DIPLOMARBEIT

Schnittstellendesign einer kognitiven Architektur

an der
Fakultät für Elektrotechnik und Informationstechnik,
der Technischen Universität Wien
zur Erlangung des akademischen Grades
Diplom-Ingenieur/in

unter Betreuung von

O. Univ. Prof. Dipl.-Ing. Dr.techn. Dietmar Dietrich
Institutsnummer: 384
Institut für Computertechnik

und

Dipl.-Ing. Alexander Wendt
Institutsnummer: 384
Institut für Computertechnik

von

Lukas Herret BSc.
Matr. Nr. 0826530
Mühlenweg 8
2440, Gramatneusiedl

18.09.2014

_____

Kurzfassung

Das Artificial-Recognition-System (ARS) Projekt beschäftigt sich mit dem Entwickeln einer kognitiven Architektur für autonome Agenten. Diese Agenten sollen in der Lage sein Situationen in einer simulierten Welt wahrzunehmen, und auf diese in angemessener Form reagieren zu können. Dafür wurde ein, auf der psychoanalytischen Theorie aufbauendes, in funktionale Schichten unterteiltes, Modell entwickelt. Die oberste Schicht dieses Modelles bildet eine Decision Unit die von der darunterliegenden Hardware gelöst, nur durch Funktionen beschrieben wird.

Die Funktionalität des Modelles wird durch Simulation in der Multi-Agenten Simulationsumgebung MASON überprüft. Durch die dichte Vernetzung zwischen Framework und Decision Unit ist es derzeit nicht möglich diese in anderen Anwendungen zu integrieren. Es stellt sich also die Frage, ob es möglich ist, eine mit der Simulationsumgebung des ARS-Projektes eng verzahnte Decision Unit herauszulösen, und in einer alternativen Simulationsumgebung einsetzten zu können.

Mit Hilfe der Ergebnisse der vor Beginn der Arbeit definierten Use Cases kann gezeigt werden, dass es möglich ist durch den Einsatz einer Mediatorschicht zwischen Simulationsumgebung und Decision Unit diese in alternativen Simulationsumgebungen zu verwenden. Die verwendete Simulationsumgebung wurde ebenfalls aus dem Gebiet der Artificial-Life-Simulation gewählt, da der Fokus der Arbeit auf dem Design des Interfaces und nicht auf der Übertragung der ARS-Konzepte in andere Anwendungsgebiete liegt. Die Trennung der Decision Unit von der ARS-Simulationsumgebung ermöglicht jedoch nicht nur den Einsatz in der angegebenen Simulationsumgebung, sondern soll als Basis für zukünftige Projekte dienen, und das Projekt ARS für alternative Anwendungsmöglichkeiten öffnen.

Abstract

The Artificial Recognition System (ARS) project deals with the development of a cognitive architecture for autonomous agents. These agents are able to perceive there situation in a simulated world, and to respond in an appropriate manner. Therefore, a hierarchical model, built on the psychoanalytic theory, was developed. The uppermost layer of this model constitutes a decision unit which is released from the underlying hardware and is only described by functions.

The functionality of the model is verified through simulation in the multi agent simulation framework MASON. Because of the dense network between the framework and the decision unit it is currently not possible to integrate the project into other applications. This raises the question whether it is possible to separate a psychoanalytically inspired decision unit from the ARS simulation environment and use it into an alternative simulation environment.

Using the results of the defined use cases it can be shown that this is possible by a adding an interface between simulation environment and decision unit it is possible to use the decision unit in alternative environments. The used environment is also form the field of artificial life simulation, to concentrate on interface design and not on mapping the concepts to other fields. The separation of the decision from the artificial life simulation not only allows the use into artificial life projects, but will serve as a foundation for future projects, and make the project ARS useable for technical problems.

Inhaltsverzeichnis

Abkürzungen

2D	2 dimensional
3D	3 dimensional
AI	Artificial Intelligence
ARS	Artificial Recognition System
BDI	Belief, Desire and Intention
DLL	Dynamic Linked Library
MASON	Multi-Agent Simulator of Neighbourhoods
SOAR	State Operator Apply Result
SGIO	SOAR general input output
UUT	Unit under Test
XML	Extensible Markup Language

1. Einführung

Artificial-Intelligence (AI) ist ein Forschungsgebiet, das sich mit dem Lösen komplexer Problemstellungen beschäftigt. Aktuelle AI-Ansätze scheitern jedoch immer noch daran, allgemeine komplexe Probleme zu lösen [DS00]. Es gibt jedoch eine Entscheidungseinheit die mit diesen komplexen Problemen umzugehen weiß. Die menschliche Psyche ist allen AI-Ansätzen bei weiten überlegen. Aus diesem Grund hat das Institut für Computertechnik [2] der Technischen Universität Wien das Projekt Artificial-Recognition-System (ARS) [1] ins Leben gerufen. Das ambitionierte Ziel des Projektes ist es, ein Modell der menschlichen Psyche zu entwickeln. Dieses soll es ermöglichen, menschliche Informationsverarbeitung zur Lösung von technischen Problemstellungen einzusetzen. Die menschliche Psyche, als Vorbild für technische Problemlösungen zu verwenden, gibt es schon lange. Keines dieser Systeme ist jedoch in der Lage komplexe Aufgabe, wie zum Beispiel die autonome Überwachung eines Flughafens zu erfüllen. Im Sinne der Top-Down-Entwicklung ist es notwendig das System als Ganzes zu betrachten. Das ARS-Projekt verfolgt deshalb den Grundsatz, dass die Lösung komplexer Probleme nur mit einem vollständigen Modell der menschlichen Psyche möglich ist. Als Grundlage dieses Funktionsmodelles dienen die Modelle der Psychoanalyse, die auf ein technisch implementierbares Funktionsmodell herunter gebrochen werden kann. Das Ziel des ARS-Projektes zum heutigen Zeitpunkt kann es nicht sein einen Regler zu Kontrolle komplexer System zu entwickeln. Es soll jedoch ein Funktionsmodell entwickelt werden, mit dessen Hilfe die Arbeitsweise der menschlichen Psyche verstanden werden kann. Die Entwicklung dieses Modelles erfolgt, getrieben durch Fallbeschreibungen. Diese definieren Handlungsabläufe die als Basis für die Entwicklung des Funktionsmodells herangezogen werden. Die Evaluierung des Modells erfolgt in einer Artificial-Life-Simulation in der die Fallbeschreibung nachgestellt wird. Die Ergebnisse der Simulation werfen zum Teil neue Problemstellungen auf. Die Erkenntnisse daraus fließen in einer weiteren Iteration wieder in den Prozess der Modellbildung ein. Dadurch ergibt sich ein zyklischer Arbeitsprozess, der mit jeder Iteration eine Weiterentwicklung des Funktionsmodelles mit sich bringt.

1.1 Motivation

Wie bereits erwähnt wird innerhalb des ARS-Entwicklungszyklus das Funktionsmodell durch das Ausführen von Fallbeschreibungen in einer 2D Artificial-Life-Simulation verifiziert. Als Simulationsumgebung wird das Multi-Agenten Simulationsframework MASON [LCRP+05] [3] verwendet. Das MASON Framework stellt dabei lediglich Werkzeuge für Scheduling, Physics-Engine, Simulationssteuerung und Visualisierung zur Verfügung. Die darauf aufbauende Artificial-Life-Simulation

wurde im Zuge des ARS-Projektes entwickelt. Abbildung 1 zeigt einen Screenshot dieser Simulationsumgebung. Das Beispiel zeigt dabei zwei, um eine Nahrungsquelle konkurrierende, Agenten und einen Stein als Hindernis.

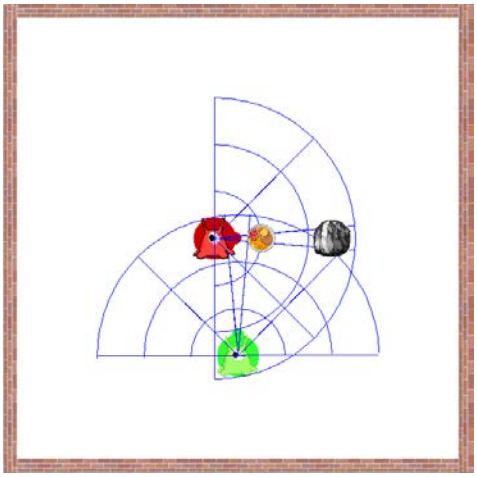

Abbildung 1: ARS-Simulation

Da die Entwicklung der Decision Unit im Fokus des Projektes steht, wurde die Simulationsumgebung immer wieder den Anforderungen dieser angepasst. Durch diesen gemeinsamen Entwicklungsprozess wurde die klare Trennung der beiden Funktionsblöcke nicht konsequent eingehalten. Abbildung 2 zeigt einen schematischen Aufbau der Simulationsumgebung mit der integrierten Decision Unit und soll die enge Kopplung verdeutlichen.

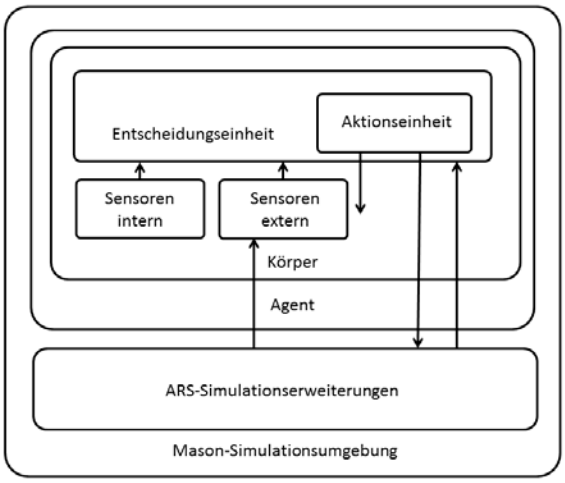

Abbildung 2: ARS-Simulationsumgebung

Statt einer klar definierten Schnittstelle gibt es mehrere Berührungspunkte die je nach Anforderung unterschiedlich ausgeführt wurden. Besonders deutlich wird die enge Kopplung zwischen Simulationsumgebung und Decision Unit dadurch, dass die Action-Engine, wie in Abbildung 2 zu sehen, in die Decision Unit integriert wurde. Dadurch, dass keine klare Schnittstelle definiert wurde, ist es nur sehr schwer möglich, die Decision Unit außerhalb der jetzigen Simulationsumgebung einzusetzen.

Im Moment ist dies sogar nur möglich wenn die ARS-Simulationsumgebung im Hintergrund ausgeführt wird.

Mögliche alternative Umgebungen wären andere Simulations- oder Visualisierungsframeworks. Hier wurde bereits versucht die Decision Unit mit Hilfe des Pogamut Frameworks [GBP11] innerhalb der Unreal Umgebung einzusetzen, um eine Vergleichbarkeit mit anderen Modellen zu ermöglichen. Eine weitere Visualisierungsumgebung wäre Second-Life [VFBD11]. Dadurch wäre es möglich das ARS-Funktionsmodell in einer realistischen Umgebung auszuführen und zu validieren. Die jetzige Simulationsumgebung wird, basierend auf den Anforderungen des Funktionsmodelles, von der Forschungsgruppe selbst entwickelt. Mit steigender Komplexität steigen auch die Anforderungen an die Simulationsumgebung und eine Eigenentwicklung ist mit hohem Aufwand verbunden. Der Umstieg auf Second-Life als Simulationsumgebung würde diesen Aufwand verringern. Außerdem würde eine realistische Simulationsumgebung die Möglichkeiten der Simulation erheblich erweitern. Weiters wäre es möglich die ARS Decision Unit zur Lösung von technischen Problemstellungen einzusetzen. Wie bereits erwähnt, gehört die Integration der ARS Decision Unit in einen technischen Prozess nicht zu den primären Zielen des Projektes. Die Decision Unit Architektur soll jedoch diese Möglichkeit nicht verwehren. Da die Regelung von komplexen Systemen das Ziel der Forschung darstellen soll.

Ein weiteres Problem bei der jetzigen Implementierung besteht darin, die simulierten Objekte unabhängig von der Simulation zu machen. Die Richtigkeit einer Simulation beruht darauf, dass die Simulationsumgebung keinen Einfluss auf die zu simulierende Funktionseinheit haben darf. Durch die enge Kopplung ist es nicht möglich die Ausführung der Decision Unit von der Simulationsumgebung abzukoppeln. Durch diese Abhängigkeit wird die Decision Unit von der Simulationsumgebung beeinflusst und es kommt zu verfälschten Simulationsergebnissen.

Es gab bereits Projekte die versucht haben die ARS Decision Unit in alternativen Umgebungen einzusetzen [Fer12] [Tor12] [Ber13]. Dies konnte jedoch nur durch ein Anpassen der Decision Unit erfolgen. Außerdem mussten Teile der Simulationsumgebung im Hintergrund mitlaufen, um eine fehlerfreie Abarbeitung des Funktionsmodelles zu gewährleisten. Die Weiterentwicklung der Decision Unit erfolgt jedoch in der Artificial-Life-Simulation. Nach einigen Weiterentwicklungsschritten waren die abgespaltenen Decision Units nicht mehr mit der aktuellen Version kompatibel und das Projekt konnte nicht mehr mit der neuesten Version der Decision Unit ausgeführt werden.

1.2 Problem Definition

Die Arbeit beschäftigt sich mit der Frage, wie sich eine aus der Artificial-Life-Simulation stammende, psychoanalytisch inspirierte Decision Unit in alternativen Umgebungen einsetzen lässt. Die Aufgabe ist es also eine Schnittstelle zu entwerfen, die es erlaubt die ARS Decision Unit sowohl in der jetzt verwendeten MASON Simulationsumgebung, als auch unabhängig von dieser in anderen Simulationsumgebungen zu verwenden. Dazu soll eine Schnittstelle zur ARS Decision Unit geschaffen werden.

Es stellt sich also die Frage wie die bidirektionale Schnittstelle einer psychoanalytisch inspirierten Decision Unit aufgebaut sein muss, um diese in alternativen Umgebungen einsetzen zu können. Da-

bei soll nicht nur der Datenfluss, sondern auch die Ausführungssteuerung und die Instanziierung der Decision Unit berücksichtigt werden. Das Problem besteht darin, dass die Simulationsumgebung und die Decision Unit gemeinsam entwickelt wurden und aufeinander aufbauen. Es wurde zwar teilweise auf eine Trennung dieser beiden Einheiten geachtet, dies wurde jedoch nicht konsequent durchgezogen. Durch den Einsatz der Schnittstelle soll die Unabhängigkeit der Funktionsblöcke gewehrleistet werden. Mit Hilfe dieser soll die Decision Unit sowohl in die ARS-Simulationsumgebung als auch in die alternative Simulationsumgebung Miklas [4] eingebaut werden. Die zur Verifikation des ARS-Modells verwendeten Fallbeschreibungen sollen in der alternativen Simulationsumgebung ebenfalls aufgebaut werden. In beiden Umgebungen sollen die Fallbeschreibungen zum selben Ergebnis kommen.

1.3 Task Setting

Die Aufgabe ist es die ARS Decision Unit aus der jetzigen Simulationsumgebung herauszulösen und für den Einsatz in anderen Umgebungen vorzubereiten. Der erste Teil der Aufgabe beschäftigt sich damit, die Decision Unit aus der Simulationsumgebung herauszulösen. Der zweite Teil der Aufgabe wird es sein, ein Interface zu spezifizieren, welches es erlaubt die Decision Unit sowohl in der jetzigen Simulationsumgebung als auch in anderen Umgebungen einsetzen zu können. Dabei muss auf jeden Fall darauf geachtet werden, dass es zu keinen parallelen Strukturen kommt, da in der weiteren Entwicklung der Decision Unit meist nur eine dieser Strukturen gewartet wird und die anderen ihre Gültigkeit verlieren. Weiters soll darauf geachtet werden, dass die Decision Unit vollkommen unabhängig von der jetzigen Simulationsumgebung eingesetzt werden kann

Folgende Schritte werden dazu nötig sein:

- Analysieren der Abhängigkeiten zwischen Simulationsumgebung und Decision Unit
- Trennen der Decision Unit von der Simulationsumgebung
- Analyse der Anforderungen der Decision Unit und der Umgebungen an die Schnittstelle
- Design einer Schnittstelle
- Anbinden der Decision Unit an die ARS-Simulationsumgebung mit Hilfe der Schnittstelle
- Anbinden der Decision Unit an eine alternative Simulationsumgebung
- Testen der Implementierung mittels Use Cases

Bei der Durchführung dieser Schritte muss darauf geachtet werden, dass folgende Punkt nicht zutreffen:

- Keine parallelen Schnittstellen für jede Decision Unit
- Keine Abhängigkeiten zwischen der Simulationsumgebung und der Decision Unit die über die definierte Schnittstelle hinaus gehen
- Keine Beeinflussung der Simulationsergebnisse durch die Schnittstelle

1.4 Methodologie

Nach der Definition der Aufgabenstellung wurde die Lösungsstrategie definiert. Als Basis der Überlegungen zur Methodik wurde das V-Model [FHKS09 S. 20] als bewährte Software Entwicklungsstrategie gewählt. Darauf aufbauend wurde der in Abbildung 3 gezeigt Ablauf definiert. Die linke Hälfte zeigt die benötigten Arbeitsschritte bis zur Implementierung. In der rechten Hälfte wird gezeigt wie die jeweiligen Arbeitsschritte evaluiert werden können.

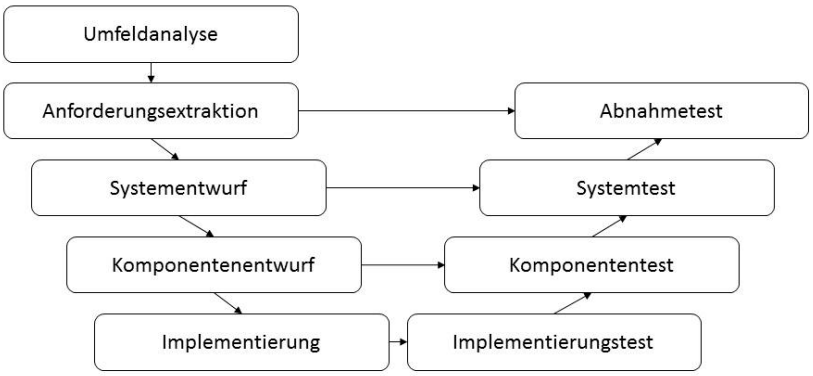

Abbildung 3: V-Model [FHKS09 S. 20]

Umfeldanalyse

Zu Beginn der Arbeit wurde eine umfassende Umfeldanalyse durchgeführt. Die Umfeldanalyse kann in die drei Bereiche Decision Unit, Simulationsumgebungen und frühere Arbeiten eingeteilt werden.

Der erste Teil beschäftigt sich mit der Analyse der Decision Unit. Einen Schwerpunkt der Analyse stellten die allgemeinen Konzepte des Modells so wie die verwendeten Datenstrukturen dar. Wobei hier besonders der Aufbau der Ein- und Ausgänge betrachtet wurde. Außerdem wurde die Ablaufsteuerung der Decision Unit betrachtet.

Im Zweiten Teil der Umfeldanalyse werden die Simulationsumgebungen näher betrachtet. Dabei wurde die auf MASON basierende Artificial Life Simulation, in der das ARS-Projekt entwickelt wird, analysiert (2.2). Hier wurden zum einen das Simulationsframework MASON und zum anderen die auf dem Framework aufbauenden Erweiterungen betrachtet. Weiters wurden die Ausführungsumgebungen Miklas (Kapitel 2.3) und Unreal (Kapitel 2.4.1) mit einbezogen.

Den dritten Teil der stellt die Analyse früherer Arbeiten dar. Am Rande des ARS-Projektes wurden bereits Projekte durchgeführt, mit dem Ziel die ARS Decision Unit in anderen Umgebungen einzusetzen (Kapitel 2.4). Da das Problem der Schnittstelle immer noch besteht, können diese Projekte zumindest teilweise als gescheitert angesehen werden. Es musste also analysiert werden, warum innerhalb dieser Projekte keine dauerhafte Lösung gefunden werden konnte.

Anforderungsextraktion

Auf Basis der drei Analyse Schritte wurden Anforderungen an die Schnittstelle definiert. Diese Anforderungen bilden den Ausgangspunkt der Modellierung und dienten in weiterer Folge als Evaluierungskriterien. Die Anforderungen lassen sich in funktionale und nicht funktionale Anforderungen unterteilen.

Systementwurf

Basierend auf den extrahierten Anforderungen wurde unter Berücksichtigung des State of the Art bezüglich Kommunikation Schnittstellen und Agentensimulation ein Modell der Schnittstelle entwickelt. Dabei musste sowohl auf die funktionalen als auch auf die nicht funktionalen Anforderungen Rücksicht genommen werden.

Komponentenentwurf

Am Ende des Systementwurfes steht ein Modell, welches zur weiteren Verarbeitung in Komponenten zerlegt werden kann. Die einzelnen Komponenten wurden basierend auf den funktionalen Anforderungen näher spezifiziert, um eine Basis für die Implementierung zu bilden.

Implementierung

Dieser Punkt teilt sich ein zwei Bereiche. Zum einen wurde das entwickelte Modell der Schnittstelle implementiert. Anschließend wurde die Implementierung an die Decision Unit angeschlossen. Um diese nun verwenden zu können, wurde die Schnittstelle zum einen mit der ARS-Simulationsumgebung und zum anderen mit Miklas zusammengeschlossen.

Implementierungstest

Im Implementierungstest wird die Richtigkeit der Implementierung anhand der von der Modellierung vorgegeben Anforderungen getestet. Diese Tests erfolgen parallel zur Implementierung für möglichst kleine unabhängige Funktionsblöcke. Dadurch können Implementierungsfehler schon zu einer sehr frühen Phase behoben werden.

Komponententest

Jede der definierten Komponenten des Modells wurde als unabhängige Einheit definiert, und kann deshalb auch so getestet werden. Dazu wurde die Implementierung der einzelnen Komponenten in eine Testumgebung eingebettet und auf Basis der in der im Komponentenentwurf definierten Anforderungen getestet. Dabei wurden die Komponenten getrennt voneinander getestet.

Systemtest

Parallel zur Anforderungsdefinition wurden Testfälle definiert, die es ermöglichten die Anforderungen zu verifizieren. Diese Testfälle zeigten zum einen das Zusammenspiel der einzelnen Komponenten, zum anderen wurde die Integration in die Decision Unit und die jeweiligen Simulationsumgebungen aufgezeigt.

Abnahmetest

Der Abnahmetest stellte die letzte Überprüfung vor der Fertigstellung dar. Dazu wurden Use Cases verwendet um die Kompatibilität zu den Simulationsumgebungen aufzuzeigen. In der ARS eigener Simulationsumgebung wird bereits mit Use Cases gearbeitet um das Modell zu verifizieren. Diese wurden als Vorbild herangezogen und in Miklas nachgestellt. Die Kompatibilität wurde aufgezeigt indem es in beiden Simulation der Use Cases zu denselben Ergebnissen kam.

2. State of the Art and Related Work

Nach der Einführung in die Thematik, folgt nun ein Überblick über relevante Arbeiten auf diesem Gebiet. Zuerst wird das Projektumfeld analysiert. Dabei werden sowohl die ARS Decision Unit als auch die verwendeten Ausführungsumgebungen betrachtet. Da es bereits Arbeiten zu dem Thema „Schnittstelle für die ARS Decision Unit“ gibt, werden diese betrachtet, um auf deren Ergebnissen aufbauen zu können. Außerdem werden Schnittstellen zu anderen kognitiven Simulationen untersucht, da diese ähnlich Anforderungen berücksichtigen müssen. Zum Abschluss werden relevante Aspekte der Softwareentwicklung, des Schnittstellendesignsund die dazugehörigen Design Pattern beschrieben.

2.1 Artificial Recognition System (ARS)

Das Projekt Artificial Recognition System (ARS) [1] beschäftigt sich mit der Entwicklung einer kognitiven Architektur. Die Idee dazu wurde vor über zehn Jahren von einem Team der Technischen Universität Wien rund um Professor Dietrich geboren. Das aus der Automatisierungstechnik stammende Team hatte die Intention, die immer größer werdende Zahl an intelligenten Knoten in der Gebäudeautomation in den Griff zu bekommen [DS00]. Das Projekt konzentrierte sich jedoch nicht darauf, ein Modell für eine spezielle Anwendung zu entwickeln. Es sollte ein Modell der Allgemeinen Künstlichen Intelligenz modelliert werden [SDW+13]. Dabei wird versucht ein Modell der menschlichen Informationsverarbeitung zu entwickeln, um dieses für technische Problemstellungen anwendbar zu machen. Die Natur als Grundlage für technische Entwicklungen zu verwenden, ist weit verbreitet. Meist werden jedoch nur Teilaspekte betrachtet, um spezielle Probleme lösen zu können. Das ARS-Projekt verfolgt jedoch einen ganzheitlichen Ansatz. Daraus ergibt sich, dass ein Modell der gesamten menschlichen Informationsverarbeitung entwickelt werden muss. Als Grundlage für die Entwicklung des Modells wurde die Psychoanalyse gewählt [DSB+13]. Wobei hier nicht die Therapieform, sondern das in der Metapsychologie beschriebene Modell der menschlichen Informationsverarbeitung herangezogen wird. Die Psychoanalyse wurde als Theorie gewählt, da sie das einzige, mit der Top Down Entwicklung kompatible, Funktionsmodell der menschlichen Psyche liefert [Deu11 S. 4]. Dieses Modell wird im ARS-Projekt nach dem top-down Prinzip in ein technisches Funktionsmodell umgesetzt. Die Vorgehensweise nach dem top-down Prinzip bedeutet, dass ausgehend von einem Modell mit hohem Abstraktionsgrad dieses Schritt für Schritt heruntergebrochen wird, bis eine Entwicklungsstufe erreicht ist, die genau genug spezifiziert ist, um dessen Funktionen implementieren zu können.

2.1.1 Modellentwicklung

Die Modellentwicklung im ARS-Projekt erfolgt in einem Interdisziplinären Team aus Psychoanalytikern und Techniker. Dabei werden in mehreren Zyklen Modellvorschläge zwischen den beiden Disziplinen ausgetauscht, und so versucht ein Modell zu finden, dass beiden Disziplinen Rechnung trägt [Muc13 S. 17]. Dabei wird, wie in der Computertechnik üblich im Top Down Verfahren vorgegangen [Lan10 S. 68].

Während der Modellierung wird bereits auf die spätere Verifizierbarkeit geachtet. Dies geschieht indem die Modellierung Use Case gesteuert erfolgt [BGSW13]. Das heißt der definierte Use Case legt die Anforderungen an das Modell fest. In Abbildung 4 ist der Use Case 1 zu sehen.

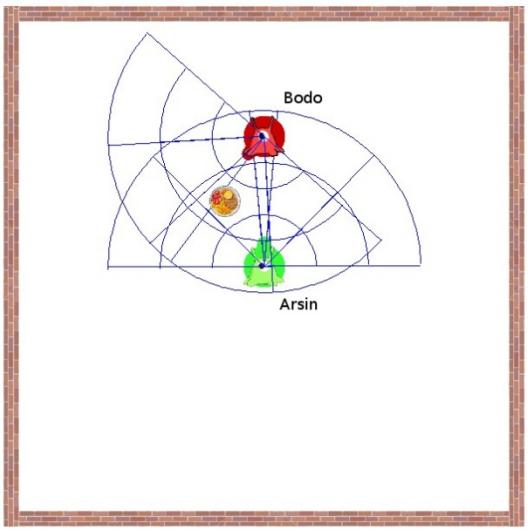

Abbildung 4: Use Case 1

Der Use Case 1 stellt zwei Agenten dar die um eine Nahrungsquelle (Schnitzel) konkurrieren. Arsin wird von der ARS Decision Unit gesteuert, während Bodo ein passiver Agent ist. Zu der in Abbildung 4 zu sehenden Ausgangssituation wurden mehrere Abläufe definiert, die vom Funktionsmodell nur durch Änderung der Eingänge erzeugt werden müssen.

- Arsin isst das Schnitzel
- Arsin flieht
- Arsin teilt das Schnitzel mit Bodo
- Arsin übergibt Bodo das Schnitzel
- Arsin denkt nur nach handelt aber nicht
- Arsin schlägt Bodo

2.1.2 Funktionsmodell

Als Ergebnis des Entwicklungsprozesses wurde ein Funktionsmodell bestehend aus drei Schichten definiert [Muc13 S. 63]. In Abbildung 5 ist das Modell mit den drei Schichten, Neural-Layer, Neuro-symbolic-Layer und Mental-Layer zu sehen. Dabei wurden nach dem Vorbild des ISO/OSI Models (siehe Kapitel 2.6.3) Interfaces (I0, I1 und I2) zu den darunter und darüberlegenden Schichten definiert. Dadurch ist es möglich die Funktionen innerhalb der Schicht auszutauschen solange die Spezifikation des Interfaces eingehalten wird. Die Entwicklungsarbeit innerhalb des ARS-Projektes konzentriert sich hauptsächlich auf die Modellierung des Mental-Layers [Muc13 S. 64]. Die Funktionen innerhalb der beiden anderen Layer können auf Grund des oben beschrieben Schichtenmodells vernachlässigt werden, solange die definierten Interfaces eingehalten werden.

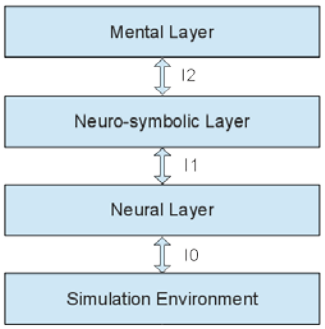

Abbildung 5: ARS-Schichtenmodell [Muc13 S. 63]

Neural-Layer

Der neuronale Layer stellt eine Beschreibung des menschlichen Nervensystems dar. Er beschreibt also die Hardware in Form von Sensoren und Aktuatoren.

Neuro-symbolc-Layer

Über den Neuro-Symbolischen Layer gibt es noch kein Beschreibungsmodell. Durch die Definition der Interfaces zum Neuronalen und Psychischen Layer kann jedoch ein Adapter entwickelt werden, der die Überführung der Neuronalen Daten in den Psychischen Layer ermöglicht.

Mental-Layer

Wie bereits erwähnt konzentriert sich die Modellierungsarbeit im ARS-Projekt hautsächlich auf den mentalen Layer. Dieser stellt das Kernstück des Funktionsmodelles dar und modelliert, basierend auf den Konzepten der Psychoanalyse, menschliche Informationsverarbeitung. Daraus ergibt sich ein Funktionsmodell, das in Anhang A zu sehen ist. Zur besseren Übersicht wurden die Funktionsblöcke in funktional zusammenhängende Tracks zusammengefasst. Diese Tracks sind in Abbildung6 und Tabelle 1 zu sehen und beschreiben den Aufbau des mentalen Layers.

Sexual Drive Track	Die Sexualtriebschiene verarbeitet Eingänge die zur Erzeugung von Sexualtrieben herangezogen werden. Dazu zählt die Signale aus den Erogenen Zonen sowie ein kontinuierlicher Libido Zufluss.

Self-Preservation-Drive-Track	Die Selbsterhaltungstriebschiene verarbeitet Signale aus den Organen des Körpers und erstellt daraus körperliche Bedürfnisse. Aus diesen werden in weitere Folge die Selbsterhaltungstriebe gebildet.
Drive-Track	In der Triebschiene werden die Inputs der Selbsterhaltungs- und Sexualtriebschiene verarbeitet und basierend auf diesen Werten Triebrepräsentanzen gebildet. Diese werden in weiterer Folge mit Erinnerungen assoziiert.
Environment-Perception-Track	In der Umgebungswahrnehmungsschiene werden Reize der Außenwelt über verschiedene Modalitäten aufgenommen und zu Wahrnehmungsobjekten gruppiert.
Body-Perception-Track	Die Körperwahrnehmungschiene verarbeitet Sensordaten aus körperinternen Sensoren.
Preception-Track	In der Wahrnehmungsschiene werden sowohl Umgebungs- als auch Körperwahrnehmungsdaten zusammengeführt und weiter verarbeitet. Die Daten werden mit Erinnerungen verglichen und mit weiteren Informationen verknüpft.
Defense-Track	Vor der Abwehrschiene werden Wahrnehmungs- und Triebinformationen miteinander verbunden und danach in die Abwehrschiene geführt. Basieren auf Über-Ich Regeln können Abwehrmechanismen ausgelöst werden, die die Inhalte der Daten verändern können.
Conversation-Track	In der Umwandlungsschiene werden die Daten in die Strukturen des Sekundärvorganges umgewandelt. Nur Daten die die Abwehr passiert haben können umgewandelt werden. Nach dieser Schiene sind die Daten bewusstseinsfähig.
Selection of Desire & Demand-Track	Der SelectionofDesire und Demand Track nutzt die Informationen des Primärvorganges um den Drang eine Triebes zu bewerten und unter Einbeziehung von Emotionen und Sozialen Regeln die Handlungsplanung vorzubereiten.
Selection-Track	Die Auswahlschiene verarbeitet die Information die ihr von den vorhergegangenen Modulen zur Verfügung gestellt wird und erstellt daraus Handlungspläne.
Imagination-Track	Durch die Rückführung von Informationen ist es möglich zu Phantasieren und Handlungsabläufte geistig durchzuspielen.
Action-Track	In der Aktionsschiene werden geplante Handlungen in Aktionsbefehle umgewandelt die von der zur Verfügung stehenden Hardware ausgeführt werden können.

Tabelle 1: Beschreibung Funktionmodule

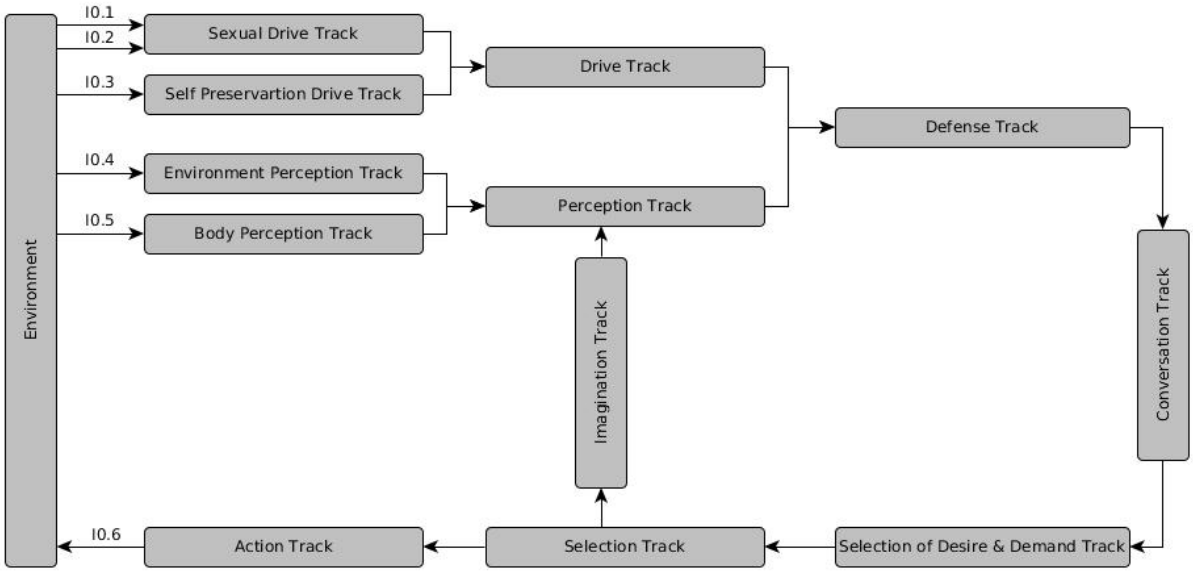

Abbildung6: Track View [BGSW13 S. 65]

2.1.3 Interfaces

Die Schnittstelle zwischen der ARS Decision Unit und dem Rest des Körpers besteht aus einem fünf-dimensionalen Input Vektor (I0.1, I0.2, I0.3, I0.4, I0.5)und einem ein-dimensionalen Output Vektor (I0.6). Abbildung 7 zeigt den Ausschnitt des Funktionsmodelles (ganzes Modell siehe Anhang A) in dem die Interfaces dargestellt sind.

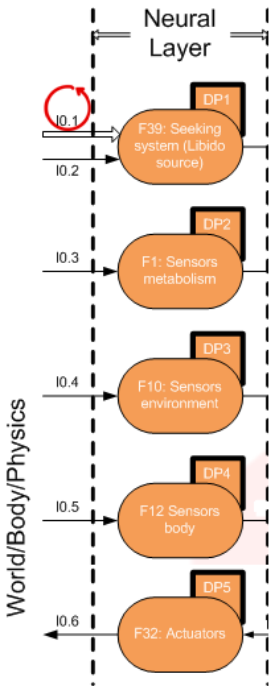

Abbildung 7: Interfaces Funktionsmodell

I0.1

Das Interface beschreibt einen double Wert, der zur Bildung der Sexualtriebe herangezogen wird. Der Kreis zeigt an, dass dieser konstante Double Wert vor jedem Modelldurchlauf generiert, und an die Module der Sexualtriebschiene übergeben wird.

I0.2

Mit I0.2 werden Signale aus den erogenen Zonen übergeben, die zur Befriedung der Sexualtriebe benötigt werden. Die Datenstruktur besteht aus dem Signalnamen und einem Double Wert. Diese Signale aus den Erogenen Zonen werden als atomare Events interpretiert und verlieren nach dem Durchlauf des Funktionsmodelles ihre Gültigkeit.

I0.3

Es werden homöostatische Werte übergeben die zu Bildung der Selbsterhaltungstriebe herangezogen werden. Die Datenstruktur besteht aus einem Signalnamen und einem Double Wert. Möglich Werte sind zum Beispiel der Füllstand des Magens oder der Blutzuckerwert. Aus diesen wird in weiterer Folge die Triebrepräsentanz des Hungertriebes generiert.

I0.4

I0.4 beinhaltet Informationen aus den Sensoren der Umwelt. Ein Bespiel für dieses Sensoren ist der visuelle Sensor. Abbildung 8 zeigt, in Form eines Objektdiagrammes, beispielhaft wie Daten aus diesem Sensor aussehen können. Es werden zwei Objekte erkannt, die jeweils über die Eigenschaften Shape, Status, Color, Orientation und Action verfügen. Der Name der bei den beiden Objekten hinterlegt ist wird nur zur Darstellung verwendet und hat keinen Einfluss auf die Verarbeitung innerhalb der Funktionsmodule. Die Objekte aus der Wahrnehmung werden im Laufe der Wahrnehmungsschiene anhand eines Erinnerungsabgleiches ihrer Eigenschaften identifiziert.

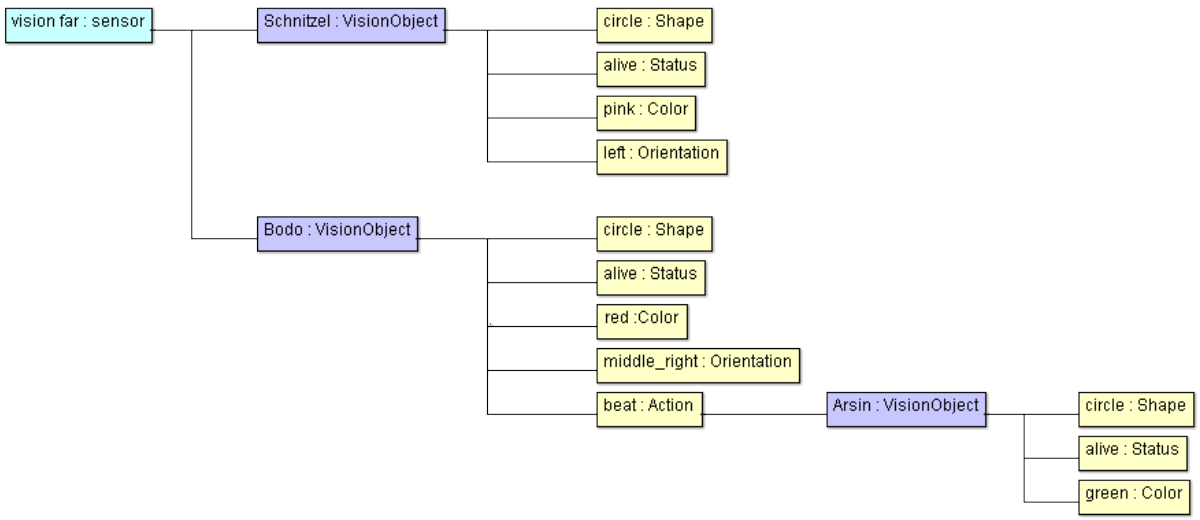

Abbildung 8: Objektdiagramm Datenstruktur I0.4

I0.5

Das Interface beschreibt Informationen aus den körperinternen Sensoren. Dabei wird dieselbe Datenstruktur wie bei I0.3 verwendet. Der Unterschied des Interfaces zu I0.3 ist, dass diese Daten nicht zur Triebgenerierung herangezogen werden, sondern als Daten der Wahrnehmung betrachtet werden.

I0.6

Über diesen Ausgang werden Aktionsbefehle an den Körper gesendet. Die Aktionsbefehle sind als String dargestellt und können, je nach Befehl, mit unterschiedlichen Eigenschaften verknüpft werden. Das als Objektdiagramm dargestellte Datum in Abbildung 9 soll anhand des Aktionsbefehles *turn* den Aufbau der Datenstruktur beschreiben. Der Aktionsbefehl ist in diesem Fall mit den Eigenschaften *Direction* und *Angle* assoziiert, die die Drehrichtung und den Drehwinkel bestimmen.

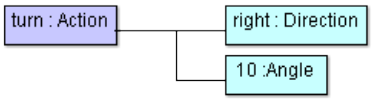

Abbildung 9: Objektdiagramm Datenstruktur I0.6

2.1.4 Ausführungsstruktur der Decision Unit

Das in Abbildung6 zu sehende Funktionsmodell wird mithilfe der Objektorientierten Programmiersprache Java implementiert. Es wird dabei in eine Decision Unit Struktur eingebettet, die die Ablaufsteuerung und die Schnittstelle zum restlichen Körper beinhaltet.

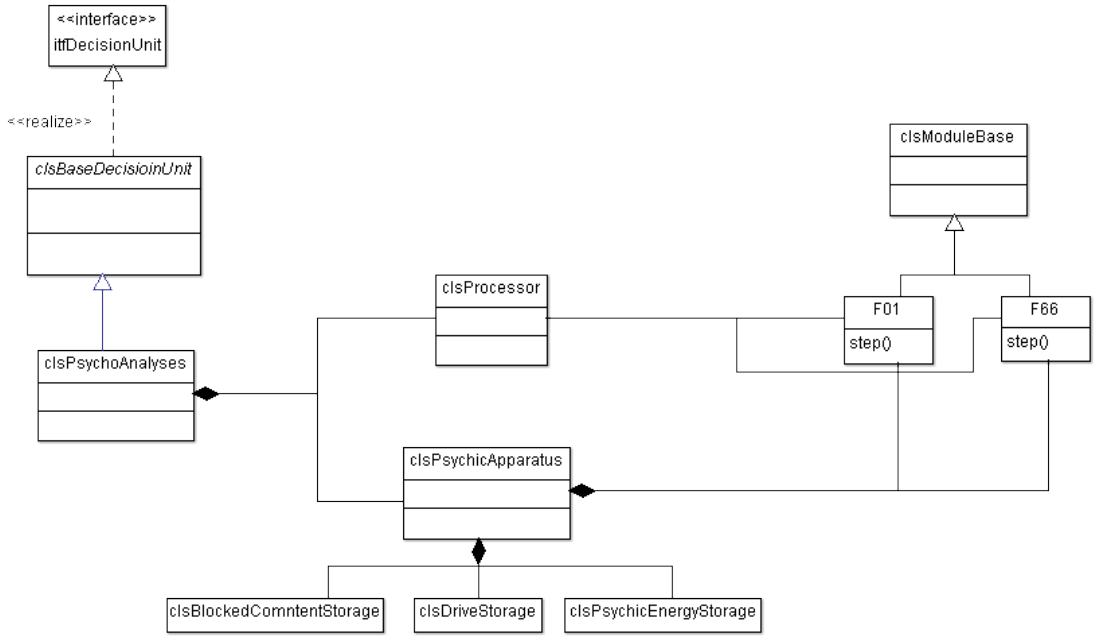

Abbildung 10: Aufbau Decision Unit [Deu11 S. 129]

In Abbildung 11 ist ein Klassendiagramm der Decision Unit zu sehen. Das Interface *itfDecisionUnit* und die abstrakte Klasse *clsBaseDecisionUnit* bilden den allgemeinen Teil, der auch von alternativen Decision Unit Ansätzen erweitert werden kann. Die Klasse *clsPsychicApparatus* bildet die Basisklasse der ARS Decision Unit. Sie beinhaltet sowohl alle Funktionsmodelle, als auch die im Modell beschriebenen Zwischenspeicher. Die Klasse *clsProcessor* ruft die einzelnen Funktionsmodule der Reihe nach auf und bestimmt so die Reihenfolge der Ausführung. Die Ausführung der Funktionsmodule erfolgt über die Funktion *step()*. Die in der Abbildung gezeigten Funktionsmodule *F01* und *F66* stehen Bespielhaft für alle Module des Funktionsmodelles.

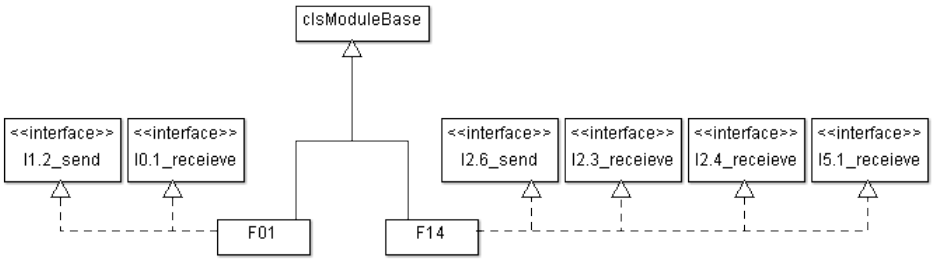

Abbildung 11: Klassendiagramm Funktionsmodule [Deu11 S. 130]

Die Kommunikation der Funktionsmodule untereinander erfolgt durch die im Funktionsmodell (siehe Anhang A) definierten Interfaces. In Abbildung 11 ist diese Struktur anhand von zwei Funktionsmodulen zu sehen. Wie diese Interfaces aufgerufen werden ist in Abbildung 12 zu sehen. Die Klassen *clsProcessor* ruft die *step()* Methode der jeweiligen Funktionsmodule der Reihe nach auf. In dieser ist die Funktion des Modules beschrieben. Am Ende dieser Methode werden die erarbeiteten Daten an das Interface des nächsten Funktionsmodules weitergegeben. Wird dieses ausgeführt, müssen bereits alle nötigen Inputdaten verfügbar sein.

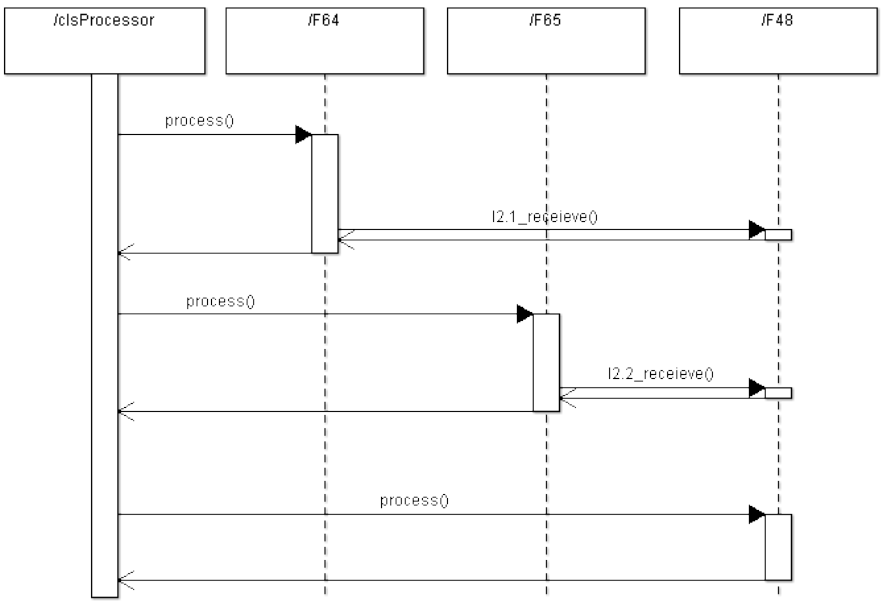

Abbildung 12: Funktionsmodulaufruf

2.2 Simulationsumgebung

Die Simulationsumgebung in der die ARS Decision Unit entwickelt wird, setzt sich aus zwei Teilen zusammen. Der Controller zur Steuerung der Simulation, sowie die Physics-Engine werden als Bibliotheken vom MASON Projekt [LCRP+05] eingebunden. Die eigentliche Artificial Life Simulation wurde gemeinsam mit der Decision Unit vom Team des ARS-Projektes entwickelt. In den nächsten Kapitel wird zuerst allgemein das Simulationsframework MASON und anschließend die Erweiterungen des ARS-Projektes beschrieben.

2.2.1 Mason Simulationsframework

Mason [3] ist ein gut skalierbares, ereignisbarsiertes Multi-Agenten Simulationstoolkit mit dessen Hilfe eine breite Palette an Multi-Agenten Simulationsszenarien abgedeckt werden sollen. Es ist ein Single Prozess System, wurde komplett in Java entwickelt und steht als Open Source Bibliothek zur Verfügung.

Das MASON Toolkit wurde auf Grund seiner guten Scheduling und Phsyics-Engine Performance und wegen seiner einfachen Erweiterbarkeit für das ARS-Projekt ausgewählt. Eine detaillierte Beschreibung des Auswahlverfahrens findet sich in [Lan10 S. 36].

Das Toolkit ist aus drei Schichten aufgebaut:

- Utility Layer
- Model Layer
- Visualisierung Layer

Utility Layer

Der Utility Layer besteht aus Klassen, die allgemeine Tools beinhalten. Wie zum Beispiel einen Zufallszahlen Generator, Datenzugriffswerkzeuge und Methoden zur Bilder oder Video Erzeugung.

Model Layer

Der Model Layer ist komplett unabhängig vom Visualisierung Layer. Dadurch ist es möglich die Simulation ohne, oder mit einer eigenen Visualisierung auszuführen. Der Aufbau der beiden Layer ist in Abbildung 13 zu sehen. Die Klasse *SimState* kapselt die gesamte Simulation. Sie besteht aus einem *Field Objekt* und einem *Scheduler*. Jeder Agent der während der Simulation ausgeführt werden soll, muss sich beim *Scheduler* registrieren. Dazu muss der Agent das Interface *Steppable* implementieren, und die jeweiligen Simulationsschritte angeben in denen er ausgeführt werden möchte. Die Ausführung kann auch in mehrere Teile unterteilt werden, die unabhängig voneinander ausgeführt werden können. Während der Ausführung des Agenten kann dieser ohne Einschränkungen auf die *SimState* Klasse und somit auch auf das *Field* zugreifen und diese beeinflussen. Die Kommunikation zwischen den Agent wird nicht definiert. Über die *SimState* Klasse kann jedoch auf die anderen Agenten zugegriffen werden. Es ist nicht unbedingt erforderlich, aber wenn sich der Agent physikalisch in der Simulationsumgebung befinden soll, dann muss er ebenfalls im *Field Objekt* regis-

triert werden. Das *Field Objekt* repräsentiert die Simulationsumgebung, und wird je nach Anforderungen an die Simulation in Subfelder unterteilt. Es können entweder die vordefinierten 2D und 3D Arrays oder auch eigene *Field* Implementierungen verwendet werden. Die vorgegeben 2D Implementierung erlaubt Quadrate, Dreiecke oder Sechsecke als Unterteilungsstrukturen. Da ein Agent jedoch nicht nur auf ein Sub-Feld beschränkt ist sind je nach Skalierung beliebige Formen der Agenten denkbar.

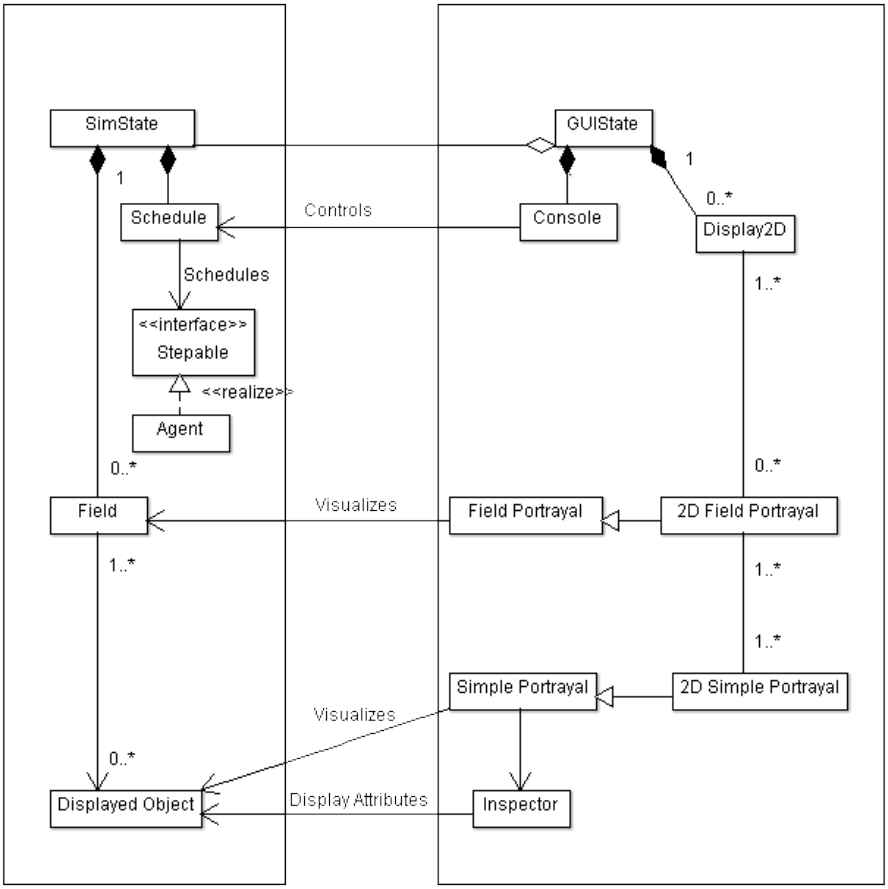

Abbildung 13: Aufbau MASON [LCRP+05 S. 4]

Visualisierung Layer

Der Visualisierung Layer beinhaltet die Visuelle Darstellung der Daten des Modelles. Da dieser unabhängig von der Simulation ist, kann die Visualisierung durch andere Visulisierungen ersetzt werden, ohne die Simulation zu beeinflussen. Weiter beinhaltet der Visualisierung Layer ein GUI zur Steuerung der Simulation (Abbildung 14).

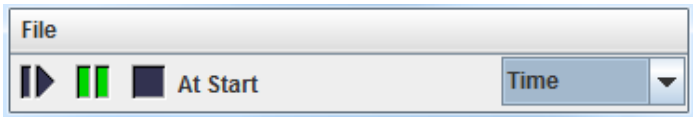

Abbildung 14: Mason Konsole

Ähnlich dem *SimState* im Model Layer gibt es im Visualisierung Layer die Klasse *GUIState*. Diese kapselt die gesamte Visualisierung, bestehende aus der Steuerkonsole und einem oder mehreren Displays. Ein Display hält ein oder mehrere Field *Portroyals* die mit dem *Field* im Model Layer verbunden sind, und dessen Visualisierung darstellen. Das *Field Portroyal* hält ein oder mehrere *SimplePortroyals* die mit den Sub-Felder im Model Layer verbunden sind und diese visualisieren. Wie in Abbildung 13 zu sehen ist, hat jede Komponente des Visualisierung Layers ein Objekt des Model Layerszugeordnet. Die Standard Implementierung des *FieldPortroyal* ermöglicht es, Inspektoren zu starten, die eine Referenz auf das zu untersuchende Objekt aus dem Model Layer erhalten. Dabei besteht die Möglichkeit sich auf die einfachen Standardinspektoren zu beschränken oder eigene Implementierungen zu verwenden.

Simulationszyklus

Jedes Simulationsobjekt wird vom Scheduler der Simulation als Softwareagent betrachtet. Da die Ausführungssteuerung der Agenten jedoch bei der Simulationsumgebung liegt muss jedes Simulationsobjekt im Scheduler registriert werden. Durch die Registrierung wird sowohl ein Eintrag im Model als auch im Visualisierungs-Layer erstellt. Die Simulationsumgebung sorgt für die Synchronisation dieser beiden Einträge. Nach der Registrierung der Simulationsobjekte kann mit der Ausführung der Simulation begonnen werden. Diese wird in Simulationszyklen unterteilt. In jedem Zyklus wird jedes Simulationsobjekt genau einmal ausgeführt. Abbildung 15 zeigt die Registrierung und den ersten Simulationszyklus. Nach der Ausführung der Simulationsobjekte wird die Visualisierung aktualisiert. Die Visualisierungsobjekte können über ihr Gegenstück im Model Layer auf die Simulationsobjekte zugreifen und so die benötigten Visualisierungsinformationen abholen. Nach demselben Prinzip arbeiten die Inspektoren, mit denen Objektinterne Daten dargestellt werden können. Nach der Abarbeitung aller Simulationsobjekte, sowie der Visualisierung und der Inspektoren wird der Simulationszykluszähler erhöht und die Ausführung der Simulationsobjekte beginnt von neuem. Mit Hilfe der in Abbildung 14 dargestellten Steuerkonsole kann der Simulationsablauf vom Benutzer gesteuert werden.

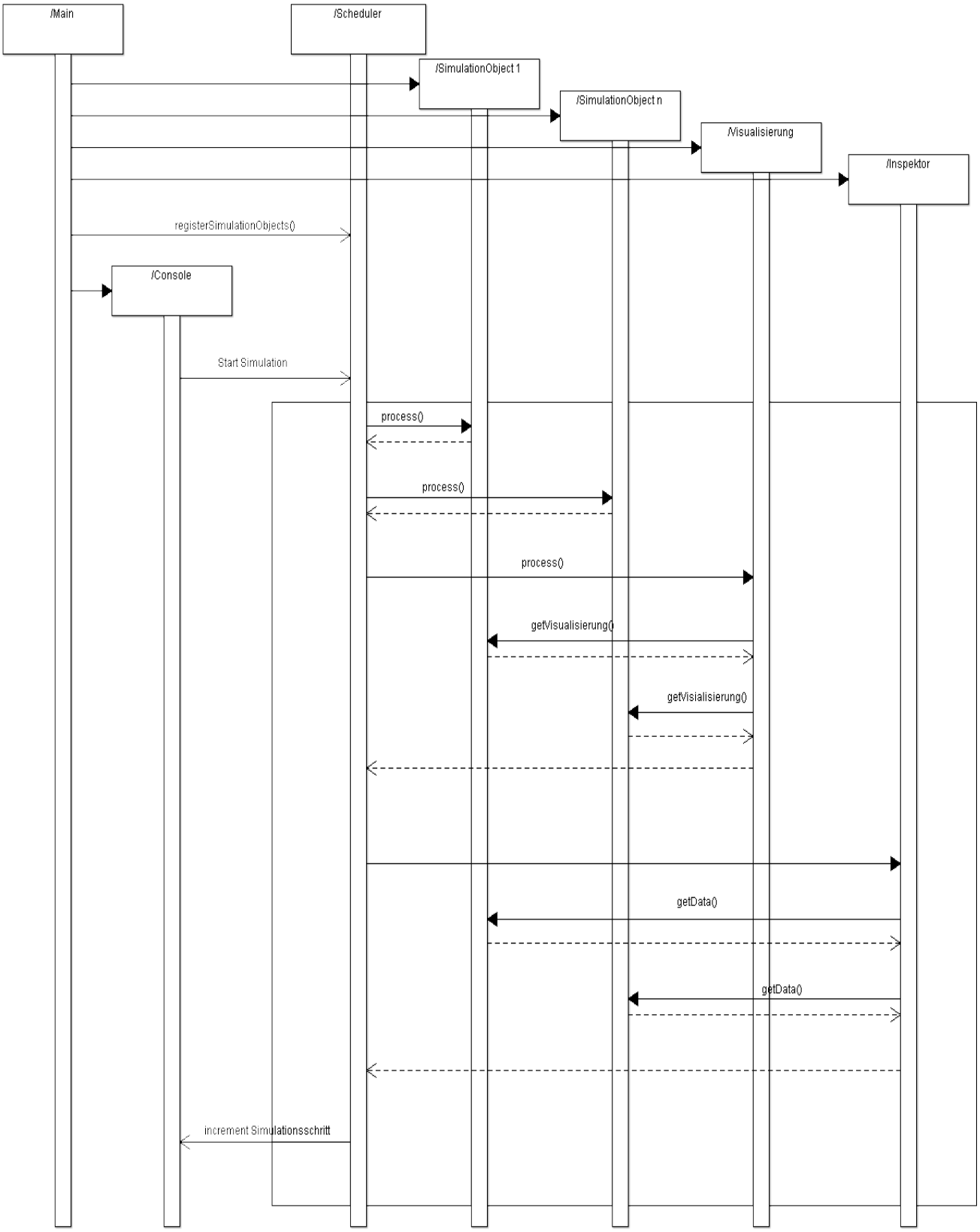

Abbildung 15: Simulationsablauf MASON

2.2.2 ARS-Simulationsarchitektur

Das MASON Simulationsframework stellt ganz allgemeine Funktionen zur Simulation zur Verfügung. Um das ARS-Modell und die dazugehörenden Use Cases simulieren zu können wird eine Artificial Life Simulation benötigt. Deshalb wurde aufbauend auf dem MASON Framework einer ARS spezifische Simulationsumgebung entwickelt. Die Entwicklung der Simulationsumgebung erfolgt dabei abgestimmt auf die Anforderungen die das Funktionsmodell und die Use Cases an die simulierte Welt haben.

Aufbau

Der funktionale Aufbau und die Ankopplung an MASON wird in Abbildung 16 dargestellt. Die Komponenten die für die Steuerung und Ausführung der Simulation verantwortlich sind, befinden sich alle im MASON Funktionsblock. An diesen angekoppelt, sind die im ARS-Projekt entwickelten Entities, die die Simulationsobjekte und Agenten darstellen. In der Abbildung ist nur eine Entity dargestellt. In der Simulation existieren die Simulationsobjekte parallel zueinander, und werden alle nach dem gezeigten Schema an die Simulationsumgebung angebunden. Die dargestellten Komponenten werden in Tabelle 2 im Detail beschrieben.

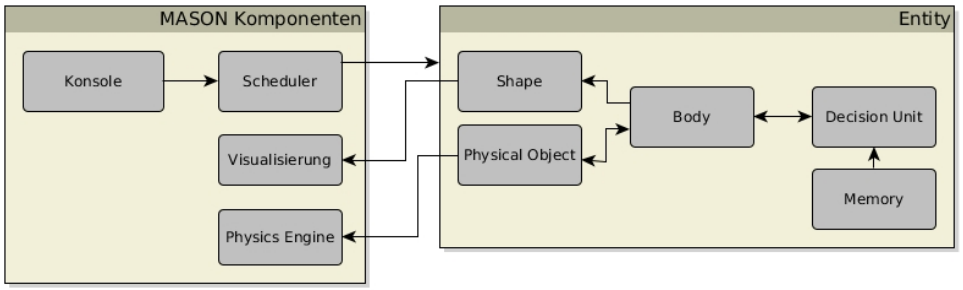

Abbildung 16: Komponentendiagram Simulationsumgebung

Mason Komponenten	Der Mason Block beschreibt das als Library eingebundene Simulationsframework. Dieses beinhalten wie in Kapitel 2.2.1 beschrieben eine *PhysicsEngine*, eine *Visualisierung*, den *Event Scheduler* und ein Steuerkonsole.
Entity	Jedes Simulationsobjekt ist durch eine *Entity* repräsentiert. Diese wird beim Scheduler registriert und in jedem Simulationsschritt aufgerufen. Die Entity besteht aus mehreren Submodulen und übernimmt die Ausführungsteuerung dieser.
Physical-Object	Das *Pysical-Object* ist Teil der *Entity* und liefert der *Physics-Engine* die nötigen Informationen für die physikalischen Berechnungen.
Shape	Das Shape Objekt ist ebenfalls Teil der *Entity* und liefert der Visualisierung Informationen über die graphische Darstellung der Entity.

Body	Im *Body* ist die eigentliche Funktionalität des Objektes gekapselt. Eine Implementierung des *Bodies* wäre der *Meat Body*. Er kommt zum Beispiel bei der Schnitzel *Entity* zum Einsatz und simuliert das essbare Fleisch des Schnitzels. Eine weitere Body Implementierung ist der *ComplexBody*. Er modelliert den Körper der aktiven Agenten und stellt ein Interface zur Decision Unit bereit. Da das Modell des Körpers nicht im Fokus des ARS-Projektes steht, wurde nicht versucht ein funktionales Modell zu entwickeln. Der Körper dient nur dazu die Inputs zu liefern, die die Decision Unit zum aktuellen Stand benötigt. Er stellt kein funktionales Modell des menschlichen Körpers dar. In Abbildung 17 ist der Aufbau des *ComplexBody* zu sehen. Dieser besteht im Wesentlichen aus vier Systemen. Das *InternalSystem* repräsentiert körperinterne Funktionen wie z.B.: Das Verdauungssystem. Das Sensor System generiert Daten aus dem *InternelSystem* (Messanger System) und aus der simulierten Welt (Externe Sensoren), und gibt diese an die Schnittstelle zur Decision Unit (*BrainSocket*) weiter. Die Aktionsbefehle der Decision Unit werden über das *BrainSocket* an die Action Engine weitergegeben. Diese wirkt sowohl auf körperinterne Systeme, als auch auf die simulierte Welt ein.
Decision Unit	In der Decision Unit wird bestimmt welche Handlungen die *Entity* ausführen soll. Eine Implementierung wäre die psychoanalytische Decision Unit des ARS-Projektes. Es besteht aber auch die Möglichkeit, andere Implementierungen zu verwenden.

Tabelle 2: Simulationskomponenten

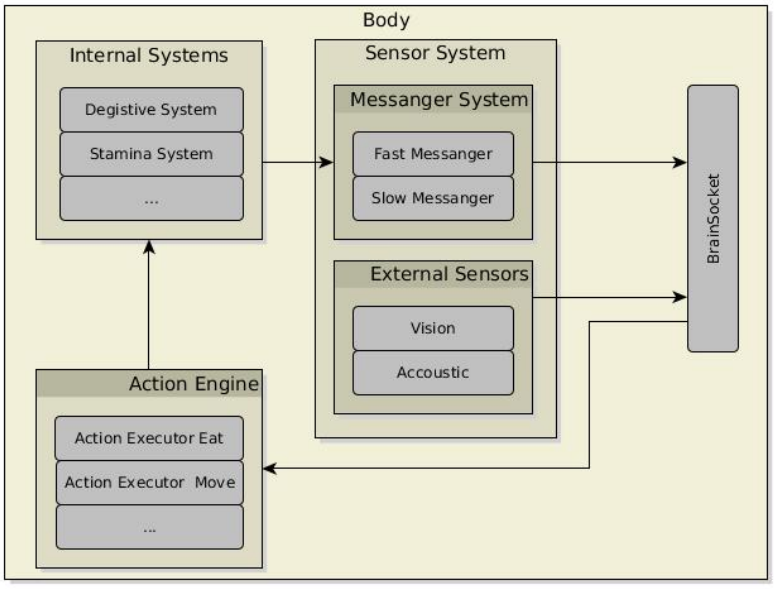

Abbildung 17: Aufbau ComplexBody

Simulationszyklus

Der MASON Simulationszyklus (Abbildung 15) wurde, um ihn an die Anforderungen einer Artificial Life Simulation anzupassen, um die in Tabelle 3 beschriebenen Subzyklen erweitert. Die Ausführung der einzelnen *Entities* wird in Teilschritte aufgeteilt. Dadurch wird erreicht, dass unabhängig von der Ausführungsreihenfolge alle *Entities* quasi parallel ausgeführt werden. Der erweiterte Simulationszyklus ist in Abbildung 18 dargestellt.

beforeSensing	Vorbereitung der Daten innerhalb des Körpers
Sensing	Die Informationen werden von den Sensoren gesammelt und zusammengeführt.
updateInternalState	Vorverarbeitung der Sensorinformationen
Processing	Ausführung der Decision Unit
Execution	Ausführung von physischen Aktionen innerhalb der Simulationsumgebung
afterStepping	Aktionen des Körpers, die aus dem Execution Schritt resultieren, werden ausgeführt

Tabelle 3: Simulationszyklen

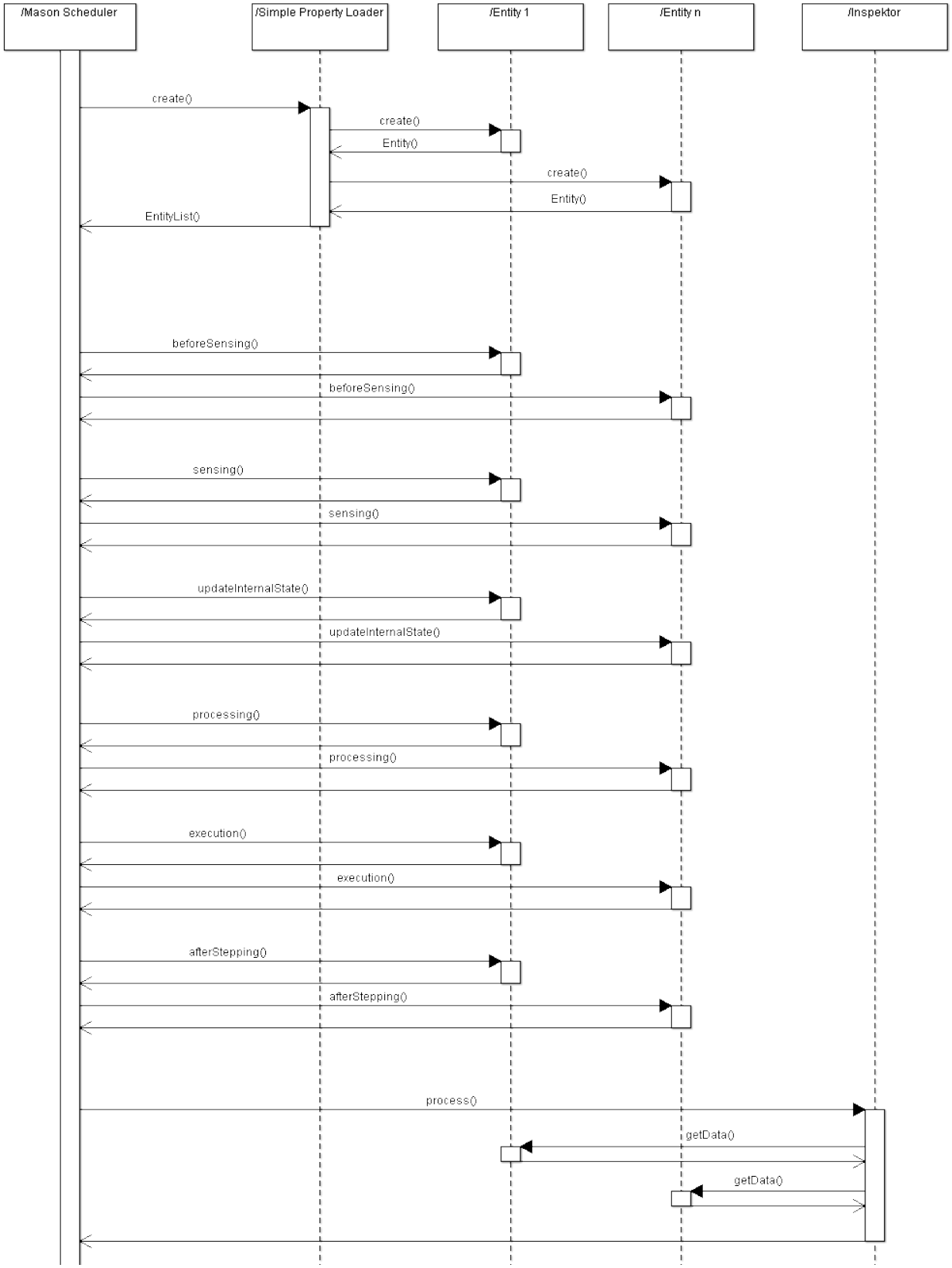

Abbildung 18: Erweiterter Simulationszyklus [Deu11 S. 118]

2.3 Miklas

Die Simulationsumgebung Miklas [4] wurde erstellt, um die ARS Decision Unit alternativ zur MASON Simulationsumgebung in einer möglichst einfachen, und hochgradig konfigurierbaren Umgebung ausführen zu können. Aus diesem Grund wurde die Miklas Simulationsumgebung als alternative Umgebung ausgewählt, um die Schnittstelle zu testen. Als Vorlage für Miklas diente das Computerspiel „Pacman“. Deshalb basiert Miklas ebenso auf der *JGameGrid* Game Engine. Aufbauend auf *JGameGrid* wurde eine hochgradig konfigurierbare Simulationsumgebung entwickelt, die es erlaubt mit wenig Programmieraufwand eine Welt mit verschiedenen Agent und Interaktionsmöglichkeiten zu erstellen. Die Entscheidungseinheiten dieser Agenten können mit beliebigen kognitiven Architekturen bestückt werden, um diese innerhalb der Simulationsumgebung auszuführen.

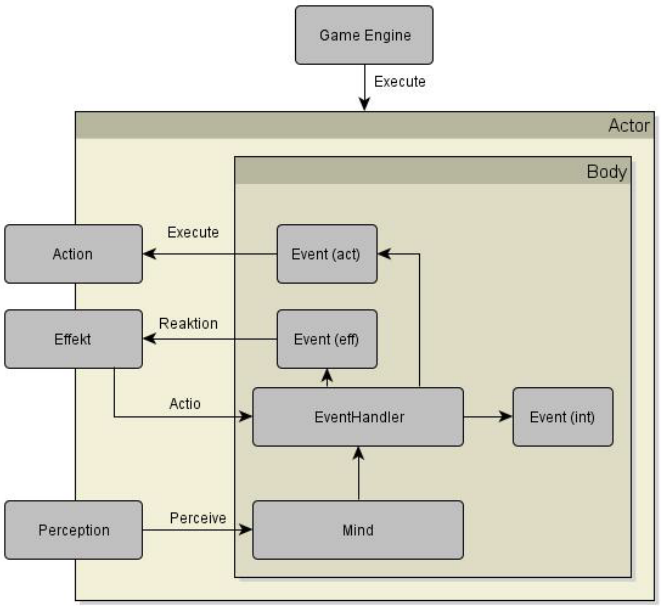

Abbildung 19: Miklas Aufbau

Sowohl aktive als auch passive Objekte innerhalb der Simulation werden als Agenten gezeichnet. Der Aufbau dieser Agenten ist in Abbildung 19 zu sehen. Die zur *JGameGrid* gehörende Game Engine übernimmt dabei die Ausführungskontrolle über die einzelnen Agenten. Jeder Agent besteht aus drei Teilen, einem Body, einem Mind und einem dreiteiligen Interface, um mit der Umwelt und mit anderen Agenten interagieren zu können. Der Body des Agenten wird durch Events bestimmt. Zu jedem dieser Events ist ein Set an Bedingungen definiert, die erfüllt sein müssen, damit dieses Event ausgeführt wird. Es wird zwischen drei Typen von Events unterschieden. Event (act) bezeichnet Events die vom Mind ausgelöst werden, und in einer Aktionsausführung des Agenten münden. Event (int) beziehen sich auf interne Zustände des Body und wirken sich auf diesen und dessen Zustandsvariablen aus. Die Aufgabe von Event (eff) ist es, Aktionen die mit dem Agenten ausgeführt werden, entgegen zu nehmen und eine Reaktion darauf zu erzeugen. Zur Interaktion mit der Umwelt und anderen Agent verfügt jeder Agent über ein dreiteiliges Interface, bestehend aus einer Action, einer Effect und einer Perception Komponente. Die Effekt Komponente ermöglicht es anderen Si-

mulationsteilnehmern, über Event (eff) Einfluss auf den Body des Agenten zu nehmen. Über die Action Komponente kann der Agent externe Aktionen innerhalb der Welt absetzten, und somit diese beeinflussen. Die Perception Komponente des Interfaces ermöglich es dem Mind seine Umwelt innerhalb eines Radius wahrzunehmen.

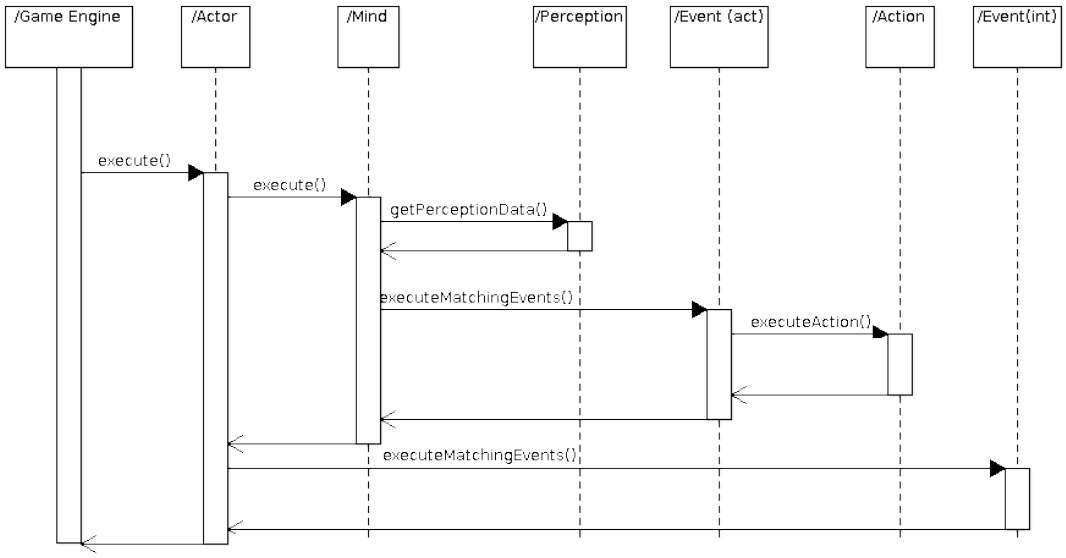

Abbildung 20: Simulationsablauf Miklas

Abbildung 20 zeigt die Ablaufsteuerung der Agentenausführung. Die Game Engine übernimmt dabei die Kontrolle und ruft der Reihe nach die registrierten Agenten auf. Umgebungs- und Körperinternedaten werden vom Mind aus dem Perception Interface angefordert, um innerhalb der Entscheidungsfindung zu einer Aktion zu kommen. Nach der Ausführung der Mind Komponente wird der interne Zustand des Agenten aktualisiert.

Die Konfiguration der Miklas Simulationsumgebung erfolgt, wie bereits oben erwähnt, über Konfigurationsfiles. Durch das hohe Maß an Konfigurierbarkeit ist es mit wenig Programmieraufwand möglich, die verschiedensten Szenarien zu schaffen. In Abbildung 21 ist beispielhaft die Konfiguration des Actors *HUMAN* mit dem Bodytype *HUMANPLAYER* und dem Mind *HUMAN* zu sehen. Die Konfiguration erlaubt es Aktoren mit beliebigen Body und Mind Kombination zu erstellen. Da der Body durch seine Events definiert wird, und diese ebenfalls im Konfigurationsfile festgelegt wird, ist die Funktionalität des Bodies über das Konfigurationsfile einstellbar. In den letzten vier Zeilen der Abbildung 21 ist zu sehen, wie Effekt, Aktion und Interne Events für den Body definiert werden können.

```
#Minds
    mind.0.mindname=HUMAN
#Actor
    actor.0.actorname=Human
    actor.0.bodytypename=HUMANPLAYER
    actor.0.worldmapchar=m
#BodyTypes
    bodytype.0.typename=HUMANPLAYER
    bodytype.0.bodytype=ANIMATEBODY
    bodytype.0.mind=HUMAN
    bodytype.0.effectonaction=BEINGATTACKED
    bodytype.0.effectonreaction=BUMPOBSTACLE,EATGOODOBJECT,EATBADOBJECT,EATFLOOR,ATTACKPLAYER
    bodytype.0.possibleaction=ACTIONEAT,ACTIONMOVEFORWARD,ACTIONTURNLEFT,ACTIONTURNRIGHT,ACTIONATTACK
    bodytype.0.bodyinternalevents=DEATH,SETHEALTH
```

Abbildung 21: Beispielhafte Konfiguration

2.4 Analyse und Vergleich früherer Versuche

Innerhalb des ARS-Projektes gab es bereits Bestrebungen die Decision Unit in externen Umgebungen einzusetzen. Leider ist aus diesem Bestrebungen nie eine allgemein Schnittstelle zur Decision Unit resultiert. In diesem Kapitel werden diese Arbeiten kurz vorgestellt und aufgezeigt warum es zu keiner Allgemeinen Schnittstelle kommen konnte.

2.4.1 Anbindung an Unreal

In den Arbeiten [Fer12] und [Tor12] wird eine mögliche Anbindung der ARS Decision Unit an die Unreal Simulationsumgebung beschrieben. Zur Realisierung dieser Anbindung wurde auf das Pogamut Framework [GBP11] zurückgegriffen (siehe Abbildung 22).

Abbildung 22: Middleware Pogamut

Das Pogamut Framework realisiert eine asynchrone Kopplung zwischen der Simulationsumgebung (Unreal) und der Decision Unit. Wobei hierbei Pogamut und die Decision Unit synchron und Unreal mit Pogamut asynchron gekoppelt sind. Daraus ergibt sich, dass die Schnittstelle über Pogamut nicht innerhalb der Simulationsumgebung des ARS-Projektes sinnvoll einsetzbar ist. Es wurde also keine einheitliche Schnittstelle, sondern eine zweite parallele Anbindung geschaffen [Fer12 S. 80]. Bei späteren Weiterentwicklungen der ARS Decision Unit wurden die Änderungen jedoch nicht auf beiden Schnittstellen, sondern nur auf die Verbindung zur ARS-Simulationsumgebung angewandt. Daraus ergibt sich, dass die Schnittstelle zu Unreal nicht mehr funktionsfähig ist.

Diese Zweigleisigkeit lässt sich auch innerhalb der Decision Unit erkennen. Hier wurden Anpassungen innerhalb der Funktionsmodule vorgenommen, um den Anforderungen der neuen Umgebung gerecht zu werden [Fer12 S. 53]. Auch dabei ist zu beobachten, dass bei Funktionsänderungen nur die aktuell relevanten Funktionen und nicht die Unreal spezifischen Funktionen gewartet werden.

Ein weiteres Problem stellt die Trennung zwischen Simulationsumgebung und Decision Unit dar. Es wurde keine klare Trennung dieser beiden Funktionseinheiten vorgenommen. Die Action-Engine der ARS-Simulationsumgebung konnte nicht von der Decision Unit getrennt werden. Diese musste also auch bei der Anbindung an Unreal ausgeführt werden [Tor12 S. 73]. Durch die konsequente Aufteilung der Funktionsblöcke in unterschiedliche Projekte können ungewollte Abhängigkeiten verhindert werden.

Die besprochenen Arbeiten sind jedoch auf Grund dieser Kritikpunkte nicht gescheitert. Der Fokus bestand darin eine Anbindung zu schaffen. Dieses Ziel wurde erreicht. Nachhaltiger wäre es jedoch gewesen den Schwerpunkt der Arbeit nicht auf die Anbindung an das externe System zu legen, sondern eine allgemein gültige Schnittstelle zu Decision Unit zu schaffen.

2.4.2 Greenhaus

In der Arbeit [Ber13] wurde versucht die ARS Decision Unit zur Steuerung eines Glashauses einzusetzen. Die Arbeit unterteilt sich dabei in zwei Bereiche. Im ersten Teil wird das ARS-Projekt allgemein betrachtet, und versucht ein Modell zu entwickeln wie die ARS Decision Unit in anderen Umgebungen eingesetzt werden kann. Der zweite Teil beschäftigt sich damit, aufbauend auf Teil 1 die Konzepte des ARS-Projektes auf die Steuerung des Glashauses zu mappen und die ARS-Implementierung zu diesem Zweck einzusetzen. Für diese Arbeit ist lediglich der erste Teil relevant. Der Ansatz dieser Arbeit ist, innerhalb der ARS-Simulationsumgebung die einzelnen Funktionsblöcke zu identifizieren und zu evaluieren, welche der Funktionsblöcke als Decision Unit angesehen werden sollen [Ber13 S. 21]. Anschließend werden diese Funktionsblöcke zu einer Blackbox zusammengefasst, um diese in alternativen Umgebungen einsetzen zu können.

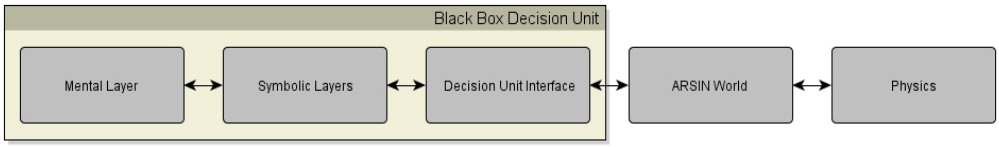

Abbildung 23: Blackbox Model [Ber13 S. 36]

Abbildung 23 zeigt dabei die identifizierten Komponenten, sowie die in der Arbeit definierte Blackbox. Dieses Modell der Blackbox wurde dabei zwar in den Glashaus Implementierung verwendet, in der ARS-Simulationsumgebung wurden jedoch keine Veränderungen vorgenommen [Ber13 S. 65]. Im Zuge der Arbeit wurde der Zugriff auf die Implementierung des Funktionsmodelles nicht betrachtet.

Das Hauptaugenmerk wurde auf die Identifikation der Grenze zwischen Simulation (ARS Decision Unit) und Applikation gelegt und diese wurde auch definiert. Die Kopplung zwischen Simulationsumgebung und Decision Unit blieb also in der ARS-Simulationsumgebung erhalten.

2.5 Schnittstellen anderen kognitiven Architekturen

Parallel zur im ARS-Projekt entwickelten, Decision Unit gibt es andere kognitive Architekturen [JW06] die ebenfalls in Artificial Life Simulationen eingesetzt werden. Die Schnittstellen zwischen diesen Architekturen und deren Simulationsumgebungen haben ähnliche Anforderungen, und können daher als Vorbild für die Schnittstelle zur ARS Decision Unit dienen.

2.5.1 SOAR General Input/Output

Die SGIO (SOAR General Input/Output) Schnittstelle [LAB+00] wurde entwickelt um die Kognitive Architektur SOAR [LNR87] mit Spieleumgebungen, wie zum Beispiel der Unreal Engine zu verbinden. Die SGIO Schnittstelle unterscheiden dabei die Ausführung in unterschiedlichen Prozessen und die Ausführung im selben Prozess.

Ausführung in unterschiedlichen Prozessen

Die Spieleumgebung und das SOAR System werden in unterschiedlichen Prozessen ausgeführt. Diese Prozesse können auf demselben Rechner, aber auch auf beliebigen Rechnern im Netzwerk ausgeführt werden. Das hat den Vorteil, da zum einen die Ausführung der Spieleumgebung nicht durch das SOAR Systems gebremst, und zum anderen, die Rechenleistung auf mehrere Rechner / Prozessoren aufgeteilt werden kann. Ein Nachteil ergibt sich aus dem Aufwand die beiden Prozesse zu synchronisieren und aus dem benötigten Kommunikationsoverhead. Abbildung 24 zeigt den Aufbau der Schnittstelle. Die SGIO Schnittstelle wird in zwei Teile unterteilt. Der eine Teil wird als DLL implementiert und kann an die Spieleumgebung angehängt werden um mit dieser zu kommunizieren. Der zweite Teil ist an das SOAR System gekoppelt. Die beiden Teile dienen als Brücke zwischen Spielumgebung und SOAR System und kommunizieren über eine Socket Verbindung.

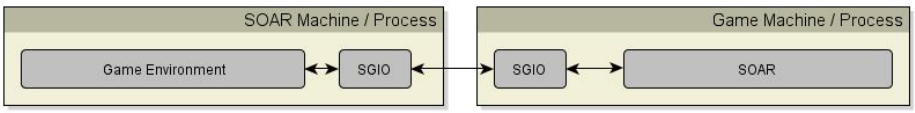

Abbildung 24: SGIO Konfiguration 1 [LAB+00 S. 4]

Ausführung im selben Prozess

Die zweite Konfigurationsmöglichkeit beschreibt die Ausführung der Spieleumgebung und des SOAR System im selben Prozess. SOAR ist komplett in die Spielumgebung integriert und von dieser abgängig. Der Vorteil dieser Konfiguration ist es, dass kein Synchronisations- bzw. Kommunikationsoverhead auftritt. Abbildung 25 zeigt den Aufbau der Konfiguration. Die SGIO wird dabei wird in Konfiguration 1 als DLL an die Spieleumgebung angehängt. Hier wird jedoch nicht mit einem Socket kommuniziert, sondern direkt die Funktion des SOAR Systems aufgerufen.

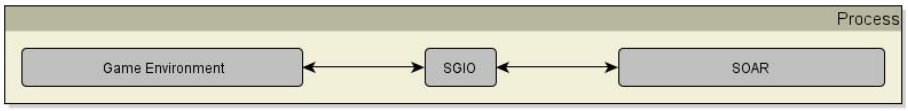

Abbildung 25: SGIO Konfiguration 2 [LAB+00 S. 4]

2.5.2 BDI Body-Mind Interface

Die in [TNB03] vorgestellte Schnittstelle verbindet die kognitive Architektur BDI [RG+95] mit autonomen Agenten in Computerspielen. Die Spielumgebung beinhaltet die Logik für den Körper des Agenten und die BDI Implementierung stellt die Decision Unit dar. Dabei wird eine synchrone Kommunikation zwischen Spielumgebung und Decision Unit beschrieben, die Ausführung der Spielumgebung wartet also bis die Decision Units ihre Berechnungen beendet haben. Die Kommunikation unterteilt sich dabei in vier Schritte, und ist in Abbildung 26 zu sehen. Die Agenten werden der Reihe nach ausgeführt. Im ersten Schritt der Kommunikation werden die Wahrnehmungsinformationen des Körpers an das Interface gesendet. Dieser wandelt die Informationen in eine, für die Decision Unit verständliche Symbolik um, und weist sie der richtigen Decision Unit zu. Nachdem Entscheidungsprozess innerhalb der Decision Unit wird ein Aktionsbefehl an das Interface übergeben. Aus diesem wird, im letzten Schritt innerhalb des Interfaces, das Verhalten des Agenten generiert. Da in diesem Schnittstellenmodell alle Körper und Decision Units über eine Schnittstelle kommunizieren, ist es notwendig, einen Identifizierungsmechanismus zu verwenden, um während der Kommunikation den Körper mit der richtigen Decision Unit zu verbinden.

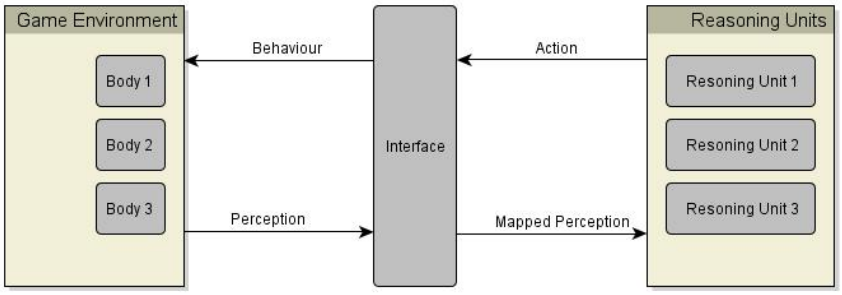

Abbildung 26: Body-Mind Interface BDI [TNB03 S. 3]

2.6 Software Design Patterns

Software Design Pattern [Cop98] stellen in der Informatik eine bewährte Methode dar um an Problemstellungen heranzugehen. Dazu werden wiederkehrende ähnliche Problemstellungen zusammengefasst, und als allgemeine Problemstellung definiert. Zur Lösung gibt es bereits bewährte und beschriebene Lösungsstrategien. Um diese anwenden zu können ist es nötig, das konkrete Problem zu verallgemeinern, um es mit den beschriebenen Problemstellungen vergleichen zu können. Wenn eine Übereinstimmung gefunden wurde, kann die beschriebene Lösungsstrategie auf die konkrete Prob-

lemstellung angewandt werden. Für die aktuelle Problemstellung ist es wichtig einen Blick auf vorhandene Software Design Pattern zu nehmen, da es sich dabei um erprobte Lösungsansätze handelt, und somit sowohl der Designaufwand verringert, als auch die Qualität der Lösung verbessert werden kann. Mit einer höheren Qualität ist in dem Fall vor allem die Wartbarkeit und Wiederverwendbarkeit des Ergebnisses gemeint. Dies ist im Projekt ARS besonders wichtig, da die Entwickler innerhalb des Teams sehr oft wechseln und so eine Einarbeitung erleichtert werden kann.

2.6.1 Factory Methode Pattern

Das Factory Methode Pattern [HJVG02 S. 107 - 116] gehört zur Familie der Erzeugermuster. Diese beschäftigen sich mit dem Erzeugen von Objekten mit Möglichkeiten die, die eines Konstruktors übersteigen.

Problemstellung

Die vom Factory Methode abgedeckte Problemstellung kann ganz allgemein als dynamisches Erstellen von Objekten bezeichnet werden. Objekte sollen im Programmcode nicht statisch durch Konstruktor Aufrufe, sondern zur Laufzeit konfigurierbar erzeugt werden.

Lösungsansatz

Das Factory Methode Pattern beschäftigt sich ganz allgemein mit dem dynamischen Erstellen von Objekten. Dabei wird ein Factory Klasse verwendet, die mit Hilfe einer create() Methode die Schnittstelle zur Objekterzeugung zur Verfügung stellt.

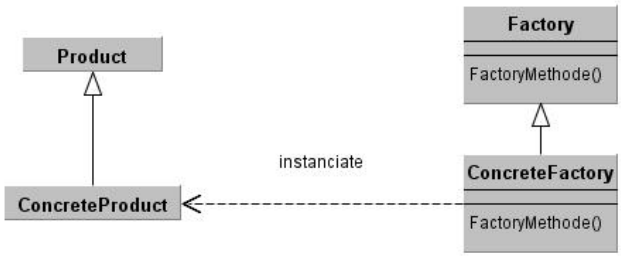

Abbildung 27: Factory Methode Pattern [HJVG02 S. 107]

Abbildung 27 zeigt den strukturellen Aufbau des Patterns. Die Klasse Factory stellt mit *FactoryMethode()* die Erzeuger Schnittstelle zur Verfügung. Diese kann zur Laufzeit mit konkreten Factory Klassen gefüllt werden, um dynamisch das passende Objekt zu erzeugen.

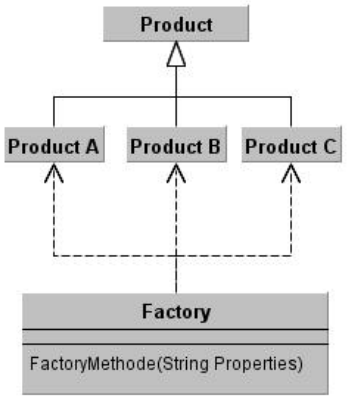

Abbildung 28: Factory Methode Pattern 2 [HJVG02 S. 110]

Ein weitere Anwendungsfall des Factory Methode Patterns wird in Abbildung 28 beschrieben. Mit Hilfe einer Factory Methode ist es, gesteuert durch den Konfiguratuionsstring Properties, möglich Objekte der Klassen Produkt A, Produkt B und Produkt C zu erzeugen.

2.6.2 Adapter Pattern

Das Adapter Pattern [HJVG02 S. 139 - 150] ist ein Software Design Pattern und gehört zur Familie der Strukturmuster. Die Design-Pattern dieser Kategorie stellen Muster für das In-Beziehung-Setzten von Software Komponenten dar. Das Adapter Pattern im Speziellen definiert eine Lösungsmöglichkeit um zwei unterschiedliche Schnittstellen zusammenzuführen.

Problemstellung

Ein Dienst bietet ein Service unter einer klar definierten Schnittstelle (S1) an. Ein Client möchte über eine von ihm definierte Schnittstelle (S2) auf dieses Service zugreifen. Die Schnittstellen S1 und S2 sind untereinander nicht kompatibel. In diesem Fall müsste man eine der beiden Schnittstellen dahingehend anpassen, dass diese mit der anderen Schnittstelle kompatibel ist. Oft ist eine Anpassung der Schnittstellen nicht möglich. Da zum Beispiel sowohl Client, als auch Server bereits in kompilierter Form existieren, und nicht mehr verändert werden können. Ein weiter Fall wäre es, dass das Anpassen einer Schnittstelle zwar möglich ist, es jedoch aus designtechnischen Gründen verhindert werden soll. Ein Beispiel: Ein Client und Server werden in unterschiedlichen Kombinationen mit unterschiedlichen Schnittstellen eingesetzt, aber nicht für jede Kombination wird eine eigene Schnittstelle definiert.

Lösungsansatz

Der Lösungsansatz des Adapter Patterns, beschreibt eine Adapterklasse, um die Schnittstelle des Dienstes in die des Clients überzuführen. Dafür werden im Pattern zwei mögliche Ansätze vorgeschlagen.

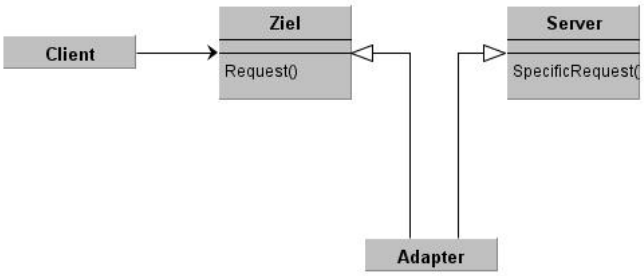

Abbildung 29: Klassen Adapter [HJVG02 S. 141]

Der in Abbildung 29 zu sehende Klassen Adapter realisiert die Schnittstellenanpassung durch Mehrfachvererbung. Dabei erbt die Adapterklasse sowohl von der Ziel, als auch von der Server Schnittstelle. Eine Implementierung dieser Art des Adapter Patterns ist leider in JAVA nur sehr bedingt möglich, da keine Mehrfachvererbung unterstützt wird.

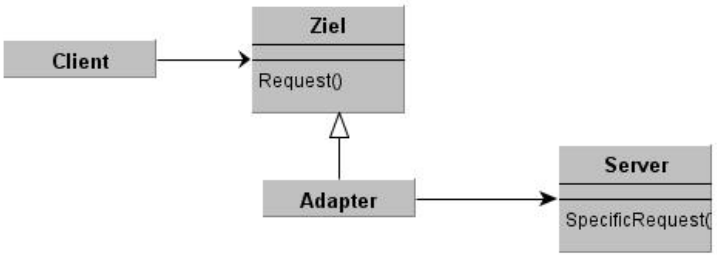

Abbildung 30: Objekt Adapter [HJVG02 S. 141]

Die zweite Variante des Adapter Patterns ist in Abbildung 30 zu sehen und stellt einen Objekt Adapter dar. Der Adapter erbt von der Ziel-Schnittstelle und kann dadurch an dessen Stelle verwendet werden. Außerdem hält er eine Referenz auf das Server Objekt. In den Implementierungen der ererbten Methoden wird auf die vom Server zur Verfügung gestellte Schnittstelle zugegriffen. Diese Adaptierung kann von der reinen Überführung in andere Namensgebungen, bis hin zur Konvertierung der Daten oder der Aufrufungsstruktur gehen.

In weiterer Folge wird nur mehr auf das Objekt Adapter Pattern Bezug genommen, da das Klassen Adapter Pattern für JAVA wenig geeignet, und aus diesem Grund für unsere Implementierung nicht relevant ist.

Vorteile

Folgende Vorteile ergeben sich durch die Verwendung des Adapter Patterns

- Ein Server kann über Adapterklassen mit unterschiedlichen Clientsystemen über dasselbe Interface kommunizieren.
- Über die Adapterklassen können Daten Konvertierungsfunktionen gekapselt werden.

Nachteile

Über folgende Nachteile muss man sich bei der Verwendung dieses Patterns bewusst sein.

- Durch die fehlende Mehrfachvererbung in JAVA muss für jede Adaptierung eine unabhängige Adapterklasse verwendet werden. Adapter Hierarchien sind nicht möglich.
- Der zusätzliche Zwischenschritt erschwert das Debugging im Fehlerfall.

2.6.3 Layered-Architecture-Pattern

Das Architektur Design Pattern der Layer [SSRB13 S. 31-41] beschäftigt sich mit der Strukturierung von Software. Dabei werden Funktionen einer Software zu Funktionsgruppen zusammengefasst. Diese Funktionsgruppen (oder Layer) kommunizieren über klar definierte Interfaces. Dadurch lassen sich beliebige Layer Implementierungen miteinander kombinieren, solange die Interfacedefinition eingehalten wird. Wie bereits in Kapitel 2.1.1 erwähnt, hat das Layered-Architecture-Pattern einen hohen Stellenwert im Projekt ARS. Das in 2.1.2 vorgestellte Funktionsmodell beruht auf den Prinzipien dieses Entwurfsmusters.

Problemstellung

Man stelle sich ein System mit unterschiedlichen High Level und Low Level Funktionen vor, wobei die High-Level Funktionen auf den Low-Level Funktionen aufbauen. Möglich Low-Level Funktionen wären Speicherzugriff oder der Zugriff auf ein Kommunikationsmedium. Die darauf aufbauenden High-Level Funktionen wären zum Beispiel Benutzerinteraktion oder Steuerungsfunktionen. Die typische Abarbeitungsreihenfolge geht von den Low-Level zu den High-Level Funktionen. Es besteht also eine vertikale Verbindung zwischen den Funktionen. Weiters wird bei solchen Systemen eine horizontale Verbindung zwischen den Funktionen gleichen Typs verlangt. Ein Beispiel dazu wäre die horizontale Verbindung gleicher Layer in Kommunikationsmodellen wie dem ISO/OSI Modell. In solchen Systemen müssen folgende Punkte beachtet werden:

- Änderungen der Implementierung einer Funktion sollen keinen Einfluss auf andere Funktionen haben
- Komponenten sollen austauschbar sein, ohne den Rest des Systems zu beeinflussen
- Ähnliche Funktionen sollen gruppiert werden, um die Wartbarkeit und Übersichtlichkeit zu gewährleisten
- Komponenten sollen von unterschiedlichen Entwicklern hergestellt werden können

Lösungsansatz

Ein Lösungsansatz der die beschrieben Punkte beachtet, besteht darin, das Set an Funktionen nach ihrem Abstraktionsniveau in Schichten (Layer) aufzuteilen. Diese Layer werden, beginnend mit dem niedrigsten Abstraktionsniveau, vertikal aufeinander gesetzt. Dabei entsteht eine Struktur in der jeder Layer nur mit dem darüber und darunterliegenden Layer kommunizieren darf (siehe Abbildung 31). Die Interfaces zwischen den Layern müssen definiert sein. Die jeweiligen Funktionen innerhalb des Layers können beliebig variiert werden, solange die Interface Spezifikation eingehalten wird.

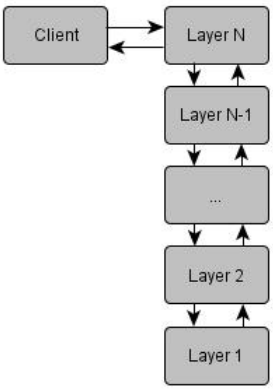

Abbildung 31: Schichtenarchitektur [SSRB13 S. 35]

Durch diesen Ansatz können folgende Szenarien abgedeckt werden.

1. **Top- Down Communication:** Ein Client richtet seine Anfrage an Layer N. Dieser kann die Anfrage nicht alleine beantworten. Er teilt diese in Anfragen eines niedrigeren Abstraktionslevels auf und richtet diese weiter an Layer N-1. Eine Anfrage an Layer N wird also in mehrere Anfragen an Layer N-1 aufgeteilt. Dies wird bis Layer 1 fortgesetzt. In umgekehrter Richtung werden die Einzelanfragen wieder zur Ausgangsanfrage des Clients zusammengesetzt.
2. **Bottom-Up Communication**: Die Kette der Aktionen beginnt bei Layer 1. Zum Beispiel wenn ein Geräte Treiber einen Input erkennt. Dabei können, ähnlich wie bei der Aufteilung der Anfragen aus Punkt 1, mehrere Inputs zu einer Aktion kombiniert werden.
3. **Teilweiser Aufruf Top-Down:** Dieses Szenario beschreibt den Fall, in dem nicht die gesamte Hierarchie durchlaufen werden muss. Wenn der Layer N die benötigte Information in einem Cache Speicher abgespeichert hat, dann ist es nicht notwendig beim Layer N-1 um diese anzufragen.
4. **Teilweiser Aufruf Bottom-Up:** Ähnlich wir beim Szenario 4, nur wird die Weitergabe eines Signals vom Layer N auf den Layer N+1 nicht durchgeführt. Eine Anfrage an einem Server könnte beispielsweise nicht weitergegeben werden, wenn das Ergebnis der Anfrage bereits im Layer N gechached wurde.
5. **Zwei Stack Communication:** Dieses Szenario beschreibt die Kommunikation von zwei Kommunikationsstacks miteinander (siehe Abbildung 32). Dabei wird deutlich, dass sich die beiden Kommunikationsteilnehmer nur an Layer 1 berühren, und auf beiden Seiten alle Layer durchlaufen werden müssen.

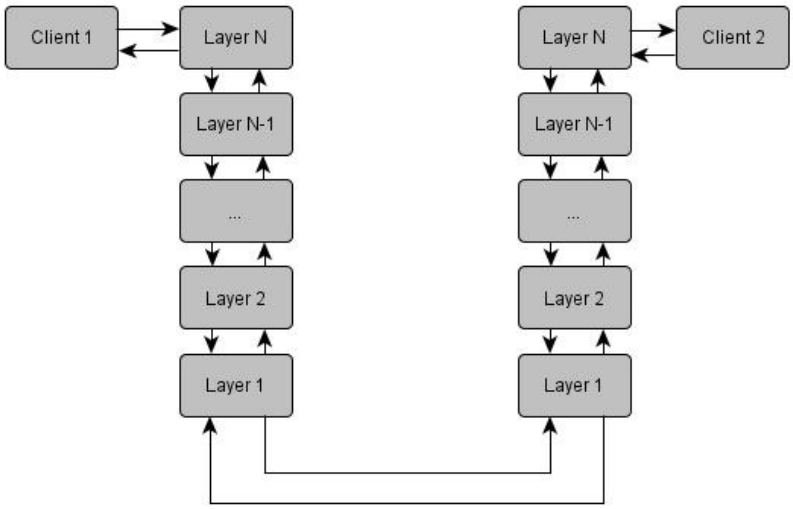

Abbildung 32: Zwei Kommunikations-Stacks [SSRB13 S. 37]

Vorteile

Aus dem, im Lösungsansatz vorgestellten Modell ergeben sich folgende Vorteile:

1. Layer Implementierungen können durch die strikte Interface Definition in unterschiedlichen Kontexten wiederverwendet werden.
2. Durch die strikten Interfaces ist es möglich die Implementierung der einzelnen Layer auszutauschen.
3. Fehler können im auftretenden Layer abgefangen werden. Ein ausbreiten auf die gesamte Software kann leichter verhindert werden
4. Die Komplexität der Software kann reduziert werden.

Nachteile

Folgende Nachteile ergeben sich aus dem Lösungsansatz, und müssen bei der Umsetzung berücksichtigt werden:

1. Kommt es zu Abhängigkeiten zwischen dem Layer N und dem Verhalten der darunterliegenden Layer, können diese nur bedingt ausgetauscht werden. Zum Beispiel wenn der Layer N in einem Datenübertragungssystem eine minimale Übertragungsrate voraussetzt.
2. Durch das einteilen in Layer entsteht kommunikationsaufwand zwischen diesen. Dieser Aufwand verringert die Effizienz der Software.

Um die Vor- und Nachteile ins richtige Verhältnis zu bekommen, sind die Anzahl und die Aufteilung der Layer von entscheidender Bedeutung.

Beispiel– ISO/OSI Modell

Als Paradebeispiel für eine in Layer aufgeteilte Kommunikationsarchitektur ist das ISO/OSI Modell zu nennen [Zim80]. In Abbildung 33 ist eine schematische Darstellung zu sehen.

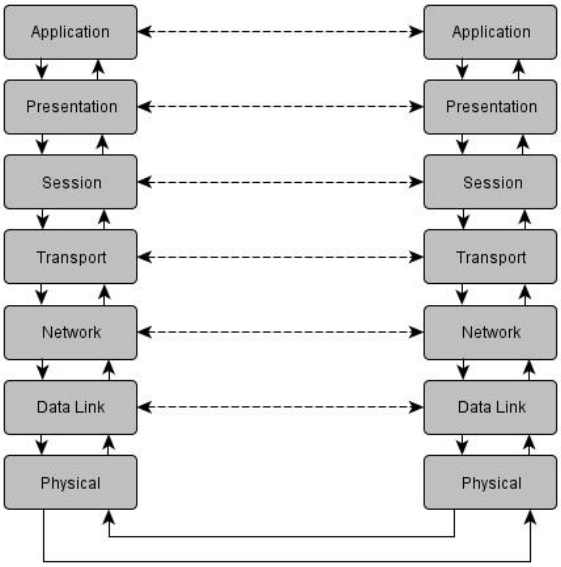

Abbildung 33: ISO/OSI Referenzmodell [Zim80 S. 3]

In [Zim80 S. 5-6] werden für die in Abbildung 33 dargestellten Layer beschrieben und folgende Aufgaben werden zugewiesen:

1. **Physical Layer** - Beschreibt mechanisch, elektrische und prozedurale Eigenschaften des Übertragungsmediums
2. **Data Link Layer** – Beschreibt die Funktionen die notwendig sind um ein Verbindung zwischen zwei Netzwerkteilnehmer aufzubauen.
3. **Network Layer** – Beschreibt das Routing zwischen Kommunikationsteilnehmer die nicht direkt mit einander verbunden sind.
4. **Transport Layer** – Bietet den darüberlegenden Layern die Möglichkeit des Zugriffs ohne die Eigenschaften des Kommunikationsmediums berücksichtigen zu müssen.
5. **Session Layer** – Bietet die Möglichkeit Sitzungen zwischen den Kommunikationsteilnehmern aufzubauen.
6. **Presentation Layer** – Ermöglich die syntaktisch korrekte Darstellung der Daten, sowie Funktion zur Datenkompression bzw. Verschlüsselung.
7. **Application Layer** – Stellt Zugriffs Funktionen für die Anwendung zur Verfügung.

3. Model und Konzepte

Ausgehend aus dem in Kapitel 1.2 beschriebenen IST Zustand und den in Kapitel 2 gewonnen Erkenntnissen kann ein neuer Aufbau (Abbildung 34) der Simulationsumgebung erstellt werden. Wie in Abbildung 34 zu sehen ist, verlangt die Decision Unit eine bidirektionale Schnittstelle zur Ein- und Ausgabe von Daten. Außerdem ist eine Trennung zwischen Kontroll- und Datenfluss notwendig.

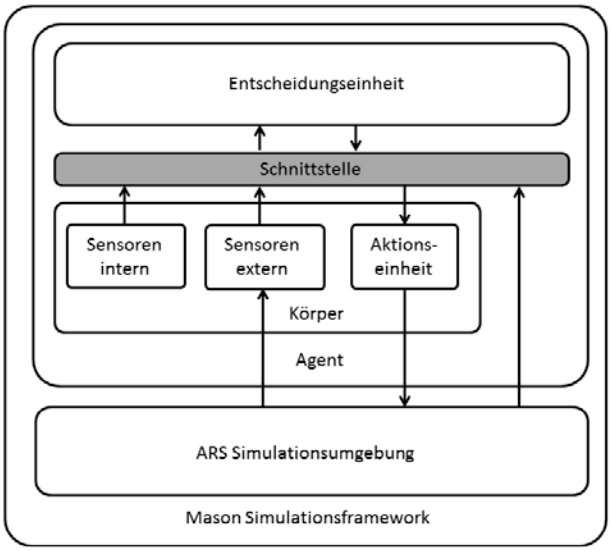

Abbildung 34: Konzept Schnittstelle

In diesem Kapitel wird das Modell der Schnittstelle zur ARS Decision Unit vorgestellt. Vor dem Beginn der Modellbildung mussten Anforderungen an die Schnittstelle definiert werden. Anschließend konnte unter Berücksichtigung von Kapitel 2.6 das Modell der Schnittstelle entwickelt werden. Weiters wird die jetzige Schnittstelle zur Decision Unit untersucht. Da diese an die Anforderungen der ARS-Simulationsumgebung angepasst ist, sind einige Eigenschaften der Schnittstelle auf die Simulationsumgebung angepasst. Alle Anpassungen, die nicht modellbedingt sind, sollen aus der Definition der Schnittstelle herausfallen.

3.1 Anforderungsanalyse

Vor dem Beginn der Modellierung mussten zuerst die Anforderungen die an die Schnittstelle gestellt werden definiert werden. Hier wurden sowohl die Anforderungen der Decision Unit selbst, als auch die der Umgebungen in denen die Decision Unit eingesetzt werden soll mit einbezogen. Ein Überblick über die herausgearbeiteten Anforderungen ist in Tabelle 4 zu sehen. In den folgenden Kapiteln werden die einzelnen Anforderungen im Detail beschrieben.

	Anforderung	**Quelle**
A01	Kontroll- und Datenfluss müssen voneinander getrennte sein.	ARS Decision Unit
A02	Die erwartete Datenstruktur muss eingehalten werden.	ARS Decision Unit
A03	Die Schnittstelle muss das Senden und Empfangen von Daten ermöglichen.	ARS Decision Unit
A10	Der Kommunikationsaufwand der Schnittstelle darf den der jetzigen Kommunikation nicht übersteigen.	ARS-Simulationsumgebung
A11	Es muss weiterhin möglich sein die Decision Unit mit Hilfe von Parametern zu konfigurieren.	ARS-Simulationsumgebung
A12	Die DU muss trotz der Verwendung einer Schnittstelle direkt aus der Simulationsumgebung instanzierbar sein.	ARS-Simulationsumgebung
A13	Mit Hilfe der Schnittstelle muss es möglich sein eine synchrone Kopplung der Kommunikationspartner zu erreichen.	ARS-Simulationsumgebung
A20	Ein Mapping von unterschiedlichen Datenstrukturen muss innerhalb der Schnittstelle erfolgen.	Alternative Umgebungen
A21	Die Schnittstelle muss es erlauben die Decision Unit als eigenen Thread auszuführen.	Alternative Umgebungen
A22	Mit Hilfe der Schnittstelle muss es möglich sein eine asynchrone Kopplung der Kommunikationspartner zu erreichen.	Alternative Umgebungen
A23	Die eigentliche Datenübertragung muss austauschbar konzipiert werden.	Alternative Umgebungen
A24	Die Decision Unit muss den unterschiedlichen Anforderungen der Ausführungsumgebungen angepasst werden können.	Alternative Umgebungen
A30	Es darf keine parallelen Schnittstellen für unterschiedliche Ausführungsumgebungen geben.	Alte Projekte
A31	Es dürfen keine Abhängigkeiten zwischen Simulationsum-	Alte Projekte

	gebung und Decision Unit existieren.	
A32	Die Funktionen der Decision Unit dürfen nicht auf die Anforderungen der Ausführungsumgebung angepasst werden.	Alte Projekte
A33	Die Schnittstelle muss für die verschiedenen Ausführungsumgebungen konfigurierbar sein.	Alte Projekt

Tabelle 4: Anforderungsanalyse

3.1.1 Anforderungen der ARS Decision Unit

Die Anforderungen A01-A03 aus Tabelle 4 wurden aus der Funktion (Kapitel 2.1.2) und der Architektur (Kapitel 2.2.2) der im ARS-Projekt entwickelten Decision Unit extrahiert.

A01 Kontroll- und Datenfluss müssen voneinander getrennte sein

Wie in Kapitel 2.2.2beschrieben wird die Decision Unit innerhalb des Simulationszyklus *Execution* ausgeführt. Dieser Simulationszyklus wird vom Scheduler der Simulation getriggert. Die Input bzw. Output Daten werden jedoch mit dem Körper des Agenten ausgetauscht. Daraus ergibt sich, dass die Ausführungskontrolle vom Datenfluss getrennt werden muss. Abbildung 35 zeigt den schematischen Ablauf der Kommunikation, der laut Anforderung A01 erreicht werden soll. Dabei ist der Körper des Agenten durch die Module *SensorEngine* und *ActionEngine* repräsentiert.

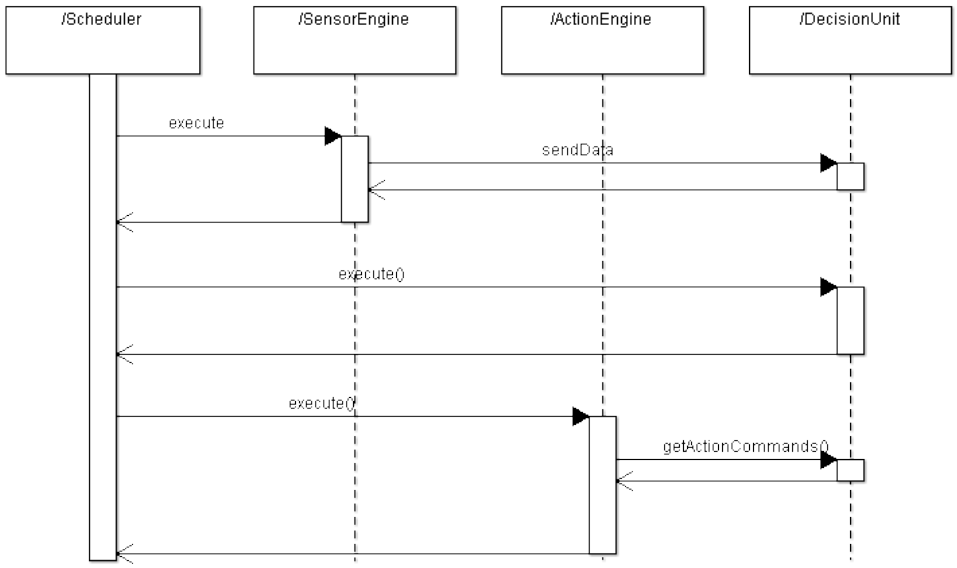

Abbildung 35: Kontroll- und Datenfluss

A02 Die erwartete Datenstruktur muss eingehalten werden

Die ARS Decision Unit ist zum jetzigen Stand der Modellierung (bzw. Implementierung) auf ein genau definiertes Set an Input Daten angewiesen. Diese Abhängigkeit resultiert nicht nur auf dem

verwendeten Interface, sondern auch auf der Weiterverarbeitung der Daten. Die Einschränkungen die durch das jetzige Interface bedingt sind, sollen aufgehoben werden. Im Zuge dieser Arbeit können jedoch Einschränkungen, die durch die weitere Verarbeitung der Daten bedingt sind, nicht behandelt werden. Bei der Modellierung des Interfaces muss also darauf geachtet werden, dass keine interfacebedingten Einschränkungen erzeugt werden, aber die Limitierungen der weiteren Informationsverarbeitung berücksichtigt werden. Daraus resultieren folgenden Subanforderungen.

- A02.1 – Die Anzahl der erwarteten Werte pro Datentyp muss eingehalten werden.
- A02.2 – Die im Modell verwendeten Bezeichnungen müssen eingehalten werden.

A03 Die Schnittstelle muss das Senden und Empfangen von Daten ermöglichen

Die Decision Unit verlangt sowohl ein Senden als auch ein Empfangen von Daten. Das hat zur Folge, dass eine bidirektionale Schnittstelle benötigt wird. In Abbildung 35 ist zu sehen, dass die Decision Unit sowohl beim Senden als auch beim Empfangen einen passiven Part einnimmt. Dabei handelt es sich jedoch nicht um einen modelltechnische, sondern um eine implementierungstechnische Lösung. Um die Möglichkeiten bei späteren Implementierungsänderungen nicht einzuschränken, soll die Schnittstelle auch eine aktiv kommunizierende Decision Unit unterstützen

3.1.2 Anforderungen der ARS-Simulationsumgebung

Da die Schnittstelle die Aufgabe hat die Decision Unit mit der ARS-Simulationsumgebung zu verbinden, müssen die Anforderungen dieser ebenfalls berücksichtigt werden. In diesem Kapitel werden die aus der Simulationsumgebung resultierenden Anforderungen A10-A13 (Tabelle 4) beschrieben.

A10 Der Kommunikationsaufwand der Schnittstelle darf den der jetzigen Kommunikation nicht übersteigen

Durch die aktuell enge Vermaschung zwischen Decision Unit und Simulationsumgebung ist der Rechenaufwand der benötigt wird, die Daten zu übertragen, im Moment sehr gering. Trennt man diese beiden logischen Einheiten auch implementierungstechnisch in zwei Funktionsblöcke entsteht zwangsweise Kommunikationsaufwand. Wird die Decision Unit innerhalb der ARS-Simulationsumgebung eingesetzt, soll um den Rechenaufwand möglichst niedrig gehalten werden. Aus diesem Grund soll die Kommunikation, wie in Abbildung 36 zu sehen, über einfache Funktionsaufrufe erfolgen. Dabei wird zwischen zwei Möglichkeiten unterschieden. Zum einen können die Daten vom Sender durch den Aufruf einer Empfängermethode über Methodenparameter übergeben werden. Und zum anderen können die Daten vom Empfänger angefragt, und über den Rückgabewert übergeben werden.

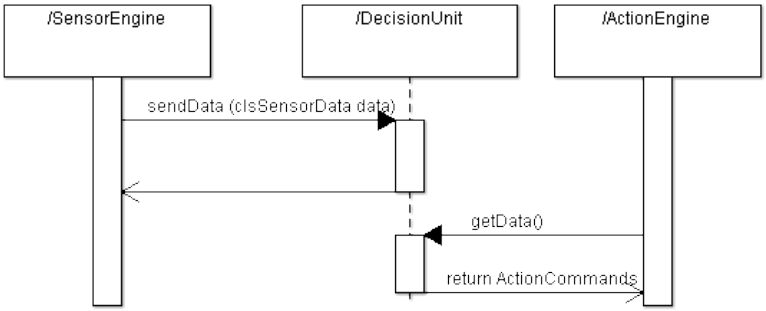

Abbildung 36: Kommunikation mittels Funktionsaufruf

A11 Es muss weiterhin möglich sein die Decision Unit mit Hilfe von Parametern zu Konfigurieren

Wie im Kapitel 2.1.1 beschrieben erfolgt die Verifizierung der Decision Unit über Use Cases. Um unterschiedliche Use Cases erzeugen zu können, sollen definierte Parameter während der Instanziierung der Decision Unit festgelegt werden können. Es muss also eine Möglichkeit geben die Decision Unit in unterschiedlichen Konfigurationen zu laden.

A12 Die Decision Unit muss trotz der Verwendung einer Schnittstelle direkt aus der Simulationsumgebung instanzierbar sein

Innerhalb der ARS-Simulationsumgebung soll die Decision Unit vom Körper des Agenten instanziiert werden. Dieser soll eine Referenz halten und eine Referenz an den Scheduler übergeben.

A13 Mit Hilfe der Schnittstelle muss es möglich sein eine synchrone Kopplung der Kommunikationspartner zu erreichen

Zwischen Körper und Decision Unit existiert keine synchrone Kopplung im eigentlichen Sinn (siehe Abbildung 37). Dabei wird die Simulationsumgebung solange in ihrer Ausführung blockiert, bis die Decision Unit die Rückgabewerte berechnet und übergeben hat.

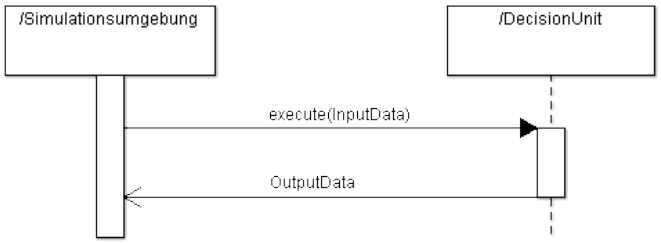

Abbildung 37: Synchrone Kommunikation

In der ARS-Simulationsumgebung wartet die Ausführung des Körpers nicht auf die Ausführung der Decision Unit. Durch die Simulationszyklen (Kapitel 2.2.2) wird jedoch dasselbe Verhalten wie bei einer synchronen Kopplung erzielt. Der Körper wird im Prinzip in zwei Funktionsblöcke geteilt. Ein Teil wird vor der Decision Unit und der andere Teil danach aufgerufen. Diese Reihenfolge wird vom

Scheduler vorgegeben und kann für jeden Simulationszyklus garantiert werden. Das ist wichtig, da es sonst zu einer nicht deterministischen Simulation kommt (siehe Kapitel 2.2.2).

3.1.3 Anforderungen aus alternativen Umgebungen

Da die Decision Unit in Zukunft nicht nur in der ARS-Simulationsumgebung eingesetzt werden soll, sondern auch in alternativen Umgebungen, müssen natürlich auch diese Umgebungen in die Modellierung der Schnittstelle mit einbezogen werden. Folgende Umgebungen wurden berücksichtigt:

- Unreal (Kapitel 2.4.1)
- Glashaus (Kapitel 2.4.2)
- Miklas (Kapitel 2.3)

A20 Ein Mapping von unterschiedlichen Datenstrukturen muss innerhalb der Schnittstelle erfolgen

Es soll möglich sein ohne funktionale Anpassung der Decision Unit, diese mit Hilfe der Schnittstelle zu verbinden. Die unterschiedlichen Konzepte und Bezeichnungen müssen aufeinander gemappt werden. Als Beispiel dafür kann das in Kapitel 2.5.2 vorgestellte BDI Body-Mind-Interface dienen. Dabei wir ebenfalls das Mapping der Input- und Ouputdaten innerhalb der Schnittstelle vorgenommen.

A21 Die Schnittstelle muss es erlauben die Decision Unit als eigenen Thread auszuführen

Es soll möglich sein die Decision Unit als eigene Thread auszuführen, um die parallele Ausführung der Umgebung und einer, oder mehrere Decision Units, zu ermöglichen. Dies steigert zum einen die Performance auf Multiprozessor Systemen, zum anderen ermöglicht es erst die in A22 geforderte asynchrone Kopplung.

A22 Mit Hilfe der Schnittstelle muss es möglich sein eine asynchrone Kopplung der Kommunikationspartner zu erreichen

Die asynchrone Kopplung ermöglicht es die Umgebung schneller oder langsamer laufen zu lassen als die Decision Unit. In Abbildung 38 ist der Ablauf einer asynchronen Kommunikation dargestellt. Sowohl Simulationsumgebung als auch Decision Unit laufen unabhängig voneinander. Die Kommunikationspakete werden zu zeitlich nicht spezifizierten Zeitpunkten übertragen.

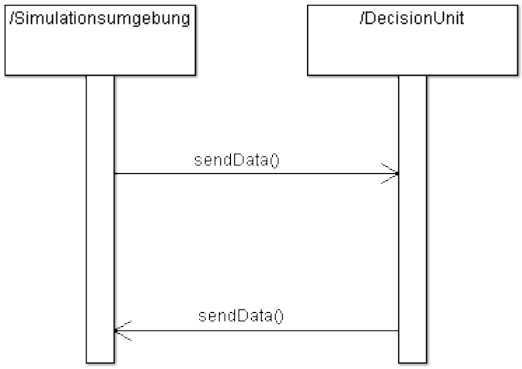

Abbildung 38: Asynchrone Kommunikation

In Systemen mit mehreren Agenten (bzw. Decision Units) kann es nicht dazu kommen, dass eine langsame Decision Unit das gesamte System bremst. Die Unreal Engine arbeitet nach diesem Prinzip.

A23 Die eigentliche Datenübertragung muss austauschbar konzipiert werden

A21 und A23 schaffen die Möglichkeit die Decision Unit und die Umgebung auf unterschiedlichen Systemen oder Umgebungen laufen zu lassen. Dies bedingt jedoch eine komplexere Art der Kommunikation als den jetzt verwendeten Funktionsaufruf. Die Schnittstelle soll also die Möglichkeit bieten die Kommunikationsmethodik beliebig austauschen zu können.

A24 Die Decision Unit muss den unterschiedlichen Anforderungen der Ausführungsumgebungen angepasst werden können

In verschieden Umfeldern gibt es unterschiedliche Konzepte die unterschiedliche Konfigurationen der Decision Unit bedingen. Es soll möglich sein, durch Konfiguration der Decision Unit die erwarteten Inputs auf die Umgebung abzustimmen. Dies kann jedoch nur im abgeschlossenen Rahmen geschehen, da die Informationsverarbeitung der ARS-Implementierung auf bestimmte Strukturen und Begrifflichkeiten angewiesen ist. Diese Einschränkungen können im Zuge dieser Arbeit nicht für das gesamte Funktionsmodell entfernt werden. In Grundzügen soll jedoch eine Anpassung möglich sein.

3.1.4 Lernen aus früheren Projekten

Im Laufe des ARS-Projektes wurden bereits parallele Projekte (siehe Kapitel 2.4) gestartet, die sich zur Aufgabe (oder zumindest als Teilaufgabe) gestellt haben eine allgemeine Schnittstelle zur Decision Unit zu modellieren. Die Anforderungen dieses Kapitel wurden aus diesen Projekten extrahiert um aus diesen zu lernen.

A30 Es darf keine parallelen Schnittstellen für unterschiedliche Ausführungsumgebungen geben

Es darf keine parallel gültigen Schnittstellen zu den einzelnen Umgebungen geben. Die Vergangenheit zeigt, dass bei Veränderungen meist nur die aktuell verwendete Struktur weiterentwickelt wird. Alle parallel dazu geführten Schnittstellen werden nicht gewartet und verlieren so ihre Gültigkeit. Die neuen Entwicklungen der Decision Unit können also nicht in die alternative Umgebung übernommen werden.

A31 Es dürfen keine Abhängigkeiten zwischen Simulationsumgebung und Decision Unit existieren

Es soll keine Abhängigkeiten zwischen der ARS Decision Unit und der Simulationsumgebung geben. Diese Abhängigkeiten existieren, da beide aufeinander abgestimmt entwickelt wurden und dabei nicht auf ein klares Interface geachtet wurde. In früheren Projekten wurde es nicht geschafft diese Abhängigkeiten vollständig aufzubrechen. Dadurch wurden Teile der ARS-Simulationsumgebung ausgeführt, auch wenn die Decision Unit in anderen Umgebungen eingesetzt wurde.

A32 Die Funktionen der DU dürfen nicht auf die Anforderungen der Ausführungsumgebung angepasst werden

Einzelne Funktionen der Decision Unit dürfen nicht auf die Anforderungen bestimmter Umgebungen angepasst werden. Das führt, ähnlich wie in A30 zu parallelen Strukturen, die im Falle von Änderungen nicht gewartet werden. Wird die Decision Unit in vielen unterschiedlichen Umgebungen eingesetzt, dann würde das zu einer Menge an Anpassungen führen, die nicht mehr überschaubar wären.

A33 Die Schnittstelle muss für die verschiedenen Ausführungsumgebungen konfigurierbar sein

Aufgrund der vielen verschiedenen Einsatzgebiete für die die Schnittstelle konzipiert werden soll, muss die Möglichkeit bestehen die Schnittstelle zu konfigurieren. Definierte Funktionsblöcke sollen beliebig miteinander kombiniert werden können.

3.2 Systementwurf

Um die in Kapitel 3.1 definierten Anforderungen umsetzten zu können, musste eine Architekturmodell der Schnittstelle definiert werden. Ganz allgemein soll laut Anforderung A01 der Kontroll- und Datenfluss getrennt werden. Daraus ergibt sich der in Abbildung 39 zu sehende Aufbau.

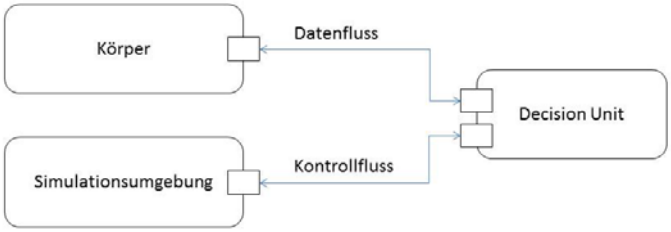

Abbildung 39: Aufbau Schnittstelle

Abbildung 39 zeigt den Schnittstellenaufbau und die beispielhafte Einbindung in die ARS-Simulationsumgebung. Es werden jeweils für Kontroll- und Datenfluss getrennte Kommunikationskanäle erzeugt. Jeder dieser beiden Kommunikationskanäle ist bidirektional ausgeführt, um der Anforderung A03 gerecht zu werden. In weiterer Folge wird ein Modell dieser Kommunikationskanäle beschrieben. Die Definition des Modells wurde dabei so flexibel gehalten, dass es sowohl für Kontroll- als auch Datenfluss übernommen werden kann. Dadurch kann auch die Anforderung A33 abgedeckt werden, die eine Konfigurierbarkeit der Schnittstelle für unterschiedliche Ausführungsumgebungen fordert. Um diese Flexibilität erreichen zu können, ist ein konfigurierbares Design unbedingt notwendig. Dafür wurde das ISO/OSI Modell (Kapitel 2.6.3) als Vorbild gewählt, da dessen Schichtenaufbau sich gut eignet, um diesen unterschiedlichen Anforderungen gerecht zu werden.

Abgeleitet vom Prinzip des ISO/OSI Modells wurde das in Abbildung 40 zu sehende Kommunikationsmodell definiert. Dazu wurden die Layer Kopplung, Datenpuffer, Datenkonvertierung und Kommunikation definiert, die im Kapitel 3.5 genauer spezifiziert werden. In Abbildung 40 ist zu sehen, dass diese Layer innerhalb der Schnittstelle gekapselt sind. Die Layer sind durch ihre Interfaces zu den benachbarten Layern definiert und können innerhalb dieser Definition mit beliebiger Funktionalität befüllt werden. Die Schnittstelle muss für jeden Kommunikationspartner erzeugt werden und kann über den Kommunikationsport von diesem angesprochen werden.

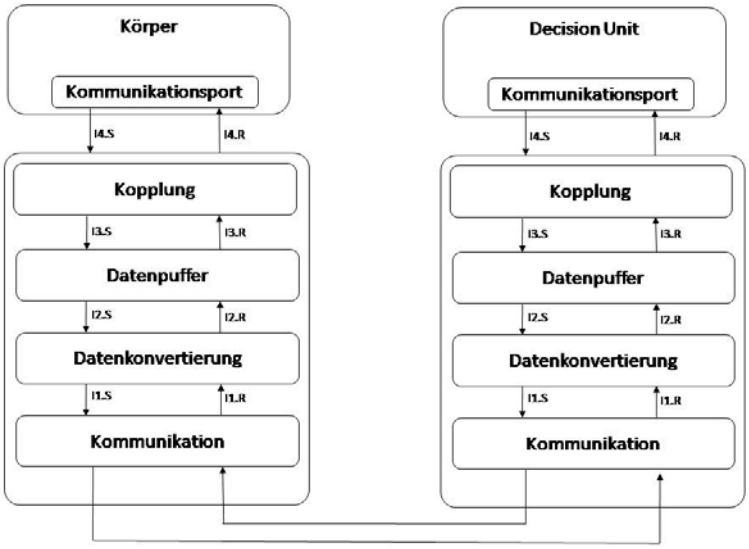

Abbildung 40: Schnittstellen Model

3.2.1 Datenstrukturen

Zur Übertragung der Daten zwischen den einzelnen Kommunikationslayern wurde die in Abbildung 41 zu sehende Datenstruktur definiert. Der *DataContainer* bildet dabei ein Interface mit der die Daten angesprochen werden können. Die Eigentliche Repräsentanz der Daten ist dahinter versteckt und könnte ausgetauscht werden. Die Daten werden durch Objekte der Klassen *DataPoint* repräsentiert. Durch die Assoziation auf sich selbst ist es möglich, mit nur einer Klasse komplexe Baumstrukturen aufzuspannen.

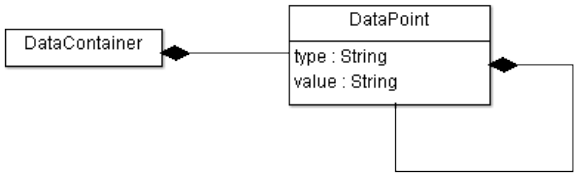

Abbildung 41: Datenstruktur Schnittstelle

Diese Datenstruktur wurde gewählt, da sich mit Hilfe dieser alle assoziativen Ein- bzw. Ausgangsdaten (siehe Kapitel 2.1.3) der Decision Unit beschreiben lassen. In Abbildung 42 ist am Beispiel eines Vision Sensor Inputs ein Objektdiagramm der Datenstruktur zu sehen.

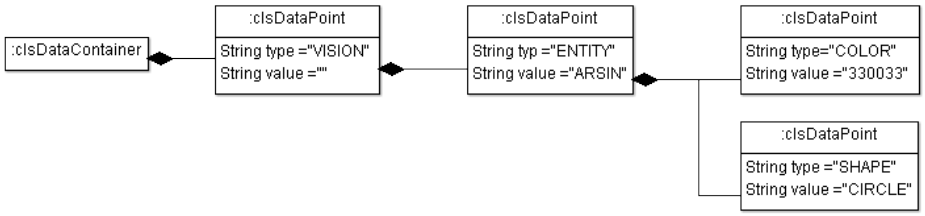

Abbildung 42: Objektdiagramm Datenstruktur

3.2.2 Schnittstellendefinitionen

In diesem Kapitel werden die in Abbildung 40 definierten Schnittstellen zwischen den Komponenten spezifiziert. Eine Spezifikation dieser Schnittstellen ist nötig, da nur so die Komponenten als eigenständige Funktionseinheiten betrachtet werden können, und trotzdem ein Zusammenspiel der Komponenten ermöglicht wird.

I4.S Kommunikationsport - Kopplungslayer

Über das Interface I4.S wird vom Kommunikationsport der Sendevorgang gestartet, und die zu sendenden Daten übergeben. Als Datenformat ist der in Kapitel 3.2.1 definierte *DataContainer* definiert.

I4.R Kopplungslayer -Kommunikationsport

Das Interface I4.R übergibt dem Kommunikationsport die Empfangenen Daten. Die Interfacedefinition stellt zwei Arten der Datenübergabe zur Verfügung. Die eine ermöglicht ein Übergeben der

Daten von der Schnittstelle an den Kommunikationsport. Dabei steuert die Schnittstelle den Aufruf der Funktion. Andererseits besteht die Möglichkeit, dass der Kommunikationsport aktiv die Daten von der Schnittstelle anfordert. Dabei kann die Schnittstelle den Kommunikationsport informieren, wenn abholbereite Daten vorliegen. Als Datenformat wird ebenfalls der in Kapitel 3.2.1 definierte *DataContainer* verwendet.

I3.S Kopplungslayer - Datenpufferlayer

Das Interface I3.S definiert die Übergabe der Daten zwischen Kopplung- und Datenpuffer- Layer in Senderichtung. Als Datenformat wird der in Kapitel 3.2.1 definierte *DataContainer* verwendet.

I3.R Datenpufferlayer - Kopplungslayer

Dieses Interface besteht aus zwei Teilen. Zum einen wird der Kopplungslayer über das Eintreffen neuer Daten informiert, und zum anderen kann der Kopplungslayer Daten anfordern. Die an den Kopplungslayer übergebenen Daten müssen bereits die von der Anwendung erwartete Struktur aufweisen. Im Kopplungslayer wird diese nicht mehr verändert. Dieser dient lediglich zu Ablaufsteuerung. Als Datenformat bei der Datenübertragung wird der in Kapitel 3.2.1 definierte *DataContainer* verwendet. Die Signalisierung neuer Daten wird mittels Funktionsaufruf realisiert.

I2.x Datenpufferlayer - Datenkonvertierungslayer

Die Interfaces I2.S und I2.R definieren den Datenaustausch zwischen Datenpuffer- und Datenkonvertierung-Layer in Sende- bzw. in Empfangsrichtung. Als verwendete Datenstruktur wird der in Kapitel 3.2.1 definierte *DataContainer* verwendet.

I1.x Datenkonvertierungslayer - Datenübertragungslayer

Die Interfaces I1.S und I1.R definieren den Datenaustausch zwischen Datenkonvertierung- und Kommunikation-Layer in Sende- bzw. in Empfangsrichtung. Die Daten werden jeweils als *String* übergeben.

3.2.3 Erzeugung und Konfiguration

Die Konfiguration der Schnittstelle erfolgt bei ihrer Erzeugung. Eine Änderung der Konfiguration im laufenden Betrieb ist nicht vorgesehen. Bei der Erzeugung muss für jeden Layer eine Implementierung instanziiert und die jeweiligen Interfaces zusammengeschalten werden. Um dies zu erleichtern und Konfiguration definieren zu können, wird auf das Factory Methode Pattern zurückgegriffen (siehe Kapitel 2.6.1). Dabei wird ein Factory definiert, die vordefinierte Konfigurationen erzeugen kann. Wird die benötigte Layerkombination noch nicht durch eine der Factory Methoden abgedeckt, kann diese natürlich auch manuell zusammengestellt werden.

3.3 Nicht Funktionale Requirements

In diesem Kapitel soll gezeigt werden, dass nicht nur die Anforderungen, die direkt in Funktionen münden umgesetzt wurden, sondern auch alle anderen aus Kapitel 3.1. Die in Tabelle

5eingetragenen nicht funktionale Anforderungen konnten nicht über die Definition der Komponenten erfüllt werden.

	Anforderung
A11	Es muss weiterhin möglich sein die DU mit Hilfe von Parametern zu konfigurieren.
A24	Die Decision Unit muss den unterschiedlichen Anforderungen der Ausführungsumgebungen angepasst werden können.
A30	Es darf keine parallelen Schnittstellen für unterschiedliche Ausführungsumgebungen geben.
A31	Es dürfen keine Abhängigkeiten zwischen Simulationsumgebung und DU existieren.
A32	Die Funktionen der DU dürfen nicht auf die Anforderungen der Ausführungsumgebung angepasst werden.
A33	Die Schnittstelle muss für die verschiedenen Ausführungsumgebungen konfigurierbar sein.

Tabelle 5: Nicht Funktionale Anforderungen

3.3.1 A11 und A24

Die Anforderungen A11 fordert die Konfigurierbarkeit der Decision Unit. Diese ist innerhalb der Decision Unit über Konfigurationsfiles möglich. Die Auswahl dieser kann entweder bei der Instanzierung oder über das Kontrollinterface erfolgen. Die Schnittstelle schränkt die Konfigurierbarkeit also in keinster Weise sein. Die in A24 geforderte Anpassung der Decision Unit an die Ausführungsumgebungen kann zum jetzigen Stand des Projektes nicht eingehalten werden. Der Grund dafür liegt jedoch nicht im funktionalen Umfang der Schnittstelle, sondern in den Einschränkungen der Decision Unit. Die Schnittstelle schränkt die Konfigurierbarkeit jedoch in keinster Weise ein.

3.3.2 A30

Die Anforderung A30 verlangt, dass es keine parallelen Strukturen der Datenübertragung geben soll. Das bedeutet in unserem Fall, dass die Decision Unit nur eine Schnittstelle zur Übertragung von Daten haben soll. In unterschiedlichen Ausführungsumgebungen soll jede Art der Steuerung oder Kommunikation über diese eine Schnittstelle erfolgen. Diese Anforderung kann eingehalten werden, da die definierte Schnittstelle, durch verschiedenen Layer Implementierung auf die Anforderungen der Ausführungsumgebungen angepasst werden kann. Somit ist es nicht nötig, parallele Schnittstellen für jede Ausführungsumgebung zu implementieren.

3.3.3 A31

In Anforderung A31 wird gefordert, dass die Decision Unit unabhängig von der aktuellen Simulationsumgebung ausgeführt werden soll. Dies wird durch die Einführung der Schnittstellen für Kontroll- und Datenfluss implizit sichergestellt. Da jegliche Art der Kommunikation zwischen Ausfüh-

rungsumgebung und Decision Unit nur über diese Schnittstellen erfolgt gibt es darüber hinaus keine Abhängigkeiten dieser beiden.

3.3.4 A32

In Anforderung A32 wird gefordert, dass die Decision Unit funktional nicht auf die Bedürfnisse der Ausführungsumgebung angepasst werden darf. Durch den Einsatz einer Schnittstelle kann natürlich nicht verhindert werden, dass dies geschieht. Da die einzelnen Layer der Schnittstelle austauschbar sind, ist eine Anpassung der Decision Unit jedoch nicht notwendig. Die nötige Flexibilität für den Einsatz in unterschiedlichen Umgebungen sitzt bereits in der Schnittstelle.

3.4 Analyse der existierenden Schnittstelle

Bevor das in Kapitel 3.2 definierte Schnittstellemodell auf die ARS Decision Unit angewandt werden kann, muss die existierende Schnittstelle analysiert werden. Aufbauend auf der Analyse in Kapitel 2.1.3 wird das Zusammenspiel zwischen Decision Unit und Simulationsumgebung analysiert, um Schnittstelleneigenschaften zu finden, die aufgrund der Simulationsumgebung bedingt sind. Sind diese Eigenschaften nicht durch das ARS-Funktionsmodel begründet sondern nur durch den Aufbau der Simulationsumgebung, dann soll die Schnittstelle um diese Punkte bereinigt werden. Im Folgenden werden die Analyse dieser Punkte und deren Bereinigungen beschrieben.

3.4.1 Vision Sensor

Die Sensordaten werden wie in Kapitel 2.1.3 beschrieben und gruppiert nach den jeweiligen Sensoren übertragen.

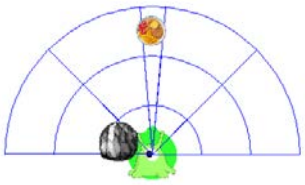

Abbildung 43: Visuelle Sensoren

In Abbildung 43 ist der Agent, zwei Simulationsobjekte und als blaues Radar die drei visuellen Sensoren zu sehen. Aufgrund des Simulator Aufbaues gibt es nicht einen visuellen Sensor, sondern jeder der drei Bereiche ist als eigener Sensor registriert. In weiterer Folge werden alle Sensoren abgerufen und übergeben ihre Daten an die Decision Unit. Die Daten für die in Abbildung 43 abgebildete Situation ist in Abbildung 44 zu sehen.

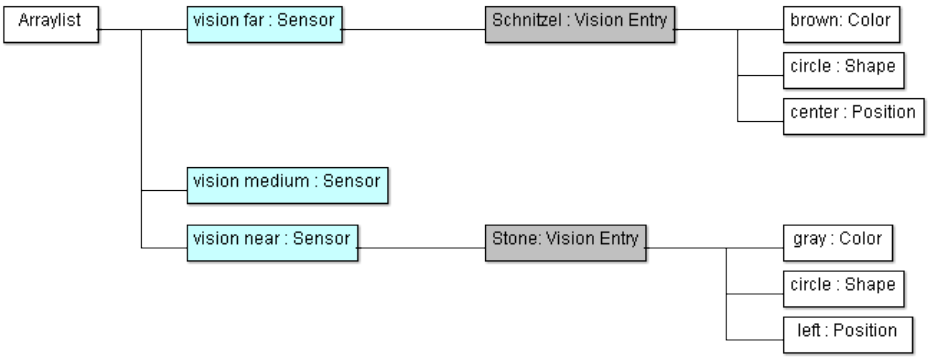

Abbildung 44: Visuelle Sensoren

Basierend auf diesen Daten werden innerhalb der Decision Unit die Objekte aus den Sensoren ausgelesen, und ihnen die Distanz des jeweiligen Sensors angehängt. Diese Unterteilung in drei verschiedene visuelle Sensoren ist nicht durch das ARS-Funktionsmodell, sondern nur durch den Simulator Aufbau bedingt. Aus dieser Unterteilung ergibt sich die schlechte Skalierbarkeit der Positionswerte. Es ist nicht ohne weiteres möglich, eine andere Skalierung zu verwenden bzw. auf höher dimensionale Positionswerte umzusteigen.

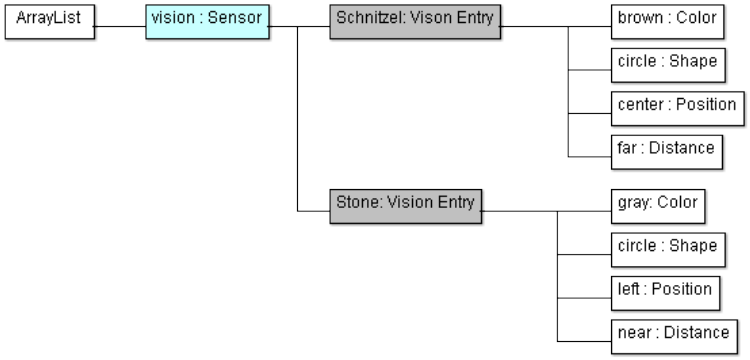

Abbildung 45: Visueller Sensor neu

In Abbildung 45 ist das Beispiel in der bereinigten Datenstruktur zu sehen. Anstatt der drei Sensoren gibt es nur noch einen visuellen Sensor. Die Information über die Entfernung wird mit Hilfe einer zusätzlichen Eigenschaft weitergegeben. Die beiden Eigenschaften *Position* und *Distance* bestimmen nun die relative Position des Objektes. Diese beiden Eigenschaften lassen sich nun beliebig skalieren, bzw. um zusätzliche Dimensionen erweitern.

3.4.2 Sequenced Actions

Das im Funktionsmodell beschriebene Interface I0.6 zeigt in welcher Form Aktionsbefehle von der Decision Unit an die Action Engine übergeben werden. Es wird dabei eine Liste an Aktionen übergeben. Einer der Untertypen von Aktion ist jedoch eine weitere Listenstruktur. Das hat zur Folge, dass sehr unübersichtliche verschachtelte Listen an Aktionsbefehlen übergeben werden. Begründet

liegt dies in der Struktur der Action Engine, die für diese Verschachtelungen entwickelt wurde. Da es für diese unübersichtliche Struktur keine modelltechnischen Gründe gibt, wird diese durch eine einfache Liste an Aktionsbefehlen ersetzt.

3.5 Komponentenentwurf

Basierend auf den Schnittstellendefinition aus Kapitel 3.2.2 können die in Abbildung 40 definierten Komponenten spezifiziert werden. In weiterer Folge werden die Komponentenspezifikationen als Basis für die Implementierung herangezogen.

3.5.1 Körper/ Decision Unit

Diese beiden Funktionseinheiten beschreiben die Kommunikationspartner die die Schnittstelle nutzen. Sie dienen in Abbildung 40 als Beispiel, wie die Schnittstelle eingesetzt werden kann. Da die Einbindung der Schnittstelle über den Kommunikationsport erfolgt und dieser nicht Teil des Schnittstellemodels ist muss das Interface zum Kommunikationspartner nicht definiert werden. Dies erlaubt größtmögliche Flexibilität beim Einbinden des Kommunikationsports in die Anwendungsfunktion.

3.5.2 Kommunikationsport

Eine strikte Trennung zwischen Funktionen und Daten ermöglicht es die Eingangs- und Ausgangsdaten der Decision Unit durch Konfiguration an die jeweiligen Umgebungen anzupassen. Diese strikte Trennung findet jedoch nur auf Funktionsmodellebene statt. In der aktuellen Implementierung des Funktionsmodelles ist diese Trennung nicht konsequent durchgezogen. Aus diesem Grund ist es nicht möglich die Decision Unit im benötigten Ausmaß zu konfigurieren. Daraus folgt, dass um die Anforderungen A02 und A20 einhalten zu können, ein Mapping der Daten auf Schnittstellenebene erfolgen muss. Dabei werden die vom Körper kommenden Signale auf die erwarteten Eingangssignale der Decision Unit überführt. Diese Aufgabe wird aus dem eigentlichen Schnittstellenmodell in den Kommunikationsport ausgelagert. Dieser ist nicht Teil des Modells. Er stellt eine Verbindung zwischen Kommunikationsteilnehmer und der Schnittstelle her. Da der Zugriff auf die Schnittstelle über ein definiertes Interface (I4.S und I4.R) erfolgt, bietet der Kommunikationsport eine flexible Möglichkeit der Integration in die eigentliche Anwendung. Dabei implementiert er sowohl das von der Schnittstelle als auch von der Anwendung vorgegebene Interface. Er stellt also einen Adapter zwischen den beiden Funktionsblöcken dar. Dabei werden sowohl die Funktionsaufrufe als auch der Datenfluss adaptiert. Für die Implementierung des Kommunikationsports kann das Adapter Pattern herangezogen werden (siehe Kapitel 2.6.2).

3.5.3 Kopplung

Der oberste Layer der Kommunikation bestimmt die Kopplung zwischen den Kommunikationsteilnehmern. Er stellt also sowohl die A13 geforderte synchrone Kommunikation als auch die in A22 geforderte asynchrone Kommunikation zur Verfügung. Diese beiden Anforderungen werden in unterschiedlichen Implementierungen des Layers umgesetzt. Je nach Konfiguration der Schnittstelle

werden die benötigten Implementierungen geladen. Die Art der Kopplung wird von der Wahl der Implementierung auf Empfängerseite festgelegt. Wird die Decision Unit in einem eigenen Thread ausgeführt, dann hat dieser Layer die Aufgabe die Kommunikation der beiden Threads zu synchronisieren (siehe Anforderung A21).

3.5.4 Datenpuffer

Dieser Layer stellt einen Datenpuffer zu Verfügung. Dieser ist vor allem bei einer Konfiguration als asynchrone Schnittstelle notwendig, da dabei die Eingangsdaten auf Empfängerseite bis zu Abholung von der Anwendung zwischengespeichert werden müssen. Wie diese Zwischenspeicherung und die Weitergabe der Daten erfolgt, wird von diesem Layer bestimmt. Durch diese Pufferung der Daten kann die in Anforderungen A02 geforderte Einhaltung der Datenstruktur auch bei asynchroner Kommunikation sichergestellt werden. In Abbildung 46 ist ein möglicher Ablauf einer asynchronen Datenübertragung zu sehen, der die zwei Problemfälle aufzeigt. Im ersten Fall sendet der Sender zwei oder mehrere Male hintereinander, ohne, dass der Empfänger die Daten ausliest. Im zweiten Fall liest der Empfänger zwei oder mehrere Male hintereinander Daten, ohne dass der Sender neue Daten senden konnte. Werden mehrere Werte hintereinander gesendet, dann ist immer der letzte Wert verfügbar. Der Wert bleibt so lange für Abfragen verfügbar, bis ein neuer Wert empfangen wird.

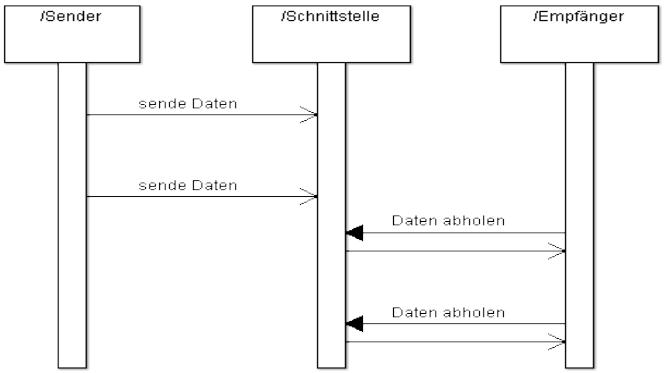

Abbildung 46: Asynchrone Datenübertragung

In diesem Fall können unterschiedliche Pufferungstypen nicht in getrennten Implementierungen realisiert werden. Die verschiedenen Typen müssen parallel in einer Implementierung verwendet werden. Es muss also je nach einkommenden Daten festgelegt werden welcher Puffertyp angewendet werden soll. Die Pufferung der Daten erfolgt dabei nur auf Empfängerseite. Die ausgehenden Daten des Senders werden nicht gepuffert.

3.5.5 Datenkonvertierung

In diesem Layer werden die Daten in ein Format gewandelt, dass es ermöglicht diese zu versenden. Dieser Layer wurde als Zwischenschritt vor der eigentlichen Datenübertragung eigeführt um diese

flexibler gestalten zu können. Für bestimmte Datenübertragungsverfahren müssen die Daten in serialisierbarer Form vorliegen um diese übertragen zu können. Es ist nicht vorgesehen in diesem Layer in die Semantik der Daten einzugreifen. Diese werden lediglich in ein übertragbares Datenformat konvertiert.

3.5.6 Kommunikation

Dieser Layer stellt die eigentliche Datenübertragung dar. Darin kann sowohl die Anforderung A10 als auch A23 realisiert werden. Ähnlich dem Kopplungslayer ist auch hier keine parallele Ausführung dieser beiden Funktionen notwendig. Sie werden also in unterschiedlichen Implementierungen des Layers umgesetzt. Je nach Konfiguration kann die passende Implementierung eingesetzt werden. Das hat den Vorteil, dass die Art der Kommunikation auf die jeweiligen Anforderungen angepasst werden kann. Es aber keine parallele Kommunikationskanäle geben kann.

In einer Implementierung wird die Anforderung A10 umgesetzt, laut dieser die Daten mit wenig Kommunikationsoverhead übertragen werden sollen. Der geringste mögliche Aufwand entsteht indem die Daten über einen Funktionsaufruf übergeben werden. Eine weitere Implementierung realisiert die Anforderung A23 die besagt, dass eine Kommunikation zwischen Prozessen möglich sein soll. Der höhere Kommunikationsoverhead wird dabei in Kauf genommen. Beispielhaft wird eine Socket Kommunikation gewählt. Es besteht jedoch auch die Möglichkeit jede andere Art der Interprozesskommunikation zu implementieren.

Die Verbindungen der beiden Kommunikationslayer sind die einzigen Berührungspunkte der beiden Kommunikationsteilnehmer. Dadurch ist sichergestellt, dass jede Art der Kommunikation alle Layer durchlaufen muss. Das Interface zwischen diesen beiden Kommunikationslayern wird nicht näher in dem Modell spezifiziert. Das ermöglicht größtmögliche Flexibilität in der Auswahl des Übertragungsverfahrens. Wie genau dieses Interface aussieht wird erst durch die die Art des Übertragungsverfahrens bzw. dessen Implementierung definiert.

4. Implementation

Das Kapitel Implementierung beschreibt die Umsetzung des in Kapitel 3 definierten Kommunikationsmodels, sowie dessen Integration in die Simulationsumgebungen. Außerdem werden die Aktionen beschrieben, die nötig waren um die Decision Unit und die aktuelle Simulationsumgebung voneinander zu trennen.

4.1 Schnittstellenstruktur

Das in Kapitel 3.2 definierte Funktionsmodell wurde, wie in Abbildung 47 zu sehen ist, umgesetzt. Die Klasse *clsCommunicationInterface* kapselt die gesamte Schnittstellenlogik und stellt mit den Methoden *recvData()* und *sendData()* das Interface für die Anwendung zur Verfügung. Da das Schnittstellenmodell auch die Möglichkeit bietet die Anwendung durch die Schnittstelle anzustoßen wurde das Interface *itfCommunicationPartner* definiert über dessen Methode *recvData()* die Anwendung angestoßen werden kann. Auf Anwendungsseite dient nicht die Anwendung selbst als Kommunikationspartner für die Schnittstelle, sondern einer Kommunikationsport. Dieser implementiert das Interface *itfCommunicationPartner* und die Interfaces der Anwendungsseite. So besteht die Möglichkeit die Schnittstelle bestmöglich in die Anwendung integrieren zu können.

Das *clsCommunicationInterface* setzt sich aus den Interfaces (*itfLayer1*, *itfLayer2*, *itfLayer3* und *itfLayer4*) für die vier Layer (Definition der Layer siehe Kapitel 3.2.2) zusammen. Diese Interfaces können jeweils mit den benötigten Implementierungen gefüllt werden. Die in Tabelle 6 beschrieben Implementierungen wurden erstellt, und werden in Kapitel 4.2 noch näher ausgeführt.

Implementierung	**Layer**	**Funktion**
clsProcedureCall	Layer 1	Übergabe der Daten durch Methodenaufruf
clsXMLData	Layer 2	Erzeugen einer XML Struktur zur Datenübertragung
clsBufferContainer	Layer 3	Pufferung der Daten auf Empfangsseite
clsBlocking	Layer 4	Blockierende Kopplung
clsNonBlocking	Layer 4	Nicht blockierende Kopplung

Tabelle 6: Layer Implementierungen

Über die Klasse clsFactory besteht die Möglichkeit Konfigurationen der Schnittstelle zu erzeugen (siehe Kapitel 3.2.3). Die benötigten Konfigurationen für die ARS-Simulationsumgebung und den Miklas Simulator wurden erstellt. Die Factory Klasse kann jedoch erweitert werden um Konfigurationen für andere Umgebung zu ermöglichen.

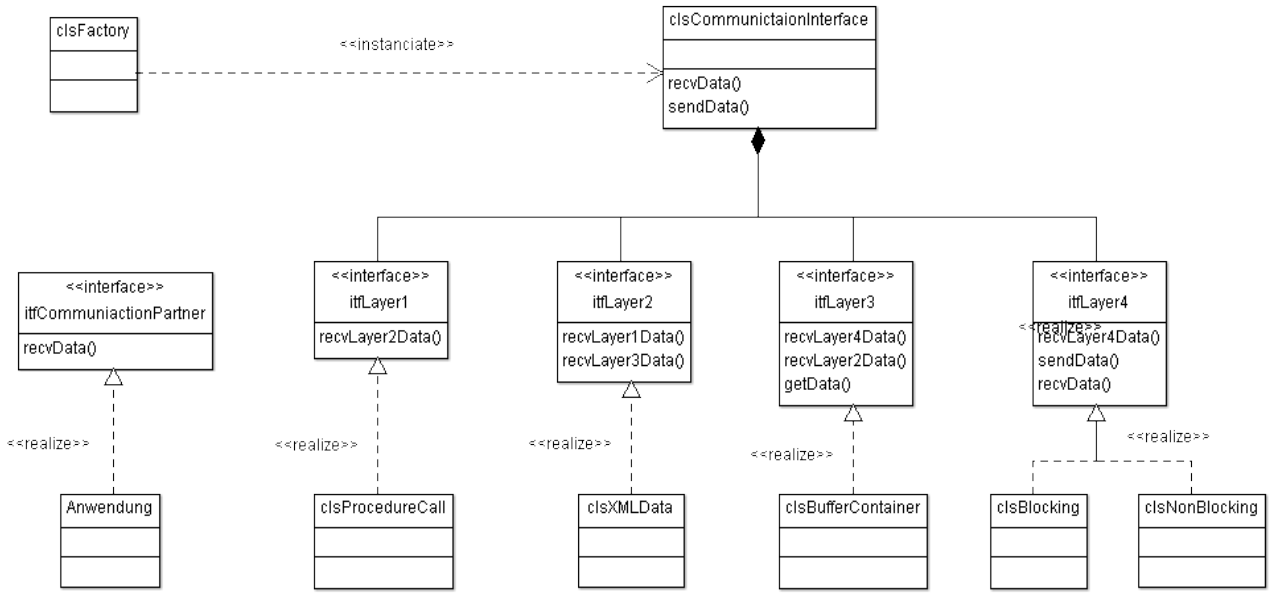

Abbildung 47: Implementierung Schnittstelle

4.2 Layer Implementierungen

In diesem Kapitel werden die einzelnen Implementierungen der Layer beschrieben. Diese Implementierungen wurden verwendet, um die Decision Unit in der ARS-Simulationsumgebung und dem Miklas Simulator einsetzen zu können. Durch die Struktur über Interfaces können jedoch beliebige Implementierungen erstellt werden, um möglichen Anforderungen in alternativen Umgebungen gerecht zu werden. Eine Auflistung der Layer findet sich in Tabelle 6.

4.2.1 clsProcedureCall

Bei diesem Layer handelt es sich um eine Implementierung des Interfaces *itfLayer1*. Der Layer ermöglicht, dass zwei Kommunikationspartner mit dem geringsten möglichen Overhead Informationen austauschen können. Es wurde der Methodenaufruf als Kommunikationsstrategie gewählt. Da es sich um eine Layer 1 Kommunikation handelt, werden die Daten direkt mit dem Layer 1 des anderen Kommunikationspartners ausgetauscht. Die Implementierung definiert die Funktion *recvData()*. Diese wird von der Gegenseite aufgerufen, um Daten zu übertragen. Der Rückgabewert der Funktion stellt dabei die Antwort des Kommunikationspartners dar.

4.2.2 clsXMLData

Die Klasse *clsXMLData* stellt eine Implementierung des Datenkonvertierungslayers dar. Dieser Layer ist für die Konvertierung der Übertragungsdaten in ein für die Übertragung geeignetes Datenformat verantwortlich. Als Datenformat für diese Implementierung wird XML verwendet, da sich die innerhalb der Schnittstelle verwendete assoziative Datenstruktur mit wenig Aufwand in XML konvertieren lässt, und dies mit vielen Datenübertragungstechniken vereinbar ist. Die Form der Konvertierung ist ein Abbildung 48 zu sehen. Es muss sowohl die Encodierung in Senderichtung als auch die Decodierung in Empfangsrichtung unterstützt werden.

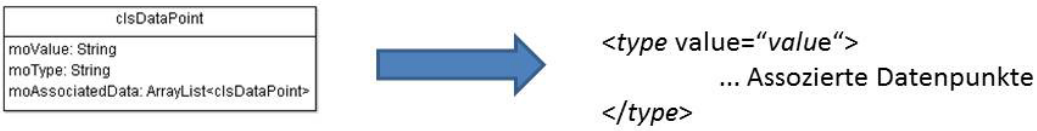

Abbildung 48: XML Struktur

Encodierung

Der Encodierungsschritt wandelt die zu sendenden Daten aus der Datenstruktur *clsDataPoint* in eine XML Struktur um. Die Membervariablen *value* und *type* der Klasse *clsDataPoint* werden dabei wie in Abbildung 49 zu sehen in das XML Datenformat umgewandelt. Assoziierte Datenpunkte werden als Kinderelemente innerhalb der Elterntags eingeschlossen.

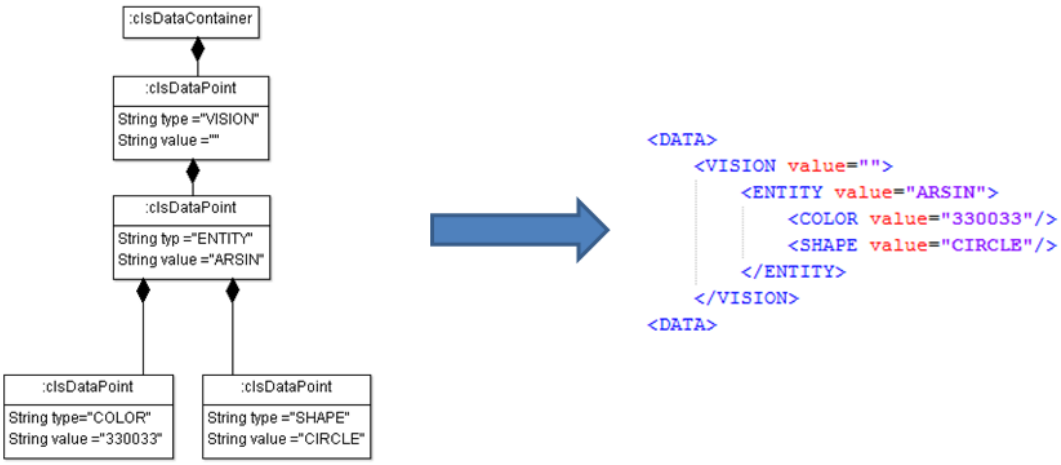

Abbildung 49: XML Codierung Beispiel

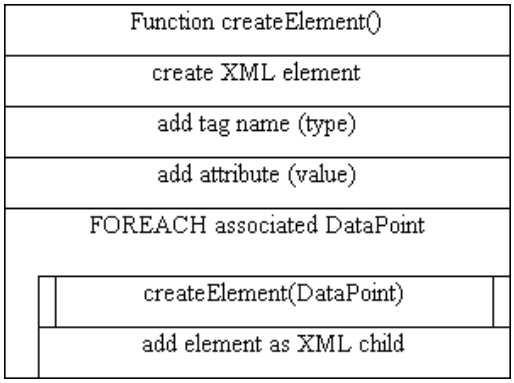

Abbildung 50: Struktogramm Kodierung

Die Abbildung 50 zeigt den implementierten, rekursiven Kodierungsalgorithmus. Die Funktion *createElement()* wird rekursiv für alle assoziierten Datenpunkte aufgerufen. Dadurch wird sichergestellt, dass der gesamte Baum an Datenpunkten konvertiert wird.

Decodierung

In umgekehrter Richtung wird der empfangene XML Baum in die *clsDataPoint* Datenstrukturumgewandelt. Dies erfolgt ebenfalls rekursiv mit der Funktion *createDataPoint()*. Der in Abbildung 51 dargestellt Algorithmus traversiert ähnlich der Encodierung durch den XML Baum, und liefert die Daten als assoziierte *clsDataPoints* zurück.

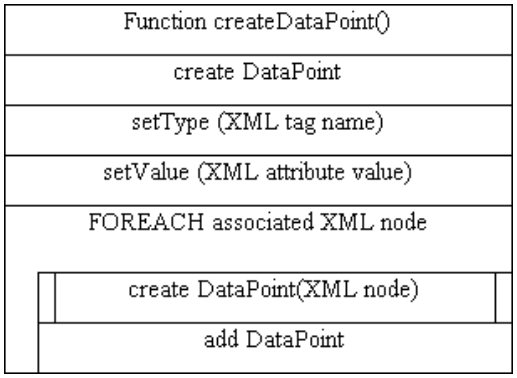

Abbildung 51: XML Dekodierung

4.2.3 clsBufferContainer

Die Klasse *clsBufferContainer* stellt eine Implementierung des Layer 3 Interfaces dar. Es werden lediglich empfangene Daten gepuffert. Gesendete Daten werden ungepuffert an den unteren Layer weitergereicht. In Abbildung 52 ist der Aufbau zu sehen. Der *clsBufferContainer* besteht aus einer Liste an *itfBuffer*. Diese können sowohl *clsEventBuffer* als auch *clsSignalBuffer* sein. Für jeden Typ

in der Liste an empfangenen Datenpunkten gibt es einen Puffer. Ist für einen Typ noch kein Puffer vorhanden, dann wird der Standardpuffer angelegt. Werden Datenpunkte empfangen, dann werden diese in die Puffer gespielt. Je nach Implementierung werden die Datenpunkte dann innerhalb des Puffers abgespeichert. Nachdem alle Datenpunkte in Puffern abgelegt wurden, wird der nächst höhere Layer über den Datenempfang informiert. Dieser kann dann je nach Implementierung die Daten aus den Puffern anfordern.

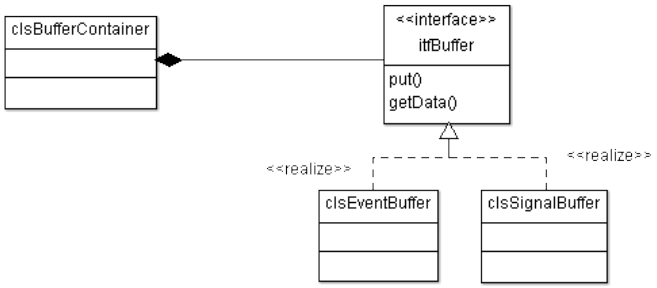

Abbildung 52: clsBufferContainer

clsEventBuffer

Die *put()* Methode des Puffers fügt die übergeben Daten zu einer Liste hinzu. Die *getData()* Methode gibt alle Elemente der Liste zurück, und löscht die Liste. Folgendes Beispiel soll diese Funktionalität veranschaulichen.

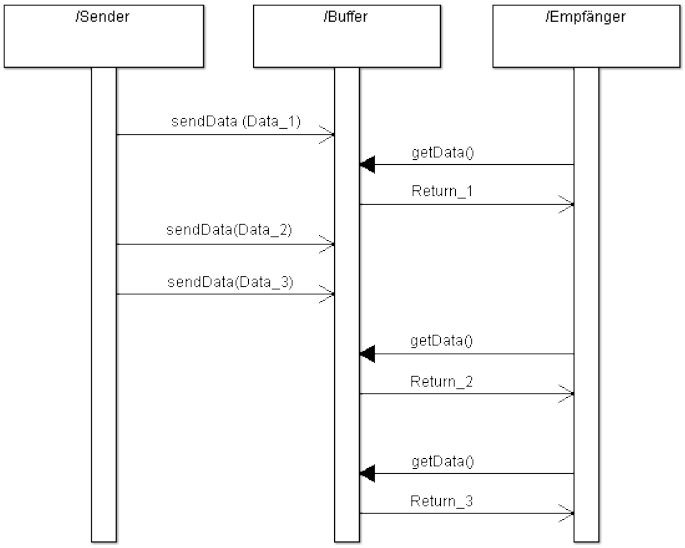

Abbildung 53: Puffer Anwendungsfall

Input	**Wert**
Data_1	Bloodsugar: 0.3
Data_2	Bloodsugar: 0.5

Data_3	Bloodsugar: 0.5

Tabelle 7: Inputdefinition clsEventBuffer

Output	**Wert**
Return_1	Bloodsugar: 0.3
Return_2	Bloodsugar: 0.5
Return_3	Bloodsugar: 0.5

Tabelle 8: Returnwerte clsEventBuffer

Die in Abbildung 53 dargestellt Sequenz wird mit den in Tabelle 7 beschrieben Input Werten ausgeführt. Dadurch ergeben sich die in Tabelle 8 beschriebenen Returnwerte. Dabei kann man erkennen, dass der Wert bei schreibendem Zugriff überschrieben wird und bei lesendem Zugriff erhalten bleibt.

clsSignalBuffer

Die *put()* Methode überschreibt das aktuelle Element aus dem Speicher. Die *getData()* Methode gibt das Element zurück. Es bleibt jedoch für weitere Anfragen im Speicher erhalten.

Input	**Wert**
Data_1	ActionCommand: EAT
Data_2	ActionCommand: MOVE_FOWARD
Data_3	ActionCommand: TURN_RIGHT

Tabelle 9: Inputdefinition clsSignalBuffer

Output	**Wert**
Return_1	ActionCommand: EAT
Return_2	ActionCommand: MOVE_FORWARD, TURN_RIGHT
Return_3	ActionCommand:

Tabelle 10: Returnwerte clsSignalBuffer

Hier dient ebenfalls die in Abbildung 53 dargestellte Sequenz als Beispiel. Es werden jedoch die Inputwerte aus Tabelle 9 herangezogen um den Output aus Tabelle 10 zu erhalten. Dabei ist zu erkennen, dass der schreibende Zugriff die Liste des Puffers erweitert und der lesende Zugriff die Liste löscht.

4.2.4 clsBlocking

Die Klasse *clsBlocking* implementiert das *itfLayer4*. Sie implementiert den blockierenden Funktionsaufruf. Dabei werden die vom darunter liegenden Layer empfangenen Daten an die nächste Schicht weitergeleitet. Der Rückgabewert wird erst dann erzeugt, wenn von der Schicht darüber ein Rückgabewert erhalten wurde. Das hat zur Folge, dass dieser Layer so lange blockiert, bis die Anwendung einen Antwort generiert und diese zurückgegeben hat.

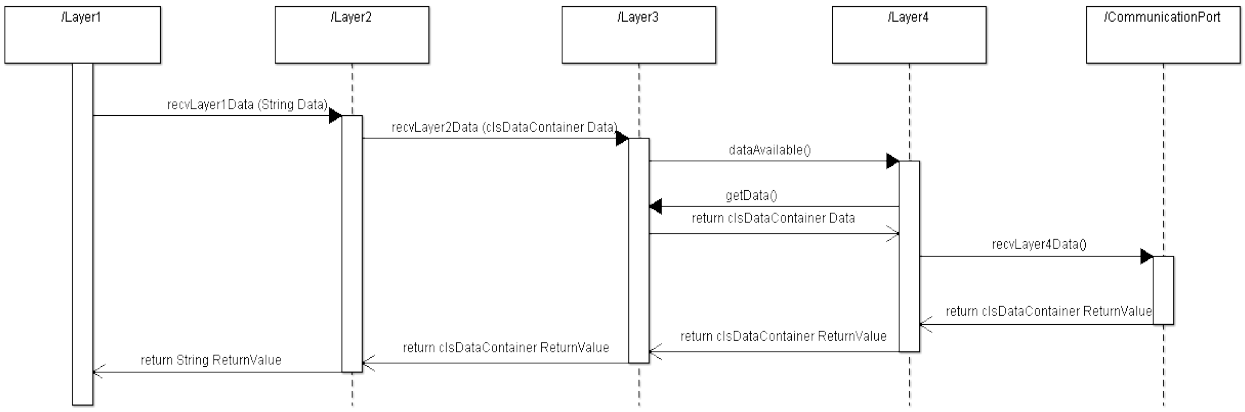

Abbildung 54: Kommunikationsablauf blockierend

Abbildung 54 zeigt die Reihenfolge der Funktionsaufrufe. Dabei ist zu erkennen, dass die Übergabe der Daten von Layer 3 auf Layer 4 einen Sonderfall darstellen. Die Daten werden nicht wie bei den anderen Schnittstellen durch geschachtelte Funktionsaufrufe übergeben. In diesem Fall holt der Layer 4 die Daten des Layers 3 ab. Dadurch ist sowohl die blockierende als auch die nicht blockierende Implementierung realisierbar.

4.2.5 clsNonBlocking

Die Klasse clsNonBlocking stellt eine Implementierung des itfLayer4 Interfaces dar. Dabei wird, zum Unterscheid zur clsBlocking Klasse, ein nicht blockierender Kommunikationsablauf umgesetzt. Abbildung 55 zeigt den Ablauf dieser nicht blockierenden Kommunikation. In der *dataAvailable()* Funktion von Layer 4 werden in diesem Fall nicht die Daten von Layer 3 abgeholt, sondern es wird sofort der Rückgabewert erstellt und zurückgegeben. Dabei kann es sich natürlich nicht um die Beantwortung der Anfrage handeln, da diese noch nicht bearbeitet werden konnte. Es wird lediglich ein Statuscode zurückgegeben. Die empfangenen Daten können unabhängig von der Datenübertragung vom Empfänger abgeholt werden. Der Empfänger kann die Daten bearbeiten und die Antwort in gleicher Weise zurücksenden. Während dieser Zeit wird der Sender nicht blockiert und kann seine Ausführung fortsetzen.

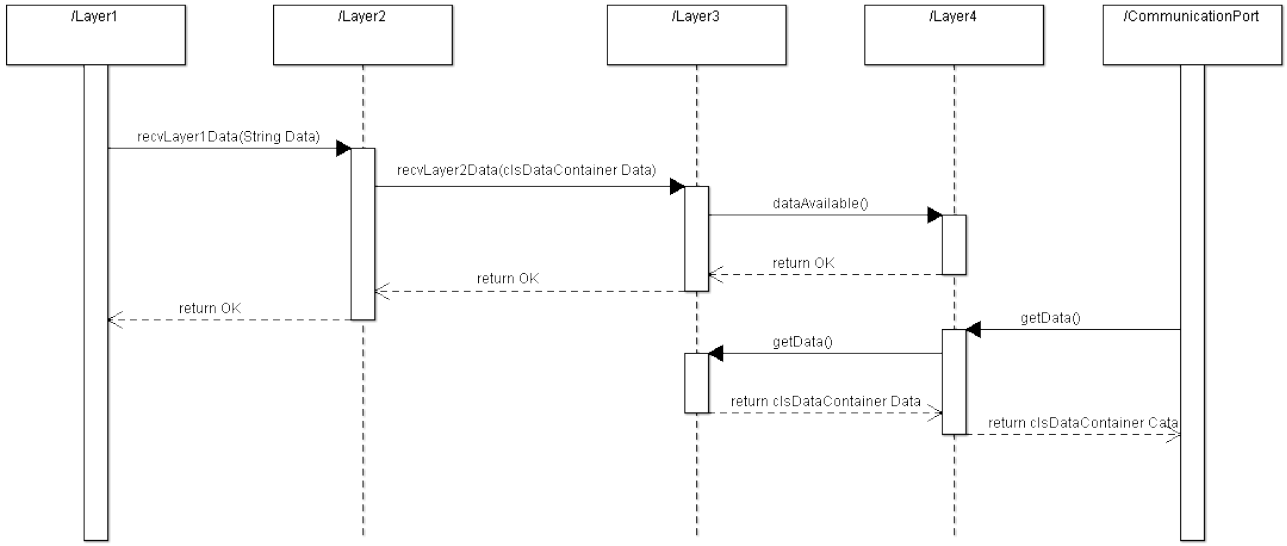

Abbildung 55: Kommunikationsablauf nicht blockierend

4.3 Decision Unit und Simulationsumgebung trennen

Wie in Kapitel 2 festgestellt, besteht zwischen der Decision Unit und der Simulationsumgebung eine enge Vernetzung. Bevor die in Kapitel 4.2 beschriebene Implementierung mit der Decision Unit verbunden werden kann, muss diese aus der aktuellen Simulationsumgebung herausgelöst werden.

4.3.1 Decision Unit Interface

Das Decision Unit Interface definiert wie die Decision Unit innerhalb der Simulation angesprochen werden kann. Durch die Verwendung von Interfaces ist es möglich, die Definition der Schnittstelle von der Implementierung zu trennen. Dabei wird nur das Interface in die Anwendung integriert. Die Implementierung kann je nach Verwendungszweck ausgetauscht werden. Bei der bestehenden Implementierung waren durch dieses Interface nicht nur die Referenzierung der Decision Unit, sondern auch die Kommunikation mit dem Körper und die Ausführungssteuerung definiert. Im linken Teil der Abbildung 56 ist die bestehende Interface Struktur dargestellt. Die Methode *process()*dient dabei zur Ablaufsteuerung und die Methode *update()* zur Übergabe der Sensordaten. Die Aktionsbefehle werden in diesem Interface nicht an den Körper übergeben. Die Decision Unit übernimmt direkt die Steuerung der Action-Engines. Das Interface *itfDecisionUnit* wird innerhalb des Körpers verwendet. Es besteht also auch die Möglichkeit andere Decision Unit Implementierung mit dem Körper zu verbinden.

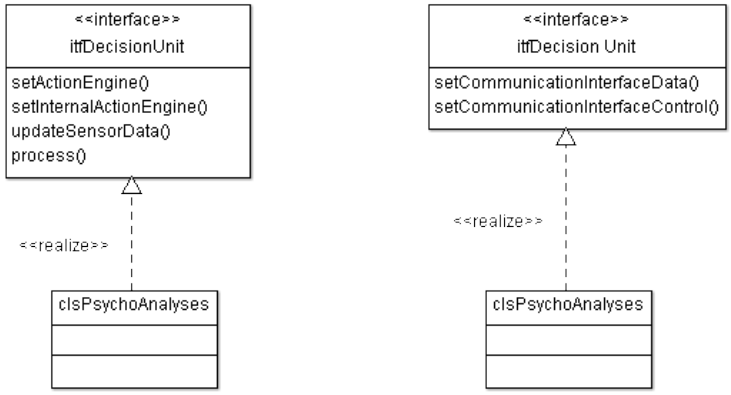

Abbildung 56: Decision Unit Interface

Im rechten Teil der Abbildung 56 ist das veränderte neue Decision Unit Interface zu sehen. Die einzigen Methoden die noch zu Verfügung gestellt werden, sind Methoden zum Setzen der beiden Kommunikationschnittstellen. Die Möglichkeit Sensordaten, Aktionsbefehle und Ablaufsteuerungsinformationen auszutauschen, wird über die beiden Kommunikationschnittstellen zur Verfügung gestellt.

4.3.2 Action Engine

Die Verschränkung zwischen Simulationsumgebung und Decision Unit wird besonders bei der Action Engine deutlich. Diese ist innerhalb des Körpers für die Ausführung von Aktionsbefehlen verantwortlich [Dön09]. Innerhalb der Ausführung dieser Befehle muss auf alle Bereich des Körpers zugegriffen werden können. Es werden sowohl Körperinterne Systeme als auch die Visualisierung oder die physikalischen Eigenschaften des Agenten bearbeitet. Man sieht also, dass die Action Engine stark in den Körper integriert ist, und somit eine starke Abhängigkeit dieser beiden Komponenten besteht. Die Implementierung des Körpers wird wiederum sehr stark von der Ausführungsumgebung und deren Eigenschaften definiert. Die Implementierung der Action Engine ändert sich also je nach Ausführungsumgebung. Daraus folgt, dass die Verbindung zwischen Decision Unit und Action Engine möglichst lose sein muss, um die Decision Unit nicht zu beeinflussen.

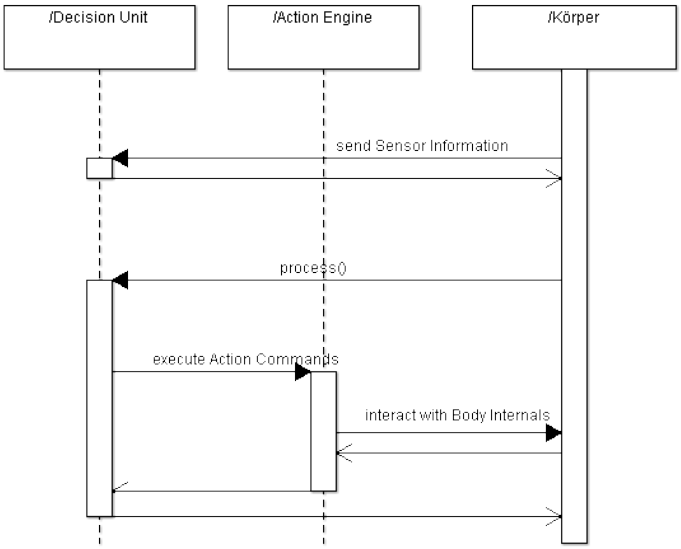

Abbildung 57: Bisherige Einbindung der Action Engine

In Abbildung 57 ist der Kommunikationsablauf zwischen Körper, Decision Unit und Action Engine dargestellt. Es ist zu erkennen, dass die Decision Unit durch eine Referenz auf die Action Engine die Ausführung der Aktionen übernimmt. Diese Referenz auf die Action Engine würde jedoch dem Prinzip der in 3.2 definierten Schnittstelle widersprechen. Außerdem wäre die Unabhängigkeit der Decision Unit von der Simulationsumgebung nicht mehr gegeben.

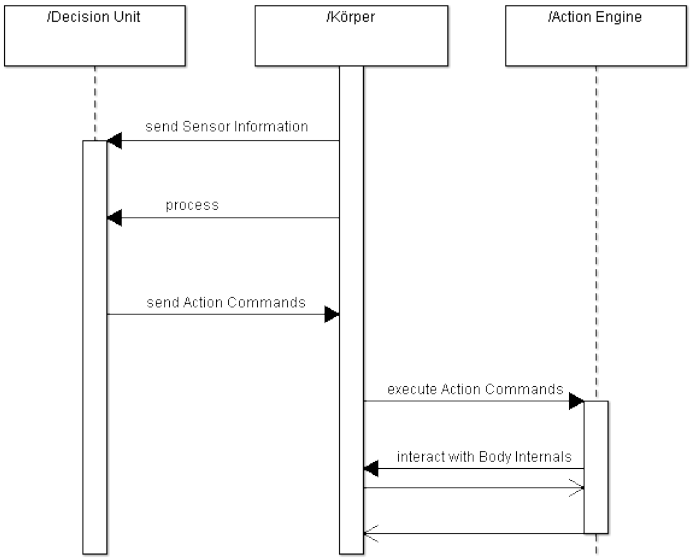

Abbildung 58: Neue Einbindung der Action Engine

Dadurch ist es notwendig die Action Engine von der Decision Unit zu lösen, und an den Körper zu koppeln. Abbildung 58 zeigt die neu definiert Kommunikationssequenz. Die Kontrolle der Action Engine wird dabei komplett an den Körper übergeben. Dadurch lässt sich diese, ohne die Decision

Unit zu verändern, auf die Anforderungen des Körpers (bzw. der Ausführungsumgebung) anpassen. Die Kommunikation erfolgt komplett über die definierte Datenschnittstelle zwischen Decision Unit und Körper. Dabei werden die Aktionsbefehle an den Körper übergeben und dieser steuert anschließende die Ausführung der Action Engine.

4.4 Anbindung der Schnittstelle an die Decision Unit

Nach der Implementierung der Schnittstelle kann diese nun verwendet werden um die Kommunikation zwischen Simulationsumgebung und Decision Unit zu realisieren. Zuerst muss die Schnittstelle an die Decision Unit angekoppelt werden (Kapitel 4.4). Im Anschluss kann der offenen Teil der Schnittstelle mit den jeweiligen Simulationsumgebungen verbunden werden (Kapitel 4.5 und 4.6).

4.4.1 Kommunikationsports

Der Kommunikationsport stellt eine Implementierung des in Kapitel 3.5.2 beschriebenen Konzeptes dar. Dieser implementiert sowohl das von der Kommunikationsschnittselle definierte Interface, als auch die Anbindung an die Decision Unit. In dieser Implementierung werden weder die Struktur, noch der Inhalt der Daten in irgendeiner Weise verändert. Die geforderte Struktur der Daten muss von der Anwendung, in die die Decision Unit integriert werden soll, geliefert werden. Der Kommunikationsport hat in diesem Fall die Aufgabe, sich in die Methodenaufrufe der Decision Unit einzufügen.

Wie in Kapitel 3.2 wird die Kommunikation der Decision Unit in zwei Teile aufgeteilt. Daher werden auch zwei Kommunikationsports mit unterschiedlichen Aufgaben benötigt. Abbildung 59 zeigt das Zusammenspiel der beiden Kommunikationsports, der Kommunikationsschnittstellen und der Decision Unit. Der *clsCommunicationPortControl* wird über neue Ausführungsbefehle informiert. Dieser führt anschließend einen Schritt der Decision Unit aus. Über den *clsCommunicationPortData* werden innerhalb der Ausführung sowohl die Sensordaten abgeholt, als auch die Aktionsbefehle an den Körper übermittelt.

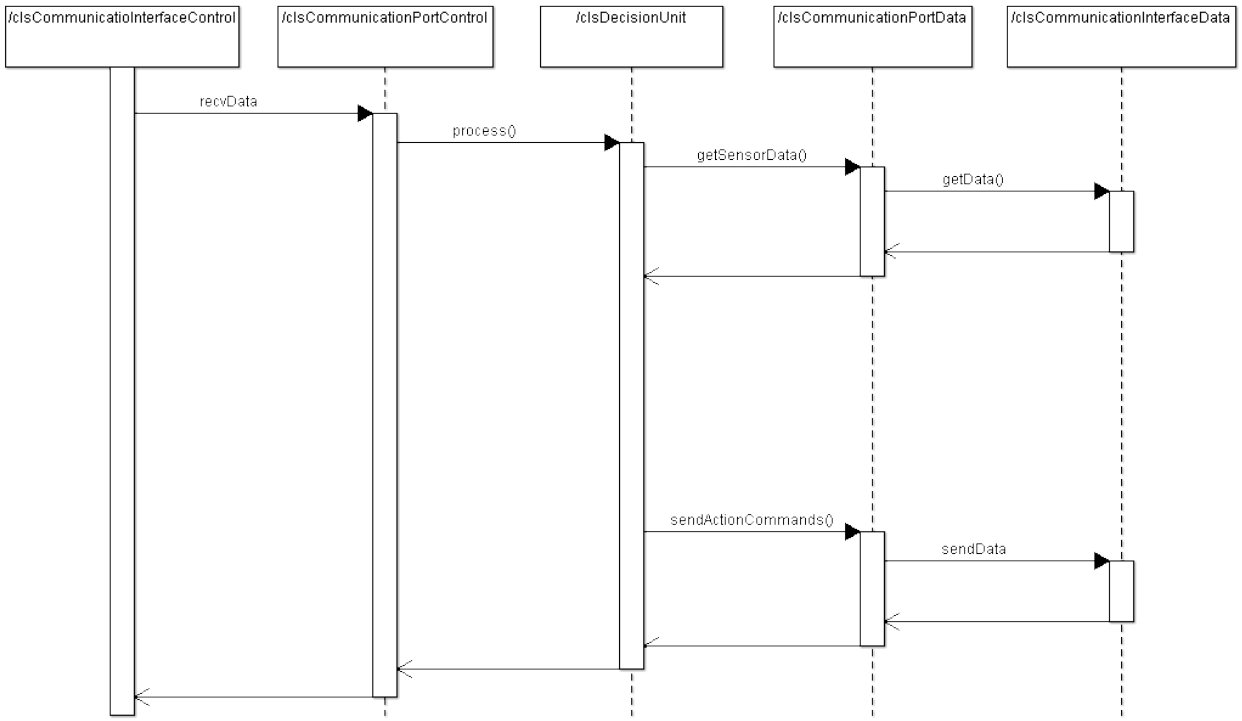

Abbildung 59: Kommunikationspots der Decision Unit

4.4.2 Integration in die Decision Unit

Vor der Umstellung der Decision Unit auf das in Kapitel 3 definierte Kommunikationsmodell wurden innerhalb der Dateninterfaces die Datentypen des Körpers verwendet. Da es durch das Komunikationsmodell zu einer klaren Trennung dieser beiden Funktionsblöcke gekommen ist, mussten diese Interfaces auf die neuen Datentypen angepasst werden. Die Eingangs- bzw. Ausgangsinterfaces (I0.1, I0.2, I0.3, I0.4, I0.5 und I0.6 siehe Abbildung 60) wurden auf den Typ *clsDataContainer* geändert. Somit ist es möglich, die Daten aus der Kommunikationsschnittstelle direkt mit dem Funktionsmodell zu verbinden.

Abbildung 60 zeigt die ersten beiden Layer des Funktionsmodelles. Die Module F39, F1, F10, F12 und F32 im neuronalen Layer haben in der aktuellen Implementierung der Decision Unit keine Funktion. Es werden lediglich die Eingangsdaten an das nächste Modul weitergegeben. Aus diesem Grund wurden die Interfaces zum Neuro-Symbolischen Layer (I1.1, I1.2, I1.3, I1.4 und I1.5)ebenfalls auf den Typ *clsDataContainer* geändert. In den Modulen F40, F2, F11 und F13 im Neuro-Symbolischen Layer erfolgt die Konvertierung der Datenstruktur *clsDataContainer* in die Datenstruktur der jeweiligen Neurosymbole. In F31 wird die Neuro-Symbolische Beschreibung der Aktionsbefehle in die Datenstruktur *clsDataContainer* umgewandelt, um diese über die Kommunikationsschnittstelle an den Körper senden zu können.

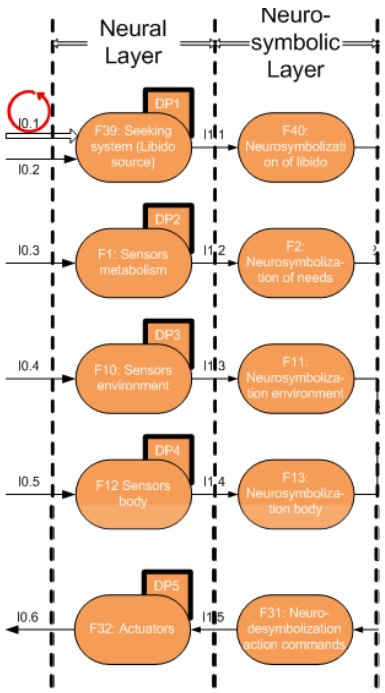

Abbildung 60: Interfaces Decision Unit

4.5 Anbindung der Schnittstelle an die ARS-Simulationsumgebung

In ähnlicher Weise wie die Kommunikationschnittstelle mit der Decision Unit verbunden ist, ist diese auch mit der Simulationsumgebung verbunden. Auch hier werden Kommunikationsports eingesetzt, um vom Interface der Schnittstelle zu abstrahieren.

4.5.1 Kommunikationsports

Auf der Seite der Simulationsumgebung zeigt sich der Grund, warum es nötigt ist die Schnittstelle auf zwei Teile aufzuteilen. Der Datenteil der Schnittstelle ist mit dem Körper des Agenten verbunden, und stellt die Übertragung der Sensor und Aktor Information zu Verfügung. Der Kontrollteil der Schnittstelle ist mit dem MASON Scheduler verbunden und überträgt die Steuerkommandos an die Decision Unit.

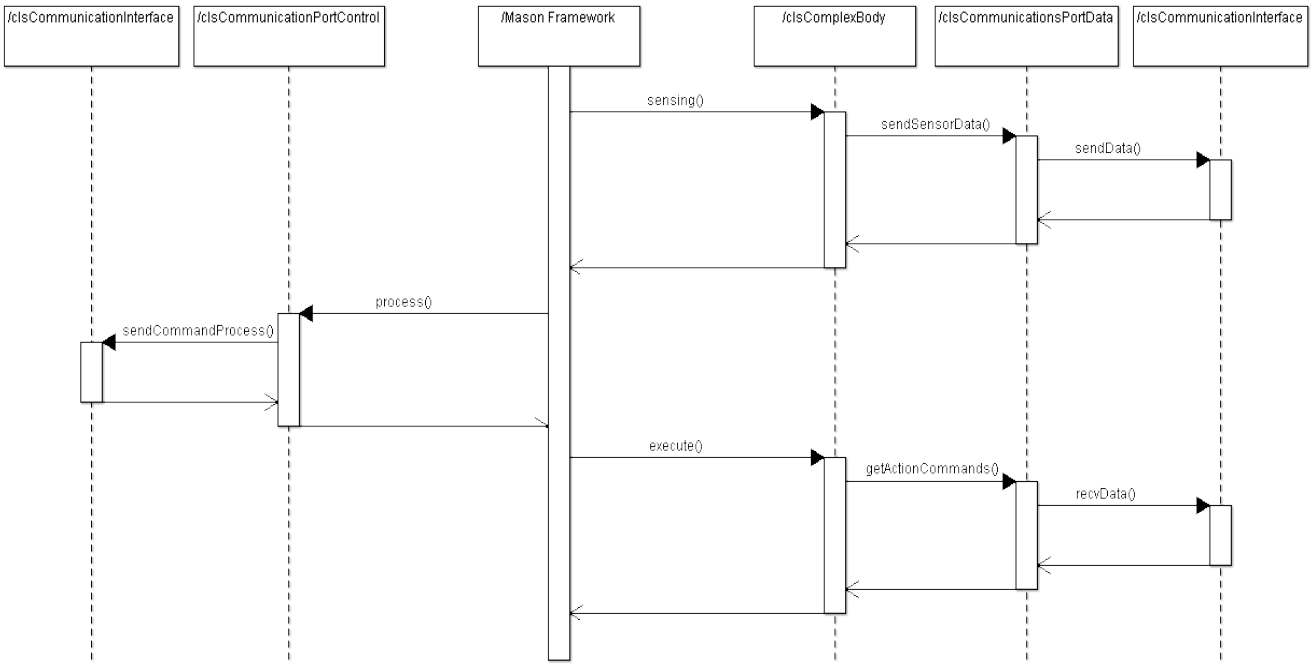

Abbildung 61: Kommunikationsports Simulationsumgebung

Abbildung 61 zeigt den Ablauf innerhalb eines Simulation Schrittes. Der Scheduler innerhalb des MASON Frameworks ist wie in 2.2.2 beschrieben für die Abarbeitungsreihenfolge verantwortlich. Zuerst wird der Sensorteil des Körpers ausgeführt, der über den *clsCommunicationPortData* Sensor Informationen an die Decision Unit sendet. Anschließend wird über den *clsCommunicationPortControl* die Ausführung der Decision Unit angestoßen. Die Aktionsbefehle werden wieder über den *clsCommunicationPortData* abgeholt.

4.5.2 Integration in den Körper

Die im Körper verwendeten Daten müssen in den Kommunikationsports in den Datentyp *clsDataContainer* konvertiert werden, um diese übertragen zu können. Die Struktur der Daten wird von der Decision Unit vorgegeben. Da diese für bestimmte Ausführungsumgebung nicht mehr angepasst werden soll, muss eine Umwandlung der Daten bereits im Kommunikationsport der Anwendung erfolgen. Da die Simulationsumgebung jedoch parallel zur Decision Unit entwickelt wurde sind die von ihr gelieferten Daten mit den Anforderungen der Decision Unit zum jetzigen Zeitpunkt ident. Durch eine Weiterentwicklung einer der beiden Funktionsblöcke könnte jedoch eine Anpassung an dieser Stelle erforderlich werden.

4.6 Anbindung der Schnittstelle an Miklas

Die Simulationsumgebung Miklas (Kapitel 2.3) stellt über das Interface *ExternalMindControlInterface* eine Schnittstelle zur Einbindung von externen Decision Units zur Verfügung. Dazu wurde eine Klasse *ARSMind* eingeführt, die das Interface implementiert und über zwei Kommunikationsports die Verbindung zur ARS Decision Unit herstellt.

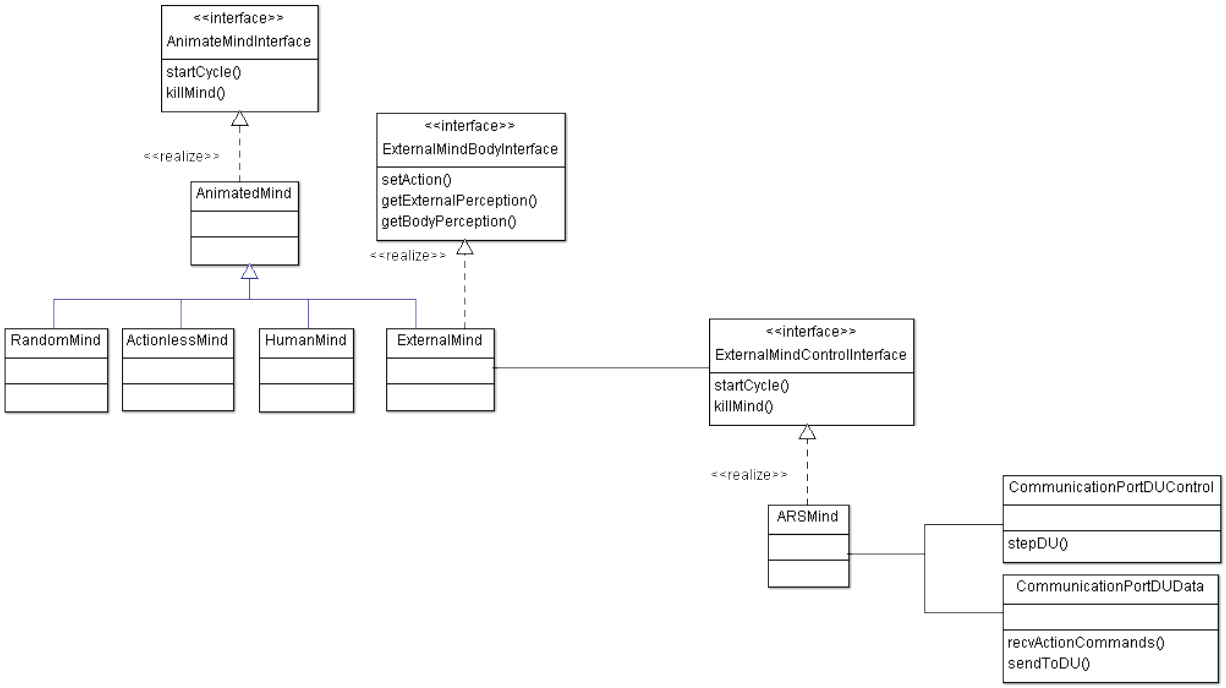

Abbildung 62: Miklas Integration

Abbildung 62 zeigt dabei den Aufbau der Miklas Simulationsumgebung und die Integration der ARS Decision Unit über die Klasse *ARSMind*. Ganz allgemein wird ein Interface *AnimatedMindInterface* zu Einbindung von Decision Units zur Verfügung gestellt. Für die Integration von externen Decision Units ist jedoch das Interface *ExternalMindControlInterface* relevant. Über diese werden externe Decision Unit Implementierungen angesprochen. Zum Unterschied zur ARS-Simulationsumgebungen müssen in Miklas die Struktur der Daten auf die Simulationsumgebung angepasst werden. Abbildung 63 zeigt die Datenstruktur die verwendet wird, um externe Decision Units mit Daten aus der Simulationsumgebung zu versorgen. Es können drei Typen von Daten ausgetauscht werden. Daten der visuellen Wahrnehmung werden über das *ExternalPerceptionInterface* ausgetauscht. Je ein Objekt in der Wahrnehmung wird als ein Objekt des Typs *ExternalPerceptionInterface* repräsentiert. Es können also beliebig viele *ExternalPerceptionInterface* Objekt mit *ExternalMindBodyInterface* verknüpft werden. Für jedes Objekt der Wahrnehmung werden die relative Position, sowie Informationen über den Körper und die Art des Objektes zur Verfügung gestellt. Für Körper bezogene Daten wird das *BodyPerceptionInterface* verwendet. Es ist ein Wert für Health und

für Pain abrufbar. Dem dritten Typen der Datenübertragung stellen die Aktionsbefehle dar. Mit Hilfe der Funktion *setAction()* können diese an den Körper übergeben werden, der die dazugehörenden Aktionen ausführt. Die beschriebene Datenstruktur muss auf die in Kapitel 2.1.3 beschrieben Interface Struktur angepasst werden. Dies geschieht im *CommunicationPortDUData.*

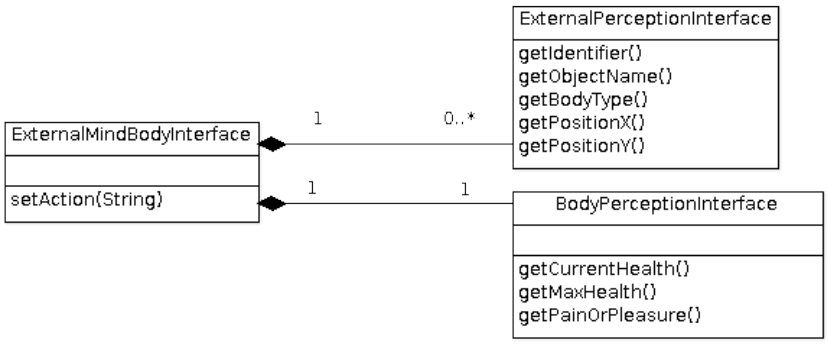

Abbildung 63: Body Mind Interface Miklas

5. Simulation und Ergebnisse

In der in Kapitel 1.4 beschrieben Vorgehensweise wurde definiert, dass die Richtigkeit des entwickelten und implementierten Schnittstellenmodells in einem vierstufigen Prozess evaluiert werden soll. Die vier Stufen des Testprozesses werden den Stufen der Entwicklung gegenübergestellt und evaluieren diese. Der Implementierungstest stellt die erste Teststufe dar. Dabei handelt es sich um Softwaretests die direkt während der Implementierung der einzelnen Softwarekomponenten erfolgen. Durch den Implementierungstest kann die Richtigkeit der Software auf unterster Ebene sichergestellt werden. Eine genaue Beschreibung allen durchgeführten Implementierungstest würde den Rahmen dieser Arbeit sprengen.

5.1 Komponententest

Der Komponententest evaluiert die Richtigkeit der in Kapitel 3.5 definierten Komponenten. Dazu werden die einzelnen Komponenten als Blackbox betrachtet in eine Testumgebung eingebettet und gegen ihre Spezifikation getestet.

5.1.1 Testaufbau

Abbildung 64 zeigt den prinzipiellen Aufbau für den Test einer Komponente. Dabei stellt die UUT (Unit Under Test) die zu testenden Komponente dar. Innerhalb der Testumgebung werden die benötigten Inputs erzeugt und an die UUT angelegt. Die daraus folgenden Outputs werden ausgewertet und mit der Spezifikation verglichen.

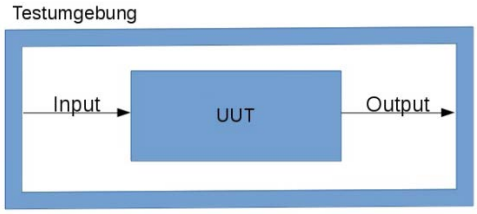

Abbildung 64: Testumgebung

5.1.2 Komponente Kopplung Synchron

Bei der Komponente Kopplung Synchron (Kapitel 4.2.4) handelt es sich um eine Implementierung des Kopplungslayer (Kapitel 3.5.3). Als Ergebnis des Testes soll gezeigt werden, dass die Ausführung des Layers 3 (und somit in weiterer Folge auch die Ausführung aller darunter liegenden Layer) solange blockiert wird, bis die Komponente eine Antwort gesendet hat. Abbildung 65 zeigt die Ausgaben während der Testausführung. Die Funktionalität und Schnittstelle von Layer3 und Layer4 werden in diesem Fall von der Testumgebung nachgebildet. Es ist zu sehen, dass die Ausführung von Layer 3 erst fortgesetzt wird, wenn die Antwort auf die Anfrage erzeugt und zurückgegeben wurde.

```
[Layer3][TRACE] - Daten an UUT gesendet
[UUT][TRACE] - Daten empfangen
[UUT][TRACE] - Daten weitegesendet
[Layer4][TRACE] - Daten empfangen und zurückgesendet
[UUT][TRACE] - Daten empfangen
[UUT][TRACE] - Daten weitegesendet
[Layer3][TRACE] - Antwort empfangen
```

Abbildung 65: Testergebnis Kopplung Synchron

5.1.3 Komponente Kopplung Asynchron

Die Komponente Kopplung Asynchron (Kapitel 4.2.5) ist ebenfalls eine Implementierung des Kopplungslayers (Kapitel 3.5.3). Es soll gezeigt werden, dass der darunter liegende Layer nicht blockiert wird bis die Antwort generiert wurde. Er soll seine Ausführung fortsetzen und zu einem späteren Zeitpunkt die Antwort entgegen nehmen können. Der Ablauf des Testes ist analog zu 5.1.2. In Abbildung 66 ist das Ergebnis des Testes zu sehen. Man kann erkennen, dass nach dem Empfang der Daten in der UUT die Ausführung von Layer 3 fortgesetzt wird. Die Antwort wird ganz am Ende des Testes zu einem späteren Zeitpunkt übergeben

```
[Layer3][TRACE] - Daten an UUT gesendet
[UUT][TRACE] - Daten empfangen
[Layer3][TRACE] - Ausführung fortgesetzt
[UUT][TRACE] - Daten weitegesendet
[Layer4][TRACE] - Daten empfangen und zurückgesendet
[UUT][TRACE] - Daten empfangen
[UUT][TRACE] - Daten weitegesendet
[Layer3][TRACE] - Daten empfangen
```

Abbildung 66: Testergebnis Kopplung Asynchron

5.1.4 Komponente Datenpuffer

Die Komponente Datenpuffer (Kapitel 4.2.3) ist eine Implementierung des Layers Datenpuffer (Kapitel 3.5.4). Bei dem Test werden zwei Konfigurationen unterschieden, die beide parallel in dieser Komponente existieren und je nach empfangenem Datum aufgerufen werden.

Eventpuffer

Der Eventpuffer wird für Daten verwendet, die nur einmalige Gültigkeit haben und konsumierend ausgelesen werden sollen. Außerdem soll es möglich sein mehrere Daten zwischen zu speichern und diese auf einmal abzufragen. In Abbildung 67 ist das Ergebnis des Testes zu sehen. Im ersten Teil des Testes kann gezeigt werden, dass durch mehrmaliges Senden von Daten diese zusammengefügt werden. Im zweiten Teil ist zu sehen, dass das Abfragen der Daten konsumierend erfolgt.

```
[Layer2][TRACE] - Sende - 'Eins'
[UUT][TRACE] - Daten empfangen
[Layer2][TRACE] - Sende - 'Zwei'
[UUT][TRACE] - Daten empfangen
[Layer4][TRACE] - Daten abholen
[Layer4][TRACE] - Empfangene Daten: Eins, Zwei
[Layer4][TRACE] - Daten abholen
[Layer4][TRACE] - Empfangene Daten:
```

Abbildung 67: Testergebnis Eventpuffer

Signalpuffer

Im Signalpuffer werden gespeicherte alte Daten von neueren überschrieben, diese werden jedoch durch ein Abfragen der Daten nicht verbraucht, und bleiben für spätere Zugriffe gespeichert. Abbildung 68 zeigt den Ablauf des Tests. Das gespeicherte Datum „Eins“ wird von dem neueren Datum „Zwei“ überschrieben. Dieses bleibt auch bei mehrmaligen Abfragen gespeichert.

```
[Layer2][TRACE] - Sende - 'Eins'
[UUT][TRACE] - Daten empfangen
[Layer2][TRACE] - Sende - 'Zwei'
[UUT][TRACE] - Daten empfangen
[Layer4][TRACE] - Daten abholen
[Layer4][TRACE] - Empfangene Daten: Zwei
[Layer4][TRACE] - Daten abholen
[Layer4][TRACE] - Empfangene Daten: Zwei
```

Abbildung 68: Testergebnis Signalpuffer

5.1.5 Komponente XML-Datenkonvertierung

Bei der Komponente XML-Datenkonvertierung (Kapitel 4.2.2) handelt es sich um eine Implementierung des Layers 2 (Kapitel 3.5.5). Die Empfangenen Daten von Layer 3 sollen für die Übertragung in ein serialisierbares Format konvertiert werden. Diese Konvertierung muss in beide Richtungen möglich sein wobei durch Konvertierung und Rückkonvertierung wieder das Ausgangsdatum entstehen soll. Abbildung 69 zeigt das Ergebnis des Tests. Es ist zu sehen, dass die Inputdaten im Format *clsDataContainer* in XML konvertiert und wieder zurück konvertiert werden. Am Ende des Tests steht wieder das Ausgangsdatum.

```
[Layer3][TRACE] - Sende: VISION - ENTITY:ARSIN - COLOR:RED; SHAPE:CIRCLE; ALIVE:TRUE
[UUT][TRACE] - Daten empfangen
[UUT][TRACE] - Daten in XML String konvertiert
[Layer1][TRACE] - Daten empfangen: <VISION> <ENTITY value=ARSIN> </COLOR value=RED> </SHAPE value=CIRCLE> </ALIVE value=TRUE> </ENTITY> </VISION>
[Layer1][TRACE] - Daten zurückgesendet
[UUT][TRACE] - Daten empfangen
[UUT][TRACE] - XML String in clsDataContainer konvertiert
[Layer3][TRACE] - Empfangen: VISION - ENTITY:ARSIN - COLOR:RED; SHAPE:CIRCLE; ALIVE:TRUE
```

Abbildung 69: Testergebnis Datenkonvertierung

5.1.6 Komponente Datenübertragung

Die Komponente Datenübertragung (Kapitel 4.2.1) stellt eine Implementierung der Layer 1 (Kapitel 3.5.6) dar. Die Aufgabe der Komponente ist es die eigentliche Datenübertragung durchzuführen. Abbildung 70 zeigt das Ergebnis des Tests. Die Daten werden in der Simulationsumgebung vom Layer 2 geniert und an die UUT übergeben. Diese sendet die Daten an den Kommunikationspartner.

```
[Layer2][TRACE] - Sende: 'Eins'
[UUT][TRACE] - Daten empfangen
[UUT][TRACE] - Sende Daten an Kommunikationspartner
[Communication Partner][TRACE] - Daten empfangen
```

Abbildung 70: Testergebnis Datenübertragung

5.2 Systemtest

Der Systemtest betrachtet das System als ganzes und verwendet als Evaluierungskriterium die in Kapitel 3.1 erarbeitetet Anforderungen. Aus diesen Anforderungen wurden Testfälle erarbeiten, die im folgenden Kapitel beschrieben werden. Jeder Testfall ist einer oder mehreren Anforderungen zugeordnet und hat die Aufgabe diese zu evaluieren. Zusätzlich zu den Anforderungen werden für jeden Testfall ein erwartetes Ergebnis sowie eine Ausführungsbeschreibung definiert. Diese Testfälle wurden wie in Kapitel 1.4 vorgesehen parallel zu den Anforderungen definiert um ein späteres Evaluieren dieser zu ermöglichen.

5.2.1 Testfall 1 – Kontroll- und Datenfluss

Anforderungen	A01 – Trennen von Kontroll- und Datenfluss
Ausführungsbeschreibung	Die Decision Unit wird sowohl mit dem MASON Scheduler als auch mit dem Körper des Agenten verbunden. Der Scheduler gibt den Ausführungstakt vor und der Körper liefert Sensordaten und empfängt Aktionsbefehle.
Erwartetes Ergebnis	Die Ausführung der Decision Unit wird unabhängig von den Input Daten getriggert. Das Empfangen von Daten löste keine Ausführung des Funktionsmodelles aus.

Tabelle 11: Testfall 1

Der im Testfall 1 erzeugten Logging Daten sind in Abbildung 71 zu sehen. Die ersten drei Logging Einträge zeigen an, dass Sensordaten über die Datenschnittstelle an die Decision Unit gesendet worden sind. Diese Daten wurden zwar von der Decision Unit empfangen, die Ausführung dieser wurde jedoch nicht ausgelöst. Erst durch das, über die Kontrollschittstelle gesendete, Kommando *PROCESS* wurde die Ausführung der Decision Unit ausgelöst. Weiters ist zu sehen, dass nach dem Abarbeiten der Funktionsmodule die errechneten Aktionsbefehle an den Körper gesendet werden.

```
[BrainSocket][TRACE] (clsBrainSocket.java:136) - Send Sensor Data to Decision Unit
[BrainSocket][TRACE] (clsBrainSocket.java:136) - Send Sensor Data to Decision Unit
[BrainSocket][TRACE] (clsBrainSocket.java:136) - Send Sensor Data to Decision Unit
[BrainSocket][TRACE] (clsBrainSocket.java:129) - Send Control Command PROCESS
[DecisionUnit][TRACE] (clsPsychoAnalysis.java:142) - Start Decision Unit Process
[DecisionUnit][TRACE] (clsPsychoAnalysis.java:147) - Send Action Commands
```

Abbildung 71: Logging Output Testfall 1

5.2.2 Testfall 2 - Datenstruktur

Anforderungen	A02 – Einhalten der Erwarteten Datenstruktur
Ausführungsbeschreibung	Es werden mehrmals hintereinander Input Daten an die Decision Unit gesendet, ohne die Ausführung dieser zu triggern. Danach wird mehrmals hintereinander die Decision Unit ausgeführt ohne neue Inputdaten zu erhalten.
Erwartetes Ergebnis	Während des ersten Teiles des Test wird, trotz mehrmaligem senden der Daten, die Struktur eingehalten, und die mehrfache Empfangenen Daten zusammengeführt. Im zweiten Teil des Testes bleibt die Datenstruktur erhalten obwohl die Decision Unit ohne frische Daten zu bekommen, mehrmals ausgeführt wird.

Tabelle 12: Testfall 2

Um die Übersichtlichkeit zu erhalten, wird das Ergebnis dieses Testfalles anhand der Daten der Vision Sensoren gezeigt. Abbildung 72 zeigt einen Ausschnitt aus den Inspektoren in F10. Da in diesem Modul die Daten aus den externen Sensoren zum ersten Mal innerhalb des Funktionsmodelles verarbeitet werden, können diese mit Hilfe von Inspektoren grafisch dargestellt werden. Die nach dem ersten und nach dem zweiten Teil des Testes abgegriffenen Daten sind ident und stimmen mit den gesendeten Daten überein.

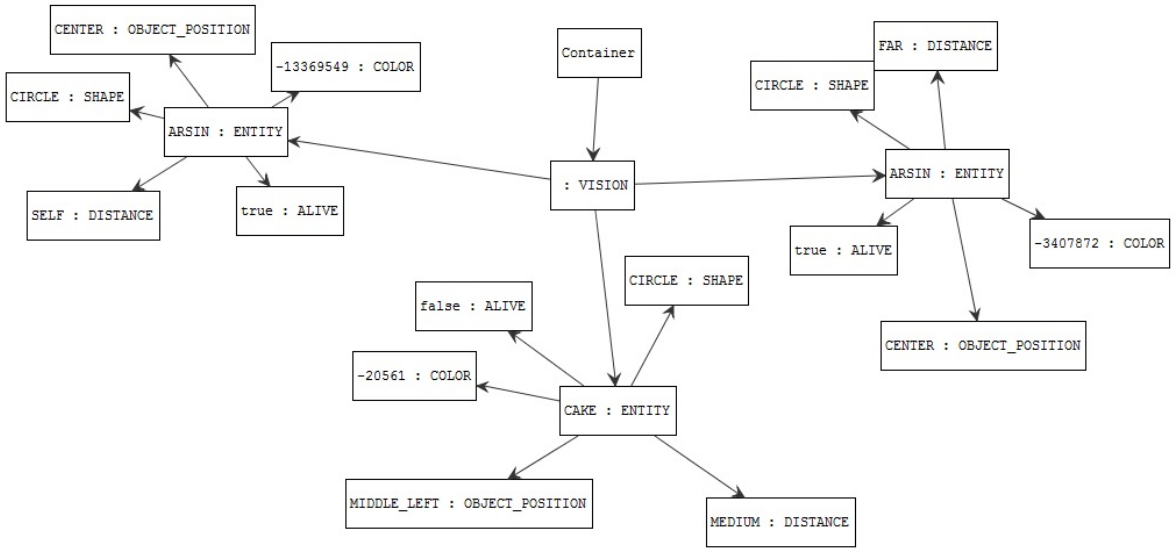

Abbildung 72: Empfangene Daten

5.2.3 Testfall 3 – Senden und Empfangen von Daten

Anforderungen	A03 – Senden und Empfangen von Daten
Ausführungsbeschreibung	Es werden Inputdaten an die Decision Unit gesendet und Output Daten ausgewertet.
Erwartetes Ergebnis	Die empfangenen Inputdaten werden an die Decision Unit und die gesendeten Output Daten werden an den Körper weitergegeben.

Tabelle 13: Testfall3

Dieser Test muss nicht explizit ausgeführt werden, da er implizit in den Testfällen 2 und 4 enthalten ist. Im Testfall 2 wird die Struktur der empfangenen Daten aufgezeigt, dass ein korrektes Empfangen der Daten bedingt. In Testfall 4 werden die unterschiedlichen Handlungsausgänge gezeigt, die nur durch ein erfolgreiches Senden von Aktionsbefehlen möglich sind.

5.2.4 Testfall 4 – Kommunikationsoverhead

Die Anforderung A10 verlangt, dass der Kommunikationsaufwand der jetzigen Schnittstelle nicht überstiegen werden soll. Diese Anforderung konnte nicht zur Gänze erfüllt werden, da es durch die Anordnung von Layern zu einem höheren Kommunikationsaufwand kommt. Der Nachteil wurde jedoch so gering als möglich gehalten, da es eine Implementierung für den untersten Layer gibt, die durch Funktionsaufrufe keinen zusätzlichen Rechenaufwand verursacht.

5.2.5 Testfall 5 – Konfiguration der Decision Unit

Anforderungen	A11 – Konfiguration von Parametern innerhalb der Decision Unit
Ausführungsbeschreibung	Die Decision Unit wird für beiden Handlungsausgänge unterschiedlich konfiguriert und ausgeführt. Die Kommunikationsschnittstelle soll dabei keinen Einfluss auf die Konfigurierbarkeit haben.
Erwartetes Ergebnis	Durch die unterschiedlichen Konfigurationen kommt es zu unterschiedlichen Handlungsausgängen.

Tabelle 14: Testfall 5

Die Decision Unit lässt sich mit Hilfe von Textfiles für die jeweiligen Handlungsausgänge konfigurieren. Abbildung 73 zeigt die beiden Handlungsausgänge, die nur durch Änderung der Konfigurationsfiles erzeugt wurden. Dadurch kann gezeigt werden, dass die Kommunikationsschnittstelle die Konfigurierbarkeit der Decision Unit nicht einschränkt.

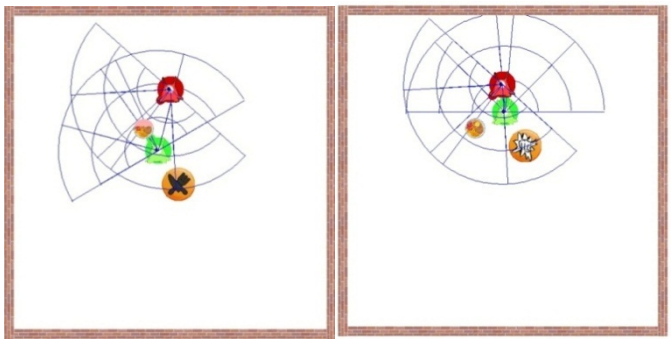

Abbildung 73: Handlungsausgänge

5.2.6 Testfall 6 – Instanzierbarkeit der Decision Unit

Die Anforderung A12 verlangt eine direkte Instanzierbarkeit der Decision Unit aus der Simulationsumgebung. Diese Anforderung wurde insofern erfüllt, da die Simulationsumgebung für den gesamten Aufbau der Simulation verantwortlich ist. Es wird also innerhalb der Simulationsumgebung sowohl die Schnittstelle als auch die Decision Unit instanziert.

5.2.7 Testfall 7 – Synchrone Kopplung

Anforderungen	A13 – Synchrone Kopplung
Ausführungsbeschreibung	Die Schnittstelle wird für mit Hilfe der clsBlocking Implementierung von Layer 4 für synchrone Kommunikation konfiguriert. Dies kann anhand der Kontrollfluss-

	schnittstelle beobachtet werden.
Erwartetes Ergebnis	Der Sender wird solange blockiert, bis der Empfänger die Daten verarbeitet hat und einen Rückgabewert zurückgegeben hat.

Tabelle 15: Testfall 7

Mit Hilfe dieses Testfalles soll die Verwendung der Schnittstelle zur synchronen Kommunikation aufgezeigt werden. Dazu wird die Kontrollschnittstelle näher betrachtet, da diese als synchrone Kommunikation konfiguriert wurde. Abbildung 74 zeigt die relevanten Loggingausgaben der Simulation. Dabei ist zu sehen, dass der Funktionsblock *BrainSocket* mit der Ausführung auf die Antwort der *Decision Unit* wartet. Die Kommunikation der beiden kann also als synchron bezeichnet werden.

```
[BrainSocket][TRACE] (clsBrainSocket.java:129) - Send Control Command PROCESS
[DecisionUnit][TRACE] (clsPsychoAnalysis.java:142) - Start Decision Unit Process
[DecisionUnit][TRACE] (clsPsychoAnalysis.java:147) - Send Action Commands
[BrainSocket][TRACE] (clsBrainSocket.java:131) - Continue with Body execution
```

Abbildung 74: Logging Ausgabe Testfall 5

5.2.8 Testfall 8 – Mapping der Daten

Anforderungen	A20 – Mapping der Daten
Ausführungsbeschreibung	Die Decision Unit wird an eine Simulationsumgebung mit angeschlossen, in der nicht kompatible Datenstrukturen verwendet werden (Miklas).
Erwartetes Ergebnis	Durch ein durchgeführtes Mapping ist die Kommunikation trotz der verschiedenen Datenstrukturen möglich.

Tabelle 16: Testfall 8

Um das Mapping der Kommunikationsdaten aufzuzeigen, wird die Ausgabe des Vision Sensors der Simulationsumgebung Miklas näher betrachtet. In Abbildung 75 ist die Datenstruktur zu sehen die der Vision Sensor erzeugt. Diese wird mit den Abbildung 72 zu sehenden empfangenen Inputdaten aus den Inspektoren von F10 verglichen. Es kann also gezeigt werden, dass trotz der unterschiedlichen Datenstrukturen eine Kommunikation der beiden Funktionsblöcke möglich ist.

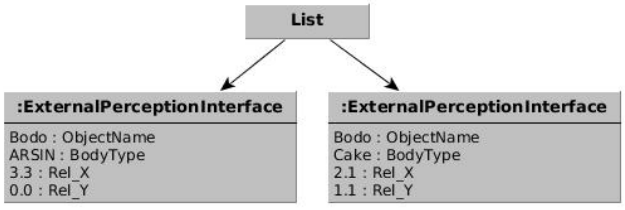

Abbildung 75: Datenstruktur Miklas

5.2.9 Testfall 9 – Decision Unit als eigener Thread

Anforderungen	A21 – Decision Unit als eigener Thread A22 – Asynchrone Kopplung
Ausführungsbeschreibung	Die Decision Unit wird als eigene Thread gestartet und die Schnittstelle wird mit der clsNonBlocking Implementierung von Layer 4 für asynchrone Kommunikation konfiguriert und ausgeführt.
Erwartetes Ergebnis	Der Sender wird nach dem Senden nicht blockiert. Die Antwort des Empfängers wird asynchron empfangen.

Tabelle 17: Testfall 9

In diesem Testfall soll gezeigt werden, dass sich die Decision Unit als eigener Thread ausführen lässt und die Kommunikationsschnittstelle für die asynchrone Kommunikation einsetzbar ist. In Abbildung 76 sind die relevanten Logging Ausgaben des Testes zu sehen. Dabei ist zu erkennen, dass der Funktionsblock *clsBrainSocket* seine Berechnungen nach dem Senden der Daten fortsetzt und die Antwort der Decision Unit asynchron dazu empfängt.

```
[BrainSocket][TRACE] (clsBrainSocket.java:136) - Send Sensor Data to Decision Unit
[BrainSocket][TRACE] (clsBrainSocket.java:138) - Continue with Body execution
[DecisionUnit][TRACE] (clsPsychoAnalysis.java:142) - Start Decision Unit Process
[DecisionUnit][TRACE] (clsPsychoAnalysis.java:147) - Send Action Commands
[BrainSocket][TRACE] (clsBrainSocket.java:143) - Action Commands receieved
```

Abbildung 76: Logging Ausgabe Testfall 7

5.2.10 Testfall 10 – Austauschen der Kommunikationslogik

Anforderungen	A23 – Austauschbare Kommunikationslogik
Ausführungsbeschreibung	Es wird dasselbe Testsetting wie im Testfall 2 verwendet. Mit dem Unterschied, dass die Kommunikationslogik durch eine andere Implementierung ausgetauscht wird.
Erwartetes Ergebnis	Trotz geänderter Kommunikationsimplementierung soll es zu denselben Ergebnissen wie in Testfall 2 kommen.

Tabelle 18: Testfall 10

Für diesen Test wurde der Testfall 3 wiederholt. Mit dem Unterscheid, dass eine neue Layer 1 Implementierung verwendet wurde. Die Ergebnisse des Tests zeigen, dass der Kommunikationslayer ausgetauscht werden kann.

5.2.11 Testfall 11 – Anpassbarkeit der Decision Unit

Anforderungen	A24 – Anpassbarkeit der Decision Unit
Ausführungsbeschreibung	Durch Konfiguration muss die Decision Unit an unterschiedliche Szenarien angepasst werden können.
Erwartetes Ergebnis	Die Decision Unit soll sich durch Konfiguration unterschiedlich verhalten.

Tabelle 19: Testfall 11

Durch das Anpassen der Konfigurationsparameter soll es möglich sein die Decision Unit für unterschiedliche Szenarien zu konfigurieren. Das Ergebnis dieses Tests ist in Abbildung 78 und Abbildung 79 zu sehen. Die Decision Unit reagiert auf dieselbe Ausgangssituartion, je nach Konfiguration, mit unterschiedlichen Handlungsausgängen.

5.2.12 Testfall 12 – Keine parallelen Schnittstellen

Anforderung A30 fordert, dass es keine parallelen Schnittstellen für die unterschiedlichen Ausführungsumgebungen geben soll. Diese Anforderung kann leider nicht über einen Komponententest überprüft werden. Ein Blick in den Quellcode der Anbindung an die Decision Unit zeigt jedoch, dass es keine parallelen Schnittstellen gibt.

5.2.13 Testfall 13 – Keine Abhängigkeiten

Anforderungen	A31 – Keine Abhängigkeiten
Ausführungsbeschreibung	Die Decision Unit wird unabhängig von der Simulationsumgebung in die Miklas eingebaut und ausgeführt.
Erwartetes Ergebnis	Für die Ausführung der Decision Unit müssen keine Pakete aus der ARS-Simulationsumgebung geladen werden.

Tabelle 20: Testfall 13

Dazu wurden zwei Java Pakete erstellt. Ein Paket beinhaltet die Decision Unit und ein Paket beinhaltet die Kommunikationsschnittstelle. Es konnte gezeigt werden, dass das Paket der Kommunikationsschnittstelle vollkommen unabhängig in anderen Projekten eingesetzt werden kann. Das Paket

der Decision Unit kann in Kombination mit der Kommunikationsschnittstelle in anderen Projekten eingesetzt werden. Dabei war eine Referenz zu Java Klassen, die die ARS-Simulationsumgebung oder den ARS-Körper zugerechnet werden können, nicht notwendig. Dadurch konnte gezeigt werden, dass keine Abhängigkeit zwischen diesen Funktionsblöcken bestehen.

5.2.14 Testfall 14 – Kein Anpassen der Decision Unit

Anforderung A32 fordert, dass die Decision Unit nicht auf die unterschiedlichen Anforderungen der Ausführungsumgebungen angepasst werden darf. Auch hier kann diese Anforderung nicht mittels Test belegt werden. Ein Blick in den Source Code bestätigt jedoch, dass keine Anpassung stattgefunden hat.

5.2.15 Testfall 15 – Anpassbarkeit der Schnittstelle

Anforderungen	A33 – Anpassbarkeit der Schnittstelle
Ausführungsbeschreibung	Die Schnittstelle wird durch Austausch einzelner Layer an die Anforderungen der Umgebung angepasst.
Erwartetes Ergebnis	Trotz sich ändernden Umgebungsanforderungen soll die Decision Unit korrekt arbeiten

Tabelle 21: Testfall 15

Durch die Anpassbarkeit der Schnittstelle soll es möglich sein die Decision Unit in unterschiedlichen Umgebungen ausführen zu können. Das Ergebnis dieses Tests ist in Abbildung 78 und Abbildung 79 zu sehen. Die Decision Unit wird in unterschiedlichen Umgebungen mit unterschiedlichen Anforderungen ausgeführt, die Entscheidungen bleiben jedoch gleich.

5.3 Abnahmetest

Im Projekt ARS ist es üblich das Modell über Use Cases zu evaluieren. Diese Use Cases beschrieben die allgemeine Situation die zur Testausführung geschaffen werden soll. Da im Zuge der Tests nicht die Funktionen der Decision Unit sondern die Richtigkeit der Kommunikationsschnittstelle evaluiert werden soll, wird nicht nur die Ausgangssituation, sondern auch der Ablauf des Use Cases beschrieben und als gegeben angenommen. Wie die Decision Unit im Zusammenspiel mit der Simulationsumgebung zu diesem Handlungsablauf kommt, ist nicht Teil der Aufgabenstellung und wird daher auch nicht evaluiert oder bewertet. Ausgehend von diesen Beschreibungen wurde, mittels Konfiguration in jeder der beiden Simulationsumgebungen diese Situation geschaffen.

Dieser Test soll zeigen, dass unabhängig von der Ausführungsumgebung die Decision Unit zu denselben Ergebnissen kommt. Damit kann gezeigt werden, dass die Simulationsumgebung die Decision Unit nicht beeinflusst und somit eine Unabhängigkeit der Komponenten erreicht werden konnte.

5.3.1 Ausgangssituation

Die Ausgangslage des Use Cases ist eine Situation mit zwei Agenten und einer Nahrungsquelle deren Anordnung in Abbildung 77 zu sehen ist. Der Agent Arsin (unten) beinhalten die ARS Decision Unit und somit auch die Implementierung des Kommunikationsmodelles. Bei dem zweiten Agenten (oben) und der Nahrungsquelle (Schnitzel) handelt es sich um passive Objekte, deren Zweck es ist die Entscheidungen der Decision Unit von Arsin zu beeinflussen. Die in weiterer Folge ausgewerteten Daten stammen also lediglich von dem aktiven Agenten, da nur dieser eine Implementierung der zu testenden Kommunikationsschnittstelle in sich trägt.

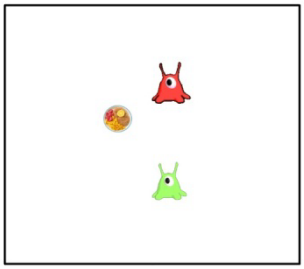

Abbildung 77: Ausgangslage Use Case

Aufbauend auf diese Ausgangssituation werden zwei alternative Handlungsabläufe definiert. Diese beiden Abläufe werden durch Konfiguration der Decision Unit erreicht. Aber auch hier gilt: Wie die Decision Unit zu den Handlungsabläufen kommt, ist nicht Teil des Testes und wird weder evaluiert noch bewertet.

5.3.2 Handlungsablauf 1

Arsin nimmt sowohl Bodo als auch die Nahrungsquelle wahr und entscheidet sich das Schnitzel zu essen. Im ersten Teil des Handlungsablaufes bewegt sich Arsin auf das Schnitzel zu und bleibt vor diesem stehen. Dazu verwendet der die Aktionsbefehle *TURN_LEFT* und *TURN_RIGHT* um die Fortbewegungsrichtung festzulegen und den Befehl *MOVE_FORWARD* um einen Schritt zu gehen. Sobald Arsin vor dem Schnitzel steht, beginnt er mit dem Aktionsbefehl *EAT* diesen zu essen.

5.3.3 Handlungsablauf 2

Arsin nimmt, wie in Handlungsablauf 1, sowohl Bodo als auch die Nahrungsquelle wahr, entscheidet sich jedoch diesmal Bodo zu schlagen. Im ersten Teil des Handlungsablaufes bewegt sich Arsin auf Bodo zu. Er verwendet dabei dieselben Mechanismen wie in Handlungsablauf 1. Sobald Arsin vor Bodo steht, beginnt er mit dem Aktionsbefehl *BEAT* auf diesen einzuschlagen.

5.3.4 ARS-Simulationsumgebung

Die Welt innerhalb der ARS-Simulationsumgebung kann mit Hilfe von Konfigurationsdateien definiert werden. In diesen werden die Objekte sowie deren Position und Eigenschaften festgelegt. Das Ergebnis dieser Konfiguration ist in Abbildung 78 (linkes Bild) zu sehen. Ausgehend aus der Anfangssituation läuft die Simulation, und kommt abhängig von der Konfiguration der Decision Unit zu unterschiedlichen Ausgängen. Die beiden Ausgänge des Testes sind in dem mittleren und rechten Bild der Abbildung 78 zu sehen. Damit soll abschließend aufgezeigt werden, dass die Kommunikationsschnittstelle die Szenarien, des im Projekt ARS definierten Use Cases unterstützt und innerhalb von diesem eingesetzt werden kann.

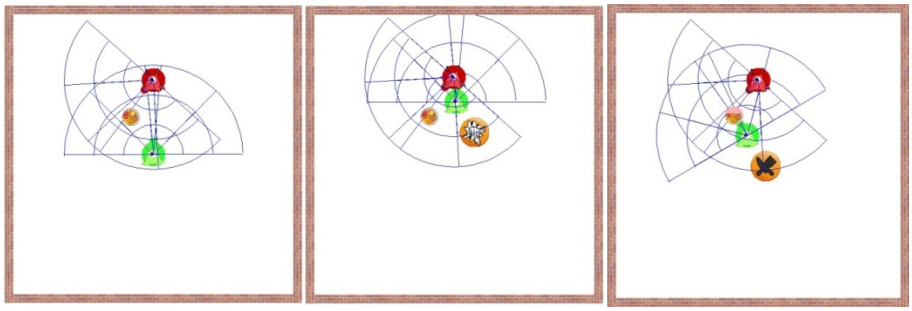

Abbildung 78: Simulationsumgabung MASON

5.3.5 Miklas

Die Welt innerhalb der Simulationsumgebung Miklas kann ebenfalls über Konfigurationsdateien definiert werden. Durch den hohen Grad an Parametrisierbarkeit besteht sogar die Möglichkeit externe Decision Units zu definieren, die zur Compilezeit noch nicht eingebunden waren. In Abbildung 79 (linkes Bild) ist das Resultat der Konfiguration zu sehen. Die Konfiguration der Decision Unit erfolgt auf dieselbe Weise wie in der MASON Simulationsumgebung. Aus diesem Grund führt die Simulation auch zu denselben Ergebnissen. Dadurch kann gezeigt werden, dass durch den Einsatz der Kommunikationsschnittstelle die Ausführung und Konfigurierbarkeit der Decision Unit unabhängig von der Simulationsumgebung ist.

Abbildung 79: Miklas

5.3.6 Testergebnis

Mit Hilfe des abschließenden Abnahmetests konnte die Funktionalität der Schnittstelle innerhalb der Simulationsumgebungen aufgezeigt werden. Abbildung 78 und Abbildung 79 zeigen, wie sich die Decision Unit in den unterschiedlichen Simulationsumgebungen verhält. Dadurch, dass die Ergebnisse der Ausführungen in beiden Umgebungen gleich sind, kann gezeigt werden, dass eine Unabhängigkeit der Decision Unit und der Simulationsumgebung mit Hilfe der Schnittstelle erreicht werden konnte.

6. Conclusio

Basierend auf der Problemstellung (Kapitel 1.2) und dem Task-Setting (Kapitel 1.3) wurde mit Hilfe der im Kapitel 1.4 definierten Vorgangsweise, das Modell einer Schnittstelle, für eine psychoanalytisch inspirierte Decision Unit modelliert (Kapitel 3) und implementiert (Kapitel 4). Bei der Modellierung und Implementierung wurden sowohl auf bewährte Software Design Pattern (Kapitel 2.6) als auch auf Lösungsstrategien von anderen kognitiven Architekturen (Kapitel 2.5) zurückgegriffen. Die Evaluierung der Implementierung des Models erfolgt in Kapitel 5 nach der in Kapitel 1.4 beschriebenen Vorgehensweise. Um die Arbeit abzuschließen, wird in diesem Kapitel über die gewonnenen Ergebnisse diskutiert, und ein Ausblick auf zukünftige Aufgabenstellungen in diesem Gebiet gegeben.

6.1 Zusammenfassung

Die Arbeit wurde im Zuge des ARS-Projektes durchgeführt. Dieses beschäftigt sich mit der Entwicklung eines psychoanalytisch inspirierten Models der menschlichen Informations-verarbeitung. Dieses wird innerhalb einer Artificial-Life Simulation ausgeführt und evaluiert. Als Simulationsumgebung kommt eine, auf den Framework MASON basierende Eigenentwicklung zum Einsatz. Durch die parallele Entwicklung von Simulationsumgebung und Decision Unit ist es im Laufe des Projektes zu einer engen Kopplung dieser beiden Funktionsblöcke gekommen. Aus diesem Grund ist es nicht möglich die Decision Unit in anderen Simulations- bzw. Ausführungsumgebungen einzusetzen. Im weiteren Verlauf des Projektes wird es aber immer interessanter die erarbeiteten Konzepte auch in alternative Umgebung einsetzen zu können. Mögliche alternative Ausführungsumgebungen wären komplexe Artificial-Life Simulationen wie Second-Life oder auch technische Problemstellungen bei denen konventionelle Regelsystem an ihre Grenzen stoßen. Aus diesem Grund wurde im Zuge dieser Arbeit die Abhängigkeit der Decision Unit zur Simulationsumgebung mit Hilfe einer Schnittstelle aufgebrochen. Um die Unabhängigkeit der Decision Unit zur aktuellen Simulationsumgebung aufzuzeigen, wurde die Ausführungs-umgebung Miklas gewählt, in der das ARS-Modell als Entscheidungseinheit eines Agenten eingesetzt werden soll. Die Miklas Umgebung eignet sich für diese Aufgabe, da sie ebenfalls ein Framework zur Artificial-Life Simulation darstellt. Dies hat den Vorteil, dass der Fokus der Arbeit auf die Entwicklung der Schnittstelle gelegt werden konnte, und kein Mapping der Konzepte stattfinden musste. In früheren Projekten (Kapitel 2.4) wurde der Fokus oft falsch gesetzt. So konnte kein allgemein gültiges Schnittstellenmodell entwickelt werden.

Als Vorgehensweise wurde ein Konzept basierend auf dem V-Modell gewählt, da sich dieses in der Software-Entwicklung bestens bewährt hat. Das V-Modell wurde lediglich auf die Anforderungen dieses Projektes angepasst.

Vor Beginn des Schnittstellendesigns musste sowohl die Decision Unit als auch die Ausführungsumgebungen analysiert werden. Da im Zuge des ARS-Projektes bereits Arbeiten zu diesem Thema gemacht wurden, wurde untersucht warum diese nicht die gewünschte Unabhängigkeit zur Folge hatten. Diese Recherchen ergaben, dass bei den untersuchten Projekten der falsche Fokus gesetzt wurde. Die Projekte beschäftigten sich damit die ARS Decision Unit in anderen Ausführungsdomänen einzusetzen. Der Fokus der Arbeiten wurde darauf gelegt die Konzepte des ARS-Projektes auf die Konzepte der neuen Ausführungsdomäne zu überführen. Das Problem der Schnittstelle wurde meist nur als implementierungstechnisches Problem gesehen. Aus diesem Grund wurde kein Schnittstellenmodell entwickelt, sondern lediglich auf den bestehenden Code aufgesetzt. Dies führte jedoch nicht zu einer sauberen Schnittstelle, sondern zu einer genau auf dieses Problem angepassten Lösung. Deshalb wurde in dieser Arbeit ebenfalls eine Artificial-Life Simulation als Anwendungsgebiet gewählt. Weiters wurde recherchiert wie andere kognitive Architekturen (die ähnlich Anforderungen an ihre Schnittstelle definieren) dieses Problem gelöst haben. Dabei wurden speziell die SGIO (SOAR General Input/Output) und das BDI Body-Mind-Interface betrachtet. Einige der beschriebenen Schnittstellenfeatures konnten auch in das ARS-Schnittstellemodell übernommen werden. So wurde zum Beispiel in der SGIO die Socket-kommunikation zwischen unterschiedlichen Prozessen beschrieben. Diese wurde im Zuge der Modellierung der untersten Kommunikationsebene ebenfalls berücksichtigt. Weiters konnte das vom BDI Body-Mind-Interface beschriebene Mapping aufgegriffen und in den Communication-Port integriert werden. Außerdem wurde evaluiert, wie sich bestimmte Entwurfsmuster der Softwareentwicklung auf diese Problemstellung anwenden lassen. Hier ist vor allem das Layerd-Architecture-Pattern zu erwähnen. Dieses bildet das Grundgerüst des Schnittstellenmodells und ermöglicht, Dank der im Pattern beschriebenen hohen Konfigurierbarkeit, das Anpassen der Schnittstelle an alternative Umgebungen.

Auf Basis der zuvor beschriebenen Analysearbeit, wurden in weiterer Folge Anforderungen an die zu erstellende Schnittstelle definiert. Diese Anforderungen wurden aus den Eigenschaften der Decision Unit, der Simulationsumgebungen ARS und Miklas und aus den Fehlern früherer Arbeiten abgeleitet. Basierend auf diesen Anforderungen wurde ein Schnittstellenmodell entwickelt, welches sich am Schichtenmodell des ISO/OSI Modells orientiert. Aufgrund dieser Sichten ist eine Anpassung der Schnittstelle an die jeweiligen Ausführungsumgebungen möglich. Dies hat zur Folge, dass zum einen die Decision Unit funktional nicht angepasst werden muss, und zum anderen es nicht notwendig ist, für jede Ausführungsumgebung eine separate Schnittstelle zu implementieren. Die Schwierigkeit bei der Aufgabe lag daran, dass alle Ausführungsumgebungen zu berücksichtigen waren. Dabei wurden die ARS-Simulationsumgebung, Miklas, und das Pagamut-Framework zur Anbindung an die Unreal-Game-Engine betrachtet. Weiters wurde versucht die Schnittstelle möglichst generisch zu gestalten, um keine Einschränkungen für zukünftige Anwendungsumgebungen zu erzeugen.

Bevor das entwickelte Schnittstellenmodell auf die Decision Unit angewandt werden konnte, musste diese von der Simulationsumgebung getrennt werden. Im Zuge der gemeinsamen Entwicklung von Decision Unit und Simulationsumgebungen ist es zu einer engen Vermaschung dieser beiden Funk-

tionsblöcke gekommen. Die zu Beginn definierte Schnittstelle wurde nicht konsequent eingehalten. Für die jeweiligen Probleme wurden parallele Schnittstellen implementiert, die nur für genau diese Problemstellung ausgelegt wurden. Das Problem lag dabei jedoch nicht an der anfänglichen Schnittstellendefinition, sondern an dem Willen der Entwickler diese einzuhalten und gegebenenfalls zu erweitern. Daraus kann gelernt werden, dass es nicht genügt eine Schnittstelle zu definieren. Diese muss außerdem ausreichend dokumentiert werden, sodass zukünftige Entwickler in der Lage sind diese zu verstehen und auf dieser aufzubauen. In den meisten Fällen ist es nicht möglich durch Programmiertechniken sicherzustellen, dass die Schnittstelle nicht umgangen wird. Aus diesem Grund müssen auch zukünftige Entwickler darauf achten die Schnittstelle nicht zu umgeben, sondern diese bei Bedarf zu erweitern. Deshalb wurde in dieser Arbeit ein großer Wert auf die Konfigurierbarkeit der Schnittstelle gelegt, sodass bei zukünftigen Anforderungen die Schnittstelle möglichst einfach erweitert werden kann.

Nach der Implementierung des Schnittstellenmodells konnte diese an die Decision Unit und an die Simulationsumgebung angeschlossen werden. Dazu wurden auf beiden Seiten Kommunikationsports verwendet. Diese ermöglichen es, das klar definierte Interface zur Kommunikationsschnittstelle mit der eigentlichen Anwendung zu verbinden. Dadurch kann die Anbindung an die Schnittstelle tief in die Struktur der Anwendung integriert werden.

Um den Vorgaben des V-Modelles gerecht zu werden wurde ein vier stufiger Evaluierungsprozess eingesetzt. Im ersten Schritt wurde direkt auf Softwareebene die Korrektheit der einzelnen Programmteile evaluiert. Im Komponententest wurden die einzelnen Schichten des Schnittstellenmodelles in einem Blackbox Testverfahren auf ihre Interface Spezifikationen getestet. Parallel zur Erstellung der Anforderungen wurden Systemtests definiert, die die Einhaltung der jeweiligen Anforderung sicherstellen. Abschließend wurde das Verhalten des gesamten Systems im Abnahmetest, basierend auf zuvor definierten Use Cases, evaluiert.

Abschließend kann gesagt werden, dass mit Hilfe dieser Schnittstelle es nun möglich ist, die ARS Decision Unit in unterschiedlichen Ausführungsumgebungen einzusetzen. Diese Aussage ist jedoch mit Einschränkungen verbunden. Durch die funktionale Beschreibung der ARS Decision Unit sollte es möglich sein, diese auf die Daten der Simulationsumgebung anpassen zu können. Dies sollte ohne Eingriffe in die Funktionen nur durch eine geänderte Konfiguration möglich sein. Die aktuelle Implementierung der ARS Decision Unit erfüllt diese Eigenschaften jedoch nicht in alle Bereichen. Daher ist ein Mapping der Daten aus der Simulationsumgebung auf die in ARS verwendeten Daten notwendig. Leider ist es im Zuge dieser Arbeit nicht möglich diese Abhängigkeit aufzubrechen. Es wurde jedoch darauf geachtet, dass die Schnittstelle eine zukünftige Konfigurierbarkeit nicht einschränkt.

6.2 Ausblick

Wie bereits in der Zusammenfassung erwähnt, ist eine Verwendung der ARS Decision Unit in alternativen Umgebungen nur durch ein Mapping der Daten möglich. Als Beispiel dient in diesem Fall der Hungertrieb des Agenten. In der Artificial Life Simulation dient der Hungertrieb dazu den Magenfüllstand des Agenten und das Bedürfnis der Nahrungsaufnahme zu repräsentieren. Wird die

Decision Unit jedoch zum Beispiel zur Steuerung eines Glashauses (siehe Kapitel 2.4.2) eingesetzt, dann wird dazu kein Hungertrieb benötigt. In der jetzigen Form ist es nötig einen Aspekt aus der Glashaussteuerung auf den vorhandenen Hungertrieb zu Mappen und diese innerhalb des Funktionsmodelles zu verwenden. Besser wäre es jedoch nicht den Hungertrieb, sondern direkt den Aspekt der Glashaussteuerung auf das Konzept der Triebe abzubilden. Die Aufgabe von zukünftigen Projekten sollte dieses Mapping der Daten durch eine bessere Konfigurierbarkeit der Decision Unit ersetzen.

Zur Visualisierung interner Daten wird im ARS-Projekt das von MASON zur Verfügung gestellte Konzept der Inspektoren verwendet. Das hat zur Folge, dass wenn die Decision Unit unabhängig von der auf MASON basierenden Simulationsumgebung ausgeführt wird, diese Inspektoren nicht mehr zur Verfügung stehen. Eine Visualisierung interner Vorgänge wäre aber in jeder möglichen Ausführungsumgebung von Vorteil. Da ohne diese Visualisierung die Vorgänge innerhalb der Decision Unit nicht dargestellt werden können und diese nur als Blackbox betrachtet werden kann. Die Vielfallt an Input und Output Werten und die Komplexität der Entscheidungsfindung lässt jedoch bei dieser Betrachtungsweise keine Schlüsse auf die inneren Vorgänge zu. Die Aufgabe zukünftiger Arbeiten sollte es also sein ein Inspektoren-Konzept zu entwickeln, dass unabhängig, von der auf MASON basierenden ARS-Simulationsumgebung, eingesetzt werden kann.

Basierend auf der in dieser Arbeit entwickelten allgemein gültigen Schnittstelle zur ARS Decision Unit ergeben sich neue Möglichkeiten für das ARS-Projekt. Betrachtet man die Ausführung der Simulation innerhalb der MASON Simulationsumgebung dann fällt auf, dass das Funktionsmodell mehrerer Durchläufe benötigt um zu einer Entscheidung zu kommen. Da die Simulationsumgebung während dieser Durchläufe ebenfalls weiterläuft, können die Eingangsdaten nicht so lange fixiert werden, bis die Entscheidung getroffen wurde. Durch die Art dieser Simulation wird das Ergebnis beeinflusst. Die Simulationsumgebung muss so langsam sein, dass die Decision Unit genug Zeit hat ihre Ergebnisse zu berechnen. Durch das Ergebnis dieser Arbeit ist es möglich diese Korrektur durchzuführen.

Eine weitere Anwendungsmöglichkeit der Schnittstelle wäre die Anbindung an eine andere Simulationsumgebung. Dadurch wäre es möglich das ARS-Funktionsmodell in einer möglichst realistischen Umgebung auszuführen und zu validieren. Mit steigender Komplexität steigen auch die Anforderungen an die Simulationsumgebung und eine Eigenentwicklung ist mit hohem Aufwand verbunden. Der Umstieg auf Second-Life als Simulationsumgebung würde diesen Aufwand verringern. Außerdem würde eine möglichst realistische Simulationsumgebung die Möglichkeiten der Simulation erheblich erweitern. Mit Hilfe der entwickelten Schnittstelle ist es möglich die ARS Decision Unit auch in Second-Life einzusetzen.

Weiters wäre es möglich die ARS Decision Unit zur Lösung von technischen Problemstellungen einzusetzen. Als Beispiel kann hier entweder die bereits erwähnte Steuerung eines Glashauses oder die zu Beginn des ARS-Projektes ins Auge gefasste Hausautomation dienen.

Literaturverzeichnis

[Ber13] Marc FreixasBerenguer. *Artificial Recognition System Analysis for Applications like Greenhouse*. PhD thesis, Vienna University of Technology, 2013.

[BGSW13] Dietmar Bruckner, Friedrich Gelbard, Samer Schaat, and Alexander Wendt. Validation of cognitive architectures by use cases: Examplified with the psychoanalytically-inspired ars model implementation. In *Industrial Electronics (ISIE), 2013 IEEE International Symposium on*, pages 1–6. IEEE, 2013.

[Cop98] James O Coplien. Software design patterns: Common questions and answers. *The Patterns Handbook: Techniques, Strategies, and Applications. Cambridge University Press, NY*, pages 311–320, 1998.

[Deu11] Tobias Deutsch. *Human Bionically Inspired Autonomous Agents*. PhD thesis, Vienna University of Technology, 2011.

[Dön09] Benjamin Dönz. *Actuators for an Artificial Life Simulation*. PhD thesis, Faculty of Electrical Engineering, Vienna University of Technology, 2009.

[DS00] Dietmar Dietrich and Tilo Sauter. Evolution potentials for fieldbus systems. In*Factory Communication Systems, 2000. Proceedings. 2000 IEEE International Workshop on*, page 343. IEEE, 2000.

[DSB+13] Dietmar Dietrich, Samer Schaat, Dietmar Bruckner, Klaus Doblhammer, and Georg Fodor. The current state of psychoanalytically-inspired ai. In *Industrial Electronics Society, IECON 2013-39th Annual Conference of the IEEE*, pages 6666–6671. IEEE, 2013.

[Fer12] Enrique Muñoz Fernández. *Implementation of the Artificial Recognition System Decision Unit in a Complex Simulation Environment*. PhD thesis, Vienna University of Technology, 2012.

[FHKS09] Jan Friedrich, Ulrike Hammerschall, Marco Kuhrmann, and Marc Sihling. Das v-modell xt. In *Das V-Modell XT*, pages 1–32. Springer, 2009.

[GBP11] Jakub Gemrot, Cyril Brom, and TomášPlch. A periphery of pogamut: From bots to agents and back again. In *Agents for games and simulations II*, pages 19–37. Springer, 2011.

[HJVG02] Richard Helm, Ralph Johnson, John Vlissides, and Erich Gamma. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 2002.

[JW06] Randolph M Jones and Robert E Wray.Comparative analysis of frameworks for knowledge-intensive intelligent agents. *AI magazine*, 27(2):57, 2006.

[LAB+00] John E Laird, MazinAssanie, Benjamin Bachelor, Nathan Benninghoff, Syed Enam, Bradley Jones, Alex Kerfoot, Colin Lauver, Brian Magerko, Jeff Sheiman, et al. A test bed for developing intelligent synthetic characters. *AnnArbor*, 1001:48109–2110, 2000.

[Lan10] Roland Lang. *A Decision Unit for Autonomous Agents Based on the Theory of Psychoanalysis*. PhD thesis, Vienna University of Technology, 2010.

[LCRP+05] Sean Luke, Claudio Cioffi-Revilla, LiviuPanait, Keith Sullivan, and Gabriel Balan. Mason: A multiagent simulation environment. *Simulation*, 81(7):517–527, 2005.

[LNR87] John E Laird, Allen Newell, and Paul S Rosenbloom. Soar: An architecture for general intelligence. *Artificial intelligence*, 33(1):1–64, 1987.

[Muc13] Clemens Muchitsch. *Human-like Perception for Psychoanalytically Inspired Reasoning Unit.* PhD thesis, Vienna University of Technology, 2013.

[RG+95] Anand S Rao, Michael P Georgeff, et al. Bdi agents: From theory to practice. In *ICMAS*, volume 95, pages 312–319, 1995.

[SDW+13] Samer Schaat, Klaus Doblhammer, Alexander Wendt, Friedrich Gelbard, Lukas Herret, and Dietmar Bruckner. A psychoanalytically-inspired motivational and emotional system for autonomous agents. In *Industrial Electronics Society, IECON 2013-39th Annual Conference of the IEEE*, pages 6648–6653. IEEE, 2013.

[SSRB13] Douglas C Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*, volume 2. John Wiley & Sons, 2013.

[TNB03] Jorge A Torres, Luciana P Nedel, and Rafael H Bordini. Using the bdi architecture to produce autonomous characters in virtual worlds. In *Intelligent Virtual Agents*, pages 197–201. Springer, 2003.

[Tor12] Ernest OrtuñoTorra. *Game world implementation of Artificial Recognition Systemmodel*. PhD thesis, Vienna University of Technology, 2012.

[VFBD11] Matteo Varvello, Stefano Ferrari, Ernst Biersack, and Christophe Diot. Exploring second life. *IEEE/ACM Transactions on Networking (TON)*, 19(1):80–91, 2011.

[Zim80] Hubert Zimmermann. Osi reference model–the iso model of architecture for open systems interconnection. *Communications, IEEE Transactions on*, 28(4):425–432, 1980.

Internet Referenzen

[1] Artificial Recognition System. *Homepge, 13.8.2014.*www.ars.ict.tuwien.ac.at

[2] Institut für Computertechnik. *Homepage, 13.8.2014* www.ict.tuwien.ac.at

[3] MASON Framework. *Homepage, 13.8.2014*. cs.gmu.edu/~eclab/projects/mason/

[4] Miklas Game Engine. *Source Code Publikation, 13.8.2014.* https://code.google.com/p/miklas/

Erklärung

Hiermit erkläre ich, dass die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt wurde. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Wien, 18. September 2014

Lukas Herret