A comparative look at model transformation languages

Matthew Stephan
Queen's University
Kingston, Canada
matthew.stephan@queensu.ca

Andrew Stevenson
Queen's University
Kingston, Canada
andrews@cs.queensu.ca

ABSTRACT

Model-driven development is an emerging area in software development that provides a way to express system requirements and architecture at a high level of abstraction through models. It involves using these models as the primary artifacts during the development process. One aspect that is holding back MDD from more wide-spread adoption is the lack of a well established and easy way of performing model to model (M2M) transformations. We propose to explore and compare popular M2M model transformation languages in existence: EMT , Kermeta, and ATL. Each of these languages support transformation of Ecore models within the Eclipse Modeling Framework (EMF). We attempt to implement the same transformation rule on identical meta models in each of these languages to achieve the appropriate transformed model. We provide our observations in using each tool to perform the transformation and comment on each language/tool's expressive power, ease of use, and modularity. We conclude by noting that ATL is our language / tool of choice because it strikes a balance between ease of use and expressive power and still allows for modularity. We believe this, in conjunction with ATL's role in the official Eclipse M2M project will lead to wide-spread use of ATL and, hopefully, a step forward in M2M transformations.

Categories and Subject Descriptors

D.2.12 [**Software Engineering**]: [Interface definition languages]

General Terms

Languages

Keywords

Model transformations, Model-driven development, model transformation languages, meta modeling

1. INTRODUCTION

Model-driven development(MDD) is a paradigm in which software is constructed in a model-centric fashion. Models are the main artifacts of interest to stakeholders and system developers and the underlying code that implements the system is generated from these models. In a perfect model-centric development project, models are the only elements that change as the implementing software code is generated and updated automatically.

One large area of importance in MDD is the notion of model transformations, that is, the ability to transform one model to another through modifying the model itself or by creating a new version of the model. Model transformations can be split into two categories, model-to-model transformations and model-to-text/code transformation [5]. The model-to-text transformation task, sometimes referred to as code generation, is relatively more mature and explored compared to model-to-model(M2M) transformations.

Progressing in the area of model-to-model transformations is crucial in continuing to foster the adoption of MDD as noted in [5, 9] as these are important operations. Although there are many classifications of model transformation approaches [3], all with much promise, it is still unclear which approach is the best for M2M transformations in practice or if some are best suited to specific contexts.

In this paper, we utilize and evaluate 3 fairly well known M2M model transformation languages each belonging to a different category of M2M as defined in [3]. The languages that we compare, along with their corresponding categories, are EMF Model Transformation Framework [1], a graph-transformation based approach; KerMeta [7], a direct-manipulation approach; and Atlas [6] (ATL), which is a hybrid approach and is part of the official Eclipse M2M project.

Section 2 begins by first providing a brief summary of M2M transformations and Eclipse Modeling Framework (EMF) Ecore models and continues with the M2M approaches that we discuss in the remainder of the paper. Section 3 outlines the procedure that we will follow in order to perform the comparison, which includes a discussion of the M2M transformation that will be attempted on all 3 languages and the various dimensions that the 3 approaches will be compared on. Section 4 presents the observations we make when using each of the respective approaches to accomplish the M2M transformation. The paper concludes with Section 5, in which we summarize our observations and chose which tool we consider the best given our criteria.

2. BACKGROUND

This section provides brief background information on M2M transformations, EMF Ecore, and the three languages we use.

2.1 M2M Transformations

M2M transformation refers to the process of modifying a model to create either a new model or update itself [8]. Generally model transformation can be seen as an operation that takes existing model elements, follows some guidelines or rules as to what should be modified and how it should modified, and then produces either an entirely new model containing appropriate elements or the modified version of the model with the new/modified elements. Model-to-model transformations involve the process of going from platform-independent models (PIM)s to either other PIMs or from PIMs to platform-specific models (PSM)s not in the form of code.

2.2 Ecore

Ecore [10] is Eclipse's implementation of the Essential Meta-Object Facility (EMOF). It provides the M2 layer, which is the Ecore language that allows modelers to develop meta models, called Ecore models. Ecore models are the M1 layer and can be used to specify systems or a family of systems that can later be instantiated (M0 layer). That is, Eclipse's implementation allows users to develop Ecore meta models (M1 layer) to allow for instantiation of system models (M0 layer) that can be developed within an Eclipse application. A completed Ecore (meta) model allows for code generation, validation, and instantiation of Ecore instance models. It is widely used in both industry and research and all three of the model transformation languages discussed in this project work with Ecore models.

2.3 Languages Used

The three modeling transformation languages used by us for this project all work with Ecore models as we figure this will provide a common foundation for comparison. We attempt to select languages that are prevalent in research and industry, based on quantity of documentation and other information sources, and that are different enough that a comparison will exhibit the benefits and drawbacks of various approaches to achieving M2M transformation.

2.3.1 Atlas Transformation Language

The ATLAS Transformation Language (ATL) is one of three transformation engines within the Eclipse Model-To-Model subproject, which, itself, is part of the Eclipse Modeling Project. The other two engines are based on the Query/View/Transformation (QVT) standard set by the Object Management Group, but their implementations are still relatively immature so we consider them out of scope for the work in this paper.

ATL is a ruled-based domain-specific language that describes transformations from one Ecore metamodel to another Ecore metamodel. It supports an interesting mix of declarative and imperative language paradigms, and provides many useful built-in collection operations such as iterators, filters, and common set operators. An ATL program is composed of rules that indicate how model elements are identified and subsequently modified in order to obtain the target model.

Figure 1 illustrates a sample rule we have taken from the ATL user guide. The rule has a source and a target metamodel, MMAuthor and MMPerson, respectively. The *Author*s within the MMAuthor metamodel, denoted by !Author following the source Metamodel name, are represented by the variable a. *Person*s in the target metamodel, bound to variable p, are new model objects that have their features *name* and *surname* set to name and surname found in the corresponding Author object.

There are three types of rules in ATL: Matched rules, such as the one presented in Figure 1 that are automatically called; lazy rules, which are ones only executed when called by another rule; and called rules, which do not include a source pattern and may contain optional local variables/parameters. There are two types of execution modes in ATL: the normal mode and the refining mode. The refining mode is used to focus on specific transformations the program, for example, to transform one source model element into one target model element and focus while leaving the rest implicitly copied.

```
rule Author {
        from
                a : MMAuthor!Author
        to
                p : MMPerson!Person (
                        name <- a.name,
                        surname <- a.surname
                )
}
```

Figure 1: Sample ATL Rule

2.3.2 EMF Model Transformation Framework

The EMF Model Transformation (EMT) framework provides an entirely graphical interface to describe in-place transformation rules on an Ecore model. It includes a compiler and a code generator so that the transformations created within it can be reused. EMT is built upon AGG [11], which is a graph transformation tool.

As shown in Figure 2, the Eclipse editor is divided into Left-Hand Side (LHS) and Right-Hand Side (RHS) panes that show how model elements are structured before and after the transformation. Optionally, a negative application condition (NAC) pane can be used to show the one or more situations where a rule should not be applied. Model elements in the LHS can be mapped to corresponding elements in the RHS and NAC section to indicate their identity, and are represented visually by having the same colour in those panes. Unmapped elements in the LHS are deleted in the transformation and unmapped elements in the RHS are created. The example in Figure 2, is an example of a transformation that will move all *Eclass*es with *name* equal to 'n' to the package with its *name* equal to 'p'. The NAC pane indicates that any packages already containing the Eclass to be moved can be ignored. EMT has been

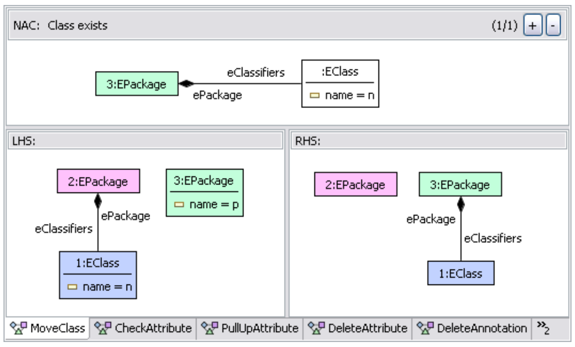

Figure 2: Move Class Example [1]

approved for inclusion in the Eclipse Modeling Framework under the name EMF-Tiger, with an initial release scheduled for Summer 2010.

2.3.3 *KerMeta*

Kermeta is a language intended for meta modeling but provides many features found in a general-purpose programming language. It allows for navigation and manipulation of a model in an object-oriented way with a syntax heavily influenced by Eiffel. Like ATL, KerMeta is strongly typed and supports built-in collection modifiers. M2M model transformation is accomplished in KerMeta by using actual instances located within the target model in conjunction with those from the source model. Users then implement a transformation model that uses the elements in both the source and the target meta models to create the desired elements in the target meta model.

Figure 3 provides a snippet of some model transformation code used in the example from [7] of transforming object oriented classes to database tables. It contains elements from both the source metamodel, for example classes bound to the *c* variable, and the target metamodel, the *table* variable.

3. PROPOSED PROCEDURE

As a use case we decided to transform a tree model into a list model. For the source and target meta models, we use the ones provided in the ATL example [4] but do not follow the rest of the example. That is, we develop the ATL transformation independently. We choose this example because it represents the common case where a software project wants to change the underlying platform-independent implementation. Without a transformation language or approach to accomplish this it can be quite difficult to do manually. Using a common use case for these tests ensures that any differences we observe will be a result of the transformation tool and not due to a difference in the source or target meta models.

The bulk of the testing phase will likely be taken up learning how to use each tool and in particular how to express our desired transformation in each tool's syntax. However, this aspect of learning is also important as it will help show the ease of adoption of each of the tools. Running the tests should be a relatively simple task of invoking the transformation engine within each tool. The output of

```
class Class2RDBMS
{
        /** The trace of the transformation */
        reference class2table : Trace<Class, Table
        >
        /** Set of keys of the output model */
        reference fkeys : Collection<FKey>
        operation transform(inputModel :
        ClassModel) : RDBMSModel is do
        // Initialize the trace
        class2table := Trace<Class, Table>.new
        class2table.create
        fkeys := Set<FKey>.new
        result := RDBMSModel.new
        // Create tables
        getAllClasses(inputModel).select{ c | c.
        is_persistent }.each{ c |
                var table : Table init Table.new
                table.name := c.name
                class2table.storeTrace(c, table)
                result.table.add(table)
        }
        // Create columns
        getAllClasses(inputModel).select{ c | c.
        is_persistent }.each{ c |
                createColumns(class2table.
                getTargetElem(c), c, "")
        }
        // Create foreign keys
        fkeys.each{ k | k.createFKeyColumns }
        end
        [...]
}
```

Figure 3: Sample KerMeta Code

the transformation will then be compared to our expected output. If they do not match, the transformation rules may need to be debugged if we notice they do not accurately express the intended transformation.

3.1 Method of Evaluation

Expressing model transformations well is a difficult problem because of the complexities involved. The transformation writer needs to have a mental map of both the source and target meta models simultaneously and visualize the transition from one to the other. The transformation language should do as much as possible to make this task easier. We focus on three independent evaluation areas for comparison:

1. Expressiveness: The expressiveness of the language determines the complexity of the transformations that can be performed by the tool. We expect all the languages to have sufficient expressive power to represent the transformation we wish to perform, but we will also examine language elements that provide additional expressiveness. For example, we will investigate whether each language has the ability to apply transformations recursively to newly created model elements. A tool that supports the choice to execute transformations recursively would be more expressive than a language that forces one option in all cases.

2. Ease of Use: The simple design and small learning curve of a tool is important to its widespread acceptance, especially in the field of model-driven development where the chain of tools used, that is, compilers, interpreters, parsers, IDEs, modelers, meta-modelers, is already extensive. For example, is it more natural to use a graphical user interface or syntax to express model transformations? For syntactic transformation languages we will be considering whether the structures and concrete syntax are intuitive and lend themselves well to expressing transformations. Finally, we will investigate the cohesion between the various tools and the Eclipse/EMF framework. A tool that integrates well into Eclipse and reuses concepts and terminology familiar to EMF modelers will make it much easier to learn compared to one that doesn't leverage existing EMF concepts.

3. Modularization: Cuadrado and Molina [2] suggest that modularization is an important criteria for a model transformation language because it promotes reusability and helps manage complexity. Modularization can come in many forms and depends on the nature of the transformation language used. Modularization constructs can be built into the transformation language itself, or modularity can be achieved through adopted practices such as the way transformations are decomposed.

These evaluation criteria above are of a qualitative nature rather than a quantitative one, but taken together should provide a good insight into the maturity and effectiveness of the various transformation tools.

4. OBSERVATIONS

The following sections detail our impressions and experiences working with ATL, EMT, and Kermeta in light of our evaluation criteria.

4.1 Atlas Transformation Language

s One immediate positive we found in using ATL is the complexity of model transformations can be managed by ATL in several ways, the most basic being the use of helper functions. As the name implies, helper functions are used by modelers to modularize transformation rules by hiding periphery details in a separate function and calling that function within the rule body. This greatly increases the readability and comprehension of the transformation rule and is often used to iterate and collect elements from a repeating or recursive model structure.

Much of the power behind ATL comes from its imperative language constructs, but we found that this makes the transformation code harder to read and understand. The philosophy of ATL is to avoid these constructs in favour of splitting the complex transformation into two or more simpler transformations that run in sequence. By having modelers transform the source model first into an intermediate model and then into the target model allows for the complexity of the transformation to be divided among the various transformation steps and can then be expressed using the more desirable functional constructs of ATL.

The distribution of ATL comes with a transformation debugger and, similarly to Eclipse's Java debugger, it allows the user to set breakpoints and view the variable values of the currently executing thread. The debugger reflects the implementation of the ATL resolve algorithm rather than a model-level view of how the transformation is evolving. The debugger is beneficial if the user is already familiar with the ATL resolve algorithm or if they want to learn how it works but seems removed from the metamodel that the user will be more familiar with. It does, however, provide excellent correspondence between the ATL statements and the execution state of the transformation engine.

ATL is a well-rounded transformation language, with an easy to use declarative syntax and a more expressive imperative syntax. Modularity is achieved using helper functions and transformation staging.

4.2 EMF Model Transformation Framework

We found EMT's graphical rule editor to be far more intuitive and comprehensible than a syntax-based transformation language. The user interacts with elements of a model familiar to them and the visual differences between the LHS and RHS are far more concrete than mentally interpreting a syntax-based rule and visualizing how it affects the model. By eliminating this cognitive burden from the user, EMT's visual editor allows users naturally sto grasp and manipulate more complex transformations easier than they could with a similar transformation expressed in a syntax-based transformation language.

EMT is best suited to when the source and target meta models are the same. It is possible to write transformation rules from one metamodel to another, but this requires the user to first create a reference metamodel that relates the source metamodel elements to the target metamodel elements.

Unfortunately EMT does not currently have supporting tools such as a debugger, profiler, test framework, et cetera. The background work required to interpret and run the transformation rules must be written by the developer in Java manually and the feature to automatically generate code to run the transformation appears to be broken. With these difficulties, we were not able to run our transformation rules in the current build.

The EMT editor is easy and natural to use, but the tool support and expressiveness leave a lot to be desired. These issues should improve as the tool matures.

4.3 Kermeta

Kermeta is a powerful language but takes a while to pick up. Unlike ATL and EMT, the metamodel is defined within the Kermeta language itself rather than in a separate Ecore file, although, an Ecore-equivalent Kermeta model is available to help bridge the two formats.

Kermeta transformations are achieved using its object-oriented model navigation commands and imperative language constructs. The lack of explicit transformation rules sets it apart from both ATL and EMT. As a model transformation language Kermeta is expressive and powerful, however it lacks the simplicity of a rule-based language where the before and after model is clearly defined.

The programming-language nature of Kermeta allows it to utilize common modularization paradigms from existing programming languages, such as object-oriented programming, aspect-oriented programming, and generics. All these techniques can be brought to bear on the model transformation problem to help reduce complexity and increase understandability.

Kermeta's more mature programming library also includes useful tools such as a unit test framework called KUnit, based on Java's JUnit framework. A test suite can be created to exercise and verify model transformations, which is especially important for regression testing if the metamodel or model constraints should change.

5. CONCLUSIONS

We compared briefly three model transformation tools, ATL; EMT; and Kermeta, for expressiveness, ease of use, and modularity by attempting the same transformation in each language. Each tool had some positive and negative characteristics. As a fully fledged programming language, Kermeta is powerful and expressive, but takes the longest to learn and was not designed with transformations in mind. EMT's visual editor is very easy to use but the lack of automatic build tools makes running the transformations problematic. As a rule-based language with declarative and imperative constructs, ATL strikes a good balance between ease of use and expressiveness. All three tools showed instances of modularity, albeit in different forms. Of all three tools used to perform the transformation, we choose ATL as the best because it is not overly difficult to use and is still quite powerful. We found the modularity to be relatively equal to the other two tools. Seeing as ATL is being integrated into official Eclipse's M2M project stream, we believe that it will eventually establish itself as a prevalent approach to perform M2M transformations and will hopefully prove to be the next step forward in M2M transformations and MDD adoption.

6. REFERENCES

[1] E. Biermann, K. Ehrig, C. Kohler, G. Kuhns, G. Taentzer, and E. Weiss. Graphical definition of in-place transformations in the eclipse modeling framework. *Lecture Notes in Computer Science*, 4199:425, 2006.

[2] J. Cuadrado and J. Molina. Modularization of model transformations through a phasing mechanism. *Software and Systems Modeling*, 8(3):325–345, 2009.

[3] K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, 2003.

[4] C. Faure and F. Allilaire. The tree to list example. `http://www.eclipse.org/m2m/atl/basicExamples_Patterns/article.php?file=Tree2List/index.html`, July 2007.

[5] A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood. Transformation: The missing link of MDA. *Lecture notes in computer science*, pages 90–105, 2002.

[6] F. Jouault and I. Kurtev. Transforming models with ATL. *Lecture Notes in Computer Science*, 3844:128, 2006.

[7] P. Muller, F. Fleurey, D. Vojtisek, Z. Drey, D. Pollet, F. Fondement, P. Studer, and J. Jézéquel. On executable meta-languages applied to model transformations. In *Model Transformations In Practice Workshop*, 2005.

[8] I. Porres. Model refactorings as rule-based update transformations. *Lecture Notes in Computer Science*, pages 159–174, 2003.

[9] B. Selic. The pragmatics of model-driven development. *IEEE software*, 20(5):19–25, 2003.

[10] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF Eclipse Modeling Framework*. Addison Wesley, second edition, 2009.

[11] G. Taentzer. AGG: A graph transformation environment for modeling and validation of software. *Lecture Notes in Computer Science*, 3062:446–453, 2004.