Softw Syst Model (2019) 18:7–10
https://doi.org/10.1007/s10270-017-0589-6

GUEST EDITORIAL

Editorial to theme issue on model-driven engineering of component-based software systems

Federico Ciccozzi[1] · **Jan Carlson**[1] · **Patrizio Pelliccione**[2,3] · **Massimo Tivoli**[4]

Received: 16 February 2017 / Accepted: 20 February 2017 / Published online: 14 March 2017

Abstract This theme issue aims at providing a forum for disseminating latest trends in the use and combination of model-driven engineering (MDE) and component-based software engineering (CBSE). One of the main aims of MDE is to increase productivity in the development of complex systems, while reducing the time to market. Regarding CBSE, one of the main goals is to deliver and then support the exploitation of reusable "off-the-shelf" software components to be incorporated into larger applications. An effective interplay of MDE and CBSE can bring benefits to both communities: on the one hand, the CBSE community would benefit from implementation and automation capabilities of MDE, and on the other hand, MDE would benefit from the foundational nature of CBSE. In total, we received 23 submissions to this theme issue, and each submission was reviewed by at least three reviewers. Thanks to the high quality of the submissions that we received, we could eventually accept six papers for publication.

✉ Federico Ciccozzi
federico.ciccozzi@mdh.se

Jan Carlson
jan.carlson@mdh.se

Patrizio Pelliccione
patrizio.pelliccione@gu.se

Massimo Tivoli
massimo.tivoli@univaq.it

[1] Mälardalen University, Västersås, Sweden

[2] Chalmers University of Technology, Gothenburg, Sweden

[3] University of Gothenburg, Gothenburg, Sweden

[4] University of L'Aquila, L'Aquila, Italy

1 Model-driven engineering

Model-driven engineering (MDE) is an established methodology to increase productivity of complex systems while reducing the time to market. MDE enables and suggests a shift from code-centric approaches to a more human-centric development, where models represents artifacts closer to human understanding that can be programmatically read and exploited for simplifying the design, implementation, and execution of software systems.

MDE permits to systematically concentrate on different levels of abstractions, each providing a view for specific stakeholders, for instance (i) improving usability, (ii) enabling customizability in different and specific domains, (iii) promoting reusability of the different algorithms, methods, and techniques, (iv) managing variability and complexity both at design-time and run-time, and (v) managing qualities like evolvability, changeability and configurability, modifiability, scalability, power consumption, and dependability.

Models and model transformations are the core development artifacts in MDE.

Models are defined with concepts that are much less bound to the underlying implementation technology and are closer to the problem domain. This makes the models easier to specify, understand, and maintain, helping the understanding of complex problems and their potential solutions through abstraction. More specifically, MDE promotes a shift from code written in third generation programming languages to models expressed in domain-specific modeling languages (DSMLs).

DSMLs use metamodels to define the modeling concepts, as well as the relations them, and their semantics. A metamodel is an abstraction that highlights the characteristics of well-formed models, which are said to conform to their

metamodel like a program conforms to the grammar of its programming language.

In MDE, development steps are automated by model transformations. A model transformation produces a non-empty set of target artifacts (i.e., models and/or text) from a non-empty set of source models. For example, by focusing on the software architecture domain, practitioners might take advantage of model transformations in order to obtain program code, alternative model descriptions, deployment configurations, inputs for analysis tools from their software architecture models. An important distinction of model transformations is that a model can be transformed either horizontally or vertically. Horizontal transformation means that the source model is transformed into a model or another type of artifact at the same level of abstraction. Vertical transformation means that the source model is transformed into a model or another type of artifact at another level of abstraction. It is important to highlight that the definition of model transformations is considered to be a one-time effort done at metamodeling level; therefore, practitioners act as users of model transformations, which are defined by MDE tool providers and domain experts.

2 Component-based software engineering

Component-based software engineering (CBSE) originally emerged as a software discipline to deliver reusable "off-the-shelf" software components to be incorporated into larger applications. The main focus has been on effective and general-purpose reuse of components within a large variety of different applications. Nowadays, especially with the increasing development of cyber physical systems (CPSs) and the Cloud, CBSE continues to attract interest and evolve as a software approach/methodology for the rapid and dynamic or "on-the-fly" assembly of flexible software systems.

From the original design needs, mostly focused on promoting effective and efficient[1] reuse of available third-party pieces of software, the attention of software engineers moved toward the definition of approaches and development of methods to add, remove, replace, modify, and assemble components dynamically, during operation. For instance, in the domain of CPSs, the strong connection between the computational and physical entities has been recognized, leading to the development of hybrid component frameworks. Such frameworks aim to be capable to take into account and reason on both the event-based and discrete properties of computational entities and the time-based and continuous properties of physical entities.

From facing challenges introduced by the limitations of the previously leveraged object-oriented technologies, such as loose coupling, independent software reuse, seamless integration of heterogeneous software, and so on and so forth, CBSE evolved, and indeed is still evolving, to address issues related to support the dynamicity, high interaction, and safety and dependability concerns of modern software systems. Very often, this led to rethinking widely adopted CBSE development processes to relax the traditional division among development phases by moving some activities from design-time to deployment- and run-time. In this new and more dynamic development settings, the use of models at run-time has been found a key factor. In this direction, recent research focused on the definition of novel software development processes and methods to build highly dynamic and evolvable component-based systems.

The kind of systems targeted ranges from component-based systems structured through component-connector styles to service-oriented and thing-based systems composed by means of either orchestration or choreography. For both kinds of systems, MDE technologies are deeply exploited, as well as the usage of models at run-time, to support analysis and automated synthesis methods for the production of the correct (with respect to functional and extra-functional properties) component/services integration and coordination logic. These recent streams of research show that the interplay of MDE and CBSE is becoming ever more important to address the complexity and high dynamicity of modern software systems, and their dependability as well.

3 Interplay of MDE and CBSE

MDE and CBSE can be considered as two orthogonal ways of reducing development complexity: the former shifts the focus of application development from source code to models in order to bring system reasoning closer to domain-specific concepts; the latter breaks down the set of desired features and their intricacy into smaller sub-modules, called components, from which the application can be built-up and incrementally enhanced.

When exploiting these development approaches, numerous different modeling notations and consequently several software models may be involved during the software life cycle, from requirements to specification, from analysis to code. On the one hand, effectively dealing with all the heterogeneous modeling notations that describe software systems needs to bring component-based principles at the level of the software model landscape. This is achieved by supporting, e.g., the specification of model interdependencies, and their retrieval, as well as enabling interoperability between the different notations used for specifying the software. On the other hand, MDE techniques can bring to the CBSE process

[1] With respect to time-to-market, integration costs, quality attributes, etc.

the possibility to effectively reuse and integrate third-party model entities as well as to boost automation in the development process through powerful model transformations.

An effective interplay of CBSE and MDE approaches would bring benefits to both research communities. On the one hand, the research results of CBSE would benefit from the implementation and automation capabilities of MDE. This will permit to scale up best practices of CBSE to large-scale systems. On the other hand, MDE would benefit from the foundational nature of CBSE approaches. Summarizing, an effective interplay of CBSE and MDE approaches could help in handling the intricacy of modern software systems, thus reducing costs and risks by: (i) enabling efficient modeling and analysis of functional as well as extra-functional properties such as, for instance, safety, reliability, availability and dependability, (ii) improving reusability through the definition and implementation of components loosely coupled into assemblies, (iii) providing automation where applicable (and favorable) in the development process. In the last fifteen years, such a cooperation has been covered by a large number of works and recognized as extremely promising; tools and frameworks have been developed for supporting this kind of integrated development process.

Nevertheless, when exploiting interplay of MDE and CBSE, clashes arise due to misalignments in the related terminology but also, and more importantly, due to differences in some of their basic assumptions and focal points.

4 In this issue

The papers in this issue address a wide range of challenges related to the combination or interplay of component-based and model-driven software engineering, but there are also themes that appear in several of the papers, including modeling and ensuring extra-functional properties, in particular timing, code generation, and model transformations.

In *A Framework to Specify System Requirements using Natural Interpretation of UML/MARTE Diagram*, Aamir M. Khan, Frédéric Mallet, and Muhammad Rashid propose a framework where early informal natural language component specifications can be represented in a graphical form with well-defined semantics to reduce ambiguity. The graphical formalism, based on UML and MARTE, is complemented by a set of primitive property patterns providing support for correctly expressing common temporal properties. The framework is implemented in an Eclipse-based tool in the form of a model transformation plugin with analysis and code generation capabilities.

In *Supporting Timing Analysis of Vehicular Embedded Systems through the Refinement of Timing Constraints*, Saad Mubeen, Thomas Nolte, Mikael Sjödin, John Lundbäck, and Kurt-Lennart Lundbäck, address the representation of temporal specifications from another perspective. Based on the EAST-ADL and TADL2 modeling languages and the Rubus Component model, they describe how timing information can be refined between abstraction levels and how the underlying component model can be exploited to specify additional timing relevant details. The approach allows end-to-end timing analysis at a higher abstraction level and in earlier development phases.

In *Synthesis of Verifiable Concurrent Java Components from Formal Models*, Julio Mariño, Raúl N. N. Alborodo, Lars-Åke Fredlund, and Ángel Herranz present a methodology for semi-automated generation of concurrent Java components from high-level models of component behavior and interaction. They describe and compare three realization alternatives: one based on synchronized methods, one using priority monitors to achieve a more fine-grained control over the synchronization and one based on a third-party message passing library. Moreover, they describe the integration of the proposed approach with state-of-the-art program verification techniques, by which the correctness of the code templates can be ensured.

In *Multi-Objective Exploration of Architectural Designs by Composition of Model Transformations*, Smail Rahmoun, Asma Mehiaoui-Hamitou, Etienne Borde, Laurent Pautet, and Elie Soubiran propose an automated approach that allows the construction of new model transformations as a composition of existing ones. The approach is used in the domain of real-time embedded systems, where the early analysis of Extra-Functional Properties (EFPs) is of paramount importance. In this context, the early validation of EFPs often results in facing a multi-objective optimization problem with a very large number of potential solutions, each of them corresponding to a model transformation alternative. To overcome the issue to write all the possible model transformation alternatives, the proposed approach is used to automate the production of architectural alternatives, each of them fulfilling specific EFPs. Each alternative is produced by following a component-based approach where the basic components are basic transformation alternatives and added value architectural alternatives are built by performing a more complex transformation obtained as a composition of the basic ones. This approach allows us to deal also with conflicting EFPs, hence making the process of finding a suitable architectural solution tractable. The approach has been prototyped in RAMES, and the experimental results shown in the paper are promising.

In *Using Internal Domain-Specific Languages to inherit Tool Support and Modularity for Model Transformations*, Georg Hinkel, Thomas Goldschmidt, Erik Burger, and Ralf Reussner analyze important issues related to the adoption of model transformation in industry. In particular, the conducted study aims to establish: (i) what type of tool support can be reused from model transformation languages implemented

as internal DSLs; (ii) the impact of embedding model transformation concepts into object-oriented ones on the quality of the derived tool support; and (iii) how to reuse modularization constructs from object-oriented languages in model transformation languages. The study has been carried on by considering the NET Modeling Framework Transformation Language.

In *Transactional Execution of Hierarchical Reconfigurations in Cyber-Physical Systems*, Christian Heinzemann, Steffen Becker, and Andreas Volk present an extension of the MechatronicUML component model conceived to support safe reconfiguration of encapsulated hierarchical component-based CPSs. The proposed solution allows us to deal with both real-time and ACI properties and allows the preservation of the continuous nature of the physical system environment by introducing feedback controller components. That is, reconfiguration can be performed without interrupting the execution of the system. For CPSs, this is a key aspect. The effectiveness of the approach is shown by applying it at work on a RailCab system.

Acknowledgements We would like to thank all authors who submitted papers, and the reviewers for their efforts and high quality reviews. Finally, we would like to thank Martin Schindler for his excellent support throughout the process of putting together this theme issue.