Standardizing SDM for Model Transformations

Hans Schippers*
Formal Techniques in Software Engineering
Universiteit Antwerpen, Belgium
Hans.Schippers@ua.ac.be

Pieter Van Gorp
Formal Techniques in Software Engineering
Universiteit Antwerpen, Belgium
Pieter.VanGorp@ua.ac.be

*Research Assistant of the Research Foundation - Flanders (FWO - Vlaanderen)

ABSTRACT

Transformations are a key technology in model driven software engineering since they are used to implement refinements for platform independence, restructurings for software migration and weavings for aspect composition. By considering transformations as models, one can develop transformations in the same paradigm as conventional applications. In this paper, we illustrate how Fujaba's language for graph rewriting has been applied for the CASE tool independent development of model transformations.[1]

1. MODEL TRANSFORMATION IN FUJABA

As Sendall and Kozaczynski state [2], model transformation can be seen as the *heart and soul* of model driven software development. Model transformations therefore deserve to be treated as first class entities in software development.

Considering transformations as models [3], recent experiments [4] have shown that Story Driven Modeling (SDM [5]) can be used as a language for the visual development of refactorings (which are a particular kind of "horizontal" model transformations). However, SDM's implementation in Fujaba suffers from two significant problems. First, the SDM metamodel in Fujaba is non-standard and it is only implicitly present in the source code. As a consequence, only the Fujaba editor is suitable to create and store SDM instances. Second, the Fujaba code generator exclusively generates code conforming to non-standard conventions, meaning it can solely be deployed on the Fujaba repository.

2. A NEW SDM COMPILER BASED ON MDA STANDARDS

Both these problems can be overcome by making use of a few MDA standards. More specifically UML, as an alternative for SDM, MOF for standardized (meta)model access and storage, and finally JMI as the binding between MOF and the Java programming language.

[1]A more elaborated paper on this work has been accepted at the ICGT'04 workshop on Software Evolution through Transformations [1].

2.1 UML Profile for Model Transformation

The first issue has been tackled by designing a UML profile for SDM, the aim being to resemble the SDM concrete syntax as closely as possible, while keeping the semantics in place. Thus, an attempt was made to associate each SDM construct with a suitable UML counterpart. To handle different variations of the same construct (such as *for each* activities versus *code* activities versus normal story activities), UML stereotypes have been used to make the distinction.

Mapping the *control flow* part of SDM proved to be fairly straightforward, because of the presence of activity diagrams in the UML standard. For the Story *primitives*, the closest match were object (collaboration) diagrams. However, these don't seem to be available in every CASE tool, and even if they are, they often offer less visual features (such as displaying attribute assignments), than class diagrams. Therefore, the latter were the preferred candidate.

The fact that the semantics of UML class diagrams in the context of model transformation differ somewhat from their conventional usage, does not really pose a problem, as people probably know what context they are dealing with. Furthermore, the model transformation semantics are formalized in OCL, although this currently needs to be checked in a separate tool, as CASE tools typically do not allow the addition of meta-constraints yet. As an illustration, Table 1 lists part of the actual mapping of SDM to UML.

SDM Construct	UML Construct
Story Activity	ActionState
ForEach Activity	ActionState with «for each» stereotype
Unbound object	UmlClass
Bound Object	UmlClass with «bound» stereotype

Table 1: Extract from SDM to UML mapping

It should be clear that any UML compliant CASE tool can now be used to create SDM instances. Additionally, since UML complies to the MOF [6] standard, any MOF repository can be employed for storing the models in a standardized way. For example, the NetBeans MetaData Repository (MDR [7]) is an open source Java implementation of MOF (or JMI [8] to be more precise) that is used in several UML and MDA tools [9, 10]. Note that the latter

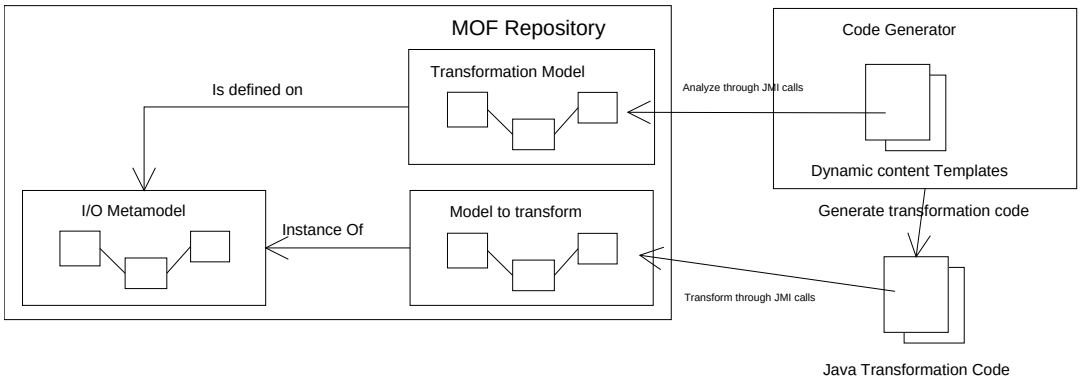

Figure 1: Code Generation Process

does not put any additional requirements on the CASE tool being used. Having a CASE tool which makes use of a MOF repository internally, is certainly convenient if it makes its API accessible to the code generator, but this can easily be circumvented by exporting UML models to XMI, and importing the result in an external MOF repository. Obviously, in that case, XMI export capabilities need to be available in the CASE tool in question, but the majority of tools do provide this feature nowadays.

2.2 Generation of Transformation Code

To solve the second problem, the Fujaba code generator was replaced by another open source solution called AndroMDA [11] for two reasons. On the one hand, AndroMDA was designed to get the information necessary to generate code from MOF compliant models inside a MOF repository. On the other hand, at the heart of the code generator is a set of dynamic content templates, which can easily be replaced in order to support different target platforms.

```
1   <#if transform.isCodeState(state)>
2           <#-- Code state => process its entry action
3                   and put the result here -->
4           <#list transform.getProcessedStatements(state) as statmnt>
5   <@indent/>          ${statmnt}
6           </#list>
7   <#elseif transform.isLinkState(state)/>
8   <@indent/>          _jcmtg_success${nd} =
9   <@indent/>                  ${transform.getMethodCallEntryAction(state)};
10  <#else/>
11          <#-- State with Transformation Primitive (diagram) -->
12          <#local primPkg = transform.getTransPrimitivePackage(state)/>
13          <#include "TransPrimitive.ftl">
14  </#if>
```

Figure 2: Dynamic Content Template Sample

The code generation process is depicted in Figure 1. The MOF repository (MDR) could be seen as the starting point, as it holds the transformation specification (transformation model). The dynamic content templates contain directives in order to extract information from this model, and based hereon, deliver a java source file with the actual transformation code. A sample of such a template can be found in Figure 2. It handles a UML ActionState (the *state* variable), by first checking what kind of activity it actually is, and then performing the appropriate action. To find out the required information, the template calls upon a helper object, accessible via the *transform* variable, which does the actual querying of the transformation model. The result consists of all text not surrounded by any type of brackets (just *_jcmtg_success* in this case), as well as the (string-formatted) result of any calls surrounded by ${}.

At the moment, only one set of templates is available, which is responsible for the generation of Java code conforming to the JMI standard, which is a mapping of MOF to Java. In other words, the resulting code is capable of accessing a JMI-compliant MOF repository such as MDR, and performing the actual transformation upon a model stored within there. Thus, model transformation code can be generated for any model, instantiating any metamodel, stored in a MOF repository which can be accessed through JMI interfaces, while other configurations (like EMF [12]) can be supported by writing a new set of templates. Note that currently, only intra-metamodel transformations are supported, that is, transforming an instance of a certain metamodel into another instance of the same metamodel.

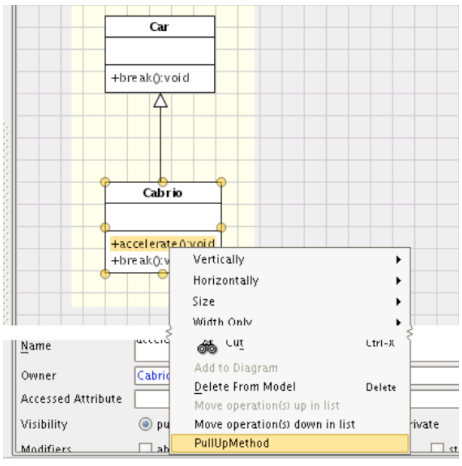

Figure 3: Refactoring Plugin generated for Poseidon.

3. EXAMPLE TRANSFORMATION

In order to validate the CASE tool independent approach to transformation development, we used the MagicDraw UML tool [13] to specify the “Pull Up Method” refactoring that illustrated the original Fujaba approach in [4]. Figure 3 shows the plugin that the new compiler generated for the Poseidon UML tool [9]. This com-

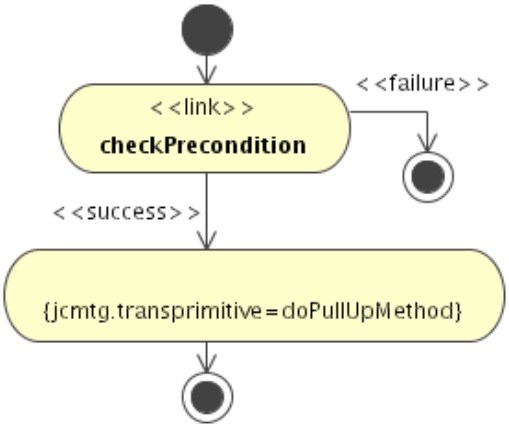

Figure 4: Transformation Flow edited in MagicDraw.

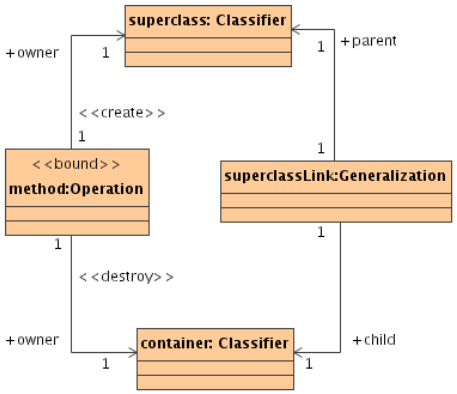

Figure 5: Transformation Primitive edited in MagicDraw.

prises the model transformation code on the one hand, and some plumbing code on the other hand, the latter generated by a separate so-called AndroMDA "cartridge". Obviously, Poseidon could be used for the specification as well, instead of MagicDraw, but we wanted to stress interoperability between several tools.

Figure 4 visualizes that the actual transformation should only be applied if the input model satisfies a certain precondition (The «link» stereotype merely indicates that the specification of this precondition is to be found in a separate diagram, pointed to by means of a tagged value). Basically, the precondition is satisfied if and only if it makes sense to perform a Pull Up Method. In other words if, for example, the class owning the method in question, does not have a superclass, the precondition would fail. Equivalent to Fujaba, the underlying semantics are based on programmed (or controlled) graph rewriting [14].

Figure 5 displays the primitive graph transformation rule that removes a method from its containing class and adds it to the list of methods from the superclass: The bound object named *method* represents the method which should be pulled up, and serves as a starting point for the lookup of its containing class *container* via the UML meta-association *owner*. Once *container* is bound too, its superclass is looked up in a similar way. The precondition guarantees that it is indeed possible to bind all objects. At that point, the owner of the method is changed from *container* to *superclass*, indicated by the «create» and «destroy» stereotypes. This completes the transformation.

Some problems, for example complex numerical calculations, seem to be solved more easily when using a conventional programming language like Java than when using graph rewriting. To illustrate that one does not have to choose for one approach exclusively, a part of the precondition is currently implemented using Java that integrates with the generated part of the transformation. It can also be specified completely using visual SDM constructs equivalent to those in Figure 4 and 5.

4. CONCLUSIONS AND FUTURE WORK

We can conclude that the model driven engineering techniques for platform independence can be applied to model transformations as well. This enables the developers of refactorings, normalizations, refinements and other kinds of program transformations to bypass CASE tool vendor lock-in.

Future work that will affect MDA and graph rewriting practitioners includes the integration of OCL into SDM, the possible support for additional visual language constructs, repository-platform independent Java transformations and the development of a dedicated GUI. Concerning the latter, it would be interesting to replace Fujaba's code generators for class and story diagrams by the discussed open source alternatives since it will reuse Fujaba's powerful editor. At the same time, Fujaba's models would be more interchangeable by reusing mature XMI serializers from NetBeans' MDR. Finally, the generated code could be deployed to any JMI repository and support for other repositories could be added with moderate effort.

The OCL integration would improve the transformation tool in two ways. Firstly, it would remove the current dependency on an external OCL tool to evaluate the well-formedness of model transformations. Secondly, constraints within the transformation model could be expressed not only in plain Java but also in OCL. The hybrid graphical/textual language would support the complete specification of repository independent model transformations.

The code driven Java approach, described in the context of the precondition of our example in Section 3, currently suffers from direct dependence on the framework of the target repository. Building Java wrappers [15] around the metaclasses of different repository platforms (JMI, EMF, ...) is a solution that is not efficient in terms of developer effort when these classes are implemented manually. One could follow a hybrid model/code driven approach by generating such wrappers on the one hand and writing pieces of manual transformation code that use the generated wrappers on the other hand. The current compiler does not use such a wrapper-based approach but generates code that calls the repository-platform specific metaclasses directly. However, if the Java "backdoor" turns out to be desirable in the long term, one can extend the compiler to generate wrappers that could be used to write repository-platform independent transformation fragments in Java.

Future work that will affect maintainers of the SDM compiler includes applying vertical model transformation on the compiler itself. It is motivated by the objective to reuse the graph matching and rewriting algorithm for different target repositories and to make the dynamic content templates as trivial as possible. This work

should lead to more insight into how one can maximally reuse and specialize parts of MDA code generators.

5. ACKNOWLEDGMENTS

This work has been sponsored by the Belgian national fund for scientific research (FWO) under grants “Foundations of Software Evolution” and “A Formal Foundation for Software Refactoring”. Other sponsoring was provided by the European research training network “Syntactic and Semantic Integration of Visual Modeling Techniques (SegraVis)”.

6. REFERENCES

[1] Hans Schippers, Pieter Van Gorp, and Dirk Janssens. Leveraging UML profiles to generate plugins from visual model transformations, October 2004. Accepted at Software Evolution through Transformations (SETra). Satellite of the 2nd Intl. Conference on Graph Transformation.

[2] S. Sendall and W. Kozaczynski. Model Transformation - The Heart and Soul of Model-Driven Software Development. *IEEE Software, Special Issue on Model Driven Software Development*, pages 42–45, Sept/Oct 2003.

[3] Jean Bézivin and Sébastien Gérard. A preliminary identification of MDA components. In *Proc. Generative Techniques in the context of Model Driven Architecture*, 2002.

[4] Pieter Van Gorp, Niels Van Eetvelde, and Dirk Janssens. Implementing Refactorings as Graph Rewrite Rules on a Platform Independent Metamodel. In *Proceedings of the 1st International Fujaba Days*, University of Kassel, Germany, October 2003.

[5] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In *Proceedings of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT)*, volume 1764 of *LNCS*, pages 296–309. Springer Verlag, November 1998.

[6] Object Management Group. Meta-Object Facility Specification, April 2002. version 1.4. document ID formal/02-04-03.

[7] Sun Microsystems. NetBeans Metadata Repository, 2002. <http://mdr.netbeans.org/>.

[8] Sun Microsystems. Java Metadata Interface Specification, June 2002. document ID JSR-40.

[9] Gentleware. Poseidon for UML, version 2.2, 2004. <http://www.gentleware.com>.

[10] Compuware. OptimalJ. <http://www.compuware.com/products/optimalj/>, 2004.

[11] M. Bohlen. AndroMDA - from UML to Deployable Components, version 2.1.2, 2003. <http://andromda.sourceforge.net>.

[12] Bill Moore, David Dean, Anna Gerber, Gunnar Wagenknecht, and Philippe Vanderheyden. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Redbooks. International Business Machines, January 2004.

[13] No Magic. Magicdraw. <http://www.magicdraw.com/>, 2004.

[14] Horst Bunke. Programmed graph grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*, pages 155–166. Springer-Verlag, 1979.

[15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, chapter 4, pages 139–150. Professional Computing Series. Addison-Wesley, 1995.