A Code Merger to Support Reverse Engineering Towards Model-driven Software Development

Oliver Haase, Nikolaus Moll and Paul Zerr
HTWG Konstanz, University of Applied Sciences, Computer Science Department, Braunegger Str. 55, 78464 Konstanz, Germany

Keywords: Iterative Model-driven Migration, Reverse Engineering, Code Merging.

Abstract: Model-driven engineering is a promising approach whose feasibility for commercial development is currently being validated. While most approaches discuss forward-engineering steps, only little research has been done on model-driven software migration. More precisely, it is unclear how to transform — or reverse engineer — existing code into generated and hand-crafted artifacts. We present an iterative approach to this problem. Assuming some evolving high-level representations of a software legacy system, code generators may produce a second version of the system to an extend where hand-crafted code is still necessary for completion. In this report we present a code merger that completes the generated code by reusing the implementation of the software legacy system.

1 INTRODUCTION

Model-Driven Software Development (MDSD) is a promising engineering approach, in particular when multiple variants of a product family have to be developed and maintained. Various papers describe how MDSD can improve software maintenance but also software flexibility, productivity and reliability, see, e.g., (van Deursen and Klint, 1998; Kieburtz et al., 1996; Ladd and Ramming, 1994; Krueger, 1992). With MDSD, a software system is described by *meta-models*, *models*, *generators*, as well as *hand-crafted code*, i.e. those parts of the code that cannot or make little sense to be expressed on the model level. While most MDSD approaches focus on forward engineering, i.e. the development of new software, only little research has been done on how to transform — or reverse engineer — existing code into the aforementioned MDSD artifacts.

Reverse engineering existing code towards MDSD is not always the prudent thing to do. Many productive software systems, and in particular those that are comparably stable, well-designed and hence well maintainable and extensible, are better left untouched. Our main goal, in contrast, is to aid the reverse engineering of software that is used for product families and thus is adapted, configured, and user tailored in various ways. Often, companies use reference systems that are adapted and configured for new customers; just as often, this configuration process has grown over time and has become highly complicated. Expressing the reference system in appropriate models and meta-models can substantially improve the production of new variants; moreover, MDSD provides a significantly higher degree of platform independence for these product families. This is especially true if the high-level representation is expressed in one or several *domain-specific languages* (DSLs).

The process of reverse engineering towards MDSD must be iterative. A typical, complex software system cannot be transformed into a combination of DSLs (meta-models), models, and generators in a single step. Instead, the reverse engineer will start small, extract individual features from only a few classes, have only part of the code generated, and have that generated code combined with the hand-crafted legacy code. Step by step, the DSLs, models, and generators will grow, and the percentage of hand-crafted code will decrease.

Please note that we focus our considerations on Java as the currently most widely spread programming language. The underlying principles are, however, applicable to other object-oriented or modular languages as well. Please also note that this paper is not about DSL design and features extraction, but about the iterative merging of hand-crafted and generated code such that the resulting software provides the same functionality as the legacy code base. In a

Haase O., Moll N. and Zerr P..
A Code Merger to Support Reverse Engineering Towards Model-driven Software Development.
DOI: 10.5220/0004309000830088
In *Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development* (MODELSWARD-2013), pages 83-88
ISBN: 978-989-8565-42-6
Copyright © 2013 SCITEPRESS (Science and Technology Publications, Lda.)

productive environment, operability of the system at any time in the iterative process is absolutely instrumental.

The merging process is far from trivial. This is partly owing to the iterative nature of the approach that leaves us with incomplete, generated artifacts, and partly because of the very nature of most legacy systems that simply are not as regular and well-behaved as a strictly forward engineered system would be. Our approach provides solutions to both of these problems, as we will show in the subsequent sections.

2 ITERATIVE REVERSE ENGINEERING

As described in the introduction, we consider reverse engineering towards MDSD an iterative process. This process uses, creates, and modifies several artifacts, which exist in different versions, each of which corresponds to the respective iteration in the overall process:

- S_i represents the initial source code for iteration i. This source code comprises both manual and generated code. The original legacy code S_0 can be considered a special case that contains only manual code.
- DSL_i is a domain specific language that covers those features that are generable in iteration i.
- M_i is the model that describes those features of the software system that have been extracted onto the model level up until the i-th iteration of the process; of course, M_i must conform to DSL_i.
- GEN_i is the generator that accompanies DSL_i and that transforms M_i into generated code.
- G_i is the code generated by GEN_i.

In the following, we use the terms S_i and G_i to denote the code of a software system in its entirety, or only an individual class, depending on the context. We do this in order not to unnecessarily complicate out terminology. Each iterative cycle i consists of two phases that are described below and graphically shown in figure 1.

During this iterative process, the model level artifacts DSL_i, M_i, and GEN_i, as well as the proportion of generated code G_i will grow successively. Complementarily, the amount of manual code will decrease with each step.

- **Phase 1: Feature Extraction**
In each iteration cycle, the reverse engineer aims to advance the generated code by extracting additional features. Technically speaking, the engineer replaces a DSL, a generator, or a model by a newer version. Often, advancing one of these components also requires changes in the other ones, e.g. adding a new meta-model entity to a DSL requires the code generator to cover this new entity as well. The feature extraction is indicated by the dotted arrows in figure 1.

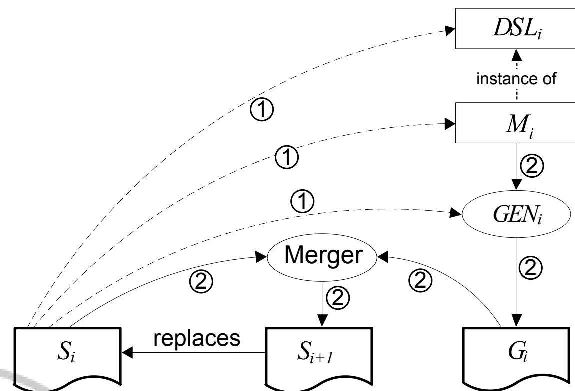

Figure 1: The two phases of the iterative reverse engineering process towards MDSD.

- **Phase 2: Code Generation and Merging**
The second phase is indicated by the solid arrows in figure 1. First, the generator, GEN_i, transforms the model, M_i, into code, G_i. Roughly speaking, G_i will be a subset of the software system S_i. In order to obtain a new version, S_{i+1}, of the software, a merger combines the generated code G_i with the existing codebase S_i. For that purpose, the merger complements the generated code with those manual code pieces from S_i that have (not yet) been generated. This new code version, S_{i+1}, then replaces S_i in the next iteration cycle. Typically, the proportion of generated code in S_{i+1} will be greater than in S_i.

The merging process is far from trivial, because in real world systems, manually crafted code hardly ever is as well-structured and regular as generated code. As a simple example consider the get and set methods in a fairly large codebase: They might follow the usual naming conventions for 90% of all attributes, but might be named differently for the remaining 10% (e.g. `readX()` instead of `getX()`). With a naive reverse engineering and merging process, the generation of getters and setters would not be possible at all due to the rather few exceptions, because the generator would not be able to distinguish between the two categories. Our merger, in contrast, can deal with irregularities like these, as will be explained

in detail in section 4.

Executing the above described iterative process allows the reverse engineer to pull more and more features of the software system onto the model level. As more and more generated code replaces manual code, each new revision of S_i simplifies the maintainability of the software system, and eases the generation of new code variants.

As with any kind of code refactoring, it is important to support the reverse engineering process towards MDSD by continuous unit testing. The reason for this is that generated method implementations can vary from the original code, as long as the methods provide the same functionality. See section 4 for further information.

It should have become obvious that the generated code will almost by definition be incomplete. To be able to adequately deal with this incompleteness in the merging step, we assume that for a given *method* in S_i, the generator either (i) does not generate the method at all, or (ii) generates only the declaration of the method using its method header[1], or (iii) generates the complete method definition, i.e. both header and body. There is no generation of partial method bodies. This assumption is, however, not a serious restriction, because each method that could be generated partly can be refactored and decomposed in a preparatory step. For *fields*, either only the declaration or the declaration together with an initializer is generated. Finally, *inner classes* are generated recursively within their containing classes.

3 SEPARATING GENERATED AND MANUAL CODE

When forward engineering a software system with MDSD, the partly generated code needs to be completed with hand-crafted code. One approach is to manually add code directly into the generated source code. However, this technique is widely considered problematic, mainly for two reasons: (1) The hand-crafted regions of the source file must not be overwritten when the artifact is regenerated, and (2) putting generated code under version control is generally avoided. Therefore, most authors (e.g., (Völter, 2009)) recommend the strict separation of hand-crafted and generated code into different source files. In the following, we briefly discuss the two separation techniques that are commonly propagated, and evaluate their applicability for model-driven *reverse engineering*.

1. *Inheritance* is probably the most widely-used technique to separate generated and manual code. The idea is to generate a base class which can be extended by a concrete subclass implementation that contains the manual code. For reverse engineering, this would mean to decompose an existing legacy class A into a generated base class, A', and a hand-crafted subclass A''. This decomposition, however, leads to a variety of technical problems, one of which has to do with access rights: Assume a private field in a legacy class A that after reverse engineering can be generated into the new base class A'. To give subclass A'' access to this field, its access right must be modified from `private` to `protected`. This modification, however, might have undesired side-effects somewhere downstreams the potentially complex pre-existing type hierarchy.
2. *Composition and delegation* is another approach where a composed object uses an associated *delegate object* to delegate its tasks to. With model driven engineering, the composed object contains the generated code, whereas the delegate object contains the hand-crafted code. In (Walter and Haase, 2008), this technique is used to transform existing legacy code into a model-driven structure. This approach, however, leads to even more serious problems with access rights than inheritance, owing to the fact that object identity gets lost with composition and delegation, because each object in the legacy system gets split across two separate objects. As a consequence, each formerly private field must be made not only protected but at least package-private if the other object needs access to it, too.

In summary, these separation techniques are not adequate for reverse engineering, because decomposing existing legacy classes into new classes, be it by subclassing or by composition, is a very disruptive technique that can have unexpected and undesired effects on the correctness of the overall system. In addition, on a semantic level, both approaches are meant for other purposes than separating generated from manual code which makes their application for code separation questionable: (1) Inheritance is meant to model *is-a relationships*; however, there is no such relationship between the generated and the manual code portion of a class. (2) Composition is meant to model *part-of relationships*, which is also not the appropriate relationship between the two code portions. In contrast, manual and generated code are two *equal* parts that together form the functionality of a class.

[1]The method header consists of modifiers, generic type parameters, result type, method identifier, parameters and the thrown exceptions.(Gosling et al., 2005, Page 210)

As a result, we do not consider code separation neither by inheritance nor by composition appropriate for reverse engineering. These techniques might rather be suitable for model-driven forward engineering, if at all. In fact, there seems to be a recent trend not to split classes for code separation even for forward engineering, for the reasons discussed above. For reverse engineering, in any instance, we have decided for the co-existence of generated and manual code in single source files, and to mitigate the potential drawbacks through a set of supporting techniques and tools:

- From a software developer's point of view, the distinction between generated and hand-crafted code must be explicit. We use the standard annotation `javax.annotations.Generated` to mark the generated portions of the code.
- In each iteration, our code merger makes sure the manual code is not overwritten unintentionally.
- We have developed an Eclipse editor that can gray out and fold in generated code automatically, in order to draw the developer's attention to the manual portions of the source code.
- The same editor protects the generated code against modification by making it read-only. Of course, a developer can always remove a `@Generated` annotation and then modify the formerly protected code portion. We find it, however, acceptable to protect developers against unintentional modifications of generated code while leaving open a back door for intentional changes.

One issue that remains open with mixed source files is versioning control, because putting generated code under versioning control is generally to be avoided. We believe, however, that (1) the associated effects, such as new versions that are only due to the (changed) generated part of a source file, are often overrated, and that (2) developing a versioning control system that understands the standard `@Generated` annotation and acts accordingly would be an interesting and worthwhile task for model driven development in general.

4 MERGING PROCESS

As we have discussed so far, in each step of our iterative approach the partly generated code G_i has to be combined with the current version of the source code S_i. In this section, we describe this merging process.

Methods in G_i can be generated either *completely* or *incompletely*. An incomplete method consists of a method header only and is generated without a body[2]. A common example for incomplete methods are hook methods. The source code S_i, in contrast, contains *complete* members only. These members can be (1) hand-crafted, (2) completely generated or (3) partly generated and partly hand-crafted.
A *method* in S_{i+1}, i.e. in the result of the merging of S_i and G_i, is

- **completely generated**, if the generated method in G_i is complete (consists of both a method header and body).
- **partly generated and partly handcrafted**, if the generated method in G_i is incomplete (has only a method header, but does not have a method body).

A *field* in S_{i+1} is

- **completely generated**, if the field is initialized in G_i, or the field is neither initialized in S_i nor in G_i.
- **partly generated and partly handcrafted**, if the field is initialized in S_i, but not initialized in G_i.

Completely generated elements are marked by the following annotation:

```
@Generated("de.htwgkn.mdre.gen")
```

Incompletely generated elements, on the other hand, are marked by the following annotation:

```
@Generated("de.htwgkn.mdre.gen",
    value="declaration")
```

Bringing S_i and G_i together is in fact a merge operation, which leads to a new source code S_{i+1}. Ideally, S_{i+1} contains more generated members than S_i.

The merging process is done per class, so all classes from G_i will be merged with their counterparts in S_i. Classes, in turn, are merged per member. For each class member in G_i, a corresponding member in S_i is searched. A corresponding member has the same name and - if it is a method - the same signature, too. There are, however, situations where no corresponding member in S_i can be found. Typical reasons are:

- **The Semantically Corresponding Member in S_i has a Different Name.** Such a *misnamed* member can occur because legacy code is typically not as regular as generated code. As a simple example, consider a legacy system where *most* getters follow the usual naming convention, `getX()`. However, for a specific field, `y`, the getter has been named `readY()`. For this attribute, the generated getter, `getY()`, is misnamed (with respect to the legacy code).

[2]The method `private void generated();` is an example for an incompletely generated method. Obviously, the code for such a method is syntactically incorrect, because non-abstract methods must have a body.

To avoid misnamed members in G_i, renaming the corresponding members in S_i before reverse engineering may be a solution. This works, however, only for internal APIs; in these cases a modern IDE can automatically refactor the method as well as all its users. For a method that is part of a public API, renaming is generally not an option.

- **There is no Corresponding Member in S_i.** The member that has been generated into G_i does not exist in S_i. Such a *superfluous* member can occur due to inconsistencies in the models and generators — caused by mistakes or oversimplifications during the reverse engineering step. As a very simple example, consider a class where *most* private fields have getters. With an oversimplified model, meta-model or generator, getters for *all* attributes will be generated. Such a superfluous member can, however, become useful when it comes to the forward engineering of future variants of the reference system.

The merger is not able to classify such a situation on its own, so the engineer has to decide how to proceed with the current member from G_i. If it is a misnamed member, the engineer indicates the corresponding member in S_i. If the member is superfluous, the engineer has the option to copy it to S_{i+1} anyway, or ignore it.

A special situation arises when a whole class that does not exist in S_i is generated. This happens, e.g., when an evolved version of a generator generates a new utility class to factor out common tasks. Again, the engineer decides whether to copy the completely generated class, or to ignore it.

4.1 Rules

These rules explain how S_i and G_i are merged to obtain the next iteration, S_{i+1}:

1. **Methods:**
   (a) **Equal Signature, Completely Generated.** G_i becomes S_{i+1}, and is marked as completely generated.
   (b) **Equal Signature, Partly Generated.** S_{i+1} is a combination of G_i's method header and S_i's body. It is annotated as partly generated.
   (c) **Misnamed.** On one hand, the legacy method name must not be changed automatically[3], and on the other hand, it's generally not possible for a merger to adjust the generated code to use the legacy method. As a result, there will be two methods for the same task in S_{i+1}, one using the legacy name, the other using the generated name. To avoid code redundancy, the former legacy method will be a completely generated delegation method, which calls the newly added generated method.
   Whether the other method is partly or completely generated, depends on whether the generated method has a body or not. If it has no method body, the former body of the legacy method will be used and the method will be partly generated.
   (d) **Copied.** A superfluous method, which is copied from G_i to S_{i+1}, will be annotated as completely generated.
2. **Fields:**
   (a) **Equal Name, Completely Generated.** G_i becomes S_{i+1}, and is marked as completely generated.
   (b) **Equal Name, Partly Generated.** S_{i+1} is a combination of G_i's declaration and S_i's initialization. It is annotated as partly generated.
   (c) **Misnamed.** The names of legacy fields are mapped to the generated fields. The legacy names will be kept, and the names of the generated fields will be replaced by the legacy names in every completely generated method in S_{i+1}.
   (d) **Copied.** A superfluous field, which is copied from G_i to S_{i+1}, will be annotated as completely generated.
3. **Types:**
   (a) **Inner Types.** are merged recursively
   (b) **Copied Types.** All members of a superfluous class, which is copied from G_i to S_{i+1}, will be annotated as completely generated.

Legacy elements which correspond to misnamed generated elements will be marked by the following annotation:

```
@Generated("de.htwgkn.mdre.gen",
    value="mappedTo:qualified.element.name")
```

4.2 Implementation

The reverse engineering tools are implemented as an Eclipse bundle, *MDREclipse*. The plugin contains the merger, an editor extension and supports auto folding. The merger requires two source directories that contain the legacy code S_i and the generated code G_i. In addition, the target directory for the merged code S_{i+1} has to be specified. If the target is not empty, the engineer is asked whether it should be cleared first. At the beginning of the merging process, all files from the

[3]Methods might be part of an API, on which other projects depend.

source S_i are copied to the target. Then, the generated files are merged into S_{i+1}. If the merger does not find a corresponding member for a generated member, the engineer will be asked how to proceed by a dialog. The merging progress is shown in a text console. The code parsing is based on the Java Development Tools (JDT) provided by Eclipse. JDT (Eclipse Foundation, 2010) allows to create abstract syntax trees (ASTs) from Java sources. These trees are traversed using the visitor pattern (Gamma et al., 1995).

The editor extension highlights generated code by changing its background color. This color can be configured in the plugin's preferences. Also, the extension protects the generated code from most manual changes. Typing, overwriting and deleting parts of generated code will be prevented. All of these features can be disabled in the preferences.

The last component of *MDREclipse* is the auto folding of generated methods. To use this feature, Eclipse's default folding has to be disabled and the *MDREclipse* folding activated. Generated methods will then automatically be folded in when a Java source file is opened.

5 CONCLUSIONS AND FUTURE WORK

This paper presents an iterative approach to transform reference systems into MDSD artifacts. We outlined our basic ideas in Section 2 and argued subsequently why separating manual and generated code into different files is problematic for our approach. In Section 4 we presented a code merger. We explained various merging situations and discussed under what circumstances merging cannot be done completely automatically.

We do not claim that our approach solves all problems in the field of reverse engineering towards MDSD. Code separation and code merging during an iterative process is a rather technical aspect; feature extraction, on the other hand, is a semantical challenge that is very hard or impractical to solve at a generic level.

Please note, that even though in each iteration step we perform only equivalence preserving modifications, the approach can very well be interspersed with forward engineering steps, as long as the reverse and forward engineering steps are performed separately one after the other. During a forward engineering iteration, the engineer can modify the model level artifacts, i.e. DSLs, models, and generators, as well as the manually crafted code. The modified software can then, in a next step, again be reverse engineered towards a greater MDSD proportion.

Finally, it should be noted that the success of the reverse engineering process strongly depends on the quality of the code base. If the reference system is well structured and coded, feature extraction becomes easier and the model level artifacts become better structured as well.

ACKNOWLEDGEMENTS

This research has been funded by the German BMBF (Ministry for Education and Research) under the umbrella of the *IngenieurNachwuchs* program. In addition, we would like to thank our industrial partners, Seitenbau GmbH and Sybit GmbH, for their support.

REFERENCES

Eclipse Foundation (accessed on July 26th 2010). *Java Development Tools*. http://www.eclipse.org/jdt/.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Gosling, J., Joy, B., Steele, G., and Bracha, G. (2005). *The Java Language Specification (Third Edition)*. Addison-Wesley.

Kieburtz, R., McKinney, L., Bell, J., Hook, J., Kotov, A., Lewis, J., Oliva, D., Sheard, T., Smith, I., and Walton, L. (1996). A software engineering experiment in software component generation. *Software Engineering, International Conference on*, 0:542.

Krueger, C. W. (1992). Software reuse. *ACM Comput. Surv.*, 24(2):131–183.

Ladd, D. A. and Ramming, J. C. (1994). Two application languages in software production. In *VHLLS'94: Proceedings of the USENIX 1994 Very High Level Languages Symposium Proceedings on USENIX 1994 Very High Level Languages Symposium Proceedings*, pages 10–10, Berkeley, CA, USA. USENIX Association.

van Deursen, A. and Klint, P. (1998). Little languages: little maintenance. *Journal of Software Maintenance*, 10(2):75–92.

Völter, M. (2009). MD* best practices. *Journal of Object Technology*, 8(6):79–102.

Walter, R. and Haase, O. (2008). How to make legacy code MDSD-ready. In *Second Workshop on MDSD Today*.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS