Technische Universität Berlin

Forschungsberichte der Fakultät IV – Elektrotechnik und Informatik

Generation of Simulation Views for Domain Specic Modeling Languages based on the Eclipse Modeling Framework

Claudia Ermel, Enrico Biermann, Karsten Ehrig
Jonas Hurrelmann

Bericht-Nr. 2009 – 17
ISSN 1436-9915

Generation of Simulation Views for Domain Specific Modeling Languages based on the Eclipse Modeling Framework

Claudia Ermel, Enrico Biermann, Jonas Hurrelmann, Karsten Ehrig

Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin, Germany
tigerprj@tu-berlin.de

Abstract. The generation of tools for domain specific modeling languages (DSMLs) is a key issue in model-driven development. Various tools already support the generation of domain-specific visual editors from models, but tool generation for visual behavior modeling languages is not yet supported in a satisfactory way. In this paper we propose a generic approach to specify DSML environments visually by models and transformation rules based on the Eclipse Modeling Framework (EMF). Editing rules define the behavior of generated visual editors, whereas simulation rules describe a model's operational semantics. From a DSML definition (model and transformation rules), an Eclipse plug-in is generated, implementing a visual DSML environment including an editor and (possibly multiple) simulators for different simulation views on the model. We present the basic components of Tiger2, our EMF-based generation environment, and demonstrate the environment generation process for a small DSML modeling the behavior of ants in an ant hill.

1 Introduction

Domain specific modeling languages (DSML) are of growing importance for software engineering, and the rapid development of DSML tools is a key issue in model-driven development. Meta-tools including MetaEdit+ [28], DiaGen [24], AToM^3 [21], GME [22], Marama [17] and DSL Tools [23] have been developed to support rapid specification and generation of DSML tools. Moreover, the Eclipse Modeling Framework EMF [11] has recently come to be a quasi-standard for meta-modeling in practice, consisting of an implementation of core concepts based on MOF. The above

mentioned meta-tools do not yet take the EMF format for meta-modeling into account, which poses difficulties when existing EMF models serve as basis for tool generation. A notable exception is the EMF-based editor generator GMF [10] which is widely used in ECLIPSE projects on model-driven software development, such as the UML2Tools subproject [13]. In contrast to editor generation, the generation of tools for visual behavior modeling languages based on EMF is not yet supported in a satisfactory way. Visual behavior models are the basis for model simulation with the purpose to validate the model behavior with respect to its requirements. In this paper we propose a generic approach to specify behavior-modeling environments by EMF models and EMF model transformation based on graph transformation rules. Graph transformation has been investigated as a fundamental concept for programming, specification, concurrency, distribution, visual modeling and model transformation [14,15].

In our modeling environment, a set of EMF transformation rules called editing rules define the editing commands of the generated visual editor, i.e. the model syntax; on the other hand, a set of simulation rules describe a model's operational semantics. For automatic simulation, rule application is controlled by activity diagrams, where simple activities denote rule applications. From a DSML definition (EMF model, view definitions and EMF transformation rules), an ECLIPSE plug-in is generated, implementing a visual DSML environment including an editor and (possibly multiple) simulation views on the model.

The structure of this paper is as follows: After reviewing graph and EMF transformation concepts in Sect. 2, we introduce our running example, a small DSML modeling the behavior of ants in an ant hill, in Sect. 3. Sect. 4 defines the EMF models used for DSML specification, and Sect. 5 presents TIGER2 [29], our generation environment based on EMF and EMF model transformation. We give an outlook to future work and conclude the paper in Sect. 7.

2 EMF Transformation based on Graph Transformation Concepts

For editing and simulation we use a rule based approach based on algebraic graph transformation concepts [14]. In this section, we introduce the main notions of modeling by graph transformation (Sect. 2.1) and extend these notions to model EMF model transformation based on graph transformation in Sect. 2.2.

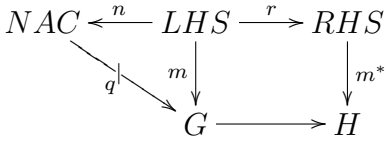
2.1 Modeling by Graph Transformation

A domain-specific visual language (VL) is modeled by a type graph defining the underlying visual alphabet, i.e. the symbols (node types) and relations (edge types)

which are available. Sentences or diagrams of the VL are given by graphs typed over (i.e. conforming to) the type graph. Such a VL type graph (which may also contain multiplicities and inheritance arcs) corresponds closely to a meta-model. Node types may be attributed by attribute types.

On the basis of a type graph defining a VL, and instance graphs typed over this type graph representing different states of a model, step-wise simulation is now described by graph transformation between these states. The main idea of graph transformation is the rule-based modification of graphs where each application of a graph transformation rule leads to a graph transformation step. The core of a graph transformation rule ($LHS \xrightarrow{r} RHS$) is a pair of graphs (LHS, RHS), called left-hand side and right-hand side, and an injective (partial) graph morphism $r : LHS \to RHS$. A graph morphism consists of structure-preserving mappings from nodes in LHS to nodes in RHS, such that for an edge from node n_1 to node n_2 in LHS which is preserved by the rule, we have a corresponding edge from node $r(n_1)$ to $r(n_2)$ in RHS. In our approach, all graph morphisms are injective, i.e. they do not merge elements. Applying the rule ($LHS \xrightarrow{r} RHS$) means to find a match of LHS in the source graph and to replace this matched part in the source graph by the corresponding RHS, thus transforming the source graph into the target graph of the graph transformation. Intuitively, the application of rule r to graph G via a match m from LHS to G deletes the image $m(LHS)$ from G and replaces it by a copy of the right-hand side $m^*(RHS)$. Note that a rule may only be applied if the so-called *gluing condition* is satisfied, i.e. the deletion step must not leave *dangling edges*.

Definition 1. *Graph Transformation Let ($LHS \xrightarrow{r} RHS$) be a typed graph transformation rule and G a typed graph with a typed graph morphism $LHS \xrightarrow{m} G$, called match. A* graph transformation step $G \overset{r,m}{\Longrightarrow} H$ *from G to a typed graph H via rule p, match m, and co-match m^* is shown in the diagram below. The rule r may be extended by a set of* negative application conditions *(NACs) [18,14]. A match $LHS \xrightarrow{m} G$ satisfies a NAC with the injective NAC morphism $LHS \xrightarrow{n} NAC$, if there is no graph morphism $NAC \xrightarrow{q} G$ with $q \circ n = m$.*



An example of a rule application is shown in Fig. 1, where mappings are indicated by corresponding numbers in the graphs. The LHS matches its Field *node to* Field *node 2 in G. Hence, the node of type* Ant *in H, which is created by the rule, is linked to* Field *2 and to node* AntWorld.

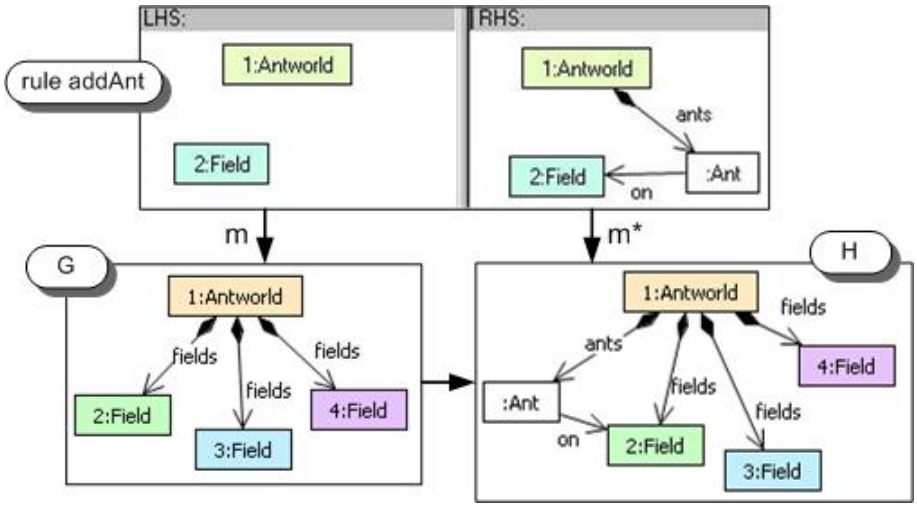


Fig. 1. Rule application example

A sequence $G_0 \Rightarrow G_1 \Rightarrow .. \Rightarrow G_n$ *of graph transformation steps is called* graph transformation*, denoted as* $G_0 \overset{*}{\Rightarrow} G_n$*.*

A rule may be extended by input parameters, i.e. variables used to compute new attribute values for nodes in the right-hand side. When the rule is applied, the input parameters have to be bound to concrete values.

2.2 From Graph to EMF Transformation

The Eclipse Modeling Framework EMF [11] is a modeling and code generation facility for building tools and other applications based on a structured data model. From a model specification described by a class diagram, EMF provides tools and runtime support to produce a set of Java classes for the model, a set of adapter classes that enable viewing and command-based editing of the model, and a basic (tree-based) editor. EMF provides the foundation for interoperability with other EMF-based tools, e.g. OCL checkers.

Although EMF provides basic operations for modifying EMF based models, it is still difficult to define more complex operations on these models. Our approach uses the recently developed Eclipse plug-in [30,7] supporting modeling and code generation for EMF model transformations, based on structured data models and graph transformation concepts. The conceptual differences between modeling based on typed, attributed graphs and object-based modeling as performed by EMF are shown in Table 1.

Table 1. Mapping EMF notions to graph terminology

EMF notion	Graph term
Model	Type graph with attribution, inheritance, multiplicities. Edges can be marked as containments.
Model instance	Typed, attributed graph with containment edges
Class	Node in type graph
Object	Node in typed graph
Association	Edge in type graph (with possible multiplicities or containment mark)
Reference	Edge in typed graph that must not violate certain multiplicity and containment constraints.

Classes in an EMF model correspond to nodes in a typing graph. Associations between classes can be seen as edges in a type graph. Generalizations and multiplicity constraints of association ends can also be defined in the type graph. Objects as instantiations of classes of an EMF model are comparable to nodes in a graph which is typed by a type graph. Objects can be linked to each other by setting reference values. Such references correspond to edges in a typed attributed graph.

Usually, EMF models have containment constraints in addition, which do not occur in plain graph transformation. Containment relations, i.e. aggregations, define an ownership relation between objects. Thereby, they induce a tree structure in model instantiations. In MOF and EMF, this tree structure is further used to implement a mapping to XML, known as XMI (XML Meta data Interchange) [31]. Containment implies a few constraints for model instantiations that must be ensured at run-time. As semantical constraints for containment edges, the MOF specification [26] states the following:

- *"An object may have at most one container."*
- *"Cyclic containment is invalid."*

EMF provides full implementations of instance models. These implementations always ensure these constraints.

In [8], containment constraints of EMF model transformations are translated to a special kind of graph transformation rules such that their application leads to consistent transformation results only, i.e. they must not delete contained objects without deleting their containment relations as well, and they must not generate objects without relating them to precisely one container. Moreover, containment cycles must not be produced by rule applications. In [8] we formally define well-formed graph transformation rules with containment (called *EMF transformation rules* from now on) and show that their application does not violate the EMF containment con-

straints stated above. Hence, in this paper we will use EMF transformation rules only and thus can be sure that we always have valid EMF instance models. Note that EMF transformation rule applications change an EMF model instance *in-place*, i.e. the model instance is modified directly, without copying it before. Moreover, we can export EMF transformation rules to the graph transformation analyzer AGG [4] in order to verify important transformation properties such as dependencies and confluence of rule applications [14].

3 Running Example: Ant World

The AntWorld simulation is a case study designed as benchmark for the comparison of graph transformation tools at the *Graph-Based Tools* Workshop GraBaTs 2008 [32]. The AntWorld simulation consists of an ant hill sitting in the middle of a large area. The ants are moving around searching for food. If an ant finds food, it brings the food home to its ant hill in order to grow new ants. On its way home, the ant drops pheromones marking the path to the food reservoir. If an ant without food leaves the hill or if a searching ant hits a pheromone mark, the ant follows the pheromone path to the food. This behavior already results in the well-known ant trails. The area in which the ants move is modeled by a grid of nodes. In order to enable the ants to go home on a straight path, if they have found some food, the area grid looks like a spider's web with the ant hill in its center, see Fig. 2.

The AntWorld simulation works in rounds. Within each round, each ant does one move. The ant behavior depends on the following modes:

- If the ant has no food and is on a field with food, it takes one piece of food and enters the food carrying mode. It may still move within the current round.
- If the ant carries some food, it follows the links towards the 'inner' circle. During its way home, on each visited grid node (including the *food* node, the ant drops 1024 parts of pheromones. This guides other ants to the food place.
- If ant with food is on the hill node, it drops the food and enters the search mode. It may leave the hill within the same round.
- An ant without food is in search mode. The ant checks the neighbor node(s) of the next outer circle for pheromones. If there are neighbor nodes on the next outer circle with more than 9 parts of pheromones, the ant chooses one of these fields, randomly.
- If the ant is in search mode and no outer neighbor has sufficient pheromones, the ant moves to any neighbor field, randomly. However, an ant without food shall not enter the ant hill.

Initially, the area grid consists only of the hill and the first two circles. The hill contains eight ants. No food is provided on the grid. Whenever during one round

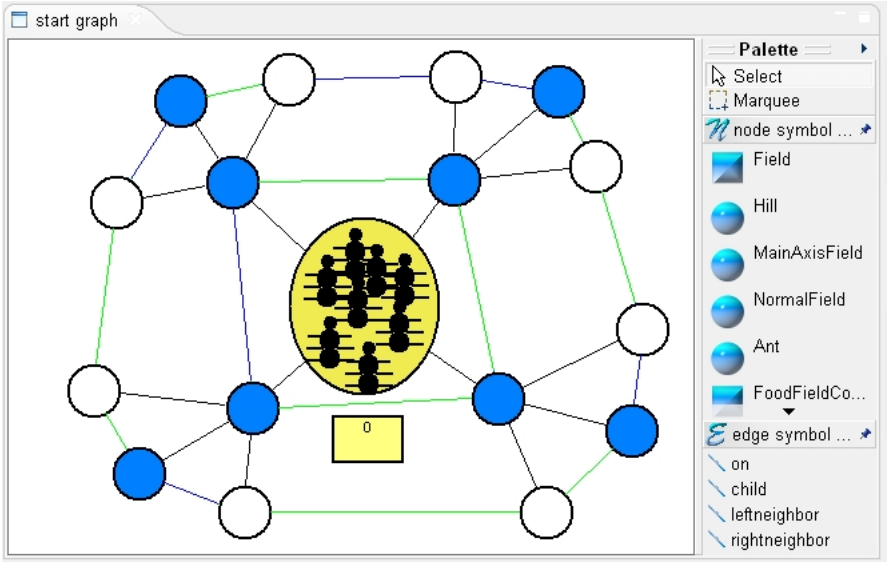


Fig. 2. AntWorld example

an ant enters the currently outmost circle (i.e. the border of the yet known area), a new circle of nodes is created. During the creation of this next circle, every 10th node shall carry 100 parts of food. After each round, the pheromones evaporate. This is needed in order to erase an old ant trail once the food has drained. After each round, the hill consumes the food brought to it and it creates one new ant per delivered food part.

3.1 The Ant World Meta Model

Fig. 3 shows the EMF model we used for our rules. All objects in our ant world are contained in a class called *AntWorld*. We model *Ants, Fields, Food* and *Pheromones* as classes. *Fields* are further divided into their specific roles like *Hill, Normal* fields and *Exit* fields which form the main four axis in the grid starting from the *Hill*.

3.2 Editing Rules

For building the start system for our simulation shown in Fig. 2, we provide some basic editing rules (see Fig. 4). Rule *addHill* is applicable only once (modeled by a suitable NAC). Rules *addNormal* and *addMainField* produce new fields, linked by applying rule *linkFields*. Ants are produced by rule *addAnt* (see Fig. 1). We do not provide editing rules for food and pheromones, since they are produced during simulation only.

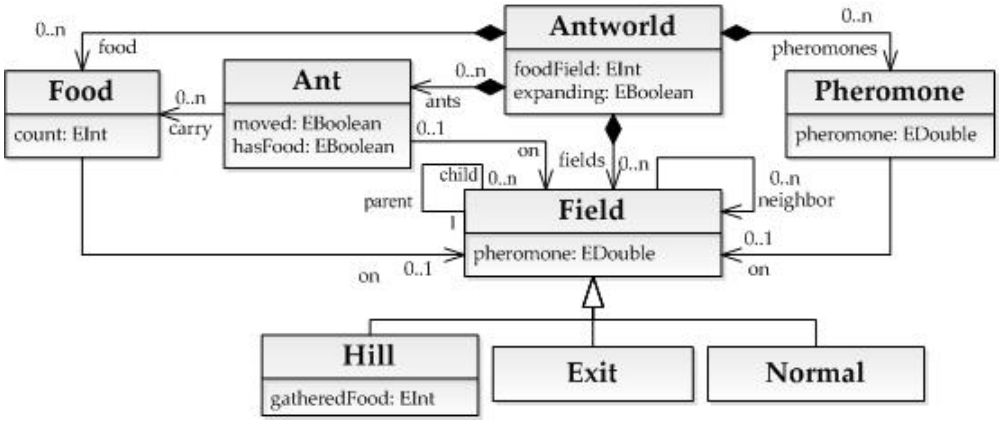


Fig. 3. EMF Model for the Ant World VL

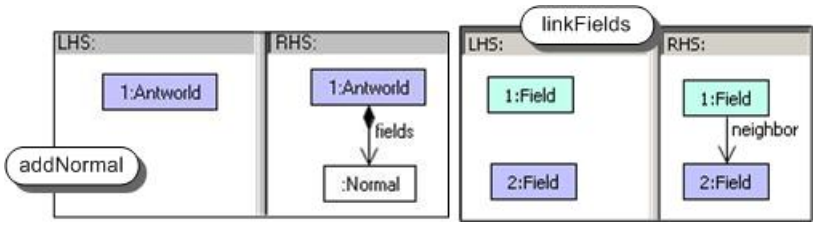


Fig. 4. Basic editing rules

3.3 Simulation Rules

We have six rules for ant movement (see Fig. 5). In search mode, an ant can move up or down or along a circle as long as there are no pheromones on any child field from the ant's position. If there are pheromones, rule *followTrail* is used instead of the other movement rules to guide the ant to a food place. If the ant already picked up some food, one of two *carryFood* rules is used, either refreshing an already existing pheromone trail while the ant moves towards the hill; or creating a new pheromone path.

AntWorld management consists of expanding the world if an ant moves towards the outer edge, create new food supplies on the outer ring, and decay or remove old pheromone trails. The world is expanded by three rules: *expandExit* (see Fig. 6), *expandNormal* and *makeCircle* to connect newly generated outer nodes (not depicted). While expanding the world, the number of newly created fields is stored and afterwards new food supplies (100 parts) are created on the outer ring by rule *createFood*. At last, each pheromone occurrence is reduced by factor 0.95 (rule *decayPhero*). Pheromone amounts below 9 are dropped completely.

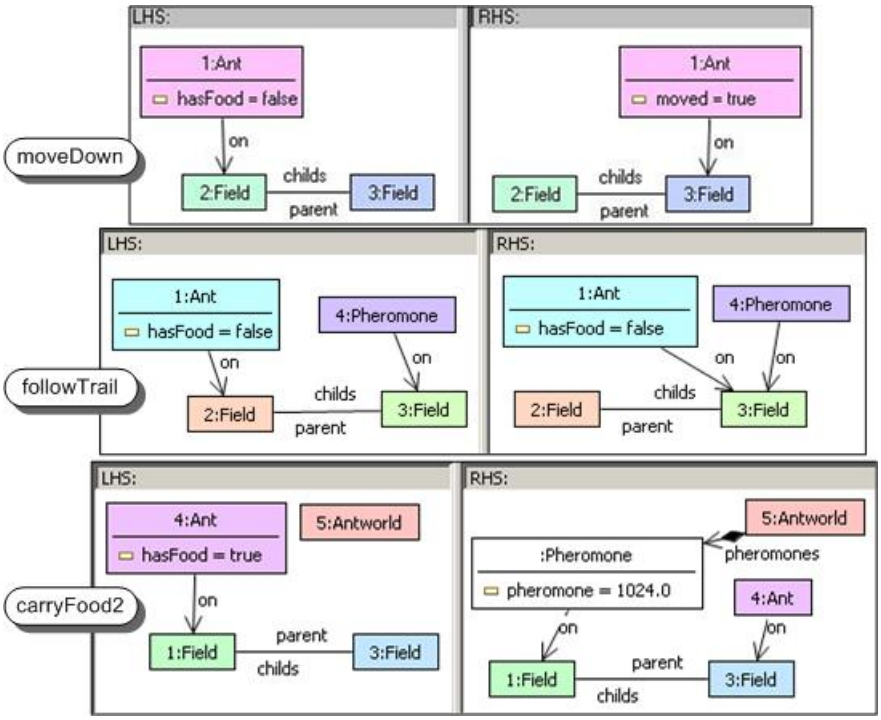


Fig. 5. Rules for ant movement

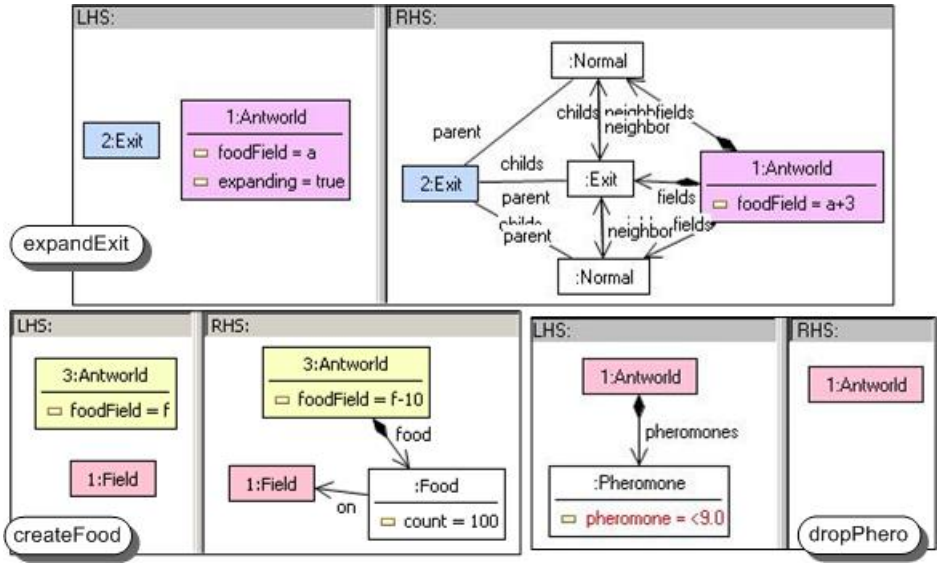


Fig. 6. Rules for AntWorld management

If an ant comes to a field containing food, it picks up one part of the food (rule *pickFood*). On the hill, a food-carrying ant drops the food (rule *dropFood*) and creates a new ant using the *createAnt* rule.

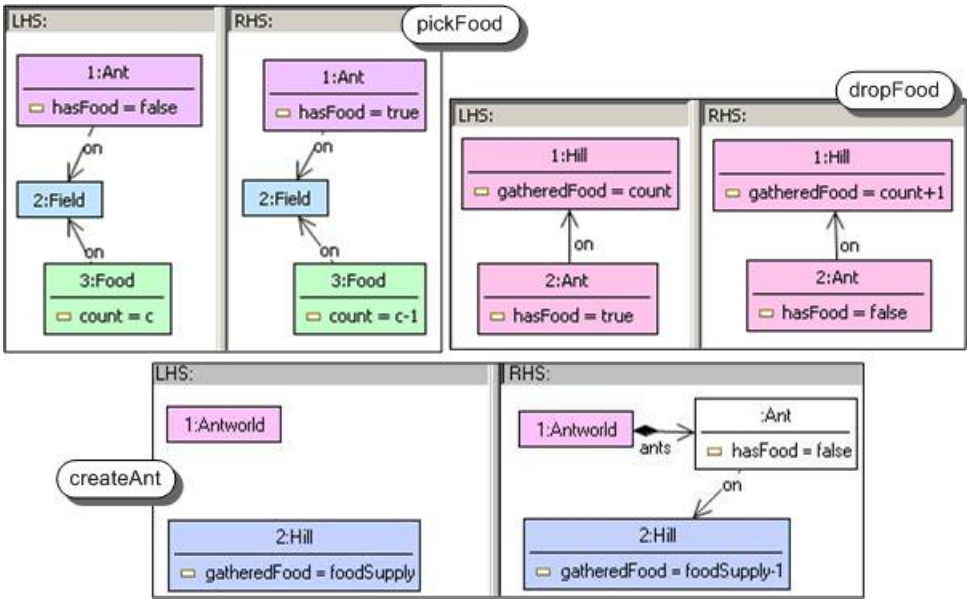


Fig. 7. Rules for food transportation

3.4 Control Structure Model

For the rule control flow of simulation rules, we define activity diagrams for the different task groups as defined in the previous section.
In Fig. 8, the main activity diagram *PlayRound* models that each round consists of the phases ant movement, food transportation and ant creation, and AntWorld management (possibly expanding the grid), Each of the hierarchical activities in the *PlayRound* activity diagram stands for a refined activity diagram controlling the rule applications of the respective phase.

In the activity diagram *MoveAnts* for ant movement (Fig. 9), we allow each ant exactly one rule application. If the ant cannot carry food or follow a trail, it is in search mode and will choose one of the possible matches to move up, down or to one of its neighbor fields.

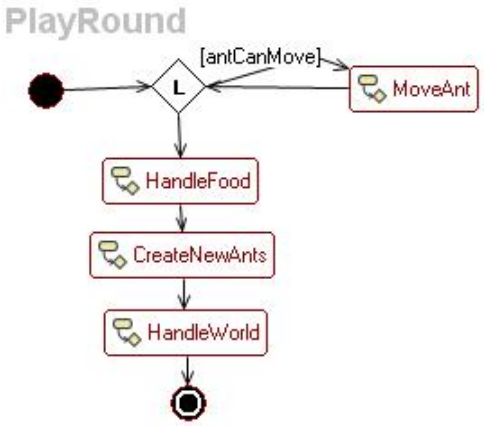


Fig. 8. Main activity diagram *PlayRound*

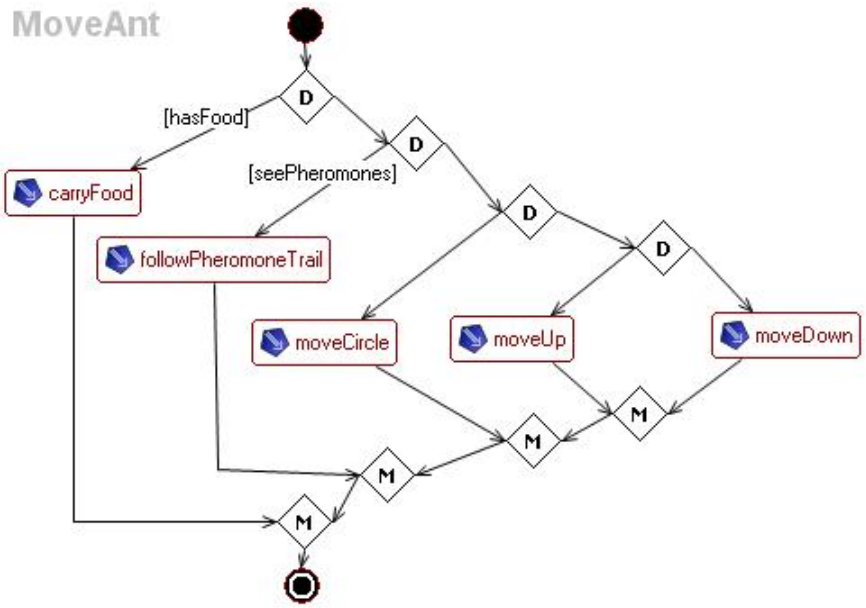


Fig. 9. Activity diagram *MoveAnts*

The rules controlled by the *HandleWorld* activity diagram (Fig. 10) expand the AntWorld if there is an ant on the outmost ring. Food is created on the newly created outer ring and pheromone trails are decayed and removed.

After all ants have moved they can still pick up or drop food on the hill. In the last case, new baby ants are created (Fig. 11).

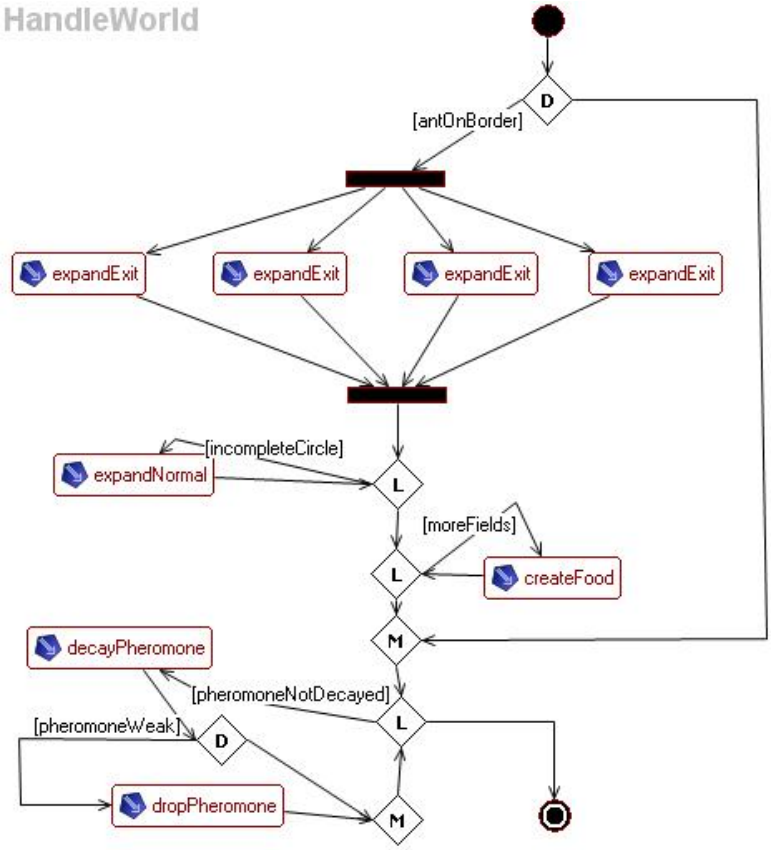


Fig. 10. Activity diagram *HandleWorld*

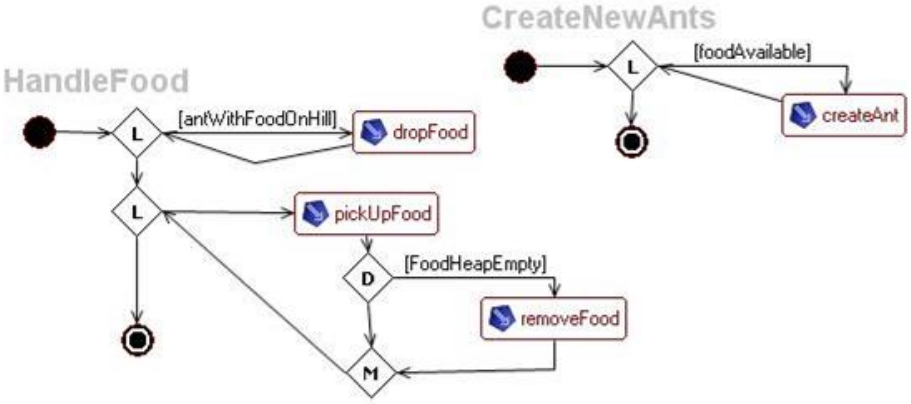


Fig. 11. Diagrams *HandleFood* and *CreateNewAnts*

4 Modeling DSML Environments

In this section, we describe the EMF models underlying our approach to DSML specification and tool generation. As demonstrated in the last section, we aim at a VL specification (Fig. 12) based on EMF, consisting of the parts

- *Language*: An EMF model for the VL, EMF transformation rules for editing operations and for simulation steps, and a set of activity diagrams specifying the application of simulation rules.
- *Visualization*: Mappings from EMF model elements to GEF figures and connections which are supposed to realize their visual appearance in the generated environment.
- *Views*: Mappings from EMF model elements to views for different visualizations,

Moreover, a VL specification also contains Session information. Strictly they are not part of the pure VLSpec, but are used to store the session in the same model instance and by that in the same file.

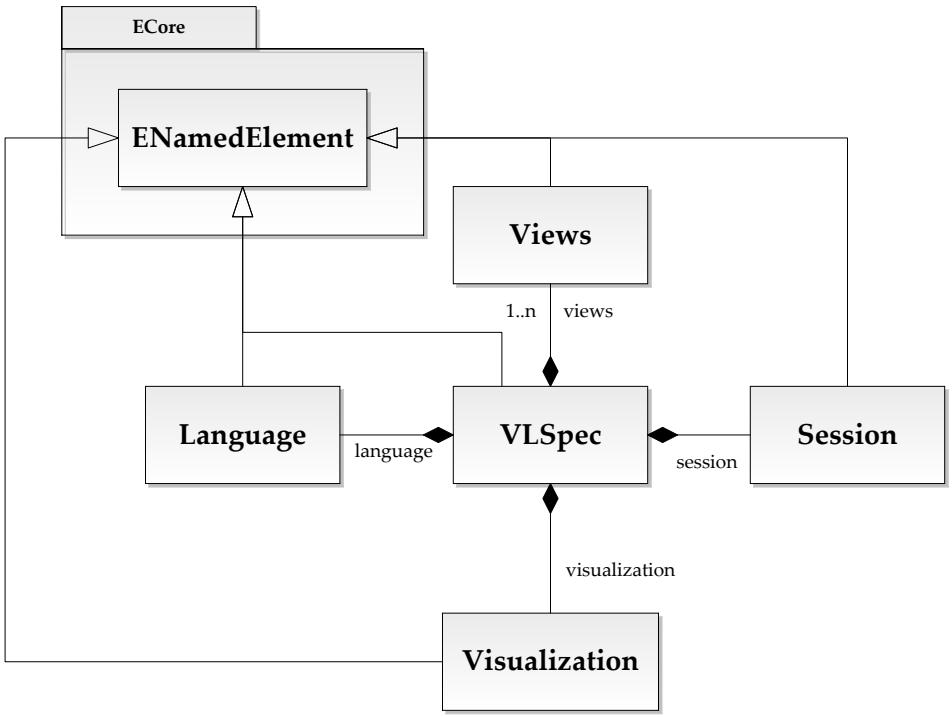


Fig. 12. Core model of a VL specification

4.1 Language

An overview of the main Language components is shown in Fig. 13.

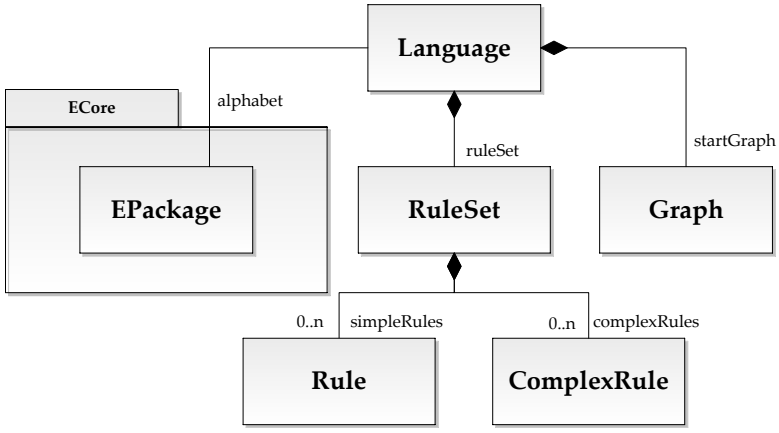


Fig. 13. Language model

A language consists of syntax and semantics. We define the syntax by an alphabet (the EMF model) and editing rules, and give the semantics in form of operational simulation rules, controlled by activity diagrams. In order to define the *alphabet* of the *Language*, an EPackage (EMF model) is used. The language does not hold the EPackage as containment but as pure association and thus the alphabet may be loaded from and can be stored to an external file. This design decision has the purpose to ease the creation and modification of the model with external tools.

For our AntWorld example, Fig. 3 is showing the EMF model representing the VL's alphabet.

Besides the *alphabet*, the *Language* contains the *Rule* and *Graph* definition (see Fig. 14).

A *Graph* consists of *Nodes* and *Edges*, which in turn are a mapping to the elements of the alphabet. Each *Node* has a set of incoming and outgoing *Edges* to form the graph structure. As the *Language* defines an EMF model that is used by having an EPackage as its alphabet, each *Node* has to be assigned to a corresponding EClass and each *Edge* has to be assigned to a corresponding EReference. To complete the graph grammar, a start graph of the language is defined using the *Graph* structure. As AntWorld start graph, the graph shown in Fig. 2 may be defined. Note that Fig. 2 shows the start graph in its concrete visualization.

Each *Rule* consists of an LHS, an RHS and a number of NACs which are all *Graphs*. *Mappings* are used to define morphism between *Graphs*. To keep the model

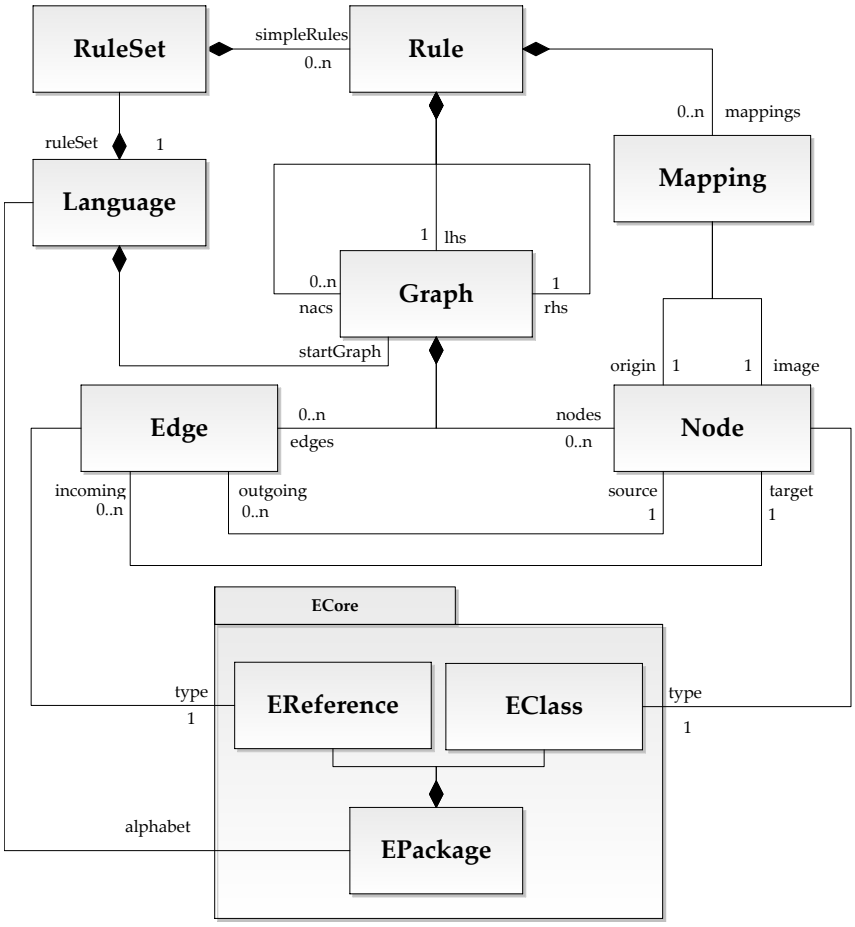


Fig. 14. Rules and Graphs of a Language

simple and efficient, there are no internal constraints to make every mapping a valid graph morphism (see [14]). Checking whether a morphism is valid and whether it is actually part of a graph transformation has to be done algorithmically. A valid *Rule* can now be used to perform basic editing operations of the language such as creation and deletion rules, or atomic simulation steps. Naturally, more complex EMF transformation rules can be defined allowing for more sophisticated operations, e.g. to realize model transformations (see [16,6,7]).

Sample rules for the AntWorld have been presented in Sect. 3. The instance model corresponding to the rule *addNormal* in Fig. 4 is shown in Fig. 15.

In order to control the application of simulation rules, activity diagrams are used (see Sect. 3.4 for examples). The formal background for refining activities by graph transformation rules is given in [19]. The main idea is to refine a *SimpleActivity* via a

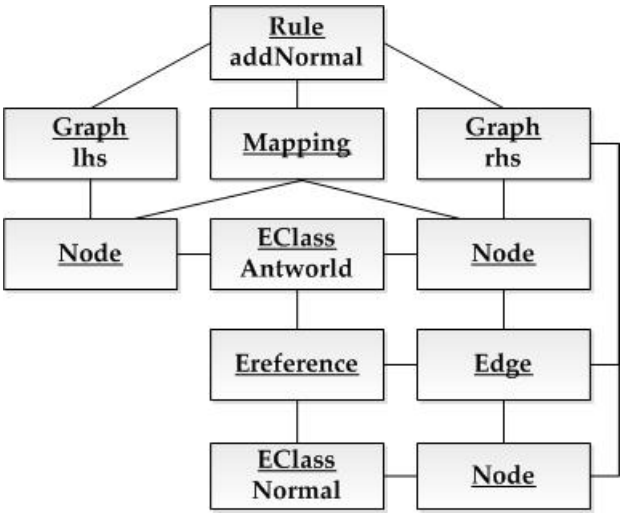


Fig. 15. Instance model for rule *addNormal*

rule that is applied when the activity is executed. *Activities* are linked by *Next* edges (see Fig. 16). To control the flow, *DecisionActivities* and *LoopActivities* with graph constraints at their outgoing *Next* edges are used. Graph constraints are are of the form $P \rightarrow C$, where P is the premise and C is its conclusion of the graph constraint. A step in the activity diagram can only be performed if the corresponding graph constraint is fulfilled, i.e. if in the case that P is found in the current graph, then also the conclusion C is found there as an extension of P.

4.2 Visualization

To achieve the visual representation of the language elements (see Fig. 17), different kind of shapes can be defined. *Figures* are used to represent *Nodes* in a *Graph* and *Connections* are used to visualize *Edges*. They directly relate to their counterparts in Draw2d, the graphic toolkit that comes with the Eclipse Graphical Editing Framework GEF [12].

4.3 Views

With the visual elements of a *Language* defined, a mapping assigns each symbol of the alphabet used, to its visual representation. Each *View* provides a distinct mapping of the model elements, which allows subsequently having different visual representations of the same model within a single editor. At least one view has to be defined in order to have a working editor. Since the alphabet is an EPackage, there

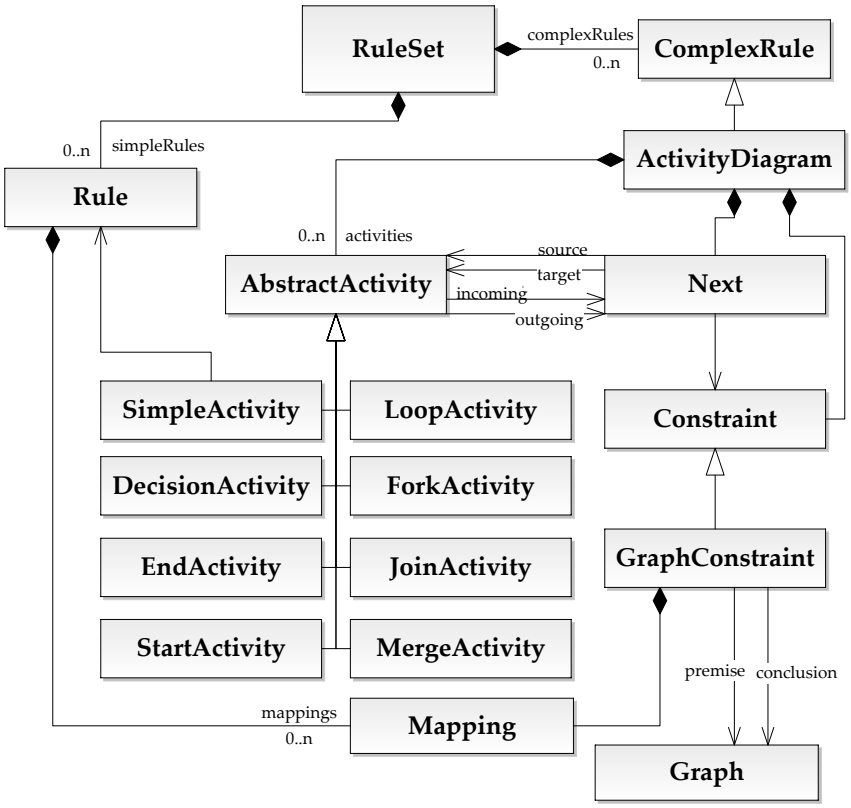


Fig. 16. Simulation defined by activity diagrams

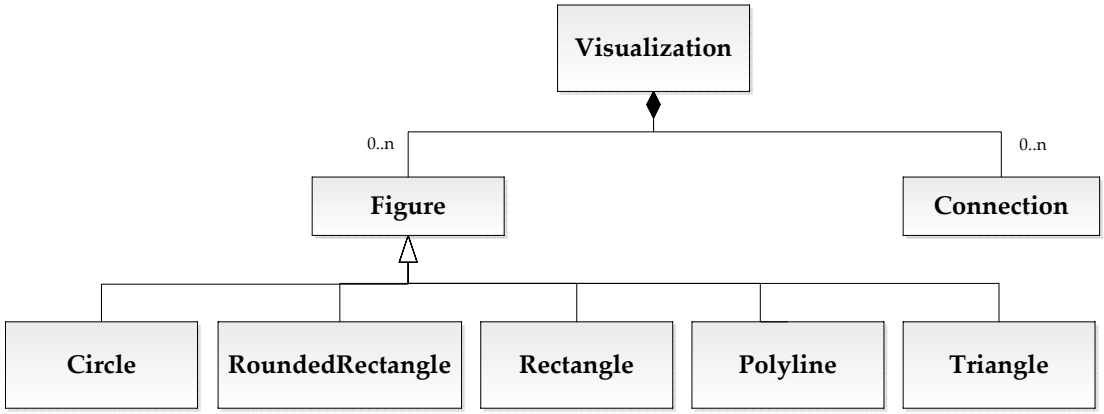


Fig. 17. Visualization of Language Elements

is a mapping between an *EClass* to one or more *Figures* and a mapping between an *EReference* to one or more *Connections* (see Fig. 18).

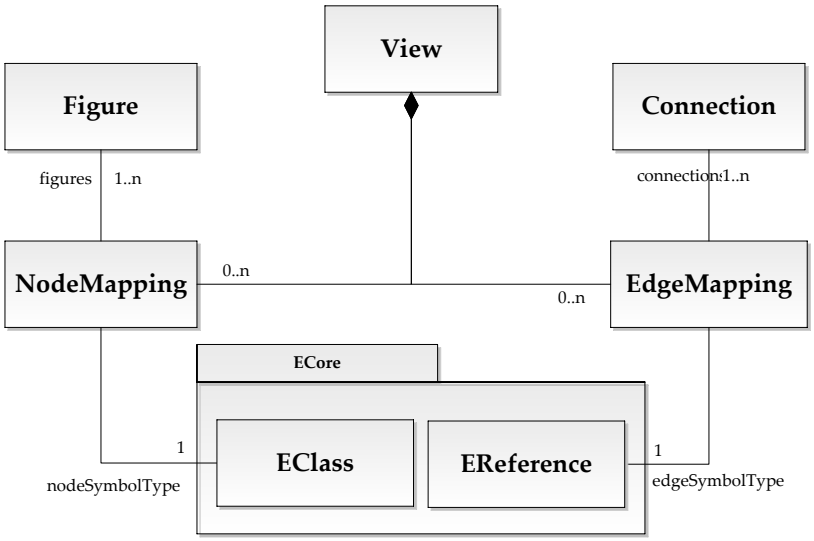


Fig. 18. Views for Language Elements

For our AntWorld view, model element *Ant* is mapped to an (invisible) *Rectangle* figure containing three *Circles* (the body parts) and three *Polylines* (the six legs), see Fig. 2.

5 The Generation Environment

A VL specification based on a meta model in combination with a rule-based specification of editor commands and refined activity diagrams for simulation is used in our generation environment Tiger 2 *(Transformation-based Generation of Environments)* to generate a corresponding visual modeling environment for the specified visual DSML.

Tiger 2 combines precise VL specification techniques using graph transformation concepts with the meta-modeling based on EMF and sophisticated graphical editor development features offered by GEF. Whereas our previous Tiger tool [29] was an editor generator only, in Tiger 2 also controlled units of simulation rules can be specified, and automatic simulation can be visualized in different views at the same time.

The architecture of Tiger 2 consists of the three components *Designer* (where the modeler defines the DSML), *Generator* (translating a VL Specification to Java code) and the *Generated Tool Environment* (an Eclipse plug-in containing a visual editor and simulation views for the specified VL).

5.1 Designer

In the current state, the *Designer* is mainly tree-based, except for the definition of rules and activity diagrams, where visual editors exist already (see the screenshots in Sect. 3). Future versions will allow VL designers the visual definition of visualizations and views, as well. Views are created by adding them via a right-click on the *VLSpec* tree node. For adding the mapping, the corresponding EClass or EReference respectively has to be determined.

5.2 Generator

Generating Code from the EMF Model EMF models can be directly translated to Java code, using the Java Emitter Templates JET [9]. The generated code can be seen as a run-time data model of the structure defined in the class diagrams. The code generation provides a complete implementation that manages the life cycle of objects (create, delete, set attributes etc.), while ensuring multiplicity and containment constraints. Further, a persistence API is provided, implementing load / save operations for model instances. The standard format for EMF models / model instances is XML / XMI. The code generated by EMF can be extended at any point. An EMF model is translated into a EPackage. Such a package contains all information defined in the model. EMF Classes are translated to Java classes that contain all attributes and references defined in the model. To each package there is a run-time factory, which is used for creating objects. Furthermore, an EPackage is identified using a unique namespace URI, also defined in the model.

As first step of the generation process, the GeneratorModel has to be created, as it contains the settings for the code generator that are not stored in the EMF model. The first important setting is the output path, where the project resides and the generated code should be created. Other basic settings, like enabling the code formatting are also done here.

```
protected void initalizeGenerator() -
genModel = GenModelFactory.eINSTANCE.createGenModel();
genModel.setModelDirectory(targetPath);
genModel.setCanGenerate(true);
genModel.setCodeFormatting(true);
genModel.setForceOverwrite(true);   "
```

The GeneratorModel has to be initialized with an actual EMF model via its EPackage:

```
public void setModel(EPackage ePackage)-
genModel.initialize(Collections.singleton(ePackage));   "
```

Finally the Generator can be invoked by setting the GeneratorModel as input and attaching a progress monitor.

```
public void startGenerator()-
Generator generator = new Generator();
generator.setInput(genModel);
Monitor monitor = CodeGenUtil.EclipseUtil.createMonitor(progressMonitor,1);
generator.generate(genModel,GenBaseGeneratorAdapter.MODEL_PROJECT_TYPE,
    monitor);  "
```

Code for EMF Transformation Rules The transformation rules created by the TIGER Designer are copied to the generated editor project as an XMI file. The structure of the XMI file is similar to that shown in Fig. 14. When the generated editor is opened, the rules are loaded as well. The execution of the rules is handled by the EMF TIGER transformation engine which initializes an instance of the class GenericRule with the content of a rule.

```
GenericRule genericRule = new GenericRule(tigerEngine, rule);
```

After a GenericRule has been initialized in that way it is possible to call the execute()-method which will search for a valid occurence of the LHS of the rule in an EMF instance and also apply the model changes that are specified by the rule. When searching for a match, NACs and attribute conditions are respected as well.

```
public boolean execute() -
 if (!isExecuted) -
  match = findMatch();
  if (match == null)-return false;"
        else -comatch = generateModelChanges();"
  modelChange.applyChanges();
  isExecuted = true;
  return true; "
 return false;   "
```

Model changes are performed according to the rule, for example adding a new edge:

```
for (Edge edge: rule.getRhs().getEdges())-
  if (!ModelHelper.isEdgeMapped(rule, edge)) -
    modelChange.addObjectChange(comatchNodeMapping.get(edge.getSource()),
        edge.getType(), comatchNodeMapping.get(edge.getTarget())); "  "
```

This part of the method generateModelChanges() prepares a model change for every edge that was not yet present in the rule's LHS. Afterwards the changes are executed by modelChange.applyChanges(). Analogously, addition and deletion of nodes and edges are performed, as prescribed by the rule.

Control flow between different rules is modeled by transformation units. Each unit represents a different kind of control flow. For example a sequential unit will execute each subunit once in a given order and is equivalent to a number of sequential activities in an activity diagram. A decision-merge construct, for instance, is represented by a conditional unit:

```
ConditionalUnit cUnit = (ConditionalUnit) trafoUnit;
TrafoUnit ifUnit = cUnit.getIf();
GenericUnit genericIfUnit = createGenericUnitFor(ifUnit);
if (genericIfUnit.execute()) -
  TrafoUnit thenUnit = cUnit.getThen();
  GenericUnit genericThenUnit = createGenericUnitFor(thenUnit);
  result = genericThenUnit.execute();
" else -
  if (cUnit.getElse() != null) -
  TrafoUnit elseUnit = cUnit.getElse();
  GenericUnit genericElseUnit = createGenericUnitFor(elseUnit);
  result = genericElseUnit.execute();
" "
```

Different units can be nested in any way with the innermost units representing single rules.

Eclipse Plug-In and Code for Views The main information needed to create an actual ECLIPSE plug-in is stored in the plug-in manifest file plugin.xml in the project root directory. The manifest defines the extension points that are used by the plug-in. The most important one for our purposes is the org.eclipse.ui.editors extension point. As the editor is used to control the life-time of a plug-in instance as well handles the actual model instance that is being edited. The extension point mainly defines a unique identifier, the name of the plug-in, a file extension that should be associated with model instances, an icon that will be displayed for the file extension, the actual class that implements the main EditorPart, and a contributor class that implements an ActionBarContributor.

The second important extension point is org.eclipse.ui.views that is used for the actual visual editor components that live within the environment. When creating new files in the workspace (when using File → New), a customWizard can be defined via the org.eclipse.ui.newWizard extension point.

The overall layout of the views is handled via an ECLIPSE perspective. The controlling class sets up the default position of the editor and the views within the whole workspace. This is realized by using our framework MUVITOR (Multi-View-Editor) [25] that is built on top of GEF. MUVITOR generalizes recurring code fragments for many editor features. It supports nested modes with multiple graphical viewers and animated simulation of model behavior. The architecture is designed in a way that encapsulates complex underlying chains of commands in GEF and simplifies the interaction with the ECLIPSE workbench.

5.3 Generated Tool Environment

Fig. 19 shows a sample view for the AntWorld simulation. Activity diagrams can be evoked for automatic application of controlled rule sequences, and their effect can be viewed in different views at the same time.

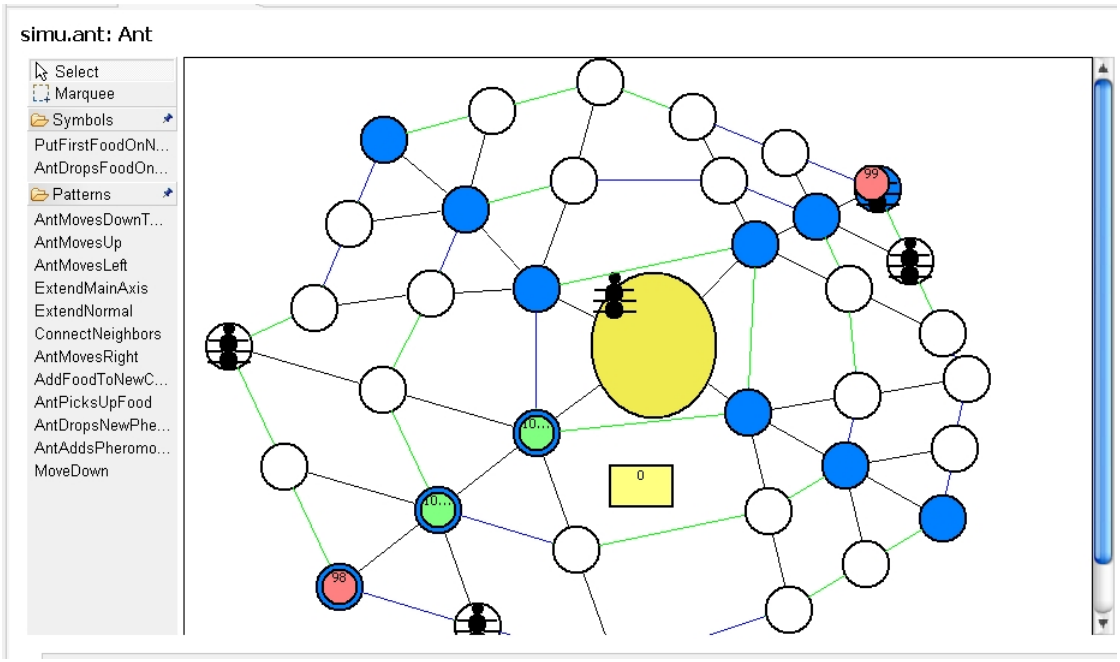


Fig. 19. Generated AntWorld simulation view

6 Related Work

Apart from GMF [10], also the Topcased modeler generator of the OpenEmbeDD [3] MDE platform provides graphical patterns for common parts of user specific EMF domain models and thus allows to easily create a basic graphical editor. However, multi-view graphical editors need a more flexible and user friendly way to define domain specific editor environments including editor operations, simulation and model transformation tools. These requirements can be more adequately fulfilled by graph transformation tools [27] providing a graphical way to define complex operations for editing, simulation, and model transformation of domain specific languages based on a well-defined theoretical background [14]. Up to now, a com-

prehensive generation framework combining graph transformation and EMF-based meta-modeling for visual environment generation has not yet been implemented.

For model transformations, ATL [2] is widely used as part of the ECLIPSE Model-To-Model Transformation (M2M) [1] project. While ATL uses a textual syntax for model transformation, TIGER 2 with its underlying EMF TIGER transformation engine provides a graphical way to define model transformations between different DSMLs based on EMF.

7 Conclusion and Future Work

We have introduced the underlying concepts and implementation of an EMF-based generator of modeling tool environments for visual DSMLs. Our implementation TIGER2 is an ongoing project. Of paramount importance is the completion of a mainly visual and intuitive *Designer* component. This will be followed by a comprehensive user evaluation to better compare the generation environment to existing approaches. Since TIGER2 is an ECLIPSE plug-in, based on the MUVITOR framework [25], exactly like the TIGER2-generated environments, a challenging task is to design and generate TIGER2 by TIGER2 in order to prove its adequateness and flexibility. Some issues concerning the visualization design are still open: As stated in [5], we aim at a flexible way to map changing visualizations to model elements by making figure properties like size or color depend on values of the model element's attributes. Hence, we will refine our view model in order to allow also mappings from attributes to figure properties. Likewise, rule control by activity diagrams can be improved by also considering the object flow (see [20]). This can help to not only define the order of rule applications, but also to define (parts of) the matches for successor rules.

References

1. Model To Model: The M2M subproject of the Eclipse Modeling Project, http://www.eclipse.org/m2m/
2. ATL: The Atlas Transformation Language (2008), http://www.eclipse.org/m2m/atl/
3. Openembedd: Model driven engineering open-source platform for real-time & embedded systems (2009), http://openembedd.org
4. AGG (2009), http://tfs.cs.tu-berlin.de/agg
5. Biermann, E., Ehrig, K., Ermel, C., Hurrelmann, J.: Flexible Visualization of Automatic Simulation based on Structured Graph Transformation. In: Proc. IEEE Symposium on Visual Languages and Human-Centric Computing. pp. 21–29. IEEE Computer Society (2008)

6. Biermann, E., Ehrig, K., Köhler, C., Kuhns, G., Taentzer, G., Weiss, E.: EMF Model Refactoring based on Graph Transformation Concepts. In: Proc. Workshop on Software Evolution through Transformations. vol. 3. ECEASST (2006)
7. Biermann, E., Ehrig, K., Köhler, C., Kuhns, G., Taentzer, G., Weiss, E.: Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. In: Proc. Conf. on Model Driven Engineering Languages and Systems. LNCS, vol. 4199, pp. 425–439. Springer (2006)
8. Biermann, E., Ermel, C., Taentzer, G.: Precise Semantics of EMF Model Transformations by Graph Transformation. In: Czarnecki, K. (ed.) Proc. Conf. on Model Driven Engineering Languages and Systems. LNCS, vol. 5301, pp. 53–67. Springer (2008)
9. Eclipse Consortium: Java Emitter Templates (JET) – Version 2.0.1 (2003), http://www.eclipse.org/emf
10. Eclipse Consortium: Eclipse Graphical Modeling Framework (GMF) (2007), http://www.eclipse.org/gmf
11. Eclipse Consortium: Eclipse Modeling Framework (EMF) – Version 2.4 (2008), http://www.eclipse.org/emf
12. Eclipse Consortium: Eclipse Graphical Editing Framework (GEF) – Version 3.4 (2009), http://www.eclipse.org/gef
13. Eclipse Modeling Foundation: MDT-UML2Tools (2009), http://wiki.eclipse.org/MDT-UML2Tools
14. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs in Theor. Comp. Science, Springer (2006)
15. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.): Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools. World Scientific (1999)
16. Ehrig, K., Biermann, E., Taentzer, G., Ermel, C., Köhler, C.: The EMF Model Transformation Framework. In: Proc. AGTIVE 2007 Symposium. LNCS, vol. 5088, pp. 66–68. Springer (2008)
17. Grundy, J., Hosking, J., Huh, J., Na-Liu Li, K.: Marama: An Exclipse Metatoolset for Generating Multi-view Environments. In: Proc. Conf. on Software Engineering (ICSE). pp. 819–822. ACM Press (2008)
18. Habel, A., Heckel, R., Taentzer, G.: Graph Grammars with Negative Application Conditions. Special issue of Fundamenta Informaticae 26(3,4), 287–313 (1996)
19. Jurack, S., Lambers, L., Mehner, K., Taentzer, G.: Sufficient Criteria for Consistent Behavior Modeling with Refined Activity Diagrams. In: Proc. ACM/IEEE 11th Conf. on Model Driven Engineering Languages and Systems. LNCS, vol. 5301, pp. 341–355. Springer (2008)

20. Jurack, S., Lambers, L., Mehner, K., Taentzer, G., Wierse, G.: Object Flow Definition for Refined Activity Diagrams . In: Proc. Fundamental Approaches to Software Engineering. lncs, vol. 5503, pp. 49–63. Springer (2009)
21. de Lara, J., Vangheluwe, H., Alfonseca, M.: Meta-Modelling and Graph Grammars for Multi-Paradigm Modelling in AToM^3. Software and System Modeling 3(3), 194–209 (2004)
22. Ledeczi, A., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing domain-specific design environments. IEEE Computer pp. 44–51 (2001)
23. Microsoft: Domain specific language tools, http://msdn.microsoft.com/vstudio/DSLTools/
24. Minas, M.: DiaGen / DiaMeta – The Diagram Editor Generator (2007), http://www.unibw.de/inf2/DiaGen/
25. Modica, T., Biermann, E., Ermel, C.: An ECLIPSE Framework for Rapid Development of Rich-featured GEF Editors based on EMF Models (2009), http://tfs.cs.tu-berlin.de/publikationen/Papers09/MBE09.pdf
26. Meta Object Facility (MOF) Core Specification 2.0. http://www.omg.org/technology/documents/modeling_spec_catalog.htm#MOF (2008)
27. Taentzer, G.: Characterizing Tools for Visual Modeling Techniques. In: Lecture Notes of SegraVis Advanced School on Visual Modelling Techniques (2006)
28. Tolvanen, J., Rossi, M.: MetaEdit+: Defining and Using Domain-Specific Modeling Languages and Code Generators. In: Proc. Conf. on Object-oriented programming, systems, languages, and applications. pp. 92–93. ACM Press (2003)
29. TU Berlin: Tiger: Generating Visual Environments in Eclipse (2005), http://www.tfs.cs.tu-berlin.de/tigerprj
30. TU Berlin: EMF Model Transformation Framework (2009), http://tfs.cs.tu-berlin.de/emftrans
31. MOF 2.0 / XMI Mapping Specification. http://www.omg.org/technology/documents/formal/xmi.htm (2008)
32. Zündorf, A.: AntWorld. University of Kassel (2008), http://www.se.eecs.uni-kassel.de/~fujabawiki/index.php/AntWorld