8. Implementierung eines Simulationssystems

In diesem Kapitel wird ein implementiertes Simulationssystem vorgestellt, das es in seiner Gesamtheit ermöglicht, in UML spezifizierte agentenbasierte Simulationsmodelle automatisch ausführen zu lassen. Die Implementierung des in Java geschriebenen Simulators, die den Charakter eines *proof of concept* und nicht eines fertigen Softwareproduktes hat, wurde dahingehend erweitert, dass die mit Hilfe eines UML-Tools erstellte Spezifikation direkt ausführbar ist.

Der Kern des Simulators ist eine Java-Programmbibliothek, die eine Klasse enthält, die die Ausführung eines agentenbasierten Simulationsmodells organisiert. Für die Wahl von Java sprach die damit erzielbare Plattformunabhängigkeit. Performanzaspekte, die gegen Java gesprochen hätten, spielen aufgrund des Charakters eines *proof of concept* der Implementierung eine untergeordnete Rolle.

Ein konkretes Simulationssystem wird ausgeführt, indem die verwendeten Typen (Agenten- und Objekttypen, Ereignis-, Wahrnehmungs- und Aktionstypen) als Unterklassen der generischen Typklassen realisiert werden. Um zu erreichen, dass man als Modellierer vollständig in UML spezifizieren kann und dieses Modell dann direkt ausführen kann, wird das Format XMI [OMG04b] verwendet, eine standardisierte XML-basierte Sprache, die eine textuelle Repräsentation eines UML-Modells darstellt. Viele der heutigen UML-Tools verwenden XMI direkt für die Speicherung der UML-Modelle oder bieten zumindest die Möglichkeit, ein UML-Modell in das Format XMI zu exportieren. Mit Hilfe von XSL-Transformationen wird dann das im XMI-Format vorliegende UML-Modell zunächst in ein XML-basiertes Zwischenformat und dann direkt in vom Simulator verwendbaren Java-Code transformiert. XSL-Transformationen sind eine XML-Anwendung, die es ermöglicht, Dokumente aus einem XML-basierten Format in ein anderes Format gemäß gewisser Transformationsregeln zu überführen.

Als Basis der Entwicklung diente zunächst die Entwicklung eines Simulationssystems für die Diskrete-Ereignis-Simulation. Implementiert wurde dabei das Simulationssystem *DESim* (*Discrete Event Simulator)* gemäß der in Abschnitt 2.1 beschriebenen Formalisierung der Diskreten-Ereignis-Simulation in der Programmiersprache Java.

Darauf aufbauend wurde ein entsprechendes Simulationssystem *OBSim* (*Object Based Simulator)* gemäß der in Abschnitt 3.4.1 beschriebenen Formalisierung der Objektbasierten Simulation entwickelt.

Schließlich wurde das Simulationssystem *ABSim* (*Agent Based Simulator)* gemäß der in Abschnitt 4.3.1 beschriebenen Formalisierung der Agentenbasierten Simulation entwickelt, das Bestandteil des entwickelten Systems zur Agentenbasierten Simulation ist.

Der jeweilige Kern von *DESim*, *OBSim* und *ABSim* besteht aus einer kleinen Menge von abstrakten Java-Klassen, die ein Benutzer mittels Vererbung konkretisieren kann. Der Kern entspricht somit einem Simulationstool wie z.B. *Silk* [Kil00], bei dem der Benutzer ein konkretes Simulationssystem mit Hilfe eines Frameworks programmieren muss.

Das Ziel des hier vorgestellten Simulationssystems ist es hingegen, einem Benutzer die Last zu nehmen, programmieren zu müssen. Das konkrete Agentenbasierte Simulationssystem, also die konkreten Agenten- und Objekttypen, Ereignis-, Wahrnehmungs- und Aktionstypen sowie die konkreten Regeln für den Umgebungssimulator und die Agentensimulatoren, können deshalb in einem XML-basierten Format, *ABSimML* (*Agent Based Simulation Markup Language*) genannt, spezifiziert werden, aus dem dann mittels XSL-Transformationen konkrete Java-Klassen erzeugt werden können, die im Simulator *ABSim* ablaufen.

Um dem Benutzer auch noch die Last zu nehmen, Simulationssysteme in einer XML-basierten, nicht-grafischen Auszeichnungssprache zu spezifizieren, wird ihm die Möglichkeit gegeben, mittels eines UML-Tools das konkrete Simulationssystem als UML-Modell zu spezifizieren und aus diesem eine XMI-Repräsentation zu erzeugen. Diese wiederum kann mittels XSLT in *ABSimML* transformiert werden.

Damit ein Benutzer das Verhalten eines simulierten Systems beobachten kann, ist es sinnvoll, dass der Simulator einen Output erzeugt. Man kann den Output in Statistik-Output, Visualisierung und

Systemmonitor aufteilen. Die vorliegende Implementierung enthält allerdings nur einen textuellen Output auf der Systemkonsole.

Um zu untermauern, dass Agentenbasierte Simulation tatsächlich auf die Diskrete-Ereignis-Simulation abbildbar ist, könnte man für die Abbildung eine Transformation mittels XSLT nutzen. Dabei würde von *OBSimML* direkt nach *DESimML* transformiert, während von *ABSimML* nach *OBSimML* und dann nach *DESimML* transformiert wird. Eine Transformation wäre auch von Java-Code zu Java-Code denkbar. Auf all diese Transformationen wurde im tatsächlich implementierten System allerdings verzichtet. Abbildung 65 zeigt alle möglichen Transformationen, wobei die tatsächlich realisierten Systembestandteile und Transformationen in grau und die nicht realisierten in weiß eingezeichnet sind. Wie man der Abbildung entnehmen kann, wurde also ein komplettes Simulationssystem für die Agentenbasierte Simulation sowie für die Objektbasierte Simulation entwickelt. Für die Diskrete-Ereignis-Simulation wurde nur der Kern entwickelt, ohne die Möglichkeit zu bieten, XML-basiert oder UML-basiert Simulationssysteme zu spezifizieren.

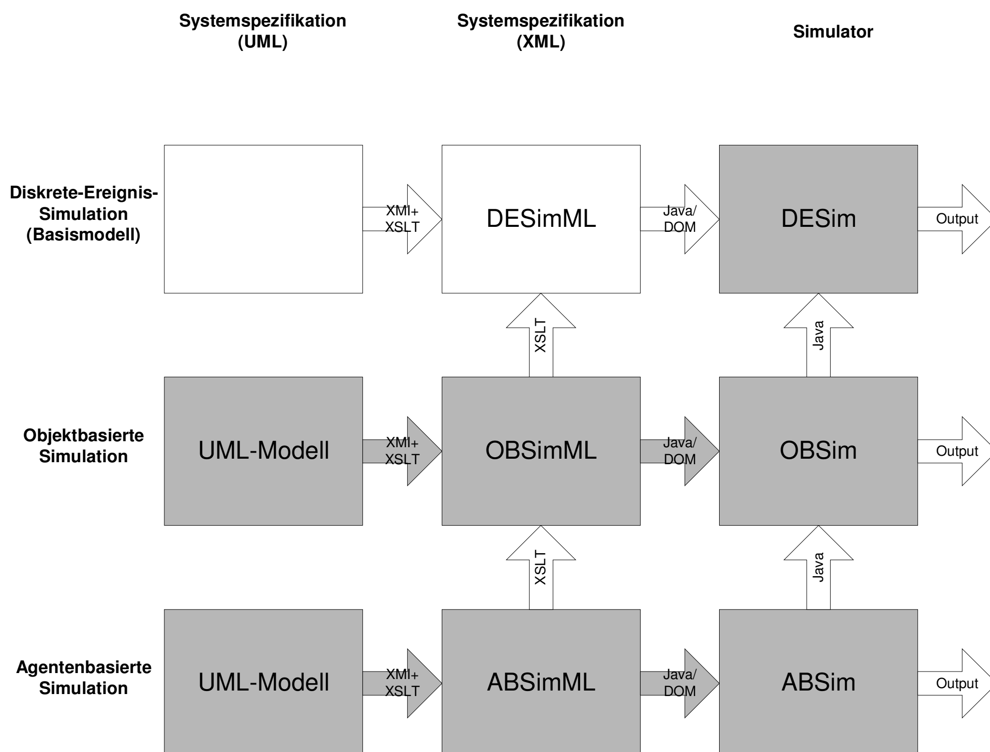

Abbildung 65: Mögliche und realisierte Bestandteile und Transformationen im Simulationssystem

In Abschnitt 8.1 wird zunächst der Kern des Simulators *DESim* vorgestellt. Es folgen in Abschnitt 8.2 der Kern des Simulators *OBSim* und in Abschnitt 8.3 der Kern des Simulators *ABSim*. In Abschnitt 8.4 wird vorgestellt, wie im implementierten System in UML ein Simulationssystem spezifiziert wird. Aufgrund der Unzulänglichkeit der existierenden Tools ergeben sich hier nämlich Abweichungen zur in Abschnitt 7.1 vorgestellten Sprache. In Abschnitt 8.5 wird das XML-basierte Format *ABSimML*, das eine vollständige Repräsentation eines konkreten Agentenbasierten Simulationssystems dargestellt, vorgestellt. Abschnitt 8.6 zeigt auf, wie man aus einem im Format XMI vorliegenden UML-Modell, das ein gemäß Abschnitt 8.4 spezifiziertes Agentenbasiertes Simulationssystem darstellt, mit Hilfe von XSLT in das Format *ABSimML* transformiert. Schließlich wird in Abschnitt 8.7 gezeigt, wie man aus einem im Format *ABSimML* vorliegenden Agentenbasierten Simulationssystem Code erzeugt, der im Simulator *ABSim* lauffähig ist. Abschnitt 8.8 stellt eine komplexe Simulationsstudie Fahrerloser Transportsysteme, die mit dem Simulator *ABSim* erstellt wurde, vor.

8.1 Der Kern des Simulators DESim

Der Kern des Simulators *DESim* besteht aus einer kleinen Menge von abstrakten Java-Klassen, die ein Benutzer[85] mittels Vererbung konkretisieren kann.

Eine dieser abstrakten Klassen ist die Klasse *SimulationSystem*, die das gesamte Simulationssystem darstellt. Sie besteht aus einer Menge von zukünftigen Ereignissen *event_set*, die als Typ die abstrakte Klasse *Event* haben[86]. Die Menge selbst hat den Typ *EventSet*. Ferner hat die Klasse *SimulationSystem* einen Zustand, deren Typ die abstrakte Klasse *State* ist[87], sowie die aktuelle Simulationszeit *time*. Methoden sind die abstrakte Methode *createInitialState()*, die einen ggf. zufälligen Startzustand erzeugt, sowie die unveränderlichen Methoden *f()*, *initialize()* und *simulate()*. Dabei entspricht *f()* der Ausführung der Transitionsfunktion f. Dass *f()* unveränderlich ist, scheint auf den ersten Blick zu überraschen, da ja gerade durch Angabe der Transitionsfunktion f ein konkretes Simulationssystem realisiert wird. In der Realisierung von *f()* wird jedoch die Aufgabe der Ausführung eines Simulationsschrittes an das entsprechende *Event*-Objekt delegiert, das dann wiederum vom Benutzer implementiert wird. Die Methode *initialize()* setzt den Startzustand durch Aufruf des abstrakten *crateInitialState()*. Die Methode *simulate()* entspricht der kompletten Ausführung der Simulation, wobei zunächst mittels *initialize()* der Startzustand gesetzt wird und dann solange die Transitionsfunktion f auf das jeweils nächste Ereignis angewendet wird, wie die Menge der zukünftigen Ereignisse nicht leer ist. Die Ausführung von f beinhaltet die Änderung des Systemzustands sowie das Hinzufügen von Folgeereignissen zur Menge von zukünftigen Ereignissen.

Die abstrakte Klasse Event hat einen Ausführungszeitpunkt *time*. Ferner besitzt sie eine abstrakte Methode *simulate()*, die der Ausführung der Transitionsfunktion f entspricht. Sie gibt ein Tupel[88], besteht aus dem neuen Zustand und den neu zu erzeugenden Ereignissen zurück.

Abbildung 66 zeigt den Entwurf des Simulationssystems *DESim* als UML-Diagramm.

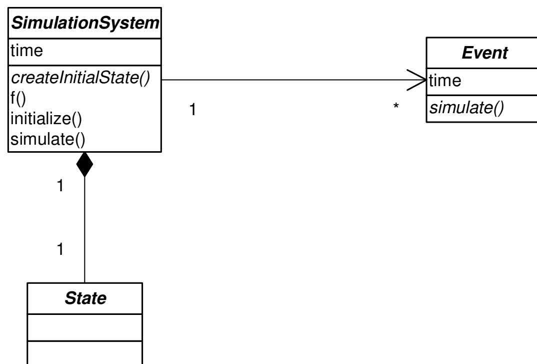

Abbildung 66: UML-Klassendiagramm für den Entwurf des Simulationssystems *DESim*

8.2 Der Kern des Simulators OBSim

Wie der Kern des Simulators *DESim* besteht auch der Kern des Simulators *OBSim* aus einer kleinen Menge von abstrakten Java-Klassen, die ein Benutzer mittels Vererbung konkretisieren kann.

Auch hier ist die Kernklasse dieser abstrakten Klassen ist die Klasse *SimulationSystem*, die das gesamte Simulationssystem darstellt. Sie besteht ebenfalls aus einer Menge von zukünftigen Ereignissen *event_set*, die als Typ die abstrakte Klasse *Event* haben. Die Menge selbst hat den Typ

[85] Benutzer kann hier ein Anwender sein, der sein konkretes Simulationssystem per Hand in Java programmiert. Benutzer kann aber auch ein Programm sein, das den Java-Code, den der DESim-Kern zur Ausführung benötigt, aus einem anderen Format, z.B. DESimML erzeugt.

[86] Natürlich haben die konkreten Ereignisse den Typ einer konkreten Klasse, die Unterklasse der abstrakten Klasse *Event* ist.

[87] Natürlich hat der konkrete Zustand den Typ einer konkreten Klasse, die Unterklasse der abstrakten Klasse *State* ist.

[88] Da es in Java keine Tupel-Typen gibt, wird stattdessen ein Objekt vom Typ *Result* zurückgegeben, das die beiden Komponenten des Rückgabewerts enthält.

EventSet. Ferner hat die Klasse *SimulationSystem* eine Menge von Objekten, die den Typ *SimObject*[89] haben, sowie die aktuelle Simulationszeit *time*. Methoden sind die abstrakte Methode *createInitialState()*, die einen ggf. zufälligen Startzustand erzeugt, sowie die unveränderlichen Methoden *executeEvent()*, *initialize()* und *simulate()*. In der Realisierung von *executeEvent()* wird die Aufgabe der Ausführung eines Ereignisses an das entsprechende *Event*-Objekt delegiert, das dann wiederum Reaktionsregeln vom Typ *ReactionRule* anwendet, deren konkrete Realisierungen vom Benutzer implementiert werden. Die Methode *initialize()* setzt den Startzustand durch Aufruf des abstrakten *crateInitialState()*. Die Methode *simulate()* entspricht der kompletten Ausführung der Simulation, wobei zunächst mittels *initialize()* der Startzustand gesetzt wird und dann solange die Methode *executeEvent()* des jeweils nächsten Ereignisses angewendet wird, wie die Menge der zukünftigen Ereignisse nicht leer ist. Somit ergibt sich für die Methode *simulate()* folgender kurzer und prägnanter und gut lesbarer Code[90] in Abbildung 67.

```
public final void simulate() {
        initialize();
        while (!event_set.isEmpty()) {
                Event e_min = event_set.extractMin();
                time = e_min.time;
                EventSet new_events = executeEvent(e_min);
                event_set.addAllEvents(new_events);
        }
}
```

Abbildung 67: Code für *simulate()* in der Klasse *SimulationSystem* im Simulator *OBSim*.

Die Klasse *ReactionRule* ist eine abstrakte Klasse, die die Bestandteile einer Reaktionsregel auf Methoden abbildet, wie in Abbildung 68 dargestellt. Dabei wird erst mittels *triggeredByEventType()* festgestellt, ob die Reaktionsregel überhaupt für diesen Typ von Ereignissen bestimmt ist, mittels *triggerCondition()*, ob die Trigger-Bedingung durch das Ereignis erfüllt ist, mittels *reactedByObjectType()*, ob das Objekt den Typ hat, der auf dieses Ereignis reagiert und mittels *reactorCondition()*, ob die Zustandsbedingung für diese Reaktionsregel erfüllt ist. Ist das der Fall, so wird mittels *reactorEffect()* der Zustandseffekt direkt auf dem übergebenen Objekt hergestellt und *resultingEvents()* gibt die Menge der Foleereignisse der Reaktionsregel zurück.

```
public abstract boolean triggeredByEventType(Event e);
public abstract boolean triggerCondition(Event e);
public abstract boolean reactedByObjectType(SimObject o);
public abstract boolean reactorCondition(SimObject o, Event e);
public abstract void reactorEffect(SimObject o, Event e);
public abstract EventSet resultingEvents(SimObject o, Event e);
```

Abbildung 68: Methoden der abstrakten Klasse *ReactionRule* im Simulator *OBSim*

Die abstrakte Klasse *Event* hat einen Ausführungszeitpunkt *time*, eine Menge von Reaktionsregeln *rules,* eine abstrakte Methode *doOneStep()* und eine unveränderliche Methode *simulate()*. Die Methode *doOneStep()* entspricht der Ausführung des Ereignisses. Sie gibt eine Menge von Folgeereignissen zurück. In den beiden ebenfalls abstrakten Unterklassen *ExogenousEvent* und *ResultingEvent* wird *doOneStep()* konkretisiert durch Aufruf der Methode *simulate()* des Events, wobei im Falle des *ExogenousEvent* vorher noch das nächste Auftreten dieses Stranges des exogenen Ereignisses in die Menge der Folgeereignisse aufgenommen wird. Die unveränderliche Methode *simulate()* sucht die passende Reaktionsregel sowie das passende Reaktorobjekt, wendet auf ihm den

[89] Um Verwechslungen mit *Object*, der Oberklasse aller Klassen in Java zu vermeiden, wurde dieser Name gewählt.

[90] Weggelassen wurden Kommentare sowie Anweisungen, die die Visualisierung des Simulationssystems betreffen.

Reaktoreffekt der Regel an und gibt die Folgeereignisse der Regel zurück. Somit ergibt sich für die Methode *simulate()* folgender kurzer und prägnanter und gut lesbarer Code[91] in Abbildung 69.

```
for (int i=0; i<rules.length; i++) {
        ReactionRule r = rules[i];
        if (r.triggeredByEventType(this)) {
                if (r. triggerCondition (this)) {
                        for (int j=0; j < objects.length; j++) {
                                SimObject o = objects[j];
                                if (r. reactedByObjectType (o)) {
                                        if (r. reactorCondition (o, this)) {
                                                r. reactorEffect (o, this);
                                                return r.resultingEvents(o, this);
                                        }
                                }
                        }
                }
        }
}
throw new RuntimeException("no matching reaction rule");
```

Abbildung 69: Code für *simulate()* in der Klasse *Event* im Simulator *OBSim*.

Abbildung 70 zeigt den Entwurf des Simulationssystems *OBSim* als UML-Diagramm.

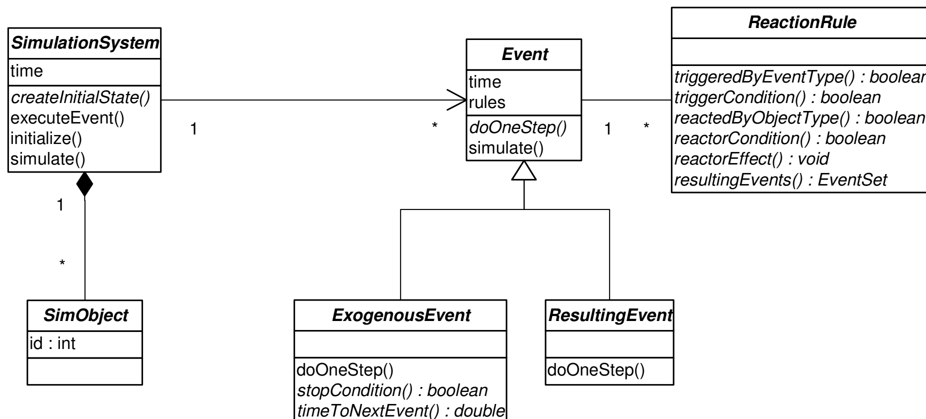

Abbildung 70: UML-Klassendiagramm für den Entwurf des Simulationssystems *OBSim*

8.3 Der Kern des Simulators ABSim

Wie die Kerne der Simulatoren *DESim* und *OBSim* besteht auch der Kern des Simulators *ABSim* aus einer Menge von abstrakten Java-Klassen, die ein Benutzer mittels Vererbung konkretisieren kann.

8.3.1 Package absimulation

Auch im Simulator *ABSim* ist die Kernklasse dieser abstrakten Klassen die Klasse *SimulationSystem*, die das gesamte Simulationssystem darstellt. Sie besteht aus einem Umgebungssimulator vom Typ *EnvSimulator* und mehreren Agentensimulatoren vom Typ *AgentSimulator*. Ferner hat die Klasse *SimulationSystem* die aktuelle Simulationszeit *clock*. Sie verwaltet ferner Referenzen zu den aktuell auftretenden Aktionen, Nachrichten und Wahrnehmungen, die zwischen dem Umgebungssmulator und den Agentensimulatoren ausgetauscht werden. Methoden sind die abstrakten Methoden

[91] Weggelassen wurden auch hier Kommentare sowie Anweisungen, die die Visualisierung des Simulationssystems betreffen.

createInitialEnvironmentState() und *createInitialAgentStates()*, die einen ggf. zufälligen Startzustand für die Umgebung und für die internen Agenten erzeugen, sowie die unveränderlichen Methoden *envSimulatorStep()*, *agentSimulatorStep()*, *initialize()* und *simulate()*.

Die Methode *envSimulatorStep()* stellt dabei einen Schritt des Umgebungssimulators dar. In der Realisierung von *envSimulatorStep()* wird die gleichnamige Methode des Umgebungssimulators mit den Aktionen und verschickten Nachrichten der Agenten als Parameter aufgerufen. Die als Ergebnis erhaltene Wahrnehmungsmenge wird anschließend den entsprechenden internen Agenten zugeordnet.

Die Methode *agentSimulatorStep()* stellt dabei einen Schritt aller Agentenssimulatoren dar. In der Realisierung von *agentSimulatorStep()* wird die gleichnamige Methode der Agentensimulatoren mit den Wahrnehmungen für den Agenten als Parameter aufgerufen. Die als Ergebnis erhaltenen Mengen von Aktionen und verschickten Nachrichten werden anschließend zusammengefasst, wobei bei den Nachrichten noch anhand der Verlustwahrscheinlichkeit und Übertragungsdauer bestimmt wird, ob und wann die Nachricht beim Empfänger ankommt.

Die Methode *initialize()* setzt die Startzustände durch Aufruf von *createInitialEnvironmentState()* und *createInitialAgentStates()*. Die Methode *simulate()* entspricht der kompletten Ausführung der Simulation, wobei zunächst mittels *initialize()* der Startzustand gesetzt wird und dann in einer Endlosschleife die Simulationszeit um eine Zyklenlänge weitergezählt wird und die Methoden *envSimulatorStep()* und *agentSimulatorStep()* aufgerufen werden.

Die Klasse *SimulationSystem* befindet sich – neben der hier außen vor gelassenen Visualisierung – als einzige Klasse im Hauptpackage *absimulation*, dessen Entwurf in Abbildung 71 dargestellt ist.

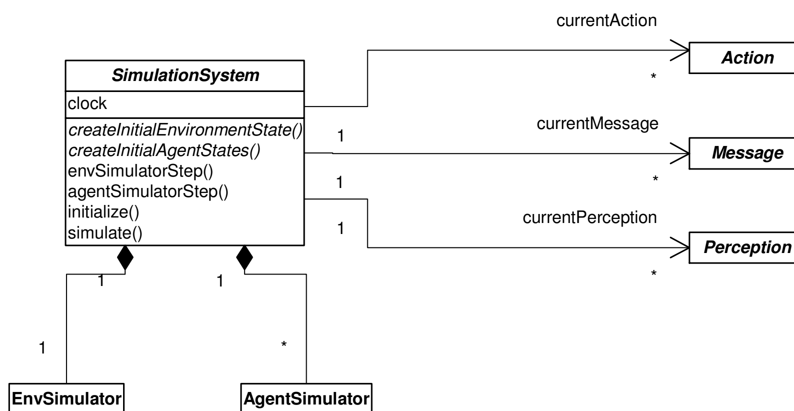

Abbildung 71: UML-Klassendiagramm für das Package *absimulation* im Simulationssystem *ABSim*

8.3.2 Package absimulation.theinterface

Im Package *absimulation.theinterface*[92], dessen Entwurf in Abbildung 72 dargestellt ist, befinden sich die Typen der Klassen, die zwischen dem Umgebungssimulator und den Agentensimulatoren ausgetauscht werden. Das sind die Klassen *Action* für Aktionen, *Perception* für Wahrnehmungen und *Message* für Nachrichten. Ein spezielles Ereignis, das das Ausführen einer Aktion darstellt, ist ein *ActionEvent*, eine spezielle Wahrnehmung, die den Empfang einer Nachricht darstellt ist eine *MessagePerception*.

[92] Der Name *theinterface* wurde gewählt, weil der Name *interface* als Packagename in Java nicht zulässig ist.

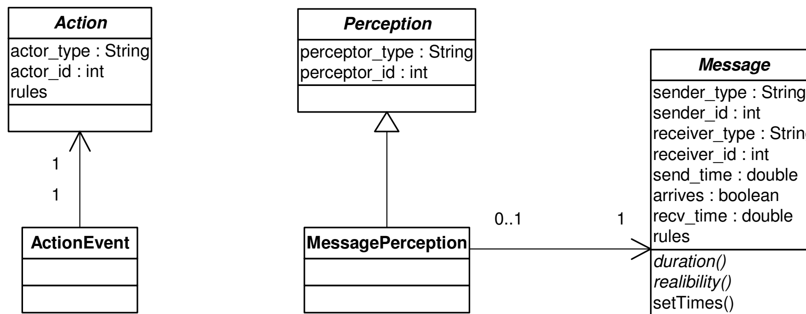

Abbildung 72: UML-Klassendiagramm für das Package *absimulation.theinterface* im Simulationssystem *ABSim*

8.3.3 Package absimulation.envsimulator

Der Umgebungssimulator und alle für die Ausführung der Simulation auf Umgebungssimulatorseite erforderlichen Klassen befinden sich im Package *absimulation.envsimulator*, dessen Entwurf in Abbildung 73 dargestellt ist. Der Umgebungssimulator verwaltet eine Menge von (externen) Agenten und von Objekten (Klasse *SimAgent* bzw. *SimObject*), eine Menge von zukünftigen Folgeereignissen *resulting_events*, von (Strängen von) exogenen Ereignissen *exogenous_events* sowie von zukünftig bei ihrem Empfänger ankommenden Nachrichten *future_messages*.

Wichtigste Methode ist *envSimulatorStep()*, die vom Simulationssystem aufgerufen wird. In ihr wird zunächst die private Methode *currentEvents()* aufgerufen, die die aktuellen Ereignisse dieses Zyklus ermittelt. Anschließend wird die Aufgabe der Ausführung der aktuellen Ereignisse an die entsprechenden *EnvEvent*-Objekte delegiert, die dann wiederum Reaktionsregeln vom Typ *EnvRule* anwenden, deren konkrete Realisierungen Unterklassen des abstrakten Typs *EnvRule* sind. Die subsummierten Folgeereignisse werden zu den *resulting_events* hinzugefügt und die subsummierten Wahrnehmungen als Ergebnis zurückgegeben.

Ereignisse haben den Typ der abstrakten Klasse *EnvEvent*, die zwei ebenfalls abstrakte Unterklassen *ExogenousEvent* und *ResultingEvent* hat sowie zwei konkrete Unterklassen, zum einen *ActionEvent*, die die Ausführung einer Aktion durch einen Agenten darstellt sowie zum anderen *MessageEvent*, die den Empfang einer Nachricht durch einen Agenten darstellt. MessageEvent hat aufgrund der 20festgelegten Reaktion des Umgebungssimulators auf Nachrichten eine feste Reaktionsregel vom Typ *MessageRule*, die keinen Zustandseffekt hat, keine Folgeereignisse erzeugt und eine *MessagePerception* für den Empfänger der Nachricht erzeugt.

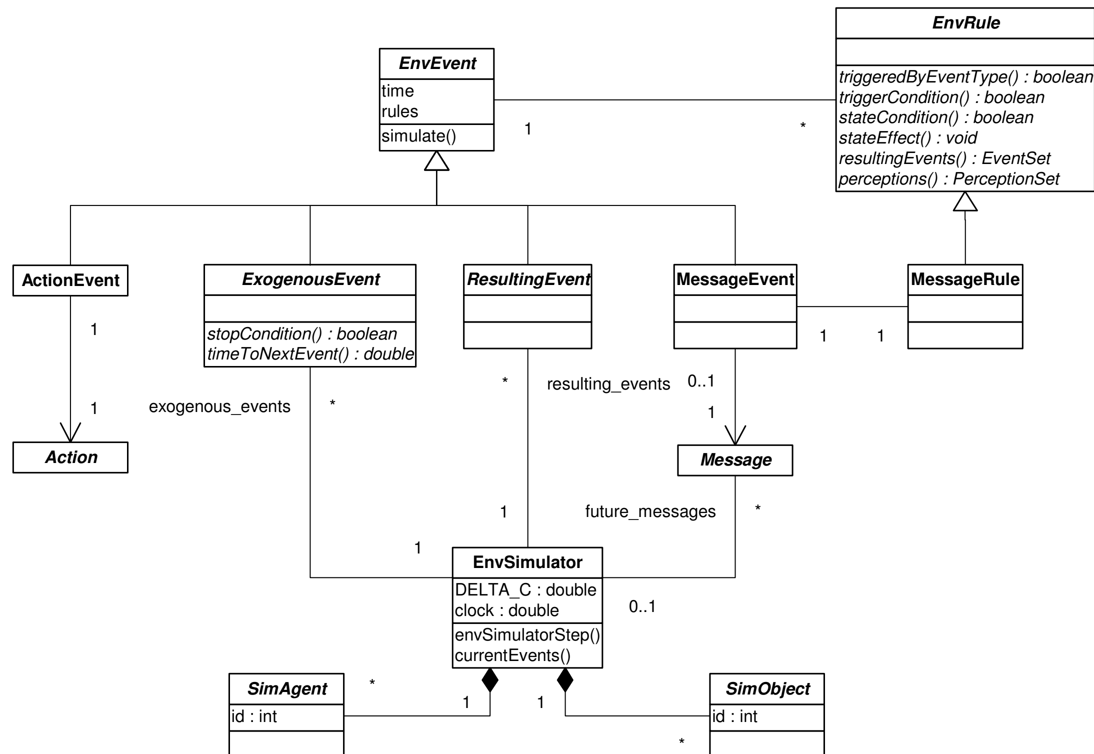

Abbildung 73: UML-Klassendiagramm für das Package *absimulation.envsimulator* im Simulationssystem *ABSim*

8.3.4 Package absimulation.agentsimulator

Die Agentensimulatoren und alle für die Ausführung der Simulation auf Agentensimulatorenseite erforderlichen Klassen befinden sich im Package *absimulation.agentsimulator*, dessen Entwurf in Abbildung 74 dargestellt ist. Ein Agentensimulator hat eine Referenz auf den von ihm simulierten (internen) Agenten (Klasse *InternalAgent*) und verwaltet eine Menge von zukünftigen Folge-Zeitereignissen *resulting_time_events* und von (Strängen von) periodischen Zeitereignissen *periodical_time_events*.

Wichtigste Methode ist *agentSimulatorStep()*, die vom Simulationssystem aufgerufen wird. In ihr wird zunächst die private Methode *currentEvents()* aufgerufen, die die aktuellen internen Ereignisse dieses Zyklus ermittelt. Anschließend wird die Aufgabe der Ausführung der aktuellen Ereignisse an die entsprechenden *InternalEvent*-Objekte delegiert, die dann wiederum Reaktionsregeln vom Typ *AgentRule* anwenden, die dann wiederum Reaktionsregeln vom Typ *AgentRule* anwenden. Die subsummierten Folge-Zeitereignisse werden zu den *resulting_time_events* hinzugefügt und die subsummierten Nachrichten und subsummierten Aktionen als Ergebnis zurückgegeben.

Interne Ereignisse haben den Typ der abstrakten Klasse *InternalEvent*, die drei ebenfalls abstrakte Unterklassen *PeriodicalTimeEvent*, *ResultingTimeEvent* und *Perception* hat, wobei *Perception* mit *MessagePerception* eine konkrete Unterklasse hat, die eine spezielle Wahrnehmung, den Empfang einer Nachricht, darstellt.

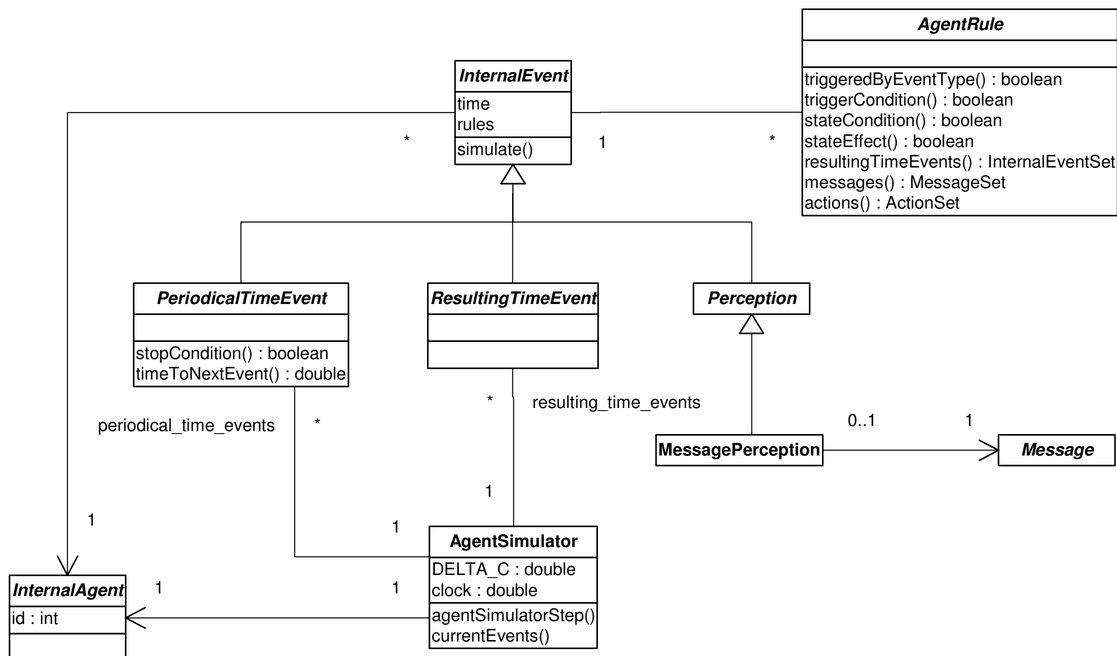

Abbildung 74: UML-Klassendiagramm für das Package *absimulation.agentsimulator* im Simulationssystem *ABSim*

8.4 Spezifikation mittels UML[93]

Wie bereits in Abschnitt 7.1 vorgestellt lassen sich Agentenbasierte Simulationssysteme visuell in Form eines UML-Profils spezifizieren. Das Prinzip des hier vorgestellten Simulationssystems besteht nun darin, mittels eines UML-Tools das Simulationssystem gemäß des UML-Profils zu spezifizieren und in Form des XML-basierten Formats XMI zu exportieren, das dann weiterverarbeitet werden kann. Was sich in der Theorie einfach anhört, hat sich in der Praxis leider als schwierig erwiesen, da nicht alle UML-Tools den UML-Standard komplett unterstützen und nicht immer alle Elemente des Modells sich auch im XMI-Export wiedergefunden haben. In diesem Abschnitt werden deshalb vorhandene Tools auf ihre Tauglichkeit untersucht und Einschränkungen formuliert, die sich durch die Verwendung dieser Tools ergeben.

In Abschnitt 8.4.1 werden zunächst die Anforderungen an die UML-Tools aufgezählt. In Abschnitt 8.4.2 werden dann vier spezielle UML-Tools, die besonders vielversprechend erschienen vorgestellt und auf die Erfüllung der Anforderungen hin untersucht. Abschnitt 8.4.3 fasst die Ergebnisse zusammen, stellt das Ergebnis der Toolauswahl dar und begründet sie. In Abschnitt 8.4.4 werden Einschränkungen formuliert, die sich aufgrund der Toolauswahl ergeben.

8.4.1 Anforderungen an die UML-Tools

In diesem Abschnitt werden die Anforderungen, die für die Realisierung des Simulationssystems erforderlich sind, aufgezählt. Dabei sind nur die Anforderungen genannt, die einerseits kritisch sind, also nicht ohnehin von jedem UML-Tool erfüllt werden und andererseits für die Spezifikation des Simulationssystems besonders wichtig sind.

8.4.1.1 n-äre Assoziationen

Um die Regeln für den Umgebungssimulator und die Agentensimulatoren darzustellen, müssen Assoziationen mit beliebig vielen Enden möglich sein, da es beliebig viele erzeugte Folgeereignisse und Wahrnehmungen auf Seiten des Umgebungssimulators bzw. Nachrichten, Aktionen und interne Folge-Zeitereignisse auf Seiten des Agentensimulators geben kann.

[93] Die Ergebnisse dieses Abschnitts basieren auf [Nguy04].

8.4.1.2 Objekte im Klassendiagramm

Objekte müssen in einem separaten Objektdiagramm oder innerhalb eines Klassendiagramms spezifiziert werden können. Das ist auf Seite des Umgebungssimulators erforderlich, um den Umgebungszustand, also die Menge der vorhandenen Agenten und Objekte, ihre initialen Zustände sowie ferner die Stränge von exogenen Ereignissen darzustellen. Auf der Seite der Agentensimulatoren ist für die initialen internen Agentenzustände sowie die Stränge von periodischen Zeitereignissen erforderlich.

8.4.1.3 Selbstdefinierte Stereotypen

Um Klassen, Objekte und Assoziationsenden mit den Stereotypen <<Agent>>, <<Object>>, <<Message>>, <<Sender>>, <<Receiver>>, <<EnvironmentalEvent>>, <<ExogenousEvent>>, <<ActionEvent>>, <<Actor>>, <<PerceptualEvent>>, <<Perceptor>>, <<Trigger>>, <<InvolvedEntity>>, <<Perception>> und <<ResultingEvent>> auf Umgebungssimulatorseite bzw. <<IMAgent>>, <<TimeEvent>>, <<PeriodicalTimeEvent>>, <<PerceptualEvent>>, <<Perceptor>>, <<OutgoingMessage>>, <<Receiver>>, <<Trigger>>, <<Reactor>>, <<ResultingAction>> und <<ResultingTimeEvent>> auf Agentensimulatorseite versehen zu können, ist es erforderlich, im UML-Tool Stereotypes selbst definieren zu können.

8.4.1.4 OCL-Unterstützung

Um auf der Seite des Umgebungssimulators bei Reaktionsregeln die Ereignisbedingung, Zustandsbedingungen, Zustandseffekte sowie die Parameter und den Zeitpunkt von Folgeereignissen und Wahrnehmungen festlegen zu können, ist es erforderlich, OCL-Constraints den einzelnen Assoziationsenden zuweisen zu können. Für Assertions bei Ereignis- und Aktionstypen müssen ferner OCL-Constraints auch Klassen zuordenbar sein.

Auf der Seite der Agentensimulatoren wird OCL-Unterstützung bei den Assertions für Wahrnehmungs- und Nachrichtentypen benötigt sowie bei Reaktionsregeln für die Ereignisbedingung, für die Zustandsbedingung und den Zustandseffekt, sowie die Parameter und den Zeitpunkt von internen Folge-Zeitereignissen, Nachrichten und Aktionen.

8.4.1.5 Vollständiger XMI-Export

Um das in UML formulierte Simulationsmodell weiterverarbeiten zu können, ist es erforderlich, dass das Modell in das Format XMI exportiert werden kann. Im exportierten Modell müssen alle für die Simulation relevanten Informationen des Modells enthalten sein.

8.4.2 Untersuchte UML-Tools

In diesem Abschnitt werden mit Rational Rose, Together, ArgoUML und Microsoft Visio vier spezielle UML-Tools vorgestellt und auf die Erfüllung der Anforderungen hin untersucht.

8.4.2.1 Rational Rose

IBM Rational Rose [Rat04] ist ein sehr verbreitetes und mächtiges Werkzeug zur Softwaremodellierung.

Es sind bei Rational Rose nur binäre, aber keine höherwertigen Assoziationen möglich. Objekte können zwar im Sequenz- bzw. Kollaborationsdiagramm, nicht aber im Klassendiagramm modelliert werden. OCL-Constraints können zwar formuliert werden, werden aber nicht auf syntaktische und semantische Korrektheit überprüft. Im XMI-Export erscheinen die spezifizierten Constraints nicht.

8.4.2.2 Together

Together Control Center [Bor04] ist ein Softwaremodellierungstool mit einer integrierten Entwicklungsumgebung der Firma Borland.

Es sind bei Together wie bei Rational Rose nur binäre, aber keine höherwertigen Assoziationen möglich. Objekte können ebenfalls zwar im Sequenz- bzw. Kollaborationsdiagramm, nicht aber im

Klassendiagramm modelliert werden. Auch OCL-Constraints können zwar formuliert werden, werden aber nicht auf syntaktische und semantische Korrektheit überprüft. Im XMI-Export erscheinen die spezifizierten Constraints nicht.

8.4.2.3 ArgoUML

ArgoUML [Tig04] ist eine in Java geschriebene Open-Source-Software zur UML-Modellierung.

Es sind bei ArgoUML nur binäre, aber keine höherwertigen Assoziationen möglich. Objekte können zwar im Sequenz- bzw. Kollaborationsdiagramm, nicht aber im Klassendiagramm modelliert werden. OCL-Constraints werden mit einem Syntaxchecker auf Gültigkeit überprüft und sind auch im XMI-Export enthalten.

8.4.2.4 Microsoft Visio

Visio [Mic04] ist ein Visualisierungs-, Zeichen- und Modellierungstool der Firma Microsoft, das durch ein UML-Add-On auch die Modellierung in UML ermöglicht.

Bei Visio sind Assoziationen mit maximal acht beteiligten Klassen möglich. Objekte können sowohl im Sequenz- bzw. Kollaborationsdiagramm als auch im Klassendiagramm modelliert werden. Bei Constraints kann zwar formuliert werden, dass sie in OCL formuliert sind, sie werden aber nicht auf syntaktische und semantische Korrektheit überprüft. Visio bietet keinen Export des UML-Modells nach XMI an.

8.4.3 Übersicht und Auswahl

Die folgende Tabelle (Abbildung 75) bietet eine Übersicht über die Erfüllung der Anforderungen bei den untersuchten UML-Tools. Eingeklammerte Häkchen bedeuten dabei, dass eine Anforderung zumindest teilweise erfüllt wird.

	Rational Rose	**Together**	**ArgoUML**	**Microsoft Visio**
n-äre Assoziationen	- (n=2)	- (n=2)	- (n=2)	√ ($n \leq 8$)
Objekte im Klassendiagramm	-	-	-	√
Selbstdefinierte Stereotypen	√	√	√	√
OCL-Unterstützung	-	-	√	(√)
Vollständiger XMI-Export	(√)	(√)	√	-

Abbildung 75: Erfüllung der Anforderungen bei den untersuchten UML-Tools

Microsoft Visio erfüllt als einziges der untersuchten Tools die ersten beiden Anforderungen, kommt aber trotzdem nicht in Frage, da es die wichtigste Anforderung, vollständigen XMI-Export zu leisten, nicht erfüllt. Als einziges der untersuchten Tools leistet dies *ArgoUML*, das zugleich auch die beste OCL-Unterstützung bietet. Deshalb wurde *ArgoUML* als einziges verwendbares Modellierungstool für die Implementierung ausgewählt.

8.4.4 Einschränkungen aufgrund der Auswahl

Das Ergebnis aus Abschnitt 8.4.3 ist ernüchternd, da man doch zumindest in der Theorie hätte erwarten können, jedes einigermaßen komplexe UML-Tool für die Agentenbasierte Simulation verwenden zu können. In der Praxis bot sich dagegen mit *ArgoUML* nur ein einziges Tool an und auch dieses erfüllt nicht alle Anforderungen vollständig. Deshalb kann ein Modellierer nicht einfach das vorgestellte UML-Profil aus Abschnitt 7.1 verwenden, sondern muss bei einigen Modellelementen die

eingeschränkten Fähigkeiten des verwendeten Tools berücksichtigen, was in diesem Abschnitt vorgestellt wird.

8.4.4.1 Instanzen von Agenten und Objekten

In *ArgoUML* ist es nicht möglich, ein Objektdiagramm anzulegen oder Objekte in ein Klassendiagramm einzufügen. Man muss sich damit behelfen, für jede Instanz eines Agententypen bzw. eines Objekttypen, die modelliert werden soll, statt eines Objekts (wie in Abbildung 40 bzw. Abbildung 48 in Abschnitt 7.1 dargestellt) eine eigene Klasse anzulegen. Der Name der Klasse setzt sich aus dem Namen der Instanz, gefolgt von einem Doppelpunkt und dem Namen der Klasse, zu der die Instanz gehört zusammen. Dies setzt voraus, dass weder der gewählte Name der Instanz noch der Name der Klasse einen Doppelpunkt enthalten, was jedoch keine echte Einschränkung ist, da z.B. die Programmiersprache Java keinen Doppelpunkt in Klassennamen erlaubt und allgemein die Verwendung eines Doppelpunkts innerhalb eines Klassennamens nicht besonders sinnvoll ist. *ArgoUML* erlaubt dies aber und so kann ein Benutzer dort durch die Verwendung eines Doppelpunkts im Klassennamen ausdrücken, dass es sich in Wirklichkeit um die Modellierung einer Instanz handelt. Initialwerte der Instanz des Agenten- bzw. Objektstyps sowie die ID des Agenten bzw. des Objekts werden als Initialwerte der als Hilfskonstruktion eingefügten Klasse modelliert. Abbildung 76 zeigt links einen (externen) Agenten als Objekt im UML-Objektdiagramm, wie er auch in Abbildung 40 in Abschnitt 7.1 vorkommt und rechts den entsprechenden Agenten als Hilfskonstruktion als Klasse im UML-Klassendiagramm. Für Objekte aus dem Umgebungszustand und für interne Agenten gilt die gleiche Hilfskonstruktion.

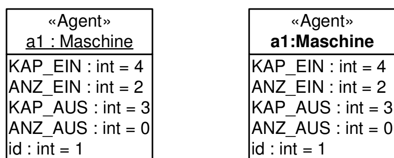

Abbildung 76: Ein Agent im UML-Objektdiagramm und im UML-Klassendiagramm

8.4.4.2 Stränge von exogenen Ereignissen und periodischen Zeitereignissen

Für Stränge von exogenen Ereignissen und Stränge von periodischen Zeitereignissen, die wie in Abschnitt 7.1 dargestellt als Objekte im UML-Objektdiagramm dargestellt werden sollen, gilt sinngemäß die gleiche Hilfskonstruktion, die bereits in Abschnitt 8.4.4.1 für Agenten und Objekte vorgestellt wurde. Sie werden durch eine Klasse dargestellt, deren Name sich aus einem Namen des Stranges, einem Doppelpunkt und dem Namen des exogenen Ereignistyps bzw. des periodischen Zeitereignistyps zusammensetzt.

8.4.4.3 Reaktionsregeln

In *ArgoUML* ist es nicht möglich, mehrstellige Assoziationen zu erstellen, was aber für die Spezifikation von Reaktionsregeln für den Umgebungssimulator und die Agentensimulatoren notwendig ist. Man behilft sich damit, dass man die n-äre Assoziation durch n binäre Assoziationen ersetzt, deren eines Ende jeweils eine neu zu erstellende Klasse ist, die der Reaktionsregel entspricht. Diese Klasse wird mit dem Stereotype <<EnvRule>> bzw. <<AgentRule>> versehen. Abbildung 77 zeigt, wie die Regel aus Abbildung 38 durch n binäre Assoziationen umgesetzt wird.

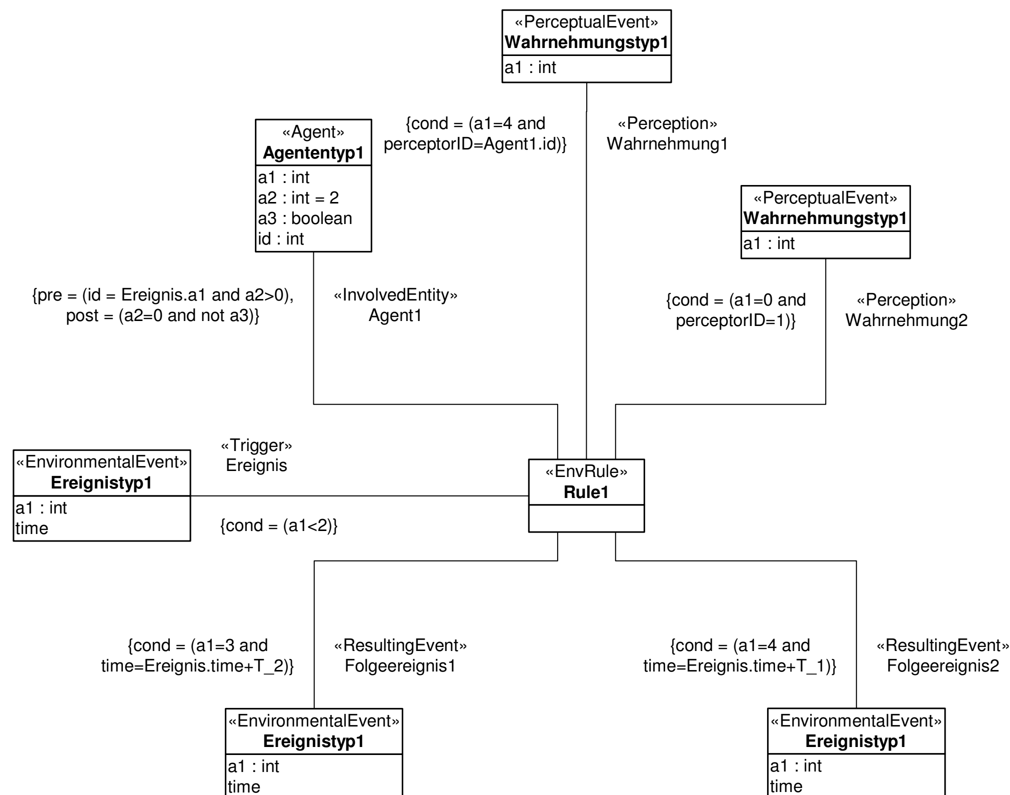

Abbildung 77: Eine Regel für den Umgebungssimulator im UML-Klassendiagramm mit binären Assoziationen

8.5 Das Format ABSimML[94]

In diesem Abschnitt wird das XML-basierte Format *ABSimML* (*Agent Based Simulation Markup Language*), das eine vollständige Repräsentation eines konkreten Agentenbasierten Simulationssystems darstellt, vorgestellt.

ABSimML ist zugleich der Name des Wurzelknotens für das XML-Dokument. Neben den üblichen Subelementen *title*, *author*, *creationDate* und *version* gibt es *classes*, *initialState* und *rules*. Abbildung 78 zeigt den Wurzelknoten mit seinen Subelementen grafisch in einer Baumdarstellung. Das Subelement *classes* beinhaltet Entitäts- und Ereignistypen und wird in Abschnitt 8.5.1 beschrieben. Der (initiale) Systemzustand wird im Subelement *initialState* spezifiziert und in Abschnitt 8.5.2 vorgestellt. Schließlich werden die vorhandenen Regeln (Subelement *rules*) im Abschnitt 8.5.3 behandelt.

Abbildung 78: Wurzelknoten im Format *ABSimML*

8.5.1 Entitäts- und Ereignistypen

Das Subelement *classes* stellt einen Container für alle an der Simulation beteiligten Entitäts- und Ereignistypen dar. Das sind Objekt-, Agenten-, Ereignis-, Wahrnehmungs-, Aktions- und Nachrichtentypen, die mit den Tags *object*, *agent* bzw. *internalAgent*, *event* bzw. *internalEvent*,

[94] Die Ergebnisse dieses Abschnitts basieren auf [Nguy04].

perception, *action* und *message* versehen werden. Die Entitätstypen (*object*, *agent*, *internalAgent*) haben dabei einen Namen (*name*) und beliebig viele Attribute (*attribute*), die wiederum einen Namen (*name*), einen Typ (*type*) und optional einen Defaultwert für den initialen Wert (*defaultvalue*) besitzen. Die Ereignistypen (*event*, *internalEvent*, *perception*, *action* und *message*) haben einen Namen (*name*) und beliebig viele Parameter (*parameter*), die wiederum einen Namen (*name*) und einen Typ (*type*) besitzen. Bei *internalEvent*, *perception*, *action* und *message* ist ferner angegeben, zu welchem bzw. welchen Agententypen sie gehören. Bei *event* und *internalEvent* wird durch einen Stereotyp (stereotype) zwischen exogenen Ereignissen und Folgeereignissen bzw. zwischen periodischen Zeitereignissen und Folge-Zeitereignissen unterschieden. Abbildung 79 zeigt die beschrieben Subelemente des Subelements *classes* grafisch in einer Baumdarstellung.

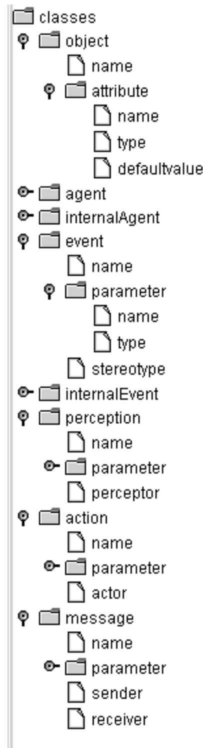

Abbildung 79: Subelement *classes* im Format *ABSimML*

8.5.2 Initialer Systemzustand

Das Subelement *initialState* beschreibt den initialen Systemzustand. Das sind die vorhandenen Agenten mit ihrem initialen externen und internen Zustand, die vorhandenen Objekte mit ihrem initialen Zustand, die Stränge von exogenen Ereignissen und für jeden Agenten die Stränge von periodischen Zeitereignissen. Sie werden mit den Tags *agentInstance*, *internalAgentInstance*, *objectInstance*, *eventInstance* und *internalEventInstance* versehen. Jede Instanz besitzt dabei ein Attribut *instanceOf*, um zu kennzeichnen, zu welchem Typ sie gehört. Die Ereignisinstanzen (Ereignisstränge) enthalten dabei Parameter (*parameter*), die wiederum einen Namen (*name*) und einen Wert (*value*) besitzen. Dabei müssen alle Parameter, die beim entsprechenden Ereignistyp aufgezählt sind, berücksichtigt werden. Die Entitätsinstanzen enthalten Attribute (*attribute*), die wiederum einen Namen (*name*) und einen Initialwert (*value*) besitzen. Dabei müssen alle Attribute, die beim entsprechenden Entitätstyp aufgezählt sind und keinen Defaultwert besitzen, berücksichtigt werden. Abbildung 80 zeigt die beschrieben Subelemente des Subelements *initialState* grafisch in einer Baumdarstellung.

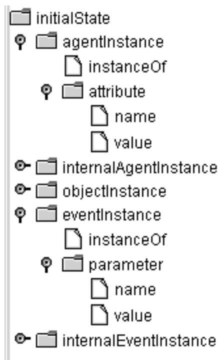

Abbildung 80: Subelement *initialState* im Format *ABSimML*

8.5.3 Regeln

Das Subelement *rules* beschreibt die im System verwendeten Regeln. Das sind zum einen Regeln für den Umgebungssimulator (*envRule*) und zum anderen Regeln für die Agentensimulatoren (*agentRule*). Eine Regel für den Umgebungssimulator besitzt ein die Regel triggerndes Ereignis (*trigger*), eine Ereignisbedingung (*triggercondition*), beliebig viele betroffene Entitäten (*involvedEntity*), beliebig viele Folgeereignisse (*resultingEvent*) und beliebig viele Wahrnehmungen (*perception*). Jede betroffene Entität hat dabei eine Zustandsbedingung (*precondition*) und einen Zustandseffekt (*postcondition*). Jedes Folgeereignis und jede Wahrnehmung hat eine Bedingung, mit der seine Parameter beschrieben werden (*resultingEventCond* bzw. *perceptionCond*). Durch die Wahrnehmungsbedingung wird außerdem beschrieben, für welchen Agenten sie gilt.

Eine Regel für einen Agentensimulator besitzt den Agententyp, zu dem die Regel gehört (*agent*), ein die Regel triggerndes Ereignis (*trigger*), eine Ereignisbedingung (*triggercondition*), eine Zustandsbedingung (*precondition*), einen Zustandseffekt (*postcondition*), beliebig viele Folge-Zeitereignisse (*resultingTimeEvent*), beliebig viele Aktionen (*action*) und beliebig viele ausgehende Nachrichten (*outgoingMessage*). Jedes Folge-Zeitereignis, jede Aktion und jede Nachricht hat eine Bedingung, mit der seine Parameter beschrieben werden (*resultingTimeEventCond*, *actionCond* bzw. *messageCond*). Durch die Nachrichtenbedingung wird außerdem beschrieben, welcher Agent der Empfänger ist. Abbildung 81 zeigt die beschrieben Subelemente des Subelements *rules* grafisch in einer Baumdarstellung.

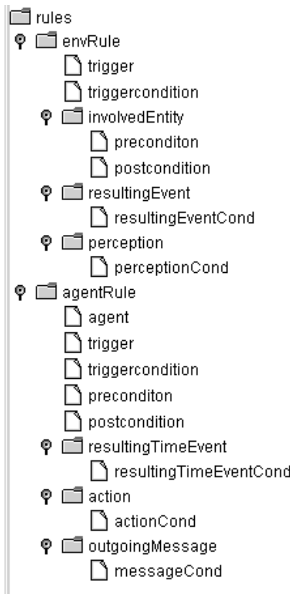

Abbildung 81: Subelement *rules* im Format *ABSimML*

8.6 Transformation von XMI nach ABSimML[95]

In diesem Abschnitt wird aufgezeigt, wie man aus einem im Format XMI [OMG04b] vorliegenden UML-Modell, das ein gemäß Abschnitt 8.4 spezifiziertes Agentenbasiertes Simulationssystem darstellt, mit Hilfe von XSLT in das in Abschnitt 8.5 vorgestellte Format *ABSimML* transformiert. Bei der Transformation müssen die relevanten Informationen im XMI-Dokument gesucht und in die diesen Informationen entsprechenden Elemente in ABSimML abgebildet werden.

In XMI werden die Elemente eines UML-Modells mit IDs versehen, um von mehreren Stellen des Dokuments referenziert werden zu können. Damit kann redundante Informationsspeicherung vermieden werden. Allerdings stellt sich dadurch bei der Transformation das Problem, zusammengehörige Elemente zu finden. Denn die Hauptaufgabe der Transformation ist es, über das XMI-Dokument verteilte Informationen, die zu einer logischen Einheit (z.B. einem Entitäts-, einem Ereignistyp oder einer Regel) gehören, an einer Stelle im ABSimML-Dokument zusammenzufassen. Abbildung 82 stellt dies grafisch dar. Durch die XPath-Funktion *key()* kann man dies lösen, ohne das Dokument bzw. ein Subelement mehrmals durchlaufen zu müssen.

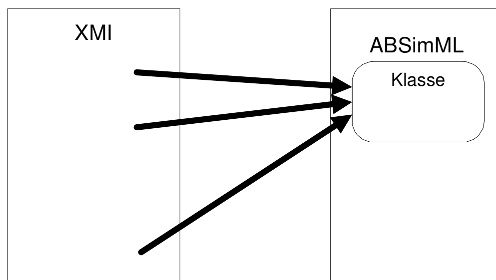

Abbildung 82: Zusammenfassung von zusammengehörigen Informationen bei der Transformation von XMI nach ABSimML

Zur Anwendung der Transformation wird *Xalan* [Xalan04] eingesetzt, ein XSLT-Prozessor des *Apache XML-Projekts*. Da die Applikation in Java implementiert ist, wird hierbei die Java-Version des XSLT-Prozessors verwendet. Somit kann man Xalan innerhalb eines Javaprogramms aufrufen.

8.7 Transformation von ABSimML nach Java-Code[96]

In diesem Abschnitt wird gezeigt, wie man aus einem im Format *ABSimML* vorliegenden Agentenbasierten Simulationssystem Code erzeugt, der im Simulator *ABSim* lauffähig ist.

Da bereits im Format ABSimML in den einzelnen Container-Elementen eine Gruppierung vorgenommen wurde und die Informationen bereits im Gegensatz zum Format XMI strukturiert vorliegen, besteht die Hauptaufgabe dieses Arbeitsschritts darin, die Informationen an die Java-Syntax und an die Klassen des Kerns des Simulators *ABSim* anzupassen.

Die Implementierung benutzt *Xerces* [Xerces04], einen XML-Parser des *Apache XML-Projekts*. Mit Hilfe dieses Parsers wird das ABSimML-Dokument in eine DOM2-Repräsentation umgewandelt. Für jedes Klassen-Element werden Wrapperklassen zur Verfügung gestellt, um direkt auf den Inhalt zugreifen zu können. Im Falle einer eventuellen Änderung des XML-Schemas müssen so nur die betreffenden Wrapperklassen geändert werden. Eine Änderung der Klassenerzeugung ist hingegen nicht erforderlich. Durch die Wrapperklassen wird nicht nur sichergestellt, dass die Informationen über den Namen der Klasse, sowie deren Attribute und Werte vorliegen, sondern es werden auch spezielle Anforderungen überprüft. Dazu gehört z.B., dass Instanzen von Agenten einen Wert des Attributs *id* zugewiesen haben müssen.

Eine Klassenschablone (Template) gibt die Struktur einer Java-Klasse vor. Die Aufgabe der Klassenerzeugung besteht darin, fehlende Informationen in der Schablone aufzufüllen. Abgesehen vom Klassennamen und den Attributen können dies auch die Zugehörigkeit zu einem Package oder die Angabe einer Basisklasse sein.

[95] Die Ergebnisse dieses Abschnitts basieren auf [Nguy04].

[96] Die Ergebnisse dieses Abschnitts basieren auf [Nguy04].

Zur Anpassung an die Klassen des Kerns des Simulators *ABSim* gehört unter anderem auch die Erweiterung der entsprechenden Basisklasse. So erbt beispielsweise jedes exogene Ereignis von der Klasse *absimulation.envsimulator.ExogenousEvent*. Es muss darauf geachtet werden, dass die erbende Klasse ererbte Attribute, wie z.B. *time* für Ereignisse, zwar in ihrem Code verwendet, aber nicht neu definiert, da in Java zwar geerbte Methoden, aber nicht geerbte Attribute überschrieben werden können. Im Falle einer Neudefinition wäre das vom Simulator verwendete Attribut der Oberklasse verdeckt, so dass die Änderungen am gleichnamigen Attribut der Unterklasse ohne Effekt blieben.

Nach erfolgreichem Abschluss der Kapselung der Informationen werden die einzelnen Java-Klassen generiert. Mit Ausnahme der Bedingungen werden alle benötigten Informationen direkt aus den Wrapperklassen entnommen.

Die in OCL formulierten Bedingungen für Regeln sind ein schwieriger Transformationsschritt, da zwischen der Transformation von Zustandsbedingungen und Zustandseffekten zu unterscheiden ist. Speziell für Zustandseffekte gibt es keine triviale Vorgehensweise, beliebige in OCL formulierte Bedingungen in Java-Code zu transformieren.[97] Die Notwendigkeit der Unterscheidung lässt sich am besten anhand des Beispiels der Gleichheit erläutern. Hierbei wird eine in OCL formulierte Gleichheit der Form $a = e$ betrachtet, wobei a eine Variable und e einen Ausdruck darstellt. Innerhalb einer Vorbedingung verwendet, hätte die Gleichheit in Java den Charakter eines Vergleichs und würde als $a == e$ umgesetzt. Denn der Zweck einer Vorbedingung im Java-Code es, zu überprüfen, ob die Bedingung erfüllt ist und den entsprechenden booleschen Wert zurückzugeben. Innerhalb einer Nachbedingung hingegen hätte die Gleichheit in Java den Charakter einer Zuweisung und würde als $a = e$ umgesetzt. Der Zweck einer Nachbedingung im Java-Code ist es, sicherzustellen, dass die Bedingung anschließend erfüllt ist. Dies wird dadurch erreicht, dass die Variable per Zuweisung auf den gewünschten Wert gesetzt wird.

Um OCL-Ausdrücke besser bearbeiten und abbilden zu können, wurde ein Parser mit Hilfe des Parsergenerators *JavaCC* [Java04] erzeugt. Der Parser erkennt eine Untermenge von OCL, die in Abbildung 83 dargestellt wird.

<table>
<tr><td>Start</td><td>::=</td><td><LOGICALEXPRESSION></td></tr>
<tr><td>LOGICALEXPRESSION</td><td>::=</td><td><RELATIONALEXPRESSION> (("and" | "or" | "xor" | "implies") <RELATIONALEXPRESSION>)*</td></tr>
<tr><td>RELATIONALEXPRESSION</td><td>::=</td><td><ADDITIVEEXPRESSION> (("=" | ">" | "<" | ">=" | "<=" | "<>") <ADDITIVEEXPRESSION>)?</td></tr>
<tr><td>ADDITIVEEXPRESSION</td><td>::=</td><td><MULTIPLICATIVEEXPRESSION> (("-" | "+") <MULTIPLICATIVEEXPRESSION>)*</td></tr>
<tr><td>MULTIPLICATIVEEXPRESSION</td><td>::=</td><td><UNARYEXPRESSION> (("*" | "/") <UNARYEXPRESSION>)*</td></tr>
<tr><td>UNARYEXPRESSION</td><td>::=</td><td>(("-" | "not") <POSTFIXEXPRESSION> | <POSTFIXEXPRESSION>)</td></tr>
<tr><td>POSTFIXEXPRESSION</td><td>::=</td><td><PRIMARYEXPRESSION> (("." | "->") <PROPERTYCALL>)*</td></tr>
<tr><td>PRIMARYEXPRESSION</td><td>::=</td><td>(<LITERAL> | <PROPERTYCALL> | "(" <LOGICALEXPRESSION> ")")</td></tr>
<tr><td>LITERAL</td><td>::=</td><td>(<STRING> | <NUMBER>)</td></tr>
<tr><td>PROPERTYCALL</td><td>::=</td><td>(<NAME> ("::" <NAME>)*)</td></tr>
<tr><td>NAME</td><td>::=</td><td><CHAR> (<CHAR> | <DIGIT>)*</td></tr>
</table>

Abbildung 83: Grammatik der vom Parser erkannten OCL-Untermenge

[97] Da sich das allgemeine Problem auf das Erfüllbarkeitsproblem boolescher Formeln reduzieren lässt, ist es sogar NP-vollständig.

Im Folgenden wird anhand des Beispiels einer Regel aus dem FTS-Beispiel (Regel 85: Wahrnehmung_TSkommtAn.1) das Ergebnis der Transformation gezeigt. In Abbildung 84 sieht man, wie diese Regel im ABSimML-Dokument repräsentiert ist.

```
<agentRule>
        <name>TSkommtAn1</name>
        <agent>Maschine</agent>
        <trigger>TSKommtAn</trigger>
        <precondition> context TSkommtAn1 inv precondition: self.o1.prodZust = leerlauf
        </precondition>
        <postcondition> context TSKommtAn1 inv postcondition: self.o1.bestellteTS =
        self.o1.bestellteTS@pre and self.o1.prodZust = produzierend </postcondition>
        <resultingTimeEvent>
                <name>ueberpruefenBA</name>
                <resultingTimeEventCond>time=clock</resultingTimeEventCond>
        </resultingTimeEvent>
        <action>
                <name>aufnehmenTS</name>
        </action>
</agentRule>
```

Abbildung 84: Regel Wahrnehmung TSkommtAn.1 im Format ABSimML

Ein Ausschnitt des Java-Codes, der dieser Regel entspricht und der Ergebnis des Transformationsschritts ist, wird in Abbildung 85 dargestellt.

```
public class TSKommtAn1 extends AgentRule {
        public boolean triggeredByEventType (InternalEvent e) {
                return (e instanceof TSKommtAn);
        }
        public boolean triggerCondition (InternalEvent e) {
                return true;
        }
        public boolean stateCondition (InternalEvent e, InternalAgent a) {
                Maschine agent = (Maschine) a;
                return (agent.prodZust == leerlauf);
        }
        ...
}
```

Abbildung 85: Regel Wahrnehmung TSkommtAn.1 als Java-Code

Durch die in diesem Kapitel beschrieben Arbeitsschritte ist es somit möglich, ein visuell spezifiziertes Simulationsmodell (UML-Modell) automatisch in ein Simulationsprogramm (Java-Code) zu transformieren und dieses vom Simulator (ABSim) ausführen zu lassen.

8.8 Simulationsstudie Fahrerloser Transportsysteme

In diesem Abschnitt wird eine Simulationsstudie Fahrerloser Transportsysteme (FTS), die mit Hilfe des Simulators ABSim erstellt wurde, vorgestellt. Diese Studie soll zeigen, dass es möglich ist, mit dem Simulator komplexe Systeme von miteinander und mit ihrer Umwelt interagierenden Agenten zu simulieren.

8.8.1 Beschreibung Grundszenario

Untersucht wurde ein FTS in der Fertigungslogistik. Die Produktionshalle ist 250 m x 250 m groß und beinhaltet 18 Maschinen, ein Eingangslager und ein Ausgangslager. Der Fahrkurs besteht aus 36

quadratisch angeordneten Knoten, die durch 120 Segmente miteinander verbunden sind. In Abbildung 86 ist das Simulationsszenario grafisch dargestellt.

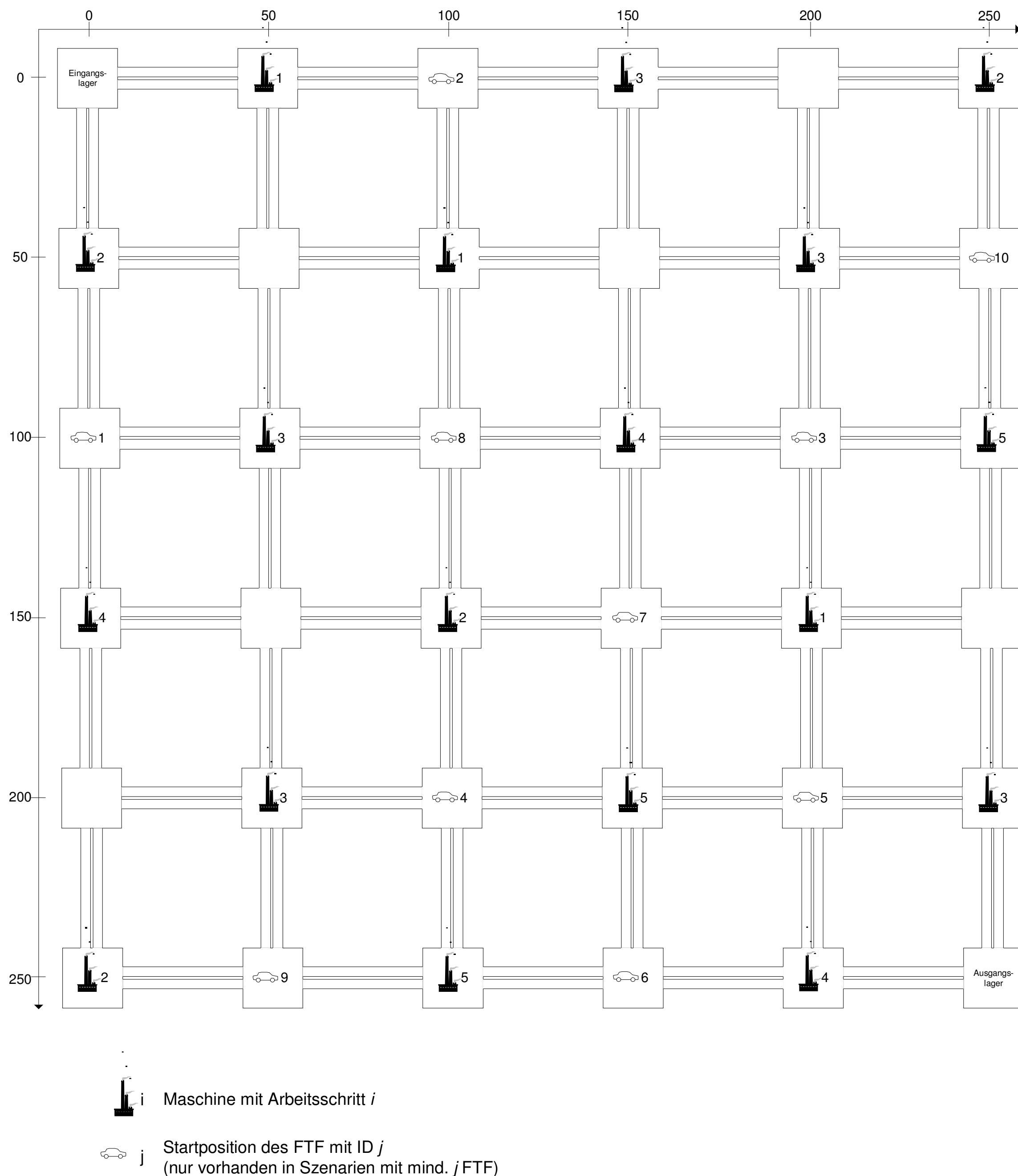

Abbildung 86: Grafische Darstellung des Simulationsszenarios

Für die Produktion werden fünf Arbeitsschritte benötigt. Offensichtlich scheint die Anordnung der Maschinen für die vorgestellte Produktion nicht optimal zu sein. Es wird hier jedoch unterstellt, dass die Anordnung der Maschinen fest ist und somit diesbezüglich keine Optimierungsmöglichkeiten bestehen. Dies kann etwa der Fall sein, wenn in täglich wechselnden Produktionsprozessen die Transportstücke die Maschinen in einer unterschiedlichen Reihenfolge durchlaufen oder wenn ein Umstellen der Maschinen technisch nicht möglich oder zu teuer ist.

Die Maschinen haben alle sowohl einen Eingangspuffer als auch einen Ausgangspuffer der Größe 5. Die Puffer sind im Initialzustand leer. Die Maschinen erteilen einen Beschaffungsauftrag, wenn sich nur noch 2 Transportstücke in ihrem Eingangspuffer befinden (Konstante C_EIN = 2). Sie erteilen einen Abholauftrag, wenn sich nur noch 2 freie Plätze in ihrem Ausgangspuffer befinden, also der Ausgangspuffer mit 3 Transportstücken belegt ist (Konstante C_AUS = 3). Die Bearbeitungszeit der Maschinen hängt vom Arbeitsschritt ab. Bei Schritt 1 ist das Intervall [230s;250s], bei Schritt 2 [350s;370s], bei Schritt 3 [470s;490s], bei Schritt 4 [310s;330s] und bei Schritt 5 [290s;310s].

Die FTF haben eine Geschwindigkeit von 5 m/s. Sie benötigen 2s, um über einen Knoten zu fahren (Konstante c_1), 1,5 s, um in einen Knoten einzufahren und stehen zu bleiben (Konstante c_2) und 1s, um aus einem Knoten auszufahren, wenn sie auf ihm stehen (Konstante c_3). Die Anzahl der FTF wurde zwischen 1 und 10 variiert. Ferner wurde das FTS mit 15 FTF und mit 20 FTF simuliert.

Es gibt einen Transportauftrags-Manager (TA-Manager), der für das gesamte FTS zuständig ist. Der TA-Manager schreibt die Transportaufträge für eine Dauer von 20s aus (Konstante c_{bewerb}). Kann aus einem Abholauftrag bzw. Beschaffungsauftrag kein Transportauftrag generiert werden, so versucht der TA-Manager dies nach 100s erneut (Konstanten c_{aa_neu} bzw. c_{ba_neu}).

Die Kommunikation zwischen den Agenten ist zuverlässig, die Kommunikationsdauer beträgt 0,05 s. Die Simulation läuft in Zyklen der Länge 0,1 s ab.

8.8.2 Ergebnisse Grundszenario

Simuliert wurde jeweils eine Produktionszeit von 24 Stunden, wobei gezählt wurde, wie viele fertig produzierte Transportstücke im Ausgangslager abgelegt wurden. In Abbildung 87 ist jeweils der Durchschnitt von 10 Simulationsläufen bei Verwendung von 1-10 FTF sowie von 15 und 20 FTF zu sehen.

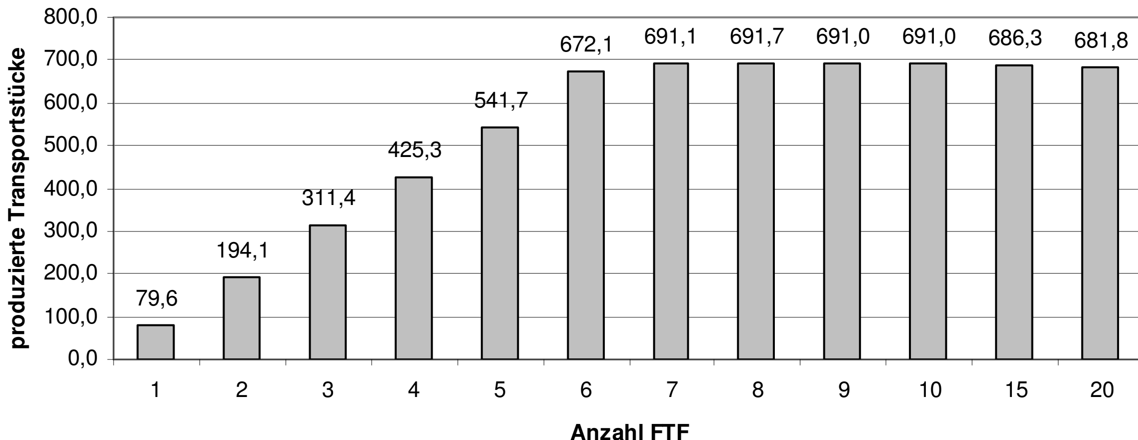

Abbildung 87: Anzahl der produzierten Transportstücke in Abhängigkeit von der Anzahl der FTF

Wie man sieht, wird das Maximum bei einer Anzahl an FTF zwischen 7 und 10 erreicht. Offensichtlich sind 7 FTF ausreichend, um das Maximum an Transportstücken, das aufgrund der Produktionsgeschwindigkeit der Maschinen notwendig ist, zu produzieren. Verwendet man sehr viele FTF (15 bzw. 20), dann sinkt die Anzahl der produzierten Transportstücke sogar wieder leicht, da sich die FTF dann teilweise gegenseitig behindern.

Aufgrund des dargestellten Ergebnisses scheint es sinnvoll zu sein, 7 FTF einzusetzen. Aus wirtschaftlichen Gründen kann es jedoch sinnvoll sein, unter dieser Zahl zu bleiben. Lässt sich etwa pro produziertem Transportstück ein Deckungsbeitrag von 10€ erzielen und entstehen durch Anschaffung und Betrieb eines FTF täglich Kosten in Höhe von 400€, so könnte mit 6 FTF (4.321 €) der beste Beitrag zum Betriebsergebnis erzielt werden. Bei 7 FTF betrüge der Beitrag zum Betriebsergebnis lediglich 4.111 €. Obwohl durch den Einsatz eines 7. FTF mehr Transportstücke produziert werden könnten, wäre die Anschaffung eines weiteren FTF unter diesen Annahmen nicht rentabel.

Bei der Simulation wurde ferner der zeitliche Verlauf der Anzahl der produzierten Transportstücke aufgezeichnet. Dies hat zum Ziel, die Dauer der Einschwingphase des Systems zu ermitteln. Abbildung 88 zeigt das Ergebnis bei Verwendung von 6 FTF.

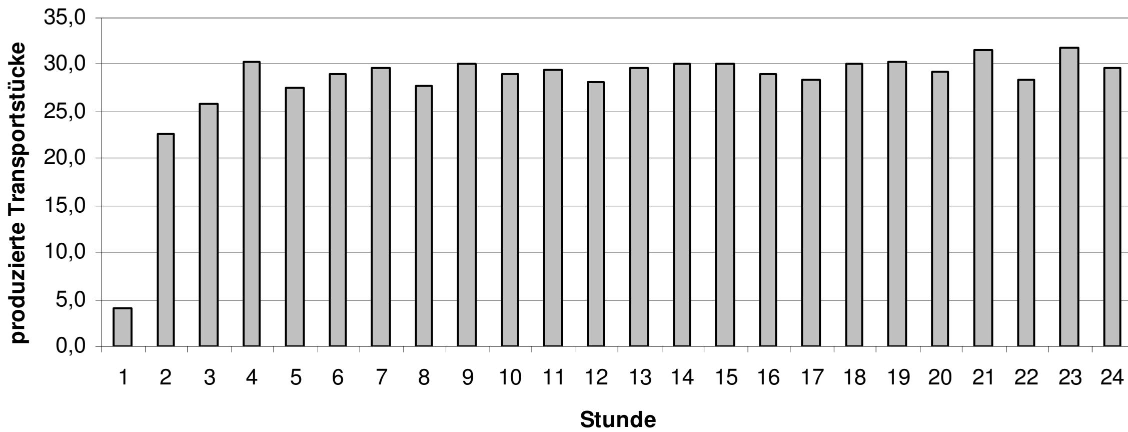

Abbildung 88: Produzierte Transportstücke pro Stunde bei Verwendung von 6 FTF

Da am Anfang die Puffer der Maschinen leer sind, dauert es eine Weile, ehe alle Maschinen arbeiten können. In den ersten 3 Stunden, insbesondere in der ersten Stunde, werden deshalb deutlich weniger produzierte Transportstücke im Ausgangslager abgelegt als in den verbleibenden 21 Stunden, in denen die Ergebnisse bis auf statistische Schwankungen konstant sind.

Beim Vergleich des zeitlichen Verlaufs bei 7 FTF und bei 20 FTF fällt auf, dass die geringere Anzahl an produzierten Transportstücken bei 20 FTF ausschließlich aus den ersten 3 Stunden resultiert. Da am Anfang alle Transportaufträge im Eingangslager starten, behindern sich viele FTF dort besonders stark.

8.8.3 Erweiterungen Grundszenario

In weiteren Simulationsläufen wurde untersucht, inwieweit die Effizienz des FTS weiter erhöht werden kann. Grundlage dieser Untersuchungen ist dabei stets das Grundszenario mit 6 FTF. Es gibt zwei Gründe, warum nicht 7 FTF als Grundlage gewählt wurden. Zum einen erscheint der Einsatz eines 7. FTF wirtschaftlich nicht sinnvoll, zum anderen besteht bei 7 FTF kein Optimierungsspielraum, da ja ohnehin dort die aufgrund der Produktionsgeschwindigkeit der Maschinen maximal mögliche Anzahl an Transportstücken erzielt wurde. Ziel ist es deshalb, auch mit 6 FTF dichter als bisher an das Maximum heranzukommen.

Schwellen der Maschinen

Im Grundszenario haben die Maschinen für C_EIN bzw. C_AUS die Werte 2 bzw. 3 verwendet, so dass ein Beschaffungsauftrag bzw. Abholauftrag erteilt wurde, wenn sich nur noch 2 Transportstücke im Eingangspuffer befunden haben bzw. nur noch 2 Plätze im Ausgangspuffer frei waren. In weiteren Simulationsläufen wurde untersucht, wie sich eine Änderung dieser Schwellwerte auswirkt. Zum einen wäre es denkbar, dass es sinnvoll ist, C_EIN zu senken und C_AUS zu erhöhen, damit Beschaffungsaufträge bzw. Abholaufträge erst dann erteilt werden, wenn es wirklich nötig ist. Damit könnte verhindert werden, dass ein Transportauftrag, der wichtig ist, damit eine Maschine weiter produzieren kann, nicht vergeben werden kann, weil noch andere Transportaufträge, die noch nicht dringend sind, ausgeführt werden. Zum anderen wäre es aber auch denkbar, dass es sinnvoll ist, C_EIN zu erhöhen und C_AUS zu senken, damit frühzeitig ein TA erteilt werden kann. Alle möglichen Werte für C_EIN und C_AUS wurden untersucht, jedoch nur in Kombinationen, in denen schrittweise der Zeitpunkt der Auftragserteilung vom spätestmöglichen zum frühestmöglichen nach vorne verlegt wurde.

C_EIN	C_AUS	produzierte Transport-stücke	
0	5	641,7	spätestmögliche Zeitpunkte der Auftragserteilung
1	4	654,9	
2	3	672,1	
3	2	670,5	
4	1	677,9	frühestmögliche Zeitpunkte der Auftragserteilung

Abbildung 89: Anzahl der produzierten Transportstücke in Abhängigkeit von den Schwellwerten der Maschinen

In Abbildung 89 sind einige Kombinationen für Werte von C_EIN und C_AUS mit ihren Ergebnissen aufgelistet. Wie man sieht ist es sinnvoll, einen Beschaffungsauftrag bzw. Abholauftrag so früh wie möglich zu erteilen. In der Kombination in der letzten Zeile (C_EIN = 4, C_AUS = 1) wird ein Beschaffungsauftrag bereits erteilt, sobald ein einziger Platz im Eingangspuffer frei ist. Ein Abholauftrag wird erteilt, wenn sich ein einziges Transportstück im Ausgangspuffer befindet.

Anzahl der TA-Manager

Im Grundszenario war ein TA-Manager für die Erteilung der Transportaufträge für das gesamte FTS zuständig. Durch die Verwendung mehrerer TA-Manager kann eine höhere Ausfallsicherheit erzielt werden. Es wurde untersucht, wie sich die Verwendung mehrerer TA-Manager auf die Effizienz des Systems auswirkt, wenn keine Ausfälle vorliegen. Dabei wurden Szenarien mit 2 und 4 TA-Managern untersucht. Abbildung 90 zeigt grafisch die Zuordnung der TA-Manager zu den Maschinen auf.

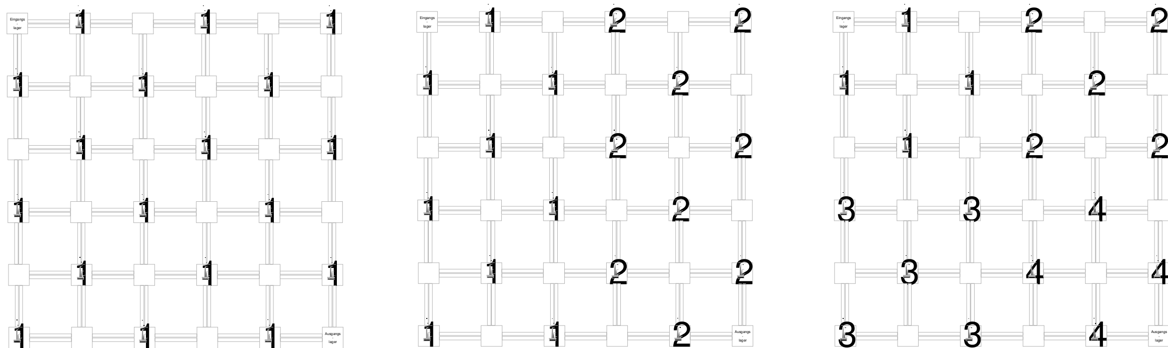

Abbildung 90: Zuordnung der TA-Manager zu den Maschinen bei 1, 2 und 4 TA-Managern

Dabei ist zu beachten, dass im Szenario mit 4 TA-Managern die Gebiete eines TA-Managers jeweils nicht den kompletten Produktionsprozess abdecken, so dass auch Maschinen, die von anderen TA-Managern bedient werden, zur Abgabe oder Annahme eines Transportstücks verwendet werden müssen.

Anzahl TA-Manager	produzierte Transport-stücke
1	672,1
2	669,0
4	669,1

Abbildung 91: Anzahl der produzierten Transportstücke in Abhängigkeit von der Anzahl der TA-Manager

In Abbildung 91 sind die Ergebnisse aufgelistet. Wie man sieht, sinkt die Effizienz bei mehreren TA-Managern leicht. Dies liegt insbesondere daran, dass sie über kein globales Wissen verfügen und dass Abholaufträge und Beschaffungsaufträge von Maschinen, die ihre Aufträge an unterschiedliche TA-Manager erteilt haben, nicht zu einem Transportauftrag zusammengefasst werden.

Austausch von Transportaufträgen[98]

Bei der im Grundszenario verwendeten Strategie ist es möglich, dass die Auftragsvergabe nicht optimal ist. So kann es zum einen vorkommen, dass ein FTF sich für einen Transportauftrag, dessen Startort sich in seiner unmittelbaren Nähe befindet, nicht beworben hat, weil es zum Zeitpunkt der Ausschreibung gerade belegt war. Zum anderen kann ein FTF den Zuschlag für mehrere Transportaufträge erteilt bekommen. Es führt dann stets den Auftrag aus, für den es zuerst den Zuschlag erteilt bekommen hat, auch wenn dieser Auftrag genauso gut von einem anderen FTF ausgeführt werden könnte, dies beim zweiten Auftrag aber nicht der Fall ist. Insbesondere bei mehreren TA-Managern ist eine nicht optimale Auftragsvergabe möglich.

In dieser Variante fragt jedes FTF, das einen Auftrag erteilt bekommen hat, bei den anderen FTF nach, ob sie ihrerseits gerade einen Transportauftrag ausführen und, wenn ja, ob sich die Gesamtbearbeitungszeit beider Aufträge verbessern würde, wenn die beiden FTF ihre Aufträge tauschen würden. Ist dies der Fall, so tauscht das FTF seinen Auftrag mit dem FTF, bei dem die größte Ersparnis erzielbar ist.

Es wurden die Szenarien mit 1, 2 und 4 TA-Managern daraufhin untersucht, ob sich ihre Effizienz durch den Austausch von Transportaufträgen verbessern lässt. Der Timeout der FTF für den Austausch beträgt dabei 1s.

Anzahl TA-Manager	produzierte Transportstücke ohne Austausch (6 FTF)	produzierte Transportstücke mit Austausch (6 FTF)	produzierte Transportstücke ohne Austausch (5 FTF)	produzierte Transportstücke mit Austausch (5 FTF)
1	672,1	690,0	541,7	574,4
2	669,0	690,0	542,3	574,6
4	669,1	690,5	542,0	578,1

Abbildung 92: Anzahl der produzierten Transportstücke mit und ohne Austausch von Transportaufträgen

In Abbildung 92 sind die Ergebnisse aufgelistet. Wie man sieht, lassen sich die Ergebnisse durch den Austausch von Transportaufträgen erheblich verbessern, so dass nun auch mit 6 FTF das Optimum an produzierten Transportstücken erreicht werden kann, für das sonst 7 FTF erforderlich sind. Zum Vergleich wurden auch Simulationsläufe mit 5 FTF durchgeführt, um zu sehen, ob durch den Austausch von Transportaufträgen auch hier das Optimum erreicht werden kann. Die Ergebnisse zeigen, dass auch hier eine Verbesserung erreicht werden kann, allerdings das Optimum nicht erreicht wird. In einem weiteren Simulationslauf (mit 6 FTF und 1 TA-Manager) wurde gezählt, wie oft es zum Tausch von Transportaufträgen gekommen ist. Bei 4297 erteilten Transportaufträgen wurde 951 mal getauscht (22,1%).

8.8.4 Zusammenfassung

In dieser Simulationsstudie wurde ein Fahrerloses Transportsystem über einen Zeitraum von 24 Stunden simuliert und als Ergebnis die Anzahl der produzierten Transportstücke ermittelt. Es wurde gezeigt, dass unter den gegebenen Annahmen das Optimum an produzierten Transportstücken bei Verwendung von mindestens 7 FTF erzielt werden konnte. Bei sehr vielen FTF sinkt das Ergebnis wieder, da sich die FTF vor allem in der Anfangsphase gegenseitig behindern. Mit 6 FTF konnten ca. 20 Transportstücke pro Tag weniger produziert werden. Lässt man zu, dass die FTF die Auftragsvergabe nachträglich optimieren, indem sie untereinander Transportaufträge tauschen, so lässt sich auch mit 6 FTF das optimale Ergebnis erzielen.

Diese Studie hat gezeigt, dass es möglich ist, mit dem Simulator komplexe Systeme zu simulieren, in denen autonome Agenten mittels Nachrichten miteinander kommunizieren, ihre Umwelt wahrnehmen, Aktionen ausführen, einen internen Zustand besitzen und autonom Entscheidungen ausführen. Für die

[98] Die Reaktionsregeln, die in dieser Variante neu hinzugekommen sind oder sich verändert haben, befinden sich im Anhang in Abschnitt B.11.

Domäne Fahrerloser Transportsysteme wurde ferner gezeigt, dass sich mit einer dezentralen Steuerung, in der die FTF sich bei TA-Managern um Transportaufträge bewerben und anschließend untereinander über den Austausch von Transportaufträgen verhandeln, effiziente Ergebnisse erzielen lassen.