Proceedings of the 2011 Winter Simulation Conference
S. Jain, R.R. Creasey, J. Himmelspach, K.P. White, and M. Fu, eds.

OVERVIEW OF TECHNIQUES FOR MODEL-DRIVEN DEVELOPMENT OF A SIMULATION PACKAGE

Dipl.-Inf. Pascal Weyprecht
Prof. Dr. rer. nat. Oliver Rose

Dresden University of Technology
Department for Modeling and Simulation
Dresden, Saxony, Germany

ABSTRACT

We propose model-driven development as a good choice for developing a simulator with decreased development time and increased stability and maintainability compared to traditional development techniques. Although the meta-model for the simulation model is not always known or well defined in most commercial or academic simulation software packages, all simulators use such a meta-model throughout different components of the simulator like the model editor or the simulation core. Model-driven development uses a clearly defined meta-model as a basis for generating different artifacts, ranging from executable source code to documentation files. In this paper, we present a software architecture based on the Eclipse Modeling Framework (EMF) in combination with the Graphical Modeling Framework (GMF) as basic model-driven frameworks for data-layer and graphical user interface of a simulation software package.

1 INTRODUCTION

First we want provide an overview of the technologies described in this paper. The Eclipse Modelling Framework (EMF) is only one of the exiting frameworks for model-driven development, but it is the one used in this paper so all described technologies are related to the EMF. Of course this is not a complete list of all technologies, tools or projects related to EMF. In this paper a very simple simulator will be developed with only four meta-model elements. It will be called SimpleSimulator and used as an example within all following sections.

The model-driven development process starts with a meta-model. In EMF this is defined as an Ecore Model. To make things clear, the Ecore model is the meta-model of our simulator. A model in the context of the simulator would be a simulation model, which can be simulated later on. In contrast, the meta-model of Ecore is the syntax definition of Ecore itself. Figure 1 shows the relations between all of them. There are several ways to edit the Ecore model, which will be described in the section 2 Eclipse Modeling Framework. With the help of this model definition in Ecore one can create a Generation Model where the parameters for the generating process are set. When this is done the model classes can be generated. These are Java class representations of the model.

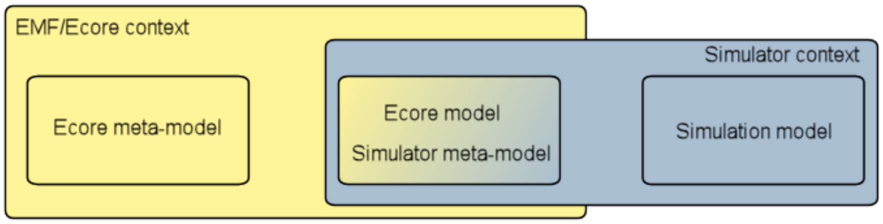


Figure 1: Ecore model vs. Simulator meta-model

In section 3 Graphical Modeling Framework a graphical editor for the simulation models will be described. Therefore four more configuration files are needed, but the result is a fully functional graphical editor with features like undo/redo, actions and toolbars, properties and outline view, shortcuts, validation and more.

As we now proposed a way to create and edit simulation models, the next step is to execute them. One possibility is to just write code that uses the model for simulation. But since we use a Model-driven solutions, an execution meta-model is created. This meta-model is also defined in Ecore and generated with EMF. However, several custom operations are needed. These will be written with eJava, which is described in the corresponding section.

Beyond the basic components that are needed for building a simulator, there are several additional features that would be nice to have. With the help of EMFText it is very easy to create a text editor with features like syntax highlighting and auto completion. In section 5 EMFText a second editor for the models will be created, giving the user the freedom to choose his preferred editing method.

Another recommendation for a simulator is documentation for the user. The task of Java Emitter Templates is not exactly to build documentation but to generate any kind of text artifact, which can also be used as a skeleton for documentation. Section 6 Java Emitter Templates will explain how to write a template and what the possibilities of Java Emitter Templates are.

This is not the first paper introducing code generating methods on developing simulation solutions. In Vangheluwe & de Lara (2002) a similar approach is introduced relying on their own tool ATOM3. Because Model-driven development was relatively new at that time (see Poole (2001)), there did not yet exist a well-developed tool chain. Nowadays there are. One example is the Eclipse Modeling Framework that is described in the following section.

2 ECLIPSE MODELING FRAMEWORK

EMF is the model-driven solution developed by the Eclipse Community. It has been developed over several years. Like in every model-driven solution, the starting point is a meta-model. This model can be edited in several ways: via a hierarchical editor, a text editor or a graphical editor. Because the graphical editor is close to the representation of the Unified Modeling Language (UML) it is a good starting point. The diagrams are very similar to class diagrams in UML. See Figure 2 for the model of the SimpleSimulator example. Every block represents a class which can have attributes and operations. These attributes can have a set of primitive types like EInt, EBoolean or EString, which are mappings of the Java types int, boolean and String. This model is based on the Ecore meta-model and can be considered as a Platform Independent Model (PIM). When this model is defined, a Generator Model can be built, which is in contrast to the first Model a Platform Dependent Model (PDM) for generating Java code. In the PDM options for the generation process like default package, source directory and several other building properties can be set. There is only a hierarchical editor for the Generator Model (see Figure 3). When the building options are set, several artifacts can be generated. First of all the Java class representation of the defined model, including getters and setters, and methods for loading and saving the data of the model in a XMI file. Also a hierarchical editor can be generated for simple editing in our new domain.

As already mentioned above, Figure 2 is a graphical representation of the SimpleSimulator model. The top level element of the this model is the class Model. This class represents the simulation model later on. It contains three kinds of elements: Tokens, ModelElements and ModelElementLinks, which will be explained now. A Token is the object flowing through the simulation later on. It has a name, so there can be for example “red” and “green” Tokens. The elements these Tokens flow through are ModelElements, which can also have a name like “machine one” or “machine two”. The class ModelElement itself is abstract, but it has four inheritors that will be introduced now. At the Source, Tokens will be created with a given interval. It can be linked to a Token, to decide which kind of Tokens will be created. The destruction of the Tokens happens when they reach a Sink. The next element is a Queue, where elements can be stored. This Queue can have a limited size, given by the attribute size. The Delay holds Tokens for a specified period of time. All ModelElements can receive and send Tokens. The relation where the Token goes next is defined with a ModelElmentLink. Therefore ModelElementLinks have a source and a target ModelElement. At this point there is no logic defined, how the model will work, only which model elements it contains and how they are connected. The logic will be defined in the section 4 eJava.

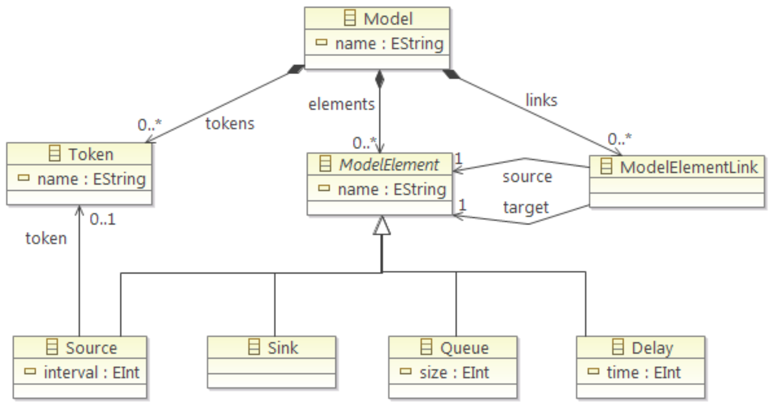
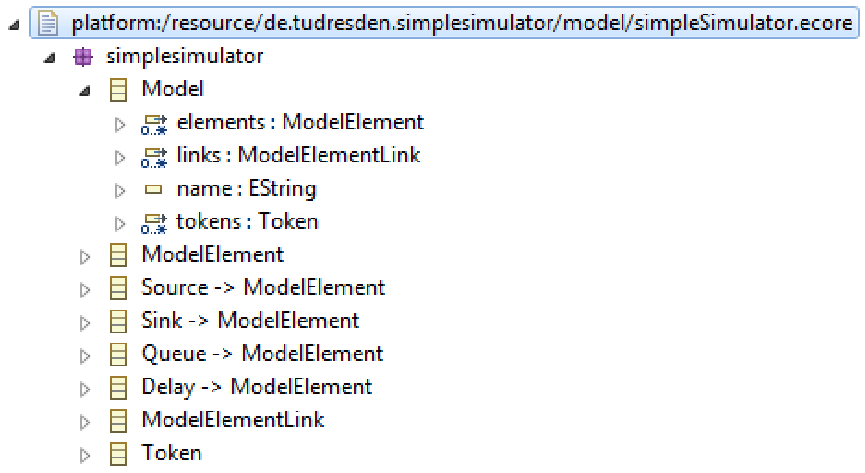


Figure 2: Graphical Representation of the SimpleSimulator Ecore Model



```
platform:/resource/de.tudresden.simplesimulator/model/simpleSimulator.ecore
  simplesimulator
    Model
      elements : ModelElement
      links : ModelElementLink
      name : EString
      tokens : Token
    ModelElement
    Source -> ModelElement
    Sink -> ModelElement
    Queue -> ModelElement
    Delay -> ModelElement
    ModelElementLink
    Token
```

Figure 3: Hierarchical Representation of the SimpleSimulator Ecore Model

3 GRAPHICAL MODELING FRAMEWORK

As discussed in the previous section, defining a model and generating a Java class representation as well as a hierarchical editor is very simple. Building a graphical editor is a little more complex. The Graphical Modeling Framework (GMF) needs four more files, to define the editor. Figure 4 shows the dashboard provided by GMF and thereby gives an overview of the steps necessary for the generation. The four files required for the process will be briefly described in this section. Although it will not be a complete documentation, we will give a brief overview of the possibilities.

The first file is an element description, where all visual elements are described. In the SimpleSimulator example these are boxes for the model elements, a circle for the token, a dashed line for the connection between source and token and lines with arrow heads for the model element links.

In the second file existing tool elements are defined. Tools in this context are the buttons that create the elements like a Source in the graphical editor later on. These tools get a name and an image, and they can also be grouped for improved usability. In this file entries for the Main Menu, Popup Menus and Toolbars can be defined, but these features are not used in the SimpleSimulator example.

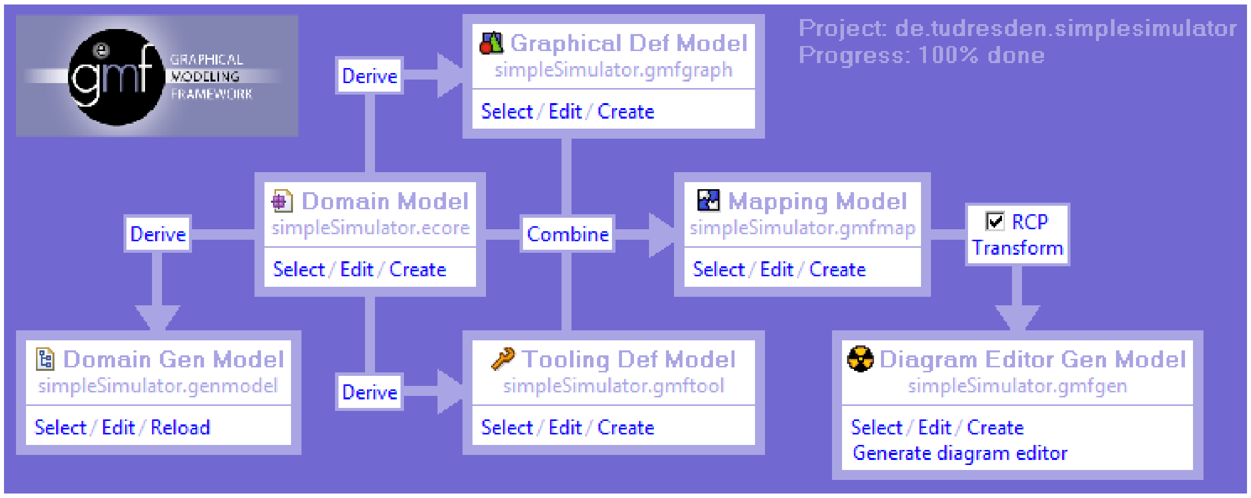


Figure 4: GMF Dashboard

None of these files has links to the Ecore model, neither the visual elements nor the tools are already linked to specific model elements. This is done in the mapping definition file ".gmfmap". In this file Node Mapping elements can be added, that map model elements from the Ecore file to Diagram Nodes from the visual description file and Tools from the tool definition file. There can also be other mappings for connections between elements and canvases, and like above that Ecore elements get mapped to visual and tool elements.

Like the ecore files in EMF, the description files for the graphical editor in GMF do not contain platform specific information, like Java in this case. This information get added in the ".gmfgen" file. Similar to the ".genmodel" file in EMF the platform independent meta-model, in this case the three files described above plus the Ecore model itself, are transformed into a platform dependent meta-model. The resulting file enables us to generate the ready to use graphical editor. For the SimpleSimulator example, this editor can be seen in Figure 5.

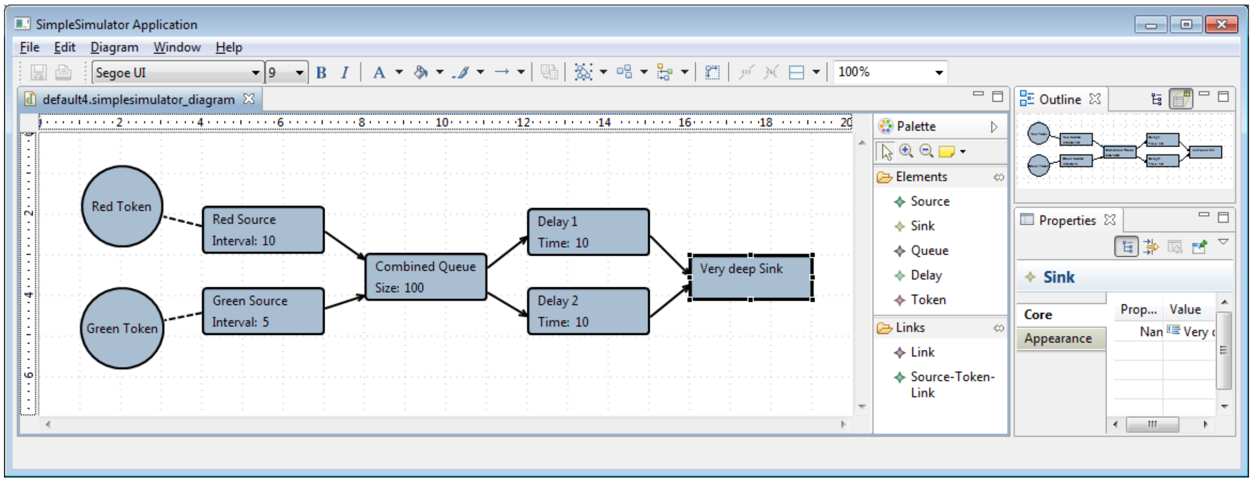


Figure 5: SimpleSelector graphical editor generated with GMF

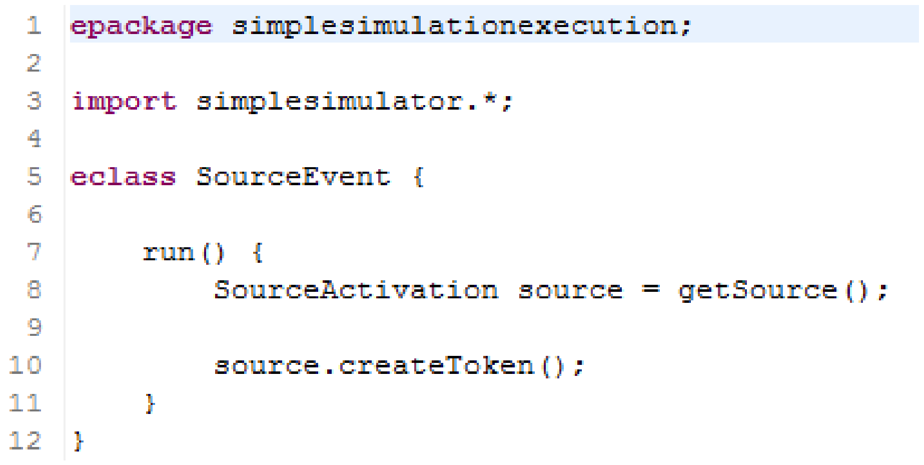
4 EJAVA

The next step in the creation of a simulator is to create the simulation engine, so that the created models with the graphical editor can be simulated. This executable code can just be written by hand, but as we will show how to work with a model-driven approach, the EOperations of EMF will be used to model the behavior of the simulation.

The usual way to implement the EOperations without the help of further tools is to generate the code and fill in the missing method bodies of the EOperation methods. This mixture of generated and hand-written code is not recommended due to several reasons. Although the hand-written code can be protected from being overwritten during the next code generation by adding a @generate NOT tag on top of the method, there are still a lot of occasions where the code can became invalid.

The problem can be solved using eJava. It allows to specify the behavior of EOperations with a textual representation of a model based on Java and Ecore. The eJava files are saved next to the Ecore model. The represented code will then be automatically put into the Generation Model, so the EMF code generator can be used without any changes. The syntax of the eJava files looks exactly like usual Java statements except some small changes for differentiating between them. The tag package is now called epackage, so is the keyword class now called eclass. Another difference is that method heads do not have parameters or return types any more, as they are already defined in the Ecore model and should not be defined twice, to minimize inconsistency. Of course the eJava Editor validates not only syntax, but also existing references within the eJava and the Ecore model and marks them directly. This way hand-written and generated code are cleanly separated, so that no compilation problems occur after the generation process.

For adding semantics to the SimpleSimulator example a second meta-model was created that follows the Visitor Pattern (see Gamma, Helm, & Johnson (1994)), to separate data from algorithms. This new meta-model is called execution model and is a simplified version of fUML (for specification see http://www.omg.org/spec/FUML/), a standard from the Object Management Group (OMG) for an executable subset of UML. Figure 6 shows one source code file of the execution meta-model of the Simple-Simulator example. There is an EClass for every meta-model element of the syntax meta-model, for example SourceActivation for Source. Additionally there is a Clock, several Events like the one in Figure 6 and more. This paper will not go into more detail on this topic but the interested reader is referred to fUML itself.

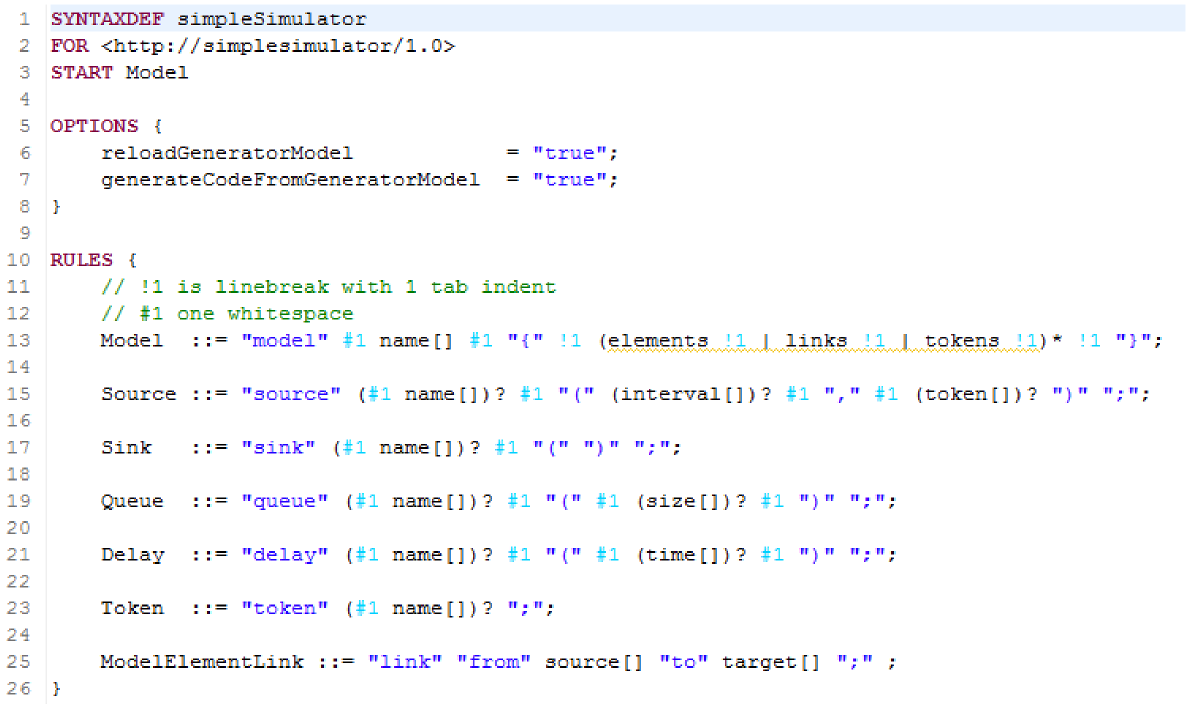


```
epackage simplesimulationexecution;

import simplesimulator.*;

eclass SourceEvent {

    run() {
        SourceActivation source = getSource();

        source.createToken();
    }
}
```

Figure 6: eJava example (SourceEvent.eJava)

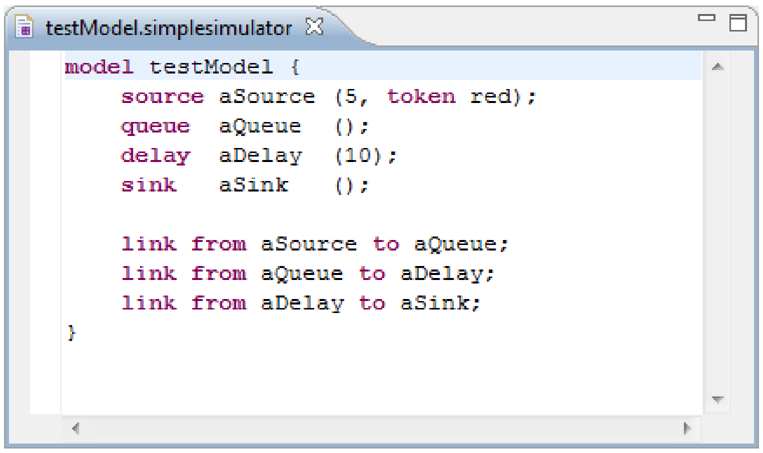
5 EMFTEXT

The eJava tool shows the importance of textual representations of models. The textual editor of eJava itself is a very good example for generated editors. It is created with EMFText, that generates a textual editor with syntax highlighting and auto completion based on an Ecore model and a syntax definition file. The syntax for the SimpleSimulator example can be found in Figure 7 as you can see it is quite simple and the only file except the Ecore model itself needed for the generation process of the text editor. The generated editor of the SimpleSimulator example is shown in Figure 8. More information about EMFText can be found at http://www.emftext.org.



```
SYNTAXDEF simpleSimulator
FOR <http://simplesimulator/1.0>
START Model

OPTIONS {
    reloadGeneratorModel             = "true";
    generateCodeFromGeneratorModel   = "true";
}

RULES {
    // !1 is linebreak with 1 tab indent
    // #1 one whitespace
    Model  ::= "model" #1 name[] #1 "{" !1 (elements !1 | links !1 | tokens !1)* !1 "}";

    Source ::= "source" (#1 name[])? #1 "(" (interval[])? #1 "," #1 (token[])? ")" ";";

    Sink   ::= "sink" (#1 name[])? #1 "(" ")" ";";

    Queue  ::= "queue" (#1 name[])? #1 "(" #1 (size[])? #1 ")" ";";

    Delay  ::= "delay" (#1 name[])? #1 "(" #1 (time[])? #1 ")" ";";

    Token  ::= "token" (#1 name[])? ";";

    ModelElementLink ::= "link" "from" source[] "to" target[] ";" ;
}
```

Figure 7: EMFText syntax of the SimpleSimulator example



```
model testModel {
    source aSource (5, token red);
    queue  aQueue  ();
    delay  aDelay  (10);
    sink   aSink   ();

    link from aSource to aQueue;
    link from aQueue to aDelay;
    link from aDelay to aSink;
}
```

Figure 8: SimpleSimulator text editor generated with EMFText

6 JAVA EMITTER TEMPLATES

The last tool that will be described here is Java Emitter Templates (JET). It is basically a generator-generator. The workflow of JET is shown in Figure 9. The user written templates get transformed into executable Java code, that generates the content of the template and fills out the gaps according to the template. This can be used in two different ways. The first way is, that both the transformation and generation step are done during developing time. In this case the output is the artifact of interest and the Java code does not need to be deployed. An example for this is documentation of a product. Here only the generated HTML files need to be shipped to the customer. The second way to use JET is to do just the transformation step during developing time and use the Java classes during runtime to generate artifacts then. A simulator project can use this technique to generate textual reports after simulation runs. A more detailed tutorial can be found at Popma (2004)

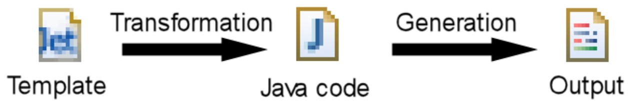
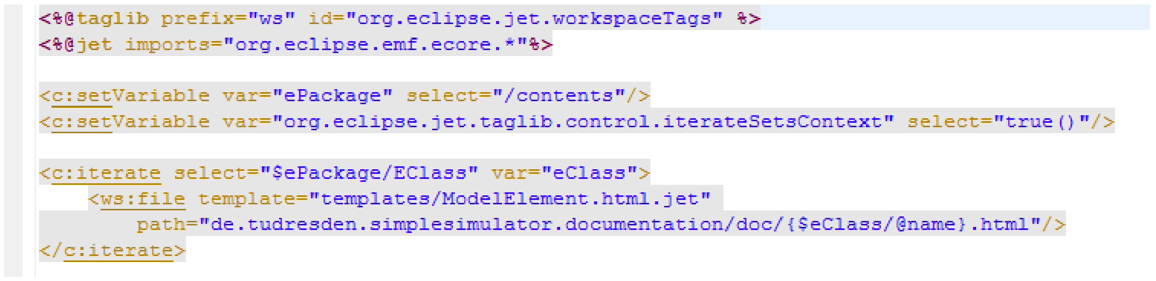


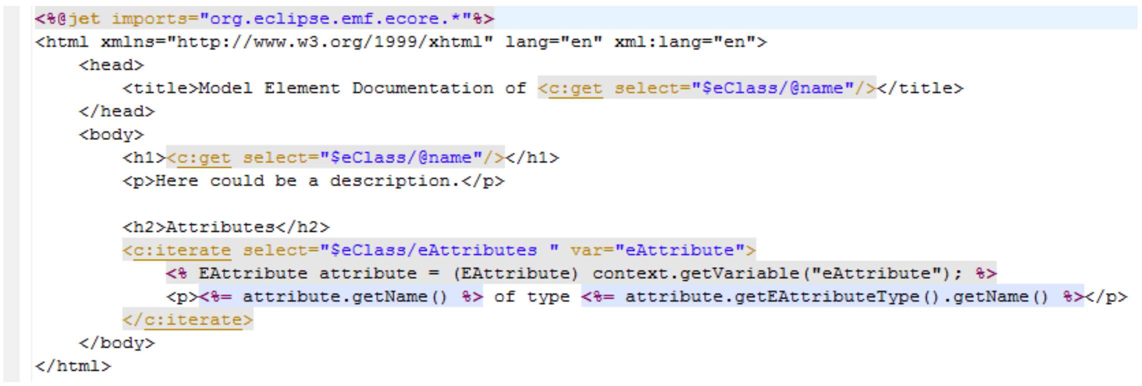
Figure 9: JET Workflow

For the SimpleSimulator example we used the first way and generated documentation for every element of the meta-model. Therefore two files are needed. The first one is “main.jet”, which defines the workflow and is shown in Figure 10. Any preparations and actions on the model are placed in this file. In the SimpleSimulator example it is an iteration over all EClasses of the model and the command to generate one output file per EClass with the template “ModelElement.html.jet”. The content of the JET file for the model element is shown in Figure 11. It contains mainly HTML code with some commands to get properties from the EClass and list all attributes that the EClass contains. The complexity of the template can of course be increased and for example comments on the model can be added to the output. Or there can be generated different files, like an “index.html” for example.



```
<%@taglib prefix="ws" id="org.eclipse.jet.workspaceTags" %>
<%@jet imports="org.eclipse.emf.ecore.*"%>

<c:setVariable var="ePackage" select="/contents"/>
<c:setVariable var="org.eclipse.jet.taglib.control.iterateSetsContext" select="true()"/>

<c:iterate select="$ePackage/EClass" var="eClass">
    <ws:file template="templates/ModelElement.html.jet"
        path="de.tudresden.simplesimulator.documentation/doc/{$eClass/@name}.html"/>
</c:iterate>
```

Figure 10: Content of "main.jet"



```
<%@jet imports="org.eclipse.emf.ecore.*"%>
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
    <head>
        <title>Model Element Documentation of <c:get select="$eClass/@name"/></title>
    </head>
    <body>
        <h1><c:get select="$eClass/@name"/></h1>
        <p>Here could be a description.</p>

        <h2>Attributes</h2>
        <c:iterate select="$eClass/eAttributes " var="eAttribute">
            <% EAttribute attribute = (EAttribute) context.getVariable("eAttribute"); %>
            <p><%= attribute.getName() %> of type <%= attribute.getEAttributeType().getName() %></p>
        </c:iterate>
    </body>
</html>
```

Figure 11: Content of "ModelElement.html.jet"

7 SUMMARY

We presented a set of tools and frameworks that support model-driven development, a technique perfectly suited for the development of a simulator. Since the common meta-model needed for simulators is also the focal point of model-driven development. We could demonstrate, that with the definition of only a few files, a lot of artifacts like editors can be generated. This paper shows the workflow of model-driven development with EMF for a simple simulator, but this is only an exemplary demonstration for the larger projects. One of these projects is the development of a SysML based discrete-event simulator solution. SysML is a general-purpose graphical modeling language for systems engineering applications, developed by the OMG (see http://www.sysml.org/). More information about the SysML simulator project can be found in (Weyprecht and Rose 2011).

REFERENCES

Gamma, E., R. Helm, and R. E. Johnson. 1994. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman.

OMG, 2011 "Semantics of a Foundational Subset for Executable UML Models (FUML)." http://www.omg.org/spec/FUML/ (accessed January 2011).

—. *SysML - Open Source Specification Project.* 2010. http://www.sysml.org/ (accessed January 2011).

Poole, J. D., 2001. "Model-Driven Architecture: Vision, Standards And Emerging Technologies." *ECOOP 2001.*

Popma, R., 2004. *JET Tutorial Part 1 (Introduction to JET).* http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html (accessed January 2011).

Software Technology Group, Dresden University of Technology. *EMFText.* 2011. http://www.emftext.org (accessed January 2011).

Vangheluwe, H., and J. de Lara, 2002. "Meta-models are models too." *Proceedings of the 2002 Winter Simulation Conference*.

Weyprecht, P., and O. Rose, 2011. "Model-driven Development of Simulation Solution based on SysML starting with the Simulation Core." *2011 Spring Simulation Multiconference*.

AUTHOR BIOGRAPHIES

PASCAL WEYPRECHT is a PhD student at the Dresden University of Technology. He is a member of the scientific staff at the Chair for Modeling and Simulation. Before that he worked for D-SIMLAB Technologies Pte Ltd, Singapore. He received his Diploma (similar to M.S.) in computer science from Dresden University of Technology, Germany in 2008. His research interests are SysML as a general purpose simulation language, as well as parallel and distributed simulations.
His e-mail address is pascal.weyprecht@tu-dresden.de

OLIVER ROSE holds the Chair for Modeling and Simulation at the Institute of Applied Computer Science of the Dresden University of Technology, Germany. He received an M.S. degree in applied mathematics and a Ph.D. degree in computer science from Würzburg University, Germany. His research focuses on the operational modeling, analysis and material flow control of complex manufacturing facilities, in particular, semiconductor factories. He is a member of IEEE, INFORMS Simulation Society, ASIM, and GI. He will be the General Chair of WSC 2012 in Berlin.
Web address: www.simulation-dresden.com.