Mechanisms for Expressing Variability in Models and MDD Tool Chains

Markus Voelter
Independent Consultant
Ziegelaecker 11
89520 Heidenheim, Germany
www.voelter.de
voelter@acm.org

Iris Groher
Siemens AG, CT SE 2
Otto-Hahn-Ring 6
81730 München,
Germany
iris.groher.ext@siemens.com

Bernd Kolb
Independent Consultant
Franz-Marc-Straße 35
89520 Heidenheim, Germany
www.kolbware.de
b.kolb@kolbware.de

ABSTRACT

Building embedded systems using model-driven development is highly desirable for a number of reasons, among them performance, penalty free abstraction, architecture enforcement, and global constraint evaluation. However, embedded systems are often built in several variants. It is thus essential to be able to express variations of systems in a coherent way, even when the system is built using model-driven development. In this paper, we describe a number of mechanisms for expressing variability in the context of MDD. This includes the expression of model variants, extending models with additional information, as well as describing variants of code generators.

1. Introduction and Motivation

Model-Driven Development (MDD) [1] for embedded systems is not fundamentally different from MDD for other systems. However, MDD is especially well suited for embedded systems for the following reasons:

- Generators can generate resource-optimized code; domain-specific abstractions in the models can be removed from the generated systems, and thus do not result in performance penalties.
- MDD supports building architecturally coherent systems because generators generate code based on rules in the templates and transformations and the meta models define application architecture in a formal way. This is important to be able to make any reasonable quality of service (QoS) assurances.
- Complex, non-local constraints and optimizations (e.g. for implementing certain QoS properties) can be considered in the generator; one can even use simulations to "optimize" dynamic system properties.
- By restricting the expressive power of the (modeling) language, the models as well as the generated code can become verifiable in the sense that certain mission-critical properties of the system can be shown to be true for all possible states of the system.
- MDD cannot just include (models of) the software, but also (models of) the hardware system, resulting in an integrated systems engineering approach.
- MDD can help to implement the variability in embedded systems resulting from the requirement that a system has to run in different (system and hardware) environments.

In this paper, we focus on the last item, namely handling variability. While handling variability in systems is a challenge for every domain, it is especially prevalent in embedded systems since many of them are part of a product line [4]. Product lines contain a portfolio of similar systems that differ in well-defined ways. Also, target environments are typically much more heterogeneous than in enterprise systems, as a consequence of optimizing for per-unit cost as opposed to development effort. It is quite typical that a given piece of functionality has to be able to run on different processor types, different operating systems, different memory layouts, different network/bus systems or with completely different UIs.

In classic (i.e. non model-driven) software development, variability is often implemented using well-known runtime mechanisms such as indirection, polymorphism, the well-known GoF design patterns [5], frameworks, or reflection. Of course, these solutions are often not feasible for embedded systems because either the language does not support them (polymorphism or reflection is not supported in C) or because they would result in too much performance overhead. While other means exist (such as various forms of pointer or preprocessor magic in C), MDD provides an additional opportunity, namely handling the variability in the generation process. This involves varying models, meta models, code generation templates, and model transformations. Since these mechanisms reside on a higher level of abstraction where system descriptions are still less detailed, describing variability on that level is simpler and more concise.

In this paper we outline a selection of mechanisms that can be used to express variability in models and code generators. The remainder of the paper is organized as follows: Section 2 introduces different mechanisms for model extension, while Section 3 looks at mechanisms for code generator variation. Section 4 provides an outlook on a methodology for model-driven software product line engineering.

2. Mechanisms for Model Extension

This section demonstrates a set of model extension mechanisms describing ways of how variants of models or meta models can be defined. Each mechanism is explained with a short description, an illustrative figure as well as with a short discussion of how the

approach is implemented in a tool chain built on Eclipse EMF [6] and openArchitectureWare [7].

2.1 Specialization

A new meta model is created, extending classes defined in some other (base) meta model (c.f. Figure 1). This is useful if you want to specialize a complete language and work with that new language in your system. A typical candidate for extension is the UML meta model.

There is a significant disadvantage to this approach: You cannot remove items from the base meta model you do not need in your language. This is an especially serious problem with complex base meta models such as UML.

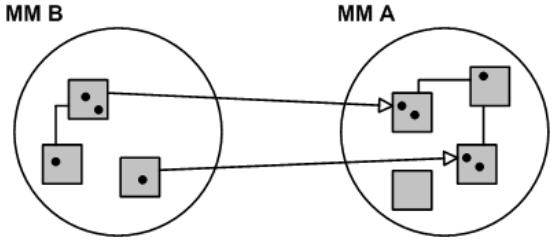

Figure 1. Specialization

EMF specifics: Ecore does not provide a means to have one meta model package "extend" another one. You can only extend single meta classes. This means that you have to define a new meta model package and reference meta classes defined in another one to have your new classes extend the original ones. Your meta classes will use the new package's name for qualification. The old meta classes (those "inherited" from the original meta model) will still be available under the name of the old package. Thus, you have to work with two meta model packages. This can be a problem in some tool environments.

2.1.1 Extension Functions

A set of functions are defined that calculate derived properties (c.f. Figure 2). Depending on the tooling, they can be accessed as if they were properties. Since the functions – and thus the derived properties – are defined outside the meta model in a separate file, the original meta model does not need to be changed. Note, however, that since the extensions are *functions*, you cannot store additional information with the model; you can only calculate derived values from information already in the model.

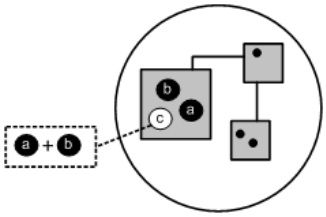

Figure 2. Extension functions

EMF specifics: EMF does not provide native support for this approach. However, using oAW's Xtend [7] facility you can achieve the desired effect. From within oAW, you can access the "derived properties", i.e. access the functions almost as if they were regular properties. You have to use *()* after the name.

2.2 Weaving

An aspect weaver is used to weave additional properties, relationships or meta classes into the base meta model (c.f. Figure 3). Depending on the weaver, you can add new properties, new relationships and also new meta classes.

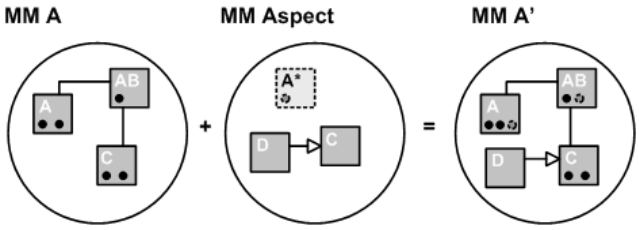

Figure 3. Weaving

EMF specifics: For EMF, there are several model weavers available, such as oAW's XWeave [7, 2] or AMMA's AMW [8]. The aspect elements are actually physically woven into the original model, physically altering its structure. The result of the weaving process is an updated model. Subsequent tooling cannot tell the difference between a woven model and a "normal" model. For more information on model weaving and XWeave see [2].

2.3 Joining

Two or more existing meta models are taken and relationships are added to join them (c.f. Figure 4). The meta models keep their own identities. Subsequent tools must be able to work with several meta models. Note that the two (or more) partial models do not need to know about the other ones. A tool creates the link between the two.

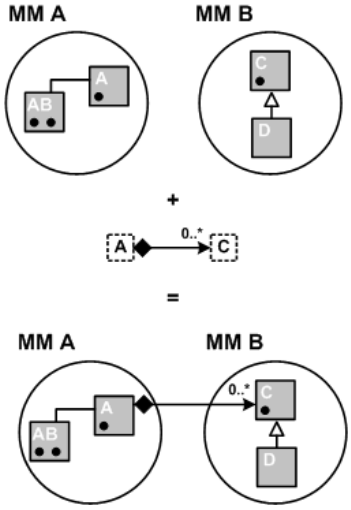

Figure 4. Joining

EMF specifics: In EMF, the join-process is invasive. Since references cannot live on their own, they are always owned by a meta class. In Figure 4 above, *MM A* would need to be changed, since the metaclass *A* will obtain a new reference.

2.4 Dynamic Properties

A set of name-value pairs is associated with a meta model element (c.f. Figure 5). This allows the storage of all kinds of additional information with model elements. The values can be primitive values or even additional model fragments.

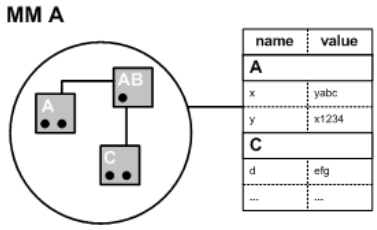

Figure 5. Dynamic Properties

EMF- specifics: EMF supports this feature in some way. Any model element can have one or several *annotations*. However, the value of an annotation is a string. oAW, to the contrary, provides a library that can store any number of name-value pairs with any model element. The value can be anything, including a model fragment.

2.5 Annotations

Annotations are "external" models that store additional information about a model element of the original model. In order to establish the relationship with the original model, the annotation meta model either contains a reference to the target meta class, or references the target by some unique (typically qualified) name.

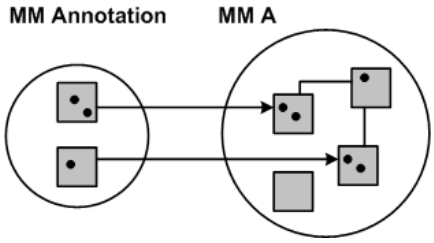

Figure 6. Annotations

EMF specifics: In EMF, a model can reference elements in another model by using inter-resource references.

3. Mechanisms for Generator Variation

After looking at the description of variants of models, let us now look at the definition of variations of generators. Note, that this is actually a feasible approach in MDD, since the generators are typically domain- or project specific. In traditional software development, the analogy would be a modification of the compiler – which is infeasible.

Code is typically generated via code generation templates (c.f. Figure 7).

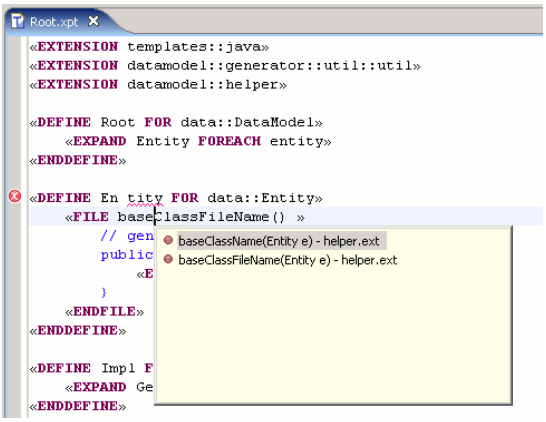

Figure 7. Code generation templates

Templates are a mixture of three kinds of code: code that should be generated into the target file, code that controls the template execution, and code that accesses the underlying model.

It is useful to adopt an OO-approach to template languages where a specific template is executed in the context of a specific model element. It is convenient to be able to reference that element by an implicit *this* reference. So, whenever you access a model property without any further qualification, it is by default resolved on the current *this* model element. As a consequence, a template definition always includes the metaclass for which it is defined (just as a method in an OO program that is always attached to a class).

So, for example, if you are in the context of a *Struct* element (something defined in your meta model), you might have a template that generates an implementation for structs. The signature for the template might be defined as

```
<<TEMPLATE cppClass FOR mymetamodel.Struct>>
```

The expression <<*name*>> in the template body resolves to the name of the particular *Struct*. If you have a separate template that can generate method signatures, then this other template might be defined as

```
<<TEMPLATE cppSignature FOR mymetamodel.Method>>
```

You can call that template from the original one, e.g. by writing

```
<<EXECUTE cppSignature FOREACH methods>>
```

where *methods* is the struct's property that returns the set of all methods for the particular struct. Every element in the foreach statement becomes the *this* object of the called template.

Based on this template mechanism, the following two variability mechanisms can be implemented.

3.1 Template Polymorphism

Once you have "object oriented templates" in place (as described in the previous section) you can also support template polymorphism. This relieves developers from writing all kinds of *type-if*s (using *instanceof*, *oclIsKindOf* and the like) in the templates. Type-ifs are bad because, just as in OO programming, you have to revisit all the type-ifs if you add a new subclass. Using template polymorphism, you can write things such as

```
...
<<EXECUTE methods FOREACH properties>>
...

<<TEMPLATE methods FOR ReadOnlyProperty>>
  ...generate only a getter method...
<<END>>

<<TEMPLATE methods FOR Property>>
  ... generate getters and setters ...
<<END>>
```

This example assumes that *ReadOnlyProperty* is a sub-metatype of *Property*. If at some point you introduce a *DerivedProperty* metaclass, you will only need to provide specialized templates (*TEMPLATE ... FOR DerivedProperty*) where the code for derived properties differs from "normal" properties.

Again, this best practice helps to keep your generator code small and maintainable, especially in the face of complex metamodels or complex target code generation requirements.

3.2 Template AOP

Assume you are generating code for embedded systems, whose code must be able to run on several different hardware architectures. The generated code is C. So, the code for the various platforms is mostly the same, but there are small (and not so small) differences scattered through the code. The classical solution is to have templates that look somewhat like the following:

```
...
all kinds of C stuff here that is common to all
the supported platforms
...

<<IF platform=="68HC11">>
  ...68HC11-specific code here
<<ELSEIF platform=="8051">>
  ...8051-specific code here
<<END>>
```

As you can see, the platform specific aspects are scattered through the code – they crosscut the template structure. So, as we all know, AOP can help to localize and modularize crosscutting concerns. Therefore, what we need is AO support in template languages. This allows us to put all the "general" stuff into the core templates, and then weave the platform-specific stuff in – all the platform-specific template code will be modularized in aspect templates.

In openArchitectureWare we support a special template definition syntax [ref]:

```
<<AROUND pointcut FOR type>>
  to-be-generated-code
  <<targetDef.proceed()>>
<<END>>
```

Note that the pointcut supports several wildcards and the type respects the polymorphism explained above.

3.3 Transformation AOP

In the same way AOP is useful for code generation templates to express variations, it is also useful for model-to-model transformations. We are not going to elaborate this further here, since it is conceptually similar to Template-AOP, and because our toolkit does not support it yet.

4. Towards a methodology for MD-SPLE

The mechanisms outlined above constitute some of the basic mechanisms of how to introduce variability into models as well as the transformer/generator. These techniques are useful for localized variability. However, for large-scale variability, in the context of Software Product Line Engineering (SPLE), these basic operations must be integrated to a consistent whole, and coordinated with the more traditional variability mechanisms. The backbone of describing the variability should be implemented using a feature model. Consequently, we need to be able to tie the mechanisms outlined above to a central feature model. This approach, as well as other aspects of model-driven product line engineering are described in more detail in [3].

5. REFERENCES

[1] Stahl, T., and Völter, M. *Model-Driven Software Development.* Wiley & Sons, 2006.

[2] Groher, I., Voelter, M., *XWeave – Models and Aspects in Concert*, Workshop on AO Modeling, AOSD 2007, and http://www.voelter.de/data/workshops/AOM2007.pdf

[3] Voelter, M., Groher, I., *Product Line Implementation using Aspect-Oriented and Model-Driven Software Development*, submitted to SPLC 2007, and http://www.voelter.de/data/pub/VoelterGroher_SPLEwithAOandMDD.pdf

[4] Pohl, K., Böckle, G., and v. d. Linden, F., *Software Product Line Engineering Foundations, Principles, and Techniques.* Berlin: Springer, 2005.

[5] Gamma E,, Helm R., Johnson R., Vlissides J., *Design Patterns*, Addison-Wesley 1994

[6] *Eclipse Modeling Framework (EMF)* website, http://eclipse.org/emf

[7] *openArchitectureWare (oAW)* website, http://www.eclipse.org/gmt/oaw/

[8] Didonet del Fabro, M., Bézivin , J. and Valduriez, P., *Weaving Models with the Eclipse AMW plugin*, In Proceedings of the Eclipse Summit Europe (Eclipse Modeling Symposium), Esslingen, Germany, October 2006.