Entwicklung einer Sprache mittels Metamodell und Model-to-Code Transformator für MAS

Daniel Glake
daniel.glake@haw-hamburg.de

Hamburg University of Applied Sciences,
Dept. Computer Science,
Berliner Tor 7,
20099 Hamburg, Germany
http://mars-group.org

Zusammenfassung. Diese Arbeit präsentiert eine domänenspezifische Sprache für Multi-Agenten Simulationen, für das Multi-Agenten Simulations Framework MARS (Multi-Agent Research and Simulation). Es konzentriert sich auf die Beschreibung individueller Agenten und ihres internen Verhaltens zuzüglich variablen- und konstanten- mentalen Zuständen. Hauptfokus liegt auf einem kleinen Modellkörper der allgemein akzeptierte Konzepte und korrespondierende MARS spezifische Mittel enthält. Die MARS DSL übernimmt diese Herausforderung und bildet dabei die Basis als Integrationsplattform aufbauend auf einem MARS spezifischen Metamodell als Ausgangspunkt. Ein Model-to-Code Transformator wird eingesetzt um anschließend die Generierung der notwendigen Implementierung zu übernehmen.

Schlüsselwörter: Multi-Agent Simulation, Modelling, DSL, Code-Generation

1 Einleitung

Die agentenbasierte Modellierung und Simulation hat sich zu einem wichtigen Forschungsfeld etabliert. Zur gleichen Zeit hat sich das Interesse an dem Prozess der Modellierung verstärkt, was von Seiten einer angepassten Modellrepräsentation und aus Sicht der Benutzbarkeit des zugrundeliegenden Simulations Frameworks immer noch eine Herausforderung darstellt [16]. Einige Forschungsgruppen haben mit textuellen und graphischen Werkzeugen versucht diese Kluft zu reduzieren, indem Modelle, soweit möglich, ohne technische Implementierungsdetails erzeugt werden. Es bleibt jedoch die Frage offen wie ein Modellierer von seinem erdachten, konzeptionellen Modell zu einem ausführbaren Simulationsmodell kommt, was aus der Konzeptbeschreibung in die Implementierung führt, ohne ein tiefes technisches Wissen bezüglich der Plattform zu benötigen [13].

1.1 Problemstellung

Ziel dieser Arbeit ist die Vorstellung einer neuen domänenspezifischen Modellierungssprache für agentenbasierte MARS Simulationen [11]. Diese auf dem

Konzept der Layerbildung basierenden Modelle, bieten innerhalb des Agenten die Möglichkeit einer individuellen Agentenbeschreibung, die mit dieser Sprache abstrahiert wird. Zuvor waren Kenntnisse über technische Abhängigkeiten sowie allgemeine softwaretechnische Programmierkenntnisse der eingesetzten C# Sprache notwendig. Die individuelle Beschreibung von Agenten steht im Mittelpunkt und baut auf dem vorhandenen MARS Framework auf, was mittels einer zusätzlichen Modellierungsschicht in Form eines Metamodells einen Datenkontrakt bildet. Ein Model-to-Code Transformator überführt diese plattformabhängige Repräsentation in seinen ausführbaren Code. Die Notwendigkeit einer zusätzlichen DSL liegt darin begründet, dass vielfach komplexe Konzepte zur Beschreibung von Agenten zwar vorliegen, aber im bisherigen MARS Modelling Workflow darauf ausgelegt sind, vom Modellierer selbst implementiert zu werden. Dieses Problem findet sich dabei nicht nur allein in der Simulationswelt, sondern generell für die Sprachentwicklung [16]. Beispiel: Eine typische Interaktion zwischen Agenten und die Modellierung der Verarbeitung sensorischer Daten die eintreffen. Allesamt müssen diese vom Modellierer sowohl auf fachlicher als auch technischer Seite verantwortet werden. Dies wiederum erzeugt den zusätzlichen Nachteil, dass deren Implementierungen an das betrachtete Modell gekoppelt ist und somit nur durch aufwendige Refaktorisierungs-Schritte generalisiert werden kann. Die Auslagerung in eine dafür vorgesehene Sprache erlaubt eine striktere Trennung der fachlichen Modellierung auf der einen Seite als auch der technischen Realisierung von Konzepten auf der anderen Seite, eingebettet in einer modellgetriebenen Architektur erlaubt sie die automatisierte Transformation in die entsprechende Implementation [3][4].

2 MARS Transformation Workflow

Ein wichtiger Vorteil in der Verwendung einer DSL Schicht zur Modellierung der Agenten basiert auf der Transformationspipeline die damit einhergeht, und automatisiert von einem Modell in ein Zielmodell überführt. Das Konzept wird auch als Model Driven Software Development (MDSD) oder Model Driven Development bezeichnet und soll mit dieser DSL ein Teil zum Modellierungsprozess für Simulationen beitragen. Die Wichtigkeit dieser modellgetriebenen Entwicklung wurde bereits von Markus Voelter verdeutlicht, der damit sowohl Produktivität als auch Systemkonsistenz hervorhebt [16]. Die Transformation ist innerhalb der Model Driven Architecture (MDA) [6] eingebettet und wird darin von einem Modell in das jeweilige Zielmodell überführt. Angelehnt an [18] wurde hierzu eine zusätzliches Generatormodell ergänzt, was der eigentlichen technischen Implementierung entspricht und sich in einer Datenstruktur wiederspiegelt, die alle notwendigen Codefragmente beinhaltet um eine ausführbare Simulation zu erhalten. Abbildung 1 zeigt den modifizierten Prozess dazu.

Es existieren drei Basismodelle die zur MDA-orientierten Simulation notwendig sind. Das Computational Independent Model (**CIM**) repräsentiert die Problemdomäne der Simulation und beinhaltet sowohl die Fragestellung als auch Menge verfügbarer Daten, wie beispielsweise Messungen. Das Platform Inde-

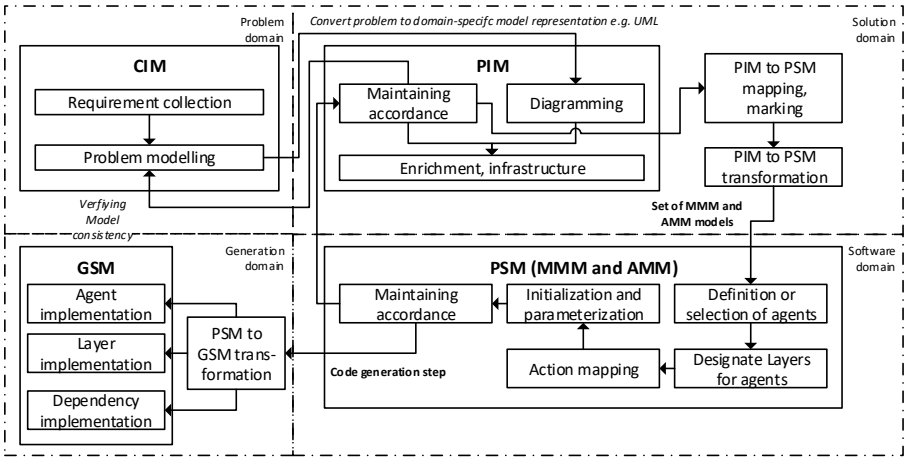

Abb. 1. Komonentenmodell des MDA aus [14] ergänzt um ein Generatormodell, übernommen aus [6].

pendent Model (**PIM**) repräsentiert das konzeptionelle Modell der betrachteten Fragestellung unter Einbezug der beteiligten Entitäten und Relationen. Ein oder mehrere Plattform spezifische Modelle (**PSM**) wie eben das MARS Metamodell sowie das komplementäre Generator spezifische Modell (**GSM**), wie das in 4.5 beschriebene Modell zur strukturierten Codeerzeugung. Weitere Details zu diesem Workflow und der Sinnhaftigkeit der modellgetriebenen Simulationsentwicklung findet sich in [6] [5] sowie [16].

3 MARS Metamodell

Das in [6] beschriebene Metamodell dient als Abstraktion der zugrundeliegenden Modellierung von MARS Simulationen, was die Definition von Agenten und deren Verhalten, verschiedene Layertypen, Interaktionen zwischen Agenten sowie Sensordefinitionen zur Abfrage der Umwelt umfasst.

Es wird eine strikte Trennung zwischen Modell- und Agentendefinition gefordert die separat und damit einer individuellen Agentenbeschreibung folgen [8]. Es werden aktive und passive Aktionen beschrieben, die in Abschnitt 4.4 näher erläutert werden. Das Agentenverhalten und jegliche Ausdrücke müssen jedoch als freiliegender Codeabschnitt näher erläutert werden. Dies ist auch noch ein Problem, da die eigene Beschreibung in diese überführt werden muss. Ein Agent besitzt Zustände sowie Metainformationen zur Abfrage aus einem Repository und kann von anderen Agententypen erben.

Die Modelldefinition sieht die Definition von Sensoren und damit der Abhängigkeiten zwischen Layer vor, auf denen die jeweiligen Agenten sitzen. Layertypen

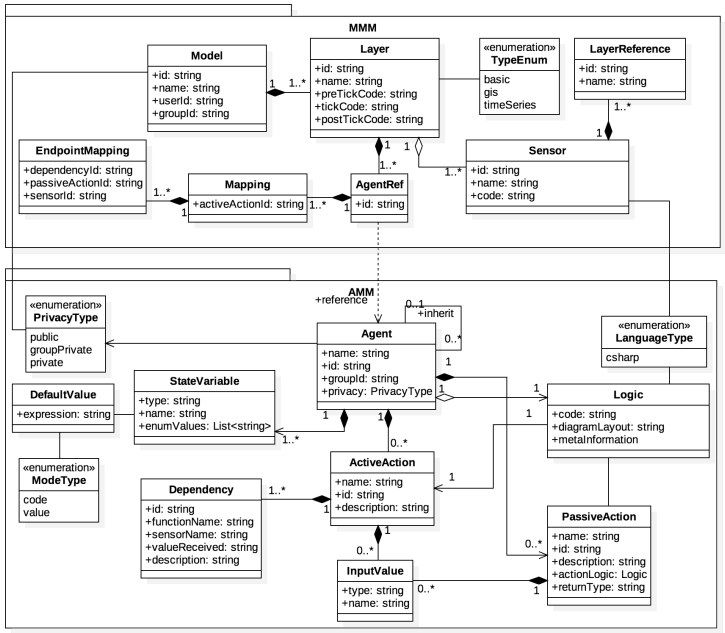

Abb. 2. PSM des MARS Frameworks, übernommen aus [6].

können sowohl Datenquellen (TIME-SERIE) oder normale Bewegungslayer sein. Agentenreferenzen werden gesetzt um letztendlich das vollständige Modell zu erhalten. Weitere Details finden sich allesamt in [6].

4 MARS DSL

Die MARS DSL als Basissprache von Agentenmodellen ist eine domänen-spezifische agenten-orientierte Modellierungssprache. Sie bietet eine Reihe von Abstraktionen für spatiale Agententypen an und bewegt sich im Modellierungs-workflow in Phase eins und zwei [6]. Die MARS DSL bildet einen Modellierungslayer mit Fokus auf die Formulierung spatial und zeitsensitiver Agenten, abstrahiert somit von der darunterliegenden Layer-Technologie und den technischen Abhängigkeiten die zuvor im Code vorzufinden waren 1.1.

4.1 Modell und Layerdefinition

Inspiriert durch die Modellierungssprache GAML [8] mit seiner objektorientierten Beschreibung eines Agenten und einer Reihe von Aktionen die aus der erfüllten Bedingung resultieren, wurde im Zuge von General Purpose Sprachen wie Scala und Ruby versucht einen Syntax zu implementieren, die eine möglichst

geringe Lernkurve für den Modellierer aufweist. Die Sprache wurde hierbei durch das open-source Framework Xtext implementiert. Eine automatische Transformation in das MMM und AMM wurde eingebaut, einschließlich eines Typsystems, basierend auf dem Konzept von [2]. Die Folge ist eine startk typisierte Sprache um damit individuell mentalen Agentenzuständen zu beschreiben. Listing 4.1 und 4.2 zeigen hierzu eine Form des Wolf-Schaf-Grass Modells, was einer erweiterten Form des Räuber-Beute Modells von [17] entspricht, formuliert mit der neuen DSL. Das Beispiel zeigt die Verwendung von unterschiedlichen Sprachfeatures der MARS-DSL. Dazu gehören die allgemeine Modell und Layerdefinition, die Agentendefinition mit deren Zuständen sowie eine Verhaltensbeschreibung eines individuellen Agenten..

```
model WolveSheepModell by "author" in "MARS-GROUP"
    for "predator prey example with the new DSL"
layer WolveLayer
layer SheepLayer
layer GrassLayer
sensor SensorDefinition {
   [ SheepLayer ] => WolveLayer
   [ GrassLayer ] => SheepLayer
}
```

Abb. 3. Beispieldefinition von Basic Layern und seiner Sensordefinition mit den Layerabhängigkeiten nach der MARS DSL

Die erste Zeile enthält Metadaten wie Autorenangabe, eine Gruppenbezeichnung sowie eine interne Kurzbeschreibung des Modells. Die Layerdefinitionen ermöglichen nur eine Basisdefinition, die keine vorhandenen Layertypen des MARS Frameworks [11] sowie des MMM [6] zum Beispiel Time Series Layer zur Datenbereitstellung unterstützt, sondern nur zur Gruppierung der Agenten und dem die Zuordnung zu einem Layer genügt. Um die Interaktion und das Explorieren 4.4 der Agenten zu ermöglichen wird mit der Sensordefinition eine notwendige Layerabhängigkeit definiert, um zu wissen welche Agenten, die auf diesen Layer liegen, abfragen zu können.

4.2 Agentendefinition und mentale Zustände

Der spatiale Agententyp erlaubt die Positionierung, angegeben durch wechselseitig ausschließende Schlüsselwörter über `latitude` und `longitude` Attribute oder `grid_x` und `grid_y` Koordinaten innerhalb einer zwei-dimensionalen Gridumgebung, ähnlich zur Positionierung in [8]. Der Agent verwaltet eine Menge von mentalen Zuständen, die sich während der kontinuierlichen Simulation verändern können und wird charakterisiert durch ein `val` für konstante Zustände oder `var` für variable Zustände gekennzeichnet. Mittels `observe` wird markiert, dass die

Werte als Teil des Simulationsergebnisses gelten und zwischengespeichert werden sollen. Über `external` wird veranlasst, diesen Wert zum Simulationsbeginn entsprechend durch von außen vorhandene Daten zu parametrisieren. Aufgrund von Vorgaben des MARS Metamodells sind alle Typen primitiv typisiert [6], die zur späteren Transformation bereitgestellt werden. Referenzen zwischen den Agenten sind zwar möglich, jedoch werden diese nicht als mentale Zustände angesehen sondern dienen der Benutzbarkeit. Neue Typen können aus den primitiven zusammengesetzt werden und spezielle Typen, die der Modellierung dienlich sind, die der `date` Typ mit korrespondierender Funktionalität wie `AddYears(..)` sind ebenfalls möglich.

```
agent Sheep on SheepLayer {
   var IsAlive : bool = true
   var Energy : integer
   var Rule : string
   var EnergyMax : integer = 100
   var GrassAgents : [Grass]
   var GrassAgent : Sheep
   var TargetDistance : number
   var Hunger : integer
   reaction R1 when TargetDistance <= 1.4143 and Hunger > 20 {
   Rule := "R1 - Restrained Hunting"; Eat(GrassAgent) }
   reaction R2 when TargetDistance <= 5 and Hunger > 40 {
   Rule := "R2 - Average Hunting"; move to GrassAgent }
   reaction R3 when Hunger > 60 {
   Rule := "R3 - Aggressive Hunting"; move to GrassAgent }
   passive GetFoodValue() : integer { return Energy }
   passive Die() { IsAlive := false }

   active Eat(grass : Grass) {
   Energy := Energy + grass.GetFoodValue(); grass.Cut() }
   reflex {
   //Reduce the energy of this wolve by One
   //plus random value in every tick
   Energy := Energy - (1 + random(3))
   //If energy lower than zero then die
   if(Energy <= 0)
   IsAlive := false
   //Get all sheeps ordered by their distance
   GrassAgents := explore; GrassAgent := GrassAgents[0];
   //Get all sheeps ordered by their distance
   TargetDistance := distance(GrassAgent) }
}
```

Abb. 4. Beispieldefinition eines Sheep Agenten von Basic Layern und seiner Sensordefinition mit den Layerabhängigkeiten, formuliert mit der MARS DSL

4.3 Agentenverhalten

Mit als `reaction` deklarierte Anweisungen, geben eine Reihe von Prädikaten und Aktionen an, die allesamt aus der zuvor möglichen `reason` Methode extrahiert werden. Dieses im MARS Framework verwendete Logikkonzept stellt dabei lediglich einen Methodenrumpf bereit, innerhalb dessen pro Simulationstick die Agentenausführung durchgeführt wird [11]. Mit den `reactions` lässt sich dahingehend nun ein reaktives Verhalten explizit modellieren, auch wenn mit dem `reflex` eine weitere Abstraktion vorliegt, die direkt dieses Konzept der `reason` Methode widerspiegelt, insbesondere wenn es um die Auswertung und Analyse von Sensordaten geht, die pro Tick für die Ausführung neuer `reaction` Statements verwendet werden sollen oder für die Beendigung des Ticks zum Zuge kommen. Die Ausführungsreihenfolge der `reactions` ist hierbei nicht willkürlich, sondern ist abhängig von deren Auftreten im Modell. Diese Art der Modellierung wurde dabei durch die Sprache GAML [8] inspiriert die an dieser Stelle die Bezeichnung eines `reflex` verwendet hat, was nicht mit dem `reflex` innerhalb dieser Sprache verwechselt werden darf.

Das Prädikat und der Body des `reaction` Konstrukts erlauben darin sowohl bekannte arithmetische und logische Ausdrücke als auch domänenspezifische Operationen, wie sie im Reaction `AgeRelatedDying` zu sehen ist. Darin erlaubt ein spezielles `simtime` Literal den Zugriff auf die bisher abgelaufene Simulationszeit über die eine zeitbedingte Steuerung vorgenommen wird. Des Weiteren sind insbesondere für spatiale Agenten gesonderte Ausdrücke vorhanden, wie diese beispielhaft in der Reaction `AverageHunting` zu sehen ist. Darin findet sich ein `move to` Ausdruck, womit eine Bewegungsaktion in Richtung des naheliegendsten Grass-Agenten vorgenommen werden kann.

Die Komplexität der Modellierung

4.4 Agenteninteraktion

Entgegen des weitläufig eingesetzen Nachrichtenversands zur Agenteninteraktion, wie in [19] und [15] wird die Interaktion für die DSL über sogenannte `active` und `passive` Actions beschrieben, bei dem die beteiligten Agenten direkt die Daten anderer Agenten über deren Schnittstellen verändern. Die Variante einer Nachrichtenübermittlung mit beteiligten Aktoren wurde zwar bedacht, jedoch aus Gründen des Aufwands und der Natur der kontinuierlichen Simulation nicht weiter verfolgt.

Passive und aktive Aktionen stellen sowohl die Fähigkeiten des Agenten selbst dar und zudem seine Schnittstellendefinition mit Daten und Aktionen, die von anderen Agenten ausgeführt werden können. Dabei ist die Modellierung sowohl für Vor- und Nachbedingungen geeignet um Aktionen, die von außen durch fremde Agenten durchgeführt werden, zu kontrollieren. Beispiel: Der oben beschriebene Sheep Agent bietet eine passive Aktion zum Sterben an, indem dieser durch eine aktive Aktion eines Jägeragenten (in diesem Fall ein Wolf) zunächst den aktuellen Energielevel über die passive Aktion `GetFoodValue()` abruft und daraufhin den Agenten mittels `Die()` sterben lässt. Diese Aktion ist jedoch an die

Bedingung geknüpft, dass ein Agent in der näheren Umgebung anwesend ist um diese Friss Aktion tatsächlich auch ausführen zu können. Aus dieser Anforderung wird somit das Prädikat des `reaction` Konstrukts deutlich, der im Falle des `Sheep` und seiner `RestrainedHunting` Reaction dafür sorgt, dass nur dann eine Aktion erfolgt, wenn die Distanz zwischen den Agenten unterhalb eines gesetzten Wertes liegt. Analog gilt dies auch für den nicht betrachteten `Wolve` Agenten.

Damit andere Agenten in der Umgebung ausgemacht werden können, existiert hinsichtlich des spatialen Agententyps ein verallgemeinertes Konstrukt zur Exploration von anderen spatialen Entitäten. Über die `explore` Expression innerhalb des `reflex` Statements wird damit eine simplifizierte Explorationsaktion aufgenommen und über alle anderen Agenten eines gewählten Agententyps (hier der Grass Agent), aus der zuvor eingetragenen Sensordefinition 4.1 einhergehenden Layerabhängigkeit, exploriert und als distanzgeordnete Collection zurückgegeben. Mittels `distance` Expression wird anschließend zur naheliegendsten Entität des neu zugeordneten `Grass` Agenten der Abstand abgefragt und einem variablen State zugeordnet. Diese Zuordnung wiederum ist innerhalb des `reflex` definiert, der damit eine immer wiederkehrende Aktion beschreibt und zu diesem Zweck seine Verwendung findet.

Neben diesen sprachlichen Beschreibungsmitteln bietet die Sprache darüber hinaus die Basis bestehend aus einer Teilmenge der C# Spezifikation [9]. Diese Menge umfasst einfache arithmetische und logische Ausdrücke sowie Wenn-dann Konstrukte, Schleifenkonstrukte sowie ein Hilfsmethoden Konstrukt, die lediglich zur internen Strukturierung des Agenten Verwendung finden können und komplexe Operationen auslagern.

4.5 Transformation in das MMM und AMM

Neben den vorhandenen Beschreibungsmitteln liegt ein wichtiger Fokus der Sprache auf der Transformation von dieser Beschreibung inklusive Ausdrücken, Statements und Kreuzreferenzen untereinander in lesbare Form des MMM und des AMM 3. Während die Agentendefinition 4.2 mit deren mentalen Zuständen und der Zuordnung des Agenten zu einem Layer aus dessen Kreuzreferenz entnommen und hinterlegt werden kann, liegt eine große Herausforderung in der Transformation interner Logik und der ausgewählten Ausdrücke vor, dass diese nach der Vereinbarung vom AMM innerhalb zur Beschreibung der `AgentLogic` bereits den verwendeten C# Code enthalten müsste. Damit diese Übersetzung möglichst reibungslos funktioniert wurde aufbauend auf der Sprache Xtend und seines Generators [1] eine Kompilation jedes Einzelausdrucks von einfachen wie direkt verwendbaren `Binäroperation` und `Unäroperationen` bis hin zu komplexeren Ausdrücken wie der Exploration anderer Agenten `explore ([Agenttype])` definiert. Hinter der Kompilation jedes Einzelkonstrukts liegt nun der tatsächliche Code als komplexe Zeichenkette mit Folge von Anweisungen oder komplexen Blöcken vor, die wiederum aus einer Sequenz von Ausdrücken bestehen können. Besondere Aufmerksamkeit bekommen hierbei die `reaction` Statements, die geordnet aus dem AST gezogen und direkt als Wenn-dann Ausdruck im Code

repräsentiert wird. Das gesamte Konstrukt wird anschließend vor oder nach der `reflex` Anweisung, bedingt dadurch ob es sich um eine `pre reaction` handelt oder nicht, eingetragen und in die `AgentLogic` als Code übernommen.

5 Model-to-Code Transformation

Die Codegenerierung nimmt neben der PIM und PSM Modellierung eine zentrale Rolle ein, ist es doch dieser Transformator, der letztendlich aus der PSM Instanz (MMM und AMM) die Implementierung erzeugt [4]. Zur Realisierung jedes Implementierungsaspekts wurde dazu ein Syntax Modell entwickelt, was eine direkte Abstraktion des tatsächlichen C# Implementierung widerspiegelt und durch Erzeugung einzelner Knoten den jeweiligen Codeaspekt wiedergibt [6], dessen Idee die template-basierte und besucher-basierte Codeerzeugung vereint [10]. Es bildet somit eine zusätzliche Schicht über den C# Abstract Syntax Tree (AST) und offeriert einzelne Operationen, die nur noch mit den Daten des MMM und AMM befüllt werden müssen. Abbildung 5 zeigt dazu die Einteilung und Zuordnung jedes Elements des MMM und AMM zu seinem jeweiligen Syntax Knoten. Wichtig hierbei ist, dass es sich um ein angereichertes Modell handelt, bei dem jegliche Kanten zwischen den Knoten auch eine Rückwärtsreferenz besitzen. Aufgrund der Kreuzreferenzen innerhalb der Implementation, können diese Daten somit einfach abgefragt werden.

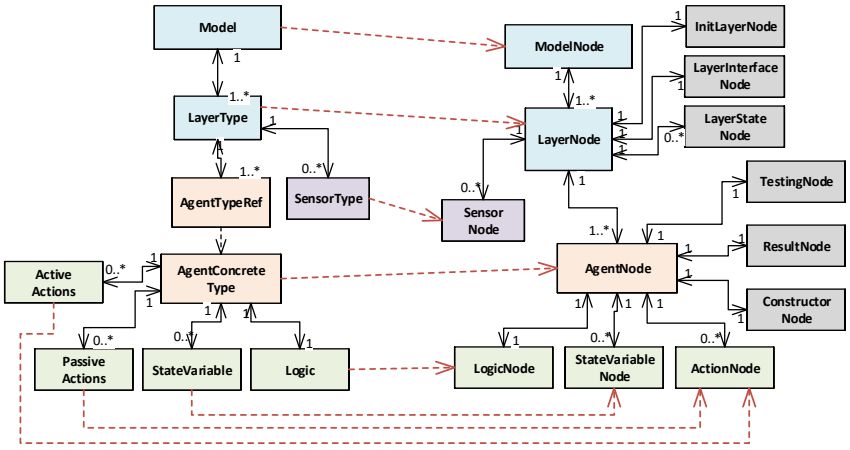

Abb. 5. Vertikaler Schnitt und Übersicht des Syntax Modells (rechts) mit seinen Assoziationen zum kombinierten MMM und AMM (links).

6 Weitere Schritte

Für die Modellierung von Agenten konnte bisher eine abstrahierte Beschreibungssprache erstellt sowie ein Referenzmodell in Form eines Räuber-Beute Modells beschrieben werden. Technische Details und das darüberlegende MDA sind allesamt in [7] näher beschrieben. Kern dieser Arbeit war neben der Analyse und der Entwicklung des PIM's zudem die Umsetzung mittels einer der zuvor untersuchten DSL-Entwicklungsinfrastruktur. Außerdem wurde ein Typsystem inklusive Validierungskomponente für die statische Semantik realisiert die in Verbindung mit dem Model-to-Model Transformator in das Metamodell überführt.

Das finale System kann damit für weitere Arbeiten herangezogen werden und neben der Analyse der Mächtigkeit der Sprache außerdem als Integrationsplattform für weitere Modellierungskonzepte, wie die Sehfähigkeit der Agenten [12] bzw. integrierte Näherungsverfahren für gewöhnliche Differentialgleichungen wie GAML, eingesetzt werden. [8]. Des Weiteren soll ein Model-to-Code Transformator nach dem Entwurf in 5 erstellt werden, der das MMM und AMM in 3 in seine Implementierung überführt, und dazu eine Reihe von Anpassungen zur Verbesserung der Ausführbarkeit des Modells vornimmt.

6.1 Thesis Outline

Innerhalb des Kontexts der Modellierung von Simulationsmodellen bleibt die Frage offen wie mit komplexeren Simulationen, wie z.B. mit kognitiven Aspekten oder unter einem sozialen organisatorischen Blickpunkt umgegangen wird, und ob diese Sprache für solche Fragestellungen ausreichend ist. Entgegen der geringeren Benutzbarkeit und Analysierbarkeit der Turing-vollständigen Sprachen [16] ist es wichtig zu evaluieren inwieweit sichergestellt wird, dass ein Modellierer sich nicht wieder in einer zu stark technischen Welt befindet und damit der größte Vorteil der DSL, nämlich der Bildung einer Projektion [4] auf das Notwendigste, wegfällt. Daraus ergibt sich eine Fragestellung die sich damit befasst, wie abstrakt genug eine Sprache für spatial ausgelegte Simulationsmodelle sein kann, ohne dabei seine Anwendungsdomäne, für die sie gedacht ist, zu vernachlässigen. Denkbar ist hier eine Studie bei der existierende Modelle und Nachbildungen dieser, mit der MARS DSL sowohl strukturell als auch semantisch verglichen werden. In diesem Zusammenhang ist der Vergleich beider Simulationsergebnisse notwendig über die untersucht wird, ob das Modell weiterhin konsistent zum geprüften Erwartungswert ist sowie der strukturellen Analyse, welche Modellaspekte vereinfacht wurden, wegfielen oder gar ergänzt werden konnten. Demnach besteht das Risiko, dass wichtige fachliche Elemente nicht in der Sprache formuliert werden und damit verfälschte Simulationsergebnisse entstehen.

7 Fazit

Diese Arbeit präsentierte eine neue domänenspezifische agenten-orientierte Modellierungssprache mit Hauptfokus auf die Beschreibung individueller spatialer

Agenten, mithilfe minimaler technischer Details. Es wurde Priorität auf eine hohe Erweiterbarkeit der Sprache gelegt um mit zukünftig angedachten Simulationsmodellen, die mit den Mitteln nicht auskommen, agieren zu können. Alles innerhalb eines dafür vorgesehene PIM und unter Berücksichtigung der faktisch großen Anzahl von Agenten-Architekturen, Interaktionsmechanismen etc., die in der Forschung angeboten werden. Die Sprache schafft eine Basis für zusätzliche aufbauende Beschreibungs-Konzepte und ist dazu innerhalb einer MDA integriert, an dessen Ende ein Model-to-Code Transformator die Überführung in eine Implementation übernimmt. Es konnte gezeigt werden, dass die Mächtigkeit der Sprache noch ungeklärt ist und wie diese mithilfe zwei möglicher Ansätze überprüft werden kann.

Literatur

1. Bettini, L.: Implementing domain-specific languages with Xtext and Xtend. Packt Publishing Ltd (2016)
2. Bettini, L., Stoll, D., Völter, M., Colameo, S.: Approaches and tools for implementing type systems in xtext. In: International Conference on Software Language Engineering. pp. 392–412. Springer (2012)
3. Drozdova, M., Kardos, M., Kurillova, Z., Bucko, B.: Transformation in model driven architecture. In: Information Systems Architecture and Technology: Proceedings of 36th International Conference on Information Systems Architecture and Technology–ISAT 2015–Part I. pp. 193–203. Springer (2016)
4. Fowler, M.: Domain-specific languages. Pearson Education (2010)
5. Gascueña, J.M., Navarro, E., Fernández-Caballero, A.: Model-driven engineering techniques for the development of multi-agent systems. Engineering Applications of Artificial Intelligence 25(1), 159–173 (2012)
6. Glake, D., Weyl, J., Hüning, C., Dohmen, C., Clemen, T.: Modeling through model transformation with mars 2.0. In: Proceedings of the 2017 Spring Simulation Multiconference. ADS '17, Society for Computer Simulation International, Virginia Beach, Virginia, USA (2017), accepted
7. Glake, D.: Entwurf und realisierung einer modellierungssprache für multi-agenten simulationen für mars. Tech. rep. (2017)
8. Grignard, A., Taillandier, P., Gaudou, B., Vo, D.A., Huynh, N.Q., Drogoul, A.: Gama 1.6: Advancing the art of complex agent-based modeling and simulation. In: International Conference on Principles and Practice of Multi-Agent Systems. pp. 117–131. Springer (2013)
9. Hejlsberg, A., Wiltamuth, S., Golde, P.: C# language specification. Addison-Wesley Longman Publishing Co., Inc. (2003)
10. Herrington, J.: Code generation in action. Manning Publications Co. (2003)
11. Hüning, C., Adebahr, M., Thiel-Clemen, T., Dalski, J., Lenfers, U.A., Grundmann, L., Dybulla, J., Kiker, G.A.: Modeling & simulation as a service with the massive multi-agent system mars. In: Proceedings of the Agent-Directed Simulation Symposium. p. 8. ADS '16, Proceedings of the 2016 Spring Simulation Multiconference, San Diego, CA, USA (2016), `http://dl.acm.org/citation.cfm?id=2972193.2972194`
12. Kuiper, D.M., Wenkstern, R.Z.: Agent vision in multi-agent based simulation systems. Autonomous Agents and Multi-Agent Systems 29(2), 161–191 (2015)

13. Lynch, C.J.: Cloud-Based Simulators: Making Simulations Accessible to Non-Experts and Experts Alike (2012), 3630–3639 (2014)
14. Nikiforova, O., Cernickins, A., Pavlova, N.: Discussing the difference between model driven architecture and model driven development in the context of supporting tools. In: Software Engineering Advances, 2009. ICSEA'09. Fourth International Conference on. pp. 446–451. IEEE (2009)
15. Rodriguez, S., Gaud, N., Galland, S.: Sarl: a general-purpose agent-oriented programming language. In: Web Intelligence (WI) and Intelligent Agent Technologies (IAT), 2014 IEEE/WIC/ACM International Joint Conferences on. vol. 3, pp. 103–110. IEEE (2014)
16. Voelter, M., Kolb, B., Szabó, T., Ratiu, D., van Deursen, A.: Lessons learned from developing mbeddr: a case study in language engineering with mps. Software & Systems Modeling pp. 1–46 (2017)
17. Volterra, V.: Variations and fluctuations of the number of individuals in animal species living together. J. Cons. Int. Explor. Mer 3(1), 3–51 (1928)
18. Wimmer, M., Burgueño, L.: Testing m2t/t2m transformations. In: International Conference on Model Driven Engineering Languages and Systems. pp. 203–219. Springer (2013)
19. Zhang, M., Verbraeck, A.: A composable prs-based agent meta-model for multi-agent simulation using the devs framework. In: Proceedings of the 2014 Symposium on Agent Directed Simulation. pp. 1:1–1:8. ADS '14, Society for Computer Simulation International, San Diego, CA, USA (2014), http://dl.acm.org/citation.cfm?id=2665049.2665050