

Bootstrapping Domain-Specific Model-Driven Software Development within Philips

Hans Jonkers
Marc Stroucken
Richard Vdovjak

Philips Research
High Tech Campus 31
5656 AE Eindhoven, The Netherlands

{Hans.Jonkers,Marc.Stroucken,Richard.Vdovjak}@philips.com

Abstract

Philips recognizes the importance of model-driven software development (MDD). Unfortunately, there seems to be a lack of mature tools that would support domain-specific MDD and allow their deployment in an incremental fashion. This paper describes the ongoing MDD research efforts at Philips, introducing VAMPIRE¹ – a light-weight model-driven approach to domain-specific software development. The VAMPIRE MDD framework is developed by Philips Research and it is currently being deployed at several Philips product divisions. The paper elaborates on the VAMPIRE modeling environment, focusing on its meta-modeling facilities, editors, and generators. Further, we summarize the lessons learned during the process of deploying our MDD framework into the product divisions.

1. Introduction

To be a successful player in a competitive business environment such as consumer electronics, medical systems, or healthcare solutions, one needs to deliver integrated and interoperable software-intensive systems, and fulfill the ever-growing demand for new and improved features in ever-decreasing time to market. Moreover, the whole range of product families is required to exhibit a similar look and feel towards the end-user. These requirements affect the product's hardware as well as its software. The proliferation of software in Philips products has been substantial and the amount of software still continues to grow at a great pace. As a consequence, the whole process in which the software is designed and constructed needs to address the aforementioned requirements efficiently.

The struggle to increase the software productivity and reliability has accompanied software development efforts since the very beginning. Among the remedies that proved to be (at least partially) successful was the raising of the abstraction level for writing code. Throughout the evolution of programming languages and design techniques (e.g. procedural languages, object-orientation, and design patterns) one can clearly see an increase in the level of abstraction at which software was written. The booming proliferation of software across many fields makes the demand for software higher than

¹ Visual Agile Model-driven Product-oriented Integrated Research Environment

ever before. The demand currently exceeds the ability to produce software by a large margin, and this “software-gap” steadily increases in time.

To address this issue, the conventional development means need to be augmented by new approaches that would enable to raise the level of abstraction even further, while bringing the software engineering discipline closer to the actual domain where it is to be applied. The combination of a higher abstraction level closely coupled with the domain knowledge introduces a so-called model-driven development (MDD) – a paradigm shift in software engineering that has the potential to become a solution to the software-gap problem. To make this happen, we need to gradually move from writing code to creating domain-specific models and generating code and other related artifacts, such as documentation, etc., from them. Using small domain-specific modeling languages, as opposed to a universal modeling language such as UML, brings the modeling discipline much closer to the domain experts and at the same time enables simpler maintenance and evolution of such models, which contributes to the desired productivity increase as well as to the agility of the model-driven development.

The rest of the paper is structured as follows. Section 2 introduces the VAMPIRE modeling framework, section 3 elaborates on VAMPIRE’s meta-modeling features defined by the *Meta Object Model*, section 4 focuses on model editors, and section 5 explains the ideas behind our code generator. Section 6 summarizes the lessons learned during the process of applying this framework in Philips product divisions. Finally, section 7 gives an overview of related work, and section 8 presents our conclusions.

2. VAMPIRE modeling framework

VAMPIRE is a light-weight model-driven approach to domain-specific software development. The VAMPIRE framework is being developed by Philips Research. It primarily aims at raising the level of abstraction at which the software for Philips products is produced, trying to increase productivity and reliability. The main idea is to capture the domain knowledge by means of models and to develop (parts of) applications by instantiating these models and generating the code, documentation, and other artifacts. VAMPIRE consists of a collection of loosely-coupled tools and in some sense represents a minimalist approach that allows us to apply MDD now and not wait till tools that would fit our needs appear.

The VAMPIRE framework is based on a very simple pattern, involving *object models*, *editors* and *generators* (see Figure 1).

Object models define the essential entities in the domain(s) of interest. There need not be a single object model capturing a complete application domain, but there can be several small object models such as models capturing the variation points of specific products.

Editors enable manual construction of model instances conforming to an object model. The editors allow users to construct these model instances in an intuitive and domain-specific way and completely hide the underlying implementation technology.

Generators facilitate the generation of various artifacts from model instances. The generators allow (parts of) the software development process to be automated, i.e. to be “driven” by the models.

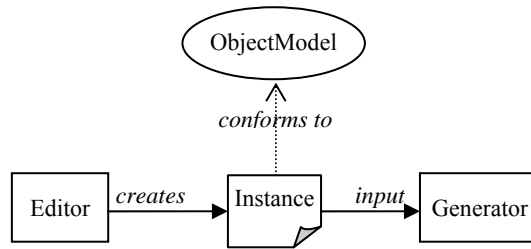


Figure 1: VAMPIRE modeling pattern.

Setting up an MDD approach like VAMPIRE requires the definition of a *meta-model* – a modeling language defining the class of models that the framework supports. In VAMPIRE, this is represented by the *Meta Object Model* (see section 3) which is used to formally describe models of a certain domain. These models can be viewed as *abstractions* of entities within the given domain. In VAMPIRE, a model constitutes a network (graph) of *types* that capture domain knowledge in a certain area (e.g., medical systems). Object models thus serve as *languages* for defining instances of models and are therefore also referred to as *domain-specific languages* [1]. An important feature of VAMPIRE is that it considers models not just as abstractions but as concrete objects from which artifacts such as executable code and documentation can be automatically generated.

The models we develop for our product domains are relatively small in size, usually not exceeding 50 mostly product-specific type definitions. However, given the fact that there are many related products (creating groups of so-called product families) it is often the case that various (related) models need to be reused, combined, or their elements simply have to refer to other models' elements.

In VAMPIRE, it is possible to combine models in different ways. Models can reference types in other models, thus building separate but linked models. Models can extend existing models, which allows the sharing of common parts (model inheritance). Finally, types can be extended with different aspects, which facilitates building models incrementally while separating concerns (Aspect Oriented Modeling).

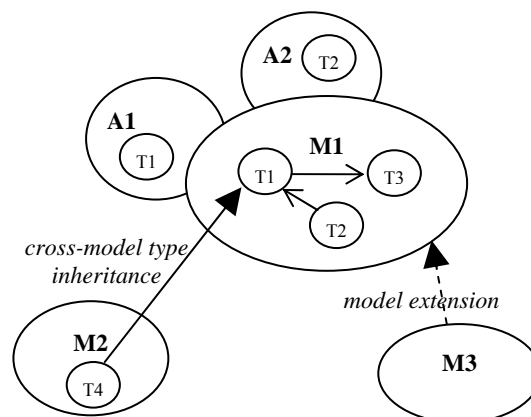


Figure 2: Aspect orientation and model reuse.

Figure 2 depicts three models (*M1*, *M2*, and *M3*). Models typically contain several interrelated type definitions, e.g. *M1* contains types *T1*, *T2*, and *T3*. As mentioned, models can be extended by various aspects. An aspect is also a model; it defines (extensions to) types that are combined with the types of the model to which the aspect is associated. For instance, aspect *A1* defines *T1* which is combined with the definition of *T1* from model *M1*; this mechanism to some extent resembles partial-classes from C# 2.0 but then at model level. The advantage of aspect orientation is the clear separation of concerns. One can simply define different aspects to already existing models, e.g. XML serialization details can be captured as an independent aspect.

Models can be also combined with, extended, or reused by other models. For instance, model type *T4* from *M2* refers to (inherits from) a type *T1* from *M1*, and model *M3* inherits all types from *M1*.

3. Meta Object Model (MOM)

The Meta Object Model (MOM) is defined as a combination of different aspects. Current aspects include the model itself, documentation and XML serialization. The documentation aspect supports the annotation of models with summaries, status, etc. The XML aspect allows for customization of the XML language of the instances. This even makes it possible to model existing XML languages like XML Schema (XSD), SVG, XML and XSLT. By so doing, the generation of such an XML file is reduced to a model transformation followed by a save-to-file operation.

The MOM is an instance of itself which brings great flexibility. It is possible to transform any object model to a MOM instance and vice versa. This enables building tools such as validators to perform additional semantic checks, normalizers that resolve model extensions by creating a single self-contained model, and so-called defaults-resolvers that add or remove default values, which makes navigating through the MOM instances easier from the code. In the code, any MOM instance can be accessed by the graph it represents. When saving an instance of an object model, the graph representation is mapped to a tree-structure imposed by XML. This is done by a special model construct that allows indicating at which location a definition of a certain item is expected and at which other locations in the model references to that item may exist. A stubs-resolver replaces all reference stubs by real references, so that at any location the item's definition becomes transparent to the programmer.

Since the MOM is a generic meta-model, it is very small with just 5 basic constructs: *class*, *union*, *list*, *enumeration* and *value type*. Classes contain attributes that can be optional. This makes it possible to test for the presence of (and even remove) an attribute from a class object. The MOM currently supports single-inheritance, but the work on supporting multiple-inheritance is in progress. This is possible because the generated C# code is fully interface-based and multiple interface inheritance is supported in C#.

Unions represent a special kind of forward type declaration. They define (closed) sets of classes or other unions; the instance of a union type is in fact an instance of exactly one type within the defined set. From the modeling perspective, unions can also be seen as a (type) choice. Unions do not have attributes of their own and support a weak form of multiple inheritance.

Lists are modeled closely to the generic list type of C# 2.0. Enumerations contain literal values. Value types are types that restrict some other basic types, e.g. the name of a C# identifier may subtype string with the restriction that it does not contain spaces.

The generated C# programming library enables a MOM-oriented reflection. So in code it is possible from any model instance to access and explore its object model, which brings more flexibility for writing artifact generators (see section 5).

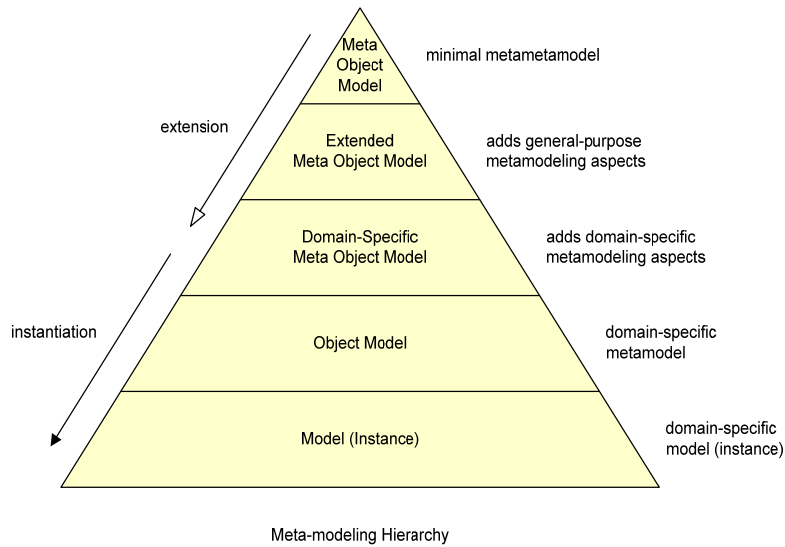


Figure 3: The MOM meta-modeling pyramid.

The MOM is extensible. Since it is described in itself, an extension to the MOM is achieved in 2-pass iterations. In the first pass the MOM is extended and new code is generated from it. In the second pass the code generator is extended to start using the new constructs. Extensions that are independent of the C# model are even simpler and can be added at any time. Therefore, extending the MOM with new aspects, e.g. layout metadata for a graphical editor is very easy. Figure 3 illustrates the extension pyramid of MOM. It starts with the MOM itself as the basic building block followed by general-purpose meta-modeling extensions such as serialization; this layer can be (optionally) extended with domain specific meta-modeling extensions. These meta-modeling facilities are then used to describe concrete domains in terms of Object Models and their instances (Models).

4. Editors

The first letter from the VAMPIRE acronym stands for “Visual”, emphasizing that the visual aspect plays a crucial role in the MDD way of working. The fact that models and their instances are edited visually (as opposed to writing code or hand-crafting XML files) makes the learning curve much less steep and the actual process of modeling much more appealing. The visual aspect also contributes to the desired agility of our approach, as it is much easier for people to reason and change models if they are represented visually.

There are several approaches to model / instance visualization, browsing, and editing. These approaches range from more generic table-based model editors (such as the one depicted in Figure 4) to more diagrammatic or pictorial editors. The latter have the potential

to offer more (domain) specific elements in the visualization, but of course then they become model dependent and therefore need to be individually tailored for every single application domain.

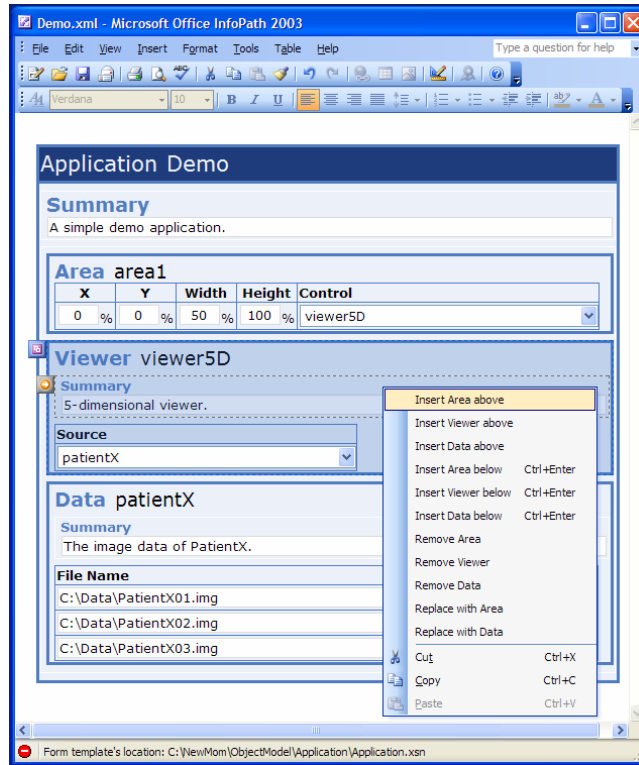


Figure 4 InfoPath-based Application Editor.

The VAMPIRE framework offers an extensible set of loosely-coupled generic tools for editing and browsing models and their instances. Most of these tools are currently based on existing XML-enabled editing suites such as InfoPath² or XMLspy³.

The generic editors provided by the framework can be made more domain-specific by extending the default generators. Future development includes a customizable suite of diagrammatic and pictorial tools for visual model editing.

5. Generators

The VAMPIRE framework includes a C# code generator which takes an object model as its input and produces a C# library of types occurring in the model together with a set of interfaces for access and manipulation. The library is fully interface-based implying that instances of model types can only be accessed / modified by interfaces [6].

The generated code provides an easy-to-use programming model for instances of an object model, where all constructs defined in the object model, are also available in C# using Intellisense of VS.NET (see Figure 5). This strongly-typed approach of model instance to C# conversion brings the benefit of compile-time checks and an early discovery of potential inconsistencies (which may occur especially when models are instantiated by humans).

² <http://www.microsoft.com/office/infopath>

³ <http://www.Altova.com/XMLSpy>

```

void TransformApplication( string inputFile, string outputFile )
{
    M.Application app = C.Application();
    app.Load( inputFile );

    foreach( M.|

```

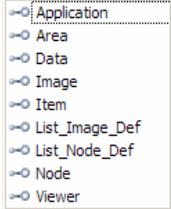


Figure 5: Model-aware Intellisense support.

Multiple views in the code provide a value-oriented, object-oriented, and an attribute-oriented way of working. The adopted interface-based approach allows switching between the different views. Construction methods are generated for the types to reduce the lines of code you have to write in a generator.

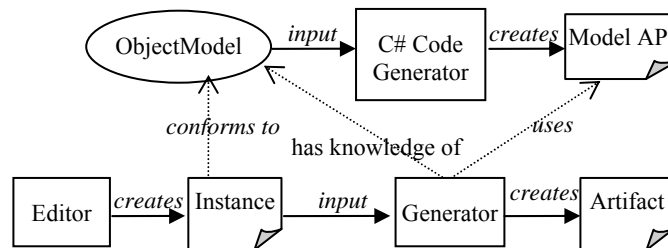


Figure 6: Artifact Generator.

Every model has its own XML representation for which an XML schema file can be generated. The generated C# code provides load and save methods for the instances of the model that conform to their XML representation. The default XML representation makes efficient use of XML attributes, elements or anonymous constructs.

Besides the C# code generator, the VAMPIRE framework offers a number of other tools such as XML schema generator, HTML documentation generator, and a generator of an SVG graphical model browser. On top of the “built-in” generators (which are generic for all instances of the MOM), one can write domain-specific generators that create domain-specific artifacts. Figure 6 shows such a generator; the generator uses the model API created by the C# generator and processes the instances of the model, creating domain-specific artifacts.

6. Applying VAMPIRE

In the past three years we have successfully applied our approach in several domains which, because of the proprietary nature of the products involved, we can only describe briefly in this paper.

During this time the approach has reached a maturity level where it can be transferred to other departments within Philips. Some departments are going to use the approach to research new models and description languages, while others will use the development environment to implement tools for their customers. Because of the fast development of models and transformations more time can be spent focusing on the content instead of the tooling.

The complexity of using MDD in existing software-intensive product lines is mainly identifying those parts in the software architecture that can be replaced by models with generated code. Certain skills, and a different way of thinking about software in general, are required. A trained eye sees possibilities for code generation almost everywhere, which requires switching between several meta-level views.

A typical target for applying MDD is the class of software that exists in many varieties in one product line, or software that changes a lot over time, e.g. with every new release. It may also be applied to develop architecture description languages (ADLs) [7].

Below we shortly summarize the lessons learned during the last few years when we were trying to introduce the VAMPIRE framework in Philips. This (not exhaustive) list can be seen as necessary prerequisites without which it would be very hard to apply MDD within an industrial environment like the Philips product divisions.

- Incremental deployment of MDD.

It is important to be able to introduce MDD in small evolutionary steps (as opposed to a sudden 100% MDD conversion). This is important especially in an environment which possesses a long history of products and usually has many obligations to third parties w.r.t. that.

- Very simple meta-modeling facility.

It is necessary to realize that the ultimate user of MDD will eventually be a (technically savvy) domain expert rather than a “purist” programmer⁴. Therefore, the learning threshold for embracing the MDD approach must be as low as possible and the actual models must be close to the domain where they are to be applied.

- The ability to combine/reuse/extend models.

Multitude of related products require their models to be also related. The agility of MDD brings even more benefits if one is able to reuse and combine existing models.

- Support the rapid development of generators (using MDD).

Tools like an efficient C# code generator or editor generators must be present in the framework to increase its usefulness, its ease of use, and ultimately to make it embraced by the community of modelers and developers.

- Software development paradigm shift.

The introduction of any paradigm shift takes time. We have to change the way in which developers perceive software, the way they think about development, and prove that the changes will bring benefits in the long run. Developers are often reluctant to change the way they work. MDD focuses on developing tools that generate (parts of) the end-product, instead of developing the product manually.

⁴ Excellent software developers will still be needed to develop MDD tools like (domain-specific) generators etc.

We believe that MDD has the potential to improve the way we create software and above all to make this process more efficient. However, as any new technology, also MDD must overcome the inertia of existing approaches, and even if all the technical ingredients are in place, it also requires the management support and their devotion to make the change happen.

7. Related work

In this short comparison we chose to focus on two MDD initiatives advocated by OMG and Microsoft, respectively. We note that there are a number of other model-driven frameworks and approaches that VAMPIRE can relate to. However, to our knowledge no existing approach offers an aspect-oriented way of modeling combined with (pure) interface-based C# code generation, features which are very important in our application domain.

MODEL DRIVEN ARCHITECTURE (OMG)

The OMG promotes model-driven architecture [2, 3]. In simple terms, their approach states that applications are developed by creating platform-independent models (PIMs) in UML. PIMs conform to domain-specific meta-models defined by UML profiles or by means of the meta-object facility (MOF). Model transformations map PIMs to platform-specific models (PSMs) and from there to code, etc. Tools exchange data by means of the XML Metadata Interchange (XMI) language. The OMG itself does not develop tools and the tool support is (to be) delivered by third parties.

MDA in some sense represents an MDD vision that other approaches can relate / comply to. VAMPIRE too adopts some of the MDA ideas, however, we consider our approach being more of a bottom-up nature (starting with a concrete domain) than top-down nature of MDA (having the universal language that needs some tailoring).

SOFTWARE FACTORIES / DSL (Microsoft)

Microsoft's *software factories* initiative [4] and the associated DSL tools [5] are very close to our approach therefore we compare it in more detail. The Microsoft approach uses a meta-language to define domain-specific languages (DSLs), which is not very different from our Meta Object Model. One important difference though, is that the VAMPIRE framework facilitates various ways of combining and reusing models; this feature is, to our knowledge, missing in DSL. The type of reflection provided by VAMPIRE and DSL also differs. While VAMPIRE allows for reflection at model level, DSL provide reflection at C# level only.

Another difference is that DSL use a template-oriented approach to artifact generation while VAMPIRE uses a (full-fledged) code-based approach. The former fits well in simpler types of artifacts such as documentation reports; however, writing templates that generate C# code is much more tedious, especially since (at the time of writing) supporting tools like Intellisense or the syntax highlighting are missing.

On top of that, DSL tools are still undergoing radical changes (to the better we believe) and their meta-model and APIs are not yet stable enough for use in production quality code. We continue to monitor the developments of the DSL tools and do not exclude the possibility to port our VAMPIRE framework onto this platform some time in the future.

8. Conclusions

In this paper we have described the essence of the VAMPIRE framework developed at Philips Research. Our approach targets both new product architectures as well as existing software intensive product lines, where the handwritten code can be incrementally replaced by generated code from models.

The framework is based on a pattern involving *models*, *editors*, and *generators* and the paper elaborated on these MDD “ingredients” in more detail. We have also summarized the lessons learned from applying our framework in Philips product divisions. Below we list some of the distinguishing features of VAMPIRE.

- By incorporating Aspect-Oriented Modeling and supporting multiple inheritance, VAMPIRE provides an easy way to build new models on top of existing ones, facilitating model extensibility and reuse.
- The generators are implemented as loosely-coupled tools associated with different model instances, and can also be combined into file- or memory-based generator pipelines, where output of one generator serves as input for another one.
- Flexible XML serialization format allows for modeling of existing standardized XML languages such as XML Schema. Creating output in such a standard XML language is as easy as building a model-to-model generator.
- Using a minimalist approach, we tailored VAMPIRE to the needs of our industrial applications. However, the framework proved to be powerful and extensible enough to be applied in different (unrelated) contexts.

In our experience, MDD has the potential to make the process of software creation much more efficient. In order to achieve that, the MDD way of thinking needs to be adopted by the developers and domain experts, some of whom will actually become application modelers.

Acknowledgements

We would like to thank our colleagues both from Philips Research and Philips product divisions for providing their valuable feedback on this work.

References

- [1] Czarnecki, K., Eisenecker, U. Generative Programming: Methods, Techniques and Applications. Addison-Wesley, 1999.
- [2] OMG Model Driven Architecture (MDA) URL: <http://www.omg.org>
- [3] Mellor, S., Scott, K., Uhl, A. Weise, D. MDA Distilled, Principles of Model Driven Architecture. Addison-Wesley Professional, 2004.
- [4] Greenfield, J., Short, K., Cook, S., Kent, S. Software Factories. Wiley, 2004.
- [5] Microsoft Domain-Specific Languages Tools. URL: <http://msdn.microsoft.com/vstudio/DSLTools/>, June 2006.
- [6] Steimann, F., Mayer, P. Patterns of Interface-Based Programming, Journal of Object Technology (JOT) Vol. 4, No. 5, July-August 2005.
- [7] Clements, P., C. A survey of architecture description languages, Proceedings of the 8th International Workshop on Software Specification and Design, Page(s):16 - 25, March 1996.