JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2009**

Vol. 8, No. 6, September-October 2009

Best Practices for DSLs and Model-Driven Development

Markus Voelter, independent/itemis

1 INTRODUCTION

In this article I describe best practices I learned over the years using DSLs for developing software. Before we start, let me outline the context. I exclusively cover external domain specific languages (DSLs), languages that are custom-defined to describe aspects of a software system. These languages can be textual or graphical, the models created with the language can be used as input for code generation, validation, simulation or interpretation. The DSLs can be intended for use by developers and architects (covering mainly architectural/technical aspects of software systems), but also by business users who are not classically considered "developers".

I explicitly exclude internal/embedded DSLs such as the ones built with Ruby, Converge or Lisp. It also does not consider tools like MPS, where you typically build DSLs by extending a Turing-complete base language (Java, in case of MPS).

The article is a highly condensed collection of best practices. For each of them, I could have written a couple of pages (in fact, many pages have been written on these and other best practices, see [1,2,3]). However, in spite of its brevity, this article reminds you of all the things you should consider when (thinking about) starting an MD* project.

Some notes on terminology. I use MD* as a common moniker for MDD, MDSD, MDE, MDA, MIC, and all the other abbreviations for basically the same approach. Models can be processed in many ways. They can be validated, transformed, generated into code, or interpreted. I use "model processing" (and the noun, "model processor") to refer to all of these with a single term. I use the term "metaware" to mean all the artifacts on the meta level. Metaware includes DSLs, meta models, editors and of course, model processors. In many cases, the overall model that describes a system is separated into a number of "model units" which I call partitions (XML files are an example). If I use the term "business" (in the context of business user, business expert or business domain), I don't specifically mean business in the sense of financials/accounting/legal, but refer to all kinds of application domains (in German: "fachliche Domänen"); they can include scientists, mechanics, automotive or, of course, financials or insurance. The term is used to contrast the programming/software domain which deals with programmers, architects and analysts.

Markus Voelter: "Best Practices for DSLs and Model-Driven Development", in *Journal of Object Technology*, vol. 8, no. 6, May-June 2005, pp. 79-102
http://www.jot.fm/issues/issue_2009_09/column6/

Each of the best practices is rated with a number of stars. The star rating is based on a small survey I did among colleagues. As of now, 10 people have replied, so the survey is not necessarily representative, but it is an indication about the confidence into the best practice. Here is what the stars mean:

- ☆☆☆ I don't think this works, I typically use a technique that contradicts this one
- ★☆☆ I haven't used this, but it sounds reasonable and I guess that is how I'd do it if I had to something like this
- ★★☆ I have used this successfully, but I am not sure it is a general best practice
- ★★★ I have used this successfully a number of times, and I am sure it is a best practice. Can't imagine not to use it.

The paper has three main sections. The first one, Designing DSLs, looks at best practices to keep in mind as you design your languages. Section two, Processing Models, looks at model checking interpretation and code generation. Section three considers a couple of things you need to keep in mind about process and organization. A final section looks at open issues and challenges in MD* world.

2 DESIGNING DSLS

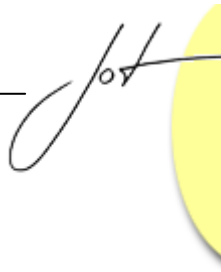
Sources for the language ★★½

How do you find out what your DSL should express? What are the relevant abstractions and notations? This is a non-trivial issue, in fact, it is the key issue in MD*. It requires a lot of experience, thought and iteration. However, there are several typical ways of how to get started.

If you're building a technical DSL, the source for a language is often an existing framework, library, architecture or architectural pattern. The knowledge often already exists, and building the DSL is mainly about formalizing the knowledge: defining a notation, putting it into a formal language, and building generators to generate parts of the (potentially complex) implementation code. In the process, you often also want to put in place reasonable defaults for some of the framework features, thereby increasing the level of abstraction and making framework use easier.

In case of business domain DSLs, you can often mine the existing (tacit) knowledge of domain experts. In domains like insurance, science or logistics, domain experts are absolutely capable of precisely expressing domain knowledge. They do it all the time, often using Excel or Word. They often have a "language" to express domain concerns, although it is usually not formal, and there's no tool support. In this context, your job is to provide formality and tooling. Similar to domain knowledge, other domain artifacts can also be exploited: for example, hardware structures or device features are good candidates for abstractions in the respective domains.

In both previous cases, it is pretty clear how the DSL is going to look like; discussions are about details, notation, how to formalize things, viewpoints, partitioning and the like (note that those things can be pretty non-trivial, too!).



However, in the remaining third case, however, we are not so lucky. If no domain knowledge is easily available, we have to do an actual domain analysis, digging our way through requirements, stakeholder "war stories" and existing applications.

For your first DSL, try to catch case one or two. Ideally, start with case one, since the people who build the DSLs and supporting tools are often the same ones as the domain experts – software architects and developers.

Limit Expressiveness ★★½

When building a DSL, make sure you're not lured into building yet another Turing-complete, general purpose language. In many cases, a purely declarative language that "states facts" about a system is good enough.

Note the difference between configuration and customization. A customization DSL provides a vocabulary which you can creatively combine into sentences of potentially arbitrary complexity. A configuration DSL consists of a well-defined set of parameters for which users can specify values (think: feature models). Configuration languages is are more limited, of course, since you cannot easily express instantiation and the relationship between things. However, they are also typically less complex. Hence, the more you can lean towards the configuration side, the easier it usually is to build model processors. It is also simpler from the user's perspective, since the apparent complexity is limited.

Be aware of the difference between precision and algorithmic completeness. Many domain experts are able to formally and precisely specify facts about their domain (the "what" of a domain) while they are not able to define (Turing-complete) algorithms to implement the system (the "how"). It is your job as a developer to provide a formal language for domain users to express facts, and then to implement generators and interpreters to map those facts into executable algorithms that are true to the knowledge they expressed. The DSL expresses the "what", the model processor adds the "how".

If there's a concern in your system for which 3GL code is the right abstraction (i.e. you need the full expressive power of a Turing-complete language with no significant semantic extensions), it is not necessarily a good idea to try and define a DSL for the concern. It is often perfectly ok to define (generate) a nice API against which developers can then write code in a 3GL. You can also generate hooks into the generated code which users can implement with 3GL code to realize some exceptional behavior. Keep the purpose of the hooks well defined, and their number limited, though!

Notation, Notation, Notation ★★½

When building DSLs, notation is extremely important. As the language designer, you care mostly about the underlying meta model, and you might not really care about the "nice syntax". But from the (domain) user's perspective, the situation is exactly opposite!

Especially (but not exclusively) in business domains, you will only be successful if and when you can tailor your notations to fit the domain – there might even be

existing notations. It is often hopeless to try and convince domain users or experts about a "better notation" – just implement what they have.

Note that this might require textual and graphical notations, Excel-like spreadsheets, form-based systems, or all of them mixed. Today's DSL tools have limitations in this respect. I am sure the next couple of years of evolution in DSL tooling will address mainly this issue. As of now, just be aware of the wide variability of notations, and try to do as best as you can given the tooling that's available.

The notation should make the expression of common concerns simple and concise and provide sensible defaults. It is ok for less common concerns to require a bit more verbosity in the notation.

When prototyping or sketching a DSL, it is often useful to start with the notation, cross-checking it with the language users.

Graphical vs. Textual Notation ★★½

Things that are described graphically are easier to comprehend than textual descriptions, right? Not really. What is most important regarding comprehensibility is the alignment of the concepts that need to be conveyed with the abstractions in the language. A well-designed textual notation can go a long way. Of course, for certain kinds of information, a graphical notation is better: relationships between entities, the timing/sequence of events or some kind of signal/data flow. On the contrary, rendering expressions graphically is a dead end (note how a graphical formula editor is somewhat of a hybrid with the way it displays fractions, matrices, integrals and the like.)

When deciding about a suitable notation, you might want to consider the following two forces: in most (but not all!) tool environments, editors for textual notations (incl. code completion, syntax highlighting and the like) are much easier to build and evolve than really user-friendly and scalable graphical editors. Textual models also integrate more easily with existing source code management and build infrastructures.

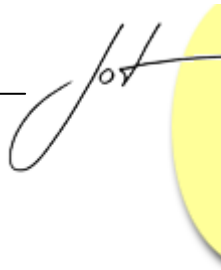
Also, instead of using full-blown graphical editing, you might want to consider textual editing plus graphical visualization (see below)

In environments where *usable* graphical editors are a lot work to build, I recommend first stabilizing the concepts and abstractions of the language with very simple editors (textual, tree, generic box/line) and then investing into a polished graphical editor.

Finally, in many systems some viewpoints will be graphical, others textual. Sometimes you will even want to mix the two forms of syntax: consider a state machine (graphical) with embedded guard expressions (textual). This can be tricky with today's tooling.

DSL Semantics (unrated)

It is not enough to define the abstractions and the notations for a DSL, you also have to define the meaning of those abstractions – the language semantics.



In some sense, the semantics of a language takes into account more knowledge about the domain than what is expressed in the language: the language only allows users to express things that are particular to the specific system/application/instance they describe with the model. The semantics, however, also takes into account the knowledge about all the stuff in the domain that is identical for every system/application/instance in that domain.

Technically it is the job of the generator, interpreter and platform to bridge this gap. However, from the perspective of the language user (who might not know specifically what a model processor does) the semantics are tacit knowledge about "how the language works" and it has to be explained as "the meaning of the language".

There are various ways of defining semantics formally, none of them being sufficiently pragmatic (as of 2008) to be useful in mainstream DSL practice. Consequently, the meaning of a language is defined in two ways: it is explained in prose and with examples towards the language users and it is tied down towards the execution platform using the code generator (which is, strictly speaking, a form of operational semantics definition, since the generator maps the language concepts to the concepts of a target language whose semantics are known) or the interpreter.

Viewpoints ★★½

A software system usually cannot be described with one notation for all relevant aspects. Also, the development process requires different aspects to be described by different roles at different times, as you want to be sure to have a clean separation of concerns. Hence it is important to identify the set of viewpoints relevant for describing the different concerns of a system, and provide notations and abstractions for each.

In some system that means that you'll define separate DSLs for each viewpoint. In other systems you'll define one language that has a number of sections, one for each viewpoint.

Whichever approach your tooling supports, viewpoints need to be connected to other viewpoints to be able to describe the overall system. Make sure those "connection points" are explicitly defined and limited in number. Also, make sure the direction of dependency is clear between the viewpoints – strict layering with unidirectional dependencies is highly recommended.

Note how this is similar to the modularization of software systems, the same rules apply: strong coherence internally, few interfaces externally and generally as little coupling as possible.

Partitioning ★★½

Like everything in software, DSLs editors and model processors don't scale arbitrarily – something you tend to forget when starting a project from a small prototype. In most scenarios, it is important to partition the overall model into separate "model units".

Partitions have consequences in many respects. They are often the unit for checkin/checkout or locking. Also, references within a partition are often direct

references, whereas cross-partition references might be implemented via proxies, (tool-enforced) name-references or generally, lazy-loading. Partition-local constraints are often checked in real-time in the editor, global constraints might only be checked upon request, maybe as part of an overall "build" process.

Also, it often makes sense to ensure that each partition is processable separately. Alternatively, it is possible to explicitly specify the set of partitions that should be processed in a given processor run (or at least a search path, a set of directories, to find the partitions, like an include path in C compilers). You might even consider a separate build step to combine the results created from the separate processing steps of the various partitions (like a C compiler: it compiles every file separately into an object file, and then the linker handles overall symbol/reference resolution and binding).

In many tools, partitioning is not completely transparent. You might have to include partitions explicitly and/or you have to make sure you don't accidentally create unintended dependencies on other partitions. Hence, it is important to consider partitioning early in the DSL/generator development process and design your metaware accordingly.

The design of a workable partitioning strategy is part of language design! Things to keep in mind in this context are: which partition changes as a consequence of specific changes of the model (changing an element name might require changes to all by-name references to that element in other partitions), where are links stored (are they always stored in the model that logically "points to" another one)?, and if not, how/where/when to control reference/link storage.

Partitions are really about physically partitioning the overall model. They can be aligned with the logical model structure (think namespace) or viewpoints, but they don't have to. For example, a partition that describes the Billing subsystem, might contain elements in several (nested) namespaces and cover several viewpoints (data structure, process, UI definition).

Evolution ★★☆

Another important aspect that is often forgotten when initiating a MD* project is the need for language evolution. If you change the language, make sure that you also have a way of adapting model processors as well as existing models.

Doing this requires any or all of the following: a strict configuration management discipline, versioning information in the models to trigger compatible model processors, keeping track of the changes as a sequence of change operations, or model migration tools to transform models based on the old language into the new language.

Whether model migration is a challenge or not depends quite a bit on the tooling. There are tools that make model evolution a very smooth, but many environments don't. Consider this when deciding about the tooling you want to use! Note that in case of textual DSLs, model migration can be achieved via regular expressions and *grep* (at least as a fallback).

It is always a good idea to minimize DSL changes that break existing models. Backward-compatibility and deprecation are techniques well worth keeping in mind in MD*-land. Note that you might be able to instrument your model processor to

collect statistics on how deprecated language features continue to be used. Once no more instances show up in models, you can safely remove the deprecated language feature.

Using a set of well-isolated viewpoint-specific DSLs prevents rippling effects on the overall model in case something changes in one DSL.

The fallacy of generic languages ★★½

Predefined, generic languages and generators are tempting – especially if you want to describe technical aspects of your system. After all, you can model everything with UML, can't you? Just add a bunch of stereotypes and tagged values…

Be careful. Using predefined languages makes you spend most of your time thinking about how your domain concepts can be shoehorned into the existing language. Also, you're being sidetracked by abstractions and notations from the existing language. Of course, some generic languages provide facilities for adaptation, like UML's profiles. Still, at least in practical tool reality, UML shines through all the time. You'll have to add a lot of constraints that prevent users from using UML features that don't make sense in your domain. Also, your language will often look like UML, since the practical reality of customizing UML tools is far from sufficient (remember: Notation, Notation, Notation!). Finally, your model processor will have to deal with the complex and big meta model of UML – profiles always add, they never remove anything.

In practice, in most cases it is much better to define your own DSL. Initially, it seems like a bit more work, but rather soon it becomes much more efficient

However, make sure you don't reinvent the exact same wheels for which standard already exists. For example, there's not much need to reinvent state charts (for state-based behavior) and sequence diagrams (to describe scenarios or text cases) – UML does a pretty good job with these. Also, for small, incremental deviations from a useful UML notation, profiles are a good choice.

So, if a suitable generic language exists, either use the existing language, or make sure your own implementation is compatible as far as possible (duck modeling: if it looks like a state machine and it behaves like a state machine, it is a state machine[1])

Learn from 3GLs ★★½

Above we discussed the fact that a DSL is not a general purpose language in disguise. However, there is still a lot we can learn from existing formalisms and languages.

Here are four examples. Most languages need some notion of scoping: for a given reference on a model element, only a subset of the type-compatible model elements constitute valid targets for the reference.

Specialization is a concept that can be applied not just to classes in OO, but also to state machines, or specifications for insurance contracts.

Also, the notion of namespaces is found in many DSLs to organize the naming scheme for model elements.

[1] Thanks to Achim Demelt for this ☺

Finally, many DSLs contain the notion of instantiation – being able to express that some concept is an instance of another concept, effectively introducing in-language type systems. As part of your constraint checks, you might have to do actual type computations and type checks.

To become a good DSL designer, it is useful to have broad knowledge about existing programming language paradigms. Please read the book Concepts, Techniques and Models of Computer Programming by Peter Van Roy and Seif Haridi.

Who are the first class citizens? ★★½☆

There are two different styles of language design: one advocates big languages with first class support for many different domain concepts. The other advocates minimal languages with few but powerful primitive features, from which bigger features are constructed by combination and made available via libraries (this is somewhat similar to the Microkernel pattern).

Here are some things to keep in mind when building DSLs. Make sure your language design is consistent in the sense that you stick to one of the two approaches throughout. Using the second approach is more complicated and requires considerable effort in finding what those basic primitive features are. Especially in business domain DSLs, the second approach often fails because business users are not used to working with few, powerful, orthogonal concepts.

In DSLs that address domains with well identifiable or well known concepts, make sure you make those concepts the first class citizens, and use appropriate notations. For example, in a DSL for programming mobile phones, make sure make sure you have native language elements all the input elements (left button, right button, 0..9 keys, joystick). Don't try to abstract this into generic "input devices".

You can combine the two approaches, however, make sure your languages retain a feeling of consistency and integrity.

Libraries ★★½

A topic related to the previous best practice is the use of libraries. Libraries are collections of instances of your DSL, intended for reuse, typically stored in a separate model partition.

Libraries help reusing model data – this is obvious. For example, in a DSL that is used to describe data structures, it is often useful to put reusable data structures (date, time, address) into a library for others to use (libraries are a form of partitioning).

However, libraries can also be used as a way to limit language complexity. Consider the above mentioned data structure DSL: instead of hard coding the primitive types *int*, *string* and *bool*, you can just implement a *primitive type* construct and make *int*, *string* and *bool* instances of that type. This allows users to add new primitive types by changing the model as opposed to changing the language – this is much less hassle!

However, if you use the library approach, make sure the model processors don't make assumptions about the structure of some of the higher-level constructs, but instead are really only based on the basic primitive features. In case of our example,

the mapping of the primitive types to the target language (e.g. Java) may need to be part of the model, otherwise you'd have to change the generator when adding a new primitive type by changing the library.

Teamwork Support ★★☆

An important aspect of your DSL tooling is support for versioning, tagging, branching, locking, comparing and merging – all aspects of working collaboratively on models. Make sure the tools you use support all of these – using the languages' concrete syntax, nobody is willing to handle these issues on an abstract syntax/meta model/tree level!

When working with business experts, repository-based systems are often very capable of addressing these issues. However, when targeting developers, the models (and the meta ware) have to interoperate with the rest of the development tools. Specifically, you need to integrate with existing source code control systems (CVS, SVN, Git and the like). Moreover, if your system is specified via models as well as manually written 3GL code, it must be possible to tag, compare and version both kinds of artifacts together to prevent running into CM hell. A tool specific repository can be a problem in such a scenario if it does not provide means to integrate with the repository for code artifacts.

Textual DSLs have a clear advantage here, since, regarding the concerns we discussed here, the models are just text (at least if they are stored as actual text files, and the textual notation is not a projection of underlying structured data).

For business users, pessimistic locking (and consequently no need for comparing and merging) might be easier to understand. In general, the decision between a pessimistic and optimistic approach should be based on the process and the collaboration use cases.

Note that good partitioning can make teamwork support much easier; the partition becomes the unit for comparison, merging or locking.

Tooling Matters! ★★★

Defining languages and notations is not enough per se – you have to provide good tool support for them, too.

DSL editors need to be able to support teamwork (see above), navigation, overviews, searching, quick-find, find-references, show usage, maybe even refactoring. For textual DSLs, your editors have to provide code completion, syntax highlighting and the like to make sure developers (who are used to powerful IDEs for their "regular" language) are willing to work with DSLs.

The same is true for the "meta developers". Make sure your environment provides a good experience for writing transformations and code generators, for example, by providing meta model-aware editors for these artifacts.

To increase usability, DSL editors need to be able to cope with wrong or incomplete models as they are entered by the users. Ideally, it should even be possible to persist them. Of course, as long as models are wrong or incomplete they cannot be processed any further. In the context of textual languages, this might mean that you

design a somewhat "looser", more tolerant grammar, and enforce correctness via constraints.

You also have to make sure the model processors are able to run as part of the nightly build (outside of the editor or tool) to integrate them into existing build environments.

3 PROCESSING MODELS

Interpretation vs. Code Generation (unrated)

When thinking about executing models, most people inherently tend towards code generation. However, interpretation is also a valid option. An interpreter is a (meta-)program that reads the model and executes code (calculations, communication, UI rendering) as it queries or traverses the model.

There's a whole bunch of tradeoffs between interpretation and code generation. Let's first look at the advantages of code generation.

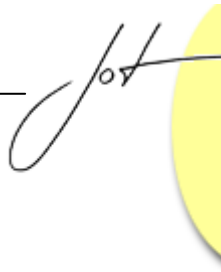
Code generation is perceived to be simpler, because the resulting code can be inspected. The templates can even be "extracted" from manually coded example applications. Generated code is also easier to debug than an interpreter (you need to use conditional breakpoints all the time). Generated code can be tailored more closely to the task at hand, and can hence be smaller and/or more efficient than an interpreter. This is especially relevant for resource-constrained environments. Finally, a code generator can work with any target platform/language, there are no changes to the target platform required (if you want to interpret, you need to run an interpreter on the target platform). More generally, using code generation, the overall MD* approach leaves no traces whatsoever in the resulting system.

Interpretation also has a number of advantages: changes in the model don't require an explicit regeneration/rebuild/retest/redeploy step, significantly shortening the turnaround time, and in some scenarios, the overall change management process. It is even possible for models to be changed from within the running application, and take effect immediately. Also, since no artifacts are generated, the build times can be much reduced. Depending on the specific case, an interpreter and the model can be smaller than generating the code.

As can be learned from programming languages, there are also potential combinations between interpretation. You can generate a lower-level representation of the model (often XML) that is subsequently interpreted (analogy: Java or CLR byte code), maybe by a previously existing interpreter. It is also conceivable to transparently generate code from within the interpreter to optimize performance (think just in time compilation, Hotspot VM). However, I have never seen this approach used in practice in the context of MD*.

Rich Domain-Specific Platform (unrated/★★½)

Code generation is a powerful tool, and a necessary ingredient to the success of model-driven development and external DSLs. However, make sure you don't generate unnecessary code.



It is always a good idea to work with a manually implemented, rich domain specific platform. It typically consists of middleware, frameworks, drivers, libraries and utilities that are taken advantage of by the generated code.

In the extreme case, the generator just generates code to populate/configure the frameworks (which might already exist, or which you have to grow together with the generator) or provides statically typed facades around otherwise dynamic data structures. Don't go too far towards this end, however: in cases where you need to consider resource or timing constraints, or when the target platform is predetermined and perhaps limited, code generation does open up a new set of options and it is often a very good option (after all, it's basically the same as compilation, and that's a proven and important technique).

Checks first and separate ★★★

In all but the most trivial cases, the structures defined by the meta model do not express the whole truth about models. Constraints – basically Boolean expressions with error messages attached – are required to validate models. It is essential that those constraints are treated as first class citizens and have their own phase during model processing.

For example, putting the constraint checks into the generator templates is bad, since it makes templates overly complicated. Also, if you have several different sets of templates (e.g. for different target languages) you'd have to put the constraints into each of them. There's usually no point in even starting up a code generator if the constraint checks don't succeed. If the model is wrong, the generated code will be wrong.

Keep in mind that it is often useful to check different constraints on different parts of the overall model at different times in the model processing chain. One example is checking certain constraints after a transformation. As another example you typically want to execute partition-local constraints interactively (e.g. when saving the partition in the editor) while global constraints should maybe be executed only on demand, because they typically take much longer to evaluate.

Check constraints as early in the processing chain as possible. The more domain-specific the model and the constraints are, the more understandable a failed constraint will be to the user. Check as many constraints as you possibly can, try to make sure that if the model validates, the resulting system is correct (this is not always possible, see runtime errors in 3GL languages, but you should strive to be as good as possible).

If you use incremental model refinement with model transformations (see Cascading below), check constraints at every level, but make sure constraints of a lower level never fail for any correct input from a higher level model – the user will not understand it.

Finally make sure you can express constraints of different severity, such as warnings and errors. Errors will typically stop the next step in the model processing chain, warnings typically won't.

Don't modify generated code ★★½

In many systems, some parts will still be expressed using code of the target language. Consequently, you have to integrate generated code and manually written code. Most tools provide what's called protected regions, marked up sections of the generated files into which you can insert manually written code that will not be overwritten when the files are regenerated.

It is often a bad idea to use them. You'll run into all kinds of problems: generated code is not a throw-away product anymore, you have to check it in, and you'll run into all kinds of funny situations with your CM system. Also, often you will accumulate a "sediment" of code that has been generated from model elements that are no longer in the model (if you don't use protected regions, you can delete the whole generated source directory from time to time, cleaning up the sediment).

Instead, add extension points into the generated code, using the composition features provided by your target language. You can e.g. generate a base class with abstract methods (requiring the user to implement them in a manually written subclass) or with empty callback methods which the user can use to customize in a subclass (for example, in user interfaces, you can return a position object for a widget, the default method returns *null*, default to the generic layout algorithm). You can delegate, implement interfaces, use *#include*, use reflection tricks, AOP or take a look at the well-known design patterns for inspiration. Some languages provide partial classes, where a class definition can be split over a generated file and a manually written file.

In the rare case where the target format does not support modularization and composition you can put the manual code literally into the model (or an external file) and have the generator paste it into the generated artifact, avoiding the need to modify it.

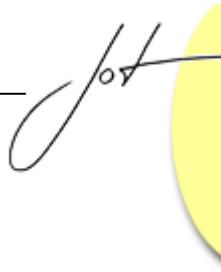
Separating generated and manually written code also has its drawbacks. For example, if you change the model and hence get different generated code, the manually written code is not automatically refactored accordingly (could be done in theory, but I haven't see it in practice). Also, the approach can result in an increased number of implementation artifacts (a generated base class and a manually written subclass), possibly increasing compilation time.

As tools become better, additional approaches might become feasible. However, as of now, the approach advocated here results in the lowest amount of headache.

Note that a similar problem can arise if you modify models resulting from a model-to-model transformation; which is why we don't recommend doing this.

Control manually written code ★½☆

Based on the previous best practices, the following can easily happen: the generator generates an abstract class from some model element. The developer is expected to subclass the generated class and implement a couple of abstract methods. The manually written subclass needs to conform to a specific naming convention. The



generator, however, just generates the base class and stops. How do you remind developers to create a subclass?

Of course, if the constructor of the concrete subclass is called from another location of the generated code, and/or if the abstract methods are invoked, you'll get compiler errors. By their nature, they are on the abstraction level of the implementation code, however. It is not always obvious what the developer has to do in terms of the model or domain.

To solve this issue, make sure there is there a way to make those conventions and idioms interactive. One way to do this is to generate checks/constraints *against the code base* and have them evaluated by the IDE. If one fails, an error message is reported to the developer. As soon as the developer implements the manual code in the required way, the error message goes away.

Another way to achieve this goal in some circumstances is to generate code that is never executed, but coerces the IDE into providing a quick fix that creates the missing artifact. For example, if you expect users to manually write a subclass of a generated class, generate a statement such as *if (false) { GeneratedBaseClass x = new ManualSubclass() }*.

Care about generated code ★★½

As we saw above, generated code is a throw-away artifact, a bit like object files in a C compiler. Well, not quite! When integrating with generated code, you will have to read the generated code, understand it (to some extent), and you will also have to debug it at some point.

Hence, make sure generated code is documented, uses good names for identifiers, and is indented correctly. All of this is relatively easy to achieve, as you have all the information you need when writing the code generator!

Making generated code adhere to the same standards as manually written code also helps to diffuse some of the skepticism against code generation that is still widespread in some organizations.

Note that in very mature environments where you generate 100% of the implementation code, the generated code is never seen by a meta ware user. In this case (and only in this case) the statements made here don't apply.

Make the code true to the model ★½☆

In many cases, you will implement constraints that validate the model in order to ensure some property of the resulting system. For example, you might check dependencies between components in an architecture model to ensure components can be exchanged in the actual system.

Of course this only works if the manually written code does not introduce dependencies that are not present in the model. In that case the "green light" from the constraint check does not help much.

To ensure that promises made by the models are kept by the code, use the following two approaches. First, generate code that does not allow violation of model promises. For example, don't expose a factory that allows components to look up and

use any other component (creating dependencies), but rather use dependency injection to supply objects for the valid dependencies expressed in the model. Second, use architecture analysis tools (dependency checkers) to validate manually written code. You can easily generate the check rules for those architecture analysis tools from the models.

Viewpoint-aware processing ★★☆

Viewpoints, as introduced above, are not just relevant for modeling. They are also important when processing models. You might want to check constraints separately for different viewpoint models. Some viewpoints might be better suited for interpretation instead of code generation. When generating code, you might want to consider generating in phases, based on the viewpoints.

For example, you should have separate code generators for the type viewpoint (once generated, developers can write manual code against the generated APIs) and the deployment viewpoint (from which you generate code that maps the API/manual code onto an execution platform), and finally interpret the state machine models within generated code by delegating to an existing state machine interpreter framework. Note that if you fail to have separate generators per viewpoint, you introduce viewpoint dependencies "through the back door", effectively creating a monolith again.

Note that there's also a separation vs. model partitions, each partition should be processable separately. If partitions and viewpoints align, this makes things especially easy.

Overall Configuration Viewpoint (unrated)

If you use viewpoints and partitions extensively, you will possibly end up with a large set of models – separate resources, that all contain parts of the overall system. For reasons of scalability and/or process, you often don't want to generate code for the whole system and/or for all viewpoints. Also, many systems can generate code for a number of target languages or environments.

In short, when running the model processor, there are often quite a number of options to specify: validate the whole model, but only with this subset of constraints; generate all the code needed for implementing the business logic for only this subsystem; or generate the deployment code for the whole system, targeting the production environment.

It is a good idea to have a separate model that captures this configuration. In some sense, it ties together the "scope of concern" for the model processor. By handling this "compiler configuration" as a model, too, you get all the benefits of modeling also for this concern, making it much more tractable than putting all of that into properties files or XML configuration files.

Care about templates ★★½

Code generation templates will be one of your central assets. They contain the knowledge of how to map domain concepts expressed in DSLs to implementation code.

Over time, they have a tendency to grow and become non-trivial. To keep the complexity in check, make sure you use well-known modularization techniques: break templates down into smaller templates that call each other, extract complex expressions into functions called by the templates and use AO to factor out cross-cutting template behavior.

Sometimes I notice that people forget about these proven techniques as soon as they go to the meta level ☺. Even worse, some of the tool builders seem to have forgotten those techniques when they built the generator tools! Make sure, when choosing a generator tool, it allows you to use those techniques for code generation templates.

Here's a specific tip: indent your templates in a way that makes sense for the template, not for the generated code. You can always run a beautifier over the generated files (at least as long as you're generating code for a language whose block structure is not based on indentation!)

Finally, by generating code against meaningful frameworks, the overall amount of template code required is reduced, improving maintainability of templates simply by having fewer of them.

M2M transformations to simplify generators ★★½

As mentioned above, generators tend to become complicated. Another way of simplifying them is to use intermediate model-to-model transformations. Two examples:

Consider the case of a state machine where you want to be able to add an "emergency stop" feature, i.e. a new transition from each existing state to a new STOP state. Don't handle this in the generator templates. Rather, write a model transformation script that preprocesses the state machine model and adds all the new transitions and the new STOP state. Once done, you can run the existing generator unchanged. You have effectively modularized the emergency stop concern into the transformation.

Second example: consider a DSL that describes hierarchical component architectures (where components are assembled from interconnected instances of other components). Most component runtime platforms don't support such hierarchical components, so you need to "flatten" the structure for execution. Instead of trying to do this in the code generator, you should consider an M2M step to do it, and then write a simpler generator that works with a flattened, non-hierarchical model.

Generally, in the case where some language features are built on top of others (see *Who are the first class citizens* above) you can reduce the higher-level constructs

to their constituent lower-level constructs, and then only provide code generators for those.

Note that for the kind of model transformations discussed here, unidirectional transformations (and hence, simpler, unidirectional transformation languages) are perfectly good enough. Bidirectional transformations are only useful in rare cases not covered in this paper.

M2M transformations for simulation (unrated)

Another important use case for model-to-model transformations is the integration of domain DSLs with existing general-purpose formalisms for which suitable validation or proofing tools exist.

For example, by transforming a specific behavior description to a state machine, you can use existing tools to automatically generate test sequences for the respective behavior. As another example consider the description of behavior for a concurrent, distributed system. By transforming it into petri nets and using suitable tools, you can make statements about reachability and liveliness of you behavior. As a third example, simulation environments are often used to verify timing or resource consumption for a specific system.

To be able to extrapolate system characteristics proven/simulated for the version of the system in the generic formalism to your original system description, you have to make sure that the simulated system is semantically equivalent to the final system being executed. So, theoretically, you have to prove that the transformations model/simulation and model/code are correct. This is very hard to actually prove, but by using a sufficient number of tests, you can show the correctness well enough for most practical purposes.

Allow for adaptations ★★☆

MD* benefits from the economies of scale. If you can write a DSL/generator once and then (re-)use it on many projects or systems, you will win big. However, as we all know, reuse is hard, because every project/system has some specifics that are not covered by the reuse candidate.

Hence, make sure you provide means for implementing *unexpected* variability in a non-invasive way.

For example, developers should be able to annotate model elements with additional information that can be used in tailored generators (e.g. store name/value pairs in a hash map for each element). Also, make sure code generation templates can be customized non-invasively to support generation of slightly-different code. This can be achieved, for example, using generator AO (the ability to contribute advice into existing generator templates) or a combination of factories and polymorphism.

Note that allowing for adaptations in all locations results in all template code being API code in the sense that developers might rely on the structure for their adaptations. As a tradeoff, you might want to mark up certain templates as “public API” or “private – don’t rely on it”.

Cascading ★★☆

Many publications advocate the idea of starting the MD* approach by defining a PIM and then transforming it into less abstract, more platform-specific PSMs, and finally to code. In my experience, it is better to start from the bottom: first define a DSL that resembles your system's software architecture (used to describe applications), and build a generator that automates the grunt work with the implementation technologies. The abstractions used in the DSL are architectural concepts of your target architecture.

In subsequent steps, build on top of that stable basis abstractions that are more business-domain specific. You can then use M2M transformations to map some aspects of those more abstract concepts to existing abstractions of your architectural language, "feeding" them into the existing generator chain. For those aspects that cannot be mapped to lower level architectural abstractions provide specific generators that generate code that acts as "business logic implementation" from the architectural generator's viewpoint (replacing some of the code that had to be manually written before).

Note that you should never ever modify the intermediate stage models. They are transitive and are typically not even stored (unless for debugging purposes). They serve as a "data extension format" between the various stages of your cascaded meta ware. If you need to put additional information into the result model, use an annotation model.

Annotation Models ★½☆

When working with model-to-model transformations you can run into some of the same problems as with code generation in that sometimes, it seems necessary to mark up the result of a transformation step manually before it is further processed. Actually changing the model after it has been created via a transformation would be an approach similar to protected regions – with similar challenges.

The better solution is to create a separate model – an annotation model – that references the elements in the intermediate model for which it specifies additional model data (effectively an additional viewpoint). The downstream processor would process the model created via the upstream model-to-model transformation and the annotation model in conjunction, understanding the semantics of the annotation model's relationship to the original model.

For example, if you create a relational data model from an object oriented data model, you might automatically derive database table names from the name of the class in the OO model. If you need to "change" some of those names, use an annotation model that specifies an alternate name. The downstream processor knows that the name in the annotation model overrides the name in the original model.

An alternative, but very related approach is to use the annotation model directly during the model-to-model transformation. In case of our example, the annotation would annotate and reference the OO model, and the transformer would take the table name specified in the annotation model into account.

Classify Behavior ★★☆

There's a tendency to use action semantic languages (ASLs) to describe system behavior on model level. However, the abstraction level of ASLs is not fundamentally different from a 3GL. Implementing functionality against a clean API is almost as good although it is, of course, implementation language specific and leads to the problem of integrating generated and manually written code. Action languages stay on the model level and hence alleviate this problem. They can also be integrated more easily into model refactoring and global constraint checking.

To become more efficient with implementing behavior, classify behavior into different kinds such as state-based or business-rule based, and provide specific DSLs for those classes of behavior. In many cases you can even generate the behavior based on a very limited set of configuration parameters.

Also, business domain specific DSLs should be used for suitable classes of behavior; as an example consider a temporal expression language for insurance contract specification.

In some sense, manually written code is just a suitable implementation language for some kind of behaviors, for which there's no more efficient way to express it.

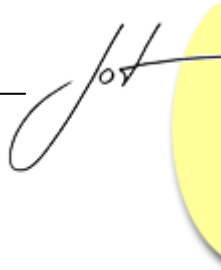
Don't forget testing ★★½

Just like in any aspect of software, testing is an important ingredient. In MD*, testing comes in different flavors, though. Here are some thoughts.

First of all, constraint checks are a form of test, a bit similar to compiler checks in classical programming, albeit easily customizable and domain specific. When testing a code generator, don't test for the syntax of the generated code, rather compile the code and write unit tests against it. This tests the generated code's semantics as opposed to its syntactic structure. You can also test model transformations by writing constraint checks against the concrete data in the models that result from a transformation. When building a generator, always keep a test model around that uses all features of the language, and write tests against this model (be aware of the coverage issue!). Building and maintaining this model and the corresponding tests is the job of those developers who build the generator, not of the generator users!

Assuming the generator is well tested and mature (see previous paragraph), then there's no need to write tests that verify the generated code in projects that use the generator. However, it is usually still useful to write tests against the overall system built via MD* - to make sure the model semantics is as expected, and to make sure them manually written code sections behave correctly.

When generating the system as well as the test, make sure you don't derive both from the same model. This might lead to a situation where a faulty test is run against a faulty system resulting in a succeeding test!



4 PROCESS AND ORGANIZATION

Iterate! ★★½

Some people use MD* as an excuse to do waterfall again. They spend months and months developing languages, tools, and frameworks. Needless to say, this is not a very successful approach. You need to iterate when developing the metaware.

Start by developing some deep understanding *of a small part* of the domain for which you build the DSL. Then build a little bit of language, build a little bit of generator and develop a small example model to verify what you just did. Ideally, implement all aspects of the metaware for each new domain requirement before focusing on new requirements.

Especially newbies to MD* tend to get languages and meta models wrong because they are not used to "think meta". You can avoid this pitfall by immediately trying out your new language feature by building an example model and developing a compatible generator.

Co-evolve concepts and language ★★½

In cases where you do a real domain analysis, i.e. when you have to find out which concepts the language shall contain, make sure you evolve the language in real time as you discuss the concepts.

Defining a language requires formalization. It requires becoming very clear – formal! – about the concepts that go into the language. In fact, building the language, because of the need for formalization, helps you become clear about the concepts in the first place. Language construction acts as a catalyst for understanding the domain!

I recommend actually building a language in real time as you analyze your domain: over the last two years I have been doing this with textual editors in the domain of software architecture, with extremely good results. As we define, evolve and verify a system's architecture with the team, I build the architecture DSL in real time.

To make this feasible, your toolkit needs to be lightweight enough so support language evolution during domain analysis workshops. Turnaround time should be minimal to avoid overhead (the more a tool uses interpretation to work with the DSL, the better). You also have to tackle the Evolution issue (see above). Textual languages, with models stored as text files, are a good option here. Model migration can be done mostly via global search and replace.

Documentation is still necessary ★★½

Building DSLs and model processors is not enough to make MD* successful. You have to communicate to the users how the DSL and the processors work. Specifically, here's what you have to document: the language structure and syntax, how to use the editors and the generators, how and where to write manual code and how to integrate it as well as platform/framework decisions (if applicable).

Please keep in mind that there are other media than paper. Screencasts, videos that show flipchart discussions, or even a regular podcast that talks about how the tools change are good choices, too.

And please keep in mind that hardly anybody reads reference documentation. If you want to be successful, make sure the majority of your documentation is example-driven or task-based.

When selecting MD* tools, make sure that the meta ware artifacts (meta models, templates, transformations, etc.) as well as your models support comments in a meaningful and scalable way.

Reviews ★★½

A DSL limits the user's freedom in some respect: they can only express things that are within the limits of DSLs. Specifically, low-level implementation decisions are not under a DSL user's control because they are handled by the model processor.

However, even with the nicest DSL, users can still make mistakes, the DSL users can still misuse the DSL (the more expressive the DSL, the bigger this risk).

So, as part of your development process, make sure you do regular model reviews. This is critical – but not limited - especially to the adoption phase when people are still learning the language and the overall approach.

Two notes: reviews are easier on DSL level than on code level. Since DSL "programs" are more concise than their equivalent specification in 3GL code, reviews become more efficient.

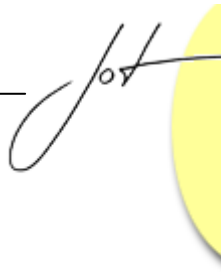
Also, if you notice recurring mistakes, things that people do in a "wrong" way regularly, you can either add a constraint check that detects the problem automatically, or (maybe even better) consider this as input to your language designers: maybe what the users expect is actually correct, and the language needs to be adapted.

Let people do what they are good at ★★½

MD* offers a chance to let everybody do what they are good at. There are several clearly defined roles, or tasks, that need to be done. Let met point out two, specifically.

Experts in a specific target technology (say, EJB on JBoss) can dig deep into the details of how to efficiently implement, configure and operate a JBoss application server. They can spend a lot of time testing, digging and tuning. Once they found out what works best, they can put their knowledge into generator templates, efficiently spreading the knowledge across the team. For the latter task, they will collaborate with generator experts and language designer – our second example role.

The language designer works with domain experts to define abstractions, notations and constraints to accurately capture domain knowledge. The language designer also works with the architect and the platform experts in defining code generators or interpreters. For the role of the language designer, be aware that there needs to be some kind of predisposition in the people who do it: not everybody is



good at "thinking meta", some people are simply more skewed towards concrete work. Make sure you use "meta people" to do the "meta work".

There's also a flip side here: you have to make sure you actually do have people on your team who are good at language design, know about the domain and understand target platforms. Otherwise the MD* approach will not deliver on its promises.

Domain Users Programming? ★★☆

We already alluded to the fact that domain users aren't programmers, but are still able to formally and precisely describe domain knowledge. Can they actually do this alone?

In many domains, usually those that have a scientific or mathematical touch, they can. In other domains you might want to shoot for a somewhat lesser goal. Instead of expecting domain users and experts to independently specify domain knowledge, you might want to pair a developer and a domain expert. The developer can help the domain expert to be precise enough to "feed" the DSL. Because the notation is free of implementation clutter, the domain expert feels much more at home than when staring at 3GL source code.

Initially, you might even want to reduce your aspirations to the point where the developer does the DSL coding based on discussions with domain experts, but then showing them the resulting model and asking confirming or disproving questions about it. Putting knowledge into formal models helps you point out decisions that need to be made, or language extensions that might be necessary.

If you're not able to teach a business domain DSL to the domain users, it might not necessarily be the domain users' fault. Maybe your language isn't really suitable to the domain. If you encounter this problem, take it as a warning sign and take a close look at your language.

Domain Users vs. Domain Experts (unrated)

When building business DSLs, people from the domain can play two different roles. They can participate in the domain analysis and the definition of the DSL itself. On the other hand, they can use the DSL to express specific domain knowledge.

It is useful to distinguish these two roles explicitly. The first role (language definition) must be filled by a domain *expert*. These are people who have typically been working in the domain for a long time, maybe in different roles, who have a deep understanding of the relevant concepts and they are able to express them precisely, and maybe formally.

The second group of people are the domain *users*. They are of course familiar with the domain, but they are typically not as experienced as the domain experts

This distinction is relevant because you typically work with the domain *experts* when defining the language, but you want the domain *users* to actually work with the language. If the experts are too far ahead of the users, the users might not be able to "follow" along, and you will not be able to roll out the language to the actual target audience.

Hence, make sure that when defining the language, you actually cross-check with real domain *users* whether they are able to work with the language.

Metaware as a product ★★½

The language, constraints, interpreters and generators are usually developed by one (smaller) group of people and used by another (larger) group of people. To make this work, consider the metaware a product developed by one group for use by another. Make sure there's a well defined release schedule, development happens in short increments, requirements and issues are reported and tracked, errors are fixed reasonably quickly, there is ample documentation (examples, examples, examples!) and there's support staff available to help with problems and the unavoidable learning curve. These things are critical for acceptance.

A specific best practice is to exchange people: from time to time, make application developers part of the generator team to appreciate the challenges of "meta", and make meta people participate in actual application development to make sure they understand if and how their metaware suits the people who do the real application development.

Compatible Organization ★★☆

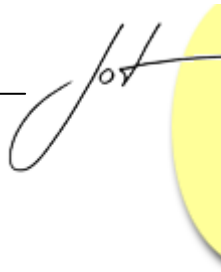
Done right, MD* requires a lot of cross-project work. In many settings the same metaware will be used in several projects or contexts. While this is of course a big plus, it also requires, that the organization is able to organize, staff, schedule and pay for cross-cutting work. A strictly project-focused organization has a very hard time finding resources for these kinds of activities. MD* is very hard to do effectively in such environments.

Make sure that the organizational structure, and the way project cost is handled, is compatible with cross-cutting activities. You might want to take a look at the Open Source communities to get inspirations of how to do this.

Forget Published Case Studies ★½☆

Many "new" approaches to software development are advertised via published case studies. While they are somewhat useful to showcase examples, they are not enough to make a real decision. DSLs are by definition *domain specific* – seeing how other people use them might not be very relevant to your situation. Some case studies even publish numbers like "we generate 90% of the code". That's of course useless. Because if modeling is 10 times more work than coding, the total effort is the same. Also, those numbers don't address lifecycle cost and quality.

The only real way to find out whether DSLs and MD* are good for you is to do a prototype. Make sure you use an agile approach and lightweight tools and ensure that 4 person weeks are enough to achieve in a meaningful result (possibly using external help if the team is new to building metaware). Look for a small, but representative example that can be extrapolated to your real system. Be sure, when looking at the resulting numbers, to add some overhead for lifecycle cost – there is non-linearity involved when extrapolating from a 4 week prototype to using the approach



strategically. But doing a prototype still gives you much more insight than reading a case study.

5 OPEN ISSUES

Before we conclude this paper, here is a set of challenges, or open issues, for which the community and the tool vendors have to find satisfactory solutions. Note that for most of the issues there's some (proposed) implementation somewhere. But it's not generally part of industry-strength tools, or even an agreed-to best practice.

Mixing Notations is still a problem. There's no tooling available to easily build DSLs that for example embed textual notations in graphical models (with complete editor support for both), or to build DSLs that use formula-editor-like, semi-graphical syntax. Intentional Software is moving in that direction, but their tooling is not generally available. And I know of no other tool.

Language Modularity and Composition is also a challenge in some environments. Especially in textual languages that operate based on parser technology, combining parsers is non-trivial. Systems like Intentional's and Jetbrains' MPS, that store (textual) models as structured meta data have an advantage here. Also, systems like MetaEdit+ can handle language modularization quite well.

Metaware Refactoring is not supported in most systems, although there's no specific reason why it couldn't. In my view it's just one of those things that needs to be done. Not conceptual challenges here.

Model/Code Refactoring is not quite that trivial. What I mean here is that if you have manually written code that depends on code that is generated form a model, and if you then change the model (and hence the generated code), what happens to the manually written code? Currently, nothing. Ideally, the manually written code is automatically changed in a way that keeps it current with regard to the changed model.

Automatic Model Migration is also not a solved issue. What do you do with your models if your language changes? Discard them, not being able to open them anymore? Open them in the new editor, but flag the places where the old model is incompatible with the new language? Automatically try to migrate? All those options exists, and while the first alternative is clearly unacceptable, I am not sure how a general best practice would look like.

Model Debugging, i.e. debugging a running system on model level is also not generally available. While you can always hand-construct specific solutions (such as debugging a state chart on an embedded device), there's no tooling available to generally support the implementation of such debuggers.

Interpretation and Code Generation are often seen as two alternatives, not as a continuum. What you maybe really want is an interpreter, where you can selectively use code generation for the parts for which interpretation is too slow – some kind of partial evaluation. There's research, but there's nothing generally available.

Handling large or many models is also a non trivial issue. How do you scale the infrastructure? How do you do impact analysis if something changes? How to you

navigate large or many models? How do you efficiently search and find? How do incrementally visualize them?

Finally, **Cartridges** is a term that get quite a bit of airplay, but it's not clear to me what it really is. A cartridge is generally described as a "generator module", but how do you combine them? How do you define the interfaces of such modules? How do you handle the situation where to cartridges have implicit dependencies through the code they generate?

So, there's a lot of challenges to work on – let's get started

6 ACKNOWLEDGEMENTS

Thanks to Steve Cook, Axel Uhl, Jos Warmer, Sven Efftinge, Bernd Kolb, Achim Demelt, Arno Haase, Juha Pekka Tolvanen, Jean Bezivin, and Peter Friese for feedback on prior versions of this article – the feedback really did help in making the article much better! I also want to thank the people who voted in response to my survey: all of the above, plus Jeff Pinkston, Boris Holzer, Gabi Taentzer, Miguel Garcia, Hajo Eichler, Jorn Bettin, Karsten Thoms, Keith Short, Anneke Kleppe and Markus Hermannsdörfer.

7 REFERENCES

[1] Kelly, Tolvanen, Domain Specific Modeling, Wiley, 2008

[2] Voelter, Stahl, Model Driven Software Development, Wiley, 2006

[3] Markus Voelter, Patterns for Model-Driven Development, http://www.voelter.de/data/pub/MDDPatterns.pdf

About the author

Markus Voelter works as an independent researcher, consultant and coach for itemis AG in Stuttgart, Germany. His focus is on software architecture, model-driven software development and domain specific languages as well as on product line engineering. Markus also regularly writes (articles, patterns, books) and speaks (trainings, conferences) on those subjects. Contact him via voelter@acm.org or www.voelter.de.