Integrating Template based Code Generation into Graphical Model Transformation

Martin Girschick (girschick@informatik.tu-darmstadt.de)
Fachgebiet Metamodellierung
Fachbereich Informatik
Technische Universität Darmstadt, Germany

Abstract: Model transformation is not only a crucial but also one of the most complicated aspects of model-driven software development (MDSD). An adequate transformation language is therefore vital to its successful application. Architecture stratification is a very flexible approach to MDSD, which applies stepwise refinement to model-based system development. In order to support refinement automation, a powerful transformation language is required. This paper presents and evaluates a novel approach to combined graphical and textual model transformation by integrating template based code generation into a graphical model transformation language. It has been implemented in the plugin "Futemplerator" for the CASE tool Fujaba.

1 Introduction

Model-driven software development is one of the most important trends in software development. Although many tools supporting it already exist, widespread application within the industry has yet to come.

Several obstacles impeding the adoption of MDSD can be identified. For instance many tools restrict the development process to predefined steps, which cannot be adapted to existing processes. Often, only a limited set of application scenarios are supported. A survey of commercial MDA tools [TA05] revealed, that most of the tools offered only a fixed set of modelling layers. In order to produce high quality software, it becomes necessary to open the tool for user supplied extensions. This includes customizable modelling languages, flexible model transformation technology and support for versatile code generation.

We propose architecture stratification [AK03] as a flexible approach to MDSD. With it software systems are described by a stack of abstraction layers, which we call "strata". The top stratum shows the most abstract view of the system, each following stratum increases detail by adding a specific "concern". A concern is addressed by a specific feature, property or aspect of the system. Possible concerns are distribution, security, persistency or user interface but also technical aspects like design patterns or framework integrations. After all concerns have been introduced into the system, the lowest stratum is reached and the system specification is complete. It can then be exported to executable code.

Each stratum contains a model of the entire system and introduces the realization of one concern. In system development, the term "model" often implies the use of graphical

modelling languages such as UML. However, this is not always the case. Although models have a predominant representation system, they often can be visualized either graphically or textually. Depending on the metamodel structure, mixed representations are possible as well. A prominent example is the combination of graphical class diagrams with textual method bodies.

Architecture stratification is not limited to a specific modelling language. Each stratum may use the optimal metamodel to present the concern modelled on it. Our current implementation of architecture stratification uses the above mentioned hierarchical metamodel, which combines graphical UML class diagrams with Java method bodies. Support for additional metamodels is planned.

The hierarchical metamodel enables us to incrementally reduce abstraction and add implementation detail to the model. These concern implementations are automatically introduced into the system by means of model transformation. They can be modified afterwards and the changes are automatically applied to subsequent strata by re-applying the transformations.

To introduce a concern the developer adds annotations to describe it. These annotations trigger “transformation rules”, which implement the concern into the system. As this transformation often changes not only the graphical part of the model but also the code, a transformation language suitable for both representations is needed. We propose a composite transformation language, which *embeds* code generation into an existing graph based transformation language, thus offering the optimal transformation language for both representation systems.

The paper is structured as follows: Section 2 illustrates the problem by means of an example. Section 3 introduces the graphical transformation language, followed by Section 4, which discusses candidates for text transformation languages. The integration into the graphical language is described in Section 5. Section 6 details the flexible stereotype concept and deals with scaleability issues. Section 7 addresses related work and the paper closes with a conclusion in Section 8.

2 Example

Architecture stratification employs model transformation to implement concerns into software systems. The example deals with the concern “user interface”: Given a data model class it implements a Java Swing dialog, which displays appropriate edit fields for the attributes of the class.

The class diagram in Figure 1 shows the data model class and a “SwingDataDialog” annotation. Each annotation (shown as an ellipse similar to UML collaborations) is parameterized using “annotation links”, which link to other model elements. In this case, the link “target” tells the annotation for which class a dialog has to be implemented.

Figure 2 shows the result after the transformation run. On the right side the transformed class diagram, on the left the generated code of the created methods is shown. As can be

Figure 1: Annotated class diagram

seen, two classes, an association, a generalization and several methods along with their method bodies have been created. In order to describe this complex transformation, a language is needed, which supports the transformation of primarily graphical models and the ability to generate source code blocks.

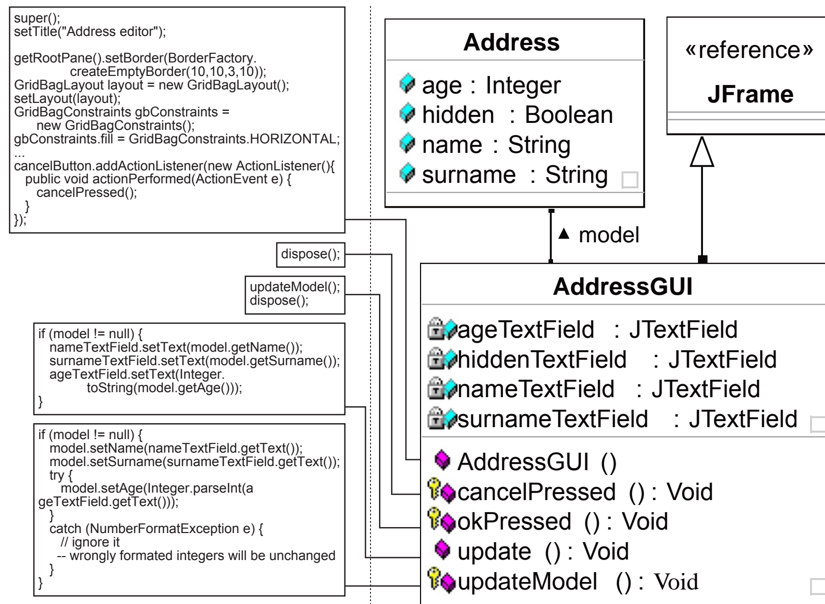

Figure 2: Result after transformation

3 Graphical Model Transformation

Most model transformation languages use text to describe the transformation, even though they transform graphical models. SPin[1], our implementation of architecture stratification, extends the CASE tool Fujaba and uses its "story driven modeling" (SDM) [FNTZ00] capabilities to describe model transformations graphically.

In order to support annotations and our hierarchical metamodel our plugin extends the Fujaba metamodel using the approach described in [BGN+04]. A simplified version of the metamodel is shown in Figure 3. Elements with a gray background belong to our SPin implementation. As can be seen, the method body is described by a list of code blocks. This simple organisation scheme allows us to clearly separate generated from hand-written code.

In addition to *generated* code blocks, which cannot be modified by the developer, transformation rules can also create so called "hook spots"[2]. These code blocks are editable by the developer and the contents are automatically retained on subsequent re-transformations (see [GKK06] for details). This adds another layer of customization to the transformation rules.

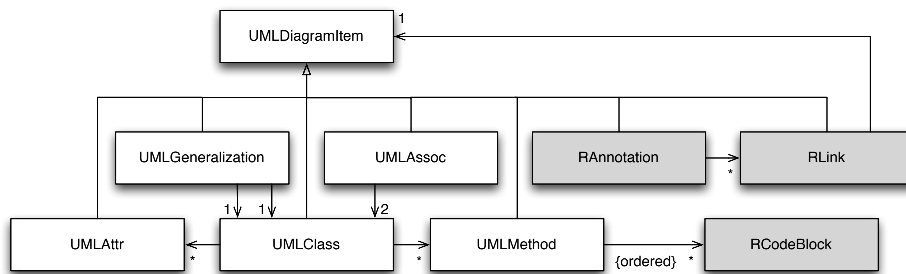

Figure 3: Simplified version of the SPin metamodel

Fujaba's story driven modeling allows the formal specification of method behaviour. It uses story diagrams—an enhanced form of UML activity diagrams—for the overall control flow. The activities within the flow describe the actual behaviour. Fujaba implements SDM and supports different activity types. One—called "story pattern"—describes graph manipulation using a notation similar to UML collaboration diagrams. It is ideal for model transformation as it combines graphical pattern matching on object structures with graph rewriting. The code generation mechanism of Fujaba exports story diagrams and the contained story patterns to executable Java code.

Figure 4 shows a part of the story diagram responsible for implementing the concern "user interface" as described in the example. The first story pattern within the diagram matches

[1] Stratification **P**lug**IN**

[2] As a combination of the "*hook* methods" from the template design pattern and framework "hot *spots*".

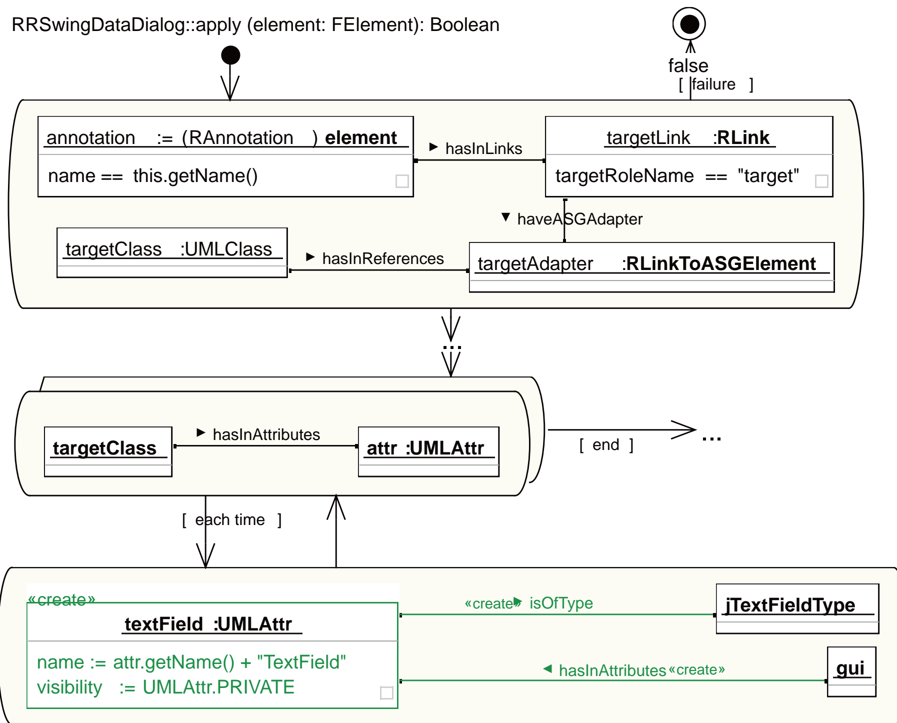

Figure 4: Part of a transformation rule

the structure shown in Figure 1. When the transformation is initiated, it finds the annotation object within the class diagram and binds the variable "targetClass" to the destination of the annotation link "target" (in this case the class "Address")[3]. The following story pattern, marked with a double border (denoting a "for each" activity), searches for all attributes within "targetClass". For each element found, it executes the third story pattern, which creates a new private attribute and attaches it to the gui class[4].

Although, at first glance, the use of an abstract, i.e. metamodel-based style, syntax is confusing, it offers in comparison to concrete syntax several advantages. The calculation of element properties can easily be expressed within the diagram because all of them are available as attributes. In addition, the notation is applicable to all metamodels compliant with the meta-metamodel. The overall control flow of story diagrams as well as the matching and creation of objects simplifies the construction of transformation rules.

However, the presented transformation rule is not complete, yet. To fully implement the concern, code for the construction of the dialog and its behavior has to be generated as well. As has already been argued in the introduction, graph rewrite systems are not suitable

[3]The intermediate object "targetAdapter" is needed for technical reasons.
[4]The creation of the class "gui" is not shown in this diagram.

for transforming or generating source code. In previous versions of SPin plain Java code was used to fill the method bodies. Figure 5 shows a statement activity responsible for generating a few lines of code.

This approach has several disadvantages: The multi-line string has to be concatenated manually, quotes and line ends have to be escaped and the mixture of the executing Java code and the Java code to be generated is confusing. The following section discusses an alternative approach for generating source code blocks.

```
String modelSetCode = "";
for (Iterator i = targetClass.iteratorOfAttrs(); i.hasNext(); attr = (UMLAttr) i.next())
{

        if (attr.getAttrType().getName().equals("Integer"))
        {
                modelSetCode += "try {\n model.set" + attr.getName() +
                "(Integer.parseInt(" + attr.getName() + "TextField.getText()));\n"
                + "   }\n   catch (NumberFormatException e) {\n" +
                "   // ignore it -- wrongly formated integers will be unchanged\n}";
        }
        else if (attr.getAttrType().getName().equals("String"))
        {
                modelSetCode += "model.set" + attr.getName() + "TextField.getText());\n";
        }
}

RCodeBlock updateModelMethodBody =
        RCodeBlockHelper.insertStatementAtStartOfMethod
        (updateModelMethod, "updateModelMethodBody", modelSetCode);
```

Figure 5: Statement Activity within a transformation rule, responsible for code generation

4 Text based Model Transformation

Method bodies are represented by a list of source code blocks, which enables clear separation of generated and hand-written parts. In most cases, when a concern is implemented by a transformation rule, new code blocks are *added* to the system. If a method already contains code blocks, new blocks can be inserted before or after one of the existing blocks.

Thus, a more detailed representation (e.g. abstract syntax trees) and full code *transformation* capabilities are not required and would lead to unnecessarily complex transformation languages. In addition, automated concern implementations can be seen as codified best practices and patterns. As a consequence, previous implementations already contain the needed code and only minor changes are required to adapt this code to the situation at hand.

Template based approaches are ideally suited for this situation as they produce parameterized text blocks. This is accomplished by mixing the output text with control elements, which are evaluated during runtime and insert calculated results into the output.

Depending on the control elements needed and the type of the output text (structured XML, unstructured plain text, source code, etc.) different approaches exist. In [Bau07] Daniel Bausch evaluated 19 template languages according to the following criteria:

Suitable for Code Generation Support for large Java code blocks including white spaces, indentations and line breaks.

Control Elements Simple markup model[5]. Support for if/else constructs, loops, macros and namespaces.

Context Transfer The parameters of the graphical transformation rule should be easily transferable to the template. The template output needs to be transferred back to the rule.

Java/Fujaba Support Availability of an API to control the template execution from Java in order to incorporate it in the transformation rule. Access to Java runtime objects (either direct or via introspection). Easily integrable into Fujaba.

Performance Template execution runtime performance.

From the 19 candidates, Velocity, WebMacro and Viento passed most of the criteria. JET and Xpand2 where discarded, because of their tight coupling with Eclipse, which made integration with Fujaba too complicated. As WebMacro and Viento are not actively maintained and Velocity already proved its capabilities for Java code generation in other projects, the latter was finally chosen. Moreover, the Fujaba plugin "CodeGen2" (described in [GSR05]) also uses Velocity to generate the executable code for Fujaba models. The existing implementation simplified the integration into Fujaba. The detailed evaluation of all suitable template languages is available in [Bau07].

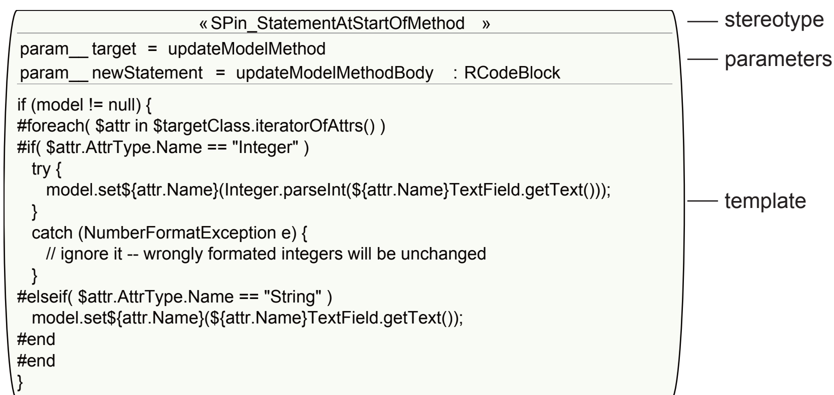

Figure 6: Template activity equivalent to the statement activity shown in Figure 5

Velocity is an interpreted template language. Markup is needed only for variables and control elements, the remaining text is copied verbatim to the output including line breaks. The lower part of Figure 6 shows the Velocity template creating the same code as the statement activity in Figure 5.

[5]Template languages employ markup characters, which separate control elements from remaining text.

Control elements start with a #, variable references with a $. Attribute values are determined by using Java introspection. Calls to Java methods (including side effects) are possible, as well. Compared to Figure 5 the template provides cleaner separation of fixed and variable text.

5 Integrating Templates

Similar to statement activities, templates are integrated into story diagrams as a new activity type. We developed a Fujaba plugin, called *Futemplerator* (FUjaba TEMPLate genERATOR), which extends the Fujaba metamodel for story diagrams with a new template activity and adds the visualization and code generation to it.

The Fujaba plugin "CodeGen2" was used for the code generation. Each story diagram describes the behaviour of one method, thus the code generation process constructs one Java method for each diagram. Each activity within the diagram creates a part of the code. The integration of template activities was accomplished by hooking into CodeGen2 and providing the necessary code generation files. The full implementation is described in [Bau07].

In addition to the template, the new activity needs to specify template parameters. Figure 6 shows such a template activity. The upper compartment contains a pop-up menu where a stereotype can be selected. This stereotype describes further processing of the output generated by the template. It also defines a set of template parameters, which are displayed in the middle compartment.

The stereotype `SPin_StatementAtStartOfMethod` for example inserts a new code block at the beginning of the method referred to by `param__target`[6]. `param__sid` specifies the Stratification ID needed for the hook spot concept mentioned in Section 3. The new code block is assigned to a local variable whose name is defined by `param__newStatement`. This enables further use of the created code block in the remaining transformation rule. For instance the stereotype `SPin_StatementAfterCodeBlock` inserts a statement after an existing block. In order to reference this block, a local variable is needed. Futemplerator already provides several stereotypes for different purposes, more stereotype definitions can be added easily. Information on the extensible stereotype concept follows in Section 6.

5.1 Context Transfer

In comparison to languages, which separate model transformation from code generation by providing two *separate* languages (e.g. OpenArchitectureWare), our integrated approach enables co-evolution of both model and code. To further simplify communication between graphical and textual parts, context information is transferred automatically from the Java

[6]The prefix "`param__`" was chosen to prevent name clashing with other variables.

Virtual Machine (JVM), which is responsible for the model transformation, to the Velocity Template Engine (VTE), which evaluates the template.

As Velocity is an interpreted language, its template parameters have to be manually fed into the VTE context. In addition to parameters defined in the middle compartment of the template activity, access to the actual model is required. The needed model elements have already been matched or created by the graphical part of the transformation rule, it therefore makes sense to transfer the context of the transformation rule to the VTE as well.

Compared to other model transformation techniques, which consist of two separate languages for model transformation and code generation this combined approach enables generation of code during the model transformation process without the need to rematch needed model elements.

The automatic transfer is possible, because CodeGen2's code generation process creates a local variable for each element within the story pattern and adds it to a list. When a template activity is executed, these variable values of the list are copied to the VTE context and after the template has been evaluated, they are retransferred back to the JVM. The latter step allows back-propagation of template evaluation side effects (e.g. creation of new elements).

The UML 2 activity diagram in Figure 7 visualizes the behaviour of the code, which is generated for a template activity. During CodeGen2's code generation of a story diagram, this code is inserted in the created Java method and then executed during model transformation. The diagram is divided into three "swim lanes" for data residing in the file system, the Java Virtual Machine and the Velocity Template Engine. The arrows denote control and data flow within the diagram, the rectangular pins on the left side of an activity show "data-in" and "-out" ports. In addition to the UML 2 notation, activities have a step counter in the upper left corner. The following section details the application of stereotypes (step 6) and macros (step 1).

6 Extensibility and Scaleability

6.1 Stereotypes

The application of template stereotypes is not limited to stratification and code generation. In the more general context of model transformation it can be used to generate other artefacts as well (e.g. build scripts or configuration files). As the list of supplied template stereotypes is extensible, additional functionalities can be implemented easily. The following simple example illustrates the use of template activities to write the template output to an arbitrary file. An elaborated version can be used to provide simple export capabilities for applications developed with Fujaba.

Each stereotype is defined by two files: A Velocity template and an XML parameter definition file. The list of available stereotypes can be extended by providing a simple Fujaba plugin containing these files and a few lines of code, which attach an additional

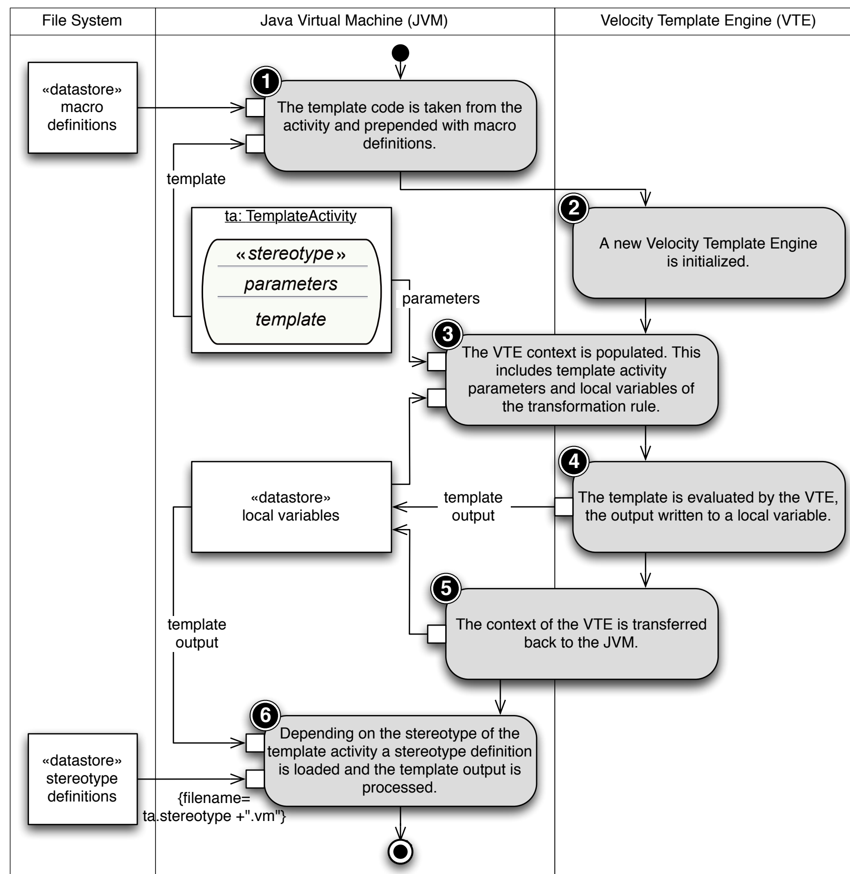

Figure 7: UML 2 activity diagram describing the behaviour of the generated template activity code

search path to Futemplerator. The new stereotypes appear in the popup folder at the top of each template activity. During code generation of the transformation rule the template associated with the selected stereotype is parsed (step 6 in Figure 7). Listing 1 shows a stereotype template, which writes the output from step 4 to the file specified by `param__filename`.

When the stereotype template shown in Listing 1 is *parsed*, `param__filename` is replaced by the parameter value and the result is inserted into the model transformation code. During *execution* of the model transformation, this code instantiates a FileWriter and writes the output (contained in the variable `futemplerator__templateOutput`) to the file.

A stereotype template is supplemented by an XML file describing the mandatory and optional parameter names and types along with a help text displayed as a tool tip within the

```
#set( $result = $imports.addToImports("java.io.FileWriter") )
   File futemplerator__filewriter = new FileWriter($!{param__filename});
   futemplerator__filewriter.write(futemplerator__templateOutput);
   futemplerator__filewriter.close();
```

Listing 1: Stereotype template: Filewriter.vm

template activity. When the stereotype is selected, the parameters are added automatically to the template activity. Listing 2 shows the parameter definition file for the template from Listing 1.

```
<?xml version="1.0" encoding="UTF-8"?>
<fpd:params xmlns:fpd="http://futemplerator.sf.net/... >
  <fpd:param    name = "filename"
                type = "java.lang.String"
             default = "&quot;output.txt&quot;"
         description = "String specifying the name of the output file." />
</fpd:params>
```

Listing 2: Stereotype parameter definition file: Filewriter.params

6.2 Macros

Step 1 in Figure 7 mentions the inclusion of macros. Velocity supports the definition of macros, which can be used similar to functions. Futemplerator automatically includes a set of global macros and macros specific to certain stereotypes in the template code. For instance, the stereotype templates for SPin provide macros, which ease the development of transformation rules. As an example, the macro in Listing 3 helps to add import statements to a class. Macros are called from templates by prepending their name with a # (e.g. `#addToImports("java.util.Iterator")`). The example also shows additional uses of Velocity in addition to plain code generation. Its ability to call Java methods from a template is used to add the import entries to a model class[7].

```
#macro( addToImports $text )
#set($result = $umlFactory.addToImportedClasses(${param__target},$!text))
#end
```

Listing 3: Macro definition: addToImports.vm

[7] `UmlFactory` provides several helper methods to create and modify model elements.

6.3 Scaleability

Due to their interpreted nature, Velocity templates are inferior to compilable statement activities. Each template has to be parsed and evaluated, in order to produce the template output. In addition, values from the model have to be determined using reflection, which further degrades the performance. We undertook a comparative analysis of two story diagrams, one with a template activity and the other with a similar statement activity.

A loop within the story diagram calls the given activity n times. The activity outputs the number n to the console. In the case of the statement activity, this is accomplished by directly calling `System.out.println`, in the case of the template activity all steps described in Figure 7 have to be executed until the stereotype template containing the call to `System.out.println` is reached. Measuring more complex examples revealed that the main overhead of template activities is created by the initialization of the Velocity engine.

Figure 8 shows the execution times for each activity type, when executed n times on a Core 2 Duo 2.33 GHz machine. For large n the overhead of template activities is quite high. However, a single transformation rule seldom contains more than a dozen template activities. Even when considering that several transformations have to be re-executed when the developer switches the stratum, the delay is still acceptable. In order to improve performance, we are planning to add a caching strategy for parsed templates to the Velocity engine.

We also conducted an analysis of memory consumption, which did not reveal any specific differences between the two approaches. A detailed description and analysis can be found in [Bau07].

n	100	1.000	10.000	100.000
statement activity	0.20 sec	0.20 sec	0.30 sec	01.20 sec
template activity	0.50 sec	1.06 sec	4.30 sec	38.00 sec

Figure 8: Execution time of statement activity vs. template activity

7 Related Work

Although we are not aware of another transformation language, which combines graphical model transformation with template based text generation, there are several tools and transformation engines, which support both model and code transformation.

Most commercial tools utilize template based languages (OptimalJ, Together) or script languages (ArcStyler) for model and code generation. The meta-CASE system Honeywell DoME follows a dual approach with a pattern based model transformation language combined with a separate code generation engine. Although the integration of a template based language is mentioned in [OSBE01], details are missing.

András Balogh and Dániel Varró describe in [BV06] the VIATRA2 framework. It currently uses a textual language for model transformation based on graph transformation. In addition, a template language is used for code generation and supports context transfer from the model transformation part, but without the ability to transfer results back to the model.

The Domain Workbench from Intentional Software presented in [SCC06] offers some similarities to the stratification approach. Their "intentional tree" contains different abstraction levels of the software under development. They are represented by different domain specific languages (DSLs) and the most concrete representation, which is similar to abstract syntax trees (ASTs), can be executed and debugged from within the Domain Workbench. Although detailed information is not available, the employed transformation approach (called "reduction") seems to be rather abstract and complicated to handle. Similar to compilers, reduction enriches the intentional tree with additional nodes until an executable form is reached.

Mens et al. [MNVE05] investigated the refactoring capabilities of Fujaba. According to their paper, the graph transformation approach is suitable for AST transformations, as well. As argued in Section 4, code *transformation* plays a less important role within architecture stratification and template based approaches employed for code *generation* are better suited. Nevertheless, code transformation is supported by the underlying technology and can be used, if needed.

8 Conclusion

In this paper we presented a novel approach to combined—graphical and textual—model transformation. The described integration of two existing transformation languages—one for graph rewriting and another for template based code generation—proved to be a feasible transformation solution to the stratification approach.

The Velocity template language offers adequate support for the generation of code blocks and Futemplerator successfully integrated it into our framework. The introduced extensible stereotype concept offers many applications within and outside the stratification approach. Although leaving room for improvements, the runtime performance is satisfactory for the currently considered test cases. A case study with a real-world software project implemented with SPin has already been started, it will provide additional information on how to improve SPin and Futemplerator.

References

[AK03] Colin Atkinson and Thomas Kühne. Aspect-Oriented Development with Stratified Frameworks. *IEEE Software*, 20(1):81–89, 2003.

[Bau07] Daniel Bausch. Integration einer Template-Sprache in das Story-Driven-Modeling-Framework von Fujaba. Bachelor thesis, Technische Universität Darmstadt, September 2007.

[BGN+04] Sven Burmester, Holger Giese, Jörg Niere, Matthias Tich, Jörg P. Wadsack, Robert Wagner, Lothar Wendehals, and Albert Zündorf. Tool integration at the meta-model level: the Fujaba approach. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(3):203–218, 2004.

[BV06] András Balogh and Dániel Varró. Advanced Model Transformation Language Constructs in the VIATRA2 Framework. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1280–1287, New York, NY, USA, 2006. ACM Press.

[FNTZ00] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language and Java. In *TAGT'98: Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations*, pages 296–309, London, UK, 2000. Springer-Verlag.

[GKK06] Martin Girschick, Thomas Kühne, and Felix Klar. Generating Systems from Multiple Levels of Abstraction. In D. Draheim and G. Weber, editors, *Conference Proceedings Trends in Enterprise Application Architecture 2006, LNCS 4473*, pages 127–141, Berlin Heidelberg, 2006. Springer-Verlag.

[GSR05] Leif Geiger, Christian Schneider, and Carsten Reckord. Template- and modelbased code generation for MDA-Tools. *3rd International Fujaba Days 2005*, September 2005.

[MNVE05] Tom Mens and Serge Demeyer Niels Van Eetvelde, Dirk Janssens. Formalising Refactorings with Graph Transformations. *Journal of Software Maintenance and Evolution*, 17(4):247–276, 2005.

[OSBE01] David Oglesby, Kirk Schloegel, Devesh Bhatt, and Eric Engstrom. A Pattern-based Framework to Address Abstraction, Reuse, and Cross-domain Aspects in Domain Specific Visual Languages. In *OOPSLA 2001 Workshop on Domain Specific Visual Languages*, Jyväskylä, Finland, 2001. Jyväskylä University Printing House.

[SCC06] Charles Simonyi, Magnus Christerson, and Shane Clifford. Intentional Software. In Peri L. Tarr and William R. Cook, editors, *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 451–464. ACM, 2006.

[TA05] Naveed Ahsan Tariq and Naeem Akhter. Comparison of Model Driven Architecture (MDA) based Tools. Master thesis, Royal institute of Technology (KTH), Stockholm, Sweden, 2005.