JOT
JOURNAL OF
OBJECT TECHNOLOGY

A Methodology for Retrofitting Generative Aspects in Existing Applications

Imke Drave, Arkadii Gerasimov, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga
Software Engineering, RWTH Aachen University, Germany

ABSTRACT Using model-based approaches and code synthesis to engineer information systems improves agile application development and evolution. However, current research lacks systematic approaches to integrate generative aspects in already existing applications. Existing approaches consider partial aspects of the engineering process, such as software language engineering or generator engineering. However, an overall approach for the model-based reconstruction of existing systems is missing. We propose a set of activities for retrofitting the model-based approach into already existing enterprise information systems. Using our experience in language engineering as well as previous generative practical realizations of applications, we have developed a methodology with three phases: problem analysis and decomposition, domain-specific language engineering and application engineering and operation. We demonstrate its practical application using a real-world enterprise information system as an example. Using our methodology developers can make structured, informed decisions when retrofitting a model-based approach into enterprise information systems.

KEYWORDS Methodology, Model-Based Software Engineering, Generators, Brown Field, Information Systems, Problem Decomposition.

1. Introduction

1.1. Motivation and Relevance

Modern software systems are pervasive systems of systems that continuously increase in size and complexity (France & Rumpe 2007; Möller et al. 2011), this growth still challenges software engineers today. The gap between the problem(s) solved by a software system and its realization (known as problem-implementation gap (France & Rumpe 2007)) still exists and contributes to the complexity of these systems. Model-Based Software Engineering (MBSE) enables automatic code synthesis and thereby offers solutions to overcome these challenges: Addressing each aspect of a problem that occurs during development through a tailored Domain-Specific Language (DSL) and utilizing models within this language for generative engineering of a software system has proven an effective means to reduce the problem-implementation gap (France & Rumpe 2007; Schmidt 2006). Abstracting from details of the implementation platform and generating code from this abstract description enables agile development and, thus, allows for efficient adaptation to changing requirements. Nevertheless, current research lacks a systematic decision-making procedure, *e.g.,* to choose between the reuse of an existing DSL or the engineering of a new one, to be used in modeling solutions to a problem and retrofitting a running Enterprise Information Systems (EISs) with implementations generated from models. A software language, general-purpose or domain-specific (Fowler 2010; Völter et al. 2013), serves a dedicated purpose (Karsai et al. 2009), has strengths and weaknesses depending on its properties, and is often used during multiple phases of the engineering process. Software engineering for modern EISs is defined by the heterogeneity of development teams and languages used to model and implement the system (Clark et al. 2015). A holistic approach to MBSE that enables exploiting the potential of code generation requires a systematic collaboration of developers (Clark et al. 2015) and explicit relationships between models, *e.g.,* through the composition of languages (Hölldobler et al. 2018) or models (Kienzle et al. 2019; Degueule et al. 2015). Choosing or engineering

JOT reference format:
Imke Drave, Arkadii Gerasimov, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. *A Methodology for Retrofitting Generative Aspects in Existing Applications.* Journal of Object Technology. Vol. 20, No. 2, 2021.
http://dx.doi.org/10.5381/jot.2021.20.2.a7

a modeling language for a specific problem aspect, therefore, requires consideration of the interrelations and overlaps of problem aspects. A methodology that (1) divides the large complex problem domain of an EIS into problem aspects, (2) offers a mechanism to choose an appropriate DSL or General Purpose Language (GPL) to handle problem aspects while regarding their interrelations, and (3) includes aspects of generator engineering as well as regeneration of the system until redeploying an improved EIS, is still missing.

1.2. Research Question and Objective

This paper, therefore, addresses the research question *how to retrofit generative aspects in existing enterprise information systems* and proposes a concrete model-based methodology for reconstructing existing applications (brown-field) through structured and informed decision-making.

The *objective* is to provide a comprehensive guide for software engineers which suggests possible activities for the transformation of a handwritten application to a model-driven application and generator infrastructure.

1.3. Main Contribution

The main contribution of this article is *a methodology for retrofitting generative aspects into existing EISs (brown-field) that can be applied throughout the phases of the development process, i.e., problem analysis, language engineering, and application engineering*.

The methodology uses the divide- and conquer principle to divide large complex problems into smaller problems that can be handled with dedicated modeling languages. It includes three phases, namely (Phase P) problem analysis and decomposition, (Phase L) domain-specific language engineering as well as (Phase A) application engineering and operation. This includes activities such as (1) coping with complex problems, (2) choosing or creating a modeling language and adapting or creating a generator, and (3) continuous regeneration, test, and deployment.

We validate our methodology by example (Shaw 2003) using *a real-world example*: an EIS from the domain of finances and controlling (Gerasimov, Heuser, et al. 2020) with a codebase of more than 500,000 lines of code. As an example for the practical realization of our methodology we use a set of languages created with MontiCore (Hölldobler & Rumpe 2017), a language workbench for the design and realization of textual DSLs. MontiCore allows for language aggregation and composition as well as for generation or synthesis of code from the models via a common infrastructure.

This work builds on our previous research results and the work of other authors, however, the proposed methodology and its application were not published before. The example application (MaCoCo project, Section 4) was already used in our publication about the maintainable co-development of front-end and back-end in generated information systems (Gerasimov, Heuser, et al. 2020). The generator framework MontiGem was published in (Gerasimov, Michael, et al. 2020). It is used to realize activities 9 and 11 in our example application in Section 4. Lessons learned about generator engineering were published in (Adam et al. 2020). Our methodology was developed bottom-up by identifying realizable underlying mechanisms of interactions (analytical generalization (Yin 2003)) between modeling languages, models and their specific characteristics and makes use of our experiences in software language engineering and practical realizations of generative approaches.

Outline. Section 2 describes the fundamentals for the practical application of our methodology, *e.g.,* the used language workbench, the example languages, and the generator framework used for the generation of an EIS. We introduce our running example and discuss challenges for generative software engineering. Section 3 shows our methodology to retrofit generative aspects in existing projects. Section 4 discusses the practical application of the approach on a real-world EIS. Section 5 critically discusses our ideas in relation to other approaches and their limitations and threats to validity. Section 6 presents related work for the different activities. The last section concludes this article.

2. Fundamentals

The *problem-implementation gap* in software occurs whenever developers implement a software solution on a lower level of abstraction than the abstraction level that is used to describe the problem (France & Rumpe 2007). Software systems today are pervasive systems of systems (France & Rumpe 2007) whose components[1] have to fulfill a heterogeneous variety of tasks (Schmidt 2006; Kienzle et al. 2019; Aniche et al. 2019). For example, EISs have to run processes, display a graphical user interface, process data, and assure security at the same time. Simultaneously, these EISs have to be highly adaptable to the ever changing corporate environment (Möller et al. 2011; Schmidt 2006). MBSE classifies the heterogeneous system tasks into problem domains and utilizes modeling languages tailored for implementing solutions in each domain individually. In MBSE, a model is an abstract representation of a system suited to define solutions to a specific set of heterogeneous tasks (Stachowiak 1973). By utilizing models in domain-specific modeling languages as primary development artifacts, MBSE raises the level of abstraction when implementing solutions and, thereby, reduces the problem-implementation gap for the specific problem domain significantly. Utilizing specialized modeling languages for each problem domain strongly separates each domain concern and allows them to be handled in multiple dimensions (Tarr et al. 1999), *e.g.,* models may regard different concerns of multiple problem domains and may be handled differently according to the phase of development. Generative software engineering utilizes models created with DSLs to automatically generate an implementation of the specified system (Czarnecki 2005), and thereby, allows one to obtain an implementation from the models directly. This section outlines fundamentals for the methodology introduced in Section 3 regarding modeling languages and generators and describes the challenges addressed by the methodology.

[1] In this paper, a component is a unit that can be independently delivered and versioned, has explicit interfaces, is largely independent of the environment can be combined with other components, is reusable, and has no externally observable state.

2.1. MontiCore Language Workbench

MontiCore (Hölldobler & Rumpe 2017; Haber, Look, et al. 2015b) is a workbench for developing compositional modeling languages (Combemale et al. 2018). MontiCore uses context-free grammars (CFGs) to simplify the definition of a language. The grammars are used for the simultaneous development of abstract and concrete syntax. The grammar of a language contains production rules that are derived from terminals and non-terminals which determine the permissible syntax of a language. MontiCore also offers options to define specific language components, which can then be combined into one language. This makes it easier to reuse, adapt or extend individual parts. The parser, abstract syntax classes, and other context dependencies are automatically generated by MontiCore and can be adjusted by hand using the TOP-mechanism (Hölldobler & Rumpe 2017), which utilizes inheritance to override or to extend the generated code. This infrastructure then allows a generator to process models of the specified DSLs. The imported models are read by the parser and then transformed into an Abstract Syntax Tree (AST). Using the infrastructure, validity checks can also check context-sensitive properties. The AST can be transformed as required for the application. The transformed AST is then used along with templates by a template engine to generate the target code.

2.2. Languages

By using the MontiCore language workbench, we have developed textual representations for several modeling languages. As language engineers, we are already familiar with the following DSLs:

Class Diagrams for Analysis (**CD4A**) (Rumpe 2016) is a DSL representing classes, their relations, attributes, and methods and is used for analysis purposes. **Object Constraint Languages (OCLs)/P (OCL/P)** is an extension of the UML OCL adapted for programming (Rumpe 2016). The **Tagging Language** is a structural DSL used to modify and provide meta-information for existing models (Greifenberg, Look, et al. 2015). It is used to extend existing models in a separate model file without modifying the original one. The tagging information can be used to generate an implementation that has more context and is platform-specific. The **GUI-DSL** (Gerasimov, Michael, et al. 2020) is a language to define Graphical User Interfaces (GUIs) and the data they display. A GUI-DSL model describes a page of a web-application user interface. It is designed to define views and connections to the business logic. These DSLs are used in our generator framework for creating an EIS and are included in our running example.

2.3. MontiGem Generator Framework

MontiGem (Adam et al. 2020) is a generator framework for enterprise management applications, a tool for the iterative, model-driven development of full-size real-world applications. The framework takes models as an input and generates code into a Run-time Environment (RTE), where the models are written in modeling languages described in the previous section. CD4A models are used as a base for the data structure and connection to the user interface, GUI-DSL models define the GUI, a validation infrastructure is generated from OCL/P models and attached to the data structure and Tagging is used to generate additional code to handle specifics of a domain.

Languages used in MontiGem are processed by MontiCore, which generates parts of the generator framework such as lexers and parsers. MontiGem can be extended by other MontiCore products in different ways, *e.g.,* by language composition (Hölldobler et al. 2018) or generator chains, where models are produced from generators using other models as input. One such extension allows the derivation of GUI models from CD4A models, which enables the rapid building of functional prototypes of an information system (Gerasimov, Michael, et al. 2020).

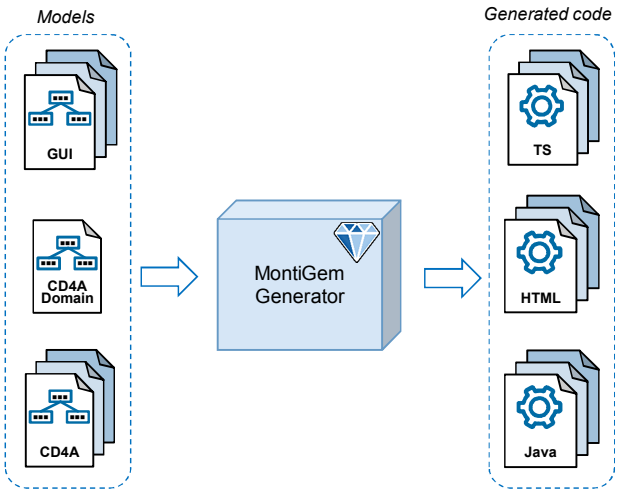

Figure 1 Generation process of MontiGem

Generally, the models used in MontiGem serve as a basis for the generation of code written in general-purpose languages. Java and TypeScript code is generated for the implementation of data structures as well as for data validation and communication between the back-end and the user interfaces. TypeScript and HTML code is generated for the implementation of GUIs.

2.4. Challenges of Using DSLs

While model-driven generative engineering reduces the problem-implementation gap and enhances the adaptation of a running EIS, some challenges, which are mostly of methodological nature (France & Rumpe 2007; Bucchiarone et al. 2020; Aniche et al. 2019), remain. The methodology proposed in Section 3 aims to meet the following challenges, which are later mentioned in corresponding activities.

Relating Models between Heterogeneous DSLs. One major challenge arises from the fact that models of complex software are heterogeneous (Aniche et al. 2019) and are often written in multiple modeling languages. A sound generative approach has to systematically relate these models, which is referred to as the *DSL-Babel challenge* in (France & Rumpe 2007). These relationships could be different in different development phases. Implicitly, (France & Rumpe 2007) names two possibilities to overcome this challenge:

– to explicate the relations between model elements in a mega-model, or
– to implement a generator that combines the models accordingly.

Supporting Evolution of Models and DSLs. Due to iterative development, changing requirements, and bug fixes, models evolve continuously during the development process (Aniche et al. 2019; Maoz et al. 2010). Effective change management still challenges MBSE. Advances in this area are divided into syntactic differencing, *e.g.,* (Taentzer et al. 2014), semantic differencing (Langer et al. 2014), and hybrid approaches (Kautz & Rumpe 2018; Maoz & Ringert 2016). Within generative MBSE, the DSLs used are continuously improved and reused for new problem aspects due to changing requirements or bug fixes. The systematic relations between the models are therefore also subject to continuous evolution. Adapting or implementing code generators that implement these model relationships require an integrated change management system for models and generators.

Reusing DSLs and Generators. Another challenge is the definition of criteria to be used when choosing an appropriate language that can be reused in the context of the problem domain. As engineering DSLs and tooling, such as generators or model analyses is a demanding process, efficient development reuses existing implementations systematically (Bucchiarone et al. 2020; Degueule et al. 2015). However, as DSLs are used to describe solutions to particular problem aspects, reusing a DSL strongly correlates to the decomposition of the problem domain into problem aspects: Decomposing the problem domain restricts the DSLs that can be reused. Vice versa, choosing to reuse a DSL requires that the problem domain is decomposed in a way such that it comprises a problem aspect to which solutions can be modeled in the respective DSL and that existing generators produce implementations of solutions to (parts of) the problem aspect. When retrofitting generative aspects into existing EISs, an effective development methodology must offer systematic criteria to choose DSLs and generators for reuse, which take the decomposition of the problem domain into account.

3. Retrofitting Generative Aspects in Existing Projects

MBSE envisions system development to rely on models that serve as primary development artifacts (Selic 2003; Völter et al. 2013; France & Rumpe 2007). Nowadays, applications evolve continuously, *e.g.,* through improvements, or added functionalities (France & Rumpe 2007). Thus, agile, iterative processes have emerged and are currently well established in the practice of software engineering (Rumpe 2017). Among others, *MBSE enables automated code synthesis and is, therefore, highly beneficial for agile development and applications that change continuously.* Therein, models lift the level of abstraction used to describe a (software) solution and generators implement the mapping between these models and the solution. Automating code synthesis for parts of a system already facilitates the extension of an existing application, which the lessons learned from industrial generative, model-based engineering of EISs published in (Adam et al. 2020) substantiate. These lessons include the fact that MBSE improves efficiency when reusing (and adapting) an existing generator, or when a generator developed in the setting of the project will be reused for other projects. In turn, generators are capable of assuring consistency between the generated artifacts, as they implement the mapping from model to code. This allows one to assure efficiency and correctness during the engineering process, because developers obtain automatic and direct feedback.

The problem domain spanned by modern EISs comprises multiple aspects of a highly variable nature (Möller et al. 2011; France & Rumpe 2007). Thereby, a problem aspect does not necessarily belong to one module of the system. Rather, one problem aspect requires specific mechanisms to handle it and may be solved across various modules of the implementation. A modular architecture of the target system does not necessarily aid in identifying the problem aspects that allow for reusing generators or modeling languages.

So far, research lacks an overall approach that integrates the engineering activities within all phases of the engineering process. Existing methodologies often consider only parts of the engineering process, *e.g.,* dividing the implementation into modules (Rogozov et al. 2020) or certain time phases such as software language engineering (Combemale et al. 2018), generator engineering, domain engineering (Kang et al. 1998) or application engineering (Czarnecki 2005). Section 6 discusses related methodologies in detail.

3.1. Agility, Phases and Iterations

The methodology assumes the existence of a running EIS and substitutes the implementation step by step by generated parts. The result is, again, a running EIS.

Our methodology covers three main phases: (Phase P) problem analysis and decomposition, (Phase L) domain-specific language engineering, and (Phase A) application engineering and operation. Whereas the first phase is more related to the problem domain, the other two phases have a stronger relation to the solution domain.

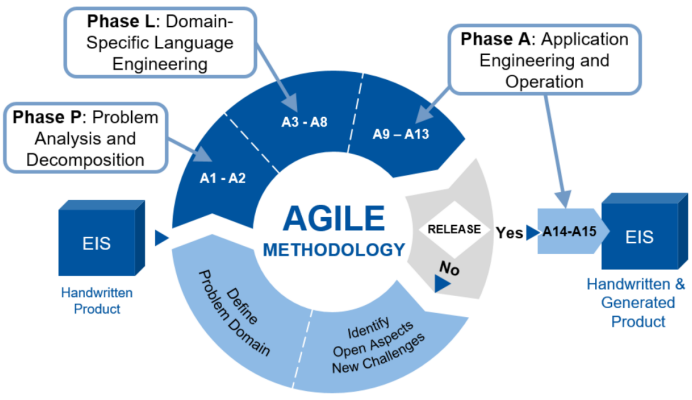

Figure 2 Agile development process with activities and phases

The proposed methodology can be mapped to an agile development process as follows: Starting with a hand-written product or the result of a previous development cycle, the problem at hand is analyzed and decomposed (phase P, activities A1 and A2 in Section 3.3). Activities for domain-specific language engineering, phase L, need to be applied (A3 to A8). Finally, activities A9 to A13 are considered for Phase A, application engineering, and operation of the product. Engineers might not follow these activities consequently, *e.g.,* they return to language engineering or generator engineering activities if the generated code is not sufficient for their problem aspect.

After completion of all necessary activities, a decision is made together with experts from the problem domain whether the product will be published or has to be refined. Upon release, activities A14 and A15 may be performed for operation (part of phase A). In case new problems arise and the product is therefore not released, these new aspects have to be identified and the process starts again at activity A1 with the updated problem.

In practice, time synchronization of DSL and application engineering is not mandatory. There exist heterogeneous development teams for phase L and A, *e.g.,* Clark et al. (Clark et al. 2015) describes these two groups of engineers: language engineers and integrators on the DSL side and system engineers on the application side. The latter might even be several groups of system engineers for different applications. Thus, such development teams may carry out activities from phases L and A independently from each other.

However, they are not entirely independent, as (intermediate) results of an activity in phase A may require repeating activities from phase L and vice versa. For example, engineering code generators is an activity of phase A but is strongly related to the activity of engineering DSLs, which is part of phase L. This is because during MBSE with code generation, not only the models evolve, but also the languages these models are written in. Changing the implementation of a DSL in phase L requires updating of the generators that process models of the evolved languages in phase A. Findings in generator development in phase A might lead to additions in the DSL in phase L.

3.2. Conceptual Model of the Main Concepts

The conceptual model in Figure 3 captures the main concepts applied in MBSE projects that utilize code generators. Following the suggestion of Mayr and Thalheim (Mayr & Thalheim 2020) to describe conceptual models, we discuss its notion space in detail.

A *problem* describes the reasons why a specific application is needed. It may comprise several *problem aspects*, which further detail the problem and challenges. Problem aspects categorize several users *requirements*, where one requirement may be categorized by multiple problem aspects. A problem aspect could be handled by an existing *library*, *handwritten code* or can be described by *models*. Therein, models describe a solution to (a part of) a problem aspect. As problem aspects may overlap, one model possibly describes solutions to (parts of) multiple problem aspects. A *modeling language* defines syntax and semantics of a set of models, *i.e.,* of those models that are written in the

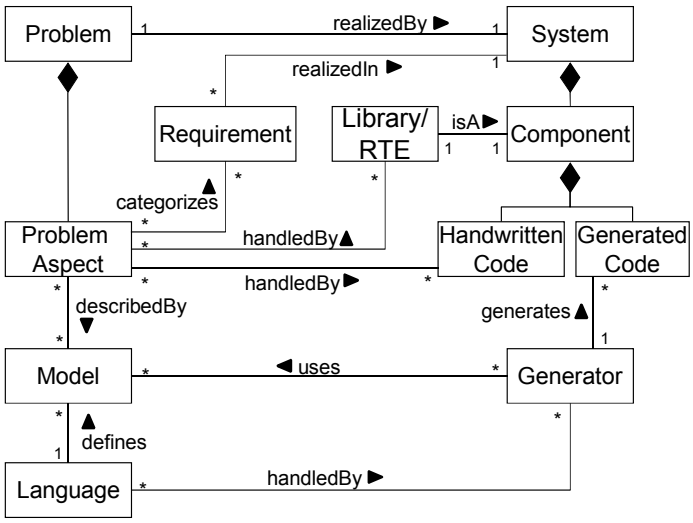

Figure 3 Conceptual model of the main concepts

language and each model belongs to exactly one language. Code *generators* use models defined by multiple languages and *generate code* in one or more programming languages. One model may be processed by multiple generators, and for one language there may exist multiple generators that process models of that language. A *system* consists of *components*, libraries, and if generated code is used typically also an *RTE*. Each component may consist of handwritten and generated code.

The described problem decomposition allows for systematic reuse of not only the components of a system, but also the techniques and tooling used to engineer them. The latter is highly beneficial as it allows one to implement new systems or components more efficiently, *e.g.,* as pointed out in (Adam et al. 2020). The development phases (Domain-Specific Language Engineering and Application Engineering) map to the concepts in Figure 3 as follows: The problem analysis and decomposition phase is concerned with identifying the problem aspects and the categorization (as well as the elicitation) of requirements. DSL engineering identifies and possibly implements the modeling languages for describing the identified problem aspects. During the application engineering and operation phase engineers create models that describe the problem aspects in the languages identified in the previous phase. A generator uses these models to generate the code of the system's components which is then finished using handwritten code that is also created in this phase. Moreover, this phase includes choosing or developing libraries and the RTE, as well as, *e.g.,* system testing or deployment.

3.3. Methodology Activities

Figure 4 shows an overview of all 16 activities in our methodology. Our methodology includes several *decision points*, where alternatives exist, and an informed decision of an engineer is needed. It allows us to handle the aforementioned challenges using a divide-and-conquer approach. Optional activities are marked with a dashed line.

The given numbering is not an explicit order in which they must be performed but rather a recommendation. The method-

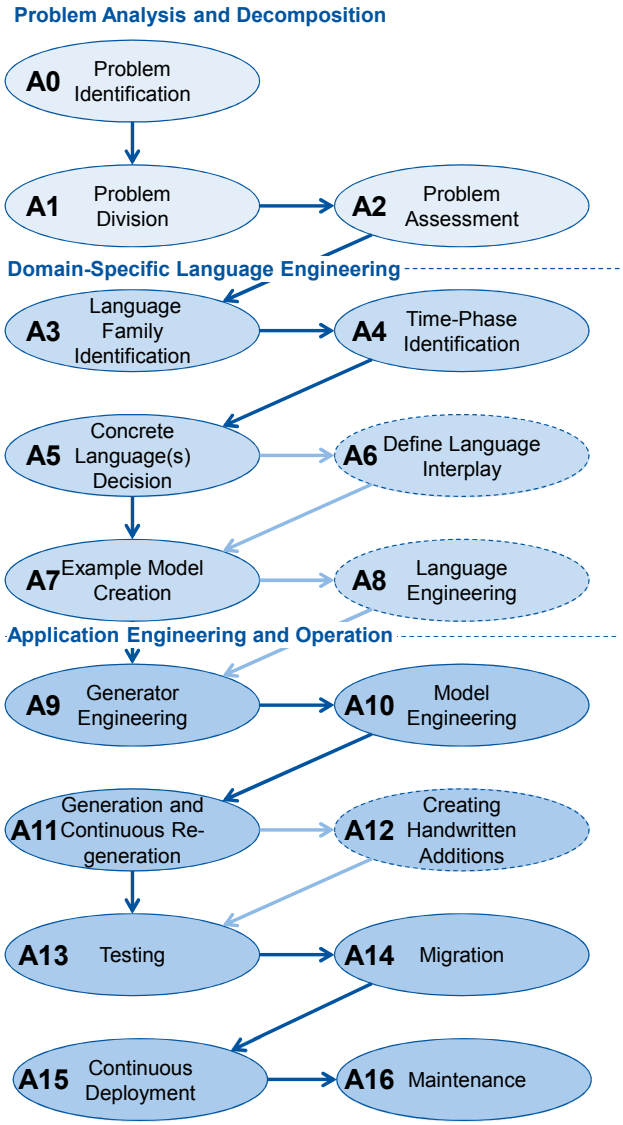

Figure 4 Overview of all 16 activities (optional ones with dotted boarders)

ology defines the essential activities that have to be customized in an agile engineering approach. Phase P includes activities 0-2, phase L activities 3-8 and phase A activities 9-16.

Activity 0: Problem Identification.

The following activities could be successfully applied if the main requirements for the future system are identified during the engineering process or have been analyzed in an already existing system. This is *Activity 0: Problem Identification*. In agile engineering processes the identification can be only partially completed before starting with activity 1 but at least all relevant requirements for one problem aspect should be identified. Waterfall-like processes allow one to identify requirements more comprehensively.

Practical Application: An easy way to identify the main problems is to investigate the requirements and system architecture of the existing application. If software engineers apply our method to an existing information system with a client/server architecture the problem identification will largely result in the same aspects mentioned in activity 1 in Section 4.

Activity 1: Problem Division. The problem has to be divided into smaller problem aspects as they are easier to handle. Figure 5 shows this general idea. Problem aspects (shown as one or more related circles) relate to specific tasks, *e.g.,* calculate some values, handle user input, or communicate with an endpoint. Moreover, there exist overarching tasks (areas in dotted lines), *e.g.,* run processes, display GUI, process data or provide security, which could spread over several components of the future system.

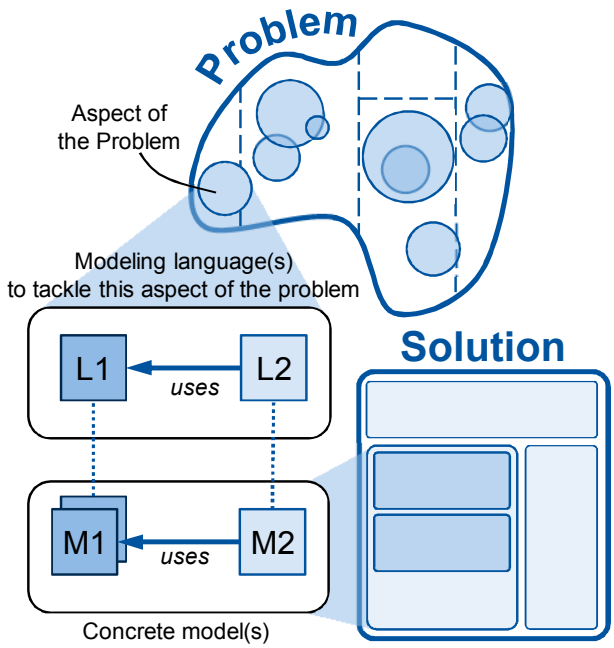

Figure 5 Decomposition of problems

In MBSE, we assume that there exist one or more languages to tackle an aspect of the problem (see Figure 5). Thus, some concrete models could be created and used as input by one or more generators to create parts of the resulting system. To make this possible, further actions of our methodology have to be applied, *e.g.,* language engineering in activity 8 and model engineering in activity 10.

Activity 2: Problem Assessment. In this activity, the developer has to decide if the problem could be handled using (a) existing libraries, (b) models, (c) handwritten code, or a combination of the above. A check if libraries which could fulfill all requirements already exist (option a) is a good way to improve an existing system or create a new one. For using models and Generative Software Engineering (GSE) (option b), it is important (i) to identify which handwritten parts are to be replaced by generated ones, or (ii) to have experience in identifying parts of a system that could be handled by models and GSE efficiently. A good candidate for (i) is, *e.g.,* repetitive code (Völter et al. 2013). The decision about parts of a system which could be handled by models and GSE (ii) is easier to make for experts with more memorized development patterns

(a.k.a. 'experience' (France & Rumpe 2007)). Variant (c), to use or write handwritten code is the last option if (a) or (b) are not feasible. This occurs, *e.g.,* for business logic which is very specific for an application domain.

Activity 3: Language Family Identification. Relevant language families have to be identified. Languages typically aim to model different kinds of system aspects. There exist language families for describing the structure or behavior of a system (Breu et al. 1998; Burgueño et al. 2019). Some approaches, *e.g.,* (Pastor et al. 2001) also allow for the definition of functional models or talk about information models (conceptual models) (Burgueño et al. 2019).

In practice, languages from all families are needed to model an entire system. If we consider a system for organizing travel in a company, *information models* help to advance the discussion about the main concepts and provide a common understanding among stakeholders. They are mainly used in the analysis phase of a project and are, thus, not further discussed in the later phases. *Structure models* are needed to describe data structures, *e.g.,* there are traveling employees, employees to handle travel requests, destination of travel, travel dates, and expenses. *Behavior models* describe processes, *e.g.,* to make a travel request or to handle travel accounts. *Functional models* consider algorithms, *e.g.,* to calculate refund of travel expenses for the employee or to determine whether a bonus program of a tour operator is advantageous. Identifying a language family before choosing a concrete language helps to leave more options open and to search for alternatives.

Activity 4: Time Phase Identification. The development of a software system is typically divided into several phases. For us, the following phases are relevant:

- *Design time:* Application developers write models and code, *e.g.,* the Class Diagram (CD) defining important concepts or an algorithm to calculate travel expenses in Java. Other approaches divide writing models and writing code into different time phases (design and implementation).
- *Compile time:* The models are processed and their syntax is checked using the grammar of the modeling languages, *e.g.,* checking if a CD includes only concepts such as classes, relations, and interfaces and only known types for attributes. Some of the models are translated into source code which is converted into byte-code, *e.g.,* the classes in CDs are used to generate data classes in Java. The concrete target programming language depends on the templates used in the generator.
- *Deployment time:* The built application including, *e.g.,* byte-code, models, HTML code, is deployed to the execution environment handling the application, *e.g.,* servers, and databases. In this step, models could be used to support the correct deployment on different machines, *e.g.,* a UML deployment diagram defines which software component should run on which hardware component.
- *Run time:* The EIS is actually running, and *e.g.,* a user can log in, see, and edit data. Some of the models can be interpreted at run time or even changed or supplemented by the user. Run-time models may be used in this phase, *e.g.,* to handle allowed workflows.

The main focus of this publication is on compile- and run-time (following (France & Rumpe 2007)), as our design time models are used as artifacts for code generation and, thus, are relevant for compile-time as well. Our methodology allows one to utilize heterogeneous languages differently, depending on the phase of development and specific problem aspects.

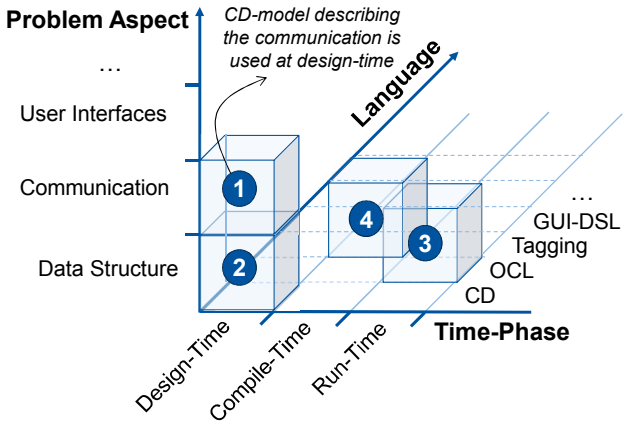

Figure 6 Models and their dimensions

The same language or even the same model could be used for different problem aspects during different phases of development. Figure 6 illustrates this with concrete examples: A CD model (1) can be used to describe a communication infrastructure between the front-end and the back-end of a client-server application and is used at design time. The same model (2) can be used to describe the data structure at design time. OCL models (3) define restrictions on the data structure and are used at run-time to validate data inputs. Tagging models (4) add platform-specific information for the data structure and are used at compile time.

Activity 5: Concrete Language(s) Decision. In this activity, we have to decide on one or more concrete modeling languages to handle our problem. We have three options for this decision (see the challenge in Section 2.4): (a) to choose one or more specific languages to handle this aspect and/or (b) to extend or adapt existing ones and/or (c) to create one or more new DSLs.

For (a) multiple languages or language combinations would typically be an option, *e.g.,* suitable languages for modeling behavior are Business Process Model and Notation (BPMN), UML activity diagrams, or Petri Nets (PNs). Before the decision for a concrete language is done, information about the languages and related tooling is needed, *e.g.,* whether there exists a language to create appropriate data structures and if it already comes with an existing generator which could be reused or adapted in activity 9. Again, experiences from preceding projects or experienced engineers help to shorten the time needed for this activity.

A language could be extended or adapted (b) if existing concepts fit well, but some aspects are missing or need to be further defined, *e.g.,* an already created CD DSL includes no method bodies or additional task types are needed in BPMN

models.

One or more new DSLs should be created (c) if existing languages do not fit the intended purpose, another representation is needed (graphical, textual), existing languages are too big (especially GPLs) or specific concepts are needed, *e.g.,* models for the description of permissions and roles in an application.

This is also the right activity to identify if one or more models have to be created dependent on the problem aspect, *e.g.,* for GUIs it would be feasible to create several models for each of the graphical interfaces.

Activity 6 (Optional): Define Language Interplay. A generative approach must handle a variety of problem aspects. To model one component, several modeling languages are needed to formally specify each aspect individually (France & Rumpe 2007). For example, CDs model structural aspects of a system while behavioral aspects are not considered. Therefore, a generative approach must integrate models of heterogeneous languages, which is reflected in the fact that GPLs, *e.g.,* UML, Systems Modeling Language (SysML) provide multiple languages for describing software systems (see challenge in Section 2.4). Problem decomposition (activity 1) therefore makes language composition necessary. To achieve language composition, techniques such as language aggregation, embedding, inheritance, extension and restriction are needed (Haber, Look, et al. 2015a; Hölldobler & Rumpe 2017).

- **Language aggregation:** Models of different languages are used together to handle problem aspects. A model in one language references a model in another language, *e.g.,* types defined in CDs are referenced in OCL. The models in this case are kept separate.
- **Language embedding:** One model may consist of several sub-languages, which have been developed independently. Thus, two or more languages are used in one model, *e.g.,* Java method bodies in CDs to specify the implementation of classes.
- **Language inheritance:** New languages are created based on an existing language by the reuse and modification of existing concepts, *e.g.,* inheriting from a basic structural language that describes the type and enriching it with CD elements, such as method signatures and interfaces.
- **Language extension:** This is a conservative form of language inheritance and leads to a higher degree of black-box reuse. One language extends another language to be able to use the language types, *e.g.,* OCL is extended by additional operators without overriding existing operators.
- **Language restriction:** One language restricts the set of models of another language and, thus, only allows a subset of models to provide better options for specific needs, *e.g.,* forbid the usage of interfaces in a class diagram. This is a non-conservative technique. Using restriction might allow one to reuse the tooling and infrastructure of the other language.

Figure 7 shows possible relations between different languages. One language could have different kinds of relations to other languages, *e.g.,* language 2 uses language 1 and can be used to restrict language 5. Moreover, languages could be transformed into other languages (with or without information loss).

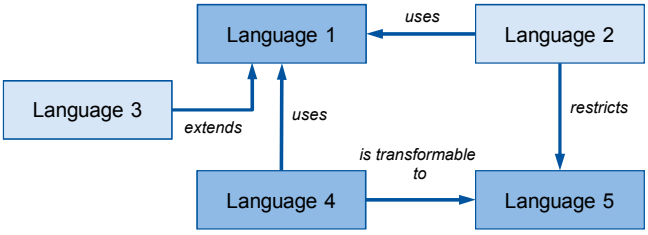

Figure 7 Relations between different modeling languages

As stated in (France & Rumpe 2007), the use of many languages requires to relate concepts across languages and a means to ensure consistency of concept representations.

Activity 7: Example Model Creation The creation of example models will help if (a) a new DSL should be developed in the next activity, namely language engineering and (b) a generator will be adapted or newly developed in activity 9, namely generator engineering. Having some example models helps in the case of (a) because the engineering of a language needs concrete examples, *i.e.,* what kind of models should be possible in a language. It helps in case of (b) because a generator needs models as inputs and has to know about the target code architecture and the relations to information in models.

Not all resulting models have to be created in this activity as it requires a high knowledge of the domain and thus, domain experts should be considered. Moreover, different project members may have the competencies and duties to handle model engineering and generator engineering (Clark et al. 2015).

Following the definition of models from Stachowiak (Stachowiak 1973), a model is an abstraction of an original made and used for a purpose. The models created in this activity have the purpose to support generator engineering for a concrete problem aspect. As MBSE and automated code generation continuously work with the created models, the purpose of the models may expand over time. Models are adapted iteratively and are needed for the creation of the system's components.

Activity 8 (Optional): Language Engineering. The decision to create a new language, or to extend or adapt an existing one is made in activity 5. There exist several methodologies on how to systematically design new DSLs, *e.g.,* (Karsai et al. 2009; Frank 2013; Michael & Mayr 2015). They include aspects such as considering the language purpose, reusing existing languages or language concepts and identification of relevant concepts in common. For more details, we refer the reader to publications about software language engineering, *e.g.,* (Haber, Look, et al. 2015a; Combemale et al. 2017; Hölldobler et al. 2018).

Activity 9: Generator Engineering. This activity, again, has several decision points regarding the reuse of and the number of generators to be developed.

Regarding generator engineering, we have three options: (1) either an existing generator is fully reused, (2) reused with adaptions or (3) newly developed. (1) is possible if the language, time phase, problem aspect, and target language(s) are kept the

same and thus, the generated code is fully usable. (2) could be considered if some of the main aspects are kept the same. This will occur, *e.g.,* if the technologies used are the same or the same modeling languages are used in input models. Moreover, generators have additional variation points, *e.g.,* a change of the templates to change the target language. The time needed for adaptions is highly variable and depends on what parts have to be changed. (3) has to be considered if a new DSL was developed, a completely new problem aspect is tackled, the target language changes or the time phase is different. In practice, the reuse of a generator or generator parts (with or without adaptations) is preferable as it is less time consuming than developing a new one (Adam et al. 2020).

Generative approaches employ generators to handle models from different languages. This can be done in two different ways:

(a) One generator takes a variety of models in heterogeneous languages as inputs or
(b) there exist multiple generators which produce compatible outputs.

In case (a), the knowledge of how the models interrelate is implemented as a part of the generator. The advantages of this approach are that it supports conventional agile development processes and the generator developer has all relevant knowledge about the interplay of models.

In case (b), each generator implements the interrelations of its input models to other models. Thus, the generator developer must rely on the developers of the other generators to also assure compatibility.

Depending on the use of the models, it could be sufficient to create an interpreter for using the models instead of a generator. This could occur, *e.g.,* for behavior models such as business process models or models which are defined by users at runtime. Interpreters could, again, be fully reused, partially reused, or developed from scratch.

Activity 10: Model Engineering. We have to create models needed for the resulting application. In practice, not all models are created sequentially. The generators include developer support to check if a model is syntactically correct (using the grammar defining the language and additional information, *e.g.,* context conditions). Thus, it is useful to continuously regenerate the code (activity 11) during model engineering. During this activity, model evolution management including syntactic (Alanen & Porres 2003; Kehrer et al. 2011) and semantic model differencing (Maoz et al. 2010; Kautz & Rumpe 2018; Acher et al. 2012; Langer et al. 2014; Maoz & Ringert 2016) enables monitoring changes to the represented system. Semantic differencing, in particular, enables one to decide, whether the modeled system has been changed to include undesired properties, *e.g.,* (Drave, Kautz, et al. 2019). It might be possible to reuse existing models without changes for another problem aspect, *e.g.,* a class diagram for the generation of the data structure and the communication infrastructure. However, this must be considered on a case-by-case basis.

Activity 11: Generation and Continuous Regeneration. The application code is generated using one or more models as input. The parser of a language checks the input model for syntactical correctness. Additional context conditions perform model checking, *i.e.,* loading referenced models to check if a concept exists.

As mentioned before, in agile approaches models are constantly adapted to new needs. Thus, a continuous regeneration of the application is needed.

Activity 12 (Optional): Creating Handwritten Additions. Handwritten code may be needed in addition to generated code. In practice, handwritten code could be necessary, *e.g.,* when the implementation needs a specific business logic, which is easier to implement directly. In a brown-field approach, the needed code might already exist and could be only adapted in this activity. Further maturing of the used DSLs could allow for replacement of previously handwritten code.

Activity 13: Testing. The components as well as the whole system have to be tested. This includes the systematic integration of components, system and user acceptance tests. The first three types have the potential for automation. MBSE approaches could again be used to create the test cases as well as the tests themselves (Rumpe 2003). The last type, user tests, will have to be done mostly manually together with users of the application.

Activity 14: Migration. Changes in the application create a need for (a) system migration and optional (b) data migration. System migration (a) includes version changes and changes to the architecture. Data migration (b) is necessary if the EIS's data structure has changed or the storage technology was replaced.

Activity 15: Continuous Deployment. Changes in the application have to be regularly deployed. This leads to continuous delivery (including manual tasks) or continuous deployment (could be fully automatic).

Activity 16: Maintenance. The models, DSLs, generators or interpreters, and the application have to be maintained. This includes the repetition of formerly mentioned activities. The maintenance includes the correction of faults and the development of additional functionality.

Use in practice. We have introduced 16 activities within our methodology for retrofitting generative aspects in existing applications. For a better understanding of how to use the methodology in practice, we show its use and lessons learned within the development of an EIS on several problem aspects.

4. Validation by example

We apply our methodology on the MaCoCo application, an EIS for financial management and controlling of university chairs and institutes (Gerasimov, Heuser, et al. 2020). EISs have to accomplish a variety of tasks, *e.g.,* process data, run processes, provide a GUI and assure security. Generative engineering of such systems, thus, requires systematic addressing of these tasks. For our approach, we designed and reused DSLs tailored for EISs in heterogeneous problem domains. This section maps the DSLs to the problem domain(s) they address and describe the strategy to integrate them within a generative engineering approach.

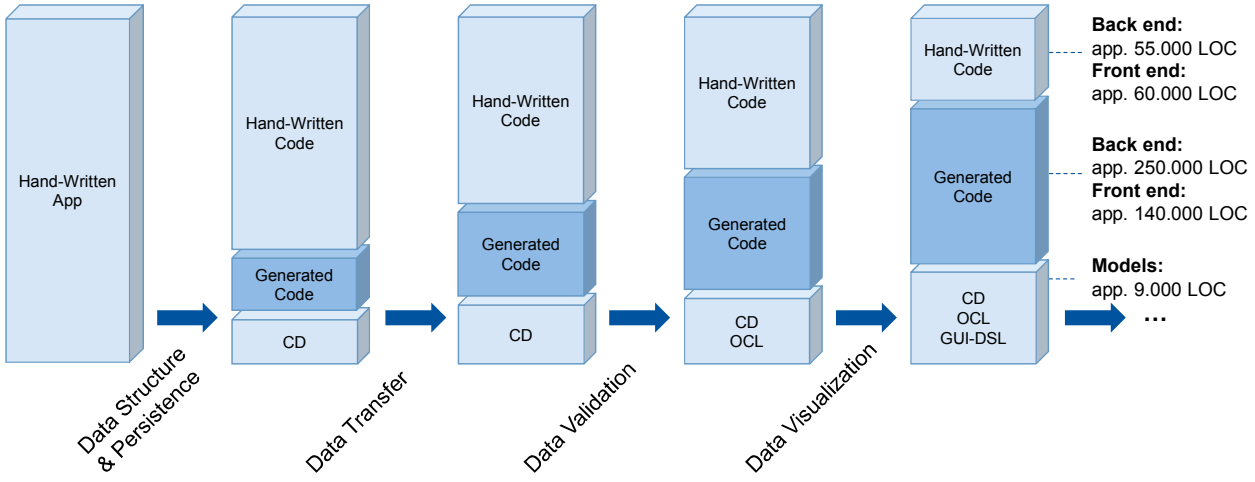

Figure 8 Development of MaCoCo using the proposed methodology

The aim of the MaCoCo project (Adam et al. 2018; Gerasimov, Heuser, et al. 2020) is to create software for financial management and controlling. We used MontiGem to retrofit generative aspects in the first version, which was a handwritten application, and further evolved the application to create new functionalities and to include further DSLs. The project started as a purely handwritten application and was adapted to incorporate the MontiGem generator framework. The development process follows an agile approach and includes strong user participation. The application is still being developed during its operation.

MaCoCo [2] is a web-based EIS which handles finances, human resources and projects. The functionality in the financial area includes financial planning, accounting, budgeting for research projects, and quarterly funding requests for approved projects. MaCoCo allows for staff administration regarding contracts, permanent positions, and organizing student assistants. Further supported processes are the completion and submission of project timesheets, requesting vacations, or staff planning on projects regarding time expenses and costs. In our project, developers hold one or more of the following roles: The **Language Engineer** defines the DSLs used by the **Application Modeler**s. The latter create models that represent solutions to dedicated aspects in the problem domain of the application. The **Generator Engineer** develops new generators which automatically derive an implementation from the models in the DSLs. A **Generator Customizer** maintains and configures existing generators. The **Tool Provider** configures and maintains the libraries and components used in the implementations of the run-time environment, the application, and the generator. Handwritten additions are handled by an **Application Programmer**.

The MaCoCo application started as an entirely handwritten prototype (Figure 8). In the first iteration of the development process, the data structure was modeled and a generator was developed to continuously generate both data structure and persistence of the application. Development of the front-end required data transfer between client and server. Thus, the generator was extended to provide the code implementing the communication. A rise in the complexity of the data model and implemented business logic resulted in the inclusion of OCL to validate data sets. In the next iteration, the GUI-DSL was developed and incorporated in the generator-cycle of the application, in order to standardize and speed up front-end development. Currently, other problem aspects are identified and both languages and models are enhanced to be able to generate new aspects which are written by hand.

Currently, about 78% of the complete application is generated. The code that still needs to be handwritten includes application-specific logic and run-time environment implementation and configuration. This includes only the lines of code in the application itself, the generator, which is used in a multitude of projects, has about 18.000 lines of code that are written by hand and <1.000 lines of generated code. Extraction of the general run-time could further save efforts for the hand-written code and enable the reuse of common implementations.

The lines of code of the current state of the application in Figure 8 differentiate between the back-end and front end. These numbers show that approximately 390.000 lines of code (78%) are generated from the models. The goal is to use domain-specific languages as much as possible to reduce the creation effort for domain experts. The graphic shows that about 9.000 lines of domain-specific models are the base for 390.000 lines of generated code.

In the following, we take a closer look on the application of the previously defined activities to the MaCoCo project.

4.1. Activity 0: Problem Identification

The original problem, which led to the development of the MaCoCo application, was that no appropriate application existed for the financial and staff controlling of small and medium university chairs. Existing accounting software included only some parts of the needed aspects for chair controlling and the Enterprise Resource Planning (ERP) system of the central university administration does not include chair-specific processes and planning possibilities. Thus, chairs used various complex

[2] Project Website: https://www.se-rwth.de/projects/MaCoCo.php

spreadsheets or different smaller self-developed applications which were error-prone, did not cover all needs for project and staff controlling, and required a lot of manual effort for double-checking bookings.

The solution for this problem was an enterprise information system for data management with a database, communication interfaces to external applications and user-friendly GUIs. We have collected the requirements for the system in an agile approach by defining each use case together with future users.

Developing this system by hand was time-consuming and required a lot of testing effort to keep all parts of the system consistent. Thus, we decided to replace parts of it step by step with generated code.

4.2. Activity 1: Problem Division

Within this activity, we can divide the problem into different smaller problem aspects to further specify each of the application's parts, *e.g.,*

- Architecture
- Communication
- (Domain) Data Structure (a)
- Data Validation (b)
- GUI (c)
- Persistence
- Privacy
- Security
- Technical Infrastructure (d)
- Tests

In the following, we focus on four problem aspects within the context of the MaCoCo project and discuss them in detail: data structure, data validation, GUI, and technical infrastructure.

a) Domain concepts which lead to the **data structure** of the application are one of the most important parts of data-centered systems. The data structure consists of multiple classes (in object-oriented programming languages) which describe the structure of the objects the application is handling. The data structure also includes classes that are used for the interaction with external libraries and a domain model. In this context, the domain model characterizes the information necessary to describe the actual user data. As a financial system, the MaCoCo project domain model includes classes like `Account` and `Budget`.

b) **Validation** is needed to ensure the validity of data which is handled by the application. In particular, user inputs must be validated to make sure the data is correct. The validity can have multiple levels:

(1) The check if an input matches the expected type, *e.g.,* the application expects a number for an account balance.
(2) The validity of a single value, *e.g.,* the length of an account name should be between two and fifty characters.
(3) The validity of a complete object or even a set of objects, *e.g.,* a sum of budgets that belong to the same account cannot be negative.

c) A **GUI** facilitates interactions with the users of an EIS. The roles of a GUI include visualization of the application's data and options for the user to interact with the system, *e.g.,* clicking a button or writing text. MaCoCo provides a rich user interface to view and operate with financial data, which includes web pages for account creation, dashboards for viewing statistics, etc.

d) The **technical infrastructure**, such as a web server which runs the MaCoCo web application is an important part of the application itself, but is usually not directly seen by the users of the system. The technical infrastructure can have an influence on the possible functionality of the application and future extensibility or scalability.

Practical Application: Each problem aspect is related to one or more requirements, therefore problem division is practically done by categorizing requirements. For example, the requirements to create, read, update and delete data in a database are together one problem aspect, namely ensuring persistence. Similar architectures such as our client-server application will have mainly the same problem aspects as specified at the beginning of this section.

4.3. Activity 2: Problem Assessment

During problem assessment, we categorize the problem aspects from activity 1 and discuss if those could be handled using models, handwritten code, or existing libraries, or a combination of the above. The problem assessment should be done for each of the problem aspects. Even when some of the aspects are similar, they could be handled using different approaches. Following is a definition of the approaches for the aforementioned problem aspects:

a) The **data structure** of MaCoCo project includes the classes which represent the domain objects themselves. Each such data class can have additional classes such as a builder. This structure qualifies for generation due to its systematic pattern.

b) To ensure consistent **data validation** in the back end and the front end of the MaCoCo application, a validation logic can be modeled and used as a source for both parts.

c) For the definition of **GUI** parts, it is necessary to rely on existing libraries to streamline the overall process of displaying and interacting with GUI elements. As a result, the definition of (custom) GUI elements in MaCoCo is done completely within the bounds of the GUI libraries. On the other hand, GUI also has a strong relation to the data structure, since one of the main purposes of a user interface is to display data. In the MaCoCo case, this relation is expressed in models to ensure consistency between a GUI and a data structure.

d) For the **web server** setup, the MaCoCo project utilizes libraries designed for this problem specifically, thus it is not necessary to model or implement a web server by hand. From this point on we will not consider this problem aspect as it is irrelevant in most of the further activities.

Practical Application: Libraries are well-suited for building code to be used in run-time environments or framework-specific library components, such as a specific communication infrastructure between server and client. The models are useful if the code to be generated has a spe-

cific pattern, but it is not easily replaceable by an existing component.

4.4. Activity 3: Language Family Identification

The identification of feasible language families for each aspect of the problem allows for a reduction of the number of languages used. In the following, we inspect what language types could handle each aspect, and if a modeling language should be used at all. In order to analyze the problem aspects identified in Section 4.3 more precisely, Table 1 represents an assignment to the language families.

	language	
problem aspects	**structure**	**behavior**
Architecture	✓	✓
Communication	✓	*P*
(Domain) Data Structure (a)	✓	✗
Data Validation (b)	*P*	*P*
GUI (c)	✓	*P*
Persistence	✓	*P*
Privacy	*P*	✓
Security	✗	*P*
Technical Infrastructure (d)	✓	✗

✓= fully suitable, ✗= not suitable, *P* = partially suitable

Table 1 Suitability of structure and behavior languages for different purposes.

To provide a detailed explanation of some specific cases in Table 1, we further discuss modeling the data structure, data validation and GUI pages in MaCoCo.

a) The **(domain) data structure** of the MaCoCo project is a static model, and it is used by the other parts of the application. The basic data structure is fixed so that the rest of the application can rely on it during run-time. The data structure already indicates the use of a structural language.

b) To describe **data validation**, we use expressions to define a valid state of an object. To enable the modeling of each of the invariants, a language has to have both: a structure part, where new invariants can be defined, and a behavior part, where the expressions for the validation are written. Using only one language family's characteristics is not enough to fulfill this requirement, thus we use a language, which can combine both, but a combination of the languages could also be used. This is expressed in Table 1 by the fact that both language families are marked as partially suitable.

c) **GUI** pages of the MaCoCo application assume that the structure of the data which should be shown is defined. A GUI language thus has to indicate which data should be used and how it is visualized. Both of those requirements can be handled by structure languages. Specific logic, *e.g.,* hiding elements, if the user does not have the required permission, has to be written in some kind of behavior language. To handle both aspects simultaneously, a language that has traits of structure and behavior language families is used in the MaCoCo project.

Practical Application: Structure languages are chosen to model, *e.g.,* data, elements, architectures, classes. Behavior languages could be further divided into aspects of behavior, *e.g.,* states, processes, or communication sequences. Functional languages are mainly used to describe algorithmic information.

In our approach, we have started with structural aspects, as they seemed to provide us with the greatest reduction of handwritten code.

4.5. Activity 4: Time Phase Identification

Another aspect to be considered is the point in time when the models of the language are to be processed. This depends partially on the problem aspect, as shown in the following examples:

a) The **data structure** of our information system is static, *i.e.,* it is already available at run-time and does not change. Therefore, corresponding models are processed during the compile-time of the application.

b) The **validation** is performed during the run-time of an application and it makes sense to evaluate validation models at the time when the check needs to be performed. However, in our MaCoCo example, the validation infrastructure is connected to a data structure, which is generated at compile-time. Thus, it is easier to ensure consistency of data types used in the validation infrastructure if models are processed at compile-time.

c) The **GUI** of the web application is static, and the functionality it provides has to be available at run-time. Although scenarios in which parts of the user interface model are interpreted at run-time are feasible, we choose code generation at compile-time to simplify GUI development of the MaCoCo application.

Practical Application: Some languages, e.g., process models, can be used for generation at compile-time but also during run-time if they are interpreted. The decision which time phase fits more influences the technological needs for changeability of the system architecture. The use of generators results in the need for re-generation if changes occur. This means a need for high changeability of the models would require a high degree of automation for the generation, deployment, and operation process. To interpret models at run-time influences the system architecture, as interpretation would require a process engine.

If handling models of a language during run-time does not bring an advantage, such models are to be processed at compile-time, which makes it easier to implement and test the end product.

4.6. Activity 5: Concrete Language(s) Decision

Once the problem is assessed and multiple language families are known, we can choose DSLs that best match our requirements.

a) The primary language for the modeling of the **data structure** in MaCoCo is CD4A. It is specifically designed to define the domain objects and associations between them. In general, models of other structure languages also have the potential to be used as a source for the generation of the data structure, *e.g.,* GUI-DSL can be used to generate classes describing data to be shown on a web page.

b) The **validation** rules in MaCoCo are best expressed with OCL/P, as the language allows one to define constraints on the domain objects and their relations.

c) For modeling the **GUI**, we did not find an appropriate textual modeling language. Thus it was clear, that we had to proceed with defining some example models showing what we would need (activities 7), implementing the DSL (activity 8), and defining the relationship between these models and already existing ones (activity 6). This process included several iterations in-between these activities.

Practical Application: There are standardized languages, such as UML or SysML, which are good candidates to choose from. Trying to sketch solutions for some use cases with models of these languages helps to understand whether the language is expressive enough to tackle a problem.

4.7. Activity 6: Define Language Interplay

In this particular step, several languages are observed at the same time, and the focus lies on their interaction.

For problem aspect a), domain objects in MaCoCo are described in a data structure modeled in the CD4A language. The data validation aims to assure that elements of data types defined in a CD4A model fulfill certain properties that need verification before the migration of new data. This causes an overlap of the problem aspects a) (data structure) and b) (data validation). Therefore, models in OCL/P reference data types defined in the data structure modeled in CD4A, which we achieved using language aggregation. Thereby, OCL/P constraints can be applied to the already defined domain objects. Figure 9 shows a simple class diagram with three attributes and an OCL/P invariant which defines that a person object is in a valid state if the age attribute is greater or equal to zero, and if the value of the birthday attribute is after today.

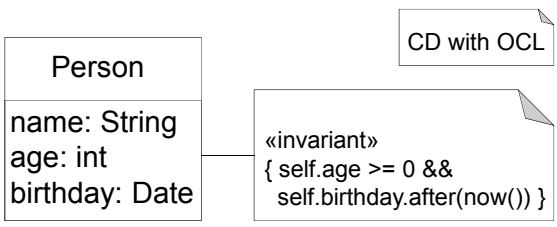

Figure 9 Example class diagram with a person class and OCL constraints

The user interface of the MaCoCo application, problem aspect c), which is generated from models in the GUI-DSL is based on the Model-View-ViewModel (MVVM) pattern (Garofalo 2011). As the user interface needs to provide or retrieve data information from the user, the problem aspects a) (data structure) and c) overlap. We solved the overlap by defining the View of the MVVM in a GUI-DSL model that references elements of the CD4A model representing the data structure, i.e., the solution to problem aspect a). To this effect, the GUI-DSL utilizes language aggregation to enable GUI developers to reference elements of a CD4A model. A model transformation derives the ViewModel of the MVVM from the CD4A model.

Practical Application: An analysis based on a use case example helps in identifying possibilities to define language interplay. Discovered shortcomings of existing languages can be resolved by a model of the other language or by an extension of the language.

Our experiences have shown, that it is important to keep the models as simple as possible and try to avoid duplication of information to reduce the development effort for developers.

4.8. Activity 7: Example Model Creation

Having executed previous steps, one can create models to describe the target system. A set of example models needs to represent solutions covering most of the elements of the corresponding problem domain.

a) The CD4A models describe classes and associations specific to an application domain. In the MaCoCo application, such models describe *e.g.,* accounts, budget classes, and the associations between them.

b) Similarly, the example OCL models define validation rules, *e.g.,* budget balance has to be higher than a certain threshold.

c) A GUI model defines the appearance of the application, the arrangement of visualization components, and how they are linked to the data structure of the application. For example, GUI models define web interfaces for account management or the navigation bar.

Practical Application: Example models should cover as many different scenarios as possible to explore the usage of a language. Within retrofitting approaches, these different scenarios can be identified by using different code snippets.

Our experiences with retrofitting alongside adding new functionalities to the application have shown, that it is not impossible to cover all scenarios with example models. However, the time spent on example models is not lost within generative or interpretative approaches, as they are reused and further evolved into the finally used models.

4.9. Activity 8: (Optional) Language Engineering

The example models serve as a source for creating a grammar, which defines a language.

a) Granted that the DSL CD4A already exists, we focus more on a language extension. For example, in the MaCoCo project, the CD4A domain model defines a database schema, which

has a direct mapping from the class diagram in the case of a relational database. The classes and associations of the model spawn database tables, which store data about domain objects and their relations respectively. The entries of the tables can be dependent on each other and can be configured to observe changes in other entries to be updated or to ignore the changes. This information is not present in CDs, therefore, the language is extended to include this information using annotations. An annotation is a string with a specific, identifying syntax, which can be attached to associations, in this example to specify the database configuration.

c) Another example from the MaCoCo application is the creation of the GUI-DSL. The GUI of the MaCoCo displays a multitude of different data sets but uses a limited set of GUI-components to do so. Therefore, the GUI-DSL was built to reduce the implementation of the user interface to a simple definition of both layout and the data source of the components within a model.

Practical Application: Our experiences have shown that the reuse of existing languages is important to be able to have a maintainable set of languages in the long run. Creating modular languages allows for creating extensions for specific needs. However, if these languages have to be changed or somehow 'misused', it is a better idea to create a new DSL for a certain purpose as we had to do it for describing the GUI.

4.10. Activity 9: Generator Engineering

To process the models and to produce the application code, the generator has to be implemented or reused.

For example, a generator for a validation infrastructure, problem aspect b), had to be defined for MaCoCo in addition to our generator for creating a data structure (Adam et al. 2020), problem aspect a). The validation infrastructure consists of the classes and their methods, which define the validation logic. To produce such code, we had to create additional templates describing classes, methods and method bodies, as well as a data structure describing these entities. Such templates and data structure are reused from the data structure generator, since it also produces classes and methods. Parts, which are specific for the validation infrastructure, such as validation logic, are not present in the generator for the data structure, and require additional implementation in form of new templates and generator code.

Regarding problem aspect c), we also had to implement a generator for the GUI and respective templates for the frontend of the application. As the code for the frontend already existed, these code parts were used as a basis for the creation of the templates. The generator engineer had to ensure that the generated code fits together with the generated code from the data structure generator.

Practical Application: A generator usually consists of the same parts, such as a model parser, AST transformations, and templates. Specific implementations vary from case to case but it helps the generator engineer to learn from existing generator code. The generator for the GUI-DSL, problem aspect c), was newly implemented as nothing similar existed before (Gerasimov, Michael, et al. 2020). Our experiences have shown that the reuse of generators or generator parts (with or without adaptations) is preferable as it is less time consuming than developing a new one (Adam et al. 2020). Developing a modular generator increases its reusability or at least of parts of it. For detailed lessons learned during generator engineering whilst developing MaCoCo, we refer the readers to (Adam et al. 2020).

4.11. Activities 10-16

Example models created during Activity 7 are used as a basis for the model engineering (**Activity 10**). The example models for the MaCoCo application were extended to become fully-fledged models used in the application. In this activity, new models are produced or existing ones adapted.

Practical Application: Creating a model is an iterative process, where each step consists of refining the model and regenerating the code. In the practical application within the MaCoCo application we have seen, that additions to models are less time-consuming than changing them. Changing models results in additional changes in the handwritten code and, especially changes in the data structure model, result in the need for data migration within the running application in a new release.

A complete model is used as an input for a generator (**Activity 11**), which produces code. The model is processed by one or multiple generators, which can handle respective DSLs. For example, in MaCoCo the CD4A domain model is used as an input for the data structure, GUI models are handled by GUI generator to output code for the web pages, *etc*.

Practical Application: As model engineering (activity 10) is an iterative process, the input models will change during the development of an application. This requires continuous re-generation and checking the generated code and the running application. Thus, it is important to provide fast generation cycles and a high degree of automation.

If the generated code does not provide all of the necessary functionality, it needs to be further extended by handwritten code (**Activity 12**). In our example the data structure is enhanced with custom getters and setters for the classes, implementing more complex business logic, and a user interface has extended functionalities to handle complex logic behind user interactions.

Practical Application: The generation process has to ensure, that handwritten additions are not overwritten when regenerating the application. One-shot generation is not useful for agile engineering processes, as most applications and their models evolve over time.

To create real-world applications, in our experience, al-

ways needs handwritten additions as specific user requirements within the business logic are not generated completely and will not be replaced by models in the future, as generation makes only sense if there exists repetitive code.

The individual components, as well as the combination of components and the whole system, are tested afterward (**Activity 13**). In MaCoCo, tests are also a subject for generation and are used alongside handwritten tests. For example, aside from the web page components, GUI models are used as a source for the generation of simple unit tests for these components.

Practical Application: The usage of existing models can ease the manual effort, as the tests or useful infrastructure for testing can be generated. This reduces the time needed for adaptions as changes in the models and handwritten code result in the need for changing hand-written tests.

Our experiences have shown that a certain amount of hand-written tests together with a refactoring process to improve the existing ones was needed at the beginning. This helped in gaining more experience in what common infrastructure aspects were needed for the tests. The generation of these infrastructure aspects was then done in a second step.

Changes in the code, including the models, languages, or the generator result in the need for migration of a system (**Activity 14**). Depending on what has been changed, different parts of the migration process can be affected. Changes in the models describing the data structure of an application may trigger data migration, *e.g.,* if a CD4A model is used to generate a database schema, the data needs to be transformed when the model is changed.

Practical Application: Until now, we did not fully automate this step and are still in the process of improving it. A high degree of automation is surely desirable but the amount of manual effort increases with the increasing complexity of the changes, especially for migrating data.

Since we already have models that describe, *e.g.,* the schema in a database, we use them to facilitate the migration process. We use delta modeling to represent differences of system variants or versions (Haber, Hölldobler, et al. 2015). Thus it is easier for us to identify changes and aspects which need to be migrated. In practice, the problem lies in the nitty-gritty which means that each migration process is accompanied by manual checks and additions to the proposed data migration code, as well as a comprehensive backup strategy.

A continuous deployment process (**Activity 15**) is often described in a model such as a configuration file, which defines phases of deployment and additional information for executing the phases, *e.g.,* a gitlab configuration file describing continuous integration workflow template. Other model-like configuration files can be used on individual deployment phases. For example, MaCoCo uses Dockerfiles (Docker Inc. n.d.) to set up containers for compiling and running the application.

Practical Application: Typically continuous deployment pipelines support fully automated deployments including steps such as build, test, and deployment (Humble & Farley 2010). Again, there exist MBSE approaches for supporting this activities (Ferry & Solberg 2017).

Maintenance (**Activity 16**) is a collective activity, which potentially includes all other activities and is continuously performed to keep the state of a system operational. The activities A1 to A15 are used to form the agile development process as shown in Figure 2.

Practical Application: Within MBSE approaches, this would also include the maintenance of the languages and generators created. Aspects such as version changes of the underlying language workbench and used DSLs might become important at a certain stage of a project, e.g., if a new version would allow expressing more in models as a previous version. The benefits of it must be well assessed together with the time required for the change, as it does not mean an increase in functionality for users in the first step within an evolving system.

To sum up, our experiences from this development process have shown that it is highly complex to replace existing code with generated code. However, in the long run, aspects that could be reached by using generators, e.g., automatic consistency between backend and frontend of an application or a modular system architecture, also increase the quality of an application and ease extensions and maintenance.

5. Discussion, Limitations and Threats to Validity

To the best of our knowledge, currently, there does not exist a concrete methodology for retrofitting generative aspects into ongoing projects at any development phase. Retrofitting generative aspects in this context means integrating and deploying implementations into an existing EIS that are generated from models. Our methodology proposes criteria, based on which one can decide whether new modeling languages have to be introduced into the ongoing development, existing languages need to be adapted or reduced to create models from which the code can be generated. These criteria give a guideline on how to choose languages in a way that minimizes the effort of changing implementations of modeling languages or tooling such as code generators and aims to maximize the reuse of such implementations. Furthermore, our approach also considers integration and deployment efforts of generated implementations into existing EIS and provides guidelines on how to minimize these efforts.

Validation method & Limitations. To evaluate our approach we have identified the "validation by example" method (Shaw 2003) as the only appropriate method. We apply the methodology to a real-world application. Because of the large scale and several years of development, approaches using comparison are not feasible for us: (1) It is not possible to time the

development of the same application by two teams in parallel where one team applies the methodology and the other team does not. (2) To use two different applications and apply them on one and not on the other is again not comparable due to the differences in application sizes and complexity. (3) To apply it on several applications after each other would be meaningful, but it is unrealistic due to the long time needed for this investigation. Furthermore, human factors, such as changing team members, might lead to bias in long-term analyses. To validate it by experiment (Wohlin et al. 2003) was not appropriate, as we do not consider the methodology in a laboratory setting or were able to assign subjects to different treatments at random. To validate it by a case study (Wohlin et al. 2003; Runeson & Höst 2009) is also inappropriate, as it is not feasible to collect data for statistical analyses from developers, who helped to improve the methodology itself and are thus biased.

Artifacts & Limitations. It was not possible to provide larger application artifacts as the code of the application is not open source, subject to the intellectual property rights (IPR) of our clients, and the application is only reachable via an internal network due to security reasons. The IPR includes the code as well as the models, thus, it is not possible to provide such artifacts. Former publications, *e.g.,* (Gerasimov, Heuser, et al. 2020) and (Gerasimov, Michael, et al. 2020) also do not provide complete models, but give some short examples of used models.

We have shown the use of our methodology in case the system or a part of the system already exists and should be replaced using generative approaches (brown-field). If the system to be created does not exist (green-field), examples for the code to be generated (needed in step 8) are also not available. This makes the engineering process of the generator more time-intensive but still possible.

Relationship between reuse of DSLs and effort. If the DSLs already exist, their application is easier and less time-consuming than creating a new DSL and all needed tooling. Also, the generator which can handle the models needs to be created using this new DSL. Dependent on the domain and languages used, the effort for writing a generator is quite high. In our experience with the development of new languages and generator infrastructure, we calculate comparable time for writing a generator, as it would have been for writing the application (see (Adam et al. 2020)).

External validity. Within this paper, we have only shown the feasibility of our methodology for one application and only for one specific type of system, namely EISs. We have applied the methodology to a full-size real-world project (Gerasimov, Heuser, et al. 2020) from the finance and controlling domain. Although only a specific case has been evaluated, our methodology provides a general description of the activities, necessary for retrofitting generative aspects into an existing application. We can provide a general description since each activity can be individually observed in other types of projects, where the experience is gathered and further compiled to be used in our example, presented in this work. We expect that our methodology could be easily applied to other domains, *e.g.,* information systems for Cyber-Physical System (CPS), smart production systems, or energy systems. However, this needs to be investigated further.

Technology restriction. We use the language workbench MontiCore for all used DSLs and for defining new DSLs. It allows us to integrate languages in a very convenient way (France & Rumpe 2007). Thus, the "enhanced tooling challenge" is less important for our application of the methodology described in Section 4. MontiCore provides an efficient meta-toolset for each DSL and, thus, facilitates language engineering activities.

To use languages without tooling that supports language composition and includes common infrastructure is more time-intensive but still allows one to make use of our methodology. In this case, to use only one modeling language for a restricted problem domain might be an option but full generative support of large parts of the system will not be possible in this case.

Team size limitations. As discussed before, there are differences in language engineering and application engineering. In larger companies, these two parts are typically developed by different teams. In Figure 4, we have discussed different roles required for these parts. Therefore it is easier to fill these roles when you have larger development teams. In smaller development teams, one person has to take over several tasks. Our methodology has no restrictions on the size of the development team(s). Our methodology can be applied flexibly in such a way that any activities can be carried out as needed. Thus, there are no restrictions on the repetition of activities.

Our methodology helps in overcoming some challenges for MBSE, *e.g.,* in (France & Rumpe 2007; Bucchiarone et al. 2020), namely

- *Relating models from heterogeneous DSLs (DSL-Babel challenge):* In activity 6, developers have to define the language interplay and systematically consider the relationships between models of same and different DSLs on a conceptual level. This includes methods to ensure consistency between related models. The interplay has to result in respective generator engineering within activity 9.
- *Support Evolution of Models and DSLs:* When evolving a model, the relations of this model to models of another DSL have to be updated. By defining the language interplay, it becomes clear which models have to be updated. During the generation process, the developer can be informed if models are not consistent anymore. A further extension could be the use of model change management within the toolchain, including semantic and syntactic differencing on models.
- *Reusing DSLs and generators:* Reuse is an important aspect of DSL engineering and application engineering. In Activity 5, DSLs which already exist can be chosen or existing ones may be extended. We provide ideas on how to make this decision. Activity 9 provides some ideas for when the reuse of generators or reuse with adaptions should be considered. This reduces the development effort.

To conclude, our example has shown that our method is applicable for retrofitting generative approaches into data-centered client/server architectures. The proposed conceptual model and methodology can be used to evaluate further retrofitting for similar or other system architectures.

6. Related Work

To the best of our knowledge, there exist no concrete methodologies for retrofitting implementations generated from models into a project. In particular, there exist no methodological considerations on how and when new or adaptations of existing modeling languages have to be introduced into a development process. (Paige et al. 2017) present four steps of a quasi typical model-driven engineering process: to construct, or to select modeling languages, to build, persist, and manage models. This suggested process is limited to the handling of models and lacks concrete activities for problem decomposition and assessment, as well as DSL engineering and generator engineering.

The literature review presented in (Neis et al. 2019) summarizes the state of the art of MBSE in software engineering for power systems applications. Even though the targeted research questions aim to reveal how source models are processed into target models or code, none of the approaches mentioned produce fully functional software applications. The review aims to signpost modeling languages used in context of the application domain of power systems, which is possibly helpful during Activities 3-5. The problem domain and mappings to the solution domain are considered in more detail in (Czarnecki 2005). This approach targets software product lines in general and emphasizes a strong relation to MBSE. Our methodology on the other hand is tailored to engineering business information systems by generating fully functional applications from a variety of modeling languages processed by a set of connected code generators. The method proposed in (Rogozov et al. 2020) modularizes the system into functions according to the enterprise's setup. The approach utilizes generative or model-driven techniques.

Several publications address one or more steps of our methodology in detail.

Activity 1: Problem Division. As pointed out in (Tarr et al. 1999), complex systems need decomposition along multiple dimensions. Decomposing a problem domain is a state of the art practice in requirements engineering (Maiden & Sutcliffe 1996). There exist approaches to automatically classify requirements as functional (FR) and non-functional (Kurtanović & Maalej 2017), distinguish between requirements and information (Winkler & Vogelsang 2016), by using WiKis to elicit, semantically structure and classify requirements (Riechert & Berger 2009).

To the best of our knowledge, approaches so far cluster existing requirements according to problem aspects. Additionally, many approaches exist that aim to systematize identifying modules of a system, *e.g.,* (Parnas 1972; Rogozov et al. 2020). A need for a system to be modular is an aspect of a problem and has to be addressed by suitable modeling languages, such as *e.g.,* Architecture Description Languages (ADLs) (Butting et al. 2017; Medvidovic & Taylor 2000; Medvidovic et al. 2007).

Activity 2: Problem Assessment. A methodology that utilizes aspect orientation to overcome the problem-implementation gap is proposed in (Elrad et al. 2002). The approach proposes decomposing the problem, which serves as a means to identify functional components of the system, whereas in our approach, the problem decomposition serves as a means to identify mechanisms to handle the problem, such as *e.g.,* models, or libraries.

Activity 3: Language Family Identification. The literature review in (Neis et al. 2019) investigated, among others, a research question regarding which modeling languages are used in model-driven and generative approaches of applications in the power systems domain. The languages found are classified into different language categories. However, the result is more of an overview of the modeling languages used in that application area and the selection of a language family is not embedded into an extensive methodology for generative model-driven engineering.

Activity 4: Time-Phase Identification. The software development life-cycle is divided into different phases, such as design-time, compile-time or run-time (Pusztai et al. 2019). Time-phases are also considered within software product line engineering. The field of software product lines commonly divides the engineering process into two major phases, *i.e.,* domain engineering, and application engineering (Apel & Kästner 2009; Czarnecki 2005; Böckle et al. 2005; Kang et al. 1998). In (Czarnecki 2005), reusable DSLs and respective generators are created or identified as a part of the domain engineering phase. Our approach however, details the engineering of DSLs in activities 3 to 6, which enables systematic reuse or creation of DSLs in general. The approach focuses on the generation of members of a system family expressed as a model, but does not necessarily make models the primary development artifacts as is the case in our approach. Our methodology aims to enable systematic retrofitting of generated aspects into already running systems, which may be a part of a system family developed using (Czarnecki 2005). An approach to interpret feature models according to a development phase has been proposed in the context of semantic differencing in (Drave, Kautz, et al. 2019), which addresses the fact that changes conducted at early development stages have a different intent than at later development stages.

Activity 5: Concrete Language(s) Decision. To the best of our knowledge, there exists no publication discussing which DSL to choose based on problem aspects. For EISs, *e.g.,* (Kolovos et al. 2019) present three interweaving DSLs for the design, deployment and manipulation as well as querying, evolving, and analyzing heterogeneous databases. Another example is (Brambilla et al. 2016) proposing a modeling language for designing web and mobile languages at once that enables the propagation of the evolution of an application's user interface across all platforms equally. For power system applications, the survey in (Neis et al. 2019) reveals many approaches that utilize such systems.

Such literature presenting custom DSLs, profiling, or the direct application of general-purpose languages such as UML or SysML to tackle specific problem domains or problem aspects of specific problem domains is common, however, respective publications often fail to point out the criteria that enable the reuse of proposed languages. These are only implicitly given by describing the problem domain or the aspects of the problem domain.

Activity 6: Define Language Interplay. Language interplay within model-driven generative engineering may be defined on

the language level, by mechanisms such as language aggregation, embedding, and inheritance (Haber, Look, et al. 2015a). However, the interplay may also take place on the artifact or model level, as identified in (Méndez Fernández et al. 2019; Butting et al. 2018)

Activities 7 and 10: Example Model Creation and Model Engineering. Model management is crucial during this activity and faces many challenges (Bucchiarone et al. 2020). For agile development, and to assure proper functioning of code generators, automation of *e.g.,* model consistency is of crucial importance. An approach based on a conceptual meta-model and relations between meta-models of the utilized languages is proposed in (Klare & Gleitze 2019).

Managing model evolution both syntactically and semantically enables version control and stepwise refinement of model artifacts during the development process, *e.g.,* (Maoz et al. 2010; Alanen & Porres 2003; Fahrenberg et al. 2011; Drave, Kautz, et al. 2019).

Another way is to derive models from existing code. Model-Driven Reverse Engineering (MDRE) (Fontana et al. 2020) starts from the source code or data base (Ristić 2017) and creates models. These can be analyzed and/or used for model-based processes in forward engineering.

Activity 8: Language Engineering. Language engineering consists of many different parts, including language aggregation, embedding, inheritance, extension, and restriction (Hölldobler & Rumpe 2017).

Extend or adapt a DSL. Concepts that generalize reusing DSLs, possibly through extensions are still in their infancy. The approach presented in (Combemale et al. 2018) formalizes a notion of language concerns that modularizes the implementation of a DSL and provides meaningful concepts for its extension. The extendability and adaptability of a DSL strongly depend on the structure of its definition and its infrastructure. Thus, reuse must already be considered when creating a DSL.

Create a DSL. Designing DSLs is an effortful task, for which (Karsai et al. 2009) gives a detailed list of guidelines, and emphasize reusing existing languages or parts of languages. There exists a variety of publications about Software Language Engineering (Kleppe 2009; Hölldobler et al. 2018), and on how to design new DSLs that fit specific purposes. Several papers discuss how best to combine concrete languages such as UML CD and OCL. A framework for model composition is provided in (Kienzle et al. 2019), which provides mechanisms that enable elaborate reuse of modeling languages and adaptation for specific problem aspects.

None of these approaches integrate the reuse or composition of models, or modeling languages in a model-driven generative methodology that aims to automatically produce fully functional business information systems from models of heterogeneous languages using a generator framework such as MontiGem.

Activity 9: Generator Engineering. Using example models and code examples as a reference when engineering code generators have proven efficient in full-size real-world MBSE projects (Adam et al. 2020), which is why we consider model creation and generator engineering to go hand in hand. The approach presented in (Eikermann et al. 2019) utilizes transformations and templates for systematic reuse of code generators. Another example presented in (Sujeeth et al. 2013) demonstrates reuse of a code generator infrastructure and related functional libraries in a form of a framework, which transforms models of different DSLs into a common representation. The code generator can also be structured to process the models in several phases and to enable model-to-model transformation, which has shown to be beneficial for generator reusability (Brambilla et al. 2016).

Activities 11 and 12: Generation and Handwritten Additions. In (Grönniger et al. 2006) we discuss mechanisms (TOP-mechanism) to enable a repetitive generation which still keeps generated and handwritten code separated, while integrating them into the product. There are also other techniques and mechanisms that allow the adaption and adjustment of generated code (Greifenberg, Hölldobler, et al. 2015).

Activity 13: Testing. Our methodology is model-based and utilizes code generators to automatically generate large parts of the EIS under development. Therefore, our methodology employs Model-Based Testing (MBT) during the testing activities. Existing techniques for MBT can be applied in this activity to generate test suites and test data from design models, or specified test models.

MBT utilizes models for the definition of test cases (Rumpe 2003; Dalal et al. 1999). Due to the model-based nature of our approach, MBT integrates very well with the proposed methodology, also mentioned in Section 3.3 and 4.11. Therefore, not only generating code but also test cases is a possible extension. In the literature, various techniques for MBT exist. A survey for techniques published between 1990 and 2007 is provided in (Dias Neto et al. 2007). These classify into techniques that

1. annotate design models to enable test case generation (Rumpe 2003; Vieira et al. 2006; Dai et al. 2004),
2. utilize DSLs to model test cases (Hartman & Nagin 2005; Briand & Labiche 2002), and
3. utilize intermediate models from which test cases are generated (Pretschner & Philipps 2005; Utting & Legeard 2007; Wieczorek & Stefanescu 2010).

The approach proposed in (Wieczorek & Stefanescu 2010) utilizes MBT for testing GUIs in service-oriented EIS. A process that utilizes a customized UML to specify business processes together with MBT techniques for generating test cases from these design models is introduced in (Mlynarski 2010). Approaches that apply MBT to generate test cases from behavioral diagrams such as UML's or SysML's activity diagrams or statecharts are, *e.g.,* proposed in (Drave, Greifenberg, et al. 2019; García-Domínguez et al. 2013; Wang et al. 2004; Kim et al. 2007; Shirole et al. 2011; Offutt & Abdurazik 1999). Test case generation from OCL specifications is proposed, *e.g.,* in (Brucker et al. 2011). Most of these approaches use code generators to obtain executable test cases. Managing test data is a major challenge in testing EISs. An MBT approach for providing test data is *e.g.,* proposed in (Wieczorek & Stefanescu 2010). Another approach that utilizes OCL specifications to generate test data is given in (Brucker et al. 2011).

Activity 14: Migration. Adaptations during the development cycles could cause the need to migrate different parts of the application. Changes in external dependencies may make it necessary to change the infrastructure, *e.g.,* the database technology, or the RTE. The evolution of the used languages and generators could lead to a migration of used models (Schonbock et al. 2015). Changes in the models need to be considered (Cicchetti et al. 2013), *e.g.,* when the database schema depends on a given data structure, meaning the schema and the existing data needs to be migrated (Herrmann et al. 2018).

Activity 15: Continuous Deployment. Continuous deployment is "the practice of continuously deploying good software builds automatically to some environment, but not necessarily to actual users" (Fitzgerald & Stol 2014). Continuous deployment is advantageous for growing projects (Savor et al. 2016) and has found several model-based approaches. For example, continuous deployment models are used in Cloud-based (Ferry & Solberg 2017) and IoT systems (Ferry & Nguyen 2019) to support dynamic provisioning, deployment, and adaptation of a system.

Activity 16: Maintenance. Maintenance effort can make up half of the total effort invested in a software system over its total lifespan (Rehman et al. 2018). Multiple projects evaluate methodologies on how to incorporate agile development methods, not only in the initial software engineering process, but also in the following maintenance phase (Rehman et al. 2018; Polo et al. 1999). (Lenarduzzi et al. 2017) give an overview on the usage of model-based approaches to maintain software of the last forty years.

7. Conclusion and Future Work

To evolve applications during their lifetime, continuous delivery and deployment in an agile way is in any case challenging (Riungu-Kalliosaari et al. 2016). If engineers want to integrate generative aspects in already existing applications, they benefit from a supporting methodology, which leads through a variety of decisions.

This paper introduces a methodology for retrofitting the model-based approach into existing systems with three phases, namely problem analysis and decomposition, DSL engineering as well as application engineering and operation. The proposed activities start with the division of large, complex problems into smaller problem aspects to be handled by DSLs or GPLs. Each problem aspect is assessed, the right language family and time phase is identified and a concrete language chosen. After defining the language interplay and creating some example models new DSLs have to be created or existing ones are extended. The next steps consider application engineering and operation of the resulting systems, including generator and model engineering.

This methodology helps to overcome the problem-implementation gap which is especially challenging for complex and large software systems. To use a divide-and-conquer approach for complex problems supports developers to make the right decisions when using model-based software engineering and code generators.

References

Acher, M., Heymans, P., Collet, P., Quinton, C., Lahire, P., & Merle, P. (2012). Feature Model Differences. In *Advanced Information Systems Engineering (CAiSE'12)* (Vol. 7328 LNCS, pp. 629–645). Springer. doi: 10.1007/978-3-642-31095-9_41

Adam, K., Michael, J., Netz, L., Rumpe, B., & Varga, S. (2020). Enterprise Information Systems in Academia and Practice: Lessons learned from a MBSE Project. In *40 Years EMISA (EMISA'19)* (Vol. P-304, pp. 59–66). Gesellschaft für Informatik e.V.

Adam, K., Netz, L., Varga, S., Michael, J., Rumpe, B., Heuser, P., & Letmathe, P. (2018). Model-Based Generation of Enterprise Information Systems. In *Enterprise Modeling and Information Systems Architectures (EMISA'18)* (Vol. 2097, pp. 75–79). CEUR-WS.org.

Alanen, M., & Porres, I. (2003). Difference and union of models. , *2863 LNCS*, 2–17. doi: 10.1007/978-3-540-45221-8_2

Aniche, M., Yoder, J., & Kon, F. (2019). Current challenges in practical object-oriented software design. In *41st Int. Conf. on Software Engineering: New Ideas and Emerging Results (ICSE-NIER'19).* IEEE. doi: 10.1109/ICSE-NIER.2019.00037

Apel, S., & Kästner, C. (2009). An Overview of Feature-Oriented Software Development. *Journal of Object Technology*, *8*(5), 49–84.

Böckle, G., Pohl, K., & van der Linden, F. (2005). A Framework for Software Product Line Engineering. In *Software Product Line Engineering: Foundations, Principles, and Techniques* (p. 19-38). Springer. doi: 10.1007/3-540-28901-1_2

Brambilla, M., Mauri, A., Franzago, M., & Muccini, H. (2016). A model-based method for seamless web and mobile experience. In (p. 33-40). doi: 10.1145/3001854.3001857

Breu, R., Grosu, R., Huber, F., Rumpe, B., & Schwerin, W. (1998). Systems, Views and Models of UML. In *Proceedings of the Unified Modeling Language, Technical Aspects and Applications* (pp. 93–109). Physica Verlag, Germany. doi: 10.1007/978-3-642-48673-9_7

Briand, L., & Labiche, Y. (2002). A UML-Based Approach to System Testing. *Software and Systems Modeling*, *1*(1), 1–57. doi: 10.1007/s11334-005-0006-0

Brucker, A. D., Krieger, M. P., Longuet, D., & Wolff, B. (2011). A specification-based test case generation method for UML/OCL. In *Models in Software Engineering (MODELS 2010)* (Vol. 6627 LNCS, pp. 334–348). Springer. doi: 10.1007/978-3-642-21210-9_33

Bucchiarone, A., Cabot, J., Paige, R. F., & Pierantonio, A. (2020). Grand challenges in model-driven engineering: an analysis of the state of the research. *Software and Systems Modeling*, *19*. doi: 10.1007/s10270-019-00773-6

Burgueño, L., Ciccozzi, F., Famelis, M., Kappel, G., Lambers, L., Mosser, S., … Wimmer, M. (2019). Contents for a Model-Based Software Engineering Body of Knowledge. *Software and Systems Modeling*, *18*(6), 3193–3205. doi: 10.1007/s10270-019-00746-9

Butting, A., Greifenberg, T., Rumpe, B., & Wortmann, A.

(2018). On the Need for Artifact Models in Model-Driven Systems Engineering Projects. In *Software technologies: Applications and foundations* (Vol. 10748 LNCS, pp. 146–153). Springer. doi: 10.1007/978-3-319-74730-9_12

Butting, A., Haber, A., Hermerschmidt, L., Kautz, O., Rumpe, B., & Wortmann, A. (2017). Systematic Language Extension Mechanisms for the MontiArc Architecture Description Language. In *Europ. Conf. on Modelling Foundations and Applications (ECMFA'17)* (pp. 53–70). Springer. doi: 10.1007/978-3-319-61482-3_4

Cicchetti, A., Di Ruscio, D., Iovino, L., & Pierantonio, A. (2013). Managing the evolution of data-intensive Web applications by model-driven techniques. *Software & Systems Modeling*, *12*(1), 53–83. doi: 10.1007/s10270-011-0193-0

Clark, T., van den Brand, M., Combemale, B., & Rumpe, B. (2015). Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages* (pp. 7–20). Springer. doi: 10.1007/978-3-319-26172-0_2

Combemale, B., Kienzle, J., Mussbacher, G., Barais, O., Bousse, E., Cazzola, W., ... Wortmann, A. (2018). Concern-Oriented Language Development (COLD): Fostering Reuse in Language Engineering. *Computer Languages, Systems & Structures*, *54*, 139 - 155. doi: 10.1016/j.cl.2018.05.004

Combemale, B., Mernik, M., & Rumpe, B. (Eds.). (2017). *Software Language Engineering (SLE'17).* Vancouver: ACM Sigplan.

Czarnecki, K. (2005). Overview of Generative Software Development. In *Unconventional Programming Paradigms.* doi: 10.1007/11527800_25

Dai, Z. R., Grabowski, J., Neukirchen, H., & Pals, H. (2004). From Design to Test with UML. In (pp. 33–49). Springer. doi: 10.1007/978-3-540-24704-3_3

Dalal, S. R., Jain, A., Karunanithi, N., Leaton, J. M., Lott, C. M., Patton, G. C., & Horowitz, B. M. (1999). Model-based testing in practice. In *21st Int. Conf. on Software Engineering (ICSE'99)* (Vol. 2, pp. 285–294). ACM Press. doi: 10.1145/302405.302640

Degueule, T., Combemale, B., Blouin, A., Barais, O., & Jézéquel, J. M. (2015). Melange: A meta-language for modular and reusable development of DSLs. In *Proc. ACM SIGPLAN Int. Conf. on Software Language Engineering (SLE 2015)* (pp. 25–36). ACM. doi: 10.1145/2814251.2814252

Dias Neto, A. C., Subramanyan, R., Vieira, M., & Travassos, G. H. (2007). A Survey on Model-based Testing Approaches: A Systematic Review. In *1st ACM Int. Workshop on Empirical Assessment of Software Engineering Languages and Technologies (WEASELTech'07), at ASE'07.* ACM Press. doi: 10.1145/1353673.1353681

Docker Inc. (n.d.). *Dockerfile reference.* Retrieved from https://docs.docker.com/engine/reference/builder (last accessed: 8.7.2020)

Drave, I., Greifenberg, T., Hillemacher, S., Kriebel, S., Kusmenko, E., Markthaler, M., ... Wortmann, A. (2019). SMArDT modeling for automotive software testing. *Software: Practice and Experience*, *49*(2), 301–328. doi: 10.1002/spe.2650

Drave, I., Kautz, O., Michael, J., & Rumpe, B. (2019). Semantic Evolution Analysis of Feature Models. In *Int. Systems and Software Product Line Conference (SPLC'19)* (pp. 245–255). ACM. doi: 10.1145/3336294.3336300

Eikermann, R., Hölldobler, K., Roth, A., & Rumpe, B. (2019). Reuse and Customization for Code Generators: Synergy by Transformations and Templates. In *Model-Driven Engineering and Software Development* (pp. 34–55). Springer. doi: 10.1007/978-3-030-11030-7_3

Elrad, T., Aldawud, O., & Bader, A. (2002). Aspect-Oriented Modeling: Bridging the Gap between Implementation and Design. In *Generative Programming and Component Engineering.* Springer. doi: 10.1007/3-540-45821-2_12

Fahrenberg, U., Legay, A., & Wąsowski, A. (2011). Vision Paper: Make a Difference! (Semantically). In *Model Driven Engineering Languages and Systems.* doi: 10.1007/978-3-642-24485-8_36

Ferry, N., & Nguyen, P. (2019). Towards Model-Based Continuous Deployment of Secure IoT Systems. In *22nd Int. Conf. on Model Driven Engineering Languages and Systems Companion (MODELS-C'19).* doi: 10.1109/MODELS-C.2019.00093

Ferry, N., & Solberg, A. (2017). Models runtime for continuous design and deployment. In *Model-Driven Development and Operation of Multi-Cloud Applications* (p. 81-–94). Springer. doi: 10.1007/978-3-319-46031-4_9

Fitzgerald, B., & Stol, K.-J. (2014). Continuous software engineering and beyond: trends and challenges. In *1st Int. Workshop on Rapid Continuous Software Engineering*. doi: 10.1145/2593812.2593813

Fontana, F. A., Bruneliere, H., Müller, H. A., & Raibulet, C. (2020). Guest editors' in-troduction to the special issue on Model Driven Engineering and Reverse Engineering: Research and Practice. *Journal of Systems and Software*, *159*. doi: 10.1016/j.jss.2019.110446

Fowler, M. (2010). *Domain Specific Languages*. Addison-Wesley Professional.

France, R., & Rumpe, B. (2007). Model-driven Development of Complex Software: A Research Roadmap. *Future of Software Engineering (FOSE'07)*, 37–54. doi: 10.1109/FOSE.2007.14

Frank, U. (2013). Domain-Specific Modeling Languages: Requirements Analysis and Design Guidelines. In *Domain Engineering.* doi: 10.1007/978-3-642-36654-3_6

García-Domínguez, A., Medina-Bulo, I., & Marcos-Bárcena, M. (2013). An Approach for Model-Driven Design and Generation of Performance Test Cases with UML and MARTE. *Communications in Computer and Information Science*, *303*, 136–150. doi: 10.1007/978-3-642-36177-7_9

Garofalo, R. (2011). *Building Enterprise Applications with Windows Presentation Foundation and the Model View ViewModel Pattern*. Microsoft Press.

Gerasimov, A., Heuser, P., Ketteniß, H., Letmathe, P., Michael, J., Netz, L., ... Varga, S. (2020). Generated Enterprise Information Systems: MDSE for Maintainable Co-Development of Frontend and Backend. In *Comp. Proc. of Modellierung 2020 Short, Workshop and Tools & Demo Papers* (pp. 22–30).

CEUR-WS.org.

Gerasimov, A., Michael, J., Netz, L., Rumpe, B., & Varga, S. (2020). Continuous Transition from Model-Driven Prototype to Full-Size Real-World Enterprise Information Systems. In *25th Am. Conf. on Information Systems (AMCIS 2020)* (pp. 1–10). AIS.

Greifenberg, T., Hölldobler, K., Kolassa, C., Look, M., Mir Seyed Nazari, P., Müller, K., ... Wortmann, A. (2015). A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)* (pp. 74–85). SciTePress.

Greifenberg, T., Look, M., Roidl, S., & Rumpe, B. (2015). Engineering Tagging Languages for DSLs. In *Conf. on Model Driven Engineering Languages and Systems (MODELS'15)* (pp. 34–43). ACM/IEEE. doi: 10.1109/MODELS.2015.7338233

Grönniger, H., Krahn, H., Rumpe, B., & Schindler, M. (2006). Integration von Modellen in einen codebasierten Softwareentwicklungsprozess. In *Modellierung 2006 conference* (pp. 67–81).

Haber, A., Hölldobler, K., Kolassa, C., Look, M., Müller, K., Rumpe, B., ... Schulze, C. (2015, October). Systematic Synthesis of Delta Modeling Languages. *Journal on Software Tools for Technology Transfer (STTT)*, *17*(5), 601–626. doi: 10.1007/s10009-015-0387-9

Haber, A., Look, M., Mir Seyed Nazari, P., Navarro Perez, A., Rumpe, B., Völkel, S., & Wortmann, A. (2015a). Composition of Heterogeneous Modeling Languages. In *Model-Driven Engineering and Software Development* (Vol. 580, pp. 45–66). Springer. doi: 10.1007/978-3-319-27869-8_3

Haber, A., Look, M., Mir Seyed Nazari, P., Navarro Perez, A., Rumpe, B., Völkel, S., & Wortmann, A. (2015b). Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)* (pp. 19–31). SciTePress. doi: 10.5220/0005225000190031

Hartman, A., & Nagin, K. (2005). The AGEDIS tools for model based testing. In *UML Modeling Languages and Applications* (Vol. 3297 LNCS, pp. 277–280). doi: 10.1007/978-3-540-31797-5_33

Herrmann, K., Voigt, H., Rausch, J., Behrend, A., & Lehner, W. (2018). Robust and simple database evolution. *Information Systems Frontiers*, *20*(1), 45–61. doi: 10.1007/s10796-016-9730-2

Hölldobler, K., & Rumpe, B. (2017). *MontiCore 5 Language Workbench Edition 2017*. Shaker Verlag.

Hölldobler, K., Rumpe, B., & Wortmann, A. (2018). Software Language Engineering in the Large: Towards Composing and Deriving Languages. *Computer Languages, Systems & Structures*, *54*, 386–405. doi: 10.1016/j.cl.2018.08.002

Humble, J., & Farley, D. (2010). *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education.

Kang, K. C., Kim, S., Lee, J., Kim, K., Shin, E., & Huh, M. (1998). FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. *Annals of Software Engineering*, *5*(1). doi: 10.1023/A:1018980625587

Karsai, G., Krahn, H., Pinkernell, C., Rumpe, B., Schindler, M., & Völkel, S. (2009). Design Guidelines for Domain Specific Languages. In *Domain-Specific Modeling Workshop (DSM'09)* (pp. 7–13). Helsinki School of Economics.

Kautz, O., & Rumpe, B. (2018). On Computing Instructions to Repair Failed Model Refinements. In *Conf. on Model Driven Engineering Languages and Systems (MODELS'18)* (pp. 289–299). ACM. doi: 10.1145/3239372.3239384

Kehrer, T., Kelter, U., & Taentzer, G. (2011). A Rule-Based Approach to the Semantic Lifting of Model Differences in the Context of Model Versioning. In *Int. Conf. on Automated Software Engineering (ASE'11)*. doi: 10.1109/ASE.2011.6100050

Kienzle, J., Mussbacher, G., Combemale, B., & Deantoni, J. (2019). A unifying framework for homogeneous model composition. *Software and Systems Modeling*, *18*(5). doi: 10.1007/s10270-018-00707-8

Kim, H., Kang, S., Baik, J., & Ko, I. (2007). Test cases generation from UML activity diagrams. In *8th ACIS Int. Conf. on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD'07)* (pp. 556–561). IEEE. doi: 10.1109/SNPD.2007.525

Klare, H., & Gleitze, J. (2019). Commonalities for Preserving Consistency of Multiple Models. In *22nd Int. Conf. on Model Driven Engineering Languages and Systems Companion (MODELS-C'19)*. doi: 10.1109/MODELS-C.2019.00058

Kleppe, A. (2009). *Software language engineering: Creating domain-specific languages using metamodels*. Addison-Wesley.

Kolovos, D., Medhat, F., Paige, R., Di Ruscio, D., Van Der Storm, T., Scholze, S., & Zolotas, A. (2019). Domain-specific languages for the design, deployment and manipulation of heterogeneous databases. In *11th Int. Workshop on Modelling in Software Engineering (MiSE '19)* (pp. 89–92). IEEE. doi: 10.1109/MiSE.2019.00021

Kurtanović, Z., & Maalej, W. (2017). Automatically Classifying Functional and Non-functional Requirements Using Supervised Machine Learning. In *25th Int. Requirements Engineering Conference (RE)*. IEEE. doi: 10.1109/RE.2017.82

Langer, P., Mayerhofer, T., & Kappel, G. (2014). A Generic Framework for Realizing Semantic Model Differencing Operators. In *PSRCMoDELs* (Vol. 1258). CEUR-WS.org.

Lenarduzzi, V., Sillitti, A., & Taibi, D. (2017). Analyzing forty years of software maintenance models. In *39th Int. Conf. on Software Engineering Companion (ICSE-C'17)* (pp. 146–148). doi: 10.1109/ICSE-C.2017.122

Maiden, N., & Sutcliffe, A. (1996). A computational mechanism for parallel problem decomposition during requirements engineering. In *8th Int. Workshop on Software Specification and Design*. doi: 10.1109/IWSSD.1996.501160

Maoz, S., & Ringert, J. O. (2016). A framework for relating syntactic and semantic model differences. *Software & System Modeling*, *17*(3). doi: 10.1007/s10270-016-0552-y

Maoz, S., Ringert, J. O., & Rumpe, B. (2010). A Manifesto for

Semantic Model Differencing. In *Int. Workshop on Models and Evolution (ME'10)* (pp. 194–203). Springer. doi: 10.1007/978-3-642-21210-9_19

Mayr, H. C., & Thalheim, B. (2020). The triptych of conceptual modeling. *Software and Systems Modeling*.

Medvidovic, N., Dashofy, E. M., & Taylor, R. N. (2007). Moving architectural description from under the technology lamppost. *Information and Software Technology*, *49*(1), 12–31. doi: 10.1016/j.infsof.2006.08.006

Medvidovic, N., & Taylor, R. N. (2000). A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, *26*. doi: 10.1109/32.825767

Méndez Fernández, D., Böhm, W., Vogelsang, A., Mund, J., Broy, M., Kuhrmann, M., & Weyer, T. (2019). Artefacts in Software Engineering: A Fundamental Positioning. *Softw. Syst. Model.*, *18*(5). doi: 10.1007/s10270-019-00714-3

Michael, J., & Mayr, H. (2015). Creating a Domain Specific Modelling Method for Ambient Assistance. In *International Conference on Advances in ICT for Emerging Regions (ICTer2015).* doi: 10.1109/ICTER.2015.7377676

Mlynarski, M. (2010). Holistic model-based testing for Business Information Systems. In *3rd Int. Conf. on Software Testing, Verification and Validation (ICST'10)* (pp. 327–330). doi: 10.1109/ICST.2010.35

Möller, D., Legner, C., & Heck, A. (2011). Understanding IT transformation - an explorative study. In *19th Europ. Conf. on Information Systems (ECIS'11).*

Neis, P., Wehrmeister, M., & Mendes, M. (2019). Model Driven Software Engineering of Power Systems Applications: Literature Review and Trends. *IEEE Access*, *7*. doi: 10.1109/ACCESS.2019.2958275

Offutt, J., & Abdurazik, A. (1999). Generating tests from UML specifications. In *«UML»'99 — The Unified Modeling Language* (Vol. 1723 LNCS, pp. 416–429). Springer Verlag. doi: 10.1007/3-540-46852-8_30

Paige, R. F., Zolotas, A., & Kolovos, D. (2017). The changing face of model-driven engineering. In *Present and Ulterior Software Engineering* (pp. 103–118). Springer. doi: 10.1007/978-3-319-67425-4_7

Parnas, D. (1972). On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, *15*. doi: 10.1145/361598.361623

Pastor, O., Gómez, J., Insfrán, E., & Pelechano, V. (2001). The OO-method approach for information systems modeling: from object-oriented conceptual modeling to automated programming. *Information Systems*, *26*(7). doi: 10.1016/S0306-4379(01)00035-7

Polo, M., Piattini, M., Ruiz, F., & Calero, C. (1999). MANTEMA: A software maintenance methodology based on the ISO/IEC 12207 standard. In *Int. Software Engineering Standards Symposium and Forum (ISESS'99)* (pp. 76–81). doi: 10.1109/SESS.1999.766580

Pretschner, A., & Philipps, J. (2005). 10 Methodological Issues in Model-Based Testing. In *Model-Based Testing of Reactive Systems* (pp. 281–291). Springer. doi: 10.1007/11498490_13

Pusztai, T., Tsigkanos, C., & Dustdar, S. (2019). Engineering Heterogeneous Internet of Things Applications: From Models to Code. In *5th Int. Conf. on Collaboration and Internet Computing (CIC)* (p. 222-231). IEEE. doi: 10.1109/CIC48465.2019.00036

Rehman, F. u., Maqbool, B., Riaz, M. Q., Qamar, U., & Abbas, M. (2018). Scrum Software Maintenance Model: Efficient Software Maintenance in Agile Methodology. In *21st Saudi Computer Society National Computer Conference (NCC'18).* doi: 10.1109/NCG.2018.8593152

Riechert, T., & Berger, T. (2009). Leveraging semantic data Wikis for distributed requirements elicitation. In *ICSE Workshop on Wikis for Software Engineering.* doi: 10.1109/WIKIS4SE.2009.5069992

Ristić, S. (2017). How to apply model-driven paradigm in information system (Re)engineering. In *14th Int. Scientific Conference on Informatics* (p. 6-11). IEEE. doi: 10.1109/INFORMATICS.2017.8327212

Riungu-Kalliosaari, L., Mäkinen, S., Lwakatare, L. E., Tiihonen, J., & Männistö, T. (2016). DevOps Adoption Benefits and Challenges in Practice: A Case Study. In *Product-Focused Software Process Improvement* (pp. 590–597). Springer International Publishing. doi: 10.1145/3210459.3210465

Rogozov, Y., Kucherov, S., Lipko, J., Belikov, A., Maakot, A., & Belikova, S. (2020). Method of designing the modular structure of the information system. *Journal of Physics: Conference Series*, *1457*. doi: 10.1088/1742-6596/1457/1/012014

Rumpe, B. (2003). Model-Based Testing of Object-Oriented Systems. In *Symposium on Formal Methods for Components and Objects (FMCO'02)* (pp. 380–402). Springer. doi: 10.1007/978-3-540-39656-7_16

Rumpe, B. (2016). *Modeling with UML: Language, Concepts, Methods*. Springer International.

Rumpe, B. (2017). *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer International. doi: 10.1007/978-3-319-58862-9

Runeson, P., & Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, *14*(2), 131–164. doi: 10.1007/s10664-008-9102-8

Savor, T., Douglas, M., Gentili, M., Williams, L., Beck, K., & Stumm, M. (2016). Continuous deployment at Facebook and OANDA. In *38th Int. Conf. on Software Engineering Companion (ICSE-C'16)* (p. 21-30). doi: 10.1145/2889160.2889223

Schmidt, D. C. (2006). *Model-driven engineering* (Vol. 39) (No. 2). doi: 10.1109/MC.2006.58

Schonbock, J., Etzlstorfer, J., Kapsammer, E., Kusel, A., Retschitzegger, W., & Schwinger, W. (2015). Model-Driven Co-evolution for Agile Development. In *48th Hawaii Int. Conf. on System Sciences (HICSS)* (pp. 5094–5103). IEEE. doi: 10.1109/HICSS.2015.603

Selic, B. (2003). The pragmatics of model-driven development. *IEEE software*, *20*(5), 19–25. doi: 10.1109/MS.2003.1231146

Shaw, M. (2003). Writing good software engineering research

papers. In *25th Int. Conf. on Software Engineering (ICSE'03)* (p. 726-736). doi: 10.1109/ICSE.2003.1201262

Shirole, M., Suthar, A., & Kumar, R. (2011). Generation of improved test cases from UML state diagram using genetic algorithm. In *4th India Software Engineering Conference (ISEC'11)* (pp. 125–134). ACM Press. doi: 10.1145/1953355.1953374

Stachowiak, H. (1973). *Allgemeine Modelltheorie*. Springer Verlag.

Sujeeth, A., Rompf, T., Brown, K., Lee, H., Chafi, H., Popic, V., ... Olukotun, K. (2013). Composition and Reuse with Compiled Domain-Specific Languages. In *Object-Oriented Programming (ECOOP'13)* (Vol. 7920 LNCS). doi: 10.1007/978-3-642-39038-8_3

Taentzer, G., Ermel, C., Langer, P., & Wimmer, M. (2014). A fundamental approach to model versioning based on graph modifications: from theory to implementation. *Software & Systems Modeling*, *13*(1). doi: 10.1007/s10270-012-0248-x

Tarr, P., Ossher, H., Harrison, W., & Sutton, S. M. (1999). *N Degrees of Separation: Multi-Dimensional Separation of Concerns* (Tech. Rep.). doi: 10.1145/302405.302457

Utting, M., & Legeard, B. (2007). *Practical Model-Based Testing*. Elsevier. doi: 10.1016/B978-0-12-372501-1.X5000-5

Vieira, M., Leduc, J., Hasling, B., Subramanyan, R., & Kazmeier, J. (2006). Automation of GUI Testing Using a Model-driven Approach. In *Int. Conf. on Software Engineering (ICSE'06)* (pp. 9–14). ACM Press. doi: 10.1145/1138929.1138932

Völter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L. C. L., ... Wachsmuth, G. (2013). *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org.

Völter, M., Stahl, T., Bettin, J., Haase, A., Helsen, S., Czarnecki, K., & von Stockfleth, B. (2013). *Model-Driven Software Development: Technology, Engineering, Management*. Wiley.

Wang, L., Yuan, J., Yu, X., Hu, J., Li, X., & Zheng, G. (2004). Generating test cases from UML activity diagram based on gray-box method. In *Asia-Pacific Software Engineering Conference (APSEC'04)* (pp. 284–291). IEEE. doi: 10.1109/APSEC.2004.55

Wieczorek, S., & Stefanescu, A. (2010). Improving testing of enterprise systems by model-based testing on graphical user interfaces. In *17th Int. Conf. and Workshops on the Engineering of Computer-Based Systems (ECBS '10)* (pp. 352–357). doi: 10.1109/REW.2016.021

Winkler, J., & Vogelsang, A. (2016). Automatic Classification of Requirements Based on Convolutional Neural Networks. In *24th Int. Requirements Engineering Conference Workshops (REW)*. doi: 10.1109/REW.2016.021

Wohlin, C., Höst, M., & Henningsson, K. (2003). Empirical Research Methods in Software Engineering. In R. Conradi & A. I. Wang (Eds.), *Empirical Methods and Studies in Software Engineering: Experiences from ESERNET* (pp. 7–23). Springer. doi: 10.1007/978-3-540-45143-3_2

Yin, R. (2003). *Case Study Research: Design and Methods*. SAGE.

About the authors

Imke Drave received her Master's Degree at Technical University Munich in 2017. Since then, she has been a research assistant at the Chair of Software Engineering (SE) at the RWTH Aachen University. Her main research interests lie in formal methods for model evolution and consistency analysis and agile methodologies for Model-Driven Systems Engineering. Furthermore, she represents SE in the Center for Systems Engineering, of whom the chair is a founding member. You can contact the author at drave@se-rwth.de or visit www.se-rwth.de.

Arkadii Gerasimov is a PhD student at the Software Engineering chair. Among his main research interests are web-engineering and generation of data centric enterprise applications using different modeling languages. You can contact the author at gerasimov@se-rwth.de or visit www.se-rwth.de.

Judith Michael is PostDoc at the Software Engineering chair and leader of the Model-Based Assistance and Information Services (MBAIS) team. Her PhD thesis at Universität Klagenfurt was about cognitive modeling for assistive systems. Her research focus is model-driven software engineering and software architectures, domain-specific languages, and (conceptual) modeling in a variety of domains and applications. Recent work deals with software architectures of assistive and information systems, digital twins and digital shadows in the production domain, privacy-preserving system design, smart assisted living, and human behavior goal modeling. You can contact the author at michael@se-rwth.de or visit www.se-rwth.de.

Lukas Netz is a PHD student at the Software Engineering Chair of the RWTH Aachen University. His main research interests are about language and corresponding generator development, that support rapid prototyping and iterative evolution of web applications. You can contact the author at netz@se-rwth.de or visit www.se-rwth.de.

Bernhard Rumpe is heading the Software Engineering department at the RWTH Aachen University, Germany. Earlier he had positions at INRIA Rennes, Colorado State University, TU Braunschweig, Vanderbilt University, Nashville, and TU Munich. His main interests are rigorous and practical software and system development methods based on adequate modeling techniques. This includes agile development methods as well as model-engineering based on UML/SysML-like notations and domain specific languages. He also helps to apply modeling, e.g. to autonomous cars, human brain simulation, BIM energy management, juristical contract digitalization, production automation, cloud, and many more.

He is author and editor of 34 books including "Agile Modeling with the UML" and "Engineering Modeling Languages: Turning Domain Knowledge into Tools". You can contact the author at rumpe@se-rwth.de or visit www.se-rwth.de.

Simon Varga is a PhD student at the Software Engineering chair. His PhD thesis is about the generation of data centric enterprise applications using different modeling languages and con-

figurable target environments. You can contact the author at varga@se-rwth.de or visit www.se-rwth.de.