Towards automated model driven development with model transformation and domain specific languages

Cuong V. Nguyen, Xhevi Qafmolla

***Abstract*— Modeling** plays a very important role in dealing with the complexity of software systems during their development and maintenance processes. As more complex models need to be developed, the importance of transformations between models grows. Model transformations allow the definition and implementation of operations on models, also provide a chain that can enable the automated development of a system from its corresponding models. In this context, approaches to model transformation techniques promise to bring productivity and efficiency to the whole process. This paper outlines practices from current model transformation approaches and the benefit of using domain specific language (DSL) for modeling purposes. We also present the approach of using hybrid transformation with to support automation of model driven development. To achieve automation development we outline a formal approach to testing model transformation with automated test data generation**.**

***Keywords*— Model** Transformation, Domain specific language, Model driven development.

I. Introduction

THE purpose of models is to present a simplified representation of an aspect of the world for a specific purpose. Nowadays, in many complex systems, a lot of aspects need to be considered from architectural to dynamic behaviors, functionalities and user interfaces. The design process can be described as the weaving of all these aspects into a detailed design model. Model-driven methods aim at automating this weaving process. Model-driven engineering (MDE) is a software development methodology, which focuses on creating models, or abstractions of something that describes the elements of a system. It is meant to increase productivity by maximizing compatibility between systems, simplifying the process of design and promoting communication between individuals and teams working on the system [4].

Model Driven Architecture (MDA) aims to separate application structure PIM (Platform Independent Model) from its functionality, PSM (Platform specific Model). The mapping between these models is realized by model transformation. The problem of model transformation based on Meta-Object Facility (MOF) can, then, be stated in the following way: “Given a source model ‘m1’ described by a meta-model ‘MM1’ we define an automatic process making it possible to obtain a model ‘m2’ conforming to a meta-model ‘MM2’; ‘MM1’ and ‘MM2’ being MOF compliant” [10]. Model transformations require specialized support in order to realize their full potential. Nowadays, there are still open issues in their foundations, semantics, structuring mechanisms that demand further research and study. Model transformations also require methodological support to integrate into existing software development practices.

In the first part of this paper we outline the classification of current model transformation approaches and give several reviews on the application of these methods. We then introduce the approach of using ATL to develop model transformation. In the next part we present the approach to testing of model transformations and discuss important issues that need to be solved to achieve a good and applicable technique for testing model transformations.

II. Classification of current model transformation approaches

To help making a decision of choosing the appropriate model transformation approach that is best suited for the requirements of a project, we need to have a comprehensive overview of transformation classification. This classification not only helps people in the domain of interest but also helps vendors and tool builders in identifying the advantages and weaknesses of their tools compared to others, researchers can also identify the limitation of a technology and improve current methodologies and formalisms. Based on the study of current model transformation technologies, the following sections depict the classification and overview on the previous work and methods in model transformation.

A. Graph transformation

Generally, (meta-) models are represented in UML

Cuong V. Nguyen is with Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, Prague, Czech Republic (phone: +420 777-102-882; e-mail: nguyevie@fel.cvut.cz).

Xhevi Qafmolla is with Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, Prague, Czech Republic (phone: +420 608-070-472; e-mail: qafmolla@fel.cvut.cz).

formalism. As a result, the models can be viewed as graphs. It is therefore natural to consider the use of graph grammars to express model transformation [6]. Graph transformation approaches are very well built theoretically. Graph transformations are usually based on matching and replacement strategy. They are based on syntactic graph rules that consist of finding a Left Hand Side graph and replacing it by a Right Hand Side graph. This approach has the power of a clear operational idea, which enhances rule specification. The complexity of this approach stems from its non-determinism in scheduling and application strategy, which requires careful consideration of termination of the transformation process and the sequence of rule application [4].

Early work involving graph transformation and models largely centered on their use in defining the semantics of different modeling diagram types, such as the continuing work of Gogolla M., Ziemann P., Kuske S. [6]. More recent work by Kuster J.M., Heckel R., Engels G. [8] has defined a more general model transformation approach using graph transformation as the underlying mechanism, allowing them to draw upon some of the properties of graph transformations in a model transformation context. Heckel R., Kuster J.M., Taentzer G., [7] have continued this work, reasoning about confluence with typed attributed graphs. Braun P. and Marschall F. [2] have proposed model transformation approaches which are essentially based upon simplified views of graph transformations, as is Agrawal A., Karsai G., Shi F.'s more mature GReAT system [1].

Although graph transformations have several interesting properties when applying to model transformations, it is still not used widely in practical situation due to the complexity and lack of structuring mechanisms. Solutions based on the graph transformation paradigm therefore have relatively little real-world usage [9].

B. API approach

This type of transformation is based firstly on Meta Object Facility specification. MOF is used in many modeling tools to create model repositories. After that Application Programming Interfaces (API) are generated for each supported meta-model. These interfaces are used to describe the model transformation process by means of programs written in an imperative language: Java, C++, etc. This approach provides the user with a set of interfaces used to describe the transformation process as a series of instructions that allow the generation of a target model from a corresponding source model. The use of APIs to describe a transformation process is a powerful solution because programming languages generally have good performance at runtime. Basically, the user must perform the entire procedure: he is in charge of the organization and description of all stages, explicitly in terms of imperative statements [4].

C. XSLT approach

Along with XML technology, XML Metadata Interchange (XMI) enables the exchange of meta-models as a standard. There is a need for bridging between XML processing and other form of data and a language for that purpose is in demand. XSL stands for EXtensible Stylesheet Language and XSLT stands for XSL Transformations. As models are described in XML format, it appears that EXtensible Stylesheet Language Transformation (XSLT) is a convenient solution for model transformation. XSLT is an appropriate standard for XML document transformation, but suffers from limitations in realizing model transformation. Moreover, XSLT data types are limited; this restricts the scope of information that must be computed during the transformation process. In a DTD, the syntax and the semantics of an XML documents are fixed, and transformation rules therefore have to deal with both [4].

The main weakness of XSLT lies in the fact that it was adequate for the simple transformations but has serious shortcomings for more advanced transformations. Recently, a formal proof was constructed that XSLT is Turing complete. However it took several years before that was proven and in practical usages the limits in XSLT make it harder to conveniently apply this approach. A final issue, which makes expressing model transformations in XSLT less than ideal, is that XML documents are represented as a tree structure; models are, in the general case, naturally describable as graphs. Although graphs can be represented by trees with link references between nodes, the difference in representation can lead to an unnatural representation of many types of model transformations [9].

D. Declarative approach

In declarative approach, the relationship between concepts in the source and the target meta-model is defined by patterns. The transformation is defined by a set of rules. A rule lays forth a pattern of source model concepts, which is then transformed into a set of elements in the target model. The sequence of the various stages of the transformation process is controlled by the user, thanks to operators that allow the carrying out of explicit transformation rules invocation. The implementation is realized by an inference engine [5]. However, one of the disadvantages of this approach is the significant amount of work burden the developer with specifying all the constraints supporting the transformation.

E. Imperative approach

As similar to imperative programming, imperative approach works in the paradigm that describes the transformation in terms of statements that change the program states. Imperative approach defines a sequence of commands to perform. An example of this approach is Transformation Rule Language (TRL). This language is in essence a standard rule-based imperative language specialized for UML-esque model transformations. This comes in several forms:

- Some of the information recorded in the new first class elements is used for additional purposes such as to create tracing information.
- Extra syntax is provided for accessing the stereotype of a UML model element.

Rules consist of a raw signature (works as the declaration of the types of the source and target model elements) and an imperative body. In TRL, the syntax and semantics of actions are similar to that of the Object Constraint Language (OCL). Moreover, there are additional control structures and side effects that add up to the syntax of TRL that makes it adequate for building model transformations. The benefit of such an approach is its relative familiarity to users, and the knowledge that largely imperative solutions traditionally lead to efficient implementations [1]. However language such as TRL on the other hand is only capable of expressing unidirectional stateless transformations, due to the imperative nature of rule actions.

F. Hybrid approach

In a declarative approach, a transformation is defined by a set of relations between the concepts of the source and target models, as described in their meta-models. The implementation is realized by an inference engine, which allows the application of the transformation to generate the target model. In an imperative approach, a transformation is described by a set of algorithms as functions or procedures that explicitly describe the sequence of transformation applications. Hybrid approaches combine the declarative and imperative approaches. The declarative approach is generally used in the definition and selection of the transformations which can be applied, while the imperative approach is well adapted to describing the transformation strategy by a control flow of execution rules, and hence to executing the transformation [4]. In hybrid languages, transformation rules mix the syntax and the semantics of the concepts they handle. ATLAS Transformation Language (ATL) as an example implements imperative bodies inside declarative shell to specify transformation. Hybrid approaches potentially have the advantages since they combine both values of declarative and imperative ones. The next section illustrates how ATL can be used to construct transformations.

III. Domain Specific Modeling Languages for Modeling

Introducing a new DSL can set up the stage for automatic generation of a part of the model artifacts and code, such as Java code, that implements the services. A general modeling language could also be used for this purpose but an appropriately designed DSL will perform the same job much more effectively.

A good domain specific modeling language needs to have several requirement features. To develop the modeling language for our purpose, the list bellow provides the most important ones. All given features should be considered during the creation of a DSL to pledge the completeness of the language.

Effectiveness: The language has to be able to deliver useable output without having to re-tailor based on specific use case. This means that the language is able to bring up good solution on specific domain and focus on solving the particular range of problems. Effectiveness also needs to guarantee the unambiguity feature of language expressions and capability to describe the problem as a whole from a higher level.

Automation and Liveness: As the modeling language can raise the level of abstraction away from programming codes by directly using domain concepts, an important aspect is the ability to generate final artifacts from these high-level specifications. This automation transformation has to fit the requirements of the specific domain. Liveness feature ensures changes from models described by the language are propagated to the next phase of development automatically.

Support Integration: The DSL has to be able to provide support via tools and platforms. The DSL needs to be able to integrate with other parts of the development process. This means that the language is used for editing, debugging, compiling and transformation as well as integrated together with other languages and platform without any heavy effort.

Once developed, a DSL can reduce the cost implied in maintaining software. In comparison to other techniques DSL is considered as one of main solutions to software reuse. On the other hand, using DSL also promotes program readability and makes program understanding easier because it is written at good abstraction level. It enables users without experience in programming, only knowledge of the concerned domain is needed to write the program. Another advantage of a DSL for modeling is the ability to generate more verification on the syntax and semantics than a general modeling language.

IV. Using ATL For Model Transformation

In this section we demonstrate the hybrid approach by using ATL. In this context, ATL can be considered as a domain specific language for model transformations. For working with ATL as a transformation language, ATL Integrated Environment was used. This environment provides a number of standard development tools that aims to ease development of ATL transformations. An ATL transformation is composed of rules that define how source model elements are matched and navigated to create and initialize the elements of the target models. To understand the concepts, the architecture of model transformation could be depicted in the following figure:

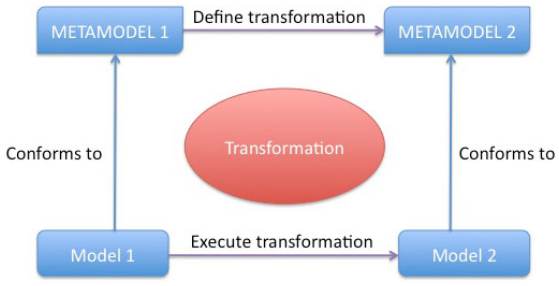

Fig. 1 Architecture of model transformation

To start building ATL transformation, we use a simple use-case of metamodel transformation from a source metamodel

M1 to target metamodel M2. The graphical presentation of M1 and M2 are depicted in fig. 2.

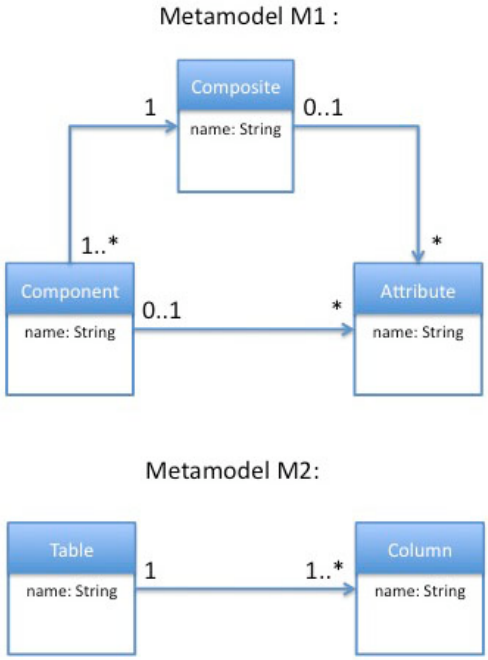

Fig. 2 Metamodels M1 and M2

Our goal is to construct the transformation from an instance I1 of metamodel M1 to instance I2 of metamodel M2 given that I1 conforms to metamodel M1 and I2 must conform to metamodel M2. Before defining the transformation itself, we need to define the source and the target metamodel M1 and M2. We use KM3 or Kernel Meta Meta Model, a neutral language that is convenient to describe metamodels and to define Domain Specific Languages for this purpose. KM3 is available under the Eclipse platform and could be easily used with ATL. The next step is to build the transformation using ATL, the transformation is defined as a module:

```
module M1toM2;
create OUT: M2 from IN: M1;
```

Inside the module, we describe the transformation process. Conceptually, the transformation process could be simplified as follows:

1. For each instance of class Composite in the IN model, create an instance in the OUT model.
2. For each instance of class Attribute we create an instance of Column in the OUT model.
3. Name of column in the OUT model is defined as the attribute name.

The transformation is built as the set of rules and helpers. A helper is an auxiliary function that computes a result needed in a rule. A rule for transforming the name attribute from the source model to the target model for our transformation is defined in ATL as:

```
rule Attribute2Column {
  from s : M1!Attribute
  to t : M2!Column (
     name <- s.name
  )
}
```

Similar process is applied to create all the rules needed for the transformation. Once the ATL transformation is created, the result of its execution will create the OUT model. ATL combines both declarative approach and imperative approach such that the declarative part is generally used in the definition and selection of the transformations which can be applied, while the imperative approach is well adapted to describing the transformation strategy by a control flow of execution rules, and hence to executing the transformation.

V. Testing of Model Transformation

As an important factor for automation in the development cycle, development of model transformations should be conducted according to standard software engineering principles. Hence, transformations need to be validated by some testing methods or else developed within the software development lifecycle.

However, currently in the domain, there is still lack of adequate techniques to support model transformation testing: testing techniques for code do not immediately carry across to the model transformation context, due to the complexity of the data under consideration [10]. To build test cases for model transformation, we need to have model instances that conform to a given metamodel. These models have to satisfy the precondition of the transformation's specification and additional constraints employed to target particular aspects of the implementation's capability.

The proposed approach aims to build a process that automatically generates a set of test models that satisfy the constraints. There are several strategies that can be used to build these models. Depending on each case, we should choose the values for the properties in a particular range; decide when objects go together in one model or when a new model should be generated. There might not be one strategy that is the best in every case. The approach in general can be described in the following way:

1. Decompose the source metamodel into more simple classes.
2. Create model parts base on simple classes decomposed in the previous step.
3. Generate complete test model from parts and metamodel specification.

One drawback of completely automatic generation of test model is the complication for testers in reading and comprehending the model. To reduce the difficulty when interpret the obtained models by a human tester, more interactive phase should be introduced. This helps the tester to better understand the input model and provide more precise data that is closer to the test requirements. To obtain a good and practical model transformation testing technique, we need to have a clear and well-defined specification that can verify the quality of testing data. An effective algorithm to build test models from the input metamodel also plays an important part in enhancing the quality of test.

VI. Conclusions

In this paper, we outlined the overview of current model transformation approaches and outlined some common weaknesses and advantages. In imperative model transformation approach, such as TRL, new elements in the target model are explicitly created; this weakens the ability to propagate changes. From the authors' perfective, declarative solutions assure no duplicates are created in such a situation. This method works well for both initial transformations and for subsequent updates. Hybrid approaches combine the declarative and imperative approaches promise to bring a lot of advantages. To demonstrate this, we introduced the usage of ATL as a hybrid transformation approach to implement model transformation. We also outlined the usage of DSL to reduce the cost implied in maintaining software. In comparison to other techniques DSL is considered as one of main solutions to software reuse and promotes program readability and makes program understanding easier because it is written at good abstraction level. For supporting automated model driven development, we also discussed an approach to testing of model transformation and identified important challenges to make model transformations dependable. From practical viewpoint, presenting more interactive phase to automatic generation of test model could bring more benefits and increase effectiveness by making model more readable and easily adaptable in the future.

It is not easy to decisively conclude which of many different approaches, if any, is the most promising. Therefore one of the main challenges for the community is simply to continue exploring different approaches to model transformations. The authors' future work and research will focus on building model transformation with hybrid and declarative approach. This will also include the research on DSL approaches to provide testing and validation of model transformation.

Acknowledgment

This work has been supported by the Department of Computer Science and Engineering, Faculty of Electrical Engineering and by the grant project of Czech Technical University in Prague number SGS12/074/OHK3/1T/13.

References

[1] Agrawal A., Karsai G., Shi F., “Graph transformations on domain-specific models”, Technical report, Institute for Software Integrated Systems, Vanderbilt University, Nov 2003.

[2] Braun P., Marschall F., “Transforming object oriented models with BOTL”, International Workshop on Graph Transformation and Visual Modeling Techniques 72(3).2002.

[3] Camillo F., Alberto M., Mario O., Iman P., “A constructive approach to Testing Model Transformation” Proc. Third International Conference on Model Transformation (ICMT2010), Springer, Jul. 2010, pp. 77-91, ISBN 978-3-642-13687-0.

[4] Dehayni M., Barbar K., Awada A., Smaili M., “Some Model Transformation Approaches: a Qualitative Critical Review”, Journal of Applied Sciences Research,Vol.5 No.11, Nov 2009, , pp. 1957-1965.

[5] Fleury, F., Steel, J., Baudry, B. “Validation in model-driven engineering: Testing model transformations”, Proc. MoDeVa 2004 (Model Design and ValidationWorkshop associated to ISSRE 2004), Nov 2004.

[6] Gogolla M., Ziemann P., Kuske S., “Towards an integrated graph based semantics for UML”, Proc. Int. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2002), Electronic Notes in Theoretical Computer Science, vol 72, 2003.

[7] Heckel R., Kuster J.M., Taentzer G., “Confluence of typed attributed graph transformation systems”, Proc. First International Conference on Graph Transformation (ICGT 02). Springer-Verlag, Oct 2003, pp 161-176.

[8] Kuster J.M., Heckel R., Engels G. “Defining and validating transformations of UML models”, Proc. IEEE Symposium on Visual Languages and Formal Methods, Oct 2003.

[9] Laurence Tratt, “Model transformations and tool integration”, Journal of Software and Systems Modelling, vol. 4(May 2005), pp. 112-122.

[10] Wikipedia, “Model Driven Engineering”, http://en.wikipedia.org/wiki/Model_Driven_Engineering, (accessed 14.12.2009).