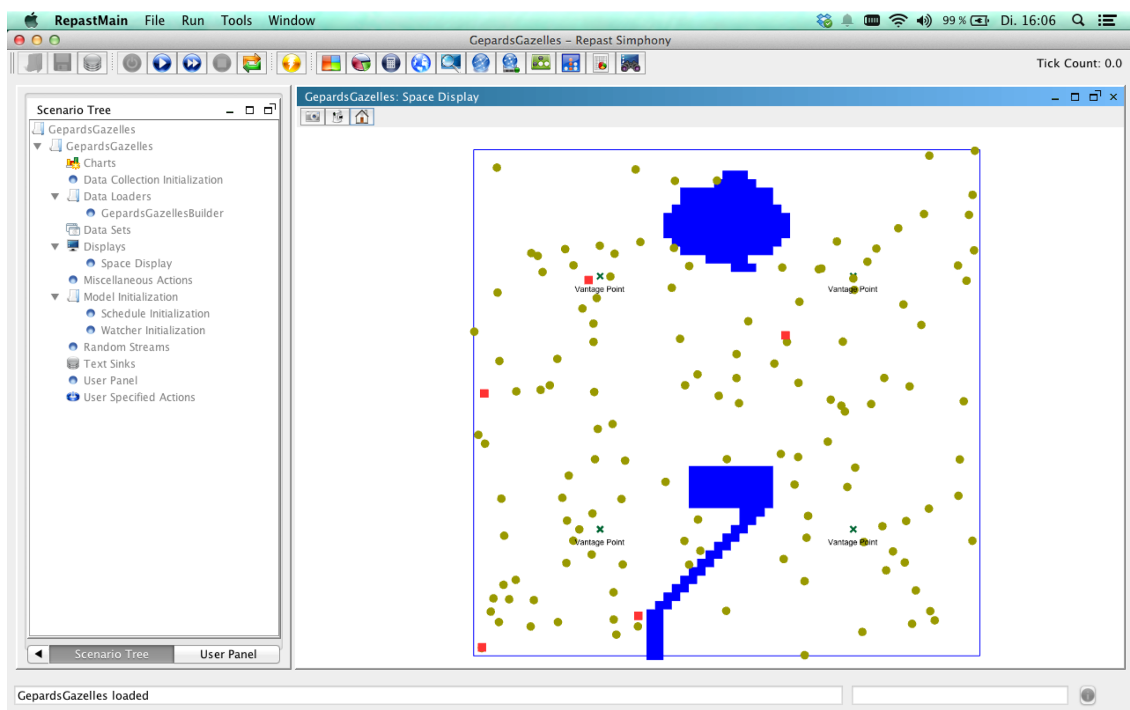


Abbildung 6.1.: Repast Simphony User Interface und Visualisierung des Simulationsszenarios

Die Geparden und Gazellen wurden als Agenten modelliert [Macal und North (2009)]. Während der Simulation wird die Bewegung der Geparden und der Gazellen bei jedem Simulationstick berechnet, aktualisiert und visualisiert. Die Aktionen, die durch die Erkennung von Kontextmustern in der Simulation initiiert werden, können dadurch in der Visualisierung entsprechend mitverfolgt werden.

6.2.2. Nachrichtenkommunikation

Zur Kommunikation zwischen dem Context-Service und der Multiagentensimulation, wird die Open Source Message Oriented Middleware *RabbitMQ*[4] verwendet. Dies dient zur Entkopplung der Kontext-Komponente vom Rest der Anwendung. Über die Middleware werden Kontextdaten als Nachrichten an den Context-Service übermittelt und dessen Antwortnachrichten von der Multiagentensimulation empfangen.

Übertragung von Ereignisnachrichten

Während der Simulation werden in den jeweiligen Simulationslayern Ereignisse generiert, die dann mit Hilfe der *RabbitMQ*-Middleware an den Context-Service übertragen werden. Der folgende Codeausschnitt (siehe Listing 6.9) zeigt das Generieren und Versenden einer Ereignisnachricht des Typs *GepardEvent* (siehe Abbildung 5.8) in der Testsimulation.

```java
// Send GepardEvent Message to the Context-Service

  String eventType = Integer.toString(2);
  String gepardId = Integer.toString(this.id);
  String gepardX = Integer.toString(this.x);
  String gepardY = Integer.toString(this.y);
  String gepardMale = Boolean.toString(this.male);
  String gepardCount = Integer.toString(this.count);
  String gepardAge = Integer.toString(this.age);
  String payload = eventType + "," + gepardId + ","
                   + gepardX + "," + gepardY + "," + gepardMale
                   + "," + gepardCount + "," + gepardAge;

  try {
    contextClient.getChannel().basicPublish("",
    ContextServiceClient.getQueueName(), null,
    payload.getBytes());
  }
  catch (IOException e) {
    e.printStackTrace();
  }
```

Listing 6.9: Codieren und Senden von *GepardEvent*-Ereignisnachrichten

[4] http://www.rabbitmq.com/download.html

Empfangen von Ereignisnachrichten

Das Empfangen von Ereignisnachrichten auf der Context-Service Seite, geschieht mit Hilfe von *EventConsumer*-Instanzen. Die in dem jeweiligen Layer (siehe Abbildung 5.4) der Multiagentensimulation versendeten *ByteMessages*, werden hier empfangen und dekodiert. Über den *EventType* wird ermittelt, um welche Art von Ereignistyp es sich bei dem empfangenen Ereignis handelt. Es wird anschließend jeweils ein entsprechendes Event-Objekt erzeugt, welches die Informationen der empfangenen Ereignisnachricht speichert. Dieses Event-Objekt wird dann an die CEP-Engine übergeben und verarbeitet. Der folgende Codeausschnitt (siehe Listing 6.10) zeigt den beschriebenen Verarbeitungsablauf von empfangenen Ereignisnachrichten.

```
1  public class EventConsumer extends Thread {
2    ...
3    public void run() {
4      while (true) {
5        ...
6        String message = new String(delivery.getBody());
7        String[] payload = message.split(";");
8        this.processEvent(payload[1]);
9      }
10   }
11
12   public int processEvent(String eventMessage) {
13     try {
14       Map<String, Object> event = new HashMap<String, Object>();
15       JsonParser jsonParser = new JsonParser();
16       JsonObject eventJson = jsonParser.parse(eventMessage)
17         .getAsJsonObject();
18       String eventType = eventJson.get("EventType").getAsString();
19       ...
20       engine.sendEvent(event, eventType);
21       return 1;
22     } catch (Exception e) {
23       System.out.println("␣[.]␣" + e.toString());
24       return -1;
25     }
26   }
27 }
```

Listing 6.10: Empfang und Decodierung von Ereignisnachrichten

6.2.3. Context-Service

Der Context-Service empfängt kontinuierlich die Snapshots des aktuellen Simulationszustands in Form von Ereignissen und analysiert diese auf Kontextmuster. Der Aufbau des Context-Service, der in Abbildung 6.2 dargestellt ist, kann grob in vier wesentliche Bestandteile unterteilt werden. Diese werden im Folgenden genannt und erläutert:

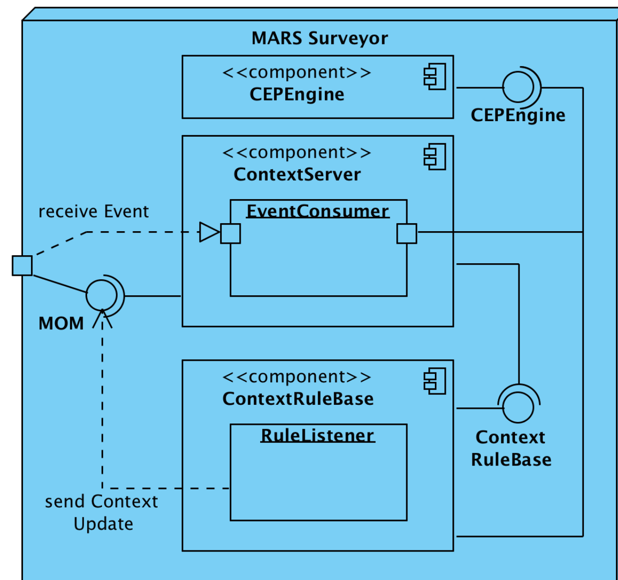


Abbildung 6.2.: Runtime Architektur des Context-Service (MARS-Surveyor)

Context-Server

Die Context-Server Komponente des Context-Service übernimmt viele wichtige Aufgaben, wie zum Beispiel:

- die Initialisierung und den Aufbau der ein- und ausgehenden Verbindungen zur Message Oriented Middleware (hier *RabbitMQ*) bzw. zur Simulation
- die Initialisierung und Konfiguration der Complex Event Processing-Engine, sowie die Registrierung der in der Simulation verwendeten Ereignistypen bei der CEP-Engine
- die Initialisierung der Regelbasis (*ContextRuleBase*) und die Registrierung der dort hinterlegten Kontextregeln bei der CEP-Engine
- die Initialisierung von Event Consumern (*EventConsumer*) und die Registrierung dieser auf die entsprechende Message-Queue für eingehende Ereignisnachrichten

CEP-Engine

Die zentrale Rolle des Context-Service, spielt die Komponente der Complex Event Processing-Engine. Alle Ereignisse, die während des Simulationsverlauf auftreten, werden hier verarbeitet und kontinuierlich auf die in der Regelbasis definierten Kontextregeln überprüft. Als CEP-Engine wird hier die Open Source Version von *Esper*[5] verwendet. Sie gilt unter dem aktuell auf dem Markt verfügbaren Angebot an CEP-Produkten, als die führende Open Source Lösung für Complex Event Processing [Cugola und Margara (2012)].

Die CEP-Engine des Context-Service hat primär die Aufgabe aus dem eingehenden Datenfluss, der sich aus einer ggf. riesigen Menge von Ereignissen der Multiagentensimulation zusammensetzt, die relevanten Informationen herauszuziehen. Am wichtigsten dabei ist das Erkennen von komplexen Ereignissen (siehe Abschnitt 2.6.3). Abbildung 6.3 stellt den Ereignisstrom (*Event Stream*) dar, der sich aus den kontinuierlich eingehenden Ereignissen der Simulation bildet. Die CEP-Engine analysiert diesen Ereignisfluss on-the-fly auf die in der Regelbasis definierten Ereignismuster.

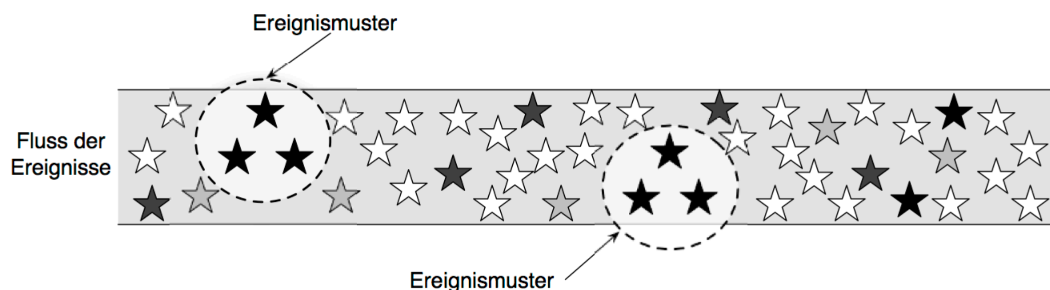


Abbildung 6.3.: Mustererkennung im Ereignisstrom (*Event Stream*) [Bruns und Dunkel (2010)]

Unmittelbar nach dem Erkennen eines komplexen Ereignismusters, soll eine Aktion initiiert werden, die als Reaktion auf das erkannte Ereignismuster folgt. Die geschieht hier mit Hilfe eines *RuleListeners* (siehe Abschnitt 6.2.3), der auf die entsprechende Ereignisregel registriert ist und eine Trigger-Nachricht an die Simulation sendet. Neben der allgemeinen Information, dass ein definierter Kontext eingetreten ist, können auch Detailinformationen der beteiligten Events an die Simulation übermittelt werden. Die Trigger-Nachricht enthält dann neben der ID der erkannten Regel weitere Informationen, die für die in der Simulation zu initiierende Reaktion relevant sind. Dies könnten zum Beispiel die IDs der beteiligten Agenten sowie deren Position sein.

[5] http://esper.codehaus.org/esper/download/download.html

ContextRuleBase

Hier werden die Kontextregeln hinterlegt, welche die Ereignismuster definieren, die erkannt werden sollen und bei deren Erkennung eine entsprechende Reaktion in der Simulation ausgeführt werden muss. Die Kontextregeln werden in Form eines Context Query Language-Statements angelegt. Alle Kontextregeln werden bei der CEP-Engine registriert und jeder Regel wird ein *RuleListener* zugewiesen. Dieser reagiert automatisch, wenn der entsprechende Kontext der ihm zugewiesenen Regel in der Simulation erkannt wurde. Das Hinzufügen von Kontextregeln, ihre Registrierung bei der CEP-Engine sowie das Zuweisen eines *RuleListener*, ist im folgenden Listing 6.2.3 gezeigt.

```
1  public class ContextRuleBase {
2
3    private EPServiceProvider engine;
4    private int ruleID;
5    private RuleListener ruleListener;
6
7    public ContextRuleBase(EPServiceProvider engine, String host,
8        int port) throws IOException {
9      this.engine = engine;
10     this.ruleID = 0;
11     ruleListener = new RuleListener(host, port);
12   }
13   ...
14   public int addContextRule(String contextRule) {
15     try {
16       ruleID++;
17       EPStatement stmt = engine.getEPAdministrator().createEPL(
18           contextRule, Integer.toString(ruleID));
19       stmt.addListener(ruleListener);
20       return ruleID;
21     } catch (Exception e) {
22       System.out.println("␣[.]␣" + e.toString());
23       return -1;
24     }
25   }
26   ...
27 }
```

Listing 6.11: Initialisierung der *ContextRuleBase* und Registrieren von Kontextregeln

Rule Listener

Ein *RuleListener* reagiert automatisch, sobald ein in der Regelbasis hinterlegtes Ereignismuster erkannt wird. Hier kann dann eine entsprechende Antwortnachricht generiert werden, die festlegt, welche Prozedur in der Simulation aufgerufen werden soll. Bei Bedarf können neue komplexe Ereignisse erzeugt und direkt wieder zur Auswertung an die CEP-Engine geschickt werden. Das folgende Listing 6.12 zeigt eine Variante eines Rule Listeners, der eine Trigger-Nachricht an die Simulation sendet. Die Nachricht enthält als Payload neben weiteren Informationen die ID der Kontextregel, die von dem Context-Service erkannt wurde. Die Simulation kann dadurch die jeweils zutreffende Prozedur als Reaktion auf die Trigger-Nachricht ausführen.

```
1  public void update(EventBean[] newEvents, EventBean[] oldEvents,
2        EPStatement rule, EPServiceProvider epsprovider) {
3      ...
4      String ruleName = rule.getName();
5
6      MapEventBean meb = (MapEventBean) newEvents[0];
7      String json = "";
8      if (meb.getProperties().size() > 1) {
9        Gson gson = new Gson();
10       HashMap<String,Object> result = new HashMap<String,Object>();
11       for (Map.Entry<String, Object> entry : meb.getProperties()
12           .entrySet()) {
13         MapEventBean value = (MapEventBean) entry.getValue();
14         result.put(entry.getKey(), value.getProperties());
15       }
16       json = gson.toJson(result);
17     }
18     ...
19     String message = ruleName + ";" + json;
20
21     try {
22       channel.basicPublish("", QUEUE_NAME, null, message.getBytes());
23     } catch (IOException e) {
24       e.printStackTrace();
25     }
26   }
```

Listing 6.12: Versenden einer Trigger-Nachricht an die Simulation

7. Experimente

In diesem Kapitel werden nun die Experimente erläutert, die mit dem im vorangegangenen Kapitel realisierten Context-Service durchgeführt werden sollen. Anhand der Ergebnisse sollen die in Abschnitt 1.1.1 aufgestellte Hypothesen verifiziert oder widerlegt werden können. Es werden die entsprechenden Messgrößen, Testszenarien und Konfigurationen beschrieben, die jeweils für die Verifizierung der Hypothesen notwendig sind.

7.1. Messgrößen

Dieser Abschnitt betrachtet zunächst die Messgrößen, die unter anderem zum Beispiel Aufschluss über die allgemeine Performance und den konkreten Durchsatz von Simulationsereignissen in der Context-Service Komponente geben sollen. Es werden die folgenden Messgrößen verwendet und protokolliert:

- **Ereignisanzahl**: Die Anzahl der Ereignisse, die von der Context-Service Komponente während des Simulationsverlaufs empfangen und verarbeitet werden. Mit Hilfe dieser Messgröße soll ermittelt werden, wie performant der Context-Service arbeitet und wie er sich bei steigender Belastung verhält.
- **Kontextregelanzahl**: Die Anzahl der während der Simulation in der Regelbasis hinterlegten Kontextregeln, die die zu erkennenden Kontextmuster definieren. Diese Messgröße soll Aufschluss darüber geben, ob und wie stark sich die Anzahl der hinterlegten Regeln auf die Verarbeitungsgeschwindigkeit auswirken kann.
- **Regelkomplexität**: Die Komplexität der in der Regelbasis hinterlegten Regeln soll ebenfalls für jede Testkonfiguration dokumentiert werden. Damit soll untersucht werden, welchen Einfluss die Komplexität der während der Simulation zu erkennenden Kontextmuster, auf die Performance des Context-Service hat.

- **Threadanzahl**: Die Anzahl verwendeter Threads soll in den entsprechenden Tests dokumentiert werden, um einschätzen zu können, wie sich diese auf die Performance des Context-Service auswirkt.
- **Zeit**: Für die verschiedenen Performancemessungen, wie zum Beispiel die Verarbeitungsgeschwindigkeit von Ereignissen, müssen die Messungen mit einem Zeitfaktor ins Verhältnis gesetzt werden. Ein Beispiel wäre hier die Messung der verarbeiteten Simulationsereignisse pro Sekunde.

7.2. Testszenarien

Um einzelne Eigenschaften des Context-Service bezüglich der aufgestellten Hypothesen evaluieren zu können, wurden verschiedene Testszenarien bzw. Experimente mit unterschiedlichen Konfigurationen entworfen. Alle Experimente wurden in *Java Virtual Machines* (JVMs) der aktuellen Java Version 7 durchgeführt. Für die Performancetests wurde als Hardware ein Rechner mit einem Intel Core i7-2600k Prozessor und 8 GB DDR3-RAM verwendet. Die CPU verfügt über 4 Kerne mit einer Taktung von jeweils 3,4 GHz pro Kern. Sie unterstützt somit die parallele Ausführung von 4 Threads bzw. 8 Threads bei Hyper-Threading.

7.2.1. Kontexterkennung

Bei diesem Test geht es um die Überprüfung der Funktionstüchtigkeit der grundlegenden Aufgabe des Context-Service, nämlich die Erkennung von Kontexten und Teilkontexten in einer Multiagentensimulation. Mit Hilfe dieses Tests, soll außerdem Hypothese 4 aus Abschnitt 1.1.1 validiert bzw. widerlegt werden. Dabei werden mehrere Aspekte betrachtet, die im Folgenden näher erläutert werden:

- **Temporale Kontexte**: Erkennung von zeitlichen Abhängigkeiten, Zusammenhängen oder Korrelationen zwischen Ereignissen in der Simulation. Dazu gehört die Analyse von Kontextmustern über einen zeitlichen Verlauf. Dabei wird sowohl das Auftreten von Mustern innerhalb von Zeitfenstern (siehe Abbildung 5.7), als auch die Erkennung von zeitlichen Abläufen selbst (siehe Abbildung 2.8) überprüft.
- **Spatiale Kontexte**: Betrifft die Erkennung von Kontextmustern zwischen räumlich (spatial) getrennten Ereignissen, die ggf. weit auseinander liegen. Hier geht es vor allem auch um die globale Erkennung von Kontexten in der Simulation, die über die räumliche Wahrnehmung der einzelnen Agenten in ihrem Sichtbereich hinausgeht.

- **Spatiotemporale Kontexte**: Hier geht es um die Erkennung von Kontextmustern, bei denen räumliche und zeitliche Zusammenhänge in Kombination betrachtet werden müssen.
- **Regelkomplexität**: Die Komplexität der in der Regelbasis hinterlegten Regeln kann variieren (siehe Beispiele in Abschnitt 4.7.1). Hier soll geprüft werden, dass sowohl einfache als auch komplexe Kontextsituationen in der Simulation gleichermaßen zuverlässig vom Context-Service erkannt werden.
- **Kontextoperationen**: Hier soll überprüft werden, wie gut sich entsprechende Operationen, wie zum Beispiel explizite Filter, bei der Abfrage von Kontext anwenden lassen und ob diese zuverlässig funktionieren.

Test

Um den Context-Service bezüglich der zuvor genannten Aspekte zu testen, wurden mehrere Kontextregeln in der Regelbasis (*ContextRuleBase*) definiert, die einen oder mehrere dieser Aspekte adressieren. Die ausgewählten Kontextregeln steigern sich dabei jeweils in ihrer Komplexität. Im Folgenden werden die definierten Regeln genannt und erläutert, was sie jeweils bezwecken:

- **Regel 1**:

```
1 "select * from KillEvent"
```

  Diese Regel ist die simpelste Form einer Kontextregel. Sie bezweckt die Benachrichtigung über jedes *KillEvent*, welches in der Simulation auftritt. Jedoch wird mit dieser Kontextregel bereits eine Filteroperation durchgeführt, da von allen auftretenden Ereignissen nur die KillEvents ausgewählt werden. Diese Regel löst jedes mal aus, wenn in der Simulation eine Gazelle von einem Geparden gerissen wird.

- **Regel 2**:

```
1 "select * from GepardEvent where x between 0 and 5"
```

  Diese Kontextregel ist ebenfalls relativ simpel aber beinhaltet bereits spatiale Constraints. Sie bezweckt die Benachrichtigung über jedes *GepardEvent*, welches im simulierten Raum zwischen den x-Koordinaten 0 und 5 auftritt. Von allen in der Simulation auftretenden Ereignissen, werden also nur die heraus gefiltert, die einen entsprechenden x-Wert aufweisen.

- **Regel 3**:

```
"select count(*) from pattern [[5]
KillEvent(x < 30, y < 30) where timer:within(3 min)]"
```

  Bei dieser Kontextregel werden neben spatialen Constraints nun zusätzlich auch temporale Constraints angegeben. Das Ereignismuster (*Pattern*), welches hier beschrieben wird, muss innerhalb von einem vorgegebenen Zeitfenster auftreten. Konkret löst diese Regel aus, wenn innerhalb von drei Minuten fünf Gazellen in dem Gebiet zwischen den x-Koordinaten 0 und 30 und den y-Koordinaten 0 und 30 gerissen werden.

- **Regel 4**:

```
"select count(*) from pattern [every
(timer:interval(30 sec) and not ([5] KillEvent))]"
```

  Mit dieser Kontextregel soll überprüft werden, ob die Abwesenheit bzw. das Nichtauftreten von Ereignissen vom Context-Service erkannt wird. Wenn in der Simulation innerhalb von einem 30 Sekunden Zeitfenster nicht fünf *KillEvents* aufgetreten sind, wird die Regel ausgelöst.

- **Regel 5**:

```
"select e1, e2 from pattern [every (e1=KillEvent(x < 30, y < 30)
-> e2=KillEvent(x > 30, y > 30)) where timer:within(10 sec)]"
```

  Diese Kontextregel definiert ein komplexeres Ereignismuster, bei dem zunächst zwei KillEvents $e1$ une $e2$ nach unterschiedlichen Koordinatenwerten aus der gesamten Ereignismenge gefiltert werden. Zusätzlich müssen zwei temporale Constraints zutreffen. KillEvent $e1$ muss vor KillEvent $e2$ stattgefunden haben und zwar innerhalb von zehn Sekunden. Treffen all diese Bedingungen zu, wird die Regel ausgelöst.

Beim dem durchgeführten Test, wurden alle in der Regelbasis hinterlegten Kontextregeln erfolgreich und zuverlässig vom Context-Service erkannt. Immer wenn ein entsprechend definiertes Ereignismuster in der Simulation aufgetreten ist, wurde die Simulation mit Hilfe einer Nachricht vom Context-Service darüber informiert. Falls nötig konnte dann in der Simulation eine entsprechende Reaktion angestoßen werden. Im Test wurden in der Simulation zum Beispiel jedes mal zehn neue Gazellen-Agenten zur Simulation hinzugefügt, sobald Regel 3 erkannt wurde.

7.2.2. Performance

Um Aussagen zur Performance des Context-Service Konzepts machen zu können, wurden mehrere Messungen mit unterschiedlichen Konfigurationen durchgeführt. Die Messungen sollen Aufschluss darüber geben, wie viele Kontextereignisse der Context-Service verarbeiten kann und wie sich die verschiedenen Konfigurationsparameter auf die Verarbeitungsgeschwindigkeit auswirken. Die verwendeten Konfigurationsparameter werden im Folgenden näher erläutert.

Ereignisanzahl

Es werden unterschiedliche Mengen von Ereignissen zur Verarbeitung an den Context-Service gesendet und die entsprechend benötigte Verarbeitungszeit dokumentiert. Dazu werden Messungen zu den Ereignismengen 100.000, 200.000, 300.000, 400.000 und 500.000 durchgeführt. Die Ereignisanzahl wird dann jeweils mit weiteren Parametern zu unterschiedlichen Konfigurationen kombiniert.

Regelanzahl

Bei den Messungen werden unterschiedliche Anzahlen von Kontextregeln in der Regelbasis des Context-Service registriert. Dadurch soll untersucht werden, ob und in welchem Ausmaß die Menge der zu überprüfenden Kontextregeln einen Einfluss auf die Verarbeitungsgeschwindigkeit hat. Es werden Konfigurationen mit 10, 100, 1.000 und 10.000 Regeln verwendet.

Regelkomplexität

Neben der Anzahl, der in der Regelbasis des Context-Service registrierten Kontextregeln, wird auch die Komplexität der Regeln variiert. Dabei werden drei verschiedene Komplexitätsstufen verwendet:

- **Leicht**: Bei dieser Konfiguration, werden Kontextregeln in Form von *Regel 1* aus Abschnitt 7.2.1 eingesetzt. Hier kommen noch keine besonderen Auswahlkriterien zum Einsatz, wie zum Beispiel eine spatiale oder temporale Angabe.
- **Mittel**: Regeln mittlerer Komplexität sind wie *Regel 3* aus Abschnitt 7.2.1 gestaltet. Hier werden bereits spatiale Bedingungen Bedingungen angegeben. Außerdem kann auch eine temporale Eingrenzung gegeben werden.

- **Komplex**: Komplexe Kontextregeln orientieren sich an der Form von *Regel 5* aus Abschnitt 7.2.1. Es werden mehrere Ereignisse auf spatiale und mehrere temporale Korrelationen bzw. Zusammenhänge überprüft.

Auslastung

Bei den Messungen werden auch verschiedene Auslastungsstufen als Konfigurationsparameter berücksichtigt. Die Auslastungsstufen beschreiben jeweils, auf wie viele empfangene und verarbeitete Kontextereignisse im Durchschnitt eine Benachrichtigung vom Context-Service erfolgt. Es werden drei verschiedene Auslastungsstufen verwendet:

- **Niedrige Auslastung**: Bei niedriger Auslastung, wird im Durchschnitt pro 500 verarbeiteten Kontextereignissen, eine Benachrichtigung vom Context-Service versendet. Das sind bei einer Menge von 500.000 Ereignissen 1.000 Benachrichtigungen über erkannte Kontextregeln.

- **Normale Auslastung**: Bei normaler Auslastung, wird im Durchschnitt nach jeweils 50 Kontextereignissen die verarbeitet wurden, eine Benachrichtigung vom Context-Service versendet. Bei einer Menge von 500.000 Ereignissen entspricht das 10.000 Benachrichtigungen über erkannte Kontextregeln.

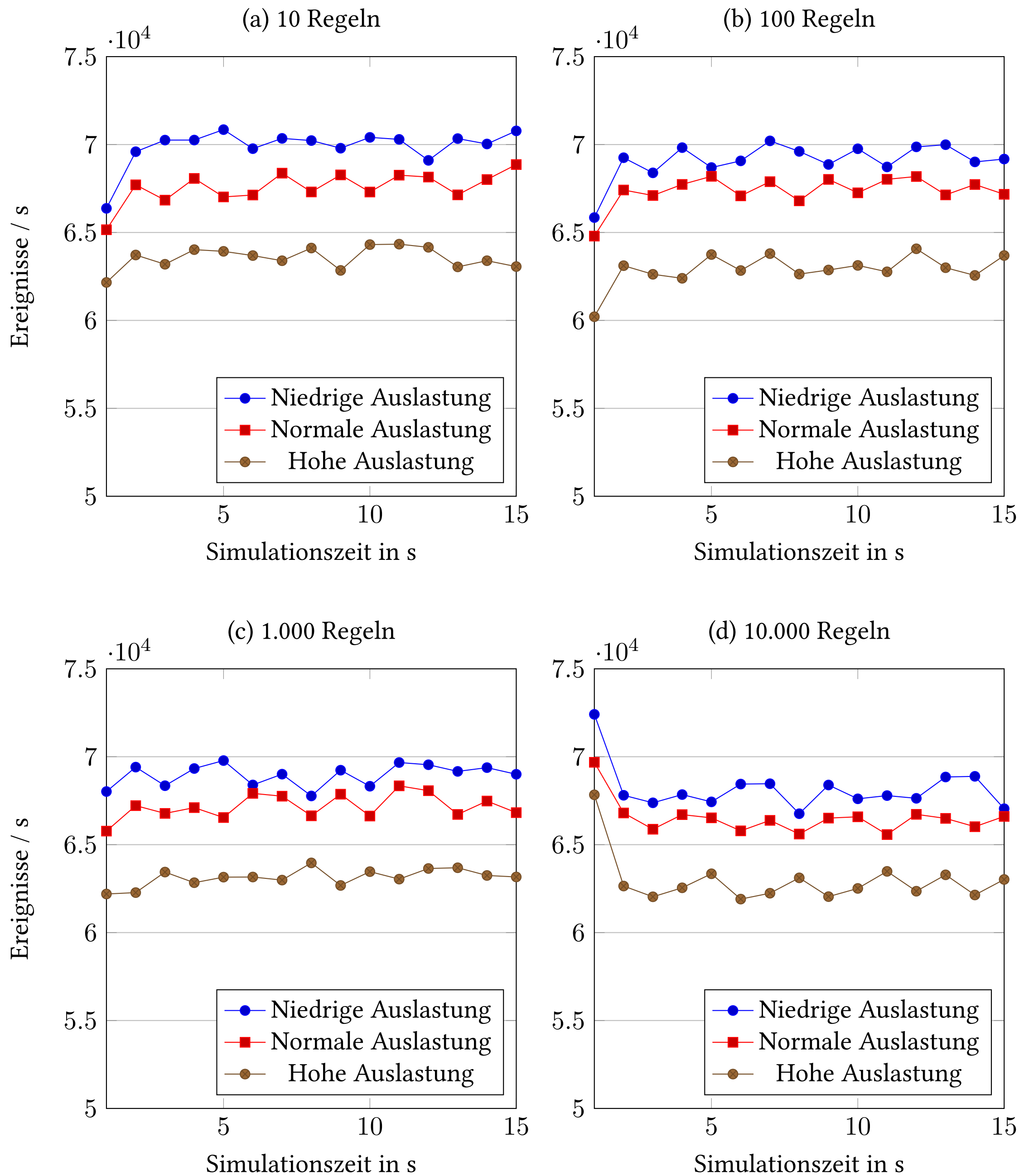
- **Hohe Auslastung**: Bei einer hohen Auslastung, versendet der Context-Service eine Benachrichtigung im Durchschnitt für jeweils 5 Kontextereignisse, die verarbeitet wurden. Dies bedeutet bei einer Menge von 500.000 Ereignissen 100.000 Benachrichtigungen über erkannte Kontextregeln, die versendet werden.

Test I

In diesem Test wird zunächst die Grundperformance des Context-Service bei der Übertragung und Verarbeitung von Ereignisnachrichten überprüft. Dazu werden Ereignisnachrichten von einem Ereignisproduzenten (*Producer*) über eine Message-Queue zu einem Ereigniskonsumenten (*Consumer*) übertragen und von der Complex Event Processing-Engine verarbeitet. Es wird jeweils gemessen, wie viele Ereignisnachrichten pro Sekunde vom Context-Service empfangen und verarbeitet wurden.

Die folgenden vier Diagramme zeigen die Messergebnisse verschiedener Konfigurationen. Bei den Messungen wurde jeweils die Anzahl, der in der Regelbasis des Context-Service registrierten Kontextregeln, um den Faktor zehn erhöht. Für jede Regelanzahl ist dabei pro

Diagramm die Menge der pro Sekunde empfangenen und verarbeiteten Ereignisse für die Auslastungsstufen *Niedrig*, *Normal* und *Hoch* (siehe Abschnitt 7.2.2) dokumentiert. Es wurden *komplexe* Kontextregeln verwendet. Die Zuordnung der entsprechenden Graphen kann der Legende entnommen werden.



Die Ergebnisse zeigen einen annähernd konstant gleichbleibenden Durchsatz von Ereignisnachrichten pro Sekunde während des gesamten Messvorgangs. Der gemessene Durchsatz variiert dabei je nach Auslastungsstufe, wobei der Unterschied im Durchsatz zwischen niedriger und normaler Auslastung, mit steigender Regelanzahl kleiner wird. Es ist außerdem zu sehen, dass der allgemeine Durchsatz von Ereignisnachrichten pro Sekunde mit steigender Regelanzahl leicht sinkt, dies jedoch nicht in einem nennenswerten Maß. Die Anzahl der Kontextregeln, die in der Regelbasis des Context-Service registriert sind, haben demnach keine großen Auswirkungen auf den Durchsatz an Ereignisnachrichten pro Sekunde.

Die folgende Tabelle 7.1 verdeutlicht dies noch einmal. Es zeigt die benötige Zeit für die Verarbeitung von 100.000, 200.000, 300.000, 400.000 und 500.000 Ereignissen bei niedriger Auslastungsstufe und komplexen Kontextregeln.

Regeln / Ereignisse	10	100	1.000	10.000
100.000	1,449 s	1,514 s	1,467 s	1,420 s
200.000	2,951 s	2,967 s	2,936 s	2,904 s
300.000	4,388 s	4,419 s	4,404 s	4,403 s
400.000	5,825 s	5,871 s	5,857 s	5,887 s
500.000	7,262 s	7,324 s	7,325 s	7,371 s

Tabelle 7.1.: Auswirkung der Regelanzahl auf die benötigte Zeit zur Verarbeitung von 100.000, 200.000, 300.000, 400.000 und 500.000 Ereignissen

Die Tabelle 7.1 zeigt, dass die benötigten Verarbeitungszeiten bei jeder Anzahl von registrierten Kontextregeln nahezu identisch sind. Die Verarbeitungsgeschwindigkeit skaliert dabei linear mit steigendem Volumen an auszuwertenden Ereignissen, sowie mit steigender Kontextregelanzahl. Auch bei den weiteren durchgeführten Messungen mit normaler oder hoher Auslastungsstufe, konnten keine nennenswerten Unterschiede bei der Verarbeitungsgeschwindigkeit bezüglich der Kontextregelanzahl festgestellt werden.

Das Diagramm aus Abbildung 7.1 zeigt die Auswirkung der Regelkomplexität auf die Verarbeitungsgeschwindigkeit von Ereignissen, bei jeweils 10.000 registrierten Kontextregeln und niedriger Auslastung.

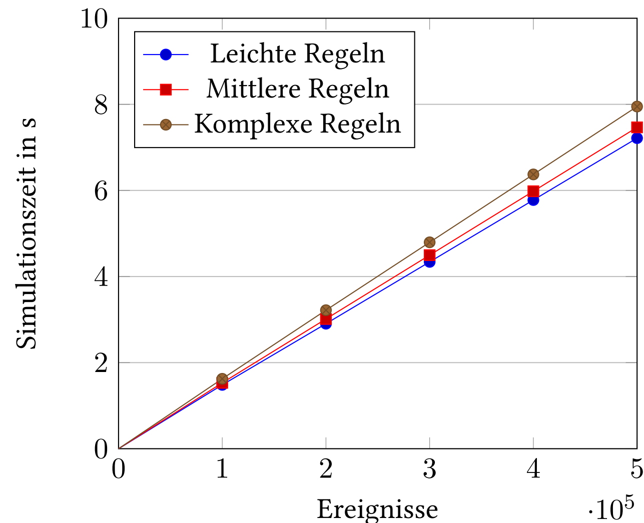


Abbildung 7.1.: Auswirkung der Regelkomplexität auf die Verarbeitungsgeschwindigkeit

Es ist zu erkennen, dass die Verarbeitungsgeschwindigkeit mit steigender Komplexitätsstufe der registrierten Kontextregeln sinkt. Außerdem ist zu sehen, dass die Differenz bei der benötigten Verarbeitungszeit zwischen den drei verschiedenen Regelkomplexitätsstufen, mit steigendem Ereignisvolumen steigt. Bei 500.000 zu verarbeitenden Ereignissen, beträgt die zeitliche Differenz zwischen der Konfiguration mit leichten Kontextregeln und der Konfiguration mit komplexen Kontextregeln 0,734 Sekunden.

Test II

Um zu untersuchen, wie sich das System bei Multithreading verhält, wurde die Übertragung von Ereignisnachrichten von mehreren *Producern* zu mehreren *Consumern* getestet. Dabei läuft jeder Producer und jeder Consumer in einem eigenen Thread und überträgt die Ereignisnachrichten jeweils über eine eigene Message-Queue. Dadurch soll der Prozess des Nachrichtenaustauschs parallelisiert und die Performance weiter gesteigert werden.

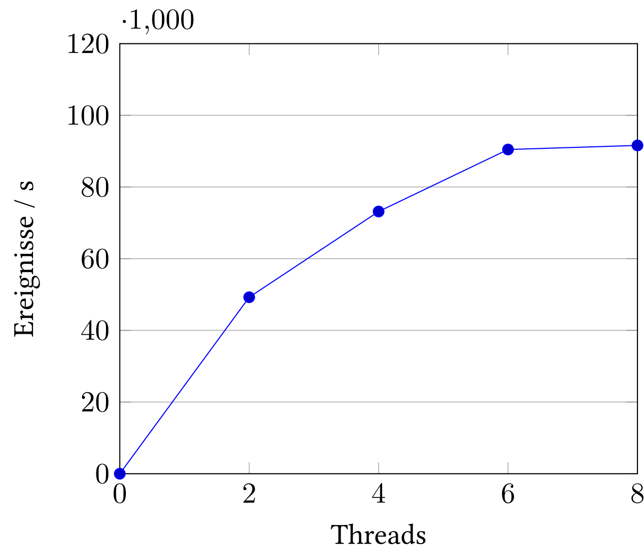


Abbildung 7.2.: Skalierbarkeit mit Multithreading

Das Diagramm aus Abbildung 7.2 zeigt das Skalierungsverhalten des Systems auf dem Testrechner. Die Messpunkte des Diagramms repräsentieren den Durchschnitt, der pro Sekunde vom Context-Service empfangenen und verarbeiteten Ereignisnachrichten. Die Messungen wurden bei 1.000 registrierten komplexen Kontextregeln und normaler Auslastung durchgeführt. Pro durchgeführter Messung, wurde jeweils ein Producer-Thread auf der Seite der Simulation und ein Consumer-Thread auf der Seite des Context-Service hinzugefügt. Die Anzahl der Threads wurde somit pro Messung um zwei erhöht. Der Graph zeigt dabei ein sublineares Skalierungsverhalten. Bei Verdoppelung der Threadanzahl, konnte auf dem verwendeten Testrechner eine durchschnittliche Performancesteigerung um $60,4\%$ erreicht werden. Bei voller Auslastung aller Prozessorkerne (siehe Abbildung 7.3), konnten so durchschnittlich 91.602 Ereignisse pro Sekunde vom Context-Service empfangen und verarbeitet werden. Die Messergebnisse der verschiedenen Konfigurationen sind in Tabelle 8.1 dokumentiert.

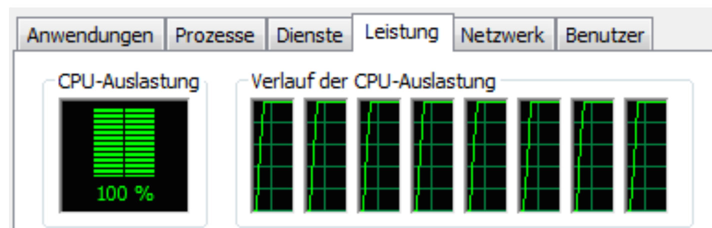


Abbildung 7.3.: Volle Auslastung aller Kerne

7.2.3. Integrierbarkeit

Die Integrierbarkeit des Context-Service Konzepts bzw. die nötigen Schritte, um eine Simulation kontextbewusst zu machen, wurden anhand der Testsimulation evaluiert. Neben der allgemeinen Integrierbarkeit des Context-Service, geht es bei diesem Test vor allem auch speziell darum, dessen nachträgliche Integrierbarkeit in eine Simulation zu überprüfen. Aus diesem Grund, wurde die in dieser Arbeit beschriebene Testsimulation zunächst vollkommen unabhängig vom eigentlichen Vorhaben, diese kontextbewusst zu machen, implementiert. Die Testsimulation wurde mit Hilfe von *Repast Simphony*, wie in den Abschnitten 5.1.2 und 5.1.3 beschrieben, realisiert. Die Simulation konnte dann bereits ausgeführt werden, jedoch ohne die Fähigkeit auf spezielle Kontexte in der Simulation zu reagieren.

Test

Um die Simulation nun mit Hilfe des Context-Service Konzepts kontextbewusst zu machen, waren die folgenden zwei Schritte notwendig:

1. **Senden von Kontextdaten**: Der erste Schritt beschreibt das Senden von Kontextdaten and den Context-Service. Je nachdem welche Kontexte in der Simulation erkannt werden sollen, müssen die dafür notwendigen Kontextdaten als Ereignisnachrichten an den Context-Service übermittelt werden. In diesem Fall sollen Kontexte erkannt werden, für deren Erkennung die Kontextinformationen und Ereignisse der simulierten Geparden und Gazellen benötigt werden. Diese Informationen werden deshalb, wie in Abschnitt 5.3 beschrieben, in Form eines entsprechenden Ereignistypen als Ereignisnachricht an den Context-Service übertragen. Das Generieren einer solchen Ereignisnachricht, sowie dessen anschließende Übertragung ist relativ einfach und ohne großen Aufwand möglich (siehe Listing in 6.2.2).

2. **Reaktion auf Kontexte**: Der zweite Schritt, um die Anwendung letztendlich kontextbewusst zu machen, ist das Festlegen einer Reaktion (*Reactive Computing*) auf erkannte Kontexte. Dazu muss in der Anwendung ein Listener implementiert werden, der auf Benachrichtigungen über erkannte Kontexte vom Context-Service lauscht. Die Benachrichtigungen enthalten die ID der erkannten Kontextregel, mit deren Hilfe eine eindeutige Zuordnung erfolgen kann. Für jede Kontextregel-ID kann dann bei Bedarf die zugehörige Reaktionslogik implementiert werden. Dies soll anhand des folgenden Listings 7.1, welches einen Ausschnitt aus dem Listener der Testsimulation zeigt, veranschaulicht werden:

```java
...
QueueingConsumer.Delivery delivery = null;
try {
  delivery = consumer.nextDelivery();
} catch (ShutdownSignalException | ConsumerCancelledException
  | InterruptedException e) {e.printStackTrace();}

String message = new String(delivery.getBody());
String[] payload = message.split(",");
int ruleId = Integer.parseInt(payload[0]);

switch (ruleId) {
  ...
  case 2:
    System.out.println("More than 5 gazelles have been " +
    "killed in the area X<30, Y<30 within 3 minutes. Added " +
    "a gepard to the simulation.");
    SimulationUpdateHelper.getInstance().setGepardsToAdd(1);
    break;

  case 3:
    System.out.println("There have not been killed at " +
    "least 5 gazelles within the last 30 seconds. Added " +
    "10 gazelles to the simulation.");
    SimulationUpdateHelper.getInstance().setGazellesToAdd(10);
    break;
    ...
}
...
```

Listing 7.1: Empfangen und Verarbeiten einer Context-Service Benachrichtigung

Das Listing zeigt die in der Testsimulation implementierte Reaktionslogik auf die Regeln 3 und 4 aus Abschnitt 7.2.1. Dabei beinhaltet *case 2* die Reaktionslogik auf Regel 3 und *case 3* die Reaktionslogik auf Regel 4, da in der Implementierung die Indexierung der Regeln bei 0 beginnt. Wenn vom Context-Service erkannt wurde, dass in den letzten 30 Sekunden nicht mindestens fünf Gazellen in der Simulation gerissen wurden, sendet er eine Benachrichtigung mit der *ruleId* 3. Die Simulation reagiert auf die Benachrichtigung in diesem Fall mit *case 3* und fügt der Simulation zehn neue Gazellen-Agenten hinzu. Die Simulation ist somit kontextbewusst.

7.2.4. Flexibilität

Der Context-Service sollte flexibel in der Anpassung an sich ändernde Simulations- bzw. Anwendungsszenarien sein. Um dies zu überprüfen, wurde eine Anpassung des Testszenarios der Simulation vorgenommen. Der Ausgangspunkt dieses Tests ist das in Kapitel 4 und 5 beschriebene Simulationsszenario, jedoch noch ohne den Ereignistyp *KillEvent* (siehe Abschnitt 5.3.3) und dessen Implementierung, sowie ohne entsprechende Kontextregeln in der Regelbasis des Context-Service. Geparden und Gazellen werden aber bereits simuliert und es können auch bereits Kontexte, die den Ereignistyp *GepardEvent* oder *GazelleEvent* voraussetzen, vom Context-Service erkannt werden.

Test

Nun soll in der Simulation zusätzlich abgebildet werden, dass die Gazellen von den Geparden gerissen werden können. Dazu wurde das Simulationsszenario entsprechend angepasst und die entsprechende Geschäftslogik implementiert. Wenn ein Gepard eine Gazelle reisst, hatte er einen Jagderfolg. Dieser Jagderfolg stellt ein spezielles Ereignis in der Simulation dar. Die Simulation soll deshalb auf verschiedenste Kontexte, die ein oder mehrere solcher Ereignisse beinhalten, entsprechend reagieren können. Der Context-Service muss also an die Änderung des Simulationsszenarios und die neuen Bedürfnisse angepasst werden. Dazu wurden die folgenden Schritte durchgeführt:

1. **Implementierung des Ereignistyps**: Zunächst muss ein neuer Ereignistyp festgelegt und implementiert werden. In diesem Test wird das Ereignis, wenn ein Gepard eine Gazelle reisst, mit Hilfe des dafür entworfenen Ereignistyps *KillEvent* abgebildet. Er beinhaltet die *ID* des Geparden und der Gazelle sowie die Position, an der der Gepard die Gazelle erlegt hat, als x und y Koordinaten. Die Implementierung des Ereignistyps, erfolgt in Form eines sogenannten *Plain Old Java Object* (POJO), also einem ganz normalen Objekt mit Attributen, Konstruktoren sowie *getter*- und *setter*-Methoden. Das folgende Listing 7.2 zeigt eine Ausschnitt aus der konkreten Implementierung des *KillEvent*-Ereignistypen:

```
1 public class KillEvent {
2
3   private int gepardId;
4   private int gazelleId;
5   private int x;
6   private int y;
7
```

```
8   public KillEvent(int gepardId, int gazelleId, int x, int y) {
9     this.gepardId = gepardId;
10    this.gazelleId = gazelleId;
11    this.x = x;
12    this.y =y;
13  }
14
15  public int getGepardId() {
16    return gepardId;
17  }
18  ...
19 }
```

Listing 7.2: Implementierung des Ereignistypen *KillEvent* als *Plain Old Java Object* (POJO)

2. **Ereignistyp Registrierung**: Als nächstes muss der neue Ereignistyp bei der Complex Event Processing-Engine registriert werden. Dies ist notwendig, damit die Engine den neuen Ereignistyp verarbeiten kann und in der Lage ist Kontexte zu erkennen, die Ereignisse dieses Typs beinhalten. Das folgende Listing 7.3 zeigt die Registrierung des *KillEvent*-Ereignistypen bei der Complex Event Processing-Engine:

```
1 Configuration configuration = new Configuration();
2 ...
3 configuration.addEventType("KillEvent",
4   KillEvent.class.getName());
5 EPServiceProvider engine = EPServiceProviderManager
6   .getDefaultProvider(configuration);
```

Listing 7.3: Registrierung des *KillEvent*-Ereignistypen bei der CEP-Engine

3. **Ereignis Parsen**: Wenn in der Simulation ein *KillEvent* auftritt, wird eine Ereignisnachricht als Byte-Message an den Context-Service gesendet. Diese Nachricht muss dann auf der Seite des Context-Service wieder in ein *KillEvent* überführt werden. Dazu müssen die einzelnen Felder der empfangenen Byte-Message bzw. Ereignisnachricht ausgelesen und ein neues *KillEvent*-Objekt mit den entsprechenden Parametern initialisiert werden. Das erstellte Ereignisobjekt wird dann an die Complex Event Processing-Engine zur Auswertung übergeben. Das folgende Listing 7.4 zeigt den beschriebenen Prozess der Verarbeitung einer *KillEvent*-Ereignisnachricht, in ein für die Complex Event Processing-Engine verständliches Ereignisobjekt:

```java
...
QueueingConsumer.Delivery delivery = null;
try {
  delivery = consumer.nextDelivery();
} catch (ShutdownSignalException | ConsumerCancelledException
  | InterruptedException e) {
    e.printStackTrace();
}
String message = new String(delivery.getBody());
String[] payload = message.split(",");
int eventType = Integer.parseInt(payload[0]);

switch (eventType) {

  case 3:
    int gepardKillId = Integer.parseInt(payload[1]);
    int gazelleKillId = Integer.parseInt(payload[2]);
    int x = Integer.parseInt(payload[3]);
    int y = Integer.parseInt(payload[4]);
    KillEvent killEvent = new KillEvent(gepardKillId,
      gazelleKillId, x, y);
    engine.sendEvent(killEvent);
    break;
  ...
}
```

Listing 7.4: Verarbeitung einer *KillEvent*-Ereignisnachricht

4. **Kontextregeln registrieren**: Der letzte Schritt ist die Registrierung von entsprechenden Kontextregeln, die KillEvents beinhalten und während der Simulation erkannt werden sollen. Beispiele für solche Regeln in der Testsimulation, wären jeweils Regel 3, 4 und 5 aus Abschnitt 7.2.1. Es können beliebige weitere Kontextregeln in der Regelbasis des Context-Service hinterlegt und registriert werden. Die Simulation wird benachrichtigt, sobald ein entsprechender Kontext erkannt wurde und kann dann darauf reagieren.

7.3. Nicht Teil der Tests

Die folgenden Punkte sind nicht Teil der Tests dieser Arbeit. Dennoch sollen die entsprechenden theoretischen Überlegungen dargestellt werden, so dass sie als weitere mögliche Untersuchungen in anderen Arbeiten aufgegriffen werden können.

7.3.1. Partitionierung

Der Context-Service kommuniziert über Nachrichten mit der Anwendung, die kontextbewusst gemacht werden soll. Dies geschieht mit Hilfe einer Message Oriented Middleware. In dieser Arbeit wurde dazu die Middleware *RabbitMQ* verwendet. Die Nachrichten werden von einer produzierenden Quelle (*Producer*) an einen konsumierenden Empfänger (*Consumer*) gesendet. Dabei gibt es die Möglichkeit Nachrichten mittels sogenanntem *Routing* nur an ausgewählte Empfänger zu senden. Dazu wird ein *Routing Key* festgelegt, mit dem dann der entsprechende Empfänger adressiert werden kann. Es ist auch möglich, dass mehrere Empfänger Nachrichten mit dem selben Routing Key empfangen. Die Empfänger können dabei alle zu einer Anwendung gehören aber auch jeweils eine eigene Anwendung für sich darstellen. Abbildung 7.6 zeigt das Routing von Nachrichten an verschiedene Empfänger mit Hilfe von Routing Keys.

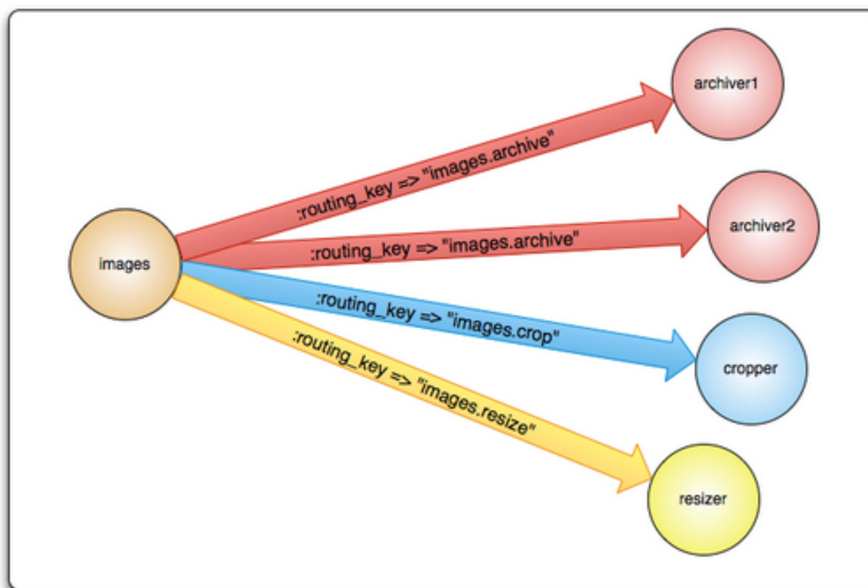


Abbildung 7.4.: Routing von Nachrichten an verschiedene Empfänger über Routing Keys[1]

[1]https://www.rabbitmq.com/tutorials/amqp-concepts.html

Es wäre nun prinzipiell denkbar, die in der Multiagentensimulation auftretenden Ereignisse, nach voneinander unabhängigen Kontextkategorien aufzuteilen. Die kategorisierten Ereignisse, könnten dann mit Hilfe von Routing an getrennte Context-Service Instanzen gesendet und ausgewertet werden. Ein mögliches Szenario wäre zum Beispiel, dass zwei Context-Service Instanzen gestartet werden. Die erste Instanz könnte dann alle Ereignisse von Geparden und Gazellen empfangen und die zweite Instanz alle Ereignisse, die zum Beispiel von Tigern und weiteren Tieren stammen. Die Ereignisse würden mittels Routing an die jeweilige Context-Service Instanz gesendet werden, die für die entsprechenden Kontexte konfiguriert ist. Dies ist in der folgenden Abbildung 7.5 dargestellt.

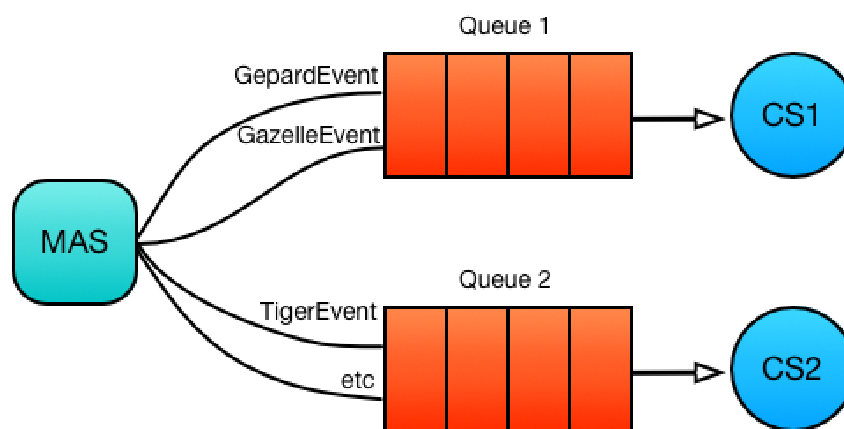


Abbildung 7.5.: Routing von Ereignisnachrichten an verschiedene Context-Services

Es ist denkbar, dass durch die Partitionierung der Ereignisse auf mehrere Context-Services ggf. eine bessere Gesamtperformance erreicht werden könnte, da die Last entsprechend verteilt wird. Dies würde somit auch eine weitere Möglichkeit der Skalierung darstellen. Die Partitionierung der Ereignisse der Multiagentensimulation, ist jedoch nicht Teil der Tests dieser Arbeit.

7.3.2. Synchronisation

Da die Architektur des Context-Service auf dem Reactive Computing Paradigma basiert, findet die Kommunikation zwischen der Multiagentensimulation und dem Context-Service asynchron statt. Daher ist es unter Umständen möglich, dass gegebenenfalls asynchrone Übertragungen von Kontextinformationen im Bezug auf Simulationsiterationen (Ticks) auftreten. Die synchrone Übertragung ist jedoch nicht Teil der Tests in dieser Arbeit, da dadurch das Reactive Computing Paradigma verletzt werden würde.

7.4. Weitere Anwendungsszenarien

Die *SURVEYOR*-Komponente (Context-Service) des MARS-Frameworks (siehe Abbildung 5.2) ist primär dafür konzipiert, Kontexte in Simulationen zur Laufzeit zu erkennen und die Simulation zu Benachrichtigen, sobald ein definierter Kontext erkannt wurde. Dadurch wird die Simulation kontextbewusst und kann auf Situationen reagieren. Es sind jedoch noch weitere Anwendungsszenarien des Context-Service in MARS denkbar.

7.4.1. Kollisionserkennung

MARS-Surveyor könnte zum Beispiel für eine einfache Kollisionserkennung von Agenten und Objekten in einer Simulation verwendet werden. Voraussetzung dafür ist, dass Ereignistypen zur Verfügung gestellt werden, die entsprechende auswertbare Daten beinhalten. Angewandt auf das in Kapitel 5 beschriebene Simulationsmodell mit Geparden- und Gazellen-Agenten, könnten die Ereignistypen *GepardEvent* und *GazelleEvent* genutzt werden. Diese beinhalten die Koordinaten der jeweiligen Agenten als x und y Werte (siehe Abschnitt 5.3). Für die Kollisionserkennung könnte dann zum Beispiel eine Kontextregel in folgender Form registriert werden:

```
1 // Register collision detection context rule
2 csc.RegisterNewContextRule("select e1, e2 from pattern
3 [every (e1=GazelleEvent -> e2=GazelleEvent(e1.x = e2.x,
4 e1.y = e2.y))]", Listener.CollisionDetectionMethod);
```

Listing 7.5: Registrieren einer Kontextregel zur Kollisionserkennung

Die Kontextregel aus Listing 7.5 feuert, wenn das *GazelleEvent* $e1$ dieselben x- und y-Koordinaten aufweist, wie das *GazelleEvent* $e2$. Falls dieser Kontext während der Simulation erkannt wird, würde die Delegate-Methode aus Listing 7.6 aufgerufen werden. In dieser Methode kann dann der Code implementiert werden, der als Reaktion auf die erkannte Kollision ausgeführt werden soll. Die beiden Ereignisse, die zum Aufruf der Methode geführt haben, sind im *results*-Parameter enthalten. Darüber kann dann herausgefunden werden, welche Agenten miteinander kollidiert sind.

```
1 public static void CollisionDetectionMethod(
2   Dictionary<string, object> results) {
3   // Do something to avoid or handle the detected collision
4 }
```

Listing 7.6: Delegate-Methode die aufgerufen wird wenn eine Kollision vom Context-Service erkannt wurde

7.4.2. Kontext-Dashboard

Eine weitere Anwendungsmöglichkeit von MARS-Surveyor, wäre die Analyse von Simulationen mit Hilfe eines Kontext-Dashboards. Dazu könnten die Benachrichtigungen des Context-Service über erkannte Kontexte in einer Datenbank gespeichert werden. Diese Daten könnten dann mit Hilfe von Tools wie *Elasticsearch*[2] und *Kibana*[3] entsprechend abgefragt und als Dashboard visualisiert werden.

Abbildung 7.6.: Beispiel eines Dashboards zur Visualisierung von Daten mit Kibana[4]

Ein solches Kontext-Dashboard könnte hilfreich sein, um Simulationsabläufe besser analysieren zu können. Auf dem Dashboard könnte zum Beispiel dargestellt werden, welcher Kontext wann und wie häufig in einer Simulation aufgetreten ist. Darüber hinaus könnten möglicherweise Zusammenhänge zwischen Kontexten, die besonders häufig gemeinsam auftreten erkannt werden. Das Dashboard könnte außerdem bei der Analyse der Auswirkung von Simulationsparametern, auf das Auftreten von speziellen Kontexten bzw. auf den Simulationsverlauf helfen. Mit den Kontextdaten, die in der Datenbank gespeichert werden, können Simulationen zu einem beliebigen Zeitpunkt nachträglich analysiert werden. Zudem könnten die Daten unterschiedlicher Simulationsdurchläufe miteinander verglichen werden.

[2]http://www.elasticsearch.org/overview/elasticsearch/
[3]http://www.elasticsearch.org/overview/kibana/
[4]http://a-bouchama.blogspot.de/2014/01/flow-activity-monitoring-logging.html

8. Ergebnisse und Diskussion

Das folgende Kapitel fasst die Ergebnisse aus dem vorangegangenen Kapitel 7 zusammen. Es werden die allgemeinen Erkenntnisse, die während der Erarbeitung des vorgestellten Konzepts für Kontexterkennung in Multiagentensimulationen erlangt wurden, erläutert und diskutiert. Motivation und Ausgangspunkt war, dass es unter den bisher bekannten Ansätzen für Kontexterkennung [Kumar und Xie (2012); Baldauf u. a. (2007)], keine adäquate Lösung für die Kontexterkennung in komplexen Multiagentensimulationen gibt. Bei der Diskussion werden daher die Ergebnisse und Erkenntnisse auch bezüglich der verwandten Arbeiten aus Kapitel 3 betrachtet.

Für die Überprüfung der zu Beginn dieser Arbeit aufgestellten Hypothesen, wurden mehrere Experimente ausgearbeitet. Die verschiedenen Experimente wurden mit der in Kapitel 5 und 6 beschriebenen Testumgebung durchgeführt. Die folgenden Abschnitte beleuchten nun jeweils die Ergebnisse der durchgeführten Experimente aus Kapitel 7, hinsichtlich der Hypothesen aus Abschnitt 1.1.1. Es soll außerdem gezeigt werden, ob die generellen Nachteile der verwandten Arbeiten (siehe Kapitel 3), mit dem eigenen Ansatz verbessert bzw. gelöst werden konnten. Dazu zählt unter anderem zum Beispiel die Möglichkeit der Formulierung temporaler Constraints in Kontextregeln [Ranganathan und Campbell (2003)], sowie die Erkennung der entsprechenden Kontexte.

Die paradigmatische Grundlage bei der Entwicklung des in dieser Arbeit vorgestellten Context-Service Konzepts und der Realisierung der Testumgebung zu dessen Evaluierung, bildete das *Reactive Computing*. Durch den Paradigmenwechsel vom *Responsive Computing*-Paradigma, welches von den bisherigen Ansätzen verfolgt wurde, zum Paradigma des *Reactive Computing*, konnten verschiedene Vorteile erreicht werden. Dabei spielten unter anderem Teilaspekte des Reactive Computing, wie zum Beispiel eine ereignisgetriebene Architektur, eine wichtige Rolle. Dies wird ebenfalls in den folgenden Abschnitten genauer dargelegt.

8.1. Vorteile des Reactive Computing Paradigma

In Abschnitt 2.7.2 des Grundlagen-Kapitels, wurden wesentliche Merkmale des Reactive Computing Paradigmas genannt. Diese Merkmale und die dadurch entstehenden positiven Effekte, die im weiteren Verlauf erläutert werden, haben sich auch bei dem in dieser Arbeit vorgestellten Context-Service Konzept gezeigt.

Das Grundelement für Reactive Computing, ist die Verwendung einer ereignisgetriebenen Architektur (*Event-Driven* Merkmal). Durch die ausschließliche Verwendung von Ereignisnachrichten zur Kommunikation zwischen der Multiagentensimulation und dem Context-Service, wurde eine sehr lose Kopplung erreicht. Dies wirkte sich wiederum positiv auf die Integrierbarkeit und somit auch auf die Modularität des Gesamtsystems aus. Der Context-Service lässt sich relativ einfach in eine Anwendung integrieren und verwenden, da es im Grunde nur zwei wesentliche Schnittstellen mit der Anwendung gibt, nämlich das Senden von Ereignisnachrichten und das Empfangen von Benachrichtigungen. Dies wurde im Experiment zur Integrierbarkeit (siehe Abschnitt 7.2.3) gezeigt.

Vergleicht man das in dieser Arbeit vorgestellte Context-Service Konzept hinsichtlich der Modularität und Integrierbarkeit mit dem *Java Context Awareness Framework* (JCAF) [Bardram (2005b)] (siehe Abschnitt 3.3), werden vor allem die folgenden Unterschiede deutlich:

- **Grad der Kopplung**: Der Grad der Kopplung ist deutlich geringer. Die hinter der Context-Service Komponente stehende technische Umsetzung, bleibt aufgrund der Kommunikation über Nachrichten für die anderen Systemkomponenten transparent. Die Interaktion zwischen Simulation und Context-Service, beschränkt sich auf das Senden von Kontextdaten in Form von Ereignisnachrichten und dem Empfangen von Benachrichtigungen. Dadurch bleibt das System leichter wartbar und erweiterbar.
- **Hochsprachenunabhängigkeit**: Die Simulation kann nun in einer beliebigen Hochsprache implementiert werden, wogegen JCAF nur in Java verwendet werden kann. Aufgrund der sehr losen Kopplung und des modularen Designs des Context-Service, ist eine nahezu unabhängige Implementierung der Anwendung möglich. Das Kontextmodell der vorgestellten Lösung verzichtet außerdem auf Interface- und Strukturvorgaben, wie bei JCAF oder Gaia (siehe Abschnitt 3.1.1 und 3.2.1). Dadurch wird eine leichtere und unabhängige Modellierung unterschiedlichster Kontextinformationen ermöglicht.

Der Vergleich zeigt, dass durch die Verwendung des Reactive Computing Paradigma, Verbesserungen in den Bereichen Modularität und Integrierbarkeit erreicht werden können.

Ein weiterer Vorteil durch die Verwendung des Reactive Computing Paradigmas, entsteht bei der Skalierbarkeit des Systems (*Scalable* Merkmal). Die reine asynchrone Kommunikation über Nachrichten, erlaubt eine Skalierung des Systems durch das Hinzufügen von Ereignisproduzenten (*Event Producer*) und -konsumenten (*Event Consumer*). Dadurch kann die Nebenläufigkeit der Ereignisverarbeitung erhöht und somit die Performance des Systems bei Bedarf entsprechend gesteigert werden (siehe Test II in Abschnitt 7.2.2). Eine vergleichbare Möglichkeit der Skalierung ist bei den Ansätzen aus Kapitel 3 nicht gegeben.

Tabelle 8.1 zeigt eine Übersicht der Messergebnisse aus den Performancetests in Abschnitt 7.2.2. Die Messergebnisse stellen die durchschnittliche Verarbeitungsgeschwindigkeit je Testkonfiguration dar. Bei den Messungen wurden in Spitzen auch Messwerte von über hunderttausend pro Sekunde verarbeiteten Ereignissen erreicht. Ein Grund für die Performance des Context-Service, ist unter anderem die nicht-blockierende Kommunikationsweise, die Teil des Reactive Computing Paradigmas ist. Der asynchrone Austausch von Ereignisnachrichten, erlaubt eine kontinuierliche Verarbeitung von Kontextinformationen mit niedrigerer Latenz und höherem Durchsatz, als bei synchroner blockierender Kommunikation und dadurch auch eine bessere Nutzung der verfügbaren Hardwareressourcen [EPTS (2013)]. JCAF verwendet dagegen zum Beispiel eine blockierende synchrone Kommunikation mittels Java RMI, die mit entsprechenden Nachteilen verknüpft ist (siehe Abschnitt 2.7.1).

Auslastung / **Regelkompl.**	**Niedrig**	**Normal**	**Hoch**
Leicht	98.866 Msgs / s	96.440 Msgs / s	81.322 Msgs / s
Mittel	95.078 Msgs / s	92.536 Msgs / s	78.947 Msgs / s
Komplex	94.564 Msgs / s	91.602 Msgs / s	76.844 Msgs / s

Tabelle 8.1.: Durchsatz von Ereignisnachrichten pro Sekunde bei jeweils 1000 registrierten Kontextregeln und 8 Threads je Regelkomplexitäts- und Auslastungsstufe

Die Ergebnisse aus den Experimenten in Abschnitt 7.2.3 und 7.2.2 befürworten letztendlich, dass durch die Verwendung des *Reactive Computing* Paradigmas, Verbesserungen in den Bereichen Modularität, Integrierbarkeit, Skalierbarkeit und Performance erreicht werden können. Die Erkenntnisse unterstützen somit Hypothese 1 aus Abschnitt 1.1.1.

8.2. Performance

Die Performancetests aus Abschnitt 7.2.2 haben gezeigt, dass der Context-Service durch die Kombination einer ereignisgesteuerten Architektur mit der Technik des Complex Event Processing in der Lage ist, große Volumen von Ereignissen mit hoher Geschwindigkeit zu analysieren und Kontextmuster in Multiagentensimulationen zu erkennen. An dieser Stelle müssen die Ergebnisse in Hinsicht auf Hypothese 2 jedoch aus verschiedenen Blickwinkeln betrachtet und bewertet werden.

Betrachtet man die Echtzeitfähigkeit des Context-Service, wobei hier unter Echtzeitfähigkeit das *Responsive* Merkmal des Reactive Computing Paradigmas verstanden wird, so ist diese durchaus gegeben. Für die Verarbeitung einzelner Ereignisse werden nur wenige Mikrosekunden benötigt, was eine hohe Geschwindigkeit und schnelle Antwortzeit darstellt. Das bedeutet, dass die Simulation grundsätzlich umgehend über erkannte Ereignismuster benachrichtigt werden kann, sobald die entsprechenden Ereignisse vom Context-Service empfangen und verarbeitet wurden.

Auf der anderen Seite zeigen die gemessenen Ergebnisse, dass die Verarbeitung von 500.000 Ereignissen mehrere Sekunden in Anspruch nimmt. Dies muss nun jeweils in Relation zu einer Ticklänge der Multiagentensimulation gesetzt werden, in der der Context-Service zur Kontexterkennung verwendet wird. Falls nun die Zeit, die für eine Simulationsiteration benötigt wird kürzer ist als die Zeit, die für die Verarbeitung aller in diesem Simulationstick generierten Ereignisse benötigt wird, dann wird die Simulation durch die Verwendung des Context-Service verlangsamt. Wenn man nun gemessene Werte von zum Beispiel rund 91.602 verarbeiteten Ereignissen pro Sekunde betrachtet (siehe Tabelle 8.1) heißt das, dass dies bei der in den Tests verwendeten Hardware die Obergrenze der Ereignisanzahl darstellt, die in einer entsprechenden Multiagentensimulation pro Sekunde generiert werden dürfte.

Als Schlussfolgerung der Ergebnisse kann für Hypothese 2 ausgesagt werden, dass der Context-Service die performante Auswertung von Kontextinformationen und Erkennung von Kontextmustern in Multiagentensimulationen bewerkstelligen kann. Die Zeitdauer die für die Auswertung benötigt wird, hängt jedoch von der pro Simulationstick erzeugten Ereignismenge ab und kann ggf. mehrere Sekunden betragen. Die Echtzeitfähigkeit kann aus diesem Grund nicht grundsätzlich gewährleistet werden, wodurch Hypothese 2 widerlegt wird.

8.3. Skalierbarkeit

Um eine Aussage zu Hypothese 3 treffen zu können, wurden in Abschnitt 7.2.2 mehrere Experimente zum Skalierungsverhalten des Context-Service durchgeführt und die jeweiligen Messergebnisse entsprechend dokumentiert. Dabei wurden die Auswirkungen von verschiedenen Parametern, wie zum Beispiel die Anzahl der registrierten Kontextregeln oder die Anzahl der laufenden Threads, auf die Verarbeitungsgeschwindigkeit des Context-Service untersucht.

In Test I wurden mehrere Messungen durchgeführt, bei denen die Anzahl der beim Context-Service registrierten Kontextregeln jeweils um den Faktor 10 erhöht wurde (10, 100, 1.000 und 10.000). Zudem wurde bei den Messungen auch die Komplexität der registrierten Kontextregeln variiert (*leicht*, *mittel* und *komplex*). Die Ergebnisse dieser Messungen haben gezeigt, dass die Anzahl der registrierten Kontextregeln, keine nennenswerten Auswirkungen auf die Verarbeitungsgeschwindigkeit des Context-Service hat. Die Messergebnisse in Tabelle 7.1 zeigen die Zeiten, die benötigt wurden, um die jeweiligen Ereignismengen zu verarbeiten. Es ist zu sehen, dass die gemessenen Zeiten bei jeder getesteten Ereignismenge, im Bezug auf die steigende Anzahl an Kontextregeln, nahezu identisch bleiben. Das bedeutet, dass hinsichtlich der Kontextregelanzahl ein super-lineares Skalierungsverhalten vorliegt. Die für die Verarbeitung der jeweiligen Ereignismengen benötigte Zeit, steigt nicht im selben Verhältnis wie die Anzahl der registrierten Kontextregeln.

In Test II wurden Messungen hinsichtlich der Skalierbarkeit des Context-Service durch Multithreading durchgeführt. Dadurch sollte untersucht werden, wie sich das Hinzufügen weiterer *EventProducer*- und *EventConsumer*-Threads auf die Performance des Context-Service auswirkt. Die Ergebnisse der Messungen sind in Abbildung 7.2 dargestellt. Der Graph zeigt ein sub-lineares Skalierungsverhalten.

Als Schlussfolgerung für Hypothese 3 aus Abschnitt 1.1.1 können daher zwei Aussagen gemacht werden:

1. Bezüglich des Skalierungsverhaltens der Verarbeitungsgeschwindigkeit des Context-Service, ist Hypothese 3 widerlegt worden. Das System skaliert zwar durch das Hinzufügen weiterer *EventProducer*- und *EventConsumer*-Threads, jedoch zeigt sich dabei ein sub-lineares Skalierungsverhalten.

2. Betrachtet man das Skalierungsverhalten des Context-Service hinsichtlich der Anzahl der registrierten Kontextregeln, so kann Hypothese 3 bestätigt werden.

8.4. Kontexterkennung

Für die Überprüfung von Hypothese 4, wurde das in Abschnitt 7.2.1 beschriebene Experiment durchgeführt. In dem Experiment wurden mehrere Kontextregeln definiert, die sich in ihrer Komplexität und den enthaltenen Bedingungen unterscheiden. Dabei wurden unter anderem Kontextregeln mit temporalen (zeitlichen), spatialen (örtlichen) sowie spatiotemporalen (zeitlichen und örtlichen) Bedingungen definiert, die während der Testsimulation vom Context-Service erkannt werden sollten.

Das durchgeführte Experiment in Abschnitt 7.2.1 hat letztendlich gezeigt, dass der entwickelte Context-Service eine zuverlässige Erkennung von definierten Kontexten und Teilkontexten in dem jeweils simulierten Szenario gewährleisten kann. Alle definierten Kontextregeln wurden vom Context-Service während der Testsimulation erkannt. Die in Abschnitt 1.1.1 formulierte Hypothese 4 kann somit also bestätigt werden.

Das Experiment hat außerdem gezeigt, dass das in dieser Arbeit vorgestellte Context-Service Konzept, eine Verbesserung gegenüber den in Kapitel 3 untersuchten Ansätzen *JCAF* und *Gaia* darstellt. Beiden Ansätzen fehlt unter anderem die Möglichkeit zur Erkennung von Kontextmustern mit temporalen Bedingungen, was in Ranganathan und Campbell (2003) explizit als offener Punkt genannt wird:

> "We are considering making the model more powerful by using first order temporal logic. This brand of logic allows writing rules that encode temporal constraints. For example, it is possible to write conditions that express the fact that some context event occurred before some other context event. Temporal logic would also allow us to impose temporal constraints between different context events. For example, it would allow us to express the fact that someone must have to enter a room before they can leave it."

Dieser offene Punkt aus Gaia, konnte in dem hier entwickelten Context-Service Konzept gelöst werden. Wie unter anderem im Kontextregel-Experiment gezeigt wurde, können nun spatiale, temporale sowie spatiotemporale Bedingungen bei der Deklaration von zu erkennenden Kontextmustern angegeben werden. Es können zum Beispiel temporale Korrelationen, wie sie in Abbildung 2.8 gezeigt sind, zwischen Ereignissen erkannt werden. Darüber hinaus können nun außerdem auch Zeitfenster - sogenannte *Sliding Windows* (siehe Abbildung 5.7) - in den Kontextregeln angegeben werden. Diese Sliding Windows erlauben es zeitliche Ausschnitte festzulegen, die auf Kontextmuster untersucht werden sollen.

8.5. Flexibilität

Zur Überprüfung von Hypothese 5, wurde das entsprechende Experiment aus Abschnitt 7.2.4 durchgeführt. In dem Experiment wurde ein bestehendes Simulationsmodell erweitert. Dabei wurde untersucht, wie flexibel sich der Context-Service an die neuen Anforderungen des erweiterten Simulationsmodells anpassen lässt und wieviel Aufwand nötig ist, um die jeweiligen Anpassungen umzusetzen.

Das Experiment zur Flexibilität hat gezeigt, dass die Context-Service Komponente eine einfache Anpassung an das jeweils gewünschte Szenario, welches simuliert werden soll ermöglicht. Zur Anpassung an das erweiterte Simulationsmodell waren lediglich vier Schritte nötig (siehe Experiment in Abschnitt 7.2.4), die zudem mit relativ wenig Implementierungsaufwand umgesetzt werden konnten. Hier ist vor allem die flexible Anpassung von bestehenden, sowie die Definition von neuen Kontextregeln zur Erkennung von wichtigen Ereignissen und speziellen Situationen in der Simulation, eine nennenswerte Verbesserung gegenüber anderen Ansätzen.

Bei Verwendung von JCAF, müsste die Implementierung der Anpassungen dagegen zwangsläufig sehr stark auf das Framework und seine Interfaces abgestimmt werden. Dies ist erforderlich, damit JCAF die Kontextinformationen verwalten und Kontextänderungen anzeigen kann. Da bei JCAF zudem die Möglichkeit fehlt, Kontexte die während einer Simulation erkannt werden sollen in Form von Kontextregeln zu formulieren, ist die Flexibilität relativ gering. Für jedes Ereignismuster das erkannt werden soll, müsste ein entsprechender Algorithmus implementiert werden, der diese Aufgabe übernimmt. Die Implementierung des jeweiligen Algorithmus könnte dann ggf. sehr kompliziert werden, vor allem für Kontextmuster mit temporalen Bedingungen.

Bei Gaia verhält es sich mit der Flexibilität, bezüglich Anpassungen des Simulationsszenarios oder anderweitigen Änderungen an der Implementierung der Anwendung, vermutlich ähnlich wie bei JCAF. Jedoch kann hier keine genaue Aussage gemacht werden, da Gaia im Gegenteil zu JCAF nicht öffentlich für Tests zur Verfügung gestellt wurde.

Der Context-Service ist somit gegenüber konventionellen Ansätzen flexibler in der Anpassung an sich ändernde Simulations- und Anwendungsszenarien. Er lässt sich mit wenig Implementierungsaufwand für neue Simulationsszenarien konfigurieren. Hinsichtlich dieses Ergebnisses, kann Hypothese 5 aus Abschnitt 1.1.1 bestätigt werden.

8.6. Eignung für Multiagentensimulationen

Die Ergebnisse der Experimente haben gezeigt, dass die in Abschnitt 4.2 genannten Anforderungen an den Context-Service erfüllt werden konnten und die in Abschnitt 4.6 analysierten Aspekte von Kontext, bei einer Verwendung des Context-Service abgedeckt werden können. Die jeweiligen Punkte werden im Folgenden noch einmal kurz betrachtet:

- **Kontextinformationen**: Der Context-Service unterstützt eine einfache Abbildung von Kontextinformationen in Form von Ereignistypen (siehe Abschnitt 5.3) und verzichtet auf Strukturvorgaben, wie bei JCAF oder Gaia. Er kann Kontextinformationen verschiedenster fachlicher Domänen verarbeiten.

- **Reasoning**: Der Context-Service unterstützt das Reasoning von Kontextinformationen [Bettini u. a. (2010)] und kann *High Level* Kontext aus *Low Level* Kontextinformationen ableiten (siehe Abschnitt 2.3).

- **Abfrage von Kontext**: Der Context-Service bietet eine einfache Möglichkeit, die für das jeweilige Simulationsszenario relevanten Kontextmuster, mittels einer SQL-basierten Context Query Language zu definieren.

- **Temporallogik**: Zeitliche Abhängigkeiten und Zusammenhänge zwischen Kontexten können vom Context-Service erkannt werden. Die Analyse von Kontextinformationen über einen längeren Zeitabschnitt Δt, statt einem singulären Zeitpunkt t wie bei JCAF und Gaia ist nun möglich. Spatiotemporale Kontextmuster können ebenso formuliert und erkannt werden.

- **Automatische Benachrichtigungen**: Der Context-Service kann die Simulation automatisch über das Erkennen bzw. Eintreten eines vordefinierten Kontextmusters informieren. Die Anwendung wird dadurch kontextbewusst (context-aware) und kann auf erkannte Kontexte reagieren (*Reactive Computing*).

- **Kommunikation über Nachrichten**: Die Kommunikation mit dem Context-Service, erfolgt über Ereignisnachrichten bzw. Nachrichtenströme (Message Queues). Die Ereignisorientierung ist außerdem ein wichtiges Mittel zur Umsetzung des Reactive Computing Paradigma und bringt weitere Vorteile mit sich, die damit zusammenhängen:
  - **Hochsprachenunabhängigkeit**: Durch die Kommunikation über Ereignisnachrichten kann der Context-Service auf Vorgaben zur Verwendung hochsprachenabhängiger Interfaces verzichten. Dadurch wird eine hochsprachenunabhängige Verwendung des Context-Service ermöglicht.

- **Lose Kopplung der Komponente**: Die hinter der Context-Service Komponente stehende technische Umsetzung bleibt aufgrund der Kommunikation über Ereignisnachrichten für die anderen Systemkomponenten transparent. Dadurch bleibt das System leichter wartbar und erweiterbar.
- **Leichte Integration**: Die Context-Service Komponente ist leicht zu verwenden. Eine Anwendung kann mit verhältnismäßig wenig Aufwand kontextbewusst gemacht werden. Dies gilt auch für den Use Case, dass eine bereits bestehende Anwendung nachträglich kontextbewusst gemacht wird.
- **Skalierbarkeit**: Die Context-Service Komponente unterstützt eine Form der Skalierbarkeit, um bei entsprechender Ereignisanzahl trotzdem eine akzeptable Performance bieten zu können.
- **Flexibilität**: Die Context-Service Komponente erlaubt eine einfache Anpassung an das jeweils gewünschte Szenario, welches simuliert werden soll. Die flexible Anpassung von bestehenden, sowie die neue Definition von Regeln zur Erkennung von wichtigen Ereignissen und speziellen Situationen in der Simulation, ist mit relativ wenig Aufwand möglich.
- **Performance**: Durch die Verwendung von Complex Event Processing, kann umgehend auf aktuelle Ereignisse reagiert und der weitere Simulationsverlauf entsprechend gesteuert werden. Durch die Ereignisorientierung und die Technologie des Complex Event Processing, ist eine Betrachtung des aktuellen Simulationszustands, anstatt einer retrospektiven Betrachtung möglich. Der Context-Service kann dadurch eine im Sinne des Reactive Computing Paradigmas nahezu echtzeitfähige Verarbeitung von Kontextdaten bewerkstelligen.

Das in dieser Arbeit vorgestellte Context-Service Konzept eignet sich somit also für einen praktischen Einsatz zur Kontexterkennung in Multiagentensimulationen.

9. Fazit und Ausblick

Die Ergebnisse dieser Arbeit haben gezeigt, dass durch die innovative Verknüpfung moderner Architekturparadigmen, wie dem Reactive Computing Paradigma und Technologien wie Complex Event Processing, eine alternative Lösung zur Kontexterkennung in Multiagentensimulationen entwickelt werden konnte. Mit dem Context-Service Konzept, konnten mehrere Vorteile gegenüber anderen Ansätzen auf dem Gebiet der *Context-Awareness*, wie zum Beispiel Gaia [Ranganathan und Campbell (2003)] oder JCAF [Bardram (2005b)] erreicht und offene Punkte gelöst werden.

Die Agenten in einer Multiagentensimulation sind per Definition von Jennings und Wooldridge (2000) kontextsensitiv, um autonom in ihrer Umwelt agieren zu können. Diese Kontextsensitivität ist jedoch auf das Bewusstsein über ihren eigenen Zustand und ihre unmittelbare Umgebung zu einem bestimmten singulären Zeitpunkt beschränkt. Mit Hilfe des Context-Service kann diese Beschränkung überwunden werden. Er ermöglicht eine globale Erfassung von Kontexten in der Simulation über definierbare Zeitabschnitte. Dadurch können auch Ereignisse bzw. Teilkontexte, die räumlich (spatial) und auch zeitlich (temporal) weit auseinander liegen, miteinander in Korrelation gebracht werden.

Die Simulation kann also nun zusätzlich auch globale Kontexte wahrnehmen und falls nötig den Simulationsablauf entsprechend anpassen. Die Kontextmuster, die während der Simulation automatisch erkannt werden sollen, können mit Hilfe von deklarativen Regeln im Context-Service definiert werden. Um die für den Simulationsverlauf relevanten Situationen erkennen und entsprechend darauf reagieren zu können, werden die dafür benötigten Kontextdaten kontinuierlich als Ereignisse an den Context-Service gesendet. Dort kann dann mit Hilfe von Complex Event Processing überprüft werden, ob entsprechend definierte Kontextmuster vorliegen. Die Anwendung kann nun über erkannte Kontexte informiert werden und eine entsprechende Reaktion in der Simulation einleiten (*Reactive Computing)*, um sich den aktuellen Gegebenheiten anzupassen. Die jeweilige Anwendung ist somit kontextbewusst (*context-aware*).

9.1. Ausblick

Das in dieser Arbeit vorgestellte Context-Service Konzept, bildet eine gute Grundlage für weitere Untersuchungen bezüglich Kontexterkennung in Multiagentensimulationen, als auch in Anwendungen aus anderen fachlichen Domänen.

Ein Punkt wäre zum Beispiel, Möglichkeiten zu finden und zu evaluieren, mit denen die Performance des Context-Service weiter gesteigert werden kann. Dabei könnte unter anderem die in Abschnitt 7.3.1 angesprochene Möglichkeit, alle Kontextinformationen einer Anwendung in mehrere voneinander unabhängige Kontextkategorien zu Partitionieren und von physikalisch verteilten Context-Service Instanzen auswerten zu lassen, näher untersucht werden. Die asynchrone Kommunikation über Nachrichten, sowie die Möglichkeit Ereignisse mittels Routing über Message-Queues an ausgewählte Empfänger zu senden, bietet dazu eine optimale Basis.

Ein weiterer interessanter Punkt wäre die Untersuchung, inwieweit sich das in dieser Arbeit vorgestellte Context-Service Konzept nutzen lassen könnte, um Systeme nach dem *Proactive Computing* Paradigma [Tennenhouse (2000)] zu realisieren. Ein proaktives System zeichnet sich durch initiatives Handeln aus, welches auf der Antizipation von Ereignissen basiert. Etwas zu antizipieren bedeutet, dass das Eintreten eines Ereignisses, einer Situation oder eines Zustands im voraus angenommen wird. Darauf aufbauend können Systeme dann vorausschauend Aktionen anstoßen. Kontextbewusstsein (*Context-Awareness*) ist zudem auch für sogenannte *Self-Adaptive* Systeme und Anwendungen [Zampunieris und Dobrican (2014)] ein wichtiger Baustein. Auch hier könnten möglicherweise weitere Untersuchungen mit Hilfe des vorgestellten Context-Service Konzepts durchgeführt werden.

A. Inhalt der DVD

Der Arbeit liegt ein Datenträger mit den folgenden Inhalten bei:

- **Messergebnisse** Die Messergebnisse der jeweiligen durchgeführten Experimente. Die Ordner und Dateien der Messungen sind entsprechend benannt, damit sie den dokumentierten Messergebnissen zugeordnet werden können.

- **Entwicklertools** Die Entwicklertools, die für die Entwicklung der Testumgebung und für die Durchführung der dokumentierten Experimente verwendet wurden. Dazu gehört die Entwicklungsumgebung *Eclipse* sowie das Multiagentensimualtionstool *Repast Simphony 2.1*.

- **Librarys** Die Librarys der verwendeten Message Oriented Middleware *RabbitMQ* und der Complex Event Processing-Engine *Esper*.

- **Quellcode** Der Quellcode der Testumgebung mit der Multiagentensimulation und dem Context-Service (MARS-Surveyor), sowie die Implementierung der Performancetests

- **Masterarbeit** Eine digitale Version der Masterarbeit als .pdf Datei

B. Danksagung

Ich möchte mich an dieser Stelle bei allen, die mich während meines Studiums begleitet und unterstützt haben, für diese wunderbare Zeit bedanken! Ein besonderer Dank gilt:

- meinen Kommilitonen Christian Vogt, Malte Nogalski und Max Jonas Werner
- dem gesamten MARS Team
- meinem Betreuer Prof. Dr. Thomas Thiel-Clemen
- meinem Zweitprüfer Prof. Dr. Stefan Sarstedt
- meinem Bruder Robert und meinen Eltern
- meinen Freunden

Literaturverzeichnis

[ComplexEvents 2013] Complex Event Processing & Real Time Intelligence. (2013). – URL http://complexevents.com/?page_id=3

[DroolsFusion 2013] Drools Fusion Temporal Reasoning - JBoss Community. (2013). – URL http://www.jboss.org/drools/drools-fusion.html

[GICEP 2013] GI Informatiklexikon - Complex Event Processing (CEP) - Gesellschaft für Informatik e.V. (2013). – URL http://www.gi.de/service/informatiklexikon/detailansicht/article/complex-event-processing-cep.html

[EPTS 2013] The Home of the Event Processing Technical Society. (2013). – URL http://www.reactivemanifesto.org/

[Abowd u. a. 1999] Abowd, Gregory D. ; Dey, Anind K. ; Brown, Peter J. ; Davies, Nigel ; Smith, Mark ; Steggles, Pete: Towards a Better Understanding of Context and Context-Awareness. In: *Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*. London, UK, UK : Springer-Verlag, 1999 (HUC '99), S. 304–307. – URL http://dl.acm.org/citation.cfm?id=647985.743843. – ISBN 3-540-66550-1

[Baldauf u. a. 2007] Baldauf, Matthias ; Dustdar, Schahram ; Rosenberg, Florian: A Survey on Context-Aware Systems. In: *Int. J. Ad Hoc Ubiquitous Comput.* 2 (2007), Juni, Nr. 4, S. 263–277. – URL http://dx.doi.org/10.1504/IJAHUC.2007.014070. – ISSN 1743-8225

[Baldowski u. a. 2014] Baldowski, Mariusz ; Busch, Jan ; Pereki, Hodabalo ; Thiel-Clemen, Thomas: Ermittlung der Waldbiomasse mit Hilfe eines spatial gemischten Indikators für den Abdoulaye Forest, Togo. In: Wittmann, J. (Hrsg.) ; Müller, M. (Hrsg.): *Simulation in Umwelt- und Geowissenschaften, Workshop Leipzig* GI (Veranst.), Shaker, 2014. – URL http://141.22.29.101/wp-content/uploads/2014/09/Ermittlung-der-Waldbiomasse-M-Baldowsk-J-Busch-H-Pereki-T-Thiel-Clemen.pdf

[Bardram 2005a] Bardram, Jakob E.: Design, Implementation, and Evaluation of the Java Context Awareness Framework (JCAF). Aabogade 34, 8200 Aarhus N, Denmark : Centre for Pervasive Healthcare, Department of Computer Science, University of Aarhus, 2005, S. 5–11. – URL http://www.daimi.au.dk/~bardram/jcaf/jcaf.v15.pdf

[Bardram 2005b] Bardram, Jakob E.: The java context awareness framework (JCAF); a service infrastructure and programming framework for context-aware applications. In: *Proceedings of the Third international conference on Pervasive Computing*. Berlin, Heidelberg : Springer-Verlag, 2005 (PERVASIVE'05), S. 98–115. – URL http://dx.doi.org/10.1007/11428572_7. – ISBN 3-540-26008-0, 978-3-540-26008-0

[Bettini u. a. 2010] Bettini, Claudio ; Brdiczka, Oliver ; Henricksen, Karen ; Indulska, Jadwiga ; Nicklas, Daniela ; Ranganathan, Anand ; Riboni, Daniele: A Survey of Context Modelling and Reasoning Techniques. In: *Pervasive Mob. Comput.* 6 (2010), April, Nr. 2, S. 161–180. – URL http://dx.doi.org/10.1016/j.pmcj.2009.06.002. – ISSN 1574-1192

[Bruns und Dunkel 2010] Bruns, R. ; Dunkel, J.: Event-Driven Architecture:. Berlin, Heidelberg : Springer-Verlag, 2010, S. 20. – ISBN 978-3-642-02439-9

[van Bunningen u. a. 2005] Bunningen, A.H. van ; Feng, L. ; Apers, P. M G.: Context for ubiquitous data management. In: *Ubiquitous Data Management, 2005. UDM 2005. International Workshop on*, April 2005, S. 17–24

[Cugola und Margara 2012] Cugola, Gianpaolo ; Margara, Alessandro: Processing Flows of Information: From Data Stream to Complex Event Processing. In: *ACM Comput. Surv.* 44 (2012), Juni, Nr. 3, S. 15:1–15:62. – URL http://doi.acm.org/10.1145/2187671.2187677. – ISSN 0360-0300

[Dey u. a. 2001] Dey, Anind K. ; Abowd, Gregory D. ; Salber, Daniel: A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-aware Applications. In: *Hum.-Comput. Interact.* 16 (2001), Dezember, Nr. 2, S. 97–166. – URL http://dx.doi.org/10.1207/S15327051HCI16234_02. – ISSN 0737-0024

[Eckhoff 2013] Eckhoff, Malte: Ein Agenten-basiertes Gepardenmodell - Konzept und räumliche Wahrnehmung. Hamburg, HH, Germany, 2013

[Engel und Etzion 2011] Engel, Yagil ; Etzion, Opher: Towards Proactive Event-driven Computing. In: *Proceedings of the 5th ACM International Conference on Distributed Event-*

based System. New York, NY, USA : ACM, 2011 (DEBS '11), S. 125–136. – URL http://doi.acm.org/10.1145/2002259.2002279. – ISBN 978-1-4503-0423-8

[Etzion und Niblett 2010] Etzion, Opher ; Niblett, Peter: *Event Processing in Action*. 1st. Greenwich, CT, USA : Manning Publications Co., 2010. – ISBN 1935182218, 9781935182214

[Eugster u. a. 2003] Eugster, Patrick T. ; Felber, Pascal A. ; Guerraoui, Rachid ; Kermarrec, Anne-Marie: The Many Faces of Publish/Subscribe. In: *ACM Comput. Surv.* 35 (2003), Juni, Nr. 2, S. 114–131. – URL http://doi.acm.org/10.1145/857076.857078. – ISSN 0360-0300

[Gu u. a. 2004] Gu, Tao ; Pung, Hung K. ; Zhang, Da Q.: A Middleware for Building Context-Aware Mobile Services. In: *In Proceedings of IEEE Vehicular Technology Conference (VTC*, 2004

[Haghighi u. a. 2006] Haghighi, P.D. ; Zaslavsky, A. ; Krishnaswamy, S.: An Evaluation of Query Languages for Context-Aware Computing. In: *Database and Expert Systems Applications, 2006. DEXA '06. 17th International Workshop on*, 2006, S. 455–462. – ISSN 1529-4188

[Hilborn u. a. 2012] Hilborn, Anne ; Pettorelli, Nathalie ; Orme, C. David L. ; Durant, Sarah M.: Stalk and chase: how hunt stages affect hunting success in Serengeti cheetah. In: *Animal Behaviour* 84 (2012), Nr. 3, S. 701 – 706. – URL http://www.sciencedirect.com/science/article/pii/S0003347212002886. – ISSN 0003-3472

[Hofer u. a. 2003] Hofer, Thomas ; Schwinger, Wieland ; Pichler, Mario ; Leonhartsberger, Gerhard ; Altmann, Josef ; Retschitzegger, Werner: Context-Awareness on Mobile Devices - the Hydrogen Approach. In: *Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03) - Track 9 - Volume 9*. Washington, DC, USA : IEEE Computer Society, 2003 (HICSS '03), S. 292.1–. – URL http://dl.acm.org/citation.cfm?id=820756.821849. – ISBN 0-7695-1874-5

[Houser 2008] Houser, A.M.: *Spoor Density, Movement and Rehabilitation of Cheetahs in Botswana*. University of Pretoria, 2008. – URL http://books.google.de/books?id=ZLoSywAACAAJ

[Hüning u. a. 2014] Hüning, Christian ; Wilmans, Jason ; Feyerabend, Nils ; Thiel-Clemen, Thomas: MARS - A next-gen multi-agent simulation framework. In: Wittmann, J. (Hrsg.) ; Müller, M. (Hrsg.): *Simulation in Umwelt- und Geowissenschaften, Workshop Leipzig* GI

(Veranst.), Shaker, 2014. – URL http://3ten.de/download/marslife/MARS-A%20next%20gen%20simulation%20framework.pdf

[Jennings und Wooldridge 2000] Jennings, Nicholas R. ; Wooldridge, Michael: On agent-based software engineering. In: *Artificial Intelligence* 117 (2000), S. 277–296

[Kumar und Xie 2012] Kumar, Anup ; Xie, Bin: *Handbook of Mobile Systems Applications and Services*. 1st. Boston, MA, USA : Auerbach Publications, 2012. – ISBN 1439801525, 9781439801529

[Luckham und Schulte 2013] Luckham, David ; Schulte, Roy: Event Processing Glossary - Version 2.0. (2013). – URL http://www.complexevents.com/wp-content/uploads/2011/08/EPTS_Event_Processing_Glossary_v2.pdf

[Luckham 2001] Luckham, David C.: *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2001. – ISBN 0201727897

[Macal und North 2009] Macal, Charles M. ; North, Michael J.: Agent-based Modeling and Simulation. In: *Winter Simulation Conference*, Winter Simulation Conference, 2009 (WSC '09), S. 86–98. – URL http://dl.acm.org/citation.cfm?id=1995456.1995474. – ISBN 978-1-4244-5771-7

[Macal und North 2011] Macal, C.M. ; North, M.J.: Introductory tutorial: Agent-based modeling and simulation. In: *Simulation Conference (WSC), Proceedings of the 2011 Winter*, Dec 2011, S. 1451–1464. – ISSN 0891-7736

[Noetzel u. a. 2013] Noetzel, Carsten ; Reintjes, Ralf ; Thiel-Clemen, Thomas: Die Rolle öffentlicher Verkehrsmittel bei der Übertragung und Verbreitung von Krankheitserregern. In: Handels, H. (Hrsg.) ; Ingenerf, J. (Hrsg.): *58. Jahrestagung der Deutschen Gesellschaft für Medizinische Informatik, Biometrie und Epidemiologie e.V. (GMDS)*, September 2013

[Orkin 2002] Orkin, Jeff: Applying goal oriented action planning in games. In: *AI Game Programming Wisdom 2*. Charles River Media, 2002, S. 217–229. – URL http://web.media.mit.edu/~{}jorkin/GOAP_draft_AIWisdom2_2003.pdf

[Pereki u. a. 2013] Pereki, Hodabalo ; Wala, Kperkouma ; Thiel-Clemen, Thomas ; Bessike, Michael P B. ; Zida, M ; Dourma, Marra ; Batawila, Komlan ; Akpagana, Koffi: Woody species diversity and important value indices in dense dry forests in Abdoulaye Wildlife Reserve (Togo , West Africa). In: *International Journal of Biodiversity and Conservation* 5 (2013),

Nr. June, S. 358–366. – URL http://www.academicjournals.org/article/article1380047402_Pereki%20et%20al.pdf

[Railsback u. a. 2006] Railsback, Steven F. ; Lytinen, Steven L. ; Jackson, Stephen K.: Agent-based Simulation Platforms: Review and Development Recommendations. In: *Simulation* 82 (2006), September, Nr. 9, S. 609–623. – URL http://dx.doi.org/10.1177/0037549706073695. – ISSN 0037-5497

[Ranganathan und Campbell 2003] Ranganathan, Anand ; Campbell, Roy H.: An infrastructure for context-awareness based on first order logic. In: *Personal Ubiquitous Comput.* 7 (2003), dec, Nr. 6, S. 353–364. – URL http://dx.doi.org/10.1007/s00779-003-0251-x. – ISSN 1617-4909

[Reichle u. a. 2008] Reichle, Roland ; Wagner, Michael ; Khan, Mohammad U. ; Geihs, Kurt ; Valla, Massimo ; Fra, Cristina ; Paspallis, Nearchos ; Papadopoulos, George A.: A Context Query Language for Pervasive Computing Environments. In: *Proceedings of the 2008 Sixth Annual IEEE International Conference on Pervasive Computing and Communications*. Washington, DC, USA : IEEE Computer Society, 2008 (PERCOM '08), S. 434–440. – URL http://dx.doi.org/10.1109/PERCOM.2008.29. – ISBN 978-0-7695-3113-7

[Russell und Norvig 2010] Russell, S.J. ; Norvig, P.: *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2010 (Prentice Hall series in artificial intelligence). – URL http://books.google.de/books?id=8jZBksh-bUMC. – ISBN 9780136042594

[Schönmann 2012] Schönmann, Frank: BDI-Architektur (Beliefs - Desires - Intentions), 2003. (2012). – URL http://www.techfak.unibielefeld.de/ags/wbski/lehre/digiSA/S03/KogMod/BDI_slides.pdf

[Strang und Linnhoff-Popien 2004] Strang, Thomas ; Linnhoff-Popien, Claudia: A Context Modeling Survey. In: *In: Workshop on Advanced Context Modelling, Reasoning and Management, UbiComp 2004 - The Sixth International Conference on Ubiquitous Computing, Nottingham/England*, 2004

[Tennenhouse 2000] Tennenhouse, David: Proactive Computing. In: *Commun. ACM* 43 (2000), Mai, Nr. 5, S. 43–50. – URL http://doi.acm.org/10.1145/332833.332837. – ISSN 0001-0782

[Thiel 2013] Thiel, Christian: Analysis of techniques for partitioning and partial synchronization in heavily distributed multi-agent based crowd simulations. In: *Master Thesis*. Hamburg University of Applied Sciences, 2013

[Thiel-Clemen u. a. 2011] Thiel-Clemen, Thomas ; Köster, Gerta ; Sarstedt, Stefan: WALK - Emotion-based pedestrian movement simulation in evacuation scenarios. In: Wittmann, Jochen (Hrsg.) ; Wohlgemuth, Volker (Hrsg.): *Simulation in Umwelt- und Geowissenschaften, Workshop Berlin* Gesellschaft für Informatik (Veranst.), Shaker, 2011, S. 103–112

[Wu u. a. 2006] Wu, Eugene ; Diao, Yanlei ; Rizvi, Shariq: High-performance Complex Event Processing over Streams. In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : ACM, 2006 (SIGMOD '06), S. 407–418. – URL http://doi.acm.org/10.1145/1142473.1142520. – ISBN 1-59593-434-0

[Zampunieris und Dobrican 2014] Zampunieris, Denis ; Dobrican, Remus-Alexandru: Moving Towards a Distributed Network of Proactive, Self-Adaptive and Context-Aware Systems. (2014), Mai, S. 22–26. – ISSN 2308-4146

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 1. Oktober 2014 Daniel Steiman