# Model Comparison: A Key Challenge for Transformation Testing and Version Control in Model Driven Software Development

Yuehua Lin, Jing Zhang, and Jeff Gray
*Dept. of Computer and Information Sciences, University of Alabama at Birmingham*
*Birmingham AL 35294-1170*
*{liny, zhangj, gray} @ cis.uab.edu*

## Abstract

As models and model transformations are elevated to first-class artifacts within the software development process, there is an increasing need to introduce core software engineering principles into modeling activities. In this position paper, we identify model comparison algorithms as a key toward addressing best practices associated with model transformation testing and version control of models.

## 1. Introduction

In the context of model driven software development (MDSD), the creation of models and model transformations is a central task that requires a mature development environment based on the best practices of software engineering principles. In a comprehensive approach to MDSD, models and model transformations must be designed, analyzed, synthesized, tested, maintained and subject to configuration management to ensure their quality. In the latter stages of the development lifecycle (i.e., the implementation stage), standard engineering processes such as testing and version control have been accepted as vital techniques toward improving quality and maintainability. However, testing and version control tools and techniques are not widely available (natively) in most modeling environments. Our position is that this situation leads to the following challenges that must be addressed to further the best practices of MDSD:

- **Lack of a Mechanism for Validating the Correctness of a Model Transformation**
  Model transformations are first class citizens in MDSD [3]. A large number of approaches and associated tools have been offered to define and perform model transformation. Typically, transformation rules and application strategies are written in a special language, called the *transformation specification*, which can be either graphical [1] or textual [5]. In model-to-model transformation, the transformation specifications are interpreted by the transformation engine to generate the *target* model(s) from the *sourc*e model(s). The effect of the transformation process is to refine the source model to contain additional details or features. It is noticeable that a transformation specification, like the code in an implementation, is written by humans and susceptible to errors. Additionally, a transformation specification may be reusable across similar domains. Therefore, it is essential to ensure the correctness of the transformation specification before it is applied to a collection of source models. However, among the literature on model transformation research, the important task of transformation testing has been neglected. The result is a lack of assurance that model transformations are free of faults and errors (i.e., determining if a particular transformation correctly transformed a source model into a new target model).

- **Version Control Tools do not Match the Structural Nature of Models**
  An essential capability for any phase of the software development lifecycle is to archive various versions of a software artifact and to revert back to previous configurations. Traditional version control and configuration management systems (e.g., CVS and Microsoft SourceSafe) have been available for many years to assist developers in this important function. However, these tools operate under a linear file-based paradigm that is purely textual while models are structurally rendered in a tree or graphical notion. Thus, there is an abstraction mismatch between currently available version control tools and the structural nature of models.

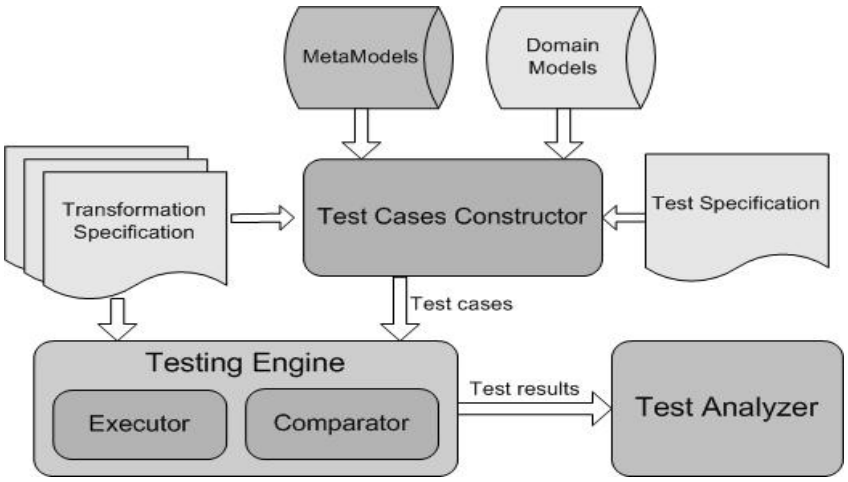
We propose model transformation testing and a model-centric version control system as effective approaches to improving the best practices of MDSD. A common requirement shared by each of these ideas is the ability to compare two different models to ascertain structural differences. The next two sections further discuss the important of transformation testing and model version control, followed by commentary on the importance of model comparison as a key facilitator for both of these best practices.

## 2. Model Transformation Testing

There are several ways to ensure the correctness of a model transformation. One possible approach is to apply formal methods in a correctness proving style of verification. However, even if a product is proved correct, it may nevertheless be subject to other forms of verification such as testing [11]. Execution-based testing has several advantages that make it an effective method to determine whether a task (e.g., model transformation) has been correctly carried out. These advantages are: 1) the relative ease with which many of the testing activities can be performed; 2) the software being developed (e.g., model transformation specifications or rules in model transformation) can be executed in its expected environment; 3) much of the testing process can be automated [6]. Within a model transformation infrastructure, it is vital to provide foundational support for testing model transformations. Otherwise, the correctness of the transformation may always be suspect, which hampers confidence in reusing the transformation.

According to the *IEEE Standard Glossary of Software Engineering Terminology*, testing is, "the process of exercising or evaluating a system by manual or automated means to verify that it satisfies specified requirements, or identify differences between expected and actual results" [11]. The basic testing activities consist of designing test cases, executing the software with those test cases, and examining the results produced by those executions [6]. Thus, we define *model transformation testing* to involve the execution of a deterministic transformation specification with test data (i.e., input to test cases) and a comparison of the actual results (i.e., the target model) with the expected output (i.e., the expected model), which must satisfy the intent of the transformation. If there are no differences between the actual target and expected models, it can be inferred that the model transformation is correct with respect to the given test specification. If there are differences between the target and expected models, the transformation specification needs to be reviewed and modified.

As an initial prototype, Figure 1 shows a framework for testing model transformation specifications that assists in generating tests, running tests, and documenting tests automatically. There are three primary components to the testing framework: test case constructor, test engine, and test analyzer. The test case constructor consumes the test specification and then produces test cases for the to-be-tested transformation specification. The generated test cases are passed to the test engine to exercise the specified test cases. Within the testing engine, there is an executor and a comparator. The executor is responsible for executing the transformation on the source model and the comparator compares the target model to the expected model and collects the results of comparison. The test analyzer visualizes the results provided by the comparator and provides a capability to navigate among any differences.




## 3. Model-Centric Version Control

It is our position that version control of models will become an increasingly important task of model development. A file-centric control system maintains an organized set of all the versions of files that are made over time. Version control systems allow developers to go back to previous versions of individual files, and to compare any two revisions to view the changes between them [13]. With respect to MDSD, model-centric version control and differentiation is concerned with storing all the versions of models, comparing any two versions to calculate the differences, and visualization of differences.

The storage and visualization features used by current version control systems are based upon the metaphor of a stream of linear text. Furthermore, the level of granularity provided to archive an artifact is at the level of a single file. This metaphor fits well with source code contained in a collection of files that make up a complete system. However, it is lacking in several respects when applied to modeling.

Modeling tools are rich in visual hierarchy and graphical representation, which are not found in the linear text files of source code representation. The current state-of-the-art for using SourceSafe or CVS to version models is archaic. Whenever a domain expert or modeling engineer desires to archive their models, they must first convert the model into an XML representation in a linear file. The XML representation of the file is then archived in the version repository. At this point, the intuitive hierarchy of a model is destroyed and replaced by links and references – the primary means for representing hierarchy and containment in a flat text file. Later, if a comparison between models is desired, the domain engineer must sort through a mass of tagged XML in order to discover differences between versions. With this approach, extra applications need to be developed to handle the results retrieved from the XML comparison in order to indicate the mapping and difference on models within the host modeling environment. Thus, the model versioning and differencing tasks become an exercise in comparing XML files.

Due to the disadvantages of XML-based model comparison, an alternative solution is to develop efficient algorithms to compare models that handle the internal data structure of model tools directly. Moreover, a challenging engineering effort will be to visualize the comparison results within the host modeling environment. Thus, model comparison and visualization of model differences are essential to model-centric version control.

## 4. Model Comparison

The main task of model comparison is to calculate the mappings and the differences between models. According to the above discussion of model transformation testing, model comparison needs to be performed to discover differences between the expected model (the expected result) and the target model (the actual result). Similarly, model comparison is also required in model-centric version control to detect changes between versioned models. Manual comparison of models is tedious, exceedingly time consuming, and susceptible to error. Thus, automation of model comparison is necessary. The following issues need to be explored for automation of model comparison and visualization of the comparison results.

- **At which level to compare models?**
  Models are usually rendered in a tree notation or a graphical notation and may be persistently stored in an intermediate language like XML. Thus, it is possible to compare models via their XML representation, but problems are raised with this approach as discussed in Section 3 (i.e., an XML document is generally a linear structure that represents hierarchy by linked identifiers to other parts of the document). This does not match the hierarchy that is visually depicted in most modeling languages and tools. One possibility is to compare models by accessing the internal representation provided by the host tool. However, such an approach could become tool-specific and not reusable.

- **What are the effective algorithms?**
  The critical problem in comparing models is to find an efficient algorithm to calculate the differences. Algorithms that assist in finding mapping and differences between XML documents can be found in [4, 12]. Algorithms for detecting differences between versions of UML are introduced in [9] and metamodel-independent algorithms regarding difference calculation between models are presented in [2]. A more exact approach is to regard model comparison as graph comparison so that model elements (e.g., vertices and edges) can be compared directly via model navigation APIs provided by the underlying modeling environment. But, existing graph matching algorithms are often too expensive for such a task [7]. Therefore, one possible solution is to loosen the constraints on graph matching to simplify the algorithm, e.g. requiring that each element of a model have a universally unique identifier (UUID) [2]. Another possible solution is to use heuristic algorithms with users' input and feedback. In a domain-specific modeling environment, the meta-model information may be used to aid the comparison by inform the elements and relationships that exist in a model.

- **Visualization of model differences**
  Source code is typically represented as a flat text file that is a sequence of linear characters. However, models are often represented visually as a hierarchical graph with complex internal representations. Several modeling tools (e.g., the Generic Modeling Environment [8]) capture the hierarchy and containment within models explicitly, whereby the modeler recursively clicks on higher-level models in order to reveal the contents of the containment. The hierarchical nature of models makes it difficult to observe visually the mapping and difference between two models. A new visualization technique is essential toward understanding the results of a model transformation testing framework, and to comprehend the changes between model versions. Currently, the commonly proposed technique to visualize the model differences is coloring [9], where different colors are used to indicate whether a model element is missing or redundant. However, a mature version control system requires more complicated techniques to discover richer information such as description and hierarchy of model differences.

## 5. Conclusion

Model transformation testing and model-centric version control are two important activities that need to be accepted as common best practices within the MDSD community. Model transformation testing helps to ensure the correctness of model transformations and version control of models assists in managing the evolution that occurs throughout the modeling process. Our position is to suggestion these activities as topics suitable for discussion at the workshop.

## References


1. Aditya Agrawal, Gábor Karsai, and Ákos Lédeczi, "An End-to-End Domain-Driven Software Development Framework," *18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) – Domain-driven Track*, Anaheim, California, October 2003, pp. 8-15.
2. Marcus Alanen and Ivan Porres, "Difference and Union of Models," *Proceedings of the UML Conference*, Springer-Verlag LNCS 2863, San Francisco, California, October 2003, pp. 2-17.
3. Jorn Bettin, "Model-Driven Software Development -- An emerging paradigm for Industrialized Software Asset Development," http://www.softmetaware.com/mdsd-and-isad.pdf, June 2004.
4. Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. "Change Detection in Hierarchically Structured Information," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Montreal, Canada, June 1996, pp. 493-504.
5. Jeff Gray, Ted Bapty, Sandeep Neema, Douglas C. Schmidt, Aniruddha Gokhale, and Balachandran Natarajan, "An Approach for Supporting Aspect-Oriented Domain Modeling," *Generative Programming and Component Engineering (GPCE 2003)*, Springer-Verlag LNCS 2830, Erfurt, Germany, September 2003, pp. 151-168.
6. Mary Jean Harrold, "Testing: A Road Map," *Proceedings of the Future of Software Engineering*, Limerick, Ireland, May 2000, pp. 61-82.
7. Samir Khuller, and Balaji Raghavachari, "Graph and Network Algorithms," *ACM Computing Surveys*, March 1996, vol. 28, no. 1, pp. 43-45.
8. Ákos Lédeczi, Arpad Bakay, Miklos Maroti, Peter Volgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gábor Karsai, "Composing Domain-Specific Design Environments," *IEEE Computer*, November 2001, pp. 44-51.
9. Dirk Ohst, Michael Welle and Udo Kelter, "Differences between Versions of UML Diagrams," *European Software Engineering Conference/Foundations of Software Engineering*, Helsinki, Finland, September 2003, pp. 227-236.
10. Stephen R. Schach, "Testing: Principles and Practice," *ACM Computing Surveys*, March 1996, vol. 28, no. 1, pp. 277-279.
11. Stephen R. Schach, *Classical and Object-Oriented Software Engineering*, 6th Edition, McGraw-Hill, 2004.
12. Yuan Wang, David J. DeWitt, Jin-Yi Ca, "X -Diff: An Effective Change Detection Algorithm for XML Documents," *Proceedings of the 19th International Conference on Data Engineering*, Bangalore, India, March 2003, pp. 519-530.
13. https://www.cvshome.org/nonav/scdocs/ddCVS_cvsglossary.html