Why Models and Code Should be Treated as Friends

Mahdi Derakhshanmanesh, Jürgen Ebert

University of Koblenz-Landau,
Institute for Software Technology

{manesh, ebert}@uni-koblenz.de

Gregor Engels

University of Paderborn,
Department of Computer Science

engels@upb.de

Abstract

Various approaches have been proposed to face the difficulties related to constructing and maintaining modern software systems. Often, they incorporate models in some part of the development or evolution process. Even the use of models at runtime seems to receive more and more attention as a way to enable the quick, systematic and automated application of change operations on software as it executes. Assuming that existing systems have been largely developed in code and that novel target architectures depend on – or even embed – models to some extent, the possible roles of models and code as well as their interaction and interchangeability need to be thoroughly examined. In this position paper, we attempt to initiate a discussion on why models and code should become closer friends.

1 Introduction

Recent approaches to the construction and maintenance of modern software systems rely on models to a large extent. The most popular approach is the Model Driven Architecture (MDA) where large parts of the system are expressed in domain-specific representations, i.e., in *models*. These models are *transformed* stepwise towards a technical solution. Furthermore, they can be changed quickly and the complex technology-specific artifacts can be regenerated. Hence, this approach supports the systematic adaptation of development artifacts. Whenever change is required, the transformation chain needs to be retraversed and a new version of the software is generated, compiled and deployed.

In addition, there are other approaches that tackle the need for quicker (and possibly smaller) change operations while the software is operating. In these works, models are not discarded before deployment but are shipped with the rest of the system. In fact, these *runtime models* [2] are integral parts of the system and they may even "drive" its execution. That is, models – such as behavioral models – are interpreted at runtime; no code is generated. The core idea can be referred to as *direct model execution*.

Depending on the followed engineering approach, the portion of used models and code varies. While we believe that the use of models at runtime can support software evolution by enabling adaptivity at different levels of granularity, we are also aware of the fact that code execution is still the established and better-performing method. Therefore, the roles of models and code as well as their dependencies need to be well-understood to utilize the advantages of each side when defining the target architecture in software development as well as modernization.

In Section 2, we give examples of five typical relationships between models and code. Implications for software design and development are covered in Section 3, before we conclude the paper in Section 4.

2 Model and Code Relationships

Reflecting on parts of our previous work, especially on the Graph-based Runtime Adaptation Framework (GRAF) [1], we may state that at least five relationship constellations will occur naturally during software development that is based on mixing models and code. The resulting *spectrum* of different cases is sketched in Figure 1.

This spectrum can be seen as describing common ways of using models and/or code, but it is also a spectrum of non-functional properties of the resulting software systems as it suggests the existence of tension between *performance* and *flexibility*. The underlying assumption is that software represented by compiled code is less expensive than interpreting (behavioral) models w.r.t. memory consumption and execution time [3]. Yet, models can be changed easier at runtime. Hence, they are more flexible. Each constellation in this spectrum is described subsequently.

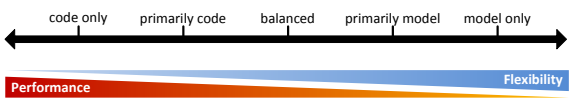

Figure 1: Model and Code Use in the Spectrum between Performance and Flexibility

Code Only. The software consists of (application) code and there are no models. Obviously, in this case, no mediation between models and code is needed. The system's interfaces are implemented conventionally in code. Furthermore, administration tasks such as inspecting or modifying the software's state are realized in code, too. For instance, they may be implemented with the programming language's reflection capabilities if supported.

Primarily Code. The code is the *primary part* (e.g., w.r.t. the control flow) but it is extended by models in places which need to be specifically flexible. For instance, parts of the behavior that are expected to be in need for (frequent) adaptation are expressed in terms of UML activity diagrams, statecharts, or Petri nets. In this case, the control flow – and potentially any kind of additional data – is redirected from code to models.

Balanced. A combination of the two previously mentioned "pure" cases is applied. Data needs to be synchronized bidirectionally between model and code parts and each side can invoke behavior implemented in the counterpart. In this case, both sides are closely integrated and wired to each other. We can imagine further cases, e.g., where behavioral models implement interfaces defined in code.

Primarily Model. The code is seen as a library of functionality and the main flow is actually dictated by models. Actions implemented in code are orchestrated by the model. Models can be seen as the primary part, whereas code represents the secondary part. For example, a model interpreter may trigger actions that are realized in code related to drivers or existing middleware. The Service Oriented Architecture (SOA) with its business process models is another example.

Model Only. The model part exists and there is no code, at least no application code. In this case, model interpreters are available – and potentially even tightly integrated with the meta-models – making models capable of stand-alone execution. This is the most flexible solution, as any change to the models impacts the software's state and execution behavior immediately. The collection of models and their interpreters *is* the program.

3 Implications

In many traditional software engineering processes, models are seen as the predecessors of code. The final product is – in the end – still made up of code. Hence, existing systems may be developed via model-based or model-driven approaches but the final outcome is usually comparable to "code only" software. To date, models can be involved in all phases of the software life-cycle: while the use of models in the early phases supports documentation and communication (model-based development) and the generation of code enables more systematic and semi-automated construction (model-driven development) the use of models at runtime supports primarily the adaptation of software via *query/transform/interpret* operations [1].

When developing or evolving systems to meet the ever-growing demand for faster turn-around times and adaptation of a "life" system, models offer attractive benefits as initially sketched. As a prerequisite, the target architecture needs to be supported by an *infrastructure* that can manage the relationships between models and code. In contrast to *models@run.time* [2], these models are not necessarily reflective.

A common approach to realizing these capabilities is to encode *model semantics* in the form of a model interpreter that is developed in the same language as the host programming language, e.g., Java. Hence, type-conversion problems do not occur and model and code are finally executed by the same virtual machine. Models are not compiled and so no redundant data needs to be stored for their execution. In the scope of a modernization project, it seems to be a desirable future goal to derive executable models from parts of the code as a basis for further evolution.

4 Concluding Remarks

As *first class entities* of a software system, models can play different roles with different relationships to code. We claim that the relationships between models and code need to be revised in this new context. In order to unleash the full potential of models (e.g., abstraction, flexible adaptation) and code (e.g., stable tools, high performance) for realizing flexible software architectures, we need a *generic infrastructure* that facilitates the integration of models and code as equally valuable constituents of tomorrow's software systems. The recently accepted DFG project *Model-Integrating Self-Adaptive Components* (MoSAiC), conducted together by the authors, will tackle this challenge.

Acknowledgments. The authors thank Thomas Iguchi for fruitful discussions and proofreading.

References

[1] Mehdi Amoui, Mahdi Derakhshanmanesh, Jürgen Ebert, and Ladan Tahvildari. Achieving Dynamic Adaptation via Management and Interpretation of Runtime Models. *Journal of Systems and Software*, 85(12):2720 – 2737, 2012.

[2] Gordon Blair, Nelly Bencomo, and Robert B. France. Models@run.time. *Computer*, 42(10):22–27, 2009.

[3] Edzard Höfig. *Interpretation of Behaviour Models at Runtime - Performance Benchmark and Case Studies.* PhD thesis, Technical University of Berlin, 2011.