

Universität Duisburg-Essen

Virtueller Weiterbildungsstudiengang Wirtschaftsinformatik (VAWi)

Projektarbeit in der Modulgruppe Entscheidungsunterstützung

Moise-DSL: Eine domänenspezifische Sprache zur Modellierung von Multiagentenorganisationen

MOISE-DSL: A domain-specific language for modelling multi-agent organisations

Vorgelegt der Fakultät für Wirtschaftswissenschaften der Universität Duisburg-Essen

Verfasser	Graffy, Silas Dr.-Gebauer-Str. 72a 55411 Bingen am Rhein 2247617
Prüfer:	Prof. Dr. Heimo Adelsberger
Tutor:	Jörg Rudnick
Abgabe:	06.10.2011

Zusammenfassung

Multiagentensysteme dienen der kollektiven Problemlösung mithilfe autonom agierender, jedoch miteinander interagierender, Programme, sogenannter Agenten.

Folglich eignen sich Multiagentensysteme in besonderem Maße für die Modellierung komplexer Systeme, die aus mehreren selbstständigen Bestandteilen bestehen. Beispiele hierfür sind Märkte, Organisationen, Insektenvölker, aber auch technische Systeme, deren Komponenten eigene Logik implementieren. Sie sind seit vielen Jahren Gegenstand der Forschung.

Der große Vorteil von Multiagentensystemen gegenüber „gewöhnlichen“, generischen Modellen besteht darin, dass sie Agenten und andere relevante Entitäten bereits als First-Class-Modellelemente kennen, d. h. dass ihnen deren Semantik klar ist. Um diesen Vorteil weiter auszubauen, wird die semantische Anreicherung bestehender Multiagentensysteme seit langem weiter vorangetrieben. Ansätze hierzu bestehen in der Abstraktion von Ressourcen, die den Agenten zu unterschiedlichsten Nutzungen zur Verfügung gestellt werden, zu sog. *Artefakten* sowie in der Agenten-unabhängigen Definition von *Multiagentenorganisationen*. Diese können analog zu Organisationsstrukturen betrachtet werden, die den Aufbau von von Menschen gebildeten Organisationen wie Firmen beschreiben.

Um die semantische Lücke zwischen Modellen und Implementierungen von Multiagentensystemen, hier speziell Multiagentenorganisationen, weiter zu schließen, Verständlichkeit der Modelle und Produktivität bei deren Erstellung zu steigern, stellt diese Arbeit MOISE-DSL vor, eine domänenspezifische Sprache zur Formulierung von Multiagentenorganisationen.

Auf Basis zugrunde gelegter, bereits bekannter Modelle für Multiagentenorganisationen wird eine geeignete Architektur für das Design der Sprache sowie ihre Integration in bestehende Multiagentensysteme vorgestellt. Den Schwerpunkt der vorliegenden Arbeit bildet die MOISE-DSL-Implementierung in Scala, einer objekt-funktionalen Hybridsprache, die sich in besonderem Maße für die gegebene Aufgabenstellung eignet. Abschließend werden die fachliche und technische Qualität der Umsetzung und ihre mögliche Bedeutung für die weitere Entwicklung von Multiagentensystemen bewertet.

Inhalt

1	Einführung	1
1.1	Zielsetzung und Motivation	1
1.2	Gliederung	2
2	Multiagentensysteme	4
2.1	Grundlagen	4
2.1.1	Agent	4
2.1.2	Agentenorientierte Programmierung	5
2.1.3	Belief-Desire-Intention	6
2.2	<i>Jason</i>	7
2.3	Agents and Artefacts	10
3	Multiagentenorganisationen	14
3.1	<i>MOISE</i>	14
3.2	<i>MOISE</i> ⁺	15
3.2.1	Struktur	15
3.2.2	Organisational Specification	16
3.2.3	Implementierungen	19
4	Domain-specific Languages	23
4.1	Motivation und Einsatz	23
4.2	Realisierungsmöglichkeiten	24
4.2.1	Interne Domain-specific Languages	24
4.2.2	Externe Domain-specific Languages	26
5	Modell und Architektur	28
5.1	Vorgehen	28
5.2	Designentscheidungen	29
5.2.1	Sprachcharakteristika und -funktionen	29

5.2.2	Integration in Multiagentensysteme	31
5.2.3	Vorbereitung der Implementierung	33
6	Implementierung	38
6.1	Grundlegende Konzepte	38
6.1.1	Allgemeines	38
6.1.2	Nutzung von Scala-Sprachfeatures und DSL-Konzepten	38
6.2	OML-Transformation	45
6.3	<code>moise.dsl</code> -Packages	46
6.3.1	Structural Specification: <code>moise.dsl.ss</code>	46
6.3.2	Functional Specification: <code>moise.dsl.fs</code>	49
6.3.3	Normative Specification: <code>moise.dsl.ns</code>	51
6.3.4	Organisational Specification und Sonstige	52
6.4	Konverter und Interpreter	53
6.4.1	Konverter	53
6.4.2	Interpreter	56
6.5	Testautomatisierung	58
6.5.1	Tests	58
6.5.2	Testabdeckung	60
6.6	Anpassungen an $\mathcal{M}\text{OISE}^+$	61
7	Bewertung und Ausblick	64
7.1	Zusammenfassung und Bewertung	64
7.1.1	Fachliche Zielerfüllung	64
7.1.2	Qualität der Umsetzung	64
7.2	Weiterentwicklungsmöglichkeiten und -bedarf	65
7.3	Multiagentensystementwicklung und -plattformen	67

Abbildungen

2.1	Beispiel für ein Grid View in <i>Jason</i> (eigenes Bildschirmfoto)	9
2.2	Interaktion zwischen Agenten-Architektur und Umwelt in <i>Jason</i> (Bordini et al., 2007, S. 103)	9
2.3	Mind Inspector in <i>Jason</i> (eigenes Bildschirmfoto)	10
2.4	Abstrakte Repräsentation und konkrete Instanzen eines Artefakts (Ricci et al., 2006, S. 209)	11
2.5	Schematische Übersicht eines MAS in Kombination mit <i>CARtAgO</i> (Ricci et al., 2008, S. 100)	12
3.1	Organizational [sic!] Structure und Organisational Entity in <i>MOISE</i> (Hannoun et al., 2000, S. 157)	15
3.2	Organisationsstruktur einer Fußballmannschaft in <i>MOISE⁺</i> (Hübner et al., 2002a, S. 122)	16
3.3	Soziales Schema einer Fußballmannschaft in <i>MOISE⁺</i> (Hübner et al., 2007, S. 377)	18
3.4	Architekturübersicht über <i>S-MOISE⁺</i> (Hübner et al., 2006, S. 70)	19
3.5	Architekturübersicht über <i>J-MOISE⁺</i> (Hübner et al., 2007, S. 386) . . .	20
3.6	Artefakte in <i>ORA4MAS</i> (Hübner et al., 2010c, S. 382)	21
5.1	Sprachabfolge bei der Verarbeitung von <i>MOISE-DSL</i> (eigene Abbildung)	33
5.2	Paketdiagramm der <i>MOISE-DSL</i> -Architektur (eigene Abbildung)	36
6.1	Sequenzdiagramm der MAS-Integration unter Nutzung des <i>MOISE-DSL</i> -Interpreters am Beispiel des GroupBoards (eigene Abbildung)	57
6.2	Vollständiger Testlauf für <i>MOISE-DSL</i> in NetBeans (eigenes Bildschirmfoto)	58
6.3	Testabdeckung für TimeTerm (eigenes Bildschirmfoto)	61
6.4	Integration von <i>MOISE-DSL</i> in <i>MOISE⁺</i> (eigenes Bildschirmfoto)	62

Tabellen

2.1	Vergleich von objekt- und agentenorientierter Programmierung (Shoham, 1993, S. 57)	6
3.1	Deontische Logik einer Fußballmannschaft in \mathcal{MOISE}^+ (Hübner et al., 2007, S. 378)	18
6.1	Implizite Konvertierungen in \mathcal{MOISE} -DSL	52

Listings

5.1	Vollständige Definition eines Ziels in <i>MOISE</i> -DSL	30
5.2	Erstellen eines GroupBoard-Artefakts in AgentSpeak	31
5.3	Erstellen eines SchemeBoard-Artefakts in AgentSpeak	31
6.1	Vollständige Implementierung von Normen in <i>MOISE</i> -DSL	38
6.2	Nutzung der Norm -Klasse in <i>MOISE</i> -DSL	40
6.3	Nutzung der Norm -Klasse in <i>MOISE</i> -DSL ohne Infixnotation	40
6.4	Setzen des Linktyps und Instanz-Rückgabe	41
6.5	Setzen der Richtung für Compatibility - und Link -Instanzen definiert in deren Basisklasse RoleRel	41
6.6	NormType -Konvertierung durch Pattern Matching	42
6.7	Trait zur abstrakten Definition möglicher Normtypen	42
6.8	Case Objects definieren konkrete Normtypen	43
6.9	Die Klasse AgentCount	43
6.10	Implicit zur Konvertierung von Ganzzahlen in AgentCount -Instanzen .	44
6.11	Bubble Word für Untergruppen-Gültigkeit	44
6.12	Nutzung des Bubble Words für Untergruppen-Gültigkeit	45
6.13	Gesamte Compatibility -Implementierung	46
6.14	Implementierung der Linktypen	47
6.15	Vollständiger FSConverter	53
6.16	UnitTest mit <i>arrange</i> , <i>act</i> und <i>assert</i>	59
6.17	Neue Methoden in OS.java zur Einbindung der <i>MOISE</i> -DSL in <i>MOISE</i> ⁺	62

Algorithmen

6.1	Rekursive Konvertierung von Gruppen	55
6.2	Implizite Erstellung von Zielen bei der Konvertierung von Plänen	55

Abkürzungen

A&A	Agents and Artefacts
AOP	Agentenorientierte Programmierung
BDI	Belief-Desire-Intention
CARtAgO	Common ARtifact Infrastructure for AGent Open environment
DSL	Domain-specific Language
FS	Functional Specification
GPL	General-purpose Programming Language
KQML	Knowledge Query and Manipulation Language
MAS	Multiagentensystem
<i>Moise</i>	Model of Organisation for multi-agent SystEms
NOPL	Normative Organisation Programming Language
NS	Normative Specification
OE	Organisational Entity
OMI	Organisation Management Infrastructure
OML	Organisation Modelling Language
ORA4MAS	Organisational Artifacts for Multi-Agent Systems
OS	Organisational Specification
SS	Structural Specification

1 Einführung

1.1 Zielsetzung und Motivation

Das Ziel der vorliegenden Arbeit ist es, eine spezialisierte domänenspezifische Sprache zur Formulierung von Organisationsstrukturen in Multiagentensystemen zu entwerfen, prototypisch umzusetzen und in ein bestehendes Multiagentensystem zu integrieren. Angestrebtes Resultat ist ein voll funktionales Gesamtsystem.

Motivation hierzu ist, das übergeordnete Gesamtziel, die semantische Lücke zwischen Modell und Implementierung von Multiagentensystemen immer weiter zu schließen, voran zu treiben. Dieser Ansatz wird bereits seit mehreren Jahrzehnten verfolgt. Dabei spielen Frameworks, spezialisierte Sprachen – konzeptionell durchaus ähnlich der hier zu entwerfenden – und Meta-Modelle eine große Rolle. Im Verlauf der Arbeit wird ausführlich auf diese Themen eingegangen (vgl. Abschnitt 1.2).

Behrens et al. (2010) veranschaulichen die Arbeiten der letzten Jahre im Bereich Multiagentensysteme anhand verschiedener Wettkämpfe, bei der Aufgaben mithilfe agentenorientierter Programmierung zu lösen waren. Dabei erkennen und unterstützen sie eine Verlagerung weg von der Betrachtung einzelner Agenten hin zu Kommunikations- und Koordinationsaspekten (S. 301, Hervorhebung im Original):

„While a lot of effort in this community was dedicated in the past to issues related to development of single-agent systems, gradually we witnessed a shift towards practical problems in multi-agent system, such as cooperation, coordination, negotiation, etc. [There is a] [...] gradual shift of focus [...] from scenarios more appropriate for benchmarking of behaviours of individual agents to *enforcing cooperation and coordination in the teams of agents*. Our ambition is to continue in this trend and we are actively discussing possibilities for adaptation and re-design of the future simulation scenarios to involve more features such as teamwork, coordination, cooperation, etc.“

Ebendiese Aspekte werden durch Werkzeuge wie die hier entworfene und entwickelte Sprache *MOISE-DSL* nachdrücklich unterstützt, indem sie den Fokus von der Program-

mierung individueller Agenten hin zur Modellierung ihrer Organisationen auf abstraktere Art und Weise verschiebt. Dabei soll die Komplexität von Multiagentenorganisationen, die teils von Agenten selbst gestaltet werden, vor dem modellierenden Benutzer so weit wie möglich verborgen werden, indem diesem eine einfache, kurze, prägnante und semantisch reichhaltige Formulierungsmöglichkeit in Form der beschriebenen Sprache zur Verfügung gestellt wird.

Handwerkliches Ziel für ihre Umsetzung ist eine ebenso klare und prägnante Modellierung ihrer Struktur sowie akzentuierte Formulierung ihrer internen Algorithmen in einer geeigneten Implementierungssprache. Dies umfasst u. a. den Verzicht auf nicht benötigte Kommentare, sehr kurze, übersichtliche und klar abgegrenzte Programmbestandteile wie Methoden oder Klassen und automatisierte Tests.

Es ist jedoch *nicht* Ziel dieser Arbeit, bestehende Modelle für Multiagentenorganisationen zu erweitern oder zu ändern. Der Fokus ist stattdessen technisch geprägt: Es wird untersucht, wie Implementierungen näher an ein Modell gerückt werden können, wobei das Modell als (statisches) Ziel der Implementierung betrachtet wird, selbst jedoch keiner Änderung unterworfen wird.

1.2 Gliederung

Die vorliegende Arbeit ist zuzüglich zu dieser Einführung in sechs Kapitel gegliedert.

Kapitel 2 beschreibt Grundlagen von Agenten und auf ihnen aufbauenden Systemen. Das Kapitel erläutert ferner die Agentenorientierte Programmierung und stellt mit *Jason* die konkrete Implementierung eines Multiagentensystems vor. Des Weiteren wird die Bedeutung von Umwelten für Agentensysteme beschrieben und mit Agents and Artefacts ein Ansatz zu deren Abstraktion vorgestellt.

Kapitel 3 gibt eine Einführung in Multiagentenorganisationen anhand des der *MOISE*-DSL zugrundeliegenden Modells, existierender Implementierungen und Modellierungssprachen.

In Kapitel 4 werden die Grundlagen und Einsatzbereiche von domänenspezifischen Sprachen erläutert. Die gängigste und implementierungstechnisch wesentlichste Klassifikation für selbige wird vorgestellt.

Kapitel 5 schließlich beschreibt das Vorgehensmodell bei der Entwicklung von *MOISE*-DSL sowie grundlegende Entscheidungen bzgl. des Entwurfs der Sprache. Weitere Themen dieses Kapitels sind grundlegende Vorentscheidungen zur Implementierung wie die verwendete Programmiersprache sowie die interne *MOISE*-DSL-Architektur und die In-

tegration in Multiagentensysteme.

Kapitel 6 widmet sich ausführlich der Art, wie *MOISE*-DSL intern implementiert ist. Weiteres Thema dieses Kapitels ist die Qualitätssicherung mittels automatisierter Tests im Zuge der Implementierung und allgemein die handwerkliche Qualität selbiger.

Themen von Kapitel 7 sind neben einer kurzen Zusammenfassung die rückblickende Bewertung aus fachlicher und technischer Sicht sowie die Beschreibung offener Punkte und möglicher Erweiterungen. Zuletzt wird die Frage nach der allgemeinen Bedeutung moderner Programmiersprachen und -paradigmen für Multiagentensysteme am Beispiel der *MOISE*-DSL angerissen.

Die in der vorliegenden Arbeit verwendete Terminologie ist der jeweils angegebenen Literatur entnommen. Da sie in den verwiesenen Arbeiten ausführlich eingeführt und in der wissenschaftlichen Gemeinschaft verbreitet ist, wird auf die formal korrekte Definition sowie die ausführliche Erläuterung einzelner Terme im Rahmen dieser Ausarbeitung verzichtet, sofern sie für das Verständnis der Arbeit nicht wesentlich sind.

2 Multiagentensysteme

2.1 Grundlagen

2.1.1 Agent

Obwohl Agenten bereits seit langem Betrachtungsgegenstand von Wissenschaft und Forschung sind, ist der Begriff *Agent* bis heute uneinheitlich definiert, wie Wooldridge (2009, S. 21) anmerkt. So sei bereits strittig, ob die Fähigkeit von Agenten, aus deren Erfahrungen zu lernen, charakteristisch für diese ist.

In der vorliegenden Arbeit sollen daher als kleinster gemeinsamer – hier jedoch ausreichender – Nenner die folgenden Eigenschaften als Charakteristika für Agenten betrachtet werden, die Wooldridge und Jennings (1995) als schwache Auffassung (*weak notion*) beschreiben. Sie definieren einen Agenten als Hardware- oder Software-System mit den folgenden Eigenschaften:

Autonomie: Agenten agieren ohne direkte Intervention von Menschen oder anderen und kontrollieren ihre Aktionen und ihren internen Status selbstständig.

Soziale Fähigkeiten: Agenten interagieren mit Dritten (i. d. R. anderen Agenten, aber potentiell auch Menschen) mittels eigener Sprachen.

Reaktionsfähigkeit: Agenten nehmen ihre Umgebung wahr und können auf Veränderungen derselben reagieren. Als Umgebung der Agenten kommen dabei die reale Welt, ein Benutzer des Agenten-beinhaltenden Software-Systems, eine Gruppe anderer Agenten, das Internet oder Kombinationen daraus in Frage.

Pro-Aktivität: Über die Fähigkeit auf Veränderungen ihrer Umgebung zu reagieren hinaus sind Agenten in der Lage, die Initiative zu ergreifen, um eigene Ziele zu erreichen.

Als Beispiel-Implementierungen für Agenten führen Wooldridge und Jennings UNIX-artige Software-Prozesse auf.

Im Folgenden wird ein Agent als Computerprogramm (Software) verstanden, ohne eine Aussage darüber zu machen, ob für dessen Ausführung ein Rechnerverbund, ein einzelner Rechner, ein Betriebssystem-Prozess, Thread oder anderes zuständig ist.

Wesentlich für Agenten ist deren Eigenschaft, nicht isoliert für sich, sondern in einem Gesamtkontext bestehend aus anderen Agenten, also in einem Multiagentensystem (MAS), zu agieren. Für die Inter-Agentenkommunikation existieren verschiedene theoretische Modelle, wie z.B. *Sprechakte*, und Protokolle wie die *Knowledge Query and Manipulation Language (KQML)* (Huhns und Stephens, 2000). Aufbauend auf Kommunikationsprotokollen, mit deren Hilfe einzelne Nachrichten übermittelt werden können, sind verschiedene Koordinationsmechanismen, darunter Tafel-Ansätze (*blackboard*) und Marktmechanismen, zur Abstimmung zwischen den einzelnen Agenten möglich.

Agenten können in unterschiedliche Klassen eingeteilt werden. Agenten im Sinne dieser Arbeit werden i. d. R. als *kognitive* Agenten bezeichnet, da sie durch Kenntnis eines (individuellen) Modells ihrer Umwelt in die Lage versetzt werden, zielgerichtet zu handeln (vgl. Abschnitt 2.1.3). Da sie in der Lage sind, dieses Umweltmodell zu *lernen*, werden sie ferner als *intelligent* bezeichnet. Aufgrund der Tatsache, dass sie Entscheidungen in Abhängigkeit davon treffen, welche Handlungsmöglichkeit sie ihrer individuellen Zielerreichung näher bringt, also den höchsten individuellen Nutzen verspricht, heißen sie des Weiteren *rational*.

2.1.2 Agentenorientierte Programmierung

Shoham (1993) schlägt Agentenorientierte Programmierung (AOP) als Spezialisierung der Objektorientierten Programmierung (OOP) vor und stellt mit der Sprache *AGENT-0* samt Interpreter eine Referenzimplementierung zur Diskussion. AOP kann als Programmierparadigma betrachtet werden, mit dessen Hilfe Systeme, deren Bestandteile sich gut zur Modellierung in Form von Agenten eignen, effektiv und effizient beschrieben werden können.

Dem generischen Status eines Objektes als grundlegender Einheit der Objektorientierung stellt Shoham den sog. *mentalen* Status eines Agenten als AOP-Äquivalent gegenüber. Er beinhaltet Entitäten wie Weltwissen, Entscheidungen, Möglichkeiten und Verpflichtungen des Agenten. Während nur generische Nachrichten zwischen Objekten ausgetauscht werden, stehen in der AOP ausdrucksstärkere Nachrichtentypen wie *informieren*, *anfordern* oder *anbieten* zur Verfügung.

Tabelle 2.1 vergleicht beide Programmierparadigmen.

	OOP	AOP
Basic unit	object	agent
Parameters defining state of basic unit	unconstrained	beliefs, commitments, capabilities, choices . . .
Process of computation	message passing and response methods	message passing and response methods
Types of message	unconstrained	inform, request, offer, promise, decline, . . .
Constraints on methods	none	honesty, consistency, . . .

Tabelle 2.1: Vergleich von objekt- und agentenorientierter Programmierung (Shoham, 1993, S. 57)

Zwar kann die AOP technisch gesehen genutzt werden, um beliebige Systeme zu modellieren – Shoham nennt als Beispiel die Formulierung eines Lichtschalters als (sehr kooperativer) Agent, der auf Verlangen hin Strom leitet, wenn er per Nachricht in Form des Umlegen des Schalters über diesen Wunsch informiert wird. Allerdings verfügt die Allgemeinheit über ein ausreichendes Verständnis des Systems *Lichtschalter*, um dieses in einem einfacheren und prägnanteren Modell zu beschreiben, das die grundlegenden mechanischen Eigenschaften des Schalters ausdrückt.

Anders sieht die Sache jedoch bei der Modellierung von Systemen aus, die aus verschiedenen einzelnen Akteuren wie Robotern, Menschen, Marktteilnehmern und anderen Organisationen – oder auch dezentralen technischen Komponenten – bestehen, die miteinander interagieren und über eigene Ziele und Wissensstände verfügen. Bei der Abbildung solcher Systeme kann das AOP-Paradigma die semantische Lücke zwischen System und Modell gegenüber allgemeinen Ansätzen wie der Objektorientierung deutlich verringern, da ihm die Typen grundlegender Modellelemente als *First-Class Citizens* bekannt sind.

Es ist Ziel der vorliegenden Arbeit diese Lücke weiter zu schließen, indem Anwendern eine Möglichkeit zur modellnäheren Formulierung des AOP-Teilaspektes der Multiagentenorganisationen aufgezeigt wird.

2.1.3 Belief-Desire-Intention

Belief-Desire-Intention (BDI) ist eine auf Rao und Georgeff (1991) zurückgehende Architektur für Agenten, die Weltwissen (*beliefs*), Ziele (*desires* bzw. *goals*) und Absichten (*intentions*) eines Agenten als dessen zentrale Elemente betrachtet.

Das Weltwissen des Agenten wird dabei durch dessen Wahrnehmung – also seiner

Wirklichkeit, nicht Realität im Sinne der neueren Wissenschaftstheorie – sowie Schlussfolgerungen gebildet und fortwährend aktualisiert. Darüber, wie diese Wissensbasis dabei konkret ausgestaltet ist, macht BDI keine Aussage.

Ziele repräsentieren die aktuelle Motivation eines Agenten und damit, was dieser erreichen *möchte*. *Goals* gehen dabei insofern über *desires* hinaus, dass für erstere Konsistenz gefordert wird, wohingegen einzelne *desires* widersprüchlich sein können (Rao und Georgeff, 1991, S. 476). Ein Agent wird folglich aus der Menge seiner *desires* immer nur solche zu aktuellen *goals* machen, die nicht in Konflikt zueinander stehen. Dies wird für den Term *Ziel* im Folgenden angenommen, sofern nicht explizit auf das Gegenteil hingewiesen wird.

Eine Intention bezeichnet einen *Plan*, den der Agent aktuell verfolgt. Diesen hat er zuvor aus einer internen Plandatenbank ausgewählt, in der ihm Pläne zur Verfügung stehen, die ihm seinen Zielen näher bringen können. Pläne können hierarchisch ineinander verschachtelt werden, also selbst andere Pläne beinhalten. Darüber hinaus bestehen Pläne aus Aktionsfolgen.

2.2 Jason

Jason ist eine Entwicklungsplattform für MAS, der die BDI-Architektur zugrunde liegt (Bordini und Hübner, 2006). *Jason* ist Open Source und dank Implementierung in Java plattformunabhängig nutzbar. Funktionen von *Jason* umfassen:

- Sprechakt-basierte Kommunikation zwischen Agenten
- Annotations für Plan-Labels, die für beliebige Auswahlfunktionen herangezogen werden können.
- Eine über ein ganzes Netzwerk verteilte Laufzeitumgebung für das MAS
- Erweiterungsmöglichkeiten der Handlungsoptionen für Agenten über sog. *interne Aktionen*
- Die Möglichkeit, die Umwelt (*environment*) des MAS mit Java in einer General-purpose Programming Language (GPL) zu definieren und damit beliebige Freiheitsgrade für selbige

Die Orientierung am BDI-Konzept wird mit Blick auf die für Agenten in *Jason* wesentlichen drei Aspekte deutlich:

Beliefs als Annahmen des Agenten über den Weltzustand finden sich gegenüber der in Abschnitt 2.1.3 gegebenen Beschreibung unverändert in *Jason* wieder. Sie werden in Form einfacher Literale oder Formulae definiert.

Goals sind konkrete Zielsetzungen des Agenten, also Dinge, die er erreichen oder – im Falle sog. *test goals* – *erfahren* möchte.

Plans sind dem Agenten mögliche Handlungsfolgen und beinhalten ferner eine Angabe des sie auslösenden Ereignisses sowie den Kontext, also die Bedingungen, unter denen das angegebene Ereignis die definierte Handlungsabfolge auslöst. Auslösende Ereignisse können dabei Änderungen der Goal- oder Beliefverhältnisse des Agenten sein.

Agentenverhalten wird in *Jason* mittels einer erweiterten Version von *AgentSpeak* formuliert, einer logischen, ebenfalls auf der BDI-Architektur basierenden, Programmiersprache, die Rao (1996) bereits vor 15 Jahren vorstellte. AgentSpeak wurde später von Bordini et al. (2002) für die Nutzung in *Jason* u. a. um die Möglichkeit erweitert, interne Aktionen (*internal actions*) zu formulieren, die via Java oder – mittels *Java Native Interface (JNI)* – beliebiger weiterer Programmiersprachen mit Systemen außerhalb der *Jason*-Plattform interagieren, beispielsweise mit End-Usern oder Legacy-Systemen.

Die MAS-Umwelt wird, wie bereits erwähnt, in Java implementiert und folgt i. d. R. dem Architekturmuster *Model View Controller* (Reenskaug, 1979), bei dem zwischen der medialen Visualisierung der Umwelt, ihrem Datenmodell und der darauf zugreifenden Steuerungslogik unterschieden wird. Für Raster-basierte Umwelten liefert *Jason* mit dem **GridWorldModel** bereits eine Basisklasse für eigene Implementierungen, die wichtige Funktionen wie die Verwaltung von Agenten und anderen Objekten bereitstellt. Abbildung 2.1 zeigt ein Beispiel für eine – ebenfalls durch *Jason* unterstützte – Visualisierung einer solchen Umwelt.

Agenten können mit der Umwelt über Aktionen und sog. *Percepts* reagieren, wie Abbildung 2.2 illustriert. Percepts stehen dabei für die Eigenschaften der Umwelt, die der entsprechende Agent aktuell wahrnehmen und auf die er reagieren kann. Diese können sich je nach Agent individuell unterscheiden.

Jason unterstützt den MAS-Entwickler bei der Arbeit mit Debugging-Tools wie dem in Abbildung 2.3 dargestellten *Mind Inspector*, der Auskunft über den mentalen Zustand eines Agenten gibt.

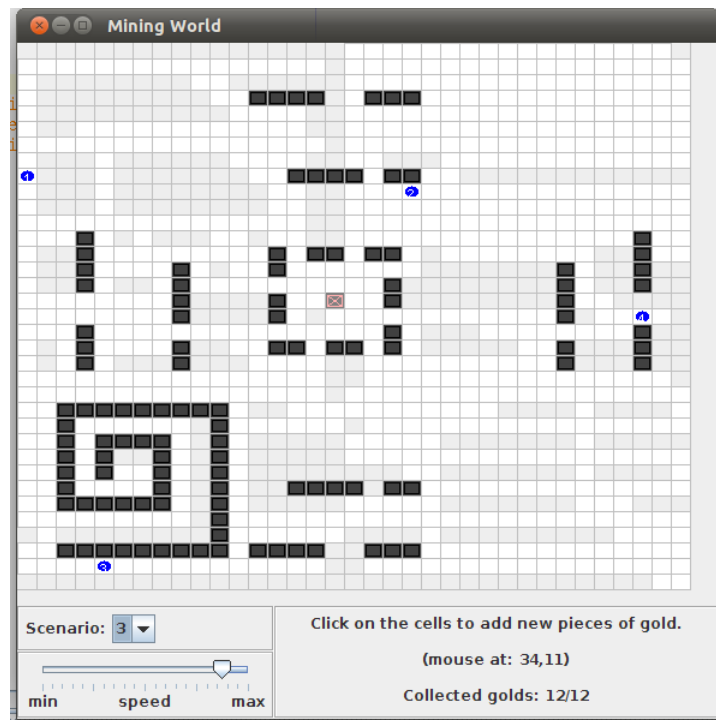


Abbildung 2.1: Beispiel für ein Grid View in *Jason* (eigenes Bildschirmfoto)

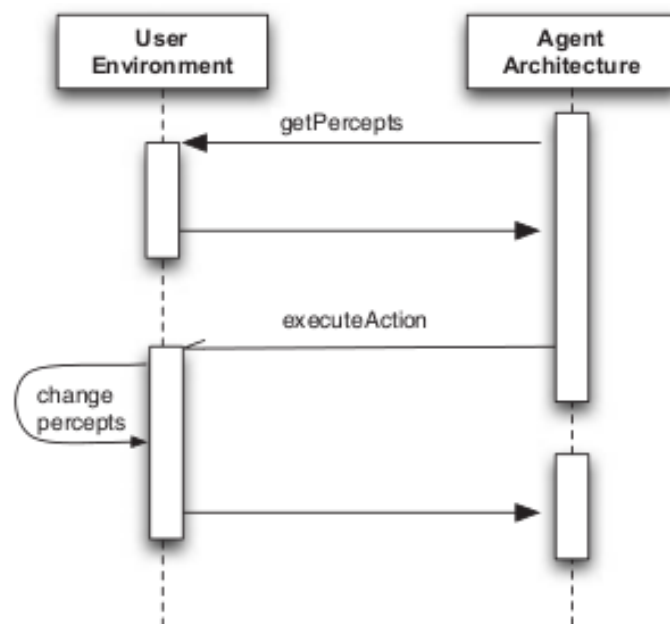


Abbildung 2.2: Interaktion zwischen Agenten-Architektur und Umwelt in *Jason* (Bordini et al., 2007, S. 103)

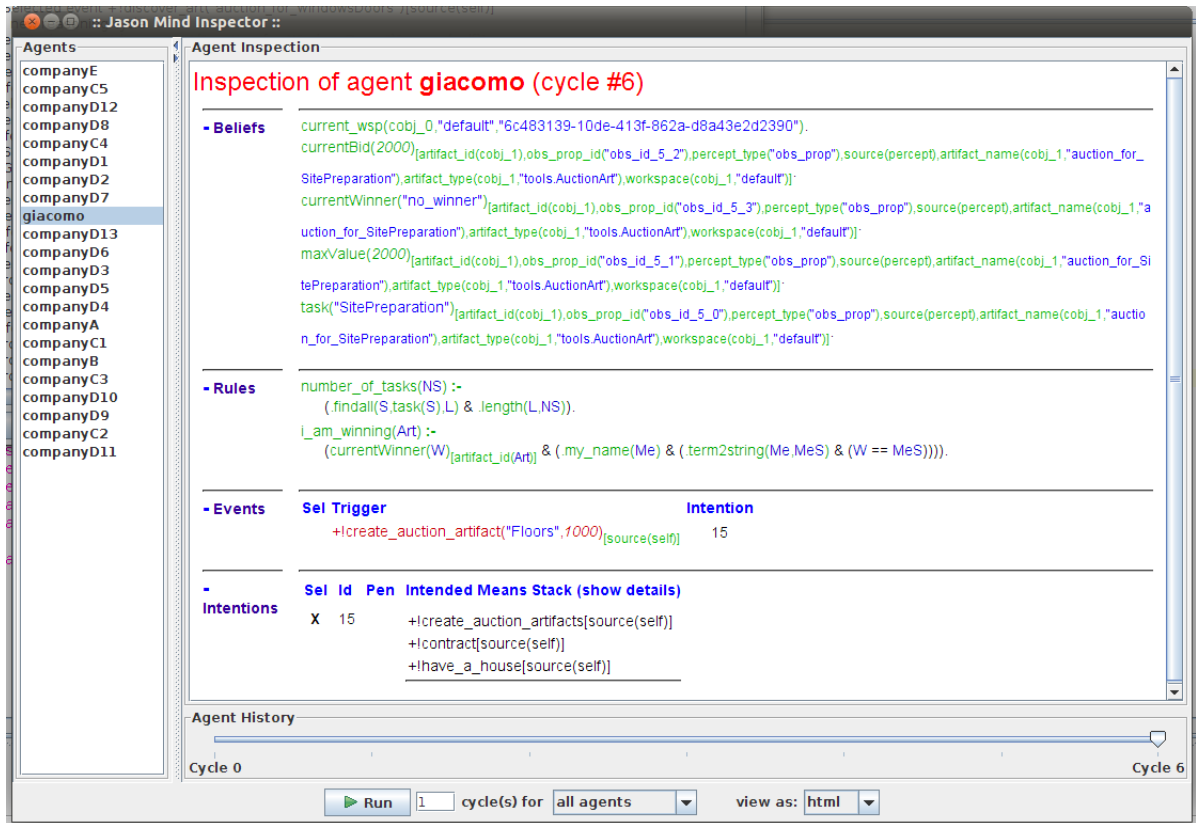


Abbildung 2.3: Mind Inspector in *Jason* (eigenes Bildschirmfoto)

2.3 Agents and Artefacts

Während *Jason* dem MAS-Entwickler mit AgentSpeak eine ausdrucksstarke Sprache zur Formulierung der einzelnen BDI-Aspekte an die Hand gibt, unterstützt es ihn bei der Implementierung der Umwelt nur wenig. Zwar stellt es ihm die in Abschnitt 2.2 genannten Klassen zur Verfügung, diese dienen jedoch in erster Linie Verwaltungs- und Visualisierungszwecken sowie weiteren Implementierungsvereinfachungen. Bei der *Modellierung* der Umwelt und ihrer Semantik unterstützen sie kaum.

Aus diesem Grund schlagen Ricci et al. (2006) mit Agents and Artefacts (A&A) ein Programmiermodell vor, das *Artefakte* als First-Class-Abstraktion in MAS-Umwelten einführt. Als Artefakte werden dabei sämtliche Objekte betrachtet, die Agenten in irgendeiner Art und Weise für sie gewinnbringend einsetzen können. Sie dienen damit als Grundbaustein für die Realisierung von MAS-Umwelten und können sowohl Koordinationsartefakte als auch konkrete Ressourcen repräsentieren.

Analog zu Klassen und Objekten werden Artefakte abstrakt definiert und für die Nutzung instanziiert. Abbildung 2.4 zeigt ein Beispiel. Dabei definiert das *Usage Inter-*

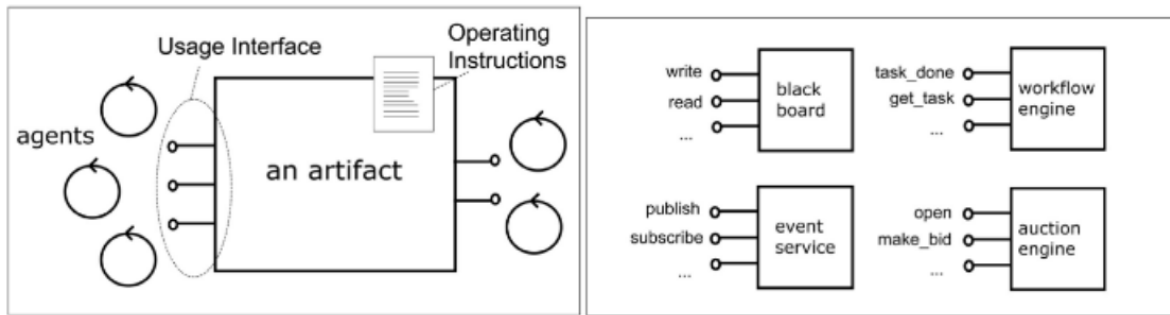


Abbildung 2.4: Abstrakte Repräsentation und konkrete Instanzen eines Artefakts (Ricci et al., 2006, S. 209)

face die möglichen Operationen, die Agenten auf Instanzen des Artefakts durchführen können, die *Operating Instructions* dagegen beschreiben, wie das Artefakt selbst seine Funktionalität erhält.

Während Ricci et al. 2006 das A&A-Model noch am Beispiel der AOP-Sprache *3APL* (Dastani et al., 2003) und dem Netzwerk-basierten Tuple Space *TuCSon* (Omicini und Zambonelli, 1999) als Koordinationsartefakt demonstrieren, stellen sie mit der Common ARtifact Infrastructure for AGent Open environment (**CARtAgO**) ein Jahr später ein generisches Framework für die Entwicklung Artefakt-basierter MAS-Umwelten inkl. Referenz-Implementierung in Java, das auch zur Programmierung der Artefakte dient, vor (Ricci et al., 2007).

2008 findet eine Integration von *Jason* und **CARtAgO** statt, sodass ein MAS zur Verfügung steht, das die Umwelt – entgegen der eigentlichen Wortbedeutung – als integralen Bestandteil des Gesamtsystems betrachtet (Ricci et al., 2008). Ricci et al. erweitern den Begriff *environment* daher zu *working environment*, weshalb im Folgenden von der Arbeitsumgebung der Agenten die Rede sein soll.

CARtAgO besteht aus drei Hauptbestandteilen:

1. Eine Programmierschnittstelle zum Erstellen und Interagieren mit der Artefakt-basierten Arbeitsumgebung erweitert die bestehende Menge der Aktionen, die Agenten ausüben können. Dazu zählen neben dem Erstellen und Zerstören von Artefakten auch das Aufrufen derer Operationen sowie das Beobachten ihrer Status und die Reaktion auf Ereignisse.
2. Eine Programmierschnittstelle zur Definition von Artefakttypen erlaubt es Entwicklern durch Ableiten von einer Basisklasse in Java eigene Artefakt-Klassen zu erstellen. Es stehen einfache Mechanismen zum Auslösen von Ereignissen zur Verfü-

gung. Ferner spielen Annotationen zum Auszeichnen der Artefakt-Operationen, deren Schritte und Ausführungsbedingungen (*guards*) sowie der Artefakt-Metadaten (mögliche Status, Ereignisse etc.) eine große Rolle.

3. Eine Laufzeitumgebung verwaltet den Lebenszyklus der Arbeitsumgebung, erzeugt und zerstört Artefakte und Agenten und routet beobachtbare Ereignisse, die von Artefakt-Instanzen erzeugt wurden. Abbildung 2.5 zeigt diese Laufzeitumgebung in der Middleware-Schicht gleichauf und integriert mit *Jason* oder alternativen Plattformen.

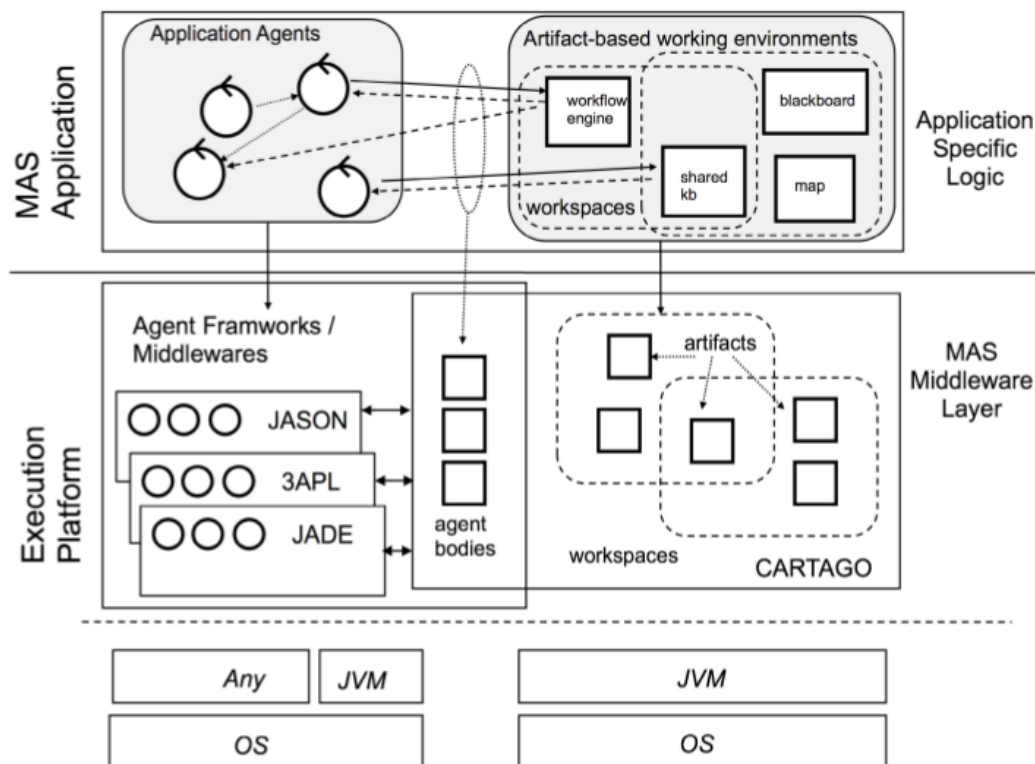


Abbildung 2.5: Schematische Übersicht eines MAS in Kombination mit CArtaGo (Ricci et al., 2008, S. 100)

Auch das A&A-Programmiermodell im Allgemeinen und sein Vertreter CArtaGo im Speziellen helfen, die semantische Lücke zwischen Modell und Implementierung von MAS weiter zu schließen, indem sie eine semantisch reichhaltigere Variante zur Definition von Arbeitsumgebungen zur Verfügung stellen. Zwar werden diese im betrachteten Fall gegenüber *Jason* nach wie vor in der GPL Java implementiert. Dennoch steht nun erstens mit der **Artefact**-Basisklasse erstmals eine semantisch belegte First-Class-Abstraktion

für Elemente der Arbeitsumgebung zur Verfügung. Zweitens ermöglichen die angesprochenen Annotationen mehr deklarative und weniger imperative Programmierung und führen damit ebenfalls zu mehr Bedeutung in den Modellen.

3 Multiagentenorganisationen

3.1 *Moise*

Das *Model of Organisation for multi-agent SystEms* (*MOISE*) ist ein Modell zur Definition von Multiagentenorganisationen mit dem Ziel, agenten- und organisationszentrierte Ansätze zur Beschreibung der Agentenkommunikation und -koordination in MAS zu vereinen (Hannoun et al., 2000).

Agenten- und organisationszentrierte Ansätze unterscheidet dabei, dass erstere davon ausgehen, dass Agenten die Verbindungen zwischen ihnen einvernehmlich ermitteln. Letztere dagegen betrachten diese als explizit vom MAS-Designer zur Entwurfszeit oder von Agenten zur Laufzeit definiert und den Agenten auferlegt.

In *MOISE* werden drei grundlegende Konzepte unterschieden:

Rollen beschreiben Verhaltens- oder Dienstklassen, die Agenten des MAS anbieten können. Dabei bestehen Rollen aus einer Menge von Missionen, die den Raum der möglichen Aktionen für den die Rolle innehabenden Agenten einschränken. Die Missionen einer Rolle werden in ihrer Stärke dahingehend unterschieden, dass ein Agent diese entweder zwingend ausführen muss oder aber die Wahl hat, ob er dies tut.

Verbindungen innerhalb einer Organisation regeln den sozialen Austausch zwischen den Inhabern der zuvor definierten Rollen. Es werden unterschiedliche Typen der Verbindungen (darunter Kommunikationswege und Weisungsbefugnisse) und sich daraus ergebende Einschränkungen sowie ihre Richtung unterschieden.

Gruppen fassen Rollen inkl. deren Missionen sowie Verbindungen zusammen und definieren ferner, ob Verbindungen nur innerhalb einer oder auch zwischen verschiedenen Gruppen gültig sind.

Zu beachten ist, dass in *MOISE* die genannten Konzepte abstrakt, also ohne Kenntnis konkreter Agenten, definiert werden. Eine solche Definition heißt *Organizational Struc-*

ture, wohingegen eine konkrete Instanz dieser Struktur *Organisational Entity (OE)* genannt wird. Abbildung 3.1 veranschaulicht diesen Unterschied anhand einer stark vereinfachten Hochschul-Organisationsstruktur als Beispiel.

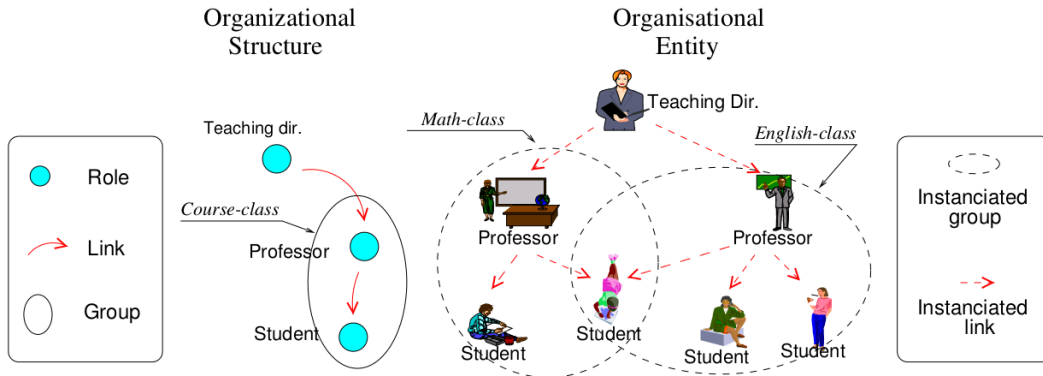


Abbildung 3.1: Organizational [sic!] Structure und Organisational Entity in *MOISE*
(Hannoun et al., 2000, S. 157)

3.2 $\mathcal{M}\text{oise}^+$

3.2.1 Struktur

Während \mathcal{MOISE} implizit zwar sowohl Organisationsstrukturen (Rollen, Verbindungen und Gruppen) als auch funktionale Organisationsaspekte (Pläne und Aufgaben in Form von Missionen) und deontische Logik (normative Aspekte wie Verpflichtung und Erlaubnis in Form von Missionsstärke) modelliert, bringt \mathcal{MOISE}^+ dieses Konzept in eine klarere Struktur (Hübner et al., 2002b,a):

Die Organisational Specification (OS) – in \mathcal{MOISE} noch als Organizational Structure bezeichnet – umfasst mit Structural Specification (SS), Functional Specification (FS) und Normative Specification (NS) alle zuvor genannten Aspekte und erlaubt es, diese in einer einzelnen, schlüssigen Spezifikation zu beschreiben. SS und FS werden in \mathcal{MOISE}^+ weitgehend unabhängig voneinander modelliert und anschließend über die NS miteinander verbunden.

3.2.2 Organisational Specification

Structural Specification

Zur Beschreibung der Organisationsstruktur erweitert \mathcal{MOISE}^+ \mathcal{MOISE} um Vererbung von Rollen, rekursive Gruppendefinitionen, Kompatibilität zwischen einzelnen Rollen sowie Kardinalitäten. So darf in der Organisationsstruktur in Abbildung 3.2 beispielsweise nur genau ein Agent die Rolle des Torhüters übernehmen; anderenfalls ist die Gruppe der Abwehr nicht mehr wohlgeformt, also gültig im Sinne ihrer Spezifikation.

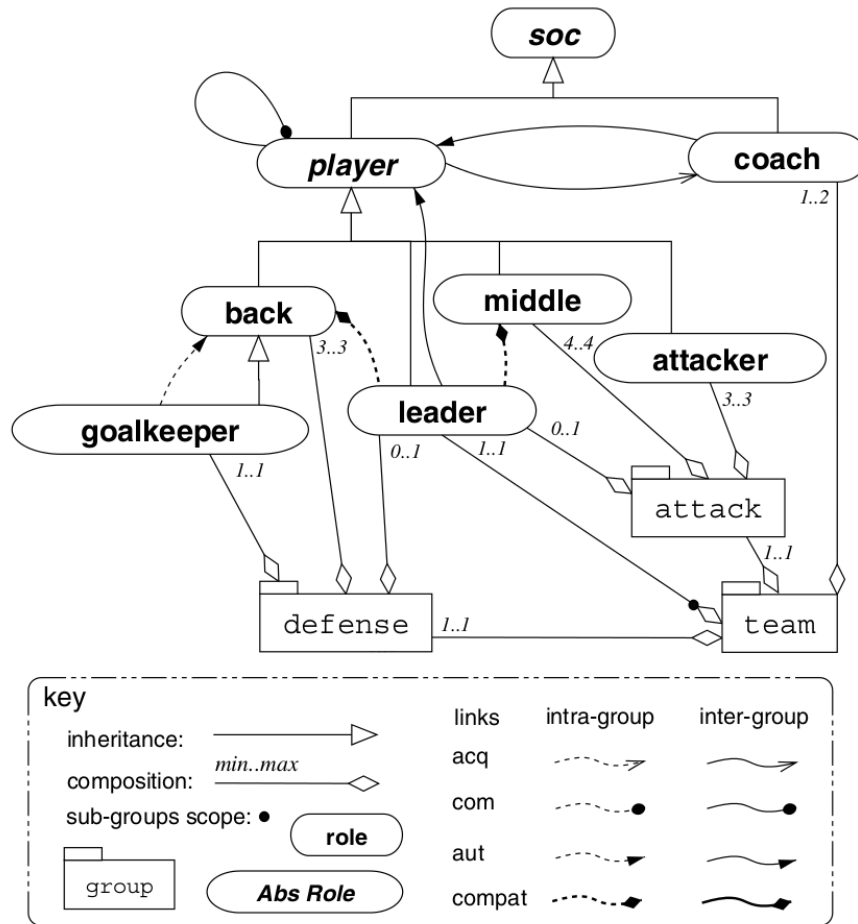


Abbildung 3.2: Organisationsstruktur einer Fußballmannschaft in \mathcal{MOISE}^+ (Hübner et al., 2002a, S. 122)

Die SS wird auf drei Ebenen definiert:

1. Die *individuelle* Ebene beschreibt das Verhalten, das einem Agenten auferlegt wird, wenn er die Rolle ausübt. Dieses ist in den Eigenschaften der Rolle festgelegt – entweder für die konkrete Rolle direkt oder indirekt über eine Rolle, von der diese

erbt. Hierunter fällt z. B., dass ein Agent, der im genannten Beispiel Mannschaftskapitän ist, gleichzeitig ein Abwehrspieler sein kann, da diese Rollen miteinander kompatibel sind.

2. Die *kollektive* Ebene beschreibt welche und wie viele Rollen jeweils in eine Gruppe eingeteilt werden. In der Abwehr-Gruppe der Fußballmannschaft müssen mindestens ein Torhüter und drei Abwehrspieler vorhanden sein, damit diese wohlgeformt ist. Die Präsenz eines Mannschaftskapitäns in dieser Gruppe ist optional; er muss jedoch im gesamten Team vorhanden sein, da für diese Gruppe – oder eine ihrer Untergruppen – genau ein Kapitän gefordert wird.
3. Auf der *sozialen* Ebene werden Rollen miteinander verbunden. Eine Verbindung kann uni- oder bidirektional zwischen einer Quell- und einer Zielrolle angeordnet sein und Autorität (wie die, die im Beispiel der Trainer über alle Spieler hat), Kommunikation oder schlicht Kenntnis voneinander bedeuten. Verbindungen können nur innerhalb oder auch zwischen Gruppen gelten und gleichermaßen optional auch für Untergruppen.

Functional Specification

Funktionale Aspekte einer Organisation beschreibt \mathcal{MOISE}^+ mithilfe eines oder mehrerer sozialer *Schemata*, die angeben wie ein MAS seine globalen organisatorischen Ziele erreicht. Ein Schema verweist auf ein Wurzel-Ziel, das i. d. R. seinerseits in mehrere weitere Teilziele zerlegt wird. Diese Teilziele können rekursiv aufgebaut und mithilfe von Sequenz-, Auswahl- und Parallelitätsoperatoren miteinander verknüpft sein.

Eine Menge nicht widersprüchlicher, also von ein und demselben Agenten gleichzeitig verfolgbarer, Ziele heißt *Mission*. Hat ein Agent eine Mission, ist er dafür verantwortlich, all ihre Ziele zu erfüllen. Der Agent auf Mission m_3 in Abbildung 3.3 verfolgt entsprechend die Ziele, den gegnerischen Strafraum zu erreichen, auf das gegnerische Tor zu schießen und (das Wurzel-Ziel) ein Tor zu erzielen.

Im sozialen Schema ist weiterhin für jede Mission festgelegt, wie viele Agenten diese jeweils ausführen sollen (in Abbildung 3.3 nicht visualisiert).

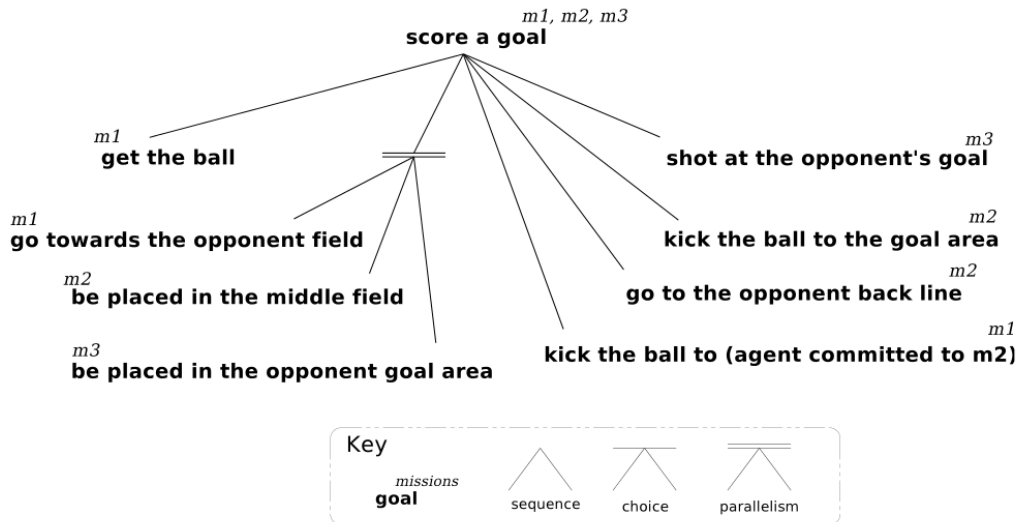


Abbildung 3.3: Soziales Schema einer Fußballmannschaft in \mathcal{MOISE}^+ (Hübner et al., 2007, S. 377)

Normative Specification

Die deontische Logik einer Organisation in \mathcal{MOISE}^+ regelt den Grad an Autonomie jedes einzelnen Agenten: Sie legt fest, welchen Rolleninhabern die Teilnahme an Missionen *erlaubt* (*permission*) oder *geboten* (*obligation*) ist.

Rolle	Relation	Mission
back	permission	m_1
middle	obligation	m_2
attacker	obligation	m_3

Tabelle 3.1: Deontische Logik einer Fußballmannschaft in \mathcal{MOISE}^+ (Hübner et al., 2007, S. 378)

So können die insgesamt drei Rollen aus dem Fußballbeispiel in Tabelle 3.1 das Schema starten, da ihnen die Teilnahme an den Missionen zu dessen Wurzel-Ziel erlaubt – oder sogar geboten – ist.

Das komplette Fußball-Beispiel macht die lose Kopplung der einzelnen Organisationsaspekte deutlich: Rollen und Ziele werden über die Indirektionsstufe der Missionen einander zugeordnet, die außerdem eine widerspruchsfreie Menge an gleichzeitig von einem Agenten verfolgten Zielen erzwingen. Rollen und Missionen dagegen werden über NS-Normen, also Erlaubnisse und Verpflichtungen, miteinander verknüpft.

3.2.3 Implementierungen

\mathcal{S} -Moise⁺

\mathcal{S} -MOISE⁺ ist die quelloffene Implementierung einer Middleware auf Basis des MOISE-Modells (Hübner et al., 2006). Sie dient als Schnittstelle zwischen den Agenten auf der einen und dem Gesamtsystem auf der anderen Seite. Agenten können sowohl die abstrakte Struktur (OS) als auch die konkrete Ausprägung (OE) der Organisationen abfragen und verändern. Die beiden Hauptkomponenten, der *OrgManager* sowie die *OrgBox*-Programmierschnittstelle, werden im Folgenden beschrieben:

OrgManager: Der OrgManager ist ein Agent, der die aktuelle OE-Ausprägung verwaltet und konsistent hält. Dies beinhaltet die Prüfung, ob Änderungen, die über die OrgBoxen anderer Agenten an ihn herangetragen werden, den in der OS definierten Bedingungen, wie beispielsweise Rollenkompatibilität, entsprechen. Falls nicht, lässt er die angefragte Änderung nicht zu.

OrgBox: Je eine OrgBox dient jedem Agenten als Schnittstelle zur Organisations- und Kommunikationsschicht. Damit abstrahiert die OrgBox gegenüber dem sie nutzenden Agenten von der tatsächlichen Implementierung der Kommunikationsprotokolle. Aus Organisationssicht stellt die OrgBox eines Agenten sicher, dass er nur von der Teilmenge der Organisation erfährt, die seine Rollen zulassen. Das bedeutet, dass potentiell jeder Agent eine andere Sicht auf die Organisation hat, da die Rollen der Agenten in der SS nur von bestimmten anderen Rollen Kenntnis haben (vgl. Abschnitt 3.2.2) oder mit ihnen kommunizieren dürfen.

Abbildung 3.4 zeigt das Zusammenspiel und den Gesamtkontext dieser Komponenten.

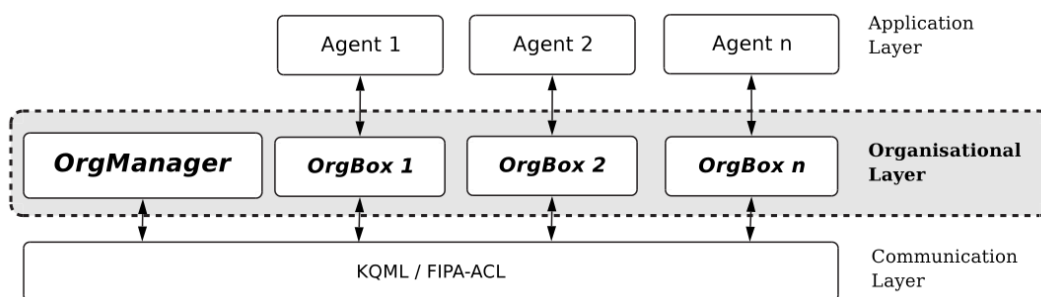


Abbildung 3.4: Architekturübersicht über \mathcal{S} -MOISE⁺ (Hübner et al., 2006, S. 70)

\mathcal{J} -Moise⁺ und ORA4MAS

\mathcal{J} -MOISE⁺ erweitert die AgentSpeak-Implementierung in *Jason* aufbauend auf \mathcal{S} -MOISE⁺ um Sprachkonstrukte zum Ändern der Organisation sowie um Ereignisse zur Reaktion auf diese Änderungen (Hübner et al., 2007). Abbildung 3.5 veranschaulicht die-

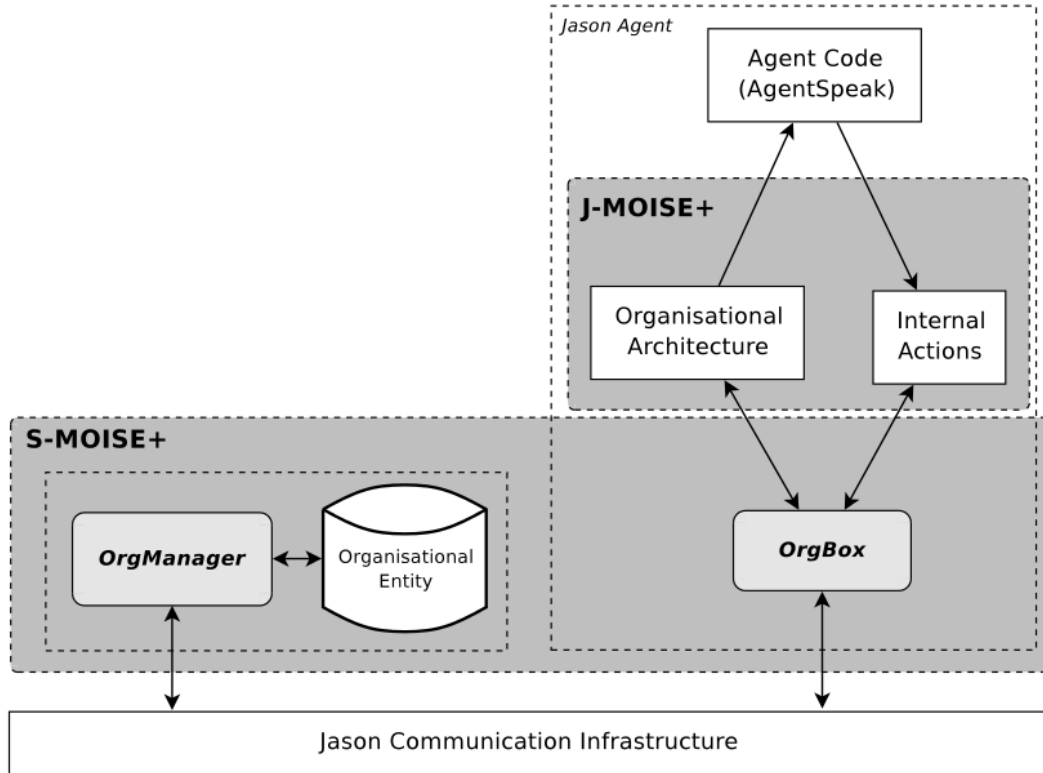


Abbildung 3.5: Architekturübersicht über \mathcal{J} -MOISE⁺ (Hübner et al., 2007, S. 386)

sen Zusammenhang. Die in \mathcal{J} -MOISE⁺ möglichen Operationen sind aus *Jason*-Sicht als interne Aktionen implementiert (vgl. Kapitel 2.2), die dann *organisatorische* Aktionen heißen. Ihre Gegenstücke, die organisatorischen Ereignisse, beinhalten das Erstellen oder Entfernen von Gruppen und Schemata sowie ihre Statusänderungen und die von Zielen, Verantwortlichkeiten, Rollen-Übernahmen etc. Auf diese Ereignisse können Agenten mit in AgentSpeak definierten Plänen reagieren.

Auf Basis von **CARTAgO** (vgl. Kapitel 2.3) führen Hübner et al. (2010c) *Organisational Artifacts for Multi-Agent Systems (ORA4MAS)* ein, das die \mathcal{J} -MOISE⁺-Organisations-ebene den Agenten über die Nutzung von Artefakten zugreifbar macht. So muss ein Agent beispielsweise ein Artefakt nutzen, um eine Rolle anzunehmen. Es ist im Weiteren Aufgabe dieses Artefakts, für die Einhaltung der für die Organisation definierten Regeln zu sorgen. Der sich ergebende Vorteil besteht darin, dass organisatorische Aspekte des

MAS nun nicht mehr in eine Ebene außer-/unterhalb des eigentlichen Modells ausgelagert werden, sondern den Agenten in Form von Artefakten gemäß A&A-Paradigma als Werkzeuge und Ressourcen (First-Class-Abstraktionen) zur Verfügung stehen. Auf das Verpacken der Organisationsaspekte in OrgBox- und OrgManager-Agenten wird nun zugunsten von Organisationsartefakten verzichtet (Kitio et al., 2008). Da diese konzeptionell etwas anderes als Agenten sind und dieser Sachverhalt sich nun auch im Modell widerspiegelt, schließt sich die semantische Lücke zwischen Modell und Realität weiter.

Abbildung 3.6 stellt die organisatorischen Artefakte zur Nutzung durch die Agenten dar. Mögliche Operationen sind durch Kreise gekennzeichnet, Eigenschaften als Recht-

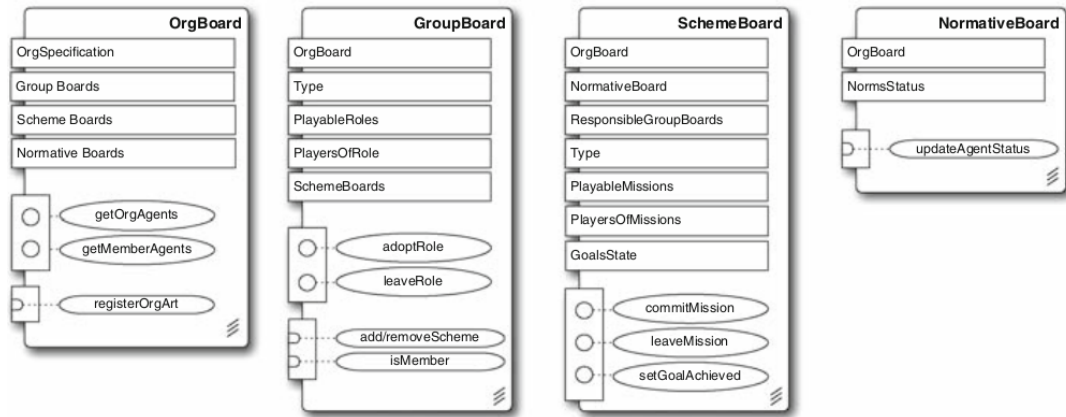


Abbildung 3.6: Artefakte in ORA4MAS (Hübner et al., 2010c, S. 382)

ecke. Halbkreise markieren I/O-Ports zum Verbinden mehrerer Artefakte miteinander. Das grundlegende Modell aus \mathcal{MOISE}^+ , bestehend aus OS, SS, FS und NS (vgl. Abschnitt 3.2.2) wird hier auf die Artefakt-Typen *OrgBoard*, *GroupBoard*, *SchemeBoard* und *NormBoard* abgebildet, über deren Operationen lesender und ändernder Zugriff auf die genannten Modellelemente ermöglicht wird.

Als naheliegenden nächsten Schritt hin zu einem einheitlichen AOP-Programmiersmodell schlagen Piunti et al. (2010) eine Kombination aus *Jason*, *CARTaGO* und \mathcal{MOISE}^+ ¹ vor. Ihre einzelnen Komponenten stellen Mechanismen bereit für die Interaktion von Agenten mit Organisations- und Umwelt- bzw. Umgebungsentitäten, für die Kontrolle der Agentenaktivitäten durch diese beiden sowie für den Einfluss von Umwelt- bzw. Umgebungsänderungen auf Organisationen und Agentenverhalten. Eine quelloffene Implementierung des Ansatzes steht als *JaCaMo* unter <http://jacamo.sourceforge.net/>

¹„ \mathcal{MOISE}^+ “ bezeichnet im Sinne dieser Kombination nicht nur das gleichnamige Modell sondern steht hier auch stellvertretend für dessen Implementierung in Software. Diese Wortbedeutung wird im Folgenden übernommen.

zum Download zur Verfügung.

Ein anschauliches Beispiel für die integrierte Verwendung der drei Komponenten findet sich bei Hübner et al. (2009).

Organisation Modelling Language

Die *Organisation Modelling Language (OML)* dient der Beschreibung von Organisationsstrukturen in \mathcal{MOISE}^+ . OML ist als Auszeichnungssprache auf Basis von XML realisiert, die zugehörige Schemadefinition findet sich in Anhang A. Es existieren Elemente für OS, SS, FS, NS, Gruppen, Ziele, Missionen, Schemata, Verbindungen etc.

OML wird zur Ausführung in die *Normative Organisation Programming Language (NOPL)* übersetzt, eine einfache Sprache, die im Wesentlichen aus einer Menge von Normen besteht (Hübner et al., 2010a,b). Zweck dieser Indirektion ist es, eine schlankere Organisation Management Infrastructure (OMI)-Implementierung in ORA4MAS zu realisieren, als sie bei direkter Nutzung von XML möglich wäre.

4 Domain-specific Languages

4.1 Motivation und Einsatz

Domain-specific Languages (DSLs), also domänenspezifische Sprachen, sind formale Sprachen, die einzig zur Formulierung von Aspekten eines bestimmten Fachbereichs, der Problemdomäne, entworfen werden. Der Sprachumfang einer DSL umfasst exakt die Problemdomäne, also keine anderen Aspekte einerseits, andererseits wird die Problemdomäne komplett abgedeckt.

Der sich aus dem Einsatz von DSLs ergebende Nutzen besteht darin, dass für die Problemdomäne relevante Entitätstypen direkt als Sprachelemente zur Verfügung stehen und nicht erst – wie bei einer General-purpose Programming Language – in einer abstrakteren Sprache formuliert werden müssen. Die Folge sind gesteigerte Produktivität bei weniger Fehlern durch den Einsatz einer DSL.

So schreiben Mernik et al. (2005, S. 317):

„By providing notations and constructs tailored toward a particular application domain, they [DSLs] offer substantial gains in expressiveness and ease of use compared with GPLs for the domain in question, with corresponding gains in productivity and reduced maintenance costs.“

Des Weiteren erfordert der Einsatz einer DSL vom Nutzer weniger Wissen – und damit weniger Einarbeitung – in Bezug auf Programmierung und auf die Problemdomäne als eine GPL. Während die Möglichkeit zum Verzicht auf Programmierkenntnisse im herkömmlichen Sinne beim Einsatz einer GPL direkt einleuchtet, erklären sich die niedrigen Anforderungen an die Kenntnis der Problemdomäne wie folgt: Da die Sprachsemantik der der Problemdomäne entspricht, lässt eine (gute) DSL inhaltlich unsinnige Formulierungen gar nicht erst zu. Selbstverständlich entbindet dies den Anwender nicht von der Pflicht, zu wissen, *was* er formulieren möchte; die Sprache leitet ihn jedoch bei der Frage, *wie* es zu formulieren ist.

Beispiele für weit verbreitete DSLs sind u. a. HTML, L^AT_EX und SQL, aber auch die bereits genannten AgentSpeak, KQML etc. Ähnlich wie (X)HTML kann auch jede an-

dere XML-basierte Sprache mit auf einen Anwendungsfall angepassten Elementen und Attributen als DSL betrachtet werden. Dies gilt auch für die Organisation Modelling Language bei \mathcal{MOISE}^+ . Der laut Fowler (2010, S. 103) jedoch wichtigste Vorteil einer DSL, die einfache Lesbarkeit für Menschen – vor allem solche ohne XML-Kenntnisse –, bleibt in all diesen Fällen aus.

4.2 Realisierungsmöglichkeiten

4.2.1 Interne Domain-specific Languages

Interne, oder auch *eingebettete*, DSLs sind in eine Wirtssprache integriert, wodurch ihnen sämtliche von dieser zur Verfügung gestellten Mechanismen und Sprachfeatures offen stehen.

Hudak (1996) betrachtet interne DSLs als die Antwort auf die Frage nach der idealen Abstraktionsstufe bei der Modellierung und Entwicklung von Software (Hervorhebung im Original):

„Although generality is good, we might ask what the ‘ideal’ abstraction for a particular application is. In my opinion, it is a *programming language* that is designed precisely for that application: one in which a person can quickly and effectively develop a complete software system. It is not general at all; it should capture precisely the semantics of the application domain—no more and no less. In my opinion, a domain-specific language is the ‘ultimate abstraction’.“

Wie viele der in der vorliegenden Arbeit geschilderten Konzepte zielen also auch DSLs darauf ab, dem Anwender eine Möglichkeit zu bieten, die Semantik seines Problemfelds direkt in maschinell verarbeitbarer Form zu nutzen.

Für die Integration interner DSLs in objektorientierte Wirtssprachen ohne Präprozessordirektiven und Meta-Programmierung nennt Dubochet (2006) drei Schlüsselaspekte:

1. Um Ausdrücke von Operanden und Operatoren der DSL einfach formulieren zu können, fordert Dubochet von der Wirtssprache die Möglichkeit, Methodenaufrufe in Infixnotation zu schreiben, also **operand₁ operator operand₂**. Die Operatoren sind dabei Ausdrücke der Wirtssprache, die den Gesamtausdruck als Methodenaufruf interpretiert: **operand₁.operator(operand₂)**.

2. Bei der Verkettung mehrerer der gerade beschriebenen Operationen sollte die Syntax der DSL sicherstellen, dass die richtige Reihenfolge eingehalten wird. Dies wird erreicht, indem jede als Operator dienende Methode ein Objekt eines Typs zurück gibt, der über seine Methoden definiert, welche Operatoren ab seiner Stelle in der Kette der Ausdrücke erlaubt sind.

So nennt Dubochet als Beispiel eine Klasse **Menge** deren Methoden **vereinigt** (**union**) und **verbund** (**join**) eine zweite Menge als Operator erwarten und selbst wieder eine Menge – die Vereinigung oder den Verbund eben – zurückgeben. Auf diesem Ergebnis sind weitere Operationen möglich. Sollen die möglichen Operatoren auf dem Ergebnis einer Operation andere sein, wird dies über den Rückgabotyp realisiert.

In Kapitel 6 wird deutlich, dass dieses Konzept in *MOISE*-DSL mehrfach zum Einsatz kommt, um die genau einmalige Angabe der für einen Ausdruck obligatorischen Attribute zu erzwingen.

3. Bei Methodenaufrufen in (stark typisierten) objektorientierten Sprachen müssen einerseits die erwarteten und übergebenen Typen kompatibel sein, andererseits muss der Typ des ersten Operands die gewünschte Operation (Methode) zur Verfügung stellen. Diese Bedingungen sind bei der Implementierung interner DSLs nicht immer leicht zu erfüllen, daher fordert Dubochet *Sichten*. Als solche bezeichnet er implizit aufgerufene Funktionen, die einen tatsächlichen in einen benötigten Typ konvertieren.

Erwartet beispielsweise bei dem Ausdruck $x.f(y)$ die Methode **f** einen anderen Typ als den von **y** oder bietet der Typ von **x** keine Methode **f** an, werden Sichten benötigt, die der Compiler automatisch aufruft. Im ersten Fall konvertiert diese **y** in einen zu **f** kompatiblen Typ, im zweiten Fall muss eine Sicht **x** in einen Typ konvertieren, der über eine Methode **f** verfügt.

Im Kontext von DSLs werden solche Sichten benötigt, wenn erstens ein Aufruf der Wirtssprache einen Datentyp als Parameter erwartet, der nicht innerhalb der DSL definiert wurde, das Parameter-Objekt aber per DSL erzeugt werden soll. Dies ist häufig der Fall, wenn DSLs genutzt werden, um Konfigurationen für Frameworks, auf deren Programmierschnittstelle kein Einfluss besteht, zu erstellen. Das zweite Einsatzgebiet von Sichten besteht darin, wenn ein Ausdruck innerhalb der DSL einen Typ erwartet, der via Sicht aus einem Typ der Wirtssprache erstellt wer-

den kann. Auf diese Art entfällt die Notwendigkeit, den Übergabeparameter erst aufwendig zu konstruieren.

Bei DSLs, die einen weniger formalen Charakter haben und eher an natürliche (englische) Sprache erinnern sollen, kommen häufig sog. *Füllwörter* (*bubble words*) zum Einsatz, die keine andere Funktion erfüllen, als dass sie die Lesbarkeit verbessern (Ghosh, 2011, S. 10:14). Es obliegt dabei dem Designer der Sprache, ob er solche – technisch eigentlich unnötigen – Worte bei der Eingabe über eigene Klassen erzwingen möchte – vgl. zweites Konzept von Dubochet (2006) – oder ob sie bei der Eingabe entfallen dürfen.

Ein nicht zu unterschätzender pragmatischer Vorteil interner DSLs besteht darin, dass sie bei ihrer Entwicklung und Nutzung von den i. d. R. ausgefeilten Werkzeugumgebungen ihrer Wirtssprache profitieren können. So stehen hier umfangreiche Debugger, Testwerkzeuge und -automatisierungsmöglichkeiten zur Fehlersuche und -vermeidung zur Verfügung. Integrierte Entwicklungsumgebungen mit Syntax-Highlightning, Wortvorschlägen etc. erleichtern die Eingabe.

4.2.2 Externe Domain-specific Languages

Externe DSLs sind solche, die nicht in eine andere Wirtssprache eingebettet sind, sondern autonom verarbeitet werden können. Dies geschieht i. d. R. mittels speziell für die jeweilige Sprache entworfener und umgesetzter Parser.

Der sich aus diesem Konzept ergebende Vorteil besteht in der komplett freien Wahl der Sprachsyntax. Bei internen DSLs bewegen sich die Gestaltungsmöglichkeiten für den Sprach-Designer in den – häufig engen – Grenzen, die die Wirtssprache ihm vorgibt. So können für die Wirtssprache reservierte Schlüsselwörter nicht genutzt werden oder es werden bestimmte Symbole zum Abschluss eines Ausdrucks erzwungen und dergleichen Einschränkungen mehr.

Nachteilig bei externen DSLs ist einerseits der häufig hohe Aufwand während der Implementierung, der jedoch durch Zuhilfenahme spezieller Werkzeuge und Frameworks gesenkt werden kann. Andererseits ist die Werkzeug-Unterstützung bei der *Nutzung* der Sprache in aller Regel schlecht. Externe XML-DSLs bilden hier die Ausnahme, da hier bereits die Sprachdefinition in einer Form (XML-Schemata) vorliegt, die es den heute üblichen XML-Editoren erlaubt, die Eingabe unterstützend und validierend zu begleiten. Umgekehrt gelten für XML-DSLs die in Abschnitt 4.1 genannten Nachteile, vor allem in Bezug auf Lesbarkeit für Menschen.

Als Weg zur besseren Eingabeunterstützung externer Nicht-XML-DSLs werden sog.

Language Workbenches vorgeschlagen, die auf Basis der Sprachdefinition entweder einen spezifischen Spracheditor neben dem eigentlichen Programm erzeugen oder aber ein Meta-Modell der Sprache, das ähnlich einem XML-Schema von verschiedenen generischen Editoren zur Eingabeunterstützung genutzt werden kann (Fowler, 2009; Hen-Tov et al., 2009).

5 Modell und Architektur

5.1 Vorgehen

Mernik et al. (2005) nennen als Phasen bei der Entwicklung einer DSL *Entscheidung*, *Analyse*, *Entwurf*, *Implementierung* und *Auslieferung*, wenngleich sie anmerken, dass die Entwicklung in der Praxis nicht als streng sequenzieller Prozess betrachtet werden kann. Auf diese Phasen wird im Folgenden eingegangen:

Entscheidung Während Mernik et al. (2005, S. 321) in erster Linie den betriebswirtschaftlichen Zielkonflikt zwischen der hohen anfänglichen Investition in eine DSL-Implementierung und den durch sie bedingten späteren Produktivitätssteigerungen als Entscheidungsproblem nennen, spricht im vorliegenden Fall wenig gegen die *MOISE*-DSL:

Zunächst kann das Für und Wider einer wissenschaftlichen Arbeit nicht in betriebswirtschaftlichen Dimensionen bewertet werden. Ferner ist die betrachtete Domäne bereits in Form der OML (vgl. Kapitel 3.2.3) formal beschrieben, sodass der für DSLs typische hohe Einarbeitungs- und Rechercheaufwand niedrig ausfällt. Des Weiteren zeigt die vorhandene OML die Eignung der Problemdomäne für die Implementierung von DSLs auf.

Schließlich sprechen die vielfältigen und langjährigen Bemühungen, die semantische Lücke zwischen Modell und Implementierung von Multiagentensystem zu schließen – von der AOP selbst über bereits bestehende DSLs wie AgentSpeak bis hin zu Agents and Artefacts (*CARTAgO*), *ORA4MAS* und *MOISE*⁺ –, aus Sicht des Autors dieser Arbeit fast zwingend für einen weiteren Schritt in diese Richtung, da das manuelle Bearbeiten einer OML-Datei in den Augen der meisten Anwender sicher nicht als Modellierung auf einem hohen, der Anwendungsdomäne angemessenen Abstraktionsniveau betrachtet werden kann.

Analyse Die Analyse der Problemdomäne gestaltet sich im vorliegenden Fall einfach, da sie bereits auf allen benötigten Ebenen ausreichend beschrieben ist: Es exis-

tieren verbale Beschreibungen der grundlegenden Konzepte und von Details, eine in der wissenschaftlichen Gemeinschaft etablierte Terminologie – wenn man von dem Begriff *Agent* absieht (vgl. Kapitel 2.1.1) – sowie formale Beschreibungen und lauffähige Implementierungen.

Entwurf Der Entwurf befasst sich mit der Frage des Zusammenhangs zwischen der betrachteten DSL auf der einen und formalen Aspekten wie der geplanten Implementierung der Sprache auf der anderen Seite. Abschnitt 5.2 geht ausführlich auf diese Fragestellungen ein.

Implementierung Die Phase der Implementierung beschreibt die eigentliche Umsetzung der zuvor entworfenen DSL als interne oder externe Variante (vgl. Kapitel 4.2). Kapitel 6 widmet sich ausschließlich diesem Aspekt.

Auslieferung Die Auslieferung einer DSL unterscheidet sich nicht von der jedes anderen Softwaresystems. Im Falle der *MOISE*-DSL muss diese nicht separat betrachtet werden, da die DSL gemeinsam mit der *MOISE*-Implementierung verteilt werden kann.

5.2 Designentscheidungen

5.2.1 Sprachcharakteristika und -funktionen

MOISE⁺ stellt mit OS, SS, FS und NS (vgl. Kapitel 3.2.2) bereits grundsätzliche Modellelemente zur Verfügung, deren Eignung für die Abbildung in einer DSL anhand von OML belegt werden kann. Daher erscheint die Übernahme selbiger als Sprachelemente für *MOISE*-DSL naheliegend und wird verfolgt.

Die von Hübner et al. (2002a, S. 124) gerne verwendete Notation für soziale Schemata „ $g_2 = g_6 \text{ , } (g_7 \mid g_8)$ “ mit g_n , $n \in \mathbb{N}$ als Zielen und „ $,$ “, „ \mid “ und (hier nicht zu sehen) „ \parallel “ als Operatoren für Sequenzen, Auswahlen und Parallelitäten weckt den Wunsch nach einer ähnlich formalen Ausdrucksmöglichkeit für die *MOISE*-DSL.

Allerdings offenbaren die meisten Konzepte in *MOISE*⁺ bei näherer Betrachtung derart viele Attribute, dass eine formale Ausdrucksweise für selbige das Hauptziel der *MOISE*-DSL, nämlich die einfache Lesbarkeit für menschliche Betrachter, konterkarieren würde. So verfügen Ziele in *MOISE*⁺ beispielsweise neben der obligatorischen Bezeichnung (hier g_n) über eine Beschreibung, einen Typ, eine Zeitvorgabe für ihre Erfüllung

sowie über eine Mindestanzahl von Agenten, die das Ziel in der Organisation zu verfolgen haben. Wenngleich einige dieser Angaben optional sind, wird offensichtlich, dass eine formale Beschreibung in Form eines Quintupels schnell unübersichtlich wird. In anderen Fällen müssten Tupel in einer formalen Darstellung der Sprache sogar verschachtelt oder verkettet werden.

Als Alternative nutzt *MOISE*-DSL daher eine an die englische Sprache angelehnte Notation, in der Ziele wie in Listing 5.1 Satz-artig formuliert werden können.

```
Goal named "site_prepared" described_as "prepare_the_site_for_the_
next_tasks" to reach in 20.minutes by_at_least 1.agent
```

Listing 5.1: Vollständige Definition eines Ziels in *MOISE*-DSL

Entsprechend beginnen *MOISE*-DSL-Ausdrücke mit einem groß geschriebenen Wort und folgen sonst Kleinschreibung. Leerzeichen innerhalb einzelner Terme werden durch Unterstriche ersetzt, ein Zeilenumbruch beendet einen Ausdruck. Innerhalb der *MOISE*-DSL-Implementierung (vgl. Kapitel 6) kommt dagegen die in der Programmierung übliche *CamelCase*-Notation zum Einsatz.

Um die gezeigte Schreibweise zu bewerkstelligen bedient sich *MOISE*-DSL des *Fluent Interface*-Entwurfsmusters, das technisch zwar einfach zu implementieren ist, sich konzeptionell jedoch stark von den in imperativen Programmiersprachen verbreiteten Programmierschnittstellen unterscheidet (Fowler, 2010, S. 69-70):

„Instead of a box of objects [...] we think linguistically of composing sentences [...]. It's this mental shift, that is the core difference between an internal DSL and just calling an API.“

Wie bereits erwähnt gibt es für Entitätstypen in *MOISE*⁺ obligatorische und optionale Attribute. In *MOISE*-DSL wird die Konvention getroffen, dass erstere immer direkt nach der Angabe des Entitätstyps (hier: **Goal**) erscheinen (in diesem Fall nur der Name des Ziels) und anschließend die übrigen Attribute.

Das letzte mit *MOISE*-DSL verfolgte Designziel besteht darin, die Sprache so stark wie möglich zu typisieren. Während in OML prinzipbedingt häufig *Magic Strings*, also Zeichenketten, deren Inhalt nicht beliebig verändert werden kann, ohne das Organisationsmodell unbewusst ungültig zu machen, zum Einsatz kommen, wird für *MOISE*-DSL gefordert, dass Parameter bereits zur Entwurfs- bzw. Übersetzungszeit auf Gültigkeit geprüft werden können.

5.2.2 Integration in Multiagentensysteme

Schnittstelle zwischen *Jason* und *Moise*⁺

Die in OML formulierte Organisational Specification wird in ein *Jason*-Multiagentensystem eingebunden, indem ein Agent mittels der von *CARTAgO* bereitgestellten Aktion *makeArtefakt* in *AgentSpeak* ein Artefakt erzeugt, dessen Typ von *ORA4MAS* (vgl. Abbildung 3.6 auf Seite 21) bereitgestellt wird.

```
makeArtifact("hsh_group", "ora4mas.nopl.GroupBoard", ["src/house-os.xml", house_group, false, true], GrArtId);
```

Listing 5.2: Erstellen eines GroupBoard-Artefakts in *AgentSpeak*

```
makeArtifact("bhsch", "ora4mas.nopl.SchemeBoard", ["src/house-os.xml", build_house_sch, false, true], SchArtId);
```

Listing 5.3: Erstellen eines SchemeBoard-Artefakts in *AgentSpeak*

Die Listings 5.2 und 5.3 demonstrieren dies für ein Group- und SchemeBoard, die als erste beide Parameter (in eckigen Klammern) die OML-Datei sowie die ID des gewünschten OML-Elements (*house_group* bzw. *build_house_sch*) übergeben bekommen.

Aus der Art der Verwendung der OS innerhalb des MAS wird deutlich, dass die Art und Weise, in der die Organisationsstruktur formuliert wurde, im Wesentlichen vor den sie nutzenden Agenten verborgen bleibt – wenn man von der Dateiendung *.xml* absieht. Diese Trennung der Belange zwischen Nutzung und Spezifikation der Organisationsstruktur gilt es beizubehalten. Aus Sicht der sie nutzenden Agenten – und ebenso aus Sicht von *CARTAgO* und *ORA4MAS* – soll eine in der *MOISE*-DSL formulierte OS sich nicht von der bisherigen OML-Variante unterscheiden. Um diese Forderung zu erfüllen, wurde ein Design gewählt, bei dem innerhalb von *MOISE*⁺ aufgrund der Dateiendung entschieden wird, ob eine OS als OML oder *MOISE*-DSL zu interpretieren ist. Auf diese Weise unterscheiden sich die beiden o. g. Listings bei Verwendung einer *MOISE*-DSL-Organisationsstruktur (außer in ihrem Dateinamen) nicht.

Der nächste Abschnitt erläutert, wie die Einhaltung dieser Forderung sichergestellt und die Integration innerhalb von *MOISE*⁺ vorgenommen wird.

Schnittstelle zwischen *Moise*⁺ und *Moise*-DSL

Die Implementierung des *MOISE*⁺-Modells liegt – wie die von *Jason*, *CARTAgO* und *ORA4MAS* – in Java vor und ist damit weitestgehend plattformunabhängig verfügbar.

Das intern genutzte Klassenmodell beinhaltet naheliegenderweise im Wesentlichen eine Klasse pro OML-Elementtyp. Diese Klassen werden beim Verarbeiten einer OML-Datei zur Laufzeit instanziiert. Die Implementierung der \mathcal{MOISE} -DSL sollte also in einer Form erfolgen, die als Ziel einer Verarbeitung das Erzeugen von Instanzen der Klassen in \mathcal{MOISE}^+ nicht nur ermöglicht, sondern wenn möglich unterstützt. Als Open Source-Implementierung steht \mathcal{MOISE}^+ beliebig tiefen technischen Anpassungen offen.

Für die Integration der \mathcal{MOISE} -DSL kommen drei grundsätzliche Varianten in Frage:

Direktes Erstellen von \mathcal{Moise}^+ -Klassen: Die erste und bei naiver Betrachtung nächstliegende Variante besteht darin, die o. g. intern in \mathcal{MOISE}^+ genutzten Klassen mittels \mathcal{MOISE} -DSL zu erzeugen und direkt in \mathcal{MOISE}^+ weiter zu verwenden. Diese Variante verspricht dank minimalen Overheads Laufzeiteffizienz. Ferner werden – vor allem interne – DSLs häufig eingesetzt, um bestehende Programmierschnittstellen, wie das \mathcal{MOISE}^+ -Klassensystem eine darstellt, komfortabler ansprechen zu können, sodass hier ein reicher Erfahrungsschatz und viele Entwurfsmuster zur Verfügung stehen.

Design eines in \mathcal{Moise}^+ -Klassen transformierbaren Klassensystems: Dieser Ansatz stellt der zuvor genannten Variante eine zusätzliche Abstraktionsschicht in Form eines eigens für die DSL entworfenen Klassensystems voran. Dies hat gegenüber dem ersten Ansatz den Vorteil beim Sprachdesign keine Zugeständnisse an das Ziel-Klassensystem machen zu müssen; der Freiheitsgrad steigt. Stattdessen wird die resultierende Komplexität in der zusätzlichen Transformationsschicht abgefangen, die einzig dazu dient, die zunächst mittels \mathcal{MOISE} -DSL erstellten Klassen in die letztendlich in \mathcal{MOISE}^+ verwendeten zu wandeln.

Design eines Klassensystems, das in OML-Code serialisiert werden kann: Bei näherer Betrachtung der \mathcal{MOISE}^+ -Implementierung offenbart sich ein gravierender Nachteil der ersten beiden Ansätze: Teile der Logik zum Erzeugen komplexerer Objektstrukturen des \mathcal{MOISE}^+ -Klassensystems sind auf genau die Methoden verteilt, deren Aufgabe eben das Parsen der OML-Dateien ist. Hierzu zählen beispielsweise das Assoziieren von Zielen mit den zugehörigen Schemata innerhalb der FS.

Die Entscheidung für eine der erstgenannten Varianten bedeutete folglich, diese Logik innerhalb der \mathcal{MOISE} -DSL-Implementierung wiederholen zu müssen, was aus Gründen der Redundanz und damit einhergehenden Problemen bzgl. Fehleranfälligkeit, Erweiterbarkeit und Wartbarkeit abzulehnen ist.

Eine naheliegende Alternative ist es, \mathcal{MOISE} -DSL als OML-Frontend zu entwerfen. Bei diesem Ansatz erzeugt die DSL ein Klassensystem, das das komfortablere Erstellen von OML erlaubt. Hierbei können zwei Alternativen unterschieden werden: Einerseits ist das manuelle Programmieren der Logik zum Erstellen von OML-Dateien möglich. Andererseits jedoch bietet jede verbreitete Programmiersprache Werkzeuge zum automatischen Generieren einer spezialisierten XML-Programmierschnittstelle auf Basis der zugehörigen XML-Schema-Definition. Die Nutzung solcher Werkzeuge ist zwar eher Implementierungsdetail als Architekturentscheidung, gleichwohl kann die Kenntnis solcher Techniken entscheidungsrelevant sein, da diese den Implementierungsaufwand und die Fehleranfälligkeit gegenüber direktem Programmieren der XML-Serialisierung signifikant senken.

Die – zumindest theoretisch mögliche – Variante, \mathcal{MOISE}^+ um das Verarbeiten eines eigenen Klassensystems zu erweitern, fällt wie die ersten beiden Ansätze aus Gründen der Redundanz – hier sogar in viel stärkerem Maße – aus.

Betrachtet man die dargestellten Alternativen, erscheint Variante 3 nicht nur aufgrund der Nachteile der anderen Kandidaten als erfolgversprechendster Ansatz, sondern bietet zusätzlich aus Architektursicht den Vorteil, dass sie durch Nutzung der explizit für die Kommunikation mit externen Spezifikationen vorgesehenen Schnittstelle (OML) die „sauberste“ Trennung der Belange bedeutet.

Abbildung 5.1 demonstriert die Abfolge sämtlicher zum Einsatz kommender Sprachen auf unterschiedlichen Abstraktionsstufen bei der Verarbeitung einer in \mathcal{MOISE} -DSL formulierten Organisational Specification zur MAS-Laufzeit.

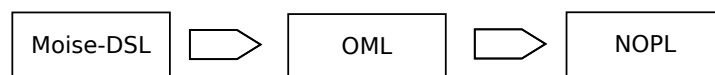


Abbildung 5.1: Sprachabfolge bei der Verarbeitung von \mathcal{MOISE} -DSL (eigene Abbildung)

5.2.3 Vorbereitung der Implementierung

Programmiersprache

Die durchgängige Nutzung der Java Virtual Machine als Laufzeitumgebung für die gesamte *Jason*-, *CARTAgO*- und \mathcal{MOISE}^+ -MAS-Umgebung setzt gleiches für sämtliche Erweiterungen wie die \mathcal{MOISE} -DSL voraus, sollen diese ernsthaft einsetzbar sein.

Zwar wären prinzipiell auch Entwicklungen in anderen ähnlich plattformunabhängigen Umgebungen denkbar, jedoch bedeuteten diese einen höheren Pflege- und Konfigurationsaufwand.

onsaufwand in Bezug auf Laufzeitumgebungen, Skripte und Konfigurationen für einzelne Plattformen etc.

Die naheliegende Variante, die *MOISE*-DSL ebenfalls in Java zu implementieren, gerät schnell gegenüber den Möglichkeiten, die die ebenfalls in einer Java Virtual Machine ausführbaren Kompilate der von Odersky et al. (2006) entworfenen objekt-funktionalen Hybridsprache *Scala* in Bezug auf DSLs bietet, ins Hintertreffen. Wampler und Payne (2009) beschreiben die Nutzung von Scala sowohl für interne als auch für externe DSLs ausführlicher.

Die Eignung einer Sprache für die Implementierung externer DSLs kann im Wesentlichen auf die Verfügbarkeit geeigneter Tools zum Erstellen von Sprachdefinitionen, Parsern etc. reduziert werden. Hier bietet Scala mit sog. *Parser Combinators* bereits von Haus aus ein mächtiges Werkzeug (Moors et al., 2008). Diese erlauben es, eine externe Sprache in einer an die erweiterte Backus-Naur-Form (ISO und IEC, 1996) angelehnten Syntax zu definieren und einfach zu verarbeiten.

Höhere Anforderungen an ihre Wirtssprache stellt die Implementierung einer internen DSL (vgl. Kapitel 4.2.1). Die dort von Dubochet geforderten Schlüsselaspekte erfüllt Scala vollumfänglich:

Methodenaufrufe mit genau einem (oder keinem, wenn es sich dabei um den letzten Aufruf in einer Methodenkette handelt) Parameter können in Scala in Infixnotation geschrieben werden, was die erste Forderung erfüllt.

Methodenverkettung ist generell in allen verbreiteten objektorientierten Programmiersprachen möglich; in Scala wird sie jedoch in besonderem Maße durch das Konstrukt **this.type** unterstützt (Odersky und Zenger, 2005), das es ermöglicht, dass bei Aufruf einer Methode auf einem Objekt einer Ableitung der die Methode definierenden Klasse dennoch der Typ der Ableitung zurückgegeben wird. Ohne dieses Konzept ist die Methodenverkettung bei Nutzung einer Vererbungshierarchie nur mit größerem Aufwand und Nachteilen hinsichtlich Wartbarkeit des Klassensystems möglich. Somit erfüllt Scala auch die zweite Forderung in besonderem Maße.

Als drittes fordert Dubochet die Möglichkeit zur Implementierung sog. *Sichten*. Scala unterstützt dieses Konzept in Form von *Implicits*, also als implizit aufzurufend gekennzeichnete Methoden, die vom Compiler nur wenn nötig zurate gezogen werden (Odersky et al., 2011, S. 480-503: Implicit Conversions and Paramters). Die Notwendigkeit ihres Aufrufs wird dabei durch die Differenz zwischen erwarteten und gefundenen Typen bzgl. Instanz oder Parameter einer aufgerufenen Methode determiniert.

Neben diesen Eigenschaften sprechen ferner

- benannte und Standard-Wert-Parameter für Methodenaufrufe (Rytz und Odersky, 2010),
- optionale Rückgaben, die hier insb. in Zusammenhang mit von Scalas Typsystem als Werttyp (als Gegenteil von Referenztyp) betrachteten *Case*-Klassen, auf die auch Pattern-Matching anwendbar ist (Odersky, 2006), genutzt werden sowie
- Mixins (in Scala *Traits* genannt),
- Typinferenz,
- in die Sprache integrierte Singletons und schließlich
- Closures

für den Einsatz von Scala – auch wenn jede andere Implementierung genauso wie interne DSLs von diesen Features profitiert. Funktionen höherer Ordnung sind in Scala als funktionaler Sprache selbstverständlich integraler Bestandteil.

Implementierung als interne oder externe DSL

Für die im Rahmen der vorliegenden Arbeit geplante Implementierung der $\mathcal{M}\mathcal{O}\mathcal{I}\mathcal{S}\mathcal{E}$ -DSL wurde von Beginn an Nutzbarkeit gefordert. Diese setzt einen Qualitätsstandard voraus, der entweder nur durch umfangreiche manuelle Tests, für die im vorliegenden Projekt jedoch keine Zeit vorgesehen war, sichergestellt werden kann oder durch Testautomatisierung.

Automatisierungswerkzeuge wie UnitTest-Frameworks stehen – gerade für die Java-Plattform – in großer Vielfalt zur Verfügung; nicht jedoch für externe DSLs.

Dies bildet gemeinsam mit der besseren Werkzeugunterstützung für das Erstellen (neben UnitTest-Frameworks auch integrierte Entwicklungsumgebungen mit Debugger etc.; vgl. Kapitel 4.2.1) und Nutzen der DSL sowie dem Designziel der Typsicherheit (vgl. Abschnitt 5.2.1) durch Überprüfung zur Übersetzungszeit der in $\mathcal{M}\mathcal{O}\mathcal{I}\mathcal{S}\mathcal{E}$ -DSL formulierten Modelle den Ausschlag für eine Implementierung als interne DSL.

Interne Architektur der DSL

Angelehnt an die originale $\mathcal{M}\mathcal{O}\mathcal{I}\mathcal{S}\mathcal{E}^+$ -Implementierung wird $\mathcal{M}\mathcal{O}\mathcal{I}\mathcal{S}\mathcal{E}$ -DSL in einzelne Pakete für OS, SS, FS und NS unterteilt.

Ferner gibt es ein Hilfspaket, das gemeinsam genutzte Programmbestandteile – hier die bereits erwähnten Implicits einerseits und andererseits eine gemeinsam genutzte Klasse für die Formulierung von Zeiteinheiten – beinhaltet.

Schließlich existiert ein Paket für das Konvertieren der *MOISE*-DSL-Klassen nach OML sowie eines, das den eigentlichen Interpreter im Sinne einer von außen nutzbaren Schnittstelle beinhaltet.

Abbildung 5.2 zeigt die Pakete inkl. ihrer Abhängigkeiten voneinander in der Übersicht. Besonderes Augenmerk liegt bei dieser Architektur darauf, dass die eigentlichen

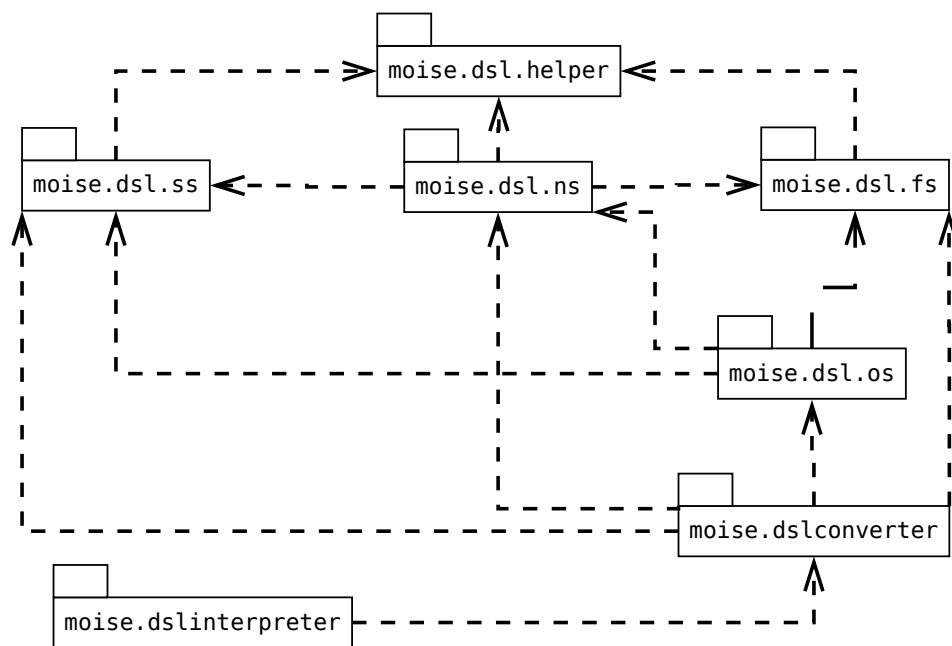


Abbildung 5.2: Paketdiagramm der *MOISE*-DSL-Architektur (eigene Abbildung)

DSL-Pakete (**moise.dsl.***) über keine externen Abhängigkeiten verfügen. Dies ermöglicht prinzipiell die Verwendung der unveränderten DSL in anderen Kontexten, wie beispielsweise einer Übersetzung in ein anderes Zielformat als OML durch Austauschen der verwendeten Konverter.

Für jedes der dargestellten Pakete existiert ferner eines mit entsprechenden UnitTests. Ein Paket mit Integrationstests stellt das korrekte Zusammenspiel der genannten und

jeweils bereits einzeln automatisiert testbaren Komponenten sicher und rundet so die Test-Suite der *MOISE*-DSL ab.

Weitere Details zu den gezeigten Paketen liefert Kapitel 6.

6 Implementierung

6.1 Grundlegende Konzepte

6.1.1 Allgemeines

Die Implementierung richtet sich in Aspekten der Code-Qualität, also beispielsweise der Länge von Klassen und Methoden, dem Tradeoff zwischen selbsterklärenden aber langen Bezeichnern und Kommentaren sowie der Einheitlichkeit von Abstraktionsleveln innerhalb eines Code-Abschnitts nach Martin (2009). Es folgen exemplarische Erläuterungen, wie die im Abschnitt zur Wahl der Programmiersprache in Kapitel 5.2.3 genannten Sprachfeatures in der Implementierung von *MOISE-DSL* genutzt werden, um die in Kapitel 5.2.1 beschriebenen Designziele zu erreichen.

Der Quellcodeverwaltung dient das frei verfügbare Versionierungswerkzeug *Subversion* (<http://subversion.apache.org/>).

6.1.2 Nutzung von Scala-Sprachfeatures und DSL-Konzepten

Infixnotation, Method Chaining und Standard-Parameter

Listing 6.1 zeigt die vollständige Implementierung des Sprachbestandteils für Normen in *MOISE-DSL*.

```
1 object Norm {  
2   def named (n: String) = EmptyNorm(n)  
3 }  
4  
5 case class EmptyNorm(val name: String) {  
6   def permits(r: Role) = NormWithoutMission(name, Permission, r)  
7   def obligates(r: Role) = NormWithoutMission(name, Obligation, r)  
8 }  
9
```

```

10 case class NormWithoutMission(val name: String, val normType:
    NormType, val role: Role) {
11     def participation_in(m: Mission) = Norm(name, normType, role, m)
12 }
13
14 case class Norm(val name: String,
    val normType: NormType,
    val role: Role,
    val mission: Mission,
    var timeConstraint: Option[TimeTerm] = None,
    var condition: Option[String] = None) {
15
16     def valid_for(t: TimeSpan) = {
17         timeConstraint = Some(t)
18         this
19     }
20
21     def valid(t: NotNumericTimeTerm) = {
22         timeConstraint = Some(t)
23         this
24     }
25
26     def with_the_condition_that (c: String) = {
27         condition = Some(c)
28         this
29     }
30
31 }

```

Listing 6.1: Vollständige Implementierung von Normen in *MOISE-DSL*

Eine Norm muss mindestens über einen Namen, einen Typ, eine Rolle und eine Mission verfügen. Optional gibt es eine Gültigkeitsdauer und eine Gültigkeitsbedingung.

Zunächst fällt auf, dass sowohl ein Objekt, also ein per Schlüsselwort definiertes Singleton, als auch eine Klasse **Norm** definiert wurden (Zeilen 1 und 14). Dieses Scala-Konstrukt heißt *Companion Object* und dient (unter anderem) dazu, die jeweilige Klasse ohne Konstruktor aufrufen zu können, indem Methoden auf dem Objekt deklariert werden (hier

named), die (hier indirekt) eine Instanz der zugehörigen Klasse erzeugen. Das Schlüsselwort **new** kann bei sog. *Case Classes* entfallen, für den Aufruf des Konstruktors generiert der Compiler eine implizit aufgerufene sog. **apply**-Methode auf dem Companion-Objekt.

Der Aufruf der Methode **named** des Norm-Objekts (Zeile 2) erstellt eine neue, noch leere Norm-Instanz (Klasse **EmptyNorm**), die ihrerseits Methoden zur Verfügung stellt, die einerseits die Rolle erwarten, auf die sich die Norm beziehen, soll und andererseits implizit über ihren Namen den Typ der Norm setzen (Zeilen 6 und 7). Ihre Rückgabe besteht in einer neuen Instanz, die eine Norm ohne Mission (Klasse **NormWithoutMission**) repräsentiert, und ausschließlich eine Methode zum Zuordnen einer Mission zu der Norm (Zeile 11) zur Verfügung stellt. Rückgabe ist schließlich die **Norm**-Instanz, auf der Methoden zum Setzen optionaler Parameter aufgerufen werden können.

Listing 6.2 zeigt, wie eine Kombination dieser Objekte und Klassen genutzt werden kann, um eine Norm mit dem Namen „n1“ zu erstellen, die die Rolle „house_owner“ verpflichtet, sich an der Mission „management_of_house_building“ innerhalb von zwei Minuten unter einer nicht näher ausgeführten Bedingung zu beteiligen.

```
Norm named "n1" obligates house_owner participation_in
    management_of_house_building valid_for 2.minutes
    with_the_condition_that "..."
```

Listing 6.2: Nutzung der **Norm**-Klasse in *MOISE*-DSL

Wie dieser Aufruf ohne Nutzung von Scalas Infixnotation aussähe, zeigt Listing 6.3. In diesem Fall ist die Nutzung von Methoden und Parametern deutlicher zu erkennen.

```
Norm.named("n1").obligates(house_owner).participation_in(
    management_of_house_building).valid_for(2.minutes).
    with_the_condition_that("...")
```

Listing 6.3: Nutzung der **Norm**-Klasse in *MOISE*-DSL ohne Infixnotation

Durch die Differenzierung in ein Objekt und drei unterschiedliche Klassen wird mittels Compiler sichergestellt, dass Normen immer alle Pflicht-Attribute enthalten. Würde ein Anwender versuchen, eine Norm nur per **Norm.named("n1")** zu erstellen, verweigerte der Compiler die Nutzung des Resultats, da eine Instanz des Typs **EmptyNorm** übergeben würde, wo der Compiler eine Instanz des Typs **Norm** erwartete.

Die Reihenfolge der letzten beiden Aufrufe (für Gültigkeitsdauer und -bedingung) darf vertauscht werden, da beide die aktuelle **Norm**-Instanz verändern und wieder zurückgeben (Zeile 28 bzw. 33).

Optionale Norm-Attribute sind als **Option[T]** typisiert (Zeilen 18 und 19), einer generischen Scala-Hilfsklasse, deren Ableitungen **None** und **Some[T]** dem Entwickler eine semantisch reichhaltigere Formulierungsmöglichkeit für bedingte Werte an die Hand geben als der in objektorientierten Sprachen häufig anzutreffende Workaround, einfach eine Instanz des jeweiligen Typs oder **null** zu nutzen. Die beiden optionalen Attribute von Normen werden standardmäßig mit **None** initialisiert.

Die gezeigte Implementierung zur Verkettung von Methoden mittels Rückgabe von **this** führt bei Vererbungshierarchien jedoch zu Problemen: In *MOISE*-DSL (genau wie in *MOISE*⁺) teilen sich Verbindungen (**Link**) und Kompatibilitäten (**Compatibility**) zwischen Rollen eine gemeinsame Basisklasse **RoleRel**, da beide ähnlich sind und zwei Rollen miteinander verbinden. Die meisten Operationen sind folglich in **RoleRel** implementiert. Anders als die Kompatibilität verfügt eine Verbindung jedoch über einen Typ (vgl. Kapitel 3.2.2), der mittels der in Listing 6.4 dargestellten Methode nach oben erläuterten Schema gesetzt werden kann.

```
def expresses(t: LinkType) = {  
  relType = Some(t)  
  this  
}
```

Listing 6.4: Setzen des Linktyps und Instanz-Rückgabe

Wurde in der Methodenverkettung jedoch unmittelbar vor dieser Methode eine aufgerufen, die in der Basisklasse **RoleRel** definiert wurde und ebenfalls **this** zurückgibt, gibt sie folglich eine **RoleRel**-typisierte Instanz zurück, für die jedoch keine Methode **expresses** definiert wurde.

Die Lösung dieses Problems liefert der Rückgabotyp **this.type**, der eine Rückgabe des aktuellen Typs statt einer Rückgabe des *die aktuelle Methode definierenden Typs* erzwingt. Listing 6.5 zeigt seine Verwendung.

```
def in (dir: RoleRelDirection): this.type = {  
  biDir = Some(dir.biDir)  
  this  
}
```

Listing 6.5: Setzen der Richtung für **Compatibility**- und **Link**-Instanzen definiert in deren Basisklasse **RoleRel**

In einigen (hier nicht gezeigten) Fällen ist es im Rahmen der Methodenverkettung aus sprachlichen Gründen wünschenswert, zwei unterschiedliche Worte bzw. Ausdrücke für die gleiche inhaltliche Aktion nutzen zu können. Dies wird durch die einfache Implementierung möglich, dass die Methode mit der einen Bezeichnung schlicht die mit der anderen Bezeichnung aufruft. Ein Beispiel zeigt Listing 6.9 auf Seite 43.

Case Classes und Pattern Matching

Case-Klassen bieten gegenüber „normalen“ Klassen neben einigen weiteren den hier ausschlaggebenden Vorteil, dass zwei unterschiedliche Instanzen einer Case-Klasse, deren per Konstruktorparameter festgelegte Attribute (siehe Zeile 14-19 in Listing 6.1 ab Seite 38) gleich sind, ebenfalls als gleich gelten. Im Scala-Typsystem gelten sie also nicht als Verweistypen, sondern als Werttypen. Dies erleichtert das Vergleichen von erwarteten und tatsächlichen Ergebnissen im Rahmen von automatisierten Tests erheblich (vgl. Abschnitt 6.5).

Ferner lassen Case-Klassen und -Objekte Pattern Matching zu, das beispielsweise bei der Konvertierung von Normtypen wie in Listing 6.6 genutzt werden kann, um für ein übergebenes Case-Objekt den zugehörigen OML-Attributwert in Form eines Strings zurückzugeben.

```
private def convertNormType(t: NormType) = t match {  
  case Obligation => "obligation"  
  case Permission => "permission"  
}
```

Listing 6.6: NormType-Konvertierung durch Pattern Matching

Traits

Traits in Scala können – je nach Sichtweise – als Interfaces mit Teil-Implementierungen oder als abstrakte Klassen mit Mehrfachvererbung betrachtet werden. Als solche eignen sie sich zur abstrakten Definition bestimmter Entitätstypen (hier Normtypen) wie die Listings 6.7 und 6.8 zeigen.

```
trait NormType
```

Listing 6.7: Trait zur abstrakten Definition möglicher Normtypen

```
case object Obligation extends NormType
case object Permission extends NormType
```

Listing 6.8: Case Objects definieren konkrete Normtypen

Zwar könnte für jeden Normtyp auch per Trait erzwungen und direkt in dessen Objekt implementiert werden, welcher OML-String ihm entspricht. Dies hätte den Vorteil, bei der Ergänzung weiterer Normtypen nur dessen Objekt hinzufügen zu müssen und nicht auch noch den in Listing 6.6 gezeigten Konverter-Code anpassen zu müssen. Jedoch würde eine solche Implementierung das Design-Ziel der Unabhängigkeit zwischen der DSL selbst, wozu der Normtyp zählt, und deren Konvertierungsziel (hier OML) verletzen. In der vorliegenden Implementierung dagegen ist der Normtyp völlig unabhängig davon, ob (und wenn ja welche) Strings in der Zielsprache für ihn genutzt werden. Nur der Konverter kennt diese Abhängigkeit.

Die Fähigkeit von Traits, bereits selbst Teil-Implementierungen zur Verfügung zu stellen, wird in *MOISE*-DSL genutzt, um mittels eines Traits **Monitorable** eine Methode **monitored_by** zur Verfügung zu stellen, die in den Klassen **Group** und **Scheme** genutzt wird. Sowohl Gruppen als auch Schemata in *MOISE*⁺ können von einem (anderen) Schema überwacht werden. Im Falle der Klasse **Group** werden gleich mehrere Traits für unterschiedliche Zwecke genutzt.

Implicits

Um Schreibweisen wie **1.agent** oder **2.agents** beispielsweise dann, wenn angegeben werden muss, wie viele Agenten ein Ziel zu verfolgen haben, zu ermöglichen, erwarten die jeweiligen Methoden als Übergabeparameter kein Integer sondern eine **AgentCount**-Instanz.

Die Klasse **AgentCount** kann nun die Methoden **agent** bzw. **agents** implementieren, wie Listing 6.9 zeigt.

```
case class AgentCount(val count: BigInt) {
  def agents = this
  def agent = agents
}
```

Listing 6.9: Die Klasse **AgentCount**

Bei Aufrufen von Methoden ohne Parameter dürfen in Scala die runden Klammern, in deren Mitte die Übergabeparameter angegeben werden, entfallen.

Damit die Integer-Literale `1` bzw. `2` diese Methoden nun aber zur Verfügung stellen können, müssen sie mittels der in Listing 6.10 gezeigten Methode implizit, also ohne weiteres Zutun des Anwenders, in `AgentCount`-Instanzen konvertiert werden¹.

```
implicit def intToAgentCount(i: Int) = AgentCount(i)
```

Listing 6.10: `Implicit` zur Konvertierung von Ganzzahlen in `AgentCount`-Instanzen

Ähnliche Implementierungen werden in `MOISE-DSL` beispielsweise genutzt, um Zeiteinheiten als `15.minutes` oder auch `1.hour and 15.minutes` schreiben zu können, wobei die eine angegebene Dauer repräsentierende Klasse `TimeSpan` mittels der Methode `and` die Addition von Dauern unterstützt. Es ist später Aufgabe des OML-Konverters die Einheit des größten gemeinsamen Teilers zu ermitteln; so würde das genannte Beispiel in OML in den String `75 minutes` konvertiert, `1.hour and 59.minutes and 60.seconds` dagegen in `2 hours`.

Bubble Words

Wie in Kapitel 4.2.1 erläutert, dienen Füllwörter nur der Lesbarkeit. In `MOISE-DSL` sind sie als Singleton mittels `object`-Schlüsselwort implementiert. Um die Übergabe eines solchen Objekts zu erzwingen, wird ihm ein Trait übergeordnet, wie Listing 6.11 zeigt. Ein Objekt kann nicht direkt als Übergabeparameter in einer Methodensignatur vorgesehen werden, da auf die Übergabe verzichtet werden könnte, würde ohnehin immer das gleiche Objekt übergeben. Ein Trait dagegen könnte potentiell mehrere Implementierungen haben, die alle übergeben werden könnten.

```
trait BubbleTraitForSubGroups
object subgroups extends BubbleTraitForSubGroups

abstract case class RoleRel( // ...
  def is_valid_for (g: BubbleTraitForSubGroups): this.type = {
    extendsToSubGroups = Some(true)
    this
  }
}
```

¹Die hier des Weiteren nötige implizite Konvertierung von `Int` nach `BigInt` für den `AgentCount`-Konstruktor erfolgt in Scala intern auf die gleiche Weise.

```

def is_not_valid_for (g: BubbleTraitForSubGroups): this.type = {
  extendsToSubGroups = Some(false)
  this
}
}

```

Listing 6.11: Bubble Word für Untergruppen-Gültigkeit

Auf diese Weise werden Schreibweisen wie Listing 6.12 möglich.

```

Link from role1 to role2 expresses communication is_valid_for
subgroups in both.directions

```

Listing 6.12: Nutzung des Bubble Words für Untergruppen-Gültigkeit

Die Alternative, auf den Übergabeparameter zu verzichten und die Methode von `is_valid_for` in `is_valid_for_subgroups` umzubenennen würde dazu führen, dass sie nicht mehr ohne weiteres in Infixnotation aufgerufen werden könnte, da Scala dies (ohne Einschränkung) nur für Methoden mit genau einem Parameter erlaubt (vgl. Abschnitt 5.2.3 auf Seite 33).

6.2 OML-Transformation

Wie in Kapitel 5.2.2 ausführlicher erläutert, wird aus *MOISE*-DSL OML zur Nutzung in *MOISE*⁺ erzeugt.

Aus Gründen des Aufwands und der Fehleranfälligkeit kommt dabei kein selbst geschriebener Code zum Einsatz sondern generierter. Hierzu werden mittels des XML-Datenbindungswerkzeugs *scalaxb*, das unter <http://scalaxb.org/> als Open Source zum Download zur Verfügung steht, alle erforderlichen Programmbestandteile erzeugt:

Klassen: *scalaxb* generiert aus der XML-Schema-Definition, die OML zugrunde liegt, Scala-Case-Klassen für Elemente und Typen.

XML-Konverter: Um Instanzen der zuvor generierten Klasse in XML serialisieren zu können, erstellt *scalaxb* Code, der auf Scalas leistungsstarker XML-Implementierung im Package `scala.xml` aufsetzt. Auch die hier nicht benötigte andere Richtung, um XML-Code in Instanzen der genannten Case-Klassen konvertieren zu können, wird selbstverständlich unterstützt.

Der mittels `scalaxb` generierte Code wird nicht direkt in `MOISE-DSL` eingebunden. Stattdessen bildet er ein eigenes Projekt, das in Form einer `jar`-Datei von der `MOISE-DSL`-Implementierung referenziert wird. Auf diese Weise ist sichergestellt, dass Änderungen am XML-Schema, die inhaltlich keine Anpassungen an `MOISE-DSL` erfordern, abgebildet werden können, indem lediglich der Code-Generator in `scalaxb` erneut aufgerufen wird.

Aus bereits erläuterten Gründen hinsichtlich einer modularen Software-Architektur werden die von `scalaxb` generierten Klassen ausschließlich im `moise.dslconverter`-Paket und nirgends sonst referenziert.

6.3 `moise.dsl`-Packages

6.3.1 Structural Specification: `moise.dsl.ss`

`RoleRel`, `Compatibility` und `Link` inkl. `LinkType`

`RoleRel` dient als abstrakte Basisklasse für Verbindungen und Kompatibilitäten, da beide zwei Rollen miteinander in Beziehung setzen. Die Klasse stellt Methoden und Attribute für Richtung, Reichweite und Untergruppengültigkeit der Beziehung zur Verfügung, wobei einige Bubble Words mit eigenen Traits und Objects zum Einsatz kommen. Ferner verfügt sie über Optionen für Beziehungstyp (Trait `RelType`) sowie Quelle und Ziel der Beziehung in Form von Rollen.

Ein Trait `RoleRelSignature` gibt die gemeinsame Signatur (mögliche Attribute und Methoden) der Klasse `RoleRel` sowie des `Link`-Objekts vor, sodass diese über die gleiche API nutzbar sind. Dies ist notwendig, da das *Objekt* selbst nicht von `RoleRel` erbt, sondern lediglich der Erzeugung von von `RoleRel` erbenden Instanzen für Verbindungen dient.

Kompatibilitäten müssen anders als Verbindungen *immer* über Quelle *und* Ziel verfügen und haben einen festen Typ, wie Listing 6.13 zeigt.

```
case object Compatibility {
  def from(r: Role) = EmptyCompatiblity(r)
}
case class EmptyCompatiblity(val from: Role) {
  def to(r: Role) = CompatibilityClass(from, r)
}
```

```

case object compatibility extends RelType
case class CompatibilityClass(val fromRole: Role,
                             val toRole: Role) extends RoleRel {
  relType = Some(compatibility)
}

```

Listing 6.13: Gesamte **Compatibility**-Implementierung

Die Nutzung erfolgt äquivalent zu Listing 6.12 auf Seite 45, nur dass keine Methode **expresses** zur Verfügung steht, da diese zur Angabe unterschiedlicher Linktypen dient.

Linktypen sind Spezialisierungen von Beziehungstypen und können entweder aus drei vorgegebenen gewählt oder selbst erstellt werden, wie Listing 6.14 zeigt.

```

trait LinkType extends RelType
case object authority extends LinkType
case object communication extends LinkType
case object acquaintance extends LinkType

object Linktype {
  def named(s: String) = CustomLinkType(s)
}

case class CustomLinkType(val name: String) extends LinkType

```

Listing 6.14: Implementierung der Linktypen

Links sind analog zu Compatibilities implementiert, können jedoch auf eine *Empty*-Klasse verzichten, da hier Quelle und Ziel optional sind. Stattdessen verfügt **LinkClass** über Methoden zum Setzen der beiden Endpunkte sowie des Linktyps.

Group

Gruppen verfügen über diverse Attribute und Methoden gemäß OML-Spezifikation. Ihre Implementierung ist straight forward und erfolgt gemäß dem bereits erläuterten Schema aus Companion-Objekt und einer zugehörigen Klasse. Gruppen können von Schemata überwacht werden und als Bezugsgröße für Kardinalitäten dienen, weshalb die Klasse **Group** die Traits **Monitorable** und **CardinalityType** (siehe Seite 48) implementiert.

Role

Rollen verfügen außer über einen Namen nur über eine Liste der Rollen, von denen sie erben. Die Methode zum Setzen selbiger ist mithilfe eines *repeated Parameter* implementiert (Odersky et al., 2011, S. 199-200). Sonst folgt die Implementierung dem gewohnten Paar aus Companion-Objekt und -Klasse.

Angabe von Minima und Maxima

OML ermöglicht für Rollen und Gruppen allgemein, Rollen in bestimmten Gruppen und für Untergruppen in übergeordneten Gruppen die Angabe von Mindest- und/oder Höchstzahlen. Zu diesem Zweck stellt das typisierbare Trait **Numerable[T]** die Methoden **at_least**, **and_at_most**, **at_most** und **exactly** zur Verfügung, die allesamt die Übergabe einer Instanz der Klasse **Number** erwarten und eines oder beide der optionalen **min**- und **max**-Attribute setzen.

Number dient dabei nur dem Zweck, statt beispielsweise **exactly 4** auch **exactly 4.times** schreiben zu können.

Ein Trait **CardinalityType** steht für die Angabe zur Verfügung, welche OML-Elemente als Kardinalitäten interpretiert werden dürfen und verlangt von diesen mindestens ein Attribut für die Bezeichnung. Genutzt werden **CardinalityType** implementierende Klassen als Typangabe für die Case-Klasse **Cardinality**, die **Numerable[Cardinality]** implementiert und ein Attribut vom Typ **CardinalityType** als Typangabe erwartet.

Die Case-Klasse **SubGroup** erwartet als Parameter die **Group**-Instanz, die die Untergruppe darstellt, und implementiert **Numerable[SubGroup]**, um Minimal- und Maximal Untergruppen anzugeben.

Die Case-Klasse **GroupRole** erwartet als Parameter eine Option auf **Role** zur Angabe der in der Gruppe auszuführenden Rolle und implementiert **Numerable[GroupRole]**, um deren Mindest- und Höchstanzahl festzulegen.

Die Selbst-Typisierung von **Numerable** ist nötig, da die Klasse **Group** über zwei Methoden gleichen Namens verfügt, die je ein Argument einer unterschiedlichen Implementierung von **Numerable** erwarten. Der Scala-Compiler in der zum Implementierungszeitpunkt aktuellen Version 2.8.1 ist jedoch nicht in der Lage, diese Überladung aufzulösen, wenn **Numerable** untypisiert ist und seine Methoden das bereits erläuterte **this.type** zurückgeben. Durch typisierte Rückgabe mittels **this.asInstanceOf[T]** kann dieses Problem umgangen werden.

SS

Companion-Objekt und -Klasse **SS** erlauben die Angabe keiner oder einer Gruppe sowie von Rollen und benutzerdefinierten Verbindungstypen in jeweils beliebiger Anzahl (inkl. Null) per repeated Parameter zur Definition der Structural Specification.

6.3.2 Functional Specification: **moise.dsl.fs**

SchemeElement, Goal und Plan

OML unterscheidet Ziele und Pläne streng voneinander. Goals stellen die eigentlichen Ziele dar und ein Plan gibt für zwei oder mehr Ziele an, mit welchem Operator (Sequenz, Auswahl, Parallelität) diese verknüpft sind. Pläne können nur Ziele beinhalten, Ziele nur Pläne. Das bedeutet, das in Kapitel 5.2.1 gezeigte Beispiel g_6 , $(g_7 \mid g_8)$ für drei miteinander in Beziehung stehende Ziele sieht tatsächlich in OML ausgedrückt sehr viel komplizierter aus, da es sinngemäß wie folgt geschrieben werden muss: $g_{w1}(\text{Plan}(\text{Sequenz}, g_6, g_{w2}(\text{Plan}(\text{Auswahl}, g_7, g_8))))$. Die Pläne wurden in *Wrapper-Goals* „verpackt“, da ein Plan keinen anderen beinhalten darf. Für MOISE-DSL soll jedoch eine an die erste Variante angelehnte Schreibweise – nur mit verständlicheren Operatoren – implementiert werden.

Folglich müssen für Ausdrücke wie den Erstgenannten implizit Varianten mit Plänen (der Zweitgenannte) erstellt werden. Aus Nutzersicht handelt es sich dann nur um verkettete Ziele, Pläne sind dem Anwender gar nicht bekannt.

Um dies zu erreichen, existiert die abstrakte Klasse **SchemeElement**, die die Methoden **then**, **or** und **parallel_with**, für die nötigen Operatoren zur Verfügung stellt. Diese Methoden erwarten selbst eine **SchemeElement**-Instanz als Operand und geben jeweils einen Plan zurück. Sowohl **Goal** als auch die ebenfalls abstrakte Klasse **Plan** erben von **SchemeElement**. Es existieren weiterhin drei (nicht abstrakte) Ableitungen von **Plan**, je eine pro Operator, also für sequenzielle, parallele und Auswahl-Pläne. Jede dieser drei Ableitungen überschreibt die Methode, die ihrem Operator entspricht. Diese Überschreibung fügt dann den ihr übergebenen Operanden in die eigene Operandenliste – alle drei Operatoren sind *n*-är – ein anstatt (wie sonst) einen neuen Plan zu erzeugen.

Pläne in OML müssen immer in ein Wrapper-Goal verpackt werden. Dieses Goal könnte theoretisch über sämtliche Attribute verfügen, die Ziele haben können (Name, Beschreibung etc.). Wenngleich in keinem MOISE⁺-Beispiel jemals Gebrauch von dieser Möglichkeit gemacht wurde, muss sie in MOISE-DSL ebenfalls abbildbar sein. Aus diesem Grund sind sämtliche Attribute nebst zugehörigen Methoden für Ziele nicht in **Goal**,

sondern bereits in der Basisklasse **SchemeElement** implementiert. Auf diese Weise stehen sie auch für *MOISE*-DSL-Pläne zur Verfügung, aus denen bei der OML-Transformation die zugehörigen Wrapper-Goals erzeugt werden (vgl. Abschnitt 6.4.1). Die zusätzliche Komplexität, die durch die Vereinfachung entsteht, aus Anwendersicht gänzlich auf Pläne verzichten zu können, wird so im Konverter abgefangen, das *MOISE*-DSL-Modell wird hiervon nicht tangiert. Während der Transformation werden auch Namen für ebendiese Goals erzeugt, da dies ein obligatorisches Attribut ist, Pläne in *MOISE*-DSL jedoch i. d. R. nicht benannt werden (wenngleich es aus Gründen der OML-Kompatibilität wegen des Wrapper-Goals möglich ist).

Zum Instanziiieren von **Goals** kommt das bereits übliche Companion-Paar zum Einsatz.

Mission

Missionen verfügen mindestens über einen Namen und ein oder mehrere OML-**Goal**-Elemente, die in *MOISE*-DSL wie beschrieben auf ein oder mehrere **SchemeElement**-Kindsinstanzen abgebildet werden.

Um diese obligatorischen Attribute zu erzwingen, kommt das in Abschnitt 6.1.2 beschriebene Verfahren mit mehreren Klassen statt nur zweier Companions zum Einsatz. Die Übergabe von einem oder mehreren **SchemeElement**-Parametern wird gelöst, indem als erster Parameter ein **SchemeElement** erwartet wird und als zweiter bis *n*-ter ein repeated Parameter vom Typ **SchemeElement**, was die leere Menge beinhaltet (der repeated Parameter allein ließe auch den vollständigen Verzicht auf eine Übergabe zu).

AgentCount und Monitorable

Die Klasse **AgentCount** dient der leserlicheren Angabe von Agentenanzahlen und ist bereits anhand des Listings 6.9 auf Seite 43 beschrieben. Das Trait **Monitorable** diene an gleicher Stelle bereits der Erläuterung des Einsatzes von Traits zum mehrfachen Nutzen einmalig implementierter Methoden.

Scheme und MonitoringScheme

Ein soziales Schema verfügt über genau ein Ziel (in *MOISE*-DSL eine **SchemeElement**-Kindsinstanz), die Option auf einen Namen und die Liste zugehöriger Missionen. Der Implementierung dienen die üblichen Companions.

Schemata und Gruppen können von einem (anderen) Schema überwacht werden. In OML geschieht dies durch Angabe des Namens des überwachenden Schemas beim überwachten Element. Dies bedingt, dass ein überwachendes Schema zwingend über einen Namen verfügen muss. Da *MOISE*-DSL zur Nutzung in *MOISE*⁺ nach OML übersetzt wird, gilt diese Implikation ebenfalls für *MOISE*-DSL. So unterscheidet sich die von **Scheme** ererbende Klasse **MonitoringScheme** von ihrem Vater darin, dass der Name hier obligatorisch ist.

Ein Schema wird durch Benennung automatisch ein **MonitoringScheme**. Der Anwender bekommt von diesem Unterschied nur dann etwas mit, wenn er versucht, ein unbenanntes Schema an die Methode **monitored_by** einer Gruppe oder eines anderen Schemas (über das Trait **Monitorable**) zu übergeben, da dies – wie gewünscht – zu einem Compiler-Fehler führt.

FS

Companion-Objekt und -Klasse **FS** erlauben die Angabe keines oder mehrerer sozialer Schemata per repeated Parameter und bilden so die Functional Specification vollständig ab.

6.3.3 Normative Specification: **moise.dsl.ns**

Norm

Da eine Norm mindestens über einen Namen, einen Typ und eine Rolle verfügt, kommen die in Abschnitt 6.1.2 erläuterten Klassen zum Einsatz, die deren Angabe erzwingen. Das Trait **NormType** mit den es implementierenden Case-Objekten **Obligation** und **Permission** vervollständigen die in Listing 6.1 auf Seite 38 dargestellte Implementierung.

NS

Companion-Objekt und -Klasse **NS** erlauben die Angabe keiner oder mehrerer Normen per repeated Parameter.

6.3.4 Organisational Specification und Sonstige

OS

Companion-Objekt und -Klasse `moise.dsl.os.OS` erlauben die Angabe genau einer SS sowie je einer Option auf einen Namen für die Organisational Specification, FS und NS. Hier wird die Zusammensetzung der OS aus ihren drei Bestandteilen SS, FS und NS sehr deutlich.

Ein intern genutztes Objekt `OSVersion` mit genau einem Attribut `versionString` dient als Konstante für die Angabe der jeweils unterstützten \mathcal{MOISE}^+ -Version (zum Zeitpunkt der Implementierung die aktuelle Version 0.8).

TimeTerm

Das Trait `moise.dsl.helper.TimeTerm` dient zur Definition von Zeitpunkten und Zeiträumen. Für numerische Zeiträume steht die Case-Klasse `TimeSpan` zur Verfügung. Die Case-Objekte `now` und `never` implementieren das Trait `NotNumericTimeTerm`, das von `TimeTerm` erbt.

Abbildung 6.3 auf Seite 61 zeigt den vollständigen Code (dort im Zusammenhang mit Testabdeckung).

Implicits

Im Objekt `moise.dsl.helper.Implicits` sind implizit aufzurufende Methoden definiert, die bei Bedarf die in Tabelle 6.1 definierten impliziten Konvertierungen vornehmen.

Quelltyp	Zieltyp
Int	TimeSpan
Int	AgentCount
Int	Number
Group	SubGroup
Role	GroupRole
CardinalityType	Cardinality
GroupRole	Cardinality
SubGroup	Cardinality

Tabelle 6.1: Implizite Konvertierungen in \mathcal{MOISE} -DSL

Die impliziten Methoden können im Wesentlichen in zwei Gruppen unterschieden werden. Die erste Gruppe besteht aus den **Int**-Konvertierungen. Diese sind nötig, um auf

Integer-Literalen die von den Zieltypen definierten Methoden aufrufen zu können. In den meisten Fällen dient dies der Leserlichkeit, im Falle der **TimeSpan** ferner der Nutzung einer oder gar mehrerer kombinierter Zeiteinheiten.

Die zweite Gruppe dient der vereinfachten Nutzung des Objektmodells, indem sie die Übergabe von Instanzen erlaubt, aus denen *inhaltlich* die erwarteten Informationen hervorgehen. Sie werden in den impliziten Methoden genutzt, um Instanzen des eigentlich erwarteten Typs zu konstruieren.

6.4 Konverter und Interpreter

6.4.1 Konverter

Grundlegende Konvertierungskonzepte

Konverter im Paket `moise.dslconverter` dienen der Konvertierung der *MOISE*-DSL-Klassen in die per `scalaxb` erstellten Klassen (siehe Abschnitt 6.2), die sich direkt auf OML-Elemente abbilden lassen.

Konverter rufen sich dabei gegenseitig auf: Zuoberst steht der **OSConverter**, der Teile seiner Arbeit an **SSConverter**, **FSConverter** und **NSConverter** delegiert, die ihrerseits spezialisierte Konverter für einzelne Entitätstypen nutzen. Alle Konverter sind als Scala-Objekt, also als Singleton, implementiert, da sie nur genau einmal benötigt werden.

Konverter machen exzessive Nutzung von benannten Parametern in Scala. Dieses auf den ersten Blick kaum nennenswerte Sprachfeature kann hier gemäß Clean Code-Paradigma der Vermeidung von Kommentaren dienen: Da die – im Folgenden und im Konverter-Code als Xb-Klassen bezeichneten – von `scalaxb` erstellten Klassen im Paket `oml` nicht dem Gestaltungsraum des *MOISE*-DSL-Entwicklers (und Code-Lesers) unterliegen, kennt er die Parameterfolgen der Konstruktoren dieser Klassen nicht. Folglich würde ein Leser des Codes Kommentare benötigen, um die Abbildung der *MOISE*-DSL auf die Xb-Klassen zu verstehen. Durch den Einsatz benannter Parameter können diese entfallen.

Listing 6.15 illustriert sämtliche zuvor genannten Aspekte anhand eines Beispiels.

```
import moise.dsl.fs.FS
import oml.{FsType => FsXb}

object FSConverter {
  def convert(fs: FS) = FsXb(properties = None,
```

```

    scheme = fs.schemes map {
      SchemeConverter.convert(_) })
  }

```

Listing 6.15: Vollständiger **FSConverter**

Zur Konvertierung von **FS**-Instanzen wird auf der Liste der Schemata die Funktion höherer Ordnung **map** aufgerufen, um die Konvertierung jeden Listeneintrags an den **SchemeConverter** zu delegieren und deren Ergebnis in eine neue Liste zu übernehmen. Diese wird dem Konstruktorparameter **scheme** der Xb-Klasse **oml.FsType** (hier per Import in **FsXb** umbenannt) übergeben.

Grundsätzlich erfolgt die Konvertierung von Traits und deren Implementierungen, die ausschließlich der typsicheren Angabe bestimmter Konstanten dienen, wie bereits in Listing 6.6 auf Seite 42 illustriert.

Nach der Erläuterung der allgemein genutzten Konzepte werden im Folgenden nur noch Konvertierungen erläutert, die entweder von den genannten Mechanismen abweichen oder diese um weitere Aspekte ergänzen.

Konvertierung von Gruppen und Untergruppen

Der **GroupConverter** konvertiert eine Gruppe inkl. ihrer Untergruppen rekursiv, wie Algorithmus 6.1 zeigt.

Konvertierung sozialer Schemata

Bei der Konvertierung von Zielen und Plänen generiert der **SchemeElementConverter** Namen für die bereits beschriebenen Wrapper-Goals. Da Namen eindeutig sein müssen, werden bereits genutzte Namen vorgehalten und nicht doppelt verwendet – auch dann, wenn ein explizit definiertes und benanntes Ziel in einem Namenskonflikt mit einem Wrapper-Goal steht. Algorithmus 6.2 verdeutlicht das Vorgehen. Die tatsächliche Implementierung erfolgt mittels verschachtelter Methoden.

Des Weiteren unterscheidet OML zwischen Zielen und Zieldefinitionen. Während Zieldefinitionen über die in **MOISE-DSL** für Ziele definierten Attribute verfügen, referenzieren OML-Ziele nur eine Definition anhand ihres Namens. Dank starker Typisierung kann diese – inhaltlich ohnehin nicht nötige – Unterscheidung in **MOISE-DSL** entfallen, da hier anstelle eines Namens direkt eine Referenz auf eine **Goal**-Instanz genutzt werden kann. Der **GoalConverter** stellt aus diesem Grund zwei Konvertierungen (nach Ziel und

```

procedure CONVERTGROUP(group)
  result  $\leftarrow$  CONVERTSUBGROUP(group,  $\emptyset$ ,  $\emptyset$ )       $\triangleright$  Min/Max nicht gesetzt ( $\emptyset$ )
  return result
end procedure

procedure CONVERTSUBGROUP(g, ming, maxg)       $\triangleright$  Min/Max f. Untergr. möglich
  r  $\leftarrow$   $\emptyset$                                  $\triangleright$  Rückgabe
  r.roles  $\leftarrow$  CONVERTROLES(g.roles)
  r.links  $\leftarrow$  CONVERTLINKS(g.links)
  for all (s, mins, maxs)  $\in$  g.subGroups do
    subgroup  $\leftarrow$  CONVERTSUBGROUP(s, mins, maxs)
    Append subgroup to r.subGroups
  end for
  r.constraints  $\leftarrow$  CONVERTCONSTRAINTS(g.cardinalities, g.compatibilities)
  r.min  $\leftarrow$  ming
  r.max  $\leftarrow$  maxg
  r.monitoringscheme  $\leftarrow$  g.monitoringScheme's name if existing or  $\emptyset$ 
  return r
end procedure

```

Algorithmus 6.1: Rekursive Konvertierung von Gruppen

```

procedure CONVERTSCHEMEELEMENT(s)
  if s is a Goal then       $\triangleright$  nicht rekursiv, da Ziele in MOISE-DSL (nicht jedoch in
  OML!) immer Blätter des Schema-Baums sind
    result  $\leftarrow$  CONVERTGOAL(s,  $\emptyset$ )       $\triangleright$  2. Parameter ist Name des zugeh. Plans
    REMEMBERNAME(s)
  else                                 $\triangleright$  s ist ein Plan
    children  $\leftarrow$   $\emptyset$ 
    for all child  $\in$  s.Children do
      elem  $\leftarrow$  CONVERTSCHEMEELEMENT(child)
      Append elem to children
    end for
    name  $\leftarrow$  GENERATEUNUSEDNAMEFOR(s)
    REMEMBERNAME(name)
    g  $\leftarrow$  CREATEWRAPPERGOAL(s, name)       $\triangleright$  setzt alle Attr. des Ziels aus s
    result  $\leftarrow$  CONVERTGOAL(g, children)
  end if
  return result
end procedure

```

Algorithmus 6.2: Implizite Erstellung von Zielen bei der Konvertierung von Plänen

nach Zieldefinition) zur Verfügung, wobei erstere schlicht den Namen der zu konvertierenden Instanz als String zurückgibt. Der übergeordnete **SchemeConverter** beschränkt sich im Wesentlichen auf Aufrufe des **SchemeElementConverters** und des hier nicht weiter erläuterten **MissionConverters**.

Konvertierung von Zeit-Ausdrücken

Der für die Konvertierung von Zeiteinheiten zuständige **TimeTermConverter** ermittelt durch Modulo-Operationen die größtmögliche Zeiteinheit für die Rückgabe von Zeitspannen (vgl. Abschnitt 6.1.2). Nicht-numerische Zeitangaben werden gemäß des bekannten Trait-nach-String-Musters konvertiert.

Umwandeln der Konvertierungsergebnisse in XML

Der sog. **XMLConverter** unterscheidet sich von den bisher beschriebenen Konvertern insofern, dass sein Ergebnistyp *keine* Xb-Klasse ist. Stattdessen fungiert er als Wrapper um alle anderen Konverter und nutzt als solcher den **OSConverter** als Einstiegspunkt. Als Eingabe dient daher eine **MOISE-DSL-OS**-Instanz, die mithilfe der kaskadierten Konverter intern in Instanzen von Xb-Klassen gewandelt wird. Diese werden im Anschluss per `scalaxb` in eine `scala.xml.Elem`-Instanz transformiert, die die Rückgabe des **XMLConverters** bildet. Er kann daher als Blackbox für die Erstellung von OML aus **MOISE-DSL** gesehen werden, die intern `scalaxb` nutzt, das jedoch vor dem Anwender verbirgt.

6.4.2 Interpreter

Der **MOISE-DSL**-Interpreter dient dazu, eine in **MOISE-DSL** formulierte und in einer separaten Textdatei abgelegte Organisational Specification zur MAS-Laufzeit in einen OML-String zu wandeln. Hierfür nutzt er intern den **XMLConverter**.

Der Interpreter stellt ferner sicher, dass die **MOISE-DSL**-Datei leserlich bleibt, indem in ihr auf das Importieren der notwendigen Pakete und auf das Definieren einer Klasse verzichtet werden kann. Hierfür ergänzt er einerseits die nötigen Importe von `ss._`, `fs._`, `ns._` und `helper.Implicits._` im Namensraum `moise.dsl` selbstständig. Der Verzicht auf eine Klassendefinition andererseits wird durch den Einsatz einer Instanz des Typs `scala.tools.nsc.Interpreter` erreicht. Diese ermöglicht das Interpretieren auch *einzelner* Scala-Ausdrücke. Innerhalb einer solchen Umgebung, werden Rückgaben

mit *res* bezeichnet und durchnummeriert, sodass der Wert der ersten Rückgabe implizit immer einer Variable **res0** zugewiesen wird. Per Konvention muss die erste (und einzige genutzte) Rückgabe einer *MOISE*-DSL-Datei die definierte OS zurückgeben, die entsprechend vom Interpreter abgefragt und weiterverarbeitet werden kann.

Das Ergebnis des *XMLConverters* wird anschließend nur noch in einen String gewandelt.

Abbildung 6.1 illustriert den gesamten Ablauf von AgentSpeak in *Jason* begonnen bis hin zur *MOISE*-DSL und ihrer internen Nutzung von *scalaxb*.

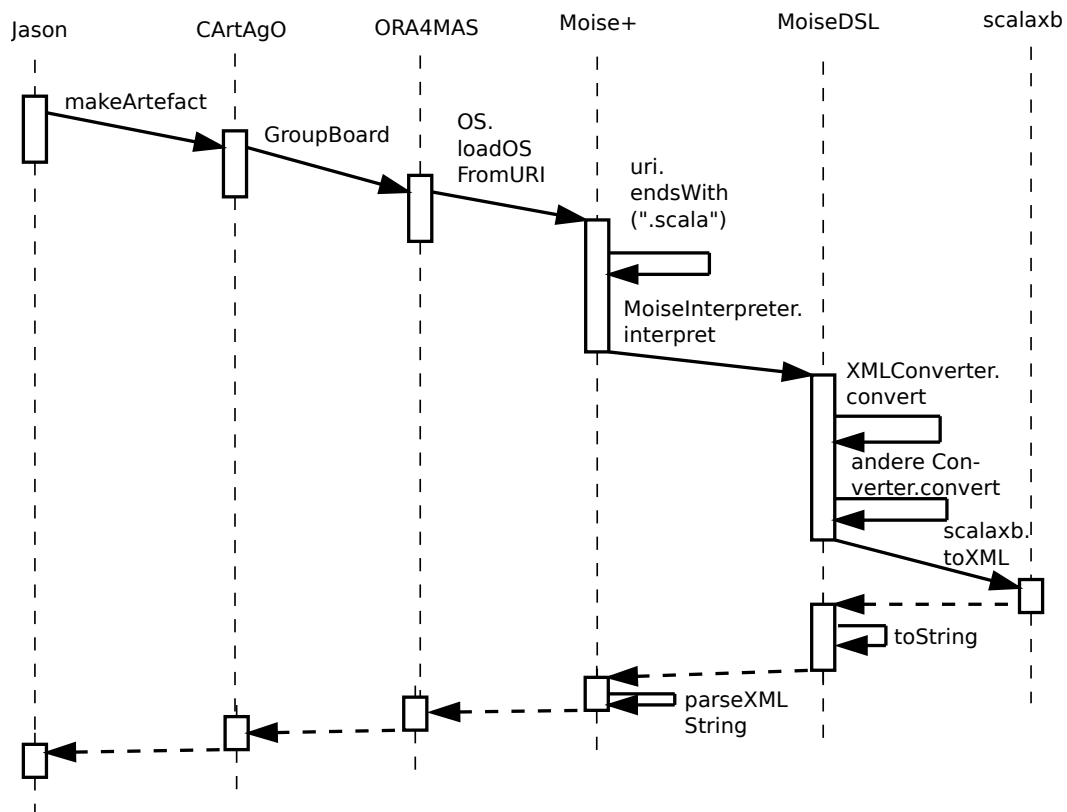


Abbildung 6.1: Sequenzdiagramm der MAS-Integration unter Nutzung des *MOISE*-DSL-Interpreters am Beispiel des *GroupBoards* (eigene Abbildung)

6.5 Testautomatisierung

6.5.1 Tests

Um die Qualität der *MOISE*-DSL-Implementierung sicher zu stellen, existieren insgesamt 114 UnitTests. Zusammengehörige Tests sind in gemeinsamen Testklassen gruppiert, die wiederum in Packages angeordnet sind. Diese Aufteilung wird im Namensschema der Tests in Abbildung 6.2 sichtbar.

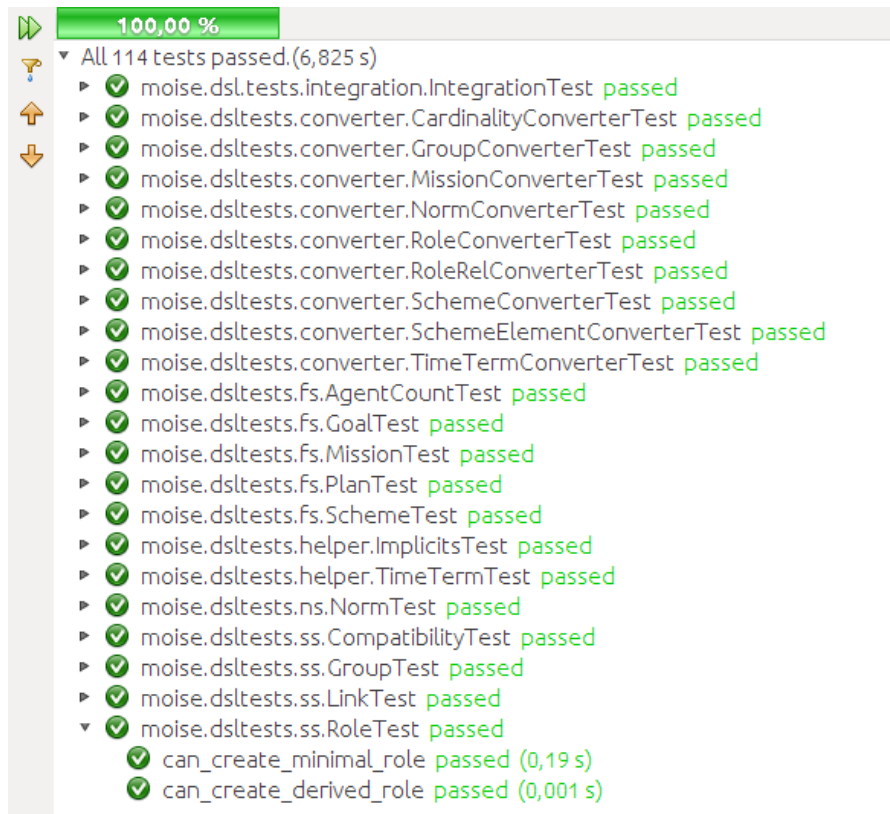


Abbildung 6.2: Vollständiger Testlauf für *MOISE*-DSL in NetBeans (eigenes Bildschirmfoto)

Technische Basis der Tests ist das UnitTesting-Framework *JUnit* für die Java-Plattform, das unter <http://www.junit.org/> als Open Source zum Download zur Verfügung steht. Es erlaubt die Definition von Tests mittels Annotationen und stellt Methoden wie `assertEquals` zum Prüfen der Ergebnisse zur Verfügung.

Um auch für Tests Übersichtlichkeit und Wartbarkeit langfristig sicherstellen zu können, folgen diese dem *3-As* bzw *Arrange-Act-Assert*-Pattern (Beck, 2003, S. 97), das Tests in drei funktionale Blöcke zur Vorbereitung der zu testenden Operation, der ei-

gentlichen Operation und dem Vergleich von erwartetem und tatsächlichem Ergebnis aufteilt. Dieses Schema wird in Listing 6.16 deutlich.

```
@Test
def can_create_norm_without_mission = {
  // arrange
  val en = Norm named "empty"
  val r = Role named "role"
  val expected = NormWithoutMission("empty", Permission, r)

  // act
  val nwm = en permits r

  // assert
  assertEquals(expected, nwm)
}
```

Listing 6.16: UnitTest mit *arrange*, *act* und *assert*

Obwohl es sich hierbei um einen technisch trivialen Ansatz – er besteht lediglich aus drei Kommentaren – handelt, zeigt die Praxis, dass er in der Lage ist, Test-Code langfristig nutzbar zu halten.

Methodennamen von Tests folgen in *MOISE-DSL* *nicht* der CamelCase-Notation, sondern bestehen aus Kleinbuchstaben; einzelne Worte sind per Unterstrich getrennt. Zwar ergibt sich daraus der Vorteil, bereits am Methodennamen einen Test von einer „normalen“ Methode unterscheiden zu können, eigentliche Ursache ist jedoch, dass der Methodename in Form eines Satzes beschreibt, *was* getestet wird. Ganze Sätze in CamelCase-Notation werden schnell unleserlich.

Das gezeigte Listing illustriert ferner den in Abschnitt 6.1.2 angesprochenen Vorteil von Case-Klassen: Dadurch, dass **NormWithoutMission** als Case-Klasse definiert wurde, kann die Gleichheit zwischen dem tatsächlichen (**nwm**) und dem erwarteten Ergebnis (**expected**) in einer einzelnen Zeile geprüft werden. Bei einer „normalen“ Klasse müssten der Name der Norm, der Typ und die Rolle separat überprüft werden, wobei für die Rolle wieder jedes Attribut einzeln verglichen werden müsste.

6.5.2 Testabdeckung

Die erreichte Testabdeckung beträgt weit über 86 Prozent des Codes. Diese Angabe ist wie folgt zu interpretieren: In Ermangelung spezieller Testabdeckungswerkzeuge für Scala kommt das Java-Tool *Cobertura* (<http://cobertura.sourceforge.net/>) zum Einsatz. Da es die Testabdeckung während der Ausführung von UnitTests auf Basis des generierten Java-Bytecode ermittelt, ist es prinzipiell für die Nutzung mit jeder Sprache, die diesen Bytecode als Kompilat erzeugt, geeignet.

Allerdings erzeugt der Scala-Compiler zusätzlichen Bytecode, der nicht auf manuell geschriebenen Code zurückgeht. Ein Beispiel hierfür sind automatisch generierte Companion-Objects bei Nutzung von Case-Klassen. Wie generell bei anderem generierten Code auch, ist dieser für Testautomatisierung und -abdeckung nicht relevant. Cobertura unterscheidet jedoch solchen Bytecode nicht von dem, dessen Ursprungscode manuell erstellt wurde, und zieht die Abdeckung im automatisch generierten Code für die Gesamtstatistik heran.

Abbildung 6.3 illustriert dies an einem Beispiel: Zwar zeigt der *Coverage Report* im unteren Teil den manuell geschriebenen Code korrekt an und hinterlegt jede Zeile mit einem Farbcode für durchlaufene Zeilen bzw. Kanten im Kontrollflussgraphen. Dennoch zeigt die Statistik im oberen Bereich niedrigere Werte als solche, die sich durch die abgebildeten Zeilen ergeben müssten.

Da Cobertura für die gesamte Implementierung der *MOISE*-DSL bereits eine (Zeilen-) Überdeckung von 86 Prozent angibt, kann davon ausgegangen werden, dass der tatsächliche Wert weit darüber liegt. Trotz dieser Hindernisse liefert Cobertura, vor allem dank der optischen Hervorhebung der bei den Tests durchlaufenen Zeilen, wichtige Hinweise auf die Qualität der UnitTests. Insbesondere bei den teils komplexeren Verfahren der Konvertierung in Instanzen von Xb-Klassen (vgl. Abschnitt 6.4.1) kann so sichergestellt werden, dass im Rahmen der Tests jede relevante Zeile durchlaufen und somit geprüft wird.

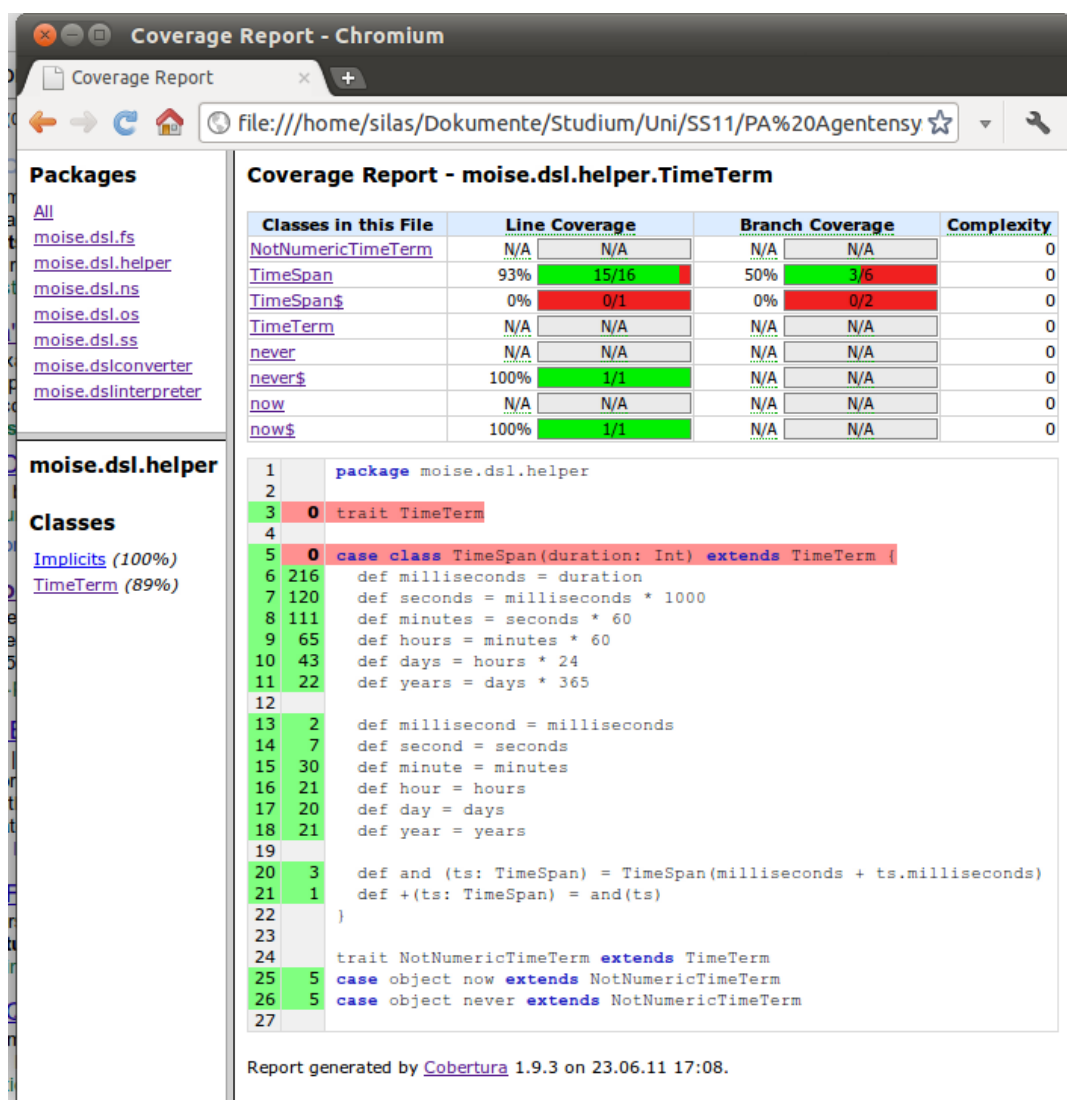


Abbildung 6.3: Testabdeckung für TimeTerm (eigenes Bildschirmfoto)

6.6 Anpassungen an $\mathcal{M}oise^+$

Designziel von $\mathcal{M}oise$ -DSL war unter anderem die für die Integration der Sprache in $\mathcal{M}oise^+$ notwendigen Anpassungen zu minimieren (vgl. Kapitel 5.2.2).

Abbildung 6.4 zeigt anhand der nur fünf geänderten Zeilen bei der Verarbeitung des OS-Elements einer OML-Datei, dass dieses Ziel als erreicht betrachtet werden kann.

Listing 6.17 zeigt darüber hinaus die ergänzten Methoden, die im Wesentlichen den vom $\mathcal{M}oise$ -DSL-Interpreter zurückgegebenen OML-String in ein XML-Dokument zur weiteren Verarbeitung konvertieren. Als reine Ergänzung wäre es ebenso möglich gewesen, die gezeigten Methoden in eine externe Bibliothek auszulagern, weshalb sie nicht

als Anpassung an MOISE^+ im engeren Sinne zu sehen sind.

Base (BASE)	3/4	Locally Modified (Based On LOCAL)
<pre> ele.appendChild(getSS().getAsDOM(document)); ele.appendChild(getFS().getAsDOM(document)); ele.appendChild(getNS().getAsDOM(document)); return ele; } public static OS loadOSFromURI(String uri) { try { Document doc = DOMUtils.getParser().parse(uri); DOMUtils.getOSSchemaValidator().validate(new DOMSource(doc)); OS os = new OS(); os.setFromDOM(doc); return os; } catch (ParserConfigurationException e) { System.err.println("Parser creation error:"+e); e.printStackTrace(); } catch (SAXException e) { System.err.println("Parser creation error:"+e); } catch (IOException e) { System.err.println("IO error:"+e); e.printStackTrace(); } catch (MoiseXMLParserException e) { System.err.println("Moise+ error:"+e); } catch (MoiseConsistencyException e) { System.err.println("Moise+ error:"+e); } catch (MoiseException e) { System.err.println("Moise+ error:"+e); e.printStackTrace(); } return null; } </pre>	<pre> 104 102 105 103 106 104 107 105 108 106 109 107 110 108 111 109 112 110 113 111 114 112 115 113 116 114 117 115 118 116 119 117 120 118 121 119 122 120 123 121 124 122 125 123 126 124 127 125 128 126 129 127 130 128 131 129 132 130 133 131 134 132 135 133 136 134 137 135 </pre>	<pre> ele.appendChild(getPropertiesAsDOM(document)); } ele.appendChild(getSS().getAsDOM(document)); ele.appendChild(getFS().getAsDOM(document)); ele.appendChild(getNS().getAsDOM(document)); return ele; } public static OS loadOSFromURI(String uri) { Document doc; try { if (uri.endsWith(".scala")) doc = readFromScalaDSLFile(uri); else doc = DOMUtils.getParser().parse(uri); DOMUtils.getOSSchemaValidator().validate(new DOMSource(doc)); OS os = new OS(); os.setFromDOM(doc); return os; } catch (ParserConfigurationException e) { System.err.println("Parser creation error:"+e); e.printStackTrace(); } catch (SAXException e) { System.err.println("Parser creation error:"+e); } catch (IOException e) { System.err.println("IO error:"+e); e.printStackTrace(); } catch (MoiseXMLParserException e) { System.err.println("Moise+ error:"+e); } catch (MoiseConsistencyException e) { System.err.println("Moise+ error:"+e); } } </pre>

Abbildung 6.4: Integration von MOISE -DSL in MOISE^+ (eigenes Bildschirmfoto)

```

public static Document readFromScalaDSLFile(String uri) throws
    SAXException, IOException {
    return loadXMLFrom(moise.dslinterpreter.MoiseInterpreter.
        interpret(uri));
}

private static Document loadXMLFrom(String xml)
    throws org.xml.sax.SAXException, java.io.IOException {
    return loadXMLFrom(new java.io.ByteArrayInputStream(xml.getBytes
        ()));
}

private static Document loadXMLFrom(java.io.InputStream is)
    throws org.xml.sax.SAXException, java.io.IOException {
    javax.xml.parsers.DocumentBuilderFactory factory =
        javax.xml.parsers.DocumentBuilderFactory.newInstance();
    factory.setNamespaceAware(true);
    javax.xml.parsers.DocumentBuilder builder = null;
    try {
        builder = factory.newDocumentBuilder();
    }

```

```

    catch (javax.xml.parsers.ParserConfigurationException ex) {
    }
    org.w3c.dom.Document doc = builder.parse(is);
    is.close();
    return doc;
}

```

Listing 6.17: Neue Methoden in OS.java zur Einbindung der \mathcal{MOISE} -DSL in \mathcal{MOISE}^+

Anhang B.1 zeigt sämtliche Änderungen als Patch. Anhang B.2 enthält zusätzlich die Korrektur eines kleinen Fehlers in der \mathcal{MOISE}^+ -Implementierung, der während der Integrationstests sichtbar wurde: Obwohl laut OML-Spezifikation erlaubt, akzeptierte \mathcal{MOISE}^+ (in der zum Zeitpunkt der Implementierung aktuellen Version 0.8) keine Links oder Compatibilities, deren Attribute für Reichweite (inter-/intragruppen), Richtung (uni-/bidirektional) oder Untergruppengültigkeit (ja/nein) nicht gesetzt waren. Die Spezifikation sieht diesem Fall statt eines Fehlers die Nutzung von Standardwerten vor.

7 Bewertung und Ausblick

7.1 Zusammenfassung und Bewertung

7.1.1 Fachliche Zielerfüllung

\mathcal{M} OISE-DSL ermöglicht es dem MAS-Entwickler bei Nutzung von \mathcal{M} OISE⁺, seine Organisational Specification statt in der für Menschen schlecht les- und formulierbaren, XML-basierten Sprache OML in einer an die natürliche englische Sprache angelehnten DSL zu schreiben.

Die Nutzung von (im Gegensatz zu XML-Dialekten einfachen) domänenspezifischen Sprachen ist zunächst einmal weder neu noch unüblich. Überall dort, wo entweder Nicht-Programmierern das Schreiben, und in erster Linie Lesen, von Programmcode (im Sinne eines *maschinell verarbeitbaren Programmmodells*) ermöglicht werden soll, oder aber dort, wo die Produktivität durch eine reichhaltigere Semantik gesteigert werden soll, kommen sie zum Einsatz.

Vor diesem Hintergrund – und wenn man bedenkt, dass DSLs wie AgentSpeak im MAS-Umfeld seit Jahrzehnten bekannt sind und auch sonst (Agents and Artefacts etc.) viel dafür getan wird, die semantische Lücke zwischen Modell und Implementierung zu verkleinern – muss es förmlich überraschen, dass die Formulierung von Organisational Specifications bis heute XML erforderte.

Das Ziel, hier eine semantisch näher am Modell liegende Implementierung zu finden und so besagte Lücke weiter zu schließen, kann daher in den Augen des Autors als erreicht betrachtet werden.

7.1.2 Qualität der Umsetzung

Implementierung

Auch die technische Qualität der Implementierung kann – trotz des Prototypenstatus – als hoch angesehen werden: Eine vollständige TestSuite auf JUnit-Basis stellt die

Funktionalität sicher und ermöglicht gefahrlose Änderungen, Erweiterungen und – ganz wichtig für guten Code – Refactoring-Maßnahmen. Die Cobertura-basierten Zweigüberdeckungstests stellen sicher, dass keine relevanten Code-Bestandteile ungetestet bleiben.

Gemäß Clean Code-Paradigma sind Methoden kurz (fast nie umfassen sie mehr als vier Ausdrücke) und selbst Traits und Klassen sind im Normalfall auf wenige Zeilen, selten eine Bildschirmseite, beschränkt – hier kommt die Ausdruckstärke der Programmiersprache Scala voll zum Zuge.

Architektur

Aus Architektursicht erscheint die gewählte Lösung ebenfalls angemessen: Die Schnittstelle zu \mathcal{MOISE}^+ nutzt mit OML exakt die in der \mathcal{MOISE}^+ -Architektur vorgesehene Schnittstelle für von außen vorgegebene Organisational Specifications. Auf diese Weise ist maximale Unabhängigkeit der \mathcal{MOISE}^+ - und \mathcal{MOISE} -DSL-Implementierungen gewährleistet. Die technischen Abhängigkeiten entsprechen exakt den inhaltlichen (OML-Sprachfeatures), ergänzen ihrerseits aber keine weiteren. Wenn eine der beiden Seiten ihre interne Implementierung ändert, determiniert dies keine Änderungsnotwendigkeit anderswo.

Aus MAS-Sicht erfolgt die Nutzung von \mathcal{MOISE} -DSL wie in Abbildung 6.1 auf Seite 57 gezeigt, transparent, da \mathcal{MOISE}^+ hiervon abstrahiert.

\mathcal{MOISE} -DSL-intern gewährleistet die Aufteilung in die Pakete `moise.dsl` und `moise.dslconverter` maximale Unabhängigkeit von der für die XML-Konvertierung genutzten scalaxb-Bibliothek einerseits und vom Konvertierungsziel OML andererseits. Sowohl die Verwendung einer anderen XML-Bibliothek als auch die Konvertierung in gänzlich andere Ziele (Programmiersprachen, Abbildungen, Dokumente etc.) lassen sich allein durch den Einsatz neuer Konverter erreichen.

7.2 Weiterentwicklungsmöglichkeiten und -bedarf

An einigen Stellen offenbart sich der Prototypencharakter der \mathcal{MOISE} -DSL. So sind beispielsweise Attribute wichtiger Klassen nicht mit dedizierten Schlüsselwörtern für die Zugriffskontrolle versehen, was in Scala implizit *public*, also öffentlichen Zugriff bedeutet. Dies wäre an sich nicht nötig, da – außer den Klassen selbst – nur die Konverter (und UnitTests) Zugriff auf diese Attribute benötigen. Scala bietet einen sehr feingranularen Zugriffsschutz, der Sichtbarkeit auf bestimmte Ebenen der Paket-Hierarchie einschränken kann. Im Zuge einer Finalisierung des aktuellen Entwicklungsstandes der \mathcal{MOISE} -DSL

wäre hier noch ein wenig Fleißarbeit angebracht, die jedoch keinen Einfluss auf aus der aktuellen Implementierung ableitbare Erkenntnisse bedeutet.

OML bietet für verschiedenen Sprachelemente die Möglichkeit, benutzerdefinierte Eigenschaften, sog. *Properties*, zu definieren. Diese bestehen schlicht aus Attribut-Wert-Paaren, deren Elemente jeweils durch Zeichenketten repräsentiert werden. Da von dieser Möglichkeit in keiner der in der Literatur oder den \mathcal{MOISE}^+ -Beispielprojekten gezeigten Organisational Specifications Gebrauch gemacht wurde, wurde sie für \mathcal{MOISE} -DSL zunächst nicht implementiert. Gleichwohl bieten Scalas Traits die leicht umzusetzende Möglichkeit, durch Definition eines zusätzlichen Traits **WithProperties** einfach und deklarativ durch „Einmischen“ ebendieses Traits zu entscheiden, welche Sprachelemente mit eigenen Eigenschaften erweiterbar sein sollen. Der Grund, warum dies nicht bereits in der vorliegenden Implementierung geschehen ist, besteht darin, dass mangels Beispiele für Eigenschaftennutzung nicht ersichtlich ist, wie und wofür diese i. d. R. eingesetzt werden. Diese Information wäre aber notwendig gewesen, um eine im Sinne des Sprachdesigns semantisch stimmige Formulierung für die Nutzung von Eigenschaften in \mathcal{MOISE} -DSL zu finden. Wie bereits erwähnt, erlaubt Scala jedoch das unkomplizierte Ergänzen dieses Features auch im Nachhinein.

Ebenfalls da genannte Beispiele den Schluss nahe legen, dass diese nicht häufig genutzt werden sowie aus Aufwandsgründen werden Gültigkeitsbedingungen für Normen in \mathcal{MOISE} -DSL genau wie in OML schlicht als Strings formuliert. Dies widerspricht eindeutig den \mathcal{MOISE} -DSL-Designzielen, auf Magic Strings zu verzichten, alles streng typisiert zu gestalten und mit Semantik zu versehen. Wie die Zeichenkette einer solchen Bedingung aufgebaut sein darf, ist in \mathcal{MOISE}^+ bekannt und wird zur Laufzeit überprüft. Daher wäre es für zukünftige Versionen möglich und wünschenswert, auch hierfür eine bereits zur Übersetzungszeit überprüfbare Formulierungsmöglichkeit für \mathcal{MOISE} -DSL zu ergänzen.

\mathcal{MOISE} -DSL-Syntaxerweiterungen zum Erzwingen einer festen Reihenfolge bei der Definition der einzelnen \mathcal{MOISE}^+ -Entitäten wären durch Nutzung mehrerer unterschiedlicher Klassen möglich in der Art derjenigen, die bereits zum Erzwingen der obligatorischen Attribute zum Einsatz kommen. Auch könnten durch den vermehrten Einsatz von Bubble Words die Unterstriche in Mehr-Wort-Bezeichnern vermieden werden, sollte das gewünscht sein.

Der \mathcal{MOISE} -DSL-Interpreter, der zur Laufzeit genutzt wird, um aus den Inhalten einer \mathcal{MOISE} -DSL-Datei einen OML-String zu erzeugen, könnte mithilfe von Caching laufzeitoptimiert werden, da er i. d. R. mehrfach für die gleiche, in der Zwischenzeit

nicht veränderte, Datei aufgerufen wird. Der Grund hierfür kann beispielsweise darin bestehen, dass zunächst ein SchemeBoard und anschließend ein GroupBoard erzeugt werden soll, weshalb zwei mal auf die Organisational Specification zugegriffen werden muss.

Einen im Vergleich zum einfachen Ersetzen der Konverter-Klassen allgemeingültigeren Ansatz zur Flexibilisierung, Erweiterung und Optimierung von internen DSLs, der besonderen Wert auf die Wiederverwendbarkeit unterschiedlicher DSL-Interpretationen legt, schlagen Hofer et al. (2008) mit *Polymorphic Embedding* vor, was die technische Seite der vielfältigen Erweiterungsmöglichkeiten interner DSLs eindrucksvoll demonstriert.

7.3 Multiagentensystementwicklung und -plattformen

Die Entwicklung von Multiagentensystemen und deren Plattformen ging in den letzten Jahren und Jahrzehnten immerzu in Richtung semantisch reichhaltigerer Modelle. Ziel war stets die Gleichheit zwischen Modell und Implementierung herzustellen, also letztendlich von Menschen leicht zu verstehende und gleichzeitig maschinenausführbare Modelle zu ermöglichen. Der Weg dazu ging über AOP selbst und zugehörige DSLs über das A&A-Meta-Modell mittels CArtaGO, Multiagentenorganisationen mit MOISE⁺ bis hin zu der Organisation Management Infrastructure ORA4MAS. MOISE-DSL geht nun den nächsten Schritt in dieser Richtung.

Interessant an MOISE-DSL ist jedoch insbesondere der Implementierungsaspekt. Genauer gesagt die Nutzung der Programmiersprache Scala. Ihr Einsatz zeigt eindrucksvoll, wie auch die bloße Auswahl einer geeigneten Implementierungstechnik – genau wie der Einsatz von DSLs – helfen kann, die Produktivität und die inhaltliche Ausdrucksstärke zu steigern (Dubochet, 2009).

Dies zeigt sich bereits bei einem Blick auf die Listing- und Algorithmenverzeichnisse dieser Arbeit: Die gute Lesbarkeit und prägnante Ausdrucksweise der Programmiersprache Scala ermöglicht es, nahezu alle Sachverhalte direkt in Form ihres Codes darzustellen. Nur bei umfangreicheren Algorithmen ist eine Darstellung in Pseudo-Code sinnvoll, wobei in diesen Fällen einerseits tatsächlich von einzelnen Aspekten der Implementierung abstrahiert wurde und andererseits die verbliebenen Aspekte in Pseudo-Code der tatsächlichen Implementierung in Scala sehr stark ähneln.

Vor diesem Hintergrund erscheint die Nutzung von Scala – oder ggf. auch anderen objekt-funktionalen Hybridsprachen für die Java-Plattform wie *Clojure* – generell als

probates Mittel für zukünftige Entwicklungen im MAS-, bzw. wegen Java speziell *Jason*-, Umfeld.

CARTAgO verbesserte durch die Nutzung vordefinierter Basisklassen, insb. jedoch durch das Ermöglichen eines deklarativen Programmierstils mittels Annotationen, bereits die Modellierung der Agentenumgebungen – wenngleich nach wie vor auf Basis von Java. Beim Planen der nächsten Entwicklungsschritte sollte nach Meinung des Autors auch hier mehr als nur ein Gedanke in die Prüfung alternativer Programmiersprachen wie der genannten investiert werden.

Literatur

- [Beck 2003] BECK, Kent: *Test-driven development : by example*. 1. Aufl. Boston, MA, USA : Addison-Wesley, 2003. – ISBN 9780321146533
- [Behrens et al. 2010] BEHRENS, Tristan ; DASTANI, Mehdi ; DIX, Jürgen ; KÖSTER, Michael ; NOVÁK, Peter: The multi-agent programming contest from 2005-2010. In: *Annals of Mathematics and Artificial Intelligence* 59 (2010), S. 277–311. <http://dx.doi.org/10.1007/s10472-010-9219-5>. – DOI 10.1007/s10472-010-9219-5. – ISSN 1012-2443
- [Bordini und Hübner 2006] BORDINI, Rafael ; HÜBNER, Jomi: BDI Agent Programming in AgentSpeak Using Jason. Version:2006. http://dx.doi.org/10.1007/11750734_9. In: TONI, Francesca (Hrsg.) ; TORRONI, Paolo (Hrsg.): *Computational Logic in Multi-Agent Systems* Bd. 3900. Springer Berlin / Heidelberg, 2006. – DOI 10.1007/11750734_9, S. 143–164
- [Bordini et al. 2002] BORDINI, Rafael H. ; BAZZAN, Ana L. C. ; O. JANNONE, Rafael de ; BASSO, Daniel M. ; VICARI, Rosa M. ; LESSER, Victor R.: AgentSpeak(XL): efficient intention selection in BDI agents via decision-theoretic task scheduling. In: *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 3*. New York, NY, USA : ACM, 2002 (AAMAS '02). – ISBN 1581134800, S. 1294–1302
- [Bordini et al. 2007] BORDINI, Rafael H. ; WOOLDRIDGE, Michael ; HÜBNER, Jomi F.: *Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology)*. 1. Aufl. Chichester, UK : John Wiley & Sons, 2007. – ISBN 0470029005
- [Dastani et al. 2003] DASTANI, Mehdi ; BOER, Frank de ; DIGNUM, Frank ; MEYER, John jules: Programming Agent Deliberation: An Approach Illustrated Using the 3APL Language. In: *In Proceedings of The Second Conference on Autonomous Agents and Multi-agent Systems (AAMAS'03)*, ACM Press, 2003, S. 97–104

- [Dubochet 2006] DUBOCHET, Gilles: On Embedding Domain-specific Languages with User-friendly Syntax. In: *Proceedings of the 1st Workshop on Domain-Specific Program Development*, 2006, S. 19–22
- [Dubochet 2009] DUBOCHET, Gilles: Computer Code as a Medium for Human Communication: Are Programming Languages Improving? In: EXTON, Chris (Hrsg.) ; BUCKLEY, Jim (Hrsg.): *Proceedings of the 21st Working Conference on the Psychology of Programmers Interest Group*. Limerick, Ireland : University of Limerick, 2009. – ISBN 9781905952168, S. 174–187
- [Fowler 2009] FOWLER, Martin: A Pedagogical Framework for Domain-Specific Languages. In: *Software, IEEE* 26 (2009), Jul.-Aug., Nr. 4, S. 13–14. <http://dx.doi.org/10.1109/MS.2009.85>. – DOI 10.1109/MS.2009.85. – ISSN 0740–7459
- [Fowler 2010] FOWLER, Martin: *Domain Specific Languages*. 1. Aufl. Boston, MA, USA : Addison-Wesley Professional, 2010. – ISBN 0321712943, 9780321712943
- [Ghosh 2011] GHOSH, Debasish: DSL for the Uninitiated. In: *Queue* 9 (2011), Jun., S. 10:10–10:21. <http://dx.doi.org/10.1145/1989748.1989750>. – DOI 10.1145/1989748.1989750. – ISSN 1542–7730
- [Hannoun et al. 2000] HANNOUN, Mahdi ; BOISSIER, Olivier ; SICHMAN, Jaime ; SAYETTAT, Claudette: MOISE: An Organizational Model for Multi-agent Systems. Version: 2000. http://dx.doi.org/10.1007/3-540-44399-1_17. In: MONARD, Maria (Hrsg.) ; SICHMAN, Jaime (Hrsg.): *Advances in Artificial Intelligence* Bd. 1952. Springer Berlin / Heidelberg, 2000. – DOI 10.1007/3-540-44399-1_17, S. 156–165
- [Hen-Tov et al. 2009] HEN-TOV, Atzmon ; LORENZ, David H. ; SCHACHTER, Lior: An interpretive domain specific language workbench. In: *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. New York, NY, USA : ACM, 2009 (OOPSLA '09). – ISBN 9781605587684, 751-752
- [Hofer et al. 2008] HOFER, Christian ; OSTERMANN, Klaus ; RENDEL, Tillmann ; MOORS, Adriaan: Polymorphic embedding of DSLs. In: *Proceedings of the 7th international conference on Generative programming and component engineering*. New York, NY, USA : ACM, 2008 (GPCE '08). – ISBN 9781605582672, S. 137–148

- [Hudak 1996] HUDAK, Paul: Building domain-specific embedded languages. In: *ACM Comput. Surv.* 28 (1996), Dec. <http://dx.doi.org/10.1145/242224.242477>. – DOI 10.1145/242224.242477. – ISSN 0360–0300
- [Huhns und Stephens 2000] *Kapitel Multiagent systems and societies of agents.* In: HUHNS, Michael N. ; STEPHENS, Larry M.: *Multiagent systems: a modern approach to distributed artificial intelligence.* 2. Aufl. Cambridge, MA, USA : MIT Press, 2000. – ISBN 9780262731317, S. 79–120
- [Hübner et al. 2010a] HÜBNER, Jomi ; BOISSIER, Olivier ; BORDINI, Rafael: From Organisation Specification to Normative Programming in Multi-Agent Organisations. Version:2010. http://dx.doi.org/10.1007/978-3-642-14977-1_11. In: DIX, Jürgen (Hrsg.) ; LEITE, João (Hrsg.) ; GOVERNATORI, Guido (Hrsg.) ; JAMROGA, Wojtek (Hrsg.): *Computational Logic in Multi-Agent Systems* Bd. 6245. Springer Berlin / Heidelberg, 2010. – DOI 10.1007/978-3-642-14977-1_11, S. 117–134
- [Hübner et al. 2010b] HÜBNER, Jomi ; BOISSIER, Olivier ; BORDINI, Rafael: A Normative Organisation Programming Language for Organisation Management Infrastructures. Version:2010. http://dx.doi.org/10.1007/978-3-642-14962-7_8. In: PADGET, Julian (Hrsg.) ; ARTIKIS, Alexander (Hrsg.) ; VASCONCELOS, Wamberto (Hrsg.) ; STATHIS, Kostas (Hrsg.) ; SILVA, Viviane da (Hrsg.) ; MATSON, Eric (Hrsg.) ; POLLERES, Axel (Hrsg.): *Coordination, Organizations, Institutions and Norms in Agent Systems V* Bd. 6069. Springer Berlin / Heidelberg, 2010. – DOI 10.1007/978-3-642-14962-7_8, S. 114–129
- [Hübner et al. 2010c] HÜBNER, Jomi ; BOISSIER, Olivier ; KITIO, Rosine ; RICCI, Alessandro: Instrumenting multi-agent organisations with organisational artifacts and agents. In: *Autonomous Agents and Multi-Agent Systems* 20 (2010), S. 369–400. <http://dx.doi.org/10.1007/s10458-009-9084-y>. – DOI 10.1007/s10458-009-9084-y. – ISSN 1387–2532
- [Hübner et al. 2002a] HÜBNER, Jomi ; SICHMAN, Jaime ; BOISSIER, Olivier: A Model for the Structural, Functional, and Deontic Specification of Organizations in Multiagent Systems. Version:2002. http://dx.doi.org/10.1007/3-540-36127-8_12. In: BITTENCOURT, Guilherme (Hrsg.) ; RAMALHO, Geber (Hrsg.): *Advances in Artificial Intelligence* Bd. 2507. Springer Berlin / Heidelberg, 2002. – DOI 10.1007/3-540-36127-8_12, S. 439–448

- [Hübner et al. 2007] HÜBNER, Jomi F. ; SICHMAN, Jaime S. ; BOISSIER, Olivier: Developing organised multiagent systems using the MOISE+ model: programming issues at the system and agent levels. In: *International Journal of Agent-Oriented Software Engineering (IJAOSE)* 1 (2007), Nr. 3/4, S. 370–395. <http://dx.doi.org/10.1504/IJAOSE.2007.016266>. – DOI 10.1504/IJAOSE.2007.016266
- [Hübner et al. 2009] HÜBNER, Jomi F. ; BORDINI, Rafael H. ; GOUVEIA, G. P. ; PEREIRA, Ricardo H. ; PICARD, Gauthier ; PIUNTI, Michele ; SICHMAN, Jaime S.: Using Jason, MOISE, and CArtAgO to develop a team of cowboys. In: DIX, Jürgen (Hrsg.) ; FISHER, Michael (Hrsg.) ; NOVAK, Peter (Hrsg.): *Proceedings of 10th International Workshop on Computational Logic in Multi-Agent Systems (CLIMA 2009)*, 2009, S. 203–207
- [Hübner et al. 2002b] HÜBNER, Jomi F. ; SICHMAN, Jaime S. ; BOISSIER, Olivier: MOISE+: towards a structural, functional, and deontic model for MAS organization. In: *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1*. New York, NY, USA : ACM, 2002 (AAMAS '02). – ISBN 1581134800, S. 501–502
- [Hübner et al. 2006] HÜBNER, Jomi F. ; SICHMAN, Jaime S. ; BOISSIER, Olivier: S-moise+: A middleware for developing organised multi-agent systems. In: *COIN I, volume 3913 of LNAI*, Springer, 2006, S. 64–78
- [ISO und IEC 1996] ISO, International Organization for S. ; IEC, International Electrotechnical C.: *Extended Backus–Naur Form: ISO/IEC 14977: 1996 (E)*. 1996. – Standard
- [Kitio et al. 2008] KITIO, Rosine ; BOISSIER, Olivier ; HÜBNER, Jomi ; RICCI, Alessandro: Organisational Artifacts and Agents for Open Multi-Agent Organisations: "Giving the Power Back to the Agents". Version: 2008. http://dx.doi.org/10.1007/978-3-540-79003-7_13. In: SICHMAN, Jaime (Hrsg.) ; PADGET, Julian (Hrsg.) ; OSSOWSKI, Sascha (Hrsg.) ; NORIEGA, Pablo (Hrsg.): *Coordination, Organizations, Institutions, and Norms in Agent Systems III* Bd. 4870. Springer Berlin / Heidelberg, 2008. – DOI 10.1007/978-3-540-79003-7_13, S. 171–186
- [Martin 2009] MARTIN, Robert C.: *Clean Code: A handbook of agile software craftsmanship*. 1. Aufl. Boston, MA, USA : Prentice Hall, 2009. – ISBN 0132350882

- [Mernik et al. 2005] MERNIK, Marjan ; HEERING, Jan ; SLOANE, Anthony M.: When and how to develop domain-specific languages. In: *ACM Comput. Surv.* 37 (2005), Dec., S. 316–344. <http://dx.doi.org/10.1145/1118890.1118892>. – DOI 10.1145/1118890.1118892. – ISSN 0360–0300
- [Moors et al. 2008] MOORS, Adriaan ; PIESENS, Frank ; ODERSKY, Martin: Parser combinators in Scala / Department of Computer Science, K.U.Leuven. Leuven, BE, Feb. 2008 (CW491). – Forschungsbericht
- [Odersky 2006] ODERSKY, Martin: The Scala Experiment – Can We Provide Better Language Support for Component Systems? In: *Proc. ACM Symposium on Principles of Programming Languages*, 2006, S. 166–167
- [Odersky et al. 2006] ODERSKY, Martin ; ALTHERR, Philippe ; CREMET, Vincent ; DRAGOS, Iulian ; DUBOCHET, Gilles ; EMIR, Burak ; MCDIRMIID, Sean ; MICHELOUD, Stéphane ; MIHAYLOV, Nikolay ; SCHINZ, Michel ; SPOON, Lex ; STENMAN, Erik ; ZENGER, Matthias: An Overview of the Scala Programming Language. 2. Aufl. / École Polytechnique Fédérale de Lausanne (EPFL). Lausanne, CH, 2006. – Forschungsbericht
- [Odersky et al. 2011] ODERSKY, Martin ; SPOON, Lex ; VENNERS, Bill: *Programming in Scala*. 2. Aufl. Mountain View, CA, USA : Artima Press, 2011. – ISBN 9780981531649
- [Odersky und Zenger 2005] ODERSKY, Martin ; ZENGER, Matthias: Scalable component abstractions. In: *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA : ACM, 2005 (OOPSLA '05). – ISBN 1595930310, S. 41–57
- [Omicini und Zambonelli 1999] OMICINI, Andrea ; ZAMBONELLI, Franco: Coordination for Internet Application Development. In: *Autonomous Agents and Multi-Agent Systems* 2 (1999), S. 251–269. <http://dx.doi.org/10.1023/A:1010060322135>. – DOI 10.1023/A:1010060322135. – ISSN 1387–2532
- [Piunti et al. 2010] PIUNTI, Michele ; BOISSIER, Olivier ; HUBNER, Jomi F. ; RICCI, Alessandro: Embodied Organizations: a unifying perspective in programming Agents, Organizations and Environments. In: NICOLETTA FORNARA, George V. (Hrsg.): *11th International Workshop on Coordination Organization Institutions and Norms in Agent Systems Coordination Organization Institutions and Norms in Agent Systems*, Springer, 2010, S. 98–114


- [Rao 1996] RAO, Anand: AgentSpeak(L): BDI agents speak out in a logical computable language. Version: 1996. <http://dx.doi.org/10.1007/BFb0031845>. In: VELDE, Walter Van d. (Hrsg.) ; PERRAM, John (Hrsg.): *Agents Breaking Away* Bd. 1038. Springer Berlin / Heidelberg, 1996. – DOI 10.1007/BFb0031845, S. 42–55
- [Rao und Georgeff 1991] RAO, Anand S. ; GEORGEFF, Michael P.: Modeling Rational Agents within a BDI-Architecture. In: ALLEN, J. (Hrsg.) ; FIKES, R. (Hrsg.) ; SANDEWALL, E. (Hrsg.): *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, 1991, S. 473–484
- [Reenskaug 1979] REENSKAUG, T.: Models-views-controllers / Xerox PARC. Palo Alto, CA, USA, 1979. – Forschungsbericht
- [Ricci et al. 2006] RICCI, Alessandro ; VIROLI, Mirko ; OMICINI, Andrea: Programming MAS with Artifacts. Version: 2006. http://dx.doi.org/10.1007/11678823_13. In: BORDINI, Rafael (Hrsg.) ; DASTANI, Mehdi (Hrsg.) ; DIX, Jürgen (Hrsg.) ; EL FALLAH SEGHROUCHNI, Amal (Hrsg.): *Programming Multi-Agent Systems* Bd. 3862. Springer Berlin / Heidelberg, 2006. – DOI 10.1007/11678823_13, S. 206–221
- [Ricci et al. 2007] RICCI, Alessandro ; VIROLI, Mirko ; OMICINI, Andrea: CArtaGO: A Framework for Prototyping Artifact-Based Environments in MAS. Version: 2007. http://dx.doi.org/10.1007/978-3-540-71103-2_4. In: WEYNS, Danny (Hrsg.) ; PARUNAK, H. (Hrsg.) ; MICHEL, Fabien (Hrsg.): *Environments for Multi-Agent Systems III* Bd. 4389. Springer Berlin / Heidelberg, 2007. – DOI 10.1007/978-3-540-71103-2_4, S. 67–86
- [Ricci et al. 2008] RICCI, Alessandro ; VIROLI, Mirko ; OMICINI, Andrea: The A&A Programming Model and Technology for Developing Agent Environments in MAS. Version: 2008. http://dx.doi.org/10.1007/978-3-540-79043-3_6. In: DASTANI, Mehdi (Hrsg.) ; EL FALLAH SEGHROUCHNI, Amal (Hrsg.) ; RICCI, Alessandro (Hrsg.) ; WINIKOFF, Michael (Hrsg.): *Programming Multi-Agent Systems* Bd. 4908. Springer Berlin / Heidelberg, 2008. – DOI 10.1007/978-3-540-79043-3_6, S. 89–106
- [Rytz und Odersky 2010] RYTZ, Lukas ; ODERSKY, Martin: Named and default arguments for polymorphic object-oriented languages: a discussion on the design implemented in the Scala language. In: *Proceedings of the 2010 ACM Symposium on Applied Computing*. New York, NY, USA : ACM, 2010 (SAC '10). – ISBN 9781605586397, S. 2090–2095

- [Shoham 1993] SHOHAM, Yoav: Agent-oriented programming. In: *Artificial Intelligence* 60 (1993), Nr. 1, S. 51–92. [http://dx.doi.org/10.1016/0004-3702\(93\)90034-9](http://dx.doi.org/10.1016/0004-3702(93)90034-9). – DOI 10.1016/0004–3702(93)90034–9. – ISSN 0004–3702
- [Wampler und Payne 2009] *Kapitel* Domain-Specific Languages in Scala. In: WAMPLER, D. ; PAYNE, A.: *Programming Scala: Scalability = Functional Programming + Objects*. 1. Aufl. Sebastopol, CA, USA : O’Reilly Media, 2009 (O’Reilly Series). – ISBN 9780596155957, S. 217–246
- [Wooldridge 2009] *Kapitel* Intelligent Agents. In: WOOLDRIDGE, Michael: *An Introduction to Multiagent Systems*. 2. Aufl. Chichester, UK : Wiley, 2009. – ISBN 9780470519462, S. 21–47
- [Wooldridge und Jennings 1995] WOOLDRIDGE, Michael ; JENNINGS, Nicholas R.: Intelligent Agents: Theory and Practice. In: *Knowledge Engineering Review* 10 (1995), Nr. 2, S. 115–152

A Anhang OML-Sprachspezifikation

A.1 XML Schema Definition

Die Datei `os.xsd` enthält die XML Schema Definition für die Organisation Modelling Language:

- `os.xsd` 

Sie wurde der \mathcal{MOISE}^+ -Distribution entnommen.

A.2 Dokumentation

Das Archiv `os-xml-doc.zip` enthält Abbildungen und HTML-Dokumente zur Beschreibung des XML Schemas in Anhang A.1:

- `os-xml-doc.zip`

Genau wie dieses ist es der \mathcal{MOISE}^+ -Distribution entnommen.

B Anhang Anpassungen an $\mathcal{M}\text{oise}^+$

B.1 Erweiterungen

Die Datei `OS.java.patch` zeigt die Erweiterungen an der Datei `OS.java` der $\mathcal{M}\text{oise}^+$ -Implementierung (auf Basis von Version 0.8), die für die Einbindung der $\mathcal{M}\text{oise}$ -DSL erforderlich waren:

- `OS.java.patch`

Ferner sind das Referenzieren des `jar`-Archivs `moise-dsl.jar` für $\mathcal{M}\text{oise}$ -DSL zur Übersetzungszeit sowie zur Laufzeit der Zugriff auf die per `scalaxb` generierten Xb-Klassen im Archiv `moise-scalaxb.jar` und die Archive `scala-library.jar` und `scala-compiler.jar` erforderlich.

B.2 Korrekturen

Die Datei `RoleRel.java.patch` zeigt die Korrekturen an der Datei `RoleRel.java` der $\mathcal{M}\text{oise}^+$ -Implementierung (auf Basis von Version 0.8), die für die Einbindung der $\mathcal{M}\text{oise}$ -DSL erforderlich sind:

- `RoleRel.java.patch`

C Anhang Beispiel „House Building“

C.1 Beschreibung

In der unter <http://jacamo.sourceforge.net> zum Download zur Verfügung stehenden Kombination aus *Jason*, *CARTAgO* und *MOISE⁺* namens *JaCaMo* dient ein Szenario, in dem ein Agent namens *giacomo* ein Haus bauen lassen möchte, als Beispiel, um die Integration der drei Bestandteile zu verdeutlichen:

Jason dient als grundlegendes Multiagentensystem, das Agentenverhalten ist Agent-Speak (und teilweise 2APL) programmiert. Mittels *CARTAgO* werden Artefakte für Auktionen, die der Vergabe von Aufträgen im Rahmen des Baus dienen, sowie zur Koordination (ORA4MAS) erzeugt und genutzt. *MOISE⁺* dient in diesem Szenario der abstrakten Beschreibung einer virtuellen Organisation, die sich aus Auftraggeber und Auftragnehmern des Bauvorhabens zusammensetzt.

C.2 OML-Formulierung

Die Datei `house-os.xml` formuliert die Organisational Specification der in Anhang C.1 genannten virtuellen Organisation in OML:

- `house-os.xml`

Sie wurde der JaCaMo-Distribution entnommen.

C.3 Moise-DSL-Formulierung

Die Datei `house-os.scala` formuliert die Organisational Specification der in Anhang C.1 genannten virtuellen Organisation in *MOISE-DSL*:

- `house-os.scala`

D Anhang *Moise*-DSL-Quelltexte

D.1 Projekt

Die Datei `moise-dsl.zip` beinhaltet das NetBeans-Projekt der *MOISE*-DSL-Implementierung:

- `moise-dsl.zip`

D.2 Xb-Klassen in `moise-scalaxb.jar`

Die Dateien `moise-scalaxb.zip` und `moise-scalaxb.jar` beinhalten das NetBeans-Projekt mit dem mittels `scalaxb` generierten Code zum Schreiben von OML-Dateien sowie das entsprechende `jar`-Archiv als Ergebnis:

- `moise-scalaxb.zip`
- `moise-scalaxb.jar`

D.3 API-Dokumentation

Das Archiv `scaladoc.zip` beinhaltet die HTML-Dokumentation der *MOISE*-DSL-API:

- `scaladoc.zip`

D.4 Test-Ergebnisse

D.4.1 UnitTests

Das Archiv `junit-report.zip` beinhaltet das HTML-Ergebnisprotokoll eines vollständigen Test-Laufs der *MOISE*-DSL-Implementierung:

- `junit-report.zip`

D.4.2 Testabdeckung

Das Archiv `cobertura-report.zip` beinhaltet das HTML-Testabdeckungsprotokoll eines vollständigen Test-Laufs der *MOISE*-DSL-Implementierung:

- `cobertura-report.zip`

E Anhang lauffähiges Multiagentensystem

Die Datei `JaMaCo_House_Building.zip` enthält ein unter *Jason* lauffähiges komplettes Beispiel-Projekt auf Basis des in Anhang C.1 beschriebenen Szenarios aus der JaCaMo-Distribution:

- `JaMaCo_House_Building.zip`

Um es auszuführen, muss lediglich die *Jason*-Projektdatei `House_Building.mas2j` gestartet werden.

Gegenüber der originalen JaCaMo-Version ohne *MOISE*-DSL kommt diese mit nur minimalen Änderungen aus:

1. Die Dateien `moise-dsl.jar`, `moise-scalaxb.jar`, `scala-library.jar` und `scala-compiler.jar` sowie die die in Anhang B beschriebenen Anpassungen beinhaltende Datei `moise.jar` wurden in das Unterverzeichnis `libs` des Projektes kopiert.
2. Die neuen Dateien (alle bis auf `moise.jar`) wurden in der *Jason*-Projektdatei `House_Building.mas2j` im Abschnitt `classpath` ergänzt.
3. Die *MOISE*-DSL-Datei `house-os.scala` (vgl. Anhang C.3) wurde in das Unterverzeichnis `src` des Projektes kopiert.
4. Der die OS nutzende Agent `giacomo` (Datei `giacomo.asl` im Unterverzeichnis `src/asl`) übergibt an `CARTAgO` beim Erstellen von Artefakten (`GroupBoard` und `SchemeBoard`) nun den Parameter `"src/house-os.scala"` anstelle des alten Parameters `"src/house-os.xml"`.

F Anhang Verwendete Hilfsmittel

Programm	Version	Zweck
Texmaker	2.2.1	L ^A T _E X-Editor zum Erstellen der schriftlichen Ausarbeitung
pdfLaTeX	3.1415926-1.40.10	Erweiterung des Textsatzprogramms T _E X, um direkt eine PDF-Datei als Ausgabe zu erzeugen
NetBeans	6.9	Integrierte Entwicklungsumgebung für die Java-Plattform; hier in Verbindung mit einem Plugin zur Scala-Unterstützung für die Implementierung der <i>MOISE</i> -DSL genutzt
Scala	2.8.1	Entwicklungsplattform für die Programmiersprache Scala, bestehend aus Compiler und Klassenbibliothek; ebenfalls für die Implementierung der <i>MOISE</i> -DSL genutzt
scalaxb	0.5.4	XML-Datenbindungswerkzeug inkl. Code-Generator für Scala; genutzt zum Erstellen der Xb-Klassen und deren Konvertierung in OML
JUnit	4.8.2-2	UnitTesting-Framework für die Java-Plattform; genutzt zur Automatisierung der <i>MOISE</i> -DSL-Tests
Cobertura	1.9.3	Zweigüberdeckungsanalysewerkzeug für die Java-Plattform; genutzt zur Ermittlung der Testabdeckung der <i>MOISE</i> -DSL-Tests
Dia	0.97.1	Anwendung zum Zeichnen von strukturierten Diagrammen; genutzt zum Erstellen eigener Abbildungen im Rahmen der schriftlichen Ausarbeitung

Eidesstattliche Versicherung

Ich versichere an Eides statt durch meine Unterschrift, dass ich die vorliegende Projektarbeit „MOISE-DSL: Eine domänenspezifische Sprache zur Modellierung von Multi-agentenorganisationen“ selbständig und ohne fremde Hilfe angefertigt und alle Stellen, die ich wörtlich oder annähernd wörtlich aus Veröffentlichungen entnommen habe, als solche kenntlich gemacht habe, mich auch keiner anderen als der angegebenen Literatur oder sonstiger Hilfsmittel bedient habe. Die Arbeit hat in dieser oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Ort, Datum

Unterschrift