ABSTRACT

The development of simulation models confronts scientists with the necessity of transforming concepts from theoretical models to executable code. Albeit modern simulation platforms provide APIs to abstract away technology, this task remains complex. Therefore a model-to-code transformation is essential, allowing domain experts to focus on their model instead of implementation details. This paper presents a multi-level transformation concept to facilitate building multi-agent simulations for domain experts. With domain-specific tool support, model ideas can be developed without managing technical requirements. This insures that the modeler is exclusively concerned with the conceptual model by utilizing a MARS (Multi-Agent Research and Simulation) Meta-Model (MMM) and Agent Meta-Model (AMM). We outline the MMM as underlying structure, discuss the foundations of model-driven development and the in-place transformation of the MMM as executed by the MARS modeling workflow. In addition, we present a model-to-code generator that creates the final simulation model.

Keywords: Multi-Agent System, Code Generator, Meta-Model, Model Transformation.

1 INTRODUCTION

In the last decades, agent-based modeling and simulation has turned into an important research field (Heath et al. 2009). At the same time, the level of interest in the modeling process itself has grown as well (Volter 2011). Apart from finding a suitable model representation, the usability aspect of modeling frameworks still is an issue. Some research groups have developed systems offering a graphical and textual user interface, so that models can be created easily without having to consider technical implementation details (Wang et al. 2006). The remaining questions is, how modelers can get from conceptual modeling to an executable
simulation in an automated way, without being obliged to have deep knowledge about the technical aspects of the system.

This question is linked to a number of challenges that should be addressed both on the modeling side as well as the technical side (Taranti et al. 2010). At the moment, a modeler has to be assisted by a technical expert to implement a concept. However, all details outside of the studied domain should be hidden for the modeler, so that the realization of the simulation idea can be implemented without any further help. While much effort has been put in developing meta-models and formal descriptions of agent-based systems, like Rodriguez, Gaud, and Galland (2014), Bae, Lee, and Moon (2012) and Grignard, Taillandier, Gaudou, Vo, Huynh, and Drogoul (2013), most of them are only for general purposes and not for domain specific modeling. In case of general purpose meta-models, the respective tools and thus also the modelers are burdened with technical details more than necessary, whereas in case of domain specific models expressiveness can be lost (Padilla et al. 2014).

2 RELATED WORK

2.1 Meta-Modeling

Various research groups have been trying to find an approach to formalize and simplify the development of agent simulation models (Kinny and Georgeff 1996). As a well-established model description language, the Unified Modeling Language (UML) serves as a basis for several toolkits and frameworks that extend the existing UML standards by adding agent-specific components. An illustrative example is the agent representation and modeling language AUML (Agent UML) (Odell et al. 2000). Accommodating key concepts of the unified modeling language, AUML realizes many features needed for the proper design of agent-oriented systems and therefore offers significant advantages over conventional UML. A related implementation of this concept is MAS-ML (da Silva and de Lucena 2003), a modeling language extending the UML scope, which is based on a meta-model supporting the agent-oriented software engineering paradigm. After creating it, the MAS-ML model can be mapped to an UML meta-model, which in return can be translated into object-oriented programming language code.

2.2 Model-Driven Design

An approach to simplify the modeling process is the INGENIAS methodology and development framework (Pavón et al. 2006). With the use of meta models and model transformation tools during the modeling process, INGENIAs focuses on model-driven development for the design of multi-agent systems. With SARL (Rodriguez et al. 2014), there is one platform-independent and general-purpose agent-programming language. The SARL language does not impose a particular execution infrastructure, but uses Janus (Galland, Gaud, Rodriguez, and Hilaire 2010) to run the SARL agents. These are based on a language meta-model. Other examples relevant in this context are AgentTool (Scott and Wood 2000) and Repast Simphony (North et al. 2007), both prominent modeling frameworks supporting graphical model design.

3 METHODS

3.1 MARS

The MARS (Multi-Agent Research and Simulation) System is an agent-based simulation framework for massive scale distributed simulations based on the concept of Modeling and Simulation as a Service (Hünig et al. 2016), (Parker and Epstein 2011). Coining the trend of moving large-scale simulations to the cloud,
MARS uses this advantage to offer a convenient web interface for creating simulations. End users can not only run simulations, but can create their own’s as well with the newly created tool set. In addition, the simulation results can be analyzed in multiple ways, including 3D visualization and visual analytics. While the second version of MARS is still under final development by the MARS research group at Hamburg University of Applied Sciences, the first edition (MARS 1.0) is already being used to research elephant behavior in a complex savanna ecosystem model.

### 3.2 Model Kruger National Park

The most advanced model currently simulated with the MARS framework relates to the Kruger National Park (KNP) in South-Africa and is developed as part of ARS AfricaE (http://www.ars-africae.org), a joint German and South-African research project. This ecological model focuses on the savannah vegetation development over a 21 year period from 1989 to 2010. Savanna vegetation structure consists of large trees, shrubs and grassland and is believed to be mainly affected by elephants, herbivores and fire. Interactions between elephants and Marula trees are of particular interest and are therefore modelled as individual entities. The large scale model consists of over 100,000 agents, simulates the whole areal extent of the Kruger National Park and is being run on an hourly temporal resolution.

Model development started two years ago and could only be done by an person with skills in programming at that time. The need for a programmer rendered the domain experts unable to extend the model on their own. Even small changes couldn’t be done if the expert lacked experience in C#. After considering teaching the domain experts computer programming another approach was taken. Using meta modeling, the structure of MARS models and thus the KNP model is predefined in order to generate a language-and MARS-specific generator model. This includes all the necessary information for execution as well as performance-relevant implementation details. Following this way, domain experts fill the meta model with their information instead of working directly on code. In addition MARS domain-specific modelling concepts can be transformed into these structure to make them applicable for the simulation models.

### 3.3 Model Transformation

The process of converting one model to another is referred to as a transformation (Fowler 2010). The concept called Model Driven Software Development (MDSD) or Model Driven Development (MDD) is very closely related to the simulation modeling process itself. The importance of its usage in software development is demonstrated by Markus Voelter. In his opinion, the MDD is a step in the right direction for enhancing productivity and system consistency (Volter 2011). The transformation is embedded within MDSD and Model Driven Architecture (MDA). Commonly, MDA and MDSD have many similarities (Wimmer and Burgueño 2013) and were introduced by the Object Management Group (OMG) in 2001 to standardize the MDSD framework.

According to Nikiforova, Cernickins, and Pavlova (2009), the MDA consists of a transformation from a conceptual model to a platform specific model, including multiple transformation steps. Supplemented by an additional generator model, the final code generation will be possible. Figure 1 shows the modified MDA process we propagate.

There are three basic models for the MDA-oriented simulation development:

1. The Computational Independent Model (CIM) represents the problem domain of the simulation, including the initial research question and a set of existing data, like measurements and results from other papers.
2. The Platform Independent Model (PIM) analyzes the domain and represents the conceptual models of the research question with their entities and relationships. A domain-specific tool developed for this purpose promotes translation into the Platform Specific Model. The MARS DSL (section 4.4) is an example for such a textual tool, which offers the modeling infrastructure and brings the diagramming into formalized agent-specific syntax.

3. One or more Platform Specific Models (PSM) are constructed by defining or selecting existing agents from the agent repository as well as domain-dependent layer types derived from the set of existing models. This corresponds to the modeling workflow as described in section 4.2.

The MDA components in Figure 1 represent activities in the MDSD. The information is exported from one component to the next, indicated by the arrow direction. Cursive captions indicate, that these steps have to be done manually (Jouault et al. 2008), (Bergmann et al. 2015), (Popoola et al. 2016). Other transformations, as from PIM to PSM as well as PSM to GSM, can be performed automatically and integrated into a modeling pipeline (Volter 2011). This is done by transforming the tool-specific model representation into the MMM or AMM as a target model, which is then converted into code by the generator.

4 RESULTS

4.1 MARS Meta-Model

This section introduces the MARS Meta-Model (MMM), an exchange format for multi-agent models. The MMM is the foundation for exchanging model concepts, drafting simulations and generating code. Within the modeling process, this meta-model will be filled with information about agents, layers and basic simulation data. Later it serves as input for the code generator which will then produce executable model code for the MARS framework. The MMM is realized in form of a standardized interchangeable JSON format.
The first component of the MARS Meta-Model is the Agent Meta-Model (AMM), which determines the data structure representing a specific agent type. The AMM provides a static view on the internal structures and properties of an agent type, as well as all actions and interactions it is capable of. No assumptions on inter-agent connections are made at this time.

An agent type itself consists of several parts starting with its name and a group Id which points out the user group the agent type belongs to. A list of structured attributes describes the agent’s inner state through state variables. The first two parameters define the type and name of the variable as shown in Figure 2. Possible types include integers, numbers, strings and enums. Additionally, mathematical or logical expressions can serve as input, interpreted at instantiation. Another option is to leave the value blank and later assign the variable with information from data sources like for example a time series dataset.

The agent’s internal behavior and its actions not directly affecting others (like walking around) are determined by the so-called logic block. This part of the AMM can be described using the programming language C# or a graphical editor which is still in development.

After assigning the agent’s state variables and simple actions, the behavior will be supplemented by more complex actions, performed in collaboration with others. For this purpose we discriminate two types of
actions. The first ones, so called active actions, are performed by the agent itself. The second type are passive actions, which are triggered when another agent tries to perform them on the current agent, for example an elephant eating leaves from a tree. In that scenario, the elephant executes the active action eat leaves and the tree agent provides the passive action leaves get eaten, so that the action can be conducted. Finding agents to perform these actions with is another subject. Different agents, like animal species for instance, have distinct ranges of perception which is represented through the concept of sensors. Agents use sensors to find agents of specific types in different ranges. Queries can be used to e.g. look for trees larger than 3m in a 2km radius. Sensors are specified similarly to agents. They have an id, a name and a code block containing the commands to execute. Last part of a sensor is the reference to a layer, used as data source.

Following this step of agent creation, the environment has to be set up, in which the agents are going to "live". MARS uses a layer concept, in which every layer represents a specific part of the environment. These layers can hold information about weather conditions, the terrain or contain agents. As mentioned before, the end users can import data to use it in simulations. These imported datasets are stored in layers and are thereby available in the simulation. Each layer offers interfaces for agents respectively layers to interact or to access information. Specifically agents use them to discover their surroundings.

Adding a layer to the MMM begins with determining its name and type. Currently we support multiple types of layers. The simplest one is the basic layer, which can be seen as the minimal effort to create a layer. Agents are typically created on basic layers because they only need an environment to live in.

The last distinctive feature is passive and active layers. In the first case, there are no methods called on the layer, so that it simply acts as an information source. Agents can query the included data to use this for their reasoning, but the layer itself does not do anything. In order to change information on the layer, like updating environment conditions on certain events, the layer has to be an active one. The layer then implements three methods for triggering actions before, while and after a tick gets executed.

4.2 Creating a Model

The MARS framework supports a modeling as a Service process that is divided into three main phases. In the first phase, the agent types used in a specific model are defined. New agent types can be designed by creating an instance of the AMM. This includes defining state variables, active and passive actions as well as its general behavior in a simulation run. It is also possible to select agent types that already exist, as all type definitions are stored for reuse in a central repository.

The second phase can be initiated when the first phase is completed and the agents are specified. This step is very important, as to make the simulation work, interactions between agents are needed in order to create a dynamic system. The key concept of this phase is the implementation of layers where agents are embedded in. Additionally, the mapping of active and passive actions and the specification of sensors are performed.

The third and last phase implicates the finalization and parametrization of the model with assignments of concrete data sets used for the initialization of layers and agents. After the MMM is complete, the code generation can be triggered, starting with a series of transformations.

4.3 Model-Driven Software Development under MARS

The modeling workflow formulated in Hüning, Adebahr, Thiel-Clemen, Dalski, Lenfers, Grundmann, Dybulla, and Kiker (2016) is supplemented by an automated code generation based on the MARS Meta-Model. It propagates a procedure in which the individual agent and layer development is in the foreground. Domain-
specific tools such as the MARS-DSL offer a selected set of existing constructs for agent definition and domain-specific validations, which will then be converted into the MMM and AMM.

The MDA architecture was expanded to include an additional generator domain with a corresponding generator model. This introduces the PSM (described in Section 3.3) and forms the transformation by means of an automatic code generation step. The respective GSM is developed for automatic code generation based on the PSM. Previously generated code does not have to go through the entire generation step again. Especially in the case of preselected and modified agent types from the repository, only changes to the respective nodes are necessary.

### 4.4 Model-to-Model Transformation: MARS DSL

MARS Base-DSL is a domain-specific, agent-oriented textual modeling language. It offers a series of design abstractions for spatial agent types, compatible with the first two phases of the MARS modeling workflow (Section 4.2). MARS Base-DSL forms a modeling layer with focus on the formulation of spatial and time sensitive agents, abstracting the underlying layer technology.

Inspired by the modeling language GAML (Grignard et al. 2013) as well as general purpose languages like Scala and Ruby, we tried to implement a syntax with a shallow learning curve. The language was created with Xtext, an open-source framework for developing DSL’s and programming languages. An automatic transformation into the MMM and AMM has been implemented, including a type system, based on Bettini, Stoll, Völter, and Colameo (2012). Therefore the language is strictly typed and allows individual description of mental agent states. AgentReactions as seen in Listing 1 describe the behavior, which is checked via predicates in every simulation tick.

```plaintext
model MicroKNP by "author" in "MARS-GROUP" for "marula_example_for_SCS"
layer MarulaLayer
spatial agent MarulaTree on MarulaLayer {
  val FruitMultiplier : int = 2
  external observe var Biomass : real
  external val HeightInM : real
  observe var Fruits : int = 0
  external var TreeAge : int
  external val GerminationDate : date

  reaction RotFruits when simtime.Months == 4 { Fruits = 0 }
  reaction CastFruits when CalculateTreeAge() > 1 {
    Fruits = Fruits + delta.Days * FruitMultiplier
  }
  reflex { CalculateBiomass() }
  reflex { CalculateTreeAge() : int {
    var ageInYears : int = simtime - GerminationDate;
    if (simtime < GerminationDate.AddYears(ageInYears)) {
      ageInYears = ageInYears - 1;
    }
    return ageInYears;
  }
}
```

Listing 1: Example code of the MARS DSL with an agent type of a Marula tree.

The first line contains meta information such as the author, a group name and an internal short description. The spatial agent type allows the positioning, carried out by means of mutually exclusive keywords for
latitude and longitude or via grid_x and grid_y coordinates in a two-dimensional grid environment, similar to GAML in Grignard, Taillandier, Gaudou, Vo, Huynh, and Drogoul (2013). The agent manages a set of mental states that are changed during the continuous simulation, characterized by a val for constant states or var for variables. Observe marks values that are saved as part of the simulation result, while external designates external parametrization. Due to the contract of the MARS Meta-Model, all types are primitives. New types can be assembled accordingly from these primitives. This way, special types, like the date type with corresponding functionalities (for example AddYears (. . .)) are possible.

Using so called reactions, a series of predicates with action bodies, all of which have been extracted from the pre-existing reason method are executed in every simulation tick (Hüning et al. 2016). This way, reactive behavior can explicitly be modeled. However, with the reflex construct this reason method concept is still possible, whereby the body of these constructs is only executed after the reactions. The execution order of the individual reactions depends on the order of their occurrence in the agent block.

To transform this agent and layer type into the PSM, we focus on the conversion of reaction and reflex bodies into corresponding C# code. With an inline-code expander (Herrington 2003), the code is parsed and wherever predefined keywords are found, the corresponding production code is inserted. The other parts are elements of the MMM and AMM and will be assigned to their respective attributes and parameters. The resulting JSON file is then automatically generated with a JSON serializer.

4.5 Model-to-Text Transformation: Viewpoint Based Code Generation

To close the gap between modeling and actual execution, an important step lies within the last model-to-text transformation to produce the final source code (Fowler 2010). Inside an abstracted model, manipulations can be made and additional code fragments can be added. The MMM instance is given as input to the model-to-text transformation process and is then transformed into several sub-trees by a combined approach of a visitor-based and template-based generation process (Herrington 2003), (Fowler 2010). Using the Microsoft Roslyn compiler, a compilation is generated in the form of C# code.

The code generation is divided into several so-called viewpoints, which define, what resources an entity of AMM and MMM has access to and what kind of behaviors can be used to solve tasks. The text generation can be controlled by defined projections on the MMM, so that parts of the input model can be accessed directly. In this way, code-specific contracts are defined in the respective nodes that are used for code generation. The transformation rules are formulated for each node of the syntax model.

4.5.1 Model-to-Text Transformation: Rule Dependencies

Transformation rules can have dependencies among each other (Fowler 2010). Such dependencies act similar to triggers, so that a transformation rule is called only if it is defined as a starting rule or it is triggered by one of the dependencies. Dependencies between rules can be specified with the generator though a register method. Besides the dependencies, the transformation rule also defines a set of pre-and post-conditions in the form of a predicate, based on the backward edges. This can be used to check for missing rule dependencies. If no trigger rule is present, the precondition would be violated and the generation would stop. The predicate is obtained in the form of a query by means of an internal language integrated query, which checks along the entities and relationships, whether or not a syntax node is needed at the respective position.
4.5.2 Model-to-Text Transformation: Meta-Model to Syntax Model

A syntax model, which is complementary to the AMM as well as the MMM, has been developed for the code generation part. For each entity of the MMM, a set of correspondences is found and represented by a set of syntax nodes. According to the approach of modeling data as aggregates (Fowler 2010), the Model will serve as an aggregate root. The generation of a new node is further refined and takes place by using a certain interface offered by the Roslyn compiler. In addition to the non-modifiable syntax nodes, an abstracted object is then generated, which comprises the information retrievable from the graph and explicitly manages the code to be generated. A concrete agent type from the AMM for instance is available as an AgentNode and allows the appending of any syntax nodes, which are necessary for the code-specific description. With the ConstructorNode, a generic node is built, including the complete internal code. It also covers querying the layer types that should be used, to inject dependent types into the currently viewed agent.

Due to the direct dependency between the AMM and the MMM, there is a set of predefined nodes that always has to be generated to run the simulation model. Every node contains a partial extension to the original model and includes the same relationships plus their cardinalities, to resolve the name and scope and to build cross-references within the code. As soon as all viewpoints are processed in the workflow, all nodes of the syntax model are collected and grouped together within a common namespace, given by the name of the model. This format, including all agent and layer classes, will then be compiled.

4.5.3 Model-to-Text Transformation: Template and Visitor-Based Code Generation

Two different strategies are pursued to support code generation in detail and to build the actual syntax nodes. In template-based code generation (TBCG), an internal parser is triggered, which fills the placeholders in a parameterizable template. The properties of all objects are coupled with the placeholder names in the template. The result is a complex string that is read with a partial parser to generate the syntax tree, which is then added to the syntax model. In addition to that, a visitor-based principle was used, where the code is generated while traversing the original textual representation. All syntax nodes are generated...
programmatically by the Roslyn API bit by bit. However, the TBCG is more commonly used, because it allows a complex generation based on a created template file. Due to the size of the simulation models, the creation of a complete template quickly becomes too complex (Herrington 2003). For this reason, the generation approach is only used to create different leaf nodes in the syntax model. That is why it is more preferable to modify this approach for systems that are processing a lot of data to accomplish different sets of tasks. Listing 2 shows a sample template format of the code excerpt that is generated for the `ResultNode`.

```csharp
<$each [state in agent.States]$>
<$when [state.Observe]$>
AgentData['<$[state.Name]$>'] = <$[state.FieldName]$>
</$when$>
</$each$>

Listing 2: Conditional template example to support code generation in the MARS framework.

The placeholder will be filled by a reflection call on the input data object. Then the template is parsed into a uniform abstract syntax tree and then gets traversed, looking for placeholders via depth first search. A placeholder is assigned with a value by a reflection call on the input. However, such calls are much slower than a direct call, which results in a performance overhead. That is why the template engine provides a tactic for minimizing the reflection usage. It is based on the runtime type generation, in which the engine generates a proxy type per external placeholder. The call will then be redirected through this proxy, which is now filled with the initial data. The proxy directly calls members of the corresponding placeholder, thereby the assignment takes place in a uniform way with no reflection involved. Additionally the proxy is lazily initialized and will later be reused again.

5 DISCUSSION

With this paper we introduced the model driven development for simulation models with a propagated AMM and MMM as well as a modeling process to instantiate them. In addition to structured formation of simulation models, domain-specific tools can be developed based on these meta-models. By that, the MMM adds an interim stage to the process of modeling and implementation of agent-based simulations. This lets domain experts view the meta-model in an abstracted way, provided by the respective tool. With the adapted MDA, we could close the gap between modeling and creation of multi-agent simulations by adding an extra GSM. This translates the meta-model instance to the respective MARS specific-code, adapted to the internal MARS concepts and services.

As to the rest, the MARS DSL has not unfolded its full power yet. On one hand, the expressiveness of the C# language used in MARS is missing. On the other hand, the internal agent logic has still to be developed in pure C# code. The disclosure of these technical details can be improved by refining the agent logic, for example, integrating it in the form of an activity flow, with predefined actions for corresponding preselected agent types. These will then be integrated into the propagated modeling workflow. Since there are no rules for performance optimization yet, this will be another task in order to fit into the MARS massive scale environment.

6 CONCLUSION

Using the model-driven development as a central approach for building simulation models, we have developed different components and concepts that can be seamlessly integrated into a modeling workflow. Build-
ing on the structured formation of simulation models (AMM & MMM), a set of domain-specific modeling tools are in development, including a graphical editor as well as a textual tool, the MARS Base-DSL. These tools are designed with the abilities of the MARS system in mind and will form the modeling environment in conjunction with the realized code generator.

REFERENCES


**AUTHOR BIOGRAPHIES**

**DANIEL GLAKE** is a master’s degree candidate at the University of Applied Sciences in Hamburg, Germany. He got his bachelor’s degree in 2015 and continued with the join of the MARS group. His research focuses on model driven development, which includes domain specific languages and impacts of UML-like notations. His email address is daniel.glake@haw-hamburg.de.

**JULIUS WEYL** is a master’s student in computer science at the University of Applied Sciences in Hamburg, Germany. After graduating at the University of Applied Sciences Worms (Germany) he relocated to Hamburg to pursue his research on agent-based models and simulations. His email address is julius.weyl@haw-hamburg.de.

**CAROLIN DOHMEN** is a bachelor’s degree candidate at Hamburg University of Applied Sciences. She is a member of the MARS group, where her main interests are modeling complex systems and developing multi-agent models with a focus on ecological and sociological simulations. Her email address is carolin.dohmen@haw-hamburg.de.

**CHRISTIAN HÜNING** holds a master’s degree in computer science and works as a research associate at the computer science department at the University of Applied Sciences in Hamburg, Germany. His research is centered around distributed systems, virtualization and cloud technologies with a focus on high-performance, large-scale and distributed multi-agent systems. His email address is christian.huening@haw-hamburg.de.

**THOMAS CLEMEN** holds a position as a full professor at the University of Applied Sciences in Hamburg, Germany. His teaching activities predominantly cover topics within information management and data science, whereas he is focusing on modeling and simulation of dynamic, complex and self-organizing systems in his interdisciplinary research. His email address is thomas.clemen@haw-hamburg.de.