Karlsruhe Institute of Technology

A Graphical Approach to Modularization and Layering of Metamodels

Bachelor's Thesis of

Amine Kechaou

at the Department of Informatics
Institute for Program Structures and Data Organization (IPD)

Reviewer:	Prof. Dr. Ralf Reussner
Second reviewer:	Jun.-Prof. Dr.-Ing. Anne Koziolek
Advisor:	Dipl.-Inform. Misha Strittmatter
Second advisor:	Dr.-Ing. Erik Burger

16. May 2017 – 15. September 2017

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 14.09.2017

. . . .

(Amine Kechaou)

Abstract

Modularity is a key aspect in software engineering as it comes with several benefits like reusability, extensibility and maintainability. Although it is a well established concept, it has not received much attention when it comes to model-driven software development. Over time, metamodels tend to evolve and grow in complexity to encompass new aspects and features. If modularization steps are not taken and metamodels are extended intrusively, they can become difficult to maintain and to extend. With the increased complexity, the modularization can become even more challenging.

In this work, we present a novel approach to assist the modeler in the task of modularization. Our approach addresses the problem from a graphical perspective. The proposed tool support displays a layered structure, where each layer has certain level of abstraction, and allows the modeler to organize metamodels inside the layers. In this layered structure, the metamodels should only depend on metamodels with the same or a higher abstraction level and should not take part in cyclical dependencies. The tool provides the modeler with full control over the modularization process and full knowledge about the relations between the metamodels, thus facilitating the modularization task greatly.

Zusammenfassung

Modularität ist ein wichtiger Aspekt der Softwaretechnik denn sie bietet mehrere Vorteile wie Wiederverwendbarkeit, Erweiterbarkeit und Wartbarkeit. Obwohl sie ein weit etabliertes Konzept ist, wurde sie in der modellgetriebenen Software-Entwicklung kaum untersucht. Im Laufe der Zeit wachsen Metamodelle und damit auch ihre Komplexität, um neue Aspekte zu umfassen. Wenn die Modularität nicht berücksichtigt ist und Metamodelle intrusiv erweitert werden, leidet ihre Wartbarkeit und Erweiterbarkeit. Mit der höheren Komplexität wird die Modularisierung selbst auch schwieriger.

In dieser Arbeit stellen wir einen neuartigen Ansatz vor, um den Modellierer bei der Metamodell-Modularisierung zu unterstützen. Unser Ansatz geht aus einer graphischen Perspektive vor. Das entwickelte Werkzeug zeigt eine geschichtete Struktur, wobei jede Schicht einen Abstraktionsgrad hat, und erlaubt den Modellierer Metamodelle innerhalb von den Schichten zu gestalten. Dabei dürfen Metamodelle nur von Metamodellen mit gleichem oder höherem Abstraktionsgrad abhängen und keine zyklische Abhängigkeiten aufweisen. Das Werkzeug gewährt dem Modellierer volle Kontrolle bei der Modularisierung und stellt alle benötigte Informationen über die Beziehungen zwischen den Metamodellen zur Verfügung, um die Modularisierung deutlich zu erleichtern.

Contents

List of Figures

1 Introduction

At the turn of the century and during the last decade, Model-Driven Software Development (MDSD) has received great attention. Its proponents usually propound a faster development and an increased software quality due to the raised level of abstraction that it provides since models are a central part in the development process [20]. With an increased complexity in the software system and the underlying metamodels, modularity becomes more and more crucial as it enables better maintainability, reusability and extensibility.

For instance, the increase in complexity in the Palladio Component Model (PCM) [10] expressed the need for a modular PCM [15]. Modularization efforts led to the conception of a layered reference structure for modular metamodels [14]. Metamodel modularization based on this structure yields sub-metamodels, or modules, that are assigned to the layers of the structure where each layer has a certain level of abstraction. The premise of the layered structure is that modules should only depend on modules from the same layer or a more abstract one and that cyclical dependencies between modules are forbidden.

However, metamodel modularization is often a challenging task since the modeler needs to have a full grasp and a global understanding of the dependencies between the different modules. Unfortunately, such requirements are barely addressed and are not fulfilled by the current tooling.

In this work, we present the MRS graphical editor, a graphical tool support for the proposed modular reference structure. Its purpose is to provide a graphical representation of the layered structure and the metamodels that it contains as well as to assist the modeler in the modularization by providing substantial information about the module dependencies and by automatically detecting the dependencies that violate the premise of the modular reference structure.

In the next chapter, we take a step back to lay down the foundations of our work. The concept behind our tool and insights about its implementation are presented respectively in chapter 3 and chapter 4. We evaluated our work by using the tool with existing metamodels in order to validate its correctness and applicability. The results of the conducted evaluation are the subject of chapter 5. Finally, we discuss related work and other modularization tools in chapter 6 and limitations of the MRS graphical editor in chapter 7 where we also give a glimpse into possible future work.

2 Foundations

2.1 Model-Driven Software Development

Model-Driven Software Development (MDSD) can be viewed as an application of the concepts of Model-Driven Engineering (MDE) to software development. The main aspect of MDE is the combination and the use of Domain-Specific Modeling Languages (DSML) and model transformations [13]. In a broader sense, a Domain Specific Language (DSL) can be unterstood as *a computer programming language of limited expressiveness focused on a particular domain* [4]. The abstract syntax of DSML is expressed through an underlying *metamodel*. The metamodel is a model of a model; that is, it describes models.

In MDSD, models do not just describe the software, but are integral part of it. Model and code coexist and evolve jointly and models are artifacts of the software [20]. Proponents of MDSD usually argue for a faster development process and a better code quality and manageability due to an increased level of abstraction. The promises made by model-driven approaches and the prominance of the Unified Modeling Language (UML) in the 90s led to the emergence of the Model-Driven Architecture (MDA), a standardization initiative by the Object Management Group (OMG) [3, 1]. In this respect, the OMG introduced the MetaObject Facility (MOF) meta-metamodel as a standard meta-metamodel that is meant to be aligned with the UML specification.

2.2 Modularity

Modularity is a well established concept in the realm of software development. It describes the degree to which a software is decomposed in separate *modules* that are to the most extent loosely coupled, so that changes on one module affects as little as possible other modules. Modularity comes with huge benefits. Parnas puts these in three categories [8]:

- Managerial: faster development and lesser communication between teams
- Product flexibility: The effect of changes on one module is minimal on others
- Comprehensibility: The system can be studied by studying one module at a time

Modularity enables key aspects like reusability, extensibility and maintainability [7]. Therefore, it is crucial to design a software system in a modular way.

These concepts usually hold for DSLs and metamodels too. Metamodels also tend to grow more and more complex to define more aspects than originally intended. Thus, the need for modularity is as much significant.

2.3 A Layered Structure for Modular Metamodels

Over the years, the Palladio Component Model (PCM) [11, 10] has grown in complexity. Many extensions were added to it, but usually in an intrusive and non-modular way. A need for modularization was expressed [15] and has led to the conception of a layered structure for modular metamodels [14]. Dubbed a modular reference structure for Component-based Architecture Description Languages, its premise is to organize metamodels, i.e. *modules*, inside this layered structure based on the following principles:

- Layers at the top hold more abstract metamodels, those at the bottom more specific ones.
- A metamodel can only depend on metamodels from the same layer or a more abstract one.
- Cyclical dependencies between metamodels are forbidden either inside a layer or across layers

In their work, Strittmatter et al. propose four different layers:

- Paradigm: the most abstract layer, holds basic structure but without semantics
- Domain: extends paradigm and adds domain semantics
- Quality: defines quality
- Analysis: provides concepts of analysis, solving or simulation.

The proposed layers are not strictly prescribed by the modular reference structure. They can be further divided and other layers can be introduced.

2.4 Eclipse Modeling Framework (EMF)

EMF is an eclipse framework that adds modeling capabilities to the Eclipse environment like describing models and code generation. At the core of EMF, the Ecore metamodel is the metamodel which is used to describe EMF models. Ecore is itself an EMF model, which makes it a meta-metamodel. It is equivalent to the Essential MOF (EMOF) specification by the OMG. In fact, the work on Ecore has greatly influenced OMG's MOF, so that it led to the specification of EMOF as being a subset of the Complete MOF (CMOF)[1]. Our work focuses on Ecore-based metamodels, but the idea behind it should also apply to EMOF metamodels.

Multiple extensions and tools build on EMF. One of these is EMF Profiles, an extension mechanism for EMF models [6]. EMF Profiles define the *emfprofile* metamodel that extends the Ecore Metamodel. The metamodel defines three classes: `Profile`, a subclass of Ecore's `EPackage`, `Stereotype`, a subclass of Ecore's `EClass` and `Extension`. A profile is basically a container for `Stereotypes` that extend `EClasses`, either by referencing another `EClass` or through so-called *tagged values*.

[1]`http://www.omg.org/ocup-2/documents/Meta-ModelingAndtheMOF.pdf`

2.5 Sirius

Sirius is an Eclipse based framework that allows users to create graphical editors for their metamodels by leveraging technologies like EMF and the Graphical Modeling Framework (GMF)[2].

Sirius comprises two main parts: the Sirius Tooling and the Sirius Runtime[3]. The former is used by *architects* to specify the graphical editors, that is, which elements from the metamodel are graphically represented, how do they look and how do they behave. The latter is used by *end users* to use already defined graphical editors.

The Sirius Runtime contains two metamodels: the representation metamodel and the description metamodel. The representation metamodel is the metamodel that defines the diagram elements. The description metamodel defines the model that the architect creates in order to specify a graphical editor. The Sirius Runtime takes as input both the modeler description model and the business model, computes the representation model and renders the diagram using GMF.

[2]https://www.eclipse.org/community/eclipse_newsletter/2017/june/article4.php

[3]https://www.eclipse.org/sirius/doc/developer/Architecture_Overview.html

3 Metamodel Modularization with the MRS Graphical Editor

3.1 Motivating Example

Over the years, the PCM has evolved to encompass new aspects and features such as additional software quality attributes [15]. Various extensions were added to the PCM but that was usually done in an intrusive manner. Such extensions weren't added as loosely coupled external modules, but the PCM itself was modified to express the new features.

In this section, we are going to showcase how inconvenient it could become to modularize an Ecore-based metamodel just by using the EMF tree editor. We do that by extracting the reliability extension [2] from the PCM into a separate metamodel called *pcmReliability*. The goal is to have two distinct metamodels *pcm* and *pcmReliability* so that *pcm* does not depend on *pcmReliability*. The steps described in [14] are going to serve as an orientation in what follows.

We first begin by importing the PCM[1] into the workspace in Eclipse and creating a new Ecore modeling project called *pcmReliability* which is going to hold the new reliability metamodel.

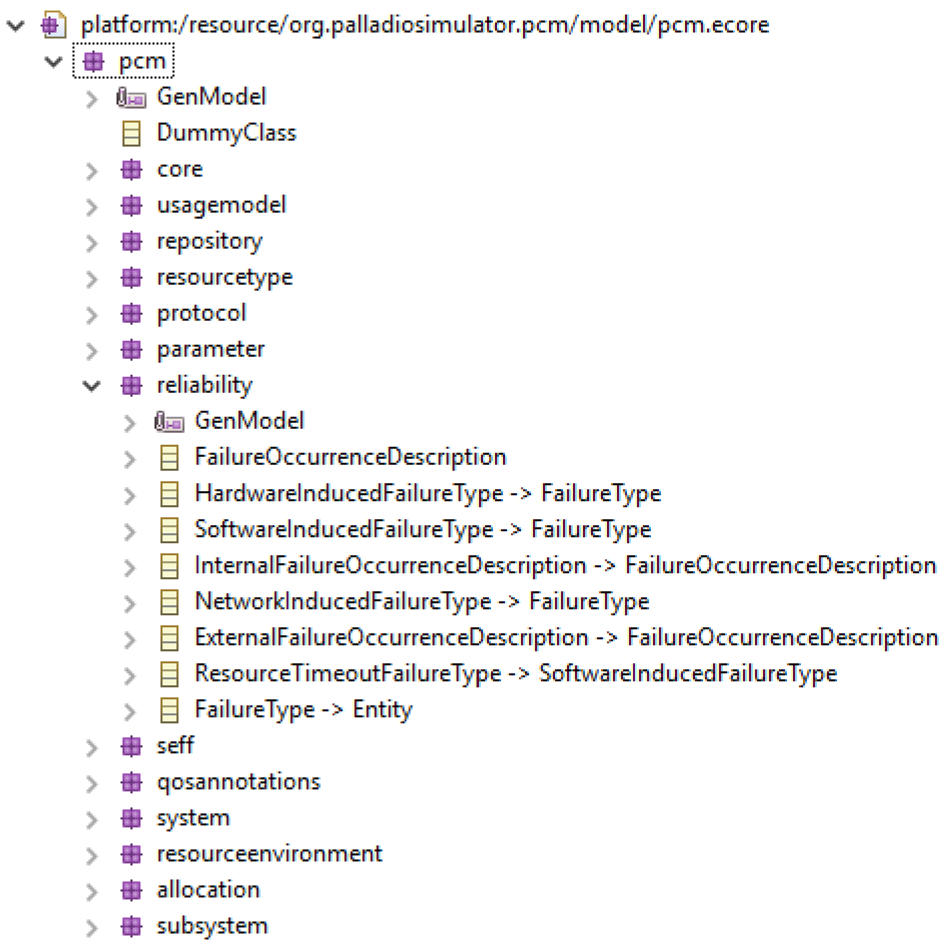
The PCM already contains the subpackage reliability which holds the foundations of the reliability extension (Figure 3.1).

However, the PCM has been enriched with reliability attributes in various other parts. These attributes usually take the form of references to `FailureOccurenceDescription` and `FailureType` or their subclasses. We focus here on `FailureOccurenceDescription` and related classes and extract these into *pcmReliability*. Classes related to `FailureType` should also be extracted into the new reliability module, as it is the case in mPCM, a modular prototype for PCM[2], or even form their own metamodel (Figure 3.2). In our example, we leave them in *pcm* to keep the example simple.

We first proceed by moving `FailureOccurenceDescription`, `InternalFailureOccurrence-Description` and `ExternalFailureOccurrenceDescription` to *pcmReliability* (Figure 3.3). Without much knowledge of the internal structure of the PCM, we immediately notice that there is no way to determine outgoing dependencies from *pcm* to *pcmReliability*. We could delete the classes of *pcmReliability*, validate the metamodel and see where did validation errors occur, but that would be a quite unorthodox and inconvenient way to modularize metamodels. Looking through all classes of *pcm* would also be too much time consuming.

[1]`https://svnserver.informatik.kit.edu/i43/svn/code/Palladio/Core/trunk/PCM/org.palladiosimulator.pcm/`

[2]`https://sdqweb.ipd.kit.edu/wiki/Palladio_Component_Model/Modular_Prototype`



```
platform:/resource/org.palladiosimulator.pcm/model/pcm.ecore
  pcm
    GenModel
    DummyClass
    core
    usagemodel
    repository
    resourcetype
    protocol
    parameter
    reliability
      GenModel
      FailureOccurrenceDescription
      HardwareInducedFailureType -> FailureType
      SoftwareInducedFailureType -> FailureType
      InternalFailureOccurrenceDescription -> FailureOccurrenceDescription
      NetworkInducedFailureType -> FailureType
      ExternalFailureOccurrenceDescription -> FailureOccurrenceDescription
      ResourceTimeoutFailureType -> SoftwareInducedFailureType
      FailureType -> Entity
    seff
    qosannotations
    system
    resourceenvironment
    allocation
    subsystem
```

Figure 3.1: Tree editor view of pcm.ecore

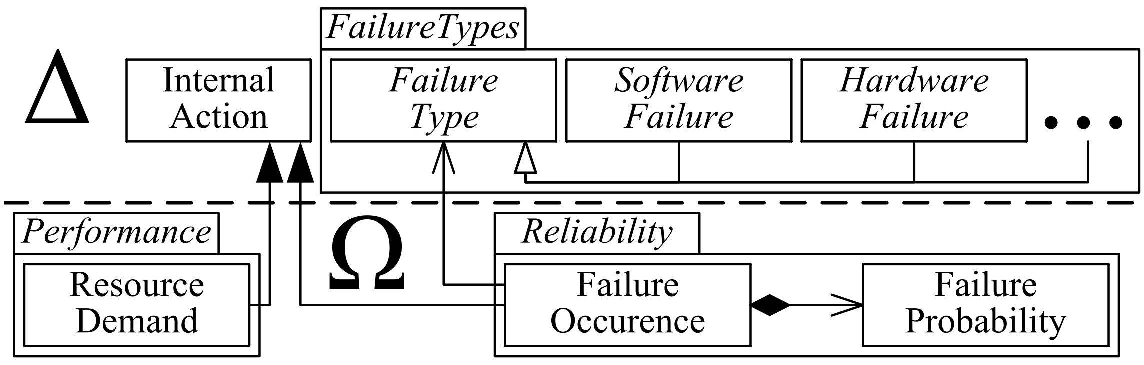


Figure 3.2: Excerpt of the Ω (Quality) Layer in the modular PCM view [14]

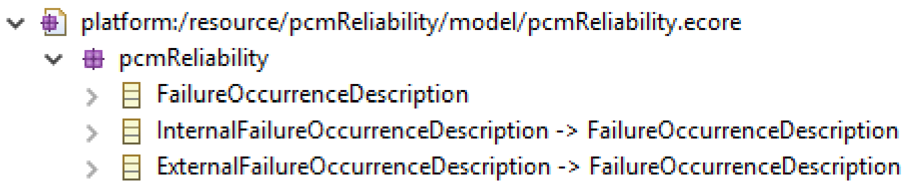

```
platform:/resource/pcmReliability/model/pcmReliability.ecore
  pcmReliability
    FailureOccurrenceDescription
    InternalFailureOccurrenceDescription -> FailureOccurrenceDescription
    ExternalFailureOccurrenceDescription -> FailureOccurrenceDescription
```

Figure 3.3: *pcmReliability* after the first modularization attempt

3.2 Concept

In addition to the problems addressed in the previous section, the EMF Tree Editor obviously doesn't support the proposed modular reference structure (Section 2.3). It doesn't offer a direct and explicit way to organize metamodels inside the layered structure. This can only be done in a purely logical manner, for example by changing the name of the metamodel so that it holds information about which layer it is assigned to. Moreover, the user has no direct knowledge of whether the conditions of the modular reference structure are held or not — that is, whether a metamodel references a metamodel from a lower layer or a cyclical dependency exists between metamodels.

For all these reasons, we have designed the MRS graphical editor with two main goals: it has to offer a graphical support for the modular reference structure and has to facilitate modularization operations. The tool is intended to be used mainly for two purposes: either to design a modular metamodel from scratch or to modularize and refactor an existing metamodel.

The MRS graphical editor offers a visual representation of a layered structure. This is a global container that can be subdivided into layers. Layers in their turn contain the metamodels, which are represented by smaller containers. The tool offers then a visualization of the relations between the metamodels. If some class from a metamodel references a class from another metamodel, an edge is drawn from the source metamodel to the target metamodel thus indicating that the source metamodel depends on the target metamodel. The same is also done if a stereotype extends a class from a metamodel with a class from another metamodel.

The tool also assists the modeler in the modularization in various ways. It warns him about possible violations of the conditions of the modular reference structure by highlighting the problematic edges in a different color. These are edges that take part in cycles and edges that go from a metamodel to another one in a lower layer. Furthermore, the tool facilitates modularization by delivering substantial information about the dependencies between metamodels, i.e., the classes that are involved and the nature of the dependency (e.g., a reference).

3.3 User Guide

To install the MRS graphical editor, please follow the instructions below:

1. Install Eclipse Oxygen Modeling Tools[3].
2. Install EMF Profiles from the update site[4].
3. Install the MRS graphical editor from the update site at [5].

Please note that the source code of the MRS graphical editor is available at `https://github.com/kit-sdq/MRS-Editor`. If you'd like to use the tool without installing it, but rather by running it from the source code, follow steps 1 and 2 then import the *mrs*, *mrs.edit*, *mrs.editor* and *mrs.custom* projects into the workspace. After that, launch an Eclipse runtime instance and import *mrs.design* in the inner workspace.

To use the MRS graphical editor, proceed as follows:

1. Create a modeling project
2. Create a MRS Model in the project (New > Other > Mrs Model) and choose "Modular Reference Structure" as the Model Object
3. Select the MRS Viewpoint (right-click on the project > Viewpoints Selection)
4. Open the representation of the Modular Reference Structure element if it was not automatically opened. For a detailed description of the capabilities of the MRS graphical editor, please refer the usage scenario provided in the next chapter.

[3]`https://www.eclipse.org/downloads/packages/eclipse-modeling-tools/oxygenr`

[4]`http://www.modelversioning.org/emf-profiles-updatesite`

[5]`https://sdqweb.ipd.kit.edu/eclipse/mrs-editor/nightly/`

4 Implementing the MRS Graphical Editor

4.1 The MRS Metamodel

The MRS graphical editor is a Sirius-based graphical editor for MRS models, that also enables the manipulation of metamodels. An MRS model is in fact a model that conforms to the MRS metamodel, an Ecore-based metamodel that we designed during this work to depict the main aspects of the modular reference structure. As shown in Figure 4.1, the MRS metamodel is composed of three `EClasses`: `ModularReferenceStructure`, `Layer` and `Metamodel`. The root object of an MRS model is an instance of `ModularReferenceStructure`. It can contain several `Layers`, which in their turn can contain several `Metamodels`. Other than containing layers, the `ModularReferenceStructure` also references the loaded EMF Profiles through the *loadedProfiles* `EReference`.

The `Metamodel EClass` holds a reference *mainPackage* to an `EPackage`, which is the main `EPackage` of the actual metamodel being represented. The *visibleEClassifiers* and *visibleEPackages* are references that are used to indicate which classes and packages from the target metamodel are being displayed. This is particularly useful for the user to only show the classes that he is interested in.

4.2 Usage Scenario

In what follows, we describe the MRS graphical editor as seen from a user perspective. Figure 4.2 shows the editor just after creating a fresh MRS model. As to how to create an MRS model, a thorough explanation is given in the user guide in Section 3.3. At the start, the diagram includes an empty container called *Modular metamodel*, which is an instance of `ModularReferenceStructure`. In this container several layers can be added via the *Add Layer* palette entry. As stated earlier, a layer can in its turn contain several metamodels. These can be essentially added in three different ways:

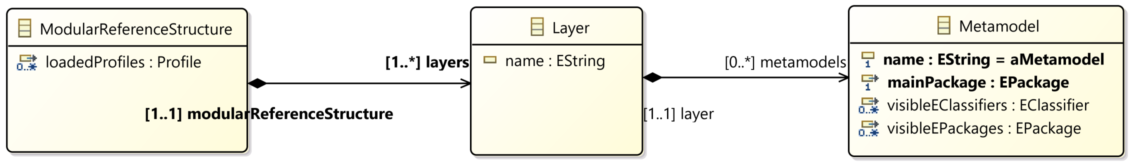

ModularReferenceStructure
loadedProfiles : Profile
[1..*] layers
[1..1] modularReferenceStructure
Layer
name : EString
[0..*] metamodels
[1..1] layer
Metamodel
name : EString = aMetamodel
mainPackage : EPackage
visibleEClassifiers : EClassifier
visibleEPackages : EPackage

Figure 4.1: Representation of the MRS metamodel

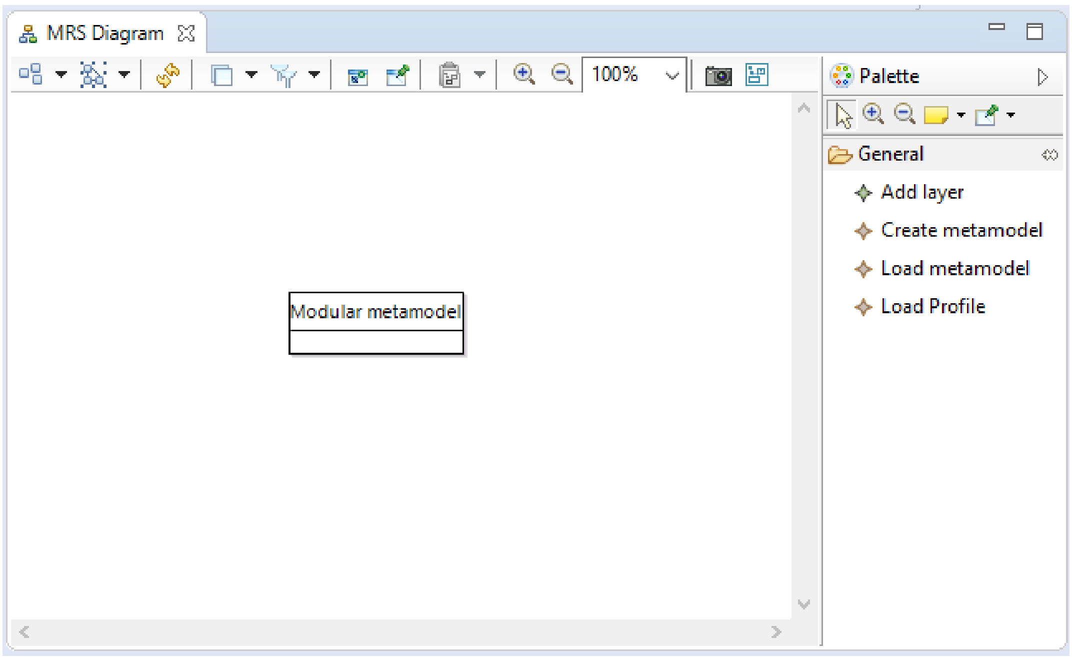


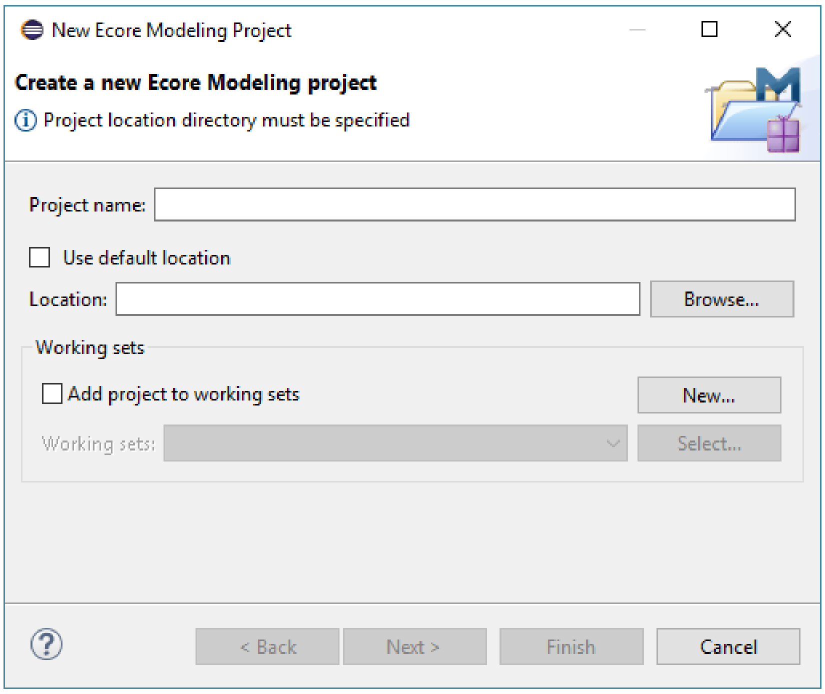
Figure 4.2: The MRS graphical editor with a blank MRS Model

a) ***Create Metamodel:*** This palette entry triggers the Ecore modeling project creation wizard (Figure 4.3) and allows the user to create an Ecore-based metamodel. It then creates a corresponding instance of the `Metamodel` `EClass` and adds it to the selected layer.

b) ***Load Metamodel:*** This palette entry prompts the user to select an Ecore file from the workspace (Figure 4.4) and adds to the chosen layer a new `Metamodel` that references the selected Ecore metamodel through the *mainPackage* reference.

c) ***Load Profile:*** Here, the user is also prompted to select a resource from the workspace. But this time, he is prompted to select a *.emfprofile_diagram* file. The referenced `EClasses` in the selected EMF Profile are then inspected and their containing metamodels are added to the chosen layer. For the sake of example, consider the simplistic EMF Profile *MyProfile* provided in Figure 4.5. It contains a stereotype called *MyStereotype*, which extends a class `A` with a class `B`. Classes `A` and `B` are contained respectively in metamodels *Metamodel1* and *Metamodel2*. When loading this profile, both metamodels are automatically added to the chosen layer (Figure 4.6). They are also linked by an edge going from *Metamodel2* to *Metamodel1* indicating that *Metamodel2* extends *Metamodel1*. The label on the edge indicates that this extension is provided by the stereotype *MyStereotype* in *MyProfile*.

Now that we have described the actions that can be taken by the user via the palette, we are able to present the various capabilities of the MRS graphical editor, that make metamodel modularization easier and more convenient. These include:



Create a new Ecore Modeling project

Project location directory must be specified

Project name:

Use default location

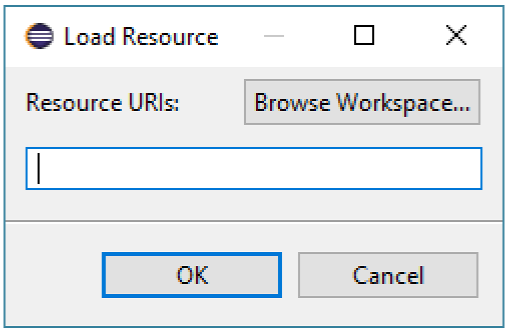
Location:

Working sets

Add project to working sets

Working sets:

Figure 4.3: The Ecore modeling project wizard



Resource URIs:

Figure 4.4: The Load Resource dialog

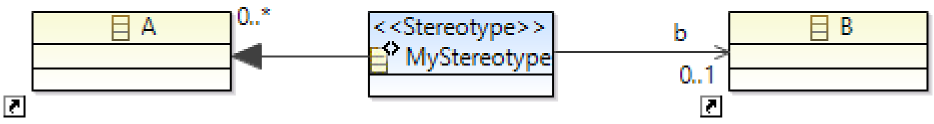


Figure 4.5: Simplistic EMF Profile

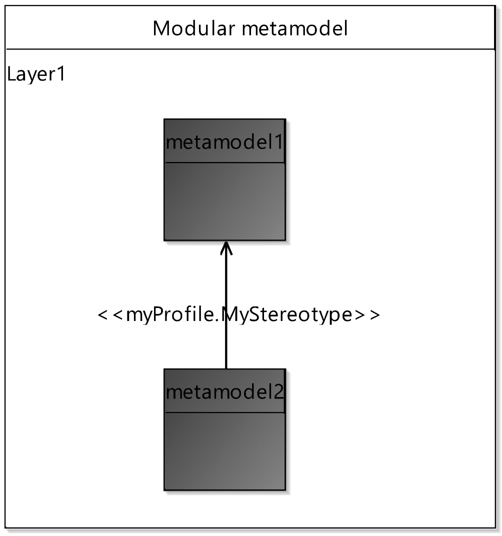


Figure 4.6: Metamodels binded by a stereotype

- Detection of dependencies that violate the modular reference structure
- Providing substantial information about the nature of dependencies: which classes are involved in the dependency and in what way
- Moving classes easily between metamodels by drag and drop

At this point, a closer definition of the term *dependency* in the context of the MRS graphical editor and Ecore-based metamodels is required. In a general sense, metamodel *A depends* on metamodel *B*, i.e. a *dependency* from *A* to *B* exists, if a loaded EMF profile contains a stereotype that extends a class from *A* and references a class from *B* or if *A* contains a `EClass` that references a `EClassifier` in *B* through the *ESuperTypes* reference, the type of a `EReference`, the return type or parameters' type of a `EOperation` or a generic type.

To demonstrate the suggested capabilities, we resume with the example provided in Section 3.1. We first proceed by loading the PCM metamodel through *Load Metamodel*. In Figure 4.7 we see that the PCM (white rectangle) and several other metamodels (gray rectangles) are added to *Layer1*. These are the metamodels on which the PCM depends and are added automatically by the MRS graphical editor. Two types of metamodels are to be distinguished:

- Metamodels represented by **white rectangles** are metamodels currently present in the workspace
- Metamodels represented by **gray rectangles** are metamodels present in the target platform (e.g. loaded plugins)

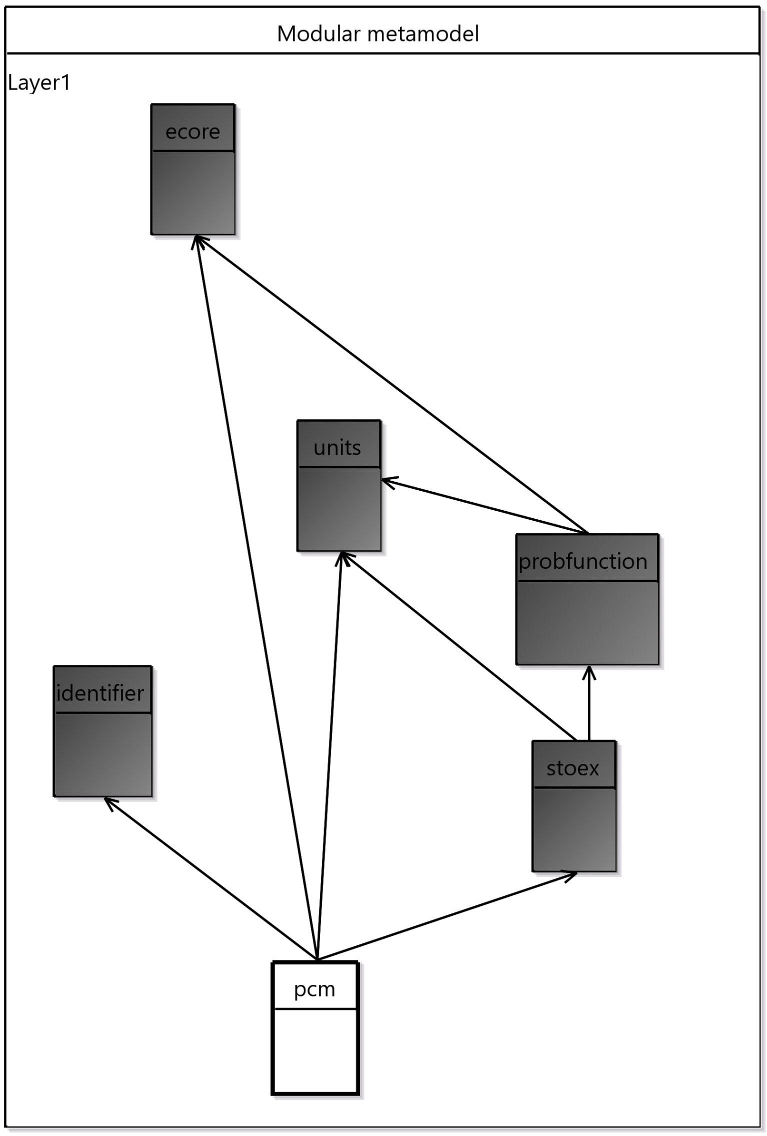


Figure 4.7: The MRS diagram after loading PCM

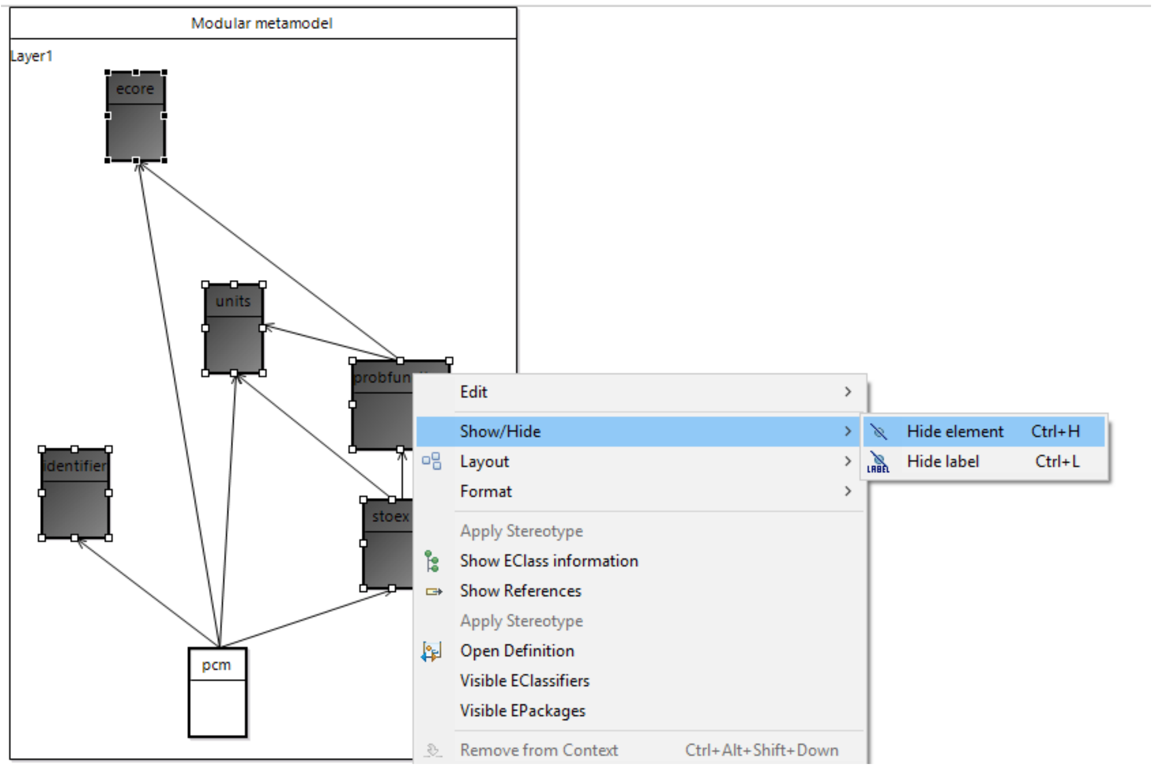


Figure 4.8: Show/Hide elements from the diagram

Although it is nice to have referenced metamodels as well automatically loaded, it can be the case that those are not needed to be dealt with. For this matter, Sirius offers a way to hide unwanted elements from the diagram. This can be achieved either via the element's context menu (*Show/Hide > Hide element*) or via the diagram's context menu (*Show/Hide*) and unselecting the unwanted elements (Figure 4.8). This only hides the elements from the diagram and doesn't actually delete them from the model.

Just as we did in Section 3.1, we also create here a new metamodel called *pcmReliability*, but this time through the *Create Metamodel* action in the MRS graphical editor. For now, since we are working with only one layer, *pcm* and *pcmReliability* will be in the same layer (Figure 4.9).

When loaded to the diagram, the contents of metamodels are by default hidden. This is why the *pcm* seems to be an empty container. The contents of a metamodel, i.e. `EClassifiers` and subpackages, can be shown via the context menu's entries *Visible EClassfiers* and *Visible EPackages*. The *reliability* subpackage is selected to be the only visible subpackage in *pcm* (Figure 4.10). All classes of the *reliability* subpackage are then selected to be visible (Figure 4.11). The result can be seen in Figure 4.12.

Upon moving `FailureOccurenceDescription`, `InternalFailureOccurrenceDescription` and `ExternalFailureOccurrenceDescription` to *pcmReliability* by ***drag and drop***, the MRS graphical editor immediately detects a cyclical dependency between *pcm* and *pcmReliability* and the identified cycle takes a red color (Figure 4.13).

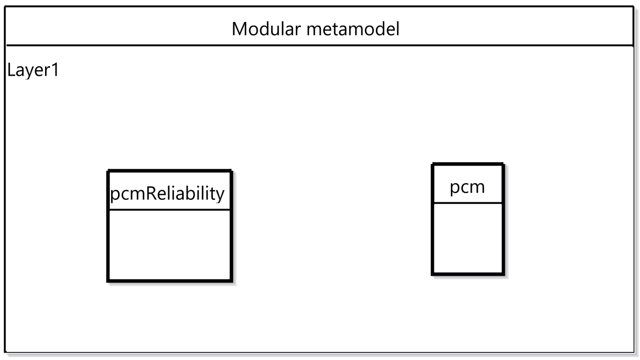


Figure 4.9: *pcm* and *pcmReliability*

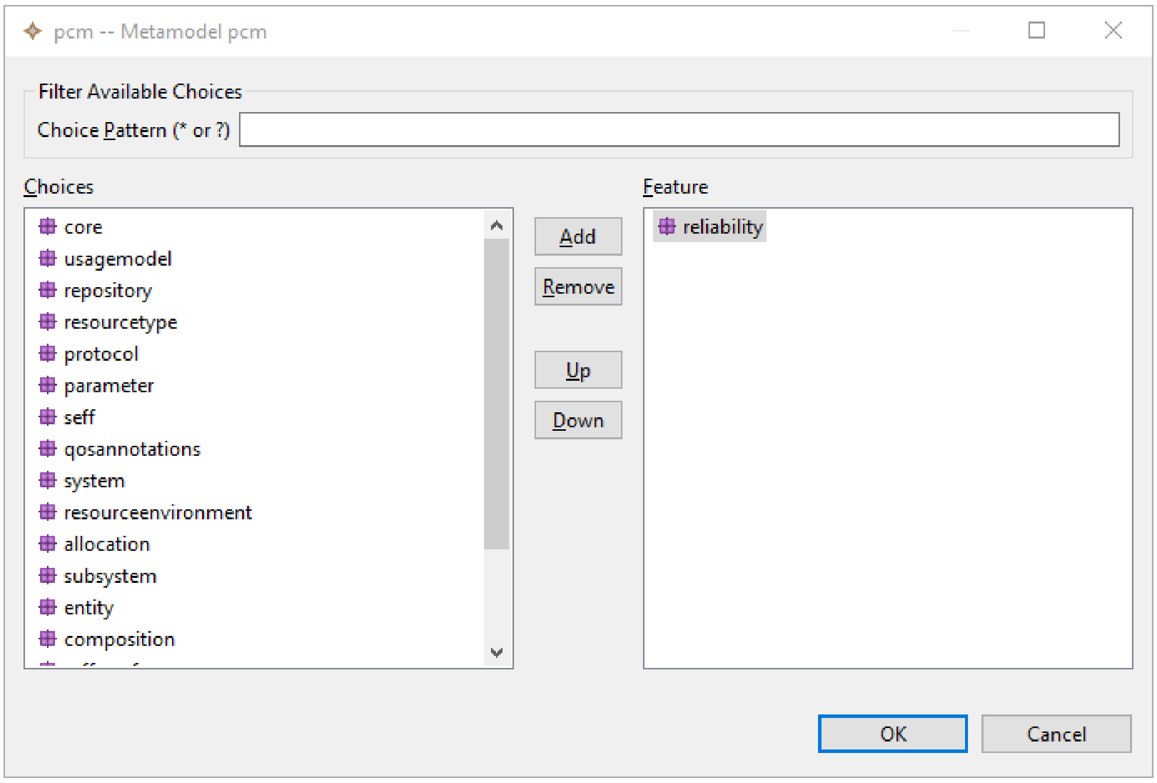
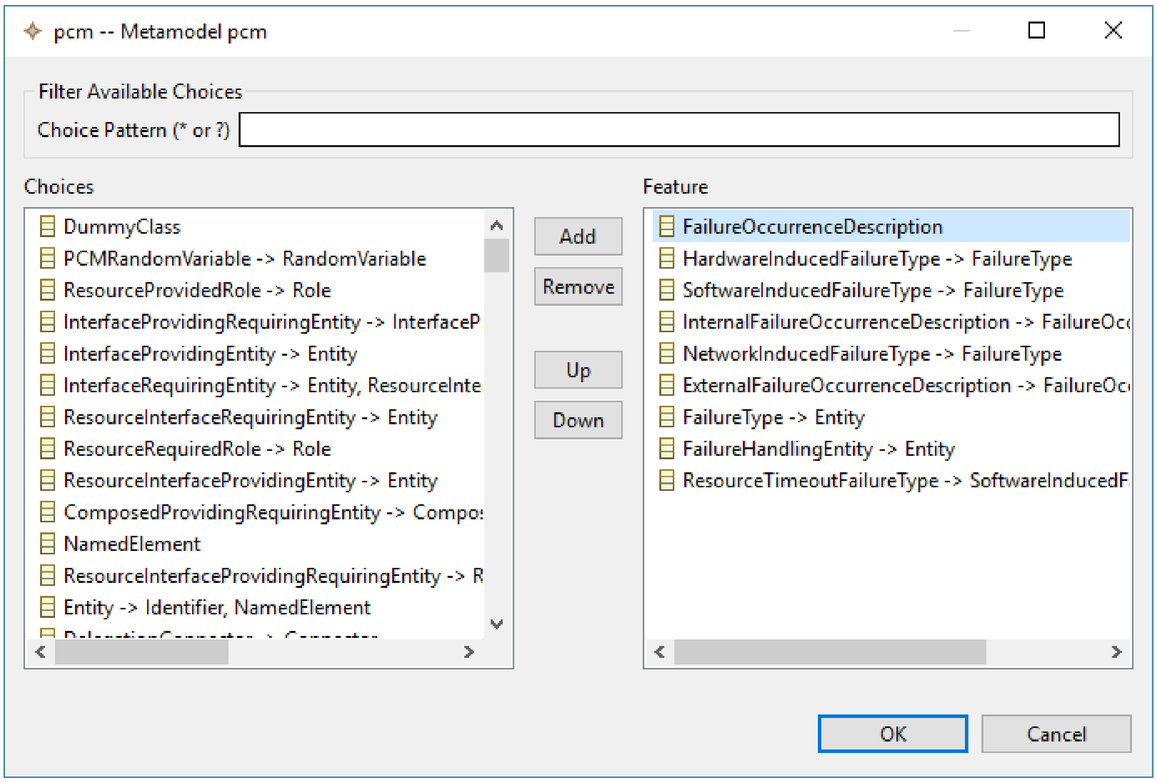


Figure 4.10: Making the *reliability* subpackage visible


pcm -- Metamodel pcm

Filter Available Choices

Choice Pattern (* or ?)

Choices
- DummyClass
- PCMRandomVariable -> RandomVariable
- ResourceProvidedRole -> Role
- InterfaceProvidingRequiringEntity -> InterfaceP
- InterfaceProvidingEntity -> Entity
- InterfaceRequiringEntity -> Entity, ResourceInte
- ResourceInterfaceRequiringEntity -> Entity
- ResourceRequiredRole -> Role
- ResourceInterfaceProvidingEntity -> Entity
- ComposedProvidingRequiringEntity -> Compos
- NamedElement
- ResourceInterfaceProvidingRequiringEntity -> R
- Entity -> Identifier, NamedElement

Feature
- FailureOccurrenceDescription
- HardwareInducedFailureType -> FailureType
- SoftwareInducedFailureType -> FailureType
- InternalFailureOccurrenceDescription -> FailureOcc
- NetworkInducedFailureType -> FailureType
- ExternalFailureOccurrenceDescription -> FailureOc
- FailureType -> Entity
- FailureHandlingEntity -> Entity
- ResourceTimeoutFailureType -> SoftwareInducedF

Figure 4.11: Making all classes of the *reliability* subpackage visible

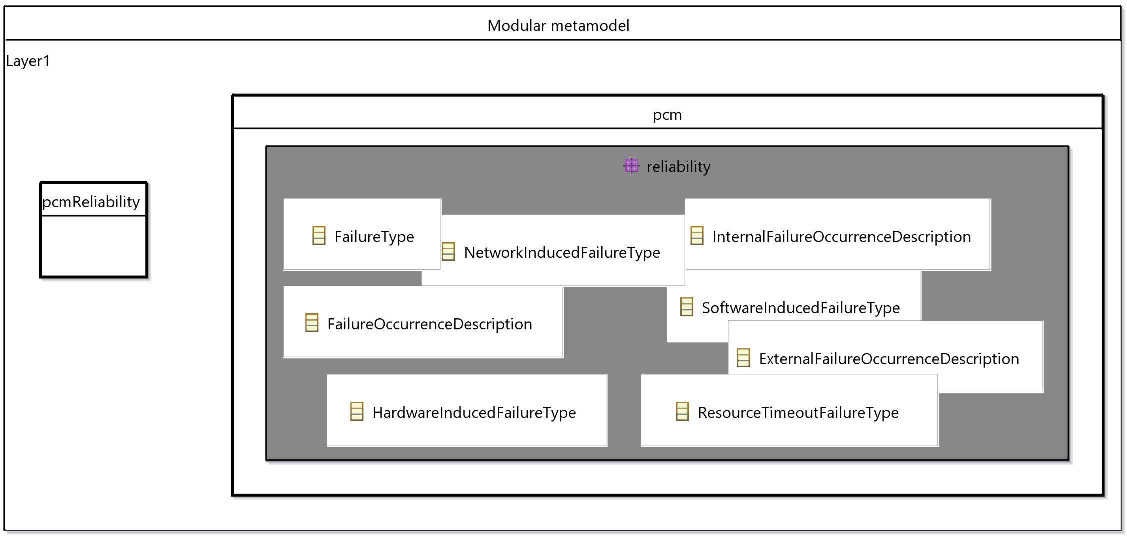

Modular metamodel
Layer1
pcmReliability
pcm
reliability
FailureType
NetworkInducedFailureType
InternalFailureOccurrenceDescription
SoftwareInducedFailureType
FailureOccurrenceDescription
ExternalFailureOccurrenceDescription
HardwareInducedFailureType
ResourceTimeoutFailureType

Figure 4.12: The diagram after making the *reliability* subpackage and its contents visible

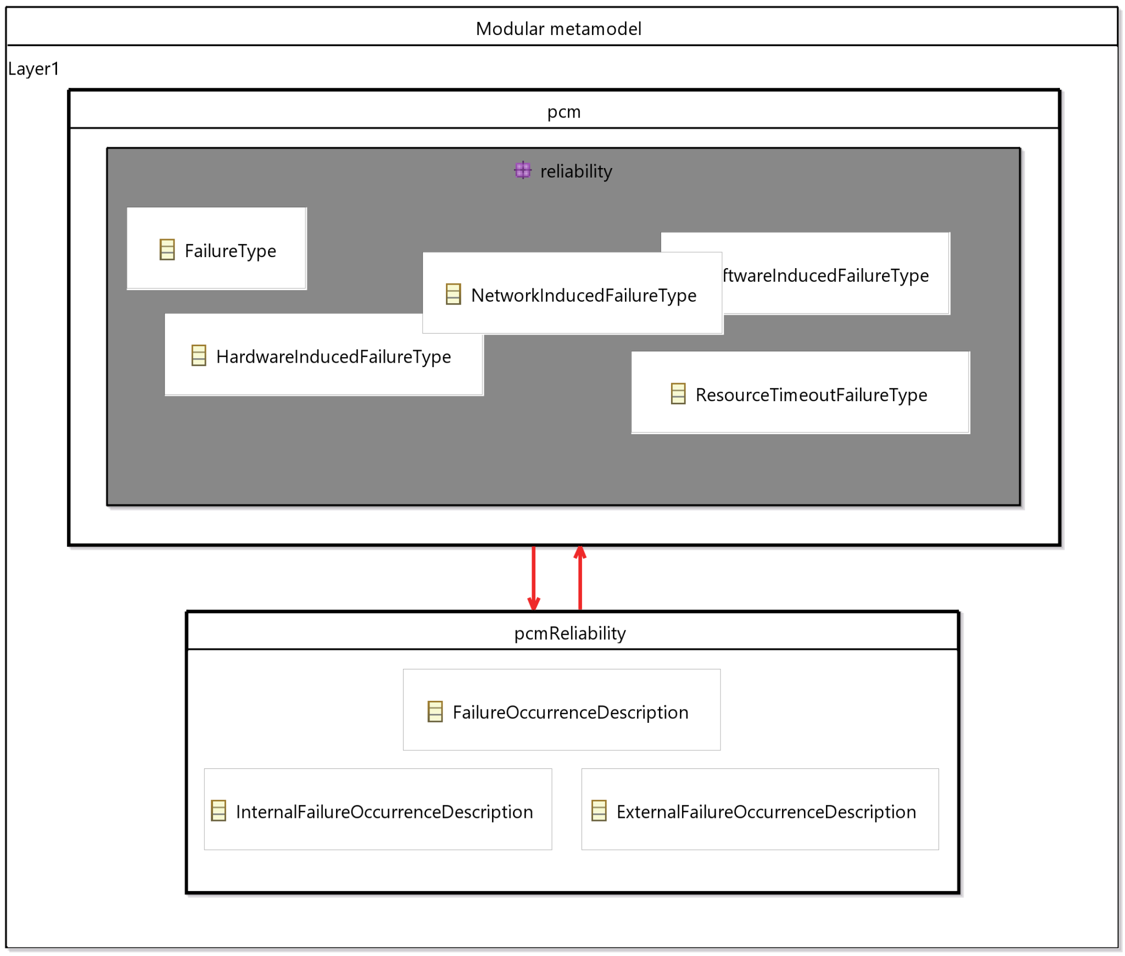

Modular metamodel
Layer1
pcm
reliability
FailureType
NetworkInducedFailureType
ftwareInducedFailureType
HardwareInducedFailureType
ResourceTimeoutFailureType
pcmReliability
FailureOccurrenceDescription
InternalFailureOccurrenceDescription
ExternalFailureOccurrenceDescription

Figure 4.13: Detected cycle between *pcm* and *pcmReliability*

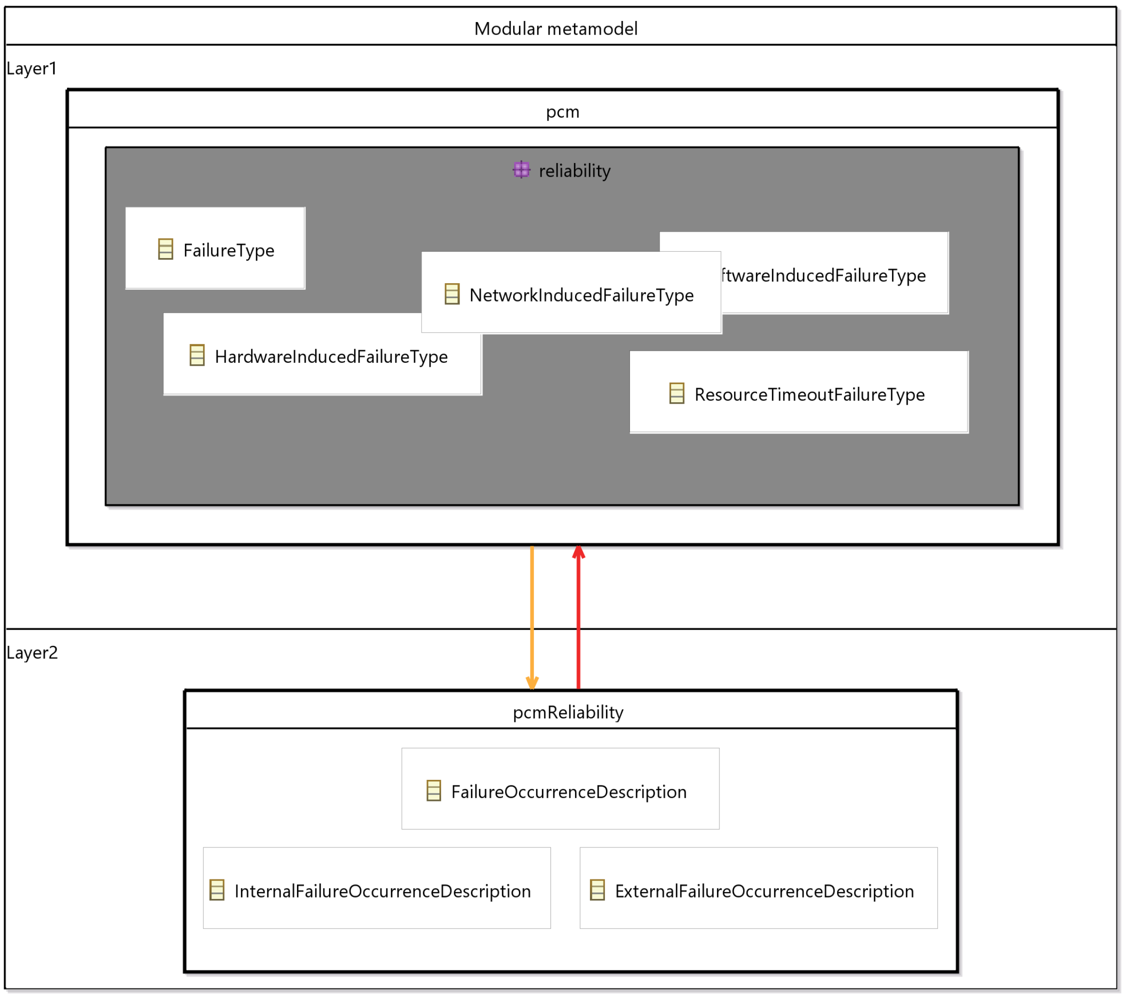

Modular metamodel
Layer1
pcm
reliability
FailureType
ftwareInducedFailureType
NetworkInducedFailureType
HardwareInducedFailureType
ResourceTimeoutFailureType
Layer2
pcmReliability
FailureOccurrenceDescription
InternalFailureOccurrenceDescription
ExternalFailureOccurrenceDescription

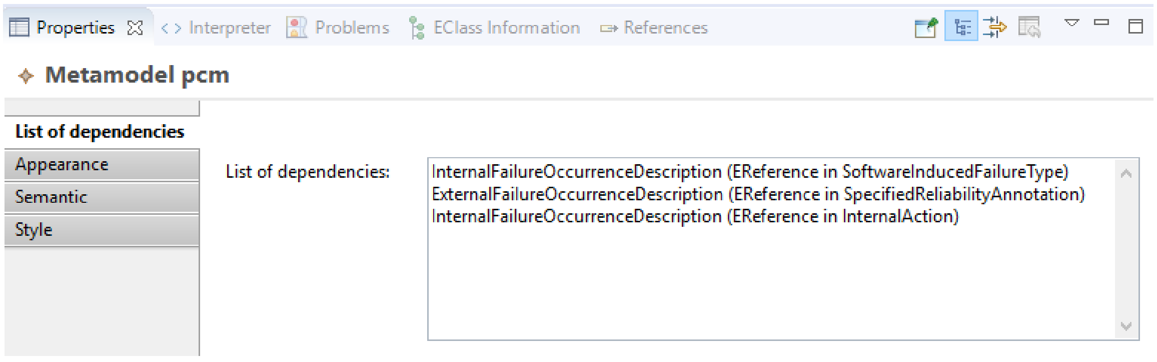
Figure 4.14: Problematic dependency detection

More generally, a dependency is colored in the following manner:

- ***red***, if it takes part of a cycle
- ***orange***, if it goes from a layer to a layer underneath it
- ***black*** otherwise.

If a dependency both takes part of a cycle and points towards a lower layer, it takes an orange color. This decision is justified by the fact that the cycle can still be identified through at least one other red dependency. This way, the user is notified about both violations (Figure 4.14).

Finally, classes that are involved in a dependency are listed in the properties view under the *"List of dependencies"* page. For that matter, the user only has to select an edge from the diagram and the classes will be listed along with the dependencies' nature following the pattern in Listing 4.1.



List of dependencies:

```
InternalFailureOccurrenceDescription (EReference in SoftwareInducedFailureType)
ExternalFailureOccurrenceDescription (EReference in SpecifiedReliabilityAnnotation)
InternalFailureOccurrenceDescription (EReference in InternalAction)
```

Figure 4.15: Properties view

```
EClassifierFromMetamodelB (DependencyType in EClassFromMetamodelA)
```

Listing 4.1: Pattern for a dependency from metamodel A to metamodel B

A dependency can have one of the following types:

- Supertype
- EReference
- Return type of EOperation
- Parameter type of EOperation
- EGenericType

For example, Figure 4.15 serves as an example to show the list of dependencies for the edge going from *pcm* to *pcmReliability*. Based upon this knowledge, the user can then take the necessary action to modularize the metamodel. In our example, we could move `SpecifiedReliabilityAnnotation` and `InternalAction` respectively from the subpackages *qos_reliability* and *seff* to *pcmReliability* since they inherit from respectively the abstractions `SpecifiedQoSAnnotation` and `AbstractInternalControlFlowAction`. In addition, we delete the `EReference` in `SoftwareInducedFailureType` to `InternalFailureOccurence` and only leave the opposite `EReference`. Manipulating this sort of details cannot be done in the MRS diagram and should be done either in the EMF tree editor or the Ecore graphical editor. The Ecore representation of a package (e.g. *reliability*) can be accessed through the MRS graphical editor via a double click.

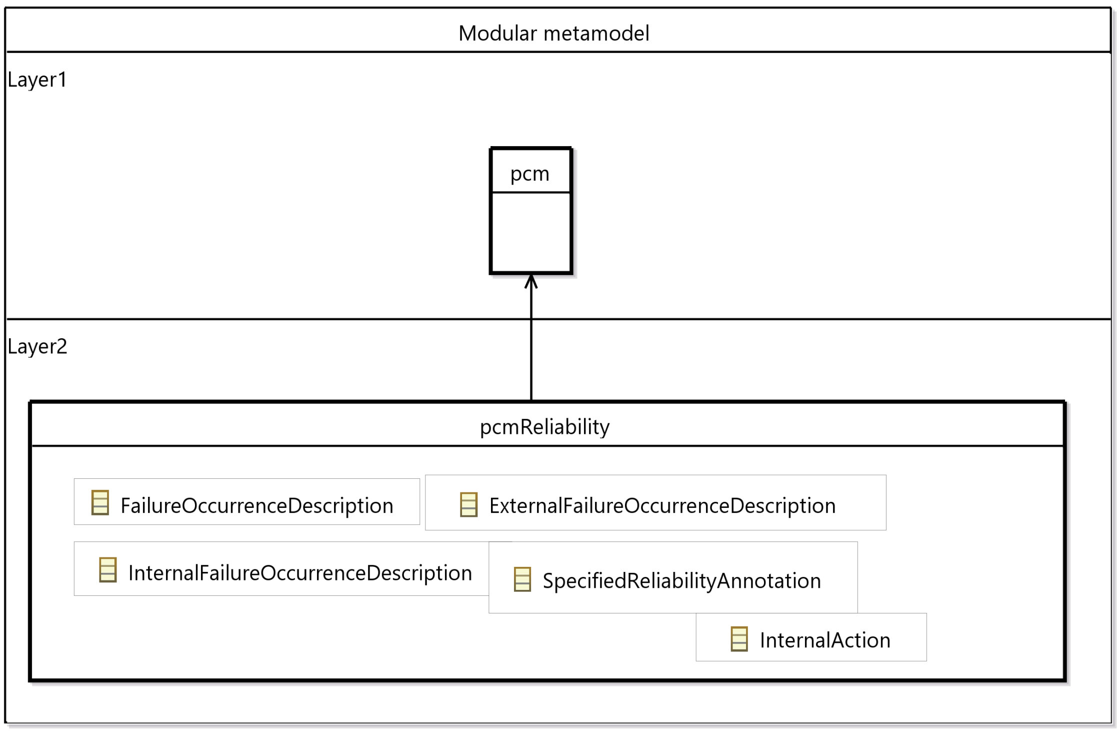

Modular metamodel
Layer1
pcm
Layer2
pcmReliability
FailureOccurrenceDescription
ExternalFailureOccurrenceDescription
InternalFailureOccurrenceDescription
SpecifiedReliabilityAnnotation
InternalAction

Figure 4.16: Final result of the modularization with the MRS graphical editor

4.3 Implementation Details

The implementation of the MRS graphical editor has the following project structure, where each project defines an Eclipse plugin:

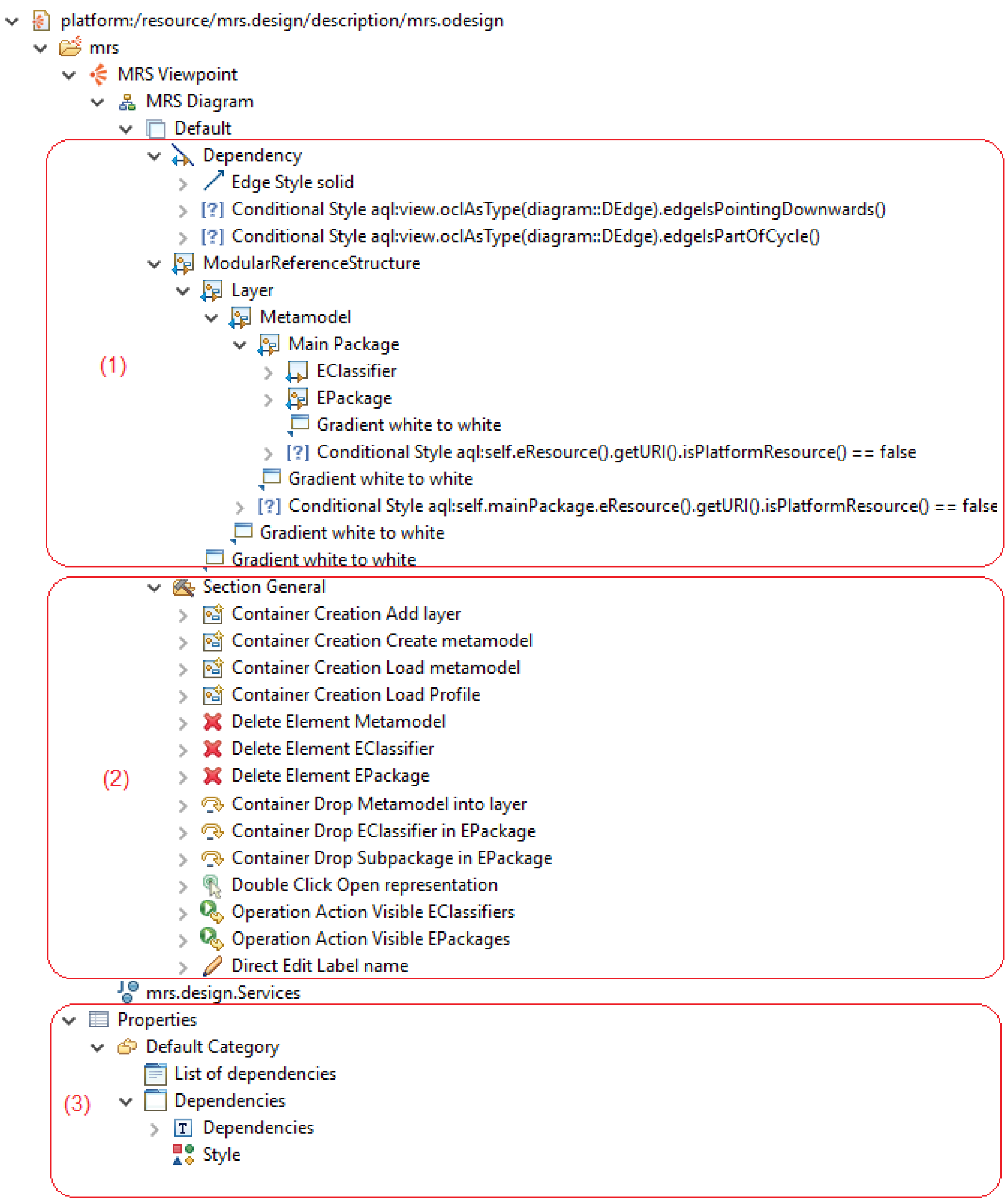
- ***mrs*** contains the MRS metamodel and the generated model code.
- ***mrs.design*** contains the definition of the graphical editor.
- ***mrs.custom*** contains external Java actions and utility classes.
- ***mrs.edit*** and ***mrs.editor*** contain the generated edit and editor code.

The central part of Sirius-based graphical editors is the Viewpoint Specification Model (VSM). Figure 4.17 shows the VSM that we developed during the course of this work. This file is located under *mrs.design/description*. It defines the graphical representation of the MRS model elements (1), the actions that the user can perform (2) and a custom properties sheet (3) to hold information about the dependencies between metamodels.

4.3.1 Diagram Elements Definition

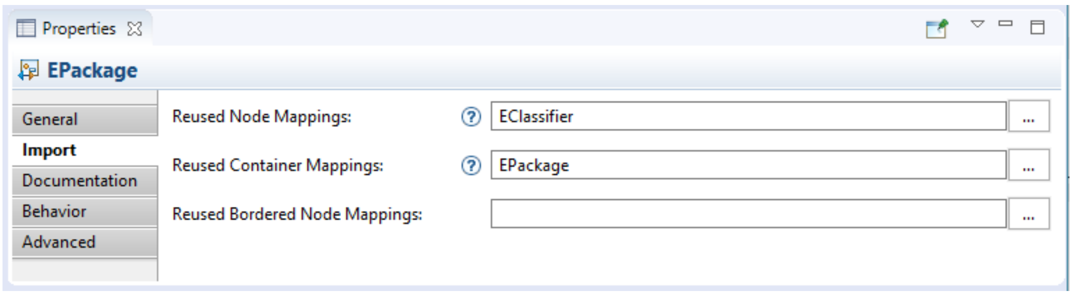
Section (1) in Figure 4.17 shows how diagram elements are defined. First of all, we define the *ModularReferenceStructure* container as the global container to hold the layered structure. It corresponds the root of the represented MRS model, i.e. an instance of the `ModularReferenceStructure EClass`. This global container has a vertical stack layout. That means that its children, the layers, are stacked vertically. The represented layers are the ones referenced in the `ModularReferenceStructure EClass` in the `layers` feature. Finally, metamodels are defined as containers inside layers. They are determined through the `metamodels` feature from the `Layer EClass`. The *Main Package* container is the graphical representation of the main `EPackage` referenced in the `mainPackage` feature. We define two possible children for the *Main Package* container: *EClassifiers* and *EPackages*. *EClassifiers* are defined as nodes, as opposed to the *EPackages* which are defined as containers, since they can contain further *EPackages* and *EClassifers*. Fortunately, Sirius allows such recursive definition via the *mapping import* feature (Figure 4.18).

Since the main package of a metamodel may contain a large number of *EClassifiers* and *EPackages*, we decided to only display certain *EClassifiers* and *EPackages* that are selected by the user and store the user's decision in the *visibleEClassifiers* and *visibleEPackages* references in the `Metamodel EClass`. Our first attempt was then to simply display the elements referenced in these lists. Soon enough, it came to our attention that if a `EClassifier` or a `EPackage` is externally renamed, for example in the EMF tree editor or the Ecore graphical editor for the containing metamodel, *visibleEClassifiers* or *visibleEPackages* are left with a dangling reference to the old object. The fact of renaming an object did not update the reference but actually created a new one. Thus, if the objects from *visibleEClassifiers* or *visibleEPackages* are directly displayed inside the main package container, some invalid elements may as well be displayed. To counter this problem, we implemented *Java services* that check which elements in the given lists are valid and by



```
platform:/resource/mrs.design/description/mrs.odesign
  mrs
    MRS Viewpoint
      MRS Diagram
        Default
          Dependency
            Edge Style solid
            [?] Conditional Style aql:view.oclAsType(diagram::DEdge).edgeIsPointingDownwards()
            [?] Conditional Style aql:view.oclAsType(diagram::DEdge).edgeIsPartOfCycle()
          ModularReferenceStructure
            Layer
              Metamodel
                Main Package
                  EClassifier
                  EPackage
                  Gradient white to white
                  [?] Conditional Style aql:self.eResource().getURI().isPlatformResource() == false
                Gradient white to white
                [?] Conditional Style aql:self.mainPackage.eResource().getURI().isPlatformResource() == false
              Gradient white to white
            Gradient white to white
          Section General
            Container Creation Add layer
            Container Creation Create metamodel
            Container Creation Load metamodel
            Container Creation Load Profile
            Delete Element Metamodel
            Delete Element EClassifier
            Delete Element EPackage
            Container Drop Metamodel into layer
            Container Drop EClassifier in EPackage
            Container Drop Subpackage in EPackage
            Double Click Open representation
            Operation Action Visible EClassifiers
            Operation Action Visible EPackages
            Direct Edit Label name
      mrs.design.Services
    Properties
      Default Category
        List of dependencies
        Dependencies
          Dependencies
          Style
```

Figure 4.17: The MRS Viewpoint Specification Model



Reused Node Mappings: EClassifier
Reused Container Mappings: EPackage
Reused Bordered Node Mappings:

Figure 4.18: Mapping import for the *EPackage* container

the same occasion delete the dangling references. A Java service[1] is simply a method that is implemented externally in a Java class and can be called inside the VSM. Listing 4.2 shows the implementation of the Java service that is used to determine the *EClassifiers* that should be visible inside a *EPackage* and to delete the dangling references. A similar method was also implemented to deal with the *EPackages*. These methods are implemented in the Java class `mrs.design.Services` which is located in the *mrs.design* plugin.

```java
    /**
     * Computes the EClassifiers that should be visible and removes all invalid references
         in the visibleEClassifiers list
     * @param ePackage the package being inspected. Can be a main package or a subpackage
     * @param metamodel the metamodel containing the EClassifiers
     * @return a list of the EClassifiers that should be visible inside ePackage
     */
    public Collection<EClassifier> getVisibleEClassifiers(EPackage ePackage, Metamodel
        metamodel) {
        // get the direct EClassifiers of ePackage
        Collection<EClassifier> eClassifiers = ePackage.getEClassifiers();

        //get all EClassifiers of the metamodel
        Collection<EClassifier> eAllClassifiers = getEAllClassifiers(metamodel.
            getMainPackage());

        //get the list of the EClassifiers that should be visible inside the metamodel
        Collection<EClassifier> visibleEClassifiers = metamodel.getVisibleEClassifiers();

        Collection<EClassifier> result = new ArrayList<EClassifier>();
        Collection<EClassifier> ghostEClassifiers = new ArrayList<EClassifier>();

        for (EClassifier visibleEClassifier : visibleEClassifiers) {
            //Mark invalid EClassifiers from the visibleEClassifiers list
            if(!eAllClassifiers.contains(visibleEClassifier))
                ghostEClassifiers.add(visibleEClassifier);

            //Mark EClassifiers that should be displayed inside ePackage
            if(eClassifiers.contains(visibleEClassifier))
                result.add(visibleEClassifier);
```

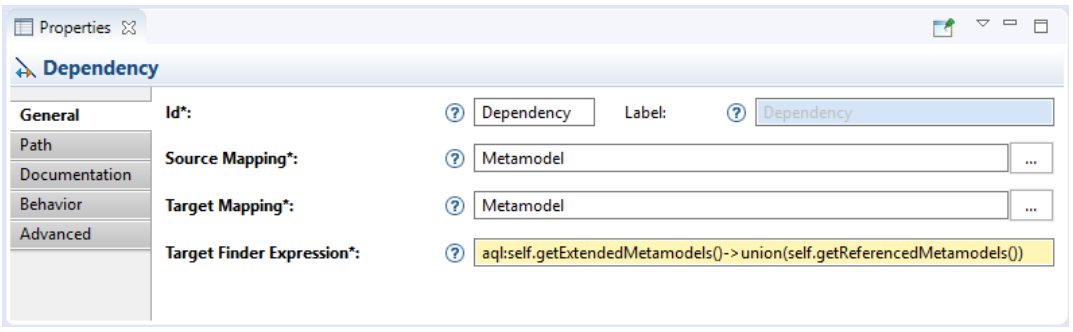
[1] `https://www.eclipse.org/sirius/doc/specifier/general/Writing_Queries.html#service_methods`

```
        }

        //Delete dangling references
        visibleEClassifiers.removeAll(ghostEClassifiers);

        return result;
    }
```

Listing 4.2: Implementation of the getVisibleEClassifiers Java service

Now that we've covered how the representation of the model element is defined, we are able to present how the graphical representation of the dependencies between metamodels is defined. The diagram element *Dependency* is responsible for this. It is a relation-based edge, meaning that it does not represent a model element, but rather a relation that we have to define ourselves. Figure 4.19 shows how this edge is defined. Both source and target elements are *Metamodels*. In the target finder expression, we compute the target candidates for a given source *Metamodel*. For this purpose, we implemented two Java services: `getExtendedMetamodels()` and `getReferencedMetamodels()`. The method `getExtendedMetamodels()` returns a set of the metamodels in the layered structure that contain each at least one class that is extended by a stereotype that references a class from the source metamodel. The stereotypes that are considered are the ones defined in the EMF Profiles that are referenced in *loadedProfiles* in the root model element. The method `getReferencedMetamodels()` returns a set of the metamodels that contain at least one `EClassifier` that is referenced by an `EClass` from the source metamodel. Finally, the target finder expression returns a union of both sets. It is also worth noting that while these services inspect the dependencies of a metamodel, they also add referenced metamodels to the structure. If they find a metamodel that is currently not loaded in the modular layered structure, they create an instance of `Metamodel` in the same layer of the metamodel being inspect and set its *mainPackage* to the main package of the found metamodel. Thus, they effectively and automatically load referenced metamodels to the layered structure, whether they are present in the workspace or in the target platform.

One of the main features of the MRS graphical editor is the detection of violations of the rules of the modular reference structure. We implement this feature by providing conditional styles to the *Dependency* edge. By default, edges have a black color. However,



Dependency

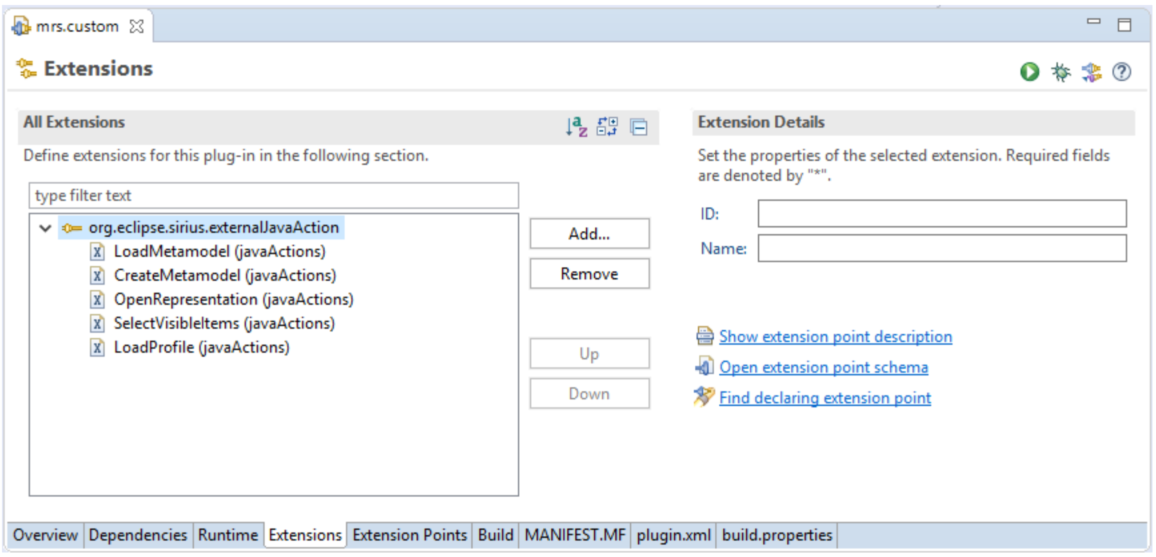
Id*: Dependency   Label: Dependency

Source Mapping*: Metamodel

Target Mapping*: Metamodel

Target Finder Expression*: aql:self.getExtendedMetamodels()->union(self.getReferencedMetamodels())

Figure 4.19: Definition of the *Dependency* diagram element


mrs.custom

Extensions

All Extensions

Define extensions for this plug-in in the following section.

type filter text

- org.eclipse.sirius.externalJavaAction
  - LoadMetamodel (javaActions)
  - CreateMetamodel (javaActions)
  - OpenRepresentation (javaActions)
  - SelectVisibleItems (javaActions)
  - LoadProfile (javaActions)

Extension Details

Set the properties of the selected extension. Required fields are denoted by "*".

ID:

Name:

Figure 4.20: External Java actions for the MRS graphical editor

if the edge goes from a metamodel to a metamodel in a lower layer, it gets painted with an orange color. This check is performed in the `edgeIsPointingDownwards()` Java Service, which takes an edge as parameter, retrieves the source and target metamodels and compares the relative position of the corresponding layers in the list of layers. The result is `true` if the index of the target layer is bigger than the index of the source layer and `false` otherwise.

The second conditional style that we provide paints the edge with a red color if it is part of a cycle. The `edgeIsPartOfCycle()` Java service takes the edge as a parameter, retrieves the source and target metamodels and performs a breadth-first search starting from the target metamodel and sees if it can reach the source metamodel. If that's the case, it returns `true`, otherwise `false`.

4.3.2 Defined User Actions

In this subsection we present how we have defined the actions that can be taken by the user in a MRS diagram. While some actions are straightforward, like the *Add Layer* action since it only creates an instance of the `Layer EClass` in the `ModularReferenceStructure`, others required more effort. The *Create Metamodel* creates an instance of the `Metamodel EClass` in the selected layer and passes it to the `CreateMetamodel` external Java action. External Java actions are plugin extensions that use the `org.eclipse.sirius.externalJavaAction` extension point and implement the `IExternalJavaAction` interface. We define our external Java actions in the *mrs.custom* plugin (Figure 4.20). Once called, the `CreateMetamodel` external java action triggers the Ecore modeling project creation wizard, retrieves the main package of the Ecore-based metamodel that the user has just created and sets it as the *main package* of the `Metamodel` instance that it received from the VSM.

In a similar fashion, the *Load Metamodel* action creates a `Metamodel` instance and passes it as a parameter to the `LoadMetamodel` external Java action, that opens a `LoadResourceFromWorkspaceDialog` which prompts the user to select an Ecore file from the workspace. The corresponding main package is then retrieved and assigned to the passed `Metamodel` instance.

The *Load Profile* action directly makes a call to the `LoadProfile` external Java action, which prompts the user to select an EMF Profile from the workspace. The external Java action then adds the profile to the *loadedProfiles* list and adds the metamodels containing the classes referenced in the profile to the selected layer. Retrieving the metamodels wasn't a straightforward task, since the `EClasses` that are referenced in the profile are loaded via their corresponding registered package and not the actual main `EPackage` of the corresponding metamodel. To get the actual metamodel, we had to first navigate from the registered package to *GenModel* using the namespace URI of the registered package, in order to retrieve the actual main `EPackage`. The method responsible for this is provided in Listing 4.3.

```java
public static EPackage getEcorePackageFromRegisteredPackage(EPackage registeredPackage,
    TransactionalEditingDomain editingDomain) {
        ResourceSet resourceSet = editingDomain.getResourceSet();
        String nsURI = registeredPackage.getNsURI();
        resourceSet.getURIConverter().getURIMap().putAll(EcorePlugin.computePlatformURIMap(
            false));

        Map<String, URI> ePackageNsURItoGenModelLocationMap = EcorePlugin.
            getEPackageNsURIToGenModelLocationMap(false);
        URI location = ePackageNsURItoGenModelLocationMap.get(nsURI);
        Resource resource = resourceSet.getResource(location, true);
        EcoreUtil.resolveAll(resource);
        EPackage ecorePackage = ((GenModel) resource.getContents().get(0)).getGenPackages().
            get(0).getEcorePackage();
        return ecorePackage;
    }
```

Listing 4.3: The method responsible for retrieving the actual main package based on the registered `EPackage`

We also redefined the default behavior of deleting graphical elements from the MRS diagram. First, the deletion of a *Metamodel* from the diagram only removes the corresponding `Metamodel` instance from its containing `Layer`. Second, we also adjusted the behavior for *EClassifiers* and *EPackages* so that their deletion does not actually delete them from their metamodels but only removes them from the *visibleEClassifiers* and *visibleEPackages* respectively.

Since Sirius does not offer a default implementation for the drag and drop features, we had to provide our own definitions for the drag and drop of *Metamodels*, *EClassifiers* and *EPackages*. The drag and drop of *Metamodels* is straightforward. It only changes its container from the old layer to the new one. In the case *EClassifiers* and *EPackages*, we first check if both the source and target metamodels are present in the workspace, since we do not want to modify target platform metamodels. If it is the case, the *EClassifiers* and *EPack-*

ages are moved to the target metamodel and the *visibleEClassifiers* and *visibleEPackages* lists are updated accordingly.

We deemed the drag and drop feature in the MRS graphical editor necessary for metamodel modularization. To some extent, the user is also allowed to edit the `EClassifiers` and the `EPackages` of the underlying metamodels in the MRS diagram by directly renaming them or through their respective default properties sheet. To have full access to the metamodels, the modeler still has to open either the EMF Tree editor or the Ecore graphical editor. For this purpose, the MRS graphical editor offers a way to open the Ecore graphical editor for any `EPackage` in the MRS diagram simply by double clicking on it. The double click makes a call to the `OpenRepresentation` external Java action that opens the representation of the package if it exists, or creates one and opens it if not. This external Java action was actually not implemented during the course of this work, but priory as we developed the Sirius-based PCM graphical editors. Along with this class, we also reuse the `SiriusCustomUtil` from prior work. These two classes were for simplicity reasons copied from `org.palladiosimulator.editors.sirius.custom` into `mrs.custom` but will be eventually subject of a broader refactoring in the future.

The default properties view for *Metamodel*s offers a way to edit the intrinsic properties defined in the `Metamodel EClass` like the *visible EClassifiers* and *visible EPackages* references. When trying to edit these, multiple `EClassifiers` and `EPackagess` are shown in the dialog, that are not necessarily part of the *Metamodel* being edited. In fact, all EClassifiers and EPackages from the metamodels loaded in the MRS diagram are displayed. For this reason, we added the context menu entries *Visible EClassifiers* and *Visible EPackages* for *Metamodels*. When the user clicks on the context menu entry, the `SelectVisibleItems` external Java action is called. It receives three parameters: the metamodel element, the list of all *EClassifiers/EPackages* inside this metamodel and the *Reference ID* of the `visibleEClassifiers/visibleEPackages` in the `Metamodel EClass`. For this matter, we implemented the Java services `getEAllClassifiers()` and `getEAllSubPackages()` that retrieves all `EClassifiers/EPackages` inside a given `EPackage` by visiting recursively its subpackages. For the *Reference ID*, we implemented Java services that simply return the ID provided in the `MrsPackage` class. For the case of `visibleEClassifier`, the *Reference ID* is given by `MrsPackage.METAMODEL__VISIBLE_ECLASSIFIERS`. Once the `SelectVisibleItems` external Java action is called, it opens a `FeatureEditorDialog`[2] and sets the value of the reference given by the *Reference ID* to the result of the dialog.

4.3.3 Properties Sheet

One crucial functionality that we propose in the MRS graphical editor is that it gives the user precise information about in which the metamodels depend on each other. This information is delivered in the properties view, as we've seen at the end of Section 4.2. Since its fourth version, Sirius offers a way to implement custom properties sheets directly in the VSM. We use this functionality to deliver the custom properties sheet *"List of dependencies"* that appears when the user select an edge from the diagram. We implemented the properties sheet so that it displays a text area widget in a read-only mode and whose content is

[2]`delivered in org.eclipse.emf.edit.ui.celleditor`

delivered by the `printDependencies()` Java service. This service retrieves the source and target *Metamodels* from the edge, computes all dependencies that go from the source to the target metamodel and returns a String representation of these dependencies.

5 Evaluation

In this chapter, we present the results of the evaluation of the MRS graphical editor. Two aspects are essentially worth validating: the correctness and the applicability of our tool. In our context, we mean by *correctness* that the metamodels resulting from the modularization are valid metamodels and are not corrupted. Besides, the information provided by the tool, in regards to the dependencies between metamodels and the detection of violations to the modular reference structure, must be correct and reliable. By *applicability* we mean that the tool fulfills the purpose it was designed for — that is, offering the user a convenient way for metamodel modularization on the basis of the modular reference structure.

The usage scenario presented in chapter 4 validates to some extent the correctness and the applicability of the MRS graphical editor. The process of extracting the *pcmReliability* module takes advantage of the cyclical dependency detection and the information provided by the properties sheet about how *pcmReliability* and *pcm* relate to each other, thus effectively validating the applicability of the tool. The correctness follows from the fact that the resulting *pcm* and *pcmReliability* are both valid Ecore-based metamodels and only *pcmReliability* depend on *pcm*.

Further, we proceeded in the evaluation in two other ways. First, we tested our tool using the prototypical modular PCM (mPCM)[1]. Second, we conducted a small evaluation session using the Smartgrid metamodel[2] [9].

5.1 Modular PCM

The need for a modular PCM has to some extent led to the work on the modular reference structure for architecture component-based architecture description languages. Prior to our work on the MRS graphical editor, modularization work on PCM resulted in mPCM, a prototypical modular PCM built on the basis of the modular reference structure. Figure 5.1 shows the resulting MRS diagram after loading the different mPCM modules. The diagram shows no violations to the modular reference structure, since mPCM was designed based on it, which confirms the correctness of the MRS graphical editor.

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

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

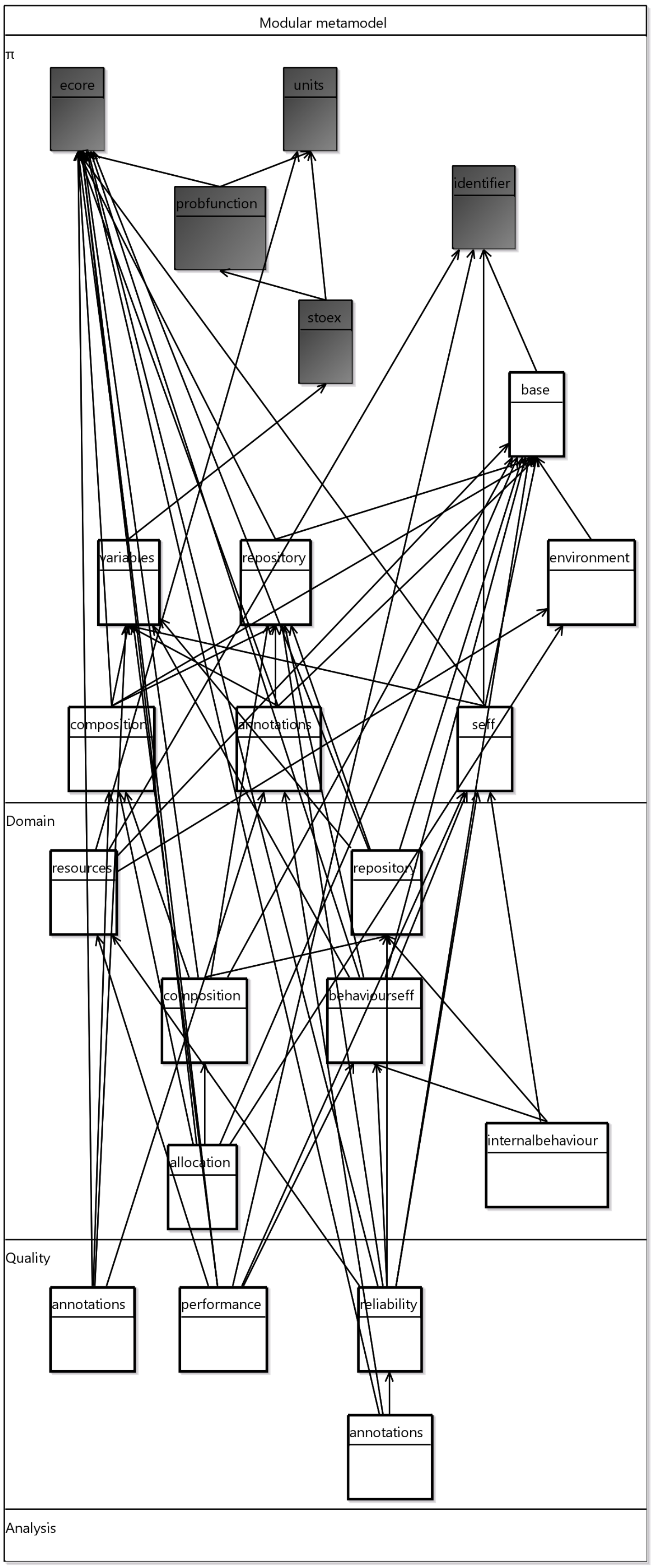


Figure 5.1: MRS Diagram of mPCM

5.2 Evaluation Session with the Smartgrid Metamodel

Using the Smartgrid metamodel (Figure 5.2, Figure 5.3, Figure 5.4), we conducted an evaluation session for the MRS graphical editor. Since we were accustomed to using the tool, we decided to get someone without prior experience with it to try it out. Mr. Rüdiger Heres kindly accepted to take part of the evaluation. Mr. Heres was provided with an Eclipse environment, where the MRS graphical editor was installed and the Smartgrid metamodel was imported into the workspace. He also was given a description of the MRS graphical editor, a user guide as well as the following instructions:

1. Create a modeling project
2. Create a MRS Model in the project (New > Other > Mrs Model) and choose "Modular Reference Structure" as the Model Object
3. Select the MRS Viewpoint (right-click on the project > Viewpoints Selection)
4. Add 3 layers via "Add Layer" palette entry and name them respectively "paradigm", "domain" and "analysis" from top to bottom.
5. Load Smartgrid Topo to the domain layer and Smartgrid Input and Output to the analysis layer.
6. Create a new metamodel called "base" in the paradigm layer.
7. Extract the classes Identifier, NamedIdentifier and NamedEntity into it.
8. Create a new metamodel called "typerepo" in the domain layer.
9. Extract the classes Repository, SmartMeterType, ConnectionType and NetworkNodeType into it.
10. Create a new metamodel called "graph" in the paradigm layer.
11. Extract the classes CommunicatingEntity, PhysicalConnection, PowerGridNode, NetworkEntity and LogicalCommunication into it.
12. Eventually resolve problematic dependencies. Tip: Use the dependency inversion principle: To reverse a dependency from class A to class B, create a class C that references both A and B and delete the A to B dependency.

Without prior knowledge of the Smartgrid metamodel and the MRS graphical editor, Mr. Heres successfully completed the evaluation task. He could easily add the layers and the metamodels described in the instructions. The MRS diagram before proceeding with the dependency inversion at the twelfth step is given in Figure 5.5. After that, he used the information provided by the tool, that is, the highlighting of dependency from *graph* to *typerepo* and the classes involved in it in order to resolve the problematic dependency. The properties sheet indicated that the `EClass` `PhysicalConnection` in *graph* had a `EReference` to `ConnectionType` in typerepo. With this information, he created a class

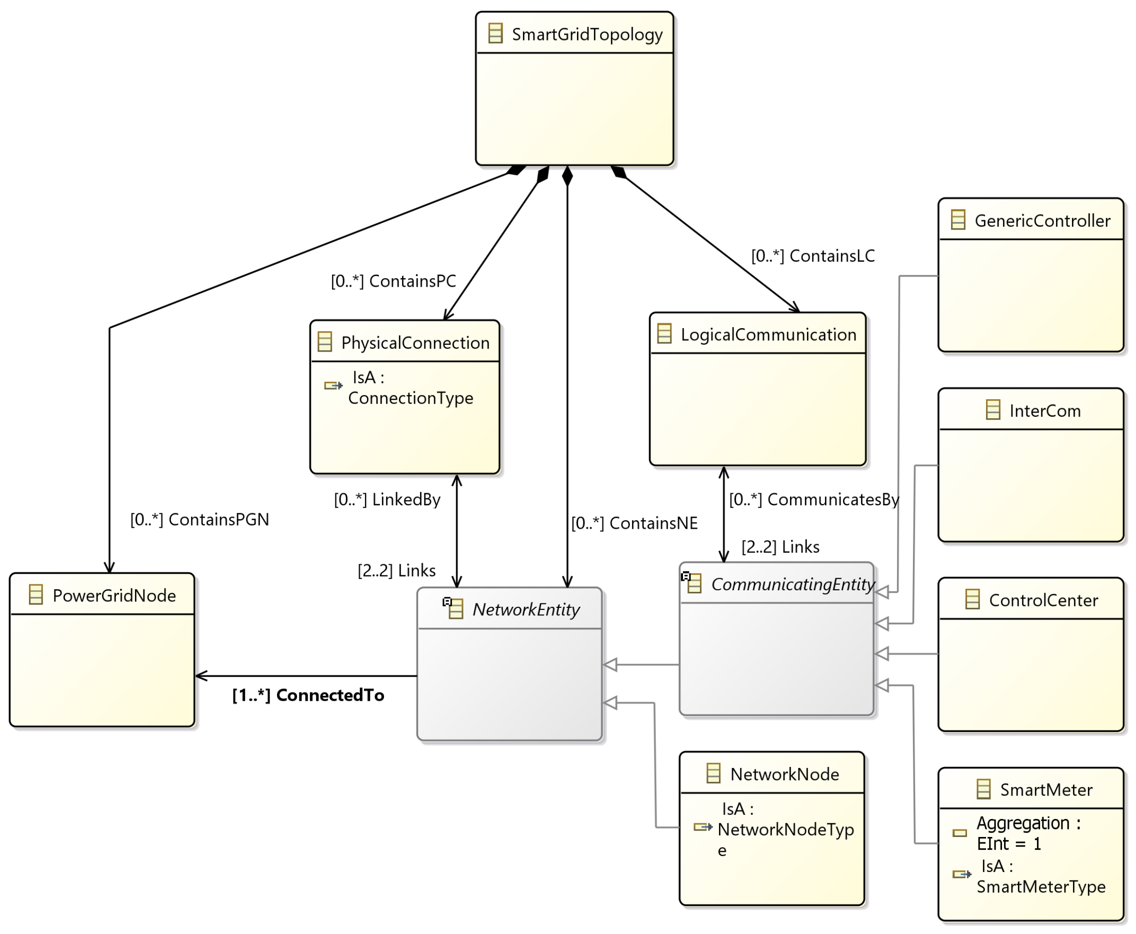


Figure 5.2: Minimal class diagram of Smartgrid Topo

`ConnectionTypeApplication` in *typerepo*, that references both `PhysicalConnection` and `ConnectionType` and deleted the `EReference` from `PhysicalConnection`to `ConnectionType`. Figure 5.6 shows the MRS diagram of Smartgrid after the dependency inversion.

This evaluation task successfully validates the applicability of our tool. By this occasion, we would like to thank Mr. Heres for participating in the evaluation and for his constructive feedback. Mr. Heres praised how the tool immediately detected the violation to the modular reference structure and how he directly had access to the information he needed to resolve the problematic dependency. He also noted that the layout of the diagram should adjust to the loaded metamodels and that re-sizing a layer should not affect the size of adjacent layers. We gladly take note of these issues as part of future work. Particularly, the last point is discussed further in chapter 7.

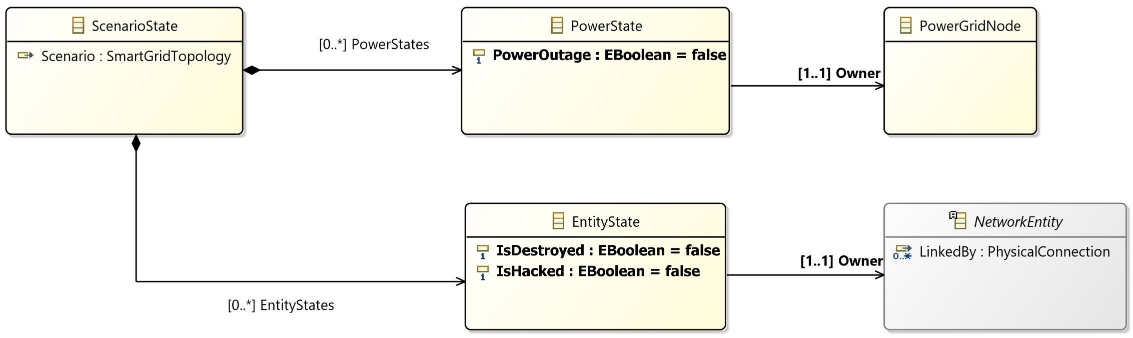

ScenarioState
Scenario : SmartGridTopology
[0..*] PowerStates
PowerState
PowerOutage : EBoolean = false
[1..1] Owner
PowerGridNode
EntityState
IsDestroyed : EBoolean = false
IsHacked : EBoolean = false
[0..*] EntityStates
[1..1] Owner
NetworkEntity
LinkedBy : PhysicalConnection

Figure 5.3: Class diagram of Smartgrid Input

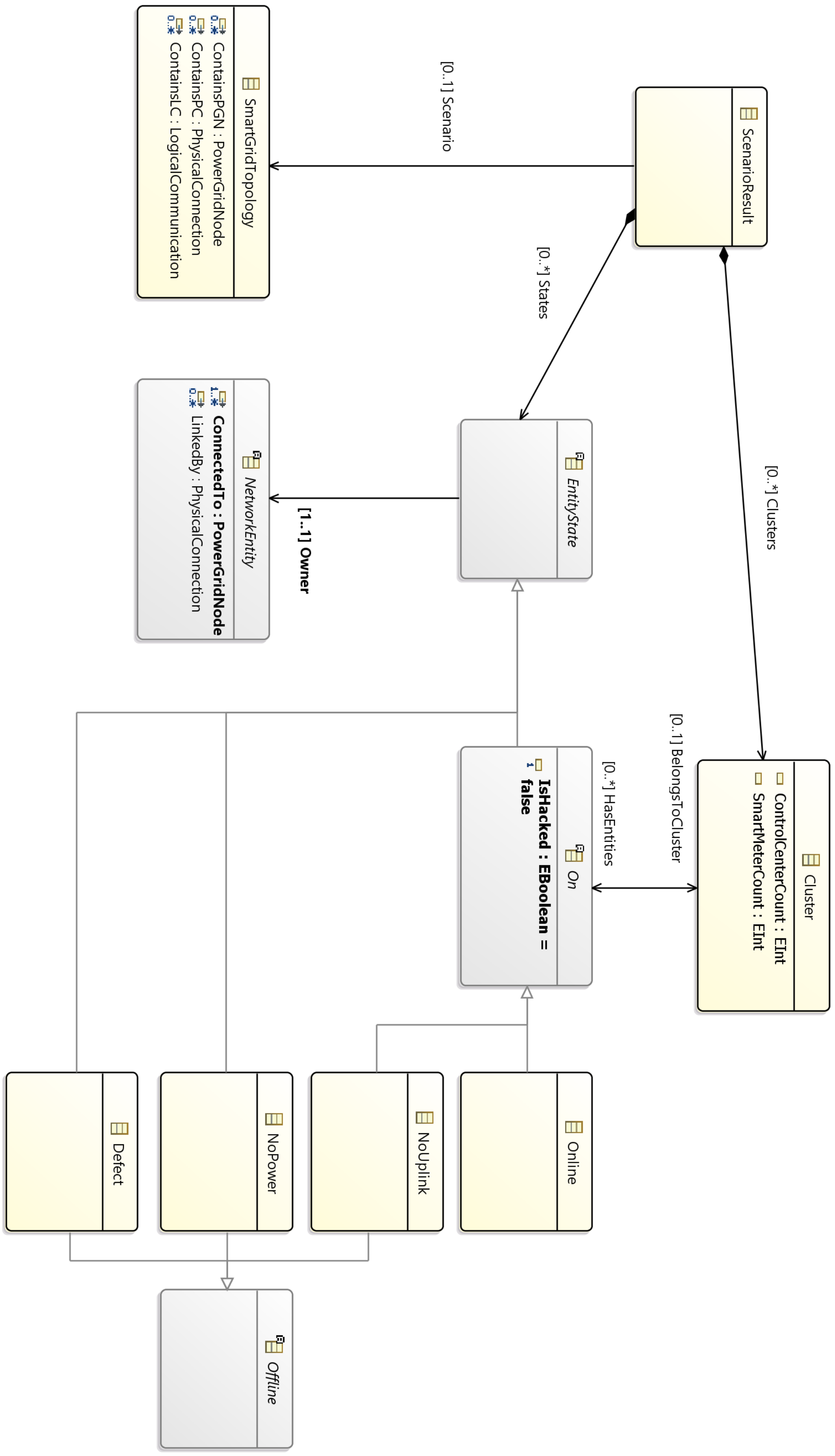


Figure 5.4: Class diagram of Smartgrid Output

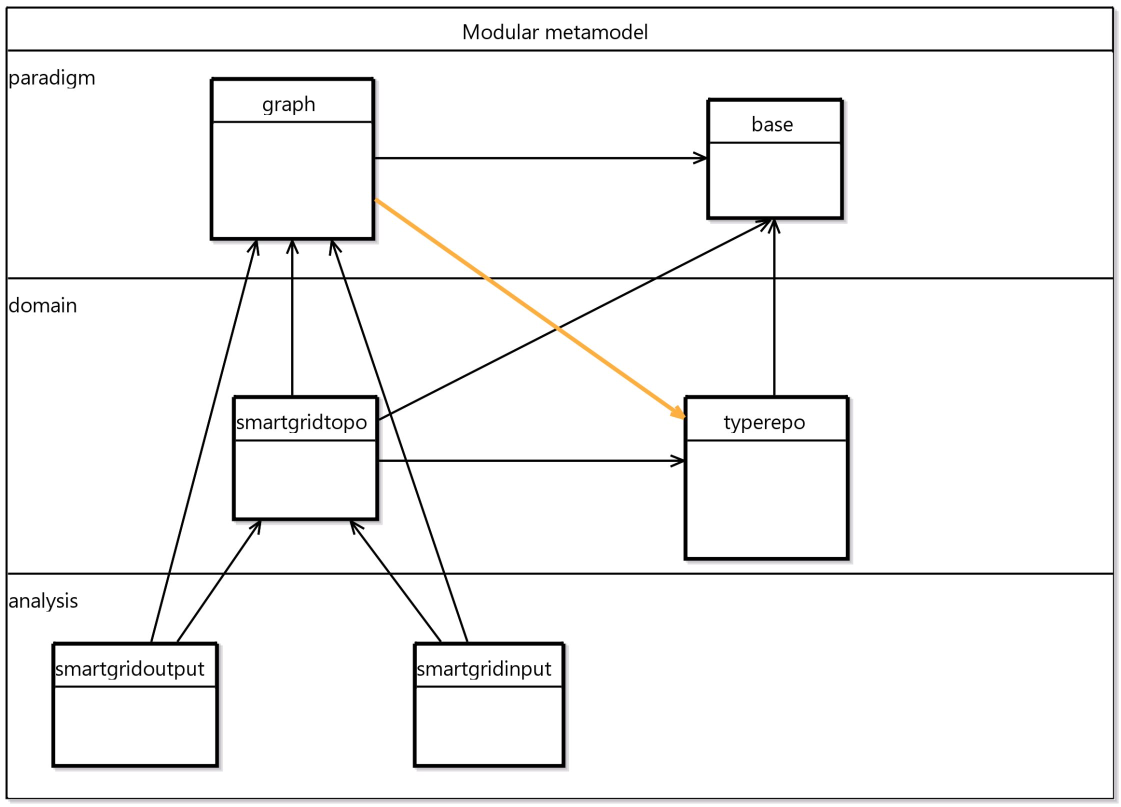


Figure 5.5: MRS Diagram of Smartgrid before proceeding with the dependency inversion

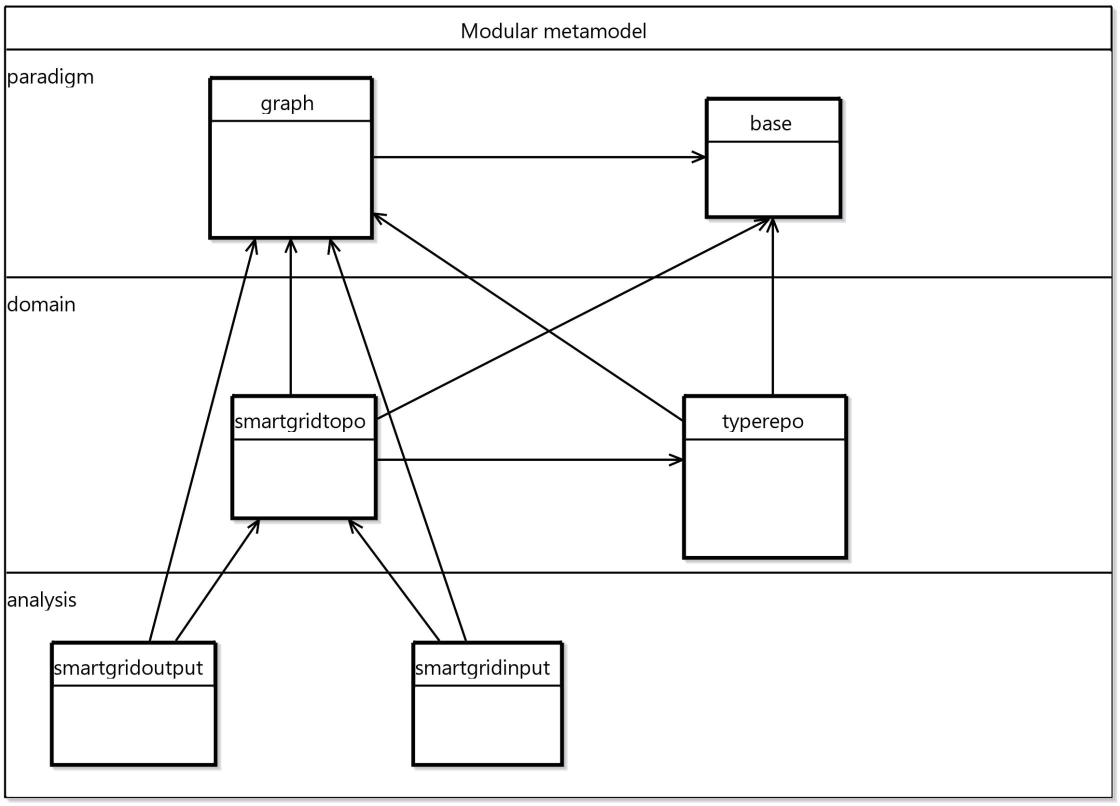


Figure 5.6: MRS Diagram of Smartgrid after the dependency inversion

6 Related Work

Modular programming is a well established concept in the realm of software development, especially in object-oriented programming. Although the basic principles of modularity also hold for DSLs and metamodels, modular metamodeling and metamodel modularization have not received much of attention. Some works have touched on the subject, but with a main focus on modularity at the model level. Garmendia et al. introduce a modular structure that divides a model into modules based on a priory chosen modularity pattern [5]. The proposed tool, EMF Splitter, allows the user to map the metamodel to a structure comprised of so-called *projects*, *packages* and *units* by annotating[1] the original metamodel. Although the tool adds a logical structure to the metamodel and separates the produced models into modules, it doesn't offer a way to effectively modularize the underlying metamodel since the information about the modularity pattern is only kept in the form of annotations and the underlying metamodel is not affected.

Strüber et al. introduce the concept of export and import interfaces at the metamodel level in order to enable information hiding at the model level [19]. The proposed Composite EMF Models[2] define these interfaces by extending the Ecore metamodel. Ecore-based metamodels can then be encapsulated and equipped with the interfaces, thus producing models that can only interact with each other through the suggested interfaces. Although this offers defacto a separation of concerns at the model level, it brings modifications to the underlying metamodel in an intrusive manner, due to the addition of the export and import interfaces and does not explicitly address the question of modularity at the metamodel level.

In other works, Strüber et al. propose a tool support for clustering metamodels [17] and a tool support for model splitting [18, 16]. While the former employs clustering techniques to split a monolithic input metamodel, the latter uses model crawling techniques and splits the metamodel based on an input description. Our approach is fundamentally different in the way that it lets the user have a full control on the modularization by providing substantial information about how metamodels relate to each other inside the layered structure and by facilitating the manipulation of metamodels. As we discuss in chapter 7, metamodel clustering techniques can be integrated in the future to the MRS graphical editor as a way to assist the user in taking modularization decisions.

[1]In a currently under development version of EMF Splitter, the mapping between metamodel elements and the structure no longer uses the annotation process, but saves the information about the modularity pattern in a model.

[2]`http://www.mathematik.uni-marburg.de/~swt/compoemf/index.php`

7 Limitations and Future Work

We see a considerable improvement potential for the MRS graphical editor. The future work can fall under three categories: the improvement of performance, the improvement of usability and the introduction of whole new features.

7.1 Performance

One of the limitations encountered in the development of the MRS graphical editor with the Sirius framework is that the Java services are meant to be stateless and the Viewpoint Specification Model (VSM) doesn't offer the possibility to hold a global state, where results of computations can be temporary stored and used across the VSM. For this reason, some computations are redundant and cannot take advantage of previous results. For example, when looking for cycles, a breadth-first search has to be done on each edge in the diagram, regardless of previous computations. During the breadth-first search, it would be more convenient to mark all encountered edges that take part in the cycle so that they wouldn't have to be investigated anymore. We can also think of a completely other mechanism for cycle detection where only one computation is done on the whole diagram on refresh, that takes into account the previous state of the diagram and the changes that have been made. Unfortunately, the diagram state cannot be stored in the VSM. We can think of storing this sort of information in a separate file or model, but that would bring the overhead of reading the file each time a change on the diagram is detected.

Another subject worth investigating is model integrity and consistency. In the current state of the MRS Metamodel, the `Metamodel` `EClass` references the main package of the target metamodel in the *mainPackage* `EReference` and `EClassifiers` and `EPackages` from this metamodel in the *visibleEClassifiers* and *visibleEPackages* `EReferences`. This raises the question of how to keep MRS models consistent with the referenced metamodels, in case these metamodels are modified or deleted. For the time being, if the referenced metamodel is deleted or not found, the corresponding *mainPackage* `EReference` is a `null` reference. We also encountered the problem where, if a `EClassifier` or a `EPackage` is renamed and it is referenced in *visibleEClassifiers* or *visibleEPackages*, then we would get a dangling reference to a `EClassifier` or a `EPackage`. We solved this problem by checking the *visibleEClassifiers* or *visibleEPackages* lists and deleting such invalid references when determining which `EClassifiers` or `EPackages` should be displayed. Another approach would be to keep the MRS model consistent using an external tool. Works on this subject include Concordance, a framework for managing model integrity [12].

7.2 Usability

We can think of a various number of ways to improve the usability and the user experience in the MRS graphical editor. The following suggestions are derived from our own experience with the tool and the feedback received at the evaluation:

- When re-sizing a layer, the size of other layers should not be affected. For example, when extending a layer to the bottom, the layer underneath it should not loose in size and should instead be displaced to the bottom with the same re-sizing amount.
- The *Load Metamodel* and *Load Profile* actions should allow multiple selection of metamodels and profiles.
- Provide a way to hide transitive dependencies from the diagram.
- To use the tool, the user usually has to create a modeling project, create a MRS Model inside this project, select the MRS Viewpoint and then open the representation. An improvement to the user experience would be to encapsulate these actions into a single one where the user only has to click on a toolbar button and enter relevant information like the project and model name. Everything else should be done automatically.
- For the time being, layers are automatically added to the bottom of the structure. The user should be able to add a layer between two layers. Furthermore, moving a layer to another position would be an interesting feature to have.

7.3 New Features

The MRS graphical editor assists the modeler by giving him a global overview on the way the metamodels in the layered structure are connected to each other. However, it doesn't make any assertions about how the metamodels should be modularized and leaves modularization decisions completely up to the modeler. As a future work, we can think of implementing a modularization assistant that inspects the metamodels in the layered structure and makes suggestions to the modeler about which classes should form their own module or how a violation to the modular reference structure should be fixed. Clustering techniques applied to metamodels can be used for this purpose. The assistant would also adapt itself according the choices of the modeler to make better predictions.

As part of future work, we can also take into account other aspects that depend on the metamodel being modularized. Such aspects may include other metamodels, model transformations, model editors and various other tools that build on the metamodel in question.

8 Conclusion and Outlook

Metamodel modularization can often be challenging and non-self-evident. In this work, we introduced a graphical tool that is intended to assist the modeler in the modularization by providing visual support for the proposed modular reference structure. We deem our approach a novel way to achieve metamodel modularization in the sense that it leaves the modeler with full control over the modularization and with extended knowledge about the relations between the metamodels in the layered structure. As to the future of the tool, we consider optimizing its performance, improving its usability and extending it with various new features like providing modularization suggestions based on metamodel clustering techniques. The discussed related work in this field can form the basis for our future work.

Bibliography

[1] Jean Bézivin. "In search of a basic principle for model driven engineering". In: *Novatica Journal, Special Issue* 5.2 (2004), pp. 21–24.

[2] Franz Brosch et al. "Architecture-based reliability prediction with the palladio component model". In: *IEEE Transactions on Software Engineering* 38.6 (2012), pp. 1319–1339.

[3] Alan W. Brown, Jim Conallen, and Dave Tropeano. "Introduction: Models, Modeling, and Model-Driven Architecture (MDA)". In: *Model-Driven Software Development.* Ed. by Sami Beydeda, Matthias Book, and Volker Gruhn. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 1–16. ISBN: 978-3-540-28554-0. DOI: 10.1007/3-540-28554-7_1. URL: https://doi.org/10.1007/3-540-28554-7_1.

[4] Martin Fowler. *Domain-specific languages.* Pearson Education, 2010.

[5] Antonio Garmendia et al. "EMF Splitter: A Structured Approach to EMF Modularity." In: *XM@ MoDELS* 1239 (2014), pp. 22–31.

[6] Philip Langer et al. "EMF Profiles: A Lightweight Extension Approach for EMF Models." In: *Journal of Object Technology* 11.1 (2012), pp. 1–29.

[7] Bertrand Meyer. *Object-oriented software construction.* Vol. 2. Prentice hall New York, 1988.

[8] David Lorge Parnas. "On the criteria to be used in decomposing systems into modules". In: *Communications of the ACM* 15.12 (1972), pp. 1053–1058.

[9] Wolfgang Raskob et al. "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.

[10] Ralf H Reussner et al. *Modeling and simulating software architectures: The Palladio approach.* MIT Press, 2016.

[11] Ralf Reussner et al. *The Palladio Component Model.* Karlsruhe Reports in Informatics, ISSN: 2190-4782. Karlsruhe, 2011. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000022503.

[12] Louis M Rose et al. "Concordance: A Framework for Managing Model Integrity." In: *ECMFA.* Springer. 2010, pp. 245–260.

[13] Douglas C Schmidt. "Model-driven engineering". In: *COMPUTER-IEEE COMPUTER SOCIETY-* 39.2 (2006), p. 25.

[14] Misha Strittmatter et al. “A Modular Reference Structure for Component-based Architecture Description Languages”. In: *2nd International Workshop on Model-Driven Engineering for Component-Based Systems (ModComp)*. CEUR, 2015, pp. 36–41. URL: `http://ceur-ws.org/Vol-1463/paper6.pdf`.

[15] Misha Strittmatter et al. “Towards a Modular Palladio Component Model”. In: *Proceedings of the Symposium on Software Performance: Joint Kieker/Palladio Days*. Ed. by Steffen Becker et al. Vol. 1083. Karlsruhe, Germany: CEUR Workshop Proceedings, Nov. 2013, pp. 49–58. URL: `http://www.kieker-palladio-days.org/`.

[16] Daniel Strüber, Michael Lukaszczyk, and Gabriele Taentzer. “Tool Support for Model Splitting using Information Retrieval and Model Crawling Techniques.” In: *BigMDE@ STAF*. 2014, pp. 44–47.

[17] Daniel Strüber, Matthias Selter, and Gabriele Taentzer. “Tool support for clustering large meta-models”. In: *Proceedings of the Workshop on Scalability in Model Driven Engineering*. ACM. 2013, p. 7.

[18] Daniel Struber et al. “Splitting models using information retrieval and model crawling techniques”. In: *International Conference on Fundamental Approaches to Software Engineering*. Springer. 2014, pp. 47–62.

[19] Daniel Strüber et al. “Managing Model and Meta-Model Components with Export and Import Interfaces.” In: *BigMDE@ STAF*. 2016, pp. 31–36.

[20] Markus Völter et al. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, 2013.