

Model Transformations Should Be More Than Just Model Generators

Jon Whittle¹ and Borislav Gajanovic²

¹ Dept of Information & Software Engineering
George Mason University
`jwhittle@ise.gmu.edu`

² Institute for Software Systems Engineering
Technische Universität Braunschweig
`b.gajanovic@sse.cs.tu-bs.de`

1 Introduction

Model transformations are an increasingly important tool in model-driven development (MDD). However, model transformations are currently only viewed as a technique for generating models (and, in many cases, only code). Little is said about guaranteeing the correctness of the generated models. Transformations are software artifacts and, as such, can contain bugs that testing will not find. This paper proposes that, in fact, model transformations should do more than just generate models. In addition, they should generate evidence that the generated models are actually correct. This evidence can take the form of precise documentation, detailed test cases, invariants that should hold true of the generated models, and, in the extreme case, proofs that those invariants do actually hold. The hypothesis is that there is enough information in the definition of a transformation to provide evidence that certain properties of the generated model are true. Such information is usually left implicit. By making that information explicit and annotating the generated model, a consumer of the model increases his/her confidence that the model does what it is supposed to do.

2 An Example

Autofilter [WS04] is a domain-specific model-to-code generator developed at NASA Ames Research Center. It generates code used in attitude control systems and has undergone pilot studies in deep space and spacecraft docking applications. Given a problem description written in a domain-specific modeling language, it selects, instantiates and composes a set of domain components that implement the problem. Given the safety-critical nature of the domain, it is imperative that the code generated be correct. In particular, there are dependencies between the components that must be satisfied in each implementation. In addition, there are global properties of the generated code that must be satisfied. Since Autofilter is a highly complex model transformation, it is not cost-effective

to formally verify the transformation using traditional methods. Instead, we propose that model transformations should generate, along with the model (or code) that they output, evidence that the model (or code) is correct. This evidence can then be processed by an independent observer such as a certification body (e.g., the FAA), code inspection teams, or automated tools.

The top third of Figure 1 is an abstract view of model-to-code transformations like Autofilter. From now on, we assume that transformations generate code, but the techniques apply to model generators as well.

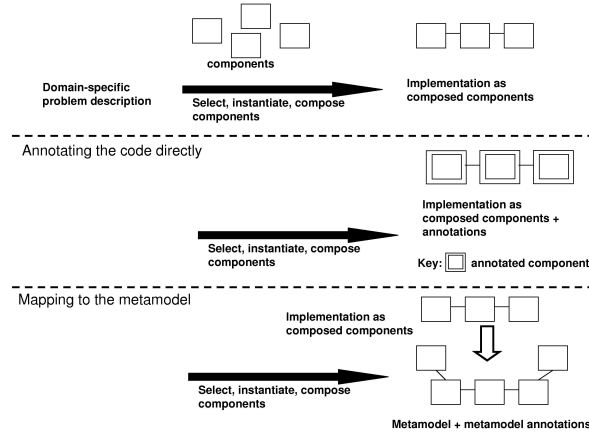


Fig. 1. Approaches to Generating Annotations.

3 Two Approaches to Annotating Generated Code

The bottom two thirds of Figure 1 illustrate two possible approaches for annotating model transformations so that they generate not only code but also evidence that the code is correct.

3.1 Annotating the Generated Code Directly

The first approach views evidence annotation as just another form of generation (see the middle third in the figure). As well as generating code, a transformation generates assertions, preconditions, postconditions or invariants for each code fragment generated. These annotations can be independently checked or taken as formal documentation. Examples might be: loop invariants for array iterations; class invariants for data classes; protocol invariants for messages. This approach works well when the evidence concerns low-level aspects of the generated code. In Autofilter, this approach has been used to annotate generated code with preconditions on component inputs, such as the fact that a sensor input noise has a certain characteristic [RVWL03]. It has also been used to provide invariants about arrays that can be used to prove array bounds safety [DFS04].

3.2 Mapping to the Metamodel

A second approach is to lift the transformation output to the meta-level (see the bottom third in the figure). For example, consider a statechart code generator that transforms UML statecharts to Java code. The Java code can be implemented using a variety of techniques — simple **case** statements, using the State Pattern etc. — but the metamodel for statecharts will always be the same. If the transformation generates annotations at the metamodel-level then the output Java code can be lifted to the metamodel where the annotations can be checked. For example, a statechart implemented in Java using the State Pattern would be abstracted to an instance of a statechart metamodel. At this abstracted level, metalevel annotations can be checked against the statechart. This approach has been used in Autofilter to check domain-specific constraints that should hold for the generated code [GWC03].

The first approach results in very detailed annotations at the implementation level. This allows code-level checking of the annotations but the annotations are highly coupled with the code implementing the transformation. The second approach decouples the transformation implementation and the annotations since the annotations can be specified entirely at the meta-level, and the metamodel is independent of the particular implementation generated. However, since the metamodel is more abstract, implementation-level annotations cannot be specified. Hence, both approaches complement each other.

4 Checking Annotations

So far, we have extended model transformations to provide annotations with the models that they generate. These annotations are assertions or evidence about the target model. The annotations can be used in different ways depending on what level of confidence is required of models generated by the transformation. At one extreme, the annotations are viewed simply as precise documentation. At the other extreme, they can be formally proved to hold on each generated model. In between these extremes, they can be used as the input to various analysis tools that may not provide full correctness guarantees but can check some properties. For example, if the annotations are given in JML, the ESC/Java tool can statically check some of the annotations.

Note that by checking the annotations for each generated model, the transformation itself is being indirectly validated. It is not feasible, in general, to formally verify transformations because generators are large, complex systems that may be written in any language. However, it is feasible, for some properties, to formally verify that those properties hold for a given generated model. The annotations are key to this. They are crucial stepping stones in a proof for the property without which the property could not be proved. For example, in Autofilter, a statistical optimality proof of the generated programs was considered — namely, that the generated code calculates the optimal estimate (in the sense of minimization of the mean-squared error) of a given state variable. This

is an important property that is practically impossible to verify using the code alone because the proof is too complex — as an informal proof, it consists of ten pages of logical steps; as a machine-checked proof, it is an order of magnitude larger. In fact, one crucial part of the proof is shown in [RVWL03] to have 2^{142} choice points, thus providing ample evidence that the proof is never likely to be obtained automatically by examining the code alone.

However, the proof can be obtained automatically given the set of annotations. Each component comes with an annotation that acts as a stepping-stone for the proof. Given the annotations, it is possible to reconstruct a proof for the entire program without the need for any search. Hence, a *user* of Autofilter can *automatically* prove that the code s/he generated is correct for the optimality property. Of course, for full confidence, the annotations must also be proved. This was done in the case of the statistical optimality property but might not be necessary in general — instead, the proof would be marked as modulo assumptions captured by the annotations.

Note that this kind of transformation verification is independent of the language used to implement the transformation. It could be written in a programming language, as a declarative set of rewrite rules, or even as a XSLT Schema. Since only the products generated by the transformation, not the transformation itself, is analyzed, the technique applies to any transformation language.

5 Summary

In previous work, the Autofilter model transformation system was augmented to provide evidence of correctness with each program that it generates. The idea can be extended to model transformations in general. Further work is being undertaken to provide general techniques for capturing annotations in generated models and proving properties using those annotations as intermediate lemmas.

References

- [DFS04] Ewen Denney, Bernd Fischer, and Johann Schumann. Using automated theorem provers to certify auto-generated aerospace software. In David A. Basin and Michaël Rusinowitch, editors, *IJCAR*, volume 3097 of *Lecture Notes in Computer Science*, pages 198–212. Springer, 2004.
- [GWC03] Emanuel Grant, Jon Whittle, and Rajani Chennamaneni. Checking program synthesizer input/output. In *3rd OOPSLA Workshop on Domain-Specific Modeling*, 2003.
- [RVWL03] Grigore Rosu, Ram Prasad Venkatesan, Jon Whittle, and Laurentiu Leustean. Certifying optimality of state estimation programs. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 301–314. Springer, 2003.
- [WS04] Jon Whittle and Johann Schumann. Automating the implementation of Kalman filter algorithms. *ACM Transactions on Mathematical Software*, 30(4):434–453, December 2004.