Foreword

Model-Driven Development is increasingly becoming an important part of today's software development landscape. While it comes in various flavors (and each with its own abbreviation: MDD, MDSD, MDE, MIC, MDA, DSM, GP), the essence is the same in all of those: use Domain-Specific Languages to make software development more efficient. Note that I didn't define the approach via code generation – that's only one aspect. Being able to verify models on the abstraction level of the domain and using notations familiar to the domain expert are both important ingredients, too.

And no, you don't do MDSD if you generate class skeletons from UML diagrams – un-profiled UML is a modeling language for *software*, and is by definition not domain-specific. Also, just drawing the pictures and not processing the models is not MDSD. DSL models, specifications or programs (or as Anneke Kleppe calls then: mograms) need to be first class citizens in the software development process: the implementation of the system must be derived from those models in an automated way.

Most of the benefits of the approach are obvious (to the readers of this foreword, at least), and I am not going to repeat the usual suspects here. Let me emphasize some of the maybe not-so-obvious ones.

A language that allows someone to express something directly enables people to talk about things much more efficiently. Case in point, over the last year, I have used textual DSLs to express software architectures. I've developed the architecture description languages together with my customers in architecture design meetings, the languages were specific to their project/platform. It's hard to overemphasize the effect this has on architects and developers alike: they now have a language to directly express architecture, architecture becomes tool-processable, and the understanding of the system's architecture improves quite a bit – the architectural concepts are clearly and formally defined as opposed to being vague concepts defined on a Wiki page or in a Word document.

With tools getting better and more mainstream, domain experts can actually become "developers" in the sense that they directly produce input into the software development tool chain. While software developers might doubt this, in many domains, experts are able to precisely and formally specify their knowledge if you give them the right language (insurance mathematicians, physical scientists, logisticians, to name a few). They might not be programmers, i.e. they might not be able to specify Turing-complete algorithms, but they are able to produce unambiguous, tool-processable representations of their domain knowledge. Getting domain experts directly into the tool chain (as opposed to having them write vague requirements documents) can be a huge boon for development efficiency.

In systems engineering, models are used all over the place. Control engineers, for example, have been working with models for a long time. Of course, their understanding of what a model is may be a bit different from ours. But nonetheless there's a huge potential for systems development once software is also specified via models – the overall system can be specified with a set of integrated models, making overall integration much simpler.

However, there are also still a number of challenges that need to be addressed if MDSD should really become the mainstream:

The whole area of scalability needs to be addressed. I am using this term very broadly: many and large models, several (integrated) viewpoints, languages evolving over time as well as keeping the overview over the models and model elements. While those problems are often specific to tool platforms, and some tools address some of those better than others, it's certainly not the case that all mainstream tools can do all of this well enough.

The seamless integration of different forms of syntax (mainly graphical and textual) is also not commonplace. Most tools can do one form of syntax well, but most can't do both. Specifically, the ability to mix graphical and textual syntax in the same diagram, with good tool support and editing usability for all forms does not really work in practice as of now. Also, modularization and reuse of parts of DSLs is not a solved problem either. To make things efficient, it is necessary to be able to compose languages: and that is including their semantics and concrete syntax, not just merging their meta models.

Finally there are the dreaded tool integration issues. Of course we have standards such as XMI (and version 2.1 actually kind of works!), but many very relevant tools don't use those standards, or don't support them wholeheartedly. In the automotive domain, for example, there's a real need to integrate with tools like Matlab/Simulink or Ascet – and XMI is not supported by those tools. Also, there's more to integration than file-based data exchange. You might want to be able to build workbenches that integrate the modeling paradigms supported by several tools into coherent workbenches, considering the various models viewpoints of the same overall system and updating the models with regards to each other in real time.

So, where does this leave us? First of all, I guess I am a big fan of open platforms and standards (on the right meta level!). Platforms like Eclipse and specifically, Eclipse Modeling are hugely important, since they have the potential to attract many players from different backgrounds – imagine a future version of Matlab/Simulink or Ascet based on EMF – we'd be quite a bit closer to the workbench I've talked about.

Also, I want to encourage everybody to focus on solving the really relevant problems in MDSD. We don't need support for automa{t|g}ic model migration or automatic derivation of architectures from requirements documents. For now, let's make the tool support for working with DSLs as good as for Java code!

Finally, I am pretty sure we're on the right track –there is progress. For example, a couple of years ago, code generation was something people considered an area for innovation. Today, code generation is basically a solved problem, we know how to write modular, maintainable code generators. If you want to advertise for your tool, you have to "show off" other features, such as model transformation or support for fancy syntax. That's a Good Thing ™, since it is a sign of progress.

Let's keep going!

Markus