A Modular Reference Structure for Component-based Architecture Description Languages

Misha Strittmatter, Kiana Rostami, Robert Heinrich and Ralf Reussner
Chair for Software Design and Quality (SDQ)
Karlsruhe Institute of Technology
Karlsruhe, Germany
{strittmatter | rostami | heinrich | reussner}@kit.edu

***Abstract*—Metamodels are used to define languages, code generation and they serve as data structures for metamodel-centric software systems. In software engineering, these metamodels are crafted, evolved and extended, e.g., by further quality dimensions or structural features. However, an ad-hoc modeling approach does not properly support metamodel reuse by extension or composition. Nor does it enforce a proper modularization which helps with tackling complexity. We present an approach to design and extend metamodels for component-based architecture description languages in a modular way. The information which is to be metamodeled is divided into paradigm, domain, quality and analysis content. We constrain the usage of dependencies and give instructions how to modularize in accordance to concerns. Related approaches try to modularize and compose transformations, generators, and tools in general. However, in the field of metamodels, little support is given. Our approach is applied to several concerns of the Palladio Component Model and an extension thereof.**

I. Introduction

In model-based software engineering (e.g., model-driven software development or software performance engineering) and in general in many fields of computer science, software is described using models. These models capture different aspects like the object-oriented design, more coarse-grained architecture, deployment and so forth. Each discipline has its own focus and may add more information to this foundation. E.g., design decisions, implementation documentation, requirements and quality related information like service level agreements for performance or security.

A *metamodel* is a model which defines the structure of other models. If a model conforms to a metamodel, the model is considered an *instance* of the metamodel. Thus, metamodels are similar to grammars, as they define languages. A prominent example is the Unified Modeling Language (UML) metamodel [12]. A sequence diagram (instance) is a model which is an instance of the UML metamodel. Our approach primarily targets MOF [13] (i.e., the meta-metamodel of UML) conforming metamodels. However, we expect that it also directly applicable to metamodels conforming to meta-metamodels which feature similar concepts to: classes, containment-, inheritance- and association relations between classes.

This work was supported by the Helmholtz Association of German Research Centers and the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future – Managed Software Evolution.

There are multiple ways to found metamodel-based languages: 1) a new metamodel is developed. 2) an existing metamodel (e.g., UML) is extended by annotations or stereotyping. 3) a variant or branch of an existing language is created. The development of new metamodels is straightforward, if they do not evolve and are isolated. However, this is seldom the case. The major problem is that growing metamodels structurally degrade over time. Extensions by annotation or stereotyping are problematic as they may result in a flat, unstructured organization of information. Branches and variants are problematic, because duplicated parts have to be maintained when the original language evolves.

An example of this is the Palladio Component Model (PCM) [1]. It is a metamodel-based language which was initially developed for the specification of component-based software architectures and their resource demands to be able to predict their performance. With time, the research focus broadened and more structural features and quality dimensions were incorporated. Some of this information was directly built into the language [2]. Other aspects were specified as extensions or wound up in branches (e.g., the integration of business processes modeling and analysis [3] as well as modeling and analysis of maintainability [4]). This impedes the structure and reuse potential of inner parts of the PCM.

There are approaches, which put forward modular, composable or extensible concepts which are more or less related to metamodels. These concepts include components [5], [6], classes of object-oriented design [7], domain specific and general purpose languages [8], transformations [9], generators [10] and simulators [11]. However, for metamodels, little support is given.

Our approach aims to tackle these problems with a reference structure for metamodels for architecture description languages. The reference structure proposes a modularization of information into layers for paradigm, domain, quality and analysis information. The layers can further be divided: e.g., for separate quality dimensions or different domains. This leads to a modular, flexible and extensible structure, which

satisfies separation of concerns and thus is better understandable and maintainable. It also increases the potential for reuse as a basis for new extensions. In addition, modularity leads to localization of change impact in the case of metamodel evolution. This does not only apply to the metamodel but to everything which is dependent on the metamodel (e.g., editors, analyzers, generators). The applicability of our reference structure is demonstrated on an selection of basic concerns and extensions of the PCM.

Our approach aims at the structuring of metamodels for the description of component-based software architecture and their qualities. However, we expect that it can also be applied to an even broader spectrum of metamodels, where the proposed decomposition is meaningful. These may be architecture description languages (ADLs) or even description languages of software-intensive systems in general.

This paper is outlined as follows: Section II describes the example scenario. Section III presents the reference structure and its concepts. Section IV applies the reference structure onto the example scenario. Section V presents related work. Section VI concludes the paper.

II. Problem Scenario

The scenario for our motivating example is the PCM. The PCM (i.e., a metamodel) is used for several analyses and simulations (see Figure 1). At the core of the PCM is a well formed construct to specify components, interfaces, and their composition. In the past, this part of the metamodel has served as a basis for new metamodel content which was emerging from new research. Initially this content was added directly to the PCM [14]. Examples of such intrusive extensions are reliability [15], event communication [16] and infrastructure components [17]. Later, as their number and diversity grew, extensions were no longer included directly in the PCM. They either came in the form of branches of the metamodel or metamodels which referenced into the PCM. Examples of such extensions are KAMP (Karlsruhe Architectural Maintainability Prediction) [4] and support for modeling business processes that interface with system services [3].

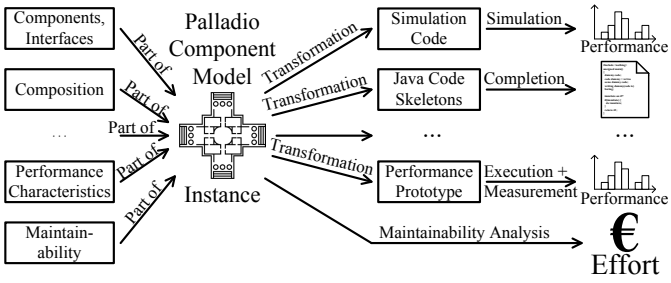

Fig. 1. Excerpt of Concerns and Capabilities of the PCM

Branches as well as intrusive extensions are problematic and have negative implication on users, developers, and researchers. Developers and researchers should have a clean base upon which they can build their extensions. If the metamodel is not modularized, it often contains content, which is irrelevant to the extension being made. This unnecessary complexity leads to a decline in understandability and may even lead to metamodeling mistakes. Further, both extension approaches are adverse to the maintainability of the metamodel. Intrusive extensions increase the complexity and external extensions often lack extension points in the base metamodel. Extension over time, regardless of intrusive or non-intrusive, increases either complexity or the amount of software artifacts that are dependent on the core metamodel. Thus, if the base metamodel is in need of restructuring but the restructuring is postponed, the cost of the refactoring will increase. This is especially critical in metamodel centric applications and can be called metamodel debt (cf. technical debt).

Negative effects of unstructured extensions on users are twofold. Users have specific needs with regards to the features of the program. They may be confused or overwhelmed when confronted with too much content irrelevant to them. Further, to support the additional metamodel content, the code of all extensions has to be shipped. For these issues, there are technical workarounds to minimize their negative impact. Shipped code of extensions may be reduced to a minimum by only including the model code. Tools and editors may be configurable to hide content which is not wanted by the user. However, this is only a workaround, as each extension needs to intrusively modify the tool in question.

For the means of the running example, the relevant concepts of the PCM will be explained in a condensed way (illustrated in Figure 2). For a complete and detailed description of the PCM, please consult the technical report [18]. At the core of the PCM are components and interfaces (I). Both are first class entities and a components may implement or require interfaces (II). Components can be placed within an architecture by creating an assembly context (III). The interfaces of these assembly contexts can be linked by connectors (IV). A component may be either atomic or composite (V). Composite components contain further assembly contexts and connectors. For the modeling of performance and reliability, atomic components contain abstractions of control flow (VI) (similar to activity diagrams or flow charts). These control flow specifications carry resource demands and failure probabilities in some of their states. To conduct performance simulation [19] of such a model, there is further information needed like the maximum simulation time or the maximum measure count.

Another part of the running example is KAMP [4]. It is an extension for the PCM that deals with maintainability. It is a tool-supported approach to semi-automatically predict change propagation in a software system. In contrast to other approaches, KAMP considers not only the architecture of a software system, but also organizational aspects (e.g., management or various roles such as tester or deployer) as well as technical artifacts (e.g., testing, build configurations, or deployment). KAMP comprises two phases: 1) *Preparation Phase*: After the user models a software architecture using the PCM (e.g., Figure 2), she or he enriches the model with context annotations such as test cases and build configuration. 2) *Change Request Analysis Phase*: After the user has modeled

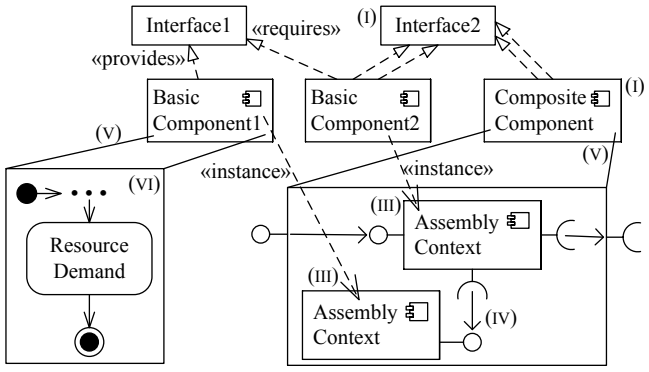

Fig. 2. Simplified Concepts of the PCM

the change in the enriched architecture model, KAMP semi-automatically calculates the change propagation and derives a task list, which is composed of tasks needed to implement the change request. This step calculates all affected artifacts annotated in the model, too.

The overall scenario of the running example is illustrated in Figure 3 and contains several main concerns: the PCM is a language to specify component-based software architecture as the common basis; performance characteristics, which are included in the PCM and partly in external sources (configurations of analysis launches); maintainability information, which is contained within an external extension. The PCM is internally structured. The primary decomposition runs along its view types (submodels). However, here it is displayed as one module with the exception of the performance results, which are stored in a modular way. This stems from the fact that due to dependency cycles within the PCM [2], it is only possible to use it as a whole. Thus, all the problems from above follow for developers and users: unnecessary complexity and its implications.

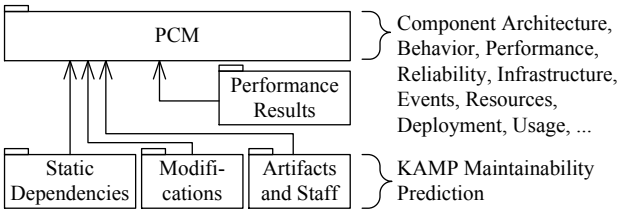

Fig. 3. Status Quo of the Scenario: Metamodel Modules

These problems are tackled by our approach. By modularizing the meta content and structuring it according to our reference structure, the concerns are separated and the extensions have a clean base. In the following sections we will present the layers of the reference structure, then build an ideal and modular version of the scenario meta model according to the reference structure.

III. Reference Structure Overview

Within our reference structure, the metamodel is subdivided into smaller parts which we call metamodel modules or just *modules*. On the technical level, these modules may manifest in their own metamodels, each of which is persisted in its own file, or they may just manifest within the package structure or comparable subdivisions within one metamodel. The contents (e.g., classes) of one module may depend on the contents of another module. This is either in the form of simple reference, containment, inheritance, or stereotype application. If at least one dependency from one module to another exists, we will regard that as the one module being *dependent* on the other.

Our reference structure organizes modules into *layers* as the topmost unit of decomposition. A layer is a set of modules. The layers are ordered with regard to the dependencies of their modules. The modules of a layer may only depend on the modules of the same or more basic layers. More basic here means, that they depend on fewer other layers. Within the scope of this paper, basic layers will be illustrated at the top. If a module of a layer depends on a module of another layer, we will regard that as the one layer being dependent on the other one. Layers may depend on all higher layers. However, it is advisable to confine the dependencies on the next higher level where possible. Circular dependencies between modules (as well as between layers) are not allowed. A circular dependency is either a result of a dependency which should be reversed, or of a strong cohesion between the modules. These modules, thus, should be considered for merging or restructuring.

A module encapsulates a set of concerns. When modularizing, two main rules should be applied. It should be meaningful to use the basic module (or modules) with and without the concerns which have been factored out. If this is not the case, the modularization is still meaningful if the basic module serves as a common foundation for multiple further modules.

The information which is formalized in metamodels can be grouped into categories depending on the type of information. Multiple decomposition dimensions exist and it not always clear which decompositions to apply and in which order. Intuitive design might modularize information in ways of infrastructure (abstract class hierarchies) vs. concrete content, views types or sub models, or semantic cohesion. With our reference structure, we propose as the primary decomposition a layering into: paradigm, domain, quality and analysis content. *Paradigm* (π) is the most basic layer. It lays the the foundation of the language by providing structure but without semantics. The *domain* (Δ) layer builds upon the paradigm and assigns semantics to its abstract structure. The *quality* (Ω) layer adds quality properties to the domain concepts. Lastly, the *analysis* (Σ) layer provides modules for input-, output- and internal state and configuration options for analyses. We settled for these layers, because they provided an intuitive primary decomposition in our research when inspecting metamodels for component-based architecture description languages and their intrusive and external extensions. The concern constellations of these layers (whereas not as explicit as proposed by our reference structure) can be found in metamodels like UML MARTE, the Descartes Metamodel and the PCM.

IV. Applying the Reference Structure

In the following subsections, we will explain the layers of the reference structure and gradually construct a modular

metamodel of our example (PCM extended by maintainability) according to these layers. Please keep in mind, that this is a simplification and a reconstruction of the PCM. Thus, modeling of concepts and dependency directions partly do not adhere to the current PCM.

A. Paradigm

The most basic layer is the paradigm layer (π). It defines foundational structure but without semantics. Thus, it is not directly usable. Not even for purposes which do not need dynamic semantics (e.g., documentation and communication). The minimal configuration of the metamodel which is meaningfully instantiable is π and the domain layer put together. First class entities in π should be abstract and no top level container (root) should be provided to avoid instantiation. Modules which cannot be instantiated directly, will be considered abstract modules. In our case π constitutes components, interfaces and their composition. However, any π is possible. It is dependent on the subject matter which is to be captured. E.g., object oriented design and behavioral formalisms.

In our example, π encompasses the modules CoreEntities and Composition. This is illustrated in Figure 4. Please keep in mind that for the sake of presentation this is an extremely simplified depiction. Wherein Composition is dependent on CoreEntities. CoreEntities contains the abstract metaclasses Component and Interface. Like in the standard UML notation, abstract classes and abstract modules are indicated by a name in italic letters. A Component may reference various Interfaces in two ways: provide and require. The Composition module defines ComposedStructures which contain AssemblyContexts and Connectors. Connectors link Interfaces of two AssemblyContexts. An AssemblyContext represents an instance of a Component.

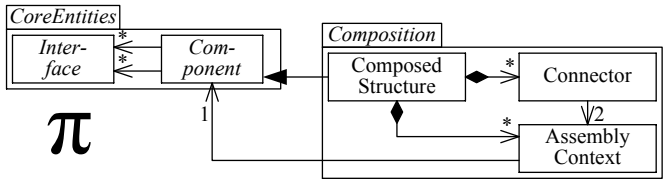

Fig. 4. The π (Paradigm) Layer

The ComposedStructure *extends* Component. In the illustration, the UML notation for stereotype application (a filled arrow) is used. However, the implementation of this relation does not necessarily have to be stereotyping. A wide range of different extension mechanisms is available with all their pros and cons. Further ones are annotation, plain inheritance, and the application of patterns like the decorator pattern or aspect-oriented extension [20].

The π layer builds the foundation of the remainder of the metamodel. It is important, that is has no outgoing dependencies into other layers. This way, it can be reused on its own.

B. Domain

The domain layer (Δ) extends π and assigns domain semantics to its abstract first class entities. By doing so, new information (i.e, attributes and relations) can be added to the derived classes. New domain concepts may be created as well. However, if these new domain concepts have an overlap with classes of other modules in Δ, or even of π it should be considered to factor that content out into a higher, more general module or even a higher layer (i.e., in this case π). In the scope of this paper, Δ will capture software systems. In general, any Δ layer is possible. E.g., embedded and mechatronic systems or cyber-physical systems. The Δ, however, has to fit the underlying π layer.

The Δ layer excerpt of the example scenario is illustrated in Figure 5. It features the ComponentRepository module, where the modeler can specify the Components and Interfaces of an architecture. Please keep in mind that the relation between Components and Interfaces is already defined in the π layer. The ComponentRepository module assigns domain semantics to the abstract CoreEntities module of the π layer providing concrete subclasses for its abstract classes. This separates the concerns of the structure of the paradigm (here component architecture) from its domain semantics, enabling the reuse of the paradigm structure. The Behavior module extends SoftwareComponents by a behavior abstraction, which is similar to a flowchart. The Behavior module is performance agnostic and is later used as a foundation for performance and reliability modeling. The Modification module is essential to the maintainability analysis. It defines Modifications of the architecture. Each concrete Modification references an element of the architecture. It further defines Propagations, which express how Modifications spread through an architecture.

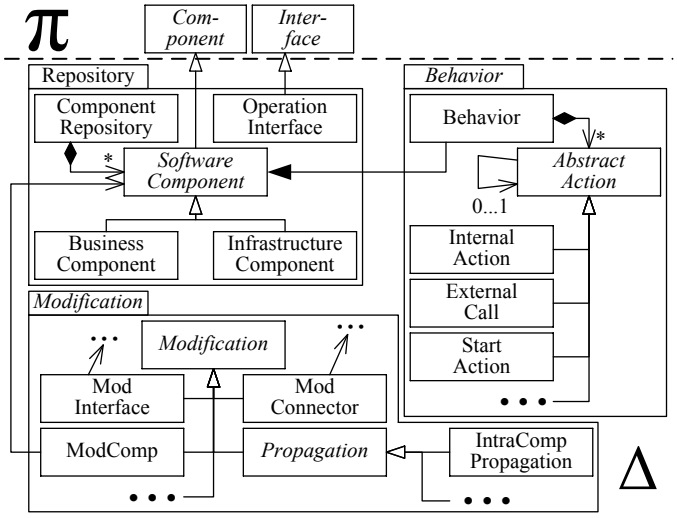

Fig. 5. Excerpt of the Δ (Domain) Layer

C. Quality

The quality layer (Ω) defines the inherent quality properties of Δ concepts. It contains primarily second class entities, which enrich first class entities of Δ. In the scope of this paper, the Ω layer constitutes performance and reliability characteristics. Possible other Ω modules which fit the π and Δ are: security and availability. Not in every circumstance a Ω layer is needed. E.g., when the metamodel is only needed

for specification of software design or for static analysis. This is the case for the KAMP maintainability analysis, which does not require information in the Ω layer.

The quality properties have to be inherent and not derived. Inherent with respect to the analyses which can be performed on the model. E.g., when considering the PCM and performance, the response time of an operation depends on many things, like resource demands within the operation, response time of external calls and so on. Thus, the response time of an operation in this consideration is not an intrinsic value, but a derived one. Therefore, the response time of operations should be moved to a lower layer. There may be quality concepts which are needed to model derived properties. These should be specified in the Ω layer. However, their instances should be contained in the Σ layer. On the other side, the resource demand (not in terms of time but in load) is independent. It may be derived from an estimate or a real word software artifact, but with regards to the analyses it cannot be derived. Therefore, it belongs in Ω and should be contained and specified there. If a model is used for solving, analysis or simulation, the model content from Ω should remain static during execution. If that is not the case, the information belongs to the analysis layer, as it is derived or state information.

Figure 6 shows an excerpt of the exemplary Ω layer. It contains the modules for Performance and Reliability. They extend InternalAction, and therefore the internal behavior of a Component, by ResourceDemand and FailureOccurence. A FailureOccurence has a probability, as well as a type. Here, an important advantage of the modularization shows. The behavior metamodule is free of performance and reliability data. As required, it can be extended by one of the two, or even both. Tools and developers are not bothered by irrelevant content. Potential future extensions can be created without the need for prearrangement.

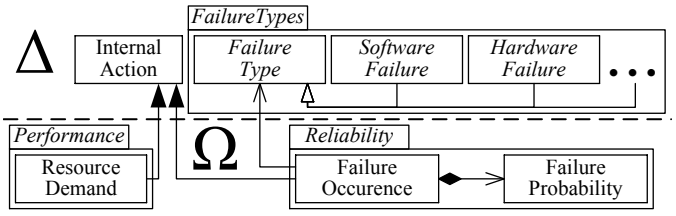

Fig. 6. Excerpt of the Ω (Quality) Layer

D. Analysis

The analysis layer (Σ) is only relevant, if models are used as a basis for analysis, solving or simulation (hereafter only referred to as analysis). Σ provides new views to specify input state, configuration, run-time state and output of an analysis. These are all possible views, but only a subset may be required by an analysis. It is possible for multiple analyses to be founded on the upper layers. Several analyses may share modules, but also posses their own ones. In the focus of this paper, the Σ layer is concerned with the performance simulation and the change impact analysis KAMP.

In Figure 7 an excerpt of the Σ layer is shown. As a really simplified version of the performance result, here, a set of OperationResponseTimes is modeled. An OperationResponseTime has a unit, which is specified in the Ω layer (the corresponding module was not shown in Figure 6). It further references the AssemblyContext and the Interface where the response time was recorded.

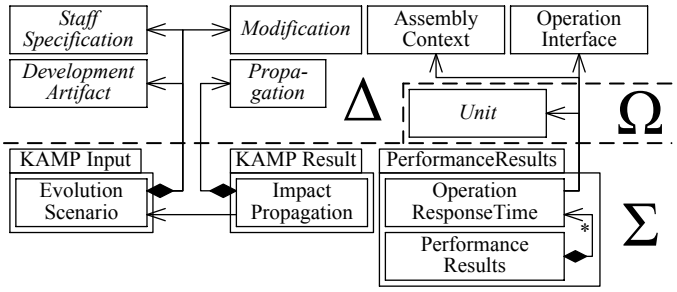

Fig. 7. Excerpt of the Σ (Analysis) Layer

The in- and output for the KAMP maintainability prediction is on the left side of Figure 7. In this circumstance, the quality layer is not needed. Both modules merely hold a root container and reference directly into the Δ layer. The EvolutionScenario of KAMP Input contains everything the analysis requires: seed Modifications, StaffSpecifications and DevelopmentArtifacts that belong to the architecture's elements. The ImpactPropagation contains the predicted extend of the architecture change in the form of Propagations. The ImpactPropagation further references the input of the scenario to enable inference of affected artifacts and staff.

E. Overall Module Structure

For an overview of all the modules of the scenario and their dependencies, see Figure 8. The modules which convey specific concerns are indicated by different levels of gray (maintainability is dark, performance is medium, reliability is light). The remaining modules cover more fundamental concerns: component-based architecture and behavior. They are the intersecting set of the metamodel elements of these analyzes. The illustration may seem much more complex, than the initial illustration in Figure 3. This is because in the initial figure, all the concerns of performance, reliability, behavior, and component architecture are contained in the big PCM module. The original structure cannot be subdivided into modules. At least not into modules according to the definition in this paper, as the subpackages of the PCM have cyclic dependencies (see [2]).

This modular structure brings many benefits. External extensions now have a clean base to build on. E.g., a security extension can now be developed without having to deal with performance and reliability. In the modular structure, extension developers have to only understand the modules they extend and to some degree modules from which they are indirectly dependent. This increases the potential of reuse of the more basic (higher) modules. When evolving the metamodel, change impact can be traced down the graph, following the depen-

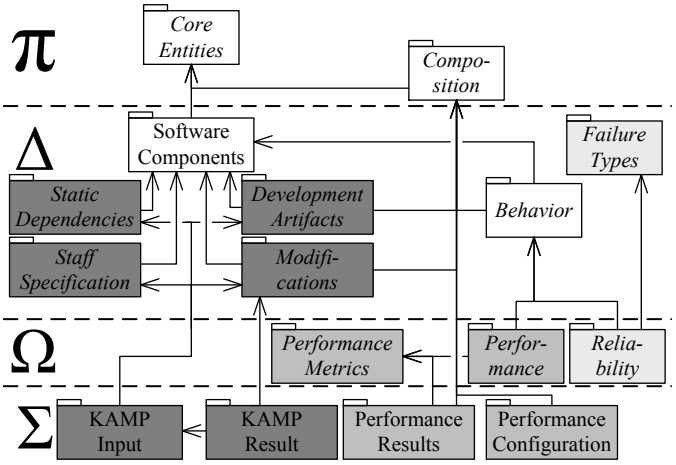

Fig. 8. The Scenario Metamodel Restructured In Concordance to the Reference Structure

dency relations. This way changes can be assessed much faster and more accurately.

V. Related Work

There are approaches, which deal with the modularization, categorization or structuring of related concepts. Coad's UML archetypes [7] for object-oriented design are used to classify classes into things, temporal concepts, roles and descriptions. Atkinson et al. propose a view-based approach for software engineering. The underlying model captures every concern into orthogonal dimensions, which are assessed through views [5]. Further, with deep modeling [21] they propose to enable instance relations within models. These deep models are layered with regards to instance levels. JetBrains MPS [8] features capabilities for the extension of DSLs. Siedersleben [6] proposed a reference structure for software architectures, where components are categorized into so-called blood types (technical, domain and library). Yet these approaches cannot be transferred to work on metamodels or they do do not offer support for metamodel extension and reuse.

There is also related work, which aims at modularizing concepts which are in direct interplay with metamodels. Jung [10] proposes a composition approach for generators. Rentschler [9] developed an approach for modular transformations. Föhrdes [11] presents a modularization into components of a performance simulator which operates on the PCM (metamodel). These approaches are especially interesting, as they deal with artifacts which are used in conjunction with metamodels. However, theses approaches cannot be directly transferred onto metamodels.

VI. Conclusion

In this paper, we proposed a reference structure for modular metamodels for component-based architecture description languages. It proposes a layering of the information into paradigm (π), domain (Δ), quality (Ω) and analysis (Σ) content. The reference structure is applied to the PCM [1] and the KAMP [4] maintainability extension. However, a remake of the PCM is not the contribution of this paper. These models were chosen to demonstrate the applicability of the reference structure. Our approach aims to be applicable to component-based architecture description languages which also express quality properties.

Our reference structure propose modularization into layers and further into modules as well as making the dependencies explicit. Dependencies are constrained to avoid dependency cycles and improve modularization. Guidance for the modularization with regards to concerns is given. The reward is a more modular metamodel which allows for better compositionality of extensions. The improved modularity leads to a reduced complexity and all its benefits: better understandability, maintainability and reusability.

Future work include an in-detail examination of the possible extension mechanisms which can be used for the extension relation. Also, we plan to develop decision support for the grouping of classes and modules into the layers. Further, an extensive metamodel will be remade according to the reference structure and its correctness and completeness proven (e.g., by finding an isomorphism). An interesting question worth investigating is if different types of extension with regards to the interplay of abstract and concrete module have implication onto potential roots elements of view types.

References

[1] S. Becker et al. "The palladio component model for model-driven performance prediction," *JSS*, Elsevier, 82(1):3–22, 2009.

[2] M. Strittmatter et al. "Identifying semantically cohesive modules within the palladio meta-model," *SSP*, 160–176, 2014.

[3] R. Heinrich et al. "Integrating business process simulation and information system simulation for performance prediction," DOI 10.1007/s10270-015-0457-1, *SoSyM*, Springer, 1–21, 2015.

[4] K. Rostami et al. "Architecture-based assessment and planning of change requests," *QoSA*, ACM, 21–30, 2015.

[5] C. Atkinson et al. "Orthographic software modeling: a practical approach to view-based development," *ENASE*, Springer, 69:206–219, 2010.

[6] J. Siedersleben *Moderne Software-Architektur: Umsichtig planen, robust bauen mit Quasar*, dpunkt, 2004.

[7] P. Coad *Java modeling in color with UML*, Prentice Hall, 1999.

[8] M. Voelter et al. "Language modularity with the mps language workbench," *ICSE*, IEEE, 1449–1450, 2012.

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

[10] R. Jung "Geco: Generator composition for aspect-oriented generators," *Doctoral Symposium - MODELS*, 2014.

[11] C. Föhrdes "Simulation components for software quality simulation in eclipse," Master's thesis, Karlsruhe Institute of Technology, 2014.

[12] OMG "UML Infrastructure Specification 2.4.1," 2011.

[13] OMG "MOF Core Specification 2.4.2," 2014.

[14] M. Strittmatter et al. "Towards a modular palladio component model," *SSP*, CEUR, 49–58, 2013.

[15] F. Brosch "Integrated software architecture-based reliability prediction for it systems," Ph.D. thesis, Karlsruhe Institute of Technology, 2012.

[16] C. Rathfelder "Modelling Event-Based Interactions in Component-Based Architectures for Quantitative System Evaluation," Ph.D. thesis, Karlsruhe Institute of Technology, 2013.

[17] M. Hauck "Extending Performance-Oriented Resource Modelling in the PCM," Diploma thesis, University of Karlsruhe, 2009.

[18] R. Reussner et al. "The Palladio Component Model," Tech. Rep., Karlsruhe Institute of Technology, 2011.

[19] M. Becker et al. "Performance analysis of self-adaptive systems for requirements validation at design-time," *QoSA*, ACM, 2013.

[20] R. Jung et al. "A method for aspect-oriented meta-model evolution," *VAO*, ACM, 19–22, 2014.

[21] C. Atkinson et al. "Melanie: Multi-level modeling and ontology engineering environment," *MW*, ACM, 7:1–7:2, 2012.