A GENERATOR FRAMEWORK FOR DOMAIN-SPECIFIC MODEL TRANSFORMATION LANGUAGES

T. Reiter, E. Kapsammer, W. Retschitzegger
Dept. of Information Systems, Johannes Kepler University, Linz, Austria
reiter@ifs.uni-linz.ac.at, ek@ifs.uni-linz.ac.at, werner@ifs.uni-linz.ac.at

W. Schwinger
Dept. of Telecooperation, Johannes Kepler University, Linz, Austria
wieland.schwinger@jku.ac.at

M. Stumptner
Adv. Computing Research Center, University of South Australia, Adelaide, Australia
mst@cs.unisa.edu.au

Keywords: Domain-specific languages, model transformation, QVT, MDA, workflow patterns.

Abstract: Domain specific languages play an important role in model driven development, as they allow to model a system using modeling constructs carrying implicit semantics specific to a domain. Consequently, possibly many reusable, domain specific languages will emerge. Thereby, certain application areas, such as business process engineering, can be jointly covered by a number of conceptually related DSLs, that are similar in a sense of sharing semantically equal concepts. Although, a crucial role in being able to use, manage and integrate all these DSLs comes to model transformation languages with QVT as one of their most prominent representatives, existing approaches have not aimed at reaping benefit of these semantically overlapping DSLs in terms of providing abstraction mechanisms for shared concepts. Therefore, as opposed to a general-purpose model transformation language sought after with the QVT-RFP, this work discusses the possibility of employing domain-specific model transformation languages. These are specifically tailored for defining transformations between metamodels sharing certain characteristics. In this context, the paper introduces a basic framework which allows generating the necessary tools to define and execute transformations written in such a domain-specific transformation language. To illustrate the approach, an example language will be introduced and its realization within the framework is shown.

1 INTRODUCTION

Model transformations play a key part in the success of model engineering in general, and for instance OMG's Model Driven Architecture (MDA) in particular. Applying MDA, the transition between PIM and PSM is facilitated by a model transformation. In an effort to standardize a language for specifying such model transformations, the OMG issued the QVT-RFP (OMG, 2002) (Queries/Views/Transformation). The QVT language allows writing transformation specifications between MOF-based metamodels. A QVT engine is then able to execute these transformations and create (or update) a target model from a source model. At the time of writing, the adoption of a QVT standard is in its final stages, but various implementations of QVT-like languages are available, such as (Bézivin, 2003), (Marschall, 2003), (INRIA Triskell, 2004). However, it is still to be seen whether a final standard finds industry-wide acceptance in the form of transformation engines, or whether vendor specific interpretations or even adaptations of the standard will emerge.

At present, the situation is somewhat comparable to the so-called 'method wars' in the nineties (Thomas, 2003), where different modeling methods like OMT, Booch, and so forth, were competing for standardization and adoption in the community. The struggle finally resulted in the emergence of the Unified Modeling Language (UML), which nowadays can be seen as the standard modeling language everyone can (more or less) agree on. The 'more or less' refers to developers who find the recent UML 2.0 version a too large and to wide-scoped standard to be reasonably applicable in software development. It can be argued, that as opposed to such general-purpose modeling languages, DSLs (domain-specific languages), for instance in the form of UML profiles or custom-built MOF-based metamodels, can contribute significantly to make model-driven software development work efficiently.

Based on this rationale and the intent to capitalize on semantically equivalent concepts inherent in various DSLs, this paper suggests the employment of *domain specific model transformation languages* (DSMTLs), which can simplify certain transformation tasks by offering appropriate abstraction mechanisms for such shared concepts. Hence, DSMTLs are tailored for certain recurring transformation tasks among a number of languages (metamodels) covering same or similar domains.

However, the focal point of this paper is not on inventing a fully-fledged DSMTL for some family of DSLs, but on motivating the approaches applicability and on furthermore proposing a generator framework that allows for the efficient implementation of DSMTLs. Nevertheless, as a running example a simple DSMTL is employed throughout the paper to motivate our intent and serve as a proof of concept for the application of the generator framework.

The rest of this paper is structured as follows: Section 2 will discuss the applicability of domain specific languages for model transformations, followed by Section 3 giving an overview of a framework for the generation of DSMTLs. Section 4 gives an example of how a DSMTL transformation definition is rendered executable. Lessons learned throughout the implementation of a prototypical framework are laid out in Section 5. Section 6 gives an overview of related work on that topic, and section 7 concludes with an outlook on future work.

2 DOMAIN SPECIFIC TRANSFORMATION LANGUAGES

Domain specific languages focus on a narrow domain only, and can therefore take certain assumptions about a domain. For instance, terms and concepts of a certain domain that implicitly carry specific semantics are immediately reused. Hence, the main advantage of a DSL is that it is accessible for persons having knowledge in a certain domain, but who are laymen (non-programmers) to general modeling per se. The disadvantage of DSLs is that considerable effort goes into devising and implementing them, as for every DSL a set of tools, such as parsers, compilers or debuggers, which support their application are required.

The idea of employing languages tailored to certain domains is not a new one (Landin, 1966), and numerous DSLs have been devised since. Meanwhile, the DSL approach has also propagated into the model engineering community. Actually, considerable controversy is abound in terms of whether MDD (model-driven development) should focus on a general-purpose modeling language, namely UML, or make use of a number of smaller, domain-specific languages expressed as metamodels.

In any case, model transformation technology contributes an essential part to the successful application of either approach. As opposed to a QVT language, which can be considered as a general-purpose model transformation language, a DSMTL offers high-level transformation language primitives. Through this raise in abstraction, transformation tasks can be made easier to describe and understand, especially considering the laymen factor. An example for a DSMTL would be some kind of 'refactoring language', that would allow to refactor programs written in a certain object-oriented programming language. For instance, certain common refactorings like changing 'for-loops' into 'while-loops' could be abstracted by that DSMTL. Furthermore, a DSMTL can find application when certain model transformation tasks do not require a full blown query language like OCL, but specialized, recurring queries suffice.

Although existing transformation languages allow defining procedure-like 'helper functions' that can for instance sum up the above mentioned functionalities, these would still be written in a general-purpose language. An important aspect in developing a DSL is to explicitly narrow the scope of a language, and disallow certain actions that would be possible in a general-purpose language. This deliberate infringement can avoid a certain language from being misused in a certain context. Examples for such misused DSLs are various scripting languages, which for instance too often prove malicious on the internet in the form of worms, ad pop-ups and the like.

To give a tangible example, in the following a simple DSMTL applied to a concrete transformation task is introduced. In this respect, domain specificity refers to the notion that the example DSMTL is tailored to be applicable for DSLs used for describing workflows, such as BPEL (BEA, 2003), UML-AD (OMG, 2005), WSFL (Leymann, 2001), XLANG (Thatte, 2001) and the like. Hence, the example DSMTL, which in the following will be referred to as WFTL ("workflow transformation language"), can capitalize on incorporating recurring patterns that emerge when specifying transformations from these workflow languages onto for instance UML Activity Diagrams (Kramler, 2005). To be able to identify such common patterns, the associated workflow languages have to cover certain semantically equivalent concepts (Fig. 1).

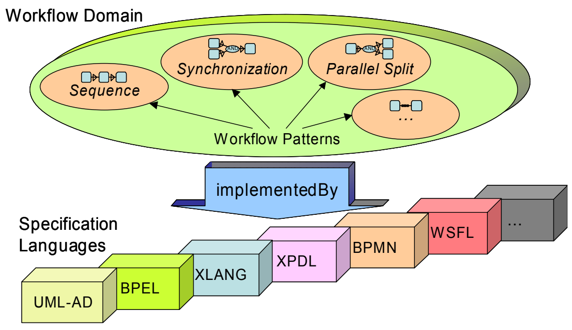

Figure 1: Workflow patterns covered by workflow languages.

Appropriate abstraction mechanisms for the WFTL can for instance be built upon workflow patterns inherent in several workflow languages, such as those proposed by (van der Aalst, 2003), which capture concepts like parallelism, sequential execution, synchronization, and so forth. The incorporation of these patterns can be seen as domain-specific assumptions representing the implicit semantics carried by the syntactic elements of the respective DSMTL. Hence, the DSMTL has to offer syntax that allows to semantically bind the respective concepts in the various workflow languages onto the generic workflow patterns. Taking the WFTL code samples in Figure 2 below, the use of the high-level §SeqExec statement binds all BPEL 'Activities' contained in Sequence, and implicitly maps them onto a semantically equivalent UML-AD representation consisting of ActionNodes connected by Transitions. The Activities of the BPEL Flow are treated analogously.

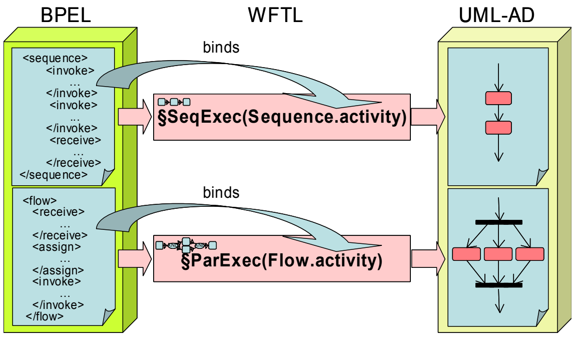

Figure 2: Workflow patterns as High-level WFTL Statements.

Although possible, an attempt to express the high-level constructs as depicted above in a general-purpose model transformation language would doubtlessly be more complicated. An imperative approach to express the §SeqExec pattern for instance, would possibly result in some form of loop iterating over the nodes to establish transitions between them, whereas a declarative definition might be counter-intuitive to find and understand.

Apart from the code snippets above, in the following we discuss further concrete syntax samples of WFTL applied to an actual example transformation from BPEL to UML Activity Diagrams. A transformation definition in WFTL consists of a set of rules, which relate concepts in the source extent with concepts in the target extent. The following excerpts of WFTL transformation code show a transformation rule mapping the BPEL concept of Process to a semantically equivalent UML construct, namely Model.

```
Process2Model                              (1)
…
variables                                  (2)
UseCase UseCase Use_Cases;
…
mapping                                    (3)
…
Process.name =: Model.name;                (4)
…
Model <> UseCase
 A_namespace_ownedElement Core;            (5)
…
Process.activity -> *;                     (6)
```

The transformation rule's title is stated in (1). Following, (2) denotes the instantiation of a UseCase model element (in the UML Use_Cases package) and binds it to the UseCase variable. The mapping (3) keyword specifies that the statements to follow actually facilitate the transformation logic. In (4) a value is assigned from left to right. (5) infers creating a link of A_namespace_ownedElement type (in package Core) between model element instances referred to by the variables Model and UseCase. (6) explicitly initiates the execution of transformation rules on every model element referred to by activity contained in Process.

The above described example shows how a DSMTL can provide specific constructs, in this case for the BPEL-to-UML transformation. With this raise of abstraction, a number of assumptions are taken that finally tailor the language to its intended application area. In the case of WFTL, the focus lies on creating an imperative language with explicit rule execution ordering and the ability to call sub-transformations. Such a stand-point can be reasonably justified when the language's intended target audience has a strong workflow engineering background, and is used to think in an algorithmic, step-wise manner. Although declarative transformation definitions can be less verbose, complex transformations can become somewhat difficult to comprehend, especially for users who are not familiar with the declarative style of 'thinking'.

For reasons of brevity and due to the fact that the introduction of a full-blown transformation language is out of scope of this paper, neither the full language definition (grammar) nor any other transformation rules than the one explained above can be elaborated on in this work. Consequently, we kindly refer the reader to (Reiter, 2005b). Nevertheless, after having motivated the approach for DSMTLs above, the following section will elaborate on our generator framework called *Marius*, which plays a key role in partly automating the realization of DSMTLs.

3 GENERATOR FRAMEWORK

For every DSL developed, an infrastructure making that language executable has to be created as well. The development of DSLs is generally following a so-called source-to-source (Spinellis, 2001) approach, meaning that code written in the DSL is translated into intermediary code of a general-purpose programming language that after compilation eventually realizes the DSL's high-level semantics. To lower the involved effort, frameworks (cf. Section 6) can be employed, which automate some of the effort involved in implementing DSLs. Basically, these language generation frameworks utilize a parser generator and some code-generation facility to produce output code. Thereby, the language developer provides specific semantics by specifying how constructs in the DSL are mapped onto an implementation in the general-purpose language. Analogously, for every DSMTL developed, an executable implementation of this language has to be generated, for which a supporting generator framework is proposed. However, we narrow the focus from providing a generic solution to the implementation of a DSL towards a framework specifically aimed at the implementation of model transformation languages. This means that the proposed generator framework offers specific support for the generation of DSMTLs, which when using generic language frameworks would require considerable adaptation and parameterization effort. For instance, one of the particular assumptions taken about implementing a DSMTL framework is, that every generated transformation language will rely on a number of rules to define transformations. Furthermore it can be assumed, that every such transformation rule will require repository access to query source models and create/update target models. Consequently, such considerations are reflected in the architecture of Marius by specific framework components, which are elaborated on in more detail in section 3.1 to 3.3.

In addition to the above taken assumption concerning transformation languages that mainly specialize the scope of our framework, a number of general design goals influence architecture and implementation related issues, too.

First of all, one of our design goals is to provide a low cost of entry for language definition. Hence, Marius is aimed at generating text-based languages by utilizing EBNF grammars, for which simple text editors suffice. Although nothing impedes the devising of additional visual syntaxes, such undertakings are out of Marius' scope, as DSLs are typically slim, focused languages that often do not necessitate a visual syntax per se. Therefore, support for language definition in terms of MOF 2.0 as requested by the QVT-RFP is out of scope as well. Furthermore, enabling higher-order transformations (transformations of transformation definitions) may be interesting for general-purpose transformation languages, but not applicable for DSMTLs.

Secondly, the transformation logic produced by Marius should be flexible and open. A source-to-source transformation approach that maps DSMTLs onto a general-purpose programming language allows developers to easily analyze, optimize, and customize the intermediary code, should the need arise to incorporate special requirements or external functionalities.

Thirdly, the definition of behavioral semantics for DSMTL constructs should be possible in a comprehensible and maintainable way. Hence, a template-based approach in which code fragments in the intermediary language are defined by the developer and assembled by Marius into a compilable language implementation is taken.

Finally, a design goal is to ensure easy deployment and application of the generated DSMTL implementation. Marius assembles translated DSMTL transformation definitions and auxiliary code pieces, such as repository access code, into a single, executable software bundle, representing a stand-alone model transformer.

Geared towards the above mentioned design goals, *Marius* is a prototype implementation developed for gaining hands-on experience (cf. Section 5) in implementing domain-specific model transformation languages. The remainder of this section deals with describing the *Marius* generator framework from an architectural point of view by explaining how the components it consists of work and interact.

As shown in Figure 3, the framework consists of components either associated with build-time, translation-time or execution-time. At build time, the implementation of a specified transformation language is generated. This means, that tools such as parser and compiler are being built, which during translation time are being used to generate model transformers from transformation definitions at translation time. Finally, at execution time these stand-alone model transformers are applied to models stored in a repository.

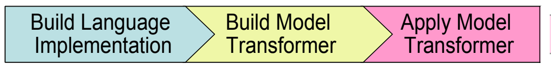

Figure 3: Marius Development Steps.

In the following, the framework's components and the activities associated with them will be described in more detail, according to what 'time' of the development phase they are active.

3.1 Build-time

The starting point for generating a language implementation is to provide a language definition. This is accomplished in terms of providing an EBNF grammar (Fig. 4). This grammar is fed to SableCC (Gagnon, 1998), an open source parser generator, which produces a Java-based parser implementation and tree-walker classes accordingly, which can serve to visit a parsed syntax tree. However, Marius makes use of the parser only and omits using the tree walker classes, as the specification of semantics in the form of Java code (intermediary language), directly in the tree walker source code, suffers from poor readability. Hence, Marius employs a template-based approach for building a 'compiler'. Java Emitter Templates (JET) (Popma, 2004) are used to generate the compilation output in a source-to-source transformation.

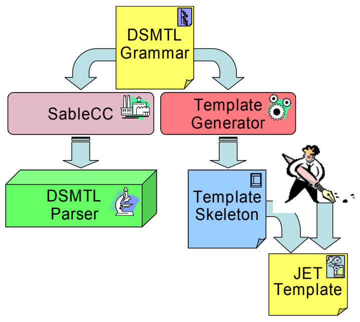

Figure 4: Build-time Component Generation.

Thereby, Marius utilizes a *template generator* which constructs a JET skeleton specific to the DSMTL's EBNF grammar that contains generic functionality to conveniently traverse parse trees and rudimentary method bodies into which a developer enters the desired Java output code.

3.2 Translation-Time

During the translation step (Fig. 5), the parser generated during build-time parses transformation definitions (1) and passes the in-memory syntax tree (2) on towards a JET template engine (3). In the next step, this engine utilizes the JET template (4) specified previously to generate Java source code (5), which represents the executable transformation definitions (6). To ensure maintainability, *traceability comments* are incorporated into the generated code that relate Java code blocks to their counterpart constructs in the DSMTL transformation definition. Furthermore, Marius provides a generic *logging mechanism* preserving a transformation's execution trace by for instance storing corresponding instances concerning source and target model elements. Logging can serve useful for lookup purposes during transformation execution as well as for subsequent debugging purposes. Similarly, Marius lays the way for utilizing specific or external, pre-existing model query mechanisms. For instance, in case of the WFTL samples introduced earlier, a custom *query resolver* is implemented and used. Code enabling to utilize either mechanism is placed into the according intermediary code by the framework as required.

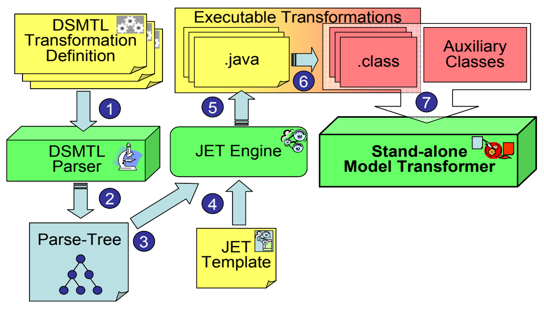
DSMTL Transformation Definition
Executable Transformations
.java
.class
Auxiliary Classes
1
2
3
4
5
6
7
DSMTL Parser
JET Engine
Stand-alone Model Transformer
Parse-Tree
JET Template

Figure 5: Generation of Stand-alone Model Transformer.

It is to note that the parsing process includes a check for syntactical correctness of the transformation definitions. A semantic analysis, however, is not performed. Nevertheless, the use of a non-declared identifier for instance will not result in a translation error, but will subsequently lead to an unsuccessful compilation of the JAVA transformation files.

Finally, after a successful compilation, the resulting class files along with *auxiliary classes* for managing repository access, factory classes for instantiating executable transformations, and generic framework base-classes are bundled into a .jar file, constituting a stand-alone, executable model-transformer (7).

3.3 Execution-Time

As a final point, during execution-time, the generated model-transformer is put to work (Fig. 6). This means, that the transformer is executed against a model repository which holds source and target models and metamodels. In the case of Marius, the NetBeans MDR (Netbeans, 2003) is employed. Repository access functionality is thereby already incorporated into the stand-alone transformer, and its execution will result in the generation of a target model from a source model.

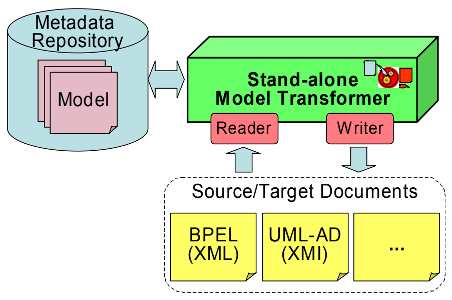

Figure 6: Repository Access & Manipulation.

This involves other components that do not directly belong to the immediate setup of *Marius*, namely *readers/writers for source and target documents*. An example thereof is for instance a reader for BPEL documents, which parses a BPEL source file and instantiates a model accordingly in the repository, requiring that a MOF-metamodel for the BPEL language has been instantiated in the repository prior to that. In our WFTL example, due to the XML-based syntax of BPEL, the actual reader parsing BPEL source files is implemented as an XSLT style sheet for readability reasons.

4 TRANSFORMATION CLASSES

As mentioned earlier, during translation-time a Java-based, stand-alone model transformer is being generated. This section takes a look at the output of the JET engine, which are the Java transformation classes. For every transformation rule, one transformation class is being generated. The structure of these classes is similar, as determined by the template they were produced from.

```
public class Process2Model  implements
Transformation {                       (1)
…
private java.lang.Object UseCase_; (2)
…
public void doMapping() { …          (3)
…
//AName2equTrafo START Line 25,
//Column 0
((RefObject)targetBase)
.refSetValue("name" , source);
//AName2equTrafo END                  (4)
```

```
…

targetPackage.refPackage("Core")
.refAssociation
("A_namespace_ownedElement")
.refAddLink((RefObject)source,
(RefObject)UseCase_);                 (5)
…
source = wrapAttributes(Process_, new
String(".activity"));                 (6)
…
myIt = source.iterator();
…
while(myIt.hasNext()) {
fromArg = myIt.next();
tr = TrafoGenerator.generate(fromArg,
this);
…
tr.doMapping();
…
```

If necessary, the generated Java source files can be manually fine-tuned. The example below shows generated Java code corresponding to the WFTL code sample in Section 2. Note, that for reasons of clarity the shown code omits exception handling and assumes, that the variables target, targetBase, targetPackage, and source have been previously resolved accordingly. In (1), the class for the Process2Model rule is declared, and (2) defines a UseCase_ helper variable. (3) is a framework method used to implement rule execution. Apart from a value assignment, (4) shows how comments are used to establish traces to the original DSMTL statements. (5) establishes a specific link between two model element instances. (6) depicts the resolution of a query (invocation of the wrapAttributes method) followed by an explicit invocation of sub-transformation rules, which are instantiated by a factory (TrafoGenerator).

5 LESSONS LEARNED

This section briefly reports on experiences gained during the implementation of our DSMTL on basis of Marius.

To evaluate the applicability of the approach taken, a case study was carried out which aimed at transforming BPEL process specifications into UML Activity Diagrams. First of all, this involved defining a MOF-based metamodel for BPEL to consequently be able to instantiate and manipulate BPEL documents in a model repository. The actual mapping from BPEL to UML-AD was largely reverse-engineered from (IBM, 2003). It is to note, that BPEL is not a laconic language, which means that to express the same semantic concepts, different syntactic constructs can be used (e.g. 'Flow'). Hence, clean round-trip engineering poses a challenge and possibly requires additional meta-information. However, our main focus was to devise the Marius framework that would allow for developing a dedicated transformation language (WFTL) suitable for the task at hand.

Generally, the creation of a domain specific language can be approached from two sides: A top-down approach will first define an abstract syntax of the language to be realized, whereas a bottom-up approach will start with the writing of concrete syntax samples, of which eventually a language grammar will be 're-engineered'. The latter approach is typically the more explorative, best suited for a rather 'agile' approach to the definition of a transformation language. In iterative cycles, new language constructs are explored. If applicable, these are incorporated into the language definition, which is then used to implement the language by means of constructing a compiler/interpreter making the language executable. If a set of test transformations runs free of errors, the next iteration can begin. During the development of WFTL this approach proved to be very applicable and produced quick results. However, languages developed according to this approach are in danger of getting fuzzy and possibly ambiguous, due to the constant growth of functionality that can result in feature-creep. Therefore, experience gained in the development of WFTL has shown how important it is to refactor the language definition every few iterations, to avoid such problems. One can see, that the bottom-up approach taken for developing a DSMTL, can become problematic when trying to implement too large, wide-scoped, language definitions. However, domain-specific languages have a narrow scope and usually a rather concise language definition, for which the employed approach will typically suffice.

6 RELATED WORK

This section will introduce related work in the field of QVT-like model transformation languages including yet available implementations thereof, and frameworks supporting language generation in general.

For specifying transformations, a distinction between declarative, imperative and hybrid languages can be made. Declarative languages allow to state how the input and the output of a transformation should be made up, without specifying how the execution should go about. On the contrary, imperative languages allow to do just that, as they let a programmer exactly specify how a source model should be transformed into a target model. A hybrid transformation language allows both declarative and imperative constructs to be intermingled, for instance a declarative language such as OCL (OMG, 2003) may be employed for selecting input model elements by means of pattern matching, whereas the specification of possibly complex transformation rules can be facilitated using imperative statements. A good example for such a hybrid approach is ATL (INRIA Atlas, 2004).

From an implementation point of view, various strategies are in use to implement model transformation engines. For reasons of language extensibility and to make it easier to comply with a final QVT syntax, ATL for instance employs a virtual machine that operates on a specific byte-code representation of ATL transformation definitions. MTF (IBM, 2004) falls into the declarative category and relies on an EMF-based (Eclipse Project, 2004) transformation engine that proceeds in two stages called mapping (evaluate relations by iterating model instances) and reconciliation (satisfy relations by deleting/modifying/creating model elements).

Aiming at flexibility, Marius does not enforce a certain style of transformation definition, but leaves it up to the user to specify semantics in a template-based approach. Thereby, a source-to-source transformation into Java allows for further flexibility and customization potential.

MoTMoT (Model driven, Template-based, Model Transformer) is a compiler generating repository manipulation code from visual transformation definitions. The compiler takes models conforming to a UML profile for Story Driven Modeling (SDM) (Shippers, 2004) and generates JMI specific Java code. However, only transformations between models conforming to the same metamodel are supported. Furthermore, MoTMoT does not aim at providing a framework for the generation of DSMTLs in general, but focuses on integration with the Fujaba tool and on making transformations specified in SDM executable.

Microsoft's work on software factories involves the notion of software product lines, which essentially means, that according to a domain-specific language a set of tools is generated to drive the software development process for fast application assembly. Although the principle of software factories aims at the implementation of domain specific languages, its focus is more wide-spread and is neither specific to MOF-based DSLs nor to implementing model transformations engines as the framework proposed in this paper does.

Sprint (Consel, 1998) is a framework for designing and implementing domain specific languages. The development of a DSL is underpinned by an iterative method encompassing several distinct steps, which finally result in an interpreter or compiler. However, the Sprint framework is not immediately applicable for model-driven development, nor does it provide specific support for the implementation of model transformation languages. Furthermore, it seems that Sprint's development method would be a somewhat heavyweight approach, compared to ours, which is rather focused on immediate, template-based realization.

The Kent Modelling Framework (KMF) (Kent, 2004) provides a set of tools to support model driven software development. KMFStudio supports the development of modeling tools, OCL4KMF provides OCL support, and YATL4KMF implements the QVT-like YATL transformation engine. Although the infrastructure provided by KMF can prove highly valuable for implementing MDD tools, to the best of our knowledge the framework does not provide a low-cost entry for the development of DSMTLs through provision of appropriate language definition, parsing and code generation facilities.

The Modeling Turnpike Project (Wada, 2005) 0 (mTurnpike) is a framework that allows to define domain-specific concepts through a combination of both modeling and programming, with the goal to raise the layer of abstraction programmers work on. Although Marius and mTurnpike both build on template-based code generation, mTurnpike is aimed towards model-driven development in general and is not tailored to implement DSMTLs. As an example, mTurnpike does not immediately provide for an EBNF-based grammar definition and the construction of an associated parser.

7 OUTLOOK AND FUTURE WORK

At current a research prototype of the Marius framework exists. As a proof of concept, a transformation language and example transformation definitions for BPEL-to-UML have been created. Future work in terms of extending the existing framework will concentrate on enabling better tool support for the generated languages. This shall also encompass to extend the existing architecture to allow external functionalities and components, e.g. OCL checkers or String manipulation libraries, to be better integrated into the implementation of a language. Furthermore, the framework shall become more flexible and allow to make use of other repository implementations than NetBeans' MDR. The same applies for specifying a model transformation language's semantics not using JMI (JSR 040, 2002) directly, but equivalent, general-purpose QVT constructs.

Apart from overhauling the existing prototype, our future work also focuses on identifying application areas where the implementation of domain specific model transformation languages can prove beneficial. In this respect, some of our current research efforts go into the direction of devising model transformation languages specific to various model integration tasks (Reiter, 2005a).

REFERENCES

van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P. Workflow Patterns. 2003. *Distributed and Parallel Databases, 14(3).*

BEA, IBM, Microsoft, SAP, Siebel, 2003. *Business Process Execution Language for Web Services Specificaion*. Version 1.1.

Bézivin, J., Dupé, G., Jouault, F., Pitette, G., Rougui, J.E., 2003. First experiments with the ATL model transformation language: Transforming XSLT into XQuery. *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture.*

Consel, C., Marlet, R., 1998. Architecturing software using a methodology for language development. *Proc. of the 10th Int. Symposium on Programming Languages, Implementations, Logics and Programs (PLILP/ALP '98).* Pisa, Italy.

Eclipse Project, 2004. *Eclipse Modeling Framework.* http://www.eclipse.org/emf/

Gagnon, E., 1998. *SableCC Java Parser Generator*, Version 2.18.2, http://sablecc.org/

JSR 040, Java Community Process, 2002. *Java Metadata Interface (JMI) Specification*, http://www.jcp.org/

Landin, P.J., 1966. The next 700 programming languages. *Commun. ACM 9 (3), 157-166.*

Leymann, F., IBM, 2001. *Web Services Flow Language.*

IBM Alphaworks, 2004. *Model Transformation Framework*, www.alphaworks.ibm.com/tech/mtf

IBM, Amsden, J., Gardner, T., Griffin, C., Iyengar, S., Knapman, J., 2003. *Draft UML 1.4 Profile for Automated Business Processes with a Mapping to BPEL 1.0.*

INRIA Atlas, Université de Nantes, 2004. *The ATL Homepage*, www.sciences.univ-nantes.fr/lina/atl/

INRIA Triskell, Université de Nantes, 2004. *MTL Engine.* modelware.inria.fr/rubrique.php3?id_rubrique=8

Kramler, G., Kapsammer, E., Retschitzegger, W., Kappel, G., 2005. Towards Using UML 2 for Modelling Web Service Collaboration Protocols. *Proc. of the First International Conference on Interoperability of Enterprise Software and Applications (INTEROP-ESA'05)*, Geneva, Switzerland.

Marschall, F., Braun, P., 2003. Model Transformations for the MDA with BOTL. *Proc. of the Workshop on Model Driven Architecture: Foundations and Applications, CTIT Technical Report TR-CTIT-03-27*, University of Twente.

Netbeans, 2003. *Netbeans Metadata Repository - MDR*, http://mdr.netbeans.org

OMG, 2003. *UML 2.0 OCL Specification, Final Adopted Specification*, www.omg.org/docs/ptc/03-10-14.pdf

OMG, 2002. *Request for Proposal: MOF 2.0 Queries / Views / Transformations RFP.* ad/2002-04-10.

OMG, 2005. *Unified Modeling Language Specification.* http://www.omg.org/uml/

Popma, R., 2004. *JET Tutorial Part 1 (Introduction to JET), JET Tutorial Part 2 (Write Code that Writes Code).* www.eclipse.org/articles/Article-ET/jet_tutorial1.html

Reiter, T., Kapsammer, E., Retschitzegger, W., Schwinger, W., 2005. Model Integration Through Mega Operations. *Proc. of the Int. Workshop on Model-driven Web Engineering (MDWE2005).* Sydney, Australia.

Reiter, T., 2005. *Transformation of Web Service Specification Languages into UML Activity Diagrams*, Master Thesis. ftp://ftp.ifs.uni-linz.ac.at/pub/diplomathesis/ reiter.pdf

Schippers, H., Van Gorp, P., Janssens, D., 2004. Leveraging UML profiles to generate plugins from visual model transformations. *Software Evolution through Transformations (SETra). Satellite of the 2nd Intl. Conference on Graph Transformation.*

Spinellis, D., 2001. Notable design patterns for domain-specific languages. *The Journal of Systems and Software 56, p. 91-99.*

Thatte, S., Microsoft, 2001. *XLANG*, Version 1.0, http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm

Thomas, D., 2003. UML – Unified or Universal Modeling Language? *Journal of Object Technology, Vol 2, No 1.*

University of Kent, 2004. *Kent Modelling Framework*, http://www.cs.kent.ac.uk/projects/kmf/index.html

Wada, H., Suzuki, J., Takada S., Doi, N., 2005. A Model Transformation Framework for Domain Specific Languages: An Approach Using UML and Attribute-Oriented Programming. *Proc. 9th World Multi-Conference on Systemics, Cybernetics and Informatics*, USA.