

Generative Software Engineering

Jean-Marc Jézéquel

**Irisa (Université de Rennes 1 et INRIA)
Campus de Beaulieu
F-35042 Rennes
<http://www.irisa.fr/prive/jezequel>**

1. Introduction

Computer-based systems have been growing in complexity at an exponential rate (roughly 10 fold increase every ten years) for more than 40 years. Outside of ivory towers, the assumption that you start software development with a well defined specification and you develop your software from scratch so that it meets its specification has no longer any believers. In the real world of software engineering, continuous change and evolution is the norm. Instead of producing one program for solving one problem at a given point in time, the real issue in software engineering is to produce families of software, either or both in the time dimension (successive versions) and/or in the spatial dimension (the variants in a product line). Depending on the context, this has to take into account the usual trade-off between time to market, cost and quality (including reliability).

There is thus a growing need for mechanisms that help to automatically construct correct programs from independently developed elements that can be combined in flexible ways to allow an easier management of changes. These mechanisms should come as complements to the structuring elements available in prevailing languages of today: functions, modules, traits, classes, packages, components, etc. Whatever structure we choose for a complex system, there will indeed be concerns that do not fit into the structure, and with traditional design methods this has a tendency to be non-functional concerns like synchronization, memory management, caching policies, monitoring etc. These are known as "crosscutting concerns", because taking care of these concerns has to be done by small portions of code distributed all over an otherwise well structured program, thus making its evolution costly, tedious and risky.

Researching evermore abstract and powerful ways of composing programs is the meat of software engineering for half a century. Important early steps were subroutines (to encapsulate actions) and records (to encapsulate data). A large step forward came with the introduction of the object-oriented concepts (classes, subclasses and virtual methods) where classes can encapsulate both data and behaviors in a very powerful, but still flexible, way. For a long time, these concepts dominated the scene, but eventually the need for additional concepts became apparent. In this chapter, we focus on model driven engineering (MDE), which complement and leverage other mechanism such generic constructs, aspect oriented programming (AOP), component based software development (CBSE), generative programming, domain specific languages.

2. Model Driven Engineering

Like in other sciences, people have indeed been relying more and more on modelling to try to master this complexity. Modeling, in the broadest sense, is the cost-effective use of a simplified representation of an aspect of the world for a specific purpose.

Modeling is not just about expressing a solution at a higher abstraction level than code. This has been useful in the past (assembly languages abstracting away from machine code, 3GL abstracting over assembly languages, etc.) and it is still useful today to get a holistic view on a large C++ program. But modeling goes well beyond that.

Modeling is indeed one of the touchstone of any scientific activity (along with validating models with respect to experiments carried out in the real world). Note by the way that the specificity of engineering is that engineers build models of artefacts that usually do not exist yet (with the ultimate goal of building them).

In engineering, one wants to break down a complex system into as many models as needed in order to address all the relevant concerns in such a way that they become understandable enough. These models may be expressed with a general purpose modeling language such as the Unified Modeling Language (UML), or with Domain Specific Languages (DSL) when it is more appropriate.

Note that the real challenge here is not on how to design the system to take a particular aspect into account: there is a huge design know-how in industry for that, often captured in the form of Design Patterns. Taking into account more than one aspect at the same time is a little bit more tricky, but many large scale successful projects in industry are there to show us that engineers do ultimately manage to sort it out (most of the time).

The real challenge in a product-line context is that the engineer wants to be able to change her mind on which version of which variant of any particular aspect she wants in the system. And she wants to do it cheaply, quickly and safely. For that, redoing by hand the tedious weaving of every aspect is not an option.

MDE does not propose to solve this problem upfront, but just to mechanize and reproduce the process experienced designers follow by hand. The idea is that when a new product has to be derived from the product-line, we can automatically replay most of this design process, just changing a few things here and there.

Usually in science, a model has a different nature than the thing it models (think of a bridge drawing vs. a concrete bridge). Only in software and in linguistics a model has the same nature as the thing it models. Because in software a model has the same nature as the thing it models, this opens the possibility to automatically derive software (and other artefacts such as test cases, performance profiles, or documentation) from its model. This property is well known from any compiler implementor (and others), but it was recently made quite popular with initiatives such as Model Integrated Computing (MIC), Microsoft's Software Factories or OMG's Model Driven Architecture (MDA), globally known as Model Driven Engineering MDE.

This requires that models are no longer informal, and that the weaving process is itself described as a program (which is as a matter of facts an executable meta-model) manipulating these models to produce a detailed design that can ultimately be transformed to code or at least test suites.

INRIA has been a pioneer in MDE research from many years. For instance, the Triskell project has developed the Kermeta metamodeling language for describing both the structure and the behavior of metamodels. It has been designed to be compliant with the OMG metamodeling language EMOF (part of the MOF 2.0 specification) and Ecore (from Eclipse). It provides an action language for specifying the behavior of models. Kermeta is intended to be used as the core language of a model oriented platform. It has been designed to be a common basis to implement Metadata languages, action languages, constraint languages or transformation language. Kermeta is statically typed, with generics as well as function types to allow OCL's forall/exist/iterate style of closures. It also directly supports model-oriented concepts like associations, multiplicities or object containment management.

As can be seen in the three contributions forming the main content of this chapter, Kermeta is widely used for supporting a large range of MDE related activities.

First, Erwan Breton and Frédéric Madiot are looking back on 10 years of MDE by presenting how Model-Driven Engineering has been used at Sodifrance.

Second, Francis Alizon, Mariano Belaunde, Grégoire Dupé, Yves Le Traon, Bertrand Nicolas, Sébastien Poivre, and Jacques Simonin are reporting on several MDE experiments in France Telecom.

Finally, Robert B. France (Colorado State University, USA) gives a more research oriented view on the subject and describes how Model Driven Engineering helps in Managing Software Complexity, through the notion of Aspect Oriented Modeling.