An Evaluation of the Graphical Modeling Framework (GMF) Based on the Development of the CORAS Tool

Fredrik Seehusen and Ketil Stølen

SINTEF Information and Communication Technology, Norway
{fredrik.seehusen,ketil.stolen}@sintef.no

Abstract. We present an evaluation of the Graphical Modeling Framework (GMF) based on our experiences in developing an editor for the risk modeling language CORAS using GMF. Our main hypothesis is that GMF shortens development time and results in more reliable and maintainable systems than alternative approaches which are not based on code generation. We conclude that the hypothesis is true, but that the answer is not as clear cut as we initially believed, and that there is still a large potential for improvement.

Keywords: Model-Driven Development, GMF, Domain Specific Languages, Evaluation.

1 Introduction

We present an evaluation of the Graphical Modeling Framework (GMF) [5] based on the experiences we got from developing a graphical editor for the risk modeling language CORAS [11]. GMF builds on the principles of Model-Driven Development (MDD) which advocate the use design time models as the basis for code generation. In the area of domain specific languages, the idea of generating code from specifications is not new; so-called parser generators have been in use for a long time. These tools usually take an EBNF-grammar as input, and produce parsers that transform concrete syntax (such as source code) into abstract syntax. The idea of GMF is similar, the main differences being that the language grammar is specified as an Ecore model (similar to a UML class diagram model) instead of EBNF, that the concrete syntax is graphical (not text-based as in the traditional case), and that GMF not only generates a parser, but also much of the language editor.

The language that we developed an editor for is called the CORAS language. The language is graphical and used for the purpose of documenting a risk analysis. The CORAS language editor was developed as part of a book release on the CORAS risk analysis method [11].

Software development in the industry is usually not based on code generation from models. We are therefore primarily interested in comparing the use of GMF to a development approach which is not based on code generation. Our main

J. Cabot and E. Visser (Eds.): ICMT 2011, LNCS 6707, pp. 152–166, 2011.

hypothesis is that GMF shortens the development time and results in more reliable and maintainable systems than alternative approaches which are not based on code generation.

Our evaluation suggests that GMF does shorten development time, but to a smaller degree than we initially expected. The main reason for this is that the code which is generated by GMF had to be modified, and this turned out to be very time consuming. Our evaluation furthermore suggests that GMF may result in more reliable systems, but that it does not result in more maintainable systems.

The main lesson learned during the development of the CORAS tool, was that there is a huge difference (in practice) between transformations that produce code that to little or no degree should be modified (like parser generators or compilers), and transformations that produce code that must be modified. In the former case, it is reasonable to hide the details of the transformation from the developer through configuration options, but in the latter case this is more problematic. In fact, we believe that in the latter case, the transformation should not only be visible to the developer, but that the developer should be expected (and accommodated) to modify the transformation much in the same way that the developer is expected to modify the generated source code.

The rest of the paper is structured as follows: in Sect. 2 we introduce basic concepts of GMF, in Sect. 3 we describe our research method and hypothesis, in Sect. 4 we describe relevant facts about the CORAS tool and its development process, in Sect. 5 and Sect. 6 we evaluate GMF w.r.t. our hypotheses and suggest improvements, respectively. Finally, in Sect. 7 we present related work, and in Sect. 8 we provide conclusions.

2 The Graphical Modeling Framework (GMF)

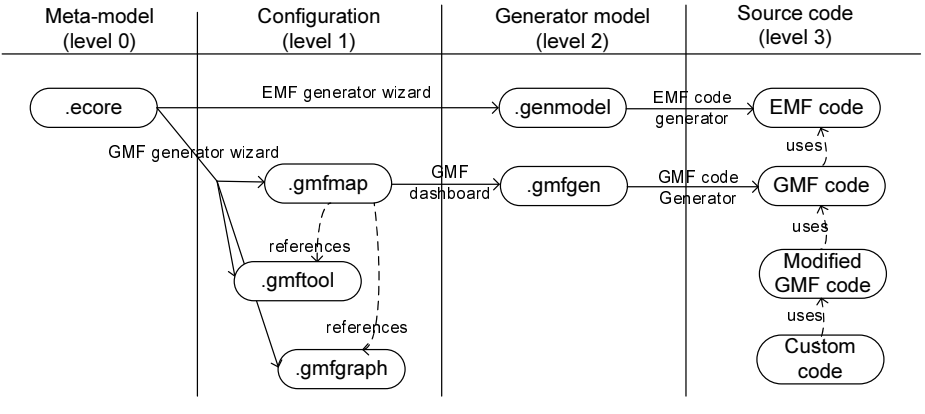

Fig. 1. Transformation in GMF

GMF is a framework for developing domain specific languages. GMF is built on the Eclipse Modeling Framework (EMF) [3] and the Graphical Editing Framework (GEF) [4]. In the context of GMF, EMF is used to define the meta model or abstract syntax of languages (expressed in Ecore), and for generating code for creating, editing, and accessing models. GEF is a framework that supports the development of graphical editors. In the context of GMF, GEF is used to implement the concrete graphical syntax of languages and for editing of concrete syntax.

The transformations of GMF are illustrated in Fig. 1. The top-most transformation consisting of the two arrows taking *.ecore* into EMF code will be referred to as the *EMF transformation* while the arrows taking *.ecore* into GMF code will be referred to as the *GMF transformation*.

As illustrated in Fig. 1, we distinguish between 4 different levels.

The meta model level (level 0). This level contains the meta model which is the basis of code generation. The meta model used by GMF must be stored in a format called Ecore.

The configuration level (level 1). This level contains the configuration model which is used by the GMF transformation (this is not used by the EMF transformation). The configuration model consists of three files: one file which is used to specify the shapes of the graphical constructs of the language (*.gmfgraph*), one file which specifies the language constructs which should appear in the tool palette of the language editor (*.gmftool*), and one file (*.gmfmap*) which maps the items of the tool palette (defined in *.gmftool*) to the graphical language constructs (defined in *.gmfgraph*) and to the meta model elements.

The default transformation from the meta model at level 0 to the configuration model at level 1 is based on a series of dialog windows where the user has to specify what elements of the meta model should be represented as arrows or nodes in the language editor.

The generator level (level 2). This level contains the code generator models for both the EMF and the GMF transformations. Both generator models contain a number of parameters that allows the user to customize the source code generated from the generator models.

The transformation from level 1 to level 2 (or from level 0 to level 2 in the case of the EMF transformation) is a one-button transformation as opposed to the dialog based transformation.

The source code level (level 3). This level contains the source code. Here we distinguish between:

- EMF code generated by the EMF transformation;
- GMF code generated by the GMF transformation, which makes use of the EMF code as well as the GEF framework;
- modified GMF code, i.e. generated GMF code which has been modified after generation;

– custom code which has not been generated from the meta model, but which makes use of the EMF-code and the GMF code.

Note that we do not distinguish between generated EMF code and modified EMF code. The reason is that we did not have to modify the generated EMF code (even though it is possible to do so).

The transformation from level 2 to level 3 is also a one-button transformation. However, the GMF transformation from level 2 to level 3 (which is written in a language called Xpand) can be customized. This was achieved by putting the files containing the transformation specification in a special folder with the same structure as the GMF transformation. When the transformation is executed, the transformation in the special folder will be used instead of the original transformation. The effect is essentially the same as modifying the original transformation directly.

In theory, all the modified GMF-code could have been specified in the transformation from level 2 to level 3, but GMF is not intended to be used in this way. Instead, GMF code which is modified after code generation can be tagged to ensure that modified code is not overwritten when code generation is executed at some later point in time.

3 Research Method and Hypotheses

According to [12], research evidence is gathered to maximize three things: the generalizability of the evidence over populations of actors; the precision of measurement of behavior; the realism of the situation or context. According the classification of research strategies of [12], our evaluation of GMF is a *field study*. Field studies are high on realism, but low on generalizability and precision.

The idea of evaluating GMF did not come about until after we had developed the CORAS tool. Therefore the development of the CORAS tool was not in any way biased by our evaluation.

We are primarily interested in comparing the use of GMF to a development approach which is not based on code generation. For us, a realistic alternative to using GMF, would have to use Eclipse and GEF. The main differences between the GMF approach and the alternative approach is that using the alternative approach, we would have to (1) write the code that uses GEF instead of generating it using GMF and (2) write the code with similar functionality as the EMF code, instead of generating it using the EMF transformation. Throughout the rest of this paper, whenever we write *the alternative development approach* (or alternative approach), we refer to the approach described above.

Note that we did not actually develop the CORAS tool using the alternative approach. Any judgment about the CORAS tool, for instance how long it would have taken to develop the CORAS tool had we used the alternative approach is therefore based on educated guesses rather than hard facts.

3.1 Hypotheses

The main reason why we chose to use GMF, was that we believed that it would shorten the development time. In fact, one of our colleagues claimed that it would only take 2 days to develop the editor using GMF; another colleague estimated that it would take about 2 weeks. Granted, our colleagues were not familiar with the requirements of the tool, but we were nevertheless given to believe that using GMF would shorten development time. Our first hypothesis is therefore:

H1: The time spent on developing the CORAS tool using GMF is shorter than the time spent on developing the tool had we followed the alternative approach.

The CORAS tool was intended to be used in industrial settings, often on the-fly during customer meetings. Consequently, the most important requirement of the CORAS tool (aside from the fact that it had to support the desired functionality) was its continuous delivery of correct service, also known as its *reliability* [2]. Our second hypothesis is therefore:

H2: The CORAS tool is more reliable when developed using GMF than it would have been if developed using the alternative approach.

From the start, it was anticipated that the CORAS tool would be extended into new prototype tools according the requirements of projects we were involved in. Already, we have made three such extensions, and further extensions are planned in the future. The situation can be described by an inheritance tree where the initial CORAS tool is the root of the tree, and each of the other tree nodes represent tool extensions that inherit functionality from their parent nodes in the tree. We are interested in evaluating how well GMF supports the maintainability of such an inheritance tree. For the purpose of our GMF evaluation, we therefore say that the *maintainability* of a tool is measure of how well it can be extended into an inheritance tree that satisfies the following requirements:

- The development of a tool in the tree should not affect any of the parents of that tool. This requirement is to make sure that the development of new tools does not cause the tools they inherit from to become unreliable.
- Updates made in a tool should automatically or semi- automatically be propagated to all its children in the inheritance tree. If we did not have this requirement, and made a change in the root node for instance, we would have to manually update all the other tools of the tree. This would be time-consuming, and it could potentially lead to reduced reliability.

Our third hypothesis is the following

H3: The CORAS tool is more maintainable when developed using GMF than it would have been if developed using the alternative approach.

4 Developing the CORAS Tool Using GMF

In this section, we present facts regarding the CORAS tool and the development setting.

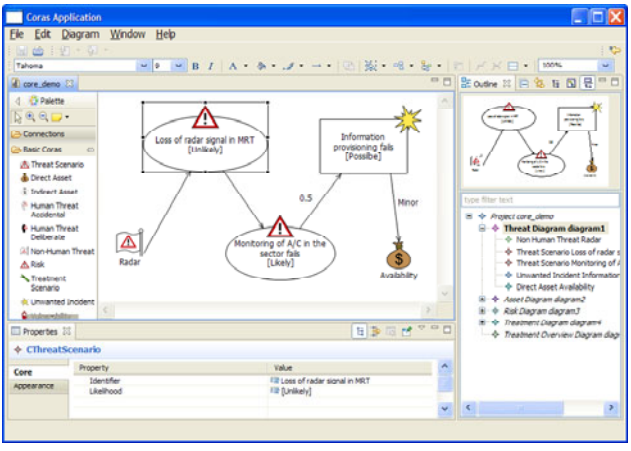

Fig. 2. Screenshot of the CORAS tool

4.1 Development Setting

Initial discussions about the CORAS tool started in September 2009. The tool requirements were completed in December 2009. The tool development was carried out by a single developer from the end of December 2009 until early May 2010. From February 2010 to May 2010, the developer worked close to full time on the CORAS tool development. The total time spent on the development was about 3.5 man months.

The developer is employed as a researcher and has a computer science background. His programming experience was mostly based on university graduate courses. The developer was somewhat familiar with Eclipse (the integrated development environment that supports GMF), but had no experience with GMF or the components that GMF builds on (EMF and GEF).

The developer was under pressure to finish the CORAS tool as quickly as possible. The development approach was based on trial and error; thereby learning exactly what was needed to get the job done. We do not argue that this is the best development approach, but merely give an account of what happened. Moreover, we believe that this situation is not uncommon in many industrial software projects.

4.2 The CORAS Tool

The CORAS tool is a graphical editor for the CORAS language which is used for the purpose of documenting risk analyses. CORAS diagrams are essentially graphs with arrows and nodes. The CORAS tool supports five kinds of diagrams, and it also supports a hierarchical structure in which one construct may be decomposed into other constructs to an arbitrary depth. A screenshot of the tool is shown in Fig. 2.

Table. 1 summarizes some statistics about the CORAS tool implementation. Here the number of modified generated code lines is estimated to be 10 percent of the code lines of the classes that had to be modified. All the other values are accurate facts (not estimations).

We have distinguished between code which has been generated by the EMF code generator, and code which has been generated by the GMF code generator. Notice that it is only the generated GMF code that had to be modified. Notice also that 18.5% of the generated GMF classes had to be modified, and that about 0.6% of the code lines had to be modified.

Table 1. Source code facts

	EMF	**GMF**	**Total**
Generated classes	131	465	595
Generated classes modified	0	86	86
Generated code lines	15220	78584	93804
Generated code lines modified	0	4767	4767
New classes	0	48	48
New code lines	0	4436	4436
Proportion of classes modified	0 %	18.50 %	14.45 %
Proportion of code lines modified	0 %	0.6 %	0.5 %

5 Evaluation of GMF

In this section, we evaluate GMF w.r.t. each of the hypotheses described in Sect. 3.

H1: The time spent on developing the CORAS tool using GMF is shorter than the time spent on developing the tool had we followed the alternative approach.

The development of the CORAS tool took about 3.5 man-months, which was more than we anticipated. Since we did have a stable meta model to begin with, it actually only took a couple of days to generate an initial version of the editor containing most of the required functionality. The problem was the huge amount of time required to modify the code that was generated.

Not much of the generated code had to be changed (see Table. 1), but finding the right places in the generated code to implement the modifications was very time consuming. It was not uncommon to spend days to search for the right place to edit, only to change a couple of lines here and there.

So, why did it take so long? One reason is that the developer had no prior experience with GMF, or the technologies used by GMF (GEF and EMF), and as it turned out, too little knowledge of Eclipse. Any of these technologies are challenging to learn on their own right; learning them all at the same time is even more challenging. Another aspect is poor documentation. In particular, the source code and the API contained very little documentation. Despite the fact that we had a book on GMF [7], this book did not address the very specific changes that we had to make to the generated code. Most of the answers we got

were found browsing the source code, as well as in blogs, forums, presentations, small tutorials found by searching the Internet.

A lot of the changes we made to the source code that was generated from GMF, had to be applied to numerous classes. That is, the same change had to be made to many classes of the same kind (usually meaning that they inherited from the same super-class contained in a library). Eventually the developer learned to modify the GMF transformations that generated the source code to implement these kinds of changes. Of course, if the developer had been aware of these things from the start, much time could have been saved. This is also true for the GMF transformation from the meta model to the generator model. Eventually we found an alternative to the GMF transformation wizard called EuGENia (which is built on top of Epsilon [9]) which allowed us to generate the graphical configuration files from an annotated meta model. This worked very well; it was very easy to learn and the execution of the transformation was reduced to pressing a button as opposed to clicking through a series of dialogs.

Up until this point, we have argued that most of the development time was spent on modifying the generated code. However, to address the hypothesis, we must ask whether GMF shortens development time. To answer this question, it is useful to distinguish between the EMF transformation and the GMF transformation (as explained in Sect. 2). The former transformation generated code which we did not have to modify. In our view, the use of the EMF transformation clearly saved us a lot of time. This should come as no surprise, since the EMF transformation is similar to well-known parser generators which are clearly useful.

The GMF transformation is more complex than the EMF transformation. First of all, it has a lot more configuration options (e.g. for customizing the graphical constructs that should be implemented). Our initial goal was to do as much of the customization as possible on the meta model level and on the configuration level (see Fig. 1), so quite some time was spent on e.g. figuring out whether or not it was possible to specify the kind of graphical construct that we needed using the configuration files (instead of having to modify the source code). It turned out that it was not possible, and eventually the configuration files were mostly ignored in favor of source code modification.

The main difference between the GMF transformation and the EMF transformation from the perspective of time use was that the code that was produced by the GMF transformation had to be modified whereas this was not the case for EMF. As already explained, the code modification turned out to be the biggest time drain. Since figuring out how to modify the code took such a long time, it is hard to say for sure whether or not it would have been faster to write all the generated code from scratch. However, if we were to make a new graphical editor with the knowledge we have today of how GMF works, we definitely think that using the GMF transformation will be faster than writing the code from scratch. We therefore believe that the hypothesis H1 is true, but that the answer is not as clear cut as we believed it to be when we started developing the tool.

H2: The CORAS tool is more reliable when developed using GMF than it would have been if developed using the alternative approach.

As mentioned in the previous section, GMF had some bugs, but these were more of the kind that slowed us down rather than making the generated code unreliable. After having tested the CORAS tool on students, we have not found any serious error in the GMF code (apart from the copy/paste functionality). The few errors that were found were caused by us rather than the GMF code.

The generated GMF code seems to be based on a lot of standard design patterns. This is probably mostly an advantage, but not necessarily if the developer who has to modify the generated code is not familiar with those patterns. Had the developer written the code from scratch, he would have used patterns that he was familiar with and could understand, and would therefore have a much better understanding of how the code works. Furthermore, the code base would likely have been smaller and more manageable.

In general we can say this: the time and effort spent on verifying the GMF transformation (and the code produced by it) is probably greater than the time and effort we would have spent on verifying the code we would have written had we followed the alternative approach. In this sense, the generated code is potentially more reliable. However, if the generated code must be modified, then the picture is less clear. The reason for this is that modifying code that is unfamiliar to the developer and which furthermore may be based on patterns that the developer is not familiar with may cause the developer to modify the code in a way that it is not intended to be modified. However, had we followed the alternative approach, we would still have to use GEF which is based on certain design patterns that must be followed in order to use the framework properly. In fact, it is probably less likely that GEF is used incorrectly in the generated code than it is that GEF had been used incorrectly had we followed the alternative approach.

As explained in Sect. 2, the first time the GMF transformations are executed, output is generated which has to be modified. When the transformations are executed again, they will overwrite parts of the output which was generated from the previous execution, and leave other parts unchanged (presumably the modified parts). Regarding the transformations from the meta model to the configuration level, and from the configuration level to the generator model, it was a bit unclear to us what parts would be overwritten, and what parts would be changed. This is of course a potential source of error. Regarding the transformation from the generator model to the source code, it was clear which parts where overwritten and which were not. The reason for this is that the developer had to explicitly annotate methods in the generated source code to indicate that they should not be overwritten during the next execution of the transformation. However, this approach does by no means guarantee that the transformation does not generate errors in the code the next time it is run. That is, changes made in the meta model or on the configuration model could result in a modification in methods that have been specified not to be overwritten.

The considerations above are potential sources of errors, but they did not as far as we know, turn out to be any problem w.r.t. to the reliability of the CORAS tool. However, it is difficult to determine whether this was a coincidence or not.

In conclusion, we believe that the hypothesis is true. The main reason for this is that we did not discover many errors which affected the reliability of the GMF code; most of the errors we found were caused by us. Source code which is generated and should be used many times is more likely to be subjected to a more detailed verification process than the code which is written from scratch. Despite the danger of modifying the generated code in a way that violates the design patterns that are used, this danger also exists if the code had been written from scratch using third party libraries such as GEF. The possibility of modifying the output of a transformation may potentially lead to errors when transformations have to be executed several times. However, according to our experience, this did not affect the reliability of the CORAS tool to a noticeable extent.

H3: The CORAS tool is more maintainable when developed using GMF than it would have been if developed using the alternative approach.

As explained in Sect. 3, the CORAS tool has been extended into three new tool versions. In the process of figuring out how to maintain these different versions, we discovered that code generation actually made things a lot more difficult for us. We will now explain this.

Consider the scenario in which we want to extend tool A into a tool B. By this we mean that we want to develop a new tool B that has a lot of the same functionality as A, but some of the functionality will be different. A standard solution to the problem is the following: package A into a component (e.g. a java jar file or an Eclipse plugin) that can be used by B, e.g. by invoking methods exposed by the public interface of the component or sub-classing public classes of A. The problem, however, is that this standard solution is not supported in the GMF code generation setting. To see this, consider Fig.3. Here tool A has been decomposed into two parts: $Model_A$ and $Code_A$ where the idea is that $Code_A$ is generated from $Model_A$ (the same applies for tool B). To extend A into B, it is possible to both extend the model and the code. Imagine that we try to apply the standard solution to the problem. We get the following scenario: First we package the source code of A into a component, then we make a folder/project for B and import component A into it. Suppose that the model has to be extended. Since it is not possible to package the model into a component like the source code, $Model_A$ is copied into the folder of B. The problem now is that when code $Code_B$ is generated from $Model_B$, $Code_B$ will contain much of the same functionality that is contained in package A and it would be very inconvenient to try to replace the functionality in $Code_B$ that overlaps with the package A with invocations to A.

The problem is most notable in the cases where $Code_A$ has been modified after it has been generated from $Model_A$. If all the modifications had been expressed in the transformation or in the model, then there would be no need to package A into a component. However, the problem is still present since GMF does not support the possibility of packaging models or transformations into components.

In summary, we can conclude that the hypothesis H3 is false. The standard way of extending tools is not supported in the GMF code generation setting.

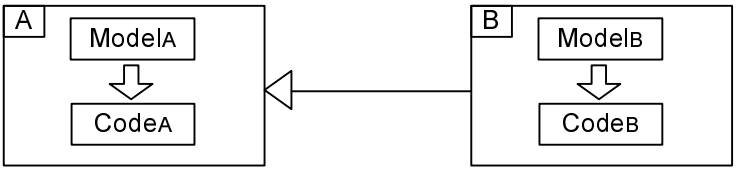

Fig. 3. Extending A into B

6 Suggested Improvements

The issues regarding poor documentation or the fact that the GMF architecture was hard to understand are not specific to transformation based development approaches. In this section, we therefore focus on the transformation aspect of the GMF framework. In particular we will focus on the GMF transformation (since we did not find any problems with the EMF transformation).

In Table. 2, we have given an overview of our main suggested improvements and the problems they are related to. In the following, we discuss each suggestion in turn.

Table 2. Summary of problems and suggested improvements

Problem	**Suggested improvement**
The dialog based GMF transformation did not work and was time consuming to use	Avoid human interaction during transformation execution
Unclear whether changes to generated artifacts would be preserved under subsequent transformation executions	Expose the transformation language to the end user / avoid changing generated artifacts
Lack of modularity support	Incorporate traditional programming language solutions to modularity

6.1 Avoid Human Interaction During Transformation Execution

The GMF transformation from the meta model to the graphical configuration models was wizard based, i.e. the user could configure the transformation through a series a dialogs. If the transformation was run a second time, the transformation would remember the dialog settings of the previous run of the transformation. Or at least that was the idea. This transformation did not work properly, but if we ignore that, we still think that the use of dialogs is bad. First, clicking through a series of dialogs each time the transformation is run is time consuming. Even though we had a stable meta model to begin with, we ended up generation the editor countless of times. Second, since the mechanisms of how the transformation worked was hidden from the user, it was unclear how a change in the dialog option would affect the transformation.

Based on our experience, we therefore think the use of transformation executors that require human interaction is a bad idea. As previously discussed, we eventually ended up using an alternative (called EuGENia) to the GMF transformation wizard for producing the graphical configuration model. This approach enabled us to annotate the meta-model with GMF-annotations, and then to generate the configuration files from the annotated meta-model without any human interaction. This approach worked far better than the GMF dialog based approach.

6.2 Expose the Transformation Language to the End User

The generated artifacts on all three levels had to be modified. For the GMF transformations that generated the configuration level artifacts and the generated model, it was unclear whether or not the modifications to the artifacts would be preserved upon subsequent executions of the transformations. One of the reasons for this was that the transformations were hidden from the user; without understanding how the transformation works it is also hard to understand how the transformation generates its output.

We therefore suggest that the transformation should be made visible to the user in the same manner as source code is often made available to programmers. Furthermore, we believe that for the GMF model transformations (i.e. the transformations that do not generate the source code), it is better to modify the transformations that generate the model artifacts (as opposed to modifying the generated artifacts) because this gives the user precise control of what the transformation generates.

This suggestion is made on the basis of our experience with using the language Epsilon Object Language (EOL) [8], which we used to customize the GMF transformations that produced the configuration level models and the generator model. This approach worked very well. EOL removed the need of modifying the generated artifacts, and enabled us precisely control the output of the transformation. Furthermore, it was not more difficult or more time consuming to customize the transformation than to modify the generated artifacts.

One might think that requiring the end user to learn a new language (the transformation language) could be a problem. However, our experience with EOL suggests otherwise. In fact, after only seeing a couple of examples of EOL code, we were able to customize the transformation as needed. Contrasting this to the months spent on modifying the generated GMF source code, the effort required to learn how to use the language becomes negligible.

Regarding the GMF transformation (written in Xpand) that generated the source code, it was fairly clear what parts of the code the Xpand transformation would change or leave unchanged when it was executed. This is because Xpand enables the user to annotate methods in the generated code to indicate whether or not the methods should be overwritten when the transformation is executed again. However, we believe that it would have been helpful increase the granularity and flexibility of this annotation scheme. For instance, it would have been

helpful indicate that certain *parts* of a method (not just the whole method) have been modified and should not be overwritten.

W.r.t. source code generators, we do not have enough evidence to determine whether or not the need of modifying the generated source code (as opposed to modifying the transformation) is a bad idea.

6.3 Incorporate Traditional Programming Language Solutions to Modularity

The GMF transformation from the generator model to the source code is can be customized/extended as explained in Sect. 2. However, we do not believe that this manner of customization is a good one; it is essentially the same as copying the source of a library into the current project and then overwriting the code. It is better to package the library into a component which can be invoked through a public interface or extended through sub-classing public classes. We believe that the transformation language should have similar extension capabilities. After all, GMF can be seen as common library, with the difference that certain parts of the code are transformations.

7 Related Work

There are not many empirical studies on applying Model-Driven Development (MDD) in industry settings. A survey of literature reporting experiences from applying MDD in industry settings can be found in [13], but there is too little evidence to allow generalization of the results. Nevertheless, many of case studies mentioned in the surveys report about a 25% productivity gain using MDD over conventional development. The paper [13] does not address GMF in particular. Papers that do consider GMF in particular are: [1,6,10,14]. Both [1] and [14] compare GMF against other domain specific language tools. According to [1], GMF is the most difficult to use, but it generates very usable editors. The paper [14] compares GMF against Microsoft DSL Tools, and concludes that GMF seems to be better accepted by participants of their case study.

In [6], the authors report on the experiences of developing a network modeling tool using GMF. They make two main conclusions: (1) a high level of programming experience is needed to use GMF properly, and that GMF should be made easier to use by domain-experts with no particular programming experience; (2) GMF has shortcomings in providing support for modeling at different abstraction levels.

In [10], the authors report on a case study where software with identical functionality was created twice: once using conventional development without code generation, and once using GMF. The study concludes that development using GMF was about 9 times faster than conventional development, and that GMF produced much higher quality of code. A threat to the validity of the study was that the conventional development was not based on EMF or GEF, but on .NET and an, according to [10], immature graphical library.

8 Conclusion

We have presented an evaluation of GMF based on our experiences in developing the CORAS tool. Our hypothesis was that GMF shortens development time, and that it results in more reliable and maintainable systems than an alternative development approach which is not based on code generation.

Our evaluation suggests that GMF does shorten development time, but to a smaller degree than we initially expected. Furthermore, we believe that the use of GMF may result in more reliable systems, but that it results in less maintainable systems according to our definition of maintainability.

For code generators that produce code that to a little or no degree should be modified (like parser generators, compilers, or EMF to a certain extent), it is perfectly reasonable to hide the transformation from the developer; letting all the work be done at the model level and in customization files. However, we believe that this is not OK for transformations (e.g. the GMF transformation) that must be extensively modified (either by modifying the transformation itself or the generated artifact). In fact, we believe that in this case, the transformations should be pushed to the forefront of the development; that the developer should be expected and accommodated to modify the transformation. GMF does offer the possibility of modifying one of the three GMF transformations, but in GMF tutorials and other documentation, this possibility is treated more like an advanced feature than an important part of the development on par with e.g. configuration model customization. In fact, the developer was not even aware of the possibility until far into the development process. Finally, we believe that the transformations should be extended in much the same way as libraries are extended when programming. Currently, this is not possible in the transformations of GMF.

Acknowledgments. The research on which this paper reports has been carried out within the DIGIT (180052/S10) and the EMERGENCY projects (187799/S10), funded by the Research Council of Norway, the MASTER and the NESSoS projects, funded from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreements FP7-216917 and FP7-256980, respectively.

References

1. Amyot, D., Farah, H., Roy, J.-F.: Evaluation of Development Tools for Domain-Specific Modeling Languages. In: Gotzhein, R., Reed, R. (eds.) SAM 2006. LNCS, vol. 4320, pp. 183–197. Springer, Heidelberg (2006)
2. Avizienis, A., Laprie, J.-C., Randell, B.: Fundamental Concepts of Dependability. Research Report No 1145, LAAS-CNRS (2001)
3. Eclipse. Eclipse modeling framework project (emf) (2011), `http://www.eclipse.org/modeling/gmp` (visited February 8, 2011)
4. Eclipse. Graphical editing framework (gef) (2011), `http://www.eclipse.org/geff` (visited February 8, 2011)

5. Eclipse. Graphical modeling project (gmp) (2011), `http://www.eclipse.org/modeling/emf` (visited February 8, 2011)
6. Evans, A., Fernández, M.A., Mohagheghi, P.: Experiences of developing a network modeling tool using the eclipse environment. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 301–312. Springer, Heidelberg (2009)
7. Gronback, R.C.: Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit. Addison-Wesley, Reading (2009)
8. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The epsilon object language (EOL). In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 128–142. Springer, Heidelberg (2006)
9. Kolvos, D.S.: An Extensible Platform for Specification of Integrated Languages for Model Management. PhD thesis, Department of Computer Science, University of York (2008)
10. Krogmann, K., Becker, S.: A case study on model-driven and conventional software development: The palladio editor. In: Proc. of tSoftware Engineering 2007 - Beiträge zu den Workshops, Fachtagung des GI-Fachbereichs Softwaretechnik, vol. 106, pp. 169–176. GI (2007)
11. Lund, M.S., Solhaug, B., Stølen, K.: Model Driven Risk Analysis - The CORAS Approach. Springer, Heidelberg (2011)
12. McGrath, J.E.: Groups: Interaction and performance. Prentice-Hall, Englewood Cliffs (1984)
13. Mohagheghi, P., Dehlen, V.: Where is the proof? - A review of experiences from applying MDE in industry. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 432–443. Springer, Heidelberg (2008)
14. Pelechano, V., Albert, M., Muñoz, J., Cetina, C.: Building tools for model driven development. comparing microsoft dsl tools and eclipse modeling plugins. In: Proc. of the Actas del Taller sobre Desarrollo de Software Dirigido por Modelos. MDA y Aplicaciones. CEUR Workshop Proceedings, vol. 227 (2007)