Karlsruhe Reports in Informatics 2018,11
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

Supplementary Material for the Evaluation of the Layered Reference Architecture for Metamodels to Tailor Quality Modeling and Analysis

Misha Strittmatter, Robert Heinrich, Ralf Reussner

2018

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

Supplementary Material for the Evaluation of the Layered Reference Architecture for Metamodels to Tailor Quality Modeling and Analysis

Technical Report

Misha Strittmatter, Robert Heinrich, Ralf Reussner
{strittmatter|heinrich|reussner}@kit.edu

18.12.2018

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Contents

1 Introduction

This technical report contains supplementary information to the evaluation of the Layered Reference Architecture for Metamodels. Chapter 2 provides detailed descriptions of the case study metamodels (original and modular version). Chapter 3 provides installation instructions for the evaluation tool. Section 4.1 presents the evolution scenarios in detail. Section 4.2 contains information about the models.

All metamodels, Modular EMF Designer diagrams, the evaluation tool, as well as the input and output data can be found online:
`https://github.com/kit-sdq/Metamodel-Reference-Architecture-Validation`

2 Case Studies

This section is concerned with the case studies that we modularized according to the reference structure. The metamodels that we used as case studies are the Palladio Component Model [7], Smart Grid Topology [5] (a DSML for modeling and resilience analysis in smart grid topologies[1]), KAMP4aPS [2] (a DSML for modeling and predicting the maintainability of automated production systems) and the BPMN2 [4] (a DSML for modeling business processes). For each metamodel we present the original metamodel, describe the modularization and present the resulting modular metamodel. In the description of the modular metamodel, we will not go into detail about transitive dependencies, as they do not influence the dependency graph.

It is important to note that we created the modular versions of the case study metamodels for the 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 internal validity of the evaluation.

To give an overview of the case studies, we applied several basic counting metrics to all metamodels. Table 2.1 shows the results. The first row shows the names of the metamodels. They are grouped after the four case studies. Within a group, the left metamodel is the original version; the right metamodel is the modularized version. The metamodel elements that were counted are listed in the first column. Although a containment is a special case of reference, the amount of containments is not included in the number of references. The dependencies row shows the sum of all dependencies (attributes, inheritances, references, containments).

2.1 Palladio Component Model

The starting point for the modularization is version 4.1[2] of the PCM.

Original Metamodel The PCM features six view types. These view types are good indicators for the topmost decomposition. We will now briefly describe these view types. For more in-depth information, please consult the respective literature [1, 8]. The *Repository* view type is used to define components and interfaces. Components provide and require Interfaces, which results in *Provided Roles* and *Required Roles*. The definitions of the Components is independent of the software systems in which they are used. The *SEFF* (Service EFFect Specification) view type enables the modeling of the behavior of the services of the components and their resource demands. It resembles a flow chart and an activity diagram.

[1] `https://sdqweb.ipd.kit.edu/wiki/Smart_Grid_Model`
[2] `https://sdqweb.ipd.kit.edu/wiki/PCM_4.1`

Metamodel	PCM	mPCM	SmartGrid	mSmartGrid	KAMP4aPS	mKAMP4aPS	BPMN2	mBPMN2
Metamodules	5	27	3	6	5	9	4	28
Packages	24	42	3	7	12	23	4	31
Classes	203	229	30	34	185	185	157	163
Attributes	56	54	9	9	14	14	135	135
Inheritances	193	194	25	25	163	163	157	162
References	198	174	15	18	117	115	134	151
Containments	120	131	11	14	101	92	103	79
Dependencies (Σ)	567	553	60	66	395	384	529	527

Table 2.1: Case Studies: Counting Metric Results

There is an abstract SEFF class, that allows for the extension of SEFFs of arbitrary type (e.g., data flow). For the sake of simplicity, however, we will address behavior describing SEFFs simply as SEFF. Systems and Composed Components can be described using the *Assembly* view type. There, Components can be instantiated (by so-called *Assembly Contexts*) and their Roles can be connected. In the *Resource Environment*, *Resource Containers*, which represent servers and workstations, their connections, and resources are modeled. In the *Allocation* view type, the Assembly Contexts of a system can then be deployed to Resource Containers of a Resource Environment. The *Usage Model* enables the modeling of behavior of the users of the system.

The module structure of the PCM is shown in Figure 2.1. It consists of five metamodules. *Identifier* provides a superclass for all classes that need an identifier attribute. *Units* defines units and provides a superclass that keeps track of a unit. *StoEx*, which is short for stochastic expression, defines arithmetic on random variables, which are used in the PCM to define and modify parameter values. *ProbFunction* defines abstractions to model probability functions, which can be used in stochastic expressions.

Around 73 % of the classes of the PCM metamodel reside in the *PCM metamodule*. This metamodule defines all main concepts of the PCM like components, interfaces, composition, assembly, resource environments, deployment, and usage models. Figure 2.2 shows the package structure of the PCM metamodule. If a package that is located within another package, it means the outer package contains the inner package. The arrows between the packages represent dependencies between the classes of the packages. The figure makes several simplifications to ensure clarity. Dependencies to and from the packages on the third nesting level (e.g., composition) count towards the dependencies of their parent packages (e.g., core in the case of composition). The figure omits transitive dependencies and dependencies to the entity package as these are very numerous. All view types of the PCM are reflected in the package structure. The Assembly view point is implemented in the Composition package.

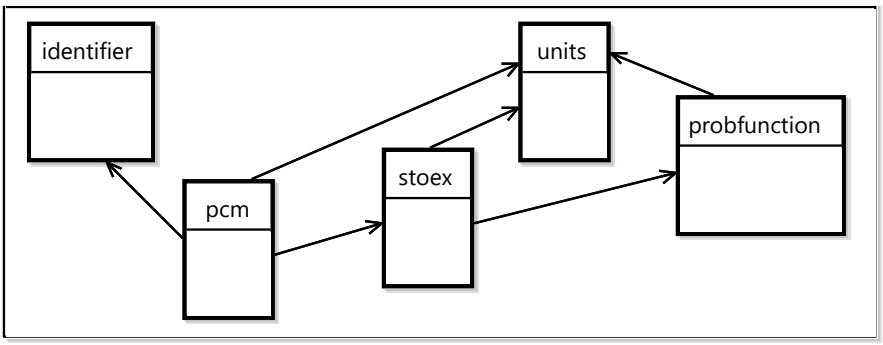

Figure 2.1: PCM Module Structure

The *PCM package* is the root package of the PCM metamodule. It simply contains the other packages. The *Core* package contains the entity and composition package, as well as class that implements random variables. *Entity* provides several abstract superclasses. *Composition*, *Repository*, *UsageModel*, *ResourceEnvironment*, *Allocation*, and *SEFF* contain mostly classes that implement their respective view types. However, they also contain classes of crosscutting features and extensions. The *System* and *Subsystem* packages contain one class each, which represents a software system and a software subsystem respectively. *ResourceType* contains classes that specify Resource Types, which are used by the Resource Containers. *Protocol* provides one single abstract class, that can be used as an extension point to define protocols [6]. It is currently unused. *Parameter* implements abstractions for the specification and manipulation of variable values. *Reliability* provides modeling of failure types and their occurrences. The *SEFF Performance* subpackage provides resource related calls as well as resource demands. This may suggest, that its parent package *SEFF* is free from resource dependent abstractions. However, it is not. *SEFF Reliability* provides abstractions to handle recovery from failures. It has the same problem as the SEFF Performance package, as the classes in SEFF still contain reliability related properties. *QoS Annotations* stands for quality of service annotations and implements a extension point for Systems. This extension point can be utilized by performance and reliability abstractions that are defined in its subpackages *QoS_Reliability* and *QoS_Performance*.

Modularization During the refactoring of the PCM, we split the PCM metamodule into 23 smaller metamodules to separate its language features [10] properly. The modularization of the PCM metamodule was driven by the effort to separate the view-points and to extract their advanced features to make them extensions. By doing so, the basic view point metamodules would be decoupled from their advanced features. The other four metamodules were already sufficiently modular and fitted well into the π layer. The number of classes in the modular 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

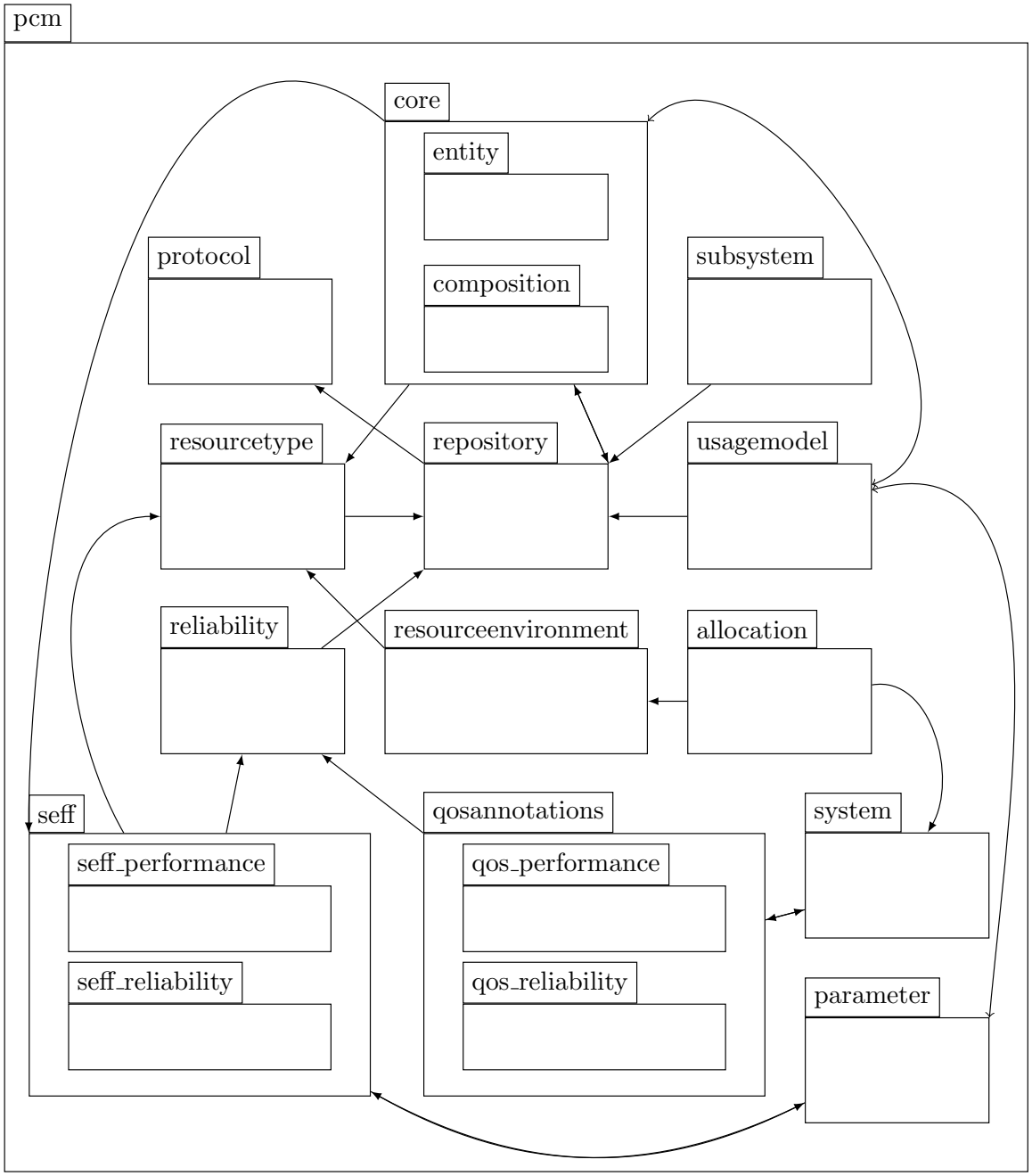

Figure 2.2: Package Structure of the PCM [10]

the reference architecture were removed or remodeled. The number of containments increased from 120 to 131, as new extending classes needed to be contained.

In the next section, we present the metamodules of the mPCM and explain how we refactored the PCM to achieve the modularization. During the refactoring, we performed refactorings and modifications of the following types many times. We will only mention the concrete operations and refactorings, if it is of special interest.

- Moving of classifiers between packages (possibly packages of different metamodules)
- Moving packages into another package (possible into another metamodule)
- Creating, deleting, renaming packages and modules
- The deletion of redundant relations that violated the constraints of the reference structure
- The reversion of dependencies that violated the constraints of the reference structure
- The creation of a new root container for a metamodule
- The creation of containments from root containers
- Renaming of classes (e.g., after we have factored out properties belonging to another concern)

Modular Metamodel Figure 2.3 shows the module structure of the mPCM. For the sake of simplicity, we have hidden transitive dependencies.

In the following we will present the resulting metamodules. For each metamodule we will explain its purpose, its dependencies, and how we created it in the refactoring process.

Paradigm The π layer contains the unaltered metamodules Identifier, Units, Probfunction and StoEx. It also contains the two basic metamodules Base and Variables that are used by many other metamodules. π further contains 5 metamodules that define view types.

Base The Base metamodule provides two superclasses that are commonly inherited from. The NamedElement class provides a name attribute. Entity inherits from NamedElement and the Identifier class (from the Identifier metamodule) to combine name and ID attributes. As almost all other metamodules use these superclasses, we will not explicitly mention dependencies to Base. The Base module does also contain a dummy class, that is not used and was only introduced to the PCM as a technical workaround. The execution engine of the transformation language QVT-R was not able to handle a root package without any classes. We did not remove the class, as it does not validate the constraints of the reference structure. Thus, by removing it, it would have harmed the internal validity of the evaluation. We created base due to the big initial horizontal split. It originates from the Entity package. It was not split as a language feature, but as a featureless metamodule that is used by other language features.

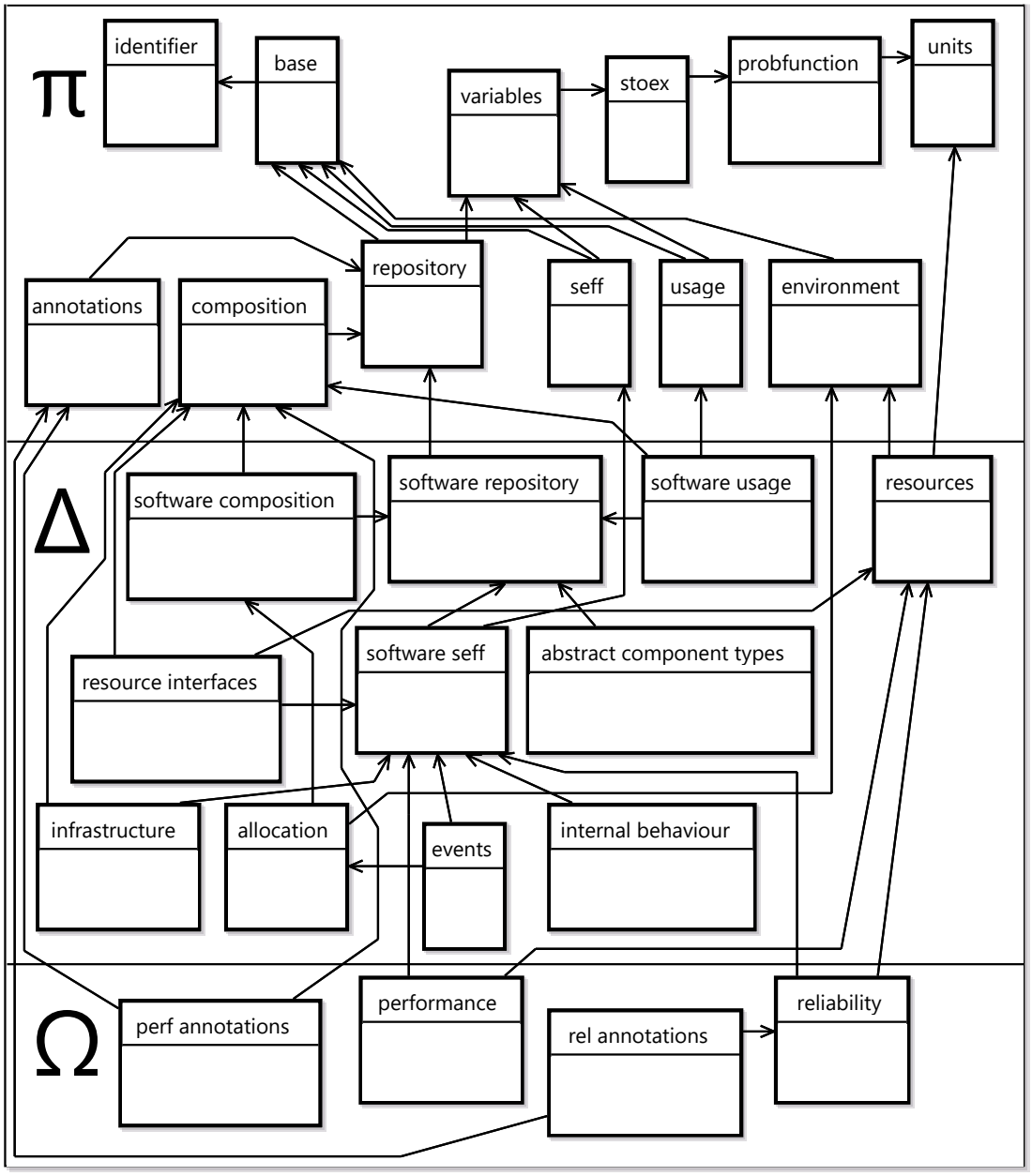

Figure 2.3: mPCM Module Structure

Variables This metamodule enables to model properties of variables. It does that on the basis of arithmetic of random variables and thus depends on the StoEx metamodule. Variables originated from the Parameter package. We factored it out due to a horizontal split to separate its language feature. The class PCMRandomVariable, which is now part of the Variables metamodule, had many outgoing container relations, which were redundant. As Variables is a π metamodule, many of the referenced containers are located in more specific layers. Container relations to such classes violated the constraints of the reference structure and had to be removed. The other container relations remained, except if they caused a dependency cycle, to not harm the internal validity of the evaluation.

Repository The Repository now contains the most basic versions of the abstractions of the former repository view type. We factored out all extensions (e.g., infrastructure, events) and content of more specific layers (e.g., software, performance, reliability). What remains are Components, Interfaces and their relations (Roles). We formed Repository in the scope of the big initial horizontal split and the subsequent paradigm extraction from its Δ counterpart.

Composition (π) This metamodule lays the abstract superclass ComposedStructure for all structures in the PCM that contain instances of components and their connectors. Composition provides the new superclass Containable. From this superclass all classes that can be contained in a ComposedStructure must inherit. This metamodule defines AssemblyContexts and Connectors as containable. Composition depends on Repository, as a AssemblyContext references a Component. Also, some ComposedStructures need Interfaces. So, a further superclass in Composition inherits from a superclass in Repository that provides Roles. Composition is transitively dependent on Variables, as a ComposedStructures may feature parameters. Composition originated from the initial horizontal split and the subsequent paradigm extraction from its Δ counterpart.

Usage, SEFF (π) The metamodules Usage and SEFF implement the domain-independent portion of their respective view types Usage Model and SEFF. Both metamodules are dependent on Variables, as they uses random variables. Both originate from the initial horizontal split and the subsequent paradigm extraction from their Δ counterpart.

Environment The environment resulted from the resource environment view type. We factored out all resource-dependent content into Δ metamodules. ResourceContainers are now Containers, LinkingResources, which connect Containers, are now Links.

Annotations Annotations contains the quality independent part of the QoS Annotations package. It established an extension for services of Signatures and is therefore dependent on the Repository metamodule. It originated from the initial horizontal split and the subsequent paradigm extraction from its Δ counterpart.

Domain The Δ layer of the mPCM provides abstractions for the domain of software components. Therefore, the Δ layer extends the view type implementing metamodules of Repository, Composition, Environment, SEFF and Usage by respective Δ modules.

Software Repository This metamodule extends its counterpart in π by domain-specific content: exceptions and interfaces that provide operation. It also defines an atomic component, that has an abstract class as a generic extension point to specify the effects of services. Although the behavior describing SEFF metamodule uses this extension point, it is not behavior-specific and can therefore be used for other kinds service effect specifications. Therefore, this metamodule is free from content of the behavior features. On its own, the Software Repository can be used to define software components their interfaces and operations. It is, however, mostly used together with composition and SEFF. Software Repository is transitively dependent on Variables, as it enables component-wide parameters for their operations. Software Repository originated from the initial horizontal split. It implements a standalone features and therefore needs to be separated from metamodules it is not dependent on.

Abstract Component Types This is a small metamodule, that defines two abstract component types. They can be used as blueprints in the component architecture of a system, as components with full service effect specifications are not yet available. As soon as they are available, they can replace the abstract components. This metamodule distinguishes implemented components from unimplemented components. Thus, it is Δ content and depends on the Software Repository instead of only depending on the Repository metamodule from π. It is transitively dependent on Repository (π) This metamodule resulted from a extension extraction from Software Repository. It is not essential for the modeling of Software Repositories, therefore it is an extension.

Resources This metamodule extends the Environment metamodule's containers and links by hardware resource specifications. These can either be used just for documentation or to simulate performance, as these resources process the resource demands that can be extended into SEFFs. In addition to its dependency to Environment, Resources also depends on Units, as for a ResourceTypes a Unit can be assigned. We performed an extension extraction to separate the Resources language feature from Environment. To achieve this, we split several classes and created a new root container.

Composition (Δ) The Composition metamodule extends its counterpart from the π layer domain-specific abstractions. It provides several concrete classes that inherit from the abstract π Composition concepts. These classes are: System, CompositeComponent, SubSystem and several Connectors. This metamodule can only be used together with Software Repository to describe how ComposedStructures (e.g., Systems and CompositeComponents) are internally structured. In addition to π Composition, this metamodule is dependent on Software Repository and transitively on Repository, as in Composition Components are instantiated into AssemblyContexts. This metamodule originated from the initial horizontal split.

Allocation The Allocation metamodule implements the Allocation view type. It provides the concepts that are necessary to deploy AssemblyContexts on Containers. Therefore it is dependent on Composition (Δ) and Environment. It is transitively dependent on Composition (π). This metamodule originated from the initial horizontal split.

SEFF (Δ) This metamodule provides many concrete classes that represent domain-specific Activities that it adds to the SEFF from π. It further extends the Software Repository by behavior as it provides a new subclass of the generic extension point that we mentioned earlier. Therefore, this metamodule depends on SEFF (π) and Software Repository. It depends transitively on Variables and Repository. This metamodule originated from the initial horizontal split.

Internal Behavior This metamodule is an extension of SEFF (Δ) and enables to model SEFFs that are not called through the interfaces of a component, but internally from other SEFFs. They are analogous to private methods in object-oriented programming. This metamodule is dependent on SEFF (Δ), as it is an extension. It is transitively dependent on SEFF (π) and Software Repository. We performed an extension extraction to remove these concepts from SEFF (Δ).

Usage (Δ) This metamodule extends its π counterpart by domain-specific concepts. It adds the description of workloads, and user-specific data. It enables the modeling of activities that call into the software system (so called EntryLevelSystemCalls). It is therefore dependent on the Software Repository, as it references Operations; and π Composition, as it references the provided role of a ComposedStructure. It is transitively dependent on Variables. This metamodule originated from the initial horizontal split.

Infrastructure This metamodule is an extension of the SEFF, Repository, and Composition view types. It introduces a new type of component, interfaces, roles, connectors, and calls. These new abstractions are called infrastructure and are used to model middleware. Besides the dependencies to the view type implementing metamodules it extends (SEFF (π and Δ), Software Repository, Repository, and Composition (π)), it is transitively dependent on Variables. Like the following cross-cutting extensions (Events and Resource Interfaces) we created this metamodule by an extension extraction. As it is a cross-cutting extension, we had to extract it from the packages of the respective view types.

Events This metamodule is an extension of the SEFF, Repository, Composition, and Allocation view types. It introduces abstractions to model event based communication. It provides event interfaces, roles, emit actions, connectors and also event channels that can be assembled and allocated. Besides the dependencies to the view type implementing metamodules it extends (SEFF (π and Δ), Repository, Allocation, and Composition (π)), it is transitively dependent on Variables.

Resource Interfaces This metamodule is an extension of the SEFF, Repository, Composition, and Environment view types. It provides modeling concepts to place resource demands on specific resources from within SEFFs. Besides the dependencies to the

view type implementing metamodules it extends (SEFF (π and Δ), Repository, and Composition (π)), it extends Resources and is transitively dependent on Variables.

Quality The quality layer contains two metamodules that implement abstractions to model Performance and Reliability. Further, two extension metamodules provide advanced concepts for Performance and Reliability respectively.

Performance The *Performance* metamodule enables the modeling of performance determining properties. This is achieved by adding resource demands to the activities within SEFFs and processing rates to Resources. This metamodule is therefore dependent on SEFF (π and Δ) and Resources. As well as transitively dependent on Variables. We created Performance due to an extension extraction, to rid the quality-independent language features SEFF and Resources from performance abstractions.

Performance Annotations The *Performance Annotations* metamodule allows to add unparametrized performance specifications to the operations of required roles of systems and to provided roles of components. Usually, the performance of a operation is determined by the resource demands of its SEFF and the processing rate and the contention on the required resources. However, it is not always possible to specify such detailed descriptions of the behavior and demand of an operation. Therefore, Performance Annotations can be used as a coarse performance abstraction. We created Performance Annotations by an extension extraction.

Reliability In short, the *Reliability* metamodule provides several failure types and modeling constructs to apply failure rates to Activities of SEFFs and to Resources. It also enables the modeling of recovery behavior after a failure. We created Reliability due to an extension extraction, to rid the quality-independent language features SEFF, Repository and Resources from reliability-specific abstractions.

Reliability Annotations This metamodule allows to specify reliability of Operations that are required by a System. It is dependent on Annotations and Reliability. We created this metamodule by an extension extraction.

Uncorrected Bad Smells and Modeling Errors As we already mentioned, we only refactored bad smells and modeling errors that violate the constraints that our approach imposes. We will now briefly elaborate on the bad smells and modeling errors that we did not fix as well as on general improvements that we did not implement. By using proper extension mechanisms, a large portion of the QoS Annotations metamodules could be dropped. The two π metamodules SEFF and Usage have a big overlap and should be consolidated. The class ResourceTimeoutFailureType has a reference to PassiveResource, which breaks modeling levels. Either ResourceTimeoutFailureType is not a FailureType but a failure occurrence, or or the reference is nonsensical. HDDProcessingResourceSpecification has redundant relations to ResourceContainer. The modules identifier and base could be merged, as they are both concerned with identity. We did not merge them, as we did not want to modify the five metamodules the original PCM metamodule is dependent on. ExceptionType is not a first-class concept, as it is not contained in a root container but

in the Signature class. This conflicts with ExceptionType being a type, as it should be possible to use instances of types from multiple places.

The following are occurrences of bad smells. ResourceInterfaceProvidingRequiringEntity is a dead class, as it is not referenced by any other class. Even if it were, it should not be abstract. Either RequiredResourceDelegationConnector or ResourceRequiredDelegationConnector is a dead class. There is a possible dead reference from Signature to FailureType. CharacterisedVariable may be a dead class. Before resolving possible dead properties and classes, they should be confirmed by searching dependent code for references. If no references are found, the class or property should be deleted, assuming there are no plans to use it in the future. There are still many redundant references, that did not cause cycles and did not violate the layering. These include redundant opposite relations and all container relations. ExceptionType might be a dead class.

Feature Model Figure 2.4 shows the feature model of the mPCM. All relations are required relations. Therefore, we have omitted the explicit required labels. Quality, View Types, Behavior, Structure, and Cross-cutting Extensions are grouping features and are therefore mandatory. We have grouped the view types into structural and behavioral features. Only the direct child features of the grouping features are classified by the grouping feature. For example, SEFF is a view type; its child feature Internal Behavior is not a view type. We placed it as a child of SEFF to demonstrate that it is an extension of SEFF and of nothing else (in contrast to the Cross-cutting Extensions). Resources and Abstract Component Types are also extensions and no view types. The Cross-cutting Extensions are advanced features and have no incoming requires relations from the rest of Δ. This means, that they could even be put in a sub-layer between Δ and Ω to enforce this decoupling. The small arrow that mark some required relations indicate that we pulled the relation up from all child features. For example, the requires relation from Quality to Resources was originally present in the Reliability and Performance feature.

For the sake of clarity in the diagram, we do not show the feature model together with the metamodule diagram. The grouping features do not have implementing metamodules. Neither has the mPCM root feature. The remaining feature nodes represent language features, are implemented by exactly one metamodule and are named like this feature. The π metamodules have no counterparts in the feature model, as they cannot be used without domain modules. Therefore they do not implement language features.

Further Decoupling Potential By looking at the feature model (Figure 2.4), more decoupling potential becomes apparent. This decoupling is not mandated by the guidelines of the reference structure, as the respective language features are intended to be used together. Such decouplings, however, increase the degree of indirection and complexity.

SEFF and Usage are dependent on Software Repository. By performing feature support extractions, the two features could be decoupled from Software Repository. This would enable the creation of system-independent Usage and SEFF Models without the need to install and load the Software Repository metamodule. For example, Usage could be decoupled quite easily by moving the EntryLevelSystemCall class into a new metamodule.

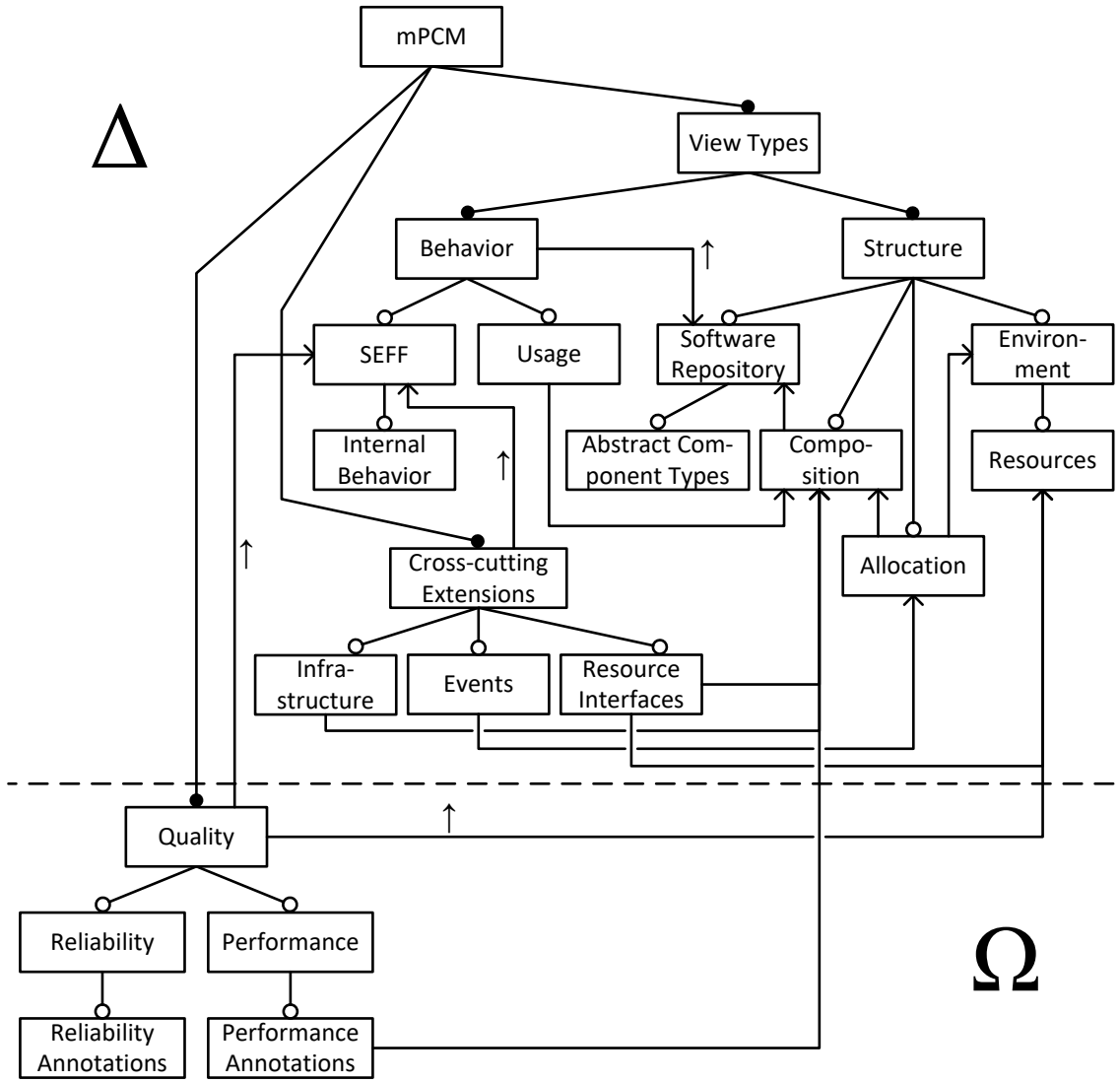

Figure 2.4: mPCM Feature Model (small arrows indicate the result of a pull up refactoring)

As EntryLevelSystemCall has no incoming dependencies within the Δ Usage metamodule, this would decouple Δ Usage from Software Repository.

The cross-cutting extensions are dependent on several view types, like the name suggests. If one of the extension features is selected, all required view type features are also selected. If it is desired to use only a subset of the view types with a specific extension, feature support extractions have to be performed to separate the parts of the respective extension that depend on the individual view types.

Both quality features are dependent on the SEFF and Resources features. By feature support extractions the parts that are dependent to these two features could be split. For example, this enables to model the performance of resources without being dependent on SEFF.

Predefined Metamodule Selections The modularization of the PCM enables a selection of language features according to the needs of the tool user. Based on the feature model in Figure 2.4, we will now present selection that fulfill the needs of certain user groups of the PCM. Of course, any selection is possible that fulfills to the constraints of a feature selection. However, these predefined selections will cover the needs of most tool users.

ADL ADL stands for architecture description language. In the context of the PCM this means the description of the component architecture without any quality information. This selection will usually used in early design stages or when reengineering the architecture of a legacy software system It consists of all structural view types: Software Repository, Composition, Allocation, Environment, and Resources. Optionally, if the description of behavior is also needed, SEFF, Usage or both can also be selected.

ADL+ This selection contains all selected features from the ADL selection with the addition of advanced features for expert tool users. It includes Abstract Component Types and all cross-cutting extensions. Optionally, if behavior is included in the ADL selection, Internal Behavior is also selected.

Performance Prediction This selection includes all view types as well as the Performance feature. As Quality is the parent feature of Performance, its required relations have to also be satisfied. Therefore, Resources is also selected. SEFF is already selected, as it is a view type.

Performance Prediction+ This is the advanced version of the Performance Prediction selection. It includes the same additional features as the ADL+ selection with the addition of Performance Annotations.

Reliability This selection is used for reliability analysis. It includes all view types, Resources, and the Reliability feature. Optionally, the advanced Δ features can be included as well as the Reliability Annotations feature.

2.2 Smart Grid Topology

Original Metamodel The Smart Grid Topology metamodel features four view types: the topology, types of devices in the topology, input state and output state. Input and output state are used by the analysis that is performed on the metamodel. It predicts the impact of the current power supply onto the smart devices in the topology.

Figure 2.5 shows the module structure of the Smart Grid Topology metamodel. It consists of three metamodules. Input and output state view types are implemented in their own metamodules. The Topo metamodule implements the device type and topology view types.

Modularization During the modularization, the input and output metamodules remained unmodified. We split the main metamodel module in several ways to separate the two view types and also to extract π metamodules. The number of classes increased from 30 to 34. The number of dependencies increased from 60 to 66.

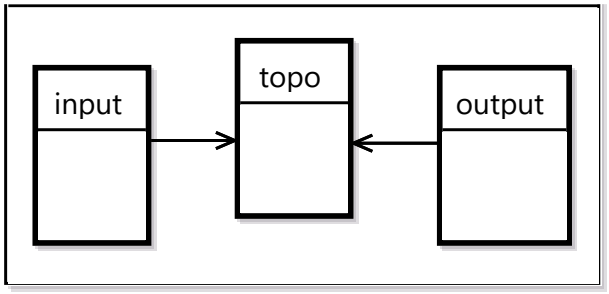

Figure 2.5: Smart Grid Topology Module Structure

Modular Metamodel Figure 2.6 shows the module structure of the modular metamodel. It populates the layers π, Δ, and Σ. In the following we will present the resulting metamodules. For each metamodule we will explain its purpose, its dependencies, and how we created it in the refactoring process.

Paradigm This layer contains the domain-independent metamodules Base and Graph.

Base This metamodule defines abstract superclasses that are used by all other metamodules. They provide name and ID attributes. As almost all other metamodules depend on Base, we will not mention incoming dependencies. This metamodules has no dependencies. Base originated from the horizontal split of Topo. It is not a language feature. We factored it out, as it used by several metamodules.

Graph This abstract metamodule defines a simple network graph structure. Nodes are connected by logical and physical connections and can be connected to power supply. Graph originated from the horizontal split of Topo.

Domain The Δ layer provides abstractions that are specific for the domain of smart grids. It contains the Topo and TypeRepo metamodules.

Topo This metamodule provides several smart-grid-specific types of devices and extends them into the graph structure by the means of subtyping. It therefore depends on Graph. This metamodule originated from the horizontal split of the original Topo metamodule.

TypeRepo TypeRepo extends SmartMeters, NetworkNodes and PhysicalConnections by Types that are stored in a Repository that is independent of concrete smart grid topologies. The extended classes lie in Topo and Graph. We factored TypeRepo out due to the horizontal split of the original Topo metamodule. Originally the devices and connections knew their types. So we performed horizontal splits to remove the type-dependent properties from the devices and connections. As this type information does not belong in the type definitions either, we created a new root container that now holds the three kinds of type applications.

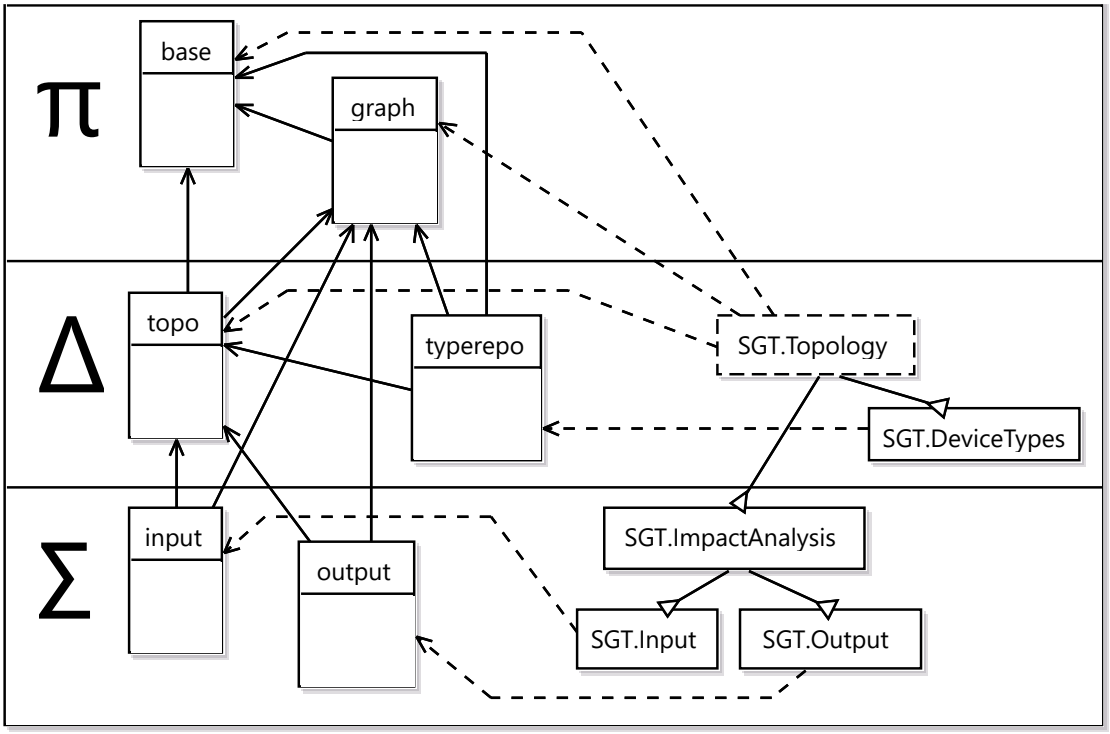

Figure 2.6: Modular Smart Grid Topology Module Structure and Feature Model

Analysis The Σ layer contains the Input and Output metamodules. We did not modify them, as they were already sufficiently modular and fit the Σ layer well.

Feature Model The feature model for the modular Smart Grid Topology metamodel is shown directly in the layered module diagram (Figure 2.6). The root node represents the Topology language feature. As the Topology language feature is always used, its feature was pulled up and merged with the formal root feature. Thus, it is implemented by the Topo metamodule and its dependencies. As the TypeRepo is an extension metamodule, it is reflected by the optional child feature DeviceTypes. ImpactAnalysis is a grouping feature node. Usually, grouping features are mandatory child features. However, it is best located on the Σ layer. Therefore it is optional, as its parent relation crosses a layer boundary. From a functional feature selection perspective, it is equivalent if the feature is placed on Δ or Σ. It is also equivalent if it is mandatory or optional as long as its children are all optional. The optional child features of ImpactAnalysis are implemented by their respective metamodule.

2.3 KAMP4aPS

Original Metamodel The KAMP4aPS metamodel features 3 view types. The Automated Production System (APS) view type is used to model the structure of such a system. The Field of Activity view type adds information about artifacts that are relevant for the evolution of the system. This includes information about the staff, tests, documentation, specifications, and further documents and files. The Modification Marks view point describes how the system is modified. Based on the information of the 3 view types, the KAMP4aPS analysis predicts the extent maintenance of the automated production system.

Figure 2.7 shows the module structure of the original KAMP4aPS metamodel. The APS, Field of Activity and APS Modification Marks metamodules implement their viewpoint. The Modification Marks metamodule is a generalized part from the KAMP metamodel that is reused by the APS Modification Marks metamodule. Basic contains superclasses that contribute name and ID attributes.

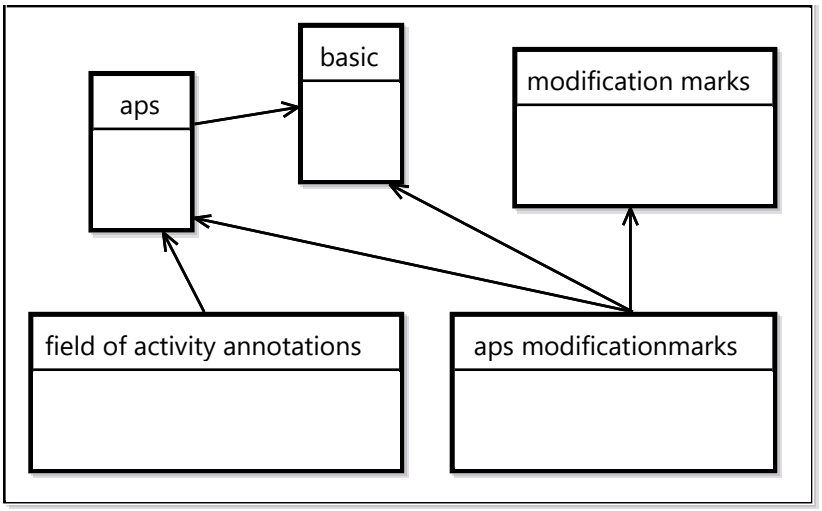

Figure 2.7: KAMP4aPS Module Structure

Modularization During the modularization, we split the APS metamodule into parts of different specificity: Automation Systems (AS), automated production systems, and a specialization for a specific kind of automated production system, called a pick an place unit (PPU). The same kind of modularization was performed on the module that describes modifications. In the scope of these two modularizations, we performed several dependency inversions to direct the module dependencies to go from the most specific to the most abstract metamodules.

The refactoring increased the number of metamodel modules from 5 to 9. The number of classes stayed constant at 185 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.

Modular Metamodel Figure 2.8 shows the module structure of the modular metamodel. It populates the layers π, Δ, and Ω. In the following, we will present metamodules that resulted from the modularization or were modified. For each metamodule we will explain its purpose, its dependencies, and how we created it in the refactoring process.

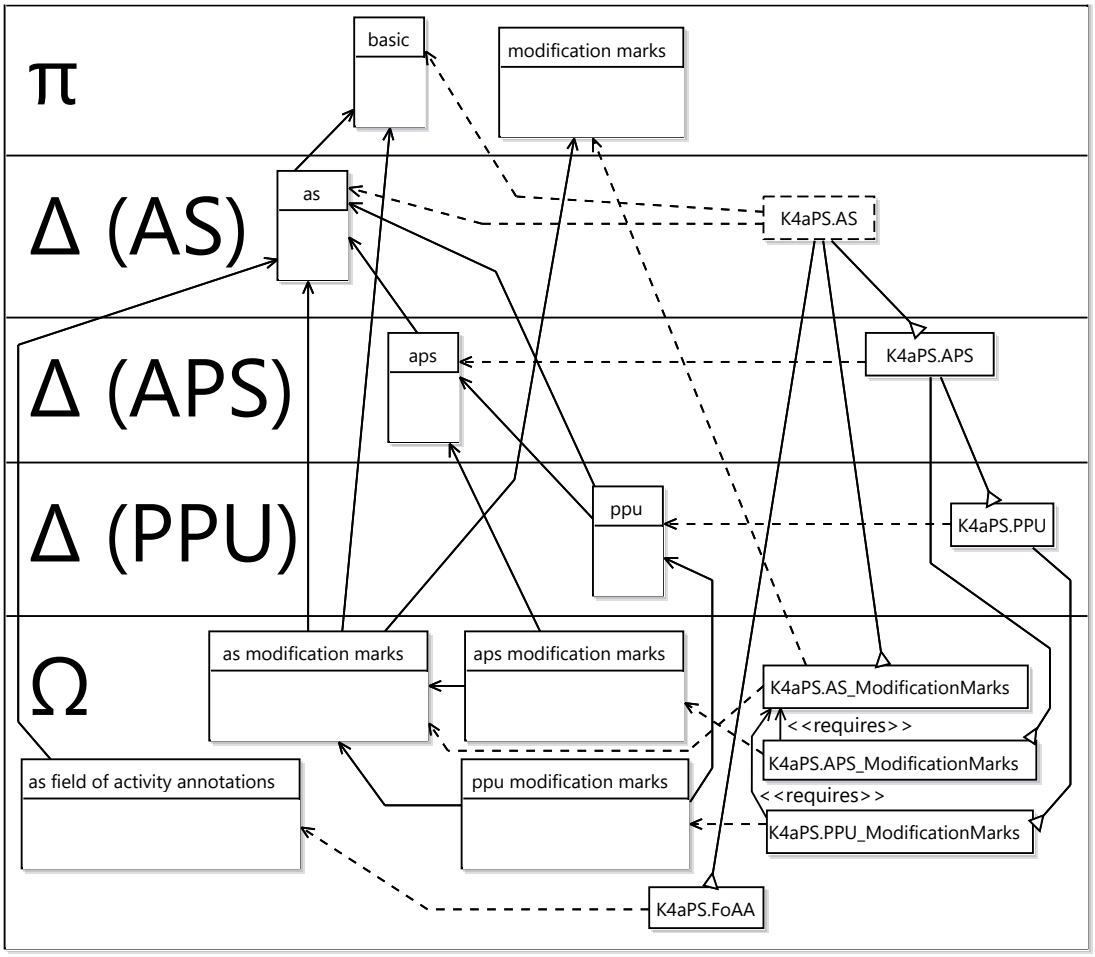

Figure 2.8: mKAMP4aPS Module Structure and Feature Model

Paradigm The π layer contains the Basic and Modification Marks metamodules. We did not change them, as they are already sufficiently modular and domain-independent.

Domain The Δ layer contains the metamodules that originated from the horizontal split of the APS metamodule. The more specific of these metamodules depend on the more abstract ones, as new subclasses are introduced and existing classes are referenced.

The Δ layer is subdivided into three sublayers to enforce the proper direction of the dependencies. This subdivision is optional. It, however, demonstrates nicely that the number of layers is not fixed to the ones that the reference structure suggests.

AS The AS metamodule contains quite general abstractions that can be used to model a wide range of automation systems. Such general modeling comes, however, with the loss of specificity.

APS The APS metamodule introduces more specific abstractions that are concerned with automated production systems.

PPU The PPU metamodule provides abstractions for pick and place units.

Quality The Ω layer contains the Field of Activity Annotations metamodule, which was not altered, as it is already sufficiently modular and only references the most abstract concepts from the AS metamodule. All metamodules of the Ω layer, are located here as they define abstractions that are needed to determine the maintainability of an automation (or more specific) system.

Modification Marks The Ω layer further contains the three metamodules that resulted from the split of the APS Modification Marks metamodule. It was split in a way to mirror the structure of the Δ layer: one metamodule for the Modification Marks of the AP metamodule, one for APS, and one for PPU. These metamodules reference their respective Δ counterpart as well as the AS Modification Marks module, as it provides superclasses.

Feature Model The feature model for mKAMP4aPS is shown directly in the layered module diagram (Figure 2.8). The root node represents the AS language feature. As the AS language feature is always used, its feature was pulled up and merged it with the formal root feature. Thus, it is implemented by the AS metamodule and its dependency Basic. The structure of the feature model pretty much mirrors the module structure. PPU is an optional child of APS. APS is an optional child of AS. AP, APS, and PPU have their respective ModificationMarks as optional children. AS, APS and PPU, their ModificationMarks and FoAA are implemented by their respective metamodules. Additionally, AS ModificationMarks is implemented by the abstract π ModificationMarks metamodule. As the APS and PPU ModificationMarks features are dependent on the AS ModificationMarks feature, they have required relation pointing towards it.

2.4 BPMN2

Original Metamodel Figure 2.9 shows the internal structure of the BPMN2 concepts that is conveyed by the standard [4]. It suggests a layered and modular structure. However, a look at the classes that implement these concepts, shows that they are often mutually or cyclically coupled by dependencies. Starting from the basic concepts in the middle, we will briefly give an overview of the concepts shown in the figure. For a more detailed explanation, please consult the standard [4].

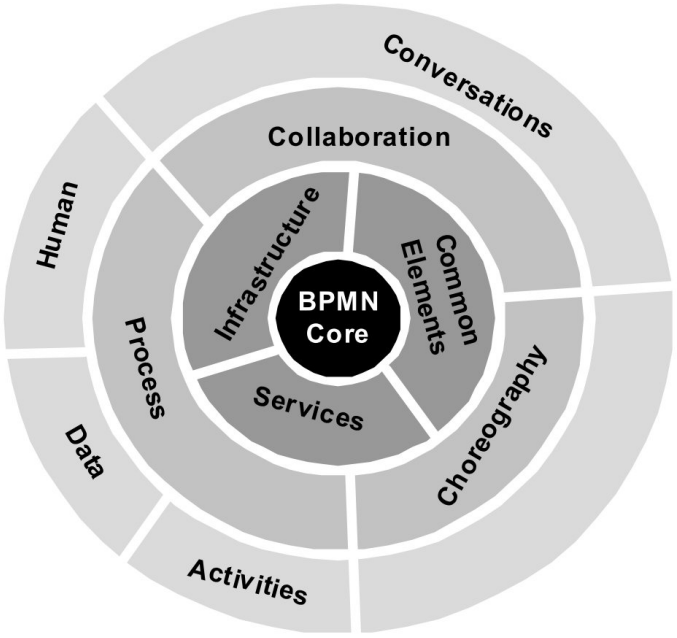

Figure 2.9: BPMN2 Concept Structure [4]

Infrastructure Infrastructure contains the most basic classes of BPMN2: Definitions, the root container of all BPMN2 models, and Import, which is used to reference external resources.

Foundation Foundation, which is not shown in the figure, provides classes that are fundamental to an abstract syntax and are needed by the three other core packages.

Commons Commons (Common Elements in the figure) provides classes that are needed by the advanced concepts Process, Choreograpy and Collaboration.

Services Services provides fundamental abstractions that are needed to model services, interfaces, and operations.

Process A Process is a sequence of activities. It is related to flow charts and activity diagrams. It consists of tasks, interactions with events, branching, loops, and many more. These elements can be partitioned into pools and lanes. A pool represents the actor who performs the process.

Collaboration Collaborations are used to model the interactions between processes and their message exchanges.

Choreography A Choreography is used to express the interaction between processes in a sequential way.

Data Data can be required by activities. It can represent information or physical objects and is used in messages.

Activities Activities are the main elements of a process. The most important activities are tasks, calls and sub-processes. Tasks are atomic activities that can be performed. Calls invoke a global process or task. Sub-processes contain a flow of activities and can be used for hierarchical decomposition.

Human Human is needed to express the involvement of persons in business processes. E.g., there are several types of tasks that have to be performed by a person.

Conversations A Conversation diagram is used to provide an overview of which pools interact with each others, but not how they interact in detail. The details of processes are usually not shown in the pools.

Figure 2.10 shows the module structure of BPMN2 version 2.0.2. It consists of 4 metamodules. We got the metamodel source from the BPMN2 Modeler Eclipse plugin[3] version 1.4.2. The main metamodule is BPMN2, which contains classes for all BPMN2 concepts. The three other metamodules are only used to express diagram information. One is BPMN specific. The others are more general and could be reused by other languages to express diagrams. There is a dependency cycle between BPMN2 and BPMN Diagram Interchange. This dependency hardly couples BPMN2 models with their diagram representation, which is undesirable.

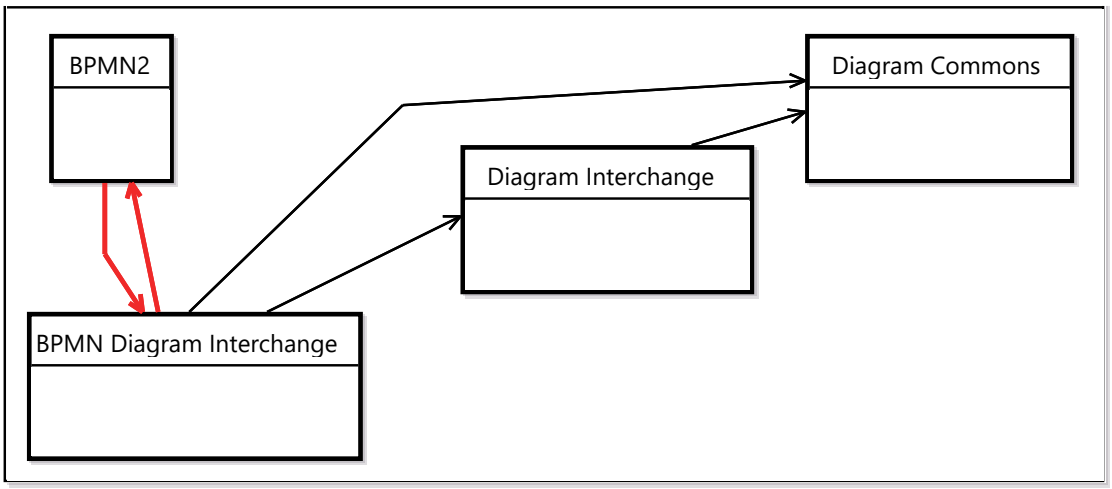

Figure 2.10: BPMN2 Module Structure

Modularization The only metamodule that we refactored is the BPMN2 metamodule. As it implemented all concepts of BPMN2, there was great modularization potential. As a starting point we modularized the metamodule into the groups of concepts that are proposed by the specification (as presented earlier).

[3] https://www.eclipse.org/bpmn2-modeler/

We reconstructed the result of the initial horizontal split in Figure 2.11. The diagram does not represent an exact state of the metamodule structure in the refactoring of the BPMN2, as in the modularization process we performed other refactorings (e.g., dependency inversion) in between the steps of the big horizontal split. The purpose of this figure is to illustrate the level of entanglement between the parts that the layering in Figure 2.9 suggests.

The final metamodules span two layers: π and Δ. We modularized the main metamodules according to its 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, as we was able to often inherit from the abstract class RootElement. RootElement is contained in to root container Definitions and therefore provides a convenient generic extension point. The number of dependencies slightly reduced from 529 to 527 (mainly because of redundant relations that violated the reference architecture).

We did not refactor the dependency from the original BPMN2 metamodule to the BPMN Diagram Interchange metamodule. Removing or inverting the dependency would have decoupled the BPMN2 metamodule completely from the diagram-related metamodules. In the evaluation, this would have improved the results for the modular metamodel significantly. However, we want to show the benefits of our approach regarding the more subtle and difficult modularization of metamodules. Although the dependency in question violates the constraints of our reference architecture, we did not want these benefits to be overshadowed by the results of such an easy refactoring.

The metamodel that We obtained contains one peculiarity that we had to resolve. It contains the class DocumentRoot, which is not covered in the standard. DocumentRoot holds a containment reference to every other class in the metamodel. This is strange, as these classes already form a proper containment hierarchy. It is also a grave bad smell, as it completely breaks the modularity. We had to remove it in both metamodels (the original and the modularized version) to get comparable results. Table 2.1 does not include the DocumentRoot and its properties.

Modular Metamodel Figure 2.12 shows the module structure of mBPMN2. For the sake of simplicity, we have hidden transitive dependencies (e.g., the dependency between BPMN Diagram Interchange and Diagram Commons).

In the following we will present the resulting metamodules. The names of the new metamodules relate strongly to concepts of the BPMN2 specification [4]. Thus, here, we will only refer to their internals where necessary. For each metamodule we will explain its purpose, its dependencies, further modularization potential where applicable, and how we created it in the refactoring process.

Paradigm Many BPMN2 concepts are not limited to the use of modeling business processes (e.g., many concepts are shared with or could extend flowcharts); thus, many metamodules are located at the paradigm layer. It was seldom the case, that a general concept contained domain information and a paradigm extraction had to be performed. Thus, many of the paradigm metamodules contain concrete classes. This is, however, justifiable for a refactored legacy metamodel.

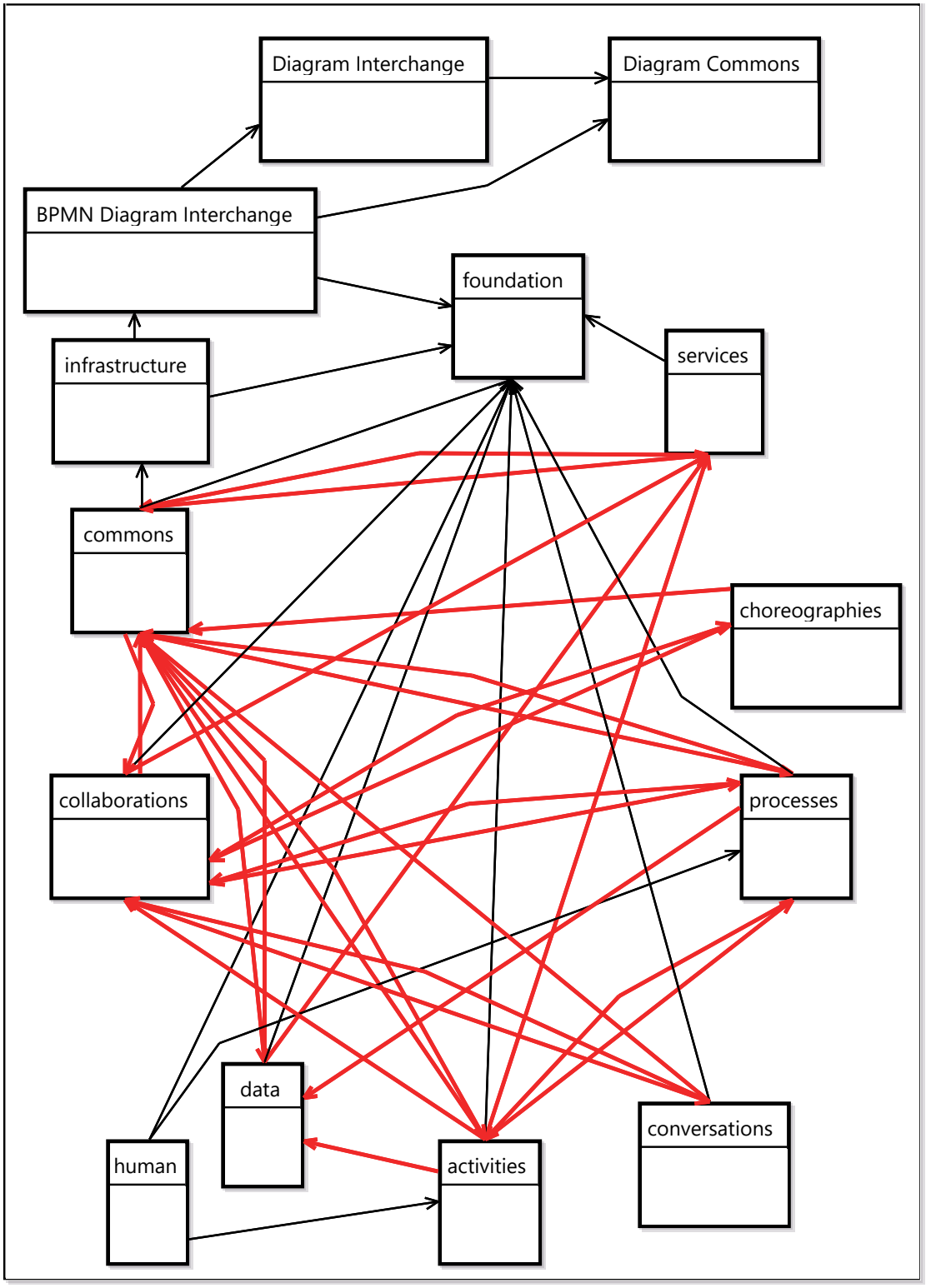

Figure 2.11: BPMN2 Module Structure after Horizontal Split According to the Structure in the Specification (Reconstructed)

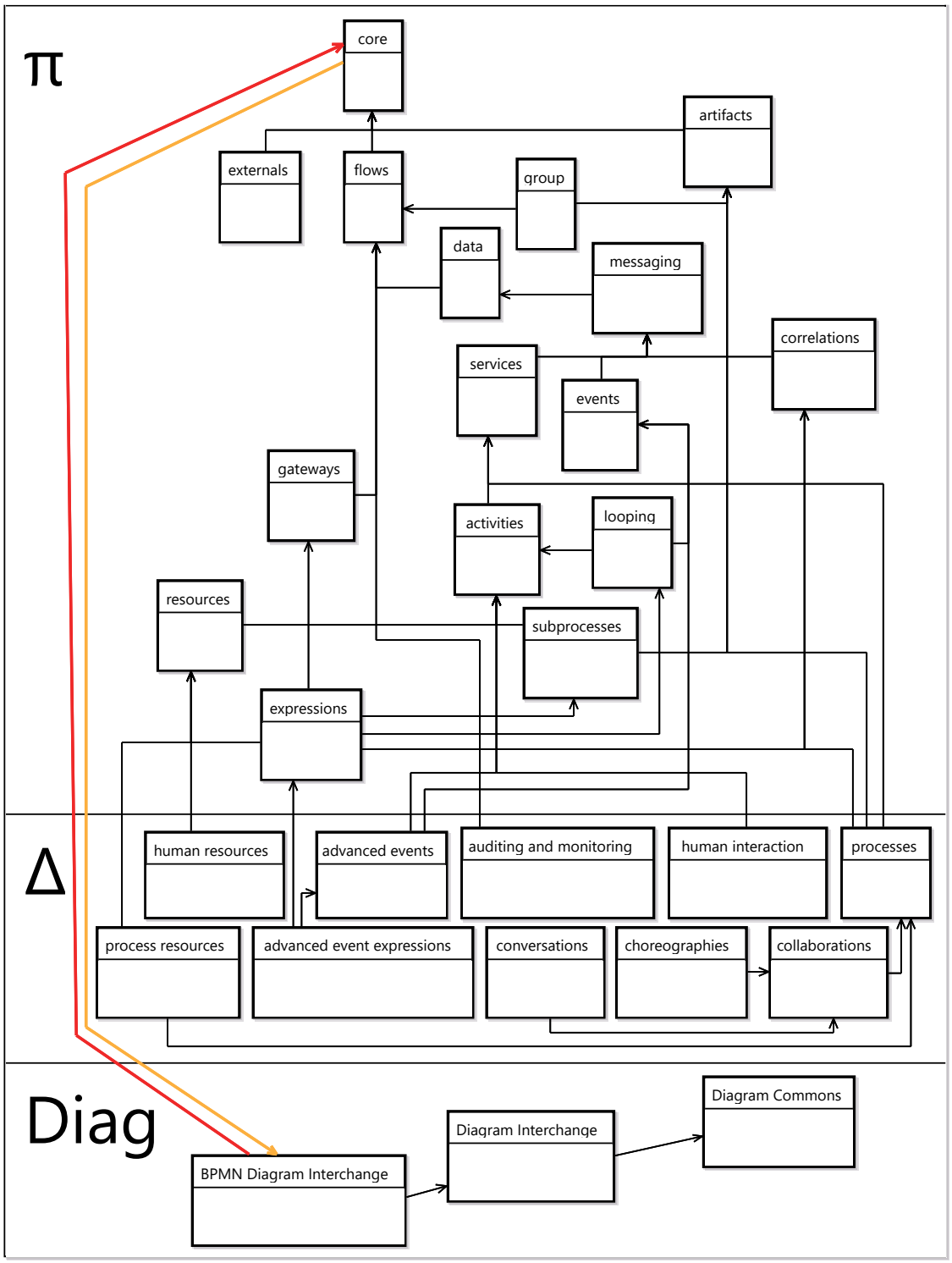

Figure 2.12: mBPMN2 Module Structure

Core This metamodule implements the most basic concepts: Definitions, which is the root container of all BPMN2 models; RootElement, the superclass for all first-class concepts; Documentation; and BaseElement, which provides an ID and a reference to documentation. Core has only one outgoing dependency, a containment to BPMN Diagram Interchange. Almost all other metamodules depend on Core. We did not explicitly factor the core package out of another metamodule. We was was the remainder of the modularization.

Artifacts This metamodule provides all BPMN2 Artifacts except Groups (i.e., Association and TextAnnotation). This metamodule is domain-independent and is therefore paradigm content. Artifacts is only dependent on Core. Five metamodules reference the Artifact metamodule. Here may be further refactoring potential in reversing these references to decouple the dependent metamodules from Artifacts. This would make Artifacts an extension metamodule. Artifacts was factored out of Core due to a horizontal split to separate language features. To make Core independent of Artifacts, we made Relationship inherit from RootElement and removed the explicit containment from Definitions.

Groups This metamodule defines Groups and the Category concept. A Group is an Artifact that groups values of a Category (i.e., FlowElements). It therefore has dependencies to Flows and Artifacts and a transitive dependency to Core. It has no incoming dependencies and is therefore a pure extension. We factored out Groups due to a feature support extraction from Artifacts (dependencies to Flow were factored out). To decouple Flows from Groups, we removed the reference from FlowElement to CategoryValue. We made the opposing reference, which was derived and transient, to a proper persistent reference.

Externals Externals provides capabilities to link external data and extend arbitrary data into BPMN2 models. These are usually used by Tools (mostly diagram editors) to store their tool-specific data, which the BPMN2 metamodel does not cover. Core is the only dependency of the Externals metamodule. With no incoming dependencies, this metamodule is a pure optional extension. Externals is the result of an extension extraction from Core. We reversed the incoming references from the BaseElement and Definitions classes of Core and introduced a new container for the now containerless classes. We further removed a redundant derived reference from ItemDefinition, which is now located in the Data metamodule, to decouple the class from Externals.

Flows Flows is a basic metamodule that defines flow sequences and abstract classes for their elements. The only dependency of Flows is to Core. We extracted this metamodule with a horizontal split to extract the respective concern. To decouple Flows from the much more specific concern of Processes and to resolve the dependencies layer violation, we removed the redundant derived reference from FlowNode to Lane.

Data This metamodule defines data, abstractions for data in- and outputs, and many more data related abstractions. The notion of data that the metamodule defines is general

enough to be considered a part of paradigm. As three classes can be part of a flow, it has inheritances on FlowElement and is dependent on the Flows metamodule. It further has a transitive dependency on Core. We performed an horizontal split to separate this metamodule.

Messaging Messaging defines abstractions for messages and their flows in a domain-independent way. It depends on the Data metamodule, as a Message can hold data. It has a transitive dependency on Core. It is possible that there is more modularization potential in this metamodule. The dependency to data could be inversed, to make data an extension of messaging. This would make data a pure extension without incoming dependencies. However, we do not have the necessary domain knowledge to decide which dependency direction is better. We separated Messaging in scope of horizontal splitting.

Gateways This metamodule introduces gateways, which can be used to fork flows. The gateways do not contain domain information and are therefore located in the paradigm layer. It is only dependent on Flows. An abstract superclass inherits from FlowElement and has several subclasses that define concrete gateways. We factored out gateways with an extension extraction. However, it could be that flows are always used together with gateways. In this case, the modularization is unnecessary and the two metamodules should be merged.

Correlations In the BPMN2 specification [4] it is written that "Correlation is used to associate a particular Message to an ongoing Conversation between two particular Process instances.". However, correlations are also used by FormalExpressions, which are paradigm concepts. This and the abstract nature of the concept contributed to our decision to assign the Correlations metamodule to the paradigm. The metamodule only depends on the Message class. It further has transitive dependencies to Data and Core. If Correlations is only seldomly used by Processes and FormalExpression, there is more modularization potential here. To perform a feature support split would decouple both metamodules from Correlations. We factored out Correlations from Messaging in the scope of an extension extraction.

Services Although there is no explicit service class in BPMN2, the content of this metamodule follows the BPMN2 specification that proposes a Services package. It defines Interfaces, which contain Operations, and service end points that can be externally extended. These abstractions are general enough to fit the paradigm layer. This metamodule depends on Messaging and transitively on Data, as an Operation may have Messages and Data as input and output. It has a further transitive dependency to Core. We created this metamodule due to horizontal decomposition.

Events The paradigm metamodule for events defines the basis on which the domain metamodule builds upon. It defines the abstract superclass and concrete classes like Start- and StopEvents. It depends on messaging, as Events can be the source and target of MessageFlows. Thus, the Events superclass inherits from InteractionNode. Transitive dependencies exist to Core, Data and Flows. We created the Events

metamodule due to a paradigm extraction, which separated it from its domain counterpart.

Activities This metamodule defines the activities within a flow. It is strongly coupled to the Services module, as Activities contains several classes that reference Operations and CallableElements as the represent or use services. Activities depends transitively on Data, Flows and Messaging. Here is, again, potential modularization potential. If service-oriented activities are not always used, they can be factored out. We extracted this metamodule with an horizontal split. To resolve a dependency cycle and a layer violation, we removed a redundant derived reference from Activity to BoundaryEvent.

Resources A Process may be performed by a Resource. This metamodule contains the domain-independent parts of the Resource concept. The metamodule depends on Activities, as a ResourceRole, which connects a Resource and a Process, references further activities that may be performed by a Resource. Resources also depends transitively on Data and Core. We made resources an extension, as it is not essential to define Processes and Activities. We separated it from Commons and inverted incoming dependencies from Activities.

Subprocesses Subprocesses are activities that contain an inner Process. This is achieved by inheriting from FlowElementsContainer of the Flow metamodule. Subprocesses has also transitive dependencies to Activities, Artifacts, and Messaging. We factored it out with a horizontal split from Activities.

Looping The Looping metamodule enables loops in flows. This modules depends on activities, as the Activities superclass can be extended by LoopCharacteristics. It is also dependent on Events, as certain loops are able to throw multiple events. Looping has transitive dependencies to Data and Core. We extracted Looping to make it an extension of Activities, as it is a rather specific feature. As loops are a specific activities, we decoupled Activities from Looping using dependency inversion. We removed the containment from the Activity superclass to the LoopCharacteristics superclass. As LoopCharacteristics was no longer contained anywhere, we created a new container class. We made the container class a subclass of RootElement (i.e,. using variant b of the referencing extension mechanism) to prevent model fragmenting. We could have also made LoopCharacteristics a subclass of RootElement, which would have reduced complexity, as no new container class would have been needed. As this has the potential to severely cluttered the set of RootElements in a Definition, we decided against it.

Expressions This metamodule implements informal and formal Expressions. FormalExpressions may be executed by a simulator or interpreted by an analyzer. Many concepts like Gateways, Subprocesses, Loops, Correlations and Resources use Expressions to express conditions. Thus, this metamodule depends on Gateways, Subprocesses, Loops, Correlations and Resources It is further transitively dependent on Data and Flows. As Expressions depends on so many advanced features, there is

more modularization potential. A drawback of the current stat of Expressions is by using or reusing it, all its dependencies are required, even if they are not needed by the user or reuser. It could be beneficial, to perform several feature support refactorings, to decouple the general concept of expressions from all the extended metamodules. The metamodule was first created, when during the big vertical split of the Commons. Expressions is a cross-cutting feature and many metamodules depended on it. However, as it is not essential for defining BPMN2 models, we conducted dependency inversion to make it a cross-cutting extension. We also made the Expressions superclass a RootElement.

Domain The Δ layer provides modeling abstractions for the domain of business processes. It contains the view type implementing metamodules Processes, Collaborations, Choreographies and Conversations. It further extends π metamodules by business process specific content like events, auditing, monitoring, and human interactions.

Resources This metamodule contains the domain-specific part of the original Resource concept. Its only purpose is to extend the Processes metamodule. As the Processes module is Δ content, this metamodule also belongs in Δ. Thus, it depends on the Process metamodule and on the Resources metamodule of π. We performed dependency inversion to decouple Process from Resources. We extracted the resulting dependency in We used a split class refactoring to separate this dependency from the ResourceRole class in order to achieve a paradigm extraction.

Resources.Human Resources.Human contributes human specific resource concepts. Its only dependency is to the Resources metamodule of π, as it uses Performer as a superclass. We created this metamodule, due to a horizontal split of the Δ Resources metamodule to separate the human specific content.

Expressions This metamodule implements a feature support of the π Expressions metamodule for Events.Advanced of Δ. It extends two events with Expression support. As the supported feature is part of Δ, this metamodule is also in Δ. It is, only dependent on Expressions of π and Events.Advanced. At first we reversed the dependencies from Events to Expressions, to make it an extension and to decouple Events from Expressions. To decouple Expressions from Events, we created this metamodule as feature support. We did this by splitting the Expressions superclass, which was carrying the reversed dependencies.

Events.Advanced This metamodule holds Events that are too BPMN specific for the π layer. It is, of course, dependent on the Events of π. It also depends on Activities, as Boundary- and CompensateEvents reference the Activity superclass. It has transitive dependencies to Core, Data, Services, and Messaging. We factored it out of Events with a paradigm extraction.

Processes This metamodule defines the Process concept, which contains LaneSets, which in turn contain Lanes. Processes is part of Δ, as it contains properties that are domain-specific. However, if a concept that is similar to Processes should be defined

for another domain, all the classes of π that Processes uses can be reused. It depends on Artifacts and Correlations, as a Process contains the Artifacts superclass and CorrelationSubscriptions. As mentioned earlier, here is further modularization potential. This metamodule further depends on Services, as a Process is a CallableElement. This metamodule is transitively dependent on Core, Data, and Flows. We separated Processes due to horizontal decomposition. We reversed a reference from Process to Collaboration, as Collaboration builds on the Process concept but not vice versa.

FlowElementsContainer from Flows is a superclass of Process. The FlowElementsContainer had a containment to the LaneSet class of Processes. To decouple Flows from Processes, we pushed down the containment to the Process class. This was possible, as the containment is not used in the other subclasses of FlowElementsContainer (Choreography and SubChoreography) as stated in the standard. Having this containment at this point in the inheritance hierarchy was not only a layer violation, but actually the unused feature of superclass smell.

Collaborations Collaborations are used to express the interaction between Processes. Thus, the Collaboration class references the Process class. Collaborations has transitive dependencies to Core, Services, Correlations, Messaging and Artifacts. We created this metamodule in the initial horizontal decomposition. Further, we remove a redundant reference from Collaboration to Choreography. This reference was used to keep track of Choreographies that exist between the Processes of a Collaboration. As these Choreographies can also be found by iterating over all Choreographies and checking which Processes are involved, this utility reference can be replaced by a helper method. This decoupled Collaboration from Choreographies and broke the dependency cylce.

Choreographies Choreographies are used to define the interaction between Processes in a sequential way. Choreographies depends on Collaborations, as a Choreography is a subclass of Collaboration. Further, the the Participants of a Collaboration are referenced by the activities of a Choreography. Choreographies has transitive dependencies to Flows, Correlations, Messaging and Artifacts. We created this metamodule in the initial horizontal decomposition.

Conversations Conversations are used to give an overview of which participants (Pools) interact with each other. It is dependent on Collaborations because of several dependencies. A Conversation expresses the interplay between several participants; a participant is a class from Collaborations. A Conversation may refer to Collaborations between participants. Conversations is transitively dependent on Core, Correlations, and Messaging. We created Conversations in the scope of the big horizontal split. Instances of Conversation classifiers were originally contained in Collaborations. To decouple Collaborations from Conversations, we used dependency inversion and created ConversationContainer as a container for all conversation specific first-class concepts.

AuditingAndMonitoring BPMN2 does not define abstractions for the modeling of auditing and monitoring information. This metamodule encapsulates one specific extension

point for each of these two concepts. It is part of Δ, as Auditing and Monitoring are business process concepts. It depends on Flows, as we created a common superclass for Auditing and Monitoring classes there.

We extracted this metamodule into an extension, as Auditing and Monitoring are seldomly used optional features. As already mentioned, we introduced the new superclass FlowAnnotation in Flows, as a generic extension point for further extension of Flow elements. We replaced containments to the Auditing and Monitoring classes from Process and FlowElement by containments to FlowAnnotation. This decoupled Processes and Flows from AuditingAndMonitoring.

It would have also been possible to simply make Auditing and Monitoring inherit from RootClass. This would have reduced the complexity. However, this would also clutter the RootClass containment in Definitions.

Technically, these two classes are even redundant. Their purpose can also be fulfilled by using the extension mechanism that is defined in Externals. As the existence of Auditing and Monitoring does not violate the reference structure, we did factor them out instead of removing them, as this would have skewed the internal validity of the evaluation.

HumanInteraction HumanInteraction provides serveral types of Tasks that are performed by humans. It is therefore dependent on Activities, as the Task and GlobalTask classes are used as superclasses. HumanInteraction is transitively dependent on Core. We created this metamodule in the scope of the initial horizontal split. It was also horizontally split from the Resources.Human metamodule to separate resource and task specific concepts.

Feature Model Figure 2.13 shows the feature model of mBPMN2. All relations are required relations. Therefore, we have omitted the explicit required labels. As the mBPMN2 occupies only the π and Δ layer, the feature diagram consists only of the Δ layer. The non-abstract metamodules of the π layer resulted in Δ features.

Extensions and View Types are grouping features and are therefore mandatory. Processes, Choreographies, Conversations and Collaborations are view types. Resources and Human are no view types but extensions.

For the sake of clarity in the diagram, we do not show the feature model together with the metamodule diagram. The two grouping features Extensions and View Types do not have implementing metamodules. Neither has the root feature. The remaining feature nodes represent language features, are implemented by exactly one metamodule and are named like this feature.

Compared with the module diagram (see Figure 2.12), the number of features is less than the number of metamodules. This is the case, as many metamodules are abstract and many other metamodules are strongly coupled to them. The metamodules Core, Services, Correlations, Artifacts, Flows, Data, and Messaging are abstract and therefore do not implement language features. As mentioned in Section 2.4, by using dependency inversion, some of these metamodules could be turned into extensions (e.g., artifacts and messaging). This would result in further feature nodes in the feature model.

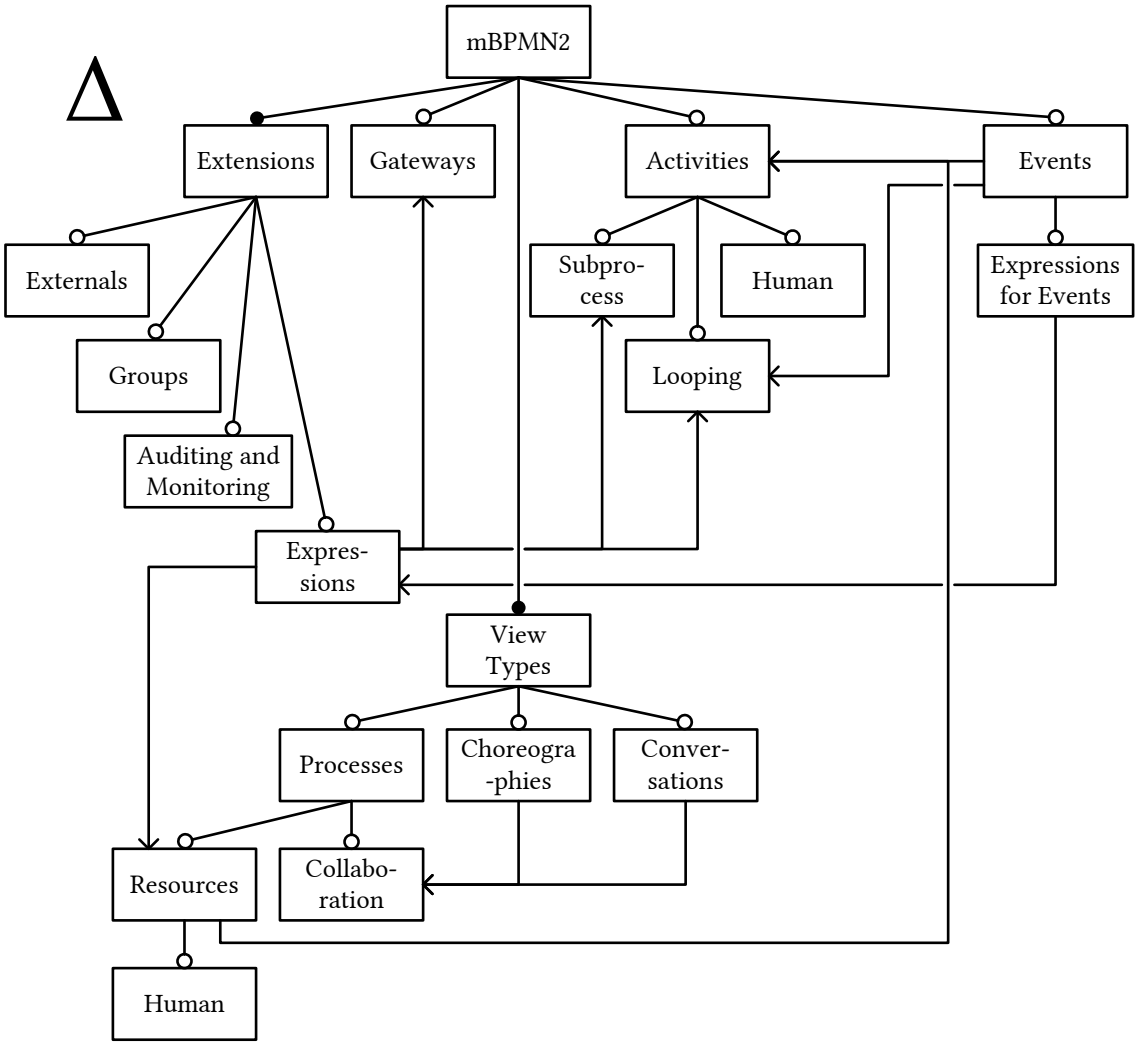

Figure 2.13: mBPMN2 Feature Model

3 Evaluation Tool

As the evaluation tool is a very special purpose tool, there is no update site for comfortable installation. The evaluation tool and its dependencies have to be installed manually. An advantage of the manual installation is the explicit control over the versions of the dependencies of the evaluation tool. This enables a more exact reproducibility of the evaluation setup.

Eclipse The evaluation tool and its dependencies are Eclipse plugins. We developed and used them with Eclipse Neon and Oxygen (4.7.2). We highly suggest using the Modeling Tools Package of Eclipse, as it provides many dependencies like the EMF.

AET All AET plugins have to be imported. These should be obtained from our fork[1].

Dependencies To get AET to compile, several plugins are required. The Generator Composition (GEKO) Framework has to be installed[2]. All Kieler Lightweight Diagrams have to be installed[3] (Ptolemy is not needed). The Xcore SDK and m2e (Maven Eclipse integration) have to be installed via the Eclipse releases update site. If any Maven errors occur, Tycho connectors have to be installed. Import all AET plugins.

Evaluation Tool All plugin projects have to be imported from the git repository[4].

Runtime Instance To enable the evaluation tool to read model files, the respective plugins also have to be imported that carry the metamodel and the model code. Finally, an inner eclipse instance can be started. Within this instance, the evaluation tool can be used.

[1] `https://github.com/MishaStrittmatter/architecture-evaluation-tool`
[2] `http://build.se.informatik.uni-kiel.de/eus/geco/snapshot`
[3] `http://rtsys.informatik.uni-kiel.de/~kieler/updatesite/release_pragmatics_2016-07/`
[4] `https://github.com/kit-sdq/Metamodel-Reference-Architecture-Validation`

4 Evaluation Data

4.1 Evolution Scenarios

In this section, we will present all evolution scenarios for the four case study metamodels. The scenarios are marked with their scenario type: extension^{+}, historical modification†, potential modification$^{\times}$ and generic modification$^{\circ}$. We will not explicitly mention the affected classes of generic modification scenarios, as they consist only of one affected class after which the scenario is named. In some scenarios it may seem that affected classes are missing. In these cases, one affected class is strongly coupled (e.g., by containment or inheritance) to the seemingly missing affected classes, so that these classes will be included in the relevant subgraph anyway.

4.1.1 Palladio Component Model

For the PCM, we collected two historical extension scenarios, elven historical modification scenarios and one potential evolution scenario. The extension scenarios for the PCM are optional extensions, i.e., they do not implement any core features of Palladio and are therefore not delivered with a standard installation of the PCM. The extension scenarios for the PCM are IntBIIS [3] and KAMP [9] (not to be confused with KAMP4aPS, which is a standalone DSML). We chose them because they are up-to-date and heterogeneous concerning the parts of the PCM they depend on. Figure 4.1 shows the module structure of the PCM and these two extensions.

The first extension is the *Integrated Business IT Impact Simulation* (IntBIIS) [3] for modeling and analyzing the performance of business processes and information systems. It consists of one metamodule, 16 classes and has 21 inter-module dependencies that target 11 classes of the PCM. It builds mostly on the user behavior defining parts of the PCM. Transferred to the mPCM, it depends on the metamodules Identifier, Base, Variables, Repository, and Usage (π and Δ). Its metamodel modules are located at the Δ, and Ω layers.

The second extension is the *Karlsruhe Architecture Maintainability Prediction* (KAMP) [9] for modeling modifications and analyzing their propagation on the software architecture level. It consists of three metamodel modules, 62 classes and has 42 inter-module dependencies that target 12 classes of the PCM. It builds on the structural parts of the PCM that belong to π and Δ. Transferred to the mPCM it depends on the metamodel modules Identifier, Base, Repository, Software Repository, Composition (π and Δ). Its metamodel modules are located at the Δ, Ω, and Σ layers.

We collected 11 historical modification scenarios for the PCM from its changelog[1]. We started with the most recent changes and selected the ones that actually changed the

[1] `https://sdqweb.ipd.kit.edu/wiki/PCM_Changelog`

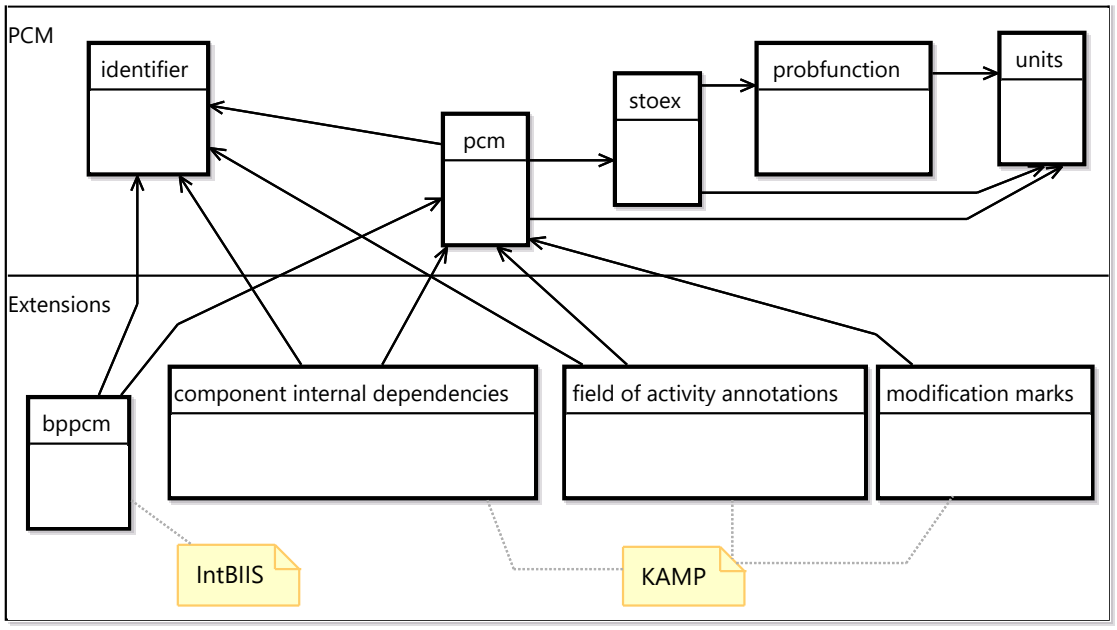

Figure 4.1: Metamodules of PCM extensions

structure of the metamodel and not just the genmodel, version numbers or namespaces. We skipped repeated modifications of the same classes. In addition, there was one proposed modification in the changelog, that we consider as a potential evolution scenario. Table 4.1 shows extension, historic and potential evolution scenarios and their respective affected classes. In the following, we present the evolution scenarios of the PCM.

The *AttributeTypes*† scenario changed types of attributes of NamedElement, Repository, ExternalCallAction, EntryLevelSystemCall from UML types to Ecore types. In the *CallAction*† modification scenario, the superclass of CallAction was changed from AbstractAction to Entity, as CallAction is not intended to be used as a stand-alone Action. The CallAction class is located in the behavior metamodel module of Δ. In the *ComLinkResType*† scenario, a supertype of CommunicationLinkResourceType (of the Resources metamodule) was changed to ResourceType (Resources) instead of ProcessingResourceType (Resources). The *LocalRoleConstraint*† scenario added OCL constraints, which check if the referenced roles belong to the component in which the calls/action is contained, to the classes InfrastructureCall, ResourceCall, and ExternalCallAction. The *MultiAllocation*× scenario aims to enable 1:n mapping of AssemblyContext to AllocationContext by changing the multiplicity of the respective reference. In the *ProcResSpec*† scenario, an inheritance relation was introduced from ProcessingResourceSpecification (Resources) to the Identifier class (Identifier). *Repository*† is a sub-scenario of the AttributeTypes scenario, where only the Repository was changed. The *ResourceDemandingBehaviour*† scenario made the ResourceDemandingBehaviour inherit from Identifier. The *ResSign*† scenario changed the multiplicity of the parameter Reference of the ResourceSignature class. The *SchedulingPolicy*† scenario removed the SchedulingPolicy Enum and created SchedulingPolicy class. In the *SyncPoint*†

Scenario Name	Affected Classes
IntBIIS^{+}	ScenarioBehaviour, CollectionDataType, NamedElement, Identifier, Entity, AbstractUserAction, PCMRandomVariable, DataType, OpenWorkload, CompositeDataType
KAMP^{+}	AssemblyConnector, DataType, RequiredRole, ProvidedRole, Entity, RepositoryComponent, Role, OperationProvidedRole, Interface, Signature, Connector, Identifier
AttributeTypes†	NamedElement, Repository, ExternalCallAction, EntryLevelSystemCall
CallAction†	CallAction, AbstractAction, Entity
ComLinkResType†	CommunicationLinkResourceType, ResourceType, ProcessingResourceType
LocalRoleConstraint†	InfrastructureCall, ResourceCall, ExternalCallAction
MultiAllocation$^{\times}$	AssemblyContext, AllocationContext
ProcResSpec†	ProcessingResourceSpecification, Identifier
Repository†	Repository
ResourceDemandingBehaviour†	ResourceDemandingBehaviour, Identifier
ResSign†	ResourceSignature
SchedulingPolicy†	SchedulingPolicy
SyncPoint†	SynchronisationPoint, Entity
UniqueCallTargets†	InfrastructureCall, ResourceCall, ParametricResourceDemand

Table 4.1: Evolution Scenarios of the PCM

scenario, a reference was created between the CallAction and the SynchronizationPoint classes. The *UniqueCallTargets*† scenario introduced OCL constraints, which check if the requested target is unique within the same action, to the InfrastructureCall, ResourceCall, and ParametricResourceDemand classes.

4.1.2 Smart Grid Topology

The Smart Grid Topology metamodel has been stable since its initial release, so we cannot deduce any modification scenarios from its version history. Following the scenario collection procedure, which we presented earlier, results in eight evolution scenarios (four potential and four generic). Table 4.2 shows the potential scenarios and their respective affected classes.

By the *AbstractType*$^{\times}$ scenario, an abstract superclass is set in place for all types in the TypeRepo. The *NewCommEntity*$^{\times}$ scenario introduces a new type of communicating device by adding a subclass to CommunicatingEntity. In the *NewPhysicalConn*$^{\times}$ scenario, an alternative to PhysicalConnection is created. As PhysicalConnection does not have an abstract superclass that would be eligible for inheritance, the root class SmartGridTopology

Scenario Name	Affected Classes
AbstractType$^{\times}$	Repository, NamedIdentifier, SmartMeterType, NetworkNodeType, ConnectionType
NewCommEntity$^{\times}$	CommunicatingEntity
NewPhysicalConn$^{\times}$	PhysicalConnection, SmartGridTopology
SmartMeter$^{\times}$	SmartMeter

Table 4.2: Evolution Scenarios of Smart Grid Topology (except Generic Scenarios)

has to also be modified. The *SmartMeter*$^{\times}$ scenario modifies the SmartMeter class by removing the aggregation attribute.

The scenarios Cluster, InputEntityState, OutputEntityState, and ScenarioResult are generic and therefore not shown in the table.

4.1.3 KAMP4aPS

For the KAMP4aPS case study, we collected 18 evolution scenarios (10 potential and eight generic). Table 4.3 shows the potential scenarios and their respective affected classes.

Scenario Name	Affected Classes
DocuApplication$^{\times}$	ComponentDocumentationFiles, InterfaceDocumentationFiles, StructureDocumentationFiles, ModuleDocumentationFiles
DocumentationFiles$^{\times}$	DocumentationFiles
FoAARepo$^{\times}$	FieldOfActivityAnnotationRepository, Entity
MechanicalAssembly$^{\times}$	MechanicalAssembly
Panel$^{\times}$	Panel, Component, ComponentRepository
ParentEntity$^{\times}$	Module, Interface, Entity
Plant$^{\times}$	Plant
Ramp$^{\times}$	Ramp, Component, MechanicalAssembly
Structure$^{\times}$	Structure, Plant
TurningTable$^{\times}$	TurningTable, Component, Module

Table 4.3: Evolution Scenarios of KAMP4aPS (except Generic Scenarios)

The scenario *DocuApplication*$^{\times}$ consists of removing the redundant container relation from all DocumentationFiles classes. In the *DocumentationFiles*$^{\times}$ scenario DocumentationFiles is changed from an interface to an abstract class. The *FoAARepo*$^{\times}$ scenario makes FieldOfActivityAnnotationRepository an Entity. In the *MechanicalAssembly*$^{\times}$ scenario the class MechanicalAssembly is moved into the MechanicalComponents package. The *Panel*$^{\times}$ scenario change the reference from the Panel class to Component to point to ComponentRepository. In the *ParentEntity*$^{\times}$ scenario the redundant or even dead reference to Entity is removed from Module and Interface. The *Plant*$^{\times}$ scenario adds structural features to Plan. For example, the redundant plantName attribute could be removed, as it is already

provided by its superclass. The *Ramp*$^{\times}$ scenario consists of moving the Ramp to the Component package and changing the superclass from MechanicalAssembly to Component, as the Ramp is not a mechanical element. In the *Structure*$^{\times}$ scenario, the redundant container relation is removed from the abstract Structure class. In the *TurningTable*$^{\times}$ scenario Component is added to the superclasses of the TurningTable class.

The following scenarios are generic. For reasons of space we had to shorten the names of some scenarios. In these cases, we put the name of the affected class into parentheses: Arm, ConveyorBelt, EtherCATSlave, HWPropagation (ChangePropagationDueToHardwareChange), ModifyMicroSwitch (ModifyMicroSwitchModule), ModifyModule, MonostableCylinder, SeedMods (KAMP4aPSSeedModifications).

4.1.4 BPMN2

The version jump from BPMN to BPMN2 (see [4]) was too big to extract any fine-grained historic modification scenarios. Also, the maturity and complexity of the metamodel made it hard to identify any potential modification scenarios. For the BPMN2 case study, we collected 23 generic evolution scenarios. These are ResAssignExp (ResourceAssignmentExpression), ComplBehDef (ComplexBehaviorDefinition), CorrSubscription (CorrelationSubscription), GlobBRuleTask (GlobalBusinessRuleTask), GlobChoreoTask (GlobalChoreographyTask), ParticipantAssoc (ParticipantAssociation), AdHocSubProc (AdHocSubProcess), ImplThrowEvent (ImplicitThrowEvent), InOutBinding (InputOutputBinding), ItemAwareElem (ItemAwareElement), Artifact, Auditing, BoundaryEvent, CategoryValue, FormalExpression, InteractionNode, LaneSet, ParallelGateway, PotentialOwner, Relationship, Rendering, RootElement, and SequenceFlow.

4.2 Models

To evaluate the *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, 30 KAMP4aPS models).

The number of BPMN2 models is much higher, because, in contrast to the other case studies, there is a public online repository with BPMN2 models[2]. For BPMN2, we have collected 103 models from internal sources and 3739 from the repository. From all these models, 46 models were invalid, could not be loaded and were therefore ignored by our analysis.

To remove potentially sensitive information from the models from internal sources, we preprocessed these models in the following way. We replaced file names by numbers. We censored model element names, labels, text annotations and documentation properties. This loss of information is irrelevant to the evaluation, as this information is not required. Instead, it is relevant which classes are instantiated. The resulting models can be found in the GitHub repository.

[2] https://github.com/camunda/bpmn-for-research/tree/1416f6f2104ae597eafa3097946140ebc2136a53

Bibliography

[1] Axel Busch, Robert Heinrich, Jörg Henss, Martin Küster, Sebastian Lehrig, Misha Strittmatter, Max Kramer, Erik Burger, and Ralf H. Reussner. "Architectural Viewpoints". In: *Modeling and Simulating Software Architectures – The Palladio Approach*. Ed. by Ralf H. Reussner, Steffen Becker, Jens Happe, Robert Heinrich, Anne Koziolek, Heiko Koziolek, Max Kramer, and Klaus Krogmann. Cambridge, MA: MIT Press, Oct. 2016. Chap. 3, pp. 37–73. URL: http://mitpress.mit.edu/books/modeling-and-simulating-software-architectures.

[2] Robert Heinrich, Sandro Koch, Suhyun Cha, Kiana Busch, Ralf Reussner, and Birgit Vogel-Heuser. "Architecture-based change impact analysis in cross-disciplinary automated production systems". In: *Journal of Systems and Software* 146 (2018), pp. 167–185. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2018.08.058. URL: http://www.sciencedirect.com/science/article/pii/S0164121218301717.

[3] Robert Heinrich, Philipp Merkle, Jörg Henss, and Barbara Paech. "Integrating business process simulation and information system simulation for performance prediction". In: *Software & Systems Modeling* 16.1 (2017), pp. 257–277. ISSN: 1619-1366. DOI: 10.1007/s10270-015-0457-1. URL: http://dx.doi.org/10.1007/s10270-015-0457-1.

[4] Object Management Group (OMG). *Business Process Model And Notation Specification (BPMN) – Version 2.0.2*. Jan. 2014. URL: http://www.omg.org/spec/BPMN/2.0.2/.

[5] Wolfgang Raskob, Valentin Bertsch, Manuel Ruppert, Misha Strittmatter, Lucia Happe, Brandon Broadnax, Stefan Wandler, and Evgenia Deines. "Security of Electricity Supply in 2030". In: *Critical Infrastructure Protection and Resilience Europe (CIPRE)*. Den Haag, Netherlands, Mar. 2015. URL: https://publikationen.bibliothek.kit.edu/1000056115.

[6] Ralf H. Reussner. "Parametrisierte Verträge zur Protokolladaption bei Software-Komponenten". PhD. Thesis. Department of Informatics, University of Karlsruhe, 2001.

[7] Ralf H. Reussner, Steffen Becker, Jens Happe, Robert Heinrich, Anne Koziolek, Heiko Koziolek, Max Kramer, and Klaus Krogmann. *Modeling and Simulating Software Architectures – The Palladio Approach*. Cambridge, MA: MIT Press, Oct. 2016. 408 pp. ISBN: 9780262034760. URL: http://mitpress.mit.edu/books/modeling-and-simulating-software-architectures.

[8] Ralf Reussner, Steffen Becker, Erik Burger, Jens Happe, Michael Hauck, Anne Koziolek, Heiko Koziolek, Klaus Krogmann, and Michael Kuperberg. *The Palladio Component Model*. Tech. rep. Karlsruhe: KIT, Fakultät für Informatik, 2011. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000022503.

[9] Kiana Rostami, Johannes Stammel, Robert Heinrich, and Ralf Reussner. "Architecture-based Assessment and Planning of Change Requests". In: *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures*. QoSA '15. Montreal, QC, Canada: ACM, 2015, pp. 21–30. ISBN: 978-1-4503-3470-9. URL: http://dl.acm.org/citation.cfm?id=2737198.

[10] Misha Strittmatter and Michael Langhammer. "Identifying Semantically Cohesive Modules within the Palladio Meta-Model". In: *Proceedings of the Symposium on Software Performance: Joint Descartes/Kieker/Palladio Days*. Ed. by Steffen Becker, Wilhelm Hasselbring, André van Hoorn, Samuel Kounev, and Ralf Reussner. Stuttgart, Germany: Universitätsbibliothek Stuttgart, Nov. 2014, pp. 160–176.