Implementing a Domain-Specific Modeling Environment For a Family of Thick-Client GUI Components

Milosz Muszynski
Tanner AG
Kemptener Str. 99
88131 Lindau im Bodensee, Germany
milosz.muszynski@tanner.de

Abstract

This paper focuses on the practical aspects of building a Domain-Specific solution for families of thick-client GUI components. The components lack business-logic, they are parts of thick-client applications and are built to be used through well-defined data oriented interfaces. We've observed that with such GUI components it is not always possible to keep the generators simple. Generating the glue code only and pushing the variability into the framework was not always feasible. We found it necessary to generate relatively large amounts of code. On the other hand, we noticed that with the characteristic for our components static variability it is possible to make the modeling process reference implementation driven. We describe an environment that focuses on using reference implementations as sources of metamodels and drivers of the generation process, effectively avoiding duplication of code between the reference implementation and the generator.

1. Introduction

In a typical domain-specific modeling process, models are expressed in a domain-specific language (often visual and defined through a metamodel). Generators use information from the model and produce code that constitutes the product. According to [5, 6], generators should be simple and concentrate on the glue code. Implementation details should not be generated, but rather pushed off into the framework. Our experiences with DSM-based generating solutions for thick-client GUI components show that such approach is not always possible or optimal. We think that for GUI components generating relatively large amounts of code is necessary. Since the source of the generated code is usually a reference implementation, problems stemming from duplication between the reference implementation and the generator typically arise. For product families with static variability, such as our GUI components, we find it easier to reverse the logic of the modeling process. Instead of performing the reference implementation to generator migration, we insert generating instructions in the reference implementation itself. We make the reference implementation a central source of information for both metamodel and the generation process. In the following sections we explain this idea in more detail.

2. GUI Component Generators

In theory, generators for Domain-Specific Modeling environments should be kept as simple as possible [5, 6, 8]. The optimal solution is when generators are implemented according to the following guidelines:

1. Variation is moved to the specification language.
2. Low-level common implementation issues are pushed into the framework.
3. Only the glue code is generated.

For GUI components, the above guidelines cannot always be followed. Moving implementation issues to the framework and generating only the glue code turns out to be hard. There are several reasons for that:

1. Moving implementation issues into the framework is time-consuming as GUIs often involve boilerplate code generated by third party GUI tools. Making such code universal and putting it in a framework requires a paradigm shift and a substantial investment.
2. Frameworks usually imply programmatic approach. We want to be able to generate GUI specifications that are not programmatic, but rather textual. For example, GUI menu creation code is easier to comprehend and manage using textual specification than a programming API.
3. We are relying on tools or environments that accept certain kind of input. For example, some GUI infrastructures accept filled out resource structures. In such case we need to generate entire structure definitions in order to satisfy environment requirements.
4. We want our GUI components to be exactly tailored to contain only the functionality that is needed.
5. We want our GUI components to conform to a company standard and be comprehensible to people who are not involved in the Domain-Specific Modeling environment.

Because of the above reasons we need generators that generate relatively large amounts of code (or other artifacts) rather than just the glue code. In our case the small generator requirement is contradictory with the optimality of the solution. With small generators we need generic and oversized frameworks. Small, tailored GUI components require precise, rich generators.

3. Variability of GUI Components

Type of the variability space is crucial in characterizing the model and the generation process for a given domain. We find GUI components that are data interface driven and lack the business logic to have static variability. Variations between members of families of such products are static rather than behavioral. Static variability is much simpler to be controlled via models as the controlling logic is less dependent on ordering and multiple relationships. In many cases models controlling static variability are trees. For GUI components the models we used were almost always trees. We also noticed that for our GUI components the number of operations used by the generator to control the code generation process was quite limited. The following generator operations were sufficient to perform the generation task:

1. Setting an output file name or part of name using a value of a model class attribute.
2. Substituting text with a value of a model class attribute at a file or file fragment scope.
3. Conditional inclusion of text depending on a value of a model class attribute.

Another implication of static variability of GUI components is a simple model-walking algorithm. In cases in which model is a tree, model walking becomes very straightforward.

We noticed that static variability allows for significant simplifications in the modeling process. Simplicity of such models is a compensation for bigger generator sizes that we find inevitable when generating the desired components.

4. Justification for a Reference Implementation Driven Generation

As a consequence of having a non-trivial generator that produces relatively large amounts of code, we come to the question of a reference implementation. It is a common practice that the code that is inserted into generators should be functional and well tested. Code that is a basis for generation is usually being developed outside of the generator in a form of reference implementation. Usually, we develop reference implementation first, debug it and test, then we insert it into the generator. Insertion of a reference implementation code into a generator (code migration) can lead to the following problems:

1. It is time-consuming, as it requires adjusting to generator's syntactical requirements. Generators usually require some form of quotations and new line indicators in the inserted code. Code migration is difficult to automate, as we need to insert code-generating instructions into the migrated code. According to [10], code migration effort amounts to an additional 20-25% of the entire reference implementation development effort.
2. It results in code duplication. Theoretically, after migration, we could scrap the reference implementation. We could regenerate it from the generator. In practice, however, new versions, variations, features and bug fixes are reasons for a constant evolution of the reference implementation. It is often impractical to make all the changes in the generator and to always work with a generated reference implementation. The role of a reference implementation in the software development process implies that it ought to be manually written software that serves as a basis for generation.

Source code inserted in a generator becomes integrated with the generator's input script. In the rest of the paper we will call such an input script "generator's source code", although what we mean here is more precisely a textual input for the generator. By 'code in generator' we mean the programming code that has been incorporated into the generator's input script.

Synchronizing the code in reference implementation and the code in generator is in general a nontrivial task. The code in generator is decorated with syntactical and functional embellishments that the code in reference implementation does not have. Because of that, comparing code with the use of tools usually does not work. Applying changes to the code in generator is time-consuming as it is in general hard to find proper places for a given change to be applied.

Theoretically we should apply changes into the generator directly. It is recommended that we treat the generated code like something disposable, just like we treat the assembler produced by a third generation language compiler [5]. In practice however, it is unavoidable to evolve the reference implementation independently of the code in the generator. The most important reason for that is comes from the logic of a software development process. Reference implementation ought to be developed in a non impeded way. Evolution of a reference implementation by changing the code in the generator would limit the developer. Such limitation would be against the intention of the process in which reference implementation is a top quality muster for other code artifacts.

Another reason for the changes to be made directly in the reference implementation is the agility of the process. Changes are often connected with the debugging activity. Debugging and testing tools do not work at the generator source level. When we work in an IDE in a debugger, we do not see the generator source code. We only see the generated code. Even if we had an IDE that is generator-aware, the generator's code would really not be what we wanted to see. Generator's code is concerned with model walking rather than with the logic of our application. In an agile process, we tend to make changes in short cycles consisting of debugging, applying changes and testing. Forcing all changes through the generator would hinder the agility of the process.

The above reasoning should not be confused with successful and entirely convincing cases of domain-specific languages that provide debugging. If a domain-specific language is behavioral in nature, we can imagine debugging at a higher abstraction level. Such debugging implies treating generated languages like disposable output. With static variability models the situation is different. Here, domain-specific language is not behavioral hence there is no notion of domain-specific debugging. At the model's abstraction level we design application structure, which is orthogonal to the application's debugging and testing process.

For non-behavioral domain-specific models it is inevitable to use the reference implementation debugging process based on the target code rather than generator's source. Generated code should run without problems and we should not need to debug it and be tempted to change it. This is fine in every aspect, but this reasoning does not apply to the reference implementation. It is thanks to the reference implementation that the generated code is of a good quality. Reference implementation must evolve and must be well debugged and tested. Therefore, it is expected that we will have to migrate code from the reference implementation to generator on a regular basis.

Since code migration from the reference implementation to the generator is time-consuming, for static variability applications such as GUI components we suggest an inverse approach. Instead of keeping the main focus in the metamodel and generator, we propose to make a reference implementation the main source for both metamodel and the generator. Rather than migrating code into the generator, we add generating instructions in the reference implementation. In the following sections we will describe the suggested environment in more detail.

5. Modeling Process

If we take the framework out of the picture, the conventional Domain-Specific Modeling process involves approximately the following steps:

1. Designing, implementing, debugging and testing the reference implementation.
2. Developing a domain-specific metamodel (or any other form of specification of a domain-specific language).
3. Writing the generation input script, migrating code from the reference implementation into the script.
4. Creating models in terms of the metamodel (or writing specifications in the domain-specific language).
5. Running the generator to obtain a working product.

Should the reference implementation change, we need to repeat the migration of the reference implementation code again.

In the reference implementation driven Domain-Specific Modeling process we have the following steps:

1. Designing, implementing, debugging and testing the reference implementation.
2. Annotating the reference implementation.
3. Parsing the reference implementation to obtain a metamodel (or any other form of specification of a domain-specific language).
4. Creating models in terms of the metamodel (or writing specifications in the domain-specific language).
5. Running the generator to obtain a working product.

As we can see, in a reference implementation driven approach, we do not create metamodel explicitly and we do not write the generation-input script explicitly. We replace these two steps with one step – annotating the reference implementation. Should there be a change in the reference implementation, we simply run the generator again. We may need to adjust annotations in the reference implementation, but we do not need to migrate the code into the generation input script again.

Our process works under assumption that some generic generation input script exists. We may sometimes need to modify it. It is not a critical step in our process and, in most cases, it will not be required. Figure 1 illustrates the reference implementation driven process.

6. Annotating the Reference Implementation.

We could imagine hybrid situations in which part of the metamodel is contained in the reference implementation and part is contained in the metamodeling tool. In our GUI component environment we assume that the entire metamodel information is contained within the reference implementation. Annotations in the reference implementation must be sufficient to express information about the metamodel. We assume that the reference implementation consists of two kinds of artifacts: files and file fragments. We annotate file and selected file fragments with the following information:

1. Metaclass name.
2. Parent relationship name.
3. Parent metaclass name.
4. Class attributes contributions.

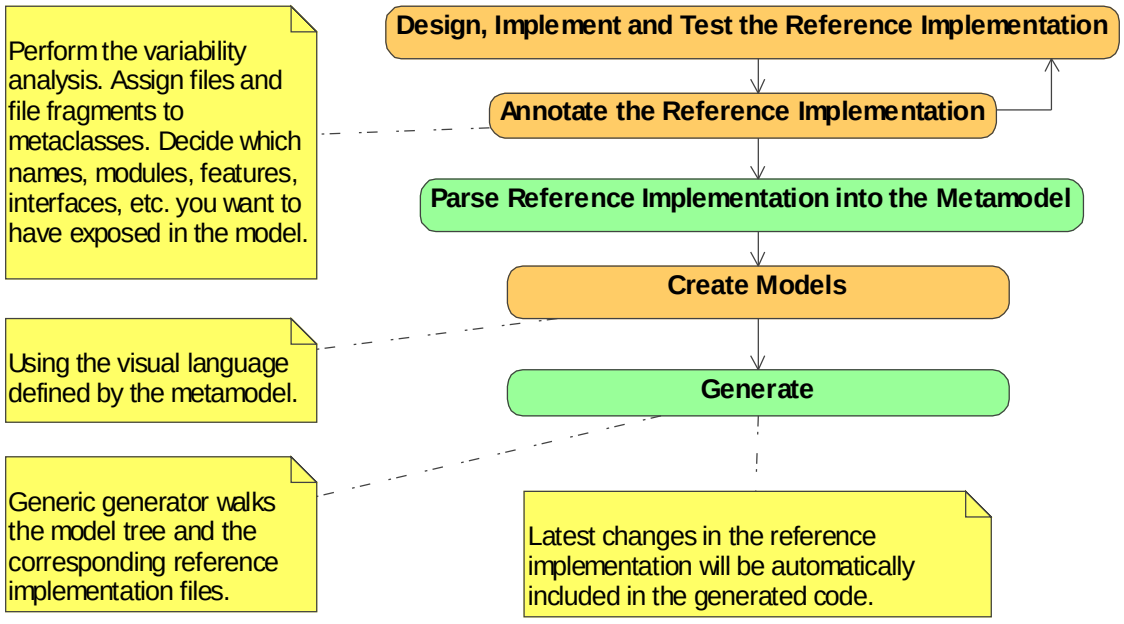

Figure 1. Reference implementation driven modeling process.

Each reference implementation file or file fragment has the following characteristics:

1. It is associated with exactly one metaclass.
2. It is associated with exactly one parent metaclass through a specified parent relationship.
3. It contributes attributes to exactly one metaclass.
4. It uses attributes of the associated metaclass and its predecessors.

Each metaclass has the following characteristics:
1. It has one or more file or file fragment associated with it.
2. It has attributes contributed by all files and file fragments associated with it.
3. It may be connected with maximum one parent metaclass through a specified relationship.

Sets of files and file fragments associated with particular metaclasses do not overlap. Annotated files and file fragments can be interpreted as a simple metamodel, in which the only relationship is a parent-child relationship. Metamodel implied by annotated files and file fragments is capable of expressing the required static variability of the GUI component product. In cases in which single relationships are not sufficient, model-walking annotations may be utilized to implement multiple relationships.

Each location in the reference implementation code has an implied associated metaclass. The following operations can be specified with regard to either an entire file or a file fragment:
1. Text substitution, driven by a value of a given model class attribute.
2. Conditional text inclusion, controlled by a Boolean value of a given model class attribute.

In addition, with regard to an entire file, its name or part of the name may be controlled by a value of a given model class attribute. We may also have annotations that force an association with a specified metaclass. Such annotations pertain to model walking.
In general, we can classify annotations by scope:
1. File
2. File fragment.

And by information usage:
1. Metamodel.
2. Code generation.
3. Metamodel and code generation.
4. Model walking.

The reference implementation parser parses annotations and generates the metamodel information. Metamodel information can then be fed into the modeling tool.

7. Generating Process

Once we have a metamodel produced by the reference implementation parser, we can create models. Models can then be walked by a generation process. The generation process uses information from the model to perform read-only operations on the reference implementation code to produce target code. Generating process performs the following steps:

- For each model node provided by the generic model walker:
  - For each reference implementation file associated with the given model node:
    - Process file scope operational annotations in the context of a metaclass associated with the file.
    - For every file fragment annotation in file:
      - Process file fragment operational annotations in the context of a metaclass associated with the fragment. If there is more than one instance of a fragment metaclass, loop through all instances and repeat the processing.

In addition to operational annotations, generator may also optionally process model-walking instructions. Generator ignores annotations pertaining exclusively to the metamodel. The connection between the model and the reference implementation is based on a list of reference implementation files associated with each metaclass. File fragments may trigger creation of some metaclasses in the metamodel extraction process, but generator is not aware of them until it processes a given file.

There are three major possible approaches when performing model walking:

- Generic walking – model walker starts from the root and then proceeds recursively to the leaves. Upon encountering fragments it switches to the fragment context, jumps to the fragment parent, loops through fragment class instances and then goes back to the previous context upon leaving the fragment.
- Walking with information embedded in reference implementation annotations – this approach is similar to the generic walking, only here the reference implementation annotations may cause the walker to skip the current flow and jump to a given model class.
- Custom walking – the user writes her or his own generator script and accesses the reference implementation files via "processReferenceImplFile(…)" kind of calls. This kind of walking allows for hybrid solutions – both reference implementation and generator-driven.

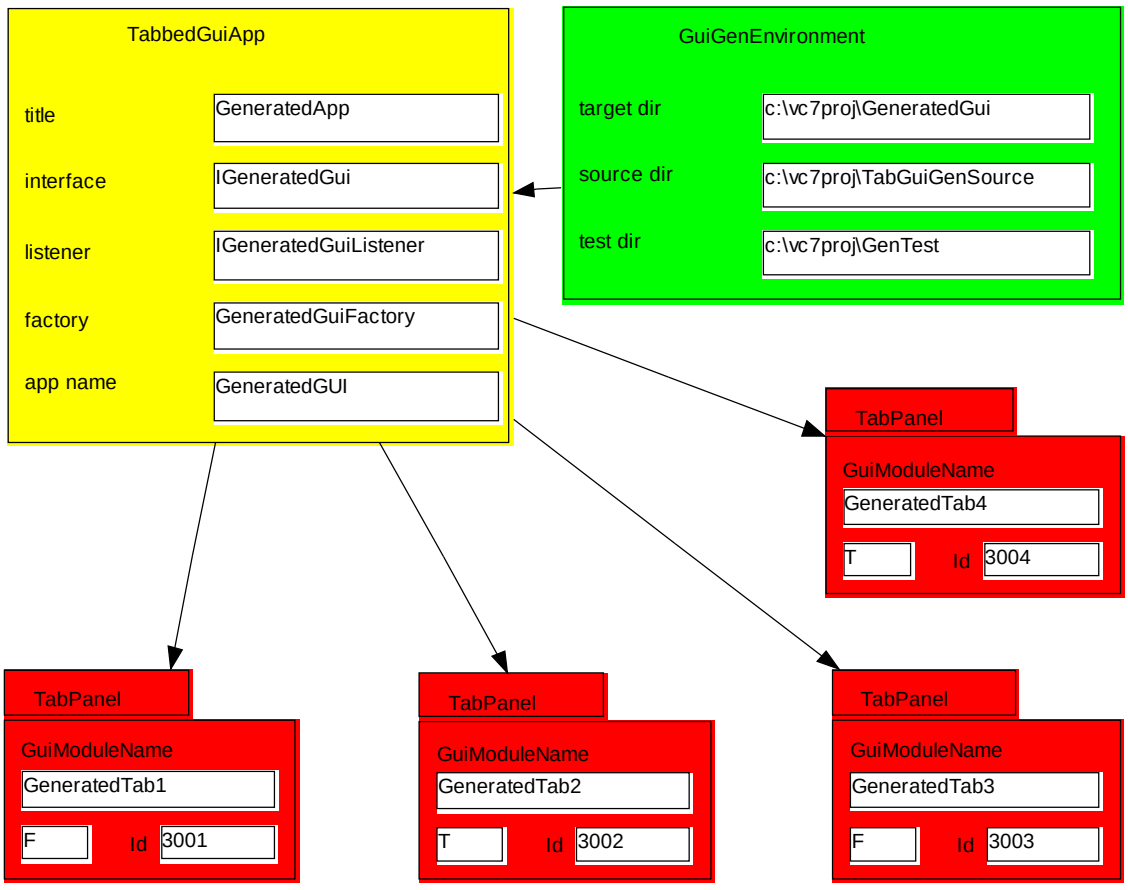

Figure 2. Simple model for reference implementation driven generation in MetaEdit+.

Consider the following example file fragment annotation:

```
/*@@fragment PrototypePanelX = $TabPanel.GuiModuleName */
      #include "PrototypePanelX.h"
/*@@endfragment */
```

In this example the default parent metaclass is the surrounding code's metaclass, hence it is not specified explicitly in the annotation. We take all instances of class "TabPanel" that are children of the parent model class and use their attribute names "GuiModuleName" to replace text "PrototypePanelX" in our fragment. Let's assume that we have a simple model as shown on Figure 2. We have four instances of class "TabPanel" being children of an instance of a metaclass "TabbedGuiApp", which happens to be an associated metaclass for the current file. As a result of this fragment processing we obtain the following code:

```
#include "GeneratedTab1.h"
#include "GeneratedTab2.h"
#include "GeneratedTab3.h"
#include "GeneratedTab4.h"
```

This processing is equivalent to the following generation input script snippet (written for the MetaEdit+ Report Generator [2]) from a DSM generation process:

```
foreach .TabPanel
{
        '       #include '"'; :GuiModuleName; '.h"'; newline;
}
```

As we can see, in the reference implementation driven approach the text '#include "PrototypePaneX.h"' needs to be written only once, in the working reference implementation. There is no need to duplicate it in the generator's input script. The annotation is all we need to perform the product generation. In the conventional approach we need to take this line into the generating script and to adjust its syntax. Such operation is easy when considering one line, but on a long run it may become a serious problem due to code duplication, longer development cycles, insufficient tool support, etc.

8. Existing Code Considerations

The migration of code from the reference implementation into the DSM generator takes significant amount of resources, especially if it needs to be performed multiple times. Elimination of the migration step has two additional advantages:

1. In environments where investment in legacy systems is significant, existing code base or its subset could be treated as a reference implementation.
2. In environments where there is a substantial production of code without the required modeling awareness and support (time pressure and/or a lack of developers with modeling skills can cause this), the modeling infrastructure could be introduced at low cost.

Both the above-mentioned types of environments are characterized by a large amount of code for which there is no modeling information or any form of modeling support. In such environments it is critical to be able to extract high abstraction level information from the

existing code. Since the reverse-engineering techniques do not increase the level of abstraction, it is usually refactoring that is being used to extract high-level information and as a code improvement technique. Refactorings introduce risk and are only feasible in the presence of a trustworthy regression test suite. Such test suites are often not available. In situations in which there is no modeling support and no regression verification mechanism in place, technique described in this paper could help in bootstrapping the modeling approach. Annotating existing code is cheap and introduces no risk. By annotating we can quickly obtain higher level modeling information for a large code base.

One possible approach is to focus on a small subset of the existing code base. Such a subset can be enriched with a test suite, refactored, and then used as a reference implementation for the rest of the system. Another approach would be to annotate a larger part of the code base and extract the metamodeling information from it. Dealing with the high level metamodel is then much easier than dealing with a low abstraction level code.

Using annotations and the approach described in this paper is limited to systems with static variability. Extracting the metamodeling information from existing code may be valuable even if the static variability information is the only information that will be extracted. DSM introduced in such a way may be limited, yet it could constitute a significant low-cost step towards modeling benefits.

9. Summary

In this paper, we are focusing on a development environment for the domain of thick-client GUI components. We have tried to adjust the typical process of Domain-Specific Modeling to our needs. In order to achieve the desired capabilities of a reference implementation driven process, such as no code duplication, tailored products, automatic reference implementation incorporation in the process, we inverted the process and made the reference implementation a central source of modeling information. As a result, we deviated from a typical DSM setup by introducing elements of Attribute-Oriented Programming [7, 9]. The solution presented in this paper offers, in our opinion, greater flexibility, as the metamodel information can be provided in a form of a reference implementation or (in a hybrid variant) both reference implementation and an explicit metamodel definition. This paper is based on experiences with metamodeling DSM solutions using the Metacase MetaEdit+ Method Workbench tool [2]. We find MetaEdit+ to be an extremely effective metamodeling environment. It was MetaEdit+ that gave us the inspiration to the approach presented here. Elegance and conceptual cleanness of MetaEdit+ allowed us to see the process freed from the unnecessary details and inspired us to further investigations towards higher efficiency of our process. We believe that the approach described in this paper has the potential of leveraging the power of Domain-Specific Modeling in problem areas which require code generation performed without the existence of thick framework layers.

Concrete concepts like reference implementation files and file fragments can be easily replaced with more abstract ones like component, part, and source [4]. The concept of a reference implementation could also be treated in a more abstract way. The idea is to provide an alternative source of metamodel information and to provide a single source of implementation. This source could be a reference implementation, but it could also be a general kind of a project or any set of artifacts.

This paper does not propose a concrete domain-specific language for thick-client GUI components. Instead, it proposes a technique to produce metamodels for such components from reference implementations. We find this approach very pragmatic as the modeling overhead is kept at minimum. Metamodels and generators can be created from running reference implementations using tools, the only extra effort is the reference implementation

annotation. Annotated reference implementation becomes a metamodel and, at the same time, the source code body of a reference implementation is being used as input for the generation process of similar products. This approach leverages the latest development of Attribute-Oriented Programming that focuses on processing source code annotations for various purposes [9].

The proposed environment has an advantage of being easy to use by technically oriented programmers. It makes it possible to focus on technical development of a working reference implementation first and then to employ an abstract modeling approach without the need of applying sophisticated modeling knowledge. Modeling-specific activities are kept at minimum and boil down to finding variability points in the reference implementation. Such variability points are designated to be attributes of easy to comprehend metaclasses. These attributes then influence and control the way new code is being generated from the existing code in a simple and understandable manner.

References

[1] Luoma, J., Kelly, S., Tolvanen, J.-P., Defining Domain-Specific Modeling Languages: Collected Experiences, Proceedings of the 4th OOPSLA Workshop on Domain-Specific Modeling (DSM'04), Vancouver, British Columbia, Canada, Oct 2004, Computer Science and Information System Reports, Technical Reports, TR-33, University of Jyväskylä, Finland, 2004

[2] MetaCase, MetaEdit+ Method Workbench Version 4.0 User's Guide, www.metacase.com

[3] Voelter, M., Bettin, J., Patterns for Model-Driven Software Development, Version 1.4, May 10, 2004, www.voelter.de

[4] pure-systems GmbH, Technical White Paper, Variant Management with pure::variants, www.pure-systems.com

[5] Tolvanen, J.-P., Making model-based code generation work, Embedded Systems Europe, Vol. 8, 60 (Aug/Sept), 2004

[6] Tolvanen, J.-P., Making model-based code generation work - Practical examples (Part 2), Embedded Systems Europe, Vol. 9, 64 (March), 2005

[7] H. Wada, J. Suzuki, S. Takada and N. Doi, "Leveraging Metamodeling and Attribute-Oriented Programming to Build a Model-driven Framework for Domain Specific Languages," In Proc. of the 8th JSSST Conference on Systems Programming and its Applications, Gunma, Japan, March 2005.

[8] Pohjonen, R., Boosting Embedded Systems Development with Domain-Specific Modeling, RTC Magazine, April 2003.

[9] Schwarz, D., Peeking Inside the Box: Attribute-Oriented Programming with Java 1.5, ON Java.com, O'Reilly Media, Inc., June 2004.

[10] Stahl, T., Voelter, M, Modellgetriebene Softwareentwicklung, Techniken, Engineering, Management, dpunkt.verlag, Heidelberg, 2005