Guest Editorial Preface

Special Issue on Model-Driven Development of Enterprise Software Systems

António Miguel Rosado da Cruz, Instituto Politécnico de Viana do Castelo, Viana do Castelo, Portugal

Modern organizations rely on software for their day to day operations, and for their tactical and strategic decision-making processes. Enterprise software systems embody that software to support organizations' business processes at any level. Engineering and developing enterprise software systems means not only to deal with the stakeholders' individual requirements, but also to ensure those requirements are aligned with the organization's strategy and its mission. Consequently, enterprise software systems are embedded with organizational knowledge, guiding the systems' behavior according to the organization's expectations, its business rules, its culture, etc.

A software development process consists of a sequence of activities and associated results that produces a software product (Sommervile, 2015), enterprise software included. This development process typically includes activities of eliciting requirements, modeling and analyzing the problem at hand, designing, modeling and prototyping a solution, and developing and testing the solution itself. Most of these activities include communicating, both with stakeholders and within the software development team. The main support for documenting and communicating systems' requirements and system's designed features are models. So, these models end up including all or most of the organizational knowledge that must be embedded into enterprise software systems.

Software models are at a higher abstraction level than code, and so it is easier for software engineers and developers, and stakeholders to reason and communicate about the software system by using a software model. These models are typically discarded after fulfilling their role of helping reasoning and communicating about the system, and ultimately after the development process has ended.

Model-driven development (MDD) is a different method (process and techniques) to software development, where models are not only aimed at reasoning and communicating about the system, but, together with code, models are productive artefacts. They are as important as code in the production of the final system. In fact, much of the code is automatically produced from models. This enables the separation of analyzing and designing the system from generating the system code. MDD models may include all the organizational knowledge required to produce the system code.

MDD approaches software development by constructing models that may be refined (transformed) through different levels of abstraction, from a platform independent level, or a computation independent level, to a platform specific model that is directly mapped to final code (Cruz, 2015; Frankel, 2003).

The first models constructed in a MDD process are platform independent models (PIM), meaning that they model a system in a platform independent way. A PIM can be defined in a computation and platform independent way. A computation independent model (CIM) does not model how to make or compute things, but only what is expected to be made, whilst a PIM may prescribe computations, provided it is made in a platform independent way. A CIM is typically only dependent of domain or business concerns, thus being independent of implementation platforms. A CIM describes the problem from the point of view of the business or domain environment. It is a model of the relevant part of the business to where a software system is going to be built, from the information system's perspective (Frankel, 2003; Warmer et al., 2003).

MDD relies on the definition of model-to-model (M2M) transformation processes that enable the transformation of PIMs to other PIMs or to platform specific models (PSMs).

From a PSM, code may be automatically generated by using a model-to-code (M2C) or model-to-text (M2T) transformation process, that is a code or documentation generation process (Warmer et al., 2003). MDD enables the collection of the knowledge acquired by an organization throughout its entire lifespan, and its storage in the form of computation or platform independent models. These models may be adapted and transformed as new business environments and constraints take place. And, at any time, they can be transformed to platform specific models when technological changes occur.

Model-driven development approaches try to solve productivity, portability, interoperability and maintenance and documentation problems (Warmer et al., 2003).

For instance, software projects productive activities are typically coding and testing, which leads the software team members to spend little time modeling or documenting the system. This becomes a problem when new team members need to maintain the software (e.g. fix bugs, enhance functionality). MDD tries to solve this problem by transferring the productive activities from coding and testing to modeling, and basing the software development process in model-to-model and model-to-text transformations. Models, being at a high abstraction level, are easier for newcomers to understand, lowering the effort and cost of software maintenance, thush increasing productivity. The portability problem has to do with the need to adapt existing systems to new technologies. By generating code from models, MDD helps maintaining the value of previous investments, as this promotes the separation of organizational knowledge in models, such as business rules, from the technical knowledge needed for developing and deploying the system to a given specific platform. The interoperability problem, the need to build several interoperating software systems, is tackled by enabling models to be component-oriented. By leveraging the notion of component from code to models it is possible to model interoperable component systems and code generated from models must also be interoperable, maintaining at code-level the interoperability modeled at model-level. Finally, the maintenance and documentation problems are mitigated by generating high level documentation from models, as low-level documentation is now currently generated from code and comments in code.

Articles in this issue:

1. A Formal Framework for Scalable Component-Based Systems

This article proposes a formal framework for designing and specifying scalable component-based systems together with the corresponding development environment and support for executing them. The article defines an incremental design methodology that considers components interfaces and their corresponding ports as the basic unit of software construction. Interfaces serve to assemble simple components to obtain either more complex ones or the entire software architecture. Additionally, interfaces supply interactions and synchronization effects on the underlying sub-system. The calculation process is also guided by changes on interfaces where the hierarchical structure of the underlying component- based system is maintained.

2. A Semantic Approach to Deploying Product-Service Systems

The second article notes that conceptual modeling may be employed for two classes of goals: (1) as input for run-time functionality (e.g., code generation) and (2) as support for design-time analysis (e.g., in business process management). This has led to a multitude of modeling languages that are conceptually redundant. Between these goals, an inherent trade-off manifests, related to selecting the most adequate language for each goal. The article advocates the substitution of the selection dilemma with an approach where the modeling method is agilely tailored for the semantic variability required to cover both run-time and design-time concerns. The semantic space enabled by such a method is exposed to model-driven systems as RDF knowledge graphs, whereas the method evolution is managed with the Agile Modeling Method Engineering framework. The argumentation is grounded in the application area of Product-Service Systems, illustrated by a project-based modeling method.

3. Combining Model Inference and Passive Testing in the Same Framework to Test Industrial Systems

The last article proposes a framework called Autofunk to test production systems by combining two approaches: model generation and passive testing. Autofunk combines the notions of expert system, formal models and machine learning to infer symbolic models, while preventing over-generalization, from the large set of events collected from a production system. Afterwards, these models are considered to passively test whether another system is conforming to the models. As the generated models do not express all the possible behaviors that should happen, the article defines conformance with four specialized implementation relations.

António Miguel Rosado da Cruz
Guest Editor
IJISMD

REFERENCES

Cruz, A. M. (2015). Use Case and User Interface Patterns for Data Oriented Applications. In S. Hammoudi, L.F. Pires, J. Filipe et al. (Eds.), *Model-Driven Engineering and Software Development, 2nd International Conference, MODELSWARD 2014*, Lisbon, Portugal, CCIS (Vol. 506, pp. 117-133). Springer International Publishing. doi:10.1007/978-3-319-25156-1_8

Frankel, D. S. (2003). *Model Driven Architecture – Applying MDA to Enterprise Computing*. Indianapolis, Indiana: Wiley Publishing, Inc.

Sommerville, I. (2015). *Software Engineering. Addison-Wesley* (10th ed.). Pearson Education.

Warmer, J., Bast, W., Pinkley, D., Herrera, M. and Kleppe, A. (2003). MDA Explained - The Model Driven Architecture: Practice and Promise. Addison-Wesley Professional, 2003.