A Layered Reference Architecture for Metamodels to Tailor Quality Modeling and Analysis

Robert Heinrich, Misha Strittmatter, Ralf Reussner

Abstract—Nearly all facets of our everyday life strongly depend on software-intensive systems. Besides correctness, highly relevant quality properties of these systems include performance, as directly perceived by the user, and maintainability, as an important decision factor for evolution. These quality properties strongly depend on architectural design decisions. Hence, to ensure high quality, research and practice is interested in approaches to analyze the system architecture for quality properties. Therefore, models of the system architecture are created and used for analysis. Many different languages (often defined by metamodels) exist to model the systems and reason on their quality. Such languages are mostly specific to quality properties, tools or development paradigms. Unfortunately, the creation of a specific model for any quality property of interest and any different tool used is simply infeasible. Current metamodels for quality modeling and analysis are often not designed to be extensible and reusable. Experience from generalizing and extending metamodels result in hard to evolve and overly complex metamodels. A systematic way of creating, extending and reusing metamodels for quality modeling and analysis, or parts of them, does not exist yet.
When comparing metamodels for different quality properties, however, substantial parts show quite similar language features. This leads to our approach to define the first reference architecture for metamodels for quality modeling and analysis. A reference architecture in software engineering provides a general architecture for a given application domain. In this paper, we investigate the applicability of modularization concepts from object-oriented design and the idea of a reference architecture to metamodels for quality modeling and analysis to systematically create, extend and reuse metamodel parts. Thus, the reference architecture allows to tailor metamodels. Requirements on the reference architecture are gathered from a historically grown metamodel. We specify modularization concepts as a foundation of the reference architecture. Detailed application guidelines are described. We argue the reference architecture supports instance compatibility and non-intrusive, independent extension of metamodels. In four case studies, we refactor historically grown metamodels and compare them to the original metamodels. The study results show the reference architecture significantly improves evolvability as well as need-specific use and reuse of metamodels.

Index Terms—Domain-Specific Modeling Language, Reference Architecture, Metamodel, Quality Analysis

1 Introduction

Software is an essential part not only of information systems but various facets of our daily life. Mobility, energy, economics, production and infrastructure strongly depend on software which is not always of high quality. Critical performance and security issues, for instance, may arise from bad software quality [1]. Examples can be found manifold in the press, like when Denver International Airport opened 16 months delayed and almost $2 billion over budget due to lack of performance and scalability of the baggage-handling software system or when 6.5 million user account credentials were leaked from LinkedIn.

A system on which software contributes essential influence is called a software-intensive system [2]. Software-intensive systems comprise besides software other domains, such as electronics and mechanics. To specify a software-intensive system in the form of a model promises many benefits like a better understanding due to abstraction, platform independence, model-based analysis for behavior and quality. Many quality properties mainly depend on a good architectural design, such as performance, availability and maintainability. Therefore, models are needed to reason early on qualities of a system. Of course, for such models, specific modeling languages are needed. Examples for modeling languages and formalisms to reason on quality are queuing networks, Petri nets, markov chains, fault trees or the Palladio Component Model (PCM) [3].

In *Model-Driven Engineering* (MDE), graphical and textual modeling languages are often defined through a metamodel. A metamodel is a model which defines the structure and characteristics of other models. The background of our research is the model-based analysis of software-intensive systems for various quality properties. A plethora of different metamodels for modeling the quality of software-intensive systems exist. Such metamodels are specific to different quality properties (e.g., performance versus reliability), tools (e.g., the Palladio bench [3] versus QPN-Tool [4]) or analysis tasks (e.g., mean time analysis versus prediction of a statistical distribution). Approaches to come to unified models, such as extending an existing well-proven metamodel, showed that metamodels for quality modeling and analysis are prone to growth due to features (e.g., new quality properties to be analyzed) being added to the modeling language over time. For example, the PCM was initially designed as a metamodel for performance analysis of software architectures. Over time, the PCM has been extended for modeling reliability, costs, maintainabil-

• *Robert Heinrich, Misha Strittmatter and Ralf Reussner are with the Karlsruhe Institute of Technology, Karlsruhe, Germany*
E-mail: {heinrich,strittmatter,reussner}@kit.edu

https://doi.org/10.1109/TSE.2019.2903797

ity, energy consumption and many other quality properties. These modifications may lead to degradation of the structure of the metamodel [5]. Feature overload, feature scattering and unconstrained creation of dependencies harm the evolvability and reusability of the metamodel.

When comparing metamodels for different quality properties, substantial parts of the models for the specification of structure and behavior of the system show similar language features. For example, in [6] modeling performance and reliability rely on the same language features for structure and behavior and merely differ in quality-specific extensions to this features. However, metamodels are seldom designed so that parts (or the entire metamodel) can be reused in different contexts. This is because to reuse a metamodel, the whole metamodel and all of its dependencies must be reused. Otherwise, dependencies are left unresolved which results in an invalid metamodel. Reusing the whole metamodel and all its dependencies is not suitable if only a subset is required. Just ignoring the irrelevant parts of the metamodel is not a feasible way either as they unnecessarily increase complexity and reduce understandability when evolving the metamodel.

Both, extension and reuse bloat the metamodel with plenty of unnecessary complexity if the metamodel is not well modularized. A systematic way of extending and reusing metamodels for quality modeling and analysis, or parts of them, while preserving their structure is required to reduce complexity.

In software engineering, reference architectures provide a general architecture for applications in a particular domain, which partially or fully implement the reference architecture [7]. For example, the Java EE architecture is a layered reference architecture which can be used as a template for various systems developed in Java. Other examples of established reference architectures are the ISO/OSI network stack and the TCP/IP protocol architecture.

A reference architecture for metamodels for quality modeling and analysis can serve as a template solution to support systematically extending and reusing parts of metamodels. There are various commonalities in object-oriented design (like for example advocated by Robert C. Martin [8]) and metamodel design. Both describe classes, their attributes and dependencies. These commonalities encourage us to transfer established concepts from object-oriented design, such as composition, acyclic dependencies, dependency inversion and layering to metamodels. Also transferring the idea of reference architectures to metamodels seems reasonable as there are various commonalities in the object-oriented modeling of structure and behavior for several domains. On a fundamental level, basic language features, like the connection of components by interfaces as well as the composition and nesting of components, hold true regardless the type of component, like software, mechanical and electrical components. Domain-specific properties may extend and refine the basic language features. Quality-related properties may extend and refine the domain-specific properties.

To the best of our knowledge, there is no reference architecture for metamodels for quality modeling and analysis in existence yet. Recent approaches to language extension and reuse (described in Section 10) do not maintain compatibility with the base language. They do not support extending the language externally and independently. Further, they are designed for language engineering in general and do not take into account specifics of a given domain or quality.

In this paper, we investigate the applicability of modularization concepts as known from object-oriented design and reference architectures as known from architectural design to metamodels. We do this by presenting the first reference architecture for metamodels for quality modeling and analysis to address shortcomings in extension and reuse that we mentioned earlier. The reference architecture leverages patterns that reoccur for several domains. It proposes a top-level decomposition of metamodels for quality modeling and analysis into four layers — paradigm, domain, quality and analysis. A layer is a set of metamodel modules which can be extended by lower-level layers and reused in different metamodels. Metamodel modules are assigned to a specific layer depending on the features they offer to the language. The modularization concepts and reference architecture are a substantial step towards our research vision of tailored quality modeling and analysis proposed in [9].

The modularization concepts are applicable to metamodels in general. They are independent of quality modeling and analysis. Also the idea of a reference architecture can be applied to any kind of metamodels as long as they face variation and extension of basic language features. The layers specification, however, strongly depends on the specific metamodels. Other scopes would result in other reference architectures. We believe metamodels for various purposes can benefit from a reference architecture. Nevertheless, our research focuses on metamodels for quality modeling and analysis as this kind of metamodels face high variability for several quality properties. This is because when applying the metamodels for modeling different systems, substantial parts of the metamodels can be reused while the quality properties of interest may differ. Furthermore, in addition to documenting quality properties, systems may be analyzed for quality properties. Often analysis configurations related to the quality properties are reflected in the metamodels. Consequently, metamodels for quality modeling and analysis are well suited for applying a layered reference architecture. Contributions provided in this paper are listed in the following.

- After introducing modeling foundations and associated roles in Section 2, we investigate shortcomings in a historically grown metamodel for quality modeling and analysis based on which we derive requirements on the reference architecture in Section 3.
- In Section 4, we first provide the foundational modularization concepts for metamodels. These concepts enable the description of language features, their relations and grouping to specify a modeling language. In contrast to related approaches, the modularization concepts enable clear distinction between language features and their implementation in metamodel modules. Then, we propose the reference architecture for metamodels for quality modeling and analysis based on the metamodel modularization concepts. In contrast to related approaches, the reference architecture provides guidance and a systematic way of extending and reusing metamodels. Furthermore, we discuss design ratio-

nale behind the reference architecture.
- Technical foundations for implementing the modularization concepts and tool support are described in Section 5.
- In Section 6, we give detailed guidelines on the application of the reference architecture. We first describe refactorings on class and metamodel level. Then, we present two processes on how to apply our approach considering two application scenarios: (1) designing a metamodel from scratch and (2) refactoring an existing metamodel.
- In Section 7, we refactor historically grown metamodels from different domains in four case studies by using the reference architecture to demonstrate its applicability.
- Based on the case studies, we argue that the reference architecture supports non-intrusive extension, instance compatibility and independent extension of metamodels in Section 8.
- In Section 9, we apply the four case studies to evaluate the reference architecture by comparing the refactored metamodels to the original metamodels. Evaluation results show that the reference architecture improves evolvability as well as need-specific use and reuse.

The paper concludes with a discussion of related work in Section 10, a summary and a description of future work in Section 11.

2 Foundations

This section introduces the foundations that are relevant to understand the approach proposed in the paper.

2.1 Modeling Foundations

Modeling Language: In MDE, languages are created and applied to efficiently design and reason about systems. Such languages capture reoccurring domain knowledge in the form of language features and patterns and are used to build models. Therefore, they are called *Domain-Specific Modeling Languages* (DSMLs). DSMLs can be subdivided into grammar-based languages and metamodel-based languages. In our research, we focus on metamodel-base DSMLs. An example of a metamodel-based DSMLs is the Automation Markup Language (AutomationML) for modeling automated production systems.
Metamodeling: A *metamodel* defines the abstractions [10] of a DSML. An instance of a DSML is a *model* that conforms to the metamodel of the DSML. Metamodels also have to be specified in a formal language. A popular metamodeling language is *EMOF* [11]. EMOF is a modeling standard released by the OMG. In its core, it provides concepts similar to that of class diagrams. In EMOF, a metamodel implements the abstractions that the language provides by *classifiers* and their properties. A classifier is either a *class*, *data type* or *enum*. An instance of a class is an *object*. Classes may have several types of *class properties* that introduce *dependencies* to other classifiers: *attribute*, which references a data type or enum, *inheritance* to another class ("is a"), *referencing* another class ("knows a"), *containment* to another class ("has part"), type parameter bounds and type arguments. These dependencies can even point to classifiers of other metamodels (inter-metamodel dependencies). We refer to a class that has a containment relation to a second class as the *container* of the second class. A class that is not contained anywhere but has outgoing containments is called a *root container*. An instance of a root container is the root of a model. The classifiers of a metamodel are organized in a package structure. A *package* can contain classifiers and other packages. In the simplest case, the hierarchy of a metamodel consists of one package.

Our approach targets EMOF-based metamodels, as they are widely used and there is an open source implementation (EMF's Ecore[1]). As EMF is open source, many supporting tools and frameworks were developed for EMF (e.g., code generators, transformation languages and editor frameworks). However, we expect that our approach is also applicable to non-EMOF metamodels that support concepts that are similar to or can be mapped to the above mentioned concepts (classifiers, attributes, references and the ability to depend on classifiers of other metamodels). This is, however, not the focus of this paper.
Feature Models: In this work, we use feature models to express the features of a language. Based on a feature model, subsets of the given features are selected to specify which features of a language are of current interest for model instantiation and tool development. A *feature model* [12] is a formalism to capture the variability and interdependencies of features of a specific subject. Except for the root feature, each feature has exactly one parent. These parent-child relations form a tree. Parent-child relations are either of the type mandatory or optional, or can be part of an alternative set or OR set [12]. A *mandatory* child feature has to be selected if its parent feature is selected. An optional child feature may be selected but is not mandatory. From the features in an *alternative set*, exactly one feature has to be selected. From the features in an *OR set*, at least one feature has to be selected. In contrast to the usual use of feature models, in the scope of our work, we allow feature sets with only one feature. The benefit is that later, more features can be added to the feature set, without having to change the child relation type. Features can also have *requires relations* and *excludes relations* to other features. Requires relations are directional. Excludes relations are mutual. A *feature selection* is a subset of the features from the feature model, that adheres to the constraints imposed by the feature relations.

2.2 Roles

We distinguish different roles that work with DSMLs.
Developer: Basically, we distinguish two developer roles – metamodel developer and tool developer – depending on how they work with metamodels. The *metamodel developer* develops and maintains the metamodel. For example, s/he creates the metamodel, fixes bugs, specifies constraints, modifies classifiers according to changing requirements and extends the metamodel by new features. The *tool developer* develops and maintains tools that work on the metamodel. S/he writes and modifies code that uses the classifiers defined in the metamodel. We use the term *developer* hereafter if we want to address both roles at the same time.

In our approach, the *metamodel developer* role is further distinguished into the *module developer* and the *metamodel architect*. The *module developer* is responsible for the internals

[1] https://www.eclipse.org/modeling/emf/

of metamodel modules. S/he creates and modifies classes and their class properties. The *metamodel architect* is responsible for module dependencies, the feature model, and the layering of metamodel modules and features. Both roles cooperate when creating or modifying module dependencies, as these are determined by the classes within a metamodel module.

User: We also refer to the role of the user of a metamodel in the paper. The user employs a metamodel via tools that operate on instances of the metamodel (e.g., editors, simulators, analyzers, generators). Thus, we address this role as *tool user*. Tool users create and modify models using editors. They process models with simulators and analyzers. Further, they transform models into other formats (e.g., code). A tool user has specific needs regarding the abstractions that are implemented by the metamodel. We refer to specific groups of abstractions that are usually used together and have a common theme as a *concern* of the tool user. Examples of concerns are the modeling of static software design, software behavior and software performance.

3 Application Scenario and Requirements Elicitation

In this section, we provide further details on the PCM to motivate an application scenario and elicit requirements on the reference architecture. The requirements we derive can be distinguished into two categories. One category refers to the reference architecture and the other aims at extension mechanisms. The reference architecture, however, cannot be used without proper extension mechanisms. So we still regard the requirements to extension mechanisms as a subset of the requirements to the reference architecture.

While we use the PCM as a representative, other historically grown metamodels for quality modeling and analysis show similar weak spots. For example, an analysis of the Capella [13] metamodel for model-based systems engineering showed it shares 12 of 14 types of metamodeling bad smells [5], [14] with the PCM[2] (e.g., multipath hierarchy, hub-like modularization and cyclically-dependent modularization).

The PCM is an established and widely used metamodel. It provides various useful features for quality modeling and analysis of component-based software architectures as described in [3]. The PCM consists of 203 classes in 24 packages. It is divided into five sub-metamodels. Around 73% of its classes reside in the biggest sub-metamodel. Starting in August 2006, the PCM has a long history of evolution. In the time from spring 2007 to fall 2012, the PCM grew from under 100 to over 200 classes [5]. There are at least 12 extensions of the PCM[3]. However, many more exist that are not publicly documented (e.g., student theses, experimental, incubation). Due to its historically grown structure, the PCM exhibits some shortcomings described hereafter.

Package Structure Erosion and Uncontrolled Growth of Dependencies: Due to repeated extensions and maintenance, the structure of the PCM eroded over time [15]. Starting with performance analysis, the scope of the PCM broadened, and more structural features and quality properties were incorporated. Initially, new features were intrusively implemented in the metamodel, e.g., for modeling reliability [6], event-based communication [16] as well as infrastructure components and middleware [17].

While intrusively implementing new features, they were placed inconsistently into the package structure of the metamodel [5]. Some new features were added to packages of similar features. Several cross-cutting concerns were scattered over the package structure, instead of being specified in a new subpackage structure. For other features, distinct packages were specified. However, these packages were not placed consistently in the existing hierarchy.

In consequence, language features are hard to grasp, if they are not adequately reflected in the package structure. For example, in case a package contains classifiers of multiple features or features are scattered over the package structure, developers are hindered in narrowing down the part of the metamodel that is relevant for their current task. Thus, the erosion of the package structure worsens understandability and, therefore, evolvability of the metamodel.

The fact that package structures allow free creation of new dependencies to other packages causes another related shortcoming – superfluous inter-package dependencies and many dependency cycles between packages accumulated in the PCM. Uncontrolled growth of dependencies hinders understandability. This is because developers, while trying to understand the semantics of a class, may follow dependencies to packages irrelevant for their objectives. Uncontrolled growth of dependencies further increases the complexity of the metamodel. Unnecessary inter-package dependencies increase coupling, which impedes evolving the metamodel and hinders developers identifying the part of the metamodel that is relevant to their task.

From these shortcomings – package structure erosion and uncontrolled growth of dependencies – we derive two requirements on the reference architecture. **R1** (Improved Evolvability): The reference architecture must improve the evolvability of metamodels. By good evolvability we understand low complexity, low coupling and high cohesion as described in further detail in Section 9. **R2** (Non-intrusive Extension): The reference architecture must ensure that metamodels are not dependent on their extensions. Extensions must not alter the extended metamodel. Thus, by implementing extensions in a non-intrusive way, we expect to prevent aforementioned adverse effects.

Instance Incompatibility: Some extensions developed in branches have never been included in the PCM master (e.g., [18], [19], [20]). The advantage is that the master does not need to be altered, and the development of master and branches is decoupled. The extension branches, however, need to be maintained to be up-to-date with the master. The situation is visualized in Figure 1 (1). Instances of metamodels from branches m' are not compatible with the tools that operate on instances of the master metamodel m. For brevity, Figure 2 shows a legend with the notational elements that are used in all the figures of this paper. So all the figures shown in the paper refer to Figure 2.

As a consequence, tools have to be branched and maintained in specific branches as well. Alternatively, a transformation between both metamodel branches needs to be

[2]https://sdqweb.ipd.kit.edu/wiki/EMOF_Bad_Smells
[3]https://sdqweb.ipd.kit.edu/wiki/PCM_AddOns

developed. It transforms parts of instances of the branched metamodel to the original metamodel so that the original tools can still be used. However, these transformations have to be maintained as well. Language features not supported by the original metamodel cannot be transformed and, therefore, also not handled by the original tooling. Instance incompatibility has also been tackled by work from the context of metamodel/model co-evolution (e.g., [21], [22]). These approaches are described in Section 10.

We derive **R3** (Instance Compatibility): The reference architecture must ensure an instance m' of an extended metamodel M' is compatible with tools that are built to operate on the base metamodel M.

Note, the original tools can only operate on the part of the instance m' that is defined in the base metamodel. If the original tools are designed to be extensible, they can be extended to also support the language features of the metamodel extension. Co-evolution of metamodels and tools is part of our research vision [9] but not in the focus of this paper. It will be investigated in respect to analytical and simulative model solvers in future work.

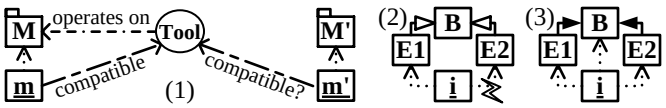

Fig. 1. Requirements: Instance Compatibility and Independent Extension

Incompatible Extensions: Some extensions to the PCM use inheritance to introduce new class properties to existing classes. Subtyping is problematic as two different extensions that subtype the same class cannot be used in combination. This is illustrated in Figure 1 (2). E1 and E2 both inherit the base class B and add some class properties. It is not possible to create an instance i of E1 and E2 (illustrated by the lightning). i is an object, as denoted by the underlined name. The only way to use the combination of both extensions is to create another class that inherits from E1 and E2. However, this means extensions cannot be developed independently, as all conflicting extensions have to be extended to make them compatible. To address this shortcoming, we specify **R4** (Independent Extension): The reference architecture must ensure that extensions can be developed independently of each other and used in combination. This is illustrated in Figure 1 (3). The base metamodel B is extended by E1 and E2. In general, it should be possible to create an instance of B that also carries the information of E1 and E2. Using an extension must not prohibit using other extensions because of technical reasons. It is allowed to forbid the use of two extensions if they are semantically conflicting. To prevent such workarounds like with the subtyping extension, an extension should only depend on another extension if the feature that is implemented by the first extension depends on the feature that is implemented by the second extension.
All or Nothing Reuse: A shortcoming of a monolithic metamodel in general, and thus also of the PCM, is the metamodel can only be reused as a whole. It is not possible to reuse parts of the metamodel. Metamodel developers are confronted with the full extent of the PCM. They stumble over features that are irrelevant to them. These irrelevant features may confuse developers, as it is not always apparent at first glance what a set of classifiers is representing. From this shortcoming, we derive **R5** (Need-specific Reuse): The reference architecture must enable a selective reuse of parts of the metamodel that are indeed needed.

When developing tools, the tool developer has to understand the metamodel. The complexity of monolithic metamodels also hinders tools developers. Moreover, users of tools based on a monolithic metamodel, are confronted with the full extent of its features. Especially optional features that are not of interest to a specific tool user could distract and confuse them. To avoid this shortcoming, we derive **R6** (Need-specific Use): The reference architecture must enable a selective use of parts of the metamodel according to the needs of the tool user, the tool developer and the dependencies of the tools.

4 Layered Reference Architecture for Metamodels

In this section, we first describe metamodel modularization concepts (Section 4.1) that are independent of the purpose of a language and, therefore, can be applied to metamodels in general. Then, we propose the reference architecture (Section 4.2) specific to metamodels used for quality modeling and analysis. Moreover, we discuss design rationale behind the reference architecture (Section 4.3).

4.1 Metamodel Modularization Concepts

General metamodel modularization concepts used as a foundation of our reference architecture are the following.
Language Features: In our terminology, a language is composed of language features. We introduce this term for the metamodel architect to be able to specify what a language should express on a conceptual level. *Language features* can either be atomic or composed. An *atomic language feature* is an abstraction of a thing to be modeled. A *composed language feature* is a set of atomic and composed language features. A language feature that covers a concern (see Section 2.2) of a tool user is called a *user language feature*. Features can have *feature dependencies* to other features. The dependencies of an atomic language feature are determined by the dependencies of the abstraction it models. The dependencies of a composed language feature are determined by the dependencies of the language features it contains. We distinguish between first-class and second-class atomic language features (in analogy to first- and second-class entities). A *first-class language feature* is an atomic language feature that is contained in a root class; a *second-class language feature* is only transitively contained.
Use of Feature Models: We use feature models known from the software product line community and domain modeling to structure language features more explicitly. In addition to the usual use of feature models, we use the dependencies of the feature nodes to restrict the dependencies of parts of the metamodel that are allocated to the feature nodes.

Almost every node in the feature model represents a language feature. Exceptions are the root feature, parents of feature sets and features that have been created to group other features. All feature dependencies have to be supported by either a parent feature relation, a requires relation

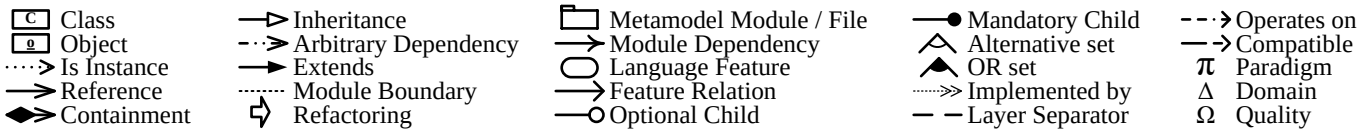

Fig. 2. Legend of the Notational Elements used in the Paper

or a path in the feature tree that consists of feature required relations and parent feature relations (in this case, we speak of a transitive feature dependency). Feature dependencies are not allowed to form cycles. In our approach, also tool users use feature models to select the language features they want to use. In contrast to a mere graph of language features and their dependencies, a feature model forces the language features into a hierarchical structure regarding the child/parent relation. Such a feature hierarchy helps tool users in feature selection, as tool users can start at the root feature and only follow down on branches that are relevant to them.

Modules and Dependencies: All feature nodes that represent language features are implemented by metamodel modules. We define a *metamodel module* as a container of packages and classifiers that has explicit dependencies. The difference between an EMOF metamodel and a metamodel module is that the dependencies between metamodel modules have to be declared explicitly and follow certain restrictions. Inspired by the acyclic dependencies principle [23], metamodel module dependencies are not allowed to form cycles. A cycle would mean that if one of the metamodel modules is used, all of the metamodel modules in the cycle have to be used, which makes the modularization meaningless. Further, the dependencies of a module must conform (directly or transitively) to the dependencies of the module's feature. We consider a metamodel that has been subdivided into metamodel modules still as a metamodel.

Classifiers of one metamodel module M may depend on classifiers of another metamodel module N. In this case, we regard M as being dependent on N. The different types of dependencies between classifiers are explained in Section 2. Additionally, we introduce a new type of dependency between two classes: the extends relation. We need this new type of dependency to reverse existing dependencies and break dependency cycles. On the level of metamodel modules and their dependencies, however, it is irrelevant precisely which types of dependencies there are between both metamodel modules. The emphasis is foremost on the presence and the direction of the dependencies. A dependency from M to N implies that when a tool uses M, N has to be installed as well.

Extends Relation: The standard EMOF dependencies are insufficient to restructure metamodel module dependencies in a way that does not violate the reference architecture. In contrast to object-oriented design [8], there is no EMOF dependency that supports the addition of new class properties to classes without violating **R3** (Instance Compatibility) or **R4** (Independent Extension). Thus, we introduce the extends relation between classes. An extends relation from one class C1 to another class C2 adds the class properties (e.g., attributes or references) of C1 to the extended class C2. More information from a technical point of view is given in Section 5.3.

Layers: A layer is a logical grouping of language features and associated metamodel modules that implement a specific semantic. Each language feature and metamodel module is allocated to exactly one layer. There can be an arbitrary number of layers. Having just one layer is equivalent to having no layering at all. The layers are ordered concerning the dependencies of their language features and metamodel modules. Similar to the layered software architecture pattern [24], feature required relations, feature parent relations, and module dependencies may only point into the same or a more basic layers (basic concerning its level of abstraction). We illustrate basic layers at the top of graphical visualizations of the layering. We do this, as by convention in class diagrams, which are commonly used to illustrate metamodels, more abstract classes are shown at the top and inheritance/generalization arrows point upward.

Relation between Modularization Concepts: Figure 3 shows how layers, language features and metamodel modules relate to each other. If every module dependency is supported by feature relations, we address this as the feature model and the module dependencies being *conform*. A module dependency from metamodel module M to metamodel module N is considered as *supported* if by following relations from the feature that is implemented by M (i.e., feature F) the feature that is implemented by N (i.e., feature G) can be reached. The only relations that can be followed are the required relation, from the requiring to the required feature, and the parent relation, from the child to the parent feature. Considering the example in Figure 3, the metamodel module M implements the language feature F, and the metamodel module N implements the language feature G. As we can reach G from F, by first following the requires relation and then following the parent relation, the dependency from M to N is supported.

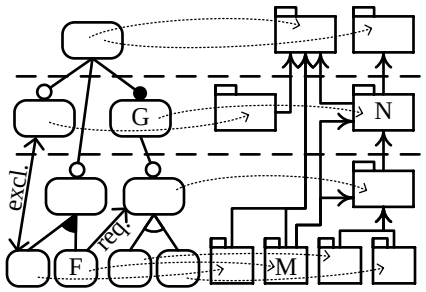

Fig. 3. Relation between the Metamodel Modularization Concepts

Terminology in Related Approaches: Related language engineering approaches bring forth their terminologies. We will briefly elaborate why we introduce some new definitions instead of relying on existing terminology. The language workbench MontiCore [25] uses the terms *language*

components, and *component grammars* to address the abstract syntax definition of language components. We chose to speak of modules instead of components, as a metamodel module cannot be instantiated multiple times (in contrast to the component concept from Component-based Software Development [3]). Of course, it is possible to have multiple other metamodel modules depending on a metamodel module M, but on the type level, M is the same from the perspective of all dependent metamodel modules. In the scope of the concern-oriented approach COLD [26], a language concern is a configurable unit of reuse that provides multiple perspectives (e.g., abstract and concrete syntax) of a language. A facet is the implementation of a perspective. In the terminology of COLD, our approach aims at abstract syntax facets. We still chose the term module, as it more strongly conveys that modularization takes place and the individual pieces are only puzzle pieces in the big picture. In addition to explicit dependency control, this is also the reason why we do not merely speak of metamodels like Degueule et al. do in [27].

Besides metamodel module our approach applies the term language feature. Based on the general meaning of feature in the context of software, we use the term user language feature as a unit of use. We did not use the term abstract syntax facet from COLD, as we want to emphasis this aspect. By using the term language feature, we separate a language part, i.e. abstraction of a thing to be modeled, from its implementation in a metamodel module.

4.2 Layers in Metamodels for Quality Modeling and Analysis

The metamodel modularization concepts from the previous subsection apply to metamodels in general with an arbitrary number of layers. Based on the metamodel modularization concepts, we give more specific guidance in this subsection by proposing a reference architecture for metamodels for quality modeling and analysis. When investigating several metamodels used for quality modeling and analysis as well as their extensions, we identified that they reflect in most cases language features from three categories – structure, behavior, and quality. Features that fall into these categories can be found in metamodels like UML MARTE [28], UMLSec [29], the Descartes Metamodel [30], the PCM [3], AutomationML [31], ROBOCOP [32], and BPMN2 [33]. Based on this observation, we decided to separate parts of a metamodel dealing with structure and behavior, quality, and the corresponding analysis into different layers in our reference architecture. Structure and behavior are further divided into paradigm and domain. In the following, we present these layers. We take a conceptual stance and, thus, mostly speak about language features. The terms metamodel module, class and class property are used, if we refer to the implementation like when examples are given.

Paradigm: The *paradigm* (π) is the most basic and most abstract layer. It specifies the foundation of the language by defining language features for reoccurring patterns of structure and behavior but without dynamic semantic. For example, in the automotive domain, components, their interfaces and connections may be specified in π without specifying whether these are software, electrical, mechanical, or other types of components. As it carries no semantics, a π layer is not intended to be used without any additional layer. The advantage of having a π layer is that π metamodel modules that originate from the development of other languages can be reused if they fit the abstractions to be modeled. This would not be possible if domain-specific semantics were located on this layer. So, if a metamodel developer is designing π, one criterion is to develop metamodel modules that could be used in a variety of domains. First-class language features of π should be abstract. Exceptions can be made if it is meaningful to instantiate such a language feature in another layer without adding further properties. It is not recommended to provide root containers in π to avoid instantiation of concrete first-class language features in π.

Domain: The *domain* (Δ) layer builds upon the paradigm layer and assigns domain-specific semantics to its abstract first-class language features. Δ builds upon structural as well as on behavioral language features. For example, by creating subclasses of the component class (e.g., for the domains of software, mechanics and electronics), the abstract component class can be enriched by domain-specific class properties (e.g., attributes or references). This will result in a metamodel module for software components, a module for mechanical components and one for electrical components. If a developer is only interested in software, the Δ layer merely includes the metamodel module for software components. It is also possible to have metamodel modules of multiple domains in the Δ layer (e.g., mechanics and electronics). A language that consists only of a π and a Δ layer can already be applied, e.g., for quality-agnostic design and documentation of a system. If atomic language features are defined in π, these have to be subtyped by first-class language features in Δ to be usable. Language features can also reuse (by containment) second-class language features of π and reference other first-class language features of π. If new atomic language features are introduced in Δ (without inheritance into π), it should be considered whether they contain an underlying pattern that can be modeled in π. Language features for modeling or analyzing quality properties, however, are not located on Δ layer but part of the layers mentioned hereafter.

Quality: The *quality* (Ω) layer defines quality properties for language features of Δ. For example, performance, reliability or security properties can be added to the component language feature. To be more specific, attributes that model resource demands can be extended to the processing step class of a service of a component to be able to evaluate the performance of the service [3]. A Ω layer is not always needed. Analyses can be conducted for structural and behavioral properties and do not always need explicit quality properties. The Ω layer contains second-class language features that enrich the first-class language features of Δ. Language features that define quality properties contained in a root container of Ω that serve as input to analyses must not change during the analysis. If they change, they model state information and have to be contained from a container in the Σ layer. Ω also models derived quality properties. However, they must not be reachable from a Ω root container by following containment relations. They will instead be contained by containers in the Σ layer.

Analysis: The analysis layer (Σ) comprises language fea-

tures used by analyses. Σ builds upon the previous layers by introducing language features to specify configuration data, runtime state, output data, and input data that does not belong to Δ language features. For example, a sensitivity analysis needs a reference to an attribute as input. The sensitivity analysis modifies the attribute's value over several analysis runs. The attribute is defined in a module located in one of the more basic layers. The reference to the attribute and the value range are defined in modules of the Σ layer. The modules of the more basic layers can be used in several analyses. Moreover, several analyses may share modules from Σ. An example of this is a performability (i.e., performance and availability) analysis that may reuse the output module of a performance analysis. Analyses may also have their own metamodel modules. On the Σ layer, new root containers, first-class language features, and second-class language features can be created as required by an analysis. Atomic language features of the other layers should be reused when possible. However, analysis-specific atomic language features should not be specializations of more basic language features. This would mean, that Σ is not adequately separated from the other layers. The only constraint that holds is the avoidance of dependency cycles.

4.3 Design Rationale

In this subsection, we explain the design rationale behind our reference architecture. They are strongly motivated by the requirements that we described in Section 3. It is important to note that several of our metamodel modularization concepts address the same requirements.

By having metamodel modules with explicit dependencies and by constraining their direction, we aim to tackle **R1** (Improved Evolvability). When developers navigate a modular metamodel, the complexity they face is reduced, compared to a large entangled metamodel. This is because they are merely confronted with the content of the metamodel module of interest and possibly with the content of metamodel modules to which dependencies exist. Furthermore, by prohibiting cycles, the coupling between metamodel modules is forced to be unidirectional. This is beneficial for the evolvability of the metamodel.

By forbidding mutual dependencies between metamodel modules, we also aim to tackle **R2** (Non-intrusive Extension), as this prevents a metamodel to be dependent on its extensions. A modular metamodel with directed cycle-free dependencies also addresses **R5** (Need-specific Reuse) and **R6** (Need-specific Use), as it is possible to use and reuse only those metamodel modules and their dependencies needed for a specific purpose.

An extends relation brings several advantages. It enables factoring out of optional content into an optional metamodel module. This addresses **R5** (Need-specific Reuse) and **R6** (Need-specific Use). If these optional modules can be implemented in a way that the base metamodel does not depend on them, **R2** (Non-intrusive Extension) is addressed. We require the concrete implementation of the extends relation to discuss **R3** (Instance Compatibility) and **R4** (Independent Extension). Therefore, we will get back to these requirements after the extends relations have been applied and we can properly assess how they satisfy the requirements (see Section 8).

Assigning the metamodel modules to specific layers has several benefits. It prevents dependencies from going into a more specific layer, the complexity of these specific layers is decoupled from the more basic layers. This addresses **R1** (Improved Evolvability). This also has the benefit, that specific layers can be exchanged or omitted to reuse more basic layers which addresses **R5** (Need-specific Reuse) and **R6** (Need-specific Use).

Regarding the specific layering for metamodels for quality modeling and analysis, the clear separation of the four layers enables decoupling and exchange of these layers. Thus, the layers π, Δ and Ω can be reused for different analyzers. π and Δ can be reused for different quality properties. π can be reused for different domains. Thus, with this specific layering we tackle **R1** (Improved Evolvability), **R5** (Need-specific Reuse), and **R6** (Need-specific Use).

5 Technical Foundation and Tool Support

Although our approach is widely based on EMOF, not all the concepts proposed in Section 4.1 are supported by EMOF. Concepts already supported by EMOF are classifier, properties of classes and dependencies to other metamodels. Concepts not supported are module, layer, dependency restriction and extension relation. This section explains how the concepts not supported by EMOF are covered by technical concepts of EMF or by tools that we developed.

5.1 Metamodel Module

We realize a metamodel module as a metamodel that is encapsulated in an Eclipse plugin. In EMF, dependencies of a class to another package or to a metamodel that resides within the same plugin are not restricted. The current graphical tooling (Ecore diagram editor) and the tree editor, however, require an explicit declaration if the content of a metamodel of another plugin is referenced. The plugin dependencies then reflect these dependencies. If a tool needs a particular set of user language features, it simply has to require the metamodel modules that implement the features. Further metamodel modules to which dependencies exist are automatically included through the plugin dependencies.

5.2 Modular EMF Designer

We developed a graphical editor, called the *Modular EMF Designer* [34], to support modularizing and layering metamodels as well as restricting dependencies. Figure 4 shows a screenshot. The editor can be used to refactor an existing metamodel or to create the module structure of a new modular metamodel from scratch. If a metamodel or metamodel module is loaded into the editor, all metamodel modules it depends on are also loaded automatically. Metamodel modules can be assigned to layers. The Modular EMF Designer visualizes dependencies between metamodel modules and highlights dependency cycles (in red) and violations against the layering (in orange). It also provides detailed information about the dependencies between two metamodel modules (which classes are dependent by what kinds of dependencies). The editor also assists in the creation of empty metamodel modules and can perform move

refactorings of classifiers and packages. When a move refactoring is performed, the editor automatically updates all incoming dependencies. The editor provides further supporting functionality like hiding of transitive dependencies, and visualization of profiles. Except for move refactorings of classifiers and packages, the editor is not intended to create or manipulate the internals of metamodel modules, as this is already covered by the standard Ecore diagram editor. In conjunction, both editors can be used to create or refactor an existing metamodel into a modular and layered form that adheres to the constraints of the reference architecture. The standard Ecore diagram editor can be invoked from within the Modular EMF Designer by double clicking on a metamodel module. Every external change to a module is immediately updated into the Modular EMF Designer.

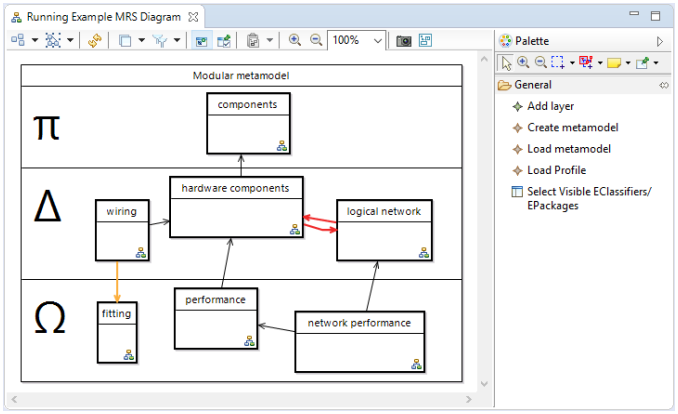

Fig. 4. Screenshot of an Exemplary Metamodel Module Overview Diagram in the Modular EMF Designer

5.3 Extension Mechanisms

Extension is a concept, which is well known in object-oriented design (e.g., stereotyping). EMOF, however, does not support an extends relation. For this reason, we identified and examined several ways on how to enable the creation of extensions with EMF's Ecore. We call these extension mechanisms. A selection of extension mechanisms that are best suited for our purpose are discussed hereafter.

The investigation of different extension mechanisms is important to our work due to the following reasons. Each mechanism has its advantages and disadvantages and, therefore, is appropriate to be applied in a given situation or not. The selection of a given extension mechanism may affect the fulfillment of the requirements described in Section 3 positively or negatively. Consequently, it may be useful to apply different extension mechanisms in the implementation of a modular metamodel.

Figure 5 (0) shows how an intrusive extension looks like. The class B (short for base class) directly owns the dependency D. Arbitrary dependency (dotted arrow) represents one of the dependencies introduced in Section 2 (i.e., attribute, reference, containment, inheritance, type bound or argument). The constellation in (0) is what we want to emulate. It could be the starting point of a modularization by factoring out D. To separate concerns, we extract D into a new class E (short for extension class) and place it in another metamodel module. In this context, we call them the base module and the extension module. This is shown in (1). The notation we use for the extends relation is a filled arrow like it is used for stereotype applications in UML [35].

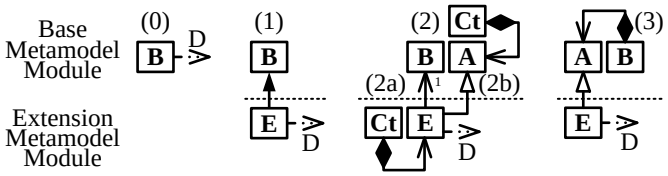

Fig. 5. Overview of Extension Mechanisms

EMF Profiles: There is no native support for stereotypes in EMOF and EMF. The extension proposed in [36] enables such support for stereotypes. Its notation coincides with (1). A profile contains a set of stereotypes. Stereotypes can be applied to a class, in the same way as in UML. Attributes and references can be specified in the stereotype. Addition of containment is not possible in the current version[4]. Therefore, new classes and their containers must be defined in a separate metamodel. The use of EMF Profiles fragments the containment tree, if a new container is introduced. As EMF Profiles adds instances of stereotypes to the extended model, it is intrusive on the model level. This means, it requires the EMF Profiles plugin and the extension metamodel module to be installed to load and modify models that contain stereotype applications. EMF Profiles offers helper methods to enable direct navigation from a B object to its stereotype applications.

Plain Referencing: The simplest way to achieve an extension is to use a reference from E to B [37]. This is shown in (2). To contain *E*, either (2a) a new container Ct is created in the extension metamodel module or (2b) an inheritance has to be established to a class A in the base metamodel module that is already contained somewhere (by Ct). A new container (2a) leads to multiple containment trees (i.e., model fragmentation) but is entirely non-intrusive on the model level. This means, extensions can be applied to models and they can still be read and modified, even when the extension metamodel module is not installed. Inheriting from a class of the base metamodel module (2b) requires the presence of an appropriate class, to which an inheritance is conceptually correct. (2b) is intrusive on the model level. As objects of E are contained in the same model as the instances of the base metamodel, the model can only be loaded when the extension metamodel module is installed. Plain referencing does not enable direct navigation from a B object to its extending E objects. This results in increased complexity of tools working on the metamodel which can be alleviated by helper methods. In contrast to EMF Profiles, the helper methods have to be implemented by developers. They have to iterate over all instances of E until the one is found that points to the B object in question. These helper methods may use hash tables to speed up the look-up of B objects.

Inheritance: If the class to be extended contains another class A that could be a conceptually correct superclass of the extension class, cross-module inheritance should be used.

[4]https://sdqweb.ipd.kit.edu/wiki/MDSDProfiles

This is shown in (3). Using inheritance keeps the containment tree of models intact and enables direct navigation to the extended language features, with the drawback of intrusiveness on the model level. If such a class does not exist, the class and a containment to it can still be created in the base metamodel. In this case, however, this extension mechanism is intrusive on the metamodel level. There are two options on how such a new class can be modeled: (i) a generic class with the multiplicity of the containment being $0..*$. The generic class can then be used as an extension point for further extensions. However, the containment mixes extensions of different types. All contained objects have to be iterated to find the desired extension instance. (ii) a class specific to the extension with a multiplicity of $0..1$. This solution does not mix various extension types but enriches the base metamodel by extension information. This can be undesirable, especially if there are multiple extensions, as the metamodel gets a new abstract class for each extension and thus the complexity rises. The multiplicity should not be set to $1..1$, as this would have the same effect as an intrusive extension (the extension has to be always instantiated).

6 Application Processes

In this section, we first describe class refactorings and metamodel module refactorings required for applying our approach. Then, we present application processes for two scenarios: (1) creation of a new metamodel, and (2) refactoring of an already existing metamodel to fit the reference architecture. The main difference between the two processes is: in the former, the feature model is constructed before the metamodel modules are implemented; in the latter, the metamodel modules already exist and are modularized hand-in-hand with an evolving version of the feature model.

On the one hand, the processes restrict the metamodel developers in their freedom. On the other hand, the processes guide the metamodel developers by providing a given structure to follow and design artifacts (e.g., feature models) to be specified. Thus, the effort for conducting these processes is higher compared to an ad-hoc approach. The additional effort caused by the processes, however, is justifiable in the long run if it ensures better evolvability and reusability of the metamodel.

6.1 Class Refactorings

For applying our approach, several refactorings are necessary to split classes, break dependency cycles and reverse the direction of dependencies. These originate in object-oriented design [8] and make use of the class extension relation that we introduced in Section 4.1. The refactorings are executed by the module developer.

Note, this section heavily refers to the subfigures of Figure 6. For the sake of brevity, in the following we only refer to the subfigure labels.

Class Split: The class split refactoring is used to separate concerns in a class. It is shown in (1). Class properties of a class C are factored out into the new class E, which extends C. Incoming dependencies remain on C. Attributes, references, and containments can be factored out without complications. Also, inheritance can be factored out; however, in EMOF it is not possible to substitute C with E. Thus, factoring out inheritances is only appropriate in cases where substitutability is not required. These cases can be identified by analyzing incoming references onto the superclass. If the superclass is not referenced by another class, the inheritance is only used to inherit the class properties of the superclass and can be factored out.

Breaking Cycles: The class split refactoring can be used to break dependency cycles. This is shown in (2i). C1 is split, and the outgoing dependency of C1 that contributed to the cycle is factored out into E. As C1 does not depend on E, the cycle is broken.

Dependency inversion can be used to break dependency cycles (2ii). Dependency inversion is explained below. Reversing one dependency in a cycle is sufficient. In the example, the dependency from C2 to C1 is inverted, which break the cycle.

Dependency Inversion: The dependency inversion principle [23] states that abstractions (class A in the figure) must not depend on specifics (S), but specifics should depend on abstractions. By transferring this principle to metamodeling, we provide several concrete refactorings for all cases in which dependencies may violate the principle. (3) illustrates the refactorings.

There are multiple ways to invert an inheritance from A to S (3a). If S is a specialization of A, the inheritance was specified in the wrong direction. Instances of A are sometimes erroneously typed with S, and the class properties from S are not needed. In this case, the inheritance can be simply inverted (3ai). Some incoming dependencies may have to be redirected from A to S depending on their meaning. If A and S implement different atomic language features, the inheritance is removed and N, a new subclass of A and S, is introduced (3aii). Incoming dependencies of A and S must be redirected to the correct class (either A, S or N). If S is only used to add class properties to A and not for typing, the inheritance can be replaced by an opposite extends relation (3aiii). For this to be feasible, there must not exist any incoming dependencies (except inheritance) to S and its superclasses.

A reference (3b) can be inverted by using a class split (3bi). The reference from A to S is factored out into the new class E. This option should be chosen, if S is a first-class language feature (i.e., the existence of an instance of S is not dependent on an instance of A). An indicator for this is when an instance of S is referenced by multiple other objects. The reference can also be inverted into an extends relation (3bii). This should be done if S is a second-class language feature and is not referenced by any other class. If S is referenced by multiple classes, a common superclass N can be introduced for these classes, which is then extended by S (3biii). Sometimes, the reference can be correctly owned by both A and S. In such cases, the reference can be inverted (3biv).

A bidirectional reference between A and S (3c) is a special case of 3b. In such cases, the reference from A to S is redundant and can be removed (only the reference from S to A remains). A containment (3d) can be inverted by replacing it by an opposing extends relation.

In (3b) and (3d), also the multiplicity of the original dependency from A to S has to be modeled correctly after the refactoring. If the multiplicity has no lower and upper

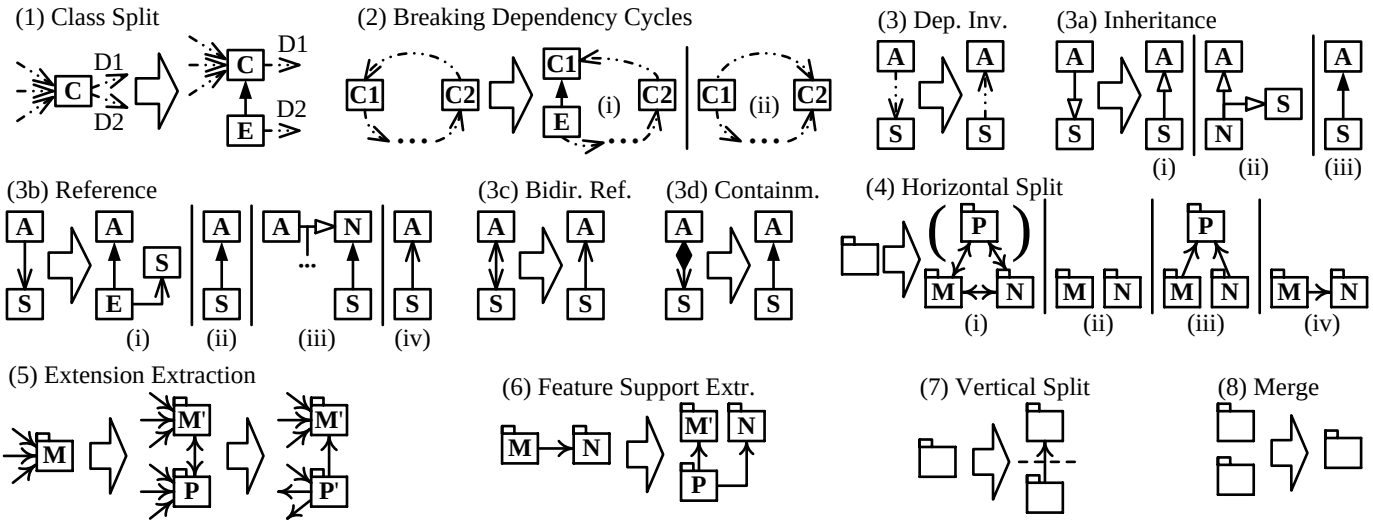

Fig. 6. Overview of Refactorings

bounds (i.e., 0..*), no further modeling is necessary, as an arbitrary number of instances of the extension can be bound to an instance of A. If there is at least an upper or lower bound, a constraint has to be defined that enforces the multiplicity.

6.2 Metamodel Module Refactorings

Our approach relies on several refactorings that modify metamodel modules, their dependencies, and content. Figure 6 illustrates these refactorings. Again, we only refer to the subfigure labels in the following. Many of them perform a split of a metamodel module, which is supported by the graphical editor Modular EMF Designer. To split a metamodel module, the metamodel architect first creates a new metamodel module and then uses the editor to move classes from the original into the new metamodel module. The editor then automatically updates incoming references on moved classes.

Horizontal Split: If there are parts of a metamodel module that can be used independently of each other, the metamodel architect must split the metamodel module. (4i) shows the potential worst case outcome. The resulting modules M and N may still share a common part P of the original module. The brackets indicate that there is not necessarily a common module P. All the metamodel modules may be mutually dependent. The metamodel architect and the module developer have to adjust the dependencies according to the dependencies of the language features that are implemented by M and N. In the simplest case (4ii), the modules are unrelated. In (4iii), both modules are dependent on a common base (P). In (4iv), one of the modules is dependent on the other.

Extension Extraction: The metamodel architect uses this refactoring, if a metamodel module M contains content that is optional (P) but cannot be used independently. Extension extraction is illustrated in (5). The metamodel architect factors out P into a new metamodel module. The remainder of M is denoted as M′. The module developer has to split classes that are essential to M′ if they contain optional class properties belonging to P. The module developer further reverses all dependencies from elements of M′ to P. If there are incoming dependencies to P from other metamodel modules, they have to be considered for dependency inversion (as depicted by the outgoing dependencies of P′).

Feature Support Extraction: Feature support extraction is a special case of the extension extraction. It is illustrated in (6). The metamodel architect can perform this refactoring, if there is a part P of a metamodel module M that is dependent on another metamodel module N and it is meaningful to use M without N. The metamodel architect separates P into its own metamodel module. The remainder of M is denoted as M′. P is dependent on M′ and N. If there are dependencies from M′ to P, the module developer must reverse them. S/he may also conduct class split refactorings to separate content of both features. As P is an extension of M′ that includes content of N, P adds support for N to M′, hence we refer to it as feature support extraction.

Vertical Split: The vertical split is illustrated in (7). The metamodel architect performs this refactoring, if a metamodel module could be assigned to multiple layers. S/he divides the metamodel module in a way that each classifier can be assigned to precisely one layer. If necessary, the module developer has to split classes. The metamodel architect assigns the resulting metamodel modules to their respective layers. If there are module dependencies that violate the layering, the module developer has to perform dependency inversion.

Merge: If there is a mandatory child feature relation between two features or a dependency cycle between metamodel modules, the metamodel architect should consider whether it is meaningful to merge those features and their metamodel modules (8). There may be various dependency constellations between the merged metamodel modules like one directional or bidirectional. There can be even no dependencies between the two metamodel modules, e.g., if abstract classes that function as ubiquitous superclasses are consolidated into one metamodel module even if they are not dependent on each other.

6.3 Creating a new Metamodel

When applying the reference architecture for the creation of a new language, we use feature models to express the variability of the language (in analogy to related approaches [38], [39]). Next, we present process steps that are meant to be performed iteratively. It can be beneficial to backtrack to a prior step (e.g., when it is discovered that a feature was forgotten or a feature can be split).

1) Language Feature Identification: First, the concerns of tool users are identified. For each language concern, a user language feature is defined. Note, this is the requirements identification phase; no technical artifacts are implemented.

2) Reuse: Readily available metamodel modules may exist in organization-internal or even public online repositories. The metamodel architect assigns metamodel modules that can be reused to implement user language features to the respective user language features. Metamodel modules to be reused can depend on other metamodel modules. The metamodel architect either assigns these metamodel modules to the same user language feature or to a new user language feature.

3) Creating the Feature Model: The metamodel architect starts the feature model by creating the root feature node and labeling it with the name of the language. For each of the identified user language features, the metamodel architect creates a feature node that is named after the language feature. From here on, we do not distinguish between features and feature nodes, because of the almost 1:1 relation between them. The metamodel architect has to declare a relation from a feature F to another feature G according to the following rules:

- **Requires Relation:** If a reused module that implements F has a dependency to a reused feature that implements G.
- **Requires Relation:** If feature F is an extension of feature G.
- **Requires Relation:** If feature F is dependent on content of feature G.
- **Excludes Relation:** If feature F prohibits the use of feature G or vice versa.

Cycles of requires relations are forbidden. The metamodel architect solves these cycles by reversing requires relations.

4) Layering: In this step, features are vertically split and assigned to layers following the guidelines in Section 4.2. The metamodel architect assigns features that contain language features only relevant to a single layer to that layer. S/he performs the following steps for each layer except for π, starting from the next basic layer.

4.a) If an unassigned feature contains features relevant to the current layer alongside with other features, s/he creates a new feature to hold the features not relevant to the current layer. S/he assigns the original feature to the current layer; the new feature remains unassigned (it will be handled further when the next layer is modularized). S/he declares a requires relation from the new feature to the original feature.

4.b) S/he reverses all feature relations coming from features of more basic layers to features of this layer.

5) Paradigm Extraction: The root feature is always part of the π layer. To form the remaining π, the metamodel architect considers for each language feature whether it contains any fundamental atomic language features or patterns. For these fundamental atomic language features and patterns, s/he creates new features in π and creates requires relations pointing to them from the dependent Δ language features.

6) Feature Grouping: Grouping of language features is either used to achieve a logical structuring (without effect on feature selection) or to form feature sets (with effects on features selection, see Section 2). Groups of features can be formed from features of the same layer. For each group, the metamodel architect creates a new feature within the same layer, and makes it the parent of each feature of the group. Grouping can be done according to multiple reasons. Multiple features could share a commonality (e.g., they are all structural abstractions, viewtypes [40], or of the same type). In some cases, groups have to be used to form feature sets (i.e., alternative sets or OR sets). If two or more features are fully interconnected with excludes relations, the metamodel architect has to use an alternative feature set. The alternative feature set then replaces all excludes relations.

7) Parent Feature Identification: First, the metamodel architect identifies all features that are direct children of the root amongst the π features by the following indicators:

- A feature contains atomic language features that are fundamental to the language.
- A feature represents a viewtype [40] (sometimes called sub-model [3]).
- A feature contains atomic language features that are shared by all viewtypes.
- A feature has no outgoing feature dependencies.

Next, s/he identifies the parents of the remaining features, which do not have a parent yet. One of the features to which a requires relation exists is usually the parent. If a feature is an extension of another feature, s/he declares a parent relation from the extending to the extended feature. In all cases, the parent relation replaces an existing dependency relation between the two features. Like the requires relations, a parent relation cannot point into a more specific layer.

8) Child Feature Type Determination: Some features already got their type in step (5). These features are either part of alternative sets or OR sets and remain this way. For the other features, which do not yet have a parent, the metamodel architect specifies the child features types as follows. The root feature has no parent but is always mandatory. Parent features of feature sets are always mandatory. Child relations that cross the π layer boundary are always OR sets (even if the parent has only one child). This enforces that π features cannot be selected on their own (but always together with at least one child). Child relations that cross the other layer boundaries are optional. If this were not the case, there would be a hard coupling between the layers. The remaining features, which do not yet have a type assigned to their child relation, are optional.

9) Feature Implementation: The module developer implements each feature by metamodel modules. Exceptions are parents of feature sets, the root feature, and features that are already completely implemented by reused metamodel modules. If the module developer introduces new module dependencies that are not conforming to the feature graph, the metamodel architect carries out the following steps. The

module developer and the metamodel architect consider the new feature dependency D from the feature F that is implemented by metamodel module M to the feature G that is implemented metamodel module N.

9.1) If the information that is modeled by D is already present in the implementation of N and is only used to ease backward navigation, the metamodel architect omits the dependency.

9.2) If there is no opposing dependency (i.e., G is not dependent on F) and the new dependency that will be introduced by D is meaningful in this specific context, s/he creates a new feature dependency from F to G.

9.3) If there is an opposing feature dependency, the metamodel architect considers dependency inversion of D (see Section 6.1).

9.4) If none of the above options are feasible, the metamodel architect declares a new feature dependency from F to G. This will result in a dependency cycle between F and G, which has to be resolved in the next step.

In case a new feature dependency is created, or an existing feature dependency is inverted, the restrictions from the layering have to be adhered to.

10) Revision and Refinement: Using the module refactorings described in Section 6.2, the metamodel architect can revise and refine the feature model to resolve issues like dependency cycles, multilayer features and features that fulfill multiple responsibilities. After each refactoring, the metamodel architect updates the feature model.

6.4 Refactor existing Metamodels

We specify the following process for refactoring an existing metamodel to fit the proposed reference architecture. The single steps of the process are not intended to be executed in a strictly sequential manner. Excerpts from the refactoring of the PCM are used to exemplify the process.

This refactoring process can be applied to a metamodel, if it has the potential to be modularized. For example, multiple language features are implemented in one metamodel module or there are dependency cycles between modules.

In this process, a feature graph is used as a predecessor stage of a proper feature model. It consists only of features and requires relations. In contrast to a feature model, its parent-child relations do not have to form a tree, because there can be multiple roots (features with no outgoing dependencies).

1) Horizontal Decomposition: First, the metamodel architect investigates the metamodel and its documentation, if existent, to identify the language features the metamodel implements. This step is prerequisite for the subsequent process steps. The metamodel architect subdivides existing metamodel modules according to the horizontal split refactoring until they only implement a single responsibility. The dependencies are not yet adjusted, this is done in a later step in the process. A good starting point for the decomposition is given by the package structure and the outline of the documentation. The result of this step is a set of metamodel modules that may be strongly interconnected and possibly contain dependency cycles. These shortcomings have to be refactored in the following steps.

For example, the largest metamodel module of the PCM was split to separate various viewtypes like resources, software repository, assembly and usage.

2) Feature Graph Creation: The metamodel architect first creates a feature for each metamodel module. Then, regardless of the module dependencies, the metamodel architect declares feature dependencies according to the guidelines and constraints of the reference architecture.

In the PCM, for example, the feature for specifying internal behaviors of a software service depends on the feature for specifying the service.

3) Dependency Alignment: In this step, the metamodel architect inspects module dependencies that are not in line with the feature graph. S/he starts with the most specific modules. These are usually the ones with the least incoming dependencies. For each incoming and outgoing dependency D on the classifier level, that is not reflected in the feature graph, the metamodel architect executes the following steps. D points from metamodel module M to N.

3.1) S/he checks whether the affected classifiers in both metamodel modules are correctly placed. If not, S/he moves the respective classifiers into the other metamodel module.

3.2) If a classifier is encountered that does not fit M nor N, s/he considers whether it either belongs to another metamodel module or whether it (and possibly further classifiers) can be factored out into a new metamodel module.

3.3) If there is a feature dependency from N to M, s/he considers dependency inversion of D.

3.4) If there is no feature dependency from N to M, s/he considers introducing a feature dependency from M to N.

3.5) If there is a feature dependency from N to M, s/he considers reversing it. If it is meaningful, s/he reverses all inter-module dependencies that go from N to M as well.

S/he updates the feature graph accordingly. The result of this step is a modular metamodel that is free of dependency cycles, and all inter-module dependencies conform to a feature dependency.

In the modularization of the PCM, there was no need for dependency alignment, as the module graph was thoroughly designed. Yet, in the modularization of other case study metamodels, we performed dependency alignment.

4) Vertical Decomposition: Metamodels in the focus of our research can be reused, at least in parts, for modeling and analyzing different quality properties or even different domains. This optional step can be performed to improve the reusability of the metamodel. The metamodel architect assigns metamodel modules that only implement language features relevant to a specific layer to that layer. On each metamodel module M that implements language features belonging to multiple layers, the metamodel architect and the module developer perform the vertical split refactoring. The metamodel architect updates the feature graph accordingly.

An example for a vertical split in the PCM is the extraction of performance-specific class properties from the resources metamodel module. Therefore, several classes were split to move the performance properties into the Δ layer.

5) Paradigm Extraction: In this step, the metamodel architect inspects Δ for atomic language features and patterns that are fundamental to the language and can be reused in other domains. Suitable candidates are often amongst the

packages that contain mostly abstract classes. If there is an atomic language feature or pattern to be factored out whose classes are not abstract, the module developer can factor it out into abstract classes from which the concrete classes then inherit. The module developer moves class properties that belong to the atomic language feature or pattern into the abstract classes, while domain-specific properties stay in the concrete classes. Incoming dependencies remain on the concrete classes. After each refactoring, the metamodel architect updates the feature graph accordingly. The result of this step is a cycle-free layered feature graph.

An example for a paradigm extraction in the PCM is to split repository. The abstract definition of components and interfaces is located at the π layer. We factored out software specific properties, like signature lists for interfaces, into the software repository module in Δ.

6) Feature Model Forming: In this step, the metamodel architect transforms the feature graph into a feature model. First, the metamodel architect creates a root feature. Then, the metamodel architect performs the steps Feature Grouping, Parent Identification and Child Feature Types Determination from Section 6.3.

7 Case Studies

In this section, we introduce case studies for applying the reference architecture before we provide a discussion of requirement in Section 8 and a detailed evaluation in Section 9. We apply the reference architecture to four metamodels: the PCM [3], Smart Grid Topology [41] (a DSML for modeling and resilience analysis in smart grid topologies), KAMP4aPS [42] (a DSML for modeling and predicting the maintainability of automated production systems) and the BPMN2 [33] (a DSML for modeling business processes). We documented the module structure of the original and modularized versions of the metamodels, links to all source metamodel files and summaries of the refactorings in a technical report [43].

It is important to note we created the modular versions of the case study metamodels primarily for evaluation. We refactored them solely according to the rules of the reference architecture. We did not fix bad smells that the reference architecture does not address as this would damage the validity of the evaluation. For the PCM, we documented errors and bad smells that we identified but did not fix [43]. Examples of such smells are redundant (container) relations that did not violate the constraints of our approach, dead classes and duplications due to missing superclasses.

We first explain the criteria for selection of metamodels and extension mechanisms. Next, we explain the stopping criteria for the modularization process in the case studies. The remainder presents the four case study metamodels. Exemplarily, we only go into detail about the metamodel module structure of the PCM due to lack of space.

Case Study Selection Criteria: One criterion for the metamodel selection for the case studies is the metamodel must be available open source. For example, the AUTOSAR [44] metamodel seems to be modularizable according to our reference architecture but is not publicly available. The metamodels should not be too excessive in size, as the effort for understanding the metamodel, acquiring the needed domain knowledge and performing the refactoring strongly increases with the size of the metamodel. So, we limited the size of case study candidates to metamodels with less than 300 classes. For example, the Capella [13] metamodel was not chosen as it defines 413 classes. Moreover, the metamodels must have some modularization potential. We also put a focus on the heterogeneity of the metamodels regarding new vs. old, small vs. big size, many vs. few layers, and different domains and analyses. We prioritized metamodels that have instances available, as they are needed for the evaluation. Further, we preferred metamodels that have a package structure in contrast to metamodels that consist only of one package that contains a high number of classifiers. Such flat metamodels are often the result when metamodels are transformed to Ecore from other metamodeling language that do not support the concept of packages (e.g., XSD).

Applied Extension Mechanisms: For the implementation of the case studies, we used the inheritance extension mechanism (see Figure 5 (3)) and the plain referencing extension mechanism variant with inheritance (2a) to an existing container where possible, as they introduce the least amount of new classes. Where these extension mechanisms could not be used, we chose the plain referencing variant with an explicit container (2b). The applicability of the profiles extension mechanism (1) is identical to the applicability of the plain referencing variant with an explicit container, as the use of both mechanisms does not depend on the presence of predefined containers. The number of new classifiers (if a stereotype is considered a classifier) introduced by both extension mechanisms is equal if the extension references further classes. If only attributes are added, the profiling requires one classifier less than referencing with a container, as the stereotype can directly contain the attributes (so-called tagged values). On the other hand, plain referencing uses the standard Ecore modeling concepts. This simplifies gathering evaluation results, as we were able to use the standard EMF API and tools to process metamodels and models. We chose plain referencing (2a and 2b) over profiles (1) because of this reason. If we had chosen profiles, the modularized versions of the metamodels would be even less complex. Note, in addition we apply inheritance extension (3).

Modularization Stopping Criteria: We refactored all four case studies until they satisfied the following criteria: (1) full vertical decomposition (each metamodel module can be assigned to exactly one layer), (2) no feature dependency and no module dependency direction violates the direction of the layering, (3) full horizontal decomposition (each metamodel module is at most as extensive as a user language feature), (4) no dependency cycles. The PCM, Smart Grid Topology, and KAMP4aPS case studies fulfill an additional criterion: (5) decoupled extensions. We used dependency inversion to decouple all metamodel modules from all other metamodel modules that represent extensions. In BPMN2 we decoupled as many extensions until we reached a point where further dependency inversion would merely decrease coupling and thus further increase the observed benefit.

7.1 PCM

The PCM has already been introduced in Section 1. For information about the size of the original metamodel see

Section 3. We split the biggest metamodel module into 23 smaller modules to separate user language features properly. The other four metamodel modules were already sufficiently modular. The number of classes in the modularized PCM (mPCM) grew from 203 to 229. This is due to splitting classes during refactoring and the creation of new containers for extensions. The number of references dropped from 198 to 174, as redundant dependencies that violated the reference architecture were removed or remodeled. The mPCM populates the layers π, Δ, and Ω. Σ is populated by extensions of the PCM, which are also used in the evaluation.

As an example of a layered and modular metamodel, we give an overview of an excerpt of the metamodel modules of the mPCM in Figure 7. Six metamodel modules of the π layer and five metamodel modules of the Δ layer are omitted for reasons of space and clarity of visualization. Also, transitive dependencies are not shown.

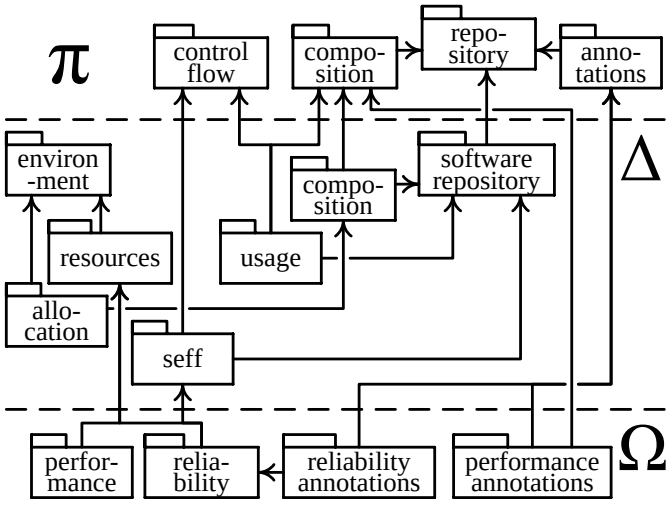

Fig. 7. Excerpt of the Metamodel Modules of the mPCM

The most relevant π metamodel modules are: the *repository*, which defines abstract components, interfaces, and roles; *composition*, which enables component composition; *control flow*, which provides a structure similar to activity diagrams. The domain (Δ) layer contains the *composition* and *software repository* metamodel modules, which extend their counterpart from the π layer and carry additional domain-specific content. The *environment* metamodel module defines execution containers and network links between them. It is extended by the *resources* metamodel module, which adds hardware resource specifications to the execution containers and network links. Using the *allocation* metamodel module, component instances (from the *composition* metamodel module) can be deployed on the execution containers of the *environment* metamodel module. The *usage* metamodel module defines usage profiles, which can be applied to interfaces from the *repository*. It reuses the *control flow* metamodel module of π, which is also reused by the *seff* metamodel module in Δ. It enables modeling the behavior of operations that components provide. The quality (Ω) layer consists of the *performance* metamodel module, which requires the *resource* extension of the *environment* metamodel module. It also adds resource demands to the *seff* specification of components. The reliability dependencies are analogous. There are also two metamodel modules which enable annotation of both quality properties in a component-based architecture. They reuse the abstract definition of annotations in the π layer.

7.2 Smart Grid Topology

This metamodel is used for impact and resilience analysis for smart grid topologies. It was chosen, as in contrast to the PCM, it is a smaller, younger, more stable and more modular metamodel, and covers a different domain. Its development started in January 2014. It was initially released in October 2015. It consists of 30 classes in 3 metamodel modules. The main metamodel module defines language features for modeling smart grid topologies. The other metamodel modules are used for the input and output state of an analysis. During the modularization, the main metamodel module was split: two π metamodel modules were factored out (one with common superclasses and one with superclasses for the graph structures of the topology) and one Δ metamodel module was factored out (it contains language features that represent the types of devices within a smart grid). The number of modules increased to 6 and the number of classes to 34. The number of dependencies increased from 60 to 66. The resulting modular metamodel populates the layers π, Δ, and Σ. The analysis operates solely on the structural parts of the topology that are defined in Δ. As mentioned in Section 4.2, metamodels for quality analysis does not necessarily need to populate the Ω layer.

7.3 KAMP4aPS

KAMP4aPS is used to model automated production systems and predict the impacts of changes in these systems. We chose KAMP4aPS, as it covers a further domain and, in contrast to the other metamodels, recently completed its initial development. It has been under development since 2016. It contains 5 metamodel modules with 185 classes in total. During the modularization, the metamodel module that describes the systems was split into parts of different specificity: automation systems (most general metamodel module), automated production systems, and a specialization for a specific kind of automated production system, called a pick an place unit (most specific). The same kind of modularization was performed on the module that describes modifications to the systems. The refactoring increased the number of metamodel modules to 9. The number of classes stayed constant as existing containers could be well utilized. The number of dependencies dropped from 395 to 390, as some redundant opposite references were removed that violated the reference architecture.

7.4 BPMN2

The Business Process Model And Notation 2 (BPMN2) is a DSML by the OMG used for modeling of business processes. It is a suitable case study metamodel, as there exist several quality analyses for BPMN2 (see [45]). In this case study, we focus on the BPMN2 metamodel, which occupies the π and Δ layer. We chose BPMN2 as it is an ISO standard, widely used and covers yet another domain. BPMN was

first released in March 2007. In January 2011, its successor BPMN2 introduced further language features (e.g., choreography, conversation, non-interrupting events and event subprocesses). It consists of 4 metamodel modules. One metamodel module defines all the concepts of BPMN2, which we call main metamodel module in the following. The other three metamodel modules are needed to model graphical diagrams. During the refactoring, the main metamodel module was modularized according to its user language features into 25 metamodel modules (resulting in 28 metamodel modules in total). 16 of these metamodel modules are on the π layer; 9 are on the Δ layer. The number of classes grew only slightly from 157 to 163. This is because often we were able to inherit from the abstract class RootElement that provides a generic extension point, as it is contained in a root class. The number of dependencies slightly reduced from 529 to 527 (mainly because of redundant relations that violated the reference architecture).

8 Discussion of the Requirements

After introducing the case studies and describing their modularization in the previous section, we now discuss whether the requirements on the reference architecture are satisfied in the case studies. The requirements **R1** (Improved Evolvability), **R5** (Need-specific Reuse) and **R6** (Need-specific Use) cannot be addressed by mere discussion but require more in-depth evaluation described in the next section.

Non-intrusive Extension: In the case studies, we implemented the modularized metamodels using inheritance where possible and both variants of plain referencing in the remaining cases. If an extends relation had to be created from one metamodel module M to another metamodel module N, it was possible to achieve this without having to modify N (for both extension mechanisms). The implementation of the extends relation did not introduce any dependencies from N to M, in the case studies. N is not dependent on M after the extends relation has been implemented. In conclusion, the extension mechanisms used in the case studies satisfied **R2** (Non-intrusive Extension).

Instance Compatibility: We observed that the extension mechanisms inheritance and plain referencing behaved differently regarding **R3** (Instance Compatibility).

In situations where we used plain referencing and created a new container, extended models were still compatible with tooling of the base metamodel. This is because instances of extending classes are contained in an instance of the new container, which is persisted in a separate model file. Therefore, **R3** (Instance Compatibility) is satisfied for plain referencing with a new container.

In situations where we used the inheritance extension mechanism or the variant of plain referencing where an inheritance relation is created to a contained class of the extended metamodel, an extended model could not be loaded by tooling of the base metamodel, when the extending metamodel module was not installed. This results from instances of the extending classes that are contained in instances of containers from the original model. The tooling tried to load all classes, encountered these unknown instances and crashed. Thus, **R3** (Instance Compatibility) is not satisfied for inheritance and and the plain referencing variant with inheritance. This is, however, only a technical shortcoming. It can be fixed by treating the case of unknown content either in the modeling runtime (e.g., EMF) or the tools. Besides ignoring the unknown content, another option is to install the extension metamodel module on the fly (e.g., in Eclipse via an update site). That way, the model can be processed and the tool user does not notice any change. Even if one would ignore these technical workarounds, this is no problem to our approach, as one can always default to plain referencing with external containers. In consequence, we consider **R3** (Instance Compatibility) satisfied by our reference architecture.

Independent Extension: We extended classes by other classes that are located in various metamodel modules. With the extension mechanisms we used, it is possible to supply an instance of such a class with the content of multiple extensions at the same time. These extensions are unaware of each other and were not developed in such a way that would explicitly allow the other extension. Thus, **R4** (Independent Extension) is satisfied by our reference architecture.

9 Evaluation

This section describes the evaluation of the reference architecture with respect to the requirements **R1** (Improved Evolvability), **R5** (Need-specific Reuse) and **R6** (Need-specific Use). Section 9.1 presents the evaluation goals and metrics. Section 9.2 explains the evaluation design. Section 9.3 presents the results and Section 9.4 interprets the results. Section 9.5 summarizes the conclusions. Threats to validity are discussed in Section 9.6.

In the evaluation section, we speak about metamodel modules and their contained packages, not language features, as we evaluate the case studies on a technical level.

9.1 Evaluation Goals and Metrics

This section presents the evaluation goals and explains how we break them down to specific metrics for: (1) evolvability and (2) need-specific use and reuse.

9.1.1 Evolvability

We first introduce the evaluation goal. Second, we give a detailed explanation of the corresponding metrics. Third, we describe the scenario-based evaluation and the extraction of metamodel parts. Fourth, we explain how the metamodel parts are transformed before we can apply the metrics.

The evaluation goal and metrics are derived from **R1** (Improved Evolvability) as follows.

Goal 1: is to analyze the metamodels for the purpose of evaluating the improvement of the metamodels' evolvability by comparing the original metamodels to the metamodels that we modularized according to the reference architecture.

While there is a variety of publications related to software evolvability, to the best of our knowledge, there is no definition of evolvability for metamodels. Breivold et al. proposed a software evolvability model [46] which outlines the sub-characteristics analyzability, integrity, changeability, extensibility, portability, and testability. The evolvability sub-characteristics of Breivold et al. are covered by

the established ISO/IEC 25010 software quality model [1] in the characteristics maintainability and portability. The characteristic maintainability in ISO/IEC 25010 covers the sub-characteristics analysability, changeability, stability and testability. Portability covers the sub-characteristics adaptability, installability, co-existence and replaceability.

Due to the lack of evolvability definitions for metamodels, we discuss the adaption of software characteristics to metamodel in the following. Adapting the evolvability characteristics of software to metamodels seems reasonable as, in analogy to any other software artifact, also metamodels face evolutionary changes due to emerging and changing requirements [47]. However, the characteristics of ISO/IEC 25010 and Breivold et al. have been specified with software products in mind. As a consequence, they partly address properties which are not valid for metamodel evolvability. The definitions of portability and its sub-characteristics focus on transferring a software product from one execution environment to another, which is not useful for evolving metamodels. Testability is the subject of recent research in the MDE community, like at the MDEbug workshop[5] initiated in 2017. Work on testability in the MDE community mainly focuses on debugging model transformations which is not in the focus of our research.

Other characteristics are well applicable to metamodels. These are analysability, changeability and stability (extensibility and integrity in [46]). Changeability and stability are often referred to as modifiability in literature.

To the best of our knowledge, there are no specific metrics that have been validated to represent metamodel quality [48], [49]. According to Cruz-Lemus et al. [50] and Briand et al. [51] analysability and modifiability of a model is affected by its cognitive complexity. Cognitive complexity of a model is hard to measure. Therefore, we follow the argumentation in [50] and refer to the amount of structural information within a model as structural complexity. Allen et al. [52] proposed metrics of information size, complexity, and coupling regarding the information entropy, based on formal definitions proposed by Briand et al. [53].

Hypergraph Metrics: The metrics proposed by Allen et al. are based on graph [54] and hypergraph [52] abstractions of models. They represent the information entropy of the graphs and hypergraphs [55]. In contrast to simple counting metrics, the metrics by Allen et al. include the interconnection of nodes and hyperedges. High entropy values indicate strong interconnection within the graphs and hypergraphs [56]. Hence, the metrics by Allen et al. are well appropriate for evaluating complexity, cohesion, and coupling in the evolution of metamodels.

A hypergraph consists of nodes and hyperedges, where a hyperedge can connect any number of nodes. We apply hypergraphs for evaluation as according to Schütt [57] and Allen et al. [52], software engineering abstractions, like set-use relations for public variables, are better represented as hypergraphs than ordinary (binary) graphs. We follow this argumentation for metamodel modules and their dependencies in our evaluation. For the evaluation, we use a hypergraph partitioned into several hypergraph modules we denote as modular hypergraph H. A hypergraph module is a set of nodes. Each node can only be contained in one of the hypergraph modules of H. We denote hyperedges crossing hypergraph module boundaries as inter-module hyperedges. Hyperedges that do not cross hypergraph module boundaries are named intra-module hyperedges.

For calculating the *complexity* of a modular hypergraph, we performed a procedure taken from [56] based on the size metric by Allen et al. In order to calculate the size of a hypergraph, we establish a pattern for each node describing the hyperedges connected or not connected to the node in form of ones and zeros. The pattern (i.e., sequence of ones and zeros) for several nodes may be identical. In that case, we aggregate them and remember the number of occurrences. Then, we calculate the probability of each pattern p by the ratio of number of occurrences and number of nodes in H [52]. Equation 1 and Equation 2 depict the metrics for size and complexity. $\mathbf{G}$ is the modular hypergraph. $\mathbf{G}_i$ is the modular hypergraph containing node i and all nodes which are connected to this node. $p_L(j)$ provides the pattern probability of node j. The size metric is first used on all G_i partial hypergraphs and then on the complete hypergraph G. Therefore, H indicates that different hypergraphs are passed to the size metric.

$$Size(H) = \sum_{j=1}^{n}(-log_2\ p_L(i)) \quad (1)$$

$$Complexity(G) = \left(\sum_{i=1}^{n} Size(G_i)\right) - Size(G) \quad (2)$$

The *coupling* of a modular hypergraph is specified as the complexity of the hypergraph with only inter-module hyperedges [52]. Following the procedure for the computation of coupling in [56], we construct a modular hypergraph H^* containing only inter-module hyperedges. Then, we calculate the complexity of H^*.

Allen [54] defines *cohesion* as the ratio of the complexity of the intra-module graph MG^o and the complexity of the complete graph $MG^{(n)}$. A complete graph is a graph for which all nodes are interconnected by edges [54]. We cannot construct a meaningful complete graph for a hypergraph. This is because a complete hypergraph would not only contain hyperedges between two nodes but also all other hyperedges for a given set of nodes [56]. Therefore, we apply the cohesion metric by Allen [54] to graphs, not hypergraphs. We follow the procedure described in [56]. First, we map the modular hypergraph H to a modular graph MG. We replace each hyperedge by a set of edges connecting all nodes that were previously connected by the hyperedge. Based on MG, we then derive a graph containing only intra-module edges MG^o and construct a complete graph $MG^{(n)}$. Cohesion is calculated as shown in Equation 3.

$$Cohesion(MG) = \frac{Complexity(MG^o)}{Complexity(MG^{(n)})} \quad (3)$$

Extraction of Relevant Subgraphs: Evolvability is not an absolute property. It is always to be considered in the context of a specific evolution scenario [58]. Because of this, we do not apply metrics on a metamodel as a whole; instead, we perform a scenario-based evaluation by applying the

[5]https://msdl.uantwerpen.be/conferences/MDEbug

metrics on the part of the metamodel that is relevant to the evolution scenario. In the following, we call the part of a metamodel that is relevant to an evolution scenario the *subgraph* of the scenario.

For each evolution scenario, we extract a subgraph as an approximation of the part of the metamodel to be inspected by the developer when s/he is conducting the evolution scenario. We form the subgraphs, starting from the classes that are modified or extended by the evolution scenario. In the following, we refer to such classes as *affected classes* of an evolution scenario. The affected classes have to be known and understood by the metamodel developer to be able to perform a modification or extension. From the affected classes, a subgraph is built by following containment references, the superclass hierarchy, dependencies due to generics, mandatory references (i.e., references having a lower multiplicity bound of at least one) and including all classes from the same package.

Metamodel Subgraph to Hypergraph Transformation: To be able to apply the metrics to a subgraph, we have to map metamodel concepts onto modular hypergraph concepts. First, all packages of the metamodel subgraph are mapped to hypergraph modules. Second, each class of the metamodel subgraph is mapped to a node, and the node is placed in the correct hypergraph module. Third, edges are constructed between the nodes. Non-generic inheritances, references, containments, type bounds and extends relations of classes of the subgraph are transformed into regular edges (hyperedges with only two ends). References to and inheritances of generic classes are transformed into a hyperedge (with potentially more than two ends due to type arguments). The ends of such a hyperedge are the class which owns the dependency, the class the dependency points at and all classes which appear in type arguments (if there are any). During the transformation of dependencies to hyperedges, classes might be dependent on other classes located outside of the relevant subgraph. This is only the case, if the dependency is a reference with a lower multiplicity of 0. References with a lower multiplicity of at least 1 are already included in the subgraph. For references with a lower multiplicity of 0, nodes are also created and placed into the right hypergraph module. Their outgoing dependencies, however, will not be transformed. Such border classes must be included, as they resemble a dependency to a part outside of the subgraph, which must be considered by the developer. However, the developer does not need to know all dependencies of the class, as they are outside of his/her scope. It can be seen as an interface to another metamodel module. Attribute types do not play a role in the understanding of the metamodel on the user language feature level and are thus ignored.

Transforming packages to hypergraph modules brings some implications. Coupling is measured between packages and cohesion is measured within packages. The alternative to transforming packages to hypergraph modules is to transform metamodel modules to hypergraph modules. However, we decided to transform packages, as several case study metamodels are monolithic. They consist of one large metamodel module and few smaller ones. These monolithic metamodels would perform very poorly when transforming metamodel modules. Thus, we decided to calculate the metrics on the basis of packages to allow a more nuanced evaluation.

9.1.2 Need-specific Use and Reuse

To evaluate **R5** (Need-specific Reuse) and **R6** (Need-specific Use), we have to show that a metamodel refactored according to the reference architecture enables more targeted use and reuse. We can evaluate both requirements together, as the acts of using a metamodel module (as a tool user or tool developer) and reusing a metamodel module (as a metamodel developer) are technically the same: Usage is only possible via a tool that has a requires dependency to the metamodel modules it uses. These requires dependencies are defined by tool developers and used by tool users (if they need the language features in question). Reuse is done by creating a requires dependency from a (possibly new) metamodel module to the reused metamodel module.

As based on a given metamodel various models can be created for specific needs, we apply models to evaluate the use and reuse of metamodel parts. To evaluate both requirements:

Goal 2: is to analyze models for the purpose of evaluating the improvement of need-specific use and reuse by comparing the original metamodel to the metamodel that is modularized according to the reference architecture.

To be able to quantify the improvement of need-specific use and reuse, we need to calculate the ratio of how much of a metamodel is used by a model. For this reason, we define the $mmUtil()$ metric (see Equation 4), which is the short form for metamodel utilization. The utilization metric divides the number of classes that a model M instantiates ($NumInstantiatedClasses()$) by the total number of classes ($NumClasses()$) of the metamodel modules that are necessary to load the model ($InstantiatedModules()$). This evaluation only regards classes, as the other classifiers are used not by instantiation but the creation of attributes in the metamodel. We use $InstantiatedModules()$, as the smallest unit of use and reuse is a metamodel module. If a model instantiates at least one class of a metamodel module, the whole metamodel module has to be used. The more classes (of a constant set of metamodel modules) are used, the higher the utilization. The best value of $mmUtil()$ is 1. This is the case when M instantiates all classes at least once. A class also counts as instantiated if it has a subclass (it does not have to be a direct subclass) that is instantiated. Each instantiated class is counted only once, regardless how often it is instantiated.

$$mmUtil(M) = \frac{NumInstantiatedClasses(M)}{NumClasses(InstantiatedModules(M))} \quad (4)$$

To compute $mmUtil()$ we infer the types (i.e., classes) of the objects in the models. We then collect all superclasses. This results in $NumInstantiatedClasses()$. We then determine which metamodel modules have to be delivered to be able to load the model ($InstantiatedModules()$). These are the metamodel modules where the instantiated classes and their superclasses are located and also all metamodel modules these modules depend on. The total count of classes in these metamodel modules is provided by $NumClasses()$.

9.2 Evaluation Design

This subsection explains the rationale behind the evaluation design. For the evolvability evaluation, it explains the types and the selection of evolution scenarios. For the use and reuse evaluation, it explains the selection of models. We already explained the selection of metamodels in Section 7.

9.2.1 Evolvability

We first give an overview of the evolution scenarios for the case studies. Then, we exemplarily describe two evolution scenarios in detail. Due to space restrictions, the full description of all evolution scenarios can be found in our technical report [43].

An evolution scenario can be either a modification or an extension. As the procedure for evaluating both types of evolution scenarios is the same, we do not distinguish between modification and extension in the evaluation. First, we gathered *historical evolution scenarios*. To collect historical modifications, we searched in available change logs of repositories for modifications. We collected historical extensions by identifying the sources of the metamodel extensions in repositories. Further, we collected what we call *potential evolution scenarios*. We did this by reviewing the metamodel and identifying classes which in the future may face a change or extension. If the search for historical and potential scenarios did not yield enough results, we randomly chose classes for modification or extension from packages that did not yet face a historical or potential evolution scenario. We denote these as *generic evolution scenarios*. Examples of generic evolution scenarios are modifications of names and multiplicities, additions of attributes and dependencies to other classes, deletions of class properties. A sufficient number of evolution scenarios is needed for a good variety in the extracted subgraphs.

For the subgraph extraction of historical modification scenarios based on the current version of a metamodel three cases can be distinguished as discussed in the following. Case 1: additions, property changes and deletions with the exception of class deletions. Case 2: class deletion in the scenario. Case 3: class deletion after the scenario.

Case 1: considering the procedure in Section 9.1.1, the evaluation of a historical modification scenario is straightforward, with the exception of deletions of classes. For example, if a property change is evaluated on a later version of the metamodel, the class is simply declared as an affected class. For the subgraph extraction procedure it is irrelevant which property of the element was changed and how it changed. Hence, it does not matter that the historical modification scenario was already applied in the past.

Case 2: although it is not as simple as the previous case, historical modification scenarios that contain class deletions can also be considered in the evaluation as follows. The deleted class is removed from the set of affected classes of the scenario, as it is no longer present in the metamodel and would cause errors in the subgraph extraction. The dependencies of the deleted class are then manually added to the affected classes according to the rules of the subgraph extraction (see Section 9.1.1). This enables the inclusion of all dependencies of the deleted class in the subgraph extraction.

Case 3: for all class deletions after the scenario, the same procedure is applied as for scenarios that contain class deletions.

By following this procedure, the subgraph of the historical modification scenario can be reproduced, assuming there was no further evolution. If there were further class deletions, the procedure has to be repeated. Thus, the same results are achieved as of an evaluation of the scenario on the actually modified metamodel. Threats to validity that may arise from evolution of the metamodel after the scenario was executed are discussed in Section 9.6.

For the PCM, we collected 13 historical evolution scenarios and one potential evolution scenario. Collected historical extension scenarios for the PCM are optional extensions, i.e., they do not implement any core features of Palladio and, therefore, are not delivered with a standard installation of the PCM. The extension scenarios for the PCM are KAMP [58] (not to be confused with KAMP4aPS, which is a standalone DSML) and IntBIIS [18]. We chose them because they are up-to-date and heterogeneous concerning the parts of the PCM they depend on. We collected 11 historical modification scenarios for the PCM from its change log[6]. We started with the most recent changes and selected the ones that actually changed the structure of the metamodel and not just the genmodel, version numbers or namespaces. We skipped repeated modification of the same class. In addition, there was one proposed modification in the change log, that we consider as a potential evolution scenario.

The Smart Grid Topology metamodel has been stable since its initial release. So we cannot deduce any historical evolution scenarios from change logs or its repository. Following the aforementioned scenario collection procedure results in eight evolution scenarios (four potential and four generic). For the KAMP4aPS case study, we collected 18 evolution scenarios (10 potential and eight generic). For BPMN2, we collected 23 generic evolution scenarios.

Exemplarily, we present two historical modification scenarios of the PCM. In the ProcResSpec scenario, an inheritance relation is introduced from ProcessingResourceSpecification to Identifier. The ProcResSpec scenario has ProcessingResourceSpecification and Identifier as affected classes. The ProcessingResourceSpecification class is located in the resources metamodel module of Δ. Identifier is located in the identifier metamodel module in π (not shown in Figure 7). In the ComLinkResType scenario, a supertype of CommunicationLinkResourceType is changed to ResourceType instead of ProcessingResourceType. The scenario has these three affected classes for the subgraph creation. The CommunicationLinkResourceType is also located in the resources metamodel module.

9.2.2 Need-specific Use and Reuse

To evaluate $mmUtil$ for the PCM, Smart Grid Topology and KAMP4aPS case studies, we collected all models that were available to us (611 PCM models, 28 Smart Grid Topology models and 30 KAMP4aPS models). The number of existing BPMN2 models is much higher, because, in contrast to the other case studies, there is a public online repository for BPMN2 models[7]. For BPMN2, we collected 103 models

[6]https://sdqweb.ipd.kit.edu/wiki/PCM_Changelog
[7]https://github.com/camunda/bpmn-for-research

from internal sources [59], [60] and 3739 models from the repository. For PCM, Smart Grid Topology and KAMP4aPS, all models were valid. For BPMN2, 46 models were invalid. These models could not be loaded and, therefore, were ignored in our evaluation.

9.3 Evaluation Results

In this subsection, we present the results of the evaluation. We use the metrics to compare the original versions of the metamodels to the modularized versions. Thus, the absolute values of the metrics are of less importance to us. The sources to our evaluation tool and the raw results can be found in our technical report [43].

9.3.1 Evolvability

The results of the hypergraph metrics analysis are shown in Figure 8 (PCM), Figure 9 (Smart Grid Topology), Figure 10 (KAMP4aPS) and Figure 11 (BPMN2). The diagrams show the results for the metrics that are labeled on the right side. The upper box contains complexity results, the middle box shows the coupling (between packages), and the lower box presents the cohesion (inside packages). The values of the metrics are plotted at the y-axis at the left side. At the x-axis, the names of the evolution scenarios are listed. The scenarios are marked with their scenario type: historical evolution†, potential evolution$^{\times}$ and generic evolution$^{\circ}$. If they produce the same subgraph, multiple scenarios result in same metric results. In these cases, only the name of the alphabetically first scenario is shown. How many scenarios produced the same result, is denoted by the number in brackets after the scenario name. For each scenario, both versions of the metamodel were evaluated: the original one (black) and the version that was modularized according to the reference architecture (gray).

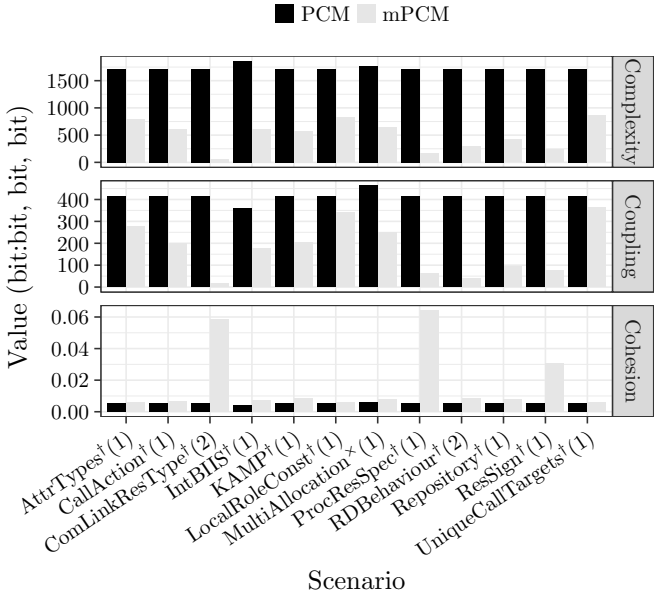

Fig. 8. Evolvability Metric Results: PCM

The unit for complexity and coupling is bit, as both metrics measure information size known from information theory. Their value range is unbounded. Low complexity and low coupling values are good. The unit for cohesion is ratio of bits, i.e. the ratio of the current cohesion compared to the cohesion of the maximal cohesive graph. Thus, its value range is between zero and one. High cohesion value is good.

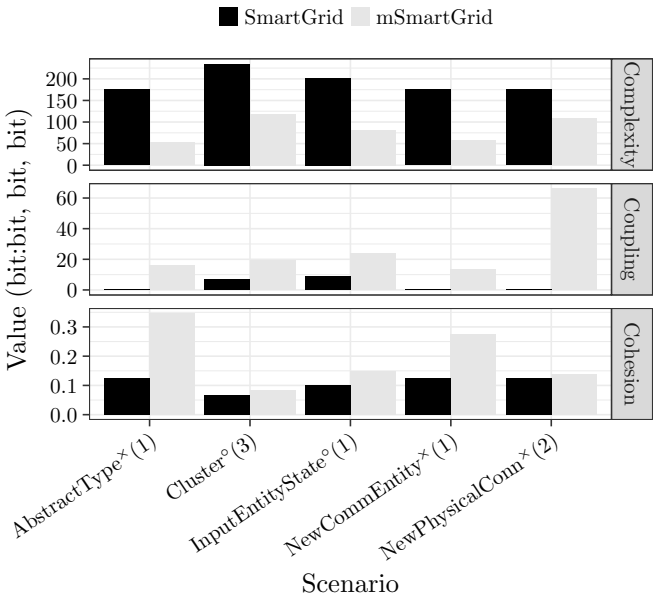

Fig. 9. Evolvability Metric Results: Smart Grid Topology

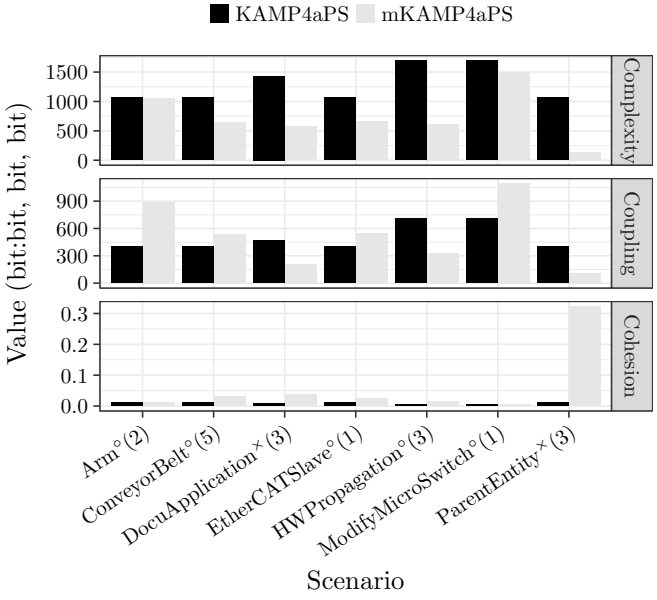

Fig. 10. Evolvability Metric Results: KAMP4aPS

9.3.2 Need-specific Use and Reuse

The results of the metamodel utilization metric are shown in Figure 12. Each case study has its own boxplot. The x-axis shows the name of the metamodel. The left one is the original version and the right one is the modularized version. The y-axis shows the scale for the $mmUtil$ metric. The unit for $mmUtil$ is ratio of classes, i.e. the ratio of instantiated classes compared to the total number of classes from all metamodel modules that have to be loaded. Thus, its value range is between zero and one. A high value is

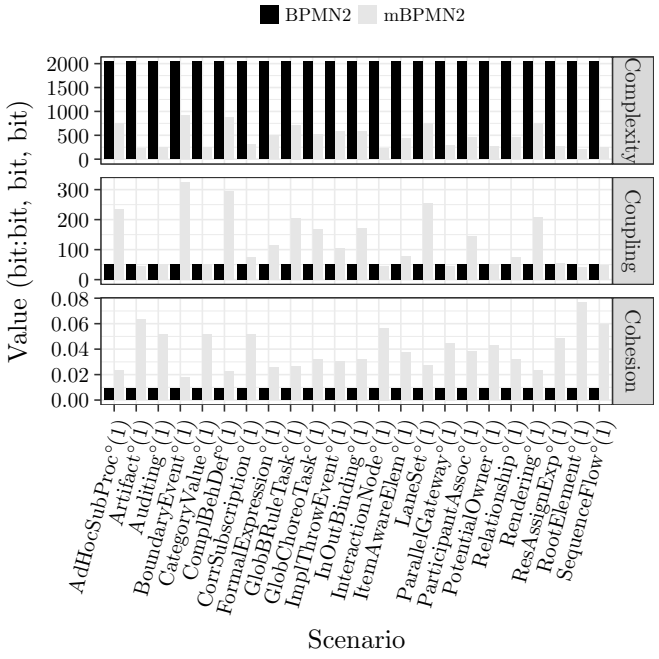

Fig. 11. Evolvability Metric Results: BPMN2

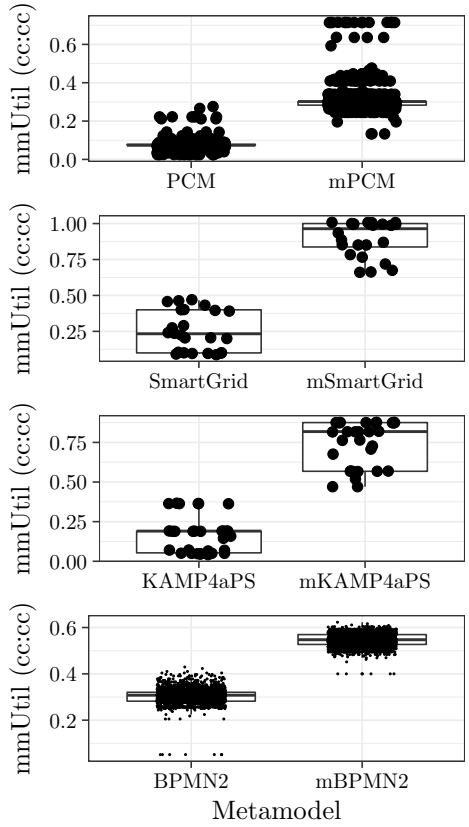

Fig. 12. Metamodel Utilization Results. cc means class count.

good. The lower and upper border of the box represent the first and third quartiles. The bar in the middle of the box shows the median. The whiskers extend from the borders of the box to the last value within 1.5 times the inter quartile range. The individual results are represented as points. We scattered the results to prevent overplotting. Thus, within results for one metamodel version, the x-position has no meaning.

9.4 Results Interpretation and Discussion

In this subsection, we interpret the results presented in the previous subsection. We discuss reasons for differences in the results between the original and modular versions of the metamodels and their implications.

9.4.1 Evolvability

Complexity: Across all case studies and for all evolution scenarios, the complexity of the modular version has decreased in comparison to the complexity of the original version of the metamodels. We attribute this to the constrainment of dependencies (layering, no cycles, conformance to language feature dependencies) and to slicing modules according to user language features. Due to these refactorings, the subgraphs of the modularized metamodels include less unnecessary language features. In many scenarios, the splitting of modules resulted in smaller package size, as before too many user language features were lumped together. The overall complexity of the modularized metamodels was not reduced. In fact, it grew due to additional indirections and class splits. However, the complexity of the parts of the metamodel that are relevant for the metamodel developer, who is working on an evolution scenario, is reduced.

Coupling: The results for the coupling metric are mixed. For the PCM, the coupling decreased in all scenarios. For the other case studies, however, there are scenarios where the coupling increased. For Smart Grid Topology, the coupling increased in four scenarios. For the remaining four scenarios (AbstractType, NewCommEntity, NewPhysicalConn, and SmartMeter), the coupling value for the original metamodel cannot be computed, as the subgraph for these scenarios consists only of one package. In these cases, the coupling cannot be compared to the coupling of the modularized version. The coupling results for KAMP4aPS increased for four scenarios and dropped for three scenarios. For BPMN2, the coupling increased for 15 scenarios, remained equal for one scenario and decreased for seven scenarios.

The mixed results for coupling are caused by different factors. Vertical module splits contribute considerably, as they turn parts of cohesion of modules into coupling. Paradigm extraction also contributes, as abstract classes are extracted and placed in another module. The resulting modules in the Δ layer are thus strongly coupled to their modules in the π layer. In some cases, the extraction of cross-cutting features contributed to the coupling. A cross-cutting feature is a feature that depends on many other features. The metamodel modules of cross-cutting features contain a package structure that mirrors the structure of metamodel modules that are extended [5]. Such structuring helps developers to navigate. These packages are strongly coupled and tend not to contain many classes. Thus, they contribute more to coupling than cohesion.

In the particular case of BPMN2, the coupling of the original metamodel is very low compared to KAMP4aPS and PCM, which have a similar size. This is a result of the main package that contains all language features except for

the ones that are concerned with graphical diagrams. The low coupling between these packages is the only contributor to the overall coupling. We split the main package in the mBPMN2. Thus, a part of the cohesion of this package was transformed into coupling which caused the growth.

As a sidenote, the BPMN2 results of all metrics for the original metamodel are constant over all scenarios. This is the case, as the package of the main metamodel module of the original BPMN2 is very large. As it is dependent on all other metamodel modules, this leads to all metamodel module to always be included in the subgraph.

Due to the dependency constraints of our reference architecture, the effect of high coupling is reasonable. To explain this, two cases of package coupling have to be distinguished: coupling of packages within a metamodel module and coupling between packages of different metamodel modules. In our reference architecture, package hierarchies within metamodel modules are only used for the logical structuring of classes to guide developers. Coupling of packages within a module can be viewed as a sort of cohesion within a module. Especially as the packages within a module are intended to be always used together. Thus, strong coupling of packages within a module does not harm the evolvability and reusability of the metamodel, even if it is bidirectional or contains cycles. One may suspect, that an increase in intra-module package coupling increases complexity and, therefore, damages evolvability. This, however, cannot be observed, as the complexity decreased across all scenarios. An increase in intra-module package coupling accompanied by the complexity remaining constant could also be obscured by a decline in cohesion. This is, however, not the case, as the cohesion increases in all scenarios. Concerning coupling between metamodel modules, the reference architecture forbids dependency cycles. This especially includes bidirectional coupling, which is the smallest form of a dependency cycle. If in the modularized version, a metamodel module (M) is coupled to another metamodel module (N), N can indeed be used without M, but M is always intended to be used together with N. Consequently, we believe strong package coupling is not a problem, if it is either package internal, or unidirectional and has been introduced by intention according to the reference architecture.

Cohesion: The values of the cohesion metric increased across all evolution scenarios of all case studies. We attribute this to the modularization according to user language features. Classes that implement the same feature tend to be related more strongly. Putting these into the same package or removing classes of other features, tends to increase the cohesion. Thus, the increase of cohesion is to be interpreted positively, as this helps developers in identifying classes that belong to user language features.

9.4.2 Need-specific Use and Reuse

For all case studies, the utilization has improved. For the Smart Grid Topology and the KAMP4aPS studies, the best utilization for the original metamodel is less than the worst utilization for the modularized metamodel. Regarding the utilization of PCM and BPMN2, for each individual model the utilization of the modular metamodel is better than the utilization of the original metamodel. We attribute the improvement of utilization to the modularization according to user language features. Models contain instances of specific language features. If the structure of the metamodel supports the use of language features independent of each other, the metamodel utilization increases. This is because EMF requires only relevant metamodel modules to load the model. These positive results show the reference architecture supports need-specific use and reuse.

9.5 Evaluation Conclusion

In summary, the results of the hypergraph analysis show positive results across all scenarios for complexity and cohesion. The results for coupling are mixed. As the reference architecture forbids dependency cycles and bidirectional coupling between modules, the increase of coupling is justifiable. The decrease in complexity helps metamodel developers when they try to understand and navigate the metamodel. The increase in cohesion shows, that packages group classes that are closely related and may evolve together. Thus, the evaluation results for goal 1 show that the reference architecture satisfies **R1** (Improved Evolvability).

Also the evaluation of metamodel utilization exhibits very positive results. The utilization improved for each model that was analyzed. Thus, the evaluation results for goal 2 show that the reference architecture satisfies **R5** (Need-specific Reuse) and **R6** (Need-specific Use).

9.6 Threats to Validity

In case study research, four aspects of validity are distinguished [61] – internal validity, external validity, construct validity, and conclusion validity.

Internal Validity: In the case studies, the metamodels have been refactored according to the reference architecture. The refactored metamodels have been compared to the original metamodels to evaluate the reference architecture. There are several ways of refactoring the original metamodels. So the refactoring may affect the evaluation results. This is why we did not fix bad smells that are not addressed by the reference architecture in the refactorings to preserve the internal validity of the evaluation.

External Validity: According to Runeson et al. [61], in case study research, the representativeness of a sample case may be sacrificed to achieve a deeper understanding and better realism of the phenomena under study. Consequently, the results achieved for the four metamodels in the case studies might not be transferable to arbitrary other cases, due to the individual properties of each case. However, the case studies give important insights and provide indicators for cases with similar properties. To be more specific, the selection of metamodels for the case studies might not be representative enough. Further, our approach might not be applicable or not be beneficial to arbitrary metamodels for quality modeling and analysis. To address this threat, we selected metamodels that are as heterogeneous as possible. See Section 7 for details on metamodel selection.

Construct Validity: Construct validity may be compromised if we merely chose case studies for which our approach works well. In the search for case studies, we encountered metamodels of different degree of modularity. As our goal was to evaluate metamodels as diverse as possible

in the case studies, we chose metamodels of varying degree of modularity. The benefits of our approach decreases, the more modular a metamodel is in its original version and the closer the metamodel modules match the granularity and dependencies of the user language features. This can be observed in the results for KAMP4aPS and Smart Grid Topology. These metamodels were already quite modular. Thus, they show smaller improvement compared to the other case studies. Nevertheless, the results gathered for KAMP4aPS and Smart Grid Topology show clear improvements when applying the reference architecture in comparison to the original metamodels. Consequently, we could show positive evaluation results also for metamodels that already had quite modular structure.

Furthermore, we selected metamodels from different domains – information systems, smart grid, production automation and business process – to ensure the reference architecture is not limited to a specific domain. The evaluation results show the metamodels from all the selected domains benefit from applying the reference architecture.

The selection of evolution scenarios for the case studies is another threat to construct validity. For the case studies, we used different types of scenarios as described in Section 9.2.1 – historical, potential and generic evolution scenarios. Historical evolution scenarios are considered a minor threat as they are derived from change logs and existing extensions to the metamodels. Thus, the metamodel actually faced this evolution in the past. Potential evolution scenarios were derived by reviewing the metamodel and identifying potential modifications and extensions. Generic modifications were derived by randomly choosing a class for modification or extension from packages that did not yet contain an affected class of an evolution scenario. The selection of potential and generic evolution scenarios might threaten the validity. However, from the evaluation results we could not identify different characteristics for potential and generic evolution scenarios in comparison to historical evolution scenarios.

We identified further threats to construct validity for the subgraph extraction and transformation. First, the subgraphs extracted for evaluation may not be an adequate approximation of the part of the metamodel that is relevant for an evolution scenario. Second, the transformation from a metamodel subgraph into a hypergraph may not map metamodel concepts to hypergraph concepts in a way that enables properly measuring the information size of the metamodel. Third, for historical evolution scenarios, there might have been evolution after a historical scenario was executed that would alter the subgraph that is extracted. These are minor threats, as the subgraph extraction and transformation is applied by the same mechanism on both, the original and the modularized metamodel. If the results for one metamodel version are skewed, the results for the other metamodel version are skewed in the same direction. As we do not focus on absolute values but comparing the original and modularized metamodel, this is acceptable.

Conclusion Validity: While analyzing the evaluation results, the effects of interpretation by a specific researcher must be eliminated. Therefore, we apply metrics based on information theory and metamodel utilization in the evaluation, which give reasonable evidence and reduce the need for interpretation. Due to the evaluation design, there is hardly an interpretation that may lead a researcher to another conclusion.

In most cases, the evaluation results depicted in the diagrams are unambiguous. Sometimes, however, the results are close enough, that they cannot be easily distinguished by merely looking at the diagram. So we described the tendencies of all results in Section 9.4. Looking at the raw evaluation data [43] makes obvious that also results close in the diagram can be clearly distinguished. For example, the complexity of the Arm scenario of the KAMP4aPS case study decreased from 1066.092 to 1059.414 bits. The cohesion of this scenario increased from 0.011 to 0.013 bits:bits. The coupling of the CategoryValue scenario of the BPMN2 case study decreased from 52.424 to 51.574 bits.

In the scenario-based evaluation, we extracted parts of the original metamodels and the refactored metamodels to be compared for several evolution scenarios. For each evolution scenario, we compared the parts of the respective metamodels that are relevant for the evolution scenario. We do not evaluate the actual effort of carrying out the evolution scenarios. This is because, in analogy to related work from software engineering [52], we assume the higher complexity and coupling and the lower cohesion of the parts of the metamodel the harder to implement the evolution scenarios. Furthermore, the actual effort for carrying out the evolution scenarios highly depends on various non-structural factors like the individual skills and experience of the metamodel developer maintaining the metamodel.

10 Related Work

Approaches from the language engineering community reuse and compose language fragments to create DSMLs. GEMOC Studio[8] allows for building and composing modeling languages based on generic language components. Puzzle [62] is a tool for refactoring DSMLs by detecting specification clones and extracting reusable language modules. EMF Refactor[9] identifies and refactors design smells based on model metrics. EMF Splitter [63] modularizes monolithic metamodels based on the structural concepts Project, Package and Unit. In contrast, the modularization in our approach is based on language features. The CORE approach [38] proposes concern-oriented reuse to specify flexible software modules and enable model-based software reuse. MontiCore [25] provides modularity concepts for DSMLs by composing existing language fragments to a new language. Melange [27] is an approach to a modular and reusable development of DSMLs by combining and subtyping existing DSML artifacts. There are further techniques that leverage previous experiences in software reuse, such as aspects (e.g., [64]), polymorphic reuse (e.g., [65]), parametric reuse (e.g., [66]), advanced composition operators [26]. Language development using these techniques is difficult because they are hard to combine due to their heterogeneity [26]. Language engineering approaches, however, do not provide any guidance for modularization and composition of the language. They do not take the specifics of a given

[8]http://www.gemoc.org/studio
[9]https://www.eclipse.org/emf-refactor

domain into account. They also do not provide support for quality modeling and analysis. For a distinction of our modularization concepts from existing language engineering approaches, we refer to Section 4.1.

Approaches like that of Strüber et al. [67] apply clustering algorithms for modularizing metamodels and models. In contrast, we transfer concepts from object-oriented design to modularize metamodels.

Jimenez-Pastor et al. [68] combine model fragmentation strategies to split models into more manageable chunks by using model abstraction and visualization mechanisms. Atkinson et al. [69] present an underlying model that captures all concerns into orthogonal dimensions which are accessed through views. The MIC framework [70] proposes different abstraction levels of modeling. Melanie [71] uses multi-level modeling to specify domain-specific and general purpose languages. However, these approaches merely target the abstraction and visualization of large MDE models by providing simpler views of the models. They do not target the modularization of the modeling language.

Work on model typing [72] and model subtyping [73] is concerned with model substitutability. This work aims at automatic model adaptation or reuse of model transformations but does not provide any support for language modularization and composition.

Metamodel/model co-evolution has been addressed in related work (e.g., in [21] and [22]). Cicchetti et al. [21] propose a transformational approach to co-evolution. The approach is based on a difference model to record the evolution of a metamodel and generates a model transformation for the co-evolution of models. Levendovszky et al. [22] describe a DSML for specifying migration rules to perform the model migration automatically. Again, approaches to metamodel/model co-evolution do not provide guidance in language modularization and composition.

There are approaches to dynamic metamodels and trace models (e.g., [74] and [75]). Hegedüs et al. [74] describe a technique for the back-annotation of analysis traces based on change-driven model transformations. Combemale et al. [75] propose an approach to trace analysis results back to the syntax and operational semantics of the original DSML. We have the reflection of analysis results in models in common with these approaches. In contrast, our reference architecture does not aim at tracing analysis results back but serving as a template to specify languages to describe analysis configurations as well as inputs and results for various quality properties.

There are also model-driven approaches to specify quality metrics (e.g., [76] and [77]). Szarnyas et al. [76] identify graph-based model metrics to distinguish real models from auto-generated synthetic ones. Basciani et al. [77] present an approach supporting the definition of custom quality models by a domain specific language to specify the aggregation of quality properties and metrics. In contrast, we do not only want to specify modeling languages for metrics but the entire system including quality metrics and their analysis.

First attempts came up for modular transformations [78] and generator composition [55]. However, these modularization concepts are not applied to the metamodels but the tooling related to the metamodels.

Approaches like JetBrains MPS [79], LISA [80] and Neverlang [81] support extensibility and reuse in general programming languages but do not support languages for quality modeling and analysis.

Configuration and reuse are central to software product lines and ecosystems. While this research is limited to the instance level, our work refers to the metamodel level. Clafer [82] is a metamodeling language with first-class support for feature modeling. Clafer has been designed to express the relation of feature models and metamodels in context of software product line engineering but does not aim to support modeling language modularization and reuse. There are approaches to language product lines (e.g., [83], [84]) that apply product line techniques for developing languages. However, these approaches share the aforementioned limitations of approaches from the language engineering community.

11 Conclusion

In this paper, we investigated the applicability of modularization concepts as known from object-oriented design and reference architectures as known from architectural design to metamodels. We proposed the first reference architecture for metamodels for quality modeling and analysis to avoid recent shortcomings in extension and reuse of metamodels. By leveraging patterns that reoccur in multiple domains, the reference architecture provides a top-level decomposition of metamodels into four layers. These layers comprise fundamental language features (π), domain-specific semantics (Δ), quality properties (Ω) as well as analysis configurations and data (Σ). Inspired by the layered architecture style, the metamodel modules are assigned to specific layers with constrained dependencies. We provided detailed guidelines on the application of the reference architecture for (1) designing a metamodel from scratch and (2) refactoring an existing metamodel. In four case studies, we compared metamodels refactored according to our reference architecture to the original metamodels. Evaluation results show that the reference architecture improves evolvability as well as need-specific use and reuse. Furthermore, based on the case studies, we argue the reference architecture contributes to non-intrusive extension, instance compatibility and independent extension of metamodels.

In the future, we will continue the modularization of existing metamodels and related tooling to expand and sharpen the reference architecture. This includes further investigation of technologies for metamodel extension. The applicability of the reference architecture to grammar-based DSMLs can be investigated. The modularization concepts proposed in this paper are independent of quality modeling and analysis and can be applied to metamodels in general. So, based on the modularization concepts, we will investigate other kinds of metamodels to identify reference architectures for other scopes. We will extend our tool support by new features for visualizing only the outgoing and incoming dependencies of a specific metamodel module to simplify working with large metamodels.

Based on the metamodel modularization concepts proposed in this paper, we will investigate the modularization

and composition of analytical and simulative solvers. Composing modular analysis tools provokes questions from theoretical computer science and formal methods community, for example on behavior preservation and termination of analyses. We will discuss these questions in the Dagstuhl Seminar #19481 in fall 2019. There are a range of ways for composing model-based analysis tools, from isolated analyses with synchronization ex-post, through online co-simulation, to fully integrated analyses [18]. How closely modular analysis tools can be coupled depends on the intensity of necessary coordination and the availability of a common basis. We will investigate a taxonomy of analysis composition techniques, define a language for modeling the interfaces of modular analyses and develop a proof-of-concept coupling of modular analysis tools.

Acknowledgments

This work was supported by the Helmholtz Association of German Research Centers and the MWK (Ministry of Science, Research and the Arts Baden-Württemberg) in the funding line Research Seed Capital (RiSC). We thank Reiner Jung for valuable discussions and support on metrics for metamodel coupling and complexity, Amine Kechaou for implementing the MRS Diagram Editor, Sandro Koch for help with the KAMP4aPS metamodel, Philipp Merkle for generous and very competent R support, Roman Pilipchuk and the Camunda Services GmbH for supplying us with BPMN2 models, and Christian Stier for excellent feedback.

References

[1] ISO/IEC, "ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models," Tech. Rep., 2010.

[2] IEEE Architecture Working Group, "IEEE Std 1471-2000, Recommended practice for architectural description of software-intensive systems," IEEE, Tech. Rep., 2000.

[3] R. H. Reussner, S. Becker, J. Happe, R. Heinrich, A. Koziolek, H. Koziolek, M. Kramer, and K. Krogmann, *Modeling and Simulating Software Architectures – The Palladio Approach*. MIT Press, 2016.

[4] F. Bause, P. Buchholz, and P. Kemper, "Qpn-tool for the specification and analysis of hierarchically combined queueing petri nets," vol. 977, 1995.

[5] M. Strittmatter, G. Hinkel, M. Langhammer, R. Jung, and R. Heinrich, "Challenges in the evolution of metamodels: Smells and anti-patterns of a historically-grown metamodel," in *10th International Workshop on Models and Evolution (ME)*. CEUR Vol-1706, 2016.

[6] F. Brosch, H. Koziolek, B. Buhnova, and R. Reussner, "Architecture-based reliability prediction with the palladio component model," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1319–1339, 2012.

[7] I. Sommerville, *Software Engineering*, 10th ed. Addison-Wesley, 2015.

[8] R. C. Martin, *Designing Object-oriented C++ Applications: Using the Booch Method*. Prentice-Hall, 1995.

[9] R. Heinrich, "Tailored quality modeling and analysis of software-intensive systems," in *30th International Conference on Software Engineering and Knowledge Engineering*. KSI, 2018, pp. 336 – 341.

[10] M. Seidl, M. Scholz, C. Huemer, and G. Kappel, *UML @ Classroom*. Springer, 2015, no. 1.

[11] Object Management Group (OMG), "MOF 2.5.1 Core Specification (formal/2016-11-01)," 2016. [Online]. Available: https://www.omg.org/spec/MOF/2.5.1/

[12] K. Czarnecki and U. W. Eisenecker, *Generative Programming*. Addison-Wesley, Reading, MA, USA, 2000.

[13] P. Roques, "MBSE with the ARCADIA Method and the Capella Tool," in *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, Toulouse, France, 2016.

[14] R. Hahn, "Bad Smells and Anti-Patterns in Metamodeling," Master's thesis, Karlsruhe Institute of Technology (KIT), 2017.

[15] M. Strittmatter and M. Langhammer, "Identifying semantically cohesive modules within the palladio meta-model," in *Symposium on Software Performance*, 2014, pp. 160–176.

[16] C. Rathfelder, *Modelling Event-Based Interactions in Component-Based Architectures for Quantitative System Evaluation*. Karlsruhe, Germany: KIT Scientific Publishing, 2013, vol. 10.

[17] M. Hauck, "Extending Performance-Oriented Resource Modelling in the Palladio Component Model," Diploma Thesis, University of Karlsruhe (TH), Germany, 2009.

[18] R. Heinrich, P. Merkle, J. Henss, and B. Paech, "Integrating business process simulation and information system simulation for performance prediction," *Software & Systems Modeling*, vol. 16, pp. 257–277, 2017.

[19] J. Kroß, A. Brunnert, and H. Krcmar, "Modeling big data systems by extending the palladio component model," *Softwaretechnik-Trends*, vol. 35, no. 3, 2015.

[20] F. Willnecker, A. Brunnert, and H. Krcmar, "Predicting energy consumption by extending the palladio component model," in *Symposium on Software Performance*, 2014, p. 177.

[21] A. Cicchetti, D. D. Ruscio, R. Eramo, and A. Pierantonio, "Automating co-evolution in model-driven engineering," in *12th International IEEE Enterprise Distributed Object Computing Conference*. IEEE, 2008, pp. 222–231.

[22] T. Levendovszky, D. Balasubramanian, A. Narayanan, F. Shi, C. van Buskirk, and G. Karsai, "A semi-formal description of migrating domain-specific models with evolving domains," *Software & Systems Modeling*, vol. 13, no. 2, pp. 807–823, 2014.

[23] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2003.

[24] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture – A System of Patterns*. Wiley, 1996.

[25] K. Hölldobler and B. Rumpe, *MontiCore 5 Language Workbench Edition 2017*, ser. Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag, 2017.

[26] B. Combemale, J. Kienzle, G. Mussbacher, O. Barais, E. Bousse, W. Cazzola, P. Collet, T. Degueule, R. Heinrich, J.-M. Jézéquel, M. Leduc, T. Mayerhofer, S. Mosser, M. Schöttle, M. Strittmatter, and A. Wortmann, "Concern-oriented language development (cold): Fostering reuse in language engineering," *Computer Languages, Systems & Structures*, 2018.

[27] T. Degueule, B. Combemale, A. Blouin, O. Barais, and J.-M. Jézéquel, "Melange: A Meta-language for Modular and Reusable Development of DSLs," in *8th International Conference on Software Language Engineering (SLE)*, 2015.

[28] Object Management Group, "UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, version 1.1," 2011.

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

[30] S. Kounev, F. Brosig, and N. Huber, "The Descartes Modeling Language," Department of Computer Science, University of Wuerzburg, Tech. Rep., 2014.

[31] R. Drath, A. Luder, J. Peschke, and L. Hundt, "Automationml - the glue for seamless automation engineering," in *2008 IEEE International Conference on Emerging Technologies and Factory Automation*, 2008, pp. 616–623.

[32] J. Gelissen and R. M. Laverty, "Robocop: Revised specification of framework and models (deliverable1.5)," Information Technology for European Advancement, Tech. Rep., 2003.

[33] Object Management Group (OMG), "Business Process Model And Notation Specification (BPMN) – Version 2.0.2," 2014.

[34] A. Kechaou and M. Strittmatter, "Modularizing and layering metamodels with the modular emf designer," in *Companion Proceedings of the 21st International Conference on Model Driven Engineering Languages and Systems*, 2018, pp. 32–36.

[35] Object Management Group, "Unified Modeling Language (UML) – Version 2.5," 2015.

[36] P. Langer, K. Wieland, M. Wimmer, and J. Cabot, "Emf profiles: A lightweight extension approach for emf models," *Journal of Object Technology*, vol. 11, no. 1, pp. 8:1–29, 2012.

[37] R. Jung, R. Heinrich, E. Schmieders, M. Strittmatter, and W. Hasselbring, "A method for aspect-oriented meta-model evolution," in *2nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*. ACM, 2014, pp. 19:19–19:22.

[38] O. Alam, J. Kienzle, and G. Mussbacher, "Concern-oriented software design," in *16th Intern. Conference on Model-Driven Engineering Languages and Systems*, vol. 8107. Springer, 2013, pp. 604–621.

[39] M. Schöttle, N. Thimmegowda, O. Alam, J. Kienzle, and G. Mussbacher, "Feature modelling and traceability for concern-driven software development with TouchCORE," in *Companion Proceedings of 14th International Conference on Modularity*, 2015, pp. 11–14.

[40] T. Goldschmidt, S. Becker, and E. Burger, "Towards a tool-oriented taxonomy of view-based modelling," in *Modellierung 2012*, vol. P-201. GI, 2012, pp. 59–74.

[41] W. Raskob, V. Bertsch, M. Ruppert, M. Strittmatter, L. Happe, B. Broadnax, S. Wandler, and E. Deines, "Security of electricity supply in 2030," in *Critical Infrastructure Protection and Resilience Europe (CIPRE)*, 2015.

[42] R. Heinrich, S. Koch, S. Cha, K. Busch, R. Reussner, and B. Vogel-Heuser, "Architecture-based change impact analysis in cross-disciplinary automated production systems," *Journal of Systems and Software*, vol. 146, pp. 167 – 185, 2018.

[43] M. Strittmatter, R. Heinrich, and R. Reussner, "Supplementary material for the evaluation of the layered reference architecture for metamodels to tailor quality modeling and analysis," KIT, Tech. Rep. 2018,11; Karlsruhe Reports in Informatics, 2018.

[44] S. Fürst, J. Mössinger, S. Bunzel, T. Weber, F. Kirschke-Biller, P. Heitkämper, G. Kinkelin, K. Nishikawa, and K. Lange, "Autosar–a worldwide standard is on the road," in *14th Intern. VDI Congress Electronic Systems for Vehicles*, vol. 62, 2009.

[45] R. Heinrich, *Aligning Business Processes and Information Systems: New Approaches to Continuous Quality Engineering*. Springer, 2014.

[46] H. P. Breivold, I. Crnkovic, and P. J. Eriksson, "Analyzing software evolvability," in *32nd Annual IEEE International Computer Software and Applications Conference*, 2008, pp. 327–330.

[47] L. Iovino, A. Pierantonio, and I. Malavolta, "On the impact significance of metamodel evolution in mde," *Journal of Object Technology*, vol. 11, pp. 3–1, 2012.

[48] G. Hinkel and M. Strittmatter, "On Using Sarkar Metrics to Evaluate the Modularity of Metamodels," in *5th International Conference on Model-Driven Engineering and Software Development*, 2017.

[49] G. Hinkel, M. Kramer, E. Burger, M. Strittmatter, and L. Happe, "An Empirical Study on the Perception of Metamodel Quality," in *4th International Conference on Model-Driven Engineering and Software Development*, 2016, pp. 145–152.

[50] J. A. Cruz-Lemus, A. Maes, M. Genero, G. Poels, and M. Piattini, "The impact of structural complexity on the understandability of uml statechart diagrams," *Information Sciences*, vol. 180, no. 11, pp. 2209–2220, 2010.

[51] L. C. Briand, J. Wüst, and H. Lounis, "Replicated case studies for investigating quality factors in object-oriented designs," *Empirical Software Engineering*, vol. 6, no. 1, pp. 11–58, 2001.

[52] E. B. Allen, S. Gottipati, and R. Govindarajan, "Measuring size, complexity, and coupling of hypergraph abstractions of software: An information-theory approach," *Software Quality Journal*, vol. 15, no. 2, pp. 179–212, 2007.

[53] L. C. Briand, S. Morasca, and V. R. Basili, "Property-based software engineering measurement," *IEEE Transactions on Software Engineering*, vol. 22, no. 1, pp. 68–86, 1996.

[54] E. B. Allen, "Measuring graph abstractions of software: an information-theory approach," in *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on*, 2002, pp. 182–193.

[55] R. Jung, R. Heinrich, and W. Hasselbring, "GECO: A generator composition approach for aspect-oriented dsls," in *9th International Conference on Model Transformation*. Springer, 2016, pp. 141–156.

[56] R. Jung, "Generator-composition for aspect-oriented domain-specific languages," Doctoral thesis, Kiel University, 2016.

[57] D. Schütt, "On a hypergraph oriented measure for applied computer science," in *COMPCON Fall '77*, 1977, pp. 295–296.

[58] K. Rostami, J. Stammel, R. Heinrich, and R. Reussner, "Architecture-based assessment and planning of change requests," in *11th International Conference on Quality of Software Architectures*. ACM, 2015, pp. 21–30.

[59] R. Pilipchuk, S. Seifermann, and R. Heinrich, "Aligning business process access control policies with enterprise architecture," in *2018 Central European Cybersecurity Conference*. ACM, 2018.

[60] R. Pilipchuk, "Coping with access control requirements in the context of mutual dependencies between business and it," in *2018 Central European Cybersecurity Conference*. ACM, 2018.

[61] P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case Study Research in Software Engineering: Guidelines and Examples*. Wiley, 2012.

[62] D. Méndez-Acuña, J. A. Galindo, B. Combemale, A. Blouin, and B. Baudry, "Puzzle: A tool for analyzing and extracting specification clones in dsls," in *15th International Conference on Software Reuse: Bridging with Social-Awareness*. Springer, 2016, pp. 393–396.

[63] A. Garmendia, E. Guerra, D. S. Kolovos, and J. de Lara, "EMF splitter: A structured approach to EMF modularity," in *3rd Workshop on Extreme Modeling*, 2014, pp. 22–31.

[64] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel, *Weaving Executability into Object-Oriented Meta-languages*. Springer, 2005, pp. 264–278.

[65] T. Degueule, B. Combemale, A. Blouin, O. Barais, and J.-M. Jzquel, "Safe model polymorphism for flexible modeling," *Comput. Lang. Syst. Struct.*, vol. 49, no. C, pp. 176–195, 2017.

[66] J. de Lara and E. Guerra, *Generic Meta-modelling with Concepts, Templates and Mixin Layers*. Springer, 2010, pp. 16–30.

[67] D. Strüber, J. Rubin, G. Taentzer, and M. Chechik, "Splitting models using information retrieval and model crawling techniques," in *Fundamental Approaches to Software Engineering*. Springer, 2014, pp. 47–62.

[68] A. Jimnez-Pastor, A. Garmendia, and J. de Lara, "Scalable model exploration for model-driven engineering," *J. Syst. Softw.*, vol. 132, no. C, pp. 204–225, 2017.

[69] C. Atkinson, D. Stoll, and P. Bostan, "Orthographic Software Modeling: A Practical Approach to View-Based Development," in *Evaluation of Novel Approaches to Software Engineering*. Springer, 2010, vol. 69, pp. 206–219.

[70] A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai, "Composing domain-specific design environments," *Computer*, vol. 34, no. 11, pp. 44–51, 2001.

[71] C. Atkinson and R. Gerbig, "Melanie: Multi-level modeling and ontology engineering environment," in *2nd Intern. Master Class on Model-Driven Engineering: Modeling Wizards*. ACM, 2012.

[72] J. Steel and J.-M. Jézéquel, "On model typing," *Software & Systems Modeling*, vol. 6, no. 4, pp. 401–413, 2007.

[73] C. Guy, B. Combemale, S. Derrien, J. Steel, and J.-M. Jézéquel, "On model subtyping," in *Modelling Foundations and Applications*. Springer, 2012, vol. 7349, pp. 400–415.

[74] A. Hegedus, G. Bergmann, I. Rath, and D. Varro, "Back-annotation of simulation traces with change-driven model transformations," in *2010 8th IEEE International Conference on Software Engineering and Formal Methods*, 2010, pp. 145–155.

[75] B. Combemale, L. Gonnord, and V. Rusu, "A generic tool for tracing executions back to a dsml's operational semantics," in *7th European Conference on Modelling Foundations and Applications*. Springer, 2011, pp. 35–51.

[76] G. Szárnyas, Z. Kővári, A. Salánki, and D. Varró, "Towards the characterization of realistic models: Evaluation of multidisciplinary graph metrics," in *19th International Conference on Model Driven Engineering Languages and Systems*. ACM, 2016, pp. 87–94.

[77] F. Basciani, J. d. Rocco, D. d. Ruscio, L. Iovino, and A. Pierantonio, "A customizable approach for the automated quality assessment of modelling artifacts," in *10th Intern. Conference on the Quality of Information and Communications Technology*, 2016, pp. 88–93.

[78] A. Rentschler, "Model Transformation Languages with Modular Information Hiding," Ph.D. dissertation, Karlsruhe Institute of Technology, Karlsruhe, Germany, 2015.

[79] M. Voelter and V. Pech, "Language modularity with the mps language workbench," in *34th International Conference on Software Engineering*. IEEE, 2012, pp. 1449–1450.

[80] M. Mernik, M. Lenič, E. Avdičaušević, and V. Žumer, "Lisa: An interactive environment for programming language development," in *Compiler Construction*. Springer, 2002, pp. 1–4.

[81] E. Vacchi and W. Cazzola, "Neverlang: A framework for feature-oriented language development," *Computer Languages, Systems & Structures*, vol. 43, pp. 1 – 40, 2015.

[82] K. Bąk, K. Czarnecki, and A. Wąsowski, "Feature and meta-models in clafer: Mixed, specialized, and coupled," in *Software Language Engineering*. Springer, 2011, pp. 102–122.

[83] T. Kühn, W. Cazzola, and D. M. Olivares, "Choosy and picky: Configuration of language product lines," in *19th International Conference on Software Product Line*. ACM, 2015, pp. 71–80.

[84] D. Méndez-Acuña, J. A. Galindo, T. Degueule, B. Combemale, and B. Baudry, "Leveraging software product lines engineering in the development of external dsls: A systematic literature review," *Computer Languages, Systems & Structures*, vol. 46, pp. 206–235, 2016.

Robert Heinrich Robert Heinrich is head of the Quality-driven System Evolution research group at Karlsruhe Institute of Technology (Germany). He holds a doctoral degree from Heidelberg University and a degree in Computer Science from University of Applied Sciences Kaiserslautern. His research interests include quality modeling and analysis across several domains, such as information systems, business processes and automated production systems. One core asset of his work is the Palladio software architecture simulator. He is involved in the organization committees of several international conferences, is reviewer for international premium journals, like IEEE Transactions on Software Engineering and IEEE Software, and is reviewer for international academic funding agencies. Robert is principal investigator or chief coordinator in several grants from German governmental funding agencies. He has (co-)authored more than 50 peer-reviewed publications and spent research visits in Chongqing (China) and Tel Aviv (Israel).

Misha Strittmatter studied Computer Science at the Karlsruhe Institute of Technology (KIT). Since 2013, he is a researcher at the Software Design and Quality (SDQ) Group of the KIT. Currently, he works on his doctoral thesis about modular metamodeling, metamodel extension, and reuse. His further research interests are software architectures, software performance, and smart grid resilience.

Ralf Reussner Ralf Reussner holds the Chair for Software-Design and -Quality at Karlsruhe Institute of Technology since 2006 and heads the Institute of Program Structures and Data Organisation. His research group works in the interplay of software architecture and predictable software quality as well as on view-based design methods for software-intensive technical systems. Ralf Reussner published over 150 peer-reviewed papers in Journals and Conferences, but also established and organised various conferences and workshops, including QoSA and WCOP. In addition, he acts as a PC member or reviewer of several conferences and journals, including IEEE Transactions on Software Engineering, IEEE Software and IEEE Computer. He founded the software architecture section of the German Informatics Society in 2006 and is speaker of its software engineering division since 2017. As scientific director of the FZI - Research Center for Information Technologies he consults various industrial partners in the areas of component based software, architectures and software quality. He is principal investigator or chief coordinator in several grants from industrial and governmental funding agencies. Ralf received offers on full professorships from University of Osnabrück, University of Hamburg and Technical University of Munich, which he all rejected. From 2003 till 2006 he held the Juniorprofessorship for Software Engineering at the University of Oldenburg, Germany, and was awarded with a grant of the Emmy-Noether young investigators excellence programme of the National German Science Foundation (DFG).