

Challenges in bootstrapping a model-driven way of software development

Dennis Wagelaar*

Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium
dennis.wagelaar@vub.ac.be

Abstract. Current MDE technologies are often demonstrated using well-known scenarios that consider the MDE infrastructure to already be in place. When starting up your own model-driven development infrastructure, because existing boxed-in tools are insufficient, for example, you will come across a number of challenges. Generally, you cannot just sit down and implement all your model transformations and other MDE infrastructure, because it simply takes too long before you get usable results. An incremental approach to putting model-driven development into place gives you the necessary “break-points”, but poses extra challenges with regard to the MDE technologies used. This paper discusses some of these challenges, such as bootstrapping a step-wise refinement chain of model transformations, bootstrapping the (usage of the) modelling language, the position of round-trip engineering and useful properties for a model transformation tool.

1 Introduction

Current MDE technologies are often demonstrated from the point of view where either the MDE infrastructure is already in place, or the MDE infrastructure is part of a ready-to-run solution that only requires you to provide some models and off you go. That leaves out the scenario where you’ll have to provide your own MDE infrastructure, such as meta-models, model transformations, configuration tools and build processes. A common reason for this scenario is that the existing ready-to-run MDE tools don’t provide (exactly) what you want and require you to do some “post-customisation” of the tool’s output. That “post-customisation” is a typical model transformation scenario, which means that writing your own model transformation definitions is suddenly within the scope of your software development process. Moving model transformation definition within the scope of your development process poses a number of challenges.

In this paper, I will elaborate on some of these challenges and relate them to each other by means of a central case study. This case study involves the model-driven development of an instant messenger application¹.

* The author’s work is part of the VariBru project, which is funded by the Institute for the encouragement of Scientific Research and Innovation of Brussels (ISRIB)

¹ <http://ssel.vub.ac.be/ssel/research:mdd:casestudies:im>

2 Bootstrapping model transformations and language abstractions

The instant messenger case study started out with a UML model and a simple Java code generator. At this point, one could identify several recurring patterns in the source model: getter and setter methods, explicit observer pattern implementations, explicit abstract factory pattern implementations, etc. There were also parts in the source model that hampered platform independence, such as explicit references to Java API (collection types, applet, AWT). These recurring patterns as well as the platform-specific parts were all candidates for abstraction and model transformation. I've decided to start with automatically generating the getters and setters.

When you define a model transformation, you already know which recurring pattern you want to generate in your output model: getters and setters in this case. Now you need to decide what language abstractions to use in your source model to represent the combination of an attribute and its getters and setters. UML uses the profile mechanism for this, where you can extend the semantics of existing language constructs with stereotypes. Let's say you define an <<EncapsulatedAttribute>> stereotype on top of the attribute language construct. You can then use a model transformation to generate getter and setter methods for each encapsulated attribute. Listing 1.1 shows what such a transformation definition could look like in ATL.

```
module Accessors;
create OUT : UML2 from IN : UML2;
...
rule PublicPropertySingle {
  from s : UML2!"uml::Property" (
    UML2!"Accessors::EncapsulatedAttribute".allInstances()
    ->select(e|e.base_Property=s)->notEmpty())
  using { baseNameS : String = s.accessorBaseNameS; }
  to t : UML2!"uml::Property" (...),
    getOp : UML2!"uml::Operation" (name <- 'get'+baseNameS,
                                   class <- s.class,
                                   ownedParameter <- Sequence{getPar}),
    getPar : UML2!"uml::Parameter" (name <- 'return',
                                     type <- s.type,
                                     direction <- #return),
    getDep : UML2!"uml::Dependency" (name <- 'Get'+baseNameS,
                                     client <- getOp,
                                     supplier <- s),
    getDepST : UML2!"Accessors::accessor" (base_Dependency <- getDep), ...
}
```

Listing 1.1. Accessors ATL transformation module

The interesting part of this transformation definition lies in the complexity of working with stereotypes. Whereas the UML modelling tools provide a user-friendly way of working with stereotypes, model transformation tools fail to hide the underlying complexity of stereotypes. Stereotypes have a meta-class representation that allows them to be instantiated at the model level: stereotype applications. The example transformation definition in Listing 1.1 applies the <<accessor>> stereotype by instantiating the UML2!"Accessors::accessor"

meta-class. In order to find stereotyped elements, the model transformation has to look up each UML2!"uml::Property" instance, as well as each UML2!"Accessors::EncapsulatedAttribute" instance, and match them against each other. This inefficient procedure is a direct result of the fact that the UML2 meta-model has no knowledge of any stereotype definitions.

Another way of introducing new language abstractions is by extending the meta-model. In the case study, that would be the UML2 meta-model. Listing 1.2 shows what the Accessors transformation definitions looks like when you use a meta-model extension². The input pattern is simpler, as it only needs to find instances of UML2!"accessors::EncapsulatedProperty", and the dependency between getters/setters and their attribute no longer requires a separate stereotype instance that links to it.

```

module Accessors2;
create OUT : UML2 from IN : UML2;
...
rule PublicPropertySingle {
  from s : UML2!"accessors::EncapsulatedProperty"
  using { baseNameS : String = s.accessorBaseNameS; }
  to t : UML2!"accessors::EncapsulatedProperty" (...),
    getOp : UML2!"uml::Operation" (name <- 'get'+baseNameS,
                                   class <- s.class,
                                   ownedParameter <- Sequence{getPar}),
    getPar : UML2!"uml::Parameter" (name <- 'return',
                                   type <- s.type,
                                   direction <- #return),
    getDep : UML2!"accessors::AccessorDependency" (name <- 'Get'+baseNameS,
                                                    client <- getOp,
                                                    supplier <- s), ...
}

```

Listing 1.2. Accessors2 ATL transformation module

The previous transformation definition examples illustrate a conflict that has existed in the MDE community for a long time: should you use UML profiles or meta-models for language extension? In the domain of program transformation, people have used modular grammars, as is demonstrated by the Stratego/XT approach to implementing language extensions and transformation definitions [5]. Considering this and the added complexity of stereotypes in the domain of model transformation, direct extension using modular meta-models seems the way to go.

However, it is not likely that UML tools are going to support unlimited meta-model extension any time soon. This is demonstrated by the fact that UML has been defined by a meta-model for over 10 years, while UML extensions are still defined as profiles today. The technical reason behind this is that in graphical languages, syntax extension is more complicated than in textual languages. Extension by stereotypes is easy from a concrete syntax point of view: just add UML keywords to the original graphical representation of the stereotyped model element. Tools for defining graphical concrete syntax, such as the Eclipse

² I still use the name UML2 to refer to the extended meta-model: UML2 is only a symbolic name that can be bound to any concrete meta-model.

Graphical Modeling Framework, are not nearly as easy to use as the tools for meta-modelling.

I believe that this paradox, where easy language extension causes complex model transformation definitions, can be mitigated by providing an automated translation from UML models with profiles to models based on “pure” meta-models. A simple transformation can generate the meta-model representation of a UML profile. A higher-order transformation can generate a transformation definition that translates stereotype instances applied to regular model elements to special model elements. The result would be a situation where you can create UML models with profiles, while you can also write the kind of transformation definitions shown in Listing 1.2.

3 Evolving a step-wise refinement chain

The previous section started out from the initial state of the instant messenger case study, which exhibited several candidates for language abstraction and model transformation: getter and setter methods, explicit observer pattern implementations, explicit abstract factory pattern implementations, explicit references to Java API (collection types, applet, AWT), etc. Now that a language abstraction and a model transformation definition are in place for getter and setter methods, a start can be made with the other abstractions/transformation definitions.

Keeping the transformation of each language abstraction you’ve introduced nicely separated in its own transformation definition reduces local complexity. You can concentrate on solving a local transformation problem. Global complexity does not reduce, however, and generally requires managing. Take the model transformation for the observer pattern, for example. This model transformation adapts the setters of each observed attribute such that the `update()` method of the observers is triggered. That means that the setter methods must already exist in the model.

Batory et al. have already pointed out that you need to manage dependencies between the different refinement steps in [1]. There even are a number of model transformation languages that support *critical pair analysis* [6][7], which provides an automated analysis of any dependencies between model transformation definitions. Unfortunately, critical pair analysis is not an easy computing task and doesn’t scale well. It is also not applicable to all model transformation languages.

Normally, transformation definitions in a step-wise refinement chain are designed to work on the output of the previous model transformation and the developer is conscious of the dependencies. When adding *alternatives* for a specific model transformation definition, or when *changing* one of the model transformation definitions in the refinement chain, the situation becomes different. Will the alternative/changed model transformation definition still provide what is required by the next transformation steps? Techniques such as critical pair

analysis may be able to tell whether the following transformation steps will still trigger, but can't say much about changes in the semantics of the outcome.

I rather believe that the dependencies between transformation steps must be concentrated in semantically rich meta-classes. For example, when the observer pattern transformation definition requires the presence of setter methods in order to adapt them, the meta-model should provide an explicit notion of setter methods. By converging the dependencies between transformation steps in semantically rich meta-classes, automated analysis of such dependencies becomes much easier.

4 To round-trip or not to round-trip

The instant messenger case study uses an incomplete code generator, that does not understand any of the UML behaviour diagrams. Instead, it uses UML's support for adding native method bodies in Java, C++, etc. to operations. Obviously, Java editing support in a UML case tool is nowhere near as advanced as what Eclipse JDT can provide. As a result, all method bodies for the instant messenger are written in Eclipse JDT, after the skeleton code is generated. These method bodies are then manually added back to the UML model, such that the next code generation cycle picks them up. This approach is a terrible maintenance nightmare and cannot be used in real situations. Merging-style code generators, such as EMF's JET and Acceleo, look nice in the beginning. They provide a way to propagate any changes in the model back to the code while merging with the existing, hand-written code. They do not provide a way to propagate code changes back to the model, however. That means that model and code can still grow apart from each other and at some point the model no longer reflects what is in the code.

A full round-trip engineering (RTE) approach can be used to solve this problem. RTE is useful in situations where the model does not provide a complete view of the software. In the case of the instant messenger, the model provides a complete specification, but still an incomplete view: the Eclipse JDT view of the method bodies is far more useful. The problem with RTE is that it is very hard to do in a general way, as is illustrated by Van Paesschen in [3] and by Antkiewicz et al. in [4]. First of all, the model transformations must be executable in both directions. There are bi-directional model transformation languages, such as QVT Relations [8] and Triple Graph Grammars [9], that support this. Writing a bi-directional transformation definition is more complex than writing a single direction transformation, however. Your output patterns have a double function as input patterns as well. As such, you must make sure that your output patterns function correctly as input patterns as well. Then, both Van Paesschen and Antkiewicz acknowledge that RTE requires more than just bi-directional transformations in the case that your model transformation definitions are not injective. Consider a simple model transformation that applies a profile to a model if and only if that profile was not yet applied. The reverse of this transformation is not possible without having the original source model available.

Recent work on RTE shows a different approach to the problem. In [10], Hettel et al. point to the possibility of doing RTE with only having a forward transformation definition and reference source and target models available. In [11], Xiong et al. actually go as far as claiming an initial implementation of such a system based on ATL. If this approach can be made to work in general, where the developer only needs to provide source model and forward model transformation definition, this removes any disadvantages that RTE has over simple forward engineering.

References

1. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering* **30** (2004) 355–371
2. Henriksson, A., Larsson, H.: A Definition of Round-trip Engineering. Technical report, Department of Computer and Information Science, Linköpings Universitet, Linköping, Sweden (2003)
3. Van Paesschen, E.: Advanced Round-Trip Engineering: An Agile Analysis-driven Approach for Dynamic Languages. PhD thesis, Vrije Universiteit Brussel, Brussels, Belgium (2006)
4. Antkiewicz, M., Czarnecki, K.: Framework-Specific Modeling Languages with Round-Trip Engineering. In: Proceedings of the ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006), Genova, Italy. Volume 4199 of Lecture Notes in Computer Science., Springer-Verlag (2006) 692–706
5. Bravenboer, M., Visser, E.: Designing Syntax Embeddings and Assimilations for Language Libraries. In: Models in Software Engineering. Workshops and Symposia at MoDELS 2007. Volume 5002 of Lecture Notes in Computer Science., Springer-Verlag (2008) 34–46
6. Mens, T., Taentzer, G., Runge, O.: Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. *Electr. Notes Theor. Comput. Sci.* **127** (2005) 113–128
7. Mens, T., Kniesel, G., Runge, O.: Langages et Modèles à Objets (LMO 2006). In: Proceedings of Langages et Modèles à Objets (LMO 2006), Nîmes, France. (2006) 167–183
8. Object Management Group, Inc.: Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification. (2005) Final Adopted Specification, ptc/05-11-01.
9. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In Tinhofer, G., ed.: Proceedings of the WG'94 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science. Volume 903 of Lecture Notes in Computer Science., Springer-Verlag (1994) 151–163
10. Hettel, T., Lawley, M., Raymond, K.: Model Synchronisation: Definitions for Round-Trip Engineering. In Vallecillo, A., Gray, J., Pierantonio, A., eds.: Proceedings of the First International Conference on Theory and Practice of Model Transformations (ICMT 2008), Zürich, Switzerland. Volume 5063 of Lecture Notes in Computer Science., Springer-Verlag (2008) 31–45
11. Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H.: Towards Automatic Model Synchronization from Model Transformations. In: Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering (ASE '07), Atlanta, Georgia, USA, ACM Press (2007) 164–173