Tailored Quality Modeling and Analysis of Software-intensive Systems

Robert Heinrich
Karlsruhe Institute of Technology
Karlsruhe, Germany
robert.heinrich@kit.edu

***Abstract*—While developing and operating software-intensive systems various concerns must be considered. Concerns like quality properties, domain, and lifecycle phase may differ from one project to another. Currently different languages and tools are required for modeling and analyzing these concerns. This results in enormous effort for model creation and maintenance. In this paper, we present a vision of tailored quality modeling and analysis by clearly separating several concerns using modular metamodels and tooling.**

***Index Terms*—domain-specific modeling language, metamodel, simulation, analysis, quality, reference architecture**

I. Introduction

Mobility, energy, and infrastructure strongly depend on software which is not always of high quality. Critical performance or security issues may arise from bad software quality. During development and operations of a software-intensive system, various concerns must be considered. Quality properties are examples of concerns which may differ from one project to another. For instance, performance is highly relevant for a web shop whereas for a storage service security may be more relevant. In the context of cloud computing and data-intensive systems, quality properties like privacy can only be assessed reasonably during operations which makes development analysis widely unrewarding. Further, software interacts with other domains (e.g., in Industry 4.0 scenarios), like a business process or mechanics and electronics of an automated production system, where only a subset of domains may be relevant for a certain software project.

For representing a software-intensive system in form of a model, a modeling language is required. Modeling languages are often defined through a metamodel. A metamodel is a model which defines the structure and characteristics of other models. If a model conforms to a metamodel, the model is considered an instance of the metamodel. Thus, a metamodel is similar to a grammar, as it defines a language.

Currently, developers and operators apply different modeling and analysis tools for each of the concerns. Each tool requires specific input models of different languages. Hence, the input models are not integrated and require enormous manual effort for creation and maintenance. Quality is mostly not represented in a domain-specific modeling language (DSML). The commonly agreed DSML in software engineering, the Unified Modeling Language (UML) [20], does not consider quality properties. Even its extensions, for example MARTE [19] or UMLSec [16], are restricted only to single quality properties. Languages and tools that comprise all possible concerns would be very large, unhandy, and hard to maintain.

DOI reference number: 10.18293/SEKE2018-020

The vision proposed in this paper targets more flexibility in Model-Driven Engineering (MDE) by using MDE adaptation capabilities to tailor DSMLs and related tooling to specific concerns. The paper addresses the concerns quality property, domain, and lifecycle phase. We aim for improving efficiency, scalability and reuse of modeling and analysis approaches by clear separation of concerns, focusing the modeling effort only on the relevant concerns, and enabling easy tool customization. Sec. II presents a motivating example before the state of the art is discussed in Sec. III. Our vision of tailored modeling and analysis of various concerns is proposed in Sec. IV and applied prototypically in Sec. V. The paper concludes in Sec. VI.

II. Motivating Example

We introduce the Common Component Modeling Example (CoCoME) as a typical software system to exemplify different configurations that may result from several concerns. We also introduce a DSML to demonstrate limitations in current modeling approaches for various concerns. CoCoME is a community case study for software architecture modeling [12], [8]. It resembles a trading system of a supermarket chain. A software system like CoCoME has to consider several concerns. Several quality properties must be satisfied such as performance, reliability, maintainability, security and privacy. CoCoME interacts with other domains – a business process and an automated production system. Software systems are typically involved in one or more business processes to satisfy business goals. In an Industry 4.0 scenario, software systems interact with automated production systems to enable customized production. An automated production system consists of software, mechanical, and electrical components. Another concern that must be considered is the system's lifecycle phases, i.e. development and operations [7]. Fig. 1 depicts three different configurations of concerns for CoCoME.

For representing CoCoME and the associated concerns in form of models we need a metamodel of a DSML. A prominent example is the UML metamodel. Another historically grown metamodel is the Palladio Component Model (PCM) [23] which we choose for modeling and analyzing CoCoME.

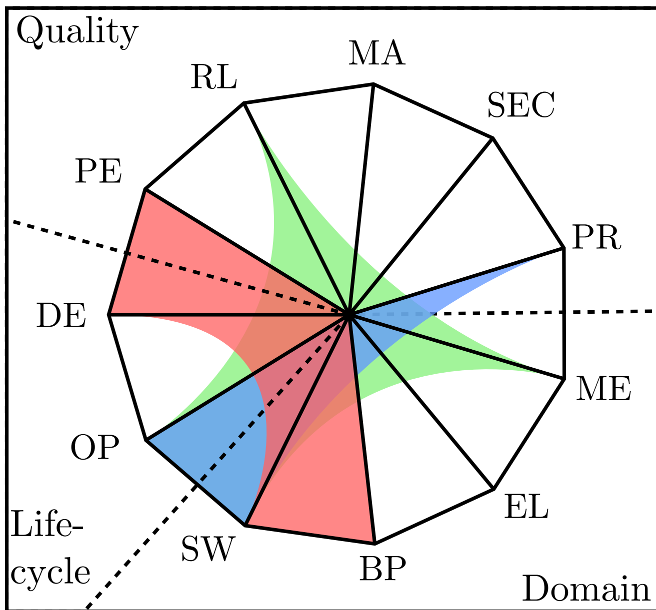

Fig. 1. Colors Represent Three Different Configurations of Concerns for Software-intensive Systems. Concerns Depicted in the Figure are Performance (PE), Reliability (RL), Maintainability (MA), Security (SEC), Privacy (PR), Mechanics (ME), Electronics (EL), Business Processes (BP), Software (SW), Operations (OP), and Development (DE)

In contrast to UML, the PCM comprises elements for reflecting analysis configurations and results. However, the PCM is limited to modeling and analyzing software systems. So we cannot model business processes or automated production systems with the original Palladio approach. Extensions of the PCM to model business processes have been proposed in [9]. Extensions for modeling mechanics and electronics are not yet planned.

Analyzing the various quality properties related to CoCoME currently requires several tools. They all need specific input models in different languages. For example, Queuing Petri Nets may be applied for performance analysis. Markov chains are used for analyzing reliability. Logic programming languages like Prolog may be used for privacy constraint checking. Consequently, high effort is required to create the different models while large parts of the models represent the same structure and behavior in different languages. Moreover, the single models are hard to compare which results in inconsistencies. The example shows existing modeling capabilities are not sufficient for various concerns. Putting all concerns into a large metamodel is just as little a solution. To prevent large and blown metamodels and enable tailoring the models and tools to specific concerns a more modular specification and composition of metamodels and related tooling is required.

III. State of the Art

Our work has intersections with three research areas – metamodel modularization, quality modeling, and runtime modeling.

Existing approaches to *metamodel modularization* do not consider architectural characteristics for modularization and do not support the interrelation between metamodel modules and tooling. The underlying model in [1] captures all concerns into orthogonal dimensions which are accessed through views. CORE [26] specifies flexible software modules for model-based and concern-oriented software reuse. Melange [5] and MontiCore [18] reuse and customize legacy modules for creating new languages. An open topic is to provide modularization concepts for DSMLs in direct interplay with related tooling. First attempts came up for modular transformations [22] and generators [15].

Quality modeling currently considers only single quality properties in metamodels and lack configurability. Literature reviews give an overview of quality modeling approaches (e.g., [17], [4]) for software systems. Quality properties are also addressed in other domains like business processes (e.g., [3]) and automated production systems (e.g., [30]). In our previous work, we developed an approach for modeling and analyzing maintainability [24]. This approach was first restricted to software systems. The approach has been extended for modeling and analyzing maintainability of business processes [25] and automated production systems [30]. Finally, the approach has been generalized by providing a methodology for domain-spanning maintainability analysis [11]. The promising potential of a generic approach for modeling and analyzing quality has already been recognized in earlier publications [21]. First approaches for generic quality modeling came up decades ago (e.g., [6]). They lack predictive analysis as formal specifications are missing. Formal specifications of quality have been proposed e.g. in [31]. Context-independent modeling of quality is described in [14]. An open topic is using commonalities between quality properties to make quality-related annotations in metamodels configurable.

Runtime modeling distinguishes approaches for reusing development models as foundation for reflecting systems during operations and approaches for model extraction from scratch using observations [10]. A comprehensive review of runtime modeling approaches is given in [29]. An open topic is reconfiguring models to changing concerns during operations while keeping all development decisions.

Further, configuration and reuse are central to *software product lines and eco systems*. While this research is limited to the instance level, our work refers to the metamodel level.

IV. A Vision of Tailored Quality Modeling and Analysis

Our vision is to provide a reference architecture for metamodels that enables clear separation of several concerns in quality modeling and analysis. Beyond considering different quality properties the reference architecture comprises multiple domains related to software (i.e., business processes, mechanics, and electronics), and life-cycle phases (i.e., development and operations). Modelers will be enabled to customize their tooling by composing modular specifications for each of the concerns as desired for their project and run

transformations to create model editors and solvers. Prior publications (cf. Sec. III) already raised the need for a generic approach to quality modeling and analysis. A reference architecture for metamodels for quality modeling and analysis is an ambitious goal. In the past it was assumed to be unattainable as quality properties were considered to be too different in their relation to system's architecture and context. Consequently, a generic approach was assumed to be too abstract for adequate predictive analysis. Recent advances, however, lead to the conclusion that the challenges can now be overcome as: (i) Research on formalization of single qualities (e.g., [2]) resulted in much deeper understanding of similarities which we can use for the reference architecture. (ii) Research on mutual quality impact between different domains (e.g., [9]) provides starting points for specification of quality-dependent inter-domain relations. (iii) A first reference architecture for single qualities in a DSML for software systems [27] is starting point to more generic investigation for different concerns. (iv) Research on simulation-based quality analysis (e.g., [23]) is foundation for a general approach to generic tooling. The vision requires innovation in several areas as detailed hereafter.

Metamodel Modularization and Composition: Foundation to the reference architecture is a concept for composition of modular metamodels. If modular metamodels are not already available, we first need to identify a set of dimensions and elaborate criteria along which metamodels can be modularized. One way is to specify dimensions based on characteristics of metamodels. As we focus on metamodels for quality modeling and analysis in various domains, dimensions like paradigm, domain, quality specification, and quality analysis appear natural. Prior work on dimensions is given in [27]. Then, existing model extension mechanisms (e.g., inheritance, profiles, stereotypes, and aspects) for composing metamodel modules created along the dimensions can be investigated and mapped to composition operators. The operators are used to build composable tooling by running transformations. Finally, we can construct the reference architecture by exploiting the concepts for metamodel modularization and composition.

Extensible DSML: The reference architecture then enables the structured extension of existing DSMLs (e.g., PCM [23]) by comprehensive specifications of quality properties. In a software DSML, systems are typically described as compositions of components by connectors made explicit through interfaces. These generic modeling concepts can be extended to support the specification of various quality properties and their context dependencies. Using modularization and composition capabilities eases the extension of a software DSML to the domains business process and automated production system by composing the corresponding modular metamodels. Further investigation is necessary on how the fundamental concepts of a DSML (i.e., composition and connectors) known from software modeling can be applied to other domains.

Reconfiguration for Operational Concerns: While building upon the extensible DSML we can reconfigure development models to changing concerns during operations while keeping all development decisions. Reconfiguration cannot be limited to DSMLs and tools but also affects monitoring probes required to observe the quality properties in the changed focus when running the system.

Modular Tooling: For providing tools tailored to specific concerns, the notion of modular metamodels for several concerns must be expanded to modular construction of modeling and analysis tools. Aforementioned concepts for metamodel modularization and composition are foundation for modular tooling. Explicitly specified dependencies between quality properties of several domains mark the points where different analytical and simulative solvers need to interact. Composable tools for modeling and analysis can be built using the composition operators. Foundations already exist for coupling simulative solvers [9].

V. Prototypical Application

This section gives concrete examples for metamodel modularization and composition to demonstrate the applicability of the reference architecture and tool modularization.

Application of the Reference Architecture: The PCM in its current form is focused on single quality properties yet does not reflect the various concerns related to our motivating example (cf. Fig. 1). For each concern we must provide specific metamodels as extensions to the PCM. For tailoring the DSML to specific concerns, the PCM and its extensions are divided into four dimensions – paradigm, domain, quality, and analysis – along which the metamodels can be modularized. Previous work limited to metamodels of software systems is given in [27] and [10].

The four dimensions have been chosen as (i) they represent generic characteristics of a DSML for quality modeling and analysis. (ii) Their hierarchical nature eases the composition of metamodels assigned to the single dimensions. In the reference architecture the modularization dimensions are represented as layers ordered hierarchically with respect to their dependencies. Metamodel modules of one layer may only depend on modules of the same or more foundational layers.

The modularization of the PCM according to the reference architecture is depicted prototypically for the domains software system and business process as well as for the quality properties maintainability and performance in Fig. 2. The figure shows a very simplified representation of the modular PCM for visualization in this paper. Each rectangle reflects a modular metamodel that may extend another metamodel and can itself be extended. Arrows depict general relationships between metamodel modules. Within a certain module the relationship is implemented by composition operators between the metaclasses of one and another module.

The paradigm layer (Π) defines the foundational concepts without any semantics, e.g. componentization. Here the component module and the activity module specify the core entities to describe structure and behavior. Composition by connectors is specified for both in the component composition and activity composition module respectively. The data module provides foundational concepts for specifying data flows such as source and sink.

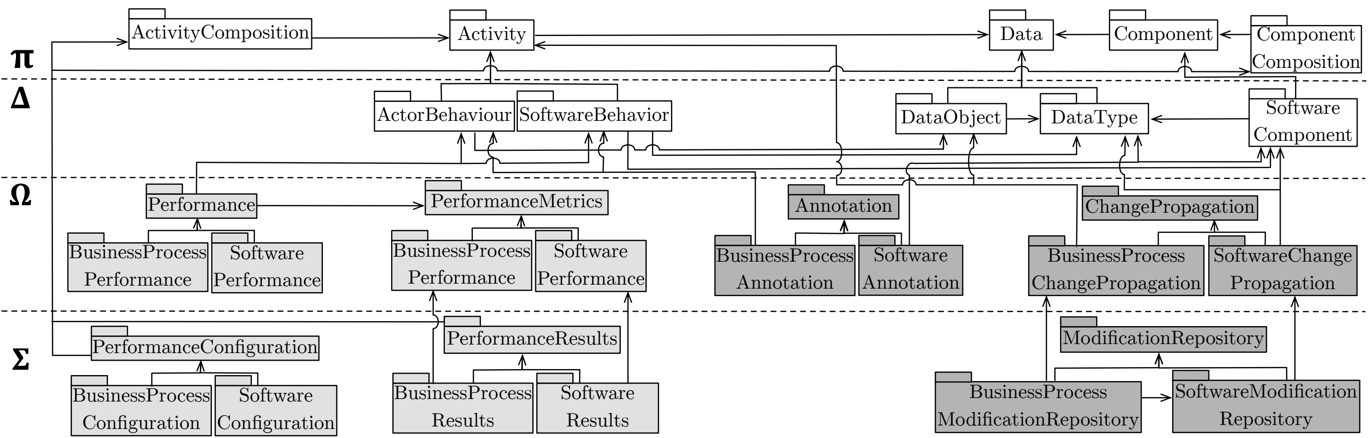

Fig. 2. Prototypical Modularization of the PCM for Different Domains and Quality Properties

The domain layer (Δ) extends Π by domain-specific semantics. Here the software components and their interfaces are specified within the software component module. The software behavior module extends the software component module by a behavior abstraction for software similar to flow charts. A business process on an atomic level comprises activities either conducted by human actors or machines (here the software system) [9]. To specify a business process, the software behavior module and the actor behavior module extend the activity module. Thus, the software behavior module serves as the connecting link between business process modeling and software modeling. Moreover, the data object and data type modules extend the generic data module by specifications of data used in business processes and software services. The modules on Δ layer are used in subsequent layers for performance and maintainability modeling.

The quality layer (Ω) defines quality properties, primarily in form of second class entities which enrich the first class entities of Δ. Quality properties on Ω are not derived by analysis [27]. Here, Ω comprises modular metamodels to reflect performance (light grey) and maintainability (dark grey) properties. The performance module comprises metamodeled performance properties which are specialized for software and business processes in the extending modules. For example, the software performance module specifies resource demands of a service while the business process performance module reflects execution time of a human activity [9]. The same structure applies to the performance metrics module which comprises abstract metrics to be specialized for software and business processes.

The annotation module contains abstract specifications of artifacts annotated to first class entities on Δ. The extending modules specialize these artifacts, e.g. test cases for software components [24] or training material for human activities [25]. Maintainability analysis in a PCM instance is conducted as change propagation analysis using the KAMP approach [24]. Once a component or activity changes the test cases or training material may change, too. Starting with a seed modification changes propagate through instances of the entities on Δ which may change instances of the entities on Ω. Elements to specify this change propagation are contained in the change propagation module and specialized by the extending modules.

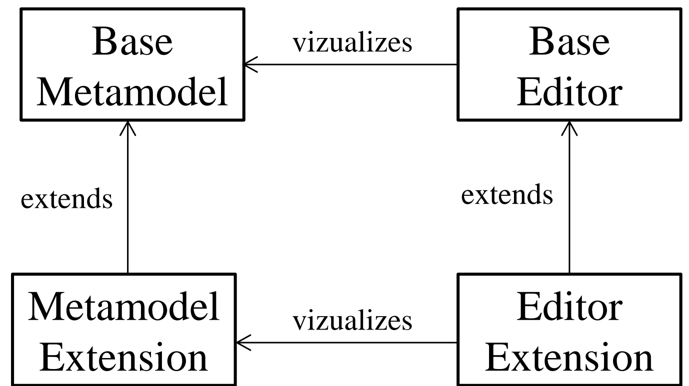

Fig. 3. Modular Base Editor and its Extension

The analysis layer (Σ) is required if models are used for analysis or simulation. It comprises metamodels for derived performance properties in the results module and the simulation configuration in the configuration module. The modification repository module reflects the origin of a change and the result of a change propagation analysis.

Modularizing the PCM according to the reference architecture is key for modularizing the related tooling as discussed hereafter. Again we use the example of software systems and business processes for demonstration.

Application of Modular Tooling: The modularized PCM and all its extensions form a directed acyclic graph. The tooling and all its extensions must mirror this structure. Next, we discuss how model editors, simulative and analytical solvers – that are closely related to metamodels – can be implemented for modular metamodels.

Model editors allow for visualizing and modifying model elements and thus are obviously related to metamodels. For exemplifying the implementation of *modular editors* we build

upon the eclipse modeling framework and the Sirius editor framework. Details on the implementation of the editors for the Palladio tooling are given in [28]. The implementation of modular editors is depicted schematically in Fig. 3. Rectangles represent modular metamodels and modular tools. Arrows reflect relationships between the modules or instances of the modules. The relationships are labeled to further specify the sort of relationship. Sirius offers the possibility to extend a diagram representation by further layers without altering the implementation of the base diagram intrusively. Analogously to a modular base metamodel and its extensions, editor extensions are packaged in their own Eclipse plugins. Applying the reference architecture, modelers can customize their tooling by selecting plug-ins for the specific metamodel modules and corresponding editors. Base metamodel in our example is those for software systems which is composed of metamodel modules on the layers Π to Δ of the reference architecture. The base editor comprises all features required for visualizing and modifying elements of software systems. The base metamodel is extended by metamodel modules on the layers Π to Δ for representing business processes. Furthermore, metamodel modules on Ω layer extend the base metamodel by performance and maintainability properties. Therefore, modular editors for business processes and the quality properties performance and maintainability extend the base editor.

More sophisticated solutions are required for modular simulations. For *modular simulations* we sketch an online co-simulation of a software system and a business process in Fig. 4. We conduct a simulation of performance properties as common in Palladio [23]. A discussion on benefits and limitations of online co-simulation is given in our previous work [9]. We have instances of two metamodels – one for the software system and one for the business process – that are composed of metamodel modules on the layers Π to Δ. Additionally, the metamodels contain metamodeled performance properties and metrics on Ω. Each metamodel has its specific modular tooling – the simulative solvers – which are interlinked by a coordinator. The coordinator is responsible for time management and model synchronization to coherently integrate the modular simulations. Simulation configuration and results are specified on Σ. The simulative solvers assure technical interoperability by providing an interoperability layer for enabling the coordinator to interact with the simulations. There already exists approaches to couple simulations based on a common runtime infrastructure, e.g. High-Level Architecture [13], which can be applied to build a coordinator.

Coupling *modular maintainability analyzes* for software systems and business processes again requires modular metamodels and analysis tools. In contrast to simulation coupling, there is no coordinator needed for synchronization. Consistent notion of time is not required in modular maintainability analyzes. Fig. 5 shows a metamodel for software systems composed of modules on layers Π to Δ. The metamodel also contains modules for artifact annotations and change propagation on Ω as well as for analysis results [24] on Σ. Instances of

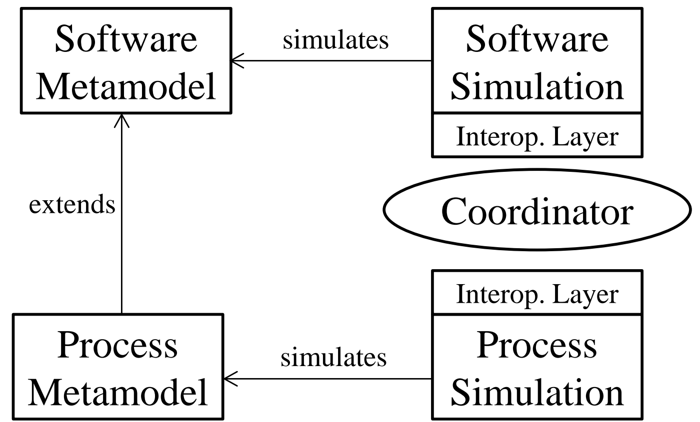

Fig. 4. Modular Performance Simulation of Software System and Business Processes

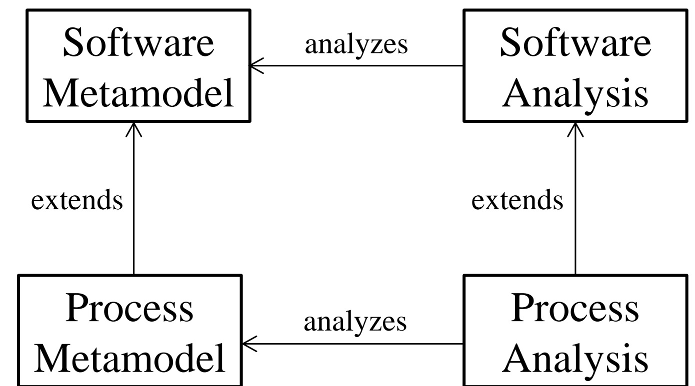

Fig. 5. Modular Maintainability Analyzes of Software Systems and Business Processes

this metamodel are analyzed by the modular software analysis tool. The metamodel for software systems may be extended by another metamodel to reflect business processes on Π to Δ. This again contains modules for annotations on Ω and analysis results on Σ. Instances of the business process metamodel are analyzed by the modular business process analysis tool which extends the software analysis tool. Due to the modular nature we can decide whether to include the metamodel modules specific to business processes or not. This allows for tailoring metamodels and tooling to specific purposes.

Reconfiguring development models to changing concerns during operations is also part of our research vision. With the iObserve approach [7] we provide first attempts for updating development models by operational observations to construct runtime models. iObserve bridges the divergent levels of abstraction in architectural models used in development and operations. An architectural model is combined with monitoring probes and used for source code generation. Monitoring data related to source code artifacts is gathered during system operation and is associated with architectural runtime model elements. Thus, iObserve allows for reusing development models during operation phase while preserving design decisions.

The iObserve approach is described in detail in [7]. Metamodel modularization and composition in this context is sketched in [10].

Findings: The prototypical application of our modularization and composition concepts for metamodels and tooling revealed several findings as summarized hereafter. Based on the PCM it is easy to see that a modularized metamodel according to our reference architecture provides many benefits compared to a metamodel without modular structure. Modularization offers a well specified base for extension as demonstrated for business processes and quality properties. Due to the hierarchical structure, modelers merely need to understand the modules they directly or indirectly extend. Side effects of changes to modules are minimized. Thus, clear separation of concerns allows modelers to focus modeling effort only on concerns relevant for a specific project. The greatest advantage for our research is that metamodel modules can be reused in different contexts. This allows for tailoring the metamodels to specific concerns. Tools can be customized easily for specific metamodels as exemplified for model editors, simulative and analytical solvers. Thus, the concepts proposed in this paper contribute to more efficient, scalable and reusable modeling and analysis of various concerns.

VI. Conclusion

In order to enable clear separation of several concerns we proposed a reference architecture for metamodels used for quality modeling and analysis. We sketched modular tooling for model editors, analytical and simulative solvers. We demonstrated the applicability of the reference architecture and modular tooling to a historically grown metamodel for software systems. Our prototypical application comprised the quality properties performance and maintainability for the two domains. Advantages of modular metamodels and tooling are a well structured base for metamodel extension, minimization of side effects in case of modification, focus only on relevant concerns, reuse of modular metamodels and tools as well as project-specific configuration of metamodels and tools.

In the future, we will continue the modularization of existing metamodels and related tooling in several case studies to evaluate, expand and sharpen our approach. This includes further investigation of technologies for metamodel composition and simulation coupling.

Acknowledgement

This work was supported by the MWK (Ministry of Science, Research and the Arts Baden-Württemberg, Germany) in the funding line Research Seed Capital (RiSC). The author thanks Kiana Busch, Misha Strittmatter and Sandro Koch for valuable discussion and support for this paper.

References

[1] C. Atkinson *et al.*, “Orthographic software modeling: A practical approach to view-based development,” in *Evaluation of Novel Approaches to Software Engineering*, vol. 69. Springer, 2010, pp. 206–219.

[2] S. Becker *et al.*, “Towards a methodology driven by dependencies of quality attributes for QoS-based analysis,” in *ICPE*. ACM, 2013, pp. 311–314.

[3] J. Cardoso *et al.*, “Modeling quality of service for workflows and web service processes,” *Journal of Web Semantics*, vol. 1, pp. 281–308, 2002.

[4] L. Dai and K. Cooper, “A survey of modeling and analysis approaches for architecting secure software systems,” *Journal of Network Security*, vol. 5, pp. 187–198, 2007.

[5] T. Degueule *et al.*, “Melange: A meta-language for modular and reusable development of DSLs,” in *SLE*. ACM, 2015, pp. 25–36.

[6] S. Frolund and J. Koistinen, “QML: A language for quality of service specification,” Hewlett-Packard Software Technology Laboratory, Tech. Rep. HPL-98-10, 1998.

[7] R. Heinrich, “Architectural run-time models for performance and privacy analysis in dynamic cloud applications,” *Perform. Eval. Rev.*, vol. 43, no. 4, pp. 13–22, 2016.

[8] R. Heinrich *et al.*, “A platform for empirical research on information system evolution,” in *SEKE*, 2015, pp. 415–420.

[9] ——, “Integrating business process simulation and information system simulation for performance prediction,” *Software & Systems Modeling*, vol. 16, no. 1, pp. 257–277, 2017.

[10] ——, *Software Architecture for Big Data and the Cloud*. Elsevier, 2017, ch. An Architectural Model-Based Approach to Quality-aware DevOps in Cloud Applications.

[11] ——, “A methodology for domain-spanning change impact analysis,” in *SEAA*. IEEE, 2018.

[12] S. Herold *et al.*, “CoCoME – the common component modeling example,” in *The Common Component Modeling Example*. Springer, 2008, pp. 16–53.

[13] IEEE, “Standard for modeling and simulation High Level Architecture,” 2000.

[14] K. Jezek *et al.*, “Towards context independent extra-functional properties descriptor for components,” *Electron. Notes Theor. Comput. Sci.*, vol. 264, no. 1, pp. 55–71, 2010.

[15] R. Jung *et al.*, “GECO: A generator composition approach for aspect-oriented DSLs,” in *ICMT*. Springer, 2016, pp. 141–156.

[16] J. Jürjens, “UMLsec: Extending uml for secure systems development,” in *UML*. Springer, 2002, pp. 412–425.

[17] H. Koziolek, “Performance evaluation of component-based software systems: A survey,” *Perform. Eval.*, vol. 67, no. 8, pp. 634–658, 2010.

[18] H. Krahn *et al.*, “MontiCore: Modular development of textual domain specific languages,” in *TOOLS EUROPE 2008*. Springer, 2008, pp. 297–315.

[19] Object Management Group, *UML Profile for MARTE*, Object Management Group Std., 2011.

[20] ——, *Unified Modeling Language, Version 2.4.1*, Object Management Group Std., Rev. 2.4.1, 2011.

[21] D. C. Petriu, “Challenges in integrating the analysis of multiple non-functional properties in model-driven software engineering,” in *WOSP-C*. ACM, 2015, pp. 41–46.

[22] A. Rentschler, *Model Transformation Languages with Modular Information Hiding*. KIT Scientific Publishing, 2015.

[23] R. H. Reussner *et al.*, *Modeling and Simulating Software Architectures – The Palladio Approach*. MIT Press, 2016.

[24] K. Rostami *et al.*, “Architecture-based assessment and planning of change requests,” in *QoSA*. ACM, 2015, pp. 21–30.

[25] ——, “Architecture-based change impact analysis in information systems and business processes,” in *ICSA*. IEEE, 2017, pp. 179–188.

[26] M. Schöttle *et al.*, “On the modularization provided by concern-oriented reuse,” in *Modularity*. ACM, 2016, pp. 184–189.

[27] M. Strittmatter *et al.*, “A modular reference structure for component-based architecture description languages,” in *ModComp*. CEUR, 2015, pp. 36–41.

[28] ——, “Extensible graphical editors for palladio,” in *7th Symposium on Software Performance*, 2016.

[29] M. Szvetits and U. Zdun, “Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime,” *Software & Systems Modeling*, vol. 15, no. 1, pp. 31–69, 2016.

[30] B. Vogel-Heuser *et al.*, “Maintenance effort estimation with KAMP4aPS for cross-disciplinary automated PLC-based production systems - a collaborative approach,” in *20th IFAC World Congress, IFAC-PapersOnLine*, vol. 50, 2017, pp. 4360–4367.

[31] S. Zschaler, “Formal specification of non-functional properties of component-based software systems,” *Software & Systems Modeling*, vol. 9, no. 2, pp. 161–201, 2010.