



Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

Daniel Glake

**MARS DSL: Eine typisierte Sprache zur Modellierung
komplexer agentenbasierter Modelle**

*Fakultät Technik und Informatik
Studiendepartment Informatik*

*Faculty of Engineering and Computer Science
Department of Computer Science*

Daniel Glake

**MARS DSL: Eine typisierte Sprache zur Modellierung
komplexer agentenbasierter Modelle**

Masterarbeit eingereicht im Rahmen der Masterprüfung

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

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

Eingereicht am: 08. Mai 2018

Daniel Glake

Thema der Arbeit

MARS DSL: Eine typisierte Sprache zur Modellierung komplexer agentenbasierter Modelle

Stichworte

DSL-Entwicklung, Multiagentensystem, Model-to-Text Transformation, Modellierung, Simulation

Kurzzusammenfassung

Mit wachsender Akzeptanz von Multiagentensystemen zur Behandlung komplexer Problem- und Fragestellungen liegt der Fokus auf die Entwicklung und der Analyse erforderlicher Werkzeuge dieser Form. Stand der Forschung ist, das konzeptionelle Agentenmodell mittels einer Agentenmethodologie zu spezifizieren und resultierenden die Artefakte als Grundlage zur manuellen Implementierung zu verwenden. Die in dieser Arbeit erörterte plattformunabhängige MARS DSL ermöglicht die Entwicklung eigener Agentenmodelle nach dem MARS Modellierungsansatz. Sie umfasst eine abstrakte Syntax, als Spezifikation des Vokbulars mit verschiedenen Invarianten und einem Modelltransformator der dies in eine ausführbare Implementierung überführt, um dadurch Fehler und Abweichungen bei einer manuellen Überführung zu vermeiden.

Daniel Glake

Title of the paper

MARS DSL: A typed language for the modeling of complex agent-based models

Keywords

DSL-Engineering, Multi-Agent-System, Model-To-Text Transformation, Modelling, Simulation

Abstract

With increasing acceptance of multi-agent systems to handle own research questions, there is a focus on the development and analysis of required tools this form. Current research is to specify the conceptual agent model using an agent methodology and to use resulting artifacts as a basis for manual implementation. The platform-independent MARS DSL discussed in this thesis enables the development of own agent models according to the MARS modeling approach. It includes an abstract syntax, as a specification of the vocabulary with different invariants and a model transformer that transforms this into a runnable implementation, thereby avoiding errors and deviations in a manual transformation.

Inhaltsverzeichnis

1. Einleitung	1
1.1. Problemstellung	1
1.2. Aufbau der Arbeit	3
2. Verwandte Arbeiten	4
2.1. Modellierungssprachen	4
2.2. Einordnung	5
3. Grundlagen	7
3.1. Agentenorientierte Modellierung	7
3.1.1. Grundstruktur	8
3.1.2. Agentenstrukturen	9
3.1.3. Organisationsstrukturen	10
3.1.4. Interaktionsstrukturen	11
3.1.5. Umgebungsstrukturen	11
3.1.6. Modellierung	12
3.2. Modellgetriebene Entwicklung	13
3.2.1. Modellgetriebene Architektur	15
3.2.2. Modelltransformation	17
4. Analyse	20
4.1. Sichten auf ein Multiagentensystem	20
4.2. Konzeptmodell der MARS DSL	21
4.3. Abstrakte Syntax und Sprachmerkmale	23
4.3.1. Multiagentensystem Sicht	25
4.3.2. Entitätssicht	27
4.3.3. Interaktionssicht	35
4.3.4. Verhaltenssicht	38
4.3.5. Workflowsicht	41
4.4. Zielsystemerzeugung nach MARS	50
4.4.1. Erzeugungsmodell	50
4.4.2. Mapping - Regeln	52
4.4.3. Templater - Konzepte	58
4.5. Validierung und Invarianten	60
4.5.1. Formale Invarianten	60
4.5.2. Informelle Invarianten und Prüfungen	69

4.6.	Analyse von Modelleigenschaften	69
4.6.1.	Agenten Eigenschaften	71
4.6.2.	Layer Eigenschaften	74
5.	Entwurf	77
5.1.	Generierungsworkflow	79
5.2.	Kontextsicht	81
5.3.	Bausteinsicht	82
5.3.1.	Bausteinsicht - Level 0	82
5.3.2.	Bausteinsicht - Sprache Level 1	82
5.3.3.	Bausteinsicht - Übersetzer Level 1	84
5.3.4.	Bausteinsicht - Übersetzer Level 2	86
5.4.	Generatorsicht	86
5.5.	Laufzeitsicht	90
5.5.1.	Laufzeitsicht - Validierung und Anreicherung	91
5.5.2.	Laufzeitsicht - Codegenerierung	92
5.5.3.	Laufzeitsicht - Reaction Ladeprozess	95
5.5.4.	Laufzeitsicht - Regelauswertung	96
5.6.	Transformationssicht	97
5.6.1.	Syntaktische Korrektheit	100
5.6.2.	Inkrementelle Erzeugung	100
6.	Evaluation	101
6.1.	Evaluationsrahmen	101
6.1.1.	Dimensionen	102
6.1.2.	Quantitative Analyse	103
6.2.	Qualitativer Workshop	104
6.2.1.	Zusammenfassung	108
6.3.	Fangmodell - Fangen und Fischen	109
6.3.1.	Motivation	109
6.3.2.	Verwandte Arbeiten	114
6.3.3.	Kernmodell	114
6.3.4.	Agentenmodell	117
6.3.5.	Layermodell	128
6.3.6.	Umgebungsmodell	130
6.3.7.	Experimente	134
6.3.8.	Zusammenfassung und Ausblick	147
6.4.	Vergleich direktes und transformiertes Modell	148
6.4.1.	Performance Vergleich	148
6.4.2.	Dynamik Vergleich	149
6.4.3.	Quantifizierter Vergleich	150
7.	Zusammenfassung und Ausblick	156

Anhang	159
A. Musterkatalog	160
A.1. Einkanal unidirektionale Interaktion	160
A.2. Einkanal bidirektionale Interaktion	160
A.3. Mehrkanal unidirektionale Nachricht	160
A.4. Mehrkanal bidirektionale Nachricht	162
A.5. Agentenevolution	162
A.6. Agentenkommando	162
A.7. Routing von Nachrichten	164
B. Modellierungsumgebung	166
B.1. Exemplarischer MARS Editor für das Modellieren in der Cloud	166
B.2. Beispiel der Integration von MathML Dokumentation	167
B.3. Beispiel einer Anwendung des integrierten Zeitreihen-Layers	168
B.4. Beispiel des Sender Empfänger Modells	169
B.5. Beispiel des angepassten Übersichtsfensters für die Modellstruktur	170
B.6. Beispiel Konstruktion eines geospatialer Fangmodell-Plots mit R	171
C. Workshop Umfrage	173
D. Danksagung	174

Tabellenverzeichnis

3.1. Unterschiedliche ABMS Definitionen und ihre Eigenschaften nach Macal (2016)	13
4.1. Essenzielle Anforderungen an die Texterzeugung	59
4.2. Ergänzte informelle Regeln zur Prüfung statischer Semantik.	70
4.3. Validierungsregeln zur Sicherstellung von konformen MARS Modellen.	76
6.1. Quantifizierung der MARS DSL	103
6.2. Bewertung der Notation und Konzepte	105
6.3. Bewertung zur Modellierungsmethodik	105
6.4. Bewertung zur Validierung und Ausführung von Modellen	106
6.5. Überschlagene Einarbeitungszeit zu MARS und der MARS DSL	106
6.6. Überschlagene Modellierungszeit mit der MARS DSL inklusive Experimente	106
6.7. Anteilige Verwendung der MARS DSL Beschreibungskonzepte	107
6.8. Übersicht verwendeter Daten und deren Quellen	110
6.9. Vergleich erzeugter Elemente des Räuber-Beute Modells gegenüber der direkten Implementierung	152
6.10. Vergleich erzeugter Elemente des Fangmodells gegenüber der direkten Implementierung	152
6.11. Vergleich der Komplexität und Kopplung von erzeugten Elementen des Räuber-Beute Modells gegenüber der direkten Implementierung sowie dem Fangmodell	153
6.12. Vergleich der Abstraktion und Instabilität mit normalisierter Abweichung von erzeugten Elementen des Räuber-Beute Modells gegenüber der direkten Implementierung sowie dem Fangmodell	154

Abbildungsverzeichnis

3.1.	Kernbausteine eines Multiagentensystems nach Jennings (2001)	8
3.2.	Metamodell der MDA nach OMG (2014)	16
4.1.	Beschreibungsmodell der Sprache, angereichert um das Klassenkonzept, eines Ressourcenmanagement einzelner Modellteile und der Möglichkeit zur Handhabung spatialer Daten	24
4.2.	Rekursivität von Agenten mit Einteilung in mehrere Ebenen und beschränkter Interaktion.	30
4.3.	Modell des Zielsystems mit Abgrenzung zu den Komponenten aus LIFE.	51
5.1.	Fachliche Außensicht auf das MDA nach Glake u. a. (2017).	77
5.2.	Technische Außensicht auf die Übersetzungselemente und Reihenfolge der Verarbeitung bis zur Ausführung von MARS DSL Ressourcen.	78
5.3.	Ergänzter Generierungsworkflow zur strukturierten und kontrollierten Transformation.	80
5.4.	Kontextsicht und Abgrenzung der DSL und des Codegenerators zur MARS Plattform und dem Modellierer. Übernommen aus (Glake (2016), geändert)	81
5.5.	Level 0 der Bausteinsicht der MARS DSL Sprachkomponente und Model-to-Code Transformator sowie Zugriffsmöglichkeiten	82
5.6.	Level 1 der Bausteinsicht des MARS DSL Sprachkerns.	84
5.7.	Level 1 der Bausteinsicht des Model-to-Code Transformators.	85
5.8.	Level 2 der Bausteinsicht der zentralen Workflow Komponente.	87
5.9.	Generatorsicht der MARS Modell- und Projektstruktur, mit verschiedenen Artefakten die durch den Codegenerator erzeugt werden.	88
5.10.	Beispiel einer Abhängigkeitsentwicklung durch Integration der Datenlayer	89
5.11.	Laufzeitsicht der Validierung und Anreicherung des Eingabemodells.	91
5.12.	Laufzeitsicht der Codegenerierung für Agenten und Layer	93
5.13.	Laufzeitsicht des Reaction Lade- und Übersetzungsvorgangs zum Simulationsstart	96
5.14.	Laufzeitsicht der angestoßenen Regelauswertung, definierter Reactions	97
5.15.	Beispiel AST eines Agenten Reflex Körpers	98
6.1.	Zusammenfassung der Umfrageergebnisse zur MARS DSL Zufriedenheit	108
6.2.	Abstraktes Wirkungsdiagramm mit Beziehungen und den Einflüssen zwischen den Zustandsgrößen.	111
6.3.	Konkretes Wirkungsdiagramm mit verfeinerten Beziehungen und Interaktionen zwischen den Zustandsgrößen.	113
6.4.	Konzeptionelles Fangmodell mit beteiligten Entitäten und Interaktionen	116

6.5.	Beispielhafte Darstellung einer diskreten Umgebung mit Fischern (schwarz), Heimathäfen (rot), Heringen (weiß) und einem Laichgebiet (gelb)	131
6.6.	Ergebnisplot der Simulation ohne dynamischen Nährstoffgehalt. Parametrisierung: $d_H = 0.15$, $a_F = 0.56$ und $d_F = 3$	135
6.7.	Ergebnis-Plots der Simulation mit dynamischem Nährstoffgehalt. Parametrisierung mit: $a_H = 0.25$, $d_H = 0.15$, $a_F = 0.56$ und $d_F = 3$	135
6.8.	Gesamtbiomasse der Heringe mit unterschiedlicher Jagdeffizienz. Parametrisierung: $a_H = 0.25$, $d_H = 0.09$, $a_F = 0.56$ und $d_F = 3$	136
6.9.	Gesamtbiomasse der Jungheringe mit Alter $age_H = \leq 6$ und ohne dynamischem Nährstoffgehalt. Parametrisierung mit: $a_H = 0.2$, $d_H = 0.15$, $a_F = 0.56$ und $d_F = 3$	137
6.10.	Langzeitverlauf der Biomasse zum Phytoplankton über die Zeit mit unterschiedlicher Umwandlungseffizienz. Parameter: $d_{ph} = 0.01$, $r_Z = 0.95$, $d_Z = 0.02$, $r_H = 0.95$, $d_H = 0.09$	138
6.11.	Phytoplankton Ausbreitung bei aktiven Heringen und Zooplankton. Parameter: $r_{ph} = 0.65$, $d_{ph} = 0.001$. Oben links $t = 50$, oben rechts $t = 100$, unten links $t = 200$, unten rechts $t = 500$, Geographischer Raum siehe 6.3.7	139
6.12.	Kumulierte Biomasse des Phytoplanktons über die Ausbreitung mit unterschiedlicher Wachstumsparametrisierung. Parameter: $B_{ph}(t_0) = 10000$, $d_{ph} = 0.005$	140
6.13.	Kumulierte Biomasse des Phytoplanktons über die Ausbreitung mit unterschiedlicher Wachstumsparametrisierung. Parameter: $B_{ph}(t_0) = 10000$, $d_{ph} = 0.005$	141
6.14.	Kumulierte Biomasse eines instabilen Nahrungsnetzes. Parameter: $r_{ph} = 0.85$, $d_{ph} = 0.01$, $r_Z = 0.95$, $d_Z = 0.02$, $r_H = 0.75$, $d_H = 0.09$	141
6.15.	Kumulierte Biomasse eines stabilen Nahrungsnetzes. Parameter: $r_{ph} = 0.75$, $d_{ph} = 0.01$, $r_Z = 0.85$, $d_Z = 0.02$, $r_H = 0.85$, $d_H = 0.09$	142
6.16.	Verteilung von Phytoplankton, Zooplankton und Hering mit Clusterbildung zum Zeitpunkt $t = 50$, Geographischer Raum siehe 6.3.7	143
6.17.	Ergebnis-Plots der Simulation ohne dynamischem Nährstoffgehalt. Parametrisierung mit: $a_F = 0.56$, $d_H = 0.15$, und $d_F = 0.12$	144
6.18.	Ergebnis-Plots der Simulation ohne dynamischem Nährstoffgehalt. Parametrisierung mit: $a_F = 0.56$, $d_H = 0.15$, und $d_F = 0.12$	145
6.19.	Positionierung von Fischern und Heringen in der Umgebung zu einem Zeitpunkt $t = 25$ mit Pfad zum naheliegendsten Hafen, Geographischer Raum siehe 6.3.7	146
6.20.	Laufzeitmessung alle 5 Ticks zum Vergleich eines direkten Modells gegenüber eines transformiertem Modells durch die MARS DSL	149
6.21.	Anzahl noch lebender Schafe I_S zu Gräsern I_G über die Zeit zum Vergleich des direkten Modells gegenüber eines transformiertem Modells durch die MARS DSL	150
6.22.	Anzahl noch lebender Wölfe I_W und Schafe I_S über die Zeit zum Vergleich des direkten Modells gegenüber eines transformiertem Modells durch die MARS DSL	151

A.1. Sender Empfänger Muster zur unidirektionalen 1:1 Kommunikation	161
A.2. Sender Empfänger Muster zur bidirektionalen 1:1 Kommunikation	161
A.3. Sender Empfänger Muster zur unidirektionalen 1:n Kommunikation	162
A.4. Sender Empfänger Muster zur bidirektionalen 1:n Kommunikation	163
A.5. Evolutionsmuster um einen Agenten in einen anderen Typ zu morphen	163
A.6. Verteilung von Kommandos ausgehend von einem verantwortlichen Agenten	164
A.7. Weiterleitung von Nachrichten mit Verlust	165
B.1. Ein integrierter Webeditor zur Modellierung innerhalb der MARS Cloud mit angebundenem Modellgenerator	166
B.2. Vervollständigungsassistent innerhalb des Online Editors inklusive Dokumenta- tion und integrierter MathML Sprache. Beispielbeschreibung der $\log_2(xyz)$ Logarithmus Definition	167
B.3. Vervollständigungsassistent innerhalb der Eclipse Zusatzlösung inklusive Do- kumentation mittels formaler MathML Sprache. Beispielbeschreibung der $\ x\ $ Absolut Definition	167
B.4. Zeitreihen-Layer integriert in ein exemplarisches Baummodell mit Abfrage von Daten und einer Agentenerzeugung	168
B.5. Beispielmodell des Sender-Empfänger Prinzips vollständig abgebildet mit der MARS DSL	169
B.6. Übersichtsfenster des aktuellen Modells, unterteilt in MARS spezifische Ele- mente, Module und eingesetzten Beschreibungselementen.	170

Listings

5.1. Partielle Vorlage zur Erzeugung eines Agentenkonstruktors.	98
6.1. Teil des Phytoplankton Modells mit Phosphor Bezug	118
6.2. Ausbreitungsmodell des Phytoplanktons	120
6.3. MARS DSL Modell eines Hering	122
6.4. MARS DSL Modell von Zooplankton	123
6.5. MARS DSL Kernverhalten des Zooplankton	124
6.6. MARS DSL Modell des Fischers	125
6.7. MARS DSL Modell der Arbeitsaktion des Fischers	126
6.8. MARS DSL Modell der aktiven Aktion zur Abbildung eines Fangversuchs für Heringe	127
6.9. MARS DSL Modell der Verkaufsaktion des Fischers	127
6.10. MARS DSL Aktion zur blockierten Fahrt	128
6.11. Layer zur Abbildung der Meeresbewohner	129
6.12. Layer zur Abbildung der Fischer mit gemeinsam geteilten Daten	129
6.13. Layer der Fischer mit gemeinsam geteilter Funktionalität	130
6.14. Raster-Layer zur Abbildung des territorialen Seegbiets	132
6.15. Raster-Layer zur Abbildung der Bathymetrie Karte	132
6.16. Raster-Layer zur Abbildung der Phosphor Konzentration	133

1. Einleitung

Systementwicklung für Multiagentensysteme (MAS) ist ein komplexes Forschungsfeld. Von einfachen Modellen mit wenigen bis keinen Interaktionen bis hin zu komplexen Welten und intelligenten nachgebildeten Verhaltensweisen erstrecken sich Modelle die durch die Modellierung einzelner Individuen entwickelt wurden. Insbesondere beim Entwurf komplexer oder gar verteilter Systeme sind MASs ein vielversprechender Ansatz und werden als leistungsfähige Technologie angesehen, indem Systeme in Form von autonomer Einheiten, die sich in einer Umgebung befinden, um ihre Ziele zu erreichen, entwickelt werden und flexible Zusammenarbeit leisten. Die Koordination zwischen Agenten kann entweder dadurch erreicht werden, dass sie miteinander in Bezug auf Protokoll- und Interaktionssprachen auf hoher Ebene interagieren, und/oder durch Pläne, die das Verhalten des Agenten insgesamt definieren, um Ziele zu erreichen. Verglichen mit dem objektorientierten Paradigma nach Jennings (2001) sind die Vorteile des agentenbasierten Rechnens die Autonomie der Anwendungskomponenten, die Bereitstellung einer besseren Trennung von Konzepten aufgrund der expliziten Berücksichtigung von Verantwortlichkeiten *Separation of Concerns* (SoC) mit Bezugnahme situationsabhängigen Handelns von Umweltressourcen und der Auseinandersetzung mit dynamischen und abstrakten Interaktionen - d.h mit modell-spezifischen anstatt mit architektonischen Konzepten.

1.1. Problemstellung

Die Entwicklung komplexer Systeme durch die Anwendung des agentenorientierten Paradigmas erfordert adäquate Modellierungstechniken und -methoden, die Schlüsselfunktionen bereitstellen, um die Komplexität bei der Entwicklung agentenbasierter Systeme zu verringern (Ahlbrecht u. a. (2014)).

Um das Fehlen ausgereifter agentenbasierter Modellentwicklungsmethodiken, -sprachen und -werkzeugen zu beheben, werden folgende Forschungsfragen näher betrachtet: **Was sind die wichtigsten und eigentlichen Kernbausteine von MASs?** Ein fehlendes Vokabular und das Unverständnis der agentenbasierten Modellierung sind sicherlich eines der Hauptgründe weswegen dieses Paradigma immer noch eine sehr hohe Einstiegshürde darstellt. Mit Vergleich diverser Beschreibungssprachen wie mit GAMA (Grignard u. a. (2013)), dem Vokabular der

GAIA Plattform aus Wooldridge (2000) oder SARL (Rodriguez u. a. (2014)) sind diverse Formen zur Formulierung möglich, bei dem ein einziges gemeinsames Konzept bleibt, der Agent. Auch die Vermischung jedes Metamodells, mit deren positiven Eigenschaften mag funktionieren, wenn die Domäne sich auf akademische Arbeiten beschränkt, nicht jedoch für industriell einsetzbare Modelle - aka Fischerei Profitabilität (Cooper und Jarre (2017a)) und (Cooper und Jarre (2017b)) oder Verkehrsanalysen (Tchappi Haman u. a. (2017)).

Was ist eine geeignete textuelle Visualisierung und Notation einer MAS-Sprache? Dieser Umstand steht in direktem Zusammenhang mit dem Endanwender, da diese Benutzbarkeit des Werkzeugs nachgiebig beeinflusst (Barišic (2013)). Die Entwicklung von agentenbasierten Systemen ist im Allgemeinen komplexer und fehleranfälliger als herkömmliches objektorientiertes Design (Jennings (2001)). Daher müssen spezifische Methoden entwickelt werden, die die Gesamtkomplexität reduzieren. Selbst wenn viele Forschungstools existieren, werden sie hauptsächlich von Grund auf neu erstellt, und es wurden nur wenige Anstrengungen unternommen, sie in integrierte *Entwicklungsumgebungen* (IDEs) wie Eclipse zu integrieren. Positive Beispiele gibt es dagegen auch (Grignard u. a. (2013)) (Wilensky (2015)).

Was ist eine adäquate Semantik, die Test-, Validierungs- und Codegenerierungsprobleme unterstützt? Der Mangel an Semantik stellt ein weiteres Hindernis dar. Sollte ein Vokabular und eine klare Notation zur Verfügung stehen, sind die generierten Artefakte in Bezug auf alle Anforderungen, die für die vollständige Codegenerierung benötigt werden, *selten vollständig* (Ahlbrecht u. a. (2014)). Eine formale Semantik kann das Verständnis der Fachexperten für ein korrektes Modell verbessern, indem sichergestellt wird, dass alle Anforderungen erfüllt werden, um die Übersetzung automatisieren zu können. Und selbst wenn Code erzeugt wird, ist normalerweise eine umfassende *Qualitätssicherung* erforderlich, die einen erheblichen Teil des Entwicklungsaufwands in Anspruch nehmen. Dieser Aufwand kann verringert werden, wenn die Validierungs- und Testeinrichtungen auf präzisen Semantiken und Techniken wie der Modellprüfung basieren. Bekannte Mittel sind zu Hauf in der Industrie verfügbar, seien es Spezifikations Sprachen wie Z oder Object-Z (O'Regan (2017)), formale Verifikation wie z.B. mittels Petri Netzen (Leroy (2014)) oder statische Prüfung (Bettini (2016)). Das Ganze wird dementsprechend schwieriger sollten *eigene Beschreibungskonzepte* in die Modellierung fallen, die das Werkzeug gegenüber anderen hervorstechen lässt, wie das gewählte Schichtenmodell der MARS-Plattform (Hüning u. a. (2016)).

Die letzte und wohl wichtigste Fragestellung ist dagegen: **Wie schließt man die Lücke zwischen agentenbasiertem Design und Implementierung?** Agentenbasierte Modelle werden immer ausgehend einer Forschungsfrage entwickelt und die Frage bleibt, wie sich daraus entwickelte konzeptionelle Modelle leicht und verständlich in die Implementierung über-

führen lassen (Glake u. a. (2017)) (Voelter u. a. (2013)). Dieser Umstand vertieft sich, selbst wenn praktische Modellierer die Methodiken zur Entwicklung befolgen kann es Schwierigkeiten kommen, die durch fehlende Reife der Werkzeuge einhergehen (Barišić u. a. (2011)).

Weitere in der Literatur bekannte Probleme sind: *Wie lässt sich die Interoperabilität zwischen bestehenden Modellen verbessern?* (Macal (2016)) bei der versucht wird eine Austauschbarkeit von Modellen zu erhalten und das als generelles (Daten-) Integrationsproblem aufgefasst werden kann. *Wie kann die Benutzerfreundlichkeit von agentenbasierten Werkzeugen verbessert werden, um das Design für Fachexperten zu vereinfachen?* (Voelter u. a. (2013)) ist dagegen ein übliches Sprach- als auch Softwareproblem, dass die Benutzbarkeit der Anwendung setzt (Starke (2015)). Mit eigenen Entwicklungsumgebungen, guten Benutzerinteraktionen oder unterschiedlichen Visualisierungen wird immer versucht die Komplexität zu verstecken und den Fokus auf das Wesentliche zu legen (Kahraman und Bilgen (2015)).

1.2. Aufbau der Arbeit

Die vorliegende Arbeit ist strukturiert in vier große Bereiche. Zunächst werden im nachfolgenden Kapitel *Verwandte Arbeiten 2* ähnliche Ergebnisse zur agentenbasierten Modellierung betrachtet, wovon ein technischer Teil in Kapitel *Grundlagen 3* seine Fortsetzung findet. Darin wird zudem auf die Kerneigenschaften dieser Art von Systementwicklung eingegangen, dessen Ausgangslage anschließend in der *Analyse 4* wieder aufgegriffen wird. Als eines der vier großen Bereiche beschäftigt sich die *Analyse* mit der Betrachtung der abstrakten Syntax inklusive Invarianten, Beschreibungs- und Abbildungsmöglichkeiten in das gewählte MARS-Zielsystem. Die technischen Feinheiten dagegen wie diese Sprache funktioniert, findet sich in Kapitel *Entwurf 5*. Architektonische Entscheidungen werden diskutiert und es wird sich technischen Besonderheiten der Sprache zugewandt. Abschließend wird in der *Evaluation 6* die Sprache kritisch durch drei Bewertungstechniken betrachtet, woraus sich im letzten Kapitel *Zusammenfassung und Ausblick* die zukünftigen Arbeiten für die Weiterentwicklung der Sprache ergeben.

2. Verwandte Arbeiten

Im Folgenden Kapitel soll auf verwandte Arbeiten mit Bezug auf domänenspezifischen Sprachen für die agentenbasierte Simulation eingegangen werden, inklusive einer eigenen Einordnung dieser Arbeit.

2.1. Modellierungssprachen

Diverse Arbeiten wurden auf dem Gebiet der Sprachentwicklung, insbesondere zum Teilgebiet agenten-orientierter Modellierungssprachen entwickelt. Allen voran die Sprache NetLogo [Wilensky \(2015\)](#), die als eines der bekannteren Derivate für die agentenbasierte Simulationenentwicklung gilt. Aus der Kombination von Oberflächenbildung inklusive Parameterfelder und dem direkten Abgriff dieser Eingaben mithilfe der eigenen Interpretersprache versucht **NetLogo** die Konstruktion von agentenbasierten Modellen aus einer Hand anzubieten. Die Sprache, abgeleitet aus der Kinderprogrammiersprache LOGO, selbst ist allerdings weniger individuenbasiert ([Macal \(2016\)](#)), sprich es wird nicht ausgehend des Verhaltens eines einzelnen Individuums die Gesamtheit des Modells betrachtet, sondern stärker auf mehrere Teilnehmer gleichermaßen eine ganze Aktionen definiert ([Macal und North \(2014\)](#)). Aufbauend auf einer großen Menge möglicher Kommandos, eingebettet in die Grammatik selbst, wird jeder Schritt einzeln zur Laufzeit interpretiert ([Aho \(2008\)](#)), bestückt durch Eingaben aus der konstruierten Oberfläche. Insgesamt ist diese Sprache dadurch wesentlich flexibler und weist mit ihrer Modellbibliothek ein umfangreiches Spektrum von Themengebieten (Ökologie, Krankheiten, Kunst, Spiele,..) auf, die dem eigenen Modell sehr förderlich sein kann, was wohl auch eine der Hauptgründe für die große Akzeptanz darstellt. Ähnlich geht die Sprache **GAML** ([Grignard u. a. \(2013\)](#)) vor, die ähnlich zum MARS System ein eigenes Simulationsframework anbietet. Vergleichbar mit der MARS DSL konzentriert sie sich auf die Beschreibung einzelner Entitäten - in **GAML** als Spezies bezeichnet - in der sich neben der Formulierung des Agentenverhaltens auch Einzelbeschreibungen zu einem explizit zu modellierenden Environment befinden, d.h. das resultierende Modell erwartet eine Spezifizierung der zugrundeliegenden Umgebung und der Verarbeitung jeder Bewegungsaktion darin. Dem steht die Sprache **SARL** ([Rodriguez u. a. \(2014\)](#)) gegenüber, die trotz des Agentenparadigmas keine domänenspezifische Modellierungssprache (DSML)

darstellt, sondern als modulare allgemeingültige Sprache fungiert. Deren Beschreibungsmittel legen den Fokus auf gültige Kerneigenschaften von Agenten (Wooldridge und Jennings (1995)), indem sie eine direkte Abstraktion zur Behandlung von Aspekten der Nebenhäufigkeitskontrolle, Verteilung, Dezentralisierung und Reaktivität aufweisen. Mit dem Konzept holonischer Agenten (Rodriguez u. a. (2011)) versucht die Sprache eine rekursive Systemstruktur zu etablieren, deren Nachrichtenaustausch sich über die Verwendung definierter *Capabilities* und *Skills* verarbeiten und steuern lässt. SARL vermittelt wohlgeformte Beschreibungsmittel für einzelne Agenten, die weitläufig anerkannte Konzepte dieses Paradigmas umsetzt und daher auch als Inspiration für die MARS DSL gilt (Glake u. a. (2017)). Aus der natürlichen Verteilung des Systems über Individuen kommt dem funktionalen Sprachparadigma, durch Wegfall potenzieller Seiteneffekte, eine besondere Bedeutung zu. FABLES, (ausgeschrieben Functional Agent-Based Language for Multi-Agent Simulations) ist eine Modellierungssprache die durch den Einsatz funktionaler Schleifen, über Rekursivität, einem blockierenden Nachrichtenempfang über Pattern-Matching und der Vermeidung von Neuzuweisungen das funktionale Paradigma und die Verhaltenslogik im Fokus hat. Es stellt Mengenkonzepte und Berechnungskonzepte in Form einer vornehmlich mathematischen Notation dar, um dadurch die Akzeptanz beim Benutzer durch bewährtes zu erhöhen. Aschermann u. a. (2016a) bieten mit ihrer **LightJason** Alternative dagegen eine BDI Sprache, die vermehrt auf die Bildung intelligenter Verhaltensweisen setzt, indem sie aufbauend auf der Sprache **AgentSpeak(L)**, die Beschreibung von Plänen erlaubt, die in eine Menge von Regeln, für den rückwärts-referenzierten Inferenzmechanismus, übersetzt werden.

2.2. Einordnung

Diese Arbeit ordnet sich in den Bereich Sprachentwicklung, mit Bezug zu agentenbasierten Modellierungssprachen, ein. Entgegen den genannten Arbeiten in 2.1 ist die Domäne spezifischer für den Bereich der Multiagentensysteme und konzentriert sich auf Individuen basierte Modelle (Macal (2016)). Es werden weitläufig anerkannten Beschreibungsmittel und Strukturierungsvorschläge mit einbezogen, die anteilig durch formale Spezifikationen der Invarianten komplettiert wird. Nicht Teil dieser Arbeit ist dagegen die Formulierung operationeller Semantiken oder theoretischer Grundlagen dieser Sprache, wie der Frage nach der Berechenbarkeit unterschiedlicher Eigenschaften - WHILE, LOOP, RAM, Turing usw. (Aho (2008)). Als Teil der Bewertung zieht die Sprache ein Fischerei Modell auf, aus dessen MARS DSL Realisierung die Machbarkeit für solche Modelle hervorgeht. Das Modell ordnet sich dabei der Modellierung ökologischer Zusammenhänge von Nahrungsnetzen ein, dass sich auf den Kreislauf von drei

2. Verwandte Arbeiten

Entitäten konzentriert und durch einen externen Einfluss die Dynamiken der Populationen betrachteten will, ähnlich der Arbeit von [Xing u. a. \(2017\)](#).

3. Grundlagen

Das folgende Kapitel spiegelt die grundlegende Struktur eines MAS an und gibt mit einer Reihe von Definition die weitläufig anerkannten Kern der agenten-orientierten Modellierung in Bezug zum Agentenkonzept an sowie Hintergründe zum Themengebiet der modellgetriebenen Entwicklung (MDD).

3.1. Agentenorientierte Modellierung

Die Entwicklung hochwertiger Modelle zur Abbildung komplexer realer Gegebenheiten ist eine schwierige Aufgabe. Viele Entwicklungsparadigmen, vor allem aus der Softwareentwicklung versuchen entsprechend diese Komplexität zu reduzieren, beispielsweise mittels *objektorientierte* Programmierung (OOP). Zwar stellt die OOP bereits einen sehr guten Schritt in die richtige Richtung dar (Jennings (2001)), die vorhandenen Konzepte reichen jedoch längst nicht aus. Nach Jennings (2001) sind die wesentlich fehlenden Merkmale:

- Die Kernkonzepte wie zum Beispiel Objekte sind zu feingranular gehalten
- Interaktionen zwischen Objekten werden als zu rigide aufgefasst
- Die Möglichkeit zum Umgang mit Organisationsstrukturen als wesentlicher Bestandteil ist umständlich

Erst durch Einführung der agentenbasierten Modellierung insbesondere zur Simulation (ABMS), ist man damit einen wesentlichen Schritt näher gerückt (Macal und North (2014)). Sie umfasst verschiedene Disziplinen die die Modellierung komplexer Systeme und Problemstellungen, durch Abbildung auf eine oder mehrere Agentenstrukturen, einzelne Prozesse und Eigenschaften besser wiedergeben kann (Aschermann u. a. (2016b)). Unabhängig von diesem gewählten Modellierungsansatz - sei er individuenbasiert, adaptiv, autonom oder interaktiv - wird das Modell nach einer grundlegenden Systemstruktur konzipiert, die auf die Eigenschaften der Agenten eingeht (Wooldridge und Jennings (1995)).

3.1.1. Grundstruktur

Eine Herausforderung in der Definition einer plattformunabhängigen Sprache für Multiagentensysteme liegt in der Entscheidung welches Konzept, abstrakt von der Zielplattform die diesen architektonischen Stil unterstützt, aufgenommen werden (Karsai u. a. (2014)). Die Kernblöcke jedes MAS sind in Abbildung 3.1 nach Jennings (2001) zusammengefasst und betreffen die Agenten als miteinander interagierende Elemente, die gruppiert in einer Umgebung existieren. Drei Strukturen werden dabei unterschieden und im Detail in Jennings (2001) und Ahlbrecht u. a. (2016) diskutiert. Die *Organisationsstrukturen (Makro-Ebene)* gibt Mechanismen zur Einteilung von Agenten in Gemeinschaften an. Die *Interaktionsstrukturen (Mesos-Ebene)* fokussieren sich dagegen auf die Agentenkommunikation untereinander, mit Protokollen und Nachrichten, während die *Agentenstruktur (Mikro-Ebene)* als zuletzt versucht beispielsweise das Schlussfolgern über Pläne und Lernmechanismen im Agenten zu realisieren. Alle Blöcke sind zunächst unabhängig vom gewählten agentenbasierten Modellierungsansatzes (Macal (2016)). Sie versuchen jedoch der nachfolgenden Definition von Agenten und Systemen gerecht zu werden.

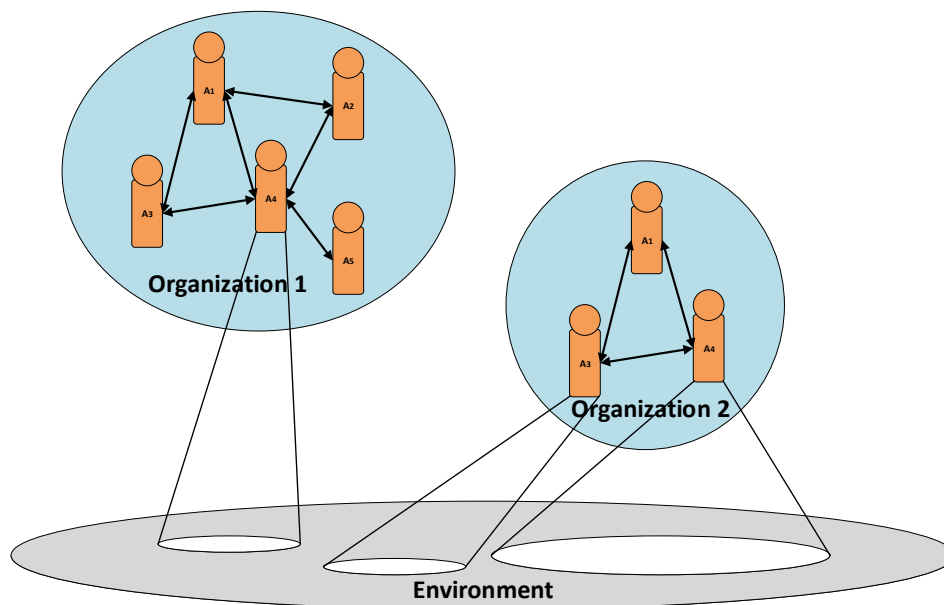


Abbildung 3.1.: Kernbausteine eines Multiagentensystems nach Jennings (2001)

Die Definition des Multiagentensystems selbst ist nach Jennings (2001) formuliert als System, das eine Menge von Agenten beherbergt, bei dem jeder eine beschränkte Menge von Fähigkeiten hat, zur Lösung von Aufgaben in *autonomer* Art und Weise.

Definition 3.1 Multiagentensystem nach Jennings (2001): *A MAS is a system that has the following properties: Each agent in a MAS has incomplete information, or capabilities for solving the problem, thus each agent has a limited viewpoint. There is no centralized control. Data is decentralized and computation is asynchronous.*

Jennings (2001) sieht ein solches System als Gesamtheit aller verfügbaren Informationen, bei dem jeder Agent nur einen Bruchteil davon innehat. Er konzentriert sich auf einzelne Agenten, wobei jeder von ihnen nur begrenzt in seinen Fähigkeiten ist, um seine Aufgaben ohne Kontrollverlust zu lösen. Jeder Agent hat seine eigenen Vorstellungen über die Umgebung, wie diese aussieht, funktioniert, und die als Grundlage für die eigenen Entscheidungen herangezogen wird.

3.1.2. Agentenstrukturen

In Anlehnung an Wooldridge und Jennings (1995) ist der Agent ein gekapseltes System, das innerhalb einer Umgebung sitzt, befähigt autonom Aktionen in dieser auszuführen, um das eigene Designziel zu erreichen. Sie sind in der Lage zum Empfang und Versenden von Nachrichten sowie zum Agieren in der Umgebung, und stehen damit folglich in einer Interaktion mit diesen Elementen. Durch Schlussfolgerungen (zum Beispiel Planen, Entscheidungsfindung und Lernen) weisen sie ein flexibles dennoch rationales Verhalten auf Zambonelli u. a. (2003). In der Literatur finden sich diverse Definitionen zu einem Agenten (Russell und Norvig (2002)) (Ahlbrecht u. a. (2016)) (Jennings (2001)). Im Folgenden werden zwei bekannte weit anerkannte Definitionen als Teil ABMS nach Macal (2016) diskutiert.

Definition 3.2 Agent nach Russell und Norvig (2002): *An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors.*

Russell und Norvig (2002) geben die allgemeinste Definition zum Stichwort *Agent* an. Sie konzentriert sich hauptsächlich auf die Interaktion der Agenten innerhalb der Umgebung. Agenten verändern diese Umgebung durch deren Effektoren und Reaktionen, die wiederum mittels Sensoren beeinflusst werden. In Bezug auf das zentrale ABMS ist zwar die Interaktion mit der Umgebung eines der wichtigen Merkmale der Agenten, jedoch gibt es wesentliche Eigenschaften, die vor allem durch die am meisten zitierte Definition aus dem MAS Bereich sichergestellt wird.

Definition 3.3 Agent nach Wooldridge und Jennings (1995) und Jennings (2001) mit folgenden Eigenschaften:

Autonomie Agenten sind unabhängig voneinander, was bedeutet, dass ohne Eingreifen anderer Mittel sie ihre Kontrolle über die *inneren* Zustände beibehalten. Er trifft *unabhängige* Entscheidungen und Aktionen, der zusätzlich nach **Wooldridge und Jennings (1995)** und **Jennings (2001)** seine eigenen Überzeugungen, Wünsche und Absichten innehält die nicht eine untergeordnete Rolle gegenüber anderen Agenten besitzen.

Reaktivität Agenten sind in der Lage auf Änderungen in der Umgebung zu reagieren. Alle im Sinne um das spezifische Ziel rechtzeitig zu erfüllen. Wenn den Agenten bewusst wird, dass ihre Pläne schiefgelaufen sind, ignorieren sie diese Tatsache nicht und versuchen weiterhin, ihre Pläne zu erfüllen, aber sie reagieren darauf, indem sie eine *alternative* Vorgehensweise wählen.

Proaktivität Agenten können ein zielgerichtetes Verhalten besitzen, indem sie selbst die Initiative ergreifen und *eigenständig* mögliche Aktionen erzeugen. Die Eigenschaft der Proaktivität schließt damit auch rein *passiv* agierende Agenten - Gras mit nur einem bereitgestellten Wert zur Repräsentation der Biomasse - die niemals eigenständig agieren aus. Das Erreichen der eigenen Ziele kann dazu andere Ziele ausnutzen, die allerdings auch im Widerspruch stehen können und aus diesem Grund zurückgezogen werden müssen (**Russell und Norvig (2002)**).

Sozialfähigkeit Agenten sind in der Lage mit anderen Agenten oder Menschen über ein bestimmtes Protokoll und Austauschformat zu interagieren und mit anderen Agenten zu kooperieren, um ein spezifisches Ziel zu erreichen. Der Informationsaustausch selbst ist dabei nicht wirklich die soziale Fähigkeit, sie entsteht erst durch das autonome Kooperieren und Verhandeln der Agenten zur Erreichung ihrer Ziele.

Wooldridge und Jennings (1995) und **Jennings (2001)** sehen Agenten als autonom an, in der Lage über Gruppen innerhalb einer Umgebung zu kommunizieren. Zudem können sie proaktiv und reaktiv agieren. Sie bezeichnen die Eigenschaften des Agenten darüber hinaus als *weak notion of agency*.

3.1.3. Organisationsstrukturen

Definition 3.4 Organisation nach Wooldridge (2000): We view an organisation as a collection of roles, that stand in certain relationships to one another, and that take part in systematic institutionalised patterns of interactions with other roles.

Wooldridge (2000) betrachtet die Organisation als Interaktionsmuster, als Teil der Interaktionsprotokolle und agentenbasierter Organisationen. Der Fokus innerhalb von Organisationen

verschiebt sich daher hauptsächlich von der internen Agentenarchitektur, wie in Abschnitt 3.1.2 beschrieben, hin zur Kommunikation zwischen einzelnen Rollen. Wie jedoch von [Ferber u. a. \(2004\)](#) festgestellt hat, fehlt der genannten Definition ein sehr wichtiges Merkmal: Ihre Partitionierung, d.h. die Art wie Grenzen zwischen Unterorganisationen gesetzt werden.

Nach [Jennings \(2001\)](#) ist jede Organisation selbst wieder ein MAS durch die die Koordination der Aufgaben zwischen den teilnehmenden Mitgliedern in Form eines gemeinsamen Verhaltens erreicht wird. Das Problem eine Organisationsstruktur zu finden, mit der jeder Agent einverstanden ist, wird dort als Differenzierungsproblem betrachtet. Verdeutlicht wird jedoch, dass nicht jedes MAS eine Organisation darstellt. [Jennings \(2001\)](#) unterscheidet nämlich zwischen der Organisation eines MAS und des MAS als eigenständige Organisation. Ersteres bezieht sich auf die interne Struktur und Koordination der Bestandteile eines MAS, während letzteres sich auf ein MAS bezieht, das sich auf seine interne Struktur stützt, um ein gemeinsames Verhalten zu bewirken.

3.1.4. Interaktionsstrukturen

Eines der wichtigsten Punkte in der Multiagenten Gemeinschaft betrifft die Frage wie und warum autonome Agenten miteinander kooperieren sollten. In [Aschermann u. a. \(2016a\)](#) wird zu dieser Frage nämlich auf ein Paradoxon hingewiesen, da die Kooperation wiederum das Einbeziehen anderer Entitäten beinhaltet und einen Teil der Autonomie aufgeben werden muss, um kooperativ entgegen einem gemeinsamen Ziel zu arbeiten. Grundlegend ist die Agenteninteraktion die Fähigkeit zur Spezifikation von Bedingungen und wünschenswerten Eigenschaften. Erst durch den Einsatz wird die Möglichkeit geboten, eigenen Ziele zu erreichen. Eine breit anerkannte Definition ist dazu nach [Lind \(2001\)](#) gegeben:

Definition 3.5 *Interaktion nach Lind (2001)*: Interaction is the mutual adaption of the behavior of agents while preserving individual constraints.

Lind betrachtet Interaktionen als die Art von Verhalten, das lediglich mit anderen Agenten zusammenhängt und daher mehr als nur der bloße Austausch von Nachrichten ist. Die Interaktion selbst wird dort als gegenseitige Anpassung betrachtet was bedeutet, dass interagierende Agenten ihr Verhalten zum Zweck der Konversation koordinieren müssen, wie zum Beispiel über ein Protokoll ([Aschermann u. a. \(2016b\)](#)).

3.1.5. Umgebungsstrukturen

Die Umgebung spielt eine zentrale Rolle innerhalb eines MAS. Es stellt den Rest des Systems abzüglich der Organisation dar und beherbergt alle Ressourcen, die überhaupt im System

nebenher vorkommen können. Dazu zählen die Agenten selbst, physikalische Objekte, kognitive Besonderheiten, Techniken als auch Normen, zum Beispiel mittels deontischer Logik mitgezählt (Aschermann u. a. (2016b)).

Definition 3.6 Umgebung nach Jennings (2001): *Before a MAS comes into existence, environment is the set of resources and phenomena that can determine whether or not the system is generated and what its structure and functioning will be. After a MAS has been generated, environment is the set of resources and phenomena that the system, as a cognitive entity, believes are outside its boundaries, and can affect its structure and functioning.*

3.1.6. Modellierung

Die Vielfalt der Anwendungen der ABMS ist groß und erstreckt sich über diverse Disziplinen in den Sozial- und Naturwissenschaften, über technische Systeme und weit über eigentliche Simulationen hinaus, wie in den Bereichen Technik, Betriebswirtschaft, Betriebsmanagement Verkehr und Ähnlichem (Macal und North (2010)) (Macal und North (2014)). Im Hinblick auf die essenziellen Eigenschaften dieser Entwicklungsmethodik zur Konstruktion von Modellen sowie der Beziehung des ABMS zu anderen Modellierungsansätzen und Simulationstechniken - analytische Modelle, diskrete Ereignissimulationen, Monte-Carlo Simulation usw. - ist es zum Teil schwer, trotz der großen Menge vorhandener Modelle ein eigenes Modell zu konstruieren (Tchappi Haman u. a. (2017)) (Wilensky (2015)) (Drogoul u. a. (2002)). Es existiert eine große Menge unterschiedlicher Ansichten, sowohl von innen als auch von außen auf diesen Ansatz (Macal und North (2014)) der versucht das ABMS besser zu differenzieren, was sich dahinter verbirgt, wie die Einsatzmöglichkeiten aussehen und wie sich die Aufgabe der Beschreibung eines Modells dadurch vereinfachen lässt. Eine solche Modellbeschreibung mit ausgehender Problematik oder Fragestellung macht letztlich ein System notwendig, die mit einer Mehrzahl von Agenten umgehen können und bei der Interaktion zwischen diesen, innerhalb oder außerhalb einer Organisationsstruktur oder der Umgebung stattfinden, wie die in vielen Arbeiten immer wieder deutlich wird (Wilensky (2015)) (Grignard u. a. (2013)). Nach Macal (2016) werden hierzu zwischen vier verschiedenen Richtungen unterschieden:

Zu jeder ABMS Richtung existieren unterschiedliche Arten von Modelle. Die Art wie welches Modell entwickelt wird, hängt jedoch von der betroffenen Fragestellung ab und dem Modell selbst (Wilensky (2015)). Mit der MARS DSL wird versucht den Ansatz der individuenorientierten Modellierung zu verfolgen, da dies mit aktuellen Sprachen wie GAML (Grignard u. a. (2013)), NetLogo (Wilensky (2015)) oder SARL (Rodriguez u. a. (2014)) nur geringfügig oder überhaupt nicht unterstützt wird.

3. Grundlagen

<i>ABMS Definition</i>	<i>Individualität</i>	<i>Verhalten</i>	<i>Interaktionen</i>	<i>Anpassbarkeit</i>
Individual ABMS	Individuelle heterogene Agenten	Vorgegebenes Skript	Beschränkt	Keine
Autonome ABMS	Individuelle heterogene Agenten	Autonome Dynamiken	Beschränkt	Keine
Interaktive ABMS	Individuelle heterogene Agenten	Autonome Dynamiken	Zwischen anderen Agenten und der Umgebung	Keine
Adaptive ABMS	Individuelle heterogene Agenten	Autonome Dynamiken	Zwischen anderen Agenten und der Umgebung	Agenten verändern das Verhalten über die Simulation

Tabelle 3.1.: Unterschiedliche ABMS Definitionen und ihre Eigenschaften nach [Macal \(2016\)](#).

3.2. Modellgetriebene Entwicklung

In Anlehnung an [Mohagheghi u. a. \(2013\)](#) wurde zu Beginn der Softwareentwicklung hauptsächlich mit Low-Level-Maschinencode programmiert, was zu maschinenorientierten Programmen führte, die aus einer Reihe von Nullen und Einsen bestanden. Eine erste Verbesserung war die Einführung von Assemblersprachen (1950-1965). Die nächste Abstraktionsebene war nach [Kleppe u. a. \(2003\)](#) und [Boydens und Steegmans \(2004\)](#) notwendig, um sich von der maschinenorientierten Art des Rechnens zu einer anwendungsorientierten Art zu bewegen. Ab diesem Zeitpunkt wurden prozedurale Programmiersprachen (3GLs) wie Pascal, Fortran und Cobol eingeführt. Jetzt war es möglich, ein Programm nahezu unabhängig vom Prozessoren zu schreiben. Anfang der 1980er Jahre wurden die ersten objektorientierten Sprachen (C ++, Smalltalk) vorgestellt, die völlig plattformunabhängige Lösungen ermöglichen. Programme, die auf bestimmten Architekturen geschrieben sind, konnten jetzt auf jede andere Plattform portiert werden, wenn die jeweilige virtuelle Maschine verfügbar ist (vgl. [Boydens und Steegmans \(2004\)](#)). In den späten 1990er Jahren wurden die Anforderungen eines spezifischen Programms auf der Grundlage von Ansätzen wie UML grafisch formulierbar. Die Notation bietet dem Anwendungsentwickler nun die Kernsystemarchitektur grafisch näher zu beschreiben und anhand der resultierenden Struktur und der Laufzeiten den Quellcode manuell zu definieren, um die architektonischen Anforderungen erfüllen zu können. Da die Ausführungsumgebung jedoch weiterhin manuelle erstellt wird, kann diese zu einer Lücke der von Domänenexperten spezifizierten Anforderungen und des tatsächlichen Systems führen, sei es aus Gründen

fehlender Fachkenntnisse oder Fähigkeiten. Damit das Aufkommen falscher Interpretation von Problemen oder falscher Modellverfeinerungen reduziert wird, wurde die Einführung der modell-getriebenen Entwicklung immer interessanter (Mohagheghi u. a. (2013)), indem automatisch die Anforderungen des Modells in ein lauffähiges Modell überführt wird. Das Synonym Model-Driven Development (MDD) (oder auch Model-Driven-Engineering (MDE)) entwickelt sich zum Stand der Praxis und ist dabei längst Teil der Entwicklungsprozesse größerer Softwaresysteme geworden, sei es zur Erzeugung von Oberflächen, der Erzeugung einer globalen Softwarestruktur mit Komponenten und Abhängigkeiten oder der Pflege des eigenen Datenmodells durch Erzeugen der Objektstruktur aus relationalen Modellen oder umgekehrt. In Zusammenhang mit Boydens und Steegmans (2004) ist MDD eine Teilmenge der Systemtechnik, in der der Prozess vornehmlich auf die Verwendung von Modellen und Modell-Engineering beruht. Die Lösungen sind damit unabhängig der gewählten Middleware, (Unternehmens-)Architektur und Zielsprache. Ein MDD Werkzeug bietet die Möglichkeit zur Differenzierung von Problemstellungen, die Verfügbarmachung von anwendbaren Mustern, syntaktische und semantische Validierungsmaßnahmen und eine Codeerzeugung. Aus dieser Grundlage heraus bietet das MDD Paradigma eine besser Lösung zur Interoperabilität und darüber hinaus zur Design Zeit, effektive Wege den Modellierer konkret auf mögliche Fehlentscheidungen hinzuweisen oder Empfehlungen zu geben. Eine Modellentwicklung wird als Modellgetrieben angesehen, wenn folgende Anforderungen erfüllt sind:

1. Der System Prozess differenziert deutlich zwischen plattformabhängigen und plattform-spezifischen Modellen
2. Modelle halten die zentrale Rolle inne, nicht nur zu Beginn der Entwicklungsphase, sondern auch während späterer Pflegemaßnahmen zur Refaktorisierung und der Weiterentwicklung.
3. Der gesamte Entwicklungsprozess findet hauptsächlich mit Modellen unterschiedlichen Abstraktionsgrad und unterschiedlichen Sichten darauf statt
4. Modelldokumentation und Beziehungen zwischen Modellen bieten eine präzise Grundlage für Verfeinerungen sowie Transformationen

Nach Kleppe u. a. (2003) und Mohagheghi u. a. (2013) bietet MDD ein großes Einflusspotential auf aktuelle Entwicklungsansätze. Nach Gargantini u. a. (2010) bietet es folgende Vorteile:

Höhere Produktivität MDD hilft bei der Verringerung des Zeitaufwands, indem es Codestrukturen und Artefakte automatisch aus den Modellen erzeugt. Selbst wenn Transfor-

mationen manuell definiert werden müssen, sorgt eine sorgfältige Planung dafür das letztendlich Kosten reduziert werden.

Pflege und Korrektur Der Einsatz von MDD hilft bei der Vermeidung von Legacy Systemen die nur mit bestimmten Technologien zusammenspielen, indem es zu einer pflegeleichteren Architektur führt, in der Änderungen schnell und konsistent vorgenommen werden und bei der die Migration von Komponenten auf neue Technologien leicht erfolgt. Leichter bedeutet in diesem Zusammenhang, dass Änderungen an der technischen Architektur allein durch Änderung einer Transformation vorgenommen werden. Die Anwendung auf alten Modellen erzeugt die neuen Systeme, unter aktualisierten Bedingungen.

Konsistenz Die Anwendung gewöhnlicher Muster und wichtiger architektonischer Entscheidungen ist eine fehleranfällige Aktion. MDD stellt die Konsistenz der Artefakte sicher, solange die Transformation deterministisch bleiben.

Anpassbarkeit Anpassbarkeit ist ein wichtiges Qualitätsmerkmal. Das Hinzufügen oder Entfernen von Funktionen ist wesentlich einfacher, da bereits der Zeitaufwand in die Automatisierung investiert wurde.

Wiederholbarkeit Der Erlös aus der MDD Arbeit durch fortschreitende Entwicklung der Transformation erhöht sich stetig. Die Verwendung getesteter Transformationen verbesserte die Möglichkeit Vorhersagen über die Entwicklung neuer Funktionalität zu treffen und damit das Risiko von architektonischen und technischen Probleme zu vermeiden, da diese bereit gelöst sind.

Langzeitwert Durch Handhabung allein über Modellen bilden sie den größten Wert, mit denen gehandelt wird. Hoch entwickelte Modelle sind reliabel zu Änderungen von Technologien und ändern sich nur, wenn sich die zugrundeliegenden Anforderungen ändern.

3.2.1. Modellgetriebene Architektur

Diverse Architekturen und Ansätze wurden zur Realisierung von MDD entwickelt (Mohagheghi u. a. (2013)) (OMG (2014)). Die wohl bekannteste ist die Architektur der Objekt Management Group (OMG) (Belaunde u. a. (2003)) die in Zusammenhang mit der Meta Object Facility (MOF) (OMG (2014)) durch Model-driven Architecture (MDA) einen Standard entwickelt haben, der versucht die Expertise aus den Modellen herauszuziehen und beizubehalten. Die MDA ist eine Instanz des MDD und bietet eine praktische Reihe von Standards zur Verbesserung der

eigenen Rentabilität sowie Kapselung vieler wertvoller Ideen - insbesondere die Vielfältigkeit von Technologien die bei Verwendung von DSL's integrierbar und möglich werden - um daraus Vorteile zu erzielen (Voelter u. a. (2013)). Drei Hauptziele sind die Portabilität, Interoperabilität und Wiederverwendbarkeit. Die MDA trennt die Eigenheiten des Systems von der konkreten Ausführungsplattform. Das Metamodell der MDA ist in Abbildung 3.2 zu sehen und beschreibt ein System entweder textuell, visuell oder in einer Kombination aus beidem.

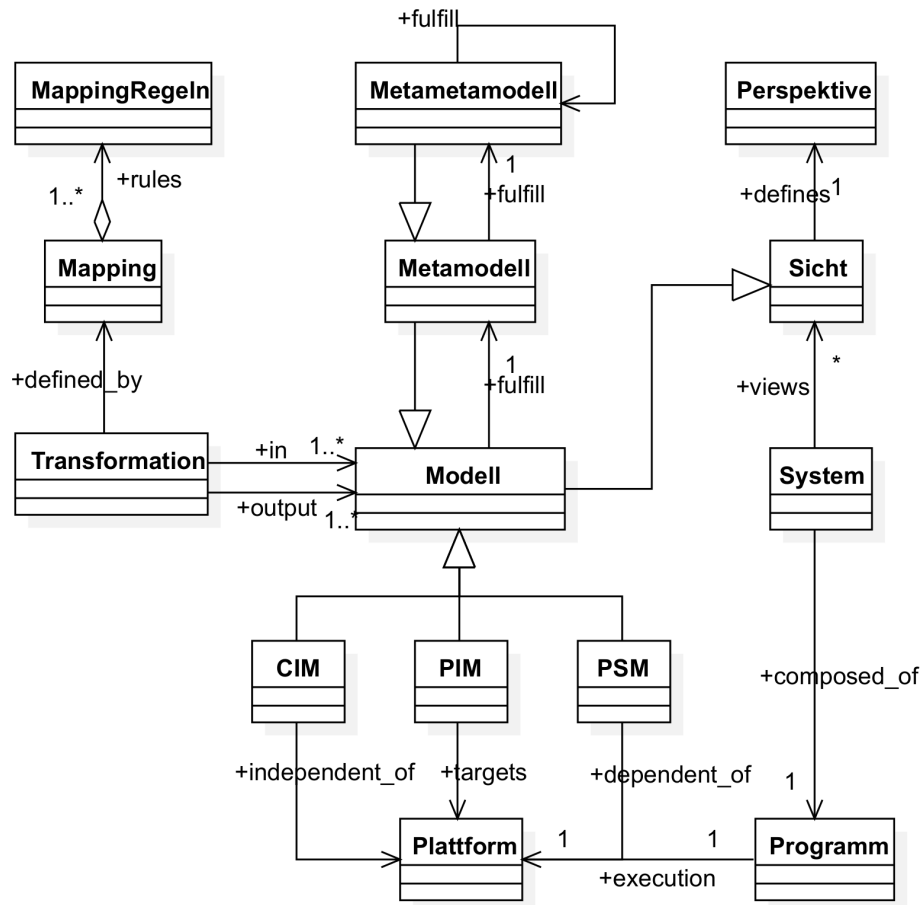


Abbildung 3.2.: Metamodell der MDA nach OMG (2014)

Es umfasst die Abstraktionsebenen des *computational independent model* (**CIM**), des *platform independent model* (**PIM**) und des *platform specific model* (**PSM**). Diese Unterteilung ist wichtig, da sie die Unabhängigkeit von der verwendeten Ausführungsplattform schafft. Alle Modelle werden jeweils in das nächste automatisch oder semi-automatisch überführt und sind ausführlich in Gläke u. a. (2017) beschrieben, als Teil der darin fachlichen Außenansicht für die MARS DSL Modellierer. Im Kern beschreibt das Metamodell ein *System* (übergreifend zu

mehreren Rechnern) aus mehreren beteiligten Modellen die alle miteinander in Beziehung stehen. Ein *System* enthält verschiedene *Sichten* darauf, die alle von der gewählten *Perspektive* oder *Gesichtspunkt* bestimmt werden. In Anlehnung an die OMG (OMG (2014)) ist die Perspektive eine Abstraktionstechnik die eine ausgewählte Menge architektonischer *Konzepte* und *Strukturierungs-Regeln* enthält, immer um eine besondere *Angelegenheit* im System zu genauer zu betrachten. Alles in allem werden diese einer *Plattform* gesammelt, die die Funktionalität für das entworfene *System* und seine Anforderungen am besten realisiert.

3.2.2. Modelltransformation

Die Lösung der Interoperabilitätsprobleme zwischen Modellen und zur Überbrückung der verschiedenen Abstraktionsebenen ist die Modelltransformationen. Sie stellt ein wichtiges Mittel im MDD-Ansatz dar und bildet nach Kleppe u. a. (2003) eine automatische Generierung eines Zielmodells aus einem Quellmodell. Sie lässt sich als Abbildung einer Menge von Modellen auf eine andere Menge von Modellen oder auf sich selbst betrachten und definiert hierfür ein Mapping, das dabei Entsprechungen zwischen Elementen oder Konzepten im Quell- und Zielmodellen umfasst. Dies geschieht über sogenannte Zuordnungsregeln, die beschreiben, wie ein oder mehrere Konstrukte in der Ausgangssprache in ein oder mehrere Konstrukte in der Zielsprache transformiert werden.

Die automatische Generierung eines Zielmodells aus einem Quellmodell kann entweder horizontal oder vertikal erfolgen, wobei horizontal bedeutet, eine Abbildung und Synchronisation von Modellen auf der gleichen Abstraktionsebene zu definieren, PIM zu PIM oder PSM zu PSM Transformationen, während vertikal beutet untergeordnete Modelle aus übergeordneten Modellen zu erzeugen, d.h. PIM- zu PSM-Transformationen. Selbst wenn horizontale und vertikale Transformationen unterschiedlich sind, kombiniert eine vollständige Transformationskette normalerweise beides, da in einem ersten Schritt vor der endgültigen Übersetzung ein oder mehrere horizontale und vertikale Transformationsschritte ausgeführt werden können, die optional mit manuell geschriebenem Code zusammengeführt werden. Das kann zum Beispiel die Zusammenlegung mehrerer Modellteile sein oder die automatische Anwendung von Optimierungen und Lösungen auf dem Modellcode. Ein Grund, warum die Erweiterung des generierten Codes mit manuell geschriebenem Code notwendig sein kann, ist, dass keine Transformation weder perfekt noch vollständig ist und man ggf. Dinge manuell ändern möchte. Ein zweiter Grund ist darin begründet, dass das Zielmodell im Vergleich zum Quellmodell aussagekräftiger ist. Der Entwickler muss also zusätzliche Details hinzufügen, die im Quellmodell nicht ausgedrückt werden können.

Neben der Unterscheidung zwischen horizontalen und vertikalen Transformationen (Mohagheghi u. a. (2013)) wird auch zwischen endogenen und exogenen Transformationen unterschieden. Endogene Transformationen sind Transformationen zwischen Modellen, die im selben Metamodell ausgedrückt werden. Endogene Transformationen sind zwischen verschiedenen Metamodellen definiert. Weitere Eigenschaften von Modelltransformationen (z.B. Automatisierungsgrad, Komplexität und Erhaltung) werden in Sarjoughian und Markid (2012) und Belaunde u. a. (2003) diskutiert. In Übereinstimmung mit Clark u. a. (2015) können Techniken zur Transformation von Modellen in drei verschiedene Arten eingeteilt werden: Transformationen von Modell zu Text wird verwendet, um Modellierungssprachen zu implementieren. Transformationen von Text zu Modellen werden verwendet, um Modelle aus (Alt-) Code zu extrahieren, häufig in Verbindung mit Reverse Engineering. Transformationen von Modellen zu Modellen werden verwendet, um Modelle zu refaktorisieren, Modelle in eine neue Modellierungssprache zu migrieren oder Modelle höherer Ebenen den Modellen auf niedrigerer Ebene zuzuordnen. Für alle drei Arten von Transformationen werden Regeln und Anwendungsstrategien in einer speziellen Sprache formuliert, der Transformationsspezifikationsprache, die entweder grafisch oder textuell sein kann OMG (2014). Die Sprache verfolgt dabei einen bestimmten Ansatz und kann demnach auch eingeteilt werden in welche Richtung die Transformation möglich ist und ob die Beschreibung deklarativ oder mehr prozedural vorgenommen wird. Mohagheghi u. a. (2013) differenziert diese nach:

- *Direkte Manipulation*: Beschreibt den einfachsten Transformationsansatz und bietet dem Benutzer nur eine beschränkte Unterstützung zur Implementierung. Häufig muss der Anwender dazu eine Abbildung in der jeweiligen Sprache aus der Zielinstanz schreiben z.B. direkt mit C# (Hejlsberg u. a. (2003)) oder Python (Rossum (1995))
- *Struktur-getriebene Transformation*: Die Sprachen dieses Ansatzes bestehen aus zwei Phasen. Zunächst wird die Struktur der Ausgabemodelle definiert, während anschließend der Inhalt des Quellmodells extrahiert und an den bestimmten Ort innerhalb des Zielmodells platziert wird. Ein Beispiel dafür wäre das Model Transformation Framework (MTF)
- *Operationeller Ansatz*: Dieser Ansatz von Sprachen ist ähnlich der direkten Manipulation. Kernmerkmal ist die zusätzliche operationelle Unterstützung, wie zum Beispiel die Abfragesprache der Object Constraint Language (OCL)

- *Vorlagen-basierter Ansatz*: Dieser Ansatz nutzt namensgebend definierte Vorlagen zur Erzeugung des Zielmodells die zur Instanziierung, insbesondere für spezifischere Formate, eingesetzt wird, z.B. mittels MOF Skripten (OMG (2014))
- *Relationaler Ansatz*: Erlaubt die Definition von Relationen zwischen Elementen von einem oder mehreren Metamodellen. Normalerweise sind Relationen bidirektional definiert, womit eine Rücktransformation in ein Quellmodell möglich ist und gar Roundtrip engineering zur Prüfung bspw. festgelegter architektonischer Entscheidung im Zielsystem, nicht ausgeschlossen wird
- *Graphbasierte Transformation*: Dieser Ansatz nutzt typisierte und mit Bezeichner versehene Graphen um darauf entweder Graphersetzungsregeln z.B mittels Grammatiken vorzunehmen, neue Graphen zu erzeugen oder bestehende zu ergänzen. Details zur Graph Transformation findet sich in Ehrig u. a. (2008)
- *Hybrider Ansatz*: Beschreibt einen Ansatz der mehrere verschiedene Ansätze kombiniert
- *Andere Varianten*: Umfasst alle Sprachen die keiner der oben beschriebenen Ansätzen entspricht. Ein Beispiel ist die häufig für Objektstrukturen und Serialisierung verwendet Extensible Stylesheet Language Transformation (XSLT) Transformation (Voelter u. a. (2013))

4. Analyse

4.1. Sichten auf ein Multiagentensystem

Die abstrakte Syntax der MARS DSL wird durch das MARS-Metamodell (Glake u. a. (2017)) definiert. Zur Weiterentwicklung dieses Modells ist dies in mehrere Sichten strukturiert, die sich jeweils auf einen bestimmten MAS-Gesichtspunkt beziehen. Durch die Gruppierung auf diese Weise, wird die Entwicklung weiterer Metamodelle durch Hinzufügen neuer Modellierungskonzepte, der Erweiterung bestehender oder Definition zusätzlicher Sichten, ermöglicht. In Bezug auf MDA werden komplexe Systeme immer aus verschiedenen Perspektiven betrachtet, und ihre Trennung in verschiedene Ansichten ist ein wirksames Mittel, um die Komplexität zu reduzieren und ihre Implementierung zu meistern. Die Bildung von Sichten erscheint als ein leistungsfähiges Mittel um die Komplexität des Systems anzugehen und das Fachwissen der Teilnehmer zu organisieren. Darüber hinaus stellt es ein Mittel zur Verfügung, um jeden spezifischen Fokus darzustellen und zu unterstützen und diese auf Modelle zu kombinieren. Diese Idee ist nicht neu und wird bereits in Starke (2015), Lind (2001) und Al-Zinati u. a. (2013) angewendet.

So werden in Lind (2001) zum Beispiel zwischen sieben Sichten unterschieden, die für die Modellierung von MAS notwendig sind. Dies ist die Systemansicht, die Umgebungsansicht, die Rollenansicht, die Interaktionsansicht, die Societysicht, die Architekturansicht und die Aufgabenansicht. Gleiches gilt für Starke (2015), dessen Sichten Vorschlag vornehmlich in Abschnitt 5.1 Verwendung finden. Für MARS werden nachfolgend unterschiedliche Perspektiven vorgeschlagen, die die Kernbausteine der MARS DSL bilden:

Multiagentensicht enthält die Kernbausteine zur Beschreibung von MAS Modellen. Insbesondere die Agenten die sich im System befinden, die Arten von Verhaltensweisen, die reaktiv und proaktiv agieren, und die Arten von Interaktionen, die für die Koordination mit anderen Agenten benötigt werden, sowie die verschiedenen Layer aus denen die Gesamtumgebung gebildet wird.

Entitätssicht definiert, wie einzelne autonome Entitäten modelliert werden können, um dadurch zugeordnete Aufgaben und Rollen innerhalb des betrachteten Modells zu lösen.

Des Weiteren definiert sie die Sicht des individuellen Agenten, auf die für ihn zur Verfügung stehenden Ressourcen mit der Frage, ob er Zugriff darauf erhält und welches Zugriffsverhalten er einsetzen kann, um seine Aufgaben zu lösen - entweder auf reaktive oder proaktive Weise. Überdies zählen auch Gruppierung (Organisationen) dazu, die im System existieren und die einzelnen Individuen zugeordnet sind, inklusive Rolle, den Aufgaben die damit verbunden sind, sowie verfügbaren temporalen, spatialen Ressourcen und die darauf definierten Operationen die von den Agenten gemeinsam geteilt werden. Darüber hinaus legt sie fest, auf welche Agentengruppen diese Zugriffe erhalten, sprich welche darunterliegenden Schichten sie überhaupt wahrnehmen können.

Interaktionssicht konzentriert sich auf den Austausch von Nachrichten zwischen autonomen Einheiten oder Gruppierungen. Dabei werden zwei Möglichkeiten geboten: Zum einen können notwendige Interaktionen heraus aus der Perspektive jeder einzelnen Entität beschrieben werden, zum anderen aus einer globalen Perspektive heraus in Bezug zu definierten Protokollen.

Verhaltenssicht beschreibt wie das interne Verhalten von Agenten definiert werden kann, wenn Aktionen mit Kontrollstrukturen oder reaktiven Aktionen kombiniert werden, um definierte Ziele zu erreichen. Die Verhaltensperspektive enthält grundlegende Konzepte aus bekannten Kontrollstrukturelementen sowie bestimmte maßgeschneiderte Konzepte zur Beschreibung von individuell agierenden Agentenprozessen.

Umgebungssicht enthält jede Art von Entität, die sich in der Umgebung befindet, und die Ressourcen und Ziele, die zwischen Agenten oder Gruppierungen geteilt werden. Die Kernumgebung befasst sich hauptsächlich mit der Definition von mehreren Schichten von eingegebene Datenlayern und den darauf befindlichen Objekten hinsichtlich ihrer Attribute und Operationen. Die Gesamtheit aus allen Datenlayern bildet die Kernumgebung und damit die abstrakte Sicht auf das Gesamtsystem, entgegen der individuellen Betrachtung eines einzelne Layer.

Über die verschiedenen Sichten wird die MARS DSL im Folgenden strukturiert und teilt dementsprechend auch seine Konzepte ein.

4.2. Konzeptmodell der MARS DSL

Grundlage jedes Sprachmerkmals ist das globale Beschreibungsmodell, in dem alle Sprach-elemente zu finden sind, die sich nicht auf die MARS-Base DSL und deren Ausdrücke, Kontrollstrukturen und Co. beziehen. Die Elemente der Sprache lassen sich in typisierte Elemente

und nicht typisierte Elemente unterscheiden und umfassen alle Konzepte die vom Modellierer ausgehend vom propagierten MDA in [Glake u. a. \(2017\)](#) im PIM zu finden sind. Das Modell wurde bereits detailliert in [Glake \(2016\)](#) und erweiternd in [Glake \(2017\)](#) beschrieben. Neue Konzepte wie das allgemein bekannte Klassenkonzept, die Einbindung von spatialen Rasterdaten und eine regelbasierte Verhaltensbeschreibung wird im nachfolgendem Abschnitt näher beschrieben. [Abbildung 4.3](#) zeigt das aktuelle Beschreibungsmodell, angereichert um diese Konzepte sowie der Erweiterung der Sprache zur Aufteilung des Modells in mehrere Dateien und damit der Entkopplung und separaten Speicherung bestimmter Elemente. Wie in [Abschnitt 4](#) bereits angedeutet wird mit der Aufteilung des Modells auf unterschiedliche Modelle auch auf Kritikpunkte der NetLogo Sprache ([Wilensky \(2015\)](#)) eingegangen. Die Einführung einer neuen regelbasierten Beschreibung ist in diesem Zuge neu und verfolgt einen neuen Ansatz wie er bereits von der Sprache ([Ahlbrecht u. a. \(2016\)](#)) und in [Aschermann u. a. \(2016b\)](#) versucht wird, jedoch mit dem Unterschied, dass dies eine rein regelbasierte Inferenz ist und Regeln deklarativ ohne gewisse Ordnung evaluiert werden ([Russell und Norvig \(2002\)](#)). Auch die Einbindung von spatialen Rasterdaten ist neu und geht auf die GIS-Erweiterung in Netlogo ([Russell und Wilensky \(2008\)](#)) bzw. die GAMA Plattform zurück ([Grignard u. a. \(2013\)](#)).

Das Modell aus [Glake \(2016\)](#) wurde ergänzt um zusätzliche Elemente zur Refaktorisierung von Aktionen innerhalb eines Agenten, aber auch zur Definition spatialer datenabhängiger Aktionen über das Konzept des *RasterLayer*. Diese Konstruktion orientiert sich an dem Konzept des Zeitreihen-Layer bietet die Möglichkeit zur Eingabe von Rasterdaten in das Modell. Dem Layer Ansatz folgend umfasst es *Methoden* und *State*. Aufgrund des gemeinsamen Speichers der im Simulationslauf eingesetzt wird, liegt damit aber auch ein Synchronisationspunkt vor, dessen Schreib- und Lesezugriffe geordnet werden müssen. Für diesen Systemaspekt wird daher eine einfache Sperre verwendet mit Verweis auf die Spezialisierung des *LockState*. Von außen bleibt die Definition eines Zustands für einen Layer aber der gleiche wie jene für einen Agenten, jedoch mit der Einschränkung und einer Warnung, dass diese nicht initialisiert oder observiert werden können. Dies kann jedoch in Zukunft durch eine realisierte Layerparametrisierung zum einen für den *RasterLayer*, als auch *TimeSeries* und dem gewöhnlich *Agenten-Layer* ermöglicht werden.

Hinzugekommen ist zuzüglich zum einen ein bewährtes Klassenkonzept, bekannt aus anderen Sprache ([Bettini \(2016\)](#)) als auch die Handhabung zum Umgang mit mehreren Ressourcen, zur Aufteilung des eigenen Modells. Ein solches ist nun als *Resource* verwaltet, das das Modell mit einer Datei verknüpft und in der globalen Umgebung *ResourceSet* für alle anderen bereithält. Entwurfs bedingt existiert eine Zusatzbeschreibung die alle nach außen sichtbaren Elemente {Agent, Layer, Objekt, ?} als Indexeintrag aufbereitet und diese Datenbeschreibung *MarsDslObjectDescription* nach außen veröffentlicht. Gekapselt wird diese durch die *ResourceDescription* die diese Elemente und deren Rückführung zum konkreten Modellobjekt kennt.

4.3. Abstrakte Syntax und Sprachmerkmale

Eines der wesentlichen Arbeiten dieser Sprache liegt in der Entwicklung der Sprachkonzepte und der Merkmale sowie dazugehöriger Semantik und der Transformation in das erhaltende Zielsystem. Dieser Abschnitt präsentiert die Syntax und Semantik der MARS DSL. Die plattformunabhängigen Konzepte, deren Attribute und formale Invarianten werden diskutiert, und die aus Forschungsperspektive zur Förderung der Entwicklung von MARS-Modellen getroffenen Entscheidungen begründet. Wie bereits in [Lind \(2001\)](#) ist mit großer Wahrscheinlichkeit eines der größten Schwierigkeiten in der Entwicklung einer plattformunabhängigen Modellierungssprache, die Definition der verwendbaren Konzepte, genauso wie deren Verwendungsausmaße die das Modell innehat. Die Konzepte sollten daher nicht zu spezifisch sein, um nur lediglich einen kleinen Teil an möglichen Modellen abzudecken, allerdings auch nicht zu allgemein gefasst, um somit nicht verwandte Details der Modellierungsmethodiken

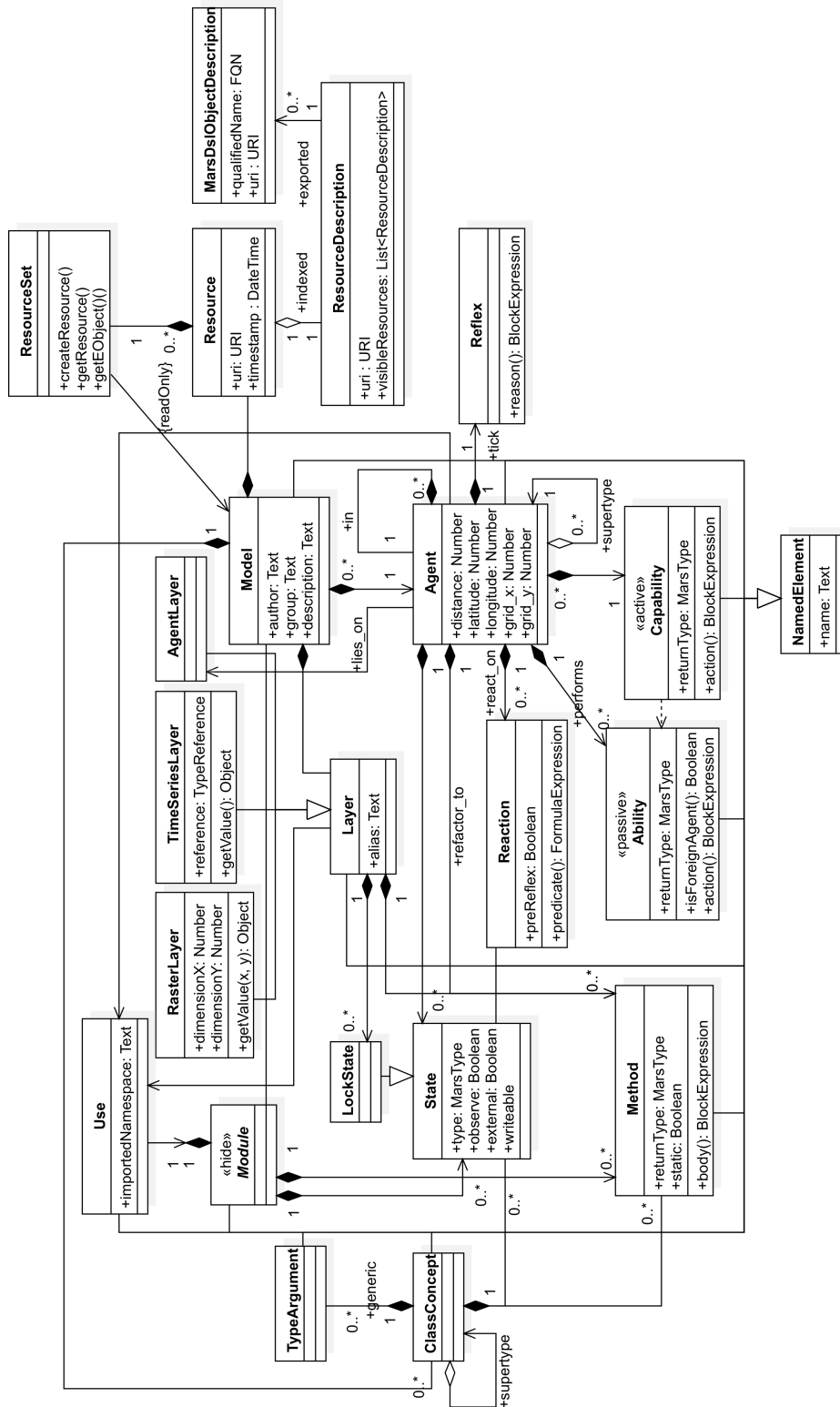


Abbildung 4.1.: Beschreibungsmodell der Sprache, angereichert um das Klassenkonzept, eines Ressourcenmanagement einzelner Modellteile und der Möglichkeit zur Handhabung spatialer Daten

im spezifischen Kontext weniger zu nützlich machen. Aus diesem Grund sind die Konstrukte plattformunabhängig und basieren auf allgemeinen Konzepten die für jedes MAS und seine Modelle relevant sind, problemspezifische Konzepte werden dabei möglichst vermieden. Die in Abschnitt 2 aus den Bereichen Agenten-basierte Softwareentwicklung, verteilter KI sowie der Kognitionswissenschaft analysierten Konzepte, begründen die Entscheidungen und stellen zugleich die weitläufige Meinung bezüglich der Auswahl dar. Ergänzend sei auf die bisherigen Arbeiten in Glake u. a. (2017), Glake (2016) und Glake (2017) hingewiesen, die neben der Aufbereitung neuer Elemente nun um eine Definition, inklusive geltender formeller und informell spezifizierter Invarianten in 4.5, erweitert werden.

4.3.1. Multiagentensystem Sicht

Ein MAS besteht in Übereinstimmung mit der Definition in Wooldridge und Jennings (1995) und Lind (2001) aus einer Sammlung von autonomen Agenten, die möglicherweise in einer dynamischen und unsicheren Umgebung angesiedelt sind, dazu in der Lage, über den Aufbau flexibler Organisationsstrukturen, über unterschiedliche Arten Interaktionen miteinander zu haben. Um diese Anforderungen aus technischer Sicht zu erfüllen, wird die Perspektive des Multiagentensystems selbst eingeführt. Diese erlaubt es MAS auf einer sehr abstrakten Ebene zu definieren, indem es die Kernbausteine der Agenten, der Layer und der Objekte einführt, die zum präzisen Entwerfen und Implementieren von MARS-Modellen notwendig sind (Glake u. a. (2017)).

Modell

Alle Elemente sind Teil einer Modelllösung zu einer spezifischen Domäne (z. B. ökologischen Modellierung, Medizin, Kunst, Verkehr, ?) (Wilensky (2015)). Damit sind sie Teil eines Gesamtmodells und auch Teil eines referenzierbaren Namensraumes sowie dazugehöriger Ressourcen. In der MARS DSL wird das vollständige Modell eines oder mehrerer Namensräume betrachtet. Abstrakt ist diese MARS DSL Modellbeschreibung nur wieder ein eigenes System mit Eingabe und Ausgabe und kann demnach auch so beschrieben werden (Aschermann u. a. (2016b)). Das Modell ist in der Lage andere Namensräume zu importieren und damit Zugriff auf deren Ressourcen zu erhalten. In 4.1 ist die abstrakte Syntax eines MARS DSL Modells definiert:

Definition 4.1 Multiagentensystem in MARS DSL

Ein Multiagentensystem wird definiert als Quintupel $A = (name, agents, layers, instances, objects, environment, imports)$, für den gilt:

- *name*: Definiert den Namen des Modell
- *agents*: Repräsentiert alle verschiedenartigen Agententypen die sich im MAS befinden
- *layers*: Repräsentiert alle verschiedenartigen Agenten-Layertypen und Umgebungen die sich im MAS befinden
- *tsLayers*: Repräsentiert alle Zeitreihen-Layertypen zur Darstellung von Zeitreihen im MAS
- *rasterLayers*: Repräsentiert alle Raster-Layertypen zur Darstellung spatialer Rasterdaten im MAS
- *objects*: Repräsentiert alle verschiedenartigen Objekttypen die sich im MAS befinden
- *uses*: Repräsentiert die Menge von anderweitig verwendeten Modellen

Definition 4.1 spezifiziert die abstrakte Syntax des MAS und stellt somit die minimalen Anforderungen dar, um den Definitionen aus [Wooldridge und Jennings \(1995\)](#) und [Galland u. a. \(2015\)](#) zu entsprechen. Das Multiagentensystem enthält Agenten, die autonom agieren und auf Aktionen, zur Lösung bestimmter Probleme, zugreifen. Darüber hinaus enthält es eine Reihe von Umgebungen, die eine grundlegende Infrastruktur bereitstellen, um Berechnungs- und Integrationsmittel für die Kommunikation und des Konzepts der Interaktion ([Caillou u. a. \(2017\)](#)) zu ermöglichen. Agenten sitzen dazu auf einem Layer ([Hüning u. a. \(2016\)](#)) der wiederum eine definierte Sicht auf eine dahinterliegende global diskrete Umgebung darstellt, worin sich Agenten gleicher Ebene 4.3.2 und zugleich Objekte einsetzen und wahrnehmen lassen. Der Grund zur Wahl einer Umgebung liegt in der breiten Verwendung und einfacheren Handhabung für verschiedene Modelltypen ([Cich u. a. \(2016\)](#)) und die Vielseitigkeit seitens der Datenstrukturen, diese zu implementieren. Durch Invarianten zur Beibehaltung der Modellstruktur ([Aschermann u. a. \(2016b\)](#)) und Übersetzbarkeit ([Aho \(2008\)](#)) wird letztlich sichergestellt, dass das beschriebene Modell auch ausführbar bleibt. Diese sind in Abschnitt 4.5 definiert, bei der versucht wurde einige essenzielle Sprachkonzepte formal zu spezifizieren.

Use

Zur Aufteilung des eigenen Modells in mehrere Modellbestandteile und zur Erhöhung der Wiederverwendbarkeit verschiedener Elemente ([Barišić u. a. \(2011\)](#)) wurde das *use* Konzept eingeführt, das einen Importmechanismus, als Mittel zur Verknüpfung unterschiedlicher Modellteilen aus voneinander unterschiedlichen Ressourcen, darstellt. Ein *use* kann dabei entweder auf den Import eines definierten Layers mit Deklaration von einem lokal geltenden

zugreifbaren Alias genutzt werden, ähnlich einem Modulimport der Sprache Python (Rossum (1995)), oder er stellt einen herkömmlichen Import dar, bei dem zwischen dateibasiertem- oder namens-gebundenem (Namensraum) Import differenziert wird (Bettini (2016)). Das MARS DSL *use* kombinierte beide Varianten, orientiert sich hauptsächlich aber an letzteres und ist definiert in 4.2:

Definition 4.2 Use in MARS DSL

Ein Use wird definiert als Tripel $A = (model, layer, alias)$, für den gilt:

- *model*: Definiert ein Modell was zur aktuellen Ressource hinzugefügt werden soll
- *layer*: Spezifiziert einen Layer der aus anderen Ressourcen als Referenz zur lokalen Ressource hinzugefügt wird
- *alias*: Spezifiziert einen, für den importierten Layer verwendeten Namen, über den referenziert werden kann

Der *use* dient der Sichtbarmachung anderer Modellelemente, insbesondere anderer Layer, um damit Entitäten 4.3.2 aus unterschiedlichen Ressourcen lokal zur Verfügung zu stellen. Ein importierter Layer bildet dabei die Besonderheit dass, eine solches Element nicht als Instanz selbst erzeugt wird, sondern zur Laufzeit, automatisch in die Agenten als Abhängigkeit hinzugefügt wird und aus diesem Grund nur die Bedarf des Layers in der aktuellen Ressource definiert werden muss.

4.3.2. Entitätssicht

Unter eigenständig typisierenden Elementen werden die Konzepte zusammengefasst, dessen Definition von selbst einen Verweis-Typen im Modell etablieren und sich damit innerhalb unterschiedlicher Modellaspekte referenzieren lassen. Sie bilden die Grundlage der Invarianten die auf der statischen Semantik geprüft werden 4.5 und elementare Inferenzregeln innerhalb des Typsystems darstellen.

Agent

Kernbaustein der Bildung von Multiagentensystemen bildet kennzeichnend der Agent als autonome Einheit, mit der Fähigkeit innerhalb einer Umgebung zu agieren. Der Agent hat einen beschränkten Zugriff auf Ressourcen die ihn umgeben, und ist mit seiner Lebenszeit an eine globale Uhr gebunden sowie der Existenz an einen gemeinsamen (Agenten-)Layer. Dieser steuert die Sichtbarkeit auf andere Agenten anderen Typs die um einen weiteren Grad vertieft

werden, wenn dieser in einer *rekursiven* Beziehung zu einem anderen Agenten steht. Als Teil eines Ganzen wird einem Agenten damit erlaubt sich an die Existenz eines anderen Agenten zu binden, in dessen inneren Agentenkontextes er agieren kann. D.h. sobald der direkt oder transitiv verknüpfte Agent aus dem System entfernt wird, scheiden auch alle darunterliegenden Agenten aus dem System. Das Konzept der holonischen Agenten ist dabei nicht neu, sondern bereits bewährt für die Entwicklung diverser komplexer Systeme eingesetzt worden, von generellen MAS [Gerber u. a. \(1999\)](#) über die Erstellung von Differentialdiagnosen in der Medizin ([Giret u. a. \(2017\)](#)) bis hin zu Kontrollsystemen intelligenter Fertigungssysteme ([Cossentino u. a. \(2008\)](#)). Ein sinnvolles Modellbeispiel findet sich auch sofort in der Medizin wieder: Würde man versuchen ein realistisches Modell von einem menschlichen Organismus zu entwerfen, müsste man im Zuge dessen auch Zellbeziehungen und Prozesse innerhalb des menschlichen Körpers abbilden. Durch diese rekursive Beziehung wären diese Zellen vollständig abhängig von der Existenz des darüber liegenden Lebewesens. Die Sichtbarkeit der tieferliegenden Agenten ist dabei beschränkt auf jene der gleichen Ebene bzw. des direkt verknüpften holonischen Agenten, der auch immer als einzige Instanz wahrgenommen werden kann. Folglich gilt für das jeweilige Environment, dass sich diese Agenten nicht mehr im gleichen globalen Environment befinden, sondern in einem eigenen innerhalb des Agenten. Die Kontextseparierung und Aufteilung der Umgebung in Submodelle wird dabei näher durch Abbildung [4.2](#) erläutert. Ein Agent wird dabei in die Hierarchie des Agentenkontext eines identifizierten Agents eingegliedert und kann darin nur innerhalb dieses beschränkten Nachrichtenraums mit anderen Agenten interagieren ([Giret u. a. \(2017\)](#)). Die abstrakte Syntax des Agenten-Konzepts ist in Definition [4.3](#) gegeben:

Definition 4.3 Agent in MARS DSL

Ein Agent wird definiert als 9-Tupel $A = (name, layer, superType, in, active, passive, states, reflex, reactions)$, für den gilt:

- *name*: Definiert den Namen des Agenten
- *layer*: Gibt den Agenten-Layer an auf dem der Agent sitzt
- *superType*: Optionaler Obertyp der den Agententyp in eine Ordnung überführt [4.3.2](#)
- *in*: Gibt einem ein Prädikat zur Identifizierung eines optionalen rekursiven Agenten an, in dessen Kontext sich dieser Agent bewegt und von dem dieser abhängig ist
- *active*: Spezifiziert die proaktiven Aktionen zur Durchführung einer Interaktion mit anderen Agenten [4.3.3](#)

- *passive*: Spezifiziert die reaktiven Aktionen an, die innerhalb einer Interaktion mit anderen Agenten zum Nachrichtempfang eingesetzt werden [4.3.3](#)
- *states*: Repräsentiert die eigene Wissensbasis, aus der zur Zielerfüllung die späteren Schlussfolgerungen gezogen werden, genauso wie Agenteneigenschaften die analysiert und extern parametrisiert werden können [4.3.3](#)
- *reflex*: Definiert eine elementare oder komplexe Agentenaktion, die explizit von der globalen Uhr angestoßen wird. [4.3.4](#)
- *reactions*: Repräsentiert die Inferenzregeln die aus dem eigenen knowledge und der sichtbaren Faktenbasis Schlussfolgerungen zieht und jeweils eine Agentenaktion definiert [4.3.4](#)

Die Definition [4.3](#) des Agenten bildet mit einer Menge von passiven und aktiven Aktionen seinen Interaktionskontext und macht Gebrauch einer *reflex* Aktion, gesteuert durch die globale Uhr um daraus eine oder mehrere Agentenaktionen durchzuführen. Ergänzend dazu lässt sich durch eine Reihe von *reaction* Konstrukten eine regelbasierte Steuerung mittels Inferenz über die Faktenbasis, des Multiagentensystem selbst, vornehmen. Abschließend liegt mit dem Vererbungskonzept ein bekanntes Designkonzept vor, abgeleitet vom Klassenkonzept [4.3.2](#), die damit auch die Typisierung des Agenten in eine Ordnung überführen. Neben dem primitiven Agentenname zur voll qualifizierten Identifizierung innerhalb des Gesamtmodells, existiert eine Referenz auf den zugrundeliegenden Agenten- *Layer*. Das *states* stellt schließlich auch privates Wissen bereit, die Informationen darstellen, die ein Agent über die Welt haben könnte, und die unter anderem dazu verwendet werden Entscheidungen zu treffen.

Um eine Laufzeitebene dahingehend zu unterstützen, wird an dieser Stelle eine bestimmte Laufzeit-Entität namens *AgentInstance*, zur Repräsentation eines bestimmten Agententyps zusätzlich definiert. Diese entstehen sobald ,das mit der MARS DSL erstellte Modell durch eine verwendete Programmiersprache ausgeführt wird. Die Definition ist in [4.4](#) gegeben.

Definition 4.4 *AgentInstance in MARS DSL*

Ein *AgentInstance* wird definiert als Quadrupel $A = (name, agentType, membersOf, members)$, für den gilt:

- *name*: Bildet den Namen der Agenteninstanz ab
- *agentType*: Spezifiziert den zugrundeliegenden Agententypen aus dem diese Instanz erzeugt wurde

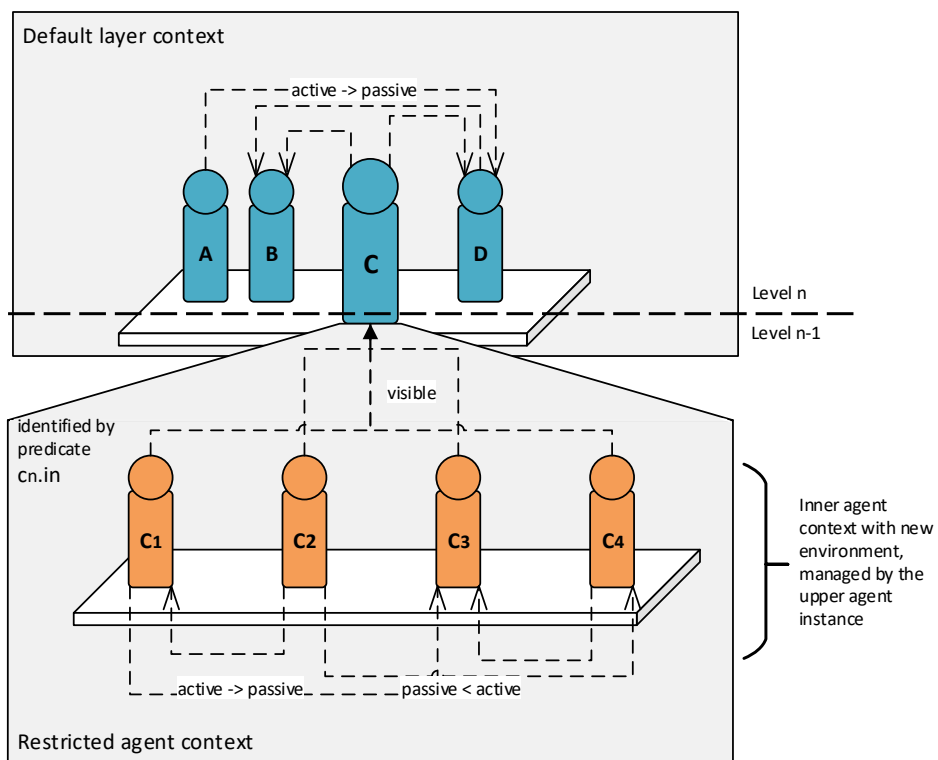


Abbildung 4.2.: Rekursivität von Agenten mit Einteilung in mehrere Ebenen und beschränkter Interaktion.

- *membersOf*: Gibt die Laufzeitinstanz an, zu der dieser Typ eine rekursive Beziehung besitzt und in dessen Kontext er lebt
- *members*: Spezifiziert jene Agenten Instanzen die sich aus der Typbindung des Agententypen zum Layer befinden

Eine Agenten Instanz ist zur Laufzeit an einen Layer gebunden, der die Sichtbarkeit auf andere Agenten innerhalb des MAS festlegt. Sollte die Instanz in einer rekursiven Beziehung stehen wird diese Sichtbarkeit nochmals beschränkt und bildet damit erneut eine Teilmenge dieser Layer Agenten.

Layer

Die Bereitstellung einer allgemeinen Modellierungsabstraktion und allgemeiner Modellierungstechnik für agentenbasierte Umgebungen ist eine sehr komplexe und schwierige Aufgabe. Der Hauptgrund ist, dass Umgebungen für verschiedene Anwendungen sehr unterschiedlich sein können, da sie irgendwie mit der zugrunde liegenden Technologie zusammenhängen (Zambonelli u. a. (2003)). Um den Definitionen in Wooldridge und Jennings (1995) zu entsprechen wird alles betrachtet was nicht direkt Teil eines Agenten oder einer Gruppe ist oder Teil der Umgebung. Daher kann eine solche Umgebung Objekte umfassen, die entweder von den Agenten verwendet werden können oder Informationen die von ihnen wahrgenommen werden können. Damit ein Metamodell für eine multiagenten-basierte Umgebung für die spezifische Domäne der ökologischen Modelle bereitgestellt werden kann, wird empfohlen eine solche Umgebung in Bezug zu öffentlich verfügbaren Ressourcen wie Objekte und Dienste, zu behandeln, die der Agentengesellschaft zur Verfügung stehen (Galland u. a. (2015)). Diese Ressourcen können von jeder berechtigten Entität, aufgerufen und verändert werden. Dazu zählen sowohl der Kernbaustein des Agenten-Layer, als auch die Dienste zur Bereitstellung von temporalen (*Time Series Layer*) und spatialen Daten (*Raster Layer*). Die abstrakte Syntax ist in 4.5 gegeben:

Definition 4.5 *Layer in MARS DSL*

Ein Agenten-Layer wird definiert als 6-Tupel $A = (name, alias, agents, dependencies, fields, methods)$, für den gilt:

- *name*: Definiert den Namen des Agenten-Layers
- *alias*: Ein spezifizierte Alias Name um einen Layer in der aktuellen Ressource über diesen Namen referenzieren zu können

- *agents*: Spezifiziert die Menge darauf agierender Agenten
- *dependencies*: Spezifiziert eine Ordnung auf sichtbare Layer, ausgehend von diesem
- *fields*: Stellt eine Menge von typisierten Datenfeldern dar, die zur Speicherung synchronisierter Interaktion (Ahlbrecht u. a. (2016)) oder einfachen Zwischenspeicherung verfügbar sind
- *methods*: Eine Menge von Operationen die der Layer zusätzlich zur Verfügung stellt

Ein solcher Agenten-Layer ist direkt Teil der Definition des Agententyps 4.3.2 selbst und damit auch nicht Teil der Umgebung. Es stellt stattdessen eine Gruppierung dar und erlaubt zum einen das Konzept der Daten-synchronisierten Interaktion für die Agenten, zum anderen legt es die Sichtbarkeit auf andere Agententypen fest die über die Bewegungslogik 4.3.5 und Explorationslogik 4.3.5 angesteuert, wahrgenommen und neu erzeugt werden können.

Die Invarianten umfassen: Globale Eindeutigkeit des Namens und lokale Eindeutigkeit eines optionalen Alias. Zudem gilt, dass die sichtbaren Layer keine zyklischen Abhängigkeiten aufweisen, sowohl direkt als auch transitiv. Die Invarianten der Felder ist äquivalent zu denen in Definition 4.9 beschrieben, während die Invarianten der Methoden äquivalent zu den in 4.12 sind.

Time Series Layer

Das Zeitreihen-Layer-Konzept der MARS DSL veranschaulicht jegliche Art von zeitbezogenen Informationen oder Ressourcen, auf die Agenten zugreifen und die sie verwenden können. Auch wenn die Kernumgebung nur die Abfragemöglichkeit von über den Durchschnitt aggregierten Daten bereitstellt, lassen sich dennoch Punktanfragen und eigene Aggregation auf parametrisierte Zeitreihen anwenden. Die Option dafür wird inhärent durch die Abfragesyntax selbst bzw. die Modellierung von daraus entstehenden Objekten und Informationen ausgerichtet. Durch den zusätzlichen Transformationsschritt zwischen der reinen Datenabfrage und der Weitergabe an den jeweiligen Agenten kann sie leicht auf die verschiedenen Arten von Anwendungseinstellungen angepasst werden. Die abstrakte Syntax von Environment wird in der folgenden Definition 4.6 angegeben:

Definition 4.6 Time Series Layer in MARS DSL

Ein Zeitreihen-Layer wird definiert als Quintupel $A = (\text{name}, \text{alias}, \text{resource}, \text{fields}, \text{operations})$, für den gilt:

- *name*: Definiert den Namen des Zeitreihen-Layers
- *alias*: Gibt den lokal identifizierbaren Namen des Layers an
- *resource*: Spezifiziert die Zeitreihe die für die Agenten zur Verfügung steht
- *fields*: Eine Menge von typisierten States die die Zeitreihe beschreibt und Werte zwischenspeichert
- *operations*: Eine Menge von Operationen die auf den Feldern angewendet oder zur Bildung eigener Abfragen auf die Zeitreihe angewendet werden können

Der Zeitreihen-Layer besitzt einen Namen sowie einen Alias über den er lokal innerhalb einer Ressource referenziert werden kann. Er kann Felder nach Definitionen 4.9 enthalten und Operationen die auf diesen Feldern ausgeführt werden. Die Frage, wie die interne Daten repräsentiert sind und welcher Form, ob äquidistant oder nicht sind an dieser Stelle unerheblich.

Als Invariante gilt insbesondere die in 4.5.1 und 4.5.1 spezifizierte Bedingung. Außerdem bilden sie die unterste Schicht im Layer Modell (Hüning u. a. (2016)) und stellen damit eine Abhängigkeit zu den oberen Agenten-Layer dar.

Raster Layer

Als eines der wesentlichen Datenelemente bildet das Raster-Layer Konzept ein Gerüst zur Integration spatialer Daten in das Modell. Ein von *außen* parametrisierter, *standardkonformer Raster* wird innerhalb des Modells zur Verfügung gestellt und durch die Abfragesyntax des Grid (Glake (2017)) bereitgestellt. Sowohl Datenabfragen als auch, entgegen der Zeitreihen-Layer 4.3.2, mögliche Veränderungen sind auf dem *Datenlayer* anwendbar und in der spezifischen minimalen Schnittstelle mit Operation eingebettet, folglich auch nur innerhalb des *Raster-Layer-Konzepts* verwendbar. Zu diesen Operationen gehören {*getValue*, *increase* und *reduce*}, bei dem vor allem eine Projektion zur Datenabfrage (*getValue*) auf verschiedene Datentypen {*string*, *bool*, *integer*, *real*, *object*} möglich ist. Die Definition für den Raster-Layer ist äquivalent zu 4.6 und gegeben in 4.3.2:

Definition 4.7 Raster-Layer in MARS DSL

Ein Raster-Layer wird definiert als Quintupel $A = (name, alias, resource, fields, operations)$, für den gilt:

- *name*: Definiert den Namen des Raster-Layers

- *alias*: Gibt den lokal identifizierbaren Namen des Layers an
- *resource*: Spezifiziert die Rasterkarte die für die Agenten zur Verfügung steht
- *fields*: Eine Menge von typisierten Feldern die die Rasterkarte beschreibt und Werte zwischenspeichert
- *operations*: Eine Menge von Operationen die auf den Feldern und den Rastern angewendet werden können

Äquivalent dazu gelten auch die Invarianten in [4.3.2](#).

Objekt

Wie bereits erwähnt existieren Agenten nicht in reiner Isolation. Stattdessen interagieren Agenten normalerweise mit Objekten oder anderen Agenten um bestimmte Aufgaben und Ziele zu erfüllen. Objekte sind Teil dieser Umgebung sowie Teil möglicher Austauschformate und werden wie folgt definiert:

Definition 4.8 *Objekt in MARS DSL*

Ein Objekt ist definiert als Quintupel $O = (name, superType, typeArguments, fields, methods)$, für den gilt:

- *name*: Definiert den Namen des Objekttyps
- *superType*: Optionaler Obertyp der den Objekttyp in eine Ordnungsrelation überführt
- *typeArguments*: Optionale Menge von Typ-Argumenten zur generischen Typisierung von Eigenschaften und Methoden
- *fields*: Eine Menge von Feldern die das Objekt beschreibt und zur Verfügung stellt
- *methods*: Eine Menge von Operationen die auf den Feldern angewendet werden können

Die Definition ist angelehnt an das bekannte Klassenkonzept und umfasst neben den Attributen auch generische Typ-Argumente, die zur Abfragezeit über das Typsystem aufgelöst werden.

4.3.3. Interaktionssicht

State

Mentale Zustände werden in der MARS DSL verwendet um die in den Agenten enthaltenen Informationen zwischenspeichern, und auch zur Parametrisierung des Agentenmodells sowie zur Spezifikation der Ausgabe. Sie können als Teil der Layer zur daten-synchronisierten Interaktion eingesetzt werden und dazu neben dem Aktiv-Passiv Konzept (Glake u. a. (2017)) vor allem einen besonderen Nachrichtenaustausch, insbesondere um für rekursive Agenten zu dienen, die unter der Bedingung des gleichen zugrundeliegenden Layers an dieser Stelle Nachrichten an ihren Holon oder an andere Agenten einer höheren Ebene weitergeben können. Die abstrakte Syntax eines State's ist definiert in 4.9:

Definition 4.9 Agent in MARS DSL

Ein State wird definiert als 6-Tupel $A = (name, type, default, observe, external, writeable)$, für den gilt:

- *name*: Definiert den Namen des mentalen Zustands
- *type*: Spezifiziert einen expliziten Typ
- *default*: Spezifiziert einen Default Ausdruck der den Initialwert des Agenten angibt und von dem optional der Typ abgeleitet wird
- *observe*: Signalisiert, ob die festgehaltenen Zustände des Agenten raus geschrieben werden sollen
- *external*: Signalisiert, ob dies als Parameter aufgefasst werden soll und von außen mit einem Wert belegt wird
- *writeable*: Signalisiert, einen nicht veränderbaren Zustand, wenn dieser einmal zugewiesen wurde

Die mentalen Zustände sind ein wesentliches Konzept zur Zwischenspeicherung von Daten zur Simulationslaufzeit, wie auch zur externen Parametrisierung des Modells. Der *name* identifiziert das Element innerhalb des verwendeten Konzepts, während der *type* einen expliziten oder inferierten Typen darstellt. Die Invarianten die gelten müssen beziehen sich auf die Eindeutigkeit der Namen und der Typ Konformität bei Zuweisungen. Das *external* und das *observe* signalisieren, ob dieser Agententyp von außen parametrisiert werden kann, und ob die

Werte bei Veränderung mit einem Zeitstempel zur Analyse raus geschrieben werden sollen. Diese Konzepte gelten dabei nur innerhalb eines Agentenkontexts und können daher nicht auf selbst definierte Objekttypen 4.3.2 angewendet werden.

Aktive Aktion

Äquivalent zur 4.3.3 wurde auch die aktive Aktion ausgiebig in Glake u. a. (2017) diskutiert und wird daher ebenfalls darauf verweisen. Eine fehlende Definition ist in 4.10 gegeben:

Definition 4.10 Aktive Aktion in MARS DSL

Eine aktive Aktion wird definiert als Quadrupel $R = (name, containingAgent, messageParameters, returnType)$, für den gilt:

- *name*: Definiert den Namen der aktiven Aktion
- *activeActor*: Gibt den initiiierenden Akteur der Interaktion an, der diese Nachrichten sendet und empfängt
- *messageParameters*: Repräsentiert optional Daten die in der Interaktion verwendet werden sollen und an den Empfänger gesendet werden
- *returnType*: Repräsentiert optionale Nachrichtentypen die aus der Interaktion erfolgen und die aus der Interaktion heraus abgeleitet werden

Die aktiven und passiven Aktionen werden letztendlich entweder auf Methodenaufrufe oder das Auslösen von Ereignissen zurückgeführt. Wichtig ist das diese Aufrufe geordnet erfolgen und innerhalb einer aktiven-passiven Abbildung. Der daraus resultierende Interaktionsgraph zwischen den Agenten stellt eines der wesentlichen Elemente zur letztendlichen Übersetzung in ein konformes Zielsystem dar. Der Vorteil dieses Aktiv-Passiv Konzepts liegt in der endogenen Modellierung unterschiedlicher Interaktionsmuster. Im Anhang 7 finden sich hierzu Beispiele für Agenteninteraktionen *Einzelübertragung*-, *Mehrübertragungs-Interaktion*-, *bilateralen*-, *multilateralen-Interaktion* sowie *Routing-Muster*.

Passive Aktion

Die Fähigkeit zur Interaktion ist eines der Kerneigenschaften von Agenten und Agentengruppen (Wooldridge und Jennings (1995)). Zwei Typen von Interaktionen können hierbei unterschieden werden: eine Protokoll-basierte Interaktion versus einer flexiblen Interaktion, wobei die MARS DSL vermehrt auf Protokoll-basierte Interaktionen verweist. Das verwendete Vokabular zum

Ausdrücken von dieser ist definiert durch das Aktiv-Passiv Modell (Glake u. a. (2017)). Dabei wird unterschieden zwischen aktiven und passiven Aktionen, bei denen die aktiven den initiiierende Senderseiten entsprechen und passive Aktionen die der empfangenden Seiten pro Nachricht. Der Nachrichtenempfang ist dabei nachfolgend definiert in 4.11:

Definition 4.11 *Passive Aktion in MARS DSL*

Eine passive Aktion wird definiert als Quintupel $R = (name, actor, messageParameters, returnType, localKnowledge)$, für den gilt:

- *name*: Definiert den Namen der passiven Aktion
- *actor*: Gibt den Akteur an, der an einer von außen initiierten Interaktion und aus Empfängersicht am Protokoll teilnimmt
- *messageParameters*: Repräsentiert die Daten der Nachricht die an diesen Agenten übermittelt werden
- *returnType*: Repräsentiert den Nachrichtentyp der aus der Interaktion optional zurückkehren kann und automatisch abgeleitet wird
- *localKnowledge*: Gibt alle Parameter des empfangenden Agenten an, auf die global innerhalb der passiven Aktion zugegriffen werden kann

Die passive Aktion repräsentiert einen Empfangspunkt einer aktiven Aktion. Ein Nachrichtenaustausch findet allein dann statt, wenn ein Aufrufereignis emittiert wird. Der Aktionskontext wechselt dazu auf den Zielagenten *actor*, der diese Nachricht verarbeitet. Das Protokoll ist hierbei implizit über die Reihenfolge der Aufrufereignisse auf den passiven Aktionen festgelegt. Die Daten die mittels des Aufrufs versendet werden sind Teil der Parametrisierung der passiven Aktion. In Glake u. a. (2017) werden diese immer ausgehend von aktiven Aktionen ausgeführt und schränken dabei die Sichtbarkeit der Agenten auf ihre eigenen Elemente der aktiven Aktionen bzw. Methoden ein. Zu weiteren Details wird auf Glake u. a. (2017) verwiesen, da dort bereits ausgiebig dieses Konzept diskutiert wurde.

Methode

Spezifiziert ein Element zur Refaktorisierung der eigenen Modelllogik und wird äquivalent zu Methoden etablierten Programmiersprachen verwendet, mit der Ausnahme, dass der Rückgabebetyp automatisch aus der Programmlogik abgeleitet wird, sofern kein Typ explizit definiert wurde. Die Definition ist in 4.12 gegeben.

Definition 4.12 *Methode in MARS DSL*

Eine Methode wird definiert als Quintupel $R = (name, parameters, localKnowledge, returnType, workflow)$, für den gilt:

- *name*: Definiert den Namen der passiven Aktion
- *parameters*: Repräsentiert die Eingabeparameter die innerhalb der Methoden verarbeitet werden item *localKnowledge*: Repräsentiert die Menge der Information auf die global innerhalb der Methode zugegriffen werden kann
- *returnType*: Repräsentiert einen optional explizit definierten Typ, der andernfalls automatisch aus der Programmlogik abgeleitet wird
- *workflow*: Spezifiziert den Workflow der durch die Methode angestoßen wird

Der Rückgabetypp wird äquivalent zu 4.3.3 und 4.3.3 aus der Programmlogik abgeleitet und der kleinste gemeinsame Typ berechnet. Es gelten die gleichen Invarianten wie zur Methodendefinition in Hejlsberg u. a. (2003) und damit auch die Invariante von fehlender Mehrdeutigkeit. Formal spezifiziert sind diese näher in Abschnitt 4.5.

4.3.4. Verhaltenssicht

Wie in Definition von Wooldridge und Jennings (1995) vorgeschlagen, sind Verhaltenselemente von besonderer Bedeutung, um Agenten ein autonomes, reaktives, proaktives und soziales Verhalten zu erlauben. Um diese Anforderungen zu erfüllen, kombiniert die MARS DSL verschiedene Verhaltenselemente innerhalb einer Verhaltenssicht. Die Verhaltensperspektive wird in zwei Teile untergliedert: Der erste und der beschriebene Kernteil umfasst den Zeitschritt der in Kombination mit der MARS-Base DSL 4.3.5 und weiteren Workflow-Elementen das Verhalten anhand von einfachen Zeit- und Lokationsbedingungen knüpft. Darin fallen auch die modellierten Systemgleichungen, die nun von einem Agenten selbst zu einem Zeitschritt t_i berechnet und für die gesamte Simulation näherungsweise gelöst wird. Der zweite Teil konzentriert sich auf eine losgelöste regelbasierte Verhaltensbeschreibung, die unabhängig der Reihenfolge von Regeldefinitionen (Fowler (2010)) ein Verhalten in Richtung eines gewissen Ziels darstellt. Beide Teile können in komplexer oder atomarer Form vorliegen und bestimmen den Ablauf des individuellen Agentenverhaltens.

Reflex

Legt die elementaren oder komplexen Aktionen des Agenten fest, die mittels globaler Uhr pro Zeitschritt angestoßen wird. Der Reflex wird dabei immer durchgeführt und ist äquivalent zur einer Reaction deren Bedingung immer zu wahr evaluiert. Bedingt durch die globale Uhr wird ein Agent immer nur einmal pro Zeitschritt angestoßen, jedoch sind rekurrente Workflows mit über Schleifen oder rekursiven Aufrufen möglich und damit ggf. endlos. Die Definition des Reflex ist nachfolgend in 4.13 gegeben:

Definition 4.13 Reflex in MARS DSL

Ein Reflex wird definiert als Quadrupel $Re = (action, knowledge, reflex, reactions)$, für den gilt:

- *action: Eine elementare oder strukturierte Aktion bei der lebendige Agenten durch die globale Uhr angestoßen werden*
- *knowledge: Repräsentiert die eigene Wissensbasis, auf die global im Reflex zugegriffen werden kann und vom umgebenden Agenten bereitgestellt wird 4.3.3*
- *reflex: Repräsentiert eine Aktion die entweder elementar oder komplex ausfallen kann und durch die globale Uhr angestoßen wird 4.3.5*
- *reactions: Repräsentiert die Inferenzregeln die aus dem eigenen knowledge und der sichtbaren Faktenbasis Schlussfolgerungen zieht und jeweils eine Agentenaktion definiert 4.3.4*

Im Reflex *reflex* wird es in der Regel immer eine komplexe Aktion geben, in der einfache Verzweigungen und, in Abhängigkeit bestimmter Simulationszeitpunkte sowie der eigenen Agentenzustände, der interne Workflow angestoßen wird oder nicht. Eine konkrete Aktionsausführung ist hierbei unerheblich, da diese auch leer ausfallen kann. Ein sinnvoller Einsatz des *reflex* liegt jedoch in Abfolge des generell akzeptierten *Sense-Reason-Act* Zykluses (Russell und Norvig (2002)). Die eigenen sensorischen Agentenzustände sollen hierbei aktualisiert werden und später in den aktiven 4.3.3 und passiven 4.3.3 bzw. in den *reactions* zu einem unabhängigen Verhalten führen (Glake u. a. (2017)).

Reaction

Entgegen der proaktiven und sequentiellen Abarbeitung einzelner Aktionen aus dem Reflex 4.3.4 ist eine Reaction ein reaktives Element, das Mithilfe eines Prädikats über die Faktenbasis des Systems ausgelöst wird. Diese Aufrufbedingung (oder auslösendes Ereignis) definiert genauestens die Vorbedingung, d.h. einen globalen F die logische Implikation und bei G die

Konklusion (Aktion) darstellt (Ahlbrecht u. a. (2016)), die ausgeführt wird sobald die Prämisse erfüllt ist (Produktionsregeln). Ein aus der Aktion heraus entstehender Nachfolgezustand, der als Aktionsausführung gelten muss. Sie sind damit eine weitverbreitete Form der Wissensrepräsentation in Form von Konditionalsätzen. Wenn (*if*) F dann (*then*) G , bei ezustand kann dafür sorgen, dass in der Folge eine andere oder die gleiche Prämisse erfüllt ist und damit ein neuer oder gleicher Nachfolgezustand erreicht wird. Dieser *match resolve act* Zyklus wird soweit ausgeführt bis keine weitere Regel mehr evaluiert werden kann oder explizit ein Abbruch erfolgt. Der Ausdruckkörper des Prädikats ist hierzu innerhalb eines Rete-Netzwerkes (Russell und Norvig (2002)) eingebettet, auf dem ein Inferenzalgorithmus die nächste Aktion ableitet und ausführt (Ahlbrecht u. a. (2016)). Es umfasst die Festlegung eines oder mehrerer zu prüfender Agententypen und kann durch Abfrage der Systemeigenschaften oder Verknüpfungen (*join*) mit anderen Fakten aus der Faktenbasis in komplexe Bedingungen überführt werden. Bei mehren gleich gültigen Regelprädikaten gilt eine zufällige Auswahl der nächsten Regeln. Die Evaluationen und Aktionen sind äquivalent zum Reflex 4.3.4 an eine globale Uhr gebunden und sie werden vor der Aktionsausführung im *reflex* durchgeführt.

Die Reactions können vom Agenten genutzt werden um bestimmte vordefinierte Ziele bzw. Systemeigenschaften zu erreichen. Zwar wird die Zielerreichung nicht direkt durch die MARS DSL unterstützt, indirekt lassen sich diese jedoch durch die Durchführung der resultierenden Aktionen erreichen. Es wird darauf hingewiesen, dass zur Durchführung auf die in 4.3.5 näher beschriebenen Workflow-Sprache mit Ausdrücken und Kontrollstrukturen sowie weiteren in Glake u. a. (2017) definierten Elementen zurückgegriffen wird. Die Definition des Reaction Konzepts ist gegeben in 4.14:

Definition 4.14 Reaction in MARS DSL

Eine Reaction wird definiert als Quintupel $R = (name, actions, matches, preCondition, localKnowledge)$, für den gilt:

- *name*: Definiert den Namen der Reaction
- *actions*: Gibt alle verknüpften atomaren oder komplexen Aktionen, zur Erreichung von bestimmten Zielen in einer Ordnung an
- *matches*: Repräsentiert optionale Agententypen auf die in der Faktenbasis geprüft werden soll
- *preCondition*: Gibt die Vorbedingung zur Ausführung der Aktionen an, die in Bezug auf das optionale Match oder der eigenen Agentenzustände gelten müssen

- *localKnowledge*: Stellt alle global innerhalb der Reaction verwendbaren Agentenparameter dar, die aus dem Match hervorgegangen sind

Eine Reaction spezifiziert den internen Prozess eines Agenten im Hinblick auf implizite Ziele, die erreicht werden müssen. Aus diesem Grund bezieht es sich auf eine Reihe von geordneten Aktionen als Teil der Prozessbeschreibung, und enthält einen Scope auf die lokale Wissensbasis des betroffenen Agenten (selbst oder fremd), um die Ausführungsreihenfolge festzulegen. Da die von den Agenten öffentlich sichtbaren Eigenschaften allein über deren passive Aktionen 4.3.3 bekannt sind und ausgehend einer aktiven Aktion 4.3.3 die Interaktion erfolgt, sind sie damit auch an daran gekoppelte Protokolle mit festgelegten Nachrichten und Anforderungen an das Senden und Empfangen gebunden. Schließlich setzt die Variable *preCondition* das Constraint um, die vor der Ausführung erfüllt sein müssen. Diese lassen sich entweder lose als logischen Ausdruck definieren oder verknüpft mittels Quantoren oder Join's in Relation zu den aus der Match Variablen erfassten Agenten stellen. Weitere Details zu den technischen Abfragen finden sich in Glake u. a. (2017) und Russell und Norvig (2002) sowie deren Entwurf in 5.5.3.

4.3.5. Workflowsicht

Im Folgenden soll die MARS-Base DSL näher erläutert werden, die eine anwendbare Workflow-Sprache für die ausführenden Aktionen festlegt. Eine solche Workflow-Spezifikation kann hierzu im weitem Sinne normalerweise aus verschiedenen Perspektiven verstanden werden: Die Kontrollfluss-Perspektive oder Prozess-Perspektive beschreibt Aktionen und ihre Ausführungsreihenfolge durch Darstellung des Ausführungsflusses. Aktionen in elementarer Form sind atomare Arbeitseinheiten, während in komplexerer Form diese modularisiert in einer Reihe von Aktionen sind. Die Datenperspektive definiert, wie Daten in der Kontrollfluss-Perspektive verarbeitet werden können. Die Ansicht ist etabliert und durch diverse bekannte Programmier- und Modellierungssprachen (Bettini (2016)) (Hejlsberg u. a. (2003)) (Voelter u. a. (2013)) (Galland u. a. (2015)) bewährt. Nachfolgend wird hierzu nur eine Teilmenge beschrieben, die zur Modellierung von Agentenmodellen zusätzlich eingeführt oder für bestehende Konzepte ergänzt wurden. Fehlende Elemente und weitere Details wurden hierzu bereits in Glake u. a. (2017) und Glake (2016) diskutiert.

Exploration

Voraussetzung für eine direkte Agenteninteraktion in der MARS DSL ist die Möglichkeit Interaktionspartner zuvor wahrnehmen zu können. Zwar lassen sich Nachrichten auch durch

einen globalen Broker an jeweils Interessierte delegieren, letztendlich muss ein Agent jedoch auch diesen Broker wahrnehmen um andere erreichen zu können. Aus diesem Grunde liegt mit dem Explorations-Ausdruck *explore* eine Abfragemöglichkeit über die Agenten innerhalb des globalen Environment wahrgenommen werden können. Der gewählte Typ kann hierbei ein gemeinsamer Obertyp oder ein spezifischer sein. Von der aktuellen Position ausgehend, wird eine Abfrage auf die zugrundeliegende Umgebung ausgeführt, die sich an die spezifizierte Typhierarchie 4.3 hält und darüber hinaus, mittels Filterprädikat, nach einer konkreten Agenteninstanz abgefragt werden kann. Dieses Prädikat setzt dazu auf das Aktiv-Passiv Modell 4.10 auf. Es wird unterschieden zwischen einer positionsabhängigen Exploration und einer positionsunabhängigen Exploration. Erster nutzt einen bestimmten Ausgangspunkt, in der Regel die Agentenposition des ausgehenden Agenten und gibt anschließend eine über die euklidische Distanz geordnete Menge von Agenten Instanzen zurück.

Definition 4.15 *Exploration in MARS DSL*

Eine Exploration wird definiert als Quadrupel $E = (agentType, where, coordinate, actor)$, für den gilt:

- *agentType*: Definiert den Agententyp der exploriert werden soll
- *where*: Repräsentiert ein Lambda Prädikat über die eine zusätzliche Filterung erfolgen kann
- *coordinate*: Repräsentiert die ausgehende Koordinate, von wo aus die Exploration erfolgen soll
- *actor*: Definiert den agierenden Agenten der diese Sensoraktion ausführt

Die Exploration anderer Agenten wird durch Angabe des *agentType* festgelegt. Dieser muss dabei einem Layer zugeordnet sein, den der ausgehende Agent überhaupt wahrnehmen kann. Mit *where* wird eine Filterung ermöglicht die zusätzlich zum jeweiligen Typ, ausgehend von den definierten Koordinaten *coordinate*, erfolgt. Diese wird andernfalls automatisch von der ausgehenden Agentenposition abgeleitet. Der Parametertyp im Lambda Prädikat muss dabei dem des explorierten Typs entsprechen. Das ausgehende Ereignis wird dabei allein durch *actor* beschrieben, der damit dem umgebenden Agenten entspricht.

Eine besondere Form nimmt das *nearest* Konzept an, es ist eine Spezialisierung dieser Exploration, das lediglich das erste Element aus der distanz-geordneten Agentenmengen darstellt.

Bewegung

Das Bewegen ist ein zentraler Bestandteil verschiedener Modelltypen (Wilensky (2015)). Dazu zählt die Möglichkeit sich innerhalb der Umgebung gleichmäßig fortzubewegen als auch sich intelligent direkt an verschiedene Punkte zu positionieren (Aschermann u. a. (2016b)). Mit Intelligent ist hierbei gemeint, dass aus der Schlussfolgerung des eigenen Zustands, z.B. eine zwischenzeitliche Berechnung für den aktuellen Zeitschritt t_i , der ausgehende Agent auf eine neue Position gesetzt wird (Tchappi Haman u. a. (2017)) und damit inhärent die global gültige Systemgleichung näherungsweise löst, wie dies beispielhaft mit Systemgleichungen zur Verkehrssimulation begründet werden kann (Weyl u. a. (2018)). Die abstrakte Syntax ist in 4.16 gegeben:

Definition 4.16 *Movement in MARS DSL*

Eine Bewegung wird definiert als Quadrupel $M = (agentType, coordinate, steps, actor)$, für den gilt:

- *agentType*: Definiert eine Agenten Instanz in dessen Richtung sich bewegt werden soll
- *coordinate*: Spezifiziert die Zielkoordinate zu der sich bewegt werden soll
- *steps*: Gibt die Anzahl der Schritte wieder die der Agent in Richtung des Ziel gehen soll
- *actor*: Definiert den agierenden Agenten der diese Bewegung ausführt

MARS DSL führt zu diesem Zweck eine *move* Expression Logik ein, über die ein Agent sich auf dem zugrundeliegenden Environment entlang bewegen kann. Sowohl mit einfachen Schritten über ein Grid, als auch komplexeren Aktion mit direkter Positionierung. Zur Bewegung zählt die Prüfung und Korrektur der Position, um den Agenten innerhalb des Environment zu halten und damit eine kontrollierte Simulationsgrenze zu schaffen. Allein durch die Lebendigkeitseigenschaft *alive* in 4.3 wird dieser entweder aus dem gesamten System entfernt oder nicht.

Erzeugung

Neben der Bewegung und Wahrnehmung ist das Erzeugen von Agenten ein essenzielles Element der MAS Methodiken (Ahlbrecht u. a. (2016)), ohne die ein solches System nicht entstehen würde. Für MARS DSL Modelle existieren hierzu zwei Möglichkeiten der Agentenerzeugung: Erstens lassen sich Agenten auf den herkömmlichen MARS-Weg zum Systemstart hinweg erzeugen (Hüning u. a. (2016)) und dort die Anzahl der Instanz parametrisieren. Zweitens

lassen sich zur Ausführungszeit des Modells neue Agenten erzeugen und diese mit verschiedenen Optionen versehen. So können sich Agenten nur für einen bestimmten Zeitraum im System befinden, um sie am Schluss unabhängig anderer wieder aus dem System zu entfernen, angegeben über ein Startzeitpunkt t_i zu welchem der Agent erzeugt werden soll und einem Endzeitpunkt, wann dieser wieder entfernt wird t_{i+n} . Fehlende Eingaben werden automatisch aus dem aktuellen Systemtick abgeleitet und Agenten leben andernfalls endlos. Die abstrakte Syntax ist in 4.17 gegeben:

Definition 4.17 *Spawn in MARS DSL*

Ein Spawn wird definiert als 6-Tupel $M = (agentType, coordinate, from, until, when, action, actor)$, für den gilt:

- *agentType*: Definiert den Agententypen der erzeugt werden soll
- *coordinate*: Spezifiziert die Zielkoordinate an dem der neue Agent in der Umgebung erscheint
- *from* : Spezifiziert den Endzeitpunkt, bis wann der Agent Teil des Systems sein soll
- *until* : Spezifiziert den Endzeitpunkt, bis wann der Agent Teil des Systems sein soll
- *actor* : Definiert den agierenden Agenten der diese Erzeugung ausführt

Die MARS DSL überwacht selbstständig die Erzeugung neuer Agenten zu einem bestimmten Zeitpunkt. Das Zeitintervall lässt sich in der jetzigen Form so verwenden, dass ein Agent auch nur für eine gewisse Zeit am Leben ist, indem für den Startzeitpunkt der aktuelle Zeitschritt $from = t_i$ eingesetzt wird und der Endzeitpunkt der erwünschten Dauer des Agentenlebens entspricht. Der Fokus auf den Tick und keiner konkreten Zeitdarstellung wie es mit dem *time-span* oder *date* Typ des Typsystems möglich wäre, ist aufgrund der möglichen Unabhängigkeit des Modells und der Systemgleichung von gewählten der Zeitdimension (Tchappi Haman u. a. (2017)) begründet, zumal der Tick immer auf eine bestimmte zeitliche Granularität {Jahr, Monat, ..., Millisekunde} projiziert werden kann. Ist eine Erzeugung vorgenommen, kann mit der *coordinate* erneut wieder eine Position angegeben werden, an welcher Position im Environment der Agent erscheinen soll, die bei fehlender Positionierung aus der aktuellen Position des *actor* abgeleitet wird. Diese wiederum spezifiziert, ähnlich zur Definition in 4.15 und 4.16 erneut die agierende Entität. Zusammengefasst bietet das MARS DSL Erzeugungskonzept breite Verwendbarkeit, die erst in Kombination mit der Bewegung und Wahrnehmung eine große Vielfalt möglicher Modelltypen (Wilensky (2015)) abbilden kann. Aber auch für sich

alleine ist die Erzeugung neuer Agenten zur Laufzeit ein essenzielles Element, insbesondere als elementarer Bestandteil der Analyse von Populationsdynamiken (Götting u. a. (2013)).

Type-Guard

Aus der starken Typen Verwendung und der Ableitung über das Typsystem, wurde im Zuge der regelbasierten Verhaltensmodellierung (Sierhuis u. a. (2009)) (Giret u. a. (2017)) ein besonderer Case-Verteiler eingeführt. Dieser *switch...case* bekannte Ausdruck wurde um das Konzept der Type Guards ergänzt, der aus einer abzufragenden Obermenge eine Typprüfung inklusive der Definition eines Prädikats zulässt. Der Case Verteiler funktioniert hierbei ähnlich zum *reaction* Konzept 4.3.4 indem es ein Match auf einen Agententypen (z.B. über die Exploration) zulässt und auf diesen ein Filterprädikat anwendet. Die abstrakte Syntax wird definiert in 4.18:

Definition 4.18 Type-Guard in MARS DSL

Eine Type-Guard Verzweigung wird definiert als 7-Tupel $TG = (parameter, switch, cases, typeGuards, predicate, action, fallThrough)$, für den gilt:

- *parameters*: Spezifiziert einen Parameter als Prüfung innerhalb des Case-Verteilers genutzt werden kann
- *switch*: Gibt den Switch Ausdruck wieder zu dessen Resultat eine Fallunterscheidung vorgenommen werden soll
- *typeGuards*: Spezifiziert die Menge der Typen auf die der Switch vor Prüfung des Prädikats passen muss
- *cases*: Spezifiziert den Case der die Typ-Prüfung und die Prädikate enthält
- *predicate*: Spezifiziert die eigentliche Fallbedingung, entweder als konkreten Wert oder zur prüfenden Ausdruck
- *action*: Repräsentiert den resultierenden Aktionskörper der bei Erfüllung ausgeführt werden soll.
- *fallThrough*: Gibt bei mehreren existierenden Type-Guards und Cases an, dass deren Prüfung mit anderen geteilt werden soll

Der *switch ... case* unterscheidet sich nur geringfügig von den beschriebene Konzept in Bettini (2016). Die Spezifikation eines Typs erlaubt jedoch die Prüfung auf eine betrachteten

Typ, die insbesondere dann Verwendung findet, sollte sich der Agent in einer Typhierarchie befinden. Das Wahrnehmen und Interagieren mit Agenten wird für viele Modelltypen immer mit diversen Typen stattfinden, die sich aus der Natur heraus automatisch in einer vererbten Ordnungsrelation befinden. Auf diese Relation wird hier Bezug genommen, indem beispielsweise für Modelle der Verkehrssimulation keine wirkliche Unterscheidung vorgenommen wird, ob das wahrgenommene Fahrzeug $T1$ oder Bürger $T2$ oder eine Bus $T3$ die aktuelle betrachtete Straße blockiert [Tchappi Haman u. a. \(2017\)](#). Erst im Nachhinein passt es sein eigenes Verhalten dem erfassten Typen entsprechend an, und umfährt vorsichtshalber das vor ihm befindliche Auto bzw. wartenden Bürger oder wartet den Bus rechtskonform ab. Andernfalls würde eine Kette aus Verzweigungen folgen müssen.

Globale Uhr

Die globale Uhr stellt einen Synchronisationspunkt aller Agenten innerhalb des Systems dar ([Ahlbrecht u. a. \(2014\)](#)). Jeder Agent wird durch eine global aktive Uhr geschaltet und kann daran seine Aktionen ausrichten. Zeitabhängige Aktionen sind dabei der häufigste Einsatz ([Aschermann u. a. \(2016a\)](#)) und werden auch als Teil komplexer Pläne eines Agenten mit involviert ([Caillou u. a. \(2017\)](#)). In MARS DSL wird diese Simulationszeit in Form eines fortlaufenden Ticks geführt, der als zeitliche Komponente jeder Differentialgleichung ([Tchappi Haman u. a. \(2017\)](#)) auftritt und dem Agenten diese näherungsweise lösen lässt. In Form eines eigenen Literals, ähnlich der Koordinatenreferenz $xcor$, $ycor$ ist sie Teil der Grammatik und vom Typsystem als Ganzzahl festgelegt. Die Definition ist in [4.19](#) gegeben und umfasst allein den Tick für den die Zeit schrittweise um 1 aktualisiert wird.

Definition 4.19 Globale Uhr in MARS DSL

Die globale Uhr wird definiert als 1-Tupel $GC = (tick)$, für den gilt:

- *tick*: Definiert den aktuellen Tick der Simulation

Da allein mit dem Tick keine konkrete Zeitplanung erfolgen kann liegt mit der *Mars Library* eine Bibliothek bereit, in der ein zusätzliches Zeitmodul *Mars.Library.Time* existiert. Diese umfasst eine Reihe nützlicher Projektionen des Ticks auf die Komponenten Jahre, Monaten Stunden, Sekunden, ..., bisherige Monate, Tage, Stunden etc. und erleichtern die Handhabung zeitbedingter Aktionen und lassen dem Agenten die Freiheit zur Kontrolle seiner Workflows und Interaktionen. Darüber hinaus ermöglicht es die Abfrage des von außen parametrisierten *startDate* und *endDate* der den Startzeitpunkt und Endzeitpunkt der Simulation wiedergibt. Die Erfassung eines einzelnen Zeitschrittes mit der Projektion auf die unterschiedlichen

Zeitgranularität kann das gesetzte δt identifizieren, auch wenn es für die meisten zeitbedingten Aktionen keinen besonderen Einfluss hat, sondern das System dahingehend einstellt, ob bspw. fix definierte Bedingungen für bestimmte Zeitpunkte ggf. übersprungen werden oder ob betrachtete Zeitintervalle - der Frühling als Reproduktionszeitraum für modellierte Tiere (de Mutsert u. a. (2017)) oder die Winterzeit für einen unsicherer Verkehrszeit (Tchappi Haman u. a. (2017)) - überhaupt auftreten.

Lambda Ausdruck

Im Zuge der Interaktionsmöglichkeiten in Abschnitt 4.3.3 und diverse Abfragemöglichkeiten auf den (Daten-)Layeren in Abschnitt 4.3.2 sowie 4.3.2 wurde das Konzept der Lambda Ausdrücke mit in die Sprache aufgenommen. Dieses, auch als anonyme Funktion bekannter Konzept, richtet sich in erster Linie an die Verwendung als Filtermöglichkeit oder Bereitstellung eines prädikatenlogischen Ausdrucks über Verweis-Referenztypen, anstelle eines konkreten Namens. In Ahlbrecht u. a. (2016) wird diese Technik dagegen auch als Möglichkeit betrachtet intelligente Schlussfolgerungen zu ziehen, indem diese als geordnete Struktur dynamisch gespeichert und Anpassungen auf dieser Struktur vorgenommen werden. Dies entspricht dem *reaction* Konzept in Abschnitt (4.3.4), der diese dazu innerhalb eines Rete-Netzwerkes einhängt (Russell und Norvig (2002)). Zudem lassen sich hiermit Aktionen sowohl zwischen Entitäten als auch innerhalb von Aktionskörpern austauschen und damit eine Erweiterung zum vereinfachten Nachrichtendesign (Glake u. a. (2017)) ansehen, indem nun keine Daten mehr ausgetauscht werden, sondern konkrete Aktionsaufforderungen, z.B. für Pläne (Aschermann u. a. (2016b)). Ein Lambda Ausdruck wird in der MARS DSL folgendermaßen definiert:

Definition 4.20 Lambda in MARS DSL

Ein Lambda Ausdruck wird definiert als 3-Tupel $L = (parameters, action, knowledge)$, für den gilt:

- *parameters*: Spezifiziert die Eingabeparameter mit optionalem Typ der andernfalls automatisch abgeleitet wurde
- *action*: Gibt eine elementare oder komplexe Aktion an die durchgeführt wird
- *knowledge*: Gibt die eigene Wissensbasis an, die global innerhalb des Aktionskörpers verwendet werden kann
- *returnType*: Gibt den automatisch abgeleiteten Rückgabetyt aus der Programmlogik an

Die Möglichkeiten aus dem Einsatz von Lambda Ausdrücken sind vielfältig. Von Funktionen höherer Ordnung bis einfachen Erweiterungen zu Mengen und Operationen darauf sind Einsatzmöglichkeiten vorgesehen. Als essenzieller Teil diverser Entwicklungssprachen (Strembeck und Zdun (2009)) hat sich deren Einsatz bereits bewährt und kann zur Ergänzung unterschiedlicher Konzepte herangezogen und dafür minimal angepasst werden.

Weitere Elemente

Weitere Elemente die in dieser Arbeit nicht exakt beschrieben wurden umfassen hauptsächlich Elemente die aus der Zielsprache (Hejlsberg u. a. (2003)) oder in anderen Spezifikationen (Alves-Foss (1999)) bereits ausgiebig behandelt wurden. Da diese Sprachelemente in Glake (2016) intensiv diskutiert werden, wird an dieser Stelle erneut darauf verwiesen. Im Allgemeinen umfassen sie jedoch:

Verzweigungen Das Workflow-Konzept der Basissprache (Glake u. a. (2017)), unterstützt die Verzweigung von Kontrollflüssen. Genau zu diesem Zweck wird das abstrakte Konzept von Verzweigungen zur Verfügung gestellt, das einen Punkt im Körper einer Aktion darstellt, bei dem sich ein einzelner Kontrollfluss in mehrere Kontrollflüsse aufteilt. Das Verzweigungskonzept wird dabei in Form bekannter *if* Ausdrücke sowie nach Abschnitt 4.3.5 auch in Form größerer Case-Verteiler *switch...case* bereitgestellt. Es legt unter anderem die Sicht des individuellen Agenten auf die für ihn zur Verfügung stehenden Ressourcen fest, mit der Frage ob er Zugriff darauf erhält und welches Zugriffsverhalten er einsetzen kann - um seine Aufgaben zu lösen - entweder auf reaktive oder proaktive Weise. Entschieden wird diese auf Basis eines Prädikats.

Schleifenkontrollstrukturen Enthält Schleifenstrukturen zur Wiederholung eines Workflows und zum Iterieren über Mengen (z.B. explorierte Agenten). Auf Basis eines Prädikats wird, ähnlich zu Hejlsberg u. a. (2003) eine Struktur bis zur Erfüllung des Prädikats ausgeführt. Schleifen lassen sich entweder mit einem fortlaufenden Index definieren oder mit einer Vor- oder Nachbedingung zum Aktionskörper.

Aufrufe Der Zugriff auf Modellelemente und Ressourcen erfolgt über Aufrufe. Diese können sich auf (Daten-)Layer, lokale und globale Variablen oder eigene States und Methoden oder Aktionen beziehen. Ein Aufruf ist entweder mit einem Empfänger verknüpft, auf dessen Typelemente zugegriffen werden soll oder es handelt sich um einen direkten Aufruf auf die eigenen referenzierbaren Elemente. Wie in Bettini (2016) wird auch hier zwischen statischen und dynamischen Aufrufen unterschieden, jedoch mit der

Zusatzmöglichkeit durch eine $(:)$ Notation anstelle einer $(.)$ Notation explizit anzugeben statisch aufgerufene Elemente eine höhere Bindung als deren dynamisch überladene Pendant zu geben.

Arithmetische- und logische Ausdrücke Umfassen eine Teilmenge von binären und unären Operatoren die aus der Mathematik und Logik bekannt sind. Dem hinzukommen sprach-agnostische Operatoren die einige Funktionen im Hintergrund verkürzen. Unter den binären Operationen sind jene aus der Arithmetik wie $(+, *, -, /)$, die arithmetische Modulo Rechnung $\%$, definiert als $a \bmod b := \lfloor \frac{a}{b} \rfloor$, eine verkürzte Exponential-Operation $**$, $^$ sowie logische Binäroperatoren wie *and*, *or*, relationale Operatoren $<$, $>$, $>=$, $<=$ und Verkürzungen der semantischen und physikalischen Gleichheit $===$, $==$, $!==$, $!=$. Zu den unären gehören der Negations Operator *not*, das Gegenstück in der Arithmetik $+$, $-$ sowie ein Inkrement- und Dekrement Operator $++$, $-$. Neben den sprach-agnostischen Operatoren zur Modulo Rechnung etc. wurde zusätzlich der Range-Operator $e1 .. e2$, $e1 .. < e2$, $e1 >.. e2$ eingeführt, über den eine Ganzzahl Sequenz mit Schrittlänge 1 erzeugt werden kann, womit folglich auch ein exklusives Range möglich sein sollte. Auf Seiten des $<$, $>$ lässt sich daher angeben, welche Seite nicht Teil der Sequenz sein soll. Schlussendlich wurde mit Einführung des Spaceship Operators $<=>$ nebenher eine Abstraktion für die Ordnungsberechnung der beteiligten Typen ermöglicht.

Einfache und komplexe Literale Zu den essenziellen Elementen fast jeder Sprache gehören neben den Operatoren auch Literale die zum Setzen beispielsweise von Standardwerten und Konstanten gebraucht werden. Diese Literale umfassen die Menge der reellen Zahlen \mathbb{R} , folglich auch der Ganzzahlen \mathbb{Z} , Zeichenketten aus dem Unicode Zeichensatz, beginnend mit `".."`, `'...'`, das für Referenztypen besondere *nil* Literal und die Menge $\{true, false\}$ der Aussagenlogik. Die einfachen Literale sind Bestandteile vieler anderer Sprachen (Voelter u. a. (2013)) (Bettini (2016)) und wurden um einige nützliche Konzepte ergänzt. Dies sind insbesondere komplexe Mengen- sowie Tupel-Literale die automatisch eine Typableitung der innewohnenden Elemente ausführt. Es fallen darunter Listenliterale `#[...]`, Set-Literale `#{...}`, Array Literale `##[...]`, direkte Abbildungen mittels `Map #\{a -> 1, b -> 2\}` sowie Tupel bis höchstens 8 Elemente wie z.B. `#(1,2,3)`, `#(xcor, ycor)`. Die in den Tupel verwendeten *xcor* und *ycor* bilden hierbei noch eine Besonderheit. Sie repräsentieren die innerhalb des Agentenkontexts aktuelle Position im Environment und sind Teil der Grammatik. Sie bilden den fixen Bestandteil der Agentendefinition in Abschnitt 4.3.2 und können zum Zwecke der Positionsabfrage bspw. als `#(xcor, ycor)` zurückgegeben werden, entsprechend der Bewegungslogik, wie in Abschnitt 4.3.5. Insgesamt erleichtert

es Handhabung lokations-bedingter Aktionen und lassen dem Agenten die Freiheit zur Kontrolle über seinen eigenen Workflow.

4.4. Zielsystemerzeugung nach MARS

Der nachfolgende Abschnitt beschreibt wie die konkrete Abbildung vom Ausgangsmodell, formuliert in der MARS DSL sowie als zusätzliches Austauschformat nach dem MARS Metamodell [Glake u. a. \(2017\)](#) vorliegend, in das ausführbare Zielsystem (MARS LIFE Umgebung) ([Hüning u. a. \(2016\)](#)) definiert ist. Die Zielstruktur und ein individuelles Mapping der Elemente im MMM und AMM wird präsentiert, die eine Transformation umfasst und damit auf der Definition von ([Kleppe u. a. \(2003\)](#)) aufsetzt und deren MDA Überlegung umsetzt.

4.4.1. Erzeugungsmodell

Zur Erzeugung eines lauffähigen Systems mithilfe eines Übersetzers ist festzustellen, wie die Struktur des lauffähigen Zielsystems aussehen soll. Dabei ist dessen Ausprägung jedoch unerheblich um nicht an eine bestimmte Menge von Modellen gebunden zu sein, sondern sie muss offen für Erweiterungen sein und auf einer abstrakteren Ebene vorgenommen werden ([Goll \(2014\)](#)). [Abbildung 4.3](#) zeigt hierzu das fachliche Erzeugungsmodell mit beteiligten Elementen und ihren Beziehungen untereinander, abgegrenzt um die orangefarbenen hervorgehobenen Komponenten, die bereits aus dem MARS-LIFE ([Hüning u. a. \(2016\)](#)) stammen.

Es wird ein System betrachtet, das zentral ein gemeinsames Environment besitzen soll. Dieses als Grid zur Verfügung gestellte Environment, wird innerhalb eines abstrakten generischen Layers zur Verfügung gestellt, von dem jeder andere konkret definierte Layertyp spezialisiert wird. Sowohl der generische Layer als auch der gemeinsame Grid Layer implementieren den von MARS LIFE vorgegebenen *ISteppedActiveLayer* und *ISteppedLayer* Kontrakt. Auf spätere Einsatzmöglichkeiten zurückzuführen, wird mit *ISteppedActiveLayer* die Möglichkeit geboten, auch für einen Layer vor, während oder nach einem Tick eine Aktion auszuführen, die sich beispielsweise auf die zunächst abgegrenzte Anforderung der in [4.2](#) erwähnten beobachtbarer Layerzustände reduzieren lässt.

Zur individuellen Agentenmodellierung wird ein abstrakter generischer Grid Agent eingeführt, der eine Spezialisierung des MARS-spezifischen *GridAgent* darstellt und dessen Initialisierung sowie wichtigste Attribute implementiert. Ausgehend davon werden wiederum die konkret definierten Agententypen, die einzig aus der MARS DSL stammenden Modellelemente enthalten, eingefügt. Dazu gehören, die aktiven- *ActiveAction* und passiven Aktionen *PassiveAction*, die in Form von Methoden übersetzt werden. Es gehören die Zustände, die als

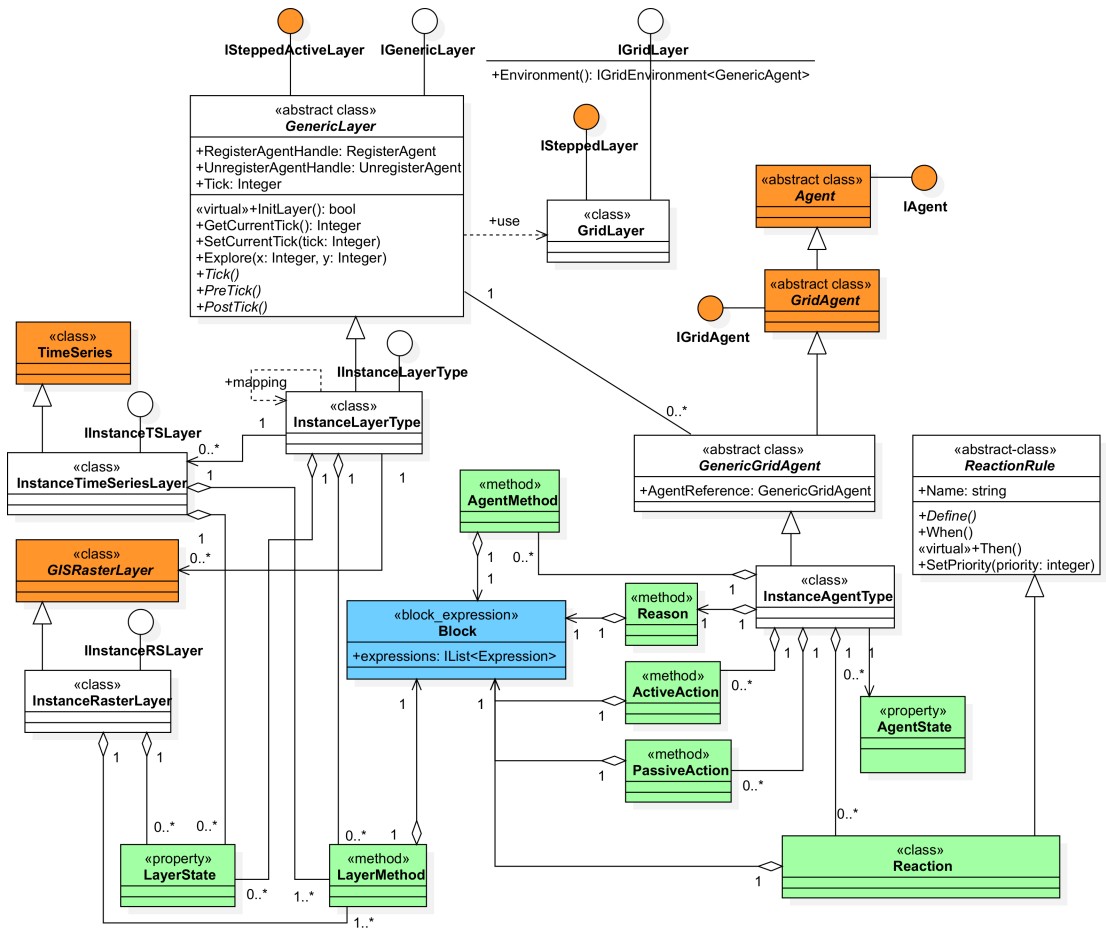


Abbildung 4.3.: Modell des Zielsystems mit Abgrenzung zu den Komponenten aus LIFE.

öffentliche *Properties* übernommen werden, dazu sowie die zentrale Reason Methode (Hüning u. a. (2016)), die sowohl den *Reflex* als auch die *Reaction* Verarbeitung enthalten. Referenzen zu den modellierten Klassen werden äquivalent übernommen und verschiedene mögliche Agentenumgebungen sind entweder direkt im Zielsystem implementiert oder es wird die öffentliche Schnittstelle der existierenden LIFE Variante angesteuert. D.h. es sind eigens für die MARS DSL entwickelte Agentenumgebungen im Zielsystem vorhanden die nicht Teil des öffentlichen MARS-LIFE sind.

Zur feingranularen Layermodellierung wurden die in 4.1 ergänzten Methoden verwendet, die zugleich von den konkreten Agententypen zur Refaktorisierung direkt übernommen wurden. Des Weiteren umfasst die Ergänzung spezifische Layerzustände, die für alle beteiligten Agenten frei zugänglich sind und damit als gemeinsamer Speicher zur Verfügung stehen, d.h. dass sowohl Schreib- als auch Lesezugriffe kontrolliert erfolgen müssen. Zu beachten ist, dass aus fachlichen Gründen zwar ein *IGenericLayer* modelliert ist, dieser jedoch im Zielsystem nie konkret auftaucht. Er dient quasi als Konvention zur Erzeugung eines MMM Agenten-Layers und wird allein zur Identifizierung innerhalb des Workflows der Codegenerierung gebraucht (Herrington (2003)).

Eine besondere Stellung nimmt der in Abbildung 4.3 vorhandene generische Layer ein, der als Komponente zur Implementierung der MARS-spezifischen Bedingungen herangezogen wird. Da die Erzeugung an sich den konkret definierten Layertypen überlassen ist, die auch das Wissen um die Agenten haben, wird anstelle einer vollständigen Initialisierung diese aufgeteilt und dabei die Instanziierung konkreter Agenten weiterhin in der Spezialisierung vorgenommen.

4.4.2. Mapping - Regeln

Zusammengefasst sind damit zwei wesentliche Abbildungen für Layer 4.2 und Agenten 4.6 inklusive ihrer implementierten Schnittstellentypen 4.3 definiert. Eine Mapping Regel besteht dabei zum einen aus dem Konzept, was auf das Zielelement der Implementierungssprache (Hejlsberg u. a. (2003)) abgebildet wird mit deren individuellen Mapping der beteiligten Attribute vom Ausgangsmodell auf das Zielmetamodell, zum anderen existiert ein strukturgetriebenes Mapping, bei dem das Ausgangselement selbst innerhalb eines anderen Kontexts in weitere Beziehungen gebracht wird.

Mapping 4.1: Modell → Namensraum

Attributgetriebene Abbildung:

- **Name:** Modellbezeichnung wird an das Ende der Standard Namensraum Bezeichnung *Mars.Modelling* angehängt
- **Layertyp** Mapping gemäß Mapping 4.2 und als eigenständiges *Projekt*, beschrieben wie in Abschnitt 5.9
- **Agententyp:** Mapping gemäß 4.6 im *Zielprojekt* des assoziierten Agentenl-Layers
- **Time Series Layer:** Mapping gemäß 4.4 und als eigenständiges *Projekt*, beschrieben wie in Abschnitt 5.9
- **Raster-Layer:** Mapping gemäß 4.5 und als eigenständiges *Projekt*, beschrieben wie in Abschnitt 5.9
- **Grid Layer** Erzeugung eines allgemeinen Grid Environments über den *Generischen Grid Agenten* inklusive *Stellvertreter Grid Agent* mit entsprechender Signatur zur externen Parametrisierung und Dimensionierung des Environments (siehe [Glake \(2017\)](#)).

Strukturgetrieben Abbildung:

- **behandelt:** Sichtbarkeit innerhalb des Zielsystems durch Bereitstellung von *Import Sammlung* aller potenzieller Namensräume, werden bei Kompilierung automatisch *wegoptimiert*
- **organisiert:** Import für jede erzeugte Datei um den eigenen *Namensraum* sichtbar zu machen

Mapping 4.2: Layer → Klasse

Attributgetriebene Abbildung:

- **Name:** Die Bezeichnung der öffentlichen *Klasse* und des implementierten *Schnittstellentyps*
- **Methoden:** Öffentliche *Methodendefinition* innerhalb der *Klasse* (siehe Mapping 4.8)

- **Sensor:** *Konstruktorinjektion* Goll (2014) mit Ordnung der direkt sichtbaren Ziellayer. Injezierte Layertypen werden auf *Typeeigenschaft* abgebildet, respektiv auf deren *Schnittstellentyp*.
- **Agenten:** Extern parametrisierte *Instanziierung von Agenten* über internen *Agenten Manager* mit Delegation eigener *Layerabhängigkeiten* an jeden Agenten
- **Zustände:** Mapping gemäß 4.7 ohne Möglichkeit diese während der Ausführung zu *observieren* oder extern zu *parametrisieren*.

Strukturgetriebe Abbildung:

- **benutzt:** Obertyp in Form des *Generischen Layer*.
- **behandelt:** Virtuelle *Layer Initialisierungs-Methode* zur Erzeugung konkreter *Agenten* aus externen Daten, indem die Basisimplementierung verwendet und um eine eigene Spezialisierung ergänzt wird
- **organisiert:** Individuelle Agentenerzeugung für jeden verknüpften Agententyp

Mapping 4.3: Layer → Schnittstelle

Attributgetriebene Abbildung:

- **Name:** Die Bezeichnung der öffentlichen *Schnittstelle*
- **Methoden:** Methodendeklaration innerhalb des Interfaces für alle von außen zugreifbaren Methoden wie die individuelle Agentenerzeugung (siehe Mapping 4.8)
- **Sensor:** Typdeklaration mit Bezeichnung der sensorischen Abhängigkeiten
- **Felder:** Öffentliche *Eigenschaft* mit *Hintergrundfeld* innerhalb der *Klasse* (siehe Mapping 4.7)

Strukturgetriebe Abbildung:

- **benutzt:** Obertyp in Form des *ISteppedActiveLayer*
- **behandelt:** Abhängigkeiten zu Agenten-Layern oder Datenlayern die auf Basis des Sensors als Projektabhängigkeit eingetragen werden soll

- **organisiert:** Änderung des Obertyps auf *ISteppedActive*, wenn es sich lediglich um einen Datenlayer handelt

Mapping 4.4: Time Series Layer → Klasse

Attributgetriebene Abbildung:

- **Name:** Die Bezeichnung der öffentlichen *Klasse* und des implementierten *Schnittstellentyps*
- **Methoden:** Öffentliche *Methodendefinition* innerhalb der *Klasse* (siehe Mapping 4.8)
- **Felder:** Öffentliche *Eigenschaft* mit *Hintergrundfeld* innerhalb der *Klasse* (siehe Mapping 4.7)

Strukturgetriebene Abbildung:

- **benutzt:** Implementiert eigene Schnittstelle die jegliche Erweiterungsmethoden enthält, ansonsten wird gemäß 4.3 erzeugt.
- **organisiert:** Erzeugung der einfachen Erweiterungsmethoden zur Verwendung und Datenabfrage in der implementierten Klasse

Mapping 4.5: Raster-Layer → Klasse

Attributgetriebene Abbildung:

- **Name:** Die Bezeichnung der öffentlichen *Klasse* und des implementierten *Schnittstellentyps*
- **Alias:** Neue öffentlich lesbare *Eigenschaft* mit Zeiger auf sich selbst
- **Methoden:** Öffentliche *Methodendefinition* innerhalb der *Klasse* (siehe Mapping 4.8)
- **Felder:** Öffentliche *Eigenschaft* mit *Hintergrundfeld* innerhalb der *Klasse* (siehe Mapping 4.7)

Strukturgetriebene Abbildung:

- **benutzt:** Implementiert eine eigene Schnittstelle die jegliche Erweiterungsmethoden enthält, ansonsten wird gemäß 4.3 erzeugt.

Mapping 4.6: Agent → Klasse

Attributgetriebene Abbildung:

- **Name:** Die Bezeichnung der öffentlichen *Klasse* und des implementierten *Schnittstellentyps*
- **Methoden:** Private modifizierte *Methodendefinition* innerhalb der *Klasse* (siehe Mapping 4.8)
- **Passive Aktionen:** Öffentliche *Methodendefinition* mit Übersetzung der Ausdrücke nach [Glake \(2016\)](#)
- **Aktive Aktionen:** Private *Methodendefinition* mit Übersetzung der Ausdrücke nach [Glake \(2016\)](#)
- **Aktiv-Passiv Mapping:** *Ziellayer* werden als Typeigenschaften über den *Schnittstellentyp* in den Agenten eingetragen und mittels darüber liegenden Layer zur Verfügung gestellt (siehe Mapping 4.2).
- **Reflex:** Mapping des Reflexbody an den Anfang der *Reason* Methode
- **Reaction:** Mapping jeder *Reaction*, geordnet als Verzweigung innerhalb der *Reason* Methode.
- **Datenlayer:** Datenlayer Referenz in Form eines statischen *Kopplungspunkt*, bezeichnet nach dem Typ und bereit zur Delegation an konkret vorhandene Implementierung
- **Zustände:** Mapping gemäß 4.7

Strukturgetriebene Abbildung:

- **benutzt:** Obertyp in Form des *Generischen Grid Agent* mit transitiver Beziehung zum abstrakten MARS spezifischen *GridAgent* Typen des *Grid* Environments

- **behandelt:** Implementierung der transitiven Schnittstelle des *GridAgent* mit Referenz auf sich selbst
- **organisiert:** Einfügen und Anmelden des Agenten in das gemeinsame *Grid* (Glake u. a. (2017)) und der MARS LIFE Umgebung

Mapping 4.7: Mentale Zustände → Typeigenschaft

Attributgetriebene Abbildung:

- **Name:** Die Bezeichnung der öffentlichen Eigenschaft
- **Typ:** Typisierung der Eigenschaft
- **Modifizierer:** Standard öffentlicher Modifizierer
- **Beschreibbar:** *Nur Lesen* Modifizierer wird auf Feld ergänzt
- **Beobachtbar:** Private *Set* Modifizierer auf Eigenschaft zur Identifikation mittels reflektiver Analyse
- **Extern initialisiert:** Vorhandenes *Set* und Konstruktorparameter innerhalb des Agenten inklusive Annotation des Konstruktors
- **Standardwert:** Übersetzung des Wertes oder Ausdrucks nach Glake (2016)

Strukturgetriebene Abbildung:

- **organisiert** optionale Felddeklaration wenn Zustand als konstant modifiziert wird

Mapping 4.8: Methode → Klassenmethode

Attributgetriebene Abbildung:

- **Name:** Die Bezeichnung innerhalb der *Methodensignatur* sowohl in der Implementierung als auch der Schnittstelle
- **Rückgabe:** Typisierung der *Methodensignatur*

- **Parameter:** Übernahme der *Typisierung* und des *Bezeichners* jedes Methodenparameters

Strukturgetriebene Abbildung:

- **organisiert** Deklaration als statisch, wenn es sich um eine Methode eines Datenlayers handelt.

4.4.3. Templater - Konzepte

Mithilfe der vorlagenbasierten Texterzeugung, werden die Beschreibungskonstrukte zum Zeitpunkt der Codegenerierung in das Äquivalent der Zielsprache transformiert. Dazu existiert eine *Template Engine* die verschiedene Vorlagen korrekt ausfüllen kann und unter Zuhilfenahme einer eigens entwickelten eingebetteten Vorlagensprache diese korrekt nach den angegebenen Konstrukten ausfüllen kann, ähnlich wie es mit der MOF *Model-to-Text* Transformation möglich ist, die jedoch zur Beschreibung des ausgehenden MOF-basierte Modells gedacht ist, was für die MARS DSL nicht gilt (OMG (2014)). Die Template Sprache bildet damit eine eigene Transformationssprache, die jedoch entgegen der eigenständigen Mächtigkeit der MOF-Sprache erst in Zusammenspiel mit den umgebenden Komponenten der MARS DSL eine korrekte Übersetzung erfüllen kann. Aus Gründen der Geschwindigkeit wurde auf eine begrenzte Menge von entwickelten Template-Konstrukten der Fokus gesetzt. Eine Auswahl und Vergleich verschiedener vorlagenbasierter Erzeugungssprachen und ihrer Sprachen findet sich im Anhang in Glake (2017).

Ausgehend von der festgelegten Struktur eines MARS Modells in Abbildung 5.9 werden zur Konstruktion der kleinteiligen Elemente, wie *Agenten*, *Klassen*, *Typeigenschaften*, *Methoden*, *Aufzählungstypen*, *Workflows* sowie *Uses* eine Reihe vereinfachender Erzeugungskonstrukte gebraucht die an dieser Stelle um mehrere Aspekte wie Variablenbindung ergänzt wurde. Die Tabelle 4.1 gibt eine Auflistung der Anforderungen an die Template Engine aus Weyl u. a. (2018) an. Hierzu wurden Erkenntnisse aus Aho (2008) mit einbezogen, ebenso aus den betrachteten Template Engines zu Razor (Palermo u. a. (2012)) und Xpand (Bettini (2016)) und der MOFM2T Übersetzung (OMG (2014)).

Für die Übersetzungssprache werden die hier eingeführten Vorlagen spezifiziert, als eine Textvorlage mit Platzhaltern für Daten. Die Platzhalter werden dynamisch zur Laufzeit an das Eingabemodell gebunden und extrahieren die dortigen enthaltenen Daten, wie zum Beispiel Bezeichner und Modifizierer. Platzhalter stellen Ausdrücke dar, die als Metamodellentitäten angegeben sind und über die Instanzen dieser Metamodellentitäten ausgewertet werden, d.h.

Konzept	Beschreibung
Definition von Platzhaltern	Platzhalter mit Namen referenzieren auf die Typeigenschaften einer Eingabeklasse. Typisierte Eigenschaften erlauben die Anwendung der assoziierten Funktionalitäten zu diesem Typ.
Definition von lokalen Variablen	Es werden Variablen zur Laufzeit eingesetzt die evaluierte Vorlagen speichern und die Berechnung Eingabe von außen entgegennehmen können.
Einsatz von Verzweigungen	Verzweigungen innerhalb der Vorlage lassen sich einsetzen, um die Texterzeugung während der Befüllung nicht programmatisch zu steuern. Die Verzweigung enthält dabei ein Prädikat.
Verwendung von Schleifen mit Separatoren, vor- und nachgeschalteten Zeichen	Eine Schleife iteriert über einen Platzhalter, hinter dem eine Collection steckt. Für jedes Element lässt sich ein Zeichen zwischen den Iterationsschritten vor Eintritt in die Schleife oder nach Austritt aus der Schleife setzen, und damit bspw. ein abschließendes Semikolon ";" für Anweisungen, oder ein Komma "," für eine Parameterliste erzeugen.
Generische Modelleingabe für Vorlage	Die Vorlage lässt sich mit jedem Eingabemodell ausführen, solange die Platzhalterkonvention eingehalten wird.
Durchreichen von Erweiterungsmethoden für Collectionstypen	Es lassen sich die Erweiterungsmethoden zur Abfrage, Filterung und Veränderung einer Collection durchreichen und innerhalb der Vorlage nutzen.

Tabelle 4.1.: Essenzielle Anforderungen an die Texterzeugung für zugrundeliegende Template Engine mit ergänzten Punkten.

ein Platzhalter ist auch an die Eingabestruktur gekoppelt. Eingebettet sind diese in dafür vorgesehene Vorlagen in denen Konstrukte aus 4.1 wieder zu finden sind. Eine ausgefüllte Vorlage bildet die sequentielle Verkettung einer komplexen Zeichenkettenkonstruktion und baut zusammen mit anderen Vorlagen das Gesamtsystems auf. Eine Vorlage kann dabei eine Spezialisierung einer anderen Vorlage sein und auch andere Vorlagen überschreiben. Anstelle der übergeordneten Vorlage wird eine überschriebene Variante verwendet, solange die Parametertypen der verwendeten Platzhalter übereinstimmen und die Auswertungsbedingungen innerhalb der Vorlage erfüllt sind.

4.5. Validierung und Invarianten

Zur Lauffähigkeit der Konzepte sind jeweils verschiedene Invarianten definiert worden. Eine Teilmenge davon wird in Tabelle 4.2 beschrieben und sind in der Sprache als Validierungsregeln hinterlegt. Alle Regeln sind einer von drei Prüftypen - Information, Warnung und Fehler - zugeordnet und werden zur Sicherstellung von Übersetzungsvorgängen auf der statischen Semantik eingesetzt. Es wurde versucht einige essenzielle Invarianten für Kernkonzepte formal zu spezifizieren. Diese sind in Abschnitt 4.5.1 fortlaufend definiert und erläutert, dazu gleichbedeutend zu Fehlern, sollten sie nicht zutreffen. Als Beschreibung wurde auf die mathematische Notation und die Spezifikationssprache der Z-Notation zurückgegriffen O'Regan (2017). Deren Elemente weisen viele Analogien zu der für die MARS DSL entwickelten Workflow-Sprache auf und beinhalten essenzielle Konzepte zur Systemspezifikation. Neben den ergänzten Regeln zur MARS DSL in Tabelle 4.2 sind diverse weitere bereits in Glake (2016) und Glake (2017) spezifiziert worden, weswegen an dieser Stelle wieder darauf verwiesen wird.

4.5.1. Formale Invarianten

Eine formale Beschreibung der wesentlichen Invarianten folgt zur konkreten Festlegung essenzieller Regeln, die die Kernstruktur eines MARS DSL Modells darstellen. Zur formalen Spezifikation wird an dieser Stelle auf die mathematische Notation inklusive der dafür festgelegten Z-Notation (O'Regan (2017)) zurückgegriffen, die explizit dazu vorgesehen ist. Alle Invarianten werden zur Design Zeit geprüft und geben direkte Rückmeldung über deren Nichterfüllung, die andernfalls keine sichere Abbildung nach 4.4.2 versichern kann (Leroy (2014)). Jede Prüfung erfolgt dabei auf der statischen Semantik.

Eindeutigkeit von Entitäten

Innerhalb eines Modells müssen die Bezeichner eindeutig sein, um Duplikate zu vermeiden. Das gilt sowohl lokal als auch global, d.h. auch übergreifend zu anderen Ressourcen, für die definierten Agententypen, Objekttypen Layer usw.:

<i>Multiagentensystem</i>
$name : \mathbb{P} NAME$ $agents : \mathbb{P} Agent; layers : \mathbb{P} Layer; rasterLayers : \mathbb{P} RasterLayer;$ $tsLayers : \mathbb{P} TimeSeriesLayer; objects : \mathbb{P} Object$
$\forall a1, a2 : agents \bullet a1.name = a2.name \wedge a1.model = a2.model \Rightarrow a1 = a2$ $\forall l1, l2 : layers \bullet l1.name = l2.name \wedge l1.model = l2.model \Rightarrow l1 = l2$ $\forall t1, t2 : tsLayers \bullet t1.name = t2.name \wedge t1.model = t2.model \Rightarrow t1 = t2$ $\forall r1, r2 : rasterLayers \bullet r1.name = r2.name \wedge r1.model = r2.model \Rightarrow r1 = r2$ $\forall o1, o2 : objects \bullet o1.name = o2.name \wedge o1.model = o2.model \Rightarrow o1 = o2$

Auf den global typisierte Elemente gilt die Gleichheit über deren vollqualifizierenden Namen. Dieser setzt sich aus der Modellbezeichnung *model* und der individuellen Elementbezeichnung *element.name* zusammen. Die Gültigkeit dieser Invariante vermeidet Duplikate und dies auch übergreifend auf weitere Ressourcen, deren Modellname ebenfalls äquivalent ist. In [Glake \(2016\)](#) ist diese Bedingung bereits sichergestellt, jedoch wegen der Entkopplung von PIM und PSM ([Glake u. a. \(2017\)](#)) an dieser Stelle nochmals formal festgelegt. Da innerhalb einer Ressource die Vollqualifizierung gilt, bleibt es damit auch möglich Entitäten mit einem gleichen Namen zu versehen, solange sie sich in unterschiedlichen Modellteilen befinden.

Use und Alias Festlegung

Innerhalb eines Modells müssen die Bezeichner eindeutig sein, um Duplikate zu vermeiden. Das gilt sowohl lokal als auch global, d.h. auch übergreifend zu anderen Ressourcen, für die definierten Agententypen, Objekttypen Layer usw.:

<i>Use</i>
$use : \mathbb{P} Use$
$(use.layer \neq null \wedge use.model = null)$ $\vee (use.layer = null \wedge use.model \neq null)$ $use.layer \neq null \Leftrightarrow use.alias \neq null$

Der *use* bezieht sich damit entweder auf ein importiertes Teilmodell oder es wird explizit ein Layer importiert für den gilt, dass zugleich ein Alias vorliegen muss.

Aktiv-Passiv Interaktion

Aus der Interaktionslogik wird das Aktiv-Passiv Mapping abgeleitet. Diese repräsentieren in der Regel das Sender-Empfänger Modell für eine Einzelkanal Verbindung von einem Agenten $a1$ zu einem anderen Agenten $a2$. Der Nachrichtenaustausch findet mittels primitiven Nachrichtentypen oder Objekttypen statt und wird in Form einfacher Methodenaufrufe modelliert. Aus der Laufzeitsicht ist diese Beschränkung nicht wichtig, da das System auch mit Agententypen als austauschbares Format umgehen kann. Jedoch geht diese Nachrichten-basierte Synchronisation gegen das eigentliche Agentenparadigma (Wooldridge und Jennings (1995)) und auch kein übliches Interaktionsmuster (Aschermann u. a. (2016b)). Für die Aktiv-Passiv Interaktion gilt damit:

$\begin{aligned} & \text{AktivPassivInteraktion} \\ & \text{activeAction} : \mathbb{P} \text{ActiveAction} \\ & \text{passiveAction} : \mathbb{P} \text{PassiveAction} \end{aligned}$
$\begin{aligned} & \forall \text{messageParam} : \text{activeAction.messageParameters} \bullet \neg \text{messageParam} \in \mathbb{P} \text{Agent} \\ & \forall \text{messageParam} : \text{passiveAction.messageParameters} \bullet \neg \text{messageParam} \in \mathbb{P} \text{Agent} \end{aligned}$

Reaction Match

Die regelbasierte Beschreibung des *reaction* Konzepts erlaubt es unabhängig von der Reihenfolge der Regeldefinition ein Verhalten zu spezifizieren (Fowler (2010)), welches über das übliche Workflow-Paradigma hinaus geht und Agenten ausgehend einem aktuellen Systemzustand Aktionen ausführen lassen (Aschermann u. a. (2016b)). Eine nach Definition 4.14 spezifizierte *reaction* umfasst dabei immer einem *match* auf einen bestimmten Agententypen, zugleich gilt erneut wieder die Abhängigkeitsbeziehung des Aktor von dem aus diese Inferenz stattfinden soll.

Reaction

$reaction : \mathbb{P} \textit{Reaction}$

Δ

$actorLayer : \textit{Reaction} \rightarrow \textit{Layer}$

$potentialLayers : \mathbb{P} \textit{Layer}$

$potentialLayers == actorLayer.dependencies \cup$

$\cup \{i : actorLayer.dependencies \bullet i.dependencies^+\}$

$\forall matchAgent : reaction.matches \bullet matchAgent \in \mathbb{P} \textit{Agent}$

$\wedge matchAgent.layer \in potentialLayers$

Für alle zu erfassenden Agententypen aus *matches* wird geprüft, ob die Sichtbarkeitsbeziehung des zugrundeliegenden Layers *actorLayer* erfüllt ist und dass es sich jeweils um einen Agententypen handelt.

Ambiguität von Operationen und Feldern

Die Mehrdeutigkeit spezifizierter Operationen und Felder muss vermieden werden. Operationen werden dabei durch deren Signatur eindeutig beschrieben. Diese umfasst den Namen, die Parametertypen sowie den Rückgabotyp. Für Operationen gilt damit:

Operation

$element^+ : \mathbb{P}\{\textit{Object}, \textit{Agent}, \textit{Layer}, \textit{RasterLayer}, \textit{TimeSeriesLayer}\}$

$\forall o1, o2 : element.operations \bullet o1.name = o2.name$

$\wedge \#o1.parameters = \#o2.parameters$

$\wedge o1.type = o2.type$

$\wedge \forall i : 1..\# : o2.parameters(i).type = o2.parameters(i).type$

$\Rightarrow o1 = o2$

Für Felder gilt vereinfachend nur die Eindeutigkeit über den Namen. Agenten und Layer beherbergen dabei States, diese sind jedoch gleichbedeutend zu den Feldern, mit der Ausnahme dass sie eines der wesentliche Systemein- und Ausgabeschnittstellen darstellen.

Field

$element^+ : \mathbb{P}\{\textit{Object}, \textit{Agent}, \textit{Layer}, \textit{RasterLayer}, \textit{TimeSeriesLayer}\}$

$\forall o1, o2 : element.fields \bullet o1.name = o2.name \Rightarrow o1 = o2$

Parametrisierung und Observierung von States

In Invariante 4.5.1 sind die mentalen Agentenzuständen und Layerzustände zwar eindeutig, jedoch wird aus der Bedingung des resultierenden MARS-Zielsystems heraus, nicht möglich komplexe Typen zu observieren oder von außen zu parametrisieren. D.h. lediglich die elementaren Datentypen in der MARS DSL Sprache lassen sich observieren und von außen mit Startwerten belegen.

<i>State</i>
$s : \mathbb{P} \textit{State}$
$s.\textit{observe} \vee s.\textit{external} \wedge s.\textit{type} \notin \mathbb{P} \textit{Agent}$
$\wedge s.\textit{type} \notin \mathbb{P} \textit{Object}$
$\wedge s.\textit{type} \notin \mathbb{P} \textit{Layer}$
$\wedge s.\textit{type} \notin \mathbb{P} \textit{RasterLayer}$
$\wedge s.\textit{type} \notin \mathbb{P} \textit{TimeSeriesLayer}$

Hierarchisierung von Agenten

Als Teil des Vererbungskonzepts kann ein Agent (*agent*) nicht mehrfach Teil seiner eigenen Typhierarchie sein und damit einen Zyklus verursachen. Des Weiteren gilt, dass als aktive autonome Agenteneinheit diese sich innerhalb der Umgebung bewegen und andere Agenten wahrnehmen können. Diese Eigenschaft soll jedoch nicht für Objekte gelten. Die Definition einer Vererbungshierarchie mit einem Objekttyp als Teil der Hierarchie geht entgegen dem verbundenem Agentenparadigma (Ahlbrecht u. a. (2016)). Für die Invariante gilt damit:

<i>Agent_{Hierarchie}</i>
$\textit{agentType}^+ : \mathbb{P} \textit{Agent}$
$\Delta \textit{inheritance}^+ : \mathbb{P} \textit{Agent}$
$\textit{inheritance}^+ == \textit{agentType} \cup \bigcup \{i : \textit{agentType} \bullet i.\textit{superType}^+\}$
$\textit{agentType}^+ : \mathbb{P} \textit{Agent}$
$\forall \textit{type} \in \textit{inheritance}^+ \bullet \textit{type} \in \mathbb{P} \textit{Agent}$
$\wedge \textit{type}.\textit{layer} = \textit{agentType}.\textit{layerself} \notin \textit{inheritance}^+$

Für alle Elemente der Vererbungskette gilt, das ein Agent nicht erneut Teil dieser Kette sein kann. Darüber hinaus stellen alle Elemente wiederum Agenten dar und keine Objekttypen und sind konsistent zum definierten Layer des betrachteten *agentType*.

Hierarchisierung von Objekten

Für die Vererbungshierarchien von Objekttypen gilt die ähnliche Invariante, mit dem Unterschied dass in deren transitiven Hüllen keine Agententypen vorliegen. Für die Hierarchie gilt damit:

$\begin{aligned} & \text{Object} \\ & \text{objectType}^+ : \mathbb{P} \text{Object} \\ & \Delta \text{inheritance}^+ : \mathbb{P} \text{Object} \end{aligned}$
$\begin{aligned} & \text{inheritance}^+ == \text{objectType} \cup \bigcup \{i : \text{objectType} \bullet i.\text{superType}^+\} \\ & \text{objectType}^+ : \mathbb{P} \text{Object} \\ & \text{self} \notin \text{inheritance}^+ \\ & \forall \text{type} \in \text{inheritance}^+ \bullet \text{type} \in \mathbb{P} \text{Object} \end{aligned}$

Die Menge der Typen der Vererbungshierarchie enthält nicht den Typen selbst und außerdem sind alle Typen wiederum ein Agententyp.

Über die Parameter der aufgerufenen Operation $o.parameters$ wird die Summe mittels Distanzfunktion d ermittelt.

Mehrdeutigkeit von Aufrufen

Das Fehlen von Mehrdeutigkeiten in Operationen reicht für die eindeutige Identifizierung eines Elements nicht aus. Sie muss auf Aufrufe erweitert werden. Hierzu wird eine Distanzfunktion $d : \text{Agent} \cup \text{Object} \rightarrow \mathbb{N}$ genutzt, die aus der Typhierarchie die Tiefe des gegebenen Typs zum aufgerufenen Parametertyp bestimmt:

$$n = \#o.parameters \sum_{i=0}^n d(parameters(i).type) \tag{4.1}$$

Für ein Aufrufziel wird das Minimum der geringsten Typdistanz ermittelt, d.h. aus der Tiefe des Vererbungsbaumes wird der Abstand zum aktuell betrachteten Parametertyps $parameters(i).type$ ermittelt und dieser Wert zusammen mit potenziell anderen Parametern in einer Summe zusammengefasst.

Vermeidung zyklischer Layer

Ähnlich der Zyklen in Vererbungshierarchie von Agenten 4.5.1 und Objekten 4.5.1 gilt diese für Layer zum Festlegen sichtbarer Agenten, erweitert jedoch auf die Menge assoziierten Layer:

Layer $\text{layer} : \mathbb{P} \text{Layer}$ Δ $\text{visibleLayers} : \mathbb{P} \text{Layer}$
$\text{visibleLayers} == \text{layer.dependencies} \cup$ $\quad \cup \{i : \text{layer.dependencies} \bullet i.\text{dependencies}^+\}$ $\text{layer} \notin \text{visibleLayers}$

Für die Invariante gilt, dass in der transitiven Hülle der Layer-Abhängigkeiten nicht der Layer selbst enthalten ist. Damit werden Zyklen in den Referenzen vermieden und der Interaktionsgraph ist ein gerichteter Graph.

Exploration von Agenten

Das Wahrnehmen von Agenten ist nur auf Agententypen anwendbar. Des Weiteren muss der zu wahrnehmende Typ in der Sichtbarkeits-Beziehung des zugrundeliegenden Layers sein.

Exploration $\text{explore} : \mathbb{P} \text{Exploration}$ Δ $\text{targetLayer} : \mathbb{P} \text{Layer}$ $\text{actorLayer} : \mathbb{P} \text{Layer}$ $\text{potentialLayers} : \mathbb{P} \text{Layer}$
$\text{explore.agentType} \neq \text{null} \Rightarrow$ $\quad \text{explore.agentType} \in \mathbb{P} \text{Agent}$ $\quad \text{targetLayer} == \text{explore.agentType.layer}$ $\quad \text{actorLayer} == \text{explore.actor.layer}$ $\quad \text{potentialLayers} == \text{actorLayer.dependencies} \cup$ $\quad \quad \cup \{i : \text{actorLayer.dependencies} \bullet i.\text{dependencies}^+\}$ $\quad \text{targetLayer} \in \text{potentialLayers}$ $\text{explore.coordinate} \neq \text{null} \Rightarrow \text{explore.coordinate} \in \mathbb{Z} \times \mathbb{Z}$ $\text{explore.where} \neq \text{null} \Rightarrow$ $\quad \# \text{explore.where.parameters} = 0$ $\quad \vee \# \text{explore.where.parameters} = 1$ $\quad \wedge \text{explore.where.parameters}(1) = \text{explore.type}$

Die Invariante sagt aus, dass für einen gegebenen zu explorierenden Agententypen explore.agentType dieser zunächst auch in der globalen Agentenmenge vorliegen muss. Zudem gilt, dass aus dem Akteur der Exploration explore.actor nach Definition 4.15 der zugrundeliegende Agenten-Layer

actorLayer in seiner spezifizierten Abhängigkeitsbeziehung der *dependencies* aus Definition 4.5 den Ziellayer *targetLayer* kennen muss. Das bezieht sich darauf, dass daraus später die Abhängigkeitsauflösung vorgenommen wird, da diese nach dem MARS-Layermodell (Hüning u. a. (2016)) immer in entsprechender Verbindung stehen, je nachdem wie deren Interaktions-Richtung verläuft, in diesem Fall eine gerichtete sensorische Interaktion. Wird eine Exploration von einem besonderen Punkt, d.h. es gibt eine explizite Ausgangskordinate *explore.coordinate* entspricht sie einer ganzzahligen Koordinate die vom Typsystem geprüft wird.

Für das Prädikat *explore.where* gilt zudem eine Besonderheit, dass sich auf die *explore.agentType* Parametrisierung stützt. Unter der Bedingung, dass ein solches Prädikat existiert gilt in der MARS DSL zusätzlich, dass entweder kein Parameter vorliegt und damit der Typ aus dem explorierten Typs automatisch abgeleitet wird oder er wurde explizit definiert und entspricht einem Typ der in Relation zur Vererbung des *explore.agentType* steht.

Bewegung von Agenten

Das Wahrnehmen von Agenten ist nur auf Agententypen anwendbar. Des Weiteren muss der zu wahrnehmende Typ in der Sichtbarkeits-Beziehung des zugrundeliegenden Layer sein.

<i>Movement</i>
<i>move</i> : \mathbb{P} <i>Movement</i>
Δ
<i>targetLayer</i> : \mathbb{P} <i>Layer</i>
<i>actorLayer</i> : \mathbb{P} <i>Layer</i>
<i>potentialLayers</i> : \mathbb{P} <i>Layer</i>
$(move.coordinate \neq null \wedge move.agentType = null)$ $\vee (move.coordinate = null \wedge move.agentType \neq null)$
<i>move.agentType</i> $\neq null \Rightarrow$
<i>move.agentType</i> $\in \mathbb{P}Agent$
<i>targetLayer</i> $== move.agentType.layer$
<i>actorLayer</i> $== move.actor.layer$
<i>potentialLayers</i> $== actorLayer.dependencies \cup$
$\bigcup \{i : actorLayer.dependencies \bullet i.dependencies^+\}$
<i>targetLayer</i> $\in potentialLayers$
<i>move.coordinate</i> $\neq null \Rightarrow$
<i>move.coordinate</i> $\in \mathbb{Z} \times \mathbb{Z}$
<i>move.steps</i> $\in \mathbb{Z}$

Für das Movement von Agenten gilt, aus zielgerichteten Bewegungen in Richtung eines Typs, dass dieser aus der Menge der Agenten stammen muss. Es gilt zudem die gleiche Sichtbarkeitsbedingung wie bereits zur Exploration in 4.5.1

Erzeugung von Agenten

Die Agentenerzeugung ist namensgebend nur auf Agententypen anwendbar, dessen spezifizierte Koordinate innerhalb des diskreten Zahlenbereichs liegt. Erzeugt werden können zudem nur Agenten, die sich nach der Invariante 4.5.1 und 4.5.1 in der jeweiligen Sichtbarkeitsbeziehung befinden.

<p><i>Spawn</i></p> <p>$spawn : \mathbb{P} \text{Spawn}$</p> <p>$\Delta$</p> <p>$targetLayer : \mathbb{P} \text{Layer}$</p> <p>$actorLayer : \mathbb{P} \text{Layer}$</p> <p>$potentialLayers : \mathbb{P} \text{Layer}$</p> <hr style="width: 20%; margin-left: 0;"/> <p>$(spawn.coordinate \neq null \wedge spawn.agentType = null)$ $\vee (spawn.coordinate = null \wedge spawn.agentType \neq null)$</p> <p>$spawn.agentType \neq null \Rightarrow$ $spawn.agentType \in \mathbb{P} \text{Agent}$ $targetLayer == spawn.agentType.layer$ $actorLayer == spawn.actor.layer$ $potentialLayers == actorLayer.dependencies \cup$ $\cup \{i : actorLayer.dependencies \bullet i.dependencies^+\}$ $targetLayer \in potentialLayers$</p> <p>$spawn.coordinate \neq null \Rightarrow$ $spawn.coordinate \in \mathbb{Z} \times \mathbb{Z}$ $spawn.from \in \mathbb{Z}$ $spawn.until \in \mathbb{Z}$</p> <p>$spawn.action \neq null \Rightarrow$ $spawn.action.coordinate \in \mathbb{Z} \times \mathbb{Z}$</p>

Für den *Spawn* von Agenten gilt ähnlich zum *Movement*, dass sich die Erzeugungsrichtung an die Sichtbarkeitsbedingung aus der Layer-Abhängigkeit richtet und nicht jeder Agententyp erzeugt werden kann. Dazu gilt, dass sich die Zielkoordinate, falls vorhanden aus der Relation $\mathbb{Z} \times \mathbb{Z}$ stammen muss, um dem diskreten Grid zu genügen. Dazu folgt, dass der aus der Definition 4.17 formulierte Zeitintervall ebenfalls eine Ganzzahl \mathbb{Z} darstellen muss, da hierüber der Tick

angegeben wird. Umgekehrt lässt sich durch Projektion $Tick(t) : Time \rightarrow \mathbb{Z}$ eine Zeitmarke in den entsprechenden Tick überführen.

4.5.2. Informelle Invarianten und Prüfungen

Weitere wichtige Invarianten die aus Gründen der Verständlichkeit informell gefasst sind, wurden in Tabelle 4.2 gesammelt. Diese sind allesamt in Regeln gefasst, die wie zu den formellen Regeln 4.5 zur Design Zeit geprüft werden und dem Modellierer eine direkte Rückmeldung über sein aktuelles Modell gibt. Diese Regeln wurden dazu in die drei Regeltypen - Information, Warnung und Fehler - eingeteilt, deren Einteilung sich in Sprachen und darauf aufbauende Analysewerkzeuge bewährt hat Bettini (2016). Alle formalen Invarianten in Abschnitt 4.5 sind dazu als Fehler-Regel aufzufassen.

- **Information** - Gibt bei Erfolg eine Information an der jeweiligen Stelle des geprüften Elements an, die zur Rückmeldung beim jeweiligen Modellierer dient. Diese können beispielsweise zur alternativen Modellierung herangezogen werden und die Nutzung vorhandener Sprachkonstrukte fördern.
- **Warnung** - Eine Warnung ist ein potenzieller Fehler, bei dem zwar das Modell übersetzungsfähig bleibt, jedoch die Auflösung der Warnung empfohlen wird, um mögliche Semantische- und Laufzeitfehler zu vermeiden (Voelter u. a. (2013)).
- **Fehler** - Ist eine Angabe um einen identifizierten Verstoß der geforderten Invarianten zu erkennen. Der Workflow zur weiteren Verarbeitung wird nicht weiter fortgeführt und damit der Model-to-Code Prozess und der Optimierer nicht angestoßen (Bettini (2016)).

Neben der Ergänzung von Sprachkonzepten zählen vor allem Prüfungen zur Sicherstellung der Codegenerierung zu den wichtigsten Aspekten dieses Projekts, um eine gemäß den Bedingungen zur Lauffähigkeit der Simulationsmodelle und der genutzten unterliegenden Sprache gerecht zu werden. Unter anderem wurde hinsichtlich des Metamodel-Austauschformats aus Glake u. a. (2017) eine Reihe von Validierungsregeln definiert, die sich später auch in der Prüfungsphase in 5.1 und der damit verbundenen Komponente wiederfinden soll. Dabei wurden folgende fachliche Prüfungen gesammelt:

4.6. Analyse von Modelleigenschaften

Ein bedeutender Vorteil durch den Einsatz einer eigenen externen DSL liegt in der Analyse der statischen Semantik, im Wesentlichen zur Prüfung von Invarianten (Leroy (2014)). Diese spezi-

Regeltyp	Regel	Beschreibung
Information	Mögliche Überlagerung bei Vergleichsoperationen von Referenz- zu Wertetypen	Ein Referenzvergleich für Wertetypen oder umgekehrt wurde modelliert der ggf. jedoch nicht gewollte ist.
Fehler	Überwachung der Anzahl von Methodenargumenten mit jenen des Aufrufs	Vergleichs der Argumente innerhalb des Aufrufs, mit jenen definierten Argumenten beim Aufrufziel.
Fehler	Einhaltung von Typkonventionen der Prädikatenlogik	Prüfung der Prädikate darauf, dass deren Rückgabe immer einen aussagenlogischen Ausdruck zurückgeben..
Fehler	Einhaltung von Typkonventionen oder Konformitätsregeln, bei typisierten und nicht typisierten Methoden	Prüfung der Rückgabe mit dem dazu assoziierten Rückgabewert innerhalb der Methode und jenem des Aufrufs.
Fehler	Überwachung auf nicht lose gekoppelte Ausdrücke	Prüfung darauf, dass keine losen Ausdrücke enthalten sind, die in keinem Kontext, entweder einer Zuweisung oder eines Aufrufs, stehen.
Fehler	Sensor Abhängigkeit enthält keine Zyklen	Strukturprüfung, ob in der Sensordefinition eine zyklische Abhängigkeit vorliegt und damit einen Bau unmöglich macht.
Fehler	Methodenaufruf auf einem Layer State	Ein State wird in Form einer Methode mit Klammerung aufgerufen und es wird versucht diesen Aufruf ggf. zu parametrisieren (syntaktische Korrektur).
Fehler	Agenten-spezifische Operationen aus dem Agenten Kontext	Operationen die zur Neupositionierung, Exploration gedacht sind, müssen innerhalb eines Agenten Kontexts passieren, Agent positioniert und exploriert selbst oder wird durch andere Agenten positioniert oder exploriert.

Tabelle 4.2.: Ergänzte informelle Regeln zur Prüfung statischer Semantik.

fizieren jedoch nur Zustände die zur korrekten Übersetzung erfüllt sein müssen, also zwingend erforderlich sind. Umgekehrt gilt, dass auch Zustände identifiziert werden können, zu denen besondere Umstände existieren. Aus den MARS DSL Modellen wurde daher eine Reihe von Eigenschaften analysiert, die aus der statischen Semantik folgt und aus denen modellabhängige Entscheidungen getroffen werden können. Nachfolgend beschriebene Eigenschaften werden zur Forcierung bestimmter Ausführungselemente verwendet. Dazu zählt der Einsatz unterschiedlicher Umgebungen *QuadTree*, *Grid*, *Kd-Tree*, *kartesisch*,...), die Zwischenspeicherung von Werten oder Abfrageergebnissen, innerhalb eines Ticks, der massive Einsatz von Method Inlining oder gar dem Wegfall nicht benötigter Elemente (Aho (2008)). Zudem werden daraus Informationen über Abhängigkeiten klar und damit Information für das Erzeugen Kompilats sichtbar. Der Optimierer des Übersetzers umfasst eine kleine Teilmenge von Semantik erhaltenen Transformationen, die durch Umstellung einen erhofften Mehrertrag entweder zur Speicher-, oder Laufzeiteffizienz bieten. Diese Eigenschaften lassen sich entgegen der bisherigen Lösung über C# leichter identifizieren, da Elemente die zur Feststellung dieser Eigenschaft führen, Teil der abstrakten Syntax, in Abschnitt 4.3, sind (Aschermann u. a. (2016a)).

4.6.1. Agenten Eigenschaften

Für den Agenten können verschiedene Eigenschaften identifiziert werden, die zusammen mit anderen Agenten das System beeinflussen. Nachfolgend sind verschiedene dazu aufgefasst die zwar alle für die statische Semantik gelten jedoch in Bezug zur echten Ausführung dennoch niemals erfüllt sein müssen. Da erst durch Auswertung der Eingabe festgestellt werden kann, ob bestimmte Bedingungen weiterhin gelten. Die Eigenschaften stehen daher in Bezug zum globalen Agenten.

Statischer und dynamischer Agent

Handelt es sich um einen beweglichen Agenten so wird im Workflow irgendwann eine *move* Aktion ausgeführt. Da erst hierdurch eine Neupositionierung erfolgt, kann das Bewegen daran festgehalten werden:

$$\begin{aligned} agentType &: \mathbb{P} Agent \\ existsmovementExp &: \mathbb{P} Move \bullet \\ movementExp.actor &= agentType \end{aligned} \tag{4.2}$$

Beinhaltet das Agentenmodell eine Bewegungsaktion wie nach Definition 4.3.5, so kann dieser Typ als beweglicher Typ aufgefasst werden. Für das resultierende Environment gilt folglich, dass bspw. der Einsatz nicht balancierte Datenstrukturen (*Kd-Bäume*) einen negativen Einfluss

auf die Laufzeit haben (Russell und Norvig (2002)) kann. Es bietet sich an auf Strukturen zurückzugreifen die mit Änderungen kostengünstig umgehen können *grid*, *quadtree*, *kartesisch*. Ergänzend sei erwähnt das die Häufigkeit der Bewegung jedoch nicht feststellbar ist. Zum einen liegt mit der globalen Parametrisierung der Tick-Frequenz f_t und de Zeitraum $t_0 \rightarrow t_n$ die Möglichkeit vor, den Aktivierungszeitpunkt des individuellen Agenten zu beeinflussen, zum anderen haben lokale Systemzustände zum Zeitpunkt t_i immer irgendeinen Einfluss auf die Agentenaktionen, weswegen hier die bekannte Trace Semantik viele Anhaltspunkte bietet Leroy (2014).

Wahrnehmbarer und nicht wahrnehmbarer Agent

Ähnlich zu in 4.6.1 feststellbaren Bewegungsdynamik kann allein durch Identifizierung von Explorationen durch den eigenen oder fremde Agententypen die Eigenschaft sichergestellt werden, ob ein Agent überhaupt durch andere wahrgenommen wird. Ein Agent ist genau dann *explorierbar* wenn gilt:

$$\begin{aligned}
 agentType^+ &: \mathbb{P} Agent \\
 \forall type &: \mathbb{P} Agent \bullet \\
 &\exists exploreExp : type \bullet exploreExp.agentType = type \\
 &\wedge type = agentType^+
 \end{aligned} \tag{4.3}$$

Aus der globalen Agentenmenge existiert ein Exploration *exploreExp* dessen verknüpfter Agententyp *agentType* dem aktuell betrachteten Type *agentType*⁺ entspricht. Da die Wahrnehmung von außen kommt und nicht Teil der Agentenbeschreibung selbst, müssen alle Typen betrachtet werden. Erneut gilt wieder, dass zwar die potenzielle Wahrnehmung als Eigenschaft festgestellt wird, jedoch eine tatsächliche Exploration nie stattfinden muss, sollte es keinen möglichen Trace geben der diesen Systemzustand erreicht.

Erzeugender Agent

Es lässt sich die Eigenschaft ableiten ob ein Agent innerhalb des Modells erzeugt wird. Diese vornehmlich aus bekannten objektorientierten Sprachen identifizierte Eigenschaft entspricht dem der Instanziierung eines Objektes und ist in der MARS DSL durch die Erzeugungslogik

4.3.5 definiert. Im Agentenmodell wird dazu lediglich nach diesem Konzept gesucht, ob ein gegebener Agent *targetAgent* auch der Zieltyp einer Erzeugungsaktion ist.

$$\begin{aligned} agentType^+ : \mathbb{P} Agent \\ \exists spawn : agentType^+ \bullet spawn.agentType = targetAgent \end{aligned} \quad (4.4)$$

Das eine Erzeugung immer im Kontext einer Bedingung geschieht, trivialerweise *true* sofern dieser von keiner bedingten Verzweigung umgeben ist, kann zwar erneut eine Erzeugung analysiert werden, jedoch nicht mit Bestimmtheit die tatsächliche Erzeugung sichergestellt werden, es sei denn es gilt:

$$\begin{aligned} agentType^+ : \mathbb{P} Agent \\ (\exists spawnExp : agentType^+ \bullet spawnExp.agentType = targetAgent \\ \wedge ((agentType.reflex.steps(\#agentType.reflex.steps) = spawnExp) \\ \vee (agentType.reflex.steps(1) = spawnExp))) \end{aligned} \quad (4.5)$$

Scheidender Agent

Das Ausscheiden eines Agenten aus dem System ist eine gesonderte Eigenschaft die neben dem Auftreten eines besonderen mentalen *IsAlive* Zustands innerhalb des Agenten auch dessen Zuordnung notwendig macht. Aus dem Modell lässt sich ableiten ob eine solche Zuweisung überhaupt existiert. Ein potenzielles Dahinscheiden ist somit genau dann gegeben wenn:

$$\begin{aligned} agentType^+ : \mathbb{P} Agent \\ \exists state : agentType.states \bullet state = IsAlive \\ \wedge (\exists assignmentExp : agentType^+ \bullet assignment.feature = state \\ \wedge isFalseLiteral(assignment.value)) \end{aligned} \quad (4.6)$$

Ein Agent *agentType⁺* wird als potenziell ausscheidender Agent deklariert, wenn er zunächst einen mentalen Zustand *IsAlive* aufweist. Existiert dieser Zustand und wird ihm eine Zuweisung *IsFalseLiteral* zugeordnet, ist an dieser Stelle sichergestellt, dass dieser Agent wenigsten den Zustand des Ausscheidens kennt. Ob er diesen erreicht hängt dagegen wieder von vorangegangenen und nachfolgenden Aktionen ab. Aus diesem Grund ist diese Eigenschaft nur von potenzieller Natur. Begründet wird der Umstand zusätzlich um die Tatsache ,dass die Aktivitäten anderer Agenten das Erreichen oder Nichterreichen diese Zustands ebenfalls beeinflussen. Existiert jedoch kein Aktiv-Passiv Mapping und damit auch keine Interaktion

zwischen dem $agentType^+$ und irgendeinem anderen, kann man dieses Ausscheiden explizit festmachen:

$$\begin{aligned}
 agentType^+ &: \mathbb{P} Agent \\
 &\exists state : agentType.states \bullet state = IsAlive \\
 &\wedge (\exists assignmentExp : agentType^+ \bullet assignment.feature = state \\
 &\wedge isFalseLiteral(assignment.value) \\
 &\wedge ((agentType.reflex.steps(\#agentType.reflex.steps) = assignmentExp) \\
 &\quad \vee agentType.reflex.steps(1) = assignmentExp))
 \end{aligned} \tag{4.7}$$

Der Unterschied zum Prädikat 4.6 ist der Zusatz $agentType.reflex.steps(\#agentType.reflex.steps) = assignmentExp$. Dieser prüft auf die Sequenz des Agententicks, ob die letzte Aktion eine Zuweisung auf das *IsAlive* darstellt.

4.6.2. Layer Eigenschaften

Die abgeleiteten Layer Eigenschaften umfassen verschiedene Aspekte zur Agentenverwaltung oder fehlender Verwendung. Als Gruppierung und Grundlage der Existenz von Agenten können diese zur Auslagerung eingesetzt werden oder zur speicher-synchronisierten Interaktion zwischen den Agenten.

Nicht besetzte Layer

Leere Layer repräsentieren jegliche Layertypen die keinerlei Verwendung haben. Dazu zählt das Fehlen von Agenten auf diesem Layer als auch fehlende Referenzen zu ausgelagerten Datenfeldern oder Operationen darauf. Würde sich die Eigenschaft beschränken und allein auf die fehlende *use* Referenz beziehen, wüsste man zwar um die fehlende Verwendung ausgelagerter Elemente, jedoch ist die Erzeugung betroffener Agenten ein externer Parameter, auf den kein Einfluss genommen werden kann. Für leere Layer wird die Bedingung daher stringenter definiert:

$$\begin{aligned}
 agentType^+ &: \mathbb{P} Agent \\
 layerType^+ &: \mathbb{P} Layer \\
 &\forall agentType : \mathbb{P} Agent \bullet agentType.layer \neq layerType
 \end{aligned} \tag{4.8}$$

Ein betrachteter Layer $layerType^+$ gilt als kein Agent sofern dieser seine Layer Referenz vorhält. Da die Existenz eines Agenten an ein Layer gebunden ist, resultiert dies auch in keiner Inkonsistenz. Erst durch einen gekoppelten Layer werden Zugriffe auf die Interna möglich.

Abhängigkeit von Layern

Wertvolle Eigenschaften wie Deadlock Freiheit, Lebendigkeit, Beschränktheit oder Reversibilität (Russell und Norvig (2002)) lassen sich nicht aus der statischen Semantik ableiten, da diese Information allein durch Einbeziehung von Eingaben möglich ist (Leroy (2014)), entweder mittels der Trace-Semantik berechneten Übergangsgraphen (Ahlbrecht u. a. (2016)) oder aus Petri-Netzen ähnlichen Erreichbarkeitsgraphen (Alves-Foss (1999)). Stattdessen lassen sich bekannte Abhängigkeitsgraphen aus der Software Entwicklung konstruieren die unter anderem auf nicht verwendete Elemente hinweisen.

$$\begin{aligned}
 layerType^+ : \mathbb{P} Layer \\
 & layerType.dependencies \cup \\
 & bigcup\{i : layerType.dependencies \bullet i.dependencies^+\}
 \end{aligned}
 \tag{4.9}$$

Die am Layer hängenden Abhängigkeiten $dependencies$ geben bereit den Abhängigkeitsgraph. Aus der transitiven Hülle folgt der globale Abhängigkeitsgraph, der neben Zyklenidentifikation über Invariante 4.5.1 auch zur Bildung des Zusammenhangsgraphen.

Bezeichnung	Beschreibung	Begründung
Ein Mapping gilt nur von aktiven zu passiven Aktionen	Prüfung darauf, dass das Mapping die Definition $Mapping = \{(a, p) \mid a \in Active \wedge p \in Passive\}$ einhält	Die Notwendigkeit liegt darin die Generierung zu vereinfachen, indem hierdurch die Abhängigkeiten zwischen den Typen für den Aufruf sichergestellt werden und damit auch gesetzte Abhängigkeiten innerhalb der zu erzeugenden Projekte. So sind bspw. zyklische Abhängigkeiten darüber auflösbar.
Sicherstellung von nicht vorhandene Duplikaten von Modellelementen	Diese Prüfung wird bereits durch die Sprache sichergestellt, ist jedoch wegen der Entkopplung von PIM und PSM (Glake u. a. (2017)) an dieser Stelle erneut notwendig. Somit gilt für alle Elemente der Metamodell-Instanz, dass deren Id's und Namen eindeutig sein müssen. Dies gilt jedoch nicht zwangsläufig für die Namen, sondern lediglich für die Elemente die innerhalb eines Typs definiert sein müssen.	Notwendig wird aus der darunterliegenden Sprache, die entsprechend eine Eindeutigkeit bei der Referenzierung benötigt.
Prüfung auf syntaktische Korrektheit der vorhandenen Logikblöcke	Eine syntaktische Prüfung von möglichem vorhandenem Code innerhalb der Verhaltensbeschreibung.	Folge aus der Beschreibung, fehlerhafter Code lässt sich nicht korrekt übersetzen.
Prüfung darauf, dass Agenten auf BASIC Layern liegen	Die Trennung von MMM und AMM (Glake u. a. (2017)) erlaubt die beliebige Zuordnung von Agenten zu einem Layer. Es muss hierbei jedoch gewährleistet sein, dass der Layer die Verantwortlichkeit eines Agenten-Layers besitzt.	Ein korrespondierender Typ wird die Agenten verwalten und ihnen die Möglichkeit geben sich innerhalb der Ausführungsumgebung dynamisch an- und abzumelden, um somit periodisch angestoßen zu werden. Andere Typen bieten diese Registrierung nicht.

Tabelle 4.3.: Validierungsregeln zur Sicherstellung von konformen MARS Modellen.

5. Entwurf

Der folgende Abschnitt befasst sich mit Entwurfsergebnissen, die sich aus der in 4 untersuchten Anforderungen ergeben haben und beschreibt die konzeptionellen und architektonische Lösung für die MARS DSL. Wie bereits in Glake (2016) und Glake (2017) werden die von Starke (2015) vorgeschlagenen Sichten genutzt, um eine überschaubare Struktur für das zu entwickelte Gesamtsystem zu schaffen. Ähnlich der in Glake (2016) enthaltenen Generatorsicht, wird diese infolge der Erweiterung der Sprache mit mehreren Ressourcen umzugehen, erneut wieder eingesetzt. Sie wird verwendet um die jeweils erzeugten Artefakte näher zu beschreiben. Des Weiteren wird ein Fokus auf die globale Sicht des verfeinerten Generierungsworkflows gesetzt, dessen Ergebnis großen Ausmaß auf die gesamtheitliche Systemarchitektur hat. Von außen wird eine MARS DSL Verarbeitung nach Abbildung 5.1 wahrgenommen, die damit das MDA aus Glake u. a. (2017) und damit den modellgetriebene Ansatz (Belaunde u. a. (2003)) umgesetzt.

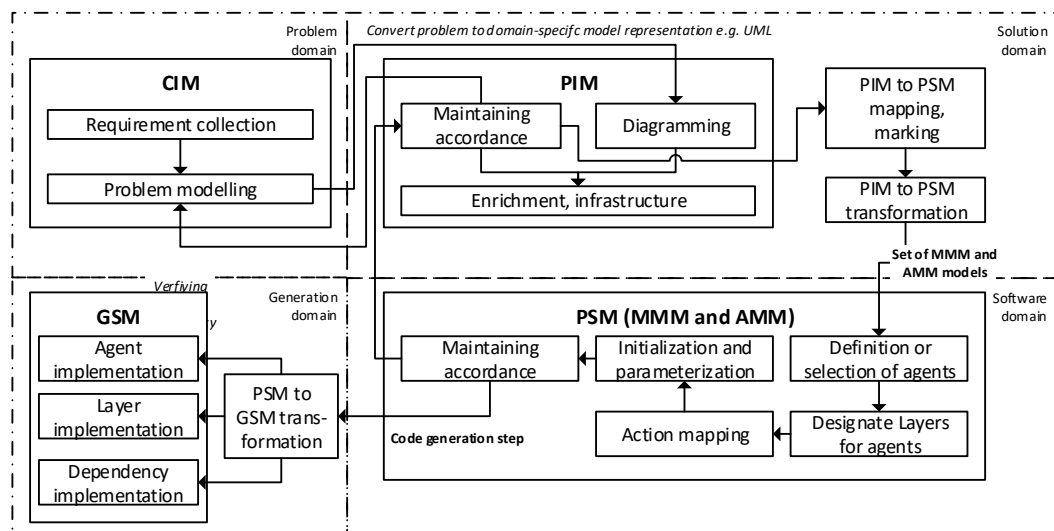


Abbildung 5.1.: Fachliche Außensicht auf das MDA nach Glake u. a. (2017).

Für den Modellierer wird zunächst die eigene offene Forschungsfrage (CIM) mit ggf. Daten und einer Idee in den Fokus gesetzt. Die Überführung in das konzeptionelle Modell wird anschließend durch MARS DSL durchgeführt, die andernfalls mit Standardwerkzeugen aka UML, ER-Diagrammen, oder eben einer eigenen Notationsform vorgenommen wird. Dieses PIM) wird automatisch in ein äquivalentes MARS-LIFE (Hüning u. a. (2016)) Zielsystem übersetzt, was damit auch die Abhängigkeit zu dieser Umgebung schafft (*platform specific model* PSM) und kann dort zur Ausführung gebracht werden. Das GSM stellt letztlich die Implementierung des Modell in C# Code (Hejlsberg u. a. (2003)) dar, das ausgeführt werden kann. Da die MARS DSL ein vollständiges PIM darstellt ist diese entsprechend unabhängig vom Zielsystem und kann bei Einführung eine neuen Abbildung auf ein anderes Zielsystems leicht auf unterschiedliche Plattformen portiert werden (Mohagheghi u. a. (2013)). Da der Fokus auf die MARS-Laufzeitumgebung gesetzt ist und dazu einige Optimierungen zur Verbesserung von Laufzeit- und Speichereffizienz vorgenommen wurde, ist eine äquivalente Abbildung nur schwer möglich (Grignard u. a. (2013)) (Wilensky (2015)), zumal die Hostsprache von MARS zur Übersetzung der MARS DSL Konzept besser geeignet geeigneter erscheinen.

Aus technischer Perspektive wird vom Benutzer die Systemumgebung und Pipeline nach Abbildung 5.2 wahrgenommen. Diese zeigt die technische Außensicht auf die Verarbeitung der MARS DSL Ressourcen bis hin zur Ausführung mit dem MARS-LIFE (Hüning u. a. (2016)) System inklusive gesonderter Werkzeuge zur Analyse und dem Aufsetzen von Experimenten.

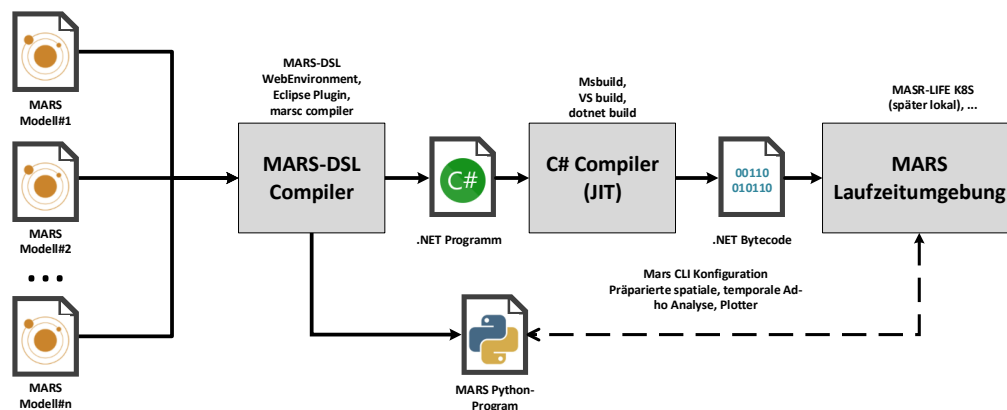


Abbildung 5.2.: Technische Außensicht auf die Übersetzungselemente und Reihenfolge der Verarbeitung bis zur Ausführung von MARS DSL Ressourcen.

Einzelne Modellteile werden über den MARS DSL Compiler in nativen C# Code übersetzt, der den Konformitätsbedingung zur Ausführbarkeit und damit auch die entsprechende Spezifikation der Zielsprache (Hejlsberg u. a. (2003)) erfüllt. Nebenbei werden eine Reihe von Python Rossum (1995) Skripten erzeugt die dazu zwei Aufgaben erfüllen. Zum einen wird eine Menge vordefinierter Zeitreihenanalysen auf das Modell vorbereitet (Drogoul u. a. (2002)) die sowohl abhängig als auch unabhängig der Observierung aus den States 4.9 in den Agenten verschiedene Analyse und Plots erzeugen. Zum anderen enthalten die Skripte vorbereitende Schritte des MARS Kommandozeilenwerkzeugs zum Bau eines MARS-Modells und Uploads. Dieser Bau stellt den Anstoß des C# JIT-Compiler dar und übersetzt das erzeugte Zielsystem in sein natives Bytecodeformat (Aho (2008)), dass von der MARS-LIFE Laufzeitumgebung ausgeführt werden kann, inklusive der verbundenen Erzeugung von Szenarien, Ausgabekonfigurationen und mehrfachen Simulationsläufen Hüning u. a. (2016).

5.1. Generierungsworkflow

Ein abgewandelter Workflow aus Glake u. a. (2017) wurde definiert, der sich auf die Erzeugung und den Bau der hochladbaren Modellartefakte beschränkt. Dabei wird eine Erzeugung vorgenommen womit anschließend das Modell an ein assoziiertes Modellierungsprojekt übergeben wird und über die vorherrschenden Werkzeuge in die MARS-Cloud hochgeladen werden kann. Abbildung 5.3 zeigt den abgewandelten Workflow zu Glake (2017) der die internen Schritte darstellt, die während der Übersetzung durchgeführt werden.

Zu Beginn wird das eingehende PIM empfangen und innerhalb eines dafür bereitgestellten Repositories abgelegt (Leroy (2014)). Dies hat den Sinn ähnlich der Datenabfrage einer Datenbank schnell von jeden Komponenten auf das ganzheitliche Modell, als Integrationslösung, zuzugreifen. Eine anschließende Validierung mit denen in 4.3 definierten Prüfungen wird durchgeführt und im Falle einer Nichterfüllung zwingender Invarianten, in Abschnitt 4.5.1 das Modell aus dem gemeinsame Repository entfernt sowie die nicht erfolgreichen Prüfungen mit entsprechenden Fehlermeldung zurückgegeben (Voelter u. a. (2013)). Sollte ein valides Modell vorliegen, wird es um verschiedene semantische Elemente angereichert. Die Anreicherung des Modells bezieht sich darauf, die losen Referenzen der zugrundeliegenden und fixen Typ-0 Schnittstellen nun zu ersetzen und die jeweiligen konkreten Entitätstypen (Typ-X) (*Agent, Layer, Method, Active-Action,...*) direkt zu referenzieren, ohne zunächst aufwändig die zugehörige technischen Auflösungen wiederkehrend durchführen zu müssen (Herrington (2003)). Das gilt sowohl für die Stellvertretertypen eines Layers, bezeichnet als *AgentReference*, als auch für das Aktiv-Passiv Mapping was losgelöst im MMM vorzufinden ist, sowie dessen Ausprägung

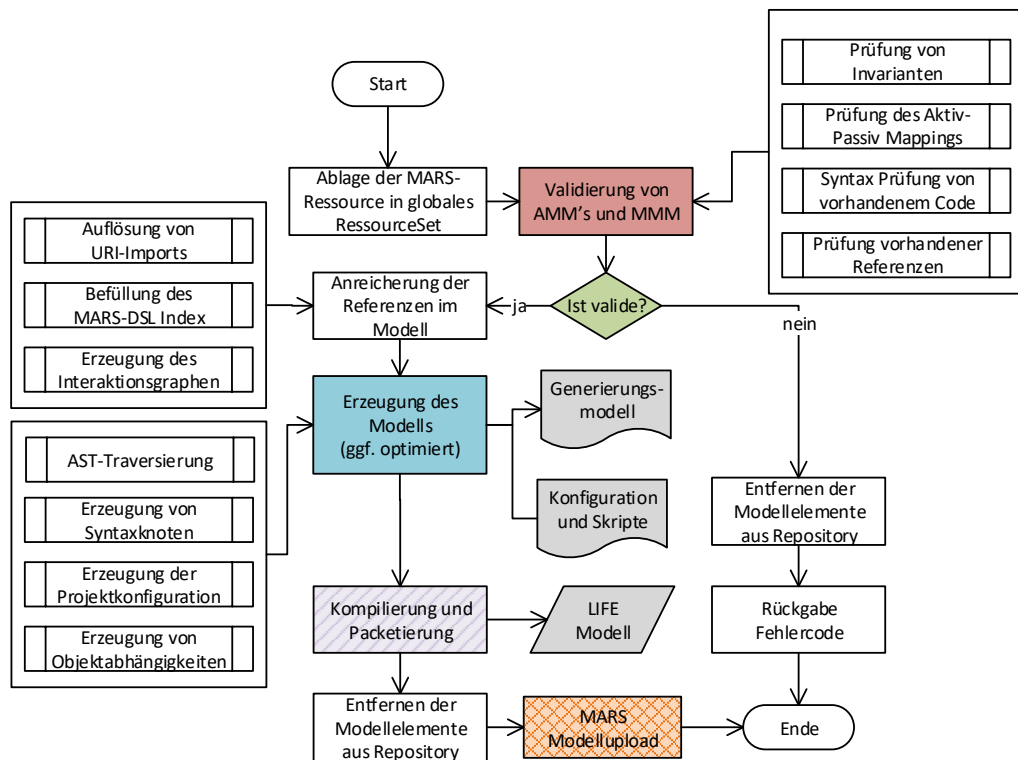


Abbildung 5.3.: Ergänzter Generierungsworkflow zur strukturierten und kontrollierten Transformation.

nur innerhalb des konkreten *AgentMetamodel* (AMM) (Glake u. a. (2017)). Insbesondere ist die Auflösung der referenzierten Ressourcen Teil dieser Anreicherung. Durch die Einteilbarkeit des eigenen Modells auf unterschiedliche Ressourcen, müssen diese auch weiterhin eine gegenseitige Referenzierung ermöglichen. Infolgedessen wird, wie in Abschnitt 4.2 bereits erläutert, ein globaler Index aufgebaut der zu jedem sichtbaren Objekt eine zusätzlich beschreibendes Objekte erzeugt und darin ablegt. Der Indexeintrag und die Anreicherung sorgen im Nachhinein für eine wertvolle Ordnung unter anderem für die Methodendeklarationen. Würde diese nicht existieren würden sich andernfalls ungeordnete Mengen durch Abarbeitung und Ausfüllen der Vorlagen ergeben, die zu inkorrekten Signaturen führen würde, bspw. zu einer einfachen gesendeten Nachricht oder einem Methode und seines jeweiligen Nachrichtenempfangs oder Aufrufs der eine strikte Parameterordnung zur Eindeutigkeit vorschreibt.

Sind alle Vorverarbeitungsschritte abgeschlossen, findet die eigentliche Übersetzung statt. Diese, nach Abschnitt 4.4 beschrieben Richtung der Transformation findet entgegen mehrerer

Teilprojekte statt, d.h es entstehen nicht nur eine oder zwei Erzeugungsartefakte aus einer ausgehenden Ressource, sondern das erzeugte Teilsystem muss innerhalb dieser Projektstruktur eingebettet sein und darüber hinaus in engen Zusammenhang mit anderen erzeugten Teilsystemen, anderer Ressourcen, stehen. In der Generierung wird hierzu eine AST-Traversierung angestoßen die für eine Übersetzung über mehrere Ressourcen hinweg arbeitet. Eine Traversierung erzeugt dabei lediglich einzelne Syntaxknoten (Klassen, Methoden, Felder, Typeigenschaften, ...), beschrieben und evaluiert nach den Vorlagen 4.4.3. Dazu gehört auch die Erzeugung jeweiliger Projektkonfigurationen mit darin gesetzten Abhängigkeiten zu anderen Projekten (Herrington (2003)) und wo diese global gefunden werden können. Das Generierungsmodell (Glake u. a. (2017)) bildet damit den Abschluss und kann hiernach mittels externem JIT Übersetzer in sein natives lesbare Format überführt werden und als hochladbares Modell für MARS-LIFE fertiggestellt werden.

5.2. Kontextsicht

Im Folgenden wird der Kontext und die vollständige Sprache nach Starke (2015) zur Systemabgrenzung der MARS DSL in Abbildung 5.4 dargestellt.

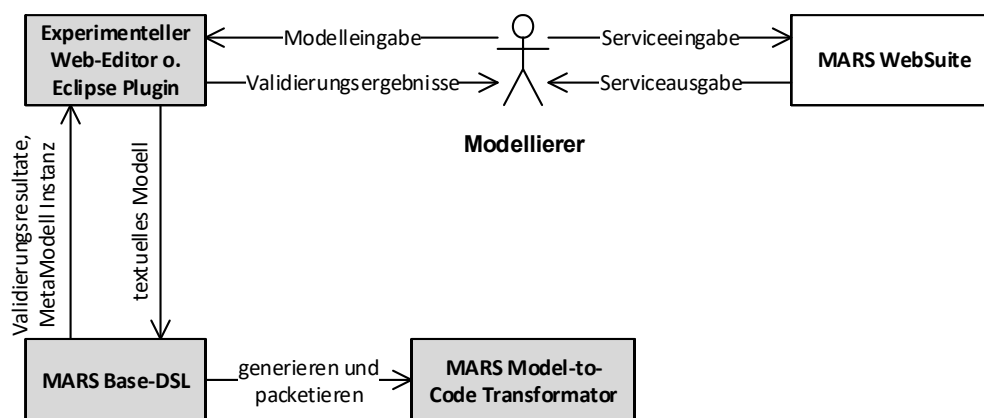


Abbildung 5.4.: Kontextsicht und Abgrenzung der DSL und des Codegenerators zur MARS Plattform und dem Modellierer. Übernommen aus (Glake (2016), geändert)

Sie stellt die gesamtheitliche Sicht mit den beiden *Zugriffsmöglichkeiten* der MARS DSL und des eigentlichen *Übersetzers* dar. Hierbei wurde der in Glake (2016) vorhandene Entwurf ergänzt um eine Zugriffsverknüpfung mithilfe eines *Plugins*, über den die Sprache verwendet

werden kann. Die restliche Abgrenzung entspricht der Einteilung in [Glake \(2017\)](#). Übersetzer und Sprachkern sind in einem System zusammengefasst und sind sowohl als *OnPremise* Lösung oder mittels experimentellem *Web-Editor* verwendbar.

5.3. Bausteinsicht

Nachfolgend werden die beteiligten Komponenten der beiden Kernkomponenten der *MARS DSL* und des entkoppelten Übersetzers vorgestellt.

5.3.1. Bausteinsicht - Level 0

Für den Übersetzer wird auf der obersten Ebene der Schnitt aus [Glake \(2017\)](#) beibehalten. Es wurde lediglich um eine neue Zugriffsmöglichkeit via Eclipse Plugin hinzugefügt.

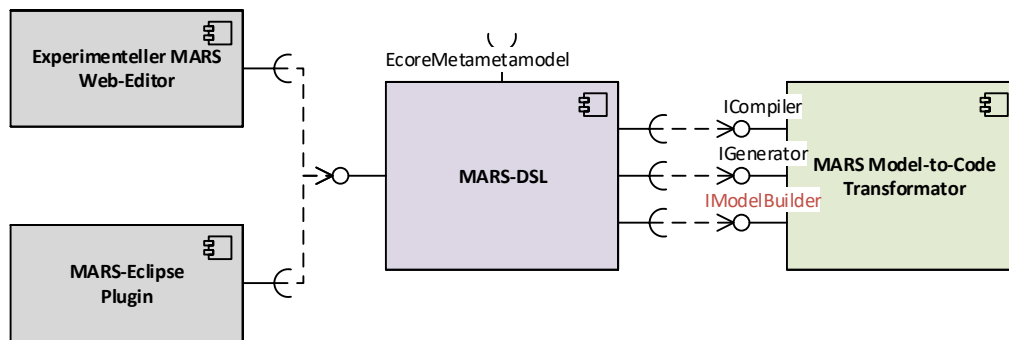


Abbildung 5.5.: Level 0 der Bausteinsicht der MARS DSL Sprachkomponente und Model-to-Code Transformator sowie zwei Zugriffsmöglichkeiten mittels experimentellem Web Editor oder eigenständigem Plugin

Der Übersetzer empfängt weiterhin dabei das Modell über die *IGenerator* und *ICompiler* Schnittstelle. Bei jedem Speichervorgang oder wenn der Generator explizit angestoßen wird, startet dieser Prozess. Dabei wird auf eine möglichst reibungslose und schnelle Erzeugung Wert gelegt. Wie in Abschnitt 5.2 ist sie in zwei Phasen eingeteilt und in die *IModelBuilder* Schnittstelle ausgelagert ([Goll \(2014\)](#)).

5.3.2. Bausteinsicht - Sprache Level 1

Mit dem *MARS DSL (MarsLang)* Kern wird die Sprache selbst mit seinen Elementen zusammengefasst. Er beinhaltet jegliche Komponenten die zur *Designzeit* die Modelleingaben und

Änderungen verarbeitet besonders die Verantwortlichkeit zur Konstruktion des aus der ANTLR (Parr (2013)) Grammatik gelesenen Textes in den AST zu transformieren (Aho (2008)). Der Kern wird dabei stets um verschiedene Aspekte der Sprache ergänzt. Mit dem Umgang *mehrerer* Ressourcen über einen gleiche oder unterschiedliche Modellnamensräume wird dem Modellierer die Möglichkeit geboten eigene Projekte aufzusetzen und innerhalb eines *Verzeichnis-basierten Workspaces* (Parr (2009)) alle beteiligten Ressourcen zu vereinen. Abbildung 5.6 zeigt den ergänzten Bausteinschnitt des Sprachkerns. Er wurde um eine Komponente zur Verwaltung der globalen Ressourcen ergänzt, von wo aus Modellteile zu anderen Ressourcen in Verbindung gebracht werden kann. Dazu existiert eine neue *Indexstruktur*, in eine abstrakte Beschreibung inklusive Referenz auf das sichtbare Modellobjekte eingetragen wird (Bettini (2016)). Der Vorteil in dieser Indexierung liegt in potenziellen Menge von Objekten die innerhalb eines Modells enthalten sein könne und der andernfalls aufwändigen Berechnung eines *globalen Scope* innerhalb des *ModelScoper*. Dieser berechnet auf Basis einer gegebenen Referenz und eines Kontextelements das als nächstes zu referenzierende Element, indem es eine Scoping Hierarchie ausgehend diese Kontextelements erzeugt und daraus das nächste Objekt wählt dessen Eingangsname auf eines der berechneten Elemente passt. Dieser Mechanismus ist zum Beispiel dazu wichtig, um potenziell mehrdeutige Aufrufe aus Invariante 4.5.1 zu vermeiden oder ggf. für den *Validator* diese zu identifizieren. Genauso legen sie die konkreten Referenzen zwischen Modellelementen fest über die bei der Übersetzung auf dem Baum traversiert werden kann. Für weiteres sei hier auf Glake (2016) verwiesen, in der eine Lazy Variante beschrieben ist, um benötigte Referenzen erst zur Abfrage tatsächliche aufzulösen. Andernfalls auch Aho (2008), die zur AST-Auflösung ein traversierendes Element einführen, dass zyklisch über das Eingabemodell wandert.

Diverse Komponenten werden durch das verwendet Xtext Framework (Bettini (2016)) bereits mit erzeugt, da diese aus der Grammatik Definition automatisch abgeleitet werden kann (Voelter u. a. (2013)). Alle automatisch erzeugten Artefakte umfassen dabei explizit das Parsing aus der zugrundeliegenden MARS DSL LL(*) Grammatik. Sie beschreibt die *Parsingregeln* zur strukturierten Eingabe einer Zeichenkette und der Transformation dieser in das MARS DSL Beschreibungsmodell 4.3. Der resultierende AST lässt sich anschließend, zur Anreicherung, Validierung, Typableitung und Übersetzung heranziehen oder wieder eine speicherbares Zeichenkettenformat serialisieren (Aho (2008)). Weiter Details zum Sprachkern findet sich in Glake (2016).

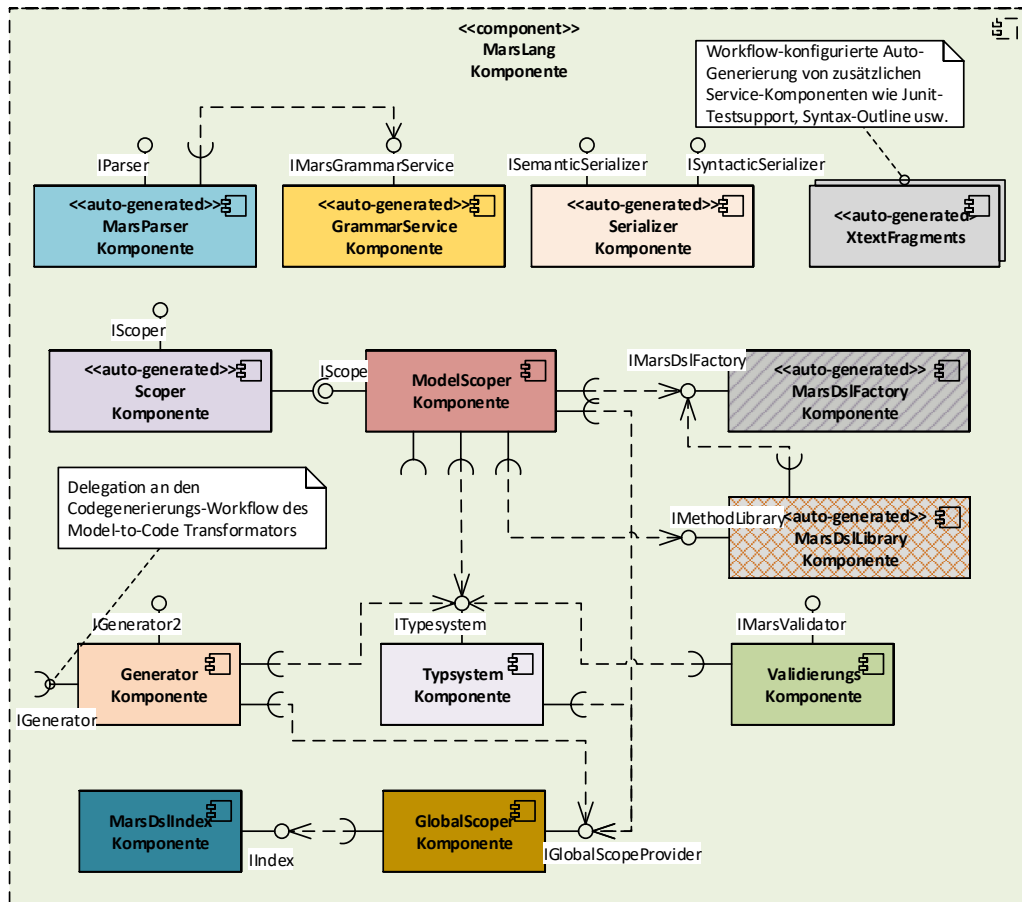


Abbildung 5.6.: Level 1 der Bausteinsicht des MARS DSL Sprachkerns.

5.3.3. Bausteinsicht - Übersetzer Level 1

Der Übersetzer ist workflow-orientiert aufgebaut, d.h. das jegliche Übersetzungsoperationen an einen Workflow weiter delegiert werden, der entsprechend die in 5.3 definierten Schritte implementiert. Diese miteinander verknüpften Schritte sind in Abbildung 5.7 dargestellt zeigen hierzu den Schnitt für Level 1 des Generators. Von außen treffen die Anfragen der DSL durch die Kernkomponente der Sprache *MARS DSL* ein und werden durch *IGenerator* Schnittstelle entgegengenommen, die unter anderem die in 5.5 vorhandenen Schnittstellen implementiert. Die unterliegenden Komponenten sind entweder als eigene Dienste verfügbar oder werden direkt durch die zugreifende Sprache konsumiert. Der gesamte Generierungsworkflow wird

dabei allein durch zentrale *WorkflowKomponente* abgearbeitet in dem Teilaufgaben wie z.B. Validierung, gesteuert *AST-Traversierung* und die Erzeugung der Syntaxknoten gekapselt sind.

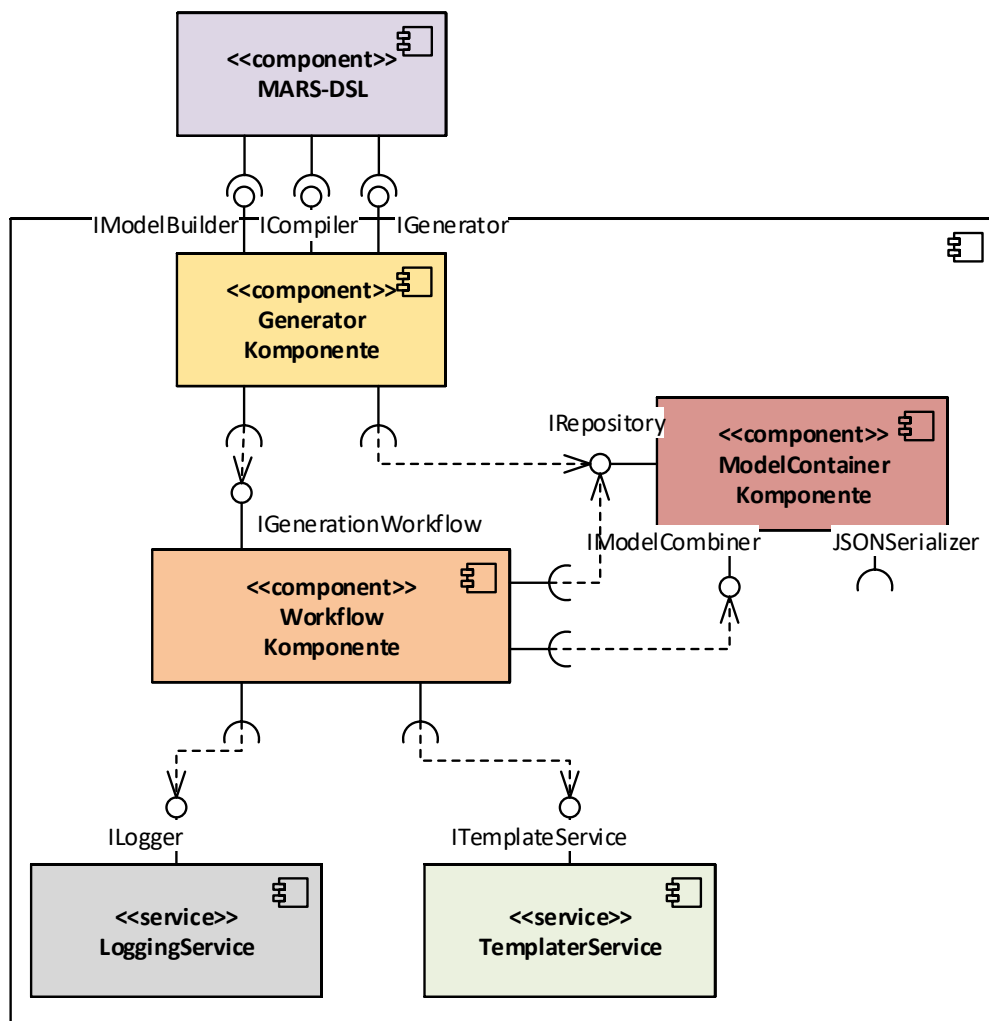


Abbildung 5.7.: Level 1 der Bausteinsicht des Model-to-Code Transformators.

Intern wird der Generator durch fünf Komponenten weiter strukturiert. Die zentrale *WorkflowKomponente* besitzt hierbei Abhängigkeiten zum entwickelten *TemplaterService* mit dem die vorlagenbasierte Texterzeugung angestoßen wird, sowie dem *LoggingService* über den ein

Log Mechanismus bereitsteht. Der entkoppelte *ModelContainer* stellt das temporäre Repository dar, über den das Modell abgelegt und zur Übersetzung abgerufen und abgefragt werden kann. Die kombinierten Teilmodelle der AMM Instanzen und MMM Instanzen sind im Zuge der Anreicherung (Glake u. a. (2017)) und Zusammenführung der Ressourcen und Einfügen in den Index darin wiederzufinden. Darüber hinaus bietet die *IModelCombiner* Schnittstelle eine Reihe von Abfragemechanismen *map, filter, first, last, sort, group, ...* auf dem AST an und bildet damit die Grundlage für die *AST-Traversierung*.

5.3.4. Bausteinsicht - Übersetzer Level 2

Die *WorkflowKomponente* ist weiter unterteilt in vier Subkomponenten, die die jeweiligen Workflows aus 5.3 implementieren. Diese Teilworkflows werden dabei vom zentralen *GenerierungsWorkflow* geschaltet und ausgeführt. Während der Validierungsschritt bei jedem Erzeugungsanstoß durchgeführt werden soll, wird die Anreicherung genauso wie der Bau entsprechend nur dann ausgeführt, sollte der vorherige Workflow-Schritt erfolgreich sein. Das gilt ebenso für die Generierung des Codes als auch den Bau, um unvorhergesehene Fehler nicht einfach zu unterdrücken. Die eigentliche Codegenerierung liegt dabei in der Verantwortung des steuernden *GenerierungsWorkflow*. Diese Komponente ist auch diejenige, die mit der genutzten Abhängigkeit zum *ITemplateService*, die damit die jeweiligen Klassen und Methoden auf Basis des Metamodells konstruiert. Abbildung 5.8 zeigt hierzu die interne Strukturierung für Level 2 der *WorkflowKomponente*.

Der Vorteil dieser Strukturierung und Trennung liegt darin, dass die eigentlichen Daten nicht mitverwaltet werden müssen. Stattdessen bietet die Komponente jegliche Funktionalität zur Texterzeugung von sich aus an, mit dem zentralen Metamodell als vereinbarter Datenkontrakt. Das macht bei einer Auslagerung der Metamodell Instanzen, z.B. in eine externe Datenbank die Verarbeitung wesentlich einfacher, da diese Komponente nicht angepasst werden muss, sondern lediglich die *ModelContainerKomponente* aus 5.7. Auch hinsichtlich der Texterzeugung ist dies unproblematisch, da wichtige Vorverarbeitungsschritte wie die Validierung und Anreicherung an diesem Punkt konzentriert sind.

5.4. Generatorsicht

Zu Übersichtlichkeit der erzeugten Artefakte wurde erneut die Generatorsicht verwendet, wie bereits aus Glake (2016) und Glake (2017). Mit dem Übersetzer werden eine Reihe verschiedener Artefakte erzeugt, die zum Teil für die Ausführung in MARS gebraucht werden und zum Teil lediglich für die interne Verarbeitung genutzt werden. Der Generator erzeugt eine

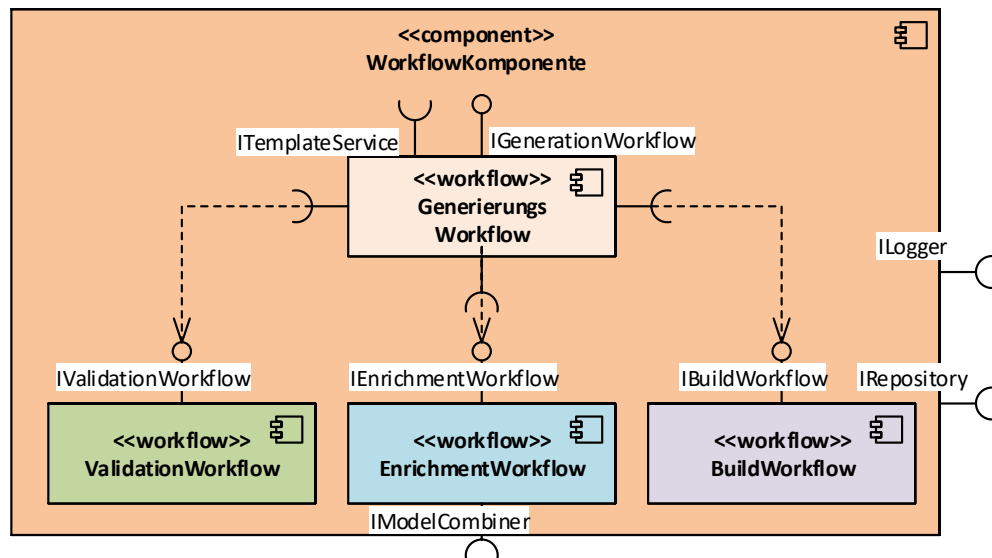


Abbildung 5.8.: Level 2 der Bausteinsicht der zentralen Workflow Komponente.

Reihe von Artefakten und passt diese an die MARS Modell Struktur und Vorgabe an, die in Verbindung mit dem MARS Modellupload steht. Überdies liegt noch das Problem, dass zwar die Metamodell Struktur aus [Glake u. a. \(2017\)](#) existiert und damit Modellcode erzeugt wird, jedoch keine Verwendung bei der Modellanalyse und dem Auffinden der Typeigenschaften und Konstruktoren, zur externen Datenbefüllung eingesetzt wird. Aus diesem Grund sind die zu erzeugenden Artefakte wie in [Abbildung 5.9](#) aufgebaut, um diesen Umstand zu umgehen.

Grob sind alle Artefakte in den Bereichen Workspace, Projekte, Code, Dienste und Konfiguration unterscheidbar. Im Zentrum steht der *Workspace* zum Vorhalten aller Bestandteile eines gesamten MARS-LIFE Modells. Innerhalb des *Workspace* wird zwischen dem Einstiegsprojekt *LIFE Starter* und dem *Models Ordner* unterschieden, in dem alle weiteren Modellkomponenten enthalten sind. Diese werden nach MARS-Konventionen durch die unterschiedlichen Layer bereitgestellt, die entsprechend zur Laufzeit als eigenständiges Projekt vorliegen, und durch den Generator darin mit drei Artefakten beschrieben werden. Diese sind neben der Projektkonfiguration im Wesentlichen die Codeartefakte für die Layer selbst und den Agenten, falls es sich um einen Agenten-Layer handelt. Um andere Agenten- und Layertypen innerhalb des gemeinsamen Namensraums zu öffnen werden in der *Projektkonfiguration* die Referenzen, auf Basis des *ActivePassive* Mappings gezogen, und wie die im [5.10](#) erläuterte Abhängigkeitsauflö-

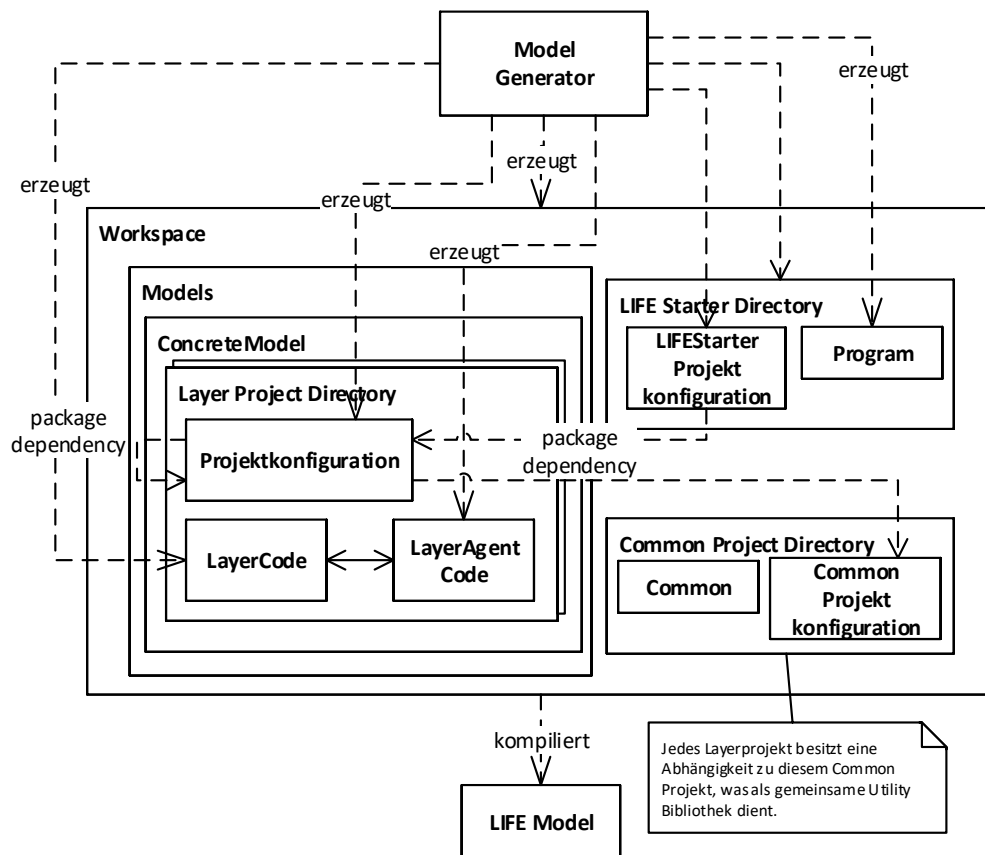


Abbildung 5.9.: Generatorsicht der MARS Modell- und Projektstruktur, mit verschiedenen Artefakten die durch den Codegenerator erzeugt werden.

sung eingetragen. Ausnahme bildet der Simulationseinstiegspunkt des *LIFE Starter* Projekts (Hüning u. a. (2016)). Dieser wird lediglich erzeugt und wird explizit für die .NETCore (Price (2016)) Umgebung mit Einstiegsmethode gebildet und zu allen anderen Projekten verlinkt. Und genau aufgrund der technischen .NET Core Umgebung müssen demnach keine weiteren Artefakte neben den gezeigten, erzeugt werden. Bspw. fällt aufgrund der Forderung das keine Änderungen zum generierten Zielsystem vorgenommen werden sollen und diese nicht in irgendeiner IDE geöffnet wird, die Erzeugung einer gemeinsamen *Solution* weg. Diese würde das Modellprojekt ganzheitlich verwalten. Trotz dessen wurde aus der Vollständigkeit heraus diese mit erzeugt und kann als schnellen alternativen Weg auch direkt verwendet werden,

um sich lediglich das Modellskelett generieren zu lassen und anschließend in der Zielsprache weiterzuarbeiten (Bettini (2016)).

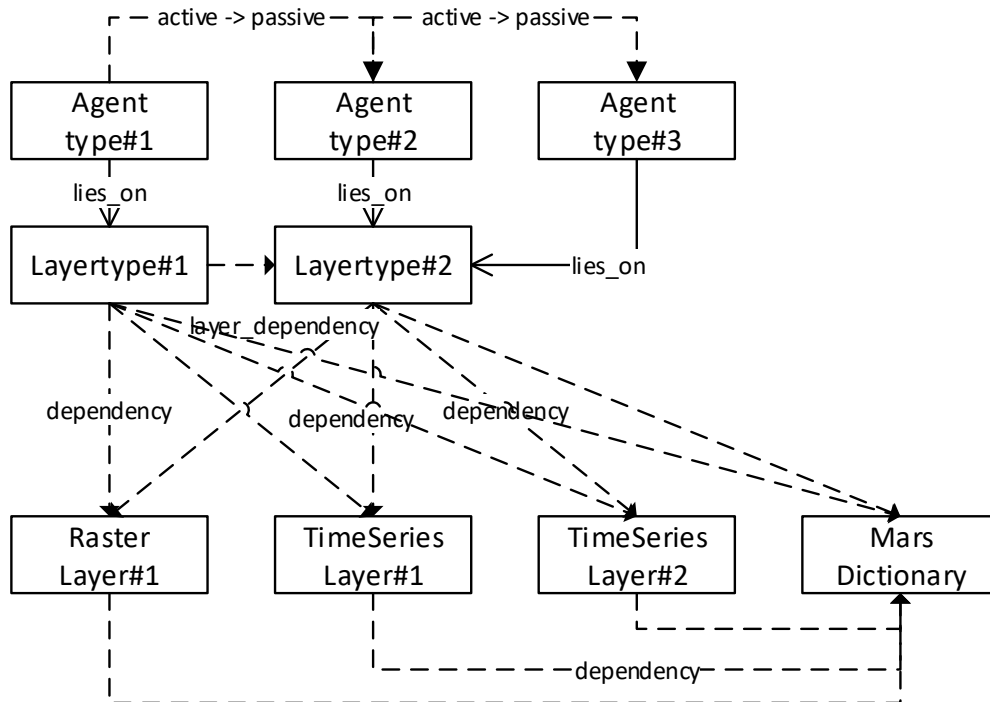


Abbildung 5.10.: Beispiel einer Abhängigkeitsentwicklung durch Integration der Datenlayer, mit drei unterschiedlichen Agententypen und zwei beteiligten Layertypen

Neben den genannten Artefakten existiert für jedes Layerprojekt eine besondere Abhängigkeit zu einem *Common* genannten Projekt, was als Bibliothek für gemeinsam genutzte Methoden dient und in dem jegliche Konnektoren oder Eigenentwicklungen die nicht explizit Teil des öffentlichen MARS-Frameworks sind enthalten sind. Sie umfasst:

Modellierte Objekttypen gemäß Definition 4.8 die zusätzlich zum eigentlichen Agentenmodell entworfen wurden und darin ggf. Verwendung finden.

MARS DSL Bibliothek die eine Verwendung von Zeitoperationen über das *Mars.Library.Time* Modul, eine ein Basis Mathematikmodul *Mars.Library.Math* mit arithmetischen Operationen $\sin(x)$, $\cos(x)$, $\text{ceiling}(x)$, $\text{round}(x)$,..., eine Teilmenge bekannter univariater stetiger

und diskreter Verteilungsfunktion $\text{emphBinomialverteilung}(x)$, $\text{Poissonverteilung}(x)$, $\text{LogarithmischeVerteilung}(x)$,... $\text{Gaussverteilung}(x)$, $\text{GammaVerteilung}(x)$, $\text{BetaVerteilung}(x)$,... und ein Umrechnungssystem für SI-Einheiten enthält.

Modell Infrastruktur zur Verwaltung aller MARS-LIFE spezifischen und technischen Aspekte zur Ausführung und Integration.

Abstrakte Schnittstelle für Collections die eine gesonderte Schnittstelle für die in der MARS DSL verwendeten Listen und Menge sowie dem Literale 4.3.5 gilt. Anders als die bekannte LINQ Konzept [Hejlsberg u. a. \(2003\)](#) umfassen diese Schnittstelle Operationen die direkt den MARS DSL Collections angewendet werden, zum Beispiel *filter*, *map*, *sort*, *head*, *tail*,

Konnektoren für Agentenumgebungen die für zur Übersetzung eine tragende Rolle in der Optimierung spielen und in Abhängigkeit der identifizieren Modelleigenschaften und Agenteneigenschaften, aus Abschnitt 4.6 unterschiedliche Datenstrukturen mithilfe eines Entscheidungsbaums auswählt.

Basis Grid-Umgebung die im Falle einer nicht entscheidbaren Umgebungsauswahl aus den anderen Möglichkeiten diese als Standard enthält.

Innerhalb jedes Codeartefakts ist die Struktur und die Beziehung nach dem Erzeugungsmodell in 4.3 aufgebaut und durch einen gemeinsamen Namensraum in der Form von `Mars.Models.{ModelName}` sichtbar. Andernfalls müsste an dieser Stelle auf eine Namensraumauflösung Bezug genommen werden, und bspw. jegliche eingetragene Abhängigkeiten als `Import` bzw. `Using` in den Code hinzugefügt werden. Durch diese Trennung reicht die Sichtbarmachung des Namensraums vom Modell aus, um keine losen Referenzen innerhalb des generierten Codes zu haben. Das Gleiche gilt für die als abstrakt markierten *GenericLayer* und das gemeinsame Environment in Form des *GridLayer* und seiner Schnittstelle. Alle Elemente sind im gleichen Namensraum zu finden und können durch die konkreten Typen referenziert werden.

5.5. Laufzeitsicht

Im Folgenden werden die wesentlichen Laufzeiten des Systems vorgestellt, die sich aus der Kommunikation zwischen den Bausteinen aus 5.3 ergeben. Sie befassen sich mit den Workflows zur Validierung und Anreicherung, dem Generierungsschritt und der ergänzen Rule Engine, die Teil des *reaction* Konzeptes ist. Essenzielle weitere Laufzeiten zum typabhängigem Scoping zur

Aufrufauflösung Typmembem, der Einbindung der MARS DSL Bibliothek und dem eigentlichen Parsingdurchlauf findet sich in den vorangegangenen Arbeiten [Glake \(2016\)](#) und [Glake \(2017\)](#).

5.5.1. Laufzeitsicht - Validierung und Anreicherung

Mit der Validierung ist ein notwendiger Schritt vorhanden, um damit nicht übersetzbaren Code zu vermeiden. Dieser befasst sich mit der Prüfung, die in 4 analysierten Bedingungen, die zur Lauffähigkeit des System genutzt wird.

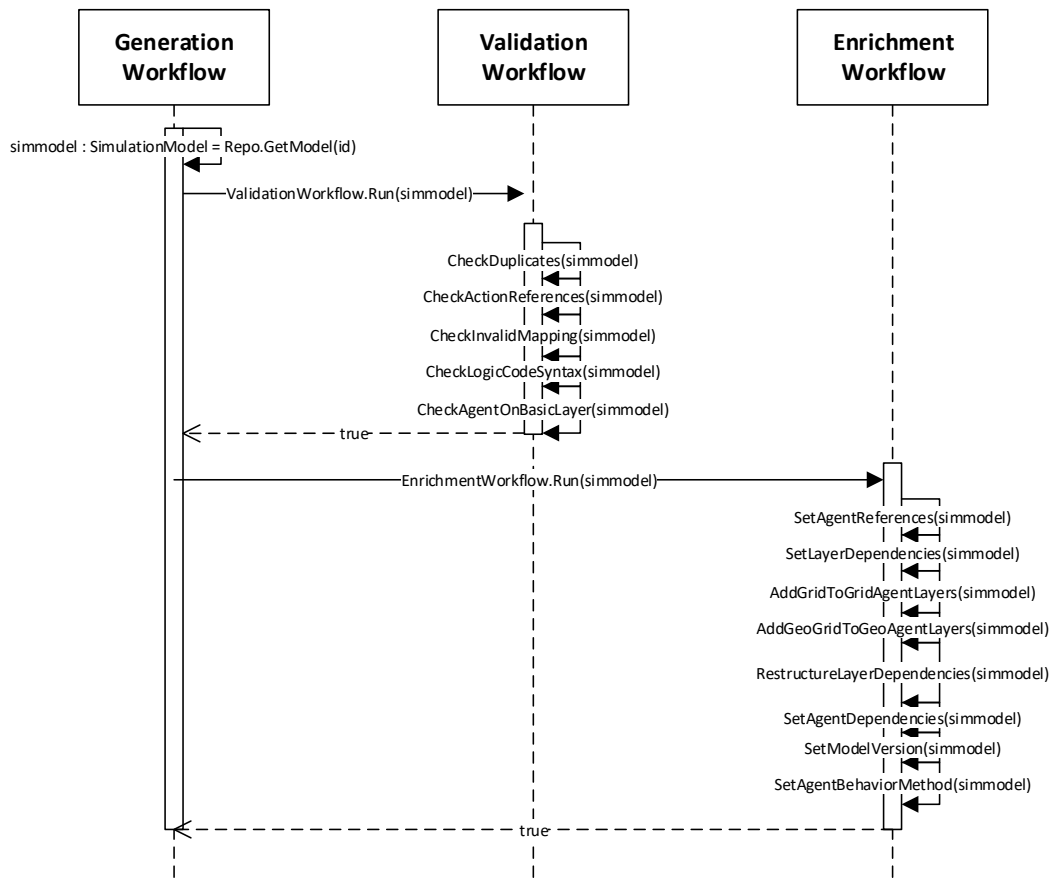


Abbildung 5.11.: Laufzeitsicht der Validierung und Anreicherung des Eingabemodells.

Die Validierung wird angestoßen durch den steuernden *GenerationWorkflow* lediglich über eine einfache *Run()* Methode, die der Workflow implementieren muss. Dieser definiert

zugleich das Modell als Eingabe, auf dem der Workflow dann arbeiten darf. Während im *ValidationWorkflow* lediglich eine Reihe von Lesezugriffen vorgenommen wird, die bestimmte Bedingungen innerhalb des Modells überprüfen (siehe Abschnitt 4.5) wird im anschließenden *EnrichmentWorkflow* dagegen auch schreibend auf dem Objektgraphen zugegriffen. Die Prüfung der Invarianten wird bereits im Vorfeld vorgenommen, da der *GenerationWorkflow* an dieser Stelle nur dann angestoßen wird, wenn die vorherige Validierung erfolgreich war. Der *EnrichmentWorkflow* führt einige Vorverarbeitungsschritte aus, bei dem er die lose gekoppelten Elementen nun durch direkte Referenzen ersetzt und dabei vor allem eine Ordnungsrelation über verschiedene Parameterlisten, und Imports hinzufügt. Das ist wichtig um beim Zugriff von Methoden die korrekte Signatur zu erhalten. Während die identifizierbaren Objekte zwar alle eine Bezeichnung haben sowie eine eindeutige ID, liegen den möglichen Parametern dabei keine Ordnungsrelation zugrunde. Stattdessen wird diese vor allem durch `RestStructureLayerDependencies` im Wesentlichen lexikographisch geordnet. Um diese Information zu speichern existiert für jeden Knoten innerhalb des Eingabemetamodells eine partielle Erweiterungsklasse vor, so bspw. für die Klassen `AgentMetamodel` und `AgentReference` aus Glake u. a. (2017) eine partielle Erweiterungsklasse mit dem gleichen Namen, die diese Eigenschaften zum Schreiben enthält.

5.5.2. Laufzeitsicht - Codegenerierung

Die eigentliche Texterzeugung wird zentral durch die `GenerationWorkflow` Komponente gesteuert, unter Zuhilfenahme des `TemplateService`, der die Vorlagen verwaltet und ausführt. Die grundlegende Verarbeitung zwischen diesen beiden Komponenten wird in Abbildung 5.12 gezeigt.

Die Codegenerierung funktioniert in zwei Phasen. Die erste Phase betrifft die Zerlegung des Domänenobjekts in seine Bestandteile und die Erzeugung von Stellvertretern (Platzhaltern), die aus der Vorlage heraus abgerufen werden und die Daten enthalten, die zweite Phase betrifft die eigentliche Ausführung und Befüllung.

Um die Platzhalter zu erzeugen wird hierzu ein *reflektiver* Ansatz gewählt, bei dem aus dem Eingabemodell (hier das kombinierte Simulationsmodell) die Typeigenschaften jeder Klasse, respektive die Entität, eingesammelt und eine Referenz auf den tatsächlichen Wert erzeugt wird. Dazu wird anfänglich die zu verwendende Vorlage mittels `CompileTemplate` vorkompiliert, sodass der Inhalt zunächst im Speicher vorliegt, anschließend werden die jeweiligen Platzhalter erzeugt. Ein solcher Platzhalter bildet dabei eine Kopplung zwischen dem eigentlichen Objektgraphen mit seinen Daten und den Platzhaltern innerhalb der verwendeten Vorlage. Jede Typeigenschaft wird dabei von der *PSM-Klasse* (Glake u. a. (2017)) abgefragt

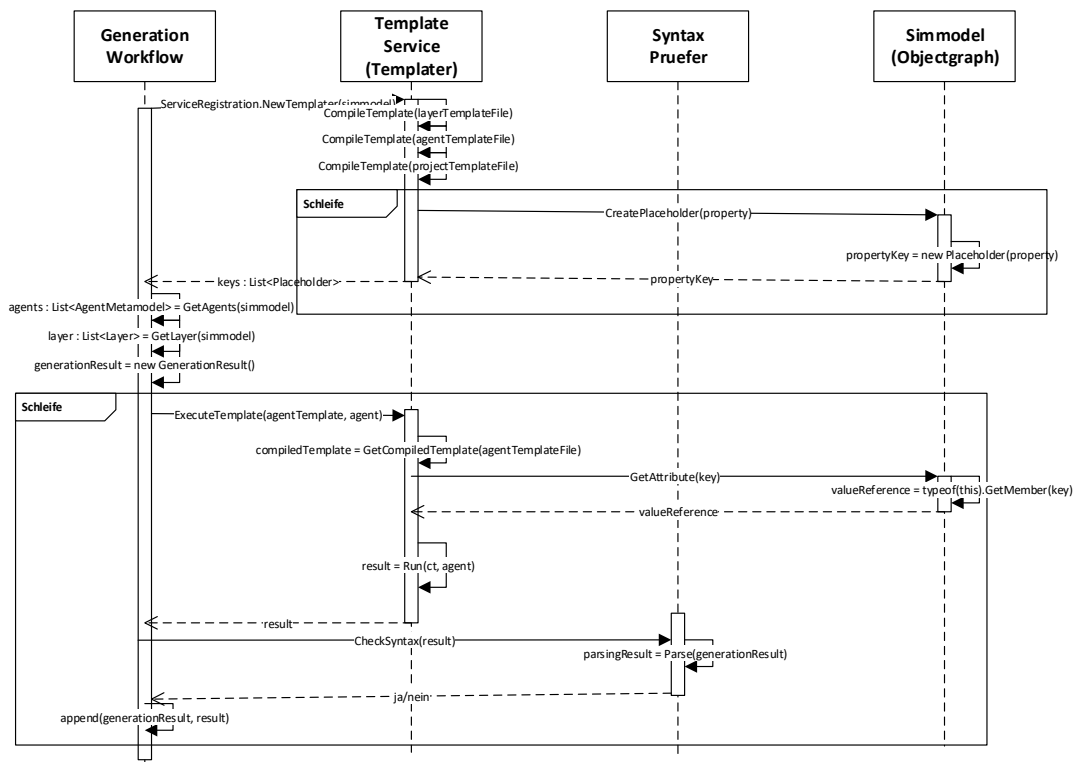


Abbildung 5.12.: Laufzeitsicht der Codegenerierung für Agenten und Layer

und durch `CreatePlaceholder` mit seinem korrespondierenden Platzhalter und einem Schlüssel, um diesen bei der Befüllung wiederzufinden, erzeugt und assoziiert. Dieser Schlüssel ist dabei einfach der Name der Typeigenschaft, da dieser aus der Sprachspezifikation her eindeutig sein muss (Hejlsberg u. a. (2003)). Am Schluss liegt eine Schlüssel-Wert Menge mit `keys : List<Placeholder>` vor, bei dem jeder Name der entsprechend in der Vorlage eingesetzt wird, auf einen Platzhalter verweist, der wiederum eine Referenz auf die tatsächliche Typeigenschaft besitzt. Dies entspricht der transitiven Abhängigkeit von: `Placeholder -> PSM-Typeigenschaft -> Wert`. Der Vorteil hinter dieser aufwändigen Vorverarbeitung liegt darin, dass zur Laufzeit ein Eingabemodell lediglich ein einziges mal per Reflektion abgefragt wird und mit den Platzhaltern ein Direktaufruf stattfindet. Andernfalls müsste bei der Befüllung für jede Ausfüllung aufwändig reflektive Abfragen auf dem Eingabemodell erfolgen, was zu einem spürbaren zeitlichen Verlust führt, insbesondere hinsichtlich der nicht-funktionalen Anforderung einer sofortigen Generierung, ohne spürbaren Verzug. Weitere

Vorteile liegen in der strikten Trennung vom Vorlagencode, also Anweisungen die innerhalb der Vorlage enthalten sind und jenen Anweisungen die aus der Abfrage der Platzhalter im Domänenmodell zu finden sind, d.h. es lässt sich Logik in Form von Methoden im Eingabemodell hinterlegen, die als Platzhalter aus der Vorlage heraus aufgerufen und evaluiert werden können. Das liegt an der Art des Aufrufs, da eine Typeigenschaft auf technischer Ebene durch ein einfaches `Get` und/oder `Set` auf ein Feld möglich wird.

In der zweiten Phase der Befüllung wird die Vorlage mittels der Platzhalter ausgefüllt und die eingetragenen Ausdrücke darin ausgeführt. Es werden bei jeglichen Typeigenschaften, die mit einem Platzhalter verbunden sind, der korrespondierende Wert hinterlegt. Zudem werden die begrenzten internen Ausdrücke identifiziert und ausgeführt. Dabei sind jegliche Konstrukte aus 4.4.3 möglich.

Die Vorlage funktioniert dabei ähnlich einem Skript, was ausgeführt wird. Die einzige Kopplung wird über die Daten erreicht, bei dem das Skript während der Ausführung diese abfragt. Ausdrücke beziehen sich dabei auf Schleifen, Verzweigungen oder gar die direkte Ausführung von Code. Möglich wird dies mittels der internen Verarbeitung indem die Vorlagen als Eingabe in die eingebettete Template Sprache gegeben wird, und ausgehend von dem erzeugtem *AST* deren Schleifen und Verzweigungen direkt in das Pendant der aktuell ausgeführten Sprachumgebung überführt wird (Leroy (2014)). Da die MARS DSL nun selbst den eigentlichen *Generierungsschritt* übernimmt, ist dies eine Verzweigung und Schleife in Java. Das *Verzweigungsprädikat* p_{if} oder das Iterationskommando i_{for} wird direkt ausgeführt als anonyme Funktion $\lambda_{if} \rightarrow eval(if)$ oder $\lambda_{for} \rightarrow next(for)$ übergeben (Russell und Norvig (2002)) und anschließend aufgerufen. Die Einführung des Lambda-Konzepts in die Generierung führt zusätzlich einen weiteren Vorteil mit sich, bei dem durch den Lambda Kalkül einhergehende Term in der Praxis durch weitere Terme, bzw. Anweisungen ergänzt werden kann und somit während der *Baumtraversierung* in Glake (2016) für die Übersetzung der Ausdrücke an dieser Stelle zusätzlicher Code hinzugefügt werden kann, ohne dass sich die Gesamtstruktur des Systems verändert. Anstelle von direkt ausgeführten Operationen wie $(+, -, *, /, \dots)$ werden diese innerhalb der anonymen Funktion gekapselt und erlauben darin ggf. Prüfungen auf aktuelle Systemzustände oder verschiedene Zwischenberechnungen, eben als eingeführte und direkt evaluierte Methode. So wurden bspw. die Kommandos der Exploration oder der Bewegung eines Agenten durch diesen Ansatz umgesetzt. Das definierte Lambda wird ausgeführt und nach den Platzhaltern der Reihe nach, wie für Abbildung 5.12 erläutert, verarbeitet.

5.5.3. Laufzeitsicht - Reaction Ladeprozess

Im Folgenden wird die Laufzeitsicht des Übersetzungsvorgangs vorgestellt, der sich seit (Glake (2017)) um die hier betrachteten Regel Engine erweitert hat und innerhalb einer MARS Simulation das Regel Inferenzsystem für Agentenverhalten zum Konzept eines *reaction* aus Abschnitt 4.3.4, darstellt. Regeln werden zum Systemstart ausgehend eines initialisierten Agenten von diesem an das Regel Repository `RuleRepository` übergeben. Die Übergabe erfolgt in Form eines Selektions Prädikats `ruleSelector` der die konkreten voll-qualifizierten Typen der Regeln im eigenen Agentenkontext angibt, nach denen im `RuleRepository` gesucht werden soll. Die Regeln werden dazu zunächst an den erzeugten Layer delegiert der eine Verknüpfung der agenten-spezifischen Regeln zur konkreten Instanz verwaltet. Durch die indirekte Abhängigkeit der Reactions kann in diesem Sinne eine potenzielle Parametrisierung erfolgen, mit der Frage. Welche Agentenregeln in einem Lauf zum Tragen kommen sollen. Der Layer delegiert diese Regeln an das `RuleRepository` weiter der auf dem beherbergenden Agentenmodell eine Reflektion ausführt. Der Aufwand ist jedoch gering, da die konkreten Typen bereits im Selektor hinterlegt sind. Es wird lediglich eine Referenz auf die konkrete und zu prüfende Regel verlangt. Auf den gesammelten Regeln `List<ReactionRule>` setzt ein Filter auf, der diese Filterung zurückgibt und an die Agenten innerhalb Layers bindet.

Die eigentliche Kompilierung erfolgt erst dann sobald alle zu verwendenden Regeln eingesammelt wurden. Sie umfasst dabei das Einpflegen und Reorganisieren der Reaction Regeln in das Rete-Netzwerk. Dieser hängt dazu jedes verknüpfte Left-Hand-Side Bedingungsprädikat (LHS) in den Evaluationsbaum, die entgegen der Faktenbasis ausgeführt werden. Das LHS wird hierzu in seine Bestandteile zerlegt und von oben nach unten, links nach rechts in den Baum als Knoten eingehängt (Russell und Norvig (2002)). Je nachdem ob eines der festgelegten Junktoren *and, or, exist, all, having, query* verwendet wurde wird dieser sogenannte α oder β Knoten in den Baum eingehängt. Diese differenzieren zwischen einfachen Selektionsbedingungen die auf den Objekten und Tupeln angewendet werde, z.B. einem Prädikat, und komplexe Verbundoperatoren deren Bedingungen aus einem Junktore (und Variablenbindung) stammt. So wird beispielsweise das Prädikat $FischfangHoch(fischer) \wedge IstGrosserFisch(fischer, hering) \vee IstNahrhaft(hering) \rightarrow PayTaxes(hering)$ in die einzelnen α -Knoten $FischfangHoch(fischer)$, $IstGrosserFisch(fischer, hering)$ sowie $IstNahrhaft(hering)$ aufgeteilt und die beiden konjugierten Aussagen $FischfangHoch(fischer) \wedge IstGrosserFisch(fischer, hering)$, $IstGrosserFisch(fischer, hering) \vee IstNahrhaft(hering)$ jeweils als β -Knoten innerhalb einer Ebene tiefer eingehängt. Sinn hinter dieser Aufteilung ist die Reduktion des Evaluationsaufwands zur Laufzeit und dem Aufbau einer effizienten Netzwerkstruktur.

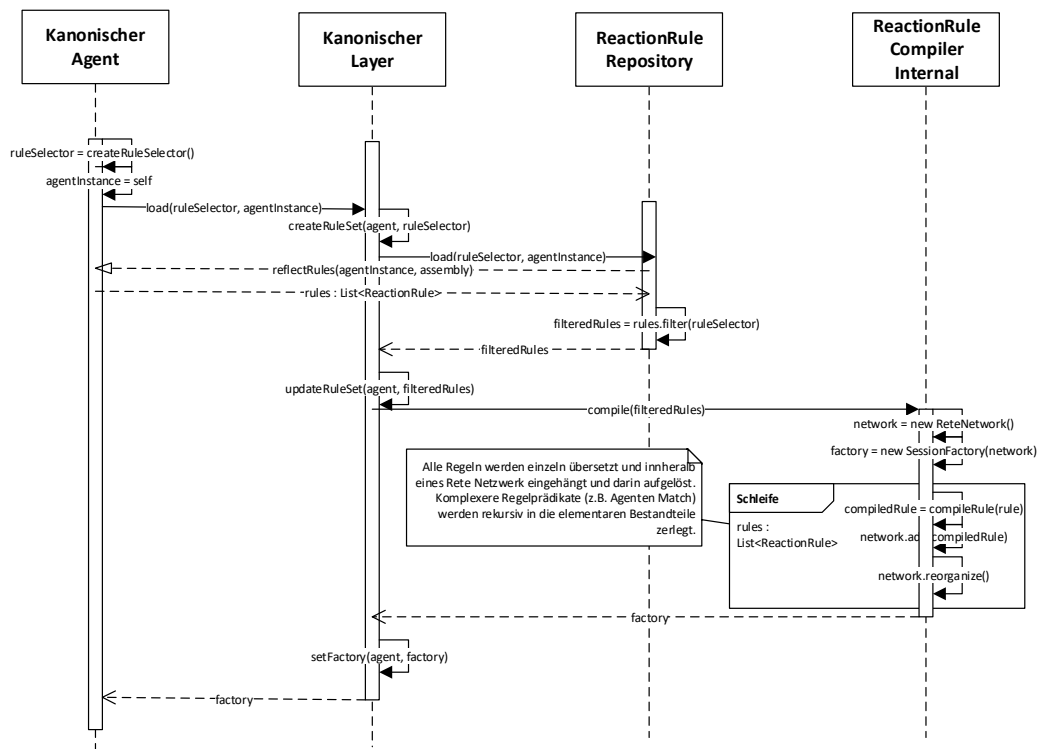


Abbildung 5.13.: Laufzeitsicht des Reaction Lade- und Übersetzungsvorgangs zum Simulationsstart

Eine `SessionFactory` wird als Repräsentant des Rete Inferenzmechanismus erzeugt, von dem aus jede anzustoßende, neue faktenbezogene Regelauswertung ausgeht. Diese Session Factory wird vom Layer an den Agenten gebunden und an diesem weitergereicht. Der Agent erzeugt sich daraus einzelne aktive Sessions, in denen er die Faktenbasis befüllt und die Regelauswertung anstößt.

5.5.4. Laufzeitsicht - Regelauswertung

Das Anstoßen einer Regelauswertung wird ausgehend eines Agenten immer zu Beginn seines Zeitschritt durchgeführt. Die zugrundeliegende Faktenbasis wird vom Inferenzalgorithmus des Rete-Netzwerks entgegen der definierten Regeln evaluiert (Russell und Norvig (2002)). Diese Auswertung erfolgt dabei nach dem **match - resolve - act** Zyklus, der bis zum Schluss oder explizitem Abbruch abgearbeitet wird. Die Faktenbasis umfasst alle sichtbaren Agenten

5. Entwurf

Instanzen die durch den beherbergenden Agenten-Layer zugegriffen werden können und über den andere Layer referenziert werden [4.3.2](#). Jeder Layer hält die Menge der Agenten vor um sie für einen Zeitschritt auch sofort als neuen Eintrag mittels Massenimport hinzufügen zu können.

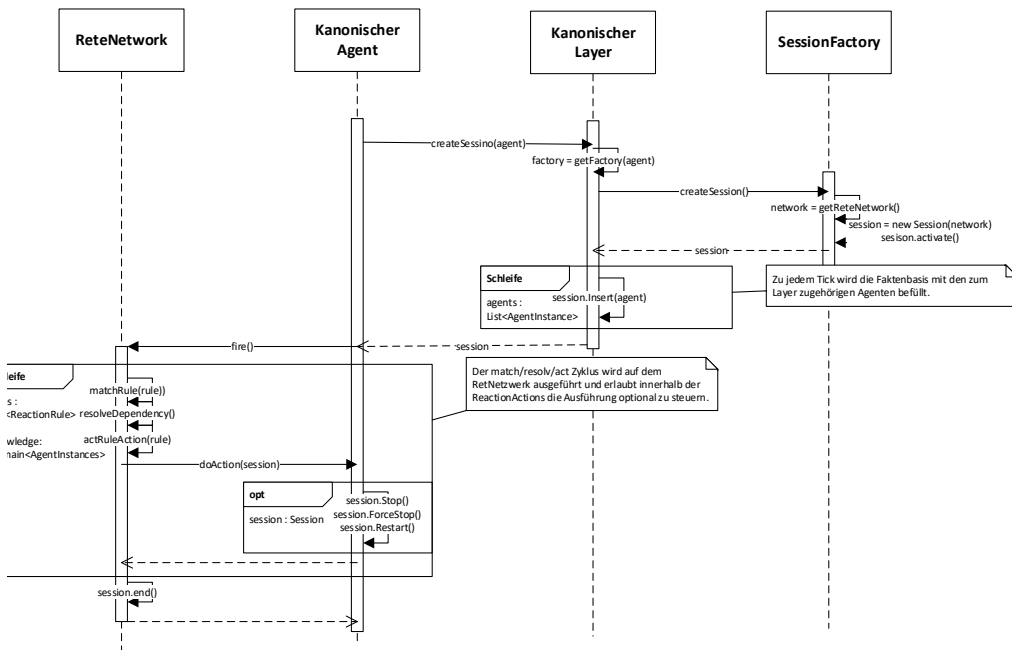


Abbildung 5.14.: Laufzeitsicht der angestoßenen Regelauswertung, definierter Reactions

Abbildung [5.14](#) zeigt den technische Seite der Regelevaluation eines Agenten. Da diese hinter einer Session gekapselt ist als Transaktion angesehen werden kann, fordert ein Agenten dazu eine neue Session über seinen Layer an. Dieser greift auf die existierende `SessionFactory` zurück, die bereits in [5.5.3](#)

5.6. Transformationssicht

Folgend sollen nun die zugrunde liegenden Lösungen der zwei Subkomponenten des entwickelten Template Service sowie des Mars Dictionary vorgestellt werden. Betrachtet werden hier insbesondere die Vorlagenkonstrukte sowie die Abbildung des Dictionary in das Zielsystem.

Die Erzeugung jedes Teilausdrucks erfolgt über eine AST-Traversierung der ausgehend einer Methoden- oder Aktionskörpers *body* dessen geordnete Ausdrücke iteriert. Die Iteration

5. Entwurf

erfolgt dabei mittels polymorphen Dispatcher, indem für eine Element und dessen Meta-Typ (z.B. *Literal*, *IfExpression*, *Switch*,...) eine Dispatcher zu einem einzelnen Ausdruck vornimmt. Diese Unterscheidung ist dabei geordnet nach dem zugrundeliegenden *Typen*. Für jeden Typ wird eine eigene Übersetzung durchgeführt die ausgehend eines lokalen Übersetzungskontexts stattfindet und auf global erzeugte Elemente Bezug nehmen kann.

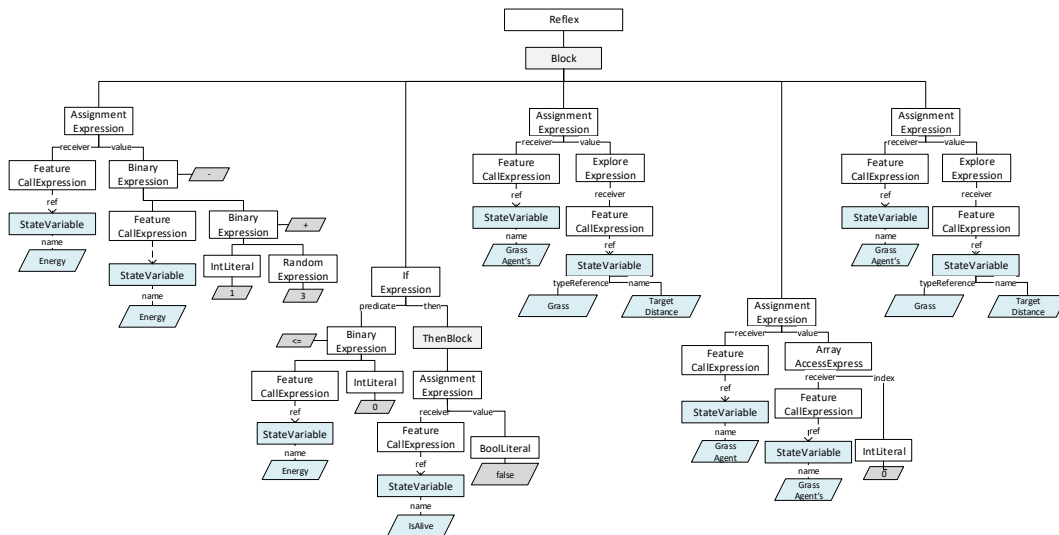


Abbildung 5.15.: Beispiel AST eines Agenten Reflex Körpers

Abbildung 5.15 zeigt ein zerlegtes Models eines Ausdruckskörpers der von links nach rechts durchlaufen wird un für jeden Typen die Fallunterscheidung erneut startet. Jedes Metaelement ist dazu mit einer Vorlage nach 5.6 verknüpft die die tatsächliche Zeichenkette wiedergibt.

Vorlagen Transformation

Der Template Dienst wird gebraucht um mittels einer vorliegenden Vorlage daraus Text zu erzeugen. Diese Vorlagen werden durch einen reflektiven Ansatz mit Daten befüllt (siehe 5.12) und ausgeführt. Es wurden mehrere vorhandene Vorlagendienste untersucht und letztendlich eine Eigenimplementierung auf Basis von Xpand (Bettini (2016)) vorgenommen, die jegliches Verhalten für eine möglichst schnelle und einfache Texterzeugung bietet. Da die Xpand Engine nicht auf die Erzeugung von konkreten C# Code ausgelegt ist sondern zur Erzeugung von jeglichem Text, werden die Vorlagen dementsprechend angepasst.

```

1      <<FOR sensor : model.simModelElements.filter(Sensor)>>
2      <<FOR sensorEntry : sensor.sensorEntry.filter[it.layer.equals(layer)]>>
      >>

```

```

3      «FOR layerDependency : sensorEntry.dataSources
4      SEPARATOR ',' AFTER ','»
5      this.«layerDependency.name»
6      «ENDFOR»
7      «ENDFOR»
8      «ENDFOR»
9      «FOR tsLayer : index.visibleTsLayer() SEPARATOR ',' AFTER ','»
10     this.«tsLayer.name»
11     «ENDFOR»
12     «FOR raster : index.visibleRaster() SEPARATOR ',' AFTER ','»
13     this.«raster.name»
14     «ENDFOR»
15
16     this
17 }, Environment);

```

Listing 5.1: Partielle Vorlage zur Erzeugung eines Agentenkonstruktors.

Die Datenbefüllung arbeitet wie in 5.12 gezeigt, indem innerhalb eines Vorverarbeitungsschrittes zunächst *reflektiv öffentliche* Typeigenschaften mit deren assoziierter Typisierung als Platzhalter registriert werden. Im *Templatedienst* werden diese lediglich abgefragt und die verknüpfte `GetMethod` aufgerufen. Das Problem an dieser Stelle ist jedoch nicht die Datenbefüllung sondern die Evaluation der eingetragenen Ausdrücke aus 4.4.3. Die Lösung für dieses Problem bietet die mit der Compiler Plattform Roslyn (Ng u. a. (2011)) inspirierte Skriptverarbeitung. Dabei wird in Form eines Interpreters diesem ein Programm als Parametereingabe übergeben, der zur Laufzeit einliest direkt ausführen kann. In Bezug auf JVM Natur der MARS DSL existiert hierzu die interne Template *Micro-DSL*. Bezogen Vorlagen wird direkt in eine Konkatenation von Zeichenketten transformiert, deren jeweilige Schleifen und Bedingungen diese steuern lässt. Listing 5.1 en Ausschnitt aus einer Vorlage der Template Engine, bei dem einem Layertypen aus dem Eingabemodell `SimmodelElement`s und seinen Platzhaltern deren Layerabhängigkeiten direkt eingepflegt wird. Dazu wird aus der gesetzten Richtung der Sichtbarkeit im `SENSOR`, ähnlich der Definition in der MARS DSL (Glake (2016)), einem aktuell betrachteten Layertypen sein genutzter Ziellayer als Parameter inklusive Typ und Namen eingetragen. Die interne Skriptverarbeitung funktioniert dabei im aktuellen ausführenden Prozess, d.h. es wird kein externer Kontext angesteuert und bietet damit eine zügige Erzeugung ohne nennenswerten Verlust.

Die Vorlage mit seinen Kommandos wird hierbei durch einen Parser eingelesen und anschließend als Kommandosequenz durchlaufen. Im Hintergrund liegen die jeweiligen Methoden, wie `filter(...)` bereits zugrunde die sich innerhalb der Vorlage direkt verwenden lassen.

5.6.1. Syntaktische Korrektheit

Für lokale und global Elemente als Zwischenschritt oder Bestandteile der Modell Infrastruktur es wichtig, das potenzielle Duplikate die sich zur Übersetzung ergeben, zu vermeiden. Solch ein Duplikat kann entweder auf eine vorher definiertes Element der Modlell Infrastruktur zurückzuführen sein oder irgendwelcher anderer mit erzeugter Artefakte die im Konflikt mit einem identifizierenden Element des Eingabemodells steht. Für die MARS DSL Grammatik wurde dazu eine syntaktische Einschränkung in den Bezeichner von Modellelementen *Agent*, *Layer*, *Variable*, *State*, *Method*, ... eingebaut um diese Konflikte zu vermeiden. Bezeichner im Eingabemodell sind beschränkt auf die Form:

$$\{ab^* \mid a \in \{a, \dots, z\} \cup \{A, \dots, Z\} \wedge b \in \{a, \dots, z\} \cup \{A, \dots, Z\} \cup \{0, \dots, 9\} \cup \{-}\} \quad (5.1)$$

Damit sind natürliche Bezeichner für Elemente wie *{Wolve, HeringModel2, Fischer_, sender_receiever, ... }* möglich, jedoch keine Bezeichnungen wie *{_Elephant, 1HeringModel, _2Fischer, ...}*. Element der Modellinfrastruktur werden im Zielsystem immer beginnend mit "_" anfangen, um damit Namenskonflikte zu vermeiden (Aho (2008)).

5.6.2. Inkrementelle Erzeugung

Die Erzeugung der Modellartefakte zum Zielsystem spezifizierten 4.4 umfasst diverse Teilkonstrukte, insbesondere jene für die Basis-DSL aus der für jede Ausdruckselement eine eigene Übersetzung angestoßen wird und innerhalb eines Artefakts eingegliedert. Da es vorkommen kann, dass komplexe Modelle, wie das in 6.3 beschriebene Evaluationsmodell entwickelt werden sollen die nicht alle Artefakte stetig neu erzeugt werden (Aho (2008)), sondern eine inkrementelle Übersetzung wird angestrebt (Karsai u. a. (2014)). Dieser Umstand wird dadurch bekräftigt dass der Großteil des Aufwands weniger auf die globalen Elemente fällt, sondern auf die einzelnen Verfeinerungsstufen, insbesondere der Definition der Workflows oder Aktionen (Bettini (2016)) mittels MARS-Base DSL (Glake (2016)).

6. Evaluation

Zur Bewertung der MARS DSL wurde zwei unterschiedliche Richtungen eingeschlagen. Zum einen wurde ein Akzeptanztest des Werkzeugs bei einer Kernzielgruppe durchgeführt der mittels einer formulierten Umfrage zu den Dimensionen und Qualitätsmerkmalen zum Evaluationsrahmen 6.1 passen. Diese stellen unterschiedliche abgeleitete Qualitätsdimensionen zur Bewertung von DSL's aus der Literatur dar die mit einer Reihe von Kriterien verknüpft sind. Der Bewertungsrahmen umfasst dabei Dimensionen zur Sprache selbst, zur Ausführung und zur Qualität. Der zweite Bewertungspart wird durch das MARS Fangmodell - Fangen und Fischen - vorgenommen. Dabei soll trotz der Notation und ggf. fehlender Konzepte die Mächtigkeit der Sprache herausgearbeitet werden und gezeigt werden, wie weit komplexe Systemmodellierung reichen kann. Für das Fangmodell wurde hierzu auf eine ausgehende modellierte Systemgleichungen zurückgegriffen in die in das Paradigma der Multiagentensysteme mit der MARS DSL verfeinert und implementiert wurde.

6.1. Evaluationsrahmen

Eine Vielzahl unterschiedlicher Bewertungsmöglichkeiten existieren für textuelle und grafische domänenspezifische Sprachen (Kahraman und Bilgen (2015)) (Barišić u. a. (2011)) (Bettini (2016)) (Karsai u. a. (2014)) die sich allesamt auf unterschiedliche Bereiche konzentrieren. So sieht die Arbeit von Kahraman und Bilgen (2015) einen Bewertungsrahmen für DSL's vor, der insbesondere auf die Qualität zur Benutzerakzeptanz setzt und analysiert wie gut ein neues Werkzeug die eigene Modellierungsfreiheit einschränkt. Mit Barišić (2013) wird vor allem eine organisatorische Lösung propagiert, bei der die Bewertung einer DSL in mehreren iterativen Schritten folgt und ein dazu entsprechender Musterkatalog eingeführt wurde, nach dem man sich richten kann. Geprüft ist dieser Vorschlag ebenfalls mithilfe einer Benutzerumfrage unter ähnlicher Zielgruppenbedingung, die für diese MARS DSL durchgeführt wurde. Zuletzt weist Karsai u. a. (2014) darauf hin, vor allem die Machbarkeit zu demonstrieren, indem eine Reihe nachzubildender Systeme gezeigt werden, die mit der neuen Sprache möglich sind oder erst werden. Viele weitere Ansätze und eine Übersicht mit weiteren Mitteln finden sich in der Arbeit zu Rodrigues u. a. (2017) die dazu eine eigenständige Bewertungstaxonomie aus diversen

Arbeiten aufbauten und die verschiedenen Bewertungstechniken in die Bereiche: *Useability Evaluation Methods*, *Software Engineering Evaluation Methods*, *Instruments* (darunter finden sich Umfragen und Interviews), *Framework Approaches*, *Evaluation Metrics*, *Data Types* und *User* eingeteilt haben.

6.1.1. Dimensionen

Zur Einteilung der MARS DSL wurde auf diese Arbeiten zurückgegriffen und eine eigene Bewertungsmatrix aufgestellt. Sie umfassen verschiedene Faktoren zur Bestimmung wesentlicher Qualitätseigenschaften einer DSL. Diese Eigenschaften unterscheiden sich dabei nur geringfügig von typischen Softwarequalitätsmerkmalen. Um den Erfolg der MARS DSL zu begründen sind eine Reihe von Erfolgsfaktoren zu DSL identifiziert, die aus der oben genannten Literatur 6.1 und aus Abschnitt 2 zur Akzeptanz abgeleitet wurden. Alle Faktoren sind spezifisch für den Einsatz einer DSL, d.h. es wurde auf allgemeine Erfolgsfaktoren wie persönliche Eigenleistung verzichtet, da sie nicht direkt von der DSL betroffen sind und stärker auf die Aspekte Ausdrucksstärke und Korrektheit Bezug nehmen. Die identifizierten Faktoren sind:

Zuverlässigkeit (I) (Spinellis (2001)) (Barišic (2013)) Eine DSL kann große Teile des Entwicklungsprozesses automatisieren, was zu weniger Fehlern führt.

Benutzbarkeit (U) (Starke (2015)) Werkzeuge und Methoden, die neben der Sprache die Modellierung unterstützen, sollten eine einfache und bequeme Bedienung aufweisen und der Modellierung förderlich sein - Projektsetup, Diagramme, Visualisierungen.

Lernfähigkeit (L) (Spinellis (2001)) (Voelter u. a. (2013)) Die DSL hilft Anwendern Domänenkonzepte zu modellieren, die sonst zeitaufwändig zu implementieren sind. Das entsprechende Kompilat wird automatisch erzeugt und auf Invarianten geprüft. Dies senkt den Aufwand und verkürzt die Zeit bis zu den Ergebnissen.

Produktivität (C) (Mohagheghi u. a. (2013)) (Barišic (2013)) Entwickler müssen eine zusätzliche Sprache lernen, die Zeit und Mühe kostet. Da sich die Domain und ihre Mittel mit der Zeit verändern, muss sich die DSL weiter entwickeln und Anwender müssen auf die Veränderungen aktualisiert werden.

Ausdrucksstärke (Fowler (2010)) (Voelter u. a. (2013)) Mit einer DSL können domänenspezifische Funktionen kompakt implementiert werden, die Sprache ist jedoch für diese Domäne spezifisch und begrenzt die möglichen Szenarien, die ausgedrückt werden können.

Dimension	Abstrakte Syntax (Metamodell)					Konkrete Syntax				Transformation	
	Submetamodell (Viewpoint)	Metaklasse (Class)	Attribute (Property)	Assoziationen (Reference)	Vererbung (Super type)	Knoten (Keywords)	Typ Ableitungsregeln	Terminale	Constraint	M2M Regeln	M2T Vorlagen
Anzahl	2	120	41	150	111	80	154	12	40	90	101

Tabelle 6.1.: Quantifizierung der MARS DSL

Wiederverwendbarkeit (R) (Spinellis (2001)) (Mohagheghi u. a. (2013)) Möglichkeit bestimmte Modellteile oder ganze Lösungen auf dieser Ebene wiederzuverwenden, anstelle von Teilen des Kompilats (Aho (2008)).

6.1.2. Quantitative Analyse

Die Sprache MARS- DSL wird auf zwei Arten bewertet, um die sprachliche Dimension des Evaluierungsrahmens abzudecken. Zunächst werden die Elemente analysiert. Dies legt die Komplexität der Sprache offen und kann in quantifizierter Form zum Vergleich herangezogen werden. Anschließend werden die Eingabe und Ausgabe der Sprache analysiert, was den Weg zur Beurteilung der erzeugten Artefakte ebnet. Tabelle 6.1 zeigt die Quantifizierung der MARS DSL über sein Metamodell aus (Glake u. a. (2017)).

Eingeteilt ist die quantifizierte Analyse in die Bereiche abstrakte und konkrete Syntax sie nehmen Bezug zum zugrundeliegenden AST der aus der Konkreten Syntax übersetzt wird. Die M2M und M2T Transformation bilden den Kern der MDD und stellen mit ihren 90 und 101 Übersetzungsregeln die elementarsten Punkte. Als notwendiges Mittel zur Übersetzung liegt aufgrund der Größe des Metamodells mit 120 *Metaklassen*, die Anzahl von 154 Ableitungsregeln nahe, zum einen die gesonderte Ableitung jedes Teilelements von der groben *Entitäts- und Typdefinition* über einzelne Schleifen und Kontrollstrukturen bis zu den Literalen. Die große Menge der Vererbung liegt nahe, dass für die DSL viele feingranulare Spezialfälle existieren. Verbessert werden kann das Metamodell dahingehend diese Überlast zu verringern und vor allem die Anzahl *Knoten (Schlüsselwörter)* von aktuell 80 zu reduzieren und die Funktionalität in die MARS DSL *Standardbibliothek* auszulagern, entgegen dem NetLogo Vorgehen (Wilensky (2015)), neue Funktionalität als Kommando in die Grammatik zu verankern und die Übersetzung

über die Metaklasse zu steuern (Bettini (2016)). So existiert mit einem Zufallsausdruck *random* ein solches Negativbeispiel, das bereits als Funktionalität in die Bibliothek verlagert wurde, jedoch noch Teil der Grammatik ist.

6.2. Qualitativer Workshop

Eine Umfrage zur Zufriedenheit und Umgang mit der MARS DSL ist eine Möglichkeit Workshopteilnehmer und andere Interessierte dahingehend zu prüfen, wie erfolgreich die MARS DSL ist. Ein Fragebogen in Bezug zu den oben betrachteten Dimensionen und Faktoren gibt Auskunft über den Erfolg der Sprache. Die Umfrage im Anhang C zu finden, wurde eingeteilt in die Bereiche DSL-Notation, Modellierung, Validierung und Ausführung. Jede Frage ist einer Dimension aus Abschnitt 6.1 zugeordnet und intervallskaliert von 0 bis 5. Der nachfolgende Abschnitt erörtert die gemittelten Ergebnisse.

Gruppe Vorkenntnisse

Das Projekt umfasste 5 Teilnehmer mit einem Abschlussminimum *MasterDegree*, *Phd* und *Phd Student*. Es war ein DSL Experte vor Ort und es wurden keine Aufgaben verteilt sondern eigene Modelle entwickelt. Direkte Rückmeldungen kamen von 3 Teilnehmern, und freundlicherweise nachträglich von 2 Teilnehmern noch nachgesendete Ergebnisse.

Entwicklungserfahrung Die Gesamtheit der bisherigen Erfahrungen im Bereich Programmierung, unabhängig der agentenbasierten Simulation, hat eine Durchschnittsbewertung in der Skala 0 bis 5 ergeben, wobei 0 bedeutet keinerlei bisherige Erfahrung und 5 absoluter Experte: 0. *Den Befragten sind Analysewerkzeuge wie die Sprache R bekannt, jedoch keine gängige Entwicklungssprache.*

ABMS Vorkenntnisse Die Gesamtheit der Vorkenntnisse im Bereich der agentenbasierten Simulation hat eine Durchschnittsbewertung ergeben, in der Skala 0 bis 5, wobei 0 bedeutet keinerlei bisherige Erfahrung und 5 absoluter Experte: 0. *Der Südafrika Workshop richtete sich in erster Linie an Einsteiger, von Unerfahrenheit wurde hier ausgegangen.*

Sprachverständnis

Die DSL stellt ein neues Werkzeug dar, womit Modelle beschrieben werden. Es ist offen, ob die Notation die Konzepte verständlich vermittelt und lesbar darstellt:

6. Evaluation

Fall	Bewertung \emptyset
Verständnis aller DSL-Elemente	3
Keine komplexen Elemente	3
Verwendung der Elemente ist einfach	2.66
Keine Unterstützung durch Experten notwendig	3.33
Elemente werden als notwendig und wichtig angesehen	3
Leichte Differenzierbarkeit der Elemente	4
Vorausgehendes Wissen ist nicht notwendig	4
Keine Verwendungsschwierigkeiten der Elemente	4
Modellcode ist lesbar	3.33
Gesamt \emptyset	3.36

Tabelle 6.2.: Bewertung der Notation und Konzepte, Skala (0 - 5; 0:keine Zustimmung und 5:hohe Zustimmung)

Fall	Bewertung \emptyset
Beschreibbarkeit des erwünschten Verhaltens	4
Verständnis der Typinferenz und der Datenkonzepte	2
Verwendbarkeit externer Daten	2
Aufwand zur Modellverfeinerung	3
Verständnis des internen Verhaltens	3.33
Keine Beschränkung der Modellierungsfreiheit	5
Gesamt \emptyset	3.22

Tabelle 6.3.: Bewertung zur Modellierungsmethodik, Skala (0 - 5; 0:keine Zustimmung und 5:hohe Zustimmung)

So hat sich nach der Umfrage ergeben, dass ein mittleres Verständnis vorhanden ist, jedoch die wirkliche Verwendung im Modell noch etwas unklar ist. Da die meisten Beschreibungsmittel versucht haben die Agentenkonzepte und Co. direkt abzubilden, liegt es nahe, dass durch die wenige ABMS Erfahrung der Interessierten, noch eine hohe Einstiegshürde existiert.

Modellierung

Die Modellierung bietet mächtige Konstrukte, die unterschiedlichen Anforderungen genügen. Die Teilnehmern wurde nach der Beschränkung in der eignen Modellierung befragt, ob das Konzeptmodell ohne viel Aufwand in die Implementierung überführt werden könne:

Eingeschränkt fühlten sich die Befragten nicht, stattdessen fehlte hauptsächlich das Verständnis zur Verwendung der Konzepte.

6. Evaluation

Fall	Bewertung $\bar{\varnothing}$
Fehleridentifizierung	4
Ausdrucksstarke Fehlermeldungen	5
Modellanpassung (Korrektur) ohne Zusatzaufwand	5
Modellneustart mit Anpassungen	4
Verständnis des internen Verhaltens	1.6
Keine technischen Fehler aufgetreten	5
Gesamt $\bar{\varnothing}$	4.16

Tabelle 6.4.: Bewertung zur Validierung und Ausführung von Modellen, Skala (0 - 5; 0:keine Zustimmung und 5:hohe Zustimmung)

$C_{Teaching}$	$C_{ReadingDocs}$	C_{Demo}	$C_{Totalnit}$
1h:30min	1h:00min	0h:45min	3h:15min

Tabelle 6.5.: Überschlagene Einarbeitungszeit zu MARS und der MARS DSL

Validierung und Ausführung

Bei keinen sind sprach-technische Fehler aufgetreten und Rückmeldungen zur Design Zeit waren *gut* bis *sehr gut* verständlich. Modellfehler sind bei allen Teilnehmern aufgetreten und konnten anhand der Rückmeldungen identifiziert werden. Die Ausführbarkeit der Modelle funktioniert, Modellanpassungen konnten hinzugefügt werden, um mit geringem Aufwand das Modell erneut zu starten.

Aus den visualisierten Ergebnissen des eigenen Modells konnte jedoch die Schlussfolgerung bezüglich dem *internen Modellverhalten* abgeleitet werden.

Zeitkosten

Die Einarbeitungszeit aus den Ergebnissen leitet sich aus der Zeit zur Modellierung über die Gesamtzeit inklusive Experimentierzeit ab. Im Durchschnitt betrug die Zeit für die Konstruktion zur Modellierung:

Der Zeitaufwand für die eigentliche Modellierung inklusive ausgeführter Experimente ist in Tabelle 6.6 gegeben:

$C_{Modelling}$	$C_{Experiments}$	$C_{Analyze}$	$C_{TotalModelling}$
2h:40min	1h:00min	0h:30min	4h:10min

Tabelle 6.6.: Überschlagene Modellierungszeit mit der MARS DSL inklusive Experimente

Konzept	Verwendung
Agentendefinition agent	1
Agenteninteraktion (passive and active)	0.8
Exploration (explore, nearest)	1
Regelbasiertes Verhalten (reaction)	0.4
Bewegung (move)	0.8
Reflex Tick (reflex)	1
Funktionsdefinition (def)	0.2
Globale Uhr (simtime)	0.4
Raster Layer (raster-layer)	0.4
Time Series Layer (ts-layer)	0
Layer (layer)	1
State Deklaration (observe, external)	0.4
Verzweigung (if)	1
Schleifen (each)	0.6
Literal (#,##)	0.4
Type-Guards (switch)	0.2
Klassenkonzept (class)	0.2

Tabelle 6.7.: Anteilige Verwendung der MARS DSL Beschreibungskonzepte

Es gilt zu beachten, dass die Modellierung und die Experimentierphase eng zusammenhängen, z.B. wegen der Modellkalibrierung und der Fehlerbehebung.

Konzepteneinsatz

Abgefragt wurde der Einsatz vorhandener Beschreibungskonzepte im eigenen Modell, die zusammengefasst in Tabelle 6.7 dargestellt sind.

Wie zu erwarten wurden bei allen Teilnehmern des ABMS-Workshops das Agentenkonzept **agent**, mit einem Verwendungsanteil von 1 genutzt und aus der Invariante folgend das **layer** Konzept ebenfalls mit 1. Interessant ist dagegen die vornehmliche Verwendung des einfachen *Reflex* **reflex** mit 1, anstelle der komplexeren *Reactions*, in dessen Workflow mit Verzweigungen **if** 1 gearbeitet wurde. Um mit anderen Agenten zu interagieren, wurden mittels **explore** oder **nearest** andere Agenten wahrgenommen, und Bewegungsaktionen **move** fanden bei 0.8 der Beteiligten im Modell ebenfalls statt. Keinerlei oder nur geringfügige Verwendung fand dagegen der Zeitreihen-Layer **ts-layer** und vorhandene Klassenkonzepte **class**.

Zufriedenheit

Einige explizit gestellte Fragen bezüglich der Zufriedenheit ergaben folgende Ergebnisse:

MARS DSL UI Zur Unterstützung der umgebenden Modellierungsaufgaben hat sich für die Akzeptanz der UI eine Durchschnittsbewertung in der Skala 0 bis 5, wobei 0 bedeutet keinerlei Hilfe und 5 absolute Hilfe, ergeben: 3.25

MARS DSL Gesamt Die Gesamtheit der MARS DSL hat eine Durchschnittsbewertung ergeben, in der Skala 0 bis 5, bei der 0 bedeutet keinerlei Zufriedenheit und 5 absolute Zufriedenheit: 3.75

MARS Gesamt Die Gesamtheit der MARS-Plattform mit Modellierungsansatz mit DSL, Experimenten, Ausführung und Datenanalyse hat eine Durchschnittsbewertung ergeben, in der Skala 0 bis 5, wobei 0 bedeutet keinerlei Zufriedenheit und 5 absolute Zufriedenheit: 4

6.2.1. Zusammenfassung

Alle Bestandteile und die nach den oben genannten Kriterien eingeteilten Fragen ergeben dies in Abbildung 6.1. Es zeigt die unterschiedlichen Merkmale und die nach Einschätzung der Modellierer wahrgenommene Ausprägung.

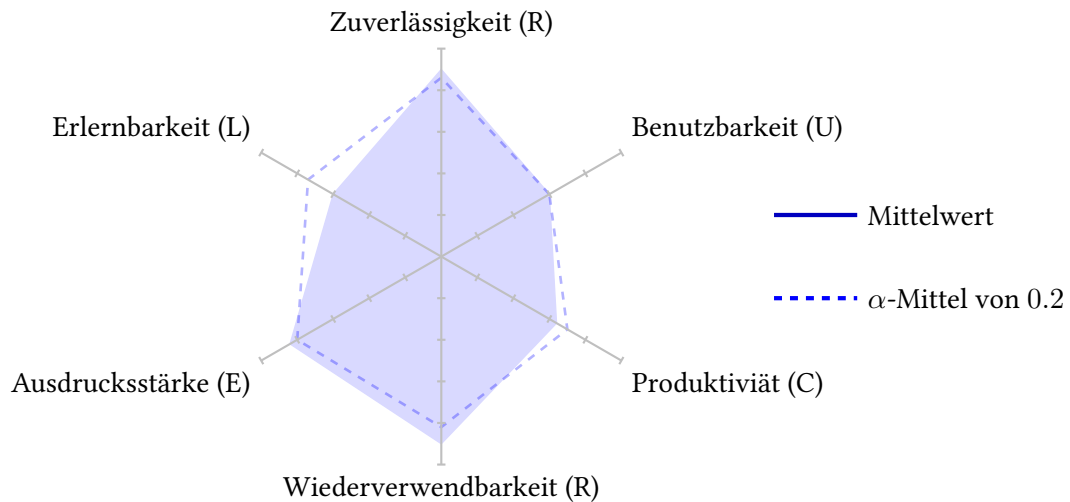


Abbildung 6.1.: Zusammenfassung der Umfrageergebnisse zur MARS DSL Zufriedenheit mit arithmetischem Mittelwert (graue Fläche) und α -getrimmten Mittelwert von 0.2

So hat sich ergeben, dass die MARS DSL unter Einsatz eines unerfahrenen Benutzers eine hohe Hürde darstellt, alle wichtigen Elemente konkret zu kennen und zu verstehen. Die *Ausdrucksstärke* beweist, dass die Konzepte zur Abbildung eigener Modelle ausreichen, genauso

wie deren Bindung untereinander, die nach Angabe der *Wiederverwendbarkeit* nicht allzu groß ist, sodass bereits entwickelte Modellteile leicht zwischen Gesamtmodellen ausgetauscht und für andere Zwecke angepasst werden können. Das in Form eines Eclipse Plugins bereitgestellte Werkzeug ist zunächst als Testumgebung im Einsatz, weswegen auf eine akzeptable Benutzeroberfläche ein geringer Fokus gelegt wurde. Folglich ergibt sich auch eine schlechtere Benutzerführung *Benutzbarkeit*, die unter anderem das Modellsetup begleitet, und über eine übliche Oberflächensteuerung es dem Modellierer an dieser Stelle einfacher machen kann. Zuletzt zeigt sich, auch nach der Prüfung auf Fehlervermeidung und Validierung aus Abschnitt 6.2, dass dieser Punkt insgesamt als positiv bewertet wurde. Auf Modellfehler und Inkonsistenzen wurde hingewiesen und anderweitige systembedingte Probleme sind während des Workshops nicht festgestellt worden.

6.3. Fangmodell - Fangen und Fischen

Als illustratives Beispiel und Bewertungsgrundlage der MARS DSL soll im Folgenden das MARS Fangmodell inklusive Modellausschnitten in der DSL präsentiert werden. Dies ist Bestandteil einer Machbarkeitsprüfung der DSL, was für ein Modell unter anderem möglich ist. Es zeigt die Konstruktionsfähigkeit zur Bildung einer eigenen Modellumgebung, mit beteiligten Entitäten und Interaktionen zwischen diesen. Es beschäftigt sich mit einer Kernproblematik der Fischerei, indem es versucht die direkten und indirekten Wechselwirkungen der beteiligten Fischer und Lebewesen aus einem Nahrungsnetz abzubilden. Zentraler Punkt dieses Abschnitts ist zudem die Präsentation einer Teilmenge der vorhandenen DSL-Elemente, integriert in einem Modell und der Anwendung dieser in verschiedenen Formen.

6.3.1. Motivation

Forschungsfrage

An der Küstenregion zur Nordsee existiert ein großer Bestand an Fischen, insbesondere Heringe, Phytokulturen und Kleintiere alias Zooplankton. Ebenso wird entlang der Küstenregion Fischfang betrieben. Während die Fischerei sich für die vorhandenen Heringe interessiert bleibt im Gewässer ein Nahrungskreislauf erhalten, worin sich Heringe von Zooplankton ernähren. Das Zooplankton ernährt sich von Phytoplankton und das Phytoplankton wird durch die Ausscheidung oder totes Gewebe mit Nährstoffen gefördert (Götting u. a. (2013)). Da Überfischung ein großes Problem, in jedweden Teilen der Welt darstellt, liegt eine offene Frage darin: Wann ein stabiles Nahrungsnetz existiert, sodass die Fischereikultur mit diesen

6. Evaluation

Daten	Typ	Beschreibung	Quelle
Phosphor-Gesamt	Punkt	Gesamte mittlere Phosphor Konzentration (2006-2014)	Melles und Lehfeldt (2014)
Städte	Punkt	Ortschaften und Städte im Bereich des Wattenmeers mit Population und Klassifizierung (Stadt, Dorf, Ortschaft,...)	Melles und Lehfeldt (2014)
Territoriales Seegebiet	Polygon	Abdeckung des territorialen Außengebiets, EU spezifisch und für Hochseefischerei geeignet	International Hydrographic Organization und Sieger (2012)
Internes Seegebiet	Polygon	Abdeckung des küstennahen Seegebiets das von Kleinbootführern befahren werden darf	International Hydrographic Organization und Sieger (2012)
Bathymetrie	Raster	Hydrographische Tiefseekarte zur Darstellung von Untiefen	Weidauer (2007)

Tabelle 6.8.: Übersicht verwendeter Daten und deren Quellen

Ressourcen auskommen kann? Dieses ökologische Interesse wird dadurch begründet, dass in der Nordsee zwar Messungen zur Grünalgen Population (Melles und Lehfeldt (2014)) existiert, jedoch nicht klar ist, in welcher Weise das Ausbreiten von Plankton die Population von Fischen und damit die Fischerei bestimmt. Zum Teil aus Umweltschutzgründen blockierte Gebiete können einen Freiraum zur Ausbreitung bieten und damit sowohl Nahrungsgrundlage für das Zooplankton und die Heringe bieten als auch damit eine Fanggrundlage für die Fischerei.

Datengrundlage

Zur Darstellung des diskreten MARS DSL Grids wird das Modell nicht auf einem frei modellierten Gebiet betrachtet, ohne Realitätsbezug, sondern es wird das Gebiet um das Niedersächsische Wattenmeer betrachtet. Daten wurden freundlicherweise vom Projekt MDI-DE (Marinedaten Infrastruktur Deutschland) zur Verfügung gestellt und umfassen weitläufige Umweltdaten um dieses Gebiet (Melles und Lehfeldt (2014)). Des Weiteren wurde zur korrekten Darstellung des Wassers und Küstengebiets der Nordseegebiete das Kartenmaterial der Internationalen Hydrographischen Organisation verwendet (International Hydrographic Organization und Sieger (2012)).

Alle Daten sind im Modell als Layer integriert, die in Abschnitt 6.3.6 mittels dem neuen Raster-Layer Konzept zu einer kongruenten Umgebung zusammengeführt ist. Das heißt alle Daten sind auf einem diskreten Grid abgebildet. Punktkarten wurden auf ein Raster über Interpolation

durch Inverse Distanzgewichtung (Russell und Norvig (2002)), mit Abstandkoeffizient $p = 2$ interpoliert. Seegebietskarten wurden zu einer Gebietskarte zusammengeführt und die Bathymetrie bietet Angaben über Tiefen, geschnitten nach der Seegebietskarte. Städte wurden aus Punktdaten auf das Raster projiziert und es werden nicht alle Punkte genutzt, sondern nur eine Teilmenge, dessen Rasterfeld ein Küstengebiet abdeckt deckt und damit eine direkte Anbindung an das Seegebiet enthält.

Wirkungsmodell

Einstiegspunkt für das erstellte Fangmodell ist das Wirkungsdiagramm, das die Beziehungen und Einflüsse beteiligter Entitäten untereinander skizziert. Abstrakt beschrieben versucht das Modell ein Nahrungsnetz aus drei beteiligten tierischen und pflanzlichen Typen unter einem externen Einfluss durch die Fischerei abzubilden.

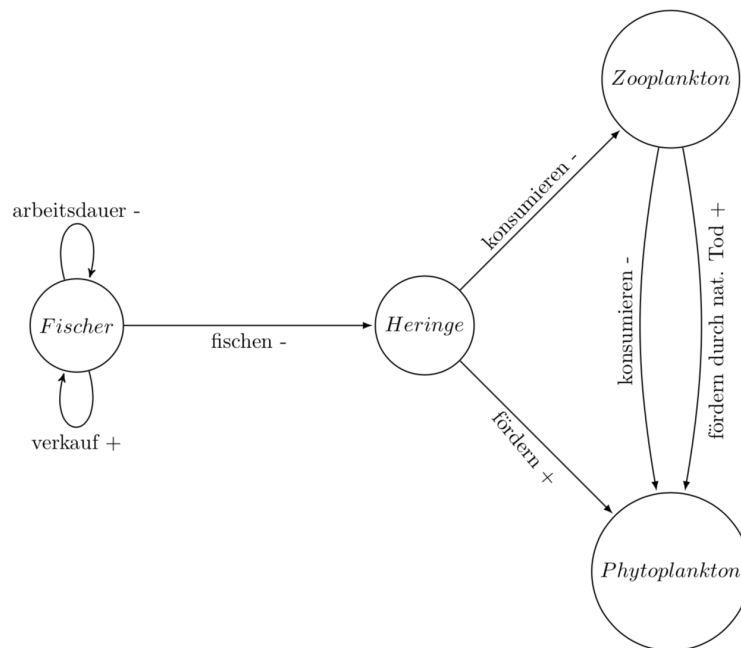


Abbildung 6.2.: Abstraktes Wirkungsdiagramm mit Beziehungen und den Einflüssen zwischen den Zustandsgrößen.

Der Graph zeigt die von außen geltenden Kerneinflüsse, die auf das Nahrungsnetz und die Fischerei *Fischer* gelten sollen. Dazu werden zunächst drei tierische Größen betrachtet, die in einem traditionellen Räuber-Beute Verhältnis zueinander stehen. Heringe konsumieren das Zooplankton, wachsen durch Aufnahme und Umwandlung der konsumierten Biomasse mittels

einer gewissen Umwandlungseffizienz und natürliches Absterben durch Respiration oder Jagd durch eine andere Beute. Der Hering nimmt zunächst nicht die komplette Biomasse auf sondern wandelt nur Anteile konsumierter Biomasse um, sodass daraus neue Heringsbiomasse erzeugt wird. Dasselbe gilt für das Zooplankton, dass mit einer gewissen Rate das verfügbare Phytoplankton konsumiert und durch den Frass der Heringe reduziert wird. Im Gegensatz zur Modellierung mittels gewöhnlichen Differentialgleichung wird hier lokations- und zeitbedingt gehandelt und die Populationen damit als nicht gleichmäßig in der Umgebung betrachtet. Das Phytoplankton ist von den verfügbaren Nährstoffen aus dem Phosphor der Umgebung abhängig, dass wiederum durch natürliches Absterben der Herings-Biomasse und totem Gewebe und ggf. der Ausscheidung wieder versorgt wird. Das gilt jedoch nicht, wenn der Fischer diese aus dem betrachteten Gebiet entfernt und herausfischt. Dieser wird durch den Fang die Heringsbiomasse aus dem System entfernen und mittels Preisbildung, abzüglich der Kosten für zum Beispiel Ausrüstung und Wartung des Kutters, seine eigene Population über das Einstellen neuer Fischer wachsen lassen. Zu Vereinfachung wird an dieser Stelle nicht auf eine konkrete Preisberechnung des Fischers gesetzt, da sich dieser vom Fangort, den jeweiligen gefangenen Fischtypen (Hering, Lachs, Dorsch,...) oder auch Verarbeitungsgrad sehr stark variiert (Götting u. a. (2013)). Wenn die entsprechende Arbeitszeit ausgeht und damit entweder das Arbeitsmaterial aufgebraucht ist (Kutter defekt, Ruhestand des Fischers, Job-Neuorientierung) fällt ein Fischer heraus und reduziert die Population. Verdeutlicht werden die gegenseitigen Beziehungen der ermittelten Zustandsgrößen innerhalb des Wachstumsmodells in Abbildung 6.2

Dieses bietet diverse Möglichkeiten es mit diversen Parametern einzustellen aus der Interaktionssicht heraus, die Einflüsse auf die Entitäten unterschiedliche abzubilden.

Als zentrale Größe bildet der Hering ein Kernelement des Systems. Dessen Wachstum ist gekoppelt sowohl am Erhalt des Phytoplanktons als auch des Zooplanktons. Beide Beutetypen sind im Bereich des möglichen, wohingegen das Zooplankton dem primären Beuteschema entspricht. Das Einfangen des Zooplanktons selbst stellt dabei große Schwierigkeit dar, ist durch die Fangchance (Ergreifungswahrscheinlichkeit) und der Tatsache, dass im aktuellen Umkreis diese Beute überhaupt vorkommt, jedoch beschränkt. Mit Anstieg der eigenen Biomasse wächst jedoch auch die Konkurrenz, und ein neuer Junghering wird im System an der aktuellen Stelle erzeugt. Dieser ist auch zunächst vom eigentlichen Fischfang befreit, da zu junge Fische generell zu klein, sowohl für Schleppnetze, Zugnetze und Fischfallen sind. Um gleiche Beute zu erhalten muss von den Heringen konkurriert werden, d.h. wer zuerst das Zooplankton fängt gewinnt. Dasselbe gilt für die Beziehung zwischen Fischer und Hering. Je größer die Fischereiflotte wird desto mehr Fischkutter befinden sich auf See und haben demnach eine höhere Chance den Hering zu ergreifen, jedoch steigt entsprechend auch die Konkurrenz, sodass Fischer

6.3.2. Verwandte Arbeiten

Die Fischerei stellt eine einflussreiche Domäne in der Ökologie dar. Wechselwirkungen unterschiedlichster Kleintiere bis zu großen Räubern in verschiedene Arealen der Welt haben enormen Einfluss auf die Lebensweise von Mensch und Umwelt gleichermaßen. Ausfälle im Fischfang oder massive Überfischung nehmen Einfluss auf örtliche Gegebenheiten. Im Bereich der agentenbasierten Simulationen haben sich eine Reihe von Modellen etabliert, die auf Wechselwirkungen der hiesiger Spezies eingehen. So haben [Xing u. a. \(2017\)](#) ein dynamisches Ökosystem konstruiert was auf das Gebiet der *Jiaozhou Bay, China* Bezug hat und deren Bathymetrie mit einbezieht. In der Arbeit von [de Mutsert u. a. \(2017\)](#) wurde sich dagegen mehr auf die Biodiversität für küstennahe Spezies konzentriert, die sich im Delta des Mississippi River, Amerika wiederfindet. Sie haben dazu mehrere Modellansätze miteinander gekoppelt und neben der Modellkalibrierung über die Parameter vor allem zwei existierende Simulatoren miteinander verknüpft, bei dem hydrographische Modellergebnisse für ein Ökosystemmodell übernommen wurden. Das eigentliche Modell versucht dagegen lediglich die Biomassen-Verteilung der einzelnen Spezies nachzubilden und umweltbedingte Veränderung zu analysieren, die durch die Fischerei einhergeht. Da die Ergebnisse insbesondere auf das Flussdelta Einfluss nimmt, sind tatsächliche Vergleiche zu dem hier betrachteten Fangmodell nur mit [Xing u. a. \(2017\)](#) oder über die beiden Arbeiten von [Cooper und Jarre \(2017a\)](#) [Cooper und Jarre \(2017b\)](#) möglich. Sie konstruierten ein Offshore Fischerei Modell für die Seefischerei Südafrikas des dortigen Seehechts, eingeteilt nach Flachwasser- und Tiefsee -Fischerei. Dazu kombinieren sie dieses Modell mit industriellen Interessen um möglichst nach profitabelsten Beziehungen zu suchen, die auch mit unterschiedlichen Treibstoffpreisen für die einzelnen Fischer umgehen kann. In beiden Fällen fehlen jedoch feingranulare Wechselwirkungen vorrätiger Nährstoffe für diese Spezies, die das Wachstum und damit auch die Profitabilität der Industrie mitbestimmt.

6.3.3. Kernmodell

Für das Fangmodell muss verdeutlicht werden, wie neue Biomasse entsteht und sich überführen lässt. Im Allgemeinen wird zur Umwandlung ein Wachstum gebraucht, sodass Biomasse über die Zeit t verändert. Diese beute-abhängige und der Menge, eigener hiesiger Fressfeinden, abhängiger Parameter stimuliert hemmt das Wachstum. Die Änderung über Zeit durch eine gewöhnliche DGL beschrieben, die für jedes Individuum das Biomassenzuwachs betrachtet, genauso wie den vollständigen Abgang mit Tod durch Frass bzw. über die Zeit hinweg allein durch Respiration. Die Kern-Systemgleichung bleibt daher gleich. Sie wird für das Agentenmo-

dell dementsprechend auf die Individuen, um damit die korrekt Berechnung einer Biomasse verfeinert und durch die agentenbasierten Konzepte teilweise ersetzt.

$$\begin{aligned}\frac{dN}{dt} &= r \cdot N - f(N, P) \cdot P \\ \frac{dP}{dt} &= k(N, P) \cdot P - g(P) \cdot P\end{aligned}\tag{6.1}$$

Die ursprüngliche zu diesem einfachen und gleich verteiltem DGL-Modell wird folgende Näherung angestrebt:

$$\begin{aligned}N(t) &= N(t - \delta t) + (r \cdot N - f(N, P)) \cdot \delta t \\ P(t) &= P(t - \delta t) + (k(N, P) \cdot P - g(P) \cdot P) \cdot \delta t\end{aligned}\tag{6.2}$$

Sie entspricht dem Kern des Räuber-Beute Modells mit einer spezifizierten funktionellen Antwort $f(N, P)$, einer formulierten Jagd $k(N, P)$ sowie einer spezifizieren Respiration des Räubers $g(P)$. Als Basalart betrachtet, wächst der Räuber über die Wachstumsrate r über eine Quell, zum Beispiel über die Aufnahme von Phosphor oder Sonnenlicht, zunächst unbeschränkt. Der Jäger wächst nur, wenn Biomasse von seiner Beute aufgenommen wird und diese in die eigen Masse überführt wird. Dieser Übergang ist durch die Funktion f und k beschrieben und steht in der Regel in einem ungleichen Verhältnis, bei der die Normalität vorgibt nicht immer die komplette Masse aufzunehmen, sondern begrenzt beispielsweise durch die eigene Verdauung, nur ein Teil davon an die Umwelt abgegeben wird. Es gilt daher $k(N, P) \leq f(N, P)$. Die funktionelle Antwort $f(N, P)$ gibt dabei das Beuteverhalten des Jägerindividuum an. In den meisten vereinfachenden Modellen ist dies meist eine Jagd mit linearem Anstieg. Da hier jedoch mit Individuen gearbeitet wird, liegt der Fokus die einzelne Interaktion zwischen einzelner Räuber und der Beute.

Abbildung 6.4 zeigt hierzu das konzeptionelle Agenten- und Layermodell mit Interaktionen zwischen den Entitäten.

Im Modell existieren vier Agententypen *Fischer*, *Hering*, *Zoplankton* und *Phytoplankton*. Der Hering sitzt zusammen mit dem Plankton auf einem eigens dafür vorgesehenen *MeeresLayer*, während die Fischer an einem spezifischen *FischerLayer* gekoppelt sind. Mit Abhängigkeit zu existierenden Raster-Layern nutzen sie Informationen aus der Umgebung. Alle *Meeresbewohner* nutzen den *PhosphorLayer* um entweder eigene Biomasse hinzufügen oder abzuziehen. Mittels *Bathemtrie* wird neben *TerritorialemSeegebiet* die Karte festgelegt. Heringe interagieren über

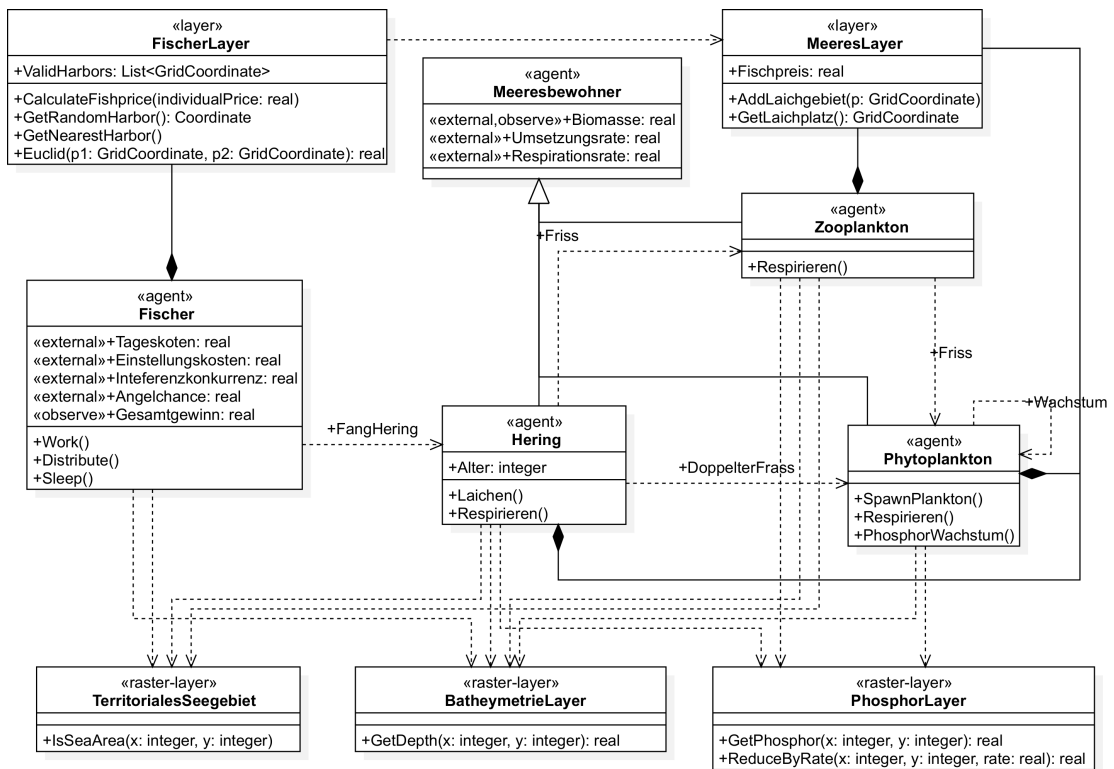


Abbildung 6.4.: Konzeptionelles Fangmodell mit beteiligten Entitäten und Interaktionen

Frass mit dem Plankton, wohingegen die *Fischer* lediglich Einfluss auf die *Heringe* haben können.

6.3.4. Agentenmodell

Die Modellierung der einzelnen Individuen ist von zentraler Bedeutung und umfasst die Modellbeschreibung des Phytoplanktons wie es sich innerhalb der Umgebung ausbreitet sowie das Zooplankton, als zusätzliche Beute für den Hering zur Verfügung steht. Kernelement ist der Hering dessen Verhalten durch seine örtliche Beute und dem Verlange zur Reproduktion an ausgewählten Stellen getrieben wird.

Phytoplankton

Das Phytoplankton oder auch einfache Alge entspricht der Basalart des Nahrungsnetzes. Jeder Schwarm (in der Folge nur als Phytoplankton benannt) ist durch ortsgebundenes Phosphor an eine fixen Position gebunden und besitzt demnach keinerlei Bewegung innerhalb des Modells - in der Folge werden auch keinerlei Strömungen in der See mit modelliert sind, die diese über das Gebiet verteilen würde. Die Ausbreitung findet alleine entgegen des verfügbaren Phosphors statt und breitet sich entlang hoher Nährstoffgrenzen und der Eingrenzung des Seegebiets aus. Da Sonneneinstrahlung als unendlich angenommen wird, ist das Wachstum allein durch örtliche Ausbreitung begrenzt, d.h. irgendwann existiert genügend Plankton (aus Gründen der Dichtbeschränkung und Nährstoffabzug usw.) und das Wachstum der eigenen Biomasse steht im Vordergrund. Die Ausgangsgleichung des Phytoplanktons ist in 6.3 gegeben:

$$\frac{dPh}{dt} = r_{Ph} \cdot Ph \cdot \left(1 - \frac{Ph}{K}\right) - a_Z \cdot Ph \cdot Z \quad (6.3)$$

K wird dabei definiert als Fanggebiet und einer spezifischen Kapazität, die die Biomasse pro Gebietseinheit wiedergibt:

$$K = \text{Fanggebiet} \cdot K_{Ph} \quad (6.4)$$

Die Kapazitätsbeschränkung wird als Maximum der Population festgelegt. Wenn keine Jäger vorhanden sind, wächst die Population in Richtung K oder fällt auf dieses Niveau, sollte zu viel Phytoplankton-Masse $Ph > K$ vorliegen. Das Wachstum des Phytoplanktons wird hauptsächlich durch das Nährstoffniveau im Wasser sowie der Sonneneinstrahlung geregelt. Durch die Kadaver der verstorbenen Heringe und des toten Zooplanktons werden neue Nährstoffe in

6. Evaluation

das Wasser eingeführt und erlauben die Ausbreitung des Planktons auf ein neues Niveau, und damit ein neues Maximum K . Hierzu wird die tote abgegebene Biomasse n_H und n_Z betrachtet, die durch Mortalität oder natürlichem Tods im Wasser übrig bleibt:

$$\begin{aligned}n_H &= m_H H(t) d_H \\n_Z &= m_Z Z(t) d_Z\end{aligned}\tag{6.5}$$

Die Kapazität zum Zeitpunkt t wird aus der toten Biomasse beider Spezies erzeugt, die mit einer Zersetzungsrate für neuen Lebensraum sorgt. Der Einfachheit wurde die individuen-spezifische Zersetzung auf die notwendigen Nährstoffe (Kohlenstoff, Sauerstoff, Phosphor etc.) zum Wachstum des Phytoplanktons vernachlässigt. Sie wird in der Regel durch eine Reihe von Reaktionsgleichungen gebildet die einen unterschiedlichen Anteil jedes Stoffe an die Umwelt abgibt, der sich wiederum durch Sonneneinstrahlung in gewünschte Stoffe neu bildet. Hier sei auf die Arbeit von [Radtke und Straškraba \(1980\)](#) verwiesen. Für die letzte Kapazität K gilt somit Folgendes:

$$K = K_{start} = \frac{\omega \cdot (n_H + n_Z)}{1 + (n_H + n_Z)}\tag{6.6}$$

K_{start} definiert den Anfangsgehalt, gesetzt durch ein verwendetes Phosphor Raster. ω gibt die maximale Differenz der Kapazität an, womit $K_{max} = K_{start} + \omega$ gilt. Die gesamte Kadaver-Masse $n_H + n_Z$ wird ins Verhältnis $(n_H + n_Z) + 1 > 0$ gestellt, sodass das Niveau zwischen K_{start} und $K_{start} + \omega$ oszilliert.

In der MARS DSL wird das Plankton als einfache Entität betrachtet. Zum Simulationsstart wird dieser erzeugt und positioniert sich zur Agenteninitialisierungsphase zufällig innerhalb des Seegebiets. Das Phytoplankton wird zu jedem Zeitschritt vom Phosphor gefördert und reduziert diese um eine gewisse extern parametrisierte Umwandlungseffizienz.

```
1 model Fangmodell
2
3 use Mars
4 use PhosphorRasterLayer as phosphor
5 use BathymetrieLayer as bathymetrie
6
7 agent Phytoplankton on MeeresLayer {
8
9     external var Beschränkt : bool
10    external observe var Biomasse : real
11    external var Umsetzungsrate : real
```

```

12     external var Respirationsrate : real
13     observe var AbgebafterPhosphor : real
14     var IsAlive : bool = true
15
16     initialize {
17         Respirationsrate = bathymetrie.GetDepth(#(xcor, ycor)) *
18             Respirationsrate
19         if(Respirationsrate < 0) Respirationsrate = 0.0001
20     }
21
22     reflex {
23         // Beschränkt auf Phosphor Vorkommen oder nicht
24         var aktuellePhosphorKapazitaet = 0.0
25         if(Beschaenkt)
26             aktuellePhosphorKapazitaet = phosphor.ReduceByRate
27                 (#(xcor, ycor), Umsetzungsrate) as real
28         else
29             aktuellePhosphorKapazitaet = phosphor.GetValue(#(
30                 xcor, ycor), Umsetzungsrate) as real
31
32         AbgebafterPhosphor = Umsetzungsrate *
33             aktuellePhosphorKapazitaet
34         if(AbgebafterPhosphor > 0) {
35             var planktonDichteAufEinemFeld =
36                 length(explore Phytoplankton where
37                     [it => return (distance(it)) <= 1])
38             var biomassenZuwachs = 1 + ((AbgebafterPhosphor *
39                 planktonDichteAufEinemFeld) / 1 +
40                 planktonDichteAufEinemFeld * aktuellePhosphorKapazitaet
41                 * 60)
42
43             Biomasse = Biomasse + biomassenZuwachs
44             println "Neue Biomasse von " + Biomasse
45                 + " durch Zuwachs von " + biomassenZuwachs
46                 + " und Phosphor Abbau" + AbgebafterPhosphor
47         }
48
49         if(Biomasse > 100) {
50             SpawnPlankton(Biomasse / 2)
51             Biomasse = Biomasse / 2
52         }

```

```

48         Respirieren ()
49     }
50     ...
51 }

```

Listing 6.1: Teil des Phytoplankton Modells mit Phosphor Bezug

Das Phytoplankton wird extern durch die *Umsetzungsrate* r_{ph} und seine initiale *Biomasse* B_{ph_i} parametrisiert. Ersteres gibt das Wachstum durch das vorhandene Phosphor der Umgebung an, wie viele davon in eigene Biomasse umgewandelt wird. Letzteres stellt die zu observierende Agenteneigenschaft dar, die zur Simulationslaufzeit in jedem δt aus dem System raus geschrieben wird. Um eine Wachstumsbeschränkung zu erreichen, wird dem *Phytoplankton* eine Dichtebegrenzung mitgegeben, die über die Anzahl vorhandener Algen in der Umgebung von $(X \pm 1, Y \pm 1)$ bestimmt wird, d.h. das Phytoplankton breitet sich nicht endlos auf einem Feld aus. Das richtet sich auch danach wie das Plankton seine Respiration berechnet. Dieser aus der Bathymetrie abhängige Wert wird gebildet durch eine Ausgangsrespiration d_{ph} die zur Initialisierung des Agenten fix über die aktuelle Tiefe verrechnet wird.

$$d_{ph_i} = \begin{cases} b_{xy} \cdot d_{ph} & \text{if } n > 0 \\ d_{ph} \cdot 0.001 & \text{if } n \leq 0 \end{cases} \quad (6.7)$$

Beim *Phytoplankton* handelt es sich um eine sterbende Entität, erkennbar an der Definition durch die gesonderte *IsAlive* Eigenschaft. Dessen Initialisierungswert ist im Normalfall gültig. Andernfalls fällt dieser Agent vollständig aus dem System raus und wird niemals in der Umgebung erzeugt.

Die Workflow-Definition des Zeitschritts (*reflex*) beschreibt das *Phytoplankton* als Agenten der zunächst das Phosphor des *PhosphorLayers* über die eigene Position abfragt und bei aktiv markierter Beschränkung *Beschaenkt* reduziert. Sollte Phosphor vorliegen wird dies zur Bildung neue Algen herangezogen. Diese dichteabhängige Beschränkung wird modelliert als Prüfung über die Nachbarzellen $(X \pm 1, Y \pm 1) \leq 5$. Das Gleiche gilt für die Verbreitungsart des *Phytoplanktons*, was ähnlich eines Ausbreitungsmodells - zum Beispiel Feuer - in jene Richtung stattfindet, wo sich das meiste Phosphor zum aktuellen Zeitpunkt t befindet. Die formulierte Spawn Aktion in 6.2 exploriert dazu auf der Umgebung nach einer Menge von Phytoplankton was sich in unmittelbarer Nähe befindet, um dazu die Anzahl des Phytoplanktons abzufragen. Der Lambda Ausdruck leitet hierzu den Typen aus der Exploration ab und fragt diese über die *euklidische distance* ab die für diesen Fall kleiner oder gleich 1 sein soll.

```

1 def SpawnPhytoplankton () {
2     var entities =

```

```

3       explore Phytoplankton where [ it => distance(it) <= 1]
4       if(Biomasse > 100 and length(entities) < 8) {
5         var fields = GetNeighbourCellsBy().Sort[it => phosphor.
6           GetValue(it)]
7         each(field in fields) {
8           if(not entities.Any[it => it.Position === field]) {
9
10          var plankton = spawn Phytoplankton at
11          plankton.SetBiomasse(Biomasse / 2)
12          Biomasse = Biomasse / 2
13          return;
14        }
15      }
16    }
17  }

```

Listing 6.2: Ausbreitungsmodell des Phytoplanktons

Wichtig für diese Aktion ist insbesondere die Frage nach den Nachbarzellen *GetNeighbourCells* die aus der aktuellen Position gezogen werden und als sortierte Liste dem Aufrufer zurückgegeben werden. Die MARS DSL Listen Implementierung bietet hierzu einen internen Sortiermechanismus $Sort((T) \Rightarrow real) : List<T>$ der dazu über einen Selektor diesen auf einen reellen Wert abbildet und dessen natürlich Ordnung nutzt.

Hering

Als zentraler Jäger liegt der Schwerpunkt des Herings im Fang von Zooplankton und alternativ von Phytoplankton sollte erstes nicht auffindbar sein. Das Modell dass die Jagd auf das Zooplankton regelt wird als funktionelle Antwort entwickelt und ist meist ein linearer Anstieg (Götting u. a. (2013)):

$$\begin{aligned}
 g_{HZ}(Z) &= a_{HZ} \cdot Z \\
 g_{HP_h}(Ph) &= a_{HP_h}
 \end{aligned}
 \tag{6.8}$$

Über eine Jagdrate $a_H Z$ und $a_H Ph$ wird die Wahrscheinlichkeit zum Fang des Planktons gesteuert.

6. Evaluation

Die Jagd selbst würde in der DGL als Handhabungszeit mit modelliert werden, die sich aus der Gesamtpopulation der betrachteten Beute wie des Zooplanktons ergibt.

$$h_H = T_{Gesamt} - h_{Behandlungszeit} \cdot Z \quad (6.9)$$

$T_{Jagdzeit}$ wird in vereinfachender Form durch $T_{Jagdzeit} = h_H \cdot H$ modelliert. Die resultierende funktionelle Antwort wäre demnach:

$$g_{HZ}(Z) = \frac{a_{HZ} \cdot Z}{1 + h_H \cdot a_{HZ} \cdot Z} \quad (6.10)$$

Im Agentenmodell weicht die Handhabungszeit h_H der Ausdehnung des Jagdgebiets des Herings über ein formuliertes Distanzprädikat zur Explorationsaktion *explore Zooplankton where it => distance(it) <= radius*], formell ähnlich zusammengefasst als $\forall a : \mathbb{P}Zooplankton \bullet d(a) <= radius$. Je größer der Jagdradius des Herings wird, desto höher wird folglich die Jagdrate bei weit entfernter Beute. Da sich der Hering keine zufällige Beute wählt, sondern möglichst den naheliegendsten, schwankt der Wert. Diese Oszillation wird bei der Respiration des Herings fortgesetzt. Genau wie beim Phytoplankton wird diese von der aktuellen Tiefe b_{xy} beeinflusst. Im Gegensatz zur Alge ist sie jedoch nicht fix sondern variiert von der aktuellen Position des Herings. Gleichung 6.11 gibt zu einer aktuellen Tiefe aus der Heringsposition x und y die Respirationrate für Tick t_i an. Im Gegensatz zur Phytokultur in Gleichung 6.7 ist bei sehr flachen Wassern $b_{xy} > 0$ die Respiration auf die Hälfte der Ausgangsrespiration festgesetzt.

$$d_{H_i} = \begin{cases} b_{xy} \cdot d_H & \text{if } b_{xy} > 0 \\ \frac{d_H}{2} & \text{ansonsten} \end{cases} \quad (6.11)$$

Je tiefer die Position ist, desto stärker wirkt diese auf die Respiration des Individuums. Begründet wird das durch die niedrigeren Temperaturen die auf diese Wechselwirkung Einfluss hat und bei geringer Wärmeeinwirkung eher hemmend ist [de Mutsert u. a. \(2017\)](#).

Mit der MARS DSL nachgebildet ergibt sich für die Agentenlogik des Hering der Ausschnitt

```

1 agent Hering on MeeresLayer {
2
3     external var Lifepoints : real
4     external var Biomasse : real = 100
5     external var Umwandlungseffizienz : real = 0.56
6     external var Fangchance : real = 0.75
7     external var Ausgangsrespiration : real = 0.009
8

```

6. Evaluation

```
9     var Laichziel : Tuple<integer , integer >
10    var NaechsterFrass : Zooplankton
11    var DoppelterFrass : Phytoplankton
12    var Alter : real = 1
13    var IsAlive : bool = true
14
15 }
16 ...
```

Listing 6.3: MARS DSL Modell eines Hering

Der Abgang durch Befischung, entspricht funktionellen Antwort $g_{FH}(H)$. Da nicht die gesamte Biomasse umgewandelt wird, muss ein zusätzlicher Parameter eingeführt werden, der die Umwandlung von Biomasse einer Spezies i zu einer Spezies j angibt und sich explizit auf die Masse des Herings und seiner Beute des Zooplanktons stützt. Definieren lässt sich diese Umwandlung mittels:

$$\omega_{ij} = \omega \cdot \frac{b_i}{b_j} \quad (6.12)$$

ω gibt dabei die individuellen spezifische Verwertung der konsumierten Masse b_i bzw. b_Z für das Zooplankton an zum Verhältnis der eigenen Masse b_j bzw. b_H für den Hering. Abgebildet ergibt sich bei verfügbaren Zooplankton aus der Umgebung die (*nearest Zooplankton*).*GetBiomasse / Biomasse*

Zooplankton

Das Zooplankton wird analog zum Hering konstruiert, mit der Ausnahme dass dieser nicht irgendwelche Laichgebiet aufsuchen muss, um sich fortzupflanzen. Stattdessen wächst dieser entgegen der Phytoplankton Population und frisst diese Algen ab. Sollte genügend Biomasse $B_i > 100$ erneut zur Verfügung stehen wird sich an Ort und Stelle fortgepflanzt, bei dem durch einfache Teilung die eigene Masse $\frac{B_i}{2}$ um die Hälfte entfällt. Erneut wird bei der Prüfung auf Zerfall mit $B_i < 1$ der sich durch den ortsabhängigen und prozentualen Biomassenabbau nach Gleichung 6.11 ergibt.

```
1 agent Zooplankton on MeeresLayer {
2
3     external observe var Biomasse : real = 100
4     external var Umsetzungsrate : real = 0.03
5     external var Respirationsrate : real = 0.015
6     external var Fangchance : real = 0.3
7     var IsAlive = true
8     ...
```

9 }

Listing 6.4: MARS DSL Modell von Zooplankton

In der MARS DSL des Abschnitts 6.4 wird das Zooplankton als ein solcher Agent definiert, der als Zwischengröße zunächst respiriert und ggf. sofort wegstirbt. Sollte das Zooplankton noch nicht ausfallen, wird nach naheliegenderem Phytoplankton gesucht auf das über die eigene Fangchance a_Z versucht wird dies im losen Gewässer abzufressen. Dieser begrenzte, über r_Z parametrisierte Biomassenabgang gibt die Umwandlung der gefressenen Biomasse pro Zeitschritt an und erhöht dahingehend die eigene Masse. Dieser Fraß funktioniert jedoch nur soweit, wie diese Algen in der Umgebung existieren. Andernfalls wird zufällig entlang des Prädikat eines erfüllten Prädikats $IsSeaAre(xcor, ycor)$ gewandert.

```

1 use TerritorialSeegebietLayer as sea
2
3 agent Zooplankton on MeeresLayer {
4 ...
5 reflex {
6     var aktuelleRespiration = Respirationsrate * bathymetrie .
7       GetNumberValue(xcor, ycor);
8     Respirieren(aktuelleRespiration)
9     if(Biomasse <= 1) IsAlive := false
10    else {
11        var phytoplankton = nearest Phytoplankton
12        if( phytoplankton != nil) {
13            if(random() <= Fangchance
14              and phytoplankton.Position == #(xcor, ycor)
15              ) {
16                Biomasse := Biomasse + phytoplankton.
17                  ReduceBiomasseByRate(Umsetzungsrate)
18            } else move to phytoplankton with random(2) + 1
19        } else move to #(xcor + random(-1,1), ycor + random(-1,1))
20        while [x,y => sea.IsSeaArea(x,y)]
21    }
22
23    if(Biomass > 100) {
24        var zooplankton = spawn Zooplankton
25        zooplankton.SetBiomass(Biomass / 2)
26        Biomass := Biomass / 2
27    }
28 }
29 ...

```

Listing 6.5: MARS DSL Kernverhalten des Zooplankton

Wird genügend Biomasse angesammelt, wird an Ort und Stelle neues Zooplankton erzeugt mit der Hälfte der eigenen Masse. An dieser Stelle wird ähnlich dem Phytoplankton die Reaktionsgleichung vereinfacht die ggf. prägenden Einfluss auf die eigene Masse haben kann.

Fischer

Der Fischer stellt eine externe Einflussgröße im gesamten Nahrungsnetz dar. Sein Ziel ist der Fang von Heringen mit Gewinnerzielung zur Deckung eigener Kosten, wie z.B. Instandhaltung der Ausrüstung (Kutter, Angelbedarf, ...). Bei jeder Möglichkeit versucht er die Fischerflotte zu vergrößern, sofern genügend Gewinn angesammelt wurde. Der Alltag jedes einzelnen Fischers ist dabei zeitbedingt. Von Morgens bis Nachmittags wird früh mit der Fischerei begonnen, während ab dem Nachmittag versucht wird den Fang zu vertreiben und die Kosten über die Einnahmen zu kalkulieren. Die Darstellung des DSL Modellcodes ist in 6.6 dargestellt. Als Hinweis muss hier angefügt werden, dass im Normalfall der Fischer morgens wieder in den Hafen einlaufen würde um seinen Fang zu verkaufen, hier wurde darauf verzichtet, da dies sehr hohe Laufzeitprobleme verursachen würde bzw. es sich um aktive Agenten handelt bei dem jeder einzelne Agent Explorations- und Bewegungsaktionen ausführen kann, dessen algorithmische Breitensuch-Lösung (Weyl u. a. (2018)) und aus Speichereffizienz nicht mit dieser Menge umgehen kann.

```

1 model Fangmodell
2
3 agent Fischer on FischerLayer {
4
5     var IsAlive : bool = true
6
7     var Haltbarkeit : real = 100
8     var BisherigerFang = new List<Hering >()
9     external var Fischpreis : real = 20
10    external var Angelchance : real = 0.56
11    external var Kosten : real
12    external var Einstellungskosten : real
13    observe var Gesamtgewinn : real
14    external var Interferenzkonkurrenz : real
15
16    reflex {
17        if (Time :: TimeOfDayHour( simtime ) >= 5

```

```

18         and Time :: TimeOfDay( simtime ) <= 15) {
19             Work()
20         }
21     else if( Time :: TimeOfDayHour( simtime ) > 15
22             and Time :: TimeOfDay( simtime ) <= 18) {
23         Distribute()
24     }
25     else {
26         Freetime()
27     }
28 }
29 ...
30 }

```

Listing 6.6: MARS DSL Modell des Fischers

In der MARS DSL wird der *Fischer* als Agent dargestellt, gruppiert über den *FischerLayer* und als sterbende Entität ebenfalls mit dem gesonderten *IsAlive* Flag definiert. Des Weiteren umfasst seine Parametrisierung eine steuerbare *Angelchance* a_F sowie einen Kostenfaktor d_F , der ähnlich der Respiration beim *Hering* und *Zooplankton* fungiert und pro Tag beim Einlauf im Hafen vom übrigen Gewinn abgezogen wird. Dem entgegen wirkt die Einnahme durch den Fang der die eigenen Kosten decken sollte. Der DSL-Ausschnitt in 6.7 zeigt den Workflow wie der Fischer versucht Heringe einzusammeln.

```

1 def Work() {
2     var hering = nearest Hering
3     var konkurrenz = nearest Fischer where [ it => distance(it) <= 30]
4     FangHering(hering, konkurrenz)
5     Haltbarkeit = Haltbarkeit - random()
6     if(Haltbarkeit <= 0) {
7         IsAlive = false
8     }
9 }

```

Listing 6.7: MARS DSL Modell der Arbeitsaktion des Fischers

In der Arbeitsphase sucht der Fischer zunächst nach naheliegender Heringsbestand, genauso wie nach Konkurrenz. Über ein Prädikat $distance(it) \leq 30$ wird sichergestellt, dass nur die Konkurrenz innerhalb eines gewissen Radius berücksichtigt wird. In der Aktion *FangHering* wird auf diese eingegangen und über die Angelchance versucht die einzufangen. Ist diese Aktion ausgeführt worden wird um einen geringfügigen Zufallswert *random*, im abgeschlossenen Intervall $[0, 1]$ die *Haltbarkeit* des Fischers reduziert. Bei jeder Fangaktion verringert sich die

eigene Haltbarkeit, bis zur Erfüllung des Prädikats *Haltbarkeit* ≤ 0 um den Fischer aus dem System zu entfernen.

```

1 active FangHering(hering : Hering, konkurrenz : Fischer) {
2     var entfernteKonkurrent = distance(konkurrenz)
3     var heringsEntfernung = distance(hering)
4     if(Math.Abs(entfernteKonkurrent - heringsEntfernung) >=
5         Interferenzkonkurrenz) {
6         if(hering.Position === #(xcor, ycor)) {
7             if(hering.GetAlter > 6
8                 and random() >= Angelchance) {
9                 BisherigerFang.Add(hering)
10                hering.Die
11            }
12        } else move to hering with 10
13    } else move to nearest Hering where [it => return it != hering]
14 }

```

Listing 6.8: MARS DSL Modell der aktiven Aktion zur Abbildung eines Fangversuchs für Heringe

Die *Einstellungskosten* des Fischers repräsentieren den Preis um neues Personal anzuwerben, die erst durch den kumulierten *Gesamtgewinn* überhaupt möglich werden. Der dazu notwendige Umsatz wird dabei durch den gleichermaßen geltenden Fischpreis gesetzt und gebildet aus dem *BisherigenFang*, verrechnet mit dem täglich schwankendem Fischpreis.

$$U_F = |Fang| \cdot p_F \cdot |\sin(\text{day}(t_i))| \quad (6.13)$$

Abzüglich der eigenen Kosten bleibt der Gewinn übrig. Diese Einstellungs- und Verkaufsphase ist in der DSL ebenfalls als eigene Aktion ausgelagert, in 6.9.

```

1 def Distribute() {
2
3     Harbor = fischerLayer.GetNearestHarbor(#(xcor, ycor))
4     if(Harbor != nil and Harbor === #(xcor, ycor)) {
5         var umsatz = BisherigeBeute.Count * IndividuellerPreis *
6             Math::Abs(Math::Sin(Time::Day(simtime)));
7         Gesamtgewinn = Gesamtgewinn + (umsatz - KumulierteKosten)
8         if(Gesamtgewinn >= Einstellungskosten) {
9             var anzahl = (Gesamtgewinn / Einstellungskosten) as
10                integer
11             while(anzahl > 0)
12                 { spawn Fischer; anzahl— }
13         }
14     }
15 }

```

```

11         }
12     }
13     } else move to Harbor with random(5,15)
14     if(Gesamtgewinn >= Einstellungskosten)
15         if(Harbor == nil) Harbor = fischerLayer.GetRandomHarbor
16         if(Harbor === #(xcor,ycor)) {
17             var anzahl = (Gesamtgewinn /
18                 Einstellungskosten) as integer
19             while(anzahl > 0) { spawn Fischer; anzahl—
20         }
21     } else move to Harbor
22     BisherigerFang = new List<Hering>()
23 }

```

Listing 6.9: MARS DSL Modell der Verkaufsaktion des Fischers

Die Bewegung des Fischers findet entgegen einer Passierbarkeit auf der Bathymetrikarte statt. Dazu wird zur Zufallsbewegung des Fischers die nächste zu befahrende Zelle auf dem *BathymetrieRasterLayer* geprüft. Ist die Bedingung für *IsPassable* erfüllt wird die Bewegung ausgeführt:

```

1 var coord = RandomNextCoord()
2 if(bathymetrie.IsPassable(coord)) move to coord

```

Listing 6.10: MARS DSL Aktion zur blockierten Fahrt

Die gesonderte Behandlung von *IsPassable* und *IsSwimmable* bezieht sich darauf, dass sich Fischer in der Nähe der Küsten sich aufgrund von zu geringer Wassertiefe nicht fortbewegen können. Diese Fahrweise ist auf dem tatsächlichen Gebiet natürlich nicht realistisch, insbesondere bei Rückfahrten zu einer Hafenanlage während der *Distribute* Phase. Es wird eine Routenberechnung benötigt um die Fischer um Halbinseln bzw. um Inseln fahren zu lassen.

6.3.5. Layermodell

Die einzelnen Agenten werden über einzelne Layer zusammengefasst. Das Fangmodell unterscheidet zwei Layer Typen. Der *MeeresLayer* beherbergt alle tierischen und pflanzlichen Entitäten {Hering, Phytoplankton, Zooplankton} und bietet den Heringen die Möglichkeit für gemeinsame Informationen. Ein gemeinsames Laichgebiet wird von den Heringen geteilt, zu denen sie zur Reproduktion zurückkehren können. Der *FischerLayer* gruppiert die Fischer im Modell und berechnet variierende Fischpreise sowie Informationen über Heimathäfen, zu den nach der Arbeit zurückgekehrt werden kann.

Meeres Layer

Der *MeeresLayer* bildet die Gruppierung über das Zooplankton, Phytoplankton und die Heringe. Für die konkrete Syntax gilt zur Definition nur das Setzen des Bezeichners.

```

1 model Fangmodell
2
3 layer MeeresLayer as meer {
4
5     var Laichziele : List<Tuple<integer , integer >> =
6         new List<Tuple<integer , integer >>()
7
8     def GetLaichplatz ()
9         => return Laichziele .Get(random(Laichziele .Size))
10
11    def AddLaichgebiet(coord : Tuple<integer , integer >)
12        => if(Laichziele .Size <= 150) Laichziele .Add(coord)
13 }

```

Listing 6.11: Layer zur Abbildung der Meeresbewohner

Die MARS DSL erlaubt durch Definition von *layer MeeresLayer* Elementen eine einfache und schnelle Form zur Formulierung von Agenten-Layern und damit einer Gruppierung unterschiedlicher Agententypen, die daran gekoppelt sind.

Fischer Layer

Der *FischerLayer* besitzt eine Abhängigkeit zum *MeeresLayer* um den Fischern die Heringe bekannt zu machen. Für die Fischer gibt es eine Berechnungsfunktion für den Fischpreis - wird bei externer Parametrisierung nicht genutzt. Ausschnitt 6.12 zeigt den definierten *FischerLayer* mit einer List von Häfen *List<Tuple<integer, integer>* sowie eine generellen Fischpreis.

```

1 model Fangmodell
2
3 use Mars
4
5 layer FischerLayer as fischerLayer => [ MeeresLayer ] {
6
7 var ValidHarbors : List<Tuple<integer , integer >>
8 var CommonFishprice : real = 10
9 ...
10 }

```

Listing 6.12: Layer zur Abbildung der Fischer mit gemeinsam geteilten Daten

Im Gegensatz zum *MeeresLayer* 6.3.5 sieht der Fischer eine Abhängigkeit zu Heringen die das Setzen einer Sichtbarkeitsbeziehung notwendig macht. Als Abhängigkeitsbeschreibung mit *fischerLayer => [MeeresLayer]* wird den darauf befindlichen Fischern die unidirektionale Wahrnehmung von Heringen ermöglicht, sprich *meeresLayer => [FischerLayer]* würde als zyklische Abhängigkeit erkannt und als Fehler darauf hingewiesen werden.

```

1
2 def CalculateFishprice (individualPrice : real)
3     => Math :: Abs (Math :: Sin (Time :: Day (simtime))) *
4         (individualPrice + CommonFishprice) / 2
5
6 def SetNewCommonFishprice (fishPrice : real)
7     => CommonFishprice = fishPrice
8
9 def AddValidHarbor (harbor : Tuple <integer , integer >)
10    => ValidHarbors .Add (harbor)
11
12 def GetNearestHarbor (origin : Tuple <integer , integer >)
13    => return ValidHarbors .Max (harbor => Euclid (origin , harbor))
14
15 def Euclid (p1 : Tuple <integer , integer >, p2 : Tuple <integer , integer >)
16    => return ((p1 .Item1 - p2 .Item1)**2 + (p1 .Item2 - p2 .Item2)**2)**0.5

```

Listing 6.13: Layer der Fischer mit gemeinsam geteilter Funktionalität

Den Fischern wird geboten über die Eingabe eines individuellen Fischpreises eine tagesabhängige Preisberechnung vorzunehmen *CalculateFishprice*. Sollten die Fischer einen Individualpreis besitzen, wird über den Mittelwert der Verkaufspreis zu den monatlichen Schwankungen berechnet. Realistischer müsste man an dieser Stelle auf ein Angebot und Nachfrage-Modell setzen - zum Beispiel als Schweinezyklus Modell - und sich sowohl am Vorkommen als auch gesetzlichen Fang- und Preisquoten richten. Weiterer Punkt der vom *FischerLayer* verantwortet wird, ist die Verwaltung valider Häfen, die als Liste im Modellcode implementiert sind und bei Abfrage einen zufälligen oder über den euklidischen Abstand naheliegendsten Hafen zur einer gegebenen Distanz zurückgibt. In der MARS DSL ist mit dem Konzept des Lambda Ausdrucks die Möglichkeit zur Bildung anonymer Funktion möglich die zur Abfrage und Manipulation von Collections sich bereits bewährt hat.

6.3.6. Umgebungsmodell

Die Umgebung wird gebildet aus einer Eingabe von Rasterkarten über das *raster-layer* Konzept 4.3.2. Es erzeugt eine dafür angepasstes Grid-Umgebung auf der jeder Agent, verteilt auf

jeweiligen Feldern liegt. Agenten können sich Felder teilen und sich in 8 Richtung fortbewegen (oben, unten, links, rechts, diagonal). Zu jedem Feld (x, y) existiert äquivalent ein Feldeintrag im eingegebenen Raster. Der Wert des Rasterfelds bestimmt das weitere Handeln und bietet das Fangmodell die Möglichkeit darüber sowohl Bewegungsaktionen als auch Berechnung auszuführen.

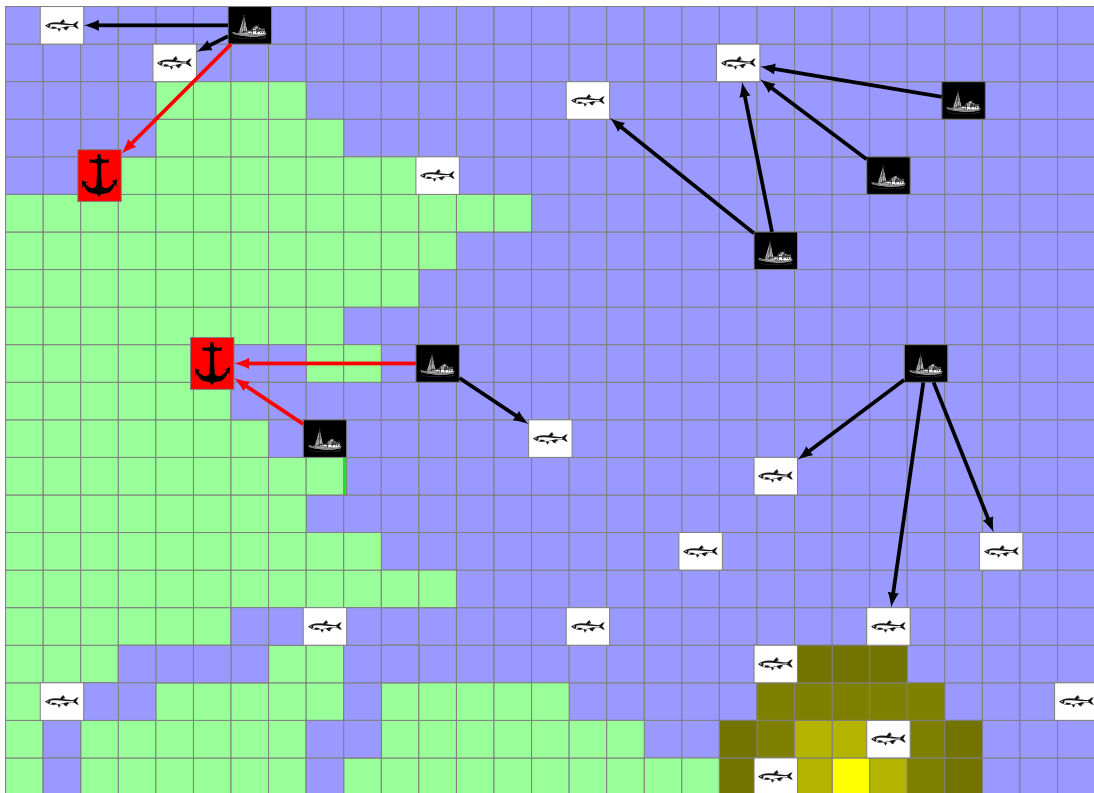


Abbildung 6.5.: Beispielhafte Darstellung einer diskreten Umgebung mit Fischern (schwarz), Heimathäfen (rot), Heringen (weiß) und einem Laichgebiet (gelb)

Das Gebiet liegt inhärent in der diskreten, DSL-spezifischen Agentenumgebung bereit, dessen eingegebenen Rasterkarten zur Unterscheidung zwischen Küsten und Seegebiet 6.3.6, einer Tiefenkarte und besonderen Positionen wie Laichgebieten und konkreten Heimathäfen direkt als Zelle markiert sind. Die Umgebung wird automatisch aus der Dimensionierung der Raster Spezifikation abgeleitet, dabei gilt, dass die Kartengröße zueinander kongruent sein muss.

Territoriales Seegebiet

Zur Unterscheidung, ob sich ein Agent auf einem Küstengebiet/Landgebiet oder Seegebiet befindet, gibt es zwei Ansätze. Zum einen lässt sich einfach über die Karte des *territorialen Seegebiets* eine Datenabfrage mit Vergleich der Differenzierungswerte ermöglichen. Je nach Rastereingabe variiert dieser Wert. Der Modellcode Ausschnitt 6.14 zeigt dazu die Eingabe eines territorialen Seegebiets.

```

1 model Fangmodell
2
3 raster-layer TerritorialSeegebietLayer as sea {
4
5     def InSeaArea(coordinate : Tuple<integer , integer >)
6         => return sea.GetBoolValue(coordinate.Item1 , coordinate.
7             Item2)
8 }

```

Listing 6.14: Raster-Layer zur Abbildung des territorialen Seegebiets

Der Raster-Layer wird definiert durch den Bezeichner sowie einen referenzierbaren Alias, über den auf die dahinterliegenden Methoden zugegriffen werden kann. Wie in Abschnitt 6.3.5 bereit dargestellt wird die Abfrage nach *InSeaArea* bei der Initialisierung des Phytoplanktons vorgenommen, um diese Entitäten gleich verteilt in der See zu haben.

Bathymetrie

Eine anderen Lösung zur Sicherstellung des Seegebiets liegt in der Verwendung über eine vorhanden *Bathymetrie* Karte. Mittels Prüfung ob sich an der entsprechende Stelle (x, y) ein Wert auf der Karte befindet, und demnach an dieser Stelle ein Wassergebiet generell existiert, unabhängig der konkreten Wassertiefe. Die Abfrage stützt sich damit auf die korrekte Beschneidung des Rasters. Fehlende Werte werden durch das in der Rasterspezifikation ausgewiesen `NO_DATA_VALUE` gekennzeichnet, von der an dieser Stelle durch die DSL abstrahiert wird. Der DSL Ausschnitt 6.15 zeigt die Integration der Bathymetrie-Karte, von außen.

```

1 model Fangmodell
2
3 raster-layer BathymetrieLayer as bathymetrie {
4
5     def IsPassable(coordinate : Tuple<integer , integer >)
6         => return bathymetrie.GetNumberValue(
7             coordinate.Item1 , coordinate.Item2) <= -4.5
8 }

```

```

9      def IsSwimmable( coordinate : Tuple<integer , integer >)
10         => return bathymetrie .GetNumberValue(
11             coordinate .Item1 , coordinate .Item2) <= 35.0
12     }

```

Listing 6.15: Raster-Layer zur Abbildung der Bathymetrie Karte

Erneut wird ein *raster-layer* definiert der über einen Alias *bathymetrie* abgefragt wird. In der Zwischenschicht wird auf die Besonderheit der Karte mit den beiden Methoden *IsPassable* und *IsSwimmable* eingegangen. Sie prüfen über die fixen Werte -4.5 und 35.0 , ob das gegebene Feld (x, y) genügend Wassertiefe bietet um sie mit einem Kutter zu befahren oder als Fisch durchschwimmen zu können. Diese Unterscheidung ist aufgrund die niedersächsischen Wattenmeers interessant, da sich Fischer an dieser Stelle nicht immer fortbewegen könne. Zur Vereinfachung wurde an dieser Stelle auf eine Modell für Ebbe und Flut verzichtet.

Nährstoffniveau

Zur Förderung des Phytoplanktons wird eine Phosphor Karte gebraucht, die das Nährstoffglied innerhalb des Nahrungsnetzes der Alge darstellt. Allein Verfügbarkeit des Phosphors beschränkt das Wachstum. Bei Zerfall anderer Populationen {Heringe und Zooplankton} durch Respiration sind die Überreste neue Grundlage die vom Phosphor Raster-Layer, mit einer spezifizierten *Umwandlungseffizienz*, wieder an die Umgebung abgegeben wird.

```

1  model Fangmodell
2
3  raster-layer PhosphorRasterLayer as phosphor {
4
5      def ReduceByRate( coordinate : Tuple<integer , integer >, rate : real)
6          {
7              var currentPhosphor = phosphor .GetNumberValue(
8                  coordinate .Item1 , coordinate .Item2)
9              if( currentPhosphor > 0) {
10                 var abbau = currentPhosphor * rate;
11                 phosphor .Reduce( coordinate .Item1 ,
12                     coordinate .Item2 , abbau)
13                 return abbau
14             }
15     }

```

Listing 6.16: Raster-Layer zur Abbildung der Phosphor Konzentration

Der Phosphor Raster-Layer bietet zwei Methoden zur Abfrage und gleichzeitigen Verringerung *ReduceByRater* des Phosphor Wertes an der aktuellen Position *coordinate*. Über die Abfrage typisierte Abfrage wird der aktuelle Phosphor Wert gelesen und bei einem existierenden Phosphor Gehalt, dieser um die Rate *rate* verringert und der abgebaute Wert zurückgegeben. Aus dem Pfad erkennt man dieser Stelle die Fähigkeit generische Rückgabetypen zu erzeugen, da nicht alle Pfade eine Rückgabe besitzen müssen.

6.3.7. Experimente

Im folgenden Abschnitt sollen verschiedene Resultate zum Modell präsentiert werden. Unter anderem wurde geprüft wie unterschiedliche Fangraten Einfluss auf das Wachstum der individuellen Agententypen hat um damit sicherzustellen, dass die resultierende MARS DSL der Semantik und der Übersetzung vollständig oder teilweise dem Wunschverhalten entspricht. Betrachtet wurde hierzu das Gebiet des Niedersächsischen Wattenmeers, entlang der Nordsee inklusive einem Anteil an der Holländischen See. Das gesamte Gebiet ist $24.676\text{km}^2 = 199\text{km} \cdot 124\text{km}$ groß, mit einer gewählte diskreten Umgebung von 500×500 Feldern. Jedes Experiment wurde mit der Startpopulation von: $H(t_0) = 300$, $H(t_0) = 750$, $Ph(t_0) = 1000$ und $F(t_0) = 10$, abweichend angegeben, ausgeführt.

Experiment - Heringswachstum

Das Phytoplankton fördert das Zooplankton und die Heringe. Das Wachstum des Phytoplanktons ist dabei lediglich vom verfügbaren Phosphor aus der Umgebung abhängig, das ohne dynamischen Nährstoffgehalt durch Absterben der Heringe keinen zusätzlichen Einfluss erhält. Stattdessen wird der Hering und damit seine Biomasse beim Absterben einfach aus dem System entfernt. Die anfänglich existierende Heringspopulation sorgt ebenfalls dafür das damit genügend Beute für die Fischer im Umlauf ist und aus diesem Grund ohne viel Aufwand und Wege diese zügig eingefangen werden können, weswegen mit der jeweiligen Fangchance auch diese zu Beginn abfallend ist. Interessant ist an dieser Stelle die Abweichung der Fangchance a_F ab dem Zeitpunkt $t = 70$. In allen drei Fällen entsteht die typische Wellenbewegung, die bereits in anderen Fischereimodellen festgestellt wurde, z.B. in [Xing u. a. \(2017\)](#) die dazu auch eine größer Artenvielfalt des chinesischen Küstengebiets mit aufnahmen.

Ist der Nährstoffgehalt auf ein dynamisches Niveau umgestellt, ergibt sich der Verlauf nach Abbildung 6.7, der damit das Phosphor an die sterbende Position um die eigene Biomasse stärkt. Der Abstand über a_F verringert sich und trennt sich erst ab Zeitpunkt $t = 80$ in unterschiedliche Verläufe, was ggf. am Wegfall der Fischer liegt, die durch weniger Fangeinnahmen nicht mehr weiter ansteigen.

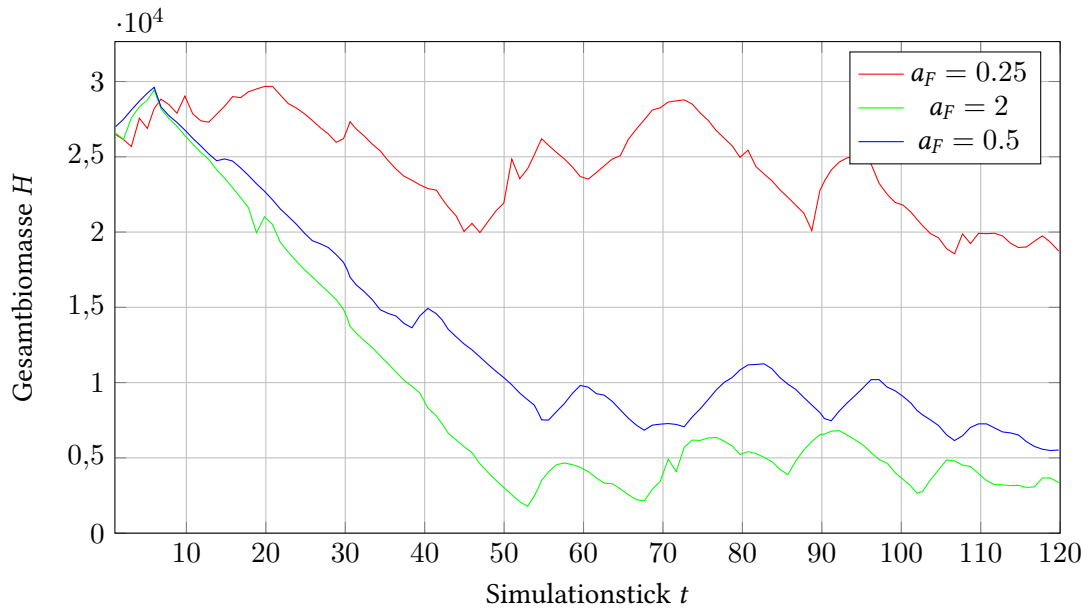


Abbildung 6.6.: Ergebnisplot der Simulation ohne dynamischen Nährstoffgehalt. Parametrisierung: $d_H = 0.15$, $a_F = 0.56$ und $d_F = 3$.

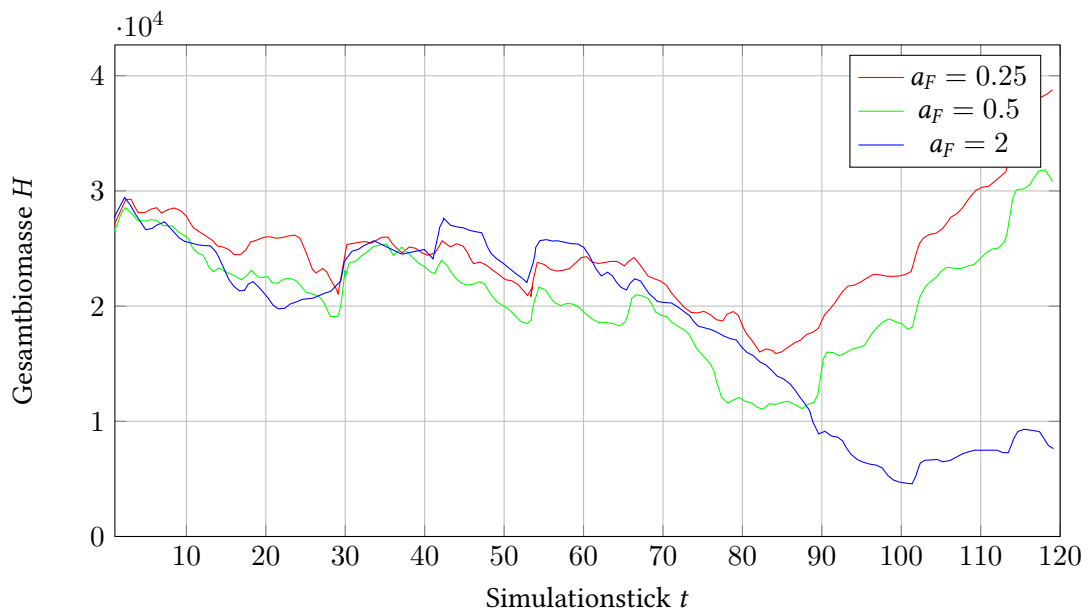


Abbildung 6.7.: Ergebnis-Plots der Simulation mit dynamischem Nährstoffgehalt. Parametrisierung mit: $a_H = 0.25$, $d_H = 0.15$, $a_F = 0.56$ und $d_F = 3$.

6. Evaluation

Der Verlauf ab Zeitpunkt $t = 40$ stellt dar, dass selbst bei halber Fangchance mit aktivem Nährstoffgehalt, die Phytoplankton Förderungen stärkeren Einfluss auf den Verlauf hat, als die eigentliche Fangchance einen Fisch einzusammeln. Dies gilt insbesondere für eine Fangchance von $a_F = 0.5$ die bis $t = 90$ sinkt, ab dort jedoch wieder zunimmt. Abbildung 6.8 zeigt hierzu eine stabile Umgebung mit aktiver Fischerei bei der die unterschiedlichen Effizienzen des Herings a_H die mit $a_F = 0.5$ korrelieren und nicht über den unrealistischen Fall von lediglich $a_H = 0.2$ ihre Beute einfangen, insbesondere da es sich hier um Kleintiere handelt. Abbildung 6.8 zeigt den Verlauf der Masse mit doppeltem Heringsfrass, der vor allem für Parameter $a_F = 1$ hervorsteht. Es ist zu erwarten, dass eine gewisse Verbesserung von 0.55 auf 0.75 existiert, jedoch sich mit optimaler Fangchance die Masse offenbar wieder zurückzieht. Das kann damit begründet werden, dass bei doppeltem Heringsfrass und einer Wahrscheinlichkeit von $a_H \leq 1$ einiges des Phytoplanktons übrig bleibt, dass im Nachhinein wieder zur Vermehrung herangezogen wird, sowohl zur Verstärkung der eigenen Spezies als auch anderer. Der Realität kommt dies sehr nahe, so hat sich nach Xing u. a. (2017) in der Bewertung ebenfalls gezeigt, dass bei optimalen Bedingungen von Jägern deren Beute kaum Erholungsmöglichkeiten bietet.

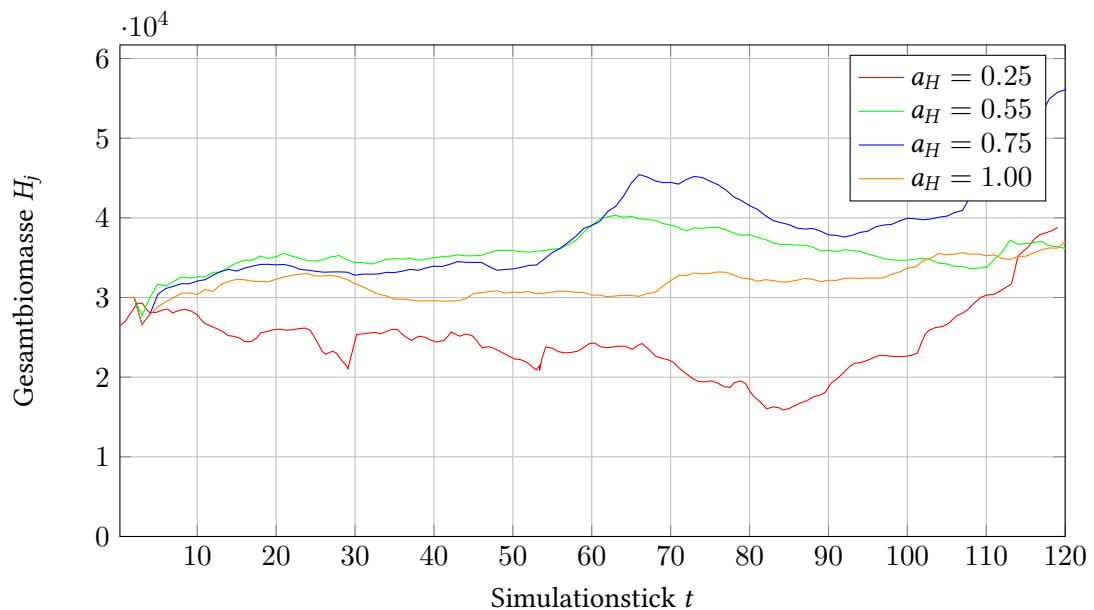


Abbildung 6.8.: Gesamtbioasse der Heringe mit unterschiedlicher Jagdeffizienz. Parametrisierung: $a_H = 0.25$, $d_H = 0.09$, $a_F = 0.56$ und $d_F = 3$.

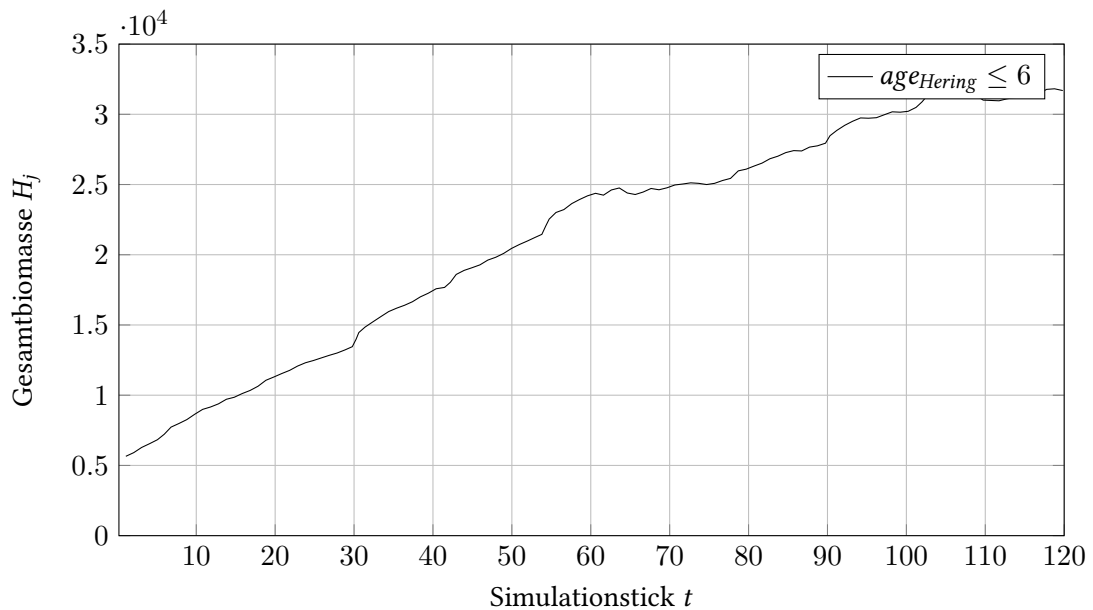


Abbildung 6.9.: Gesamtbiomasse der Jungheringe mit Alter $age_H = \leq 6$ und ohne dynamischem Nährstoffgehalt. Parametrisierung mit: $a_H = 0.2$, $d_H = 0.15$, $a_F = 0.56$ und $d_F = 3$.

Experiment - Phytoplankton Ausbreitung

Die Grundlage für die Verbreitung von Zooplankton und Heringen ist das Phytoplankton. Dessen Ausbreitungsverhalten auf Basis von 6.2 orientiert sich am Vorhandensein des Phosphors sowie der tiefenabhängigen Respiration 6.7. Je tiefer die aktuelle Position desto mehr verteilt sich die Biomasse des Phytoplanktons an der Stelle, und desto eher wird der Bereich eingenommen, der von diesen Tiefen nicht betroffen ist, genauso die Ortschaften an dem ausreichend Phosphor vorrätig ist. Abbildung 6.11 zeigt die Ausbreitung des Phytoplanktons in Zusammenspiel mit langsamen Zooplankton als Fressfeind, genauso wie dem Hering der davon profitiert.

Experiment - Phosphor Beschränkung

Das beschränkte Wachstum durch das vorrätige Phosphor in der Umgebung bildet zusammen mit der verfügbare Fläche die Kapazität des Phytoplanktons. Abbildung 6.12 zeigt das Biomassenwachstum für unterschiedliche Umsetzungsraten r_{ph} .

Die Änderung über r_{ph} entspricht dem erwarteten Verhalten, bei dem durch Zunahme der Umsetzung folglich das Phosphor verstärkt aus der Umgebung gezogen wird, sofern keine

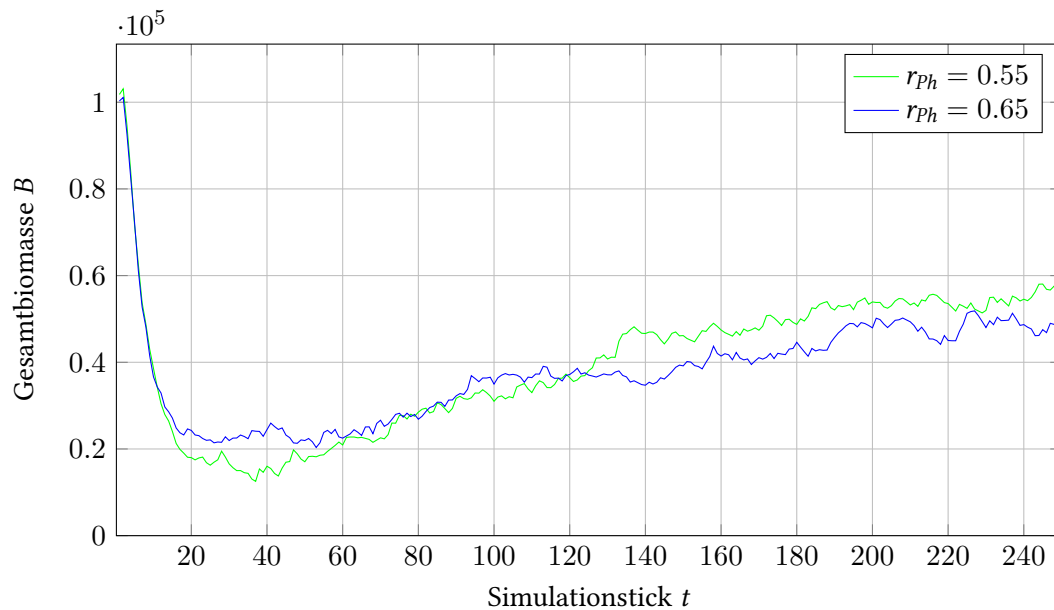


Abbildung 6.10.: Langzeitverlauf der Biomasse zum Phytoplankton über die Zeit mit unterschiedlicher Umwandlungseffizienz. Parameter: $d_{ph} = 0.01$, $r_Z = 0.95$, $d_Z = 0.02$, $r_H = 0.95$, $d_H = 0.09$

neuen Nährstoffe nachgeliefert werden. Interessant ist der Zusammenhang über die Varianz der Respirationsrate d_{ph} , wovon schlüssig die Lebensdauer betroffen ist. Abbildung 6.13

d_{ph} streckt und krümmt den Systemverlauf, der durch den raschen Anstieg über r_{ph} gestartet wird. Interessant dabei ist das konstante Verhalten durch $r_{ph} = 0.55$, das trotz des raschen Anstiegs den Biomassenabgang sehr langandauernd auf einem Niveau hält. Dies wird natürlich durch schiere Größe der Nordsee begünstigt.

Experiment - Stabiles und Instabiles Nahrungsnetz

Gepprüft werden die Parametrisierung um ein stabiles Nahrungsnetz zu erreichen, das die Oszillation gegenüber Abbildung 6.14 aufhebt. Betrachtet wird dazu die Interaktion der Planktonteilchen unter sich, im Zusammenspiel mit dem Heringsfrass.

Die Instabilität äußert sich durch die Oszillation die ab Tick $t = 40$ fortfährt und beim Ausschlagen vom Phytoplankton B_{ph} sofort das Zooplankton B_Z nachzieht. Das Wegfressen durch den Anstieg des Phytoplanktons sorgt neben der ortsabhängigen Respiration von Basis $d_Z = 0.02$ vor allem durch den Heringsfrass für den nachgelagerten Abgang. Der mäßige Anstieg der Heringsmasse zeigt dies ab $t = 90$.

6. Evaluation

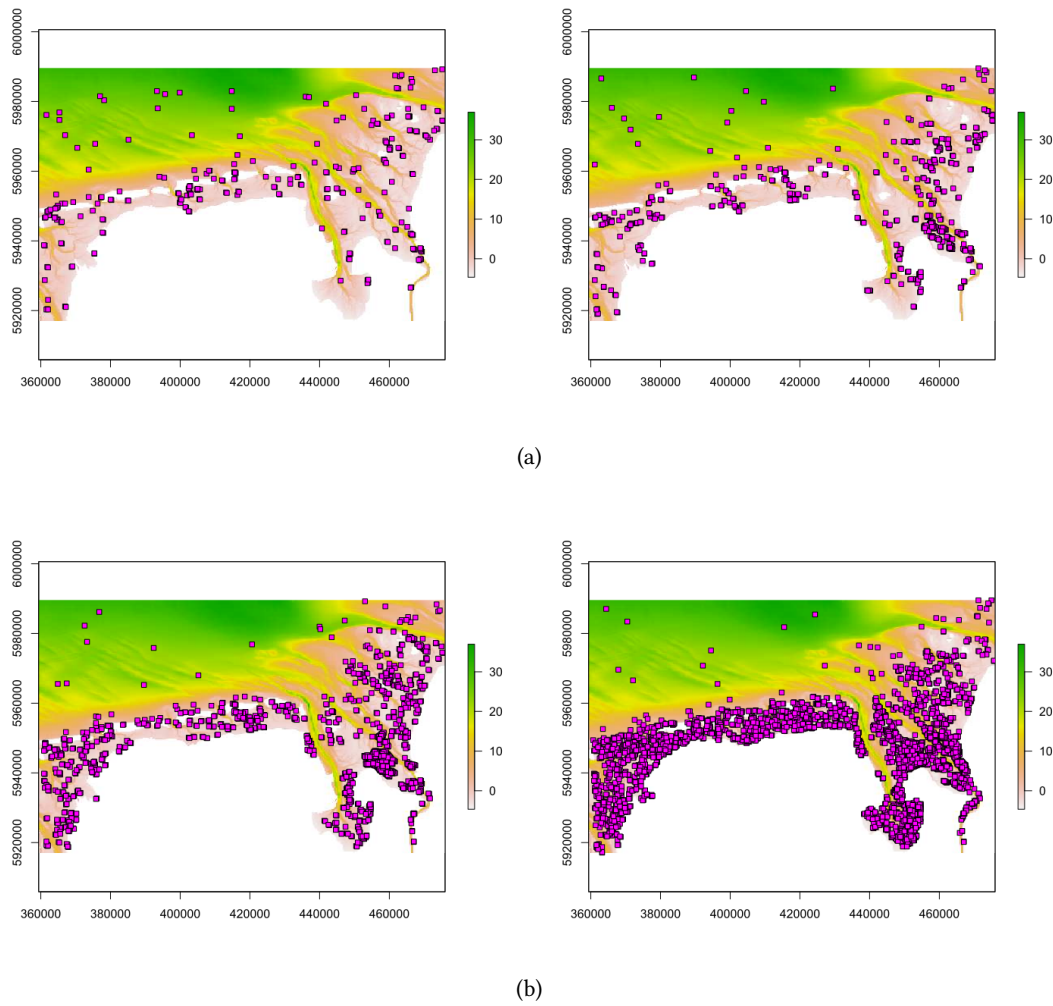


Abbildung 6.11.: Phytoplankton Ausbreitung bei aktiven Heringen und Zooplankton. Parameter: $r_{ph} = 0.65$, $d_{ph} = 0.001$. Oben links $t = 50$, oben rechts $t = 100$, unten links $t = 200$, unten rechts $t = 500$, Geographischer Raum siehe 6.3.7

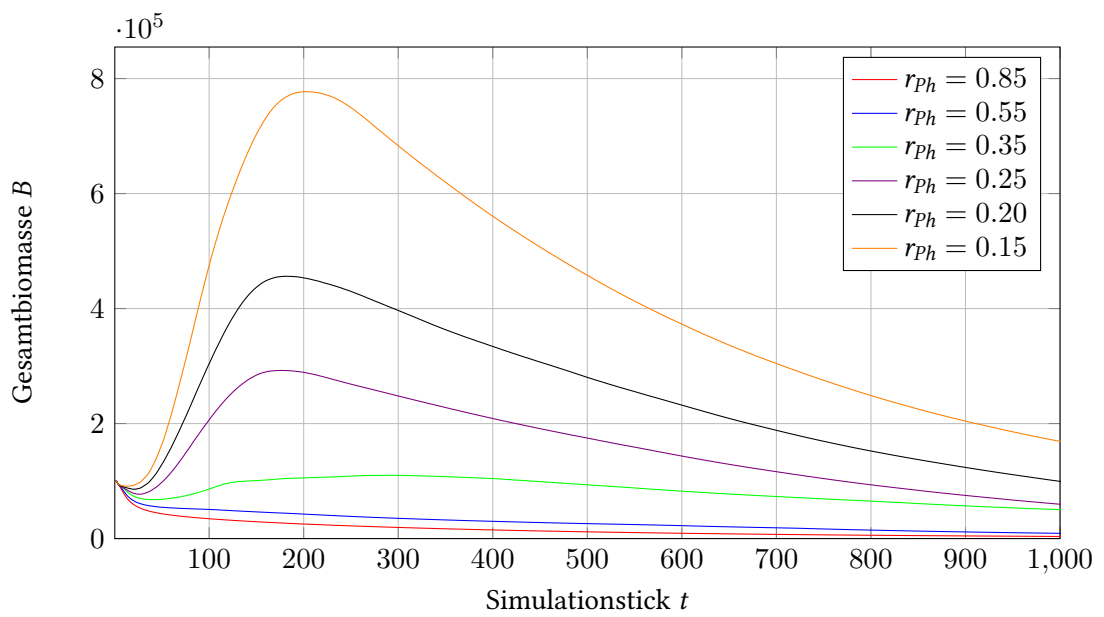


Abbildung 6.12.: Kumulierte Biomasse des Phytoplanktons über die Ausbreitung mit unterschiedlicher Wachstumsparametrisierung. Parameter: $B_{Ph}(t_0) = 10000$, $d_{Ph} = 0.005$

6. Evaluation

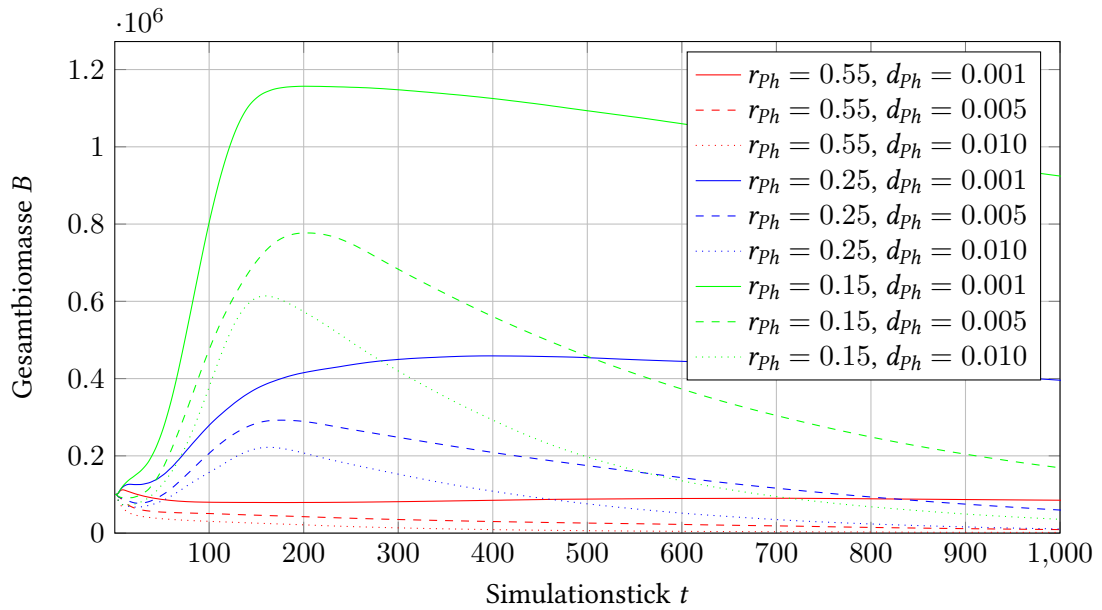


Abbildung 6.13.: Kumulierte Biomasse des Phytoplanktons über die Ausbreitung mit unterschiedlicher Wachstumsparametrisierung. Parameter: $B_{ph}(t_0) = 10000$, $d_{ph} = 0.005$

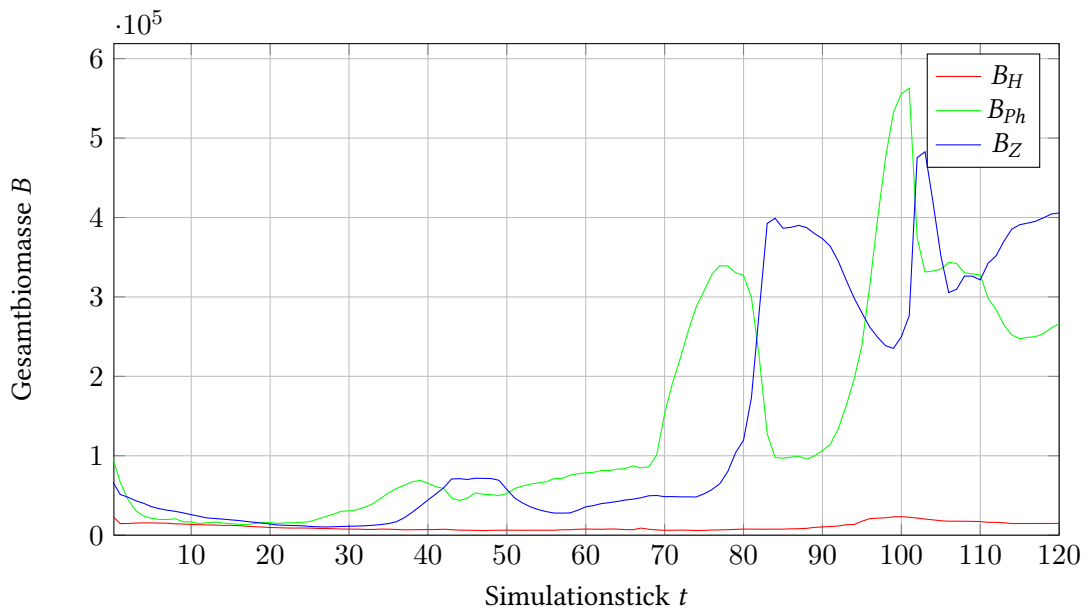


Abbildung 6.14.: Kumulierte Biomasse eines instabilen Nahrungsnetzes. Parameter: $r_{ph} = 0.85$, $d_{ph} = 0.01$, $r_Z = 0.95$, $d_Z = 0.02$, $r_H = 0.75$, $d_H = 0.09$

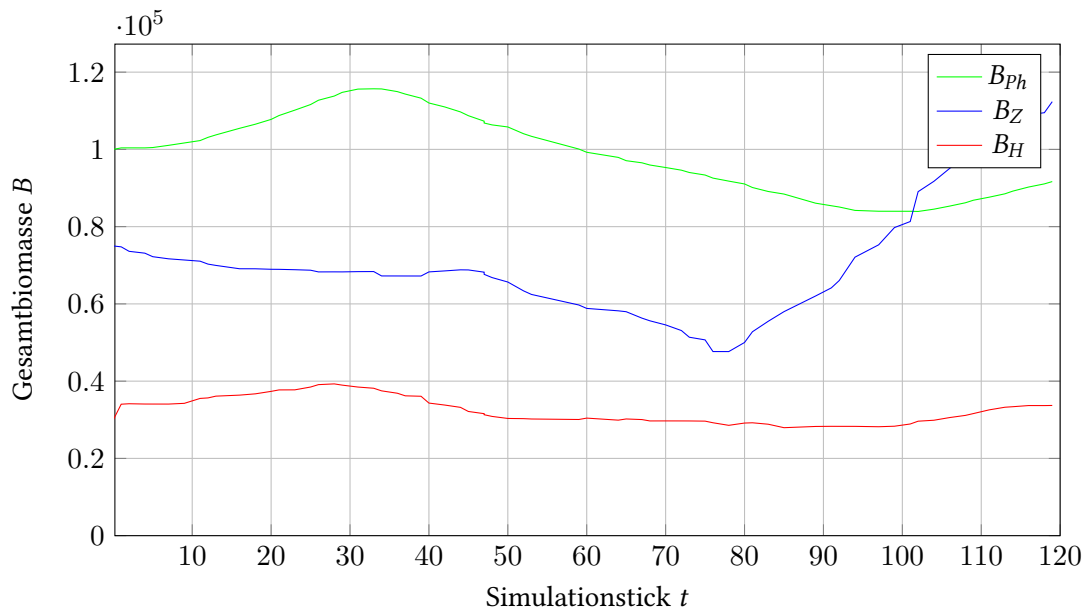


Abbildung 6.15.: Kumulierte Biomasse eines stabilen Nahrungsnetzes. Parameter: $r_{ph} = 0.75$, $d_{ph} = 0.01$, $r_Z = 0.85$, $d_Z = 0.02$, $r_H = 0.85$, $d_H = 0.09$

Abbildung 6.15 zeigt dazu den Verlauf ohne aktive Fischerei, die für die Startmasse von $B_{ph} = 100000$, $B_Z = 750000$, $B_H = 300000$ die Oszillationen entfernt, indem das Phytoplankton gedämpft wurde. Diese Dämpfung wird durch die Verteilung in Abbildung 6.16 deutlich, die sich vor allem in der Clusterbildung des Planktons äußert. Heringe wandern zwischen den Gruppen hin und her, während das Zooplankton sich an der Ausbreitung des Phytoplanktons orientiert.

Die Gruppenbildung verläuft entlang der nährstoffreichen Küstenregionen. Die Passage von Wilhelmshaven und der Weser an der östlichen Seite zeigen weniger ausgeprägte Gruppen die erst an den Kanalein- und Ausgängen, ebenso der Ems auf der östlichen Seite, einen Wachstumsanstieg verzeichnet.

Experiment - Fischpreis Variation

Zur Betrachtung der Fischer spielt der Preis eine bedeutende Rolle. Er reguliert das Einkommen der Fischer über den Verkauf der gemachten Beute, die über die Zeit angesammelt wurde. Dazu existieren zwei Modellansätze. Beim einen agiert der Fischer ähnlich einem Räuber und vereinfacht den Verkauf dahingehend, die eigene Beute sofort auf dem Meer zu verkaufen. Beim anderen muss dieser erst einen Hafen anlaufen, und dann seine Beute verkaufen. Eine Lösung

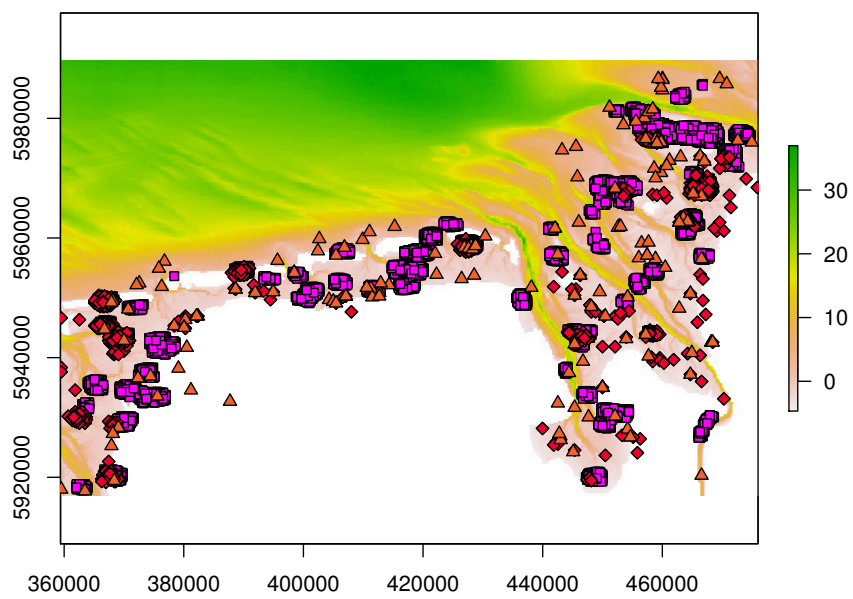


Abbildung 6.16.: Verteilung von Phytoplankton, Zooplankton und Hering mit Clusterbildung zum Zeitpunkt $t = 50$, Geographischer Raum siehe 6.3.7

die gewählt werden würde, sofern noch andere Systemeigenschaften wie ein realistischer Treibstoffverbrauch und die Untersuchung von Profitabilität im Fokus steht (Cooper und Jarre (2017b)). Nach diesem vereinfachendem Schema ergibt sich ein Verlauf nach Abbildung 6.17.

Bei fehlender zeitlichen Verzögerung T wird die erwartete Schwingung beim Heringsanstieg erreicht, bei dem dessen Anstieg direkt zum instabilen Nahrungsnetz aus 6.14 in Beziehung steht.

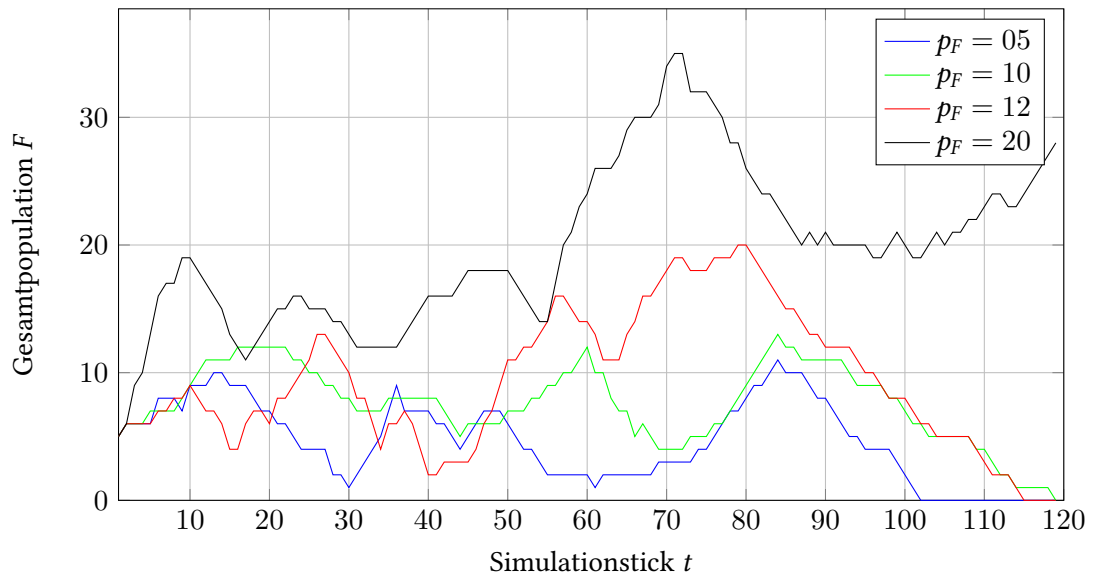


Abbildung 6.17.: Ergebnis-Plots der Simulation ohne dynamischem Nährstoffgehalt. Parametrisierung mit: $a_F = 0.56$, $d_H = 0.15$, und $d_F = 0.12$.

Die Fischerpopulation schwankt in diesem Fall ähnlich der Räuber, durch zwei Faktoren. Zum einen durch die vorrätige Anzahl initialisierter Heringe, zum anderen da sich diese bis zum Zeitschritt $t = 10$ zunächst ebenfalls ausbreiten können - sofern sie genügend Beute und Biomasse gesammelt haben. Zu erkennen ist die stufenweise Ab- und Zunahme der Fischer, damit begründet, dass einzelne Fischerindividuen zunächst einen genügend großen Fang erreichen müssen, und deren Gewinn sich zudem durch den täglich schwankenden Fischpreis ergibt. Bis genügend Gewinn erzielt wurde, vergeht immer eine gewisse Zeit. Interessant ist die fortlaufende Amplitudenentwicklung, im Besonderen um den Zeitpunkt $t = 70$ bei dem für den Fischpreis $p = 20$ ein hoher Erlös für einen raschen Anstieg sorgt und anschließend wieder abfällt. Möglicherweise hängt dies mit dem Verlust der Heringe zusammen, wie bereits in Abbildung 6.8 dargestellt, der in Zusammenhang mit dem eingeschränkten Bewegungsgebiet

des Fischers zusammenhängt, sollte er sich in der Nähe der Küstenregion aufhalten, in dem sich der Fischer selbst nie bewegen darf.

Eine anderer Modellansatz, der dem individuellen Fischereiverhalten sehr nahe kommt ist der beschriebene in Abschnitt 6.3.4. Dieser nutzt die zeitliche Verzögerung, bei dem der Fischer zunächst den Aufwand leisten muss, zurück zu einem naheliegenden Hafen zurückzukehren um seine Beute zu vertreiben. Abbildung 6.18 zeigt den zeitlichen Verlauf für unterschiedliche Fischpreise p_F .

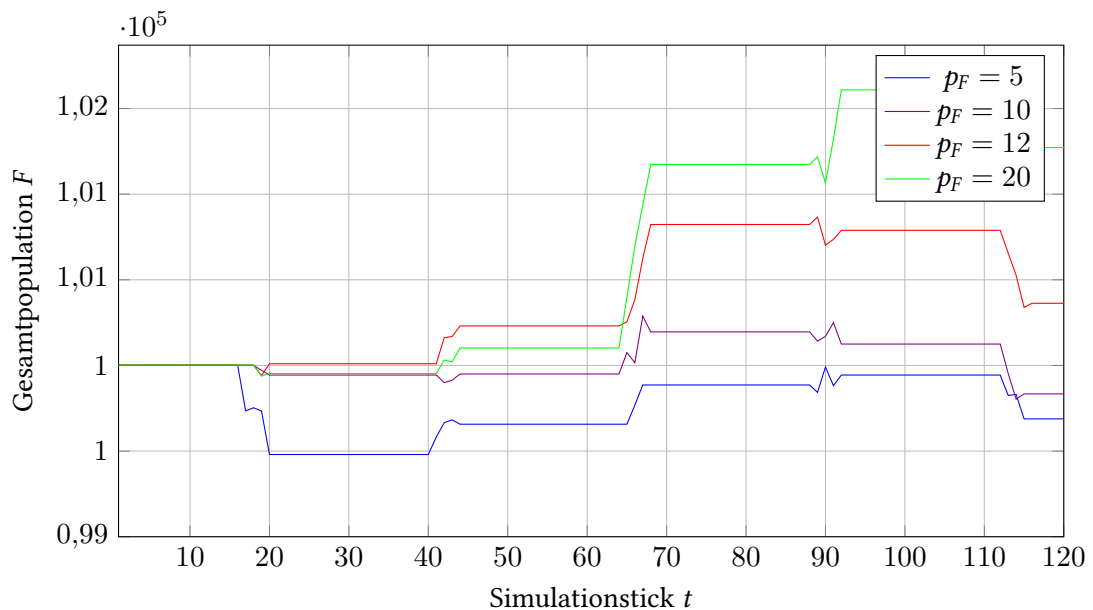


Abbildung 6.18.: Ergebnis-Plots der Simulation ohne dynamischem Nährstoffgehalt. Parametrisierung mit: $a_F = 0.56$, $d_H = 0.15$, und $d_F = 0.12$.

Leider läuft dieses Modell bei einem realistischen Anschaffungspreis von $A_F = 10.000$ darauf hinaus, dass für den betrachteten Zeitraum bis $t = 120$ nie genügend Kapital angesammelt wird um neue Fischer einzustellen. An Stelle dessen wird durch den Zeitverzug deutlich wie schrittweise im Zeitraum von 15 : 00 bis 18 : 00 Uhr eine Gewinnkalkulation stattfindet. Mit $p_F = 5$ reicht dieser zumeist nicht aus um diese kumulierten Tageskosten zu decken. Insbesondere, da auch nicht jeder Fischer rechtzeitig zum Hafen kommt. Abbildung 6.19 zeigt die Verteilung von Fischern, Heringen und dem Zooplankton zum Zeitpunkt $t = 25$ der den Beginn der Distributionsphase wiedergibt.

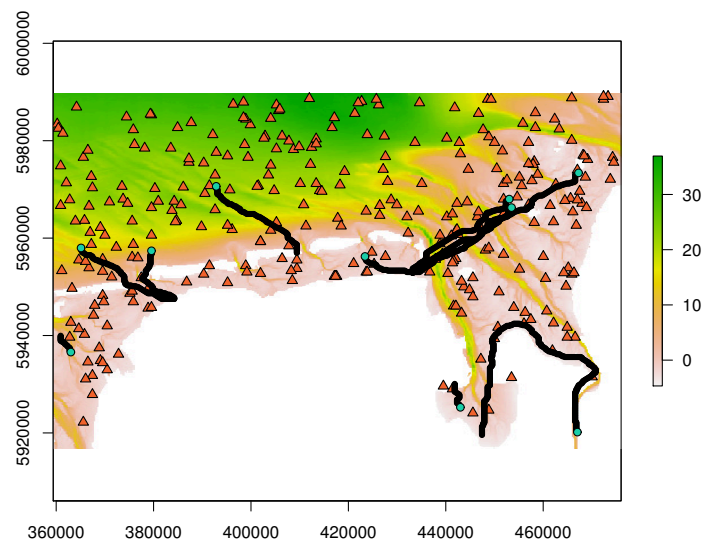


Abbildung 6.19.: Positionierung von Fischern und Heringen in der Umgebung zu einem Zeitpunkt $t = 25$ mit Pfad zum naheliegendsten Hafen, Geographischer Raum siehe [6.3.7](#)

Dieser zeitliche Verzug zeigt sich auch im Plot 6.19 der zum einem Zeitpunkt $t = 25$ die Verteilung der Fischer darstellt, genau während der Distributionsphase. Heringe sind am Anfang noch verteilt und bewegen sich in Richtung der Küste.

6.3.8. Zusammenfassung und Ausblick

Das Fangmodell zeigt illustrativ wie ein MARS DSL Modell aussehen kann und welche Art von Modelltyp unter anderem damit möglich sind. Es beweist die Machbarkeit der verschiedenen Konzepte in einem größeren Rahmen indem ein ein komplexeres Nahrungsnetz mit äußeren Einflüssen erstellt wurde, und mit offenen Fragestellung zur Fischerei und unter Bezugnahme von Daten wie Phosphor und Phytoplankton den Echtweltbezug wiederherstellt. Ein feingra-nulares Agentenmodell wurde erstellt, dass sich Jagdbeziehungen zwischen Individuen stützt und aus einer Reihe formulierter Systemgleichungen um lokations- und zeitbedingtes Handeln verfeinert wurde. Die Resultate haben gezeigt, dass wünschenswerte Wechselwirkungen zwischen den Fischern und den Heringen stattfinden und der system-typische Welleneffekt für Räuber-Beute System auftritt, insbesondere wenn Parameter leichte Schwankungen in Richtung einer ausbreitenden Spezies aufweisen $Sterberate < Geburtenrate$. Der Fischer als externe Einflussgröße machte deutlich dass trotz eines schwankenden Preises nur eine stufenweise Verbesserung möglich ist. Der Jagderfolg orientiert sich an der Änderung der Jagdeffizienz und Preisschwankungen zum Verkauf können einen massive Auswirkungen auf Fischerei haben, die damit die Heringspopulation bis zum völligen Wegfall aus dem System führen kann. In der Folge schwankt stetig nur das Zooplankton und Phytoplankton durch den Wegfall des Hering.

Als zukünftige Modellverfeinerung ist die Integration von gesonderten Reaktionsgleichung zur realistischeren Verarbeitung von Phosphor, genauso wie weitere wichtiger Elemente wie Nitrit, Phosphate, dem Salzgehalt und der Kombination daraus mit vorkommendem Sauerstoff. Die Verarbeitung findet in diesem Modell sehr trivial statt und es ist offen ob diese Besonderheiten einen beschleunigenden oder hemmenden Effekt auf das Algenwachstum hat. Weitere Verfeinerung betrifft die Einführung eines Angebot und Nachfragemodells für den Fischverkauf, der den sinus-schwankenden Fischpreis exakter reguliert. Alles mit der ausgehenden Frage wann eine Stabilisierung des Systems erreicht wird.

6.4. Vergleich direktes und transformiertes Modell

6.4.1. Performance Vergleich

Um ein aussagekräftiges Ergebnis der resultierenden Ausführungsschnelligkeit aus der Transformation zum Zielsystem in Abschnitt 4.4 zu erhalten, wurde hierzu das bekannte Räuber-Beute Modell (Glake u. a. (2017)) zur Interaktion zwischen Wölfen, Schafen, und Gräsern modelliert. Dies stellt eine exakte Nachbildung des im MARS-Umfeld verwendeten Modells dar und macht den Vergleich eines mit der MARS DSL entwickelten Modells und eines gegen MARS-Framework entwickelten Modells deutlich. Dazu liegt eine gemittelte Messung pro Zeitschritt bis $t = 200$ von 4 Durchläufen, sowohl mit dem direkt implementierten Modell als auch dem transformierten. Zusätzlich werden die Modellergebnisse zum Vergleich herangezogen um Aussagen bezüglich ähnlicher oder nicht ähnlicher Dynamiken zu treffen. Sie umfassen die jeweils kumulierte Anzahl noch existierender Agenten eines Typs in der Welt, zu jedem fünften Zeitschritt. Parametrisiert wurden beide Modellvarianten mit der Anzahl von Instanz $I_{Wolve} = 10$, $I_{Sheep} = 100$ und Grass $I_{Grass} = 1000$. Da in der MARS DSL noch keine aktiven Layer (Hüning u. a. (2016)) möglich, d.h. der Layer selbst kann aktiv werden und für diese Beispiel das Grass neu wachsen lassen, wurde zur Erfüllung dieser Anforderung ein besondere *Observer* eingefügt, der sich ähnlich dem NetLogo Wächter (Wilensky (2015)) verhält und global Zustände beobachtet, überwacht und ggf. Veränderungen vornimmt. Dazu wurde aus dem $MARS_{direkt}$ Modell die gleiche Position (0, 0) gewählt, von wo aus neue globale Aktionen durchgeführt werden.

Abbildung 6.20 zeigt hierzu die Laufzeit jedes Ticks. Das Zielmodell umfasst dabei noch keinerlei Optimierungen zur Verbesserung von Explorationen. Zu erkennen ist der anfänglich notwendige Aufwand zur Initialisierung der Agenten. Das Erzeugen und Einfügen der in $MARS_{transformiert}$ existierenden Agenten in das globale Environment verhält sich fast identisch. Anstelle der Synchronisation ggf. mehrere einzelner Umgebungen fällt dieser Aufwand jedoch an diesem Punkt weg, was vermutlich die minimale Verbesserung begründet. Zur eigentliche Simulationslaufzeit ist zu erkennen dass offenbar ein Großteil Tick Laufzeit im Zeitbereich von 1000 bis 1500 liegt. Die Explorationen von anderen Agenten stellt eines der teuersten Operationen dar, dessen Anwendung möglichst gering ausfallen sollte. In der Übersetzung wird dazu auf zuvor identifizierte Agenten zurückgegriffen bzw. bei Abfrage aller Individuen die globale Agentenmenge des gewählten Typs verwendet. Wichtiger Optimierung stellt zudem der inhärente Einsatz des Method Inlineing dar der durch automatische Entfaltung der Ausdrücke in komplexe Aufrufe einhergeht. Äquivalenter Aktionscode wird mehrfach erzeugt, weswegen Referenzen zu anderen Methoden mit kostspieligen Kopiervorgänge weniger notwendig wird.

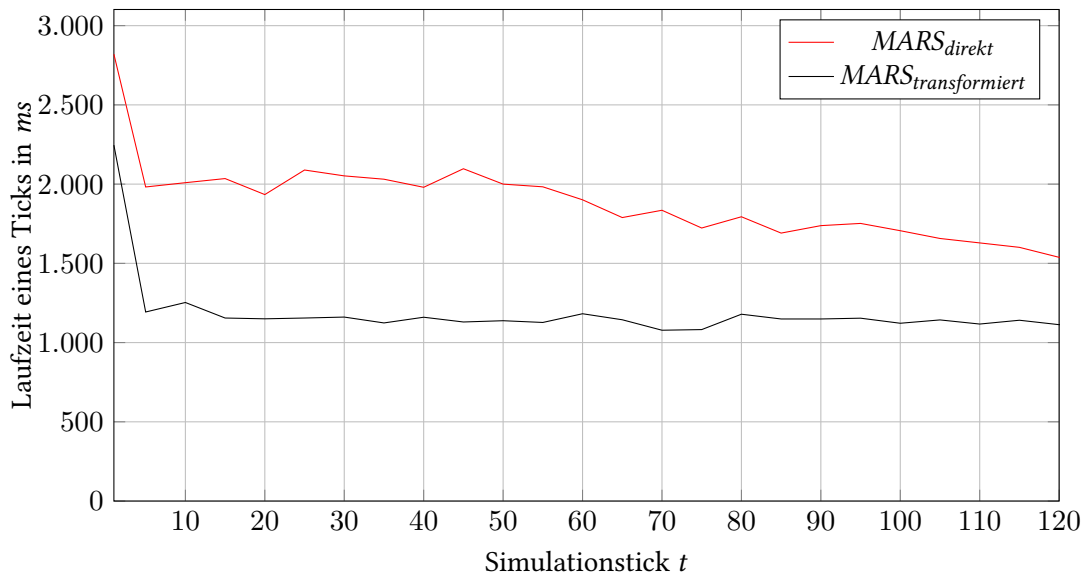


Abbildung 6.20.: Laufzeitmessung alle 5 Ticks zum Vergleich eines direkten Modells gegenüber eines transformiertem Modells durch die MARS DSL

6.4.2. Dynamik Vergleich

Da die Laufzeitverbesserung auf Grundlage eines falschen Verhaltens entstehen kann wurde zuzüglich der Messung die Dynamiken beider erstellten Modell gegenübergestellt. Essenzielle Zustandsgrößen stellen die Populationen der *Schafe*, *Wölfe* und der *Gräser* dar. Alle drei Dynamikverläufe sollten dem direkten Modell entsprechen. Nachfolgend werden hierzu die Verläufe der Populationen durch Vergleich der Schafe zu den Gräsern 6.21 und zusätzlich der Wölfe zur Anzahl der Schafe 6.22 abgebildet. Immer gegenübergestellt mit den Ergebnissen aus dem direkten Modell.

Der Vergleich beider Ergebnisse zur Populationsdynamik in Abbildung 6.21 und 6.22 zeigt die Erhaltung der Systemdynamik des übersetzten Modells. Die geringfügige Abweichung in der *Grass* und *Schaf* Population kann durch den Wegfall wiederholter Ausführung begründet werden und der zufällig gewählten Positionen. Das Setzen von Wölfen und Schafen in unmittelbarer Nähe erlaubt die schnellere Reduzierung der Zielbeute. Das gilt ebenfalls für das *Grass*. Mehrfache Wiederholungen würden diese Abweichung relativieren.

Letztendlich stellt der Dynamikvergleich fest dass die bekannte und anerkannte Räuber-Beute Systematik mit der MARS DSL abbilden lässt.

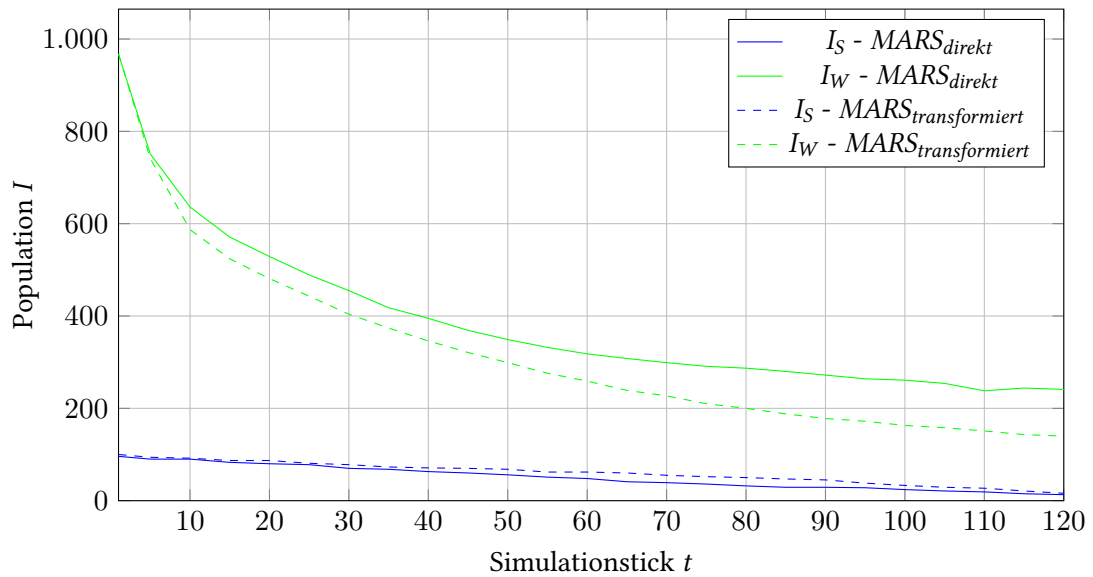


Abbildung 6.21.: Anzahl noch lebender Schafe I_S zu Gräsern I_G über die Zeit zum Vergleich des direkten Modells gegenüber eines transformiertem Modells durch die MARS DSL

6.4.3. Quantifizierter Vergleich

Analog zur Einzelbetrachtung der Sprache 6.1.2 wird im Folgenden nun im Zusammenspiel mit einem zuvor existierendem Modell die Eigenschaften des Kompilats mit dem Modell und seiner direkten Implementierung gegenübergestellt.

Artefakte und Projekte

Erster Anhaltspunkt ist der Unterschied erzeugter Artefakte und Projektelemente des *direkten* Wölfe Schafe Modells zur DSL Lösung. Tabelle 6.9 zeigt die Anzahl erzeugter Artefakte sowohl für das *direkte* als auch transformierter Modell, der im Zuge dessen nochmals eingeteilt wird in das vollständige Modell *vollständige* und reine *Agenten*-Modell. Erstes umfasst das Gesamtsystem, dass zur Ausführung in die MARS-Cloud hochgeladen wird. Letzteres stellt hauptsächlich die erzeugte *Agentenlogik* dar, die durch las Eigenleistung des Modellierer bei der Agentendefinition anfällt und gleichbedeutend erzeugt wird.

Für das Fangmodell gilt analog Tabelle 6.10. Sie vergleicht die erzeugten Artefakte gegenüber dem reinen DSL-Fangmodell. Zu sehen ist, dass die Anzahl erzeugter Operationen

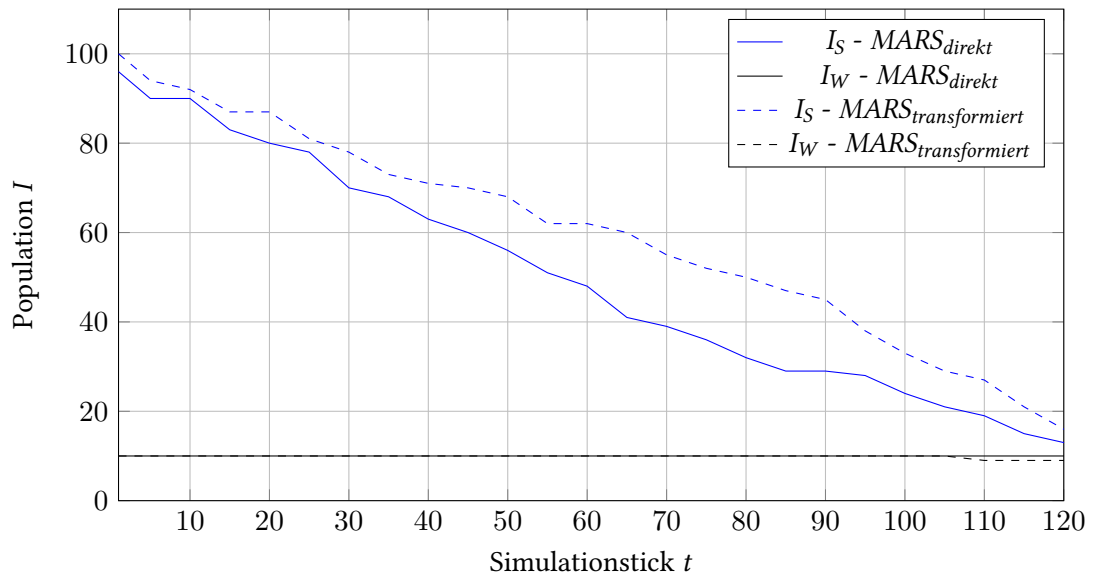


Abbildung 6.22.: Anzahl noch lebender Wölfe I_W und Schafe I_S über die Zeit zum Vergleich des direkten Modells gegenüber eines transformiertem Modells durch die MARS DSL

sehr schwankt die reine Beschreibung einer Entität alleine ausreicht, jegliche technischen Maßnahmen durch die DSL zu verstecken, der zur Lauffähigkeit gebraucht wird.

Die durch den MDD Ansatz erzeugten Elemente zeigen dass bei den *Projekten* eine ähnliche Menge herrscht, die aus der MARS Bedingung hervorgeht bei der für jeden Layer ein einzelnes Projekt vorgehalten werden muss, wie aus Abschnitt 4.4.1. Während das formulierte DSL-Modell lediglich ein Projekt zur Verwaltung der Dateien vorsieht wird zu jedem definierten Layer ein respektives Projekt aufgesetzt. Mit dem Layer assoziierte Agenten werden dagegen auf zwei *Quelldateien* aufgeteilt. Die reine Agentenbeschreibung inklusive definierter Ausdrücke, States, Actions etc. und die Layerverwaltung, die jedwede Systeminitialisierung und *Resourcenbindung* - GIS Dateien und Zeitreihen - verwaltet. Näheres dazu in Abschnitt 5.4.

Betrachtet man allein die *Agentenlogik*, in der ein Großteil der *Eigenentwicklung* des Modellierer enthalten ist, zeigt sich das der Aufwand der technischen Implementierung zur Lauffähigkeit viele Elemente erfordert, insbesondere um eine Vielzahl möglicher Modelle darstellen zu können. Die zusätzliche Anzahl von 21 für das vollständige Modell gibt Aufschluss über alle notwendigen Erweiterungen die sowohl durch die vorhandenen Konnektoren der MARS DSL Standardbibliothek begründet wird und zum anderen zusätzliche anonyme Typen

6. Evaluation

Metrik	$MARS_{direkt}$	$MARS_{transformiert}$ vollständig	$MARS_{transformiert}$ Agentenlogik	$MARS_{WölfeSchafe}$ DSL-Modell
Projekte	2	3	1	1
Quelldateien	7	5	5	1
Namensräume	3	6	1	1
Typen	11	27	6	6
Operationen	81	420	120	17
Felder	16	65	20	19

Tabelle 6.9.: Vergleich erzeugter Elemente des Räuber-Beute Modells gegenüber der direkten Implementierung

Metrik	$MARS_{Fangmodell}$ DSL-Modell	$MARS_{transformiert}$ vollständig	$MARS_{transformiert}$ Agentenlogik
Projekte	1	7	1
Quelldateien	4	10	1
Namensräume	1	9	1
Typen	8	33	12
Operationen	39	597	273
Felder	35	57	41

Tabelle 6.10.: Vergleich erzeugter Elemente des Fangmodells gegenüber der direkten Implementierung

6. Evaluation

Metrik	$MARS_{direkt}$	$MARS_{transformiert}$ vollständig	$MARS_{transformiert}$ Agentenlogik	$MARS_{Fangmodell}$
Lines of Code (LoC)	235	1335	465	1714
Ø Zyklomatische Methodenkomplexität	1.39	2.06	3.582	1.88
max Zyklomatische Methodenkomplexität (CC)	9	40	40	40
Ø Relationale Kohäsion	2.45	1.79	3.17	1.4
Ø Methoden Rank	2.45	1.79	3.17	1.4
Lack of Cohesion (LCOM)	0.48	0.3	0	0.48
Lack of Cohesion Henderson-Seller (LCOM-HS)	0.57	0.36	0	0.54

Tabelle 6.11.: Vergleich der Komplexität und Kopplung von erzeugten Elementen des Räuber-Beute Modells gegenüber der direkten Implementierung sowie dem Fangmodell

um lediglich externe Parameter empfangen zu können. So ist die Grid Feld Dimensionierung eine Stellvertreterlösung die eine solchen Zusatztypen darstellt [Glake \(2017\)](#). Sicherlich ist ein Teil davon inaktive, der durch die nachgelagerte *JIT-Kompilierung* jedoch entfällt [5.1](#).

Komplexität und Bindung

Zur Betrachtung von Komplexität und Kopplungsgrad der erzeugten Artefakte wurde auf eine Reihe gültiger Software Metriken zurückgegriffen. Details zu den Metriken finden sich unter anderem in [Starke \(2015\)](#). Sie umfassen die Metriken zur Komplexitätsbestimmung innerhalb von Methoden sowie zum Kopplungsgrad dieser in Bezug um Gesamtsystem. Die Anzahl der Codezeilen (LoC) steht den beiden zwar gegenüber, ist jedoch eine gute Überblick der tatsächlichen Methodenkörper.

Tabelle [6.11](#) zeigt die Komplexität der Methoden und gibt an, dass sich CC zunächst an die Definition der Methodenkörper und der Workflows richtet. So ist die Ø CC beim vollständigen System mit 2.06 gering genug um dennoch ein Verständnis dafür aufzubauen. Zudem geben, nach LCOM 0.3 und verfeinernd die LCOM-HS 0.36, eine geringe Kopplung im Durchschnitt an, der er in Zusammenhang mit der dennoch geringen Komplexität auch Nacharbeiten am generierten System durchführbar sind. Dem gegenübersteht steht allenfalls Einzelfälle mit

6. Evaluation

Metrik	$MARS_{direkt}$	$MARS_{transformiert}$ vollständig	$MARS_{transformiert}$ Agentenlogik	$MARS_{Fangmodell}$
Instabilität	1	0.95	1	0.81
Abstraktion	0.36	0.21	0.17	0.36
Normalisierter Abstand	0.36	0.235	0.17	0.17

Tabelle 6.12.: Vergleich der Abstraktion und Instabilität mit normalisierter Abweichung von erzeugten Elementen des Räuber-Beute Modells gegenüber der direkten Implementierung sowie dem Fangmodell

einer CC von 40 die sich durch eine hohe Verzweigung ausweisen, was jedoch aus Standard Erzeugung folgt, daran zu erkennen, dass diese Maximum für jedwede Erzeugung gilt.

Mit diesem Ergebnis lässt sich nun überlegen ob das Kompilat dennoch für Weiterentwicklung herangezogen werden kann. Die Antwort darauf ist vielschichtig und Arbeiten am erzeugten System sollten, auch bei diesen tendenzielle positiven Ergebnissen nur durchgeführt werden, sollte man ein Verständnis für die MARS Umgebung und deren technische Details vorweisen können und man nicht gewillt ist über die DSL weiterzuarbeiten. Sind stattdessen Ergänzung vorgesehen die nicht in das eigentliche Erzeugungsmodell eingreift sondern als zusätzliche Ressource z.B. auf die Typstruktur des Agentenmodells Bezug nimmt oder gar ganzen Namensräumen liegt mit der Lösung der inkrementellen Erzeugung aus Abschnitt 5.6.2 eine Basis vor auf der MDD typisch u.a. das Datenmodell aufgebaut werden kann [Mohagheghi u. a. \(2013\)](#).

Abstraktion und Instabilität

Modelle stellen immer eine Abstraktion von einer betrachteten Wirklichkeit dar und sind im dem Zuge dessen immer mit der Anforderung belastet, resilient genug zu sein auf technologische Änderungen zu reagieren und nur dann ein anderes Verhalten aufzuweisen, sofern sich die zugrundeliegenden Anforderungen ändern. Aus diesem Grund wird die *Instabilität* und *Abstraktion* der erzeugten Systeme betrachtet. Aus Sicht des Modellierers ist diese Metrik für das Erzeugungssystem zunächst irrelevant, ist jedoch zur Betrachtung der Erhaltung des Abstraktionsniveaus und der Stabilität des Erzeugungssystems von Wert und damit für Zwecke der Weiterentwicklung. Zur Prüfung der Abweichung wird zu diesem Zweck der normalisierte Abstand von *Instabilität* gegenüber *Abstraktion* genutzt. Die Tabelle 6.12 zeigt hierzu die berechneten Werte dieser Metriken.

Die Instabilität von 0.95 und 1 sowohl für das vollständige System als auch für die Agentenlogik weist sehr instabile Komponenten auf, die jedoch darauf zurückgeführt wird, dass bei diesem kleineren Modell keine Aufteilung der erzeugten Assemblies erfolgt. Mit dem Abstraktionsgrad von 0.21 und 0.17 zeigt sich auf der anderen Seite dagegen eine sehr abstrakte Komponente. Der senkrechte Abstand zur Ideallinie liegt im Bereich 0 bis 1. Mit 0 wird damit das System gesehen, das auf eine hohe Abstraktion setzt und dazu noch stabil genug ist. Alle Systeme neigen gegen 0 und weisen damit eine höhere Abstraktion und dennoch Stabilität auf, was tendenziell einen guten Wert darstellt, auch einen besseren Wert im Vergleich zur direkten Realisierung *MARS_{direkt}*.

7. Zusammenfassung und Ausblick

Die agentenbasierte Softwareentwicklung ist ein effektiver Ansatz zur Behandlung komplexer Problemstellungen mittels autonomen und intelligenten Softwareentitäten. Aufgrund wachsender Akzeptanz von Agentensystemen wird der Forschungsschwerpunkt auf dem Gebiet der agentenbasierten Entwicklung vor allem auf die Erforschung von geeignetem Entwicklungswerkzeugen gesetzt.

Aktueller Stand der Forschung ist dabei das Agentendesign mittels einer Agentenmethodologie zu spezifizieren und die resultierenden Artefakte als Grundlage zur manuellen Programmierung zu verwenden. Zu diesem Zwecke werden dann spezifische Modellierungssprachen oder Standardprogrammiersprachen wie C# oder Java genutzt. Zwei Hauptprobleme lassen sich bei diesem Ansatz identifizieren: Zum Einen führen die Fehler, die bei der manuellen Überführung von Design in eine konkrete Implementierung entstehen dazu, dass insbesondere das abstrakte Design weniger nützlich, hinsichtlich auf Nachhaltigkeit der entwickelten Softwareapplikation, wird. Zum Anderen existiert keine universelle Agentenmethodologie die ausdrucksstark genug ist, um alle Problemstellungen bei denen Agententechnologie zum Einsatz kommt, adäquat zu beschreiben.

Aus diesem Grund wurde eine Sprache, zur Modellierung komplexer agentenbasierte Modelle, namens MARS DSL. Sie erlaubt die Formulierung von Agentensystemen auf einer plattformunabhängigen und textuellen Art und Weise, die auf der sprach-getriebenen Entwicklungsinitiative basiert. Die Sprache umfasst eine abstrakte Syntax die das Vokabular definiert, eine konkrete Syntax zur Darstellung inklusive partieller und formaler Invarianten die der Sprache eine Bedeutung gibt. Die unabhängige Beschreibung erlaubt die Formulierung von Agenten und Layern in Verbindung zu spatialen und temporalen Daten. Ein generischer Übersetzer in der MARS DSL erzeugt ein konformes und ausführbares MARS-Modell durch Modelltransformationsschritte. Basierend auf Erzeugungsvorlagen wird das entworfene MAS in ausführbaren Code umgewandelt, der optional mit manuell geschriebenem Code zusammengeführt werden kann. Die MARS Plattform bietet hierzu das Rahmenwerk für erzeugte System, in dessen Metamodell die Transformation stattfindet. Zum Design-Zeitpunkt wird diese Übersetzung vorausgehend durch eine Modellierungsumgebung unterstützt, dessen Funktionalität die platt-

formunabhängige Spezifikation von MAS, die Modellvalidierung, die Modelltransformation und das nachträgliche Packen und Vorbereiten der Modellartefakte umfasst. Die Merkmale der Umgebung umfassen die Verringerung der Komplexität durch Trennung in Diagramme, einem Projektoutline, die Modellvalidierung durch Integration der von Z beschriebenen statischen Semantik, wiederverwendbare Komponenten und Erweiterbarkeit. Praktisch evaluiert wurde diese Sprache anhand eines Workshops in Südafrika für Interessierte im Bereich ABMS und durch die Modellierung eines Fischereimodells als Machbarkeitsnachweis. Die Auswertung einer Umfrage zum Workshop ergab eine hohe Ausdrucksstärke und Wiederverwendbarkeit der DSL, mit Abstrichen im Bereich zur einfachen Erlernbarkeit dieser mächtigen Mittel, die u.a. durch eine verminderte Benutzeroberfläche als auch das fehlende Verständnis rund um die ausgehenden Agentenkonzepte selbst einhergegangen ist. Mit dem Fangmodell wurde dagegen ein komplexeres Modell konstruiert, das auf die Fischerei in der Nordsee Bezug nahm und dazu ein Nahrungsnetz mit vorhandenen Interaktionen zwischen Phytoplankton, Zooplankton und Heringen abbildet. Ein auf die Heringspopulation geltender Einfluss durch die Fischer hat wünschenswerte Dynamiken und Oszillationen gezeigt, die große Ähnlichkeiten zu bisherigen Arbeiten aufwies. Gleiches gilt für die Art der Ausbreitung des Phytoplanktons und Zooplanktons an der Nordseeküste. Dieses Ergebnis hat neben dem Vergleich eines direkt implementierten Modells gegenüber einer übersetzten Form aus der MARS DSL aufgezeigt, dass trotz der Sprachabstraktion ein modelliertes Systemverhalten darin wiedererkennbar ist und nachgebildet werden kann. Der reduzierte technische Aufwand ist dabei soweit zurückgedreht, wie es das MARS DSL Metamodell in seiner Beschreibung vorsieht. Metrisch wurde belegt, dass das erzeugte System, obgleich der großen Menge erzeugter Elemente, immer noch eine handhabbare Struktur, mit geringer Kopplung, Komplexität und hoher Kohäsion aufweist, um damit auch manuelle Anpassungen ergänzen zu können.

Als Ausblick ist die Verbesserung der Transformation zu nennen, die auf die aktuellen Performanceprobleme in der Explorationssemantik Bezug nimmt, und unter Verwendung bereits identifizierter Modelleigenschaften, über die statische Semantik, eine passende technische Umgebung für die jeweiligen Agenten finden soll, wie zum Beispiel ein Kd-Tree für statische Agenten. Diese modellabhängige Optimierung eröffnet ein großes weiteres Arbeitsfeld aus dem Compilerbau, um die Leistung sowohl aus Speicher- als auch aus Laufzeiteffizienzgründen zu erhöhen. Mit den zur DSL entwickelten Umgebungen und realisierten Datenstrukturen - Kartesisch, og. Kd-tree, Quad-Tree, Grid - liegt damit bereits die Basis vor, die in der Transformation eingesetzt werden kann. Aus Benutzersicht wird die Sprache, ausgehend der Rückmeldung des Südafrika Workshops und weiterer zukünftiger Anwenderrückmeldungen, weiterentwickelt und vor allem auf einen verbesserten Einstieg gesetzt. Dieser sollte sich in der Sprachver-

wendung zeigen, um die aktuell mäßige Erlernbarkeit zu vereinfachen und die vorliegenden Beschreibungsmittel dem Anwender näher zu bringen. Vor allem für die umgebende Werkzeugunterstützung der Sprache kann hier auf bewährte UI-Techniken zurückgegriffen werden, als auch auf die Hervorhebung von domänenspezifischen Optimierungen und Vorabtransformation zur Designzeit. Schlussendlich gilt es weitere Modelle mit der MARS DSL zu entwickeln, die von der individuen basierten Modellierung profitieren und vor allem als Referenz für neue Anwender dienen, genauso wie zur Prüfung der Korrektheit des Zielsystems mit der aktuellen Spezifikation zu C# und der MARS Zielplattform. Die MARS DSL bietet damit ein wesentlich leichtgewichtiges Werkzeug zur Formulierung agentenbasierter Modelle ohne den technischen Hintergrund von MARS zu kennen. Größere Modelle, wie das hier betrachtete Fangmodell sind im Bereich des möglichen, jedoch für komplexe Berechnungen mit der MARS DSL eher unzureichend, wenn es darum geht eine große Masse mit komplexer Semantik zu simulieren, sondern eher Größenordnungen im Bereich von NetLogo zu formulieren, dessen Dimensionierung diese Sprache in erster Linie versucht hat einzufangen.

Anhang

A. Musterkatalog

Anbei wird ein Musterkatalog vorgestellt, der aus der Entwicklung sprechende Modellmuster für die MARS DSL beinhaltet und dabei einen Ausschnitt der Referenzmuster aus [Aschermann u. a. \(2016a\)](#) wiedergibt. Darunter sind Interaktionsmuster für uni- und bidirektionale Kommunikation, ein Muster zum Agentenmorphismus sowie Weiterleitung von Nachrichten und eine Variante des Kommandomusters.

A.1. Einkanal unidirektionale Interaktion

Problem Ein Agent X sendet eine Nachricht an einen *einzelnen* Empfänger Y , der logisch aus der Umgebung wahrgenommen werden kann *ohne* Antwort.

Realisierung in [A.1](#)

A.2. Einkanal bidirektionale Interaktion

Problem Ein Agent X sendet eine Nachricht an einen *einzelnen* Empfänger Y , der logisch aus der Umgebung wahrgenommen werden können und eine *mit* Antwort senden.

Realisierung in [A.2](#)

A.3. Mehrkanal unidirektionale Nachricht

Problem Ein Agent X sendet eine Nachricht an *mehrere* andere Agenten Y_1, \dots, Y_n , die alle identisch oder logisch miteinander verwandt sein können und aus der Umgebung wahrnehmbar sind, ohne Antwort vom Empfänger.

Realisierung in [A.3](#)

```
Sender.mars
model SenderReceiverModel
layer SenderReceiverLayer
agent Sender on SenderReceiverLayer {
    var ReceiverAgent : Receiver
    var Antwort : string
    active Sende()
        => ReceiverAgent.Empfange("Coffee time?", me)
    passive Antwort(answer : string)
        => Antwort = answer
    reflex { ReceiverAgent := nearest Receiver; Sende() }
}

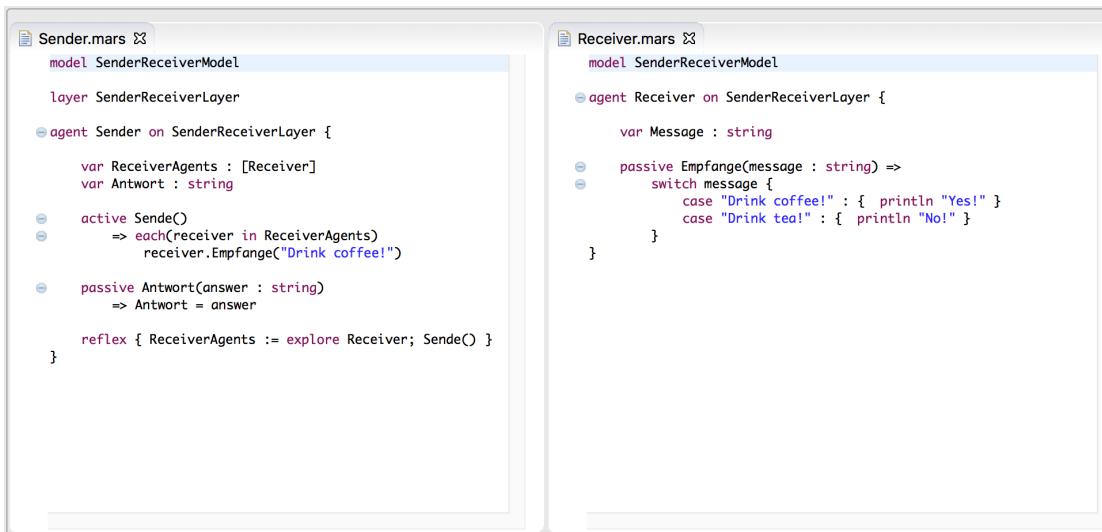
Receiver.mars
model SenderReceiverModel
agent Receiver on SenderReceiverLayer {
    var Message : string
    passive Empfang(message : string, sender : Sender) =>
        switch message {
            case "Coffee time?" : { sender.Antwort("Yes!") }
            case "Tea time?" : { sender.Antwort("No!") }
        }
}
```

Abbildung A.1.: Sender Empfänger Muster zur unidirektionalen 1:1 Kommunikation

```
Sender.mars
model SenderReceiverModel
layer SenderReceiverLayer
agent Sender on SenderReceiverLayer {
    var ReceiverAgent : Receiver
    var Antwort : string
    active Sende()
        => ReceiverAgent.Empfange("Coffee time?", me)
    passive Antwort(answer : string)
        => Antwort = answer
    reflex { ReceiverAgent := nearest Receiver; Sende() }
}

Receiver.mars
model SenderReceiverModel
agent Receiver on SenderReceiverLayer {
    var Message : string
    passive Empfang(message : string, sender : Sender) =>
        switch message {
            case "Coffee time?" : { sender.Antwort("Yes!") }
            case "Tea time?" : { sender.Antwort("No!") }
        }
}
```

Abbildung A.2.: Sender Empfänger Muster zur bidirektionalen 1:1 Kommunikation



```
Sender.mars
model SenderReceiverModel
layer SenderReceiverLayer
agent Sender on SenderReceiverLayer {
    var ReceiverAgents : [Receiver]
    var Antwort : string
    active Sende()
    => each(receiver in ReceiverAgents)
        receiver.Empfange("Drink coffee!")
    passive Antwort(answer : string)
    => Antwort = answer
    reflex { ReceiverAgents := explore Receiver; Sende() }
}

Receiver.mars
model SenderReceiverModel
agent Receiver on SenderReceiverLayer {
    var Message : string
    passive Empfang(message : string) =>
    switch message {
    case "Drink coffee!" : { println "Yes!" }
    case "Drink tea!" : { println "No!" }
    }
}
```

Abbildung A.3.: Sender Empfänger Muster zur unidirektionalen 1:n Kommunikation

A.4. Mehrkanal bidirektionale Nachricht

Problem Ein Agent X sendet eine Nachricht an *mehrere* andere Agenten Y_1, \dots, Y_n , die alle identisch oder logisch miteinander verwandt sein können und aus der Umgebung wahrnehmbar sind, inklusive Antwort vom Empfänger.

Realisierung in A.4

A.5. Agentenevolution

Problem Ein Agent X hat einen bestimmten Zustand erreicht, bei dem er den zugrundeliegenden Typ wechselt und damit seine Verhaltensweisen.

Realisierung in A.5

A.6. Agentenkommando

Problem Ein proaktiver Agent X verteilt Kommandos an einen reaktiven Agenten die anteilige Arbeit zu leisten, und zum Beispiel eine Berechnung auszuführen.

Realisierung in A.6

```

Sender.mars
model SenderReceiverModel
layer SenderReceiverLayer
agent Sender on SenderReceiverLayer {
    var ReceiverAgents : [Receiver]
    var Antwort : string

    active Sende()
    => each(receiver in ReceiverAgents)
        receiver.Empfange("Coffee time?", me)
    passive Antwort(answer : string)
    => Antwort = answer

    reflex { ReceiverAgents := explore Receiver; Sende() }
}

Receiver.mars
model SenderReceiverModel
agent Receiver on SenderReceiverLayer {
    var Message : string

    passive Empfang(message : string, sender : Sender) =>
        switch message {
        case "Coffee time?" : { sender.Antwort("Yes!") }
        case "Tea time?" : { sender.Antwort("No!") }
        }
}
    
```

Abbildung A.4.: Sender Empfänger Muster zur bidirektionalen 1:n Kommunikation

```

Raupe.mars
model Agentevolution
use Mars
layer Evolutionlayer
agent Caterpillar on Evolutionlayer {
    external var Evolutiontime : integer
    var MyVita : VitaData = new VitaData
    var IsAlive = true;

    reaction Morph when Time::Month(simtime) >= 4
    and Time::Month(simtime) <= 9 {
        (spawn Butterfly).Morph(me)
        IsAlive := false
    }

    passive Vita() => return MyVita
}

class VitaData {
    var Lifepoints : integer = 100
}

Schmetterling.mars
model Agentevolution
agent Butterfly on Evolutionlayer {
    var MyVita : VitaData

    passive Morph(caterpillar : Caterpillar)
    => MyVita := caterpillar.Vita

    reflex {
        // do something
    }
}
    
```

Abbildung A.5.: Evolutionsmuster um einen Agenten in einen anderen Typ zu morphen

```

Commander.mars
model CommandoModel
use Mars
layer CommandoLayer

agent Commander on CommandoLayer {
    var Underlings : [Underling]
    reflex {
        var command = [n : integer => return Fib(n)]
        Underlings = explore Underling
        Order(command)
    }

    active Order(command : (integer)=>integer) =>
    each(var underling in Underling) {
        underling.ReceiveCommando(new Command(command, random(100)))
    }

    def Fib(n : integer) {
        var a = 0; var b = 1; var c = 0
        if (n == 0) return a;
        for (var i = 2; i <= n; i++) { c = a + b; a = b; b = c; }
        return b;
    }
}

class Command {
    var Order : (integer)=>integer
    var OrderInfomation : integer

    initialize (order : (integer)=>integer, info : integer) {
        Order = order; OrderInfomation = info
    }
}

Underling.mars
model CommandoModel

agent Underling on CommandoLayer {
    var Command : Command

    reflex {
        Command?.Order.Invoke(Command.OrderInfomation)
    }

    passive ReceiveCommando(command : Command)
    => Command = command
}
    
```

Abbildung A.6.: Verteilung von Kommandos ausgehend von einem verantwortlichen Agenten

A.7. Routing von Nachrichten

Problem Eine Nachricht wird von einem Agenten X an einen Agenten Y gesendet, die an mehrere Agenten $Z_1 \dots Z_n$ von Y weitergeleitet werden soll. Die Nachricht kann dabei verloren gehen und Y informiert X über den Verlust.

Realisierung in A.7

```
Sender.mars
model RoutingModel
layer RoutingLayer
agent Sender on RoutingLayer {
    var Broker : Broker
    var Antwort : string
    active Sende()
    => Broker.Empfange(
        "Coffee time?", me)
    passive Antwort(answer : string)
    => Antwort = answer
    reflex { Broker := nearest Broker;
    Sende()
    }
}

Broker.mars
model RoutingModel
agent Broker on RoutingLayer {
    var ReceivedMessage : string
    var Receivers : [Receiver]
    var MessageSender : Sender
    passive Empfange(message : string,
    sender : Sender)
    ) {
        ReceivedMessage = message
        MessageSender := sender
    }
    reflex {
        Receivers = explore Receiver
    }
    active Delegate()
    => each(receiver in Receivers) {
        if(random() > 0.5) {
            receiver.Empfange(
                ReceivedMessage)
        } else {
            MessageSender
                .Antwort("Lost!")
        }
    }
}

Receiver.mars
model RoutingModel
agent Receiver on RoutingLayer {
    var Message : string
    passive Empfange(message : string) =>
    switch message {
    case "Coffee time?" : {
        println "Yes!"
    }
    case "Tea time?" : {
        println "No!"
    }
    }
}
```

Abbildung A.7.: Weiterleitung von Nachrichten mit Verlust

B. Modellierungsumgebung

B.1. Exemplarischer MARS Editor für das Modellieren in der Cloud

```
1 model SenderReceiverModel
2
3 layer SenderReceiverLayer
4
5 agent Sender on SenderReceiverLayer {
6
7     var ReceiverAgents : [Receiver]
8     var Agents : string[]
9
10    active Sende(receiver : Receiver) : bool {
11        return receiver.Empfange("Neue Nachricht_" + random(1))
12    }
13
14    reflex {
15        ReceiverAgents := explore Receiver
16        Sende(ReceiverAgents[0])
17    }
18 }
19
20
21 agent Receiver on SenderReceiverLayer {
22
23     var Message : string
24
25     passive Empfang(message : string) : bool {
26         Message := message
27         return true;
28     }
29 }
```

Save Revert Generate modified Welcome to MarsDsl-Web

Abbildung B.1.: Ein integrierter Webeditor zur Modellierung innerhalb der MARS Cloud mit angebundenem Modellgenerator

B.2. Beispiel der Integration von MathML Dokumentation

The screenshot shows a code editor on the left with the following code:

```

math.Log2
}
def Calculate
  StemBioma "Value" - STRING
  BranchBio m()Log2
  LeafBioma #
}
def Calculate
{
var treeA ;
if (treeA
if (treeA
treeAgeInYears := (treeAgeInYears + math.Log(treeAgeInY
return AgeType.ADULT;
}
    
```

On the right, a documentation panel for the `Method Log2(IntegerType) : NumberType` is displayed. It includes the title **Binary Logarithm**, a description: "Evaluate the binary logarithm of an integer number.", a parameter table with `x` and the note "Two-step method using a De Bruijn-like sequence table lookup.", a return statement: "return The binary logarithm of the input.", and a detailed explanation: "The binary logarithm ($\log_2 n$) is the power to which the number 2 must be raised to obtain the value n. That is, for any real number x: $x = \log_2 n \iff 2^x = n$. For example, the binary logarithm of 1 is 0, the binary logarithm of 2 is 1, the binary logarithm of 4 is 2, and the binary logarithm of 32 is 5. binary logarithm is the logarithm to the base 2. The binary logarithm function is the inverse function of the power of two function."

Abbildung B.2.: Vervollständigungsassistent innerhalb des Online Editors inklusive Dokumentation und integrierter MathML Sprache. Beispielbeschreibung der $\log_2(xyz)$ Logarithmus Definition

The screenshot shows the Eclipse IDE with the following code in the editor:

```

agent TestAgent on TestLayer {
  use mars.modelling.math as math
  reflex {
    var targetAgents := explore ProxyAgent
    var toAgent := targetAgents[0]
    targetAgents := nil
  }
}
    
```

The documentation panel for the `Method Abs(NumberType) : NumberType` is shown below the code. It includes the title, a description: "Returns the absolute value of a specified number.", a parameter table with `x` and the note "The number from which the absolute value shall be returned.", a return statement: "return A double-precision floating-point number, x, such that $0 \leq x < 1.7976931348623157E+308$ ", and a detailed explanation: "For any real number x the absolute value or modulus of x is denoted by $|x|$ (a vertical bar on each side of the quantity) and is defined as $|a| = \sqrt{a^2}$ ".

Abbildung B.3.: Vervollständigungsassistent innerhalb der Eclipse Zusatzlösung inklusive Dokumentation mittels formaler MathML Sprache. Beispielbeschreibung der $\|x\|$ Absolut Definition

B.3. Beispiel einer Anwendung des integrierten Zeitreihen-Layers

```
model TreeModel by "author" in "group" for "simple tree model"
layer TreeLayer as trees {
    use mars.modelling.time as time

    def GetSeason() : integer {
        if(time.Month(simtime) < 3 or time.Month(simtime) > 10) { return 1 }
        else { return 0 }
    }

    def Plant(x : integer, y : integer) : Tree {
        return spawn Tree at (x, y)
    }
}

time-series-layer PrecipitationLayer as pts {
    def GetName() : string { return "TsLayerPrecipitation" }

    def GetDisturbedValue() : number {
        return GetValue() as number
        // or simple return GetNumberValue()
    }
}
```

Abbildung B.4.: Zeitreihen-Layer integriert in ein exemplarisches Baummodell mit Abfrage von Daten und einer Agentenerzeugung

B.4. Beispiel des Sender Empfänger Modells

```
model SenderReceiverModel by "Daniel Glake" in "MARS-GROUP"

layer SenderReceiverLayer

agent Sender on SenderReceiverLayer {

    var ReceiverAgent : Receiver

    active Sende() : bool {
        return ReceiverAgent.Empfange("New message " + random(1))
    }

    reflex { ReceiverAgent := nearest Receiver; Sende() }
}

agent Receiver on SenderReceiverLayer {

    var Message : string

    passive Empfange(message : string) : bool {
        Message := message
        return true
    }
}
```

Abbildung B.5.: Beispielmodell des Sender-Empfänger Prinzips vollständig abgebildet mit der MARS DSL

B.5. Beispiel des angepassten Übersichtsfensters für die Modellstruktur

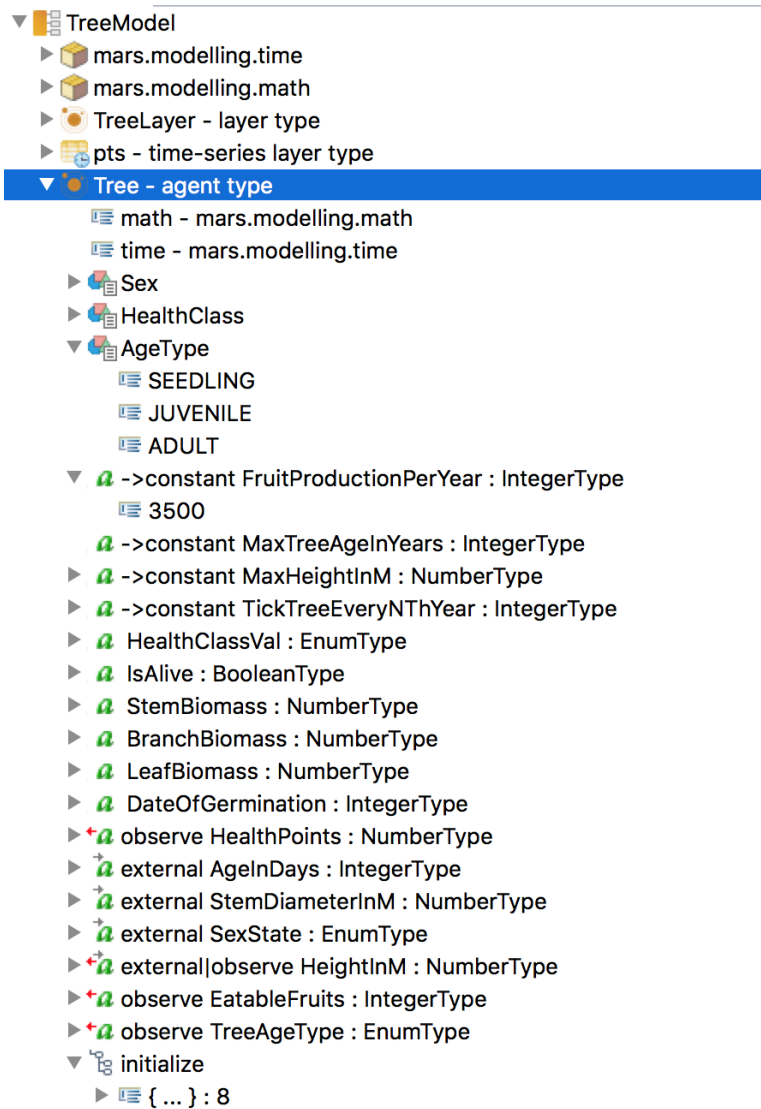


Abbildung B.6.: Übersichtsfenster des aktuellen Modells, unterteilt in MARS spezifische Elemente, Module und eingesetzten Beschreibungselementen.

B.6. Beispiel Konstruktion eines geospatialer Fangmodell-Plots mit R

```
1 # Load the library
2 library(raster)
3 library(rgdal)
4
5 # Load the input asc file, which is used withing the model
6 BH_RS <- raster("Bathymetrie.asc")
7
8 # Loaded agent positions and calculated paths for tick 50
9 root <- "output_data_50.csv"
10 phytoFile <- root
11 zooFile <- root
12 heringFile <- root
13 fischerFile <- root
14
15 # Load individual requested model results for the summary and positions
16 xyzDataPhyto <- read.csv(paste("phyto_", phytoFile, sep=""), [1],
17 header=FALSE, sep=";", stringsAsFactors = FALSE)
18 xyzDataZoo <- read.csv(paste("zoo_", zooFile, sep=""), [1],
19 header=FALSE, sep=";", stringsAsFactors = FALSE)
20 xyzDataHering <- read.csv(paste("hering_", heringFile, sep=""), [1],
21 header=FALSE, sep=";", stringsAsFactors = FALSE)
22 xyzDataFischer <- read.csv(paste("fischer_", fischerFile, sep=""), [1],
23 header=FALSE, sep=";", stringsAsFactors = FALSE)
24 xyzDataFischerPath <- read.csv("path_output50_.csv",
25 header=FALSE, sep=";", stringsAsFactors = FALSE)
26
27 # Construct a spatial data frame and project the
28 # latitude and longitude values to the
29 # coordinate reference system of the input
30 xyzSPPhyto <- SpatialPointsDataFrame(coords = xyzDataPhyto[,4:3],
31 data = xyzDataPhyto[,4:3],
32 proj4string=CRS("+proj=longlat+datum=WGS84+ellps=WGS84+towgs84=0,0,0"))
33 xyzSPZoo <- SpatialPointsDataFrame(coords = xyzDataZoo[,4:3],
34 data = xyzDataZoo[,4:3],
35 proj4string=CRS("+proj=longlat+datum=WGS84+ellps=WGS84+towgs84=0,0,0"))
36 xyzSPHering <- SpatialPointsDataFrame(coords = xyzDataHering[,4:3],
```

```
37     data = xyzDataHering[, 4:3],
38 proj4string=CRS("+proj=longlat_+datum=WGS84_+ellps=WGS84_+towgs84=0,0,0")
39 xyzSPFischer <- SpatialPointsDataFrame(coords = xyzDataFischer[, 4:3],
40     data = xyzDataFischer[, 4:3],
41 proj4string=CRS("+proj=longlat_+datum=WGS84_+ellps=WGS84_+towgs84=0,0,0"))
42 # Calculated paths from the fishers
43 xyzSPFischerPath <- SpatialPointsDataFrame(coords = xyzDataFischerPath[, 4:3],
44     data = xyzDataFischerPath[, 4:3],
45 proj4string=CRS("+proj=longlat_+datum=WGS84_+ellps=WGS84_+towgs84=0,0,0"))
46
47 # Join the raster cells with the points and set a color
48 xyzSPPhyto$Z <- BH_RS[xyzSPPhyto, ]
49 xyzSPPhyto$color <- rgb(255,0,255,maxColorValue=255, alpha=255)
50
51 xyzSPZoo$Z <- BH_RS[xyzSPZoo, ]
52 xyzSPZoo$color <- rgb(240,10,50,maxColorValue=255, alpha=255)
53
54 xyzSPHering$Z <- BH_RS[xyzSPHering, ]
55 xyzSPHering$color <- rgb(240,100,50,maxColorValue=255, alpha=255)
56
57 xyzSPFischer$Z <- BH_RS[xyzSPFischer, ]
58 xyzSPFischer$color <- rgb(28,206,170,maxColorValue=255, alpha=255)
59
60 xyzSPFischerPath$Z <- BH_RS[xyzSPFischerPath, ]
61 xyzSPFischerPath$color <- rgb(255,0,0,maxColorValue=255, alpha=255)
62
63 # Plot the input raster and add each spatial data frame
64 # with a respective individual symbol
65 plot(BH_RS)
66 plot(xyzSPPhyto, add = TRUE, bg = xyzSPPhyto$color, pch = 22, cex = 1)
67 plot(xyzSPZoo, add = TRUE, bg = xyzSPZoo$color, pch = 23, cex = 1)
68 plot(xyzSPHering, add = TRUE, bg = xyzSPHering$color, pch = 24, cex = 1)
69 plot(xyzSPFischerPath, add = TRUE, bg = xyzSPFischerPath$color, pch = 20, cex = 1)
70 plot(xyzSPFischer, add = TRUE, bg = xyzSPFischer$color, pch = 21, cex = 1)
```

C. Workshop Umfrage

Die nachfolgende Umfrage wurde im Rahmen des Südafrika Workshops ausgeteilt und von Teilnehmern ausgefüllt. Sie ist eingeteilt in die Bereiche: Vorkenntnisse der Teilnehmer, Bestimmung bisheriger Arbeiten oder bereits gesammelter Erfahrungen in der Modellierung, Verständnis sowohl der Agentenkonzepte, der Notation und der Verwendung der Beschreibungsmittel und Zufriedenheit sowie Fehlerprüfung mit einer zeitlichen Einteilung der eigenen Arbeit mit der MARS DSL.

MARS-DSL QUESTIONNAIRE

Please complete this questionnaire as conducted by MARS. Please do not write any identifying marks on the questionnaire as participants are meant to be anonymous. All information will be kept confidential. Any concerns can be communicated to (daniel.glake@haw-hamburg.de). Thank you for your time.

Background Questions

1. Which is the highest academic rank you own?

- Bachelor student
- Master student
- Ph.D. student
- Bachelor
- Master
- Ph.D.
- Other academic grade _____
Please enter the academic degree here

2. Which of these Multi-Agent Systems do you already know?

- GAMA
- NetLogo
- Swarm
- MATSim
- Repast Symphony
- MASS (Multi-Agent Simulation Suite)
- MARS (Multi-Agent Research and Simulation)
- MASON (Multi-Agent Simulation of Neighbourhoods)
- Other system _____
Please enter the system here

3. Which of these Agent-based modelling languages do you already know?

- SARL
- NetLogo
- StarLogo
- MARS-DSL
- AgentSpeak(L)
- Visual AgentTalk

- GAML (GAMA Modeling Language).....
- MAML (Multi-Agent Modelling Language).....
- FABLES (Functional Agent Based Language for Simulation).....
- Other language _____

Please enter the name

4. Have you already used another modelling- or development languages and if so, which ones?

5. How much experience do you have in software development?

Please answer on a scale of 0 to 5 - where 0 means that you have no experience, and 5 means that you have already developed complex Agent-based models yourself.

beginner						expert
	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	0	1	2	3	4	5

6. How much experience do you have in agent-based modeling?

beginner						expert
	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	0	1	2	3	4	5

If so, what was the model about?

Project specific questions

7. Was this a new MARS modelling project or built on an existing version?

Yes..... No.....

8. If you start a new MARS modelling project, how do you proceed?

9. Did the MARS DSL user interface help you modeling?

Please answer on a scale of 0 to 5 - where 0 means that the interface did not help at all, and 5 means that the interface was very good for understanding.

no help	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	helped a lot
	0	1	2	3	4	5	

10. Can you estimate the percentage of time that spent on the modelling task (not the online experiment task)?

11. Estimate the percentage of code that was generated

12. How many lines of code did your project consist of?

13. I looked into the generated code in order to be able to understand the underlying models

Yes..... No.....

14. I looked into the generated code in order to be able to write custom code

Yes..... No.....

DSL-Notation

	Totally agree	Somewhat agree		Somewhat disagree	Totally disagree
I have understood all notation elements.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
I consider the notation elements to be unnecessary complex.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
I found it easy to employ the notation elements for solving the exercises.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
I was not able to complete the exercises without frequent questions and support by an expert.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
I consider the various notation elements reasonable and necessary.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
In my opinion the various notation elements were difficult to distinguish from each other.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
I consider it easy to remember and employ the various notation elements, even without a legend.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
The modelling notation's successful usage requires a lot of previous knowledge.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
I felt very confident in employing the various notation elements.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
I experienced difficulties in using the various elements.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
The code is more readable	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Individual Based Modelling

	Totally agree	Somewhat agree	Somewhat disagree	Totally disagree	Not used
I could describe the desired agent behaviour.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
I was able to analyze the desired system properties.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
I found the the data concepts (types, inference, initialization, ...) easy to use.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
I was able to use external input data within the model.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
I was able to refine my model without much effort.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
The available concepts for my model were sufficient.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
MARS DSL restricts my freedom as modeller	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Validation and Running

	Totally agree	Somewhat agree	Somewhat disagree	Totally disagree	Not used
In case of a failure or incorrect behaviour I could quickly identify the error.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
The validation messages were meaningful enough	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
I was able to restart my model without much effort without any adjustments	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
I was able to restart my modified model without much effort	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
I was able to draw conclusions about the internal behaviour in the model from the visualized results.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Fewer errors occur	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

15. Which modeling core concepts of MARS DSL did you use?

- Agent definition (**agent**)
- Agent interaction (**passive and active**)
- Mass exploration of other agents (**explore**)
- Individual exploration of other agents (**nearest**)
- Rule based behaviour (**reaction**)
- Agent movement (**move**)
- Method definition (**def**)
- Mars library use (**use**)
- Tick based execution (**reflex**)
- Time based action (**simtime**)
- Raster layer location based data (**raster-layer**)
- Time series layer and data (**ts-layer**)
- Basic Layer for agent interaction (**layer**)
- Basic Layer for refactoring (**layer**)
- Observation of mental agent states (**observe**)
- External initialization of mental agent states (**external**)
- Other

16. Which other expressions of MARS DSL did you use?

- Switch with type guards (**switch**)
 - Execution branching (**if**)
 - Loop over iterables (**each**)
 - Literal expressions (**#,##**)
 - Basic for loop expressions (**for**)
 - Class definition (**class**)
 - Other
-
-

17. Which modeling concept did you lack in the language?

- Pattern matching
 - Message passing
 - Complex type observation
(e.g. own model objects)
 - Goal oriented action planing
(e.g. BDI agents)
 - Embedded data analysis (e.g. time series - or spatial data)
 - Direct execution of the model
 - Graphical notation
 - Other
-
-

18. **How satisfied are you - all in all - with the use of MARS-DSL as a modeling tool?**

Please answer on a scale of 0 to 5 - where 0 means that you are completely dissatisfied, and 5 means that you are completely satisfied. With values in between, you can rate your opinion.

completely						completely
dissatisfied						satisfied
	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	0	1	2	3	4	5

19. How satisfied are you - all in all - with the use of MARS as the simulation platform?

Please answer on a scale of 0 to 5 - where 0 means that you are completely dissatisfied, and 5 means that you are completely satisfied. With values in between, you can rate your opinion.

completely						completely
dissatisfied						satisfied
	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	0	1	2	3	4	5

20. Did you notice mistakes or errors in the language?

D. Danksagung

Mein Dank gilt Prof. Dr. Thomas Clemen der mich sehr motiviert hat, und die Idee bzw. den Vorschlag für eine solche Sprache hatte, und mich überzeugte der MARS Forschungsgruppe beizutreten. Um die Sprache weiter voranzutreiben ließ er mir viele Freiheiten bei der Eigenentwicklung, und gab mir immer wieder neue Impulse, und er stand mir mit seinem methodischen Wissen zur Seite. Ich möchte auch der MARS-Gruppe für deren Expertise danken, und Julius Weyl, der mit mir zusammen die Paper geschrieben hat. Außerdem gilt mein Dank auch Prof. Dr. Stefan Sarstedt der die Zweitbetreuung übernommen hat. Zuletzt möchte ich mich bei meiner Mutter für Ihre bedingungslose Unterstützung und ihre große Hilfe bedanken.

Literaturverzeichnis

- [Ahlbrecht u. a. 2016] AHLBRECHT, T. ; DIX, J. ; FIEKAS, N. ; KÖSTER, M. ; KRAUS, P. ; MÜLLER, J. P.: An architecture for scalable simulation of systems of cognitive agents. In: *International Journal of Agent-Oriented Software Engineering (IJAOSE)* 5 (2016), S. 232–265
- [Ahlbrecht u. a. 2014] AHLBRECHT, T. ; DIX, J. ; KÖSTER, M. ; KRAUS, P. ; MÜLLER, J. P.: A scalable runtime platform for multiagent-based simulation. In: DALPIAZ, F. (Hrsg.) ; RIEMSDIJK, M. B. van (Hrsg.) ; DIX, J. (Hrsg.): *Engineering Multiagent Systems II* Bd. 8758. Switzerland : Springer International Publishing, 2014, S. 81–102. – URL [/publication/2014-emas.pdf](#). – 2014-emas-maserati
- [Aho 2008] AHO, Alfred V.: *Compiler: Prinzipien, Techniken und Werkzeuge*. Pearson Deutschland GmbH, 2008
- [Al-Zinati u. a. 2013] AL-ZINATI, Mohammad ; ARAUJO, Frederico ; KUIPER, Dane ; VALENTE, Junia ; WENKSTERN, RZ: DIVAs 4.0: A multi-agent based simulation framework. In: *Proceedings of the 2013 IEEE/ACM 17th International Symposium on Distributed Simulation and Real Time Applications* IEEE Computer Society (Veranst.), 2013, S. 105–114
- [Alves-Foss 1999] ALVES-FOSS, Jim: *Formal syntax and semantics of Java*. Springer Science & Business Media, 1999
- [Aschermann u. a. 2016a] ASCHERMANN, M. ; KRAUS, P. ; MÜLLER, J. P.: LightJason: A BDI Framework Inspired by Jason / Department of Computer Science, TU Clausthal. Clausthal-Zellerfeld, Germany, October 2016 (IfI Technical Report IfI-16-04). – Forschungsbericht. – URL [/publication/2016-ifi-techreport.pdf](#)
- [Aschermann u. a. 2016b] ASCHERMANN, Malte ; KRAUS, Philipp ; MÜLLER, Jörg P.: LightJason: A BDI Framework inspired by Jason. In: *Multi-Agent Systems and Agreement Technologies: 14th Europ. Conf., EUMAS 2016, and 4rd Int. Conf., AT 2016, Valencia, Spain, 2016*, Springer International Publishing, 2016, S. tbd. – URL [/publication/2016-eumas.pdf](#)

- [Barišić 2013] BARIŠIĆ, Ankica: Evaluating the quality in use of domain-specific languages in an agile way. In: *Doctoral Symposium at the 16th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, CEUR-WS Bd. 1071, 2013, S. 1613–0073
- [Barišić u. a. 2011] BARIŠIĆ, Ankica ; AMARAL, Vasco ; GOULÃO, Miguel ; BARROCA, Bruno: Quality in use of domain-specific languages: a case study. In: *Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools* ACM (Veranst.), 2011, S. 65–72
- [Belaunde u. a. 2003] BELAUNDE, Mariano ; CASANAVE, Cory ; DSOUZA, Desmond ; DUDDY, Keith ; EL KAIM, William ; KENNEDY, Alan ; FRANK, William ; FRANKEL, David ; HAUCH, Randall ; HENDRYX, Stan u. a.: *MDA Guide Version 1.0. 1.* 2003
- [Bettini 2016] BETTINI, Lorenzo: *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016
- [Boydens und Steegmans 2004] BOYDENS, Jeroen ; STEEGMANS, Eric: Model driven architecture: The next abstraction level in programming. In: *Proceedings of the first European Conference on the Use of Modern Information and Communication Technologies*, 2004, S. 97–104
- [Caillou u. a. 2017] CAILLOU, Philippe ; GAUDOU, Benoit ; GRIGNARD, Arnaud ; TRUONG, Chi Q. ; TAILLANDIER, Patrick: A Simple-to-use BDI architecture for Agent-based Modeling and Simulation. In: *Advances in Social Simulation 2015*. Springer, 2017, S. 15–28
- [Cich u. a. 2016] CICH, Glenn ; KNAPEN, Luk ; GALLAND, Stéphane ; VUURSTAEK, Jan ; NEVEN, An ; BELLEMANS, Tom: Towards an Agent-based Model for Demand-Responsive Transport Serving Thin Flows. In: *the 5th International Workshop on Agent-based Mobility, Traffic and Transportation Models, Methodologies and Applications (ABMTRANS 2016)*, Procedia Computer Science, Elsevier, Februar 2016
- [Clark u. a. 2015] CLARK, Tony ; SAMMUT, Paul ; WILLANS, James: Applied metamodelling: a foundation for language driven development. In: *arXiv preprint arXiv:1505.00149* (2015)
- [Cooper und Jarre 2017a] COOPER, Rachel ; JARRE, Astrid: An agent-based model of the South African offshore hake trawl industry: part I model description and validation. In: *Ecological Economics* 142 (2017), S. 268–281

- [Cooper und Jarre 2017b] COOPER, Rachel ; JARRE, Astrid: An agent-based model of the South African offshore hake trawl industry: part II drivers and trade-offs in profit and risk. In: *Ecological Economics* 142 (2017), S. 257–267
- [Cossentino u. a. 2008] COSSENTINO, Massimo ; GALLAND, Stéphane ; GAUD, Nicolas ; HILAIRE, Vincent ; KOUKAM, Abderrafiâa: How to control emergence of behaviours in a holarchy. In: *Self-Adaptive and Self-Organizing Systems Workshops, 2008. SASOW 2008. Second IEEE International Conference on IEEE* (Veranst.), 2008, S. 180–185
- [Drogoul u. a. 2002] DROGOUL, Alexis ; VANBERGUE, Diane ; MEURISSE, Thomas: Multi-agent based simulation: Where are the agents? In: *International Workshop on Multi-Agent Systems and Agent-Based Simulation* Springer (Veranst.), 2002, S. 1–15
- [Ehrig u. a. 2008] EHRIG, Hartmut ; HECKEL, Reiko ; ROZENBERG, Grzegorz ; TAENTZER, Gabriele: *Graph Transformations*. Springer, 2008
- [Ferber u. a. 2004] FERBER, Jacques ; MICHEL, Fabien ; BÁEZ, José: AGRE: Integrating environments with organizations. In: *International Workshop on Environments for Multi-Agent Systems* Springer (Veranst.), 2004, S. 48–56
- [Fowler 2010] FOWLER, Martin: *Domain-specific languages*. Pearson Education, 2010
- [Galland u. a. 2015] GALLAND, Stéphane ; BALBO, Flavien ; GAUD, Nicolas ; RODRIGUEZ, Sebastian ; PICARD, Gauthier ; BOISSIER, Olivier: A multidimensional environment implementation for enhancing agent interaction. In: BORDINI, Rafael (Hrsg.) ; ELKIND, Edith (Hrsg.): *Autonomous Agents and Multiagent Systems (AAMAS15)*. Istanbul, Turkey, Mai 2015
- [Gargantini u. a. 2010] GARGANTINI, Angelo ; RICCOBENE, Elvinia ; SCANDURRA, Patrizia: Combining formal methods and MDE techniques for model-driven system design and analysis. In: *INTERNATIONAL JOURNAL* 1 (2010)
- [Gerber u. a. 1999] GERBER, Christian ; SIEKMANN, Jörg ; VIERKE, Gero: Holonic multi-agent systems. (1999)
- [Giret u. a. 2017] GIRET, Adriana ; TRENTESAUX, Damien ; SALIDO, Miguel A. ; GARCIA, Emilia ; ADAM, Emmanuel: A holonic multi-agent methodology to design sustainable intelligent manufacturing control systems. In: *Journal of Cleaner Production* 167 (2017), S. 1370–1386
- [Glake u. a. 2017] GLAKE, D. ; WEYL, J. ; HÜNING, C. ; DOHMEN, C. ; CLEMEN, T.: Modeling through Model Transformation with MARS 2.0. In: *Proceedings of the 2017 Spring Simu-*

- lation Multiconference*. Virginia Beach, Virginia, USA : Society for Computer Simulation International, 2017 (ADS '17)
- [Glake 2016] GLAKE, Daniel: Entwurf und Realisierung einer Modellierungssprache für Multi-Agenten Simulationen für MARS. 2016. – Forschungsbericht
- [Glake 2017] GLAKE, Daniel: Erweiterung der MARS DSL sowie Entwurf und Entwicklung eines Model-to-Text Transformators für MARS. 2017. – Forschungsbericht
- [Goll 2014] GOLL, Joachim: Architekturmuster. In: *Architektur-und Entwurfsmuster der Softwaretechnik*. Springer, 2014, S. 287–398
- [Götting u. a. 2013] GÖTTING, Klaus-Jürgen ; KILLIAN, Ernst F. ; SCHNETTER, Reinhard: *Einführung in die Meeresbiologie 2: Das Meer als Lebensraum und seine Nutzung*. Bd. 45. Springer-Verlag, 2013
- [Grignard u. a. 2013] GRIGNARD, Arnaud ; TAILLANDIER, Patrick ; GAUDOU, Benoit ; VO, Duc A. ; HUYNH, Nghi Q. ; DROGOUL, Alexis: GAMA 1.6: Advancing the art of complex agent-based modeling and simulation. In: *International Conference on Principles and Practice of Multi-Agent Systems* Springer (Veranst.), 2013, S. 117–131
- [Hejlsberg u. a. 2003] HEJLSBERG, Anders ; WILTAMUTH, Scott ; GOLDE, Peter: *C# language specification*. Addison-Wesley Longman Publishing Co., Inc., 2003
- [Herrington 2003] HERRINGTON, Jack: *Code Generation in Action*. Greenwich, CT, USA : Manning Publications Co., 2003. – ISBN 1930110979
- [Hüning u. a. 2016] HÜNING, Christian ; ADEBAHR, Mitja ; THIEL-CLEMEN, Thomas ; DALSKI, Jan ; LENFERS, Ulfa ; GRUNDMANN, Lukas: Modeling & simulation as a service with the massive multi-agent system MARS. In: *Proceedings of the Agent-Directed Simulation Symposium* Society for Computer Simulation International (Veranst.), 2016, S. 1
- [International Hydrographic Organization und Sieger 2012] INTERNATIONAL HYDROGRAPHIC ORGANIZATION, IHO ; SIEGER, Rainer: *Limits of oceans and seas in digitized, machine readable form*. 2012. – URL <https://doi.org/10.1594/PANGAEA.777975>
- [Jennings 2001] JENNINGS, Nicholas R.: An agent-based approach for building complex software systems. In: *Communications of the ACM* 44 (2001), Nr. 4, S. 35–41

- [Kahraman und Bilgen 2015] KAHRAMAN, Gökhan ; BILGEN, Semih: A framework for qualitative assessment of domain-specific languages. In: *Software & Systems Modeling* 14 (2015), Nr. 4, S. 1505–1526
- [Karsai u. a. 2014] KARSAI, Gabor ; KRAHN, Holger ; PINKERNELL, Claas ; RUMPE, Bernhard ; SCHINDLER, Martin ; VÖLKELE, Steven: Design guidelines for domain specific languages. In: *arXiv preprint arXiv:1409.2378* (2014)
- [Kleppe u. a. 2003] KLEPPE, Anneke G. ; WARMER, Jos B. ; BAST, Wim: *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional, 2003
- [Leroy 2014] LEROY, Xavier: *Formal Proofs of Code Generation and Verification Tools*. S. 1–4. In: GIANNAKOPOULOU, Dimitra (Hrsg.) ; SALAÜN, Gwen (Hrsg.): *Software Engineering and Formal Methods: 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings*. Cham : Springer International Publishing, 2014. – URL http://dx.doi.org/10.1007/978-3-319-10431-7_1. – ISBN 978-3-319-10431-7
- [Lind 2001] LIND, Jürgen: *Iterative software engineering for multiagent systems: the MASSIVE method*. Springer-Verlag, 2001
- [Macal und North 2014] MACAL, Charles ; NORTH, Michael: Introductory tutorial: Agent-based modeling and simulation. In: *Proceedings of the 2014 winter simulation conference* IEEE Press (Veranst.), 2014, S. 6–20
- [Macal 2016] MACAL, Charles M.: Everything you need to know about agent-based modelling and simulation. In: *Journal of Simulation* 10 (2016), Nr. 2, S. 144–156
- [Macal und North 2010] MACAL, Charles M. ; NORTH, Michael J.: Tutorial on agent-based modelling and simulation. In: *Journal of simulation* 4 (2010), Nr. 3, S. 151–162
- [Melles und Lehfeldt 2014] MELLES, Johannes ; LEHFELDT, Rainer: Marine Daten-Infrastruktur Deutschland (MDI-DE). In: *Die Küste, 82 MDI-DE* (2014), Nr. 82, S. 1–23
- [Mohagheghi u. a. 2013] MOHAGHEGHI, Parastoo ; GILANI, Wasif ; STEFANESCU, Alin ; FERNANDEZ, Miguel A.: An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases. In: *Empirical Software Engineering* 18 (2013), Nr. 1, S. 89–116
- [de Mutsert u. a. 2017] MUTSERT, Kim de ; LEWIS, Kristy ; MILROY, Scott ; BUSZOWSKI, Joe ; STEENBEEK, Jeroen: Using ecosystem modeling to evaluate trade-offs in coastal management:

- Effects of large-scale river diversions on fish and fisheries. In: *Ecological Modelling* 360 (2017), S. 14–26
- [Ng u. a. 2011] NG, Karen ; WARREN, Matt ; GOLDE, Peter ; HEJLSBERG, Anders: The Roslyn Project, Exposing the C Sharp and VB compiler's code analysis. In: *White paper, Microsoft* (2011)
- [OMG 2014] OMG, OMG: *Meta Object Facility (MOF) Core Specification*. 2014
- [O'Regan 2017] O'REGAN, Gerard: Z formal specification language. In: *Concise Guide to Formal Methods*. Springer, 2017, S. 155–171
- [Palermo u. a. 2012] PALERMO, Jeffrey ; BOGARD, Jimmy ; HEXTER, Eric ; HINZE, Matthew ; SKINNER, Jeremy: *ASP.NET MVC 4 in Action*. Manning Publications Co., 2012
- [Parr 2009] PARR, Terence: *Language implementation patterns: create your own domain-specific and general programming languages*. Pragmatic Bookshelf, 2009
- [Parr 2013] PARR, Terence: *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013
- [Price 2016] PRICE, Mark J.: *C# 6 and .NET Core 1.0: Modern Cross-Platform Development*. Packt Publishing Ltd, 2016
- [Radtke und Straškraba 1980] RADTKE, E ; STRAŠKRABA, M: Self-optimization in a phytoplankton model. In: *Ecological Modelling* 9 (1980), S. 247–268
- [Rodrigues u. a. 2017] RODRIGUES, Ildevana P. ; BORBA CAMPOS, Márcia de ; ZORZO, Avelino F.: Usability Evaluation of Domain-Specific Languages: A Systematic Literature Review. In: *International Conference on Human-Computer Interaction* Springer (Veranst.), 2017, S. 522–534
- [Rodriguez u. a. 2014] RODRIGUEZ, Sebastian ; GAUD, Nicolas ; GALLAND, Stéphane: SARL: a general-purpose agent-oriented programming language. In: *the 2014 IEEE/WIC/ACM International Conference on Intelligent Agent Technology*. Warsaw, Poland : IEEE Computer Society Press, 2014
- [Rodriguez u. a. 2011] RODRIGUEZ, Sebastian ; HILAIRE, Vincent ; GAUD, Nicolas ; GALLAND, Stéphane ; KOUKAM, Abderrafiâa: Holonic Multi-Agent Systems. In: *Self-organising Software*. Springer, 2011, S. 251–279

- [Rossum 1995] ROSSUM, Guido: Python Reference Manual. Amsterdam, The Netherlands, The Netherlands : CWI (Centre for Mathematics and Computer Science), 1995. – Forschungsbericht
- [Russell und Wilensky 2008] RUSSELL, Eric ; WILENSKY, Uri: Consuming spatial data in NetLogo using the GIS Extension. In: *The annual meeting of the Swarm Development Group*, 2008
- [Russell und Norvig 2002] RUSSELL, Stuart J. ; NORVIG, Peter: Artificial intelligence: a modern approach (International Edition). (2002)
- [Sarjoughian und Markid 2012] SARJOUGHIAN, Hessam S. ; MARKID, Abbas M.: EMF-DEVS modeling. In: *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation-DEVS Integrative M&S Symposium* Society for Computer Simulation International (Veranst.), 2012, S. 19
- [Sierhuis u. a. 2009] SIERHUIS, Maarten ; CLANCEY, William J. ; HOOF, Ron J.: *Multi-Agent Programming:: Languages, Tools and Applications*. Kap. Brahms An Agent-Oriented Language for Work Practice Simulation and Multi-Agent Systems Development, S. 73–117. Boston, MA : Springer US, 2009. – URL http://dx.doi.org/10.1007/978-0-387-89299-3_3. – ISBN 978-0-387-89299-3
- [Spinellis 2001] SPINELLIS, Diomidis: Notable design patterns for domain-specific languages. In: *Journal of systems and software* 56 (2001), Nr. 1, S. 91–99
- [Starke 2015] STARKE, Gernot: *Effektive Softwarearchitekturen: Ein praktischer Leitfaden*. Carl Hanser Verlag GmbH Co KG, 2015
- [Strembeck und Zdun 2009] STREMBECK, Mark ; ZDUN, Uwe: An approach for the systematic development of domain-specific languages. In: *Software: Practice and Experience* 39 (2009), Nr. 15, S. 1253–1292
- [Tchappi Haman u. a. 2017] TCHAPPI HAMAN, Igor ; KAMLA, Vivient c. ; GALLAND, Stéphane ; KAMGANG, Jean-Claude: Towards an Multilevel Agent-based Model for Traffic Simulation. In: *the 6th International Workshop on Agent-based Mobility, Traffic and Transportation Models, Methodologies and Applications (ABMTRANS'17)*, Springer, Februar 2017
- [Voelter u. a. 2013] VOELTER, Markus ; BENZ, Sebastian ; DIETRICH, Christian ; ENGELMANN, Birgit ; HELANDER, Mats ; KATS, Lennart C. ; VISSER, Eelco ; WACHSMUTH, Guido: *DSL*

- engineering: Designing, implementing and using domain-specific languages.* dslbook.org, 2013
- [Weidauer 2007] WEIDAUER, Alexander: Bathymetrie und Topographie Projekt IMKONOS. (2007)
- [Weyl u. a. 2018] WEYL, J. ; GLAKE, D. ; CLEMEN, T.: Agent-Based Traffic Simulation at city scale with MARS. In: *Proceedings of the 2018 Spring Simulation Multiconference*. Baltimore, Maryland, USA : Society for Computer Simulation International, 2018 (ADS '18). – accepted
- [Wilensky 2015] WILENSKY, U: NetLogo Dictionary. In: *NetLogo User Manual 3* (2015)
- [Wooldridge und Jennings 1995] WOOLDRIDGE, Michael ; JENNINGS, Nicholas R.: Intelligent agents: Theory and practice. In: *The knowledge engineering review* 10 (1995), Nr. 2, S. 115–152
- [Wooldridge 2000] WOOLDRIDGE, Michael J.: *Reasoning about rational agents*. MIT press, 2000
- [Xing u. a. 2017] XING, Lei ; ZHANG, Chongliang ; CHEN, Yong ; SHIN, Yunne-Jai ; VERLEY, Philippe ; YU, Haiqing ; REN, Yiping: An individual-based model for simulating the ecosystem dynamics of Jiaozhou Bay, China. In: *Ecological Modelling* 360 (2017), S. 120–131
- [Zambonelli u. a. 2003] ZAMBONELLI, Franco ; JENNINGS, Nicholas R. ; WOOLDRIDGE, Michael: Developing multiagent systems: The Gaia methodology. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 12 (2003), Nr. 3, S. 317–370

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

Hamburg, 08. Mai 2018

 Daniel Glake