Flexible Graphical Editors for Extensible Modular Meta Models

Master's Thesis of

B.Sc. Michael Junker

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:	M.Sc. Heiko Klare

12. April 2016 – 11. October 2016

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, 10.October 2016

. . . .

(B.Sc. Michael Junker)

Abstract

In model-driven software development, graphical editors can be used to create model instances more efficiently and intuitively than with pure XML code. These graphical editors rely on models created on the basis of a meta-model. If such a meta-model is extended invasively not only its code has to be re-generated but also the graphical editor needs to be adapted. When developing multiple extensions, the meta-model as well as the corresponding graphical editor tend to get complex and error-prone.
One way of coping with this complexity is to use modular meta-models and extending them noninvasively. However, having multiple meta-model fragments providing extended features is only half the job as equivalent graphical editors are needed as well.
This master's thesis therefore analyzes different types of extensions for meta-models as well as on graphical editor level. Next, a short analysis of extension mechanisms follows. These mechanisms are used for different realizations of extension types. Like the extension types, the mechanisms are also analyzed for both meta-models and for graphical editors. While the classification of extensions resembles one part of this thesis' concept, their mapping from meta-model level to graphical editor level marks the second part. This mapping is done in order to show possible impacts of a meta-model extension to its corresponding graphical editor.
To validate this concept, the analyzed mappings are implemented exemplarily in two different frameworks. Furthermore, the two prototypes show the different possibilities each framework has to offer when it comes to their capabilities of extension. Therefore, this thesis can also be seen as guideline for extending a given graphical editor.

Zusammenfassung

Im Bereich der modellgetriebenen Softwareentwicklung werden oftmals graphische Editoren eingesetzt um die Entwicklung von Modellen zu erleichtern. Diese graphischen Editoren sind dabei auf das den Modellen zugrunde liegende Metamodell angepasst. Falls dieses Metamodell invasiv erweitert wird, muss nicht nur der gesamte Code erneut generiert werden, der graphische Editor muss dann ebenfalls an die Erweiterung angepasst werden. Im Falle mehrerer Erweiterungen wird so nicht nur das Metamodell, sondern auch der graphische Editor komplex und fehleranfällig. Weiterhin kann es passieren, dass Endnutzer nicht alle Funktionalitäten des Metamodells und des Editors nutzen möchten, sondern nur einen Teil davon. Weitere Erweiterungen senken dann unter Umständen die Attraktivität des Gesamtprodukts.
Eine Möglichkeit dieser Komplexität entgegen zu wirken, ist die Einführung von modularen Metamodellen und damit einhergehend auch Erweiterungen nicht-invasiv zu gestalten. Nichtsdestotrotz sind modulare Metamodelle auch nur dann sinnvoll, wenn die graphischen Editoren auch entsprechend umgesetzt sind, da ansonsten dennoch die gesamte Funktionalität der Metamodelle in einem Editor steckt.
Aufgrund dieser Faktoren beschäftigt sich diese Masterarbeit mit den verschiedenen Typen von Erweiterungen auf Metamodell-Ebene sowie auf Ebene der graphischen Editoren. Neben der Klassifikation der einzelnen Erweiterungstypen wird dabei auch auf einzelne Mechanismen eingegangen, die zu bestimmten Erweiterungstypen führen können.
Während die Klassifikation der Erweiterungen eine große Rolle in dieser Arbeit spielt, müssen die Erweiterungen auf beiden Ebenen noch in Zusammenhang gebracht werden. Dabei werden mehrere Abbildungen geschaffen, die es erlauben von einer Metamodellerweiterung auf mögliche Erweiterungen auf graphischer Editorenebene zu schließen. Dadurch lassen sich die verschiedenen Auswirkungen einer Erweiterung auf Metamodellebene auf graphische Editoren besser erkennen.
Validiert wird dieses Konzept durch eine exemplarische Implementierung der Abbildungen in zwei verschiedenen Frameworks. Weiterhin zeigen die beiden implementierten Prototypen welche Möglichkeiten und Grenzen die beiden Frameworks aufweisen. Dadurch kann diese Arbeit auch als Richtlinie betrachtet werden, mit Hilfe derer die Entwicklung einer Erweiterung vereinfacht werden soll.

Contents

List of Figures

List of Tables

Listings

1 Introduction

1.1 Motivation

Nowadays, software systems tend to grow and get increasingly complex. If we take a look at the lines of code for a Windows operating system, we see that there have been four to five million in 1993 and with the release of Windows XP in 2001 there were already around 40 million [39]. While four to five million lines of code are hard to manage, 40 million seems almost impossible. Furthermore, there are countless additional features implemented in further versions of the operation system, which also increases its complexity. One way to approach this ever increasing complexity is by introducing model-driven software development (MDSD) [33]. In MDSD we use meta-models in order to describe domain-specific languages [54]. However, like actual code, these models also have to be maintained. Since a software systems usually lasts a couple of years, these models also have to evolve. One approach to handle that evolution is to expand the originally used meta-model leading to a larger meta-model, which then can be hard to understand or contains features most of the users won't even use. The approach we pursue during this thesis is to create meta-model fragments and use these as extension, to the original meta-model. That way if a user only wants to use the core features without any extension, the extensions can just be left out.
A problem that occurs when using the second approach is that the meta-model fragments are only useful if they reflect in every other part of the system as well [9]. In the context of the Palladio Component Model [3], there is the approach to modularize the current meta-model. According to Strittmatter et al [56], not only the meta-model modules are important but also their impact on the simulations or graphical editors. One part of a software system that is covered in this thesis are graphical editors. If the meta-model consists of many different modules but the graphical editor needs every single one of these modules to function, there is no gain in using meta-model fragments instead of directly extending the core meta-model. Therefore, the graphical editor must also be flexible enough to handle these meta-model extensions on editor level.
Of course, there are already different approaches considering the extension of meta-models or the extension of graphical editors. Authors such as Jiang et al [26] propose different extensions for meta-models in general, while Heinrich [21] focuses on the extension of business process editors. Furthermore, there are approaches combining the two types of extension. In [58] for example the complete IDE can be extended through an extension of the meta-model.
Existing approaches however, mostly generate a graphical editor automatically out of the given meta-models. That means, that if a meta-model is extended a complete regeneration of the graphical editor is necessary. This editor contains the core meta-model and the

extension. Aside from the fact, that the automatic generation can only be done for limited domains or simple editors, like tree editors or generic class diagrams, the user is again forced to work with the complete graphical editor, although he doesn't need certain extensions. The approach presented in this thesis should therefore not generate a graphical editor out of a meta-model but give the developer the freedom to create and combine their own versions of meta-model and graphical editor extensions. Achieving that the user is free to choose which extension he considers necessary for his project and which can be left out. Therefore, we first present a classification of possible extensions on meta-model level as well as on graphical editor level. After that we map the meta-model extensions to graphical editor extensions showing the degree of freedom the developer has when considering an extension on meta-model level. Furthermore, this mapping together with the evaluation can be seen as guidelines as to how extensions of graphical editors can be created given a certain meta-model extension.
To validate the given approach we implemented two prototypes with the help of two different frameworks which both use the same meta-model. The meta-model is composed of different modules while each module represents specific extensions that are classified within this thesis. Those extensions are then implemented for both prototypes in order to show, that our classification and mapping is valid for an extension on meta-model level and its representation in a graphical editor extension. Thereby, we do not only implement these extensions on their own but furthermore show that these extensions can be combined freely according to certain rules. The frameworks used are Graphiti [12] on the one hand and on the other hand Sirius [13]. Both of them are extensions to the Eclipse IDE and based on the Eclipse Modeling Framework (EMF).

1.2 Goals of this thesis

The goals for this thesis can be expressed as a number of research questions that are answered by this thesis. The first research question that arises is how extensions can be classified in general and then of course, how they can be further classified on meta-model level as well as on graphical editor level. The next question that has to be answered is how these extensions are mapped together on the different levels. Furthermore, the question of realization must be answered showing which mapping actually is possible to implement with the given frameworks. The last research question can only be answered by the two prototypes. Assuming there are models containing information of every available extension and try to load these models when only the basic graphical editor is available. What happens with the remaining information unknown to the graphical editor? Is it left out, shown as incomplete information or in the worst case, can the model no longer be opened?

1.3 Outline

This thesis is organized as follows. Chapter 2 captures foundations for the following chapters. Thereby, we address the field of model-driven software development. This field

also includes the terms *model* and *meta-model* as well as a short overview on domain-specific languages. Since this thesis focuses mainly on modularity and extensibility of meta-models as well as graphical editors both terms are also addressed in this chapter. The chapter concludes with an overview on both frameworks that were used for the creation of the prototypes.
Chapter 3 introduces related work on the field of meta-model and graphical editor extension. Furthermore, work on the creation of modular meta-models is discussed while another section deals with language workbenches which are workbenches with the purpose of creating DSLs.
One of the main contributions of this thesis is addressed in chapter 4 where a classification of extensions on both meta-model and graphical editor level is made. This chapter includes general, platform independent classification of extensions as well as concrete extensions for EMF meta-models or Graphiti editors.
After the classification of extensions is made the chapter that follows deals with the mapping of those extensions between meta-model level and graphical editor level.
Chapter 6 then shows the implementation of both prototypes answering the research question of which mapping actually can be implemented for which prototype. In both prototypes we implement one core editor based on a core meta-model and overall three extensions to the core editor which are also based on meta-model extensions referring to the core meta-model. Botch implementations can be seen as guidelines as to how different extensions can be implemented.
After the implementation we can analyze both frameworks on the aspects of what features are still missing and which framework should be preferred under which circumstances. Furthermore, we can also theoretically analyze further scenarios to prove that the mapping presented in chapter 5 does also hold for different scenarios. This is all done in chapter 7. At last we summarize this thesis and give an outlook on future work.

2 Foundations

This chapter covers the fundamentals of this thesis. In the beginning, we first give an overview on model-driven software development and the terminology and technology associated with it. After that, the key term *modularity* is explained. This chapter ends with an introduction to the technical foundations and frameworks used during the implementation of both prototypes.

2.1 Model-Driven Software Development

Models can have a wide variety of applications such as code generation, deriving further artifacts or documentation. In case of a documentation purpose the *Unified Modeling Language* (UML) [18] can be used to create diagrams illustrating the system. However, if the system evolves, the code changes making the UML diagram inconsistent with the code. Additional effort must then be conducted, in order to ensure the consistency of the diagrams with the code and the other derived artifacts. Since this task is an additional effort, it can also be seen as a burden to the developer.
In contrast to *model-based* software development as mentioned above, model-driven software development (MDSD) uses models not only for documentation purposes, but as key artifacts. Those models are 'abstract and formal at the same time' according to Stahl et al. in [54] p.14. This abstraction can be expressed as a reduction to the essence, meaning that a model in MDSD contains the same information as the final program code, but in a much more compact form. In order to receive valid code in the end of the modeling process, transformations are needed. For software developers the approach of first creating models and then transforming them into valid source code, has the advantage that it reduces the system's complexity. That is the result of the possibility to work on the compact and easier models than on the actual program code.
So far we covered the aspect of abstraction, but left out the aspect of formalism. In order to apply a mode-to-model or a model-to-code transformation, the model to be transformed has to meet defined criteria. Those criteria are defined in a meta-model, which can be also thought of as a domain-specific language (DSL), which describes how models of a specific domain should look like.

2.2 Models and Meta-Models

For a better understanding of the concepts in model-driven software development, a few definitions and their context are needed. The definitions include model, meta-model and meta-metamodel, which are illustrated in figure 2.1.

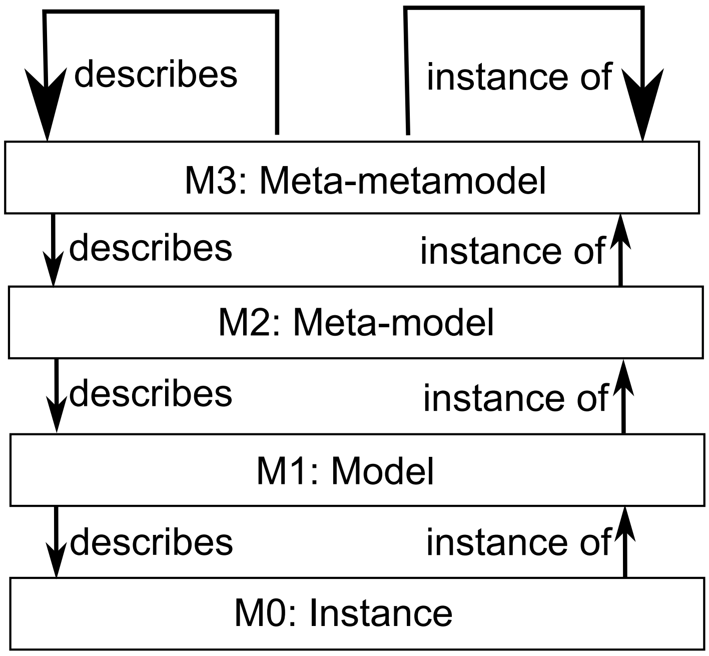


Figure 2.1: The four meta-levels of OMG described by Stahl et al. [54]

In order to describe those levels in a more concrete way, the levels are explained with the help of an example. As represented in the figure, the lowest level is the instance level, which is the actual object. Exemplary speaking, this could be an actual car. The following paragraphs describe the further levels M1 to M3 with the help of the example *car*.

Model A model can be seen as an abstraction of a real world object. According to Stachowiak [53], a model is a formal representation of an original that fulfills the properties of abstraction, homomorphism and pragmatics. These properties mean that a model abstracts from unnecessary details the original has since not every detail is actually needed, when creating models. For example, when modeling the car that should serve as toy for children, the gear drive is an unnecessary detail, but the car still should have tires. Furthermore, when creating a valid model, the developer is not allowed to give the model additional features the original doesn't have. This is covered by the property of homomorphism, which states that statements on the model also hold for the original. In our car example, unless real cars can't fly, the model car should also not be able to transform into a plane. The last property mentioned is pragmatics. This property simply means that the model always serves a specific purpose. In our example this is the purpose as children's toy.

Meta-Model With the help of a meta-model it is possible to describe models or in other words: Every model is an instance of a meta-model. A model can thereby only be created with constructs that are defined in the meta-model. Stahl et al. [54] states that a meta-model consists of the following parts:

- *Abstract Syntax*: The abstract syntax describes the elements a valid model can consist of, independent of the representation. Throughout this thesis, elements belonging to the abstract syntax of a meta-model are called *meta-classes*.
- *Concrete Syntax*: While there is only one abstract syntax of a given meta-model, there can be various concrete syntaxes. Concrete syntaxes may be different graphical representations. For example, classes in UML could be drawn with rectangles or with circles. Even textual representations as concrete syntax are possible.
- *Static Semantics*: These are semantics that can be evaluated without executing the model itself. Static semantics can for example be expressed as constraints.

When referring to the car model example from above, the meta-model of such a car would describe how valid car models can be created. This description may contain a color and the number of tires and seats a car can have. As a constraint, we can also say that the number of tires must be either three or four in order to be a car. This constraint belongs to the static semantic, since it is evaluable without executing the model.

Meta-Metamodel As the meta-model describes valid models, the meta-metamodel is used to describe meta-models. Meta-metamodels should be self-describing to prevent endless conformance sequences. Although, the *Object Management Group* (OMG) standardized the meta-metamodel *Meta Object Facility* (MOF) [17], a more common meta-metamodel that is used in practice is *Ecore*. Ecore is an implementation of the *Essential MOF* (EMOF) standard and is implemented as part of the *Eclipse Modeling Framework* (EMF) [55]. The meta-model and its extensions presented in this thesis are based on Ecore and are implemented in EMF.

As already stated, meta-models are used in order to describe models, which, in model-driven development, can be used for code generation. Although meta-models are a general concept, during this thesis they are always referred to describe a language leading to the interchangeability of these two terms. The reason behind this interchangeability is that meta-models can be considered a language describing aspects of a (software) system specific to a domain.

2.3 Domain-Specific Languages

As mentioned in section 2.1, in MDSD meta-models can be used to describe domain-specific languages. Martin Fowler [14] defines a DSL as 'a computer programming language of limited expressiveness focused on a particular domain' p.27. The key element of limited expressiveness is in contrast to general-purpose languages (GPL), such as Java or C++, which is why DSLs are usually used to build only certain aspects of a system, but not the entire system. Since DSLs are compact languages and focus only on a single domain, they are valuable programming languages for that domain and should therefore be preferred instead of GPLs. According to Fowler [14], DSLs can be divided into three categories.

- *External DSLs* are languages separate from the main language of the application it works with. Examples for external DSLs are thereby regular expressions, SQL [57] or the Palladio Component Model (PCM) [3].
- *Internal DSLs* are DSLs, which use general-purpose languages in such a way that scripts in an internal DSL are valid code in its GPL, but only use a subset of the language's features, in order to handle one aspect of the overall system. An example for this style is Lisp [43].
- *Language workbenches* are special integrated development environments (IDE) used for defining and building DSLs. Furthermore, language workbenches support working with self-defined DSLs as they come with IDE support. A few examples, which are also discussed in section 3.3, are Eclipse Xtext [4] or Jetbrains MPS [58].

Since DSLs usually cover one aspect of a system and therefore apply only to a small domain, there are a lot of different DSLs. Some of them are listed by van Deursen et al in [8], but since this paper came out in the year 2000, there are even more DSLs. For this thesis, DSLs built upon meta-models in a model-driven environment and support the use of graphical editors are primarily relevant. Although DSLs of every category mentioned above, could be of interest, language workbenches have the advantage that they usually come with additional graphical editors for model support. When creating internal or external DSLs graphical editors may still have to be designed and implemented.

2.4 Modularity

Modularity is a general concept and has been applied to a wide field of applications as for example shown in [27], where Jinghua et al investigate the concept of modularity in different sciences, such as social sciences and natural sciences. A definition of modularity for software systems is thereby given in the IEEE standard [25]:

- **Modularity**: The degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.

To put in another way, modularity is simply 'the degree to which a system's components may be separated and recombined' as stated by Schilling [47]. If analyzed from another perspective, every system can be characterized by some degree of coupling between its components. The higher the degree of coupling between the system's components, the lower the degree of modularity and vice versa. In component-based software engineering (CBSE) [1], components can be composed to modules, whereas a module encapsulates certain functionality. Not only in CBSE modularity is a topic, but also in programming languages such as Java, which already comes with concepts to support modularity as shown in Poo et al [42]. A single class can for example represent a module encapsulating certain functionality. To build the bridge between Java and CBSE, classes can be composed to components, which can be their own module or, as already stated, can be composed to bigger modules.

Since modules have a high internal cohesion because of the encapsulated functionality, but low coupling to other modules, they can be changed or substituted with other modules without a bigger impact on the rest of the system. When trying to achieve a high degree of modularity, however, Meyer et al [37] suggested five criteria, which indicate the degree of modularity. Decomposability of the problem into sub-problems, composability of modules to new systems and understandability of a module in isolation are three of the criteria of modularity. The fourth criterion is that small changes in a module should only have localized effects, while the last criterion is that faults should stay isolated in the module. These criteria apply for systems, as well as all artifacts associated with the system, such as the meta-model. There are numerous ways of creating a modular meta-model, which is discussed in section 3.2. An advantage of modules that was only marginally mentioned so far, is their capabilities of extension. When a meta-model is a self-contained entity, which a module is, it can be easily extended by every developer without regard to other extensions or compositions. Those extensions are discussed later in detail in section 4. At first, the terms extension and extensibility are explained in the next section.

2.5 Extensibility

There are various ways to define the term extensibility depending on its application context, which makes it even more important to create a common basis, since the terms extension and extensibility are used throughout the entire thesis. Just like the term modularity, there is also an IEEE definition on the term extensibility in the same standard [25].

- **Extensibility**: The ease with which a system or component can be modified to increase its storage or functional capacity.

Synonyms for extensibility are *expandability* and *extendability*. The main focus in this thesis, however, lies rather in increasing the functional capacity than in increasing the system's storage.
While extensibility describes the ease of modifying a system or component, the extension is the actual modification. Extensions are usually implemented as a result of changing or new user requirements during the system's evolution. According to Selmeci et al [50], there are different ways of how to engage these requirements. There are solutions, where there is no need of extending the system, such as configuration or personalization of options. Solutions that require extensions can be further categorized. *Modification* for example, means that the standard software is altered, which requires a deep understanding of the software. As modification on meta-model level leads to a larger and more complex single meta-model, it is not further discussed in this thesis. The other two options of extending a system are *enhancing* and *add-ons*. Enhancing means implementing predefined entry points. The original system is not changed, but the power of the extension is dependent on the entry points. An add-on on the other hand is a special set of objects, which are built on a specific version of the application (or meta-model) that offer additional features. Both of these solutions can be applied to both meta-models and graphical editors and therefore the focus in this thesis lies on these two.

2.6 Eclipse Modeling Framework

The *Eclipse Modeling Framework* (EMF) is part of the Eclipse IDE [55]. As already mentioned in section 2.2, the example meta-model and its extensions are based on Ecore and are implemented in EMF. This section therefore deals with EMF and its features. Creating a meta-model based on the EMF is very similar to creating meta-models in the UML. In addition to *meta-classes* and packages, different types of references, such as compositions or inheritance can be modeled. A meta-class can furthermore contain different attributes or operations, while each attribute has a data type. Data types can either be primitive, such as integer or Boolean or other meta-classes as well. Operations, instead, can be pictured as methods in java composed of a return type, a parameter list and a name.
If a meta-model should be extended, there is always the possibility to add the extension directly in the current meta-model in form of new packages or classes within the meta-model. The other way of extending a meta-model in general is creating a new plug-in containing the extension, which references the current meta-model. To preserve the understandability, the latter approach should be preferred. Among defining meta-models in MDSD approaches, EMF also comes with a generator tool with capabilities to generate source code out of existing meta-models. If changes have to be implemented, the developer can simply adjust the meta-model and regenerate the code out of it. The generated code manifests in different Eclipse plug-ins. It is possible to generate up to four different plug-ins for each meta-model generation.

- **Model Plug-in**: This plug-in includes all interfaces and corresponding classes to the Ecore packages, classes and enums that are described in the meta-model. For every class, there is also an additional interface generated to provide the program to an interface design principle [15].

- **Edit Plug-in**: The edit plug-in contains the UI-independent portion of the editor code in terms of item provider classes for each meta-class. Furthermore, EMF generates sample icons to represent the classes.

- **Editor Plug-in**: This plug-in contains the UI-dependent portion of the editor code. The three plug-ins already mentioned facilitate building models with the help of a tree editor instead of writing XML code.

- **Test Plug-in**: The last generated plug-in contains test code for each entity in the meta-model. For testing the JUnit framework[1] is used.

Although, it is possible in EMF to create models with a tree-based editor, developing own diagram-based graphical editors can come with more advantages. In case the meta-model is complex and offers a lot of different features, such as the PCM, a self-developed graphical editor can reduce the complexity when creating models.

[1]http://junit.org/junit4/

2.7 Graphical Editors

The idea of graphical editors supporting the development process is not new. In 1990 Brad Myers [38] already came up with a taxonomy on visual programming and program visualization. Visual programming thereby 'refers to any system that allows the user to specify a program in a two-(or more)-dimensional way' [38]. In program visualization on the other hand, the program is specified in the conventional textual way, but uses graphics to illustrate aspects of the program. In this thesis we only focus on graphical editors to support the process of creating and maintaining models in a model-driven environment. Since those models are usually transformed into executable code and are then compiled or interpreted into one-dimensional streams, graphical editors in this thesis are seen as program visualization. The advantages of graphical editors in comparison to pure textual coding are numerous. According to Smith [52] and Rimes [45], program visualization supports program understanding. This becomes clear, when thinking about the human nature. Humans can memorize and understand pictures way faster than textual representations, such as code. Together with a better program understanding, graphics tend to be a higher-level description of the desired actions, which makes the programming task a lot easier [38]. Furthermore, Shneiderman claims that the user has the impression of directly constructing the program instead of abstractly design it [51].
As already stated in the previous section, when using EMF the data that the user wants to visualize is the domain model. This can be done by using the EMF generator tool to open every domain model based on the defined meta-model with a tree-based editor. This tree-based editor can already be considered a graphical editor. However, there are other representations, such as the entity representation in a diagram, on which this thesis focuses. An entity representation of a domain model always consists of the graphical elements representing the domain model elements and their connections to each other. Graphical elements can be considered as nodes or containers depending on whether they contain other graphical elements or not. Connections are usually referred to as edges between nodes or containers. Those terms are interchangeable throughout this thesis.
The development of a graphical editor for entity representation as it is done in this thesis can have further advantages. The editor still uses the same underlying business domain model as the tree-based editor leading to the fact that if a graphical element is deleted from the editor canvas, it is also deleted from the domain model. This makes it possible to only work with both the self-defined graphical editor and the generated tree-based editor. While the generated editor displays all elements either with diamonds or icons a self-defined graphical editor can use all kinds of graphics to display certain model objects. Furthermore, a graphical editor as it is presented within in thesis comes with additional features such as different views. Views such as a properties view listing all attributes of the selected element or an outline view giving an overview on the current model are also available in the tree-based editor but other views such as the palette showing all elements that can be added to the diagram aren't. In addition to views the layout of the graphical elements can also be changed to improve the understandability. All in all, when referring to a graphical editor in this thesis all the editors' aspects are meant including the canvas containing the graphics, the toolbar, the properties view, the outline view, the palette view and possible further views belonging to the editor defined by the framework

or the developer. Another term for graphical editors in this thesis are graphical modeling workbenches.
In order to evaluate the approach in this thesis, we implemented a prototype in two different frameworks. Both frameworks, Graphiti and Sirius, are explained in the following two sections.

2.7.1 The Graphiti Framework

Graphiti is an Eclipse-based graphics framework, that enabled rapid development of state-of-the-art diagram editors for domain models [12]. Relevant for this thesis is that Graphiti can deal with EMF-based domain models easily but can also deal with any other Java-based objects on domain side as well. Although, Graphiti utilizes the *Graphical Editing Framework*[2] and *Draw2D*[3] for diagramming the user only needs to know Java coding and EMF to use the framework. To add functionality to the editor in development the user implements so called *features*. These are used for example for displaying *pictogram elements*, the graphical representation of objects in Graphiti. There are standard features the user can implement or in case additional functionality is needed there is also the possibility to implement custom features. A list of all standard features is given in the following.

- **Add**: The add feature makes it possible to add an existing model element to the diagram.

- **Create**: Besides adding an element to the diagram the create feature also creates a new corresponding business object in the underlying model.

- **Update**: When updating the business model the graphical diagram is not updated as well. In order to ensure an update on the diagram as well the update feature for these elements representing business objects has to be implemented.

- **Move**: General movement of graphical elements is already implemented. With the help of this feature however it is possible to restrict the movement of those elements. An application for this feature would be annotated elements that should always be in distance of 10 pixels to other elements for readability purposes.

- **Remove**: Remove is basically the opposite of the Add-feature. It only removes the pictogram element from the diagram but not the corresponding object in the business model.

- **Delete**: While the Remove-feature is the opposite of the Add-feature, the opposite for the Delete-feature is the Create-feature. Not only the pictogram element is removed from the diagram but the corresponding object in the business model is also removed.

[2]https://eclipse.org/gef/
[3]https://www.eclipse.org/gef/draw2d/

- **Resize**: Like the Moving-feature a standard implementation for resizing pictogram elements is also given. A useful application for redefining the resizing behavior can be that it is not allowed to make the pictogram element smaller than the total length of the characters representing the name of the element.
- **Layout**: If pictogram elements can contain other pictogram elements their representation might be confusing. With the help of the Layout-feature the user can choose how to arrange those elements depending on which and how many elements there are.

2.7.2 The Sirius Framework

Like Graphiti Sirius is also an Eclipse-based graphics framework [13]. Although both frameworks aim for an easy development of graphical editors and the visualization of EMF models their approaches are different. While the user needs to know Java coding when using Graphiti this is not necessary when using Sirius. Of course, it is possible to write Java code to add additional functionality to the editor but the main focus of Sirius lies in the definition of a model which defines the complete structure of the graphical editor the user wants to develop. The user can choose whether he wants to create a diagram, table or tree editor but for this thesis only the diagram editor matters. As already mentioned the main focus of Sirius lies in the definition of a model. This model is defined in a *.odesign* file. Besides defining the look and behavior of model elements the user can even choose different layers to represent the data on. On the default layer the basic structure of the model elements could be displayed while on an additional layer additional behavior compartments to existing elements could be represented.
When starting to define a graphical editor in Sirius a domain class has to be set for the diagram which resembles the starting point. In order to define further model elements the user has to navigate through the meta-model starting from the chosen domain class. Besides variable, feature or service expressions Sirius also offers Acceleo[4] expressions which are based on the Eclipse implementation of OCL [16].
Besides creating new diagram elements such as nodes, containers, edges or decorations to be displayed in the canvas of the graphical editor the user can also add new tools, new customizations or import existing elements. Customizations allow for adding additional styles to existing elements. These styles apply when certain properties are evaluated or selected. The import mechanism works for every diagram element that has already been defined in the same odesign. This is useful when many domain model elements have similar properties. With the help of the import mechanism these properties have only be defined once and can be reused any time. When creating a new tool a new section is created. Depending on the content of the section it is displayed in the palette view of the graphical editor or not. Tools can also be used for defining behavior on graphical elements, adding new menu buttons or adding additional functionality by using plain Java code. Getting to know the exact features of these tools is part of chapter 6.

[4]https://eclipse.org/acceleo/

3 Related Work

This chapter deals with related work for the presented approach. First of all, different meta-model extensions are discussed and analyzed. This is later used as a foundation on how meta-models can be extended in general. The next section considers modular meta-models covering also possibilities to modularize existing meta-models. Then, language workbenches are discussed as they cover the aspect of creating meta-models as DSL. Furthermore, they also come with textual or graphical editor support for the developed DSL. The next section deals with graphics frameworks, such as the two frameworks used for the implementation of the presented approach. In the last section of this chapter, extensions to graphical editors itself are discussed and how they relate to the approach in this thesis.

3.1 Extension of Meta-Models

There are various ways to extend an existing meta-model. This section covers general notes on the extension of meta-models, as well as concrete scenarios. Nevertheless, all of these approaches consider mainly one, at most two different extension mechanisms. Within this thesis, we combine all these papers and analyze all of the different mechanisms for their capabilities of extension.
According to Jiang et al. in [26], there are four different types of extension mechanisms the UML has to offer, differentiating in what the user is allowed to change. While the first level allows for manipulating the original meta-model, the second level is defined in a way an extension is used within this thesis. The original meta-model can not be manipulated, but not every element of the extension must necessarily have a parent element in the core meta-model. This supports for a modular use of meta-models. The next two levels further constrain the extensions and are not further considered. Although, Jiang uses the term *extension mechanism* it should not be confused with extension mechanisms presented in this thesis, since these concepts differ from each other.
In general, different sorts of meta-model extensions are possible. One quite obvious extension is extension by inheritance as presented by Schleicher et al [48] and Danilo et al [2]. Existing meta-classes in the core meta-model are thereby simply inherited and extended by additional functionality in the meta-model extension. Schleicher et al. uses thereby inheritance, when extending the Business Process Model and Notation meta-model to support compliance scope, while Danilo et al adds further annotations that represent quality aspects. Another way of extending an existing meta-model, is by either referencing meta-classes in the core meta-model or realizing interfaces as shown in the documentation of TOGAF [19]. Referencing can have different impacts on the extension, which are addressed in section 4.4.1. One last possible extension, that is also analyzed within this

thesis, is extension by applying the UML Profile mechanism as presented in the work of Ko et al [30]. Another important paper on this approach was published by Kramer et al [32], where the EMF-Profile mechanism is applied to the *Palladio Component Model*. This profile mechanism is also addressed later, when dealing with the classification of extensions in chapter 4 and during the implementation in chapter 6.

3.2 Modular Meta-Models

The whole basis of the approach presented in this thesis, relies on having a modular meta-model. Therefore, we should discuss general usage, composition and extension of modular meta-models within this section. When we use the term modular meta-model, we refer to a meta-model consisting of different modules, while each module is an encapsulation of functionality from other modules as presented by Colombo et al. in [6]. Furthermore, it is important to Colombo, that modules are communicating with mostly one element from other modules. Although, communication with as little entities in modules as possible is desirable, there are different approaches as to how this communication should work, especially when modular meta-models should be extended by other modules.
The authors in [29], [24] and [60] suggest building modular meta-models with the help of meta-model fragments and interfaces. Since those interfaces are used as extension points for further meta-model fragments, they support the information hiding principle. Being able to hide different information, makes it possible to use meta-models as black-boxes. Given that, developers extending the meta-model don't have to get full insight in the complete structure of the core meta-model. Kelsen et al [29] and Hessellund et al [24] only focus on the conceptual idea of modular meta-models, while Zivkovic et al [60] also suggests mechanisms to extend the meta-model. Those mechanisms are interface realization and interface subtyping but there is no suggestion given of when to use which mechanism. Within this thesis, we also cover the aspect of the different impacts each mechanism has.
A different approach is used by Henriksson et al [23], where grammars are extended in such a way, that they resemble modules. Furthermore, this approach is extended by using a transformation from the presented grammar to meta-models, whereas meta-models, due to this transformation, have the following properties. Inheritance is only used to express grammatical types, not for feature inheritance meaning that no parent meta-class has any features, such as attributes or operations. Meta-models used in this thesis, resemble grammatical types as well as they support feature inheritance. The second property mentioned by Henriksson et al [23] is that all aggregations in the generated meta-model are compositions, leading to a tree structure of the meta-model. Although, both the inheritance and referencing extension mechanisms presented in this paper are constrained, this approach can be applied to any language according to Henrisson et al.
The last approach presented by Weisemöller et al in [59] extends the MOF 2.0 itself in such a way, that required and provided interfaces can be implemented in the meta-model. Achieving that, meta-models and components in component-based software development can be designed in the same manner, leading to modular meta-model fragments.
Extensions presented within this thesis, can also be considered meta-model fragments,

as they always refer to at least one entity in the core meta-model or to another extension, which itself refers to the core meta-model. The core meta-model is itself the only meta-model not being a fragment, as it doesn't need to communicate with its extensions. Furthermore, instead of restricting meta-models to only a few extension mechanisms, in this thesis every mechanism can be used to create a communication between the core meta-model and its modules. Nevertheless, this also assumes that each user knows what he is doing as all internal meta-model entities can be extended.

3.3 Language Workbenches

In [58] Jetbrains MPS is presented. Jetbrains MPS is a language workbench, with which it is possible to define custom languages and their IDEs. This includes ways to extend, modularize or compose the language. Everyone of these building blocks is divided into structure and syntax, the type system and the generation of the building block. Extension of languages and editors is thereby presented as extension by inheritance. In this paper extension means adding new information to existing elements. Another type of building block presented in this paper is reuse. Reuse can be achieved by using templates in form of abstract classes which are extended. Those extensions work well for MPS, but in general, there are more extensions applicable to editors and meta-models which are addressed in chapter 4.
MetaEdit+ [41] is a graphical workbench also used for creating and using domain-specific languages and code generators. Meta-models can either be created graphically or form-based with a meta-modeling language created by MetaCase [40]. While language creation works good with MetaEdit+, there is a lack of modular language evolution. The only way of meta-model evolution can be achieved by changing the meta-model directly, leading to an adaption of the generated code. That is one of the main aspects we want to avoid during this thesis. Fortunately, even if elements in the meta-model are deleted older instances can still be produced, since the information stays in the instance. There is just no way further instances of the removed element can be created.
Another language engineering environment worth mentioning is MontiCore [31]. MontiCore is parser-based and can generate parsers, meta-models and editors based on extended grammar. Furthermore, two different extension mechanisms for languages are supported, which are grammar inheritance and embedding. As in MetaEdit+, those extension mechanisms apply only to the developed language, as the editors are generated from the language definition. That means, that no extensions to the editors itself are intended.
One of the last workbenches discussed is Spoofax/IMP. Spoofax/IMP is a meta-tooling suite providing DSLs for describing editor services [28]. These editor service descriptions can then be used to generate Eclipse plug-ins. The generated editors are purely textual but can be composed since a generated editor in Spoofax/IMP is always a module. Composition is thereby the only way to extend a language but has the drawback that once two editors are composed they are dependent on each other which I want to avoid during this thesis.
The last tool worth mentioning here is Xtext [4]. Xtext is built on top of the Eclipse IDE and also uses source editing, instead of graphical editing. Like some of the other workbenches

already discussed the user can not only develop his own language with Xtext but it also contains a framework for the generation of Java code.

3.4 Extension of Graphical Editors

This section discusses related work for the extension of graphical editors, as well as modular structures of graphical editors that could possibly extended. Thereby, not only graphical editors in the context of DSL modeling are considered. Although, there are plenty of language workbenches providing their own graphical editors, extending these editors is barely considered. That is due to the fact, that they are generated on the basis of the defined meta-model. However, as mentioned in section 1 there are situations where explicit extensions of graphical editors apart from meta-model extensions are needed.
As mentioned, graphical editors have a wide field of applications which is not restricted to the modeling of DSLs. Fejes et al [10] for example, present a graphical editor for man-machine interfaces of dynamic systems. The editor itself is build in a modular way where each module can be extended separately. The modules cover thereby pictures representing subsystems or processes, associations between those pictures, menus, icons, fonts and rules. These are all valid extension types for graphical editors described in that paper. However, we only deal with diagram representations as graphical editors, which leads to further and different extension types. Therefore, the work in Fejes et al [10] does not suffice for our context.
When modeling business processes, the work of Heinrich [21] introduces quality requirements as symbols. These symbols are an extension of a graphical editor for the *Business Process Model and Notation* (BPMN). A symbol for example, can thereby indicate the maturity of an activity based on errors that were found during a certain period of time. However, based on an underlying meta-model, there are not only symbols but further types of extensions for a graphical editor, that we cover within this thesis.
Another work from Refsdal compares the two Eclipse frameworks GMF and Graphiti [44]. The graphical modeling framework (GMF) is, as Graphiti and Sirius, an Eclipse-based framework to visualize models. He noticed that both of these frameworks have capabilities for extension but it is a lot easier in Graphiti since one can work with plain java code and doesn't need to know the internals of GEF and Draw2D as the underlying rendering engine.
Instead of comparing two different frameworks Lehrig extended a given meta-model and its graphical editors by architectural templates in [36]. The editor extension relies on applying different *Profiles* to the meta-model and therefore also to the editor. Although, this is also one aspect within this thesis we also cover other possibilities to extend a meta-model and its corresponding graphical editor.
The last paper to discuss is the work of Ruscio et al [46], who automates the propagation of domain-model changes by GMF model adapters. This paper uses a similar approach to the approaches of most of the previous discussed language workbenches. In the paper, GMF is extended in such a way, that the editor is automatically adapted when changes on meta-model level occur. As mentioned earlier, in this thesis we stride towards a flexible

graphical editor that isn't generated with all extensions, whenever the meta-model is extended. In that way, the user is not forced to use every extension available.

4 Classification of Extensions

As already mentioned in section 1, modular meta-models can only be of use if the other parts of a system are equally flexible. In this master's thesis, we propose a general concept for flexible graphical editors, that should help developers decide how to extend their graphical editors, if the meta-model is extended. Therefore, the first section deals with the basic approach I presumed during this thesis. Furthermore, one has to differentiate between the concept of *extension types* and *extension mechanisms*, which is described next within this chapter. In the following, all extension types and mechanisms that are considered throughout this thesis are described first on meta-model level and then on graphical editor level.

4.1 General Approach

Before going into the details of this approach, the general concept should be described. Therefore, figure 4.1 illustrates the proposed approach in an UML-like notation. When beginning to design a new system in model-driven software engineering, the developer starts off by developing the meta-model. In this thesis this is called the *core*. As soon as the core meta-model is developed, a graphical editor can be implemented in order to design domain models in a graphical manner. When an extension for the meta-model is needed, the developer has two options. Either the extension is directly added to the core meta-model in the same file or a new file containing the extension is created. The first option would lead to a larger core meta-model, which we do not want, as already discussed in chapter 1. In order for the second option to work, at least one class of the extension has to somehow connect to the existing core meta-model. This connection can be called an extension mechanism and is explained in the next section. The important part of extending the core meta-model is, that the extension references the core meta-model, but not the other way around. That way, it is still possible to add or remove extensions without an impact on the core meta-model. The core meta-model can then be considered modular, since a change in one of the extensions has no impact on the core meta-model or the other extensions. Furthermore, a change in the core meta-model only affects those extensions, that are connected to the changed classes. Of course, on graphical editor level, the extensions must work in the same way. As we can see in figure 4.1, in this approach graphical editors are realized in the same manner. There is a *core graphical editor* with different extensions. The extensions have knowledge of the core graphical editor but not the other way around. Furthermore, the core graphical editor only references the core meta-model, so that the core graphical editor can only display the core meta-models class instances. The same applies for each extension on graphical editor level. Each extension only references its corresponding meta-model extension. The last feature of the approach

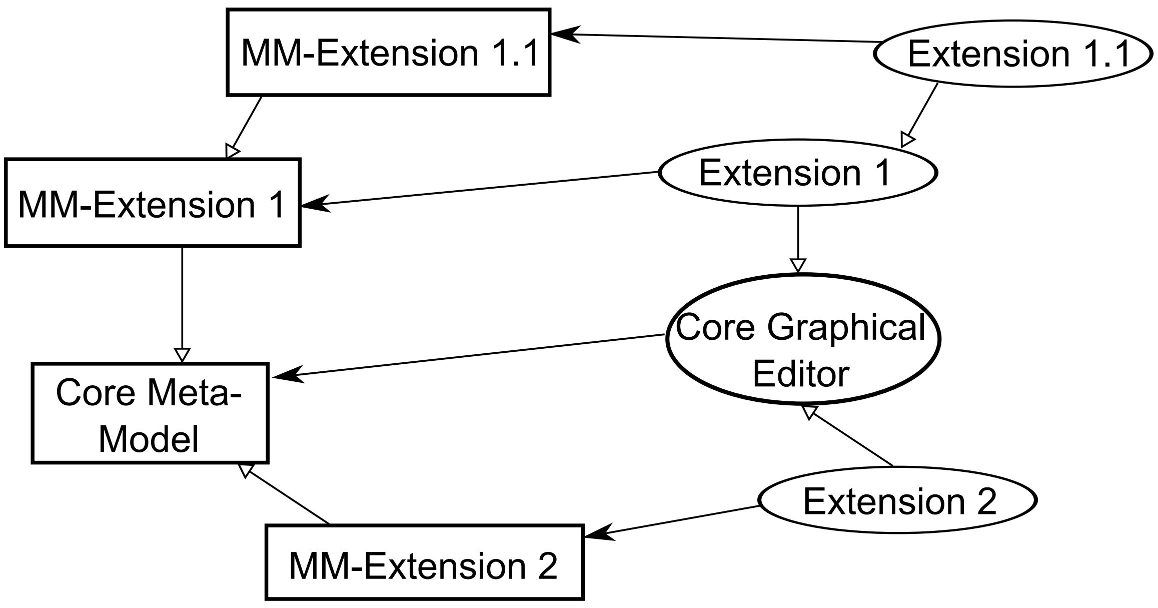


Figure 4.1: General approach showing extensions on meta-model and graphical editor level

shown by the figure, is an extension of an extension. In the figure, meta-model extension 1.1 extends the meta-model extension 1. This also works, since extension 1.1 knows indirectly about the content of the core meta-model, as it references extension 1, which references the core meta-model. The same applies again for the graphical editor. There is only one quite obvious feature concerning this aspect. Removing extension 1 would lead to the removal of extension 1.1, which of course leads also to either the removal of graphical editor extension 1.1 and 1 or at least to their deactivation, since a graphical editor can not represent anything that isn't there. A more interesting question is what happens if all meta-model extensions exist but the user only uses a subset of the graphical editor extensions. This question is technology dependent and therefore answered later in chapter 7.
After discussing the general approach of this thesis it is important to analyze how those extensions actually work. Therefore, a characterization of extensions is needed which is analyzed in the next section.

4.2 Extension Types and Mechanisms

As already stated in section 2.5, an extension can be characterized in various ways depending on the system's context. There is also the possibility to further classify the general concept of extension in order to get a more distinct view on this topic. Therefore, the terms *extension mechanism* and *extension type* in context of the presented approach are introduced in the following.

- **Extension Mechanisms** define how something is extended. Depending on the context and the technology used there can be a lot of different mechanisms that can be used to realize an extension type. Since those realizations are implemented in applications, components or other systems extension mechanisms are highly platform dependent. In the context of this thesis I concentrated on extension mechanisms for EMOF-based meta-models as well as for the frameworks Graphiti and Sirius.

- **Extension Types** are a more general concept. Extensions can be classified into different groups representing one type of extension. In contrast to extension mechanisms, extension types are platform independent meaning that it doesn't matter whether Sirius or a different framework is used. Although, extension types are independent of the platform they are still context dependent. A simple example would be an automobile and a building structure. An extension type for a building structure could be the vertical transportation of people whereas this extension type can not be applied to automobiles. Furthermore, an extension type can have multiple extension mechanisms realizing the type. Referring to the building structure, the vertical transportation can be realized by an elevator or a flight of stairs.

When putting the terms extension type, extension mechanism and the general term extension all together, we are making the restriction that an extension contains at least one instance of an extension type. In other words, an extension implements at least one extension type. Otherwise there is no increase of the previous functionality. Furthermore, an extension type is realized by an extension mechanism, which is dependent on the underlying platform. Since an extension type can be realized by more than one mechanism, the mechanism is also dependent on different quality aspects, such as usability, complexity or understandability. Nevertheless, there can be different forms of extensions. We therefore do not require, that a meta-class in the extension targets a meta-class in the core meta-model with a relation. We also consider meta-classes in the extension referencing each other with no direct relation to one of the meta-classes in the core meta-model as possible extension types.
In the following sections, extension types and mechanisms for both meta-models and graphical editors are listed and explained. For meta-models we consider only EMOF-based meta-models as they are the most common ones in model-driven software development. Furthermore, extension types for graphical editors mostly apply to Eclipse based diagram editors but there are other graphical editors where these extension types can also be applied to. Extension mechanisms are thereby also analyzed for EMF-based meta-models, the Graphiti and the Sirius framework.

4.3 Extension Types on Meta-Model Level

Since we are only considering meta-models based on the EMOF meta-metamodel, we can infer the possible extension types considered in this thesis from the *ecore* meta-model. Figure 4.2 thereby shows a simplified version of the meta-model. The first division in this simplified meta-model are the sub classes of *ENamedElement*. On the one hand there is the *EClassifier* sub-class dividing further into *EDataType* and *EClass*, whereas an *EClass* can either be abstract or an interface. An *EClass* can have multiple super-classes as indicated by the reference. On the other hand, there is the *ETypedElement* class, which

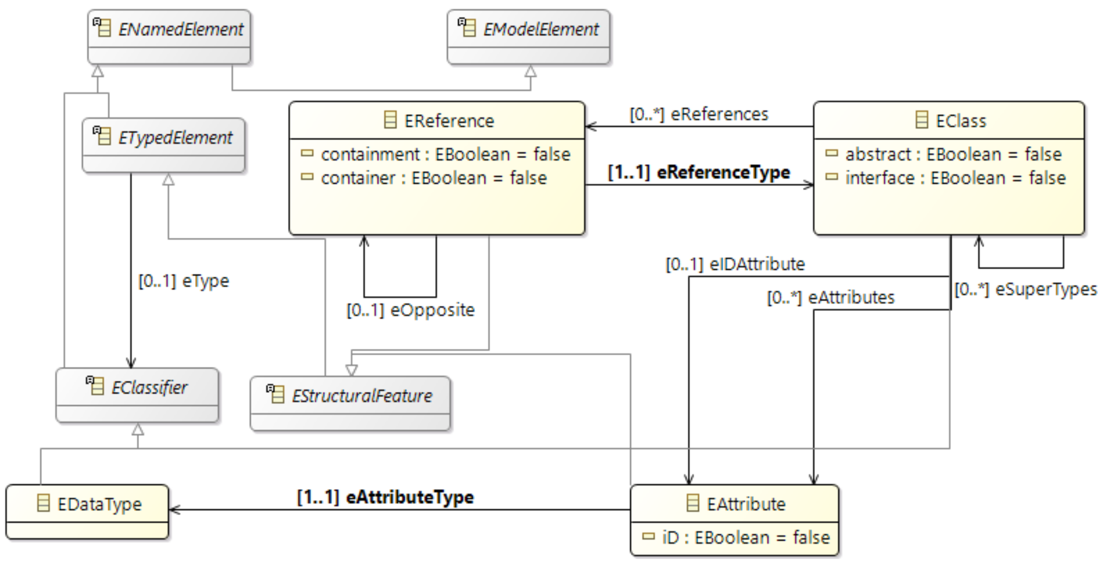


Figure 4.2: A simplified version of the ecore meta-model based on the complete meta-model in [55]

can have an *EClassifier* as type. Going down in the inheritance hierarchy, we have the *EStructuralFeature* sub-class, which divides into the sub-classes *EReference* and *EAttribute*. An attribute must always have a certain data type, while the reference always references an *EClass* as type. Furthermore, the reference can be a containment or a container.
When identifying possible extension types, we concentrate on the three meta-classes *EClass*, *EAttribute* and *EReference*, as these are not abstract and most commonly used when designing a new meta-model. Analyzing both the given meta-model and the related work in section 3.1, it becomes clear, that there are basically only two different extension types on meta-model level. Adding new information to an existing class or adding a new *EClass* as meta-class. Those types however, can be divided into more detailed extension types. Both extension types are further analyzed within the next two sections.
For illustration purposes figure 4.3 shows a minimalistic example containing all discussed extension types and mechanisms that realize these types for EMF-based meta-models. The figure is divided into five parts containing a core meta-model and its four independent extensions. The core meta-model only defines a DSL where persons can be defined. A person has a name, a unique identifier and an arbitrary number of relatives. A relative is thereby defined by his or her degree of kinship to the person standing in relation to the relative. Furthermore, a relative can visit another person and has certain topics to talk about when having a conversation with a certain person depending on the degree of kinship. The topics method in the relative class is abstract making the whole class abstract. The three remaining extensions that are provided in the figure are discussed within the next sections, where the extension types and mechanisms for EMF-based meta-models that we use in this thesis are explained.

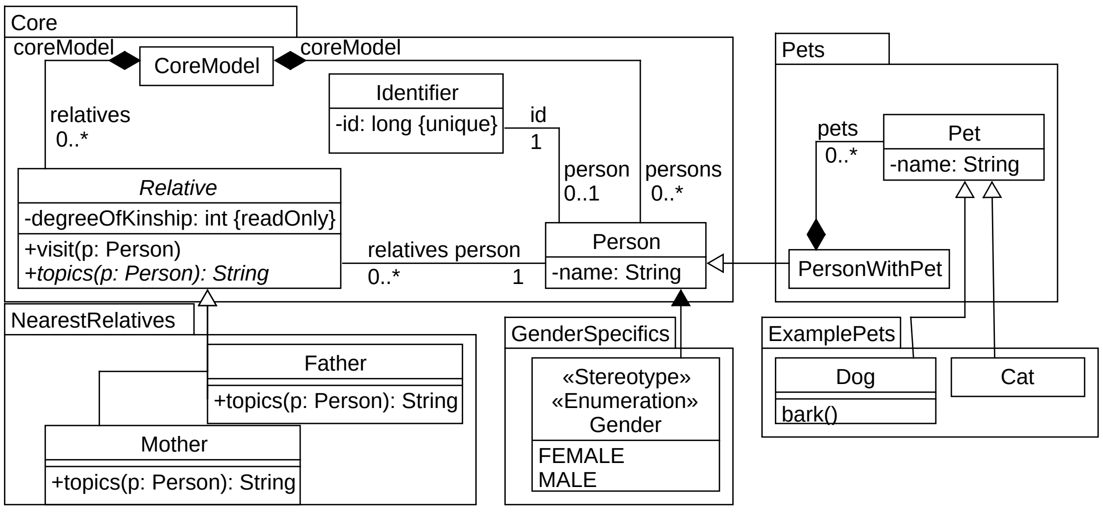


Figure 4.3: An exemplary core meta-model and three fragments extending the core meta-model

4.3.1 New Meta-Class

Whenever we create a new extension on meta-model level, we most likely create new meta-classes (except for a stereotype only extension) as we are not allowed to alter the existing core meta-model. On graphical editor level, extending a meta-class could therefore lead to all kinds of extension type implementations in the mapping. Therefore, we need to figure out how to best divide this extension type into further sub-extension types. One way of doing so is to differentiate between whether the meta-model extension should be used in the same editor as the core meta-model or if it should be represented in its own graphical editor. This, however addresses only the intention on graphical editor level and is therefore only considered in chapter 5 where the mapping is analyzed. Another point which is considered later this chapter in section 4.4.5 is the combination of this extension type together with the other extension types on meta-model level. Depending on the combination, different extension types are actually realized. Nevertheless, we divide this extension type depending on the developers intent meaning that we assume we are allowed to intrusively extend the core meta-model. If we were allowed to do that, would the meta-class now be represented as attribute, containment or as new meta-class somehow referencing the core meta-class? Whenever we would intrusively add another meta-class to the core meta-model as an extension, we say that the extension type of adding a new meta-class is fulfilled. Otherwise, one of the three other extension types considered in this section are fulfilled. From now on, when referring to the meta-class extension type, we always refer to the assumption that we can intrusively extend the given meta-model and add a new meta-class given the developers intention. Furthermore, the new meta-class is only counted among this extension type if new domain-specific information is added to the core meta-model. A counter example can be constructed given figure 4.3. Assuming there is a meta-class extending only the *Identifier* meta-class, which

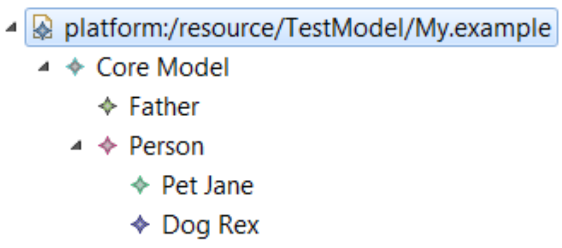


Figure 4.4: The extension type of adding a new meta-class

relates to the *Relative* and *Person* meta-class, such an extension would rather not add any new domain-specific information and can therefore not be counted as a new meta-class. One way to further differentiate this extension type apart form what we already did is considering the instances of the new meta-class on the model level. When referring to the tree editor of EMF-based meta-models we can say that a new meta-class is added when an instance of this meta-class would be added on the same level as the extended meta-class given an intrusively extended meta-model. A simple example based on the meta-model and their extensions in figure 4.3 is given by figure 4.4. Thereby we can divide this extension type into two subtypes where both have different realizations on graphical editor level. One of these subtypes is creating a new meta-class instance one level below the root node of the model while the other subtype is creating a new meta-class instance as part of another instance except the root node. Here we assume that the root node of the model is also the root node for our graphical editor. If the root node in the graphical editor is a different one than in the model, the assignment of both extension types may change. In that case, all meta-classes, whose instances are below the model instance root node don't matter anymore as the root node for the graphical editor is a different one. On the other hand, some of the other meta-classes may then switch to be instances below the graphical editors new root node. All in all, the editor's root node is crucial for the classification of the new meta-classes and these two extension types therefore are only valid when also considering their mapping to the graphical editor. Both types are further discussed within the next two sections.

4.3.1.1 New Meta-Class Instance Below Root Node

Given figure 4.4 with the corresponding meta-model in figure 4.3, we can see that any of the given extensions of the *Relative* meta-class leads to the new meta-class extension type. All instances of the *Father* or *Mother* class are shown one level below the root node, the *CoreModel*-object, which is the same level any other instance of a meta-class extending the *Relative* class would be listed.

4.3.1.2 New Meta-Class Instance as Part of Other Instance

Apart from creating new instances of meta-classes one level below the root element, the new meta-class extension type can also be applied if those instances are listed one level below any other instance given the condition that the new instance is on the same level as the extended meta-class. An example for this is also shown in figure 4.4, where an instance of the *Dog* meta-class is listed on the same level as the instance of the *Pet* meta-class. In the corresponding meta-model in figure 4.3, we can see that the *Dog* meta-class extends the given *Pet* meta-class in a different extension.

4.3.2 New Information to Existing Classes

Analyzing the new meta-class extension type we figured out that creating a new meta-class is actually dependent on the developers intent. Even newly created meta-classes in an extension may not be used as meta-class extension type. Therefore, we introduce a second main extension type being the adding of new information to existing classes in contrast to creating a new meta-class extension type. Since adding new information to existing classes is a generic term, further specification is needed. Therefore, we divide this extension type into three further types: adding a new attribute, adding a new containment and adding a new relation. All these types differ in some aspects which also reflects in the mapping of these extension types to those extension types on graphical editor level, which are discussed in chapter 5.
Depending on the information the user wants to add, different extension mechanisms can be used, which are discussed in the following sections. In case a non-primitive type other than a string is added as attribute and that type didn't exist before, there is of course the need in creating a new class in the extension representing that data type. However, this case doesn't satisfy the definition of the new meta-class extension type as the data type itself is considered to be a different extension type as the attribute who's type is the new data type. Keep in mind that all these extension types may either refer to the respective ecore meta-class in figure 4.2 or are the result of the developers intent when creating a new meta-class in the extension meta-model.

4.3.2.1 New Attribute

One of the possible extension types on meta-model level is adding a new attribute as new information to an existing class. The attribute thereby doesn't have to be listed when selecting an instance of the extended meta-class as long as the instance appears as property of the extension. Section 4.4.1 analyzes this type a bit further.
There are different attributes that are possible to add. Those can be primitive such as integer or Boolean, enumerations or other non-primitive types such as strings or other objects. When adding a new attribute to an existing class as part of an extension in the tree editor, this attribute is not necessarily shown as a new element but only as property of the extended meta-class given that the core meta-model is extended intrusively. Attributes are shown in figure 4.3 only as real attributes and not as meta-classes whose intent it is to attribute a given meta-class in the core meta-model. Examples for attributes are the *degreeOfKinship* or the *name* attributes.

4.3.2.2 New Containment

Adding a new containment is in a way the same extension type than adding a new attribute to an existing class. At first, when the new containment is added to an existing meta-class, it only resembles another attribute. The difference between the containment and the attribute mentioned in the previous section is that on model level the containment is shown one level below its container object, while the attribute is only shown as property of the container object. Since a containment is shown as a new object in the tree editor, this leads to more advantages. At first, the containment itself can have further attributes. Second, the containment can itself be used as a new container having even more elements as containment in further extensions. Given figure 4.3, a containment is shown in the *Pets* extension, where the meta-class *PersonWithPet* contains an arbitrary number of *Pet* instances.

4.3.2.3 New Relation

The extension type of adding new information is in this section given for introducing a new relation between meta-classes. Extending or altering an existing relation is not possible due to the restriction of not changing the core meta-model intrusively. Relations are always shown as references between two meta-classes. Not only the reference itself is important but also their role and cardinality. The importance of the latter is also described in chapter 5. The role of a relation, however, also gives an indication of the developers intent which is important especially if the meta-model extension developer is a different person than the developer developing the graphical editor extension. Given figure 4.3, the relation extension type is shown between any two connected meta-classes. The specific type of relation thereby represents the extension mechanism. However, those specific relations not only realize the relation extension type but can also realize other extension types, which are discussed in the next section.

4.4 Extension Mechanisms on Meta-Model Level

In this section all extension mechanisms on meta-model level are discussed. As already mentioned, extension mechanisms indicate how an extension type is realized. However, one extension mechanism can realize multiple extension types. Therefore, not only the extension mechanisms are discussed, but also how they relate to all extension types on meta-model level discussed in the section before. The extension mechanisms analyzed in the following are all considered during the implementation in chapter 6. We only cover the most basic extension mechanisms on meta-model level. Therefore, the list of extension mechanisms for meta-models may not be complete as extension types, such as decorations, are missing.

4.4.1 Referencing

Although, referencing can be one of the most simple methods to realize a certain extension type, the term referencing covers each type of association that can be used between two meta-classes. All those types need to be considered when analyzing the impact of the referencing extension mechanism. In TOGAF [19], for example, referencing with bidirectional associations can be used, while Henriksson et al [23] only allows compositions, when referencing between two entities. Since we are dealing with EMF-based meta-models, defining an aggregation between two classes is not possible and is not further discussed here. As already mentioned, referencing is done by adding an association between two meta-classes in the extension or one meta-class in the extension and one meta-class in the core meta-model. Thereby, a differentiation has to be made in which type of association is used. Because of that the first paragraph in this section deals with bidirectional associations, the second with unidirectional associations and the third paragraph focuses on compositions. Note that the relation extension type only gives a general notion that a reference would be used if the core meta-model would be extended intrusively. The reference extension mechanism always refers to one type of reference in particular such as a composition or an association.

Bidirectional Association A basic binary association is shown in figure 4.3 as association between the two classes *Person* and *Identifier.* Although this is not an extension of the core meta-model, we can see the purpose of a possible extension with a bidirectional association as extension mechanism. A bidirectional association can realize the new relation and both meta-class extension types. Since a bidirectional association can only be used between two classes in the extension, the intention of the developer doesn't change in case the core meta-model would be intrusively extended. Therefore, both meta-classes connected by the association are implementations of the one of the two meta-class extension types. Such an association can only be realized within two meta-classes in the extension, as otherwise the core meta-model would be extended intrusively, which we want to prevent. Assuming both meta-classes that are connected with a bidirectional association would realize each a new attribute, if the core meta-model was extended intrusively. Then, the association would still be needed as a relation between both attributes. The same argumentation holds for both meta-classes, if any or both would be realized as new meta-classes.

Unidirectional Association An unidirectional association is, in contrast to a bidirectional association, an association where we can only navigate from the source meta-class to the target meta-class but not the other way around. This type of association is not shown in the example above. In contrast to the bidirectional association, this type of association can occur between two meta-classes in the extension meta-model as well as between one class in the extension and another class in the core meta-model. The last one is shown in figure 4.5. The realization of the extension type with this mechanism is highly dependent on the developers intent. As the combination in figure 4.5 can also occur for both meta-classes being in the same core meta-model when intrusively extending the core. The association would then only realize the relation extension type, while the meta-class actually realizes one of the both meta-class extension types. On the other hand the example in figure 4.5 could also realize the attribute extension type. Then, both the *ExtClass* and the association, would disappear and the *CoreClass* would gain a new attribute, if intrusively extended. At last, the developer may also intent to realize the new containment extension type with the help of an unidirectional association. An indication for that is a cardinality of one and a further composition leading to another meta-class. Compositions are discussed within the next paragraph.

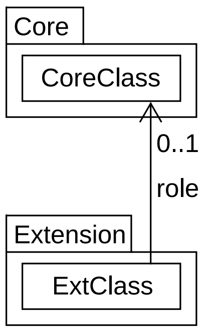


Figure 4.5: Two meta-classes in different packages related with an unidirectional association

Composition A composition defines a whole/part relationship between two classes. A part can be included in at most one composite at a time and if the composite is deleted, all of its parts are deleted. An example is shown in figure 4.3, where the *Pet* class is linked to the *PersonWithPet* class by a composition. This means that as soon as a pet exists, it has to belong to an owner, while a person with pets can have an arbitrary number of pets.
The inheritance of the person meta-class in this case is necessary, since we can not connect the composition in this context directly with the person meta-class. This would again postulate that the core meta-model knows about its extension, which we want to prevent. However, if we turn the composition around, a direct connection to the person class would be possible, but then a person would be contained inside a pet. There are situations, where such a use of compositions is semantically correct resulting in the realization of a new meta-class, which is a container for the meta-class in the core meta-model. In any case, using a composition as extension mechanism realizes the containment extension type for meta-models. That is due to the fact that on EMF model level, there will always be a new containment, when creating an instance of the extension. This also applies for compositions related to the root container meta-class. In our case, both the *Person* and the *Relative* meta-class are contained in the *CoreModel* class.
Although, a composition is a type of reference we wouldn't consider the composition realizing the relation extension type. That is due to the fact that only the meta-class and the composition combined realize the containment extension type.

As we can see, referencing allows for the realization of all extension types. However, the reference itself, can only realize the three sub-types of the new information to an existing meta-class extension type. Which type is realized exactly is on the one hand dependent on the type of reference (association or composition) and on the other hand on the type of meta-class that is created in the extension. Furthermore, as a meta-class needs to be created in order to use any type of reference to another meta-class any of the two meta-class extension types can also be realized depending on the developers intent.

4.4.2 Inheritance

Inheritance as extension mechanism is probably the classic method of extension. Like referencing the term inheritance also needs a distinguished examination as Danilo et al [2] or Schleicher et al [48] use extension on abstract or normal classes, while Zivkovic et al [60] suggests interface subtyping. Depending on the inherited class or interface this extension mechanism can also be used to realize more than one of the extension types for meta-models. The following three paragraphs thereby analyze each case.

Inheriting Abstract Meta-Classes In the example meta-model in figure 4.3, inheritance of the abstract class *Relative* is shown in the meta-model extension *NearestRelatives*. Both classes *Father* and *Mother* inherit the Relative class and implement its abstract method *topics*. This form of inheritance can be used to add a new meta-class. However, there is also the possibility to add a new containment, when inheriting an abstract meta-class. Assuming the core meta-model contains two meta-classes connected by a composition, where the composition's target is an abstract meta-class. Inheriting from this abstract meta-class leads to having a new containment as its abstract super-class is connected by a composition.

Inheriting Normal Meta-Classes Other than inheriting from an abstract meta-class, inheritance can be used on any other meta-class as well. This is done for example in figure 4.3, when inheriting from the *Person* or *Pet* meta-class. Usually this is done in order to receive a specialized version of the extended meta-class. In general, the extension type of adding a meta-class is also realized with this type of inheritance. Depending on the developers intend, the newly created class could also serve only for adding attributes to the class already in existence. The concrete realization is thereby dependent on the developers intent, if he were to intrusively extend the core meta-model. In addition to adding an attribute or a new meta-class, a new relation is also realized, inheriting a normal meta-class. This extension type is given, as the sub-class always references its super-class, because of the inheritance relation between those two. At last, there is also the possibility that a new containment is realized by inheritance under the same circumstances we explained the previous paragraph.

Inheriting Interfaces The last possibility a user has, when using inheritance is by interface subtyping. In EMF a meta-class marked as interface, always needs to be marked as abstract as well. Therefore, inheriting from an interface in EMF with another interface is equal to an abstract meta-class inheriting another abstract meta-class. Neither a new meta-class nor information to an existing class are added, as abstract classes or interfaces can be instantiated as model elements. Nevertheless, interface subtyping may be useful especially when planning to add an extension to the extension.

4.4.3 Realization

Realization means implementing a given interface and can in this case only be applied on meta-classes marked as interface. As already mentioned, when using EMF a meta-class can only be marked as interface, if it is marked as abstract as well. Therefore, realizing an interface in EMF is equal to inheriting from an abstract class with all its consequences on model level.

4.4.4 Stereotyping

The last extension mechanism for meta-models that is discussed in this thesis is stereotyping. In figure 4.3, the *Gender* enumeration stereotype is shown. When a stereotype is applied to an existing class, the class gets all the attributes and operations it had before plus every attribute and operation of each applied stereotype. In this case with the help of the *GenderSpecifics* extension, it can be defined whether a person is male or female. Since the content of a stereotype merges with the content of the classes the stereotype is applied on, there are no new classes available when instantiating a model but further attributes for those model elements the stereotype is applied to.
Stereotyping is the only extension mechanism not supported by the basic EMF editor. Since it is getting more and more popular to use stereotypes and apply profiles in UML, there is an extension to EMF that supports the profile mechanism. In EMF this is called *EMF-Profiles* and was developed by Langer et al [34], [35]. Furthermore, since the evaluation of this approach can be applied on the *Paladio Component Model*, we can use *MDSD-Profiles* [32]. MDSD-Profiles are specifically designed for the PCM environment and are therefore the obvious choice when implementing this extension mechanism.

4.4.5 Combination of Extension Mechanisms

Now that we discussed all extension mechanisms on meta-model level that are relevant for this thesis, we have to focus on the hierarchy of those mechanisms, as it is possible to use more than one extension mechanism on one extension. A meta-class in an extension could thereby be referenced to an existing meta-class in the core meta-model as well as it could realize an interface given in the core meta-model. As the different extension mechanism realize different extension types there is the possibility of a hierarchy. Since a stereotype can only be applied with the respective mechanism, there can not be a hierarchy when using stereotypes. However, the other three extension mechanisms can be combined, which makes an analysis of their hierarchy necessary.
One combination can be where two extension mechanisms are combined that realize the same extension type. In that case only this extension type is realized obviously. An example for that is an extension where the meta-class in the extension inherits a meta-class in the core meta-model and at the same time realizes a given interface in the core meta-model.
The next possible combination is referencing combined with either inheritance or realization. As we already discussed referencing can be used for all three extension types that belong to adding new information to an existing meta-class in the core meta-model. Assuming we use a composition on the one hand and on the other hand inheritance on two different core meta-classes. The composition in general realizes the new containment extension type while inheritance realizes the new meta-class extension type. If both mechanisms are used, then also both extension types are realized, as it is possible to add the new class as containment to the source of the composition and at the same time instantiate

		Extension Type			
		Attribute	Relation	Containment	Meta-Class
Extension Mechanism	Association	x	x	x	x
	Composition			x	
	Inheriting Abstract Class			x	x
	Inheriting Normal Class	x	x	x	x
	Realization			x	x
	Stereotyping	x	(x)	(x)	

Table 4.1: Summary of meta-model extension types and mechanisms realizing them

it anywhere the inherited meta-class could also be instantiated.
All in all, we can see that all extension mechanisms can be combined. Those combinations have only the effect that more than one extension type is possibly realized by a single meta-class in the extension. None of the considered extension mechanisms lies in any kind of hierarchy.

4.5 Summary of Meta-Model Extension Types and their Realizations

To sum up the previous sections, this section gives an overview on all meta-model extension types and the extension mechanisms for EMF-based meta-models realizing these extension types. For a concrete overview table 4.1 is given. The columns of the table thereby represent the extension types, while each row represents an extension mechanism for meta-models. As the table shows, an association can realize every extension type. Therefore, associations are highly dependent on the developers intent, whereas a composition always leads to realizing a containment. We do not distinguish here between unidirectional and bidirectional associations, as both can realize the same extension types as a bidirectional association can be split into two unidirectional associations.
The next extension mechanism, inheriting an abstract class, realizes only the meta-class extension type or the containment extension type depending on the meta-class the abstract meta-class is connected to. In case of inheriting a normal meta-class, the realized extension type depends on the developers intent, as it is when regarding associations. Since the realization of interfaces leads to the same goal as inheriting an abstract class this extension mechanism also realizes the new meta-class extension type and the containment extension type. The last extension mechanism, we discussed during the last sections, is stereotyping. Stereotyping with EMF-Profiles only aims for adding new attributes to existing classes, since once a stereotype is applied to a meta-class, instances of this meta-class are the same as before but with further attributes. As we use MDSD-Profiles, it is also possible to

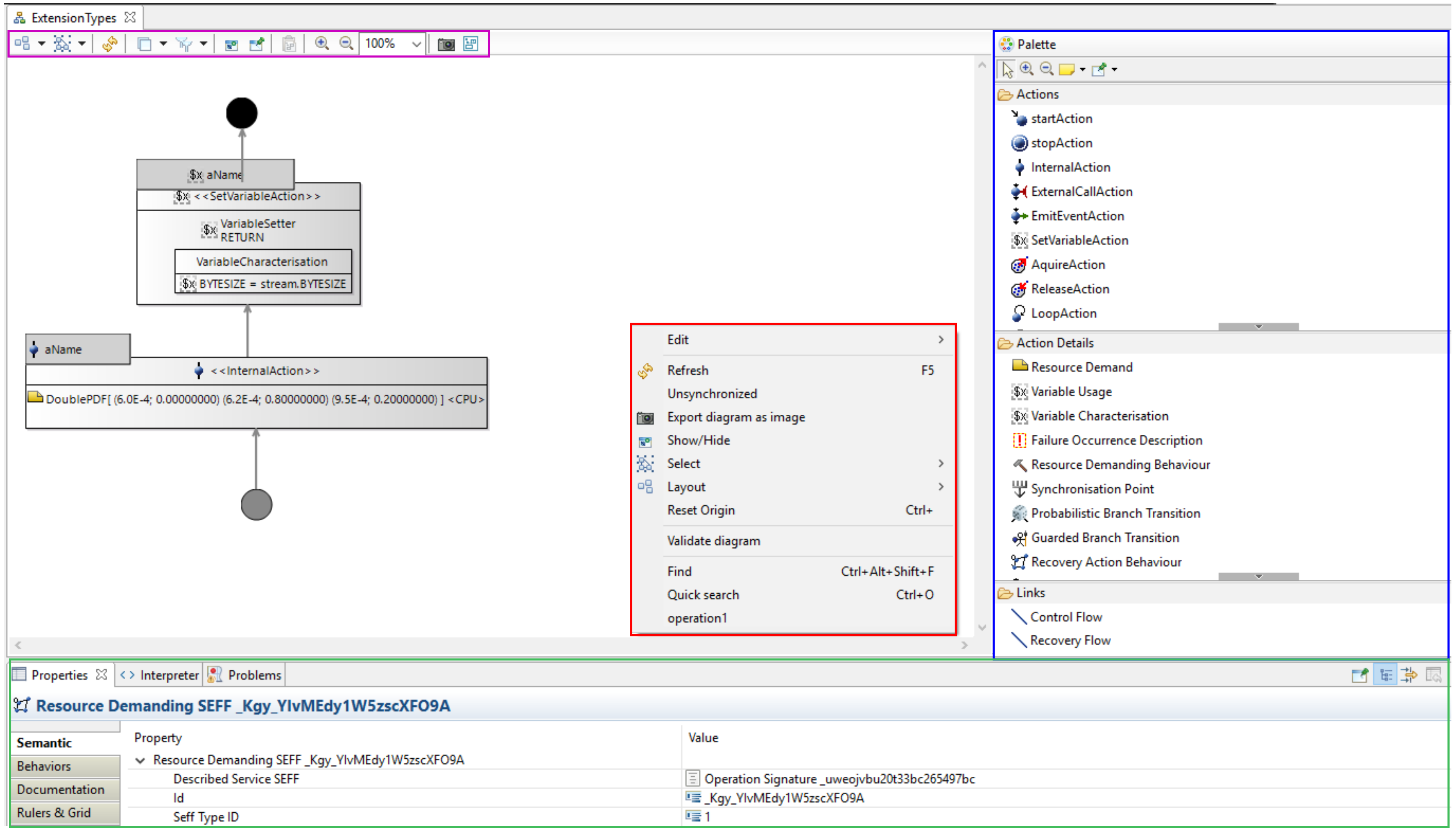
deposit relations within the stereotype. Therefore, not only the relation extension type, but also the containment extension type, could be indirectly realized. However, during the implementation we only regard stereotyping, when adding further attributes to an existing meta-class.

4.6 Extension Types for Graphical Editors

When describing extension types for graphical editors, we can also reduce those types down to the same two core types as for meta-models. One core type being adding new information to an existing graphical element and the other type being adding a new graphical element. Adding information can thereby mean that, for example, a new sub-node is added to the graphical element representing further information. In contrast, the extension type of adding a new graphical element means that the graphical element must exist independent of other existing graphical elements. A third extension type for graphical editors can be called the *structure extension type*, where the information of elements and elements itself stay the same with regard to contents, but the structure of the editor changes. This type includes changes in the layout or replacements of nodes or containers to other, possibly more comprehending structures. Since this type is independent of any extension type on meta-model level, it is only mentioned for the sake of completeness and not further divided or discussed.
Although, it is possible to reduce the extension types down to the two types mentioned above, there is no indication given, where those extension types are located in a graphical editor. Therefore, the list of extension types for graphical editors is extended by the location, where a type should be applied. A drawback that comes with this kind of partitioning is that extension types for graphical editors can not be fully generic, as different graphical editors tend to have different parts (e.g. some have a toolbar, others don't). In addition to differentiating between the location of each extension type, we also can divide the given extension types as to whether they are used for the representation of notation elements or if they can be used for the creation of elements or for altering their values. Having this in mind the following list applies to graphical editors based on the Eclipse IDE. However, many of these concepts can also be seen in other graphical editors. Furthermore, we only consider diagram representations in this thesis, as focusing on all different editors would be too complex. In order to get a better idea on what those extension types actually are, figure 4.6 shows a typical graphical editor as to how we see it. Extension types not visible in the figure are shown in the respective section.

4.6.1 Extend Existing Notation Element

An obvious extension type for graphical editors is the extension of an existing notation element. Whether the notation element is a node, container or connection does in this case make no difference. For this extension type only the representation of an existing notation element is extended meaning that possible creations are not discussed here. They resemble their own extension type, analyzed in section 4.6.3 and following.
The extension of an existing notation element can be divided into further extension types.



Resource Demanding SEFF _Kgy_YlvMEdy1W5zscXFO9A

Property	Value
Resource Demanding SEFF _Kgy_YlvMEdy1W5zscXFO9A	
Described Service SEFF	Operation Signature _uweojvbu20t33bc265497bc
Id	_Kgy_YlvMEdy1W5zscXFO9A
Seff Type ID	1

Figure 4.6: Extension types in graphical editors

One of these types being changing the appearance of a notation element in order to show further information, while another type being creating a new compartment. The next possibility a user has is to add a new annotation to an existing notation element. There are some cases not covered by the previous three extension types, which is why there is a fourth rather generic extension type, which is the representation of a sub-node or -container within the existing notation element. In all four of these cases we add new information to existing elements. All of these types are further analyzed within the next four subsections.

4.6.1.1 Change Appearance

The appearance of a graphical element can be used for the representation of certain aspects of the element in the underlying business model. Thereby, not only the color, but also labels are relevant. Changing the appearance of connections for example can indicate a different data flow or one, which capacities are at their limit. If the color of a node changes, this could indicate that a certain state of the node has altered.

4.6.1.2 New Compartment

A compartment in graphical editors can exist only inside a notation element. They are used to group sub-elements into predefined regions within a container. An example is shown in figure 4.6, where the *ResourceDemand*, characterized through the folder icon, is an element inside a compartment of the *«InternalAction»* container. A compartment can contain more than one element inside it. In this case it would be possible to add further *ResourceDemands* to the one already existent. Compartments can even be divided into further compartments making the compartment a container structure. Extending a notation element by adding a new compartment can be applied on container structures, as well as on nodes. While the extension of a container structure to support the compartment can be realized straightforward, it is a much harder task when extending a node to support a compartment. Nodes don’t contain any compartments, which is why the node itself has to be turned into a container structure first. As this is an intense intrusion to the specifications of the core graphical editor, the impact on further extensions to that node can not be foreseen. Because of that, adding a new compartment to a node should be avoided whenever possible and is not included within this thesis.

4.6.1.3 New Annotation

The next possibility we have, when extending an existing notation element, is to add a new annotation to it. In contrast to a compartment, an annotation does not define a region where similar elements are grouped. An example for an annotation is also shown in figure 4.6 on top of any of the two container nodes. Of course the annotation is not limited to be a border node on a container, but can be any node inside a container as well. The only condition is that the annotation represents an attribute of the existing graphical element and not a group of specific sub-elements or new sub-elements. Speaking in other words, an annotation only resembles information the user should be able to see to the full extend in the properties view, which narrows the annotation down to rather primitive data types or

strings that should be represented. In case of the *VariableCharacterisation* in the figure, we can say that the container element itself is not an annotation, but rather a new sub-container or a new compartment. This depends on whether it is possible to add further key-value pairs below the *BYTESIZE = stream.BYTESIZE* or if a new *VariableCharacterization* container is created. The key-value pair that is listed inside the container is by our definition an annotation, since this string resembles only an attribute of the *VariableCharacterization.*
Another sort of annotation is the introduction of a new symbol that is attached to a notation element. Symbols can represent anything, such as quality aspects as depicted by Heinrich [21]. In general, symbols should be preferred over changing the appearance of a notation element, when more than one extension can possibly be implemented for the same graphical element, as changing the color can apparently only be done once or otherwise information gets lost.
Like the compartment, an annotation can also be applied to both nodes and container structures, which comes with the same side effects in case the node has to be turned into a container structure first. On the other side, if just a bordered node should be added to the existing node, this usually is no problem, if there aren't that many extensions using bordered nodes. These would then start to overlap making it hard to read the inside of the bordered node.

4.6.1.4 New Sub-Node or -Container

Besides creating a new compartment for grouping similar elements or creating an annotation representing an attribute, we also can add nodes or container structures to existing elements for a different purpose. A new node or container could also resemble a new element which is a rather complex attribute of the existing notation element or is not even listed in the properties view. Figure 4.7 shows an example for notation elements within an existing container element that don't belong to any of the above mentioned extension types. As we can see on the right side of the *defaultUsageScenario* container, the *«ClosedWorkload»* is one example that doesn't fit into the category of being an annotation nor being a compartment. It can not be an annotation since it is obviously a container and can also not be a compartment, as there is no grouping of similar elements inside it. On the left side of the *defaultUsageScenario* container inside the compartment, there are other examples for new sub-nodes and containers which don't belong in any of the above mentioned categories for extension types. The top and the bottom node both resemble start or end nodes for processes that lie between them. None of these can be accounted for an annotation, as they resemble stand-alone elements inside the compartment. The last element is the *«SystemCallAction»* located between the start and end node. The same argumentation already used for the *«ClosedWorkload»* container can also be applied to this container. The *«SystemCallAction»* goes even further as it obviously has a compartment marked by the horizontal line.
All in all, we have now four different extension types, when it comes to the extension of existing graphical elements. All of these extension types can even be further extended either by one of the four types already mentioned or by other extension types, which are discussed during this section.

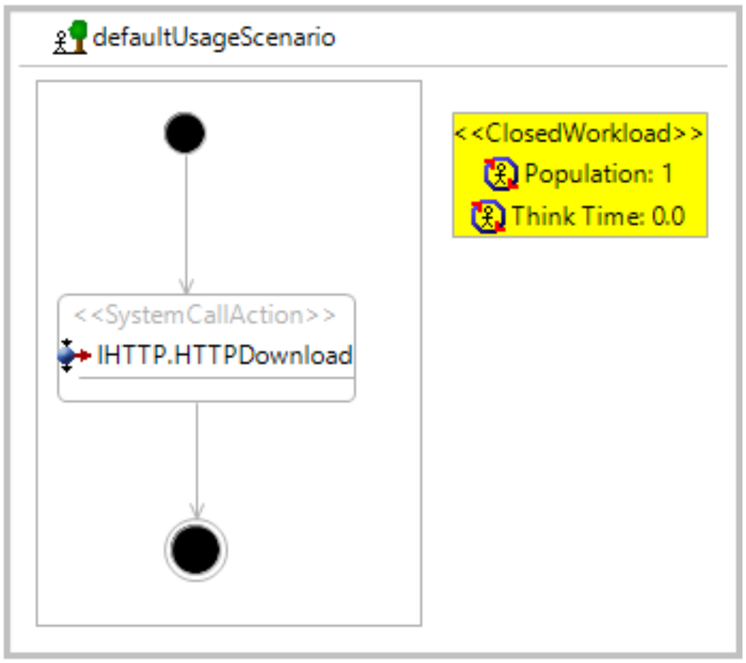


Figure 4.7: Excerpt from a usage model in the Palladio context showing notation elements inside a container

4.6.2 New Notation Element

While the previous section focused on extending an existing graphical element, this section focuses on representing notation elements that didn't exist in the core graphical editor. New notation elements, assuming they exist in the extension of the business model, are always shown directly in the diagram of the graphical editor, which is usually placed in the center of a graphical editor. Apart from representing elements of the underlying business model, the removal of a notation element also results in the deletion of the corresponding model element in the business model. Like in the previous section, we can also divide this extension type into further types. One type being the creation of a new node or container element, while the other type being the creation of a new connection. The node or container focuses on the representation of an instance element of the underlying business model, while the connection states the relation between two elements. However, if the corresponding meta-class' purpose is being a connection between other meta-classes this is also a valid reason for representing it in the diagram as connection. Furthermore, we do not differ between connections targeting notation elements inside an existing one, targeting notation elements represented directly in the diagram or a combination of both. However, when speaking of a new container or node we only mean new nodes or container structures directly inside the diagram but not inside an existing container. Creating nodes or containers inside existing containers resembles the new sub-node or -container or new compartment extension type.
Besides the extension type of creating a new notation element, there is usually also the extension type of a new palette entry realized at the same time, which is discussed within the next section.

4.6.3 Add Palette Entry

The palette is used to show the user the elements he can add to the editors diagram, where the graphical elements are displayed. Thereby, these elements are not only added to the editor but also created in the underlying business model. This is in contrast to the diagram itself, where notation elements only represent their corresponding model elements. In figure 4.6, the palette is bordered blue. Besides nodes or containers also connections can be listed within the palette that can be added to the diagram. Usually the palette displays all elements available for that graphical editor, while the diagram contains all elements already existent in the domain model to be edited. Since it is possible to add graphical elements to the diagram directly with the help of the palette, extending it also leads to the creation of new graphical notation elements like nodes, containers or connections. Furthermore, the second core extension type for graphical editors can also be realized with the palette. This is possible, since the palette can also contain elements that are used in compartments or as annotations within existing notation elements.

4.6.4 Add Properties Entry

The properties view in Eclipse is, as the name states, responsible for showing properties of a selected graphical element. In figure 4.6, the properties view is placed at the bottom of the editor and is bordered green. In the properties view not only properties are shown, but they can also be altered. Since only properties of selected elements are shown, an extension of this view can only lead to showing more properties of existing graphical elements. No new independent graphical elements can be shown, since those must be created in advance before the properties view can actually show their value. This means that only the core extension type of adding new information to existing elements can be realized by extending this view. Depending on the editor used, the properties view may update automatically, when an element extended by further attributes is selected in the editor. However, we don’t consider an automatic update as new properties entry in the sense of an extension type. Realizing this extension type therefore means that the entry can be added manually to the properties view.

4.6.5 Extension of Outline View

One of the only views not shown in figure 4.6 is the outline. An example outline is therefore shown in figure 4.8. The outline is used to give the user an overview on the diagram in focus. This also includes information on the underlying domain model. The outline is automatically updated, when new notation elements are added to the diagram. As discussed in the previous section considering the properties view, we don’t consider automatic updates as a realization of an extension type. Nevertheless, the outline view could also be extended. An extension of this view, however, can only lead to adding information to existing elements in the sense of altered representations. Of course, there is also the possibility to show new elements in the outline view that don’t exist within the actual diagram. This could be achieved by accessing and alternating this view’s code.

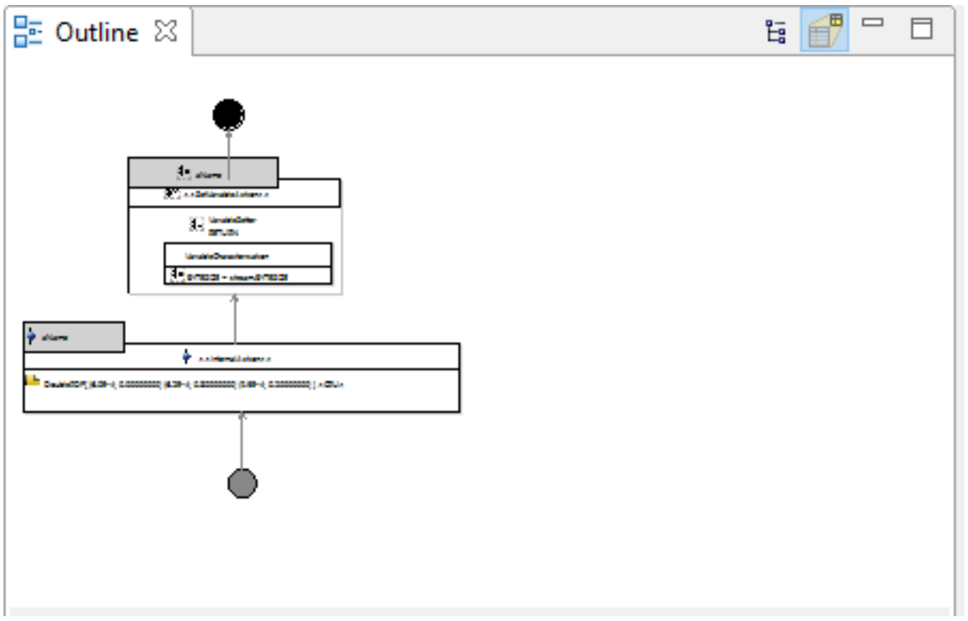


Figure 4.8: An exemplary outline view for a graphical editor

That would also violate the original purpose of this view and is therefore not considered in this thesis.

4.6.6 Add Toolbar Button

The toolbar is usually placed right above the graphical editor. In figure 4.6 it is bordered purple. Toolbar buttons can be used for various reasons. Giving the graphical elements, a better layout, showing the user helpful tips or hiding elements according to specific filters are only a few of those options. Extending the toolbar can either mean adding a new toolbar button and ,with that, new functionality or extending a button that already exists with further functionality. In either way the button can then lead to additional information that is shown for existing graphical elements. An example for this kind of extension is, as already mentioned, showing or hiding elements according to specific filters. While filters don't actually add attributes to existing elements nor create new elements, since only already existing elements are shown or hidden, another use for a toolbar can be used as extension type in the sense of this thesis. Thereby, a toolbar button automatically generates missing graphical elements of a specific type according to certain criteria. Another use for toolbar buttons is to change the state of all valid existing entities at once to save the user time.
The main difference between the toolbar and other extension types presented in this section is that the toolbar usually adds more than only one graphical element to the diagram. Therefore, it needs special requirements in order to be useful.

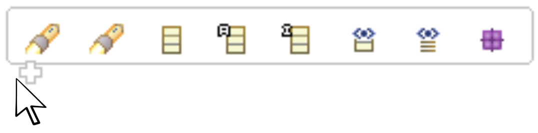


Figure 4.9: Mouse over extension type

4.6.7 Add Button to Context Dependent Menu

Adding a button to a context dependent menu like extending an existing notation element is a more or less generic term. We thereby regard context dependent menus that appear, when hovering the mouse on a notation element and pop-up windows that appear when right-clicking with the mouse on an empty diagram space or a notation element. Although we discuss both of these menus separately, they are the same extension type as it doesn't make any difference in the later mapping chapter.

Context Menu Button This part of the extension type of adding a new button to the context dependent menu is bordered red in figure 4.6. When a context menu opens due to a right-click on the diagram, besides general options or applying a layout to the whole diagram, it is possible to extend the pop-up window. Therefore, new elements can be added to the diagram. Usually a new button in the context menu isn't used for adding new elements, since it is user friendlier to add a new element via drag and drop from the palette, but for the sake of completeness it is still mentioned here.
If right-clicked on a selected graphical element, an extended context menu on the other side can add new information to the selected graphical element. In contrast to the toolbar this kind of extension always affects only one graphical element.
As mentioned before, using the palette to drag and drop a new element on the diagram or a graphical element is preferred over opening the context menu via right-clicking. However, there are still use cases for using such a context dependent menu. An example would be a list of similar elements, where one of them is automatically generated depending on other already existing elements. This could happen either within a certain graphical element or depending on all other elements in the diagram. Furthermore, such a context menu button can be used anytime a palette entry can not be used due to the framework's capabilities. Other than that we can use a context dependent menu to change an attribute's value.

Mouse Over Button Since this part of the above mentioned extension type is not shown in the overview figure of extension types for graphical editors, figure 4.9 shows an example of this type. Depending on the graphical framework that is used these mouse over buttons as presented in the figure are created automatically, as they resemble every palette entry that is valid for this location the mouse rests on. Apart from using mouse over buttons to add new notation elements to the diagram or new compartments to an existing notation element, a mouse over button can also be used to alter different information of an existing notation element. Exemplary speaking, an extension could use a mouse over button to set a Boolean attribute of a notation element to either true or false. In the same way this extension can also be used for changing the appearance of the existing notation element on demand. The difference between this extension type and the *change appearance* extension type discussed in section 4.6.1.1 is that the latter is only used for representational purposes. The mouse over extension type actually changes the appearance online when the button is pressed.

4.6.8 Create New View

Although it would be possible to extend the Eclipse platform in order to create a new view in terms of an Eclipse-based view, this is not the intention of that extension type. The views we are suggesting and analyzing, when speaking of this extension type are views that appear if and when the user, for example, double clicks on a notation element. The view should then manifest as either a wizard or a dialog window, where values of the notation element can be seen and optional edited or as a new editor, where further elements can be added. The base notation element then serves as a container for content not available in the core editor.
As we can see designing a new view can serve the purpose of adding information to an existing notation element, in case a dialog opens, where for example new key-value pairs can be altered. Furthermore, the other core extension type of adding new elements can also be realized with the help of a new view in case a new editor is opened.

4.7 Combination of Extension Types for Graphical Editors

All the above mentioned extension types for graphical editors can, like the extension types and mechanisms on meta-model level, also be combined. At first glance and with regard to the mapping, which is discussed later in chapter 5 the combination of extension types does not make any difference, as if only one extension type is used. This is due to the fact that only one extension type on meta-model level is realized with an extension type on graphical editor level. Multiple combinations of extension types on graphical editor level do not change the meta-model extension type. Of course, not every combination is possible or reasonable but this is discussed later in the mapping chapter.
Combining extension types has different advantages. In case a meta-model extension doesn't consist of only one meta-class but of more classes, which are related to the class serving as extension, using only one extension type in the graphical editor may cause the editor to get overly complex. Considering the work of Heinrich [21], where quality attributes are added to activities in a *Business Process Model and Notation* (BPMN) model. Since there are many quality attributes, such as security, portability, performance and so on it would make the notation element confusing for others to read, if more than one quality attribute is attached to the element. This is why symbols are used, which would resemble the change appearance extension type and when clicked on a symbol a new view opens containing information on that quality attribute. This way the complexity of the diagram is reduced by combining two different extension types.

4.8 Extension Mechanisms for Graphical Editors

This section includes the different extension mechanisms for both the Graphiti and the Sirius framework. Furthermore, it is also addressed which extension mechanism possibly realizes which extension type for graphical editors.

4.8.1 Extension Mechanisms in Graphiti

This section deals with potential extension mechanisms available for the Graphiti framework. There are basically two different extension mechanism. The first mechanism are extension points, a mechanism strongly supported by the Eclipse platform[1]. The second mechanism are plug-in dependencies meaning that besides a new plug-in containing the extension also a dependency to the core graphical editor can be created. Using the second extension mechanism results in further possibilities for extension, such as referencing or given certain extension points also inheritance or realization.

- **Extension Point**: Graphiti itself delivers only four different extension points that can be used for extending the framework itself. One is designed for exporting the diagram to a file, the other three for providing new diagram types or images meaning that these three can be used for actually covering extension types listed in the previous section. Fortunately, developers can define their own extension points leading to a large set of extensions to cover extension types, such as the toolbar or in general the look of a specific element. To cover the additional extension types, the developer has to use extension points provided by the Eclipse platform. Drawbacks of extension points are that some of them need a great understanding of the underlying Eclipse platform, which makes them complicated to implement. Furthermore, extension points for the toolbar or graphical elements have to be defined first. That requires a clear understanding, of where extension points should be placed regarding additional extensions.
  Since extension points are used all over Eclipse and Graphiti also supports them, it can be said that every extension type for graphical editors can be realized with the help of extension points. Although this is possible, it should be stated that the effort for realizing some of these extension types with this mechanism may be arbitrary high, whereas other extension types can be realized quite simply.

- **Plug-in Dependency** Since the Graphiti framework is based on features that have to be implemented, we can not simply reference a given feature and add information to it. However, we can reference all the existing notation elements and add further elements or different colors to it. Inheriting or realization on the other hand will only work if the inherited class is somehow added to the main class of the diagram, the *feature provider*. For such an adding we again need an extension point, which will be demonstrated later in chapter 6.

[1] `https://wiki.eclipse.org/FAQ_What_are_extensions_and_extension_points%3F`

4.8.2 Extension Mechanisms in Sirius

When using the Sirius framework, there are overall four different extension mechanisms that can or even must be combined. All these extension mechanisms are described shortly in the following. Every extension mechanism can be used within the same viewpoint or in any other viewpoint as well as long the necessary resources are loaded. This also means that every extension mechanism can be used in a different project than the project containing the core editor.

- **Diagram Import**: With the help of the diagram import mechanism, a developer, like the name states, can import a given diagram. Instead of the complete diagram representation, only the first layer of that diagram representation is imported. However, the developer has the opportunity to add further layers. Unfortunately, every change made in the imported diagram directly affects the original diagram. Therefore, we can not use this extension mechanism for our purpose and is therefore not further regarded during this thesis.

- **Java Extension**: With the help of a java extension, the developer can add a java class defining general methods that can be used wherever an Acceleo expression is needed. Although this extension doesn't actually add new attributes to existing elements or new classes, it strongly supports the diagram import and extension mechanism Sirius offers. That is because Acceleo expressions are used for labeling or creating elements or semantic expressions pointing to certain elements. Java extensions are explained in detail in the implementation chapter.

- **Extension Point**: Since Sirius as well as Graphiti are both frameworks embedded in the Eclipse platform, the developer has the same opportunities regarding extension points than with Graphiti. Although, the capabilities when defining own extension points are the identical to Sirius and Graphiti, there are a lot more extension points specific to Sirius than to Graphiti leading to an easier extension of the whole framework. However, self-defined extension points may offer further functionality, such as mouse-over buttons, which Sirius doesn't use.

- **Diagram Extension**: When extending an existing diagram with a diagram extension, nothing in the core diagram can be changed in any of its layers. However, the developer can add additional layers representing either new information to existing elements or even new elements, depending on the meta-model extension. Therefore, the diagram extension should be preferred over the diagram import mechanism and is used throughout the entire implementation of the Sirius prototype. As we can use any Sirius tools after creating a diagram extension, we can assume that the diagram extension is capable of realizing most of the extension types presented in this chapter. This assumption is proven correct during the validation in chapter 6.

5 Mapping of Extensions between Meta-Models and Graphical Editors

In the previous chapter we focused on a classification of extensions on meta-model level as well as on graphical editor level. Therefore, we distinguished between extension types and extension mechanisms. While we mapped each extension mechanism to one or more extension types on their respective levels, we still need to relate both levels to each other. This is done within this chapter. We here focus on mapping extension types on meta-model level to possible extension types on graphical editor level. Thereby, we always consider one meta-model extension type and analyze the possible resulting graphical editor extension types. Since there are five different meta-model extension types, but twelve extension types on graphical editor level, there obviously can not be a one-to-one relation between these extension types.
Although, an extension type on meta-model level exists, this doesn't necessarily mean that the extension type has to be mapped on graphical editor level. There are multiple cases, where meta-classes are not intended to be represented on graphical editor level. Therefore, given a certain extension type, there is always the possibility that no mapping to graphical editors is even wanted.
This chapter starts with the analysis of both meta-class extension types and continues with the three extension types covering new information to existing meta-classes. Each section thereby covers the possible mapping and an argumentation of why the other graphical editor extension types are not supported by a meta-model extension type.

5.1 Mapping of Meta-Class Instance Below Root Node to Graphical Editors

This section covers the mapping of extending the meta-model by the new meta-class extension type to possible extension types on graphical editor level. The extension type in focus here covers meta-classes, whose instances are placed one level below the root node chosen for the graphical editor. Usually the root node of a graphical editor is identical to the root node of the corresponding EMF tree editor. To cover all the aspects of this mapping, this section is further divided into first all supported mappings from meta-model level to graphical editor level and then an analysis of why the other extension types on graphical editor level are not supported.

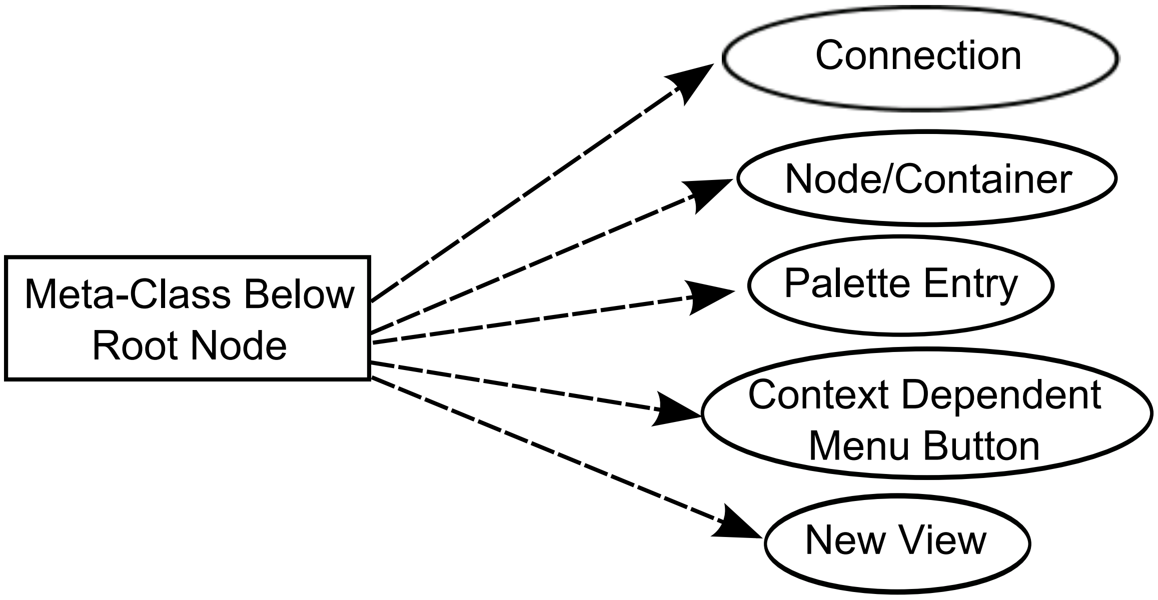


Figure 5.1: Mapping of the first meta-class extension type to graphical editor extension types

5.1.1 Supported Realizations on Graphical Editor Level

To start off, we analyze the possible realizations on graphical editor level given the new meta-class extension type. For a better overview on this mapping figure 5.1 is given. As we can see, when using this extension type, we can choose between five different realizations on graphical editor level, whereas two of them can be used for representation. The other three extension types are used for their creation. For a better overview, each of these extension types are discussed in their separate paragraph.

Notation Element If the new meta-class is supposed to be represented in the graphical editor, one obvious extension type that should be covered on graphical editor level is the new node/container extension type. Whether we use a node or container depends on the scenario. As a general notion we can say that if the meta-class contains more than one attribute or other containments, using a container as representation is appropriate. Furthermore, if possible extensions to this meta-class are thinkable, a container might be of more use than a node. As an alternative for the node/container extension type, if the meta-class resembles a connection between other meta-classes, we could represent it as connection instead of a node or container.

Palette Entry As already stated in section 4.6.3, the palette view is used to create not only a notation element in the diagram. Also the corresponding model element in the underlying business model is created. Therefore, if the user not only wants to show and possibly remove the new notation element from the diagram, a new palette entry should be created.

Context Dependent Menu Button Apart from creating a new palette entry, we can also extend any given context dependent menu by adding another button to it. This can be either a new mouse-over button, when hovering over an empty diagram space or a context menu button appearing, when right-clicking on the diagram. In any way, we can use such a button to create the notation element and its corresponding model element. As the effect of both a palette entry and the context dependent menu button are usually interchangeable for a given meta-class extension type, one or both can be used for the creation of the notation element and its corresponding model element.

New View The last possible extension type on graphical editor level, which may be the result of the new meta-class extension type, is the new view extension type. In this case the new view can be seen as a wizard that pops up, where the user has to choose between different model elements before the notation, and the corresponding model element, can be created correctly.

Now that we discussed which extension types can be realized given the first meta-class extension type, we should also analyze why the other extension types should not be preferred. This is done within the next section.

5.1.2 Unsupported Realizations on Graphical Editor Level

Now that we discussed the possible extension types on graphical editor level that can be realized, if the first new meta-class extension type on meta-model level is realized, we need to analyze why the other extension types for graphical editors can not be realized at the same time. Like in the previous section, we here also divide the unsupported extension types by paragraphs explaining why an extension type on graphical editor level should be avoided given the first meta-class extension type.

Extend Existing Notation Element It is pretty obvious that every extension type related to altering existing notation elements can not be realized by the given meta-class extension type. This is due to the fact that instances of this meta-class are direct containments of the editor's root node. Therefore, the corresponding notation element can not be part of any other existing notation element, except for the root node, which is the basis of the diagram. In conclusion, we can say that none of the four extension types concerning the extension of a notation element can be realized with the given meta-class extension type.

Add Properties Entry Since we excluded all of the four extension types concerning the extension of an existing notation element, consequently, there is no properties view to extend. As we always create a new notation element, its properties view contains everything that is needed. However, if further notation elements are added to the new container, we could alter the container's properties view to represent the new values, when selecting the container. Nevertheless, this would be in contrast with our given mapping for this meta-class extension type, as this container would be altered due to a different meta-model extension type.

Extend Outline Now that we overall discussed ten out of the twelve extension types on graphical editor level, there is only extending the outline and adding a new toolbar button left. As the automatic update of the outline view, whenever a notation element is added to the diagram or changed, is not considered an extension to this view (see section 4.6.5), there is no scenario where this view is extended to show the new meta-class. We can say that there are two basic situations, where such an extension in case of a new meta-class can be considered. First, when in addition to the notation element in the diagram specific characteristics should be displayed. If that is the case, extending the outline is not done to show the new notation element. It shows rather the specific attribute, which is again a different extension type. The second situation is when the notation element should only be displayed within the outline view, but not in the actual diagram. That would be a real extension of the outline. At the same time this mean that the new meta-class has no important contribution to the core meta-model as the corresponding notation element is only shown in a small scale in the outline for representative purposes. If that is the case, the extended meta-class can simply be omitted.

Toolbar Button The last extension type to discuss is the new toolbar button extension type. Toolbar buttons can be considered a special extension type, as their purpose varies from loading resources to a diagram, moving notation elements in order to gain a certain layout or changing the status of multiple entities. While we do not think of the first two purposes as extension type for any meta-model extension type, the last purpose is still not covered, when considering the first meta-class extension type. Besides changing the status of existing elements, of course, multiple new notation elements could be added. However, if we only regard the meta-class and its notation element, there is no indication on how many notation elements should be added when pressing the toolbar button. Therefore, a toolbar button for such a purpose would be too random to actually consider it as extension type. A last notion on toolbar buttons for the meta-class extension type is that we could alter the purpose of the toolbar so that only one notation element is added somewhere in the diagram. However, since the position would be more or less random and the toolbar button uses the same functionality as the new context menu button, we can leave out this extension type.

5.2 Mapping of Meta-Class Instances as Part of Other Instances to Graphical Editors

This section deals with the mapping of the second new meta-class extension type to extension types on graphical editor level. This extension type focuses on instances of a meta-class that are listed as part of other instances, except the chosen root node of the graphical editor. As in the previous section, we divide this section into one part analyzing the supported realizations on graphical editor level, while the second part discusses why the other extension types on graphical editor level are not supported.

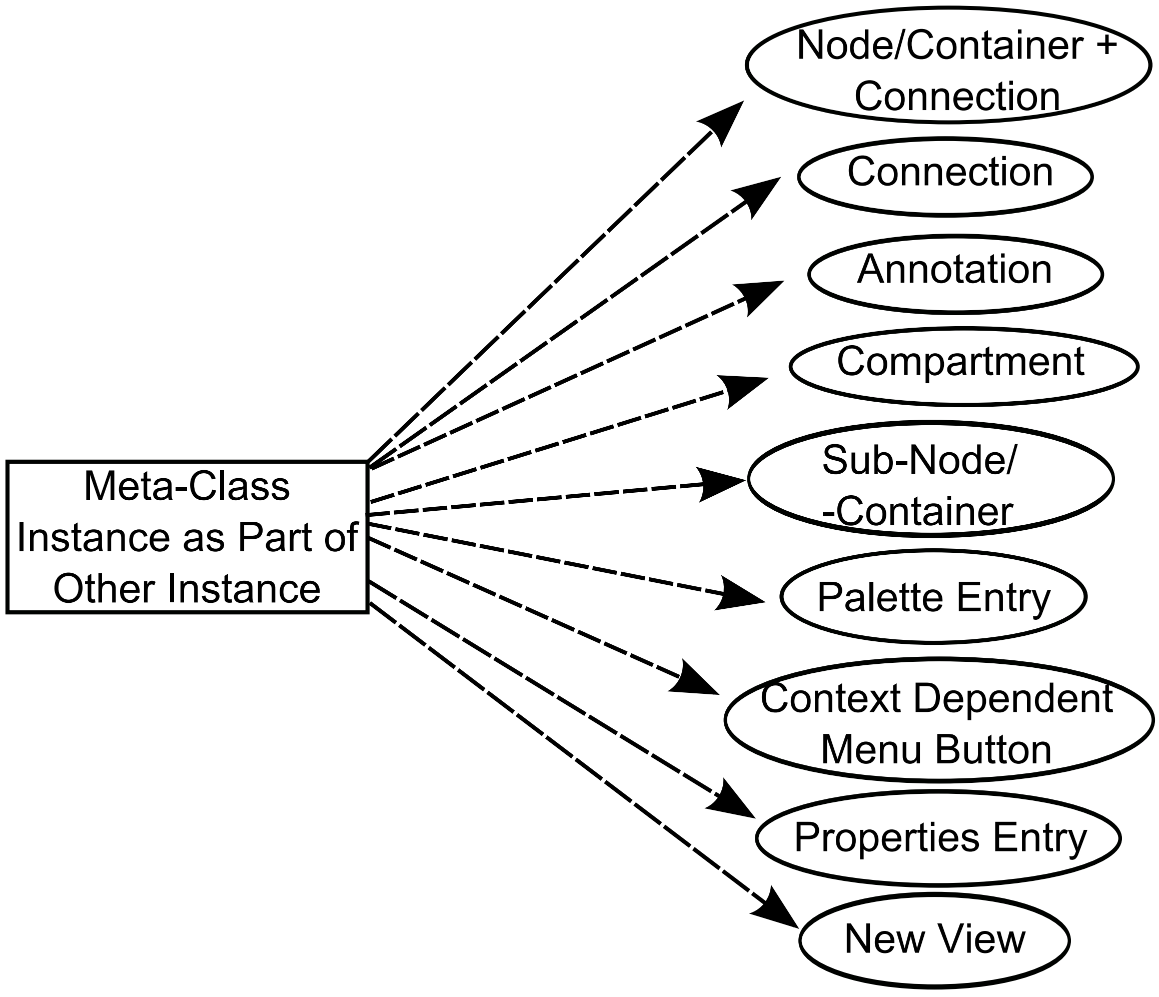

Meta-Class Instance as Part of Other Instance
Node/Container + Connection
Connection
Annotation
Compartment
Sub-Node/ -Container
Palette Entry
Context Dependent Menu Button
Properties Entry
New View

Figure 5.2: Mapping of the second meta-class extension type to graphical editor extension types

5.2.1 Supported Realizations on Graphical Editor Level

Like in the previous section, we first give an overview on the possible mappings as represented by figure 5.2. Next, we discuss in each paragraph why those extension types are supported. As there are overall nine extension types supported for this meta-model extension type, we also present table 5.1 at the end of this section. The table shortly summarizes the supported extension types and the condition under which they should be preferred.

As we can see from the figure, there are more extension types on graphical editor level that can be possibly realized, if the second new meta-class extension type is given. While there are some extension types we already discussed in the previous section, there are others that became available once instances of the meta-class are placed as part of other instances in the EMF tree editor.

Node/Container + Connection Although, the meta-class in focus is already part of another meta-class, we still have the opportunity to represent it as new notation element within the diagram. The only difference to the mapping in the previous section is that we need a connection to the notation element resembling the container meta-class as well. That way, we make sure the user knows to where this notation element and its corresponding meta-class instance belongs.
Apart from representing the meta-class instances as node or container, we, of course, can represent them also only as connections, if and when the meta-class itself should resemble a connection. Thereby, we do not differ whether a connection targets two notation elements inside an existing notation element, two notation elements directly placed in the diagram or a combination of both as stated in section 4.6.2.

Extend Existing Notation Element If we don't want to create a new notation element in the diagram including a connection, we can also extend an existing notation element. If the new meta-class' essence can be represented by a single string or another attribute, adding an annotation to the existing element is enough. The annotation could, for example, be a bordered node. On the other hand, if the meta-class explicitly adds new domain specific information that can not be covered with the representation of an annotation, a new sub-node or -container can be added.
Aside from creating new sub-nodes or containers inside a given container, there is also the possibility to create a new compartment. This is usually done if more meta-classes belong to the meta-class in focus, and they can furthermore be grouped together. In other words we can say that a new compartment is never created alone, as further sub-nodes are needed to fill the compartment. Whether those sub-nodes are directly related to any meta-class in the core meta-model or reference the meta-class in focus in any way, doesn't matter.

Creation of the Notation Element As soon as it is possible to create compartments or sub-nodes or -containers it seems likely that a palette entry or a new context dependent menu button is also created in order to create those notation elements not only in the diagram but also in the underlying business model. This argumentation is the same as in the previous section. Furthermore, a new view can be added if the meta-class should also be represented in its own diagram representation. Another purpose for a new view is if further to the meta-class related instances should be manually chosen by the user in order to create a valid notation and business model element.

Adding a Properties Entry Aside from adding a palette entry in order to create the notation element and its corresponding meta-class instance, we here have also the possibility to add a new properties entry. This can be done, since instances of this meta-class are already part of an existing instance and therefore their notation element corresponds to an already existing notation element. Therefore, the properties entry can be added to the already existing notation element, if not done automatically. A drawback is that the properties entry is only supposed to show and alter attributes of a notation element, but not to create new ones. Adding a properties entry may still be useful, if further attribute

	Notation Element + Connection	Connection	Sub-Node/-Container	Annotation	Compartment	Palette Entry	Context Dependent Menu Button	Properties Entry	New View
Representation	x	x	x	x	x				x
Intended as Connection		x							
Simple meta-class				x					
Element grouping					x				
Creation also possible						x	x		x
Manual choice of sub-elements									x
Further properties available								x	

Table 5.1: Overview on when to use which extension type on graphical editor level given the second new meta-class extension type

extension types should also be shown within this properties entry. However, with the help of the new view extension type this rule could be violated and new notation elements could be created. This is not further discussed here.

Table 5.1 again lists all possible extension types, including their condition under which they can be applied to the graphical editor. Rows five and six require any of the representation extension types, while row number six also includes the row above. The last row however does not require any of the other extension types. The further content of the table is not discussed here, as it is already mentioned in the paragraphs above. Within the next section we analyze the extension types not listed in this section and discuss why they can't or shouldn't be used for the second meta-class extension type.

5.2.2 Unsupported Realizations on Graphical Editor Level

As already done in the previous section, we here discuss those extension types on graphical editor level that are not supported by the second new meta-class extension type.

New Node/Container Although, we discussed the new notation element extension type partly in the previous section, for the sake of completeness, it is necessary to analyze it within this section. As already discussed, we can combine the new node/container extension type with the new connection extension type, in order to represent the second meta-class extension type on graphical editor level. The connection therefore targets the outer instance of our meta-class instance in focus. If the connection was left out, we could still identify the targeting notation element by adding an annotation containing the ID of the target. However, the diagram would get too complex, if this was done for every meta-class instance that is part of another instance. That is why the new node/container extension type should only be applied, if combined with a connection to its target. If placed alone without any additional extension type, there is no indication given to which other notation element the node or container belongs.

Change of Appearance Another extension type from which we should refrain, when given the second meta-class extension type, is changing the appearance of an existing notation element. Changing the appearance of an element usually indicates a change of an attribute's value. Therefore, this extension type is not supported when adding a new meta-class as part of another meta-class.

Toolbar and Outline As already mentioned in the previous section considering the first meta-class extension type, an extension of the outline view would require that our meta-class including all of its attributes is not important enough to be represented in the diagram itself. However, if that is the case, the class itself could be left out. Therefore, it is not necessary to extend the outline view given the second meta-class extension type. For an extension of the toolbar we can also apply the same argumentation as in the section before. Although, we could implement a toolbar button creating multiple instances of the meta-class as notation element, a scenario where doing so is necessary, is hard to find and therefore this extension type can also be omitted. An equivalent result can be achieved for multiple drag and drop operations from one palette entry.

5.3 Mapping of Adding Attributes to Existing Classes to Graphical Editors

After covering both extension types on meta-model level dealing with the adding of new meta-classes, we also have to discuss adding new information to existing meta-classes and their mappings to graphical editor extension types. Therefore, we start with the adding attributes to existing classes extension type. Like in the previous sections, we first start with all supported realizations on graphical editor level for this extension type, followed by a short summary of the reasons behind each extension type. In the section afterward, we discuss why the other extension types are not supported.

5.3.1 Supported Realizations on Graphical Editor Level

When adding a new attribute to an existing class, there are overall eight possibilities to realize this extension type for graphical editors. Figure 5.3 summarizes all possible mappings for this extension type. Starting from these eight extension types, we can divide those further into extension types that are necessary for the pure representation of the

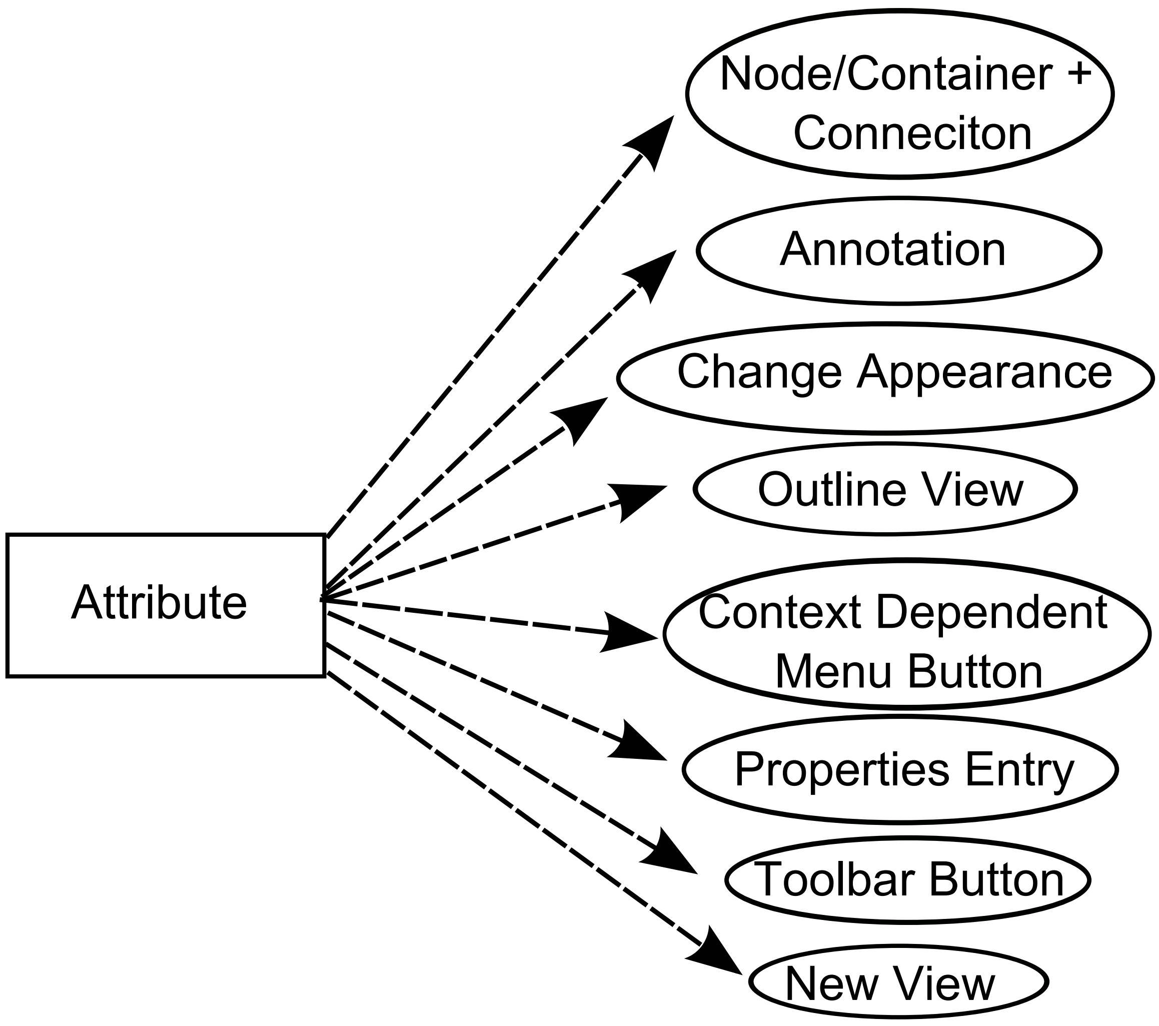


Figure 5.3: Mapping of the attribute extension type to graphical editor extension types

new attribute and extension types that are used to change the value of the attribute. All of these extension types are discussed in the following paragraphs.

Node/Container + Connection Like in the previous mapping, we here have again the possibility to represent the attribute as new notation element. However, if we do so, we again need to connect the notation element to the notation element it belongs. The targeting notation element should overall represent the meta-class whose attribute we just added. This is the same argumentation as for the second meta-class extension type presented in section 5.2.1.

Extend Existing Notation Element When representing the attribute, we can further choose between adding a new annotation or changing the appearance. Changing the appearance is mostly done, if the attribute to represent is a Boolean attribute. If the attribute changes the state of the underlying meta-class instance, the color of the notation element may be set to change. Beside the color, also the label of the node or container may change indicating an attribute, such as a counter. If the existing label shouldn't be changed, an annotation can also be added to the notation element.

Extension of the Outline View The last possibility we have for the pure representation of an attribute, is the extension of the outline view. As stated in section 4.6.5, we don't consider the automatic update of the outline view as an extension. An extension rather introduces a new feature, which is only available for the outline view. Therefore, an extension of this view is rare, since the attribute needs to be important enough to appear in the meta-model but not that important to actually be shown in the original diagram. Another scenario for extending the outline view is, if the attribute requires a smaller representation of the complete diagram. This can be the case for an attribute indicating a bottleneck in a bigger diagram.

Extending the Properties View As mentioned in section 4.6.4, the properties view satisfies both, the representation of an attribute, as well as altering its value. Therefore, extending the properties view, when introducing an attribute, should always be considered, if not done automatically. The type of attribute thereby doesn't matter as there are plenty of widgets that can be added, in order to represent any type of attribute.

Altering the Attribute's Value Changing the value of an attribute can further be done by adding a toolbar button, a context dependent menu button or a new view or simply by adding a new feature to an annotation created before to represent the attribute. With a new feature a mouse click on the annotation results in changing the label of the annotation. At the same time, the attribute's value should change, if implemented correctly. We don't consider this as an actual extension type, but for the sake of completeness it is still mentioned here.
Besides different functionality, such as loading a new model, toolbar buttons can also be used for applying or clearing an attribute's value for all relevant notation elements in the diagram. The important difference between a toolbar button and every other extension

type in this mapping is that the toolbar button applies a change to all notation elements that fulfill a certain condition. The other extension types apply their change only to one notation element containing the attribute.
As in the previous sections mentioned, a mouse-over button as well as a context menu button provide a pop-up window and are most of the times interchangeable. If the framework provides custom mouse motion listeners, we can add an icon as mouse-over button representing the state of the attribute we want to change. By clicking on the button the value of the attribute changes to a value specified in the implementation. The new context menu button does the same. If both buttons can be implemented, the mouse-over button in general should be preferred, since it is user-friendlier.

New View The last extension type realizing the attribute extension type on graphical editor level is the creation of a new view. There are various examples, where a new view can be used to alter the value of an attribute, such as views for entering and validating key-value pairs. However, all these examples require not only the representation of the attribute, but furthermore at least one feature that opens the new view. In case of altering the value of an attribute, a double click on a new annotation or even one of the other extension types responsible for altering the attribute's value can be used as feature to open the new view.

5.3.2 Unsupported Realizations on Graphical Editor Level

Since there are eight possible extension types that can be realized on graphical editor level, when adding a new attribute to an existing meta-class, there are only four extension types, which are not supported by the new attribute extension type. As before, these extension types are discussed within each of the following paragraphs.

Sub-Node/-Container Starting at the top of our classification, introducing a new sub-node or -container is by definition not possible, when realizing the new attribute extension type. New sub-nodes or -containers contain new domain specific information, whereas an annotation only specifies additional information to an existing domain class.

Adding a Compartment The next unlikely extension type is the compartment extension type. A compartment resembles a group of similar elements. Creating a compartment would be possible, if a list is added as attribute. However, depending on the list only their size may be of interest or the elements are shown in the properties view. If the content of the given list contains rather complex elements, they should rather be resembled as new containment as then further attributes need to be represented. In either way, adding a compartment because of a single attribute is unlikely.

New Notation Element As previously discussed, the combination of adding a new notation element and a connection to the corresponding notation element is valid. However, using only one of these extension types to cover the attribute extension type, is not supported. If the node resembling the attribute is used alone, then we can not decide to which element the attribute belongs. On the other hand, an attribute can not be represented by a connection. A connection requires both a source and a target. If a primitive data type is used, there can not be a source and target in the same diagram. Nevertheless, if the data type is non-primitive and can be identified to an existing notation element, representing an attribute as a connection would be possible. However, if this is the case, there should rather be a relation on meta-model level between these two meta-classes, instead of an attribute.

Palette Entry The last extension type that is not supported by the new attribute extension type is the palette entry. Since the palette is used to add notation elements that didn't exist before, there is no need in introducing a palette entry for an attribute, as attributes are always present. Only their value changes, which can not be covered by a palette entry.

5.4 Mapping of Adding a Containment to Existing Classes to Graphical Editors

In this section we analyze the mapping between adding a new containment on meta-model level and its realization possibilities for graphical editors. Like the previous section, we start by analyzing all supported realizations and continue with a discussion of why the other extension types are not supported or at least should be avoided.

5.4.1 Supported Realizations on Graphical Editor Level

Similar to the previous mapping, we here have seven supported realizations for the containment extension type. In order to preserve the structure of the previous sections, figure 5.4 shows the mapping of the new containment extension type. We here also separate this section into first, the supported extension types that are used for the pure representation and in the next paragraph the extension types used for the creation of the containment on editor level are presented. As the argumentation for some extension types are the same as in the previous sections, we only reference the paragraph, where these extension types have already been discussed.

Representation As the previous two extension types, the containment can also be realized as a combination of node or container and connection targeting its container. For a detailed explanation refer to paragraph 5.2.1. Furthermore, the containment can also be realized as connection, if the containment itself is intended as connection between two meta-class instances.
Deciding, whether a containment is realized as new sub-node or -container or as compartment, strongly depends on further meta-classes and their references towards the containment. If the containment doesn't have any further references, representing it as sub-node or sub-container is enough. If, however, there are more meta-classes related to the containment with an arbitrary cardinality, the containment can be used as compartment. On the other side, if the containment itself has an arbitrary cardinality towards

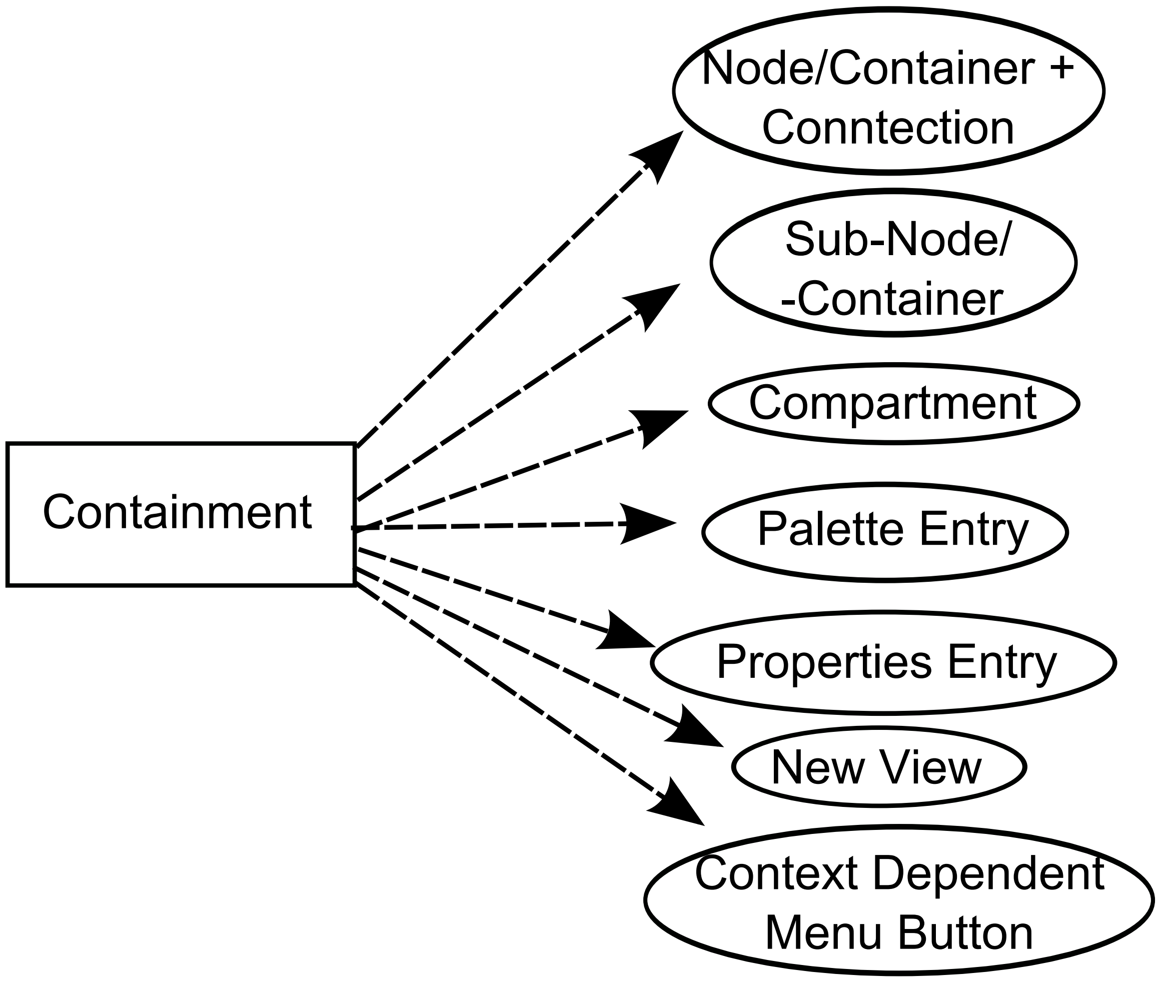


Figure 5.4: Mapping of the compartment extension type to graphical editor extension types

it container, those containment instances can also be grouped in a compartment. As a general rule, we can say that the more instances the containment may have, the better the representation as compartment is.
As last extension type that can be used for the representation of the containment, a properties entry can be added to the container element. Besides the representation of the containment and its possible sub-elements, the values of those sub-elements may also be altered in the properties view.

Creating the Containment on Editor Level If the containment should also be created in the diagram, there are the same possibilities we have, when creating an instance of a new meta-class in our diagram. Those possibilities are a new palette entry or a new context dependent menu button. Both of these extension types have already been mentioned in paragraph 5.2.1. As support for any of the both mentioned creating extension types, a new view can also be used. This view could either be a new editor with the containment as root element or a dialog for choosing the containment's sub-elements.

5.4.2 Unsupported Realizations on Graphical Editor Level

As the previous section discussed all supported extension types on graphical editor level given the containment extension type, we now discuss why the other extension types are not supported or at least are unlikely. Therefore, each of the remaining extension types are analyzed in separate paragraphs.

Node/Container The reasons that speak against only adding a node or a container to the diagram are the same as mentioned in sections 5.3.2 and 5.2.2. Of course, if the root element of the diagram changes due to a newly created editor, the containment could be realized as node or container. However, the containment wouldn't be a containment extension type anymore, but one of the meta-class extension types.

Annotation and Change of Appearance As containments are rather complex instances and usually contain further elements, simply changing the appearance of the existing notation element won't cover all the containment's features. An equal problem occurs for annotations. Instead of an attribute, a containment offers more complexity. Therefore, we can not add an annotation, in order to grasp the containment to its full extend.

Extension of Outline Extending the outline view, in order to show a containment that isn't shown anywhere else in the diagram, can be rejected with the same reasons as in section 5.1.2. Therefore, we do not discuss this extension type further.

Toolbar Button The last extension type that is unlikely to realize given the containment extension type, is the toolbar button extension type. We could implement a toolbar button serving the functionality of creating the containment in any desired way. However, we would require certain rules specifying which notation element receives the graphical representation of the containment and which does not. Furthermore, not only one containment

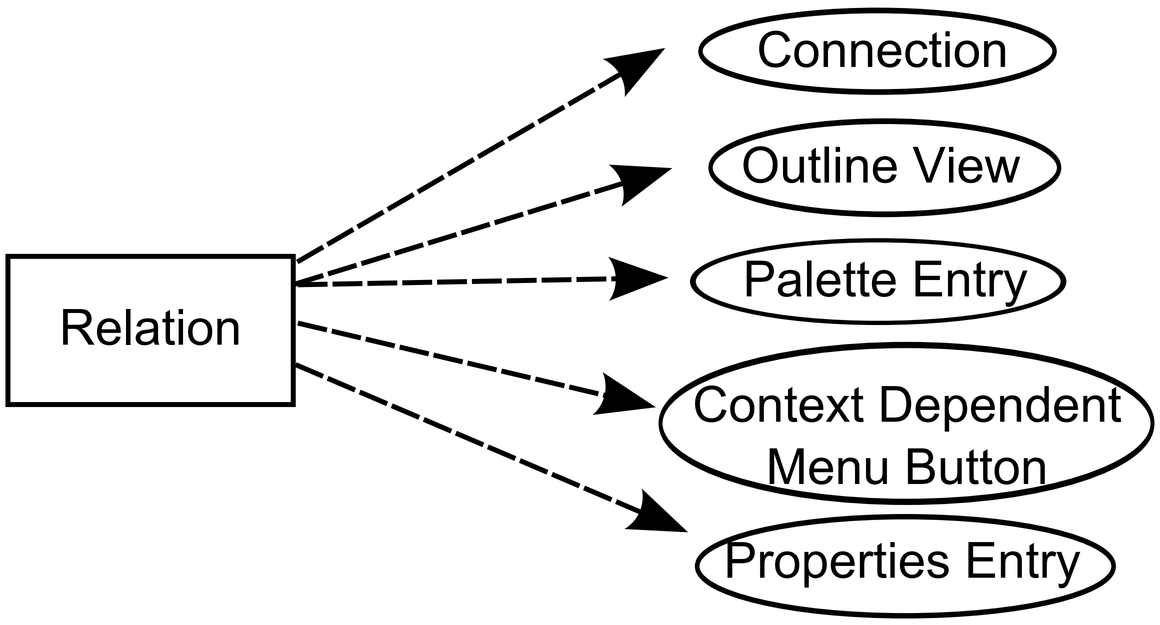


Figure 5.5: Mapping of the relation extension type to graphical editor extension types

instance would be added, but an arbitrary valid number. Although, in some scenarios this may be useful, in general, we should refrain from implementing such a toolbar button.

5.5 Mapping of the Relation Extension Type to Graphical Editors

The last extension type, whose mapping needs to be discussed is the relation extension type. Again, we first provide all the supported extension types on graphical editor level. Afterward, the present the rather unlikely or unsupported extension types.

5.5.1 Supported Realizations on GRaphical Editor Level

As in the previous sections, an overview is given by figure 5.5. When speaking of the relation extension type, it is important to remember that only the relation is relevant at that point. Of course, the relation always has a source and a target meta-class. In this section, however, we focus on representing and creating only the relation on graphical editor level. Furthermore, we need to remember that only those relations are relevant that would still be relations, if the core meta-model was extended intrusively. As we did in the previous sections, we divide this section into paragraphs analyzing the representation possibilities and the possibilities to create the relation.

Representation Given the relation extension type, we have overall two possibilities to represent the relation. The first one is the connection, while the second one is extending the outline view. Given a relation, we always can choose whether we want to represent

is as connection or if we want to represent it only indirect. An indirect representation thereby means that, considering an unidirectional association, the source of the relation is represented within an existing notation element, the target, leaving out the relation representation. Considering a composition, the target is usually represented within the source notation element. When seeing that a notation element is placed inside an existing one, the user knows that the inner notation element belongs to the outer one. Therefore, there is no reason to also represent the relation explicitly.
Another way of representing the relation extension type is by extending the outline view. This could happen, if the relation is defined as connection on graphical editor level representing a data flow. Although present in the actual diagram, an extension of the outline could therefore imply a bottleneck between some of the entities. The connection itself could then, for example, be represented in a different color than in the actual diagram.

Creation of the Relation This paragraph assumes that the relation is represented as connection. Indirect representations were already regarded in the previous sections analyzing the other extension types on meta-model level. As we represent the relation as connection, besides adding a palette entry, also a context dependent menu button can be added. As the connection is relation-based, no new element will be added to the underlying model. Moreover, the properties view of existing elements changes. Therefore, we also have the possibility on graphical editor level to extend the properties view by adding a new entry, if not done automatically. The properties entry can then show the connection between two notation elements or we could even create further connections between elements that were not connected before.

Within the next paragraph, we discuss why there is no mapping to the rest of the extension types on graphical editor level given the relation extension type.

5.5.2 Unsupported Realizations on Graphical Editor Level

During this section we analyze the seven extension types left on graphical editor level. We again, discuss those extension types only shortly, if the reasons behind not using them are similar to reasons we already mentioned during previous sections.

Node/Container Since a relation states a connection between two meta-classes, realizing the relation as node or container would cause the source and target information to get lost. Of course, we could add annotations to the node or container that state the source and the target element. However, in a larger diagram with many similar elements, we can not see which element is connected to which. Furthermore, if the elements don't have an ID, they can not be identified exactly, when using this sort of notation. This is why the developer should refrain from realizing the relation extension type as node or container.

Extend Existing Notation Element The same argumentation from above can also be applied, when thinking about a realization as sub-node or sub-container. However, as mentioned in the previous section, a relation can also be indirectly realized. Considering this, the relation, of course, is partly represented as sub-node/-container, compartment or annotation. However, as the representation is only indirect, these realizations are not considered above. At last, the extension type considering the change of appearance can also not be supported by the relation, as the semantic behind such an extension is not clear. A change in the color of a notation element, usually maps to a certain state. A relation, however, does not resemble a state.

Toolbar Button Like always, the toolbar button resembles a special extension type. There could be scenarios, where a toolbar button is needed to create connections. However, these scenarios are limited to a view. Therefore, the toolbar button is here again, not considered a likely extension type.

New View Although possible in every other mapping, a new view created because of a relation, is unlikely. As already stated, the relation is only represented directly as connection. A connection as such, relates two notation elements. Therefore, there is no reason to implement a new view to represent the connection. Even if the connection needs to be created between three notation elements, this can still be done within the normal diagram representation. A new view would only let the user choose in a dialog window, which notation elements the connection should target. Although this is a scenario for implementing a new view, it is rather unlikely and therefore, the view is not considered in the mapping above.

6 Implementation and Validation

This chapter deals with the implementation of both prototypes in Graphiti and Sirius. On the one hand, this chapter is used to validate the classification of chapter 4 as well as the mapping of those extension types in the previous chapter. On the other hand, this chapter can also be seen as guideline as to how different extensions in both Sirius and Graphiti can be implemented. To start off, we first discuss possible scenarios that can be used for the validation. After choosing a scenario, we first analyze its core meta-model and implement the corresponding graphical editor in both frameworks. When analyzing the extensions, we first take a look at the meta-model extension types. From those extension types, we infer the possible mapping to graphical editor extension types. During the implementation we choose appropriate extension types on graphical editor level with regard to the implementation of the core editor. Therefore, not every mapping can be implemented, as there are too many different mappings. This procedure is done for all three extensions highlighting the most important changes during each extension. This chapter concludes with a section that sums up the mappings validated in each framework.

6.1 Overview on Available Scenarios

To validate the concept introduced in chapters 4 and 5, we implement two prototypes. One is implemented with the help of the Graphiti framework, the other one with the Sirius framework. Both are Eclipse-based graphical editor frameworks. To show that the classification and the mapping analyzed in the previous chapter holds for any realistic scenario, we first have to choose an adequate scenario. Overall, we found three basic scenarios, which look promising on first sight. One of them is analyzed at length during this chapter, while the other scenarios are discussed shortly within the next chapter. Table 6.1 gives an overview on the possible extension types on graphical editor level that can be realized with possible extensions on meta-model level. As we can see, the *Smart Grid Resilience Framework*[1] can support all possible extension types on graphical editor level in its extensions. To be fair, we therefore used a more or less artificial extension among the two existing ones on meta-model level. Even though one of these extensions didn't exist before, the extension itself is conclusive.
The IntBIIs extension is introduced by Heinrich et al [22]. IntBIIs is an extension of the Palladio Component Model, which addresses only the *usage model*. An evaluation of this scenario is done in section 7.3.1. Like the IntBIIs extension, the security extensions mentioned in the table are also an extension of the PCM. Instead of the usage model, they address the repository and the resource environment. Those extensions are introduced by Busch et al [5] and are also further addressed in section 7.3.2.

[1] `https://svnserver.informatik.kit.edu/i43/svn/code/SmartGrid`

	Node/Container	Connection	Annotation	Compartment	Change Notation Element	Sub-Node/-Container	Palette Entry	Context Dependent Menu Button	Properties Entry	Extend Outline	New View	Toolbar Button
Smart Grid	x	x	x	x	x	x	x	x	x	x	x	x
IntBiis	x						x	x				
Security Extensions			x	x	x	x	x	x	x			

Table 6.1: Overview on available scenarios

As we can cover all the possible extension types on graphical editor level with the Smart Grid extensions, we choose this as our scenario for the implementation of both prototypes. An introduction to the Smart Grid Resilience Framework is given in the next section.

6.2 Smart Grid Resilience Framework

A smart grid is conceived as an electric grid able to deliver electricity in a controlled, smart way from points of generation to consumers [20]. Smart grids often make sense, when using renewable energy resources, such as solar plants or wind generators. There are usually more than one renewable energy resource in a defined region. Those can all be addressed separately. With a smart grid we can choose, whether we need all wind generators or ,if the production of a few is currently enough.
When designing a new smart grid, it becomes necessary to graphically represent the smart grid, as distances or possible flaws can be detected easier. Therefore, this chapter uses graphical editors to represent a smart grid based on a given meta-model. The graphical editor can also be used to run various analysis to show the impact of each analysis directly in the diagram. Furthermore, we validate our classification of chapter 4 and the mapping discussed in chapter 5 with the help of overall three extensions.
In order to build a graphical editor for smart grids, we need to define a meta-model first, on which the graphical editor is based. The meta-model given in figure 6.1 is based on previous work available at [2]. Based on the root element *SmartGridTopology*, there are overall four different containments: *PowerGridNode*, *NetworkEntity*, *PhysicalConnection* and *LogicalCommunication*. Network entities are at least connected to one power grid node, which is responsible for the generation of power. Furthermore, these network entities can be connected to each other by a physical connection. The abstract class *NeworkEntity* can

[2]`https://svnserver.informatik.kit.edu/i43/svn/code/SmartGrid`

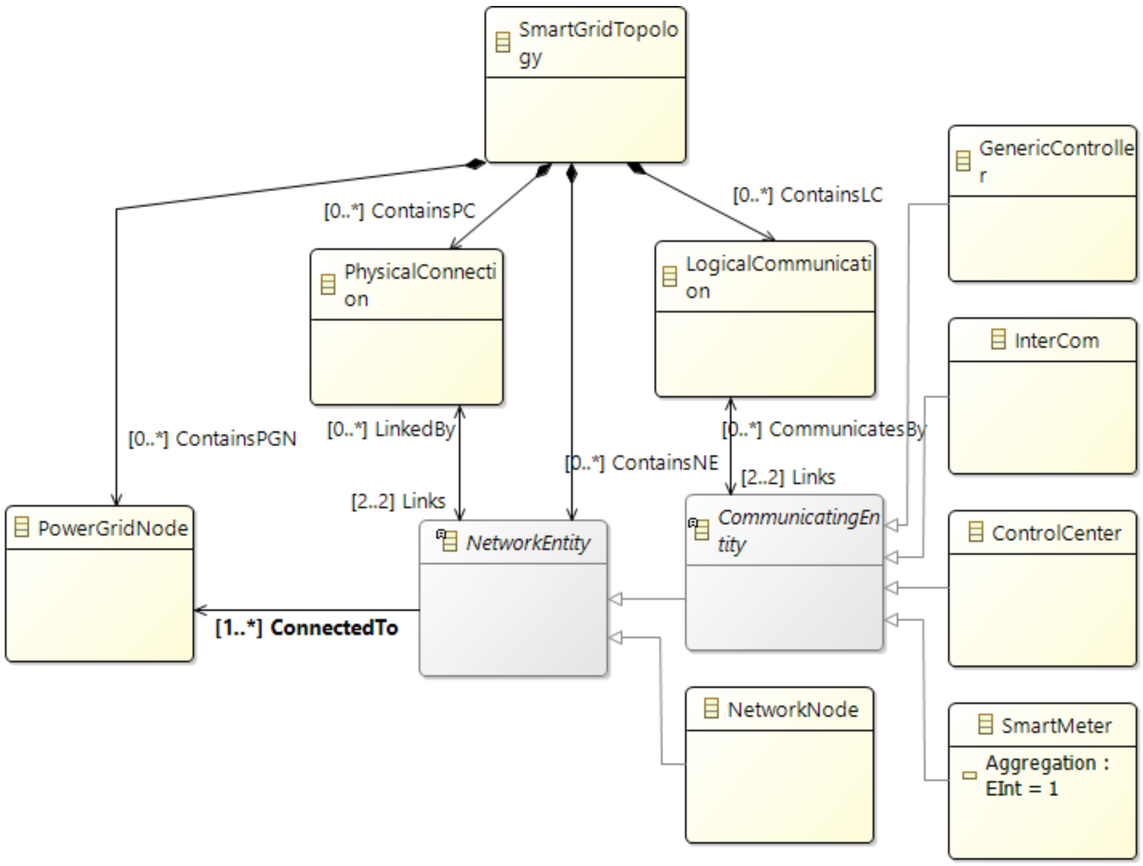


Figure 6.1: Screenshot of the smart grid core meta-model as described in 2

be divided into further concrete classes, such as *NetworkNode*, *SmartMeter*, *ControlCenter*, *InterCom* and *GenericController*. All of these entities, beside the network node, are also a *CommunicatingEntity* meaning that there can be a logical communication between two of these entities. The only attribute listed in this meta-model is the *Aggregation* of a smart meter. If the value of this attribute is higher than one, it means that the smart meter instance resembles as many smart meters as the attribute's value states.
The next section starts with the implementation of the given meta-model as Sirius-based graphical editor and continues with the implementation of the Graphiti-based editor.

6.3 Implementation of the Core Meta-Model in Graphical Editors

This section covers the implementation of the core editor for both the Graphiti and the Sirius framework. Both subsections cover the framework specific parts of the implementation. Therefore, this whole chapter can also be seen as a guideline, when similar editors have to be implemented in either of the two frameworks.
In general, when implementing a new editor, we start off by choosing the root element for each model representation. In our case this is an instance of the *SmartGridTopology* meta-class, as all the other meta-classes are contained within *SmartGridTopology*. All the other meta-classes we want to represent in the graphical editor can then be added according to the frameworks specifics.

6.3.1 Sirius Implementation

When using the Sirius framework, we can more or less click our editor together and still have the opportunity to provide java classes for even more functionality. Creating an editor in Sirius mainly consists of the following parts. First, we define a new viewpoint with which we define the connection between the different models and the editor we're about to implement. Second, we need to define all notation elements that should be represented in the diagram later. Optional, we can implement sections that also support the creation of these defined notation elements. Putting all this together and creating a diagram for a given model instance, results in the diagram shown in figure 6.2. This figure is thereby our running example throughout this entire chapter for both the Sirius and the Graphiti framework. The center of the diagram thereby shows the notation elements, while on the right side the palette view includes all notation elements that can be added to the diagram. The legend in figure 6.3 shows which notation element resembles which model element. Starting with the definition of a new viewpoint, we explain how this editor can be implemented within the next subsections. Thereby, we only briefly discuss the features of representing and adding new notation elements, as this can already be found in many tutorials throughout the internet.

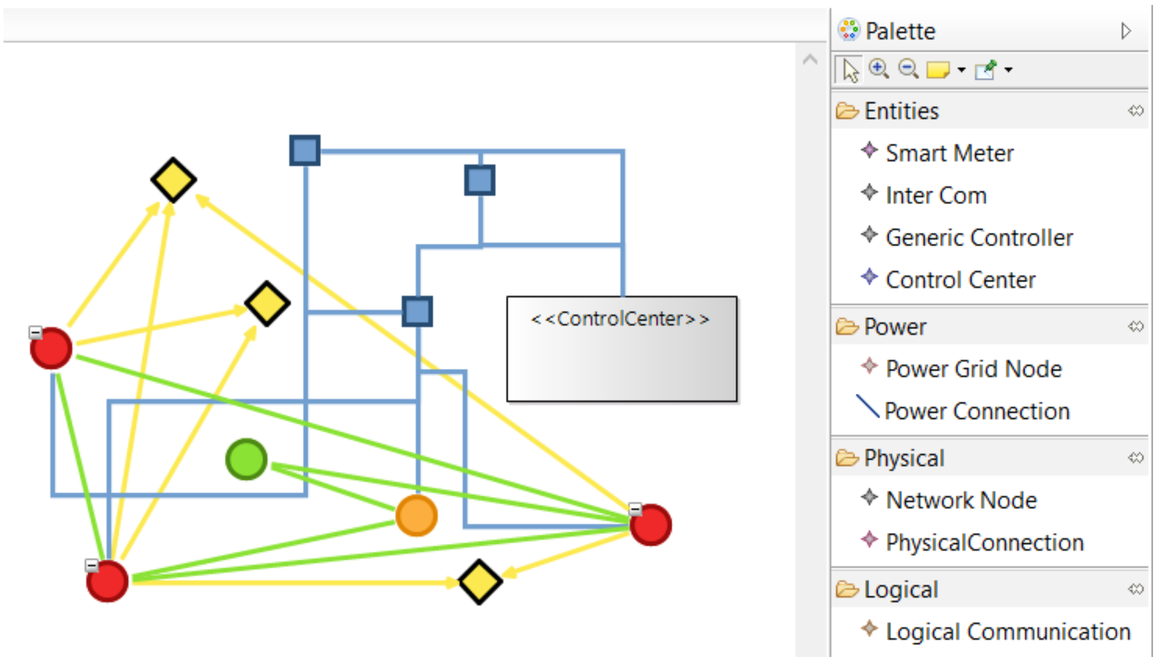


Figure 6.2: Screenshot of the resulting smart grid core editor in Sirius

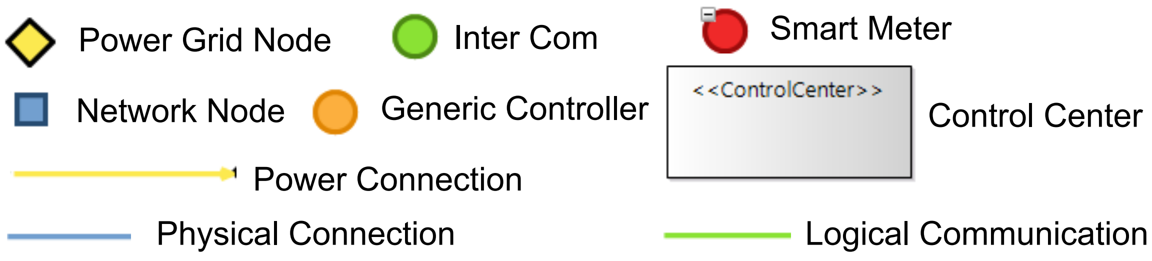
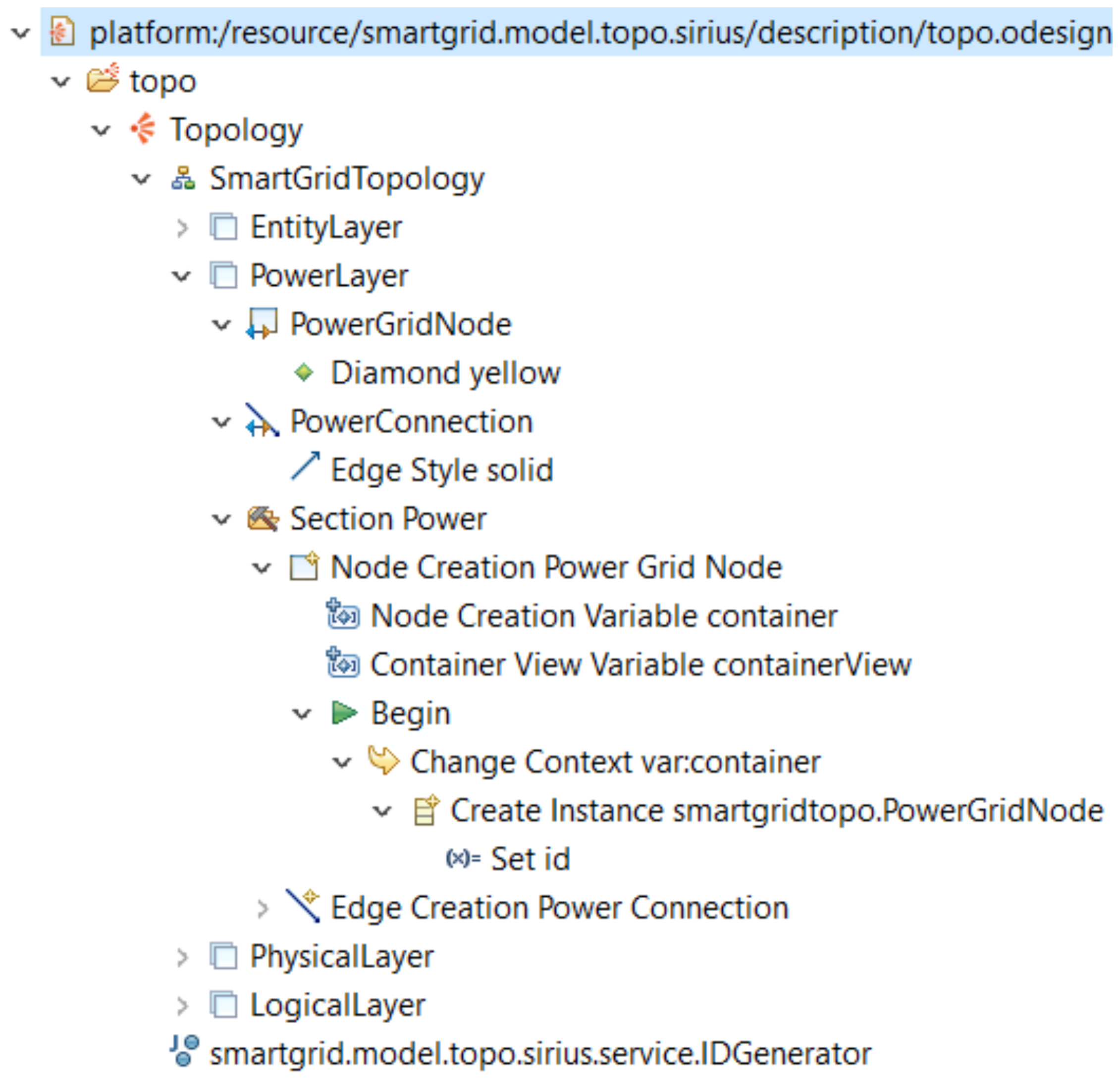


Figure 6.3: Legend of all notation elements in the Sirius core editor

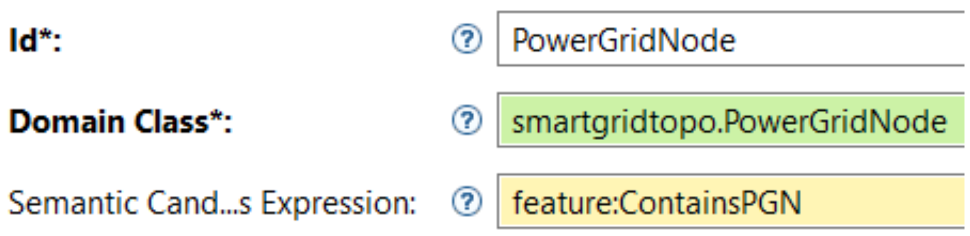


```
platform:/resource/smartgrid.model.topo.sirius/description/topo.odesign
  topo
    Topology
      SmartGridTopology
        EntityLayer
        PowerLayer
          PowerGridNode
            Diamond yellow
          PowerConnection
            Edge Style solid
          Section Power
            Node Creation Power Grid Node
              Node Creation Variable container
              Container View Variable containerView
              Begin
                Change Context var:container
                  Create Instance smartgridtopo.PowerGridNode
                    Set id
            Edge Creation Power Connection
        PhysicalLayer
        LogicalLayer
      smartgrid.model.topo.sirius.service.IDGenerator
```

Figure 6.4: Screenshot of the odesign file realizing the topography meta-model

6.3.1.1 Creating a New Viewpoint

A typical editor created with the Sirius framework is specified in an *odesign* file. Figure 6.4 thereby shows the odesign for our core editor. In order to define representation elements and their rules for creation and editing, we first have to define a viewpoint, which encapsulates all of this information. The viewpoint is shown in the figure right below the folder symbol having *topo* written next to it. The viewpoint's name we used for the definition of our core diagram editor is *Topology*. Other than the name of the viewpoint we can also define the model file extension this viewpoint can be applied to. Therefore, the names of the model file extensions have to be given. Since we only want to use this viewpoint for .smartgridtopo model files, we should insert this as model file extension or simply enter *, in order to support all file extensions. The concrete models that are actually supported, are models containing a special root node element specified in the diagram description. As we define only one diagram description with



Id*:	PowerGridNode
Domain Class*:	smartgridtopo.PowerGridNode
Semantic Cand...s Expression:	feature:ContainsPGN

Figure 6.5: Screenshot of the main properties for representing a power grid node

smartgridtopo.SmartGridTopology as root node element and those elements can only exist in .smartgridtopo model files, only these files are supported.

6.3.1.2 Representation of Notation Elements

As shortly stated in the previous subsection, we use a diagram description to define which element should be the root editor node for every diagram we want to create. Besides diagram descriptions Sirius offers further editors we could create, such as table descriptions or tree editors. Nevertheless, the extension types for graphical editors discussed in chapter 4 are based on editors that support nodes, container structures and connections and therefore we only consider diagram descriptions in this thesis.
Inside a viewpoint we can add multiple different descriptions to the viewpoint, which is why we need to identify each description by adding an ID to it. In our case the ID of the diagram description is *SmartGridTopology*, as seen in figure 6.4. When adding an ID and defining which model element is used as root node, we can start adding notation elements to the description. Adding notation elements for representation purposes requires *layers*. We have chosen to use overall four layers in this core editor. In each layer, both the representation and the creation of those notation elements specific to this layer are defined. Therefore, the first layer contains all entities like smart meter, intercom, generic controller and control center. The second layer, as seen in figure 6.4, contains the power grid node and the power connection. The third layer is responsible for the representation and creation of network nodes and physical connections, while the last layer only contains logical communications. In Sirius each of these layers can be hidden resulting in the masking of all its containing elements in the editor.
If we want to add a notation element for representation to the editor, we have to define it inside a layer. As an example, we take a closer look at the *PowerGridNode* in figure 6.4. We first define that we want the power grid node to be actually displayed as a node and add a style to it. In this case, we choose a yellow diamond as representation. For this to work, we also need to add some properties to the *PowerGridNode*. Those properties can be seen in figure 6.5. When creating a new notation element for representation purposes, we have to define an ID for this element first. After that, we need to tell the node to which meta-model element the node should refer to. In out current example this is the *smartgridtopo.PowerGridNode*, as shown in the second row of the figure. The last row in this properties sheet is the *Semantic Candidates Expression*. This is the only entry that is optional but should always be used, especially, if more than one model exists and is

opened in a Sirius diagram. The Sirius documentation for those expressions states the following:

- **Semantic Candidates Expression:** Restrict the list of elements to consider before creating the graphical elements. If it is not set, then all semantic models in session will be browsed and any element of the given type validating the precondition expression will cause the creation of a graphical element. If you set this attribute then only the elements returned by the expression evaluation will be considered.[3]

Since we didn't set a precondition, all power grid nodes of all available smartgridtopo models would be shown, if this expression is not set. With *feature:ContainsPGN*, we first address only the smartgridtopo model for which we create a representation and not all of them. Second, we restrict the represented power grid nodes to the ones that are returned, when calling the *ContainsPGN* composition in the meta-model. An equivalent semantic candidates expression, which we also use is *[self.ContainsPGN]*. When creating a new connection for representation purposes, there are a few other fields we have to fill out. First of all, we have to choose nodes and containers the connection can be applied to as source and target notation element, as well as an expression on how to find the target. These information need to be applied for both relation-based connections and element-based connections. The difference between those two is that the element-based connection corresponds to a specific meta-class leading to further information that need to be given. Those information are on the one hand the domain class, like in figure 6.5, and on the other hand an additional expression leading to the source.

6.3.1.3 Creation of Notation Elements

As our notation elements are now represented, we also want to be able to create all of these notation elements within the diagram. Therefore, we first need to create a section inside one of our layers. In this example, we provide one section to each layer. It is also possible to put all tooling concerns, such as node or container creation, delete features, double click actions and so on into one section. In our example, we consider again the power grid node and its creation. As we can see in figure 6.4, we are now in the section *Power* and inside the *Node Creation Power Grid Node* item. The first two entries consider only variables that can be used throughout the creation of the node. The first variable returns the container notation element. For our purposes, this is the only variable necessary. This is in our case the *SmartGridTopology* model element, on which this diagram is based. The third entry is the starting point of our creation. We first switch the context to our root node element under which we want to add a new power grid node. Second, we create a new instance of that node. Thereby, we have to enter a reference name, in which the new instance will be stored. As there is only one way to access all power grid nodes, we enter *ContainsPGN*, as we did for the creation of the representation of this node. The last item we use, when creating a power grid node, is to set the ID. Since we do not want, and there is no need for it, to manually type the ID for each new power grid node, we use a java service to provide one randomly. How the use of java services works, is described shortly within the next section.

[3]https://wiki.eclipse.org/Sirius/Tutorials/StarterTutorial

6.3.1.4 Java Services

Java services are always used, if the Sirius framework itself doesn't provide the required functionality. This may be because the functionality required is too specific or there are further validations required than the framework itself could provide. The basic implementation of a java service requires four steps [4], which are described below.

- **Create Java Class:** The java class we want to use as a service, according to the tutorial, should be placed in a new source folder *services* within our project containing the odesign description file. However, the service class can be placed anywhere in any project, as long as there is a dependency from our viewpoint specification project to the project containing the java service. The class itself should have a standard constructor with no arguments, as an instance of this class is created automatically by the Sirius framework whenever a method of this class is needed.

- **Implement Java Service Methods:** For the java service method to be valid and accessible from inside an editor, each method has to follow a specific signature. The method should be public and should return either a primitive data type or an ecore-based object for example *EList*, *EObject* or any subtype. Furthermore, the method needs at least one parameter which should also be an ecore-based object. The method signature for our ID generator thereby looks as described in listing 6.1

  Listing 6.1: Method signature of ID generator

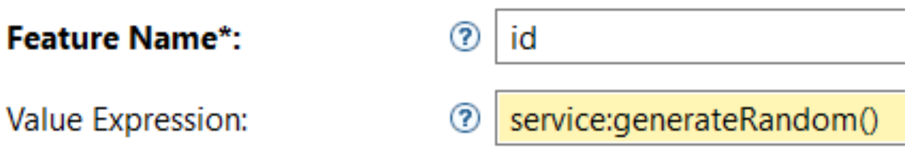
  ```java
  public int generateRandom(EObject obj)
  ```

  We here do not need to further specify the parameter, as we don't need it in order to generate a random number.

- **Let the Diagram Description Know the Service Class:** Before we can actually use the service class in our editor specification, we first need to let the editor specification know that the class actually exists. Therefore, we add a *Java Extension* in our odesign file and enter the qualified java class name of our java service. The qualified java class name is thereby the combination of the package and the class name of our service. This is shown at the bottom of figure 6.4.

- **Using the Service:** For an easier understanding of how we can use the java service, figure 6.6 is given showing the properties of a set action setting the ID of an entity with the help of a java service. The feature name is thereby the name of the attribute we want to set. In this case it is the ID. Below, in the expression field, we can use our java service. The service is thereby called by entering *service:* followed by the method we want to call. The parameter, if there is only one, doesn't need to be entered as the EObject is automatically transferred to the method by the framework. The object transferred is thereby dependent on the context. Since we now consider a new instance of a power grid node, the new power grid node instance is transferred.

[4] https://wiki.eclipse.org/Sirius/Tutorials/AdvancedTutorial



Feature Name*:	id
Value Expression:	service:generateRandom()

Figure 6.6: Screenshot of the properties of the set action when using a java service

A java service can be used wherever an expression is required. This includes semantic candidates expressions, value expressions, preconditions and further.

6.3.2 Graphiti Implementation

While we simply can create a diagram based on an existing model in Sirius, this is not possible in Graphiti at first. We can either create a new Graphiti editor that creates a new topology model from scratch with the help of a wizard or we can create a plug-in receiving an existing topology model and creating a diagram on that basis. Since we may have also existing topology models, the latter makes more sense and is discussed in subsection 6.3.2.3. However, first of all we need to define the diagram we can use for a topology model. Where we need a viewpoint specification in Sirius, we have to implement specific classes in Graphiti, which are described in the next subsection. As soon as we created the diagram, we also need to implement the representation and creation of all the notation elements, which is done in section 6.3.2.2. The implementation in Graphiti for the topology and later the input and output model extensions is based on previous work. Further information can be gained here [5]. As we are forced to write java code, when dealing with Graphiti and can not click our editor together like we can when using Sirius ,there are a lot of listings presented in all Graphiti-based sections. All of these listings are explained in detail after they are first referenced, so that each section should be understandable even without reading every line of code.

6.3.2.1 Implementing a New Smart Grid Diagram

While we needed to define a viewpoint and a diagram description in Sirius, we also have to define a basic diagram representation that can be used for smart grid topology models. In contrast to Sirius, we have to implement the java code by ourselves. Since we are using Eclipse and our new graphical editor can be seen as a plug-in to the Eclipse platform, we, of course, need to implement a few extension points, in order to place our graphical editor in the Eclipse platform. In Sirius, this is done automatically, when creating a new viewpoint specification project. For a new Graphiti editor we need to implement three extension points that are explained in the following.

org.eclipse.graphiti.ui.diagramTypes This extension point only states the diagram type ID, which is needed whenever we want to refer to any smart grid topology graphical editor based on Graphiti. There is no class that has to be implemented but only the name of

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

the diagram type, the actual self defined type, which has to be unique, and an ID for this extension point has to be defined.

org.eclipse.graphiti.ui.diagramTypeProviders According to the Graphiti developer guide[6], this extension type is used to register custom diagram type providers. The custom provider must understand the given diagram type provided by the previous extension type and therefore be suitable for editing and viewing diagrams of that type. The class we need to implement is required to implement the *IDiagramTypeProvider* interface. This class is by far the most important class of our editor, as it not only contains the standard diagram behavior instance handling the complete behavior in our diagram, but also the feature provider. The feature provider contains all sorts of notation elements we want to use in our editor. In our case, these are patterns containing all sorts of features as listed in section 2.7.1. Since we provide our own meta-model, we should also define a custom feature provider. Basically, the feature provider is implemented as follows. First, the constructor adds all patterns that we want to represent in the editor. The definition of a pattern is analyzed in section 6.3.2.2. Next, since we also want to be able to use mouse-over buttons in later extensions, we already defined a method, which adds context buttons to the mouse-over view. Therefore, another extension point is defined that all extensions can implement. This is exemplary shown in section 6.4.4.3. So far, we could already use the feature provider as it is and would be able to view and create all element-based patterns. However, the *PowerConnection* in our meta-model is not element-based but only relation-based meaning that there is no meta-class defining the power connection. Therefore, we also need to override most of the *feature* methods, namely *getAddFeature*, *getCreateConnectionFeatures*, *getRemoveFeature* and *getDeleteFeature* to adjust them to our needs. In our editors adding simply means creating a notation element, while creating also means the creation of the corresponding business model element. Removing and deleting can thereby be regarded the same way, as removing only removes the notation element from the diagram and deleting also removes the corresponding business model element. The implementation of these methods is straightforward and exemplary shown for the *getRemoveFeature* method in listing 6.2.

Listing 6.2: Get remove feature implementation regarding the relation-based power connection

```
public IRemoveFeature getRemoveFeature(final IRemoveContext context) {
  if (context.getPictogramElement() instanceof Connection) {
    Connection con = (Connection) context.getPictogramElement();
    NetworkEntity start = this.getNetworkEntity(con.getStart());
    PowerGridNode end = this.getPowerGridNode(con.getEnd());

    if (start != null && end != null) {
      return new RemovePowerConnectionFeature(this);
    }
  }
  return super.getRemoveFeature(context);
```

[6] http://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.graphiti.doc%2Fresources%2Fdocu%2Fextension-points%2FdiagramTypeProviders.html

```
}
```

The context given as parameter is always dependent on the method called. Given the *getAddFeature* method, the context would be an implementation of *IAddContext*. In the second line of the listing, we check whether the pictogram element contained in the given context is a connection. If not, we can delegate to the super class method and let the remove action be handled there. If we are dealing with a connection, we check whether the given connection starts at a network entity and ends at a power grid node. If that is given, we are dealing with a power connection. Otherwise, a logical communication or a physical connection should be removed currently. However, if the source of the connection resembles a network entity and the target resembles a power grid node, we can return a *RemovePowerConnectionFeature* instance, where the logic of removing a power connection is handled.

org.eclipse.ui.editors This extension point is the only one, which is actually optional. A Graphiti diagram would now already open, if a wizard or a topology model loader is implemented, which is shown in section 6.3.2.3. However, as we would like the diagram to be more extensible we provide our own version of a Graphiti diagram and therefore, we need to implement the given extension point. This extension point requires us to implement the *IEditorPart* interface. This is easily achieved by extending the Graphiti diagram class called *DiagramEditor*. The only method we override in this class is the *createDiagramBehavior* method, as we provide our own diagram behavior. Although, we can access the diagram behavior from the diagram provider class implemented in the previous extension point, Graphiti doesn't provide methods to set the diagram behavior in the provider class. That is why we need an extended diagram editor. Our self defined diagram behavior adds a new *ResourceSetListener*, as we know that the given editor is going to be extended. This listener lets us know, whether a resource inside the given resource set changed and needs to be saved along with the other resources. In Sirius this is automatically done, if a resource is added to the current Sirius session described later in section 6.4.2.

6.3.2.2 Creating Notation Elements in Graphiti

Like in Sirius, we need to define each node and connection we want to use separately. Instead of clicking our desired representation together, we need to implement so called *patterns* in plain java code. For each pattern we can choose whether we want to be able to only represent the underlying domain model, or, if we also want to be able to create new notation elements with a palette entry and other features, such as resizing, moving, removing or deleting of that notation element. Like in Sirius, we can link each notation element with a domain model element making it easier to implement such a delete feature. Since the procedure for the implementation of such a pattern is almost the same for each notation element, we analyze the implementation of the control center as container and the logical communication as domain model element-based connection and the power connection as relation-based connection.

Control Center The basis for each entity in our scenario is given by the abstract class *AbstractFormPattern*, which extends *AbstractPattern* and implements basic functionality needed by each pattern. Together, both of these abstract classes implement functionality, such as creating, moving, updating or resizing of all concrete patterns we want to implement. For a more detailed view on the implementation of the *AbstractPattern* class, refer to the documentation in [11]. The *AbstractFormPattern* class further implements methods, such as *canAdd* and *canCreate* that return true, if the location we want to place our pattern is the diagram itself. For using the control center in further extensions, these methods should be overridden, as the control center should be used as container.
For the representation of the control center or any other entity we take a closer look at the *add* method implemented in *AbsractFormPattern*, which is shown in listing 6.3.

Listing 6.3: Implementation of the add method in AbstractFormPattern for the representation of an entity

```
 1 public PictogramElement add(final IAddContext context) {
 2   Diagram targetDiagram = (Diagram) context.getTargetContainer();
 3   IPeCreateService peCreateService = Graphiti.getPeCreateService();
 4   ContainerShape containerShape = peCreateService.createContainerShape(targetDiagram,
         true);
 5
 6   // add a chop box anchor to the shape
 7   peCreateService.createChopboxAnchor(containerShape);
 8
 9   IGaService gaService = Graphiti.getGaService();
10   GraphicsAlgorithm shape = this.getGraphicalPatternRepresentation(containerShape);
11   gaService.setLocation(shape, context.getX(), context.getY());
12
13   this.linkObjects(containerShape, context.getNewObject());
14   return containerShape;
15 }
```

The first three lines of the given method simply create a container shape and add it to the current diagram. The container shape is important, since we then have the possibility to add further shapes in different extensions. The *ChopboxAnchor* created in line seven makes sure that all connections targeting our entity are placed in the middle of the border of our container shape. The next three code lines manage the graphical representation of the entity and its location in the diagram. This is the only part, which is specific for each entity, as each entity type should be represented differently. Therefore, the method *getGraphicalPatternRepresentation* is abstract and needs to be implemented by each entity pattern. Basically, the shape, color and possible children shapes are defined in that method, which is shown in listing 6.4. The last two lines in listing 6.3 link the current container shape with the underlying business object. In case of the control center pattern this object is the desired control center.

Listing 6.4: Implementaiton of the control center's graphics algorithm

```
protected GraphicsAlgorithm getGraphicalPatternRepresentation(ContainerShape
    containerShape) {
```

```
 2     IGaService gaService = Graphiti.getGaService();
 3     Rectangle rect = gaService.createRectangle(shape);
 4     rect.setWidth(200);
 5     rect.setHeight(100);
 6     rect.setForeground(new ColorConstant(128, 128, 128));
 7     rect.setBackground(new ColorConstant(255, 255, 255));
 8     rect.setLineWidth(ConstantProvider.shapeLineWidth);
 9
10     MultiText inside = gaService.createMultiText(rect, "<<ControlCenter>>");
11     inside.setHeight(25);
12     inside.setWidth(200);
13     inside.setX(rect.getX() + 1);
14     inside.setY(rect.getY() + 1);
15
16     return rect;
17   }
```

Regarding listing 6.4, it becomes clear that the container shape given as parameter only represents the frame for our concrete control center. In case of the control center representation, we first create a new rectangle, which we want to use as container for further nodes and set its color, size and line width. Since we want the user to know that this is a control center, we also have to add another label to it, which is done in lines ten to 14. For other entities this method is, of course, implemented differently. However, this listing should serve as basic idea of how to implement those entities.
Since we not only want to represent the control center and the other entities, but want to have a palette entry for creating these entities we also have to override the *create* method in the *ControlCenterPattern* class. As we need to create the business object, as well as the graphical representation, we start by getting our *SmartGridTopology* root container class from the current diagram. Based on that topology model we create a new control center, add it to the list of network entities in the *SmartGridTopology* object and call the *addGraphicalRepresentation* method, which does the rest for us, as we already implemented all the necessary methods.
The last step we need to do before the control center (and the other entities) are active and can be used in the diagram is to switch to our *SGSFeatureProvider* class and add the line in 6.5 to the constructor of our feature provider.

Listing 6.5: Adding a new pattern to the Graphiti diagram

```
this.addPattern(new ControlCenterPattern());
```

Logical Communication As basis for each element based connection we also use an abstract class called *AbstractConnection*, which itself extends the abstract class provided by Graphiti called *AbstractConnectionPattern*. The procedure for implementing the *create* method is similar to the implementation of the control center shown in the previous paragraph. The difference is that we not only have to create a line, instead of a rectangle, we also have to set the anchors of the connection accordingly. Therefore, we added a chop box anchor for every entity pattern, which can be addressed by the *ICreateConnectionContext*, which is given as parameter for the create method of our logical communication

pattern. Other connection contexts are given accordingly in every other method provided by *AbstractConnectionPattern*. The given context allows us to access the source, as well as the target anchor. From these given anchors and their parent pictogram elements Graphiti provides a method with which we can access the business model element according to the given pictogram element. The parent of each chop box anchor is the pictogram element representing an entity.
For the power connection, the pattern can stay the same except that in the create method we, of course, can not create a power connection in the underlying business model as the power connection only exists as relation. Therefore, we need to set the *ConnectedTo* attribute each *NetworkEntity* instance contains. That means that we can apply a connection pattern to element-based edges, such as the logical communication, as well as on relation-based edges, such as the power connection.

6.3.2.3 Creating a Representation for a Topology Model

As mentioned earlier we have two options, if we want to use a Graphiti diagram. The first one is to create a wizard and along with the creation of the diagram to create the topology model resource. The second option is to load an existing topology model and build a diagram upon that model. As this is more comfortable we only describe the second option. However, the knowledge needed on Graphiti is the same for both options. Moreover, since it is not possible to create a diagram representation like we do in Sirius, we have to implement our own plug-in providing this functionality independent on whether we want to use the wizard or loading an existing topology model. Therefore, we create a new toolbar button for loading a topology model, as it is described later in section 6.4.2 for the input model extension. After the topology model is loaded, we can start creating the diagram. As the implementation is only partly dependent on Graphiti, we more or less skip the Graphiti independent parts by only describing them shortly. Creating and filling a diagram in Graphiti with content basically consists of four steps. The first one is creating the diagram itself and connecting it to the topology scenario. Then, we need to add all necessary topology elements to the diagram. As all elements are placed in the upper left corner, we also should apply a layout algorithm to the notation elements. The last step necessary is to save the diagram. As the last two steps are mostly independent of Graphiti and can be applied to any resource, it is only mentioned for the sake of completeness. The other two steps are explained in detail in the following.

Creating a New Diagram Apart from creating a file containing the future diagram, we also need to configure the diagram itself. Therefore, listing 6.6 is given. The given listing is embedded and executed in a recording command, as otherwise an exception would occur, when executing the code.

Listing 6.6: Creation of a Graphiti diagram

```java
diagramResource.setTrackingModification(true);
final Diagram diagram = Graphiti.getPeCreateService().createDiagram("
    SmartGridSecurityDiagramType", diagramName, 10, true);
// link model and diagram
```

```
4 final PictogramLink link = PictogramsFactory.eINSTANCE.createPictogramLink();
5 link.setPictogramElement(diagram);
6 link.getBusinessObjects().add(scenario);
7 diagramResource.getContents().add(diagram);
```

The listing starts with setting the tracking modification to true, as we want to track changes in the topology model, as well as in the diagram. Line two creates the diagram itself. Therefore, we need to provide the correct diagram type ID, which we defined in our extension point explained earlier in section 6.3.2.1. Apart from the name of the diagram, we also provide information on the grid size of the later diagram and whether the diagram should snap to the grid. After the diagram creation is done, we again have to link the diagram with the current topology scenario, which is done in lines four to six. The last line in this listing only adds the diagram to the resource we created earlier. Now loading the resource automatically loads the created diagram.

Adding Relevant Entities Since we can access the topology scenario, we also have access to each entity and connection available in the current model. Therefore, the adding of these entities can be done straightforward by iterating over all entities and then adding each of them to the current diagram. Listings 6.7 and 6.8 show how the network entities can be added to the diagram.

Listing 6.7: Preparation of each entity before adding them to the diagram

```java
private void addElements(final Diagram diagram, final Object newObject) {
  final AreaContext area = this.createAreaContext();
  final AddContext add = new AddContext(area, newObject);
  add.setTargetContainer(diagram);
  add.setNewObject(newObject);
  ...
}
```

The method given in listing 6.7 takes our diagram and an entity, such as a smart meter as parameter. We then need to create an area context defining the location, where the new notation element should be placed. As we apply a layout algorithm later this position may be arbitrary inside the diagram. Next, we add an add context defining that we want to add our new entity to the diagram. To actually add the newly defined context containing the element to the diagram we again need a *RecordingCommand* with the code line given in listing 6.8 to be executed.

Listing 6.8: Adding entities from inside a recording command

```java
GraphitiHelper.getInstance().getFeatureProvider().addIfPossible(add);
```

This code line simply calls the *addIfPossible* method of our feature provider, which is responsible for adding, creating, deleting or altering all notation elements inside the diagram. The procedure for adding connections is quite similar to adding entities to the diagram. The difference between these two actions is that for a connection we need an *AddConnectionContext*. Creating a new *AddConnectionContext* requires two anchors as parameters,

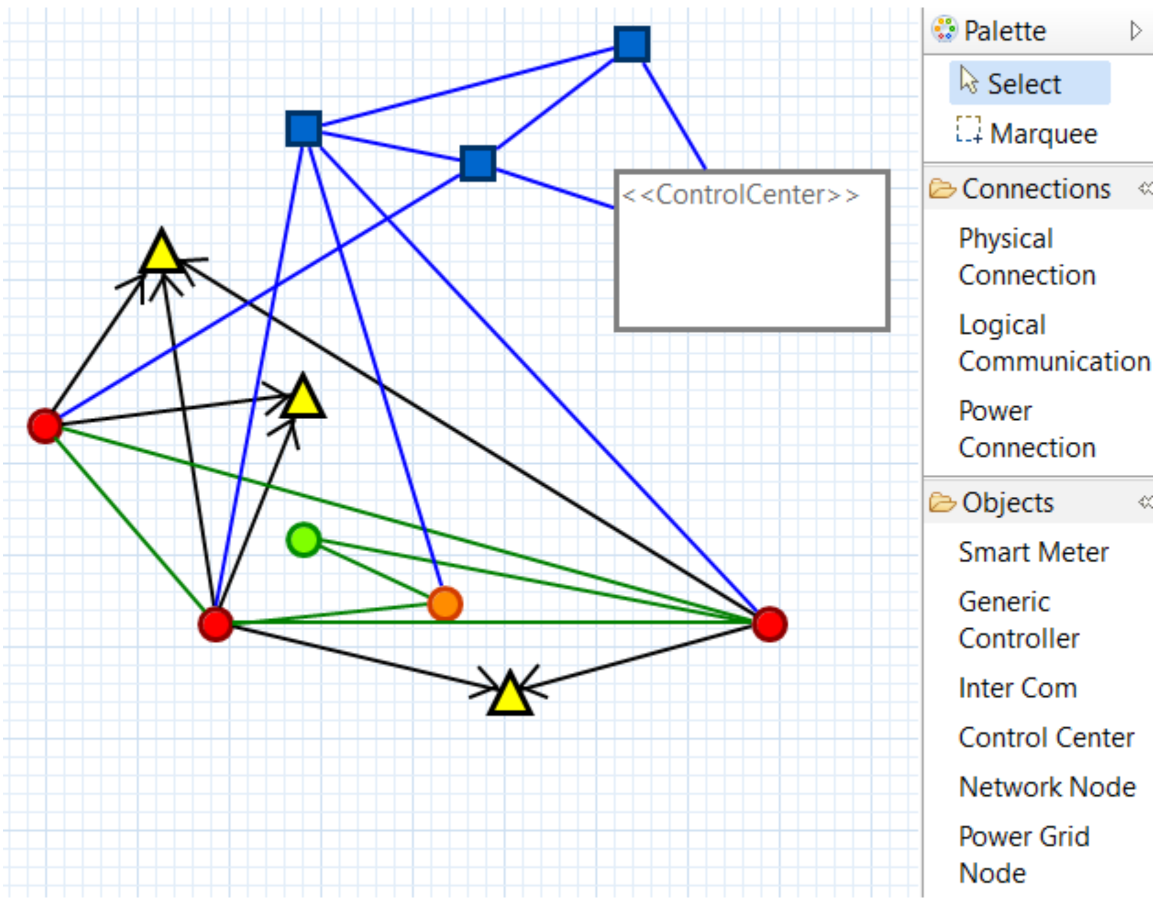


Figure 6.7: The core editor for our running example in Graphiti

which we get from receiving the pictogram elements to the business objects the connection connects. Therefore, it is important that we create the pictogram elements for the entities first, as we need them to access their anchors. Now that we have created all entities with their connections in our diagram we only need to layout them. This can be done with any layouting algorithm and is used as custom feature. This means that we only need to reference the feature provider, while the rest of the implementation is independent of Graphiti.

The result of loading our running example can be seen in figure 6.7. The elements' representations are the same in Sirius as well as in Graphiti. The only change is that the power grid nodes are represented as triangle and the power connections are black instead of yellow.

6.4 The Input Model Extension

In this section we cover the first extension for our smart grid resilience framework. Therefore, the section is divided into first the analysis of the meta-model including its extension types and mechanisms. After that we discuss how a second model is loaded to the existing diagram for both Sirius and Graphiti. Section 6.4.3 then deals with the

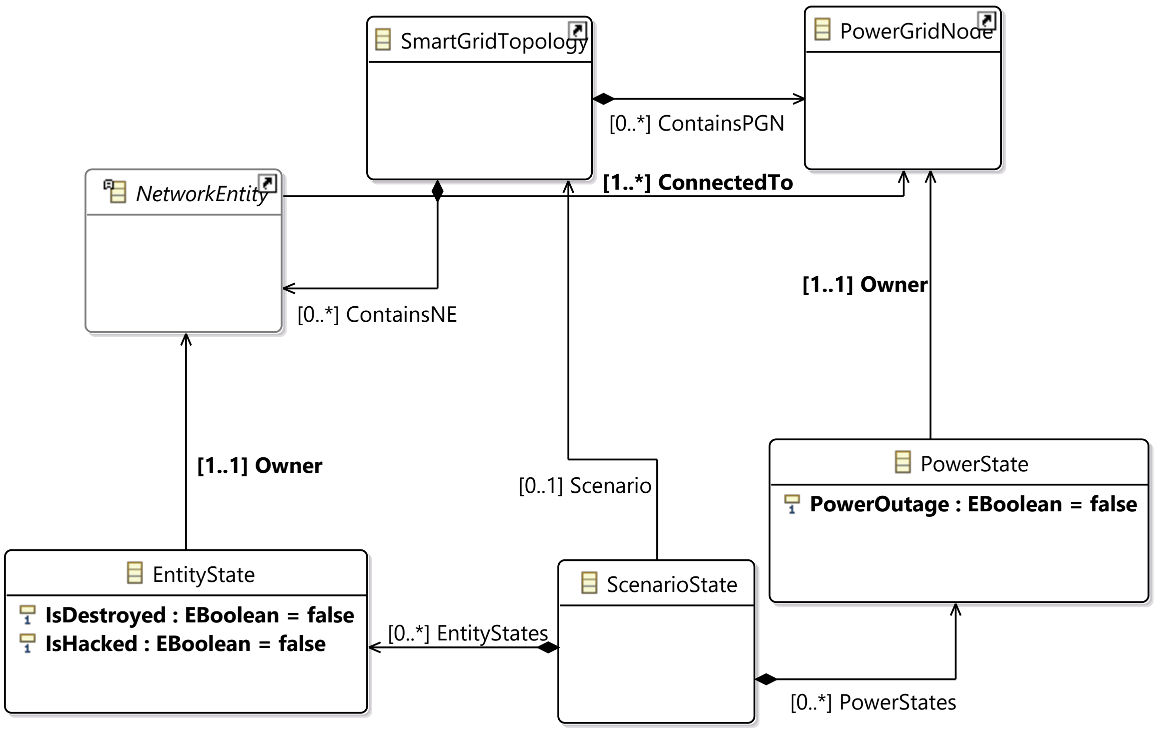


Figure 6.8: Meta-model of the input model extension

Sirius implementation of the actual editor, while the last section covers the Graphiti implementation.

6.4.1 The Input Meta-Model

The first of the three extensions for our smart grid resilience framework is the input model given by the meta-model in figure 6.8. This meta-model aims at adding states to the existing meta-model elements of the topology meta-model. The top of the figure shows those meta-classes that already exist in the core meta-model, while the bottom of the figure shows the new meta-classes available in the input model extension. As we can see, this extension only contains three additional meta-classes. Since we want to be able to create smart grid input models, there has to be a root model element. In this case the root element is *ScenarioState*, which contains the two other meta-classes. Furthermore, a *ScenarioState* references a *SmartGridTopology*, the root element of the core model. As already mentioned, the two remaining meta-classes in the extension aim at providing states to existing core model elements. On the one hand, there is a *PowerState* meta-class, which tells the modeler whether there is a power outage or not. Naturally, such a power state needs a power grid node, where this state can be applied to. This is realized by an unidirectional association to the meta-class *PowerGridNode*. If only this extension without the core meta-model would be considered, we would have a meta-class extension type with an attribute extension type and a relation extension type represented by the association. However, as we need to focus on the developers intent, we here only contribute to the attribute extension type, as the power outage should clearly be an attribute of the power grid node. The same

extension mechanism is also used to relate an *EntityState* to a *NetworkEntity*. Entity states indicate whether a given network entity is destroyed or hacked. Network entities, as seen in section 6.2, are all elements we implemented as notation elements in the core editor, such as the control center, network nodes, smart meters, generic controllers and intercoms. Since we have only one extension type, according to our classification, we need to think about how to best realize these extension types on graphical editor level. Given the attribute extension type, our mapping in section 5.3 suggests eight different extension types, where four can be used for the representation, while the other four should alter the value of the attribute. As we ca not implement all of them, we choose to change the appearance of a power grid node, if its power is out. For the other entities we also choose a change in the appearance depending on the hacked and destroyed status. The only exception thereby is the control center, where we add annotations to it regarding each status. Changing the value is implemented as context dependent menu button.
The next subsections give detailed information on the actual implementation in Sirius, as well as in Graphiti. But first, adding a new toolbar button to load possible extensions is explained.

6.4.2 Adding a Second Model to the Editor

When representing the input model in an editor, it becomes clear that for this to work also information of the corresponding topology model needs to be present. As the input model represents an extension to the topology model, we would prefer to have both models at the same time active in one diagram. If that is given, we can alter both models at the same time and get a better view on how they interact together. In Sirius this is no problem, as all models present in the same project containing a viewpoint description, are automatically loaded to the current session. We only need to check whether, for example a certain input model fits to the current topology model. However, if the model is not present in the current project, we need to load this model into the current session. In Graphiti the model needs to be loaded in any way, as there is no equivalent to the Sirius session. In order for this to work, we introduce a new toolbar button with which a second model, and possibly further models, can be loaded. The model can thereby only be loaded, if the diagram already displays a *smartgridtopology* model. Since the Sirius framework offers its own toolbar, we can add a new toolbar button at the end of the standard toolbar. In Graphiti, we have to use the basic Eclipse toolbar, as Graphiti doesn't provide its own. Therefore, both of these approaches unfortunately require a different procedure, which is why we first discuss the Sirius implementation of the toolbar button and then the Graphiti implementation on its own.

6.4.2.1 Adding a Toolbar Button to the Sirius Toolbar

For the toolbar button to work as intended, we overall need three extension points all provided by the Eclipse platform. The following enumeration lists and explains all three of them.

- *org.eclipse.ui.commands*: This extension point is needed to tell the program what to do after the button is pressed. The class implementing this command should extend the *org.eclipse.core.commands.AbstractHandler* class, as then only the execute method has to be overridden. The basic course of action is here still independent of the framework used. We first access the shell and from there open a new file dialog, from which we can choose the input model we want to load. After that, we create and load a new resource containing the file that was just chosen. The last step is to access the current editor and add the resource to the list of resources that are currently active in the editor. Since only the last part is really specific to Sirius, listing 6.9 only covers these few lines.

Listing 6.9: The adding of a new model to the current Sirius session

```
 1  IEditorPart part = PlatformUI.getWorkbench().getActiveWorkbenchWindow().getActivePage
        ().getActiveEditor();
 2  DDiagramEditor editor = (DDiagramEditor) part;
 3  final Session session = editor.getSession();
 4  TransactionalEditingDomain domain = (TransactionalEditingDomain) editor.
        getEditingDomain();
 5  final RecordingCommand cmd = new RecordingCommand(domain) {
 6
 7    @Override
 8    protected void doExecute() {
 9      session.addSemanticResource(r.getURI(), new NullProgressMonitor());
10    }
11  };
12  domain.getCommandStack().execute(cmd);
```

Lines one to three only retrieve the current session containing all loaded models and other resources, such as the current diagram. Since we can not alter the session directly, as it is currently used by the diagram, we have to contain the method call in line eight inside a *RecordingCommand*. That command is directly executed in the last line of the listing. To minimize failures, there should be a check first whether a valid input model is already loaded and remove that resource from the session before adding the new one to the list.

- *org.eclipse.core.expressions.propertyTesters*: Since the toolbar button should only be present, if a topology model is currently active in the editor, we need to implement this extension point as well. In general, this extension point is used to test a property and act accordingly. For our purpose, the property tester gets the current editor and tests, if the target model of the editor is a topology model.

- *org.eclipse.ui.menus*: The last extension point we need is the one which puts the last two extension points together. Since this extension point is a bit bigger than the other ones, in the following we analyze the code of the plugin.xml regarding this extension point, which is presented in listing 6.10.

Listing 6.10: Menu contribution extension point used to add toolbar button to the Sirius toolbar

```
 1  <extension point="org.eclipse.ui.menus">
```

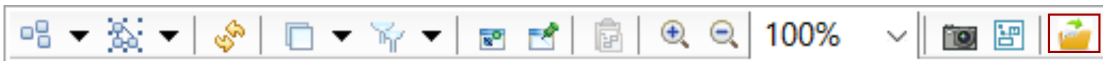


Figure 6.9: Screenshot of the extended Sirius toolbar

```
  <menuContribution
    allPopups="false"
    locationURI="toolbar:org.eclipse.sirius.diagram.ui.tabbar?after=additions">
   <command
     commandId="smartgrid.model.input.sirius.loadinputmodel"
     icon="icons/open-file-icon.png"
     style="push">
     <visibleWhen
       checkEnabled="false">
       <test
          property="smartgrid.model.input.sirius.propertyTester.isTopoLoaded">
       </test>
     </visibleWhen>
   </command>
  </menuContribution>
</extension>
```

First of all, a menu contribution is added to our extension point. The menu contribution explicitly states that our toolbar button should appear at the end of the Sirius toolbar. By pressing the toolbar button we invoke our previous written command, which here has the special attribute of only being visible if the property test returns true.

The overall result of the toolbar button implementation can be seen in figure 6.9. As we can see, the new toolbar button is listed at the end of the Sirius toolbar. The button is present as long as a topology model is shown in the current diagram. Thereby, we don't make a difference whether an element is selected or not. If the button should only be present, if no element is selected, an additional property tester is needed.

6.4.2.2 Adding a Toolbar Button to the Eclipse Toolbar

Since the standard Graphiti editor doesn't have its own toolbar, we need to either create a new extensible toolbar, when designing the core editor or we have to extend the Eclipse toolbar. Creating a new toolbar is an even harder task than adding a toolbar button to the Sirius toolbar. However, adding a new button to the Eclipse toolbar is in that way an easier task as we only need to use the ID of the Eclipse toolbar *toolbar:org.eclipse.ui.main.toolbar* and implement the button the same way as described the section before. Inside the execute method of our command we basically apply the same code as for the Sirius toolbar button. First, we retrieve the current shell and open a file dialog, where we can choose the input model. Next, we load the input model as resource and add it to the list of active models in our diagram. For Graphiti the last part is also contained in a *RecordingCommand* containing the two line shown in listing 6.11.

Listing 6.11: The adding of a new model to the current Graphiti diagram

```java
final EObject scenarioState = inputModelResource.getContents().get(0);
this.diagramContainer.getDiagramTypeProvider().getDiagram().getLink().getBusinessObjects
    ().add(scenarioState);
```

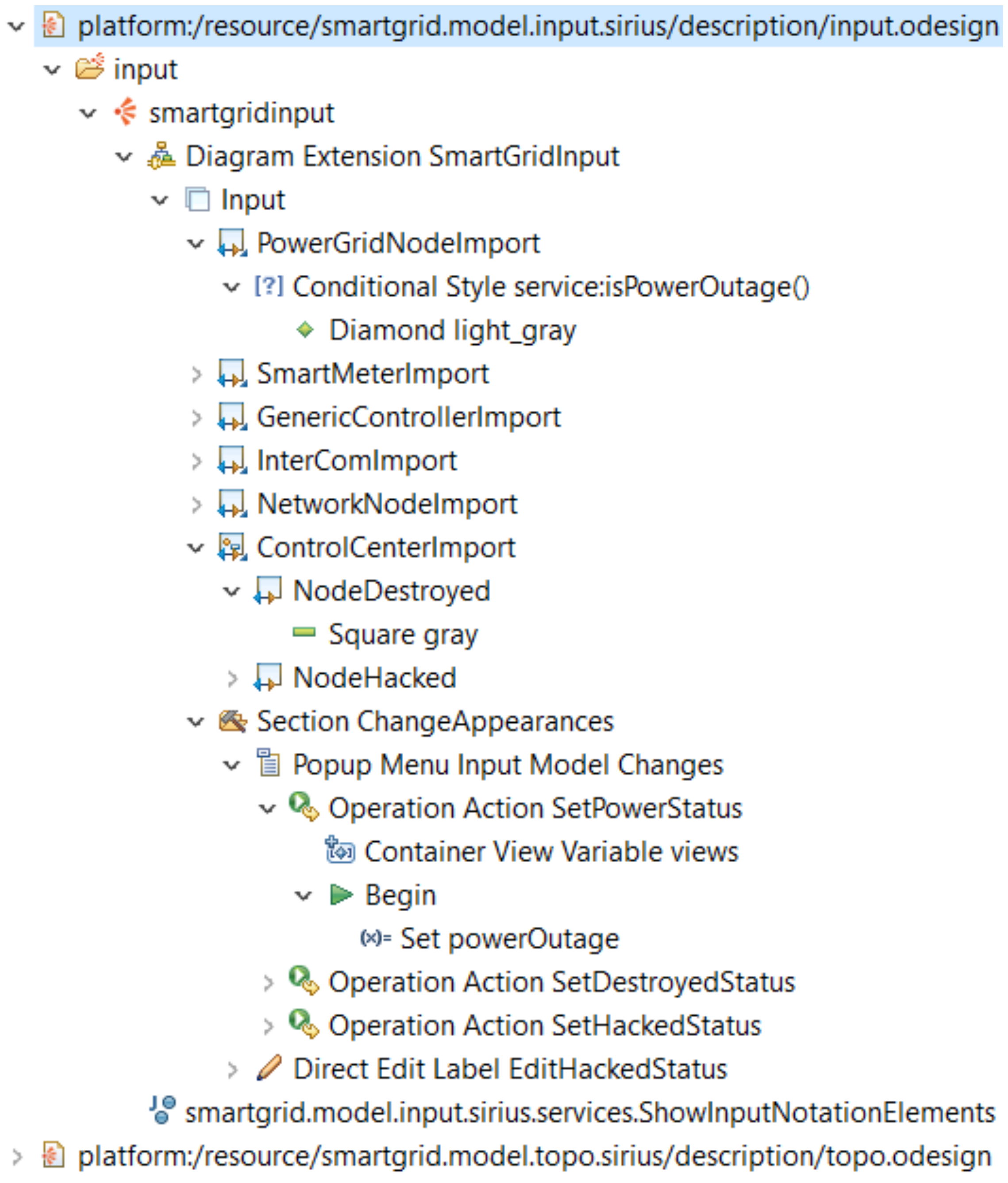
In the first line the actual scenario state object is retrieved from the resource we loaded one step earlier. The next line accesses the current diagram container, which stores the current diagram amongst other variables. From that diagram, we can access the underlying business models, such as our topology model and add the current scenario state to it. The result of this implementation is the same as in the section before with the difference that now we have a new toolbar button in the Eclipse toolbar. To minimize failures, there should be a check first whether a scenario state is already loaded and remove that scenario state before adding the new one to the list. This check does also need to be done for the Sirius implementation we discussed earlier.

6.4.3 Implementing the Input Model with Sirius

As we already discussed the possible mappings for the input model in section 6.4.1, we focus in this section on the actual implementation of these mappings in Sirius. Therefore, we first focus on how to extend a given diagram in Sirius and then focus on the change of appearance for power grid nodes, as well as on the other entities. The next subsection here deals with the possibility to change the value of an attribute from inside the diagram. Then, we discuss the adding of new annotations to the control center, as this is our only container element in the editor. For this case we also discuss possibilities to alter the values of the control center's attributes. For explanation purposes, figure 6.10 thereby shows the description file for the input extension.

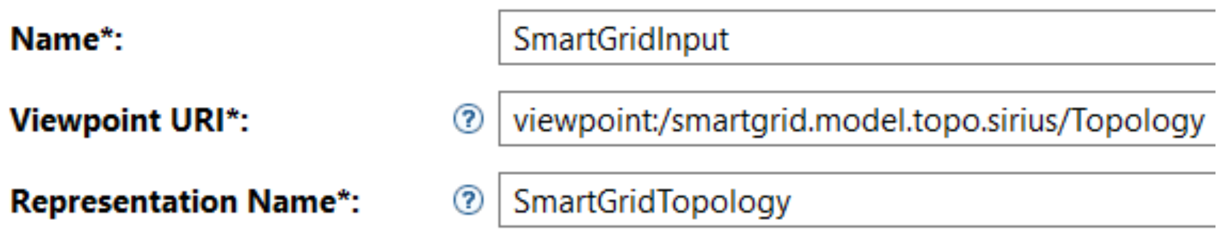
6.4.3.1 Extending a Given Diagram in Sirius

Since we already discussed how to load a second model into the current Sirius session, we now need to make sure this model can be used appropriately. Since we do not want to create an extended diagram programmatically and we don't want the core editor know the extension, we need to use the extension mechanisms given by the framework. Therefore, we first create a new plug-in with a new viewpoint specification model, as we already did, when creating the core editor. After the definition of the new viewpoint, we can add a so called *diagram extension* shown in figure 6.10, which we named *SmartGridInput*. For a better understanding of how such a diagram extension works, figure 6.11 is given. Since we want to extend the diagram description given by the topology model, we have to tell the diagram extension, where to find that description. The first properties entry thereby only states the name of our extended diagram description and can be chosen at will. The second entry specifies the viewpoint containing the diagram description we want to extend. The required URI is thereby composed of the keyword *viewpoint:/* followed by the name of the plug-in, where the viewpoint is located. The last segment of this URI simply states the viewpoint's name, where the diagram description to extend is located. Only those three segments make the URI unique, as there can be more than one description file with more



```
v platform:/resource/smartgrid.model.input.sirius/description/input.odesign
  v input
    v smartgridinput
      v Diagram Extension SmartGridInput
        v Input
          v PowerGridNodeImport
            v [?] Conditional Style service:isPowerOutage()
                Diamond light_gray
          > SmartMeterImport
          > GenericControllerImport
          > InterComImport
          > NetworkNodeImport
          v ControlCenterImport
            v NodeDestroyed
                Square gray
            > NodeHacked
          v Section ChangeAppearances
            v Popup Menu Input Model Changes
              v Operation Action SetPowerStatus
                  Container View Variable views
                v Begin
                    (x)= Set powerOutage
              > Operation Action SetDestroyedStatus
              > Operation Action SetHackedStatus
            > Direct Edit Label EditHackedStatus
        smartgrid.model.input.sirius.services.ShowInputNotationElements
> platform:/resource/smartgrid.model.topo.sirius/description/topo.odesign
```

Figure 6.10: The viewpoint description file for the input model extension



Name*: SmartGridInput

Viewpoint URI*: viewpoint:/smartgrid.model.topo.sirius/Topology

Representation Name*: SmartGridTopology

Figure 6.11: Properties of a diagram extension completed for the input model extension

than one viewpoint inside a single plug-in. The last properties entry labeled *Representation Name* should have the name of the diagram description we want to extend as value. In our case this is the *SmartGridTopology* diagram description.
After all these entries have been correctly filled, the extension is active as soon as we add the new viewpoint to the list of viewpoints of a modeling project. We should mention here that the extension uses the same root element as the core editor. That results in the use of a java service for almost all entities we want to add. In order to reference or use elements from the core diagram description, we should load this resource to our viewpoint specification. The next subsections thereby show how we can alter the appearance of a given notation element that was already defined in the core editor.

6.4.3.2 Changing the Appearance of a Notation Element

Since all attributes introduced in the input model are Boolean attributes, we can change the appearance of nodes in order to display the attributes value. A false attribute should thereby be represented in the node's original state, while the attribute being true actually changes the node's appearance. In order to change an existing element, we first have to import that element. Therefore, we first need to load the core editor into our extension. If that is done correctly, the URI of the odesign file appears at the bottom of the input model extension description, as seen in figure 6.10. After loading the core editor, we can import any of the given notation elements to our input model extension. All imported entities have the same properties as the originals in the core editor. We can then add a *conditional style* to one of these elements. This can also be seen in figure 6.10 for the *PowerGridNodeImport*. Conditional styles require a Boolean value. If the condition is met, the style is changed according to the style definition. In our case the yellow diamond of the power grid node changes to a gray diamond, if the condition is true. As we can not access the domain model elements of the input model directly, we need to add a java service to the editor. Regarding again figure 6.10 at the bottom, we added a java service class called *ShowInputNotationElements* to the diagram extension. For checking the status of a power grid node, we therefore use the service method *isPowerOutage()*. Listing 6.12 thereby shows the code for this method.

Listing 6.12: The *isPowerOutage* method of the java service class used in the input model extension

```java
public boolean isPowerOutage(PowerGridNode node) {
    boolean powerOutage = false;
    PowerState required = getCorrectPowerState(node);
    if (required != null) {
      powerOutage = required.isPowerOutage();
    }
    return powerOutage;
  }
```

The method *getCorrectPowerState()* uses the given power grid node to access its *SmartGridTopology* container element. After we retrieve the current Sirius session, which we already discussed in listing 6.9, we can access all its current semantic resources. From there, we can test each resource whether it is our scenario state container or not. Then we test, if the scenario state actually belongs to the given smart grid topology element by comparing the ID's of the current smart grid topology element with the referenced smart grid topology element from the given scenario state. If these ID's are the same, we have the correct scenario state. From there, the *getCorrectPowerState()* method simply compares each power state's owner with the given power grid node and, if there is a match, the power outage state is returned. This procedure is the same for the test, if an entity is broken or hacked. The main difference between changing the appearance of a power grid node and changing the appearance of a network entity is that we overall need three conditional styles for the network entity, in order to show all possible states and their variations. The first conditional style tests whether both attributes are true and changes the style of the node accordingly. The second and third conditional style only test whether one of these attributes are true. The conditional style testing whether both attributes are true must be placed in the first place, since Sirius performs only the first condition evaluating to true and ignores the other conditions. In order to prevent such a behavior we have to give the node in the extension in general a custom style that behaves according to certain conditions. Therefore, a custom *EditPart* and an *EditPartProvider* has to be implemented, which is not further discussed here. The result of our input model extension can be seen in figure 6.12. Gray nodes thereby represent destroyed entities, a gray diamond is as mentioned a power grid node without power. If a node is hacked, the label of the node changes to an underlined *H* and, if an entity is both destroyed and hacked, the node is gray and marked with an underlined H.
Now that we can represent additional styles, we need to can concentrate on adding annotations to the control center in order to represent the destroyed and hacked state in container elements.

6.4.3.3 Adding New Annotations to the Control Center

Adding an annotation works in a similar way than changing the appearance of an existing node. Instead of a conditional style, we simply add a new node to the imported control center. The labeling works again with the help of a method in our java service class. The method works similar as the method described in listing 6.12 with the difference that we can not return a Boolean value but must return a string for a correct labeling. The reason behind this is that we can not combine service calls with other acceleo expressions. Therefore, in case of the destroyed node, the returned string starts always with *destroyed* = and then alters between true or false depending on the value.

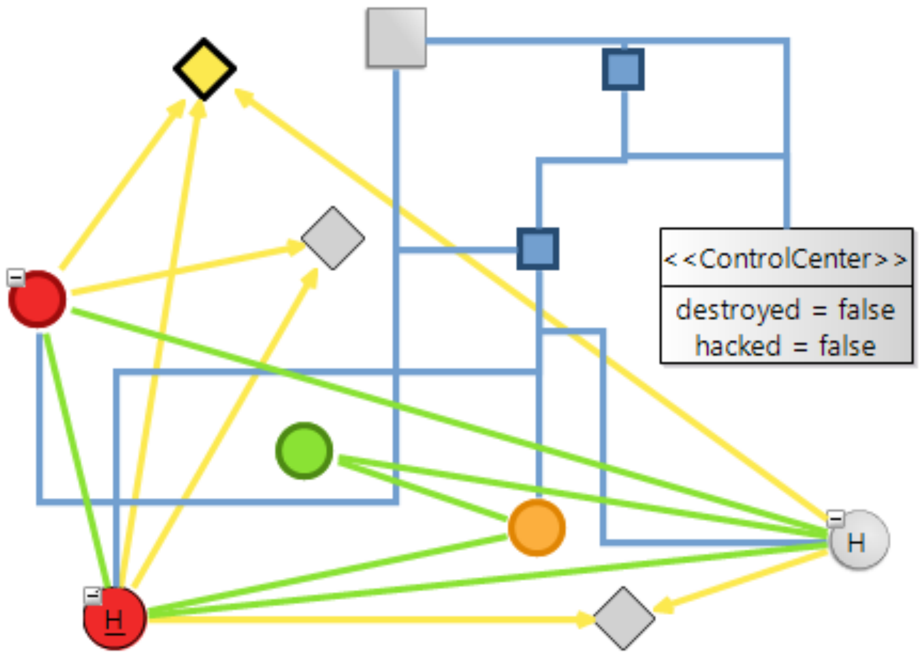


Figure 6.12: Our running example with a loaded input model

The result of the new annotations can also be seen in figure 6.12. The control center now has two additional annotations showing the destroyed and the hacked status. If this status needs to be changed in the diagram directly, there are basically two different options the developer has. The first one is to create a new context dependent menu button as described in the next section. The other, more intuitive option is to add an *element edition*. Using an element edition we can not know what the user will type into the label to edit meaning that we can not rely on a simple service call as we do, when changing a nodes appearance shown later in section 6.4.3.4. Therefore, we create an external java extension that acts according to the user's input, when the label is edited. A direct edit label action is also shown in the viewpoint specification in figure 6.10. For the external java action to work properly, we also need to transfer the user's input as parameter to the external java action. The java action basically retrieves the correct entity state for our control center and checks the argument whether it contains any forms of *true* or *false*. Then, we check if there is a difference between the input value and the current status, and if so, change the status and refresh the diagram. Refreshing the diagram programmatically works similar to retrieving the current session in Sirius. After the current editor is retrieved the representation, meaning the diagram itself, is retrieved, which can be refreshed.

6.4.3.4 Adding a New Button to a Context Dependent Menu

Changing the appearance of a node directly inside the diagram can be achieved in multiple ways. We could add a toolbar button setting the power for all power grid nodes at once. Since we already implemented a toolbar button for loading the input model and this would work in the same way, we rather choose a different extension type. Therefore, we

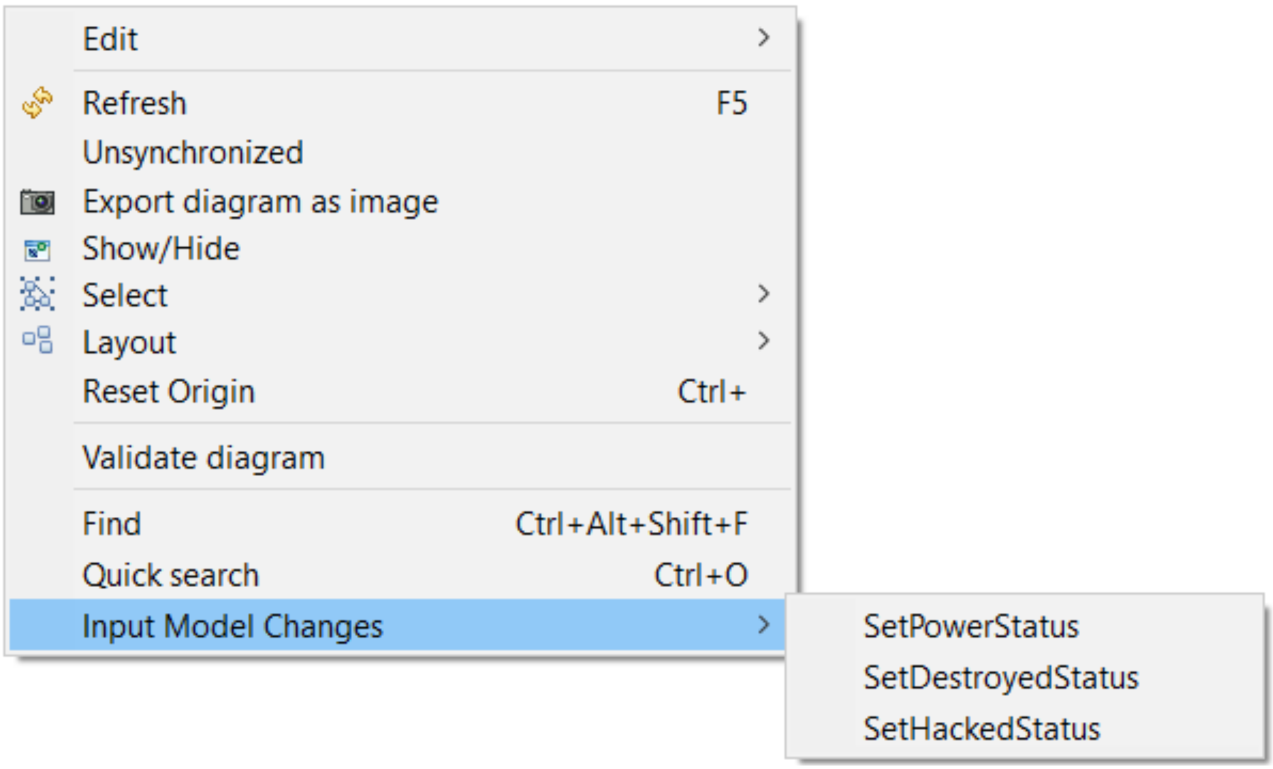


Figure 6.13: Three new context menu buttons appearing if the input layer is selected

add a new button to a context dependent menu for each appearance we want to change. Since Sirius doesn't provide support for creating own mouse-over buttons, we choose to add new buttons to the context pop-up menu. Before adding the actions itself, we first create a new pop-up menu entry for the standard context menu. This is shown in the viewpoint specification in figure 6.10 and is called *Input Model Changes*. The next step is to add actions to this menu. In the figure only the *SetPowerStatus* action is visible to its full extend, but the other actions are implemented in the same way. When starting the action, we simply use a set action to set the power status. The set action itself calls a method from our service class, which changes the current state of the power grid node. Applying these steps also for the other two states results in the menu shown in figure 6.13. There is, of course, the possibility to add a filter to these buttons making them only visible, when the correct element is selected. However, since the methods in the service class explicitly require a *PowerGridNode* for changing the power status and a *NetworkNode* for changing the destroyed or hacked status these methods are only actually called, when one of these elements is selected in advance. For the sake of completeness, listing 6.13 shows the method for changing the current power status.

Listing 6.13: Set power outage method in the java service class

```
public void setPowerOutage(PowerGridNode node) {
  PowerState state = getCorrectPowerState(node);
  state.setPowerOutage(!state.isPowerOutage());
```

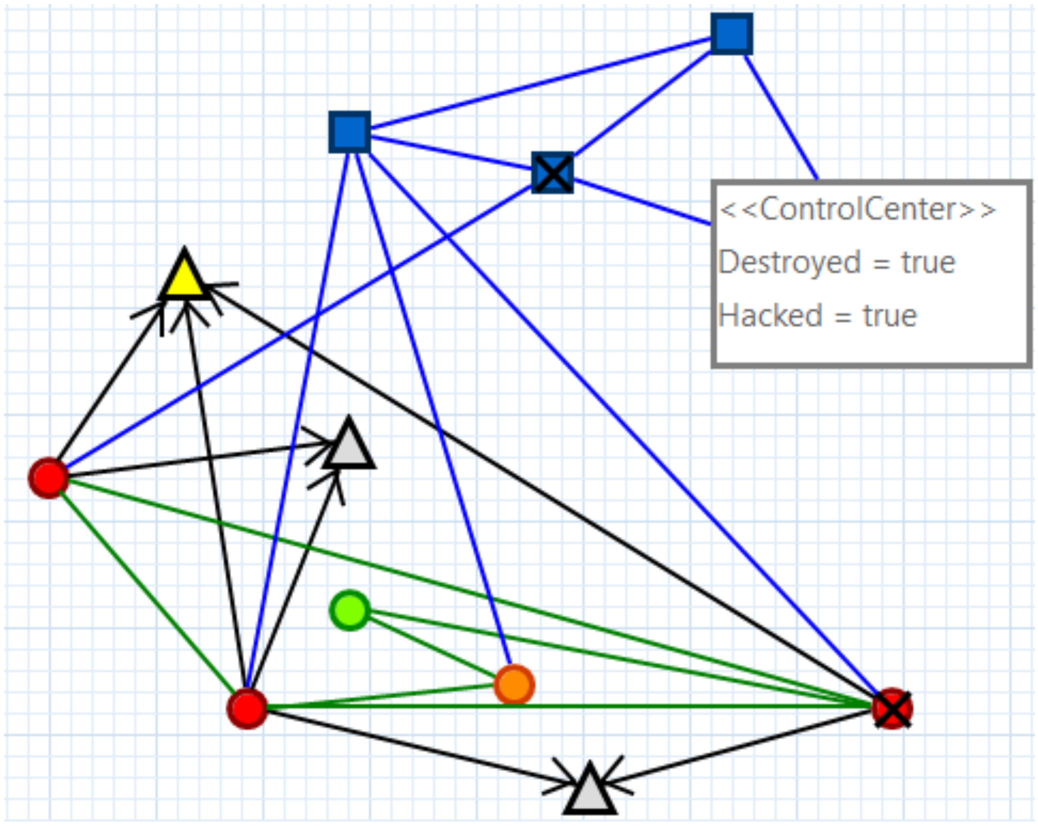


Figure 6.14: The input model diagram representation in Graphiti for our running example

```
    refreshDiagram();
}
```

At first, the method retrieves the corresponding power state to the given power grid node. Afterward, the state is changed to its opposite and last, the diagram is refreshed immediately.

6.4.4 Implementing the Input Model with Graphiti

Now that the implementation in Sirius is covered, we focus on the Graphiti implementation. As there is no such thing as a diagram extension, we only need to add a toolbar button to the Eclipse toolbar providing the functionality described in 6.11. Based on this functionality, any representation can be altered, which is described within the next two subsections. As the Graphiti implementation of the input model extension is mainly based on previous work, the appearance of some entities is different than in the Sirius implementation. Furthermore, the hacked status is left out for the entities and first introduced within the next extension. The result of the input model extension in Graphiti is presented by figure 6.14. The figure shows again the running example introduced in the section before. Throughout the next subsections, we will refer to each kind of entity considered in the extension at the appropriate point in time starting with changing the appearance of nodes.

6.4.4.1 Changing the Appearance of a Node

During runtime in Sirius every notation element in the extension gets evaluated and executed as soon as the specific layer gets active. As there are no layers in Graphiti, the model representation should be active as soon as the input model is loaded. Therefore, we need to apply all changes directly after the model is loaded. This may lead to rather complex code. If the changes should only be represented, then there is no need in creating a new feature, as we can access each shape directly and change its appearance. In the following, we analyze the *destroyed* status for network entities, which is exemplarily shown in figure 6.14 for the smart meter bottom right. The destroyed and hacked status for the control center is considered in the next section.
As we already loaded the input model, we have access to all its entity states and therefore access to every destroyed state. Assuming the destroyed state doesn't refer to a control center, we want to represent that state as two lines crossing each other over the respective network entity. Therefore, we provide a method during the loading of the input model accomplishing that task. The method is given in listing 6.14.

Listing 6.14: Drawing the destroyed status for network entities

```
 1  public void drawDestroyed(final ContainerShape containerShape) {
 2    IPeCreateService peCreateService = Graphiti.getPeCreateService();
 3    IGaService gaService = Graphiti.getGaService();
 4    // create lines
 5    Shape firstLine = peCreateService.createShape(containerShape, false);
 6    Polygon pFirst = gaService.createPolygon(firstLine, new int[] { 0, 20, 20, 0 });
 7    pFirst.setForeground(this.manageColor(ConstantProvider.FOREGROUND_BLACK));
 8    pFirst.setLineWidth(ConstantProvider.shapeLineWidth);
 9    Shape secondLine = peCreateService.createShape(containerShape, false);
10    Polygon pSecond = gaService.createPolygon(secondLine, new int[] { 0, 0, 20, 20 });
11    pSecond.setForeground(this.manageColor(ConstantProvider.FOREGROUND_BLACK));
12    pSecond.setLineWidth(ConstantProvider.shapeLineWidth);
13  }
```

After receiving the graphics algorithm service and the pictogram element service in the first two lines, we can start adding two crossing lines to the given shape representing a network entity. Both shapes *firstLine* and *secondLine* are represented as children of the given container shape meaning that their location in x and y coordinates is limited to the area of the container shape. The polygons created are used to form each line, its color and line width. The integer arrays transferred in lines five and nine contain information on each edge of the polygon. In line five, for example, the first edge starts at x=0 and y=20, while the second edge of the same polygon is placed at x=20 and y=0.
When removing the scenario state again from the current diagram, we can simply remove all children for each network entity. That is due to the fact that the only possible children are currently these two crossing lines indicating the destroyed status. An exception is the control center, which is discussed next.

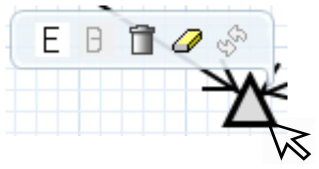


Figure 6.15: Two new mouse-over buttons for setting the power and destroyed status

6.4.4.2 Adding New Annotations to the Control Center

Adding an annotation to a container works similar to changing the appearance of a node in Graphiti. As all of our patterns representing entities are container shapes, we can add as many shapes as children as we want. Instead of lines, we add a so called *MultiText* to the control center for each possible status. The control center can thereby also be seen in figure 6.14 with its two additional states *hacked* and *destroyed*. In order to change the status of either network entity, we added mouse-over buttons that are presented within the next section.

6.4.4.3 Adding a Button to a Context Dependent Menu

As we not only want to represent a power or entity state, but also want to be able to change it, we implemented a context dependent menu button for possible state changes like we did for the Sirius prototype. The difference in this prototype is that we implement these buttons for the mouse-over context menu. This is possible, since the topology editor already offers an extension point for adding new buttons to this context menu. If our Sirius editor would also offer such a self-defined extension point, we could have implemented such a mouse-over button there as well. However, a custom *EditPart* is needed for such a behavior, which was only implemented for the Graphiti-based smart grid editor in the previous work.
As mentioned in the sections before, as long as we only want to represent the extension but not make any changes, we do not need a new feature. However, since we want to change the status directly in the editor, we need to provide a new *CustomFeature* for each possible state. Figure 6.15 shows the result of two new mouse-over buttons. The left one resembles the power state, while its neighbor represents changing the destroyed status of a node. The implementation for the hacked status is equivalent and is not considered here. We only consider here the *power state* feature. Other features can be implemented similarly. As we want the power enabled button to appear as mouse-over button, we need to implement the given *smartgridsecurity.graphiti.extension.contextbutton* extension point. The implementing contributor only adds all features that should appear as button to a list and returns it. The evaluation is done by the topology editor, which adds the new buttons to the existing ones Graphiti provides, such as the remove or delete button. In the following, the *PowerEnabledFeature*, which is able to change the power state of a given power grid node is considered.
The class itself extends *org.eclipse.graphiti.features.custom.AbstractCustomFeature*. The two important methods that need to be overridden are the execute method and *getImageID*. The latter returns a path to an icon, which is used as button in the mouse-over menu.

The execute method itself changes the state of the current power grid node and alters its appearance. Due to the evaluation of the extension point the execute method is always called whenever the button is pressed leading to a change of the current state.

6.4.4.4 Removing the Content of an Input Model

As already mentioned, Graphiti doesn't offer a layer functionality as Sirius does. Therefore, after loading another model all its features stay active as long as the diagram exists, whereas we only need to disable a certain layer in Sirius to only show the original diagram. An implementation of another toolbar button is required clearing all the changes made because of loading the input model. The *clear* button itself can be implemented the same way as the *load* button. However, instead of adding shapes or changing colors, pressing the clear button should return the original state of the diagram. In the following, we shortly discuss this process for all network entities, the power grid nodes and the scenario state itself.
As power grid nodes only gain another color, if their power is out, we need to draw the original *yellow* for every power grid node. As the input model extension has a dependency to the topology model the original color is known.
Smart meters or other entities may have two lines crossing indicating their defect in the input model. As these lines are implemented as children of the original shape we can easily remove all children from the shape restoring the original entity. This, of course, is only valid as long as we are sure that the input model extension is the only extension. Otherwise, we need to make sure that only the two crossing lines are removed. An exception for that is the control center, since in the original editor it already has a *MultiText* as child shape. Therefore, the code in listing 6.15 removes exactly the two *MultiTexts* the input model creates.

Listing 6.15: Removing both texts in the control center

```
1 for (GraphicsAlgorithm g : shape.getGraphicsAlgorithm().getGraphicsAlgorithmChildren()) {
2   if (g instanceof MultiText && ((MultiText) g).getValue().contains(value)) {
3     shape.getGraphicsAlgorithm().getGraphicsAlgorithmChildren().remove(g);
4     break;
5   }
6 }
```

The parameter *value* in line two thereby equals either *Hacked* or *Destroyed* depending on which text should be deleted.
The last action that needs to be done when clearing the input model, is the removal of the scenario state that was loaded to the diagram. After that the diagram is again reverted to its original state.

6.5 The Output Model

In this section we analyze and implement the second extension of the smart grid topology meta-model. The output model extension is thereby an extension to the input model

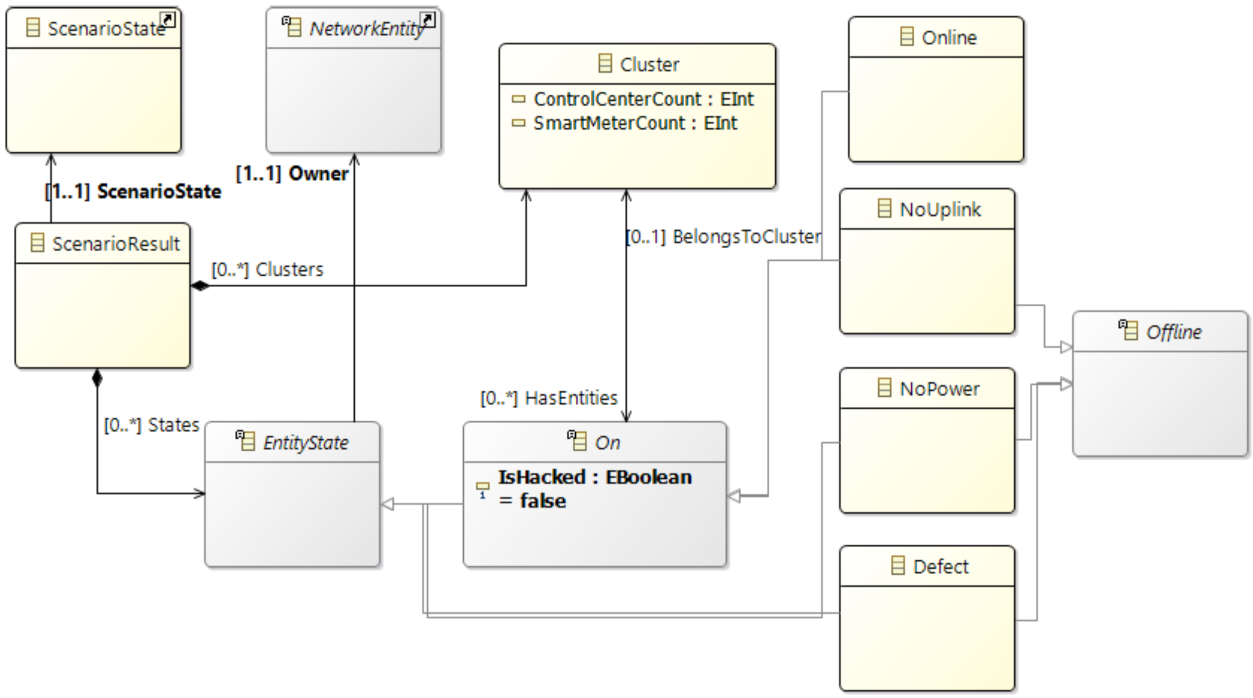


Figure 6.16: The output meta-model extension

extension meaning that the output meta-model knows the input meta-model, as well as the topology meta-model. As we did in the last section, we first analyze the given meta-model and explain its proper use. At the same time, we analyze the different mappings the output model extension has to offer. After that, we continue with the implementation in both Sirius and Graphiti leaving out or cutting short those parts that are similar to the last extension. The last subsection summarizes and explains different problems and possible solutions, when having more than one extension active at the same time.

6.5.1 The Output Meta-Model

As already mentioned, the output model extension is en extension to the input model extension. That means, in order to create a valid output model instance, we need an existing input model instance. In general, the output model resembles the estimated result given a certain input for the smart grid. In other words, the output model resembles the impact of the input model to the remaining entities. Therefore, an output model is best generated automatically, as a manual creation of an output model might lead to missing or wrongly classified nodes. Furthermore, as the representation should resemble an output resulting from a given input, there is no need in implementing any extension type that can be used for altering values of an attribute. The complete meta-model of this extension is shown in figure 6.16. On the left side we can see the main container element, the *ScenarioResult*. As the output model extension extends the input model, there is a reference from the ScenarioResult to the *ScenarioState* given. Furthermore, the *ScenarioResult* only contains two different meta-classes directly. The first is the abstract *EntityState* meta-class along

with the inheriting meta-classes *On*, *NoPower* and *Defect*. Although, this class is named identical to the meta-class in the input model, they are still different semantically. This abstract class should specifically resemble a concrete output state resulting from a different input state. For example, if a power state has a power outage all entities that are connected to this power state and no other result in having no power. The defect meta-class resembles a state, which was marked earlier as *destroyed*. The abstract meta-class *On* can be further divided into either an *Online* entity, where everything is alright or a *NoUplink* meta-class, where somehow the connection to the control center is lost. This can happen, if an entity is destroyed and only this entity has outgoing connections to the control center. Then all other entities can be considered *NoUplink* entities. Last but not least, the *EntityState* meta-class, like its equivalent in the input model, references exactly one network entity meaning that each entity state is mapped to exactly one existing network entity.
The other meta-class in figure 6.16 directly contained in the *ScenarioResult* class, is the *Cluster* meta-class indicating, which *On* state belongs to which given cluster, whereas a cluster specifically contains a number of smart meters, as well as a number of control centers as attributes.
As mentioned, we leave out some of the extensions, as they are similar to the extensions in the input model. Therefore, we here focus on the *NoUplink* and the *Cluster* meta-class. As an entity state can only resemble one concrete state at a time and given the developers intent to add further states to the model, we can infer that the *NoUplink* resembles again an attribute extension type. If extended noninvasively, each network entity would probably receive a new attribute with *EntityState* as type. Therefore, we can infer the same mapping as we did within the previous extension.
Considering the cluster class, we can further differentiate. If the cluster was intended as an actual meta-class, we could represent each instance as container surrounding its containing network entities. However, if, for the developer, the relation between the cluster and the *On* meta-class is the important aspect, we could consider this a relation extension type. If only the relation extension type is relevant, the outline view could be extended, in order to show the membership of each *On* state towards its cluster. Due to lack of time and missing framework capabilities an extension of the outline view is not done and only mentioned here for the sake of completeness. Furthermore, the implementation of further meta-classes is shown within the next extension. Therefore, we only focus on implementing the *NoUplink* meta-class.
Besides the different mapping possibilities the attribute extension type has to offer, we again want to change the appearance of the existing model to represent those kinds of information. As this is almost the same implementation we did in the section earlier, we only focus on the *NoUplink* implementation, since the rest can be implemented the same way.

6.5.2 Implementing the Output Model with Sirius

Since the implementation of the output model is almost identical to the implementation of the input model, we only cover those parts that differ from the input model. As an exemplary implementation, we only regard the *NoUplink* meta-class, as the other meta-classes that extend the *EntityState* meta-class can be implemented the same way. Therefore,

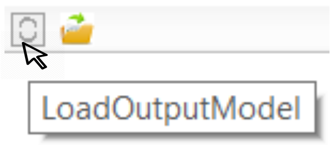


Figure 6.17: The disabled load output model button next to the load input model button

the next two subsections first deal with loading a third model to the given diagram and second, the implementation of a smart meter representation, in case there is no uplink given.

6.5.2.1 Loading a Second Extension Based on the First Extension

In section 6.4.2 we've already shown how adding a second model to the given diagram works for Sirius as well as Graphiti. Since this procedure is identical for every further model we want to add, there is no need in mentioning it here again. Nevertheless, our output model extension is an extension of the input model extension causing us to only enable the load button, if a corresponding input model is already loaded. Alternatively, we could alter the *visible when* parameter for the toolbar button to make the output button only visible, in case an input model is loaded. However, it makes more sense to already show the button assuming the corresponding plug-in exists, but disable it to show the user that further extensions are possible under the condition that an input model is loaded.
When loading a new model with the help of a toolbar button, we basically need to extend the *AbstractHandler* class. As we now have more than one extension we provide another abstract class named *LoadExtensionModel*, which implements the required *execute* method as seen in the Sirius part of section 6.4.2. Since we need to add a required file extension and name for the file chooser dialog, this part is outsourced to an abstract method that each class has to implement. In case of our new *LoadOutputModel* class, the implementation of this method is shown in listing 6.16.

Listing 6.16: Example implementation of the *setFileDialogExtension* method

```
protected void setFileDialogExtension() {
  dialog.setFilterExtensions(new String[] { "*.smartgridoutput" });
  dialog.setFilterNames(new String[] { "Output Model" });
}
```

The first line simply states that only output models can be loaded, while the second line names the given extension.
If we want to disable the toolbar button given the condition that no input model is loaded, we further need to override the `public boolean isEnabled()` method given by the *AbstractHandler* class. The procedure in this method is similar to loading a model. We first gather the current Sirius session, access all semantic resources and check whether there is already a valid input model loaded. If that is the case, we return true and otherwise false. The result can thereby be seen in figure 6.17 showing a disabled load output model button next to the load input model button known from section 6.4.2 However, it is important to

know that this procedure is decrepit, if all the models exist in the same project, as they are automatically added to the current session as soon as the diagram is opened.

6.5.2.2 Implementing the Output Model Diagram Extension

Since we want to use the output model extension whenever a corresponding input model is active, we also need to define a diagram extension like we did for the input model extension. The only difference now is that we don't extend the topology viewpoint and diagram representation directly, but the *smartgridinput* diagram extension. Therefore, the viewpoint URI and the representation name of the diagram extension are adjusted to the specifics of the input model extension. As already discussed in section 6.5.1, we again want to change the appearance of our nodes to show, whether their state is *NoUplink*, *NoPower*, *Online* or *Defect*. For the last two states we wouldn't need to change anything, since an online state can be seen as fully functional as represented by the topology diagram and a defect state resembles a destroyed node, which is already covered by the input model extension. Furthermore, we only focus on the *NoUplink* state here, since the implementation for the second state can be considered equal.
Since the output model resembles an output state for the given input state, the output model receives a higher priority than the input state. Therefore, we need to make sure that the entity state of the output model is shown instead, or at least among the state of the input or topology model. To ensure that we can not import the *SmartMeter* node of the topology diagram, as than the appearance would depend on whether the input layer would be chosen active first or the output layer. Therefore, we import the *SmartMeterImport* node from the input model extension. Doing that results in also importing every conditional style we applied earlier on. When adding a new conditional style, it gets preferred over the other existing conditional styles. Furthermore, using the output layer without the input layer now results in the same representation we would have, if the output layer isn't active at all. This is due to the fact that we imported a node from the input model extension. The result of the *NoUplink* implementation for smart meters can be seen in figure 6.18, where two out of the three smart meters are now marked as *NoUplink*. Since changing the appearance of a node may lead to a conflicting state, for example, if two extensions both want to change the color of a node, we also need to address this problem. This is done in section 6.5.4 and is considered for both the cases that the extensions know and also depend on each other and that both extensions are completely independent of each other.

6.5.3 Implementing the Output Model with Graphiti

Now that the implementation of the output model extension for the Sirius-based editor is discussed, the next focus lies on the implementation of the Graphiti-based editor. Within this section, we also divide between loading the output model based on the currently active input model and then concentrate on the actual implementation of the extension. The implementation here is based on previous work meaning that all new states are implemented for every network entity except the control center.

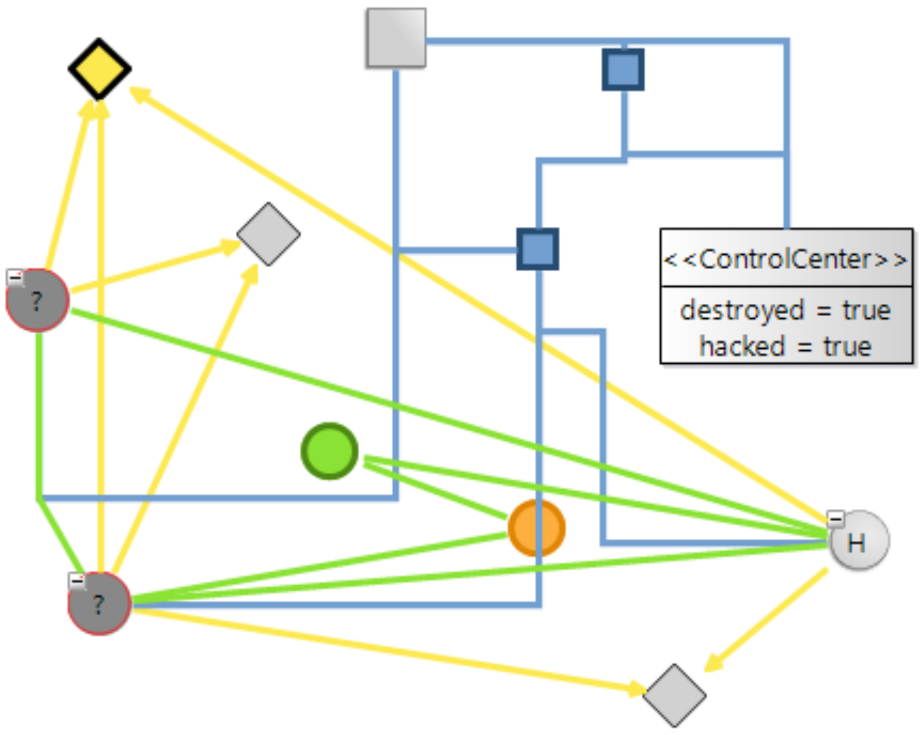


Figure 6.18: Result of the output model extension implementation in Sirius

6.5.3.1 Loading a Second Extension Based on the First Extension

As there are no layers in Graphiti, the otput model extension needs to become active, as soon as a valid output model is loaded. Loading the model is done the same way as described in section 6.4.2. The difference now is that before any dialog opens to choose an output model from, a check is required whether the diagram currently contains an input model or not. If that is the case, the standard procedure is executed including that another *clear* button gets enabled, if the output model was loaded successfully.

6.5.3.2 Implementing the Graphical Representation in Graphiti

As mentioned in the section before, we extend the input model extension only by checking whether an input model is already loaded and by that knowing which changes in the editor are made after the input model is active. Therefore, the appearance of entities can again be changed at will. That is in a way equal to the import mechanism Sirius offers. Figure 6.19 shows the result of the Graphiti implementation. Since this extension is based on previous work, there are more extensions implemented as the *NoUplink* meta-class for smart meters. However, we only analyze the *NoUplink* feature for smart meters, in order to establish a better basis for comparison of the two frameworks. In figure 6.19, the same two smart meters are marked as *NoUplink* than in the Sirius section. Furthermore, the output model in Graphiti also states the hacked status of a node with an exclamation mark, if its hacked and with a question mark, if its only an instance of *NoUplink*. Even more, the color of a *NoUplink* entity is changed to a gray with its border getting the original color of its inside.

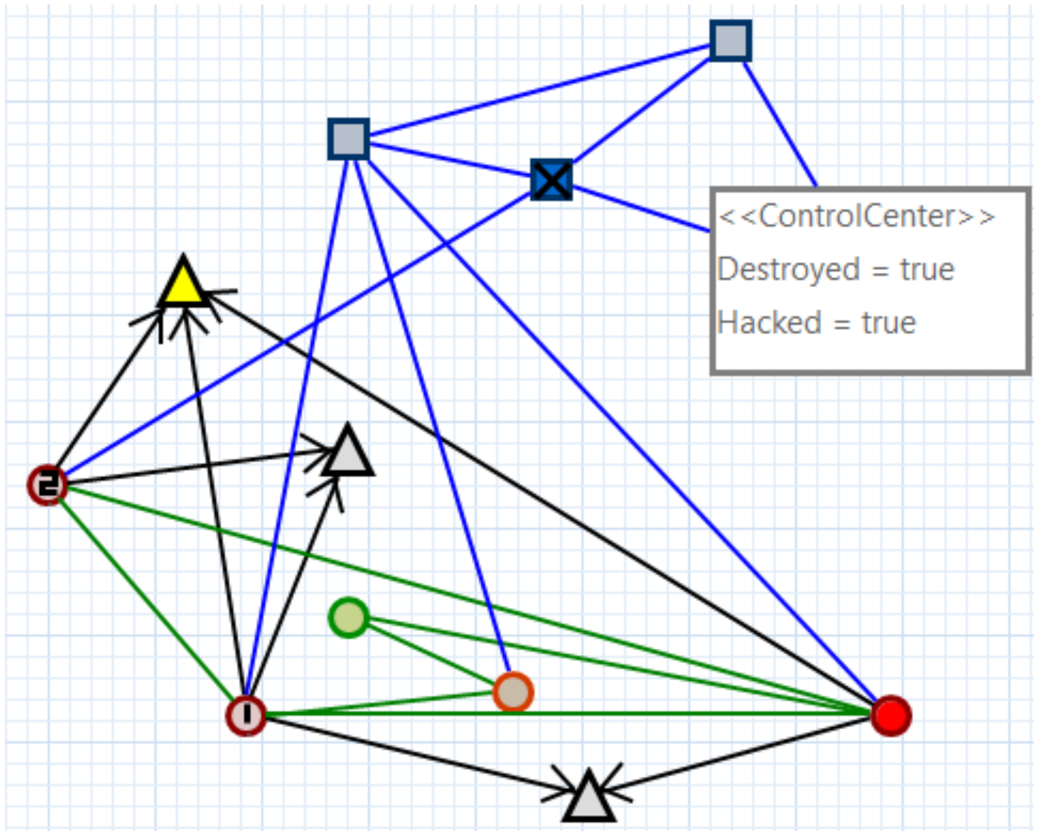


Figure 6.19: The result of the output model extension implementation in Graphiti

Technically, this extension works as a combination of the clear button in the input model and the extension implementation already discussed in the input model regarding only entities that are *NoUplink* instances in the output model. On a more concrete perspective this means that, if an entity is an instance of *NoUplink*, all its children are removed. Then, the color is changed and depending on the hacked status a question or exclamation mark is drawn.
As long as these two extensions are the only extensions and the output model extension depends on the input model extension, this implementation works. However, if another extension is added independently, the implementation gets more complex. This is addressed in the next section for Sirius as well as for Graphiti. Furthermore, a general solution is offered for that problem but not validated.

6.5.4 Problems with Two or More Active Extensions

Adding additional buttons to the Sirius or Eclipse toolbar is no problem proven by the output model extension. Moreover, we can have at least two extensions active at the same time meaning that further extensions are also possible. We already noticed a problem, when two or more extensions are active at the same time. If both extensions change the appearance of an existing node, we have to define rules to prioritize the extensions. In case of the input and output model extension, this is fairly easy, since the output model extension shows the impact of a given input and therefore should always have a higher priority than the input model. Even if we assume that both extensions have the same priority, we could manage to change the appearance accordingly, since the output model extension knows the input model extension.
In Sirius, we can therefore different conditional styles in order to derive the different appearances. If there are more appearances to adjust, we can even use the custom style for nodes for both extensions. In case of the output model extension, we extend the *EditPart* for the node in the input model extension and add the new conditions under which the appearance changes. This works well even if there are conflicting appearances, for example, that one extension requires a node to turn yellow, while the other requires the same node under a different condition to be blue. If both conditions are met, the standard Sirius behavior is to change the color according to the first condition of the first active extension that is true. In case of a custom style, the developer can decide which color must be active.
A similar solution also works for the Graphiti framework. Since we need to use plain Java code, we can access the extension directly checking whether any appearance is changed and act accordingly. Assuming we developed the core and extension editor according to certain quality aspects, this problem is as easy to solve in Graphiti as it is in Sirius.
A problem occurs, if the appearances are conflicting and the extensions don’t know about each other. Then, the possibility to extend a given custom style is not given anymore. Therefore, different rules have to be identified. One way could be to already address this problem at the core editor. We could define an extension point serving the purpose of communicating among all extensions. With that extension point an extension could identify conflicting extensions and could act accordingly assuming all extensions actually implement the given extension point. Of course, if such an extension point has to be

created afterward, when most of the extensions already exist, the effort of creating such a communication among the given extensions is huge. In the following, we describe a possible definition of such an extension point in the core editor.
To establish a communication between possible extensions, we have to define a basis for that communication. In our case an abstract class named *AbstractSmartGridExtension* is appropriate for that task. This abstract class needs to be inherited by each future extension. The abstract class contains on the one hand methods responsible for retrieving certain notation elements and on the other hand abstract methods for each extension type on graphical editor level, where a conflict may occur. Referring to the smart grid resilience framework, we first would need ID's to identify each possible entity class, such as smart meters or power grid nodes. As abstract methods we need simple Boolean methods returning true, if, for example, the color of a power grid node is possibly affected by the extension. After creating such methods, the only methods left to implement are methods that check whether at least one extension changes the color for a certain entity. Therefore, we also need a list of all extensions implementing this extension point. Listing 6.17 shows how this can be done without the core editor knowing its extensions in detail.

Listing 6.17: Example on how to get all active extensions implementing an extension point

```
1 IExtensionRegistry reg = Platform.getExtensionRegistry();
2 IExtensionPoint ep = reg.getExtensionPoint(extensionID);
3 IExtension[] extensions = ep.getExtensions();
4 ArrayList<AbstractSmartGridExtension> contributors = new ArrayList()<
      AbstractSmartGridExtension>;
5 for (IExtension ext : extensions) {
6    IConfigurationElement[] ce = ext.getConfigurationElements();
7    AbstractSmartGridExtension obj = (AbstractSmartGridExtension) ce[0].
         createExecutableExtension("class");
8    contributors.add(obj);
9 }
```

At first, we access our desired extension point in line two. The parameter *extensionID* is thereby the ID of our defined extension point. From there, we can access all available extensions and iterate over all. In line six, we access the configuration elements containing each interface or abstract class the extension point provides. Since we only provide one abstract class and nothing else, we can access that class in line seven. By calling such a method, we can check each entity classes for possible conflicts and act accordingly. As this is only a theoretical idea further validation is necessary.

6.6 Further Extension of the Smart Grid

As we couldn't cover all extension types on graphical editor level with the previous two extensions, there is a need in creating a further *artificial* extension. With the last two extensions we already covered the meta-class, attribute and relation extension type on meta-model level. Furthermore, we used mainly unidirectional associations as extension mechanisms. That leaves the containment extension type and inheritance, composition and stereotyping as extension mechanisms. On graphical editor level, we already covered

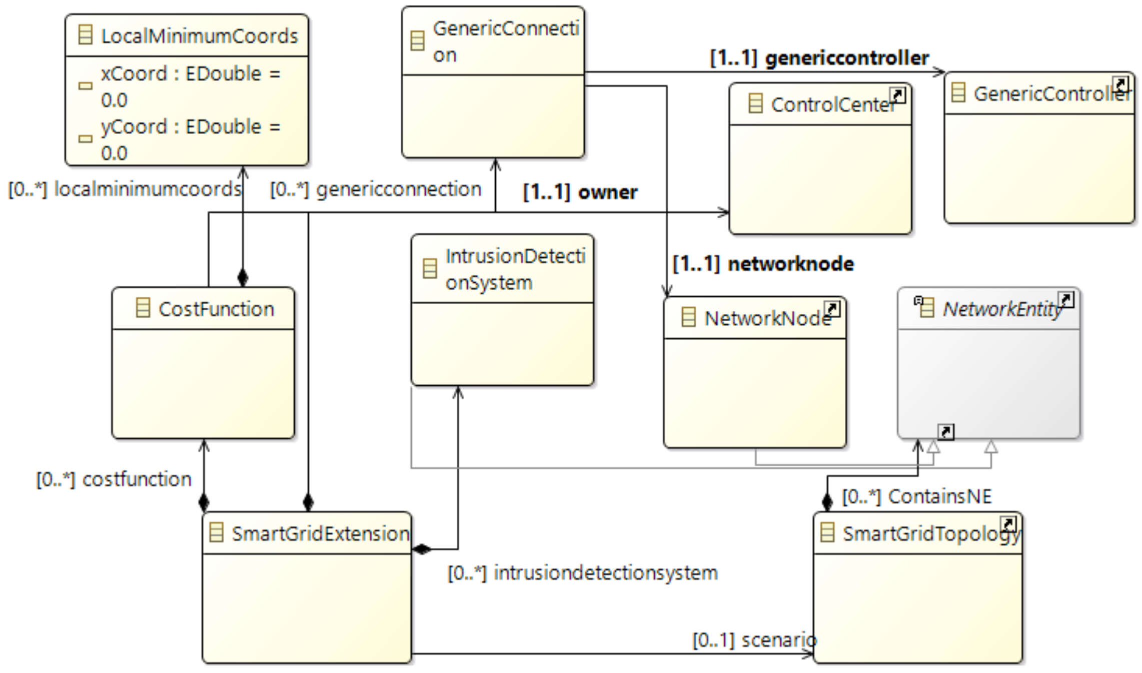


Figure 6.20: The Meta-Model for the Artificial Extension

annotations, a change of appearance as representative extension types. Toolbar buttons and context dependent menu buttons as extension types were used for creation or altering values. For the representation, there are still notation elements left including nodes/containers as well as connections and furthermore compartments, sub-nodes and an extension of the outline view. When creating or changing notation elements we need to implement a properties entry, a palette entry and a new view. Therefore, we create an extension that could possibly be implemented in the smart grid context.
The next subsection therefore deals with the meta-model of this extension. Section 6.6.2 covers the mapping of each meta-model element, as there are more different extension types than in the extensions before. After that, we regard the implementation of a new stereotype, as this is independent of the editor framework used. The last two subsections within this section then deal with the implementation in first Sirius and then in Graphiti.

6.6.1 The Artificial Extension Meta-Model

Like in the previous sections analyzing the different extensions, we first provide the meta-model considering the extension. The meta-model for our artificial extension is given in figure 6.20. This time the meta-classes contained in the topology meta-model are represented on the right side of the diagram, while the new meta-classes of the extension are shown on the left side. We, of course, again use a container element referencing the *SmartGridTopology*. Based on the *SmartGridExtension* container meta-class we have overall three different outgoing compositions. The first one on the left side is the *CostFunction*. If we find a (local) minimum of the cost function, the control center works efficient. Therefore, we need an unidirectional association to the control center on the one hand and on the

other hand another meta-class resembling the local minimums as a new meta-class. The *LocalMinimumCoords* thereby contains two attributes indicating the x and y coordinate of the local minimum.
The second outgoing composition we need to consider is the *IntrusionDetectionSystem* meta-class. As the name states this meta-class simply resembles an intrusion detection system inheriting the *NetworkEntity* meta-class of our topology meta-model.
The last meta-class we need to consider in this meta-model is the *GenericConnection*. This class should resemble a special connection between generic controllers and network nodes.
Since we want to cover all extension mechanisms and types that weren't covered in the extensions before, we also have to consider stereotyping. As EMF doesn't support stereotyping within its diagrams, we need to configure a new diagram containing our stereotype. For this extension we choose to add an emergency supply to the control center that states how long a control center lasts, when all connected power grid nodes suffer a power outage. The implementation of this stereotype is shown in section 6.6.3.

6.6.2 Mapping of the Individual Extension Types

While the previous section covered the single meta-classes that are added during this extension, this section deals with the mapping of each meta-class to possible graphical editor extension types. Considering the *IntrusionDetectionSystem* meta-class, we can see that this meta-class is in fact a direct sub-class of *NetworkEntity*. Since this meta-class is directly connected to the topology meta-model with an extension mechanism resulting in a meta-class extension type, we can not map an intrusion detection system to an existing graphical element. Therefore, we need to handle this meta-class in the same way as we did with the other network entities contained in the core editor and simply create a new node. The creation can also be handled the same way as for the other entities meaning that a new palette entry is added. Additionally, a context dependent menu button could be added. Other possible implementations can be inferred from section 5.1.
If we take a look at the *GenericConnection* meta-class, we have multiple options. As we can see, there are two unidirectional associations aiming towards the generic controller and the network node meta-class. We hereby assume that this meta-class would be added to the core meta-model in a similar way as it is presented here. The only difference would be that there would be two bidirectional associations, instead of unidirectional ones. Therefore, the *GenericConnection* realizes the second meta-class extension type, as it is part of either the network node or the generic controller class. The associations on the other hand, realize the relation extension type. The meta-class as well as the relations can be mapped to a new connection between generic controllers and network nodes. Although, we decided to implement this extension type in such a way, there are other possibilities. We could, for example, add a properties entry for each generic controller and network node indicating to which entity they are connected by a generic connection. Furthermore, we could use a bordered node to move the properties entry to the diagram. If the generic controller and network node were implemented as containers, we could also use a sub-node representing a single generic connection. Otherwise, a compartment containing all generic connections with their target would also be possible. However, as bordered nodes should mostly be

used for annotations and a class intended as connection wouldn't necessarily count as an annotation, we should refrain from adding a border node in this case. For the creation of such a connection we can simply add a palette entry or a context dependent menu button. If the connection should only be resembled in the properties view, than a connection could also be created in this view.
The last mapping of meta-classes visible in figure 6.20, is the cost function and its local minimum coordinates. The cost function itself relates to the control center. Through this association information is added to the control center the same way as in the input and output model extension. However, this time there are no Boolean attributes involved meaning that a change of the appearance of the control center or adding of just an annotation doesn't resolve this mapping. Furthermore, the *LocalMinimumCoordinates* meta-class is also connected to the cost function meta-class and needs to be considered as well. Analyzing these meta-classes and their connections step by step we come up with the following extension types for meta-models and their realizations on graphical editor level. The composition from the cost function to the coordinates results in the coordinates being a containment inside the cost function. Therefore, we can organize the coordinates as sub-nodes of the cost function. For the representation of the cost function itself we again assume that we create an extension based on the topology editor. If we created a new editor for this extension, the cost function itself would definitely be its own container. However, as an extension to the topology editor, the cost function needs to relate to the control center. If added intrusively, the cost function itself would be realized as containment extension type. Therefore, we could also create a container for the cost function and add a connection to the corresponding control center realizing the relation as connection and the cost function as notation element. Nevertheless, we can also directly map the cost function to the control center, as there is always exactly one control center corresponding to a cost function. Therefore, these two meta-classes in the extension map to a compartment inside the control center and sub-nodes resembling the coordinates. Adding and changing can here also be done by a palette entry and the properties view.
As we also cover stereotypes within this extension, we need to figure out the mapping of the power supply stereotype. As we stated in section 4.4.4 the stereotype extension mechanism realizes the attribute extension type on meta-model level. As the attribute we want to add is an integer, we need to refrain from changing the appearance but can simply add an annotation or a properties entry. The next section thereby deals with the implementation of such a stereotype.

6.6.3 Implementation of a MDSD Profile

MDSD profiles provide a non-invasive mechanism for extending a given meta-model. The approach is described by Kramer et al [32]. The definition and application of such a profile is mostly independent on the framework used. The framework specific parts are therefore only described shortly in two paragraphs. For our smart grid extension we provide a stereotype *EmergencySupply* within a new profile *EmergencySupplyProfile*. The profile itself is shown in figure 6.21. For simplicity reasons, the stereotype can only be applied to the control center. The power supply attribute should state how much time the control center can stay online after all its energy supply is shut down. In order to apply

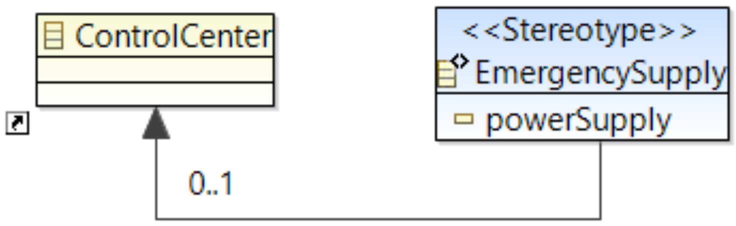


Figure 6.21: The profile used for the smart grid extension

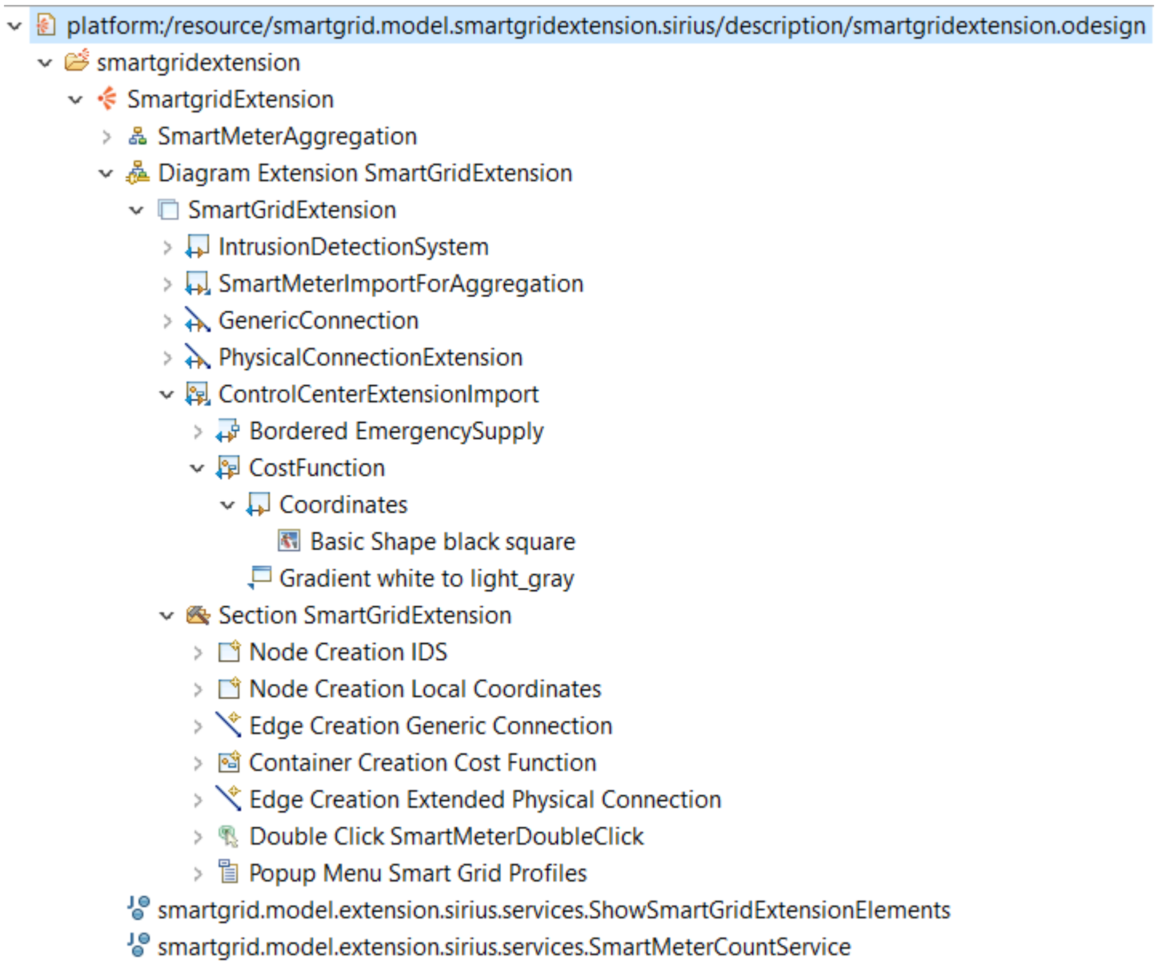
this stereotype non-invasively we need to use the MDSD profiles API. First, we need to apply the profile in general to a smart grid topology model and after that we can apply the stereotype to any control center. The appliance of a profile and stereotype is shown in listing 6.18.

Listing 6.18: Applying a profile and stereotype with MDSD Profiles

```java
IFile f1 = ResourcesPlugin.getWorkspace().getRoot().getProject(PROJECT_NAME).getFile(
    PROFILE_LOCATION);
ResourceSet set = new ResourceSetImpl();
Resource r = set.createResource(URI.createFileURI(f1.getFullPath().toString()));
try {
  r.load(null);
} catch (IOException e) {
  e.printStackTrace();
}

Profile profile = (Profile) r.getContents().get(0);
EObject currentSelection = selection.iterator().next();
ProfileAPI.applyProfile(currentSelection.eContainer().eResource(), profile);
Stereotype st = profile.getStereotypes().get(0);
StereotypeAPI.applyStereotype(currentSelection, st);
```

For this kind of appliance one requirement is that the project and path to the profile is known. An alternative would be to open a dialog window, where the user can choose the profile. After loading the profile's resource, we make use of the *ProfileAPI* and *StereotypeAPI* given by the MDSD profiles. Before applying a stereotype, we need to apply the profile first. The profile is applied to the complete model instance, whereas a specific stereotype contained in the profile is only applied to a specific model element. According to the profile definition in figure 6.21, the profile contains only one stereotype. This stereotype is applied to the currently selected element in the last line of the listing. For the sake of completeness, there should exist another method checking whether the current selection really is a control center to which the stereotype can be applied.
In order to add the stereotype as representation to the smart grid extension, we can add a context menu button calling an external java action performing the code in listing 6.18. Furthermore, a java service method is needed returning a list of control centers with applied stereotype. Whether the given stereotype is applied or not can also be checked with the given API. Figure 6.23 shows the control center with applied stereotype. We chose to simply add an annotation to it whenever the stereotype is applied.



```
platform:/resource/smartgrid.model.smartgridextension.sirius/description/smartgridextension.odesign
  smartgridextension
    SmartgridExtension
      SmartMeterAggregation
      Diagram Extension SmartGridExtension
        SmartGridExtension
          IntrusionDetectionSystem
          SmartMeterImportForAggregation
          GenericConnection
          PhysicalConnectionExtension
          ControlCenterExtensionImport
            Bordered EmergencySupply
            CostFunction
              Coordinates
                Basic Shape black square
              Gradient white to light_gray
          Section SmartGridExtension
            Node Creation IDS
            Node Creation Local Coordinates
            Edge Creation Generic Connection
            Container Creation Cost Function
            Edge Creation Extended Physical Connection
            Double Click SmartMeterDoubleClick
            Popup Menu Smart Grid Profiles
      smartgrid.model.extension.sirius.services.ShowSmartGridExtensionElements
      smartgrid.model.extension.sirius.services.SmartMeterCountService
```

Figure 6.22: The odesign of the artificial smart grid extension

A Graphiti-based solution can be implemented analogous, as we can also receive the current selection and act accordingly.

6.6.4 Sirius Implementation

This section covers the implementation of the artificial extension with the Sirius framework. We thereby do not only implement the mapping discussed in section 6.6.2 but also implement a new view. This view wasn't mentioned earlier, since it is not an extension caused by the meta-model in this section. Moreover, we add a bordered node to the smart meter indicating its aggregation and on double click a new editor should open showing all smart meters contained in the aggregation. This could be used for further extensions in case the smart meter is further extended by containments or attributes. Figure 6.22 shows the description file for the artificial extension including the diagram extension as well as the diagram description for the smart meter aggregation. As we can see, we here also use a java extension in order to represent the elements of our artificial extension. Here the java extension is named *ShowSmartGridExtensionElements*. The other java extension

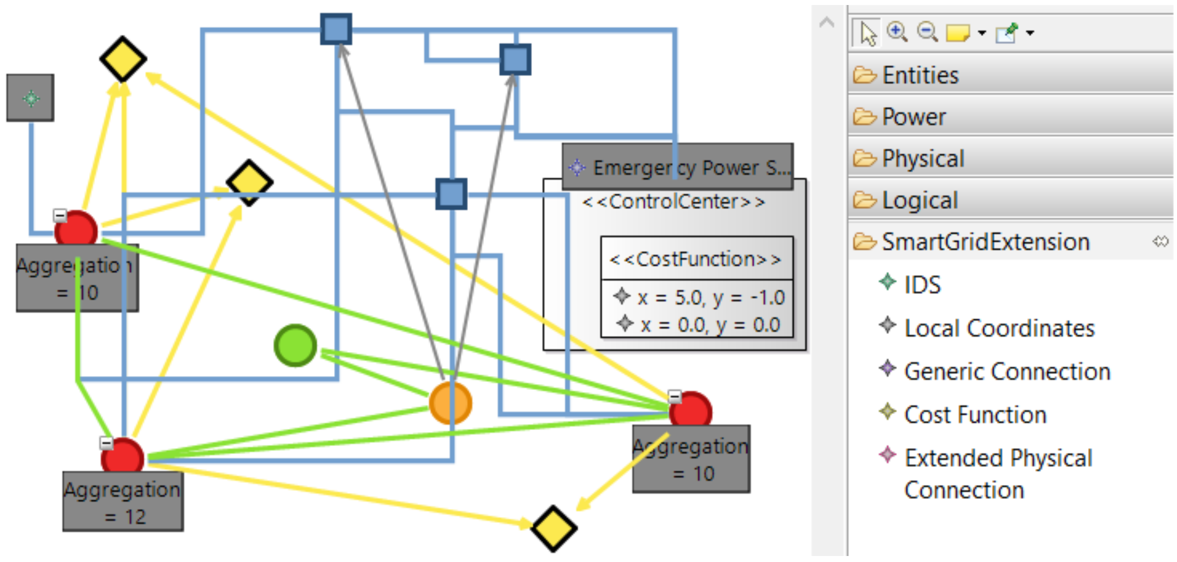


Figure 6.23: Result of the active smart grid extension layer

is responsible for the *SmartMeterAggregation* diagram description and is discussed in subsection 6.6.4.4.
The result of the active smart grid extension layer can be seen in figure 6.23. During each subsection we explain the different extensions based on our running example. Since we already discussed the diagram extension mechanism at length, we only go over the actual implementation of the elements discussed in section 6.6.2 starting with the intrusion detection system and the generic connection as new notation elements.

6.6.4.1 Adding a Notation Element

Within this subsection we discuss the adding of new notation elements. Therefore, we start with adding the intrusion detection system as a new node and continue with adding the generic connection as new connection between generic controller an network nodes.

Adding a Node As we already discussed the *IntrusionDetectionSystem* meta-class is a meta-class below root node extension type, as it extends the abstract meta-class *NetworkEntity* directly. Using the same semantic candidates expression as for all other network entities does not work here. That would only work if either the smart grid topology meta-model knows the smart grid extension or, if the smart grid extension root container extends the *SmartGridTopology* container. Therefore, we here again need a service method to receive a list of all intrusion detection systems available in the extension. This service method, however, only needs to receive the smart grid extension container and from there get all intrusion detection systems available.
After successfully representing the intrusion detection system, we also need to add another palette entry so that these intrusion detection systems can also be added to the diagram, if they didn't exist before. Therefore, we added a new node creation in our *SmartGridExtension* section. The creation of the node itself can either be done by changing

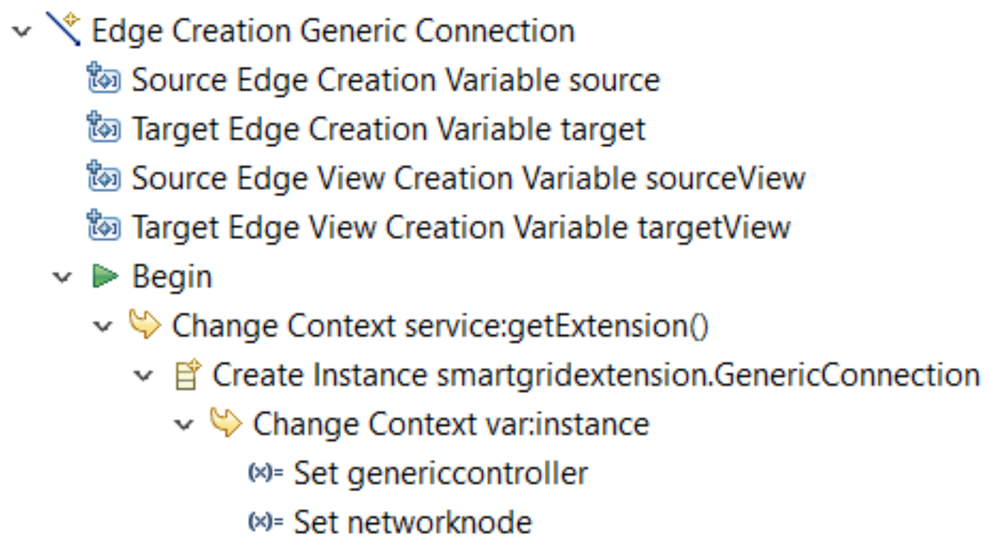
the context to the smart grid extension with a java service call and from there create a new intrusion detection system. Otherwise, we need to add a new external java action dealing with the creation. When creating a new external java action, we need to implement the *IExternalJavaAction* interface. This interface includes a *canExecute* method, which in our case always returns true, as there are no restrictions in creating an intrusion detection system. The second method *execute* is shown in listing 6.19.

Listing 6.19: execute-method for the external java action to create a new intrusion detection system

```java
1 public void execute(Collection<? extends EObject> arg0, Map<String, Object> arg1) {
2   List<? extends EObject> list = (List<? extends EObject>) arg0;
3   SmartGridTopology topo = (SmartGridTopology) list.get(0);
4   SmartGridExtension ext = ExtensionModelHelper.getAndCheckSmartGridExtension(topo);
5   IntrusionDetectionSystem ids = SmartgridextensionFactory.eINSTANCE.
        createIntrusionDetectionSystem();
6   ext.getIntrusiondetectionsystem().add(ids);
7 }
```

The first argument contains thereby our smart grid topology container element, while the second argument contains further self defined parameters, if there are any. The sequence of this method is fairly simple. First, we get the current topology container, then the smart grid extension container currently used. After creating a new intrusion detection system in line five, we add this element to the list of all intrusion detection systems within our container element.
The intrusion detection system can be seen in figure 6.23 as gray square in the diagram top left and in the palette view the creation of such a system is called *IDS*.
Although, adding a new node or container can be done almost straightforward another problem occurs. As we now have a new network entity we would assume that it is possible to also connect these intrusion detection systems with other network entities with the help of a physical connection. According to the topology meta-model in section 6.2, a physical connection connects two network entities and therefore this should work. However, since we need to use node and container mappings, when representing a new connection, only the entities covered in the topology model can be used as source and target for the physical connection. Furthermore, importing the physical connection from the topology editor does also not solve the problem, as we then are only able to change the style of the connection but not the mapping for this extension. The only current way to solve this problem is to add a new element-based edge resembling the physical connection and add the intrusion detection system as mapping, beside every other entity. After that, we also need a new edge creation for the palette in order to create new physical connections that use the intrusion detection system as source or target.

Adding a Connection Besides adding an intrusion detection system as node, we also want to add the generic connection as new element-based edge to the diagram. The procedure for the representation is quite similar to the representation of, for example, the physical connection. Since the element-based edge is in Sirius based on the corresponding domain class, we can add *[genericcontroller/]* and *[networknode/]* as source, respectively target



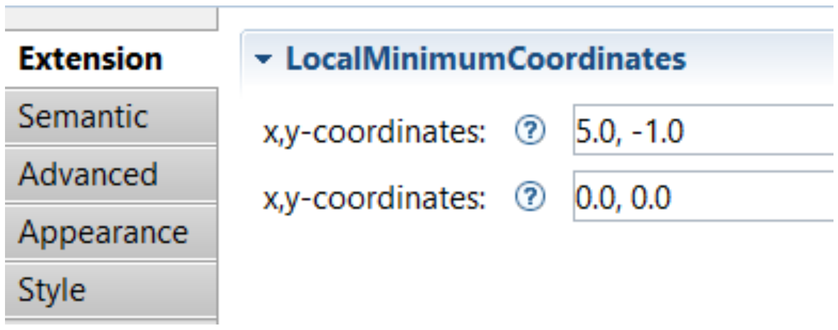
```
Edge Creation Generic Connection
    Source Edge Creation Variable source
    Target Edge Creation Variable target
    Source Edge View Creation Variable sourceView
    Target Edge View Creation Variable targetView
  Begin
    Change Context service:getExtension()
      Create Instance smartgridextension.GenericConnection
        Change Context var:instance
          Set genericcontroller
          Set networknode
```

Figure 6.24: The edge creation for the generic connection

finder expression. Those two expressions are based on the roles in our artificial extension meta-model. We only need our java service for the semantic candidates expression, which returns all generic connections for our smart grid extension model instance.
For the creation of such a connection inside the diagram we have also the possibility to either use an external java action, as we did for the intrusion detection system, or we can use the given basic Sirius actions. We chose to do the latter. Figure 6.24 shows the creation of the generic connection in our diagram extension. We first change the current context to the *SmartGridExtension* root container. From there, we first create a new generic connection instance named *instance*. We now have automatically a new variable to switch the context to and can set the generic controller and network node accordingly.
Like the intrusion detection system, the connection can also be seen in figure 6.23 in the palette view as third item and in the diagram. In the diagram we have two generic connections shown as gray arrows going from the generic controller to two of the available network nodes.

6.6.4.2 Adding a Compartment

After we created a new node as well as a new connection, we now need to discuss the creation of a new compartment and its sub-nodes inside the control center. Like in the input model extension, we first import the control center from our topology editor. Then, we can add the cost function as a new container inside the control center. Since only the local minimum coordinates are added to that container, the cost function container resembles a compartment. In order to receive the correct cost function, we again need our service class. The method *getCostFunction(ControlCenter cc)* provided by the service class simply gets all cost functions available for the current smart grid extension and then checks whether the owner of the cost function has the same ID as the given control center. If so, that cost function is returned. For representing the coordinates we don't have to use the service class, as our container element is already the cost function. Therefore, we can



LocalMinimumCoordinates

x,y-coordinates: 5.0, -1.0

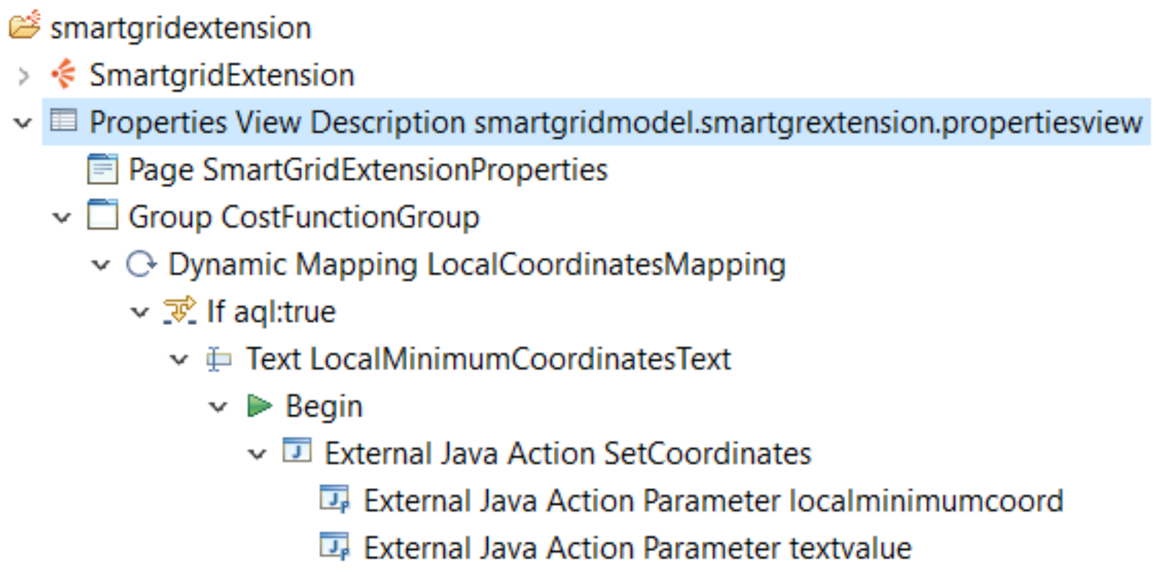
x,y-coordinates: 0.0, 0.0

Figure 6.25: Screenshot of the extended properties view

simply enter the meta-model role of the composition from the cost function meta-class to the *LocalMinimumCoordinates* meta-class. These coordinates should be represented as annotations inside the compartment.
Since we also want to create palette entries for both the cost function and the coordinates, we have several options. Besides the possibilities to either use the basic Sirius actions or an external java action, we can also vary the content of the Sirius actions. We can for example choose to only add a palette entry for the coordinates and add a conditional course of action, if the cost function doesn't exist for the current control center. Nevertheless, we can still add both the coordinates and the cost function to the palette view. The coordinates can be added like any other basic entity by changing the context to its container and then adding a new instance. This works, since the container of the coordinates is already placed in the context of our extension. When adding the cost function on the other hand, we first need to switch the context to our extension by calling the *getExtension()* method of our service class.
Like the two other extensions, both the cost function and the coordinates can be found in figure 6.23 in the palette view, as well as in the control center in the diagram. A side effect of mapping meta-model elements to new notation elements is that for these new elements the properties view is updated automatically and needn't be done as its own extension. Therefore, altering the values of the coordinates can also be done in the properties view, when clicking on one of the coordinate sub-nodes. Of course, we could also add a direct edit label, as we did in the input model extension in section 6.4.3.3.

6.6.4.3 Extending the Properties View

Extending the properties view is one of the new features Sirius 4.0 has to offer. Instead of implementing different extension points to provide an extended properties view, we now can use the mechanics Sirius offers. Within this section, we exemplary show how to add a new tab to the *cost functions* properties view that shows all available local minimum coordinates. The result is shown in figure 6.25. Each *LocalMinimumCoord* instance gets his own label and text field containing the values for the coordinates. If the corresponding meta-class would also have an ID, the label for each instance could be adjusted. In order to show the mechanics behind this result, figure 6.26 is given showing the description of our new properties view. As we can see, the properties view description contains two main



```
smartgridextension
  > SmartgridExtension
  v Properties View Description smartgridmodel.smartgrextension.propertiesview
      Page SmartGridExtensionProperties
    v Group CostFunctionGroup
      v Dynamic Mapping LocalCoordinatesMapping
        v If aql:true
          v Text LocalMinimumCoordinatesText
            v Begin
              v External Java Action SetCoordinates
                  External Java Action Parameter localminimumcoord
                  External Java Action Parameter textvalue
```

Figure 6.26: Screenshot of the properties view description in the odesign file

elements. One being a *page*, while the other one being a *group*. The page corresponds to a new tab in the existing properties view, while the group represents a section in the properties tab[7]. In the page we simply define that the domain class, where this tab should be visible is our *CostFunction* class. In the group we could limit the representation further. However, as we are only interested in all of the cost function's minimum coordinates, we do not need to specify anything further except for the group's ID. In the dynamic mapping that follows we iterate over all available *LocalMinimumCoord* instances the selected cost function has to offer. The iterator used needs to be named. During the iteration this name can be used as variable in order to access the current *LocalMinimumCoord* instance. For each iteration a condition is required for which the following widgets should be created. As we do not need any condition and want to represent all coordinates, we simply add *aql:true* as statement to the condition. The text widget that follows specifies the label *x,y-coordinate* we can see in figure 6.25 and the value of its text field. As the current instance is stored in our iterator variable, we can access the coordinates as follows in listing 6.20, where *localminimumcoords* is the name of our iterator.

Listing 6.20: AQL statement receiving the x-coordinate of the current *LocalMinimumCoord* instance

```
aql:localminimumcoords.xCoord + ', ' + localminimumcoords.yCoord
```

Sirius offers a key listener for each text widget. Therefore, if we want possible editing changes to be applied to the correct instance, we can use an external java action. This external java action receives two parameters. One parameter being the current *LocalMinimumCoords* instance, while the other parameter being the new value of our text widget. The new value is automatically stored in a variable called *newValue*, which can be accessed

[7]https://www.eclipse.org/sirius/doc/specifier/Properties_View_Description.htm

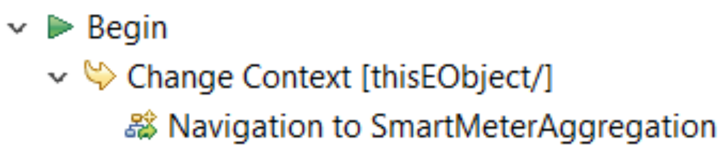

Begin
Change Context [thisEObject/]
Navigation to SmartMeterAggregation

Figure 6.27: Screenshot of the smart meter aggregation node double click action

by *var:newValue*. The external java action itself simply transforms the string value of the text field into two double values and sets the attributes of the *LocalMinimumCoords* instance accordingly.

6.6.4.4 Opening a New View as Editor

Within this section we create a view as editor by double clicking on the aggregation of a smart meter. This extension is only new on graphical editor level, since the aggregation is already a valid attribute in the topology meta-model. In order to create a double click event on the aggregation, we first have to add the aggregation as border node to the smart meter. This is done easily by importing the smart meter from the topology model and adding a new border node containing the smart meter's aggregation. Then, we need to configure our double click event. Since figure 6.22 only shows the presence of the double click event *SmartMeterDoubleClick*, we also provide figure 6.27 to show that action in detail. The double click event itself is mapped to the new aggregation node, but could also be mapped to the smart meter node as well. As we can see, the Sirius mechanism to open a new editor window is fairly simple. We make sure that we are indeed in context of the current smart meter and then use a navigation action to open the corresponding diagram. The *SmartMeterAggregation* described in the details of the double click action is the same diagram description we can see in figure 6.22 at the top. For the navigation action we can choose whether a new diagram should be created, if it doesn't exist. If that box is unchecked, then nothing happens, when double clicking on the border node, in case the smart meter doesn't have its own representation in a diagram.
Now that we know how to open a new diagram based on a smart meter, we need to take a short look at that description as well to show how multiple elements can be shown, if they are based on a single integer attribute like the aggregation. Therefore, figure 6.28 shows the diagram description of our new smart meter diagram. The description only contains one aggregation node and a section for adding additional nodes and removing existing ones. Since the aggregation attribute is only an integer, we can not use a standard semantic candidates expression to tell the diagram to represent a number of smart meters according to the value of the aggregation attribute. Therefore, we need our java service. The java service method takes the current smart meter's aggregation and creates a list of smart meters depending on the aggregation's value. For the increase of the aggregation we can use a palette entry beside the given properties view. Instead of creating a new smart meter through the palette entry, we simply increase the root smart meter's aggregation by one. As we increase and represent the aggregation of the smart meter in such a complicated way. we also need to define a delete feature managing the deletion of one of the diagram's nodes. Therefore, we can implement an external java action handling the decrease of the

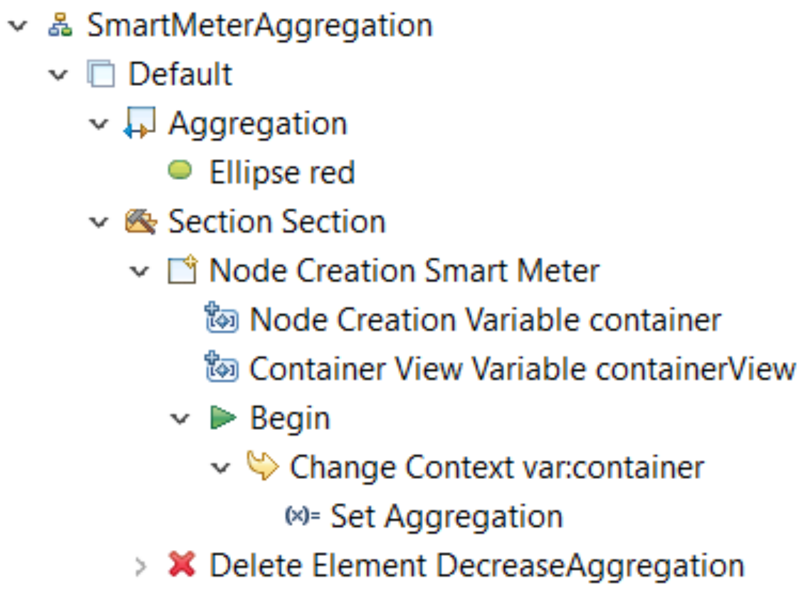

SmartMeterAggregation
Default
Aggregation
Ellipse red
Section Section
Node Creation Smart Meter
Node Creation Variable container
Container View Variable containerView
Begin
Change Context var:container
Set Aggregation
Delete Element DecreaseAggregation

Figure 6.28: Screenshot of the smart meter aggregation diagram description

aggregation. Decreasing the value the same way as we increased it doesn't work, since the diagram thinks that we are dealing with real smart meters, which are in fact linked nowhere in the underlying business model.
Such an implementation in Sirius comes with a major drawback. As smart meters inside a smart meter only exist as an integer value, the diagram behavior is broken when adding *fake* smart meters as nodes. Therefore, we can not change the layout of the represented nodes or resize them. However, this extension should only demonstrate the creation of a new editor, where in a real scenario actual domain elements are contained in the new diagram root node.

6.6.5 Graphiti Implementation

This section deals with the implementation of the artificial extension in Graphiti. Other than in Sirius, we leave out the navigation to a new editor as well as extending the properties view. The first one is excluded as we already implemented a similar feature in section 6.3.2.3, where we created a new diagram based in the selection of a topology model. The navigation to a new editor can be implemented in a similar way, where the selection is not a topology model but a smart meter in the current diagram. The properties view is excluded from this implementation, as there is no Graphiti specific API handling the representation of the properties view as there is in Sirius 4.0.
The rest of this section addresses adding new notation elements namely the intrusion detection system and the generic connection. At last, adding the cost function as new compartment is also discussed. The result of the implementation can be seen in figure 6.29, which is referenced throughout this section.

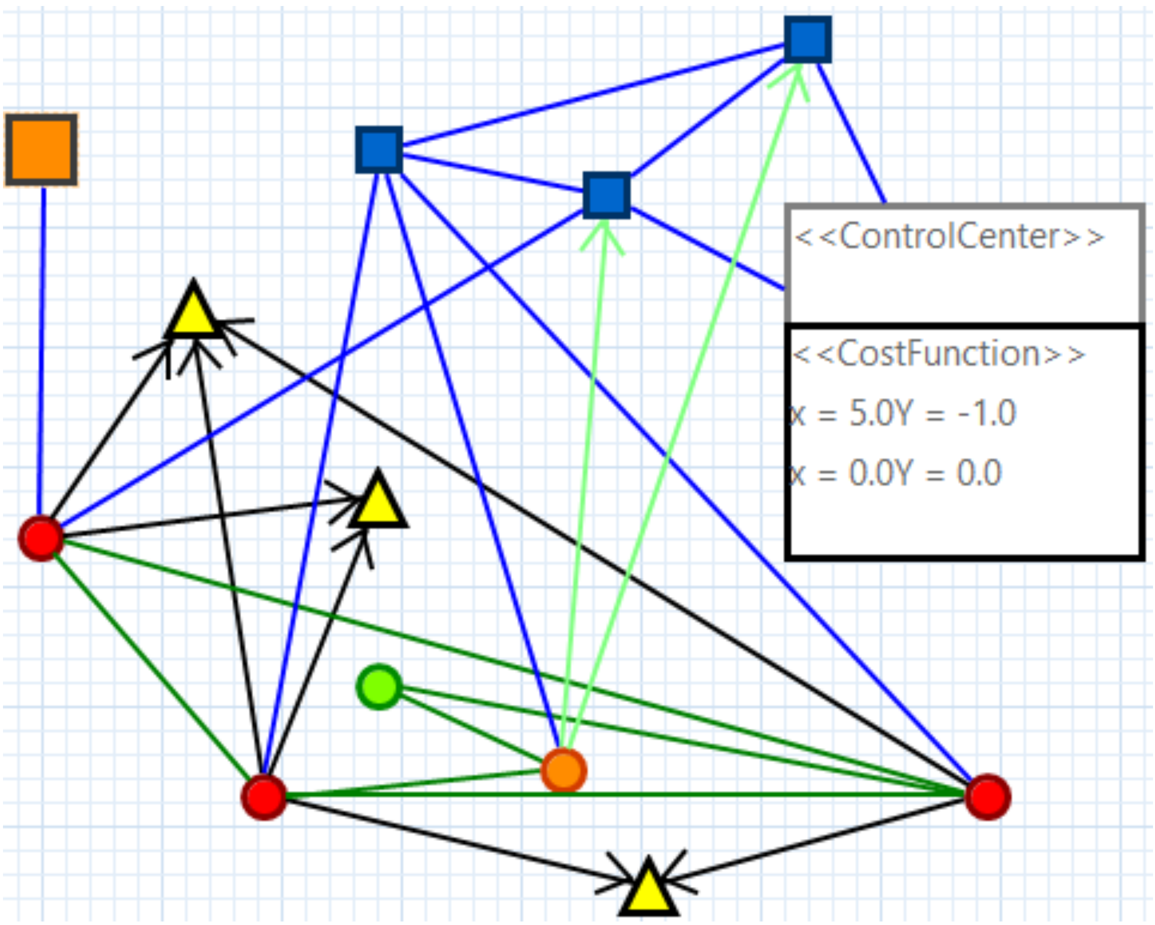


Figure 6.29: Screenshot of the artificial extension implemented in Graphiti

6.6.5.1 Adding a New Node

As in the Sirius implementation, we also want to add the intrusion detection system as node to our Graphiti implementation. The intrusion detection system is represented as orange square in figure 6.29. As we can see from the figure, there can also be a physical connection attached to the intrusion detection system. This can be done without creating an extended physical connection as we needed to do in the Sirius implementation. Adding an intrusion detection system can be done with the help of a new *add feature*. The add feature itself is implemented the same way as described in listing 6.3. The difference between a pattern and the add feature is that the pattern class contains methods for every possible feature, while the add feature only contains methods for adding the element to the diagram and checking whether it can be added or not. Furthermore, the add feature does not need to be added to the feature provider as the pattern does. In order to show the intrusion detection systems available in our diagram, we need to call the *add* method of its feature. Therefore, the code presented in listing 6.21 can be applied to any new element that should be represented in the current diagram. The code is called from our toolbar button as the elements should all be represented as son as the *smartgridextension* model is loaded.

Listing 6.21: Adding a new node from an extension to an existing diagram in Graphiti

```java
private void addAddFeature(EObject newObject, AbstractAddFeature feature, ContainerShape
    targetContainer) {
 final AreaContext area = new AreaContext();
 area.setLocation(100, 50);
 final AddContext add = new AddContext(area, newObject);
 add.setTargetContainer(targetContainer);
 add.setNewObject(newObject);
 final CommandStack commandStack = this.diagramContainer.getDiagramBehavior().
     getEditingDomain().getCommandStack();
 commandStack.execute(new RecordingCommand(this.diagramContainer.getDiagramBehavior().
     getEditingDomain()) {
  @Override
  protected void doExecute() {
      feature.add(add);
  }
 });
}
```

The first five lines of the given method create the context for the element to be represented. Therefore, its location has to be set, which should be different for each element or, alternatively, a layout algorithm has to be applied to the new elements after they got created. After setting the location, the element's container and its business object have to be set to the context. The container for an intrusion detection should always be the current diagram but for other domain model objects the container could change. The last lines of this listing again address the *CommandStack* and execute the feature's *add* method so that the new element can be added to the active diagram.
Since existing intrusion detection systems are now represented, we also want to create new ones directly in the diagram. Unfortunately, ones the core editor is implemented

Graphiti does not offer any possibility to add new elements to the palette view for creation purposes. This is caused by the feature provider, which provides only methods to receive all its known create features. However, as they are returned as fixed array, there is no possibility to add further create features. Furthermore, a feature provider can not be set a second time once the diagram is created. In order to accomplish adding elements to the palette view a custom feature provider has to be used already in the core editor providing specific methods for this task. The same problem occurs for creating a new context dependent menu button, when right clicking on the diagram's surface. Graphiti does support changes for this menu but there must be an extension point defined in the core editor so that further buttons can be added to it in an extension. The only extension point the core editor offers in that case regards changes in the mouse-over menu. However, this menu only is active, when hovering over an existing element and not while hovering over the diagram itself. Therefore, adding a new button to this menu, like we did in the input model implementation in section 6.4.4.3, is not suitable for this task.

6.6.5.2 Adding a Connection

Adding a new connection works similar to adding a new node. The only change is that instead of an *AddContext* an *AddConnectionContext* is required, which needs a source and a target anchor. For our *GenericConnection* we hereby need again an *AddFeature* and the delegation to its add method from our *LoadSmartGridExtension* class responsible for the loading of an artificial extension model.
As mentioned, an *AddConnectionContext* requires a source as well as a target anchor. We can address these by finding the source and target container shape for the connection to be created. Finding the correct container shapes is done by comparing the linked network entity's ID of each container shape with the source and target network entity's ID of the generic connection to be created.
The implementation of the *add feature* is similar to the implementation for the power connection described in section 6.3.2.2 and is therefore not further discussed here. For creating new connections directly inside the diagram we have the same problems as we did when creating a new intrusion detection system. However, in this case we can add a mouse-over button since a connection needs to be created between a source and a target entity. The creation of a new mouse-over button was also discussed in section 6.4.4.3 and is therefore not further discussed here.

6.6.5.3 Adding a Compartment

The cost function compartment and its content can be added to the control center as the intrusion detection system can be added to the diagram including the same drawbacks. However, a difference that can be made concerns the location of the compartment. As the container shape of the compartment is the control center, we can make the location of where to put the cost function more exact. Therefore, we alter the code described in listing 6.21 a bit to the listing in 6.22.

Listing 6.22: Adding a predefined region to the control center with variable location information

```
 1 if (!(targetContainer instanceof Diagram)) {
 2   int childrenY = 1;
 3   if (targetContainer.getGraphicsAlgorithm() != null
 4       && targetContainer.getGraphicsAlgorithm().getGraphicsAlgorithmChildren() != null) {
 5     childrenY = 25 * targetContainer.getGraphicsAlgorithm().getGraphicsAlgorithmChildren
           ().size();
 6   }
 7   area.setLocation(0, childrenY);
 8 } else {
 9   area.setLocation(100, 50);
10 }
```

Instead of defining a fixed location, we make the location in the area context dependent on the given container structure. If, for example, a cost function should be added to the control center, the check in line three returns true. This is due to the fact that the control center as target container has a *graphics algorithm child*, namely the *«ControlCenter»* label. For each label, and we assume that labels are the only additional children a control center shape has, we assume a height of 25 and multiply this height with the number of total children. The new location's x coordinate is then set to zero, as it should start on the left border of its container, while its y coordinate is dependent on the value of *childrenY*. The code in listing 6.22 can also be applied for the cost function's content. Then, the target container resembles the cost function and its children are dependent on the number of the cost functions optimums already added to the cost function container shape.
For the creation of such a cost function and its local minimum coordinates a mouse-over button can be applied, as we need an existing element to add the cost function to. The implementation of such a mouse-over button can be done according to section 6.4.4.3. Apart from the creation, the compartment as such can not be moved or resized. The inability to resize the compartment is due to the implementation of the topology editor but could be overridden. However, if we also want the compartment to be moveable inside the control center, we also would have to implement a so called *MoveShapeFeature*[8] specifically designed for the cost function and its content.

6.7 Summary of the Validated Mappings

During this section we sum up this chapter. Thereby, we focus on the validation of our approach. First, we give an overview on the extension mechanisms and types validated on meta-model level. Second, we compare the validated extension types on graphical editor level for both frameworks. At the end of this section, we show which mappings have been validated for which framework.

[8]http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.graphiti.doc%2Fjavadoc%2Forg%2Feclipse%2Fgraphiti%2Ffeatures%2Fimpl%2FDefaultMoveShapeFeature.html

	Meta-Class	Attribute	Containment	Relation
Association	x	x		x
Composition			x	
Inheritance	x	x		
Stereotype		x		

Table 6.2: Overview on the validation of extension types and mechanisms on meta-model level

6.7.1 Summary of the Validation on Meta-Model Level

In this section, we sum up the validation done on meta-model level. Thereby, table 6.2 is given. Both meta-class extension types are here combined, as their differentiation is only required when considering the mapping towards graphical editor extension types. Furthermore, the extension mechanism realization is left out, since interfaces in EMF are always marked as abstract and therefore, can also be considered as abstract classes for our purposes. As we can see from the table, not all possible extension mechanism to extension type mappings have been covered during our validation. If compared to table 4.1 in section 4.5, the association misses the containment extension type. Furthermore, we didn't realize the containment and relation mapping with the inheritance extension mechanism. However, all extension types on meta-model level have been validated by our extensions and all extension mechanisms have been used for the realization of at least one extension type.
The associations were used for the *GenericController* in section 6.6.2, where we covered the relation and the meta-class extension type. Furthermore, we used associations in the input model to add further attributes.
The composition was only used during the artificial extension, when considering the *CostFunction* as well as its local minimum coordinates. Inheritance was also used during the artificial extension but also when adding entity states as attributes to network entities in the output model extension.
At last, we used a stereotype in order to add another attribute to the control center, also in section 6.6.2.

6.7.2 Comparison of Extension Types on Graphical Editor Level

For a better comparison of both frameworks regarding their capabilities in realizing extension types, table 6.3 is given. The table shows all extension types regarded in section 4.6 and indicates, which of those are realizable by which framework. As we can see, all extension types responsible for the representation of a model element are realizable and have been validated in both frameworks. We only have to differentiate, when it comes to the creation or altering of element's values. For Graphiti, both the palette entry and

	Sirius	Graphiti
Node/Container	x	x
Connection	x	x
Annotation	x	x
Change Appearance	x	x
Compartment	x	x
Sub-Node/Sub-Container	x	x
Palette Entry	x	(x)
Context Dependent Menu Button	x	(x)
Toolbar Button	x	*
Properties View Entry	x	*
New View	x	x
Outline View Extension	**	**

Table 6.3: Direct comparison of the possible extension types in both frameworks

the context dependent menu button are in brackets, as their realization depends on the core editor. If the core editor provides custom extension points, whose class instances are added to the respective methods, further palette entries and context dependent menu buttons are possible in Graphiti. However, if the core editor should not be altered at all, creation of new model elements is only possible by creating a new editor in Graphiti.
The extension types marked with an asterisk may be possible but are not dependent on the framework. As we have shown in section 6.4.2, we can add a toolbar button to the Eclipse toolbar providing the same functionality as in Sirius. However, as Graphiti does not provide its own toolbar, the toolbar button is also independent of the Graphiti framework. Although, it has not been implemented within this thesis, Graphiti does not provide altering the properties view neither as extension point nor as any intern method, which needs to be overridden. On the other hand, Sirius does provide this functionality, but only as recently as the current version 4.0 was released in *Eclipse Neon*.
The last extension type the table shows is the extension of the outline view. This extension type was not implemented by any of the two prototypes, which is why we can not make any well-founded statements on this extension type for neither framework.

6.7.3 Validation of the Mapping in Sirius

This section deals with the summary of the validation of the mapping in Sirius. The validation in Graphiti is regarded in the next section. To get a better overview on the result of the validation, table 6.4 is given. Starting in the first row, we can see that we covered only two out of five different mappings given this meta-model extension type. However, the other possible mappings were a connection, a context dependent menu button and a new, which we've shown that they can be implemented, at least for other extension types. The procedure in Sirius, however, is the same so that we can safely say that the three remaining mappings can also be implemented using the Sirius framework.
Unfortunately, we only managed to cover two out of the nine different mappings the

	Node/Container	Connection	Sub-Node/-Container	Annotation	Compartment	Change Appearance	Outline View	Palette Entry	Context Dependent Menu Button	Properties Entry	Toolbar Button	New View
Meta-Class, Below Root Node	x							x				
Meta-Class, Part of Other Instance		x					x					
Attribute				x		x			x	x		x
Containment			x		x			x		x		
Relation		x						x				

Table 6.4: Overview on the validation of the mapping in Sirius

second meta-model extension type has to offer. Although, the validation of some of these mappings can be inferred from other extension type's mappings, not all of them can be validated that way.
Considering the attribute extension type we managed to cover five out of eight different mappings. The representational extension types were covered during the input model extension and the artificial extension, as was the context dependent menu button. The properties view entry was added for the coordinates of the local minimum coordinates instances for the cost function in the artificial extension. The new view, on the other side, was created for the aggregation attribute belonging to the smart meter.
The containment extension type was only realized in the artificial extension. Therefore, we covered four out of seven different mappings. The last extension type, whose mapping needs to be regarded, is the relation extension type. Here, we covered two out of five different mappings. Indirectly, the relation extension type was also realized as compartment and sub-node. However, within the mapping we only regard the direct realizations of the relation extension type.
Although, we created various toolbar buttons, they didn't actually serve the purpose of realizing any extension type. That is why they are left out in this summary. However, as we managed to implement toolbar buttons, we are confident that they can also be used, for example, for changing the status of a notation element.

	Node/Container	Connection	Sub-Node/-Container	Annotation	Compartment	Change Appearance	Outline View	Palette Entry	Context Dependent Menu Button	Properties Entry	Toolbar Button	New View
Meta-Class, Below Root Node	x											
Meta-Class, Part of Other Instance		x										
Attribute				x		x			x			
Containment			x		x			x		x		
Relation		x							x			

Table 6.5: Overview on the validation of the mapping in Graphiti

6.7.4 Validation of the Mapping in Graphiti

Like we did for the Sirius framework, we sum up the validated mappings also for the Graphiti framework. As in the previous section, we present table 6.5 for a better overview. While we were able to validate 15 out of 34 different mappings for Sirius, we only validated eleven mappings for Graphiti. This was because of the fact that the core editor would have to be extended, in order to support additional palette entries. Furthermore, Graphiti doesn't support an extension of the properties view, which is why we couldn't validate the mappings, considered with the properties view, either. However, we have managed to validate the same amount of mappings considering the pure representation, as we did for the Sirius framework.
Further mappings in Graphiti would have been possible, since we could have, for example, implemented a toolbar button serving the purpose of switching the power outage status of each power state on and off. However, to cover all mappings equally, further meta-model extensions would have been needed, as we can not realize multiple representation extension types for a single meta-model extension type.

7 Evaluation

While the previous chapter considered the implementation of both frameworks and therefore, the validation of our approach, we evaluate the insights achieved throughout this chapter. Therefore, we start with a comparison between both the Graphiti and the Sirius framework. After that, we take a look at the scenario, where more meta-model content is available, but there is no editor handling the content. Thereby, we also consider the other case, where the graphical editor extension exists, but there is no valid model for the given extension. A last section within this chapter discusses further scenarios extending given meta-models. These scenarios are only discussed theoretically and are not implemented. However, these scenarios proof again that the mapping depicted in chapter 5 is valid, as we also discuss the appliance of these scenarios to a possible Sirius implementation.

7.1 Comparison Between the Graphiti and the Sirius Framework

This section deals with the direct comparison of both frameworks used. Therefore, advantages and disadvantages of each framework are taken into consideration. After we compared the capabilities of each framework to realize the given extension types in section 6.7.2, we can also compare both frameworks towards their advantages and disadvantages. We hereby divide this section into five different parts, where each part is concerned with one aspect of creating and using an editor, as well as its extensions.

7.1.1 Creating the core editor

When creating the core editor for a given meta-model, we don't have to consider possible extensions, when creating the editor in Sirius. We can still use the complete functionality in any extension independent of the core editor's design. Furthermore, the effort creating the core editor is pretty low, as the core Sirius editors already contain a wide range of functionality. A minor disadvantage of Sirius for this concern is that the model always has to be created first. There is no feature that lets you create the diagram first and ,from there, automatically creates the model.
In Graphiti on the other hand, we have to know whether one or more extensions are created in the future. If there will be any further extensions, different extension points have to be defined that regard adding create features in the feature provider and adding context dependent menu buttons. Furthermore, the effort creating a core editor with almost the same functionality Sirius offers, is a lot higher as custom diagram, feature provider, diagram behavior and other classes have to be created. Additionally, we need to write plain java code for each notation element to be represented, whereas we can use the

predefined viewpoint description editor in Sirius to create the required notation elements with only little java writing necessary. At last, Graphiti does neither offer a mechanism for creating a diagram without an existing model nor creating a diagram for an existing model. Both features have to be implemented.

7.1.2 Toolbar

The fact that Graphiti doesn't offer a toolbar, such as Sirius, leads to more than one disadvantage. As there is no toolbar in Graphiti, we have to add buttons to the Eclipse toolbar itself causing the toolbar to appear crowded. If only one or two extensions should be added to the current diagram, this is not a problem. However, if other buttons should be added as well, a new toolbar specifically designed for the Graphiti diagram may need to be implemented. Furthermore, there is some functionality missing in Graphiti due to the missing toolbar, such as a layout button. If a new diagram, based on a model, should be created and there is no layout algorithm applied, all model elements have to be dragged from hand to their position. Other functionality such as the layer buttons in Sirius are nice to have but can be implemented in Graphiti with the help of *clear* buttons, as we did in section 6.4.4.4 as well.

7.1.3 Creating the extension

Concerning the pure creation of an extension Sirius again has the advantage that the viewpoint description editor can be used for most of the extension. However, as a different root model element is used in the extension than in the core editor it becomes necessary to use java services and external java actions which also have to be implemented with java code. In Graphiti we can add new notation elements or change existing ones the same way we did when creating the core editor. For their creation in the diagram the predefined extension points have to be implemented. The features created in the extensions need only to be called at the appropriate point in time. Therefore, the effort in creating the extension in Graphiti is reduced compared to the creation of the core editor as we don't have to implement the diagrams basics again. On the other side the effort for creating an extension in Sirius is higher than creating the core editor as we most certainly are required to write java code in order to support the extension meta-model.

7.1.4 Adding an extension model to the diagram

Adding an extension model to the diagram can be implemented in both frameworks by adding a new toolbar button. Sirius furthermore, automatically adds every valid model to the current diagram that exists in the diagram's project. Such a feature can be an advantage, since there is no need for a toolbar button but can also be disadvantageous. That is the case for many extension models referring to the diagram's underlying core model. Regarding our running example, it is possible that more than one input model exists for a given topology model in order to represent different scenarios. Currently, all input models would be loaded into the diagram, as they are all valid causing the Sirius editor to only load the first model applicable and ignoring the other ones.

Graphiti on the other hand, first needs a *ResourceSetListener* implemented in the core editor that ensures that all loaded models are saved together. Furthermore, if a desired extension should be added to the diagram, a toolbar button is necessary loading the extension to the diagram. The advantage of this method is that always only the loaded models and the core model is active. Therefore, we can take a look at multiple valid input models for a single topology model without putting more effort in the implementation.

7.1.5 Further drawbacks

The last paragraph in this section regards further drawbacks for both frameworks individually. A minor drawback in Sirius is that mouse-over buttons aren't supported by the core editor. Like Graphiti, there would be the need for a custom editor implementation to enable mouse-over buttons in Sirius. A bigger drawback in Sirius is that changes made on the same notation element, whether it is a node or a container are not commutative throughout independent extensions. That means if in our running example all three extensions are active at the same time, the control center only shows the elements of the smart grid extension layer. A possible, but untested, reason could be that Sirius prioritizes its extensions alphabetically leading to the smart grid extension having the highest priority. Graphiti on the other hand, simply adds all elements successively to the container independent of the extension loaded last.
An advantage in Graphiti, and at the same time a disadvantage for Sirius, is the activation of an extension, if no valid extension model exists. In Graphit this simply isn't possible, as all extensions need to be loaded to the diagram by the toolbar button. In Sirius on the other hand, we simply activate a certain extension layer. Even if there is no valid extension model, the layer still becomes active leading to *NullPointerExceptions*, as all java service methods can not access the extension, since it is not present in the current session. The editor itself can still be used, but as long as the extension layer is active, further exceptions will be thrown. That happens, for example, if a new *intrusion detection system* should be created, and there is no smart grid extension model to add the intrusion detection system to.
All in all, we can say that using the Sirius framework for the creation of core editor and its extensions requires a lot less effort than using the Graphiti framework. This is also due to the fact that Sirius offers a framework including UI components, such as the viewpoint description editor, while Graphiti only offers an API used within plain java code.

7.2 Content not Supported by the Current Graphical Editor

This section deals with the last research question considering content that is available as meta-model extension, but no graphical editor extension is available for representation and creation of the meta-model extension elements. Of course, if the editor doesn't know that a certain model is an instance of a meta-model extension, there will be no additional content in the editor considering this model. However, it may be useful for the user to see, whether there is additional content, as he may want to take the meta-model extension into account. As this section is theoretical, we only discuss such possibilities for the Sirius

framework.
In order for such a feature to be realized, we could use an additional diagram extension or structure our core editor accordingly. We here assume that we only extend the core meta-model. However, the presented approach can also be extended in such a way that unknown extensions of an extension become visible. Assuming that any notation element could be extended, we need to add a further notation element for each already existing notation element with a connection between the two elements. As we don't know the domain class extending our notation, element we use the same domain class as for the notation element to be possibly extended. The difference here lies only in the *semantic candidates expression*. The appropriate extension class can not be found with the help of the *acceleo query language*, as it can not be used for such general terms. Therefore, we need to implement a java service method. This service method would need to visit every resource in the current project that is not known and find possible references to the domain class, which is possibly extended. This approach would have to be done for every domain class that is represented in the current diagram. As a label for these new notation elements, we could then add their corresponding resource to show the user, where additional content can be found. If there are multiple extensions of the same type, the further represented notation elements may be redundant. This strategy, of course, only works for such extensions, where the core model element is referenced by a composition or unidirectional association. If the corresponding meta-class is extended by inheritance, another strategy needs to be found, such as trying to up-cast every class instance found. The validation of such a feature is a challenge for future work and is not further discussed here.

7.3 Further Scenarios

This section deals with additional scenarios that were not addressed within the last chapter but are still worth mentioning. Therefore, we shortly describe each scenario and its purpose. During the description, a special focus lies on the extension types on meta-model level and how they can be possibly realized in a theoretical graphical editor extension without implementing such an extension. We hereby only focus on the extension types used for representing the model elements as most of the creation extension types strongly dependent on the developers preference. Each of the presented extensions extends the *Palladio Component Model* (PCM) [3]. The PCM is an approach for performance prediction during the design time of a software system. It comprises five sub-models: repository, system assembly, resource environment, deployment and usage. The relevant models for each extension are explained shortly in the respective subsection.

7.3.1 IntBIIS

The *Integrated Business IT Impact Simulation* (IntBIIS) is an extension for the Palladio Component Model, which addresses only the usage model and is based on Heinrich et al [22]. According to Becker et al [3], the PCM usage model describes typical or critical usage scenarios and parameter values for the system in development. IntBIIs consists of two

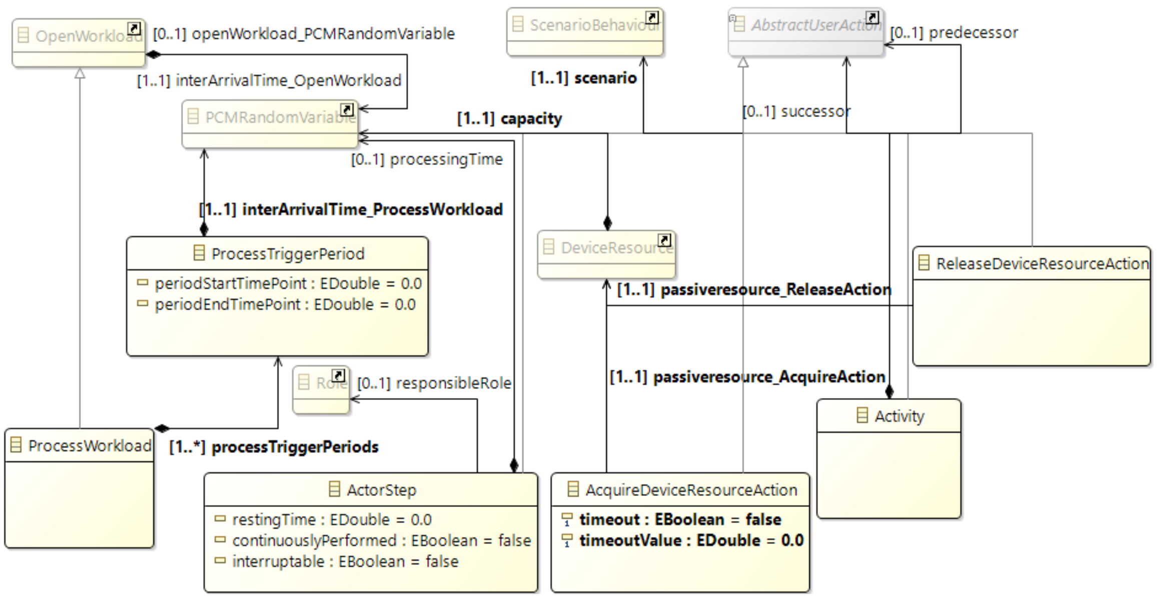

OpenWorklo
[0..1] openWorkload_PCMRandomVariable
[1..1] interArrivalTime_OpenWorkload
ScenarioBehavio
[1..1] scenario
AbstractUserActi
[0..1] predecessor
PCMRandomVaria
[1..1] capacity
[0..1] successor
[0..1] processingTime
[1..1] interArrivalTime_ProcessWorkload
ProcessTriggerPeriod
periodStartTimePoint : EDouble = 0.0
periodEndTimePoint : EDouble = 0.0
DeviceResour
[1..1] passiveresource_ReleaseAction
ReleaseDeviceResourceAction
Ro
[0..1] responsibleRole
[1..1] passiveresource_AcquireAction
Activity
ProcessWorkload
[1..*] processTriggerPeriods
ActorStep
restingTime : EDouble = 0.0
continuouslyPerformed : EBoolean = false
interruptable : EBoolean = false
AcquireDeviceResourceAction
timeout : EBoolean = false
timeoutValue : EDouble = 0.0

Figure 7.1: IntBIIS meta-model with new meta-classes on the right side and existing classes on the left side

different packages, whereas only one actually extends the PCM usage model. Therefore, only this extension is regarded. The extension offers overall six new meta-classes that can be seen in the meta-model in figure 7.1. Starting with the *Activity* meta-class, we can see that there are overall two references to meta-classes of the usage meta-model. One reference is the extension of the abstract class *AbstractUserAction*. As concrete user actions are displayed as sub-nodes or sub-containers in the usage model, the activity should be represented the same way. The same argumentation holds for *AcquireDeviceResourceAction*, *ActorStep* and *ReleaseDeviceResourceAction*, as all of these four classes extend the *AbstractUserAction* meta-class. Furthermore, an activity also references *ScenarioBehaviour* with a composition. Therefore, a scenario behavior should be realized as compartment of the activity making the activity a container element.

Regarding the *AcquireDeviceResourceAction*, as well as the *ReleaseDeviceResourceAction*, we can see another reference towards the *DeviceResource* meta-class. This meta-class is not part of the usage model, but part of the second package of this extension. However, since both resource action meta-classes inherit the *AbstractUserAction* meta-class and contain further attributes, these two could also be realized as container. For the same reasons, the *ActorStep* meta-class, together with its attributes can also be realized as container, where the referenced meta-class instances can also be represented as annotation or sub-container, depending on the realization of the *Role* meta-class.

The last meta-class directly referencing the usage model is *ProcessWorkload*. As its inherited class is also represented as sub-container, the process workload should also be represented as sub-container having a compartment containing *ProcessTriggerPeriod* meta-class instances.

As none of these classes seem to require any special instantiation all of these classes could

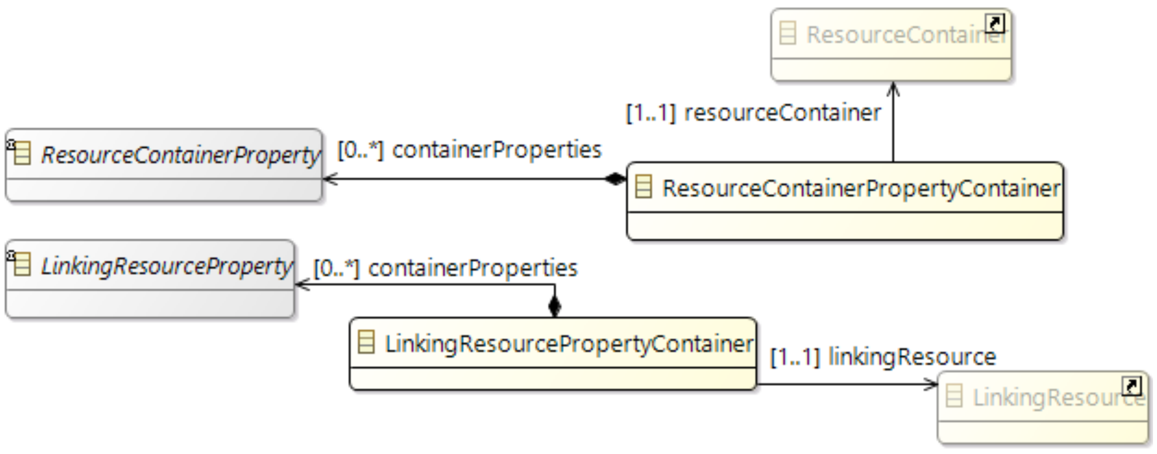

ResourceContainer
[1..1] resourceContainer
ResourceContainerProperty
[0..*] containerProperties
ResourceContainerPropertyContainer
LinkingResourceProperty
[0..*] containerProperties
LinkingResourcePropertyContainer
[1..1] linkingResource
LinkingResource

Figure 7.2: Excerpt of the *ContainerStereotypes* package of the meta-model developed by Czogalik [7]

be added to the palette view to ensure their creation. Changing any of their attributes' values could be done directly in the diagram or as a new properties entry.

7.3.2 Security Extensions

The next extension discussed in this section also deals with the Palladio Component Model. Busch et al propose an approach for assessing security of component-based software architectures [5]. To accomplish this extension on meta-model level only the *repository* and the *resource environment* models in the PCM have to be extended. The resource environment model defines resource containers and the network topology, while the repository contains interfaces and components that are used throughout the definition of the complete model instance [3]. The extension thereby focuses on adding new attributes by stereotyping. Therefore, the extension on a possible graphical editor is also limited to a few extension types. Currently, there is no meta-model given for this extension leading to the fact that all realizations according to the attribute extension type are possible assuming that stereotyping is the only extension mechanism used. If other extension types and mechanisms, such as providing relations and different references will be used, both mappings regarding relations and attributes in the sections 5.3 and 5.5 have to be taken into account.

7.3.3 Architectural Data Flow Analysis

A last extension we want to discuss within this chapter focuses on the work of Seifermann in [49] and the thesis of Czogalik in [7]. We hereby point out the different extension types by analyzing the given meta-model. As the meta-model itself is quite large only those classes are regarded that can be considered a direct extension to one of the given PCM sub-models. The meta-model consists of five different packages. Three of them are partly shown and analyzed towards their extension types within this section.
Starting with figure 7.2, we see an excerpt of the *ContainerStereotypes* package. In this

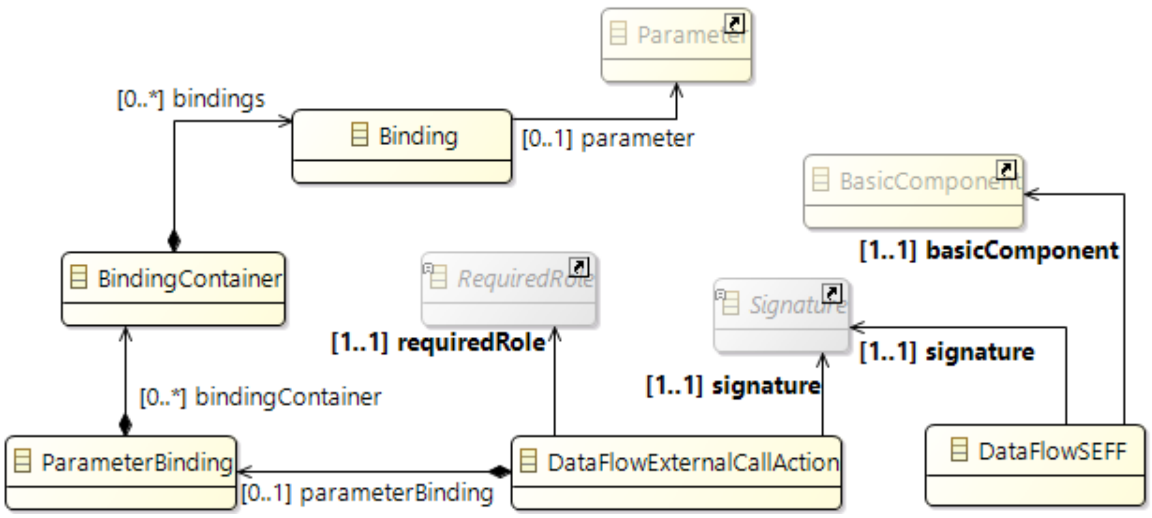


Figure 7.3: Excerpt of the *DSEFF* package of the meta-model developed by Czogalik [7]

package, there are overall two meta-classes directly referencing meta-classes of the PCM. The *LinkinResourcePropertyContainer* class references *LinkinResource*, while the *Resource-ContainerPropertyContainer* references *ResourceContainer*. The referenced classes are both contained in the resource environment sub-meta-model of the PCM. According to our classification, both references result in a relation extension type. However, as they both have a cardinality of one and a further composition attached to them, both of these meta-classes realize the containment extension type, as described in section 4.4.1. Possible ways of realizing the containment extension type on graphical editor level are analyzed in section 5.4. Since both the *LinkinResource* and the *ResourceContainer* are realized as containers, we could simply add a new compartment to each of the containers. Further instances of sub-classes of *LinkingResourceProperty* and *ResourceContainerProperty* can than be added as nodes or containers to the compartment.
The next part of the meta-model we need to analyze, is the *DSEFF* package. An excerpt of this package with all meta-classes extending the PCM is shown in figure 7.3. Within this package, there are overall four references to PCM meta-classes. All of the referenced classes are meta-class of the *repository* sub-model. Starting with the *Binding* meta-class, we can assume, that, in an intrusively extended meta-model, these two classes would be modeled the same way as they are in this meta-model. Therefore, we have a relation extension type. The *Binding* meta-class itself is contained in the *BindingContainer* class and therefore realizes the containment extension type. If mapped to a graphical editor, we could implement this class and its reference together as annotation within a compartment that corresponds to the *BindingContainer*.
Looking at the right side of figure 7.3, we see the *DataFlowSEFF* meta-class. This class uses two unidirectional associations. One targets the abstract *Signature* class, the other one targets the *BasicComponent* class. As the original *service effect specification* also uses two references in a similar way we can here also assume that the extension would be the same, if the meta-model was intrusively extended. Therefore, both associations realize a relation extension type together with a meta-class extension type. This combination can result in

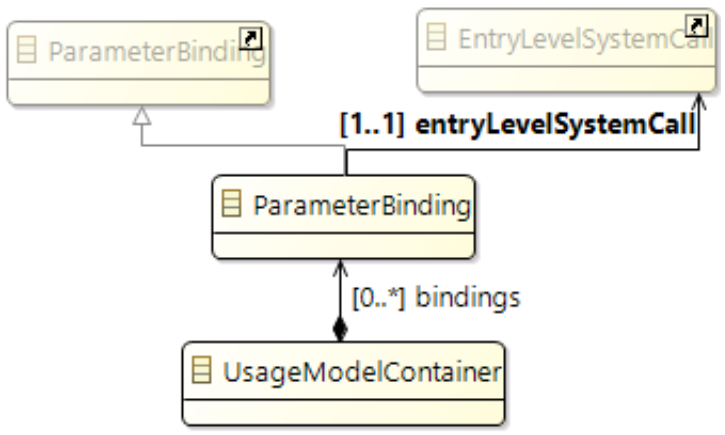


Figure 7.4: Excerpt of the *Usage* package of the meta-model developed by Czogalik [7]

the representation of the *DataFlowSEFF* as container containing each a *BasicComponent* and a *Signature*. Otherwise, both of the associations targets can be represented as independent nodes or containers with a connection to the *DataFlowSEFF*.
The last class to analyze is the *DataFlowExternalCallAction* meta-class. However, if compared to the SEFF meta-model the references in the data flow extension are interchangeable with the ones in the SEFF meta-model. Therefore, both associations linked to the *repository* classes again realize the relation extension type, while the *DataFlowExternalCallAction* resembles a meta-class extension type. The *ParameterBinding* can be realized as compartment of the *DataFlowExternalCallAction* container element, as it realizes the containment extension type.
The last package we need to discuss, is the *Usage* package shown in figure 7.4. The *Usage* package contains only one meta-class referencing the PCM usage model. On the left side, the *ParameterBinding* class inherits a class with the same name known from the *DSEFF* package shown in figure 7.3. Therefore, the meta-class in the usage package also contains further *BindingContainers*. Furthermore, as there is again one unidirectional association to the PCM meta-class with a cardinality of one, we can conclude that the *ParamterBinding* meta-class in the usage package realizes the containment extension type together with its relation to *EntryLevelSystemCall*. The mapping on graphical editor level therefore is the same, as mentioned before, where we discussed the *ContainerStereotypes* package.
The last two packages that complete this meta-model either extend meta-classes that were already discussed or extend generic meta-classes, such as *Identifier*. Those two packages are therefore not further discussed.

8 Conclusion

This last chapter concludes this thesis with a summary of its contributions and gives an overview on the possibilities of future work.

8.1 Conclusion

This thesis presented ways to extend a modular meta-model and transfer those extensions to graphical editors in the same manner. The extensions itself are designed independently from each other and extend the core meta-model or graphical editor or one of their extensions noninvasively. In contrast to related work, our graphical editors are as modular as the underlying meta-model. That means that if the user does not wish to use a certain extension, it can be left out on both, the graphical editor level as well as the meta-model level. Therefore, we started the main part of this thesis with a definition of extension types and mechanisms that were used throughout this thesis. Afterward, we discussed and analyzed a classification of extensions for meta-models as well as for graphical editors. Thereby, we introduced five different extension types on meta-model level that have an impact on a graphical editor extension. Graphical editor extension types can thereby be divided into first, the representation of a meta-model extension and second, into creating or altering the meta-model instance's value. For our understanding such a graphical editor can have up to twelve different extension types that are mostly based on the framework used for creating such an editor. Besides the extension types on each level, we also defined extension mechanisms that can be used for the realization of an extension type on their respective level.
As there was no indication given on which extension type on meta-model level results in an extension type on graphical editor level, a mapping between those two needed to be established. This was be done after the classification on both levels was completed. During the mapping, we analyzed the possible impact of a meta-model extension type on graphical editor level. The result was e shortened list of graphical editor extension types which could be implemented for a given meta-model extension type. Based on this mapping and the classification an implementation could be done. The actual implementation of course, is dependent on the developers intention, the given context and the possibilities of the framework used for the implementation.
To validate this concept and the mapping, we implemented two prototypes, each in a different framework. During the validation we have shown, that the classification on both levels is reasonable. The mapping following from this classification could also be validated for the most part, as not all mappings have been implemented.
Although, both the Sirius and the Graphiti framework are capable of implementing most of the extension types, when given a meta-model extension, we came to the conclusion

that Sirius should be preferred. Where Graphiti uses plain java code, a Sirius-based editor description is more structured and offers even more possibilities, such as a custom definition of a properties view, that comes with Sirius 4.0. Furthermore, we do not necessarily need to implement a toolbar button for loading an extension into our Sirius diagram. Sirius does automatically load every model instance in the current project, which has an active viewpoint, to the current session.
All in all, we've not only shown that the mapping between meta-model extension types and graphical editor extension types works. We also established guidelines showing how to extend graphical editors noninvasively for both the Sirius and the Graphiti framework.

8.2 Future Work

Future work based on this thesis may address several topics. On the one hand there could be future work aiming towards the classification of extensions and their mappings. On the other hand there still open issues concerning the implementation of graphical editors which can also be addressed in future work.
Starting with the classification we only focused on the main extension types and mechanisms for meta-models. However, there could be more extension types taken into account when considering the ecore meta-metamodel such as packages, classifiers or operations. Some of these may also have an impact on graphical editor extensions. Not only more extension types but also extension mechanisms can be considered on meta-model level such as decorating which wasn't mentioned in this thesis.
Among extending a modular meta-model and its impact on graphical editors there are other artifacts future work could target. Extending the meta-model may also have influences on further simulations which can also be regarded.
Now that graphical editor extensions based on meta-model extensions have been covered there are also reasonable scenarios where no meta-model extension is given but the editor may need an extension overall. That can be the case for simulations where editor support is advantageous in order to show the impact of a given input without having a further meta-model extension. In that way further functionality can be achieved.
The last three aspects of possible future work cover implementation parts of this thesis that haven't been fully evaluated. One of these two aspects was already mentioned in section 6.5.4 where two or more extensions are active at the same time but are independent of each other. The developer or the user may not want the editor's elements to change their appearance according to the last loaded extension. For that case rules have to be defined as to how to react in these situations. Section 6.5.4 indeed introduces a possible solution for this problem but this solution also needs to be validated and tested.
Besides having possible conflicts in two or more independent extensions future work may also regard content not provided by the current editor. There could be a meta-model extension with valid model instances but no graphical editor extension to represent the model. One way of dealing with such a case would be to somehow recognize the meta-model extension and represent it in the editor as cloud or other so that the user knows that there is another extension with references to the current active models.
A last future work based on this thesis may be researching possibilities to apply a graphical

editor extension also to a different core editor. An application would be a general core editor for tutorial purposes with low complexity where developers can start building simple extensions. These extensions can then be applied to another core editor with further functionality and more complexity.

Bibliography

[1] Andreas Andresen. *Komponentenbasierte Softwareentwicklung mit MDA, UML 2 und XML*. 2., neu bearb. Aufl. München: Hanser, 2004. ISBN: 3-446-22915-9. URL: `http://digitool.hbz-nrw.de:1801/webclient/DeliveryManager?pid=1589920&custom_att_2=simple_viewer`.

[2] Danilo Ardagna et al. "Project Deliverable D3. 1 Prediction Models Specification (revised version)". In: (2009). URL: `http://www.q-impress.eu/wordpress/wp-content/uploads/2009/05/D3.1-Prediction_model_specification_v20.pdf`.

[3] Steffen Becker, Heiko Koziolek, and Ralf Reussner. "The Palladio component model for model-driven performance prediction". In: *Journal of Systems and Software* 82.1 (2009). Special Issue: Software Performance - Modeling and Analysis, pp. 3–22. ISSN: 0164-1212. DOI: `http://dx.doi.org/10.1016/j.jss.2008.03.066`. URL: `http://www.sciencedirect.com/science/article/pii/S0164121208001015`.

[4] Lorenzo Bettini, ed. *Implementing domain-specific languages with Xtext and Xtend : Learn how to implement a DSL with Xtext and Xtend using easy-to-understand examples and best practices.* Community experience distilled. Bibliogr. [p. 311]-318. Index. Birmingham, UK [u.a.]: Packt Publ., 2013. ISBN: 978-1-78216-030-4; 1-78216-030-2 [Titel anhand dieser ISBN in Citavi-Projekt übernehmen].

[5] Axel Busch, Misha Strittmatter, and Anne Koziolek. "Assessing Security to Compare Architecture Alternatives of Component-Based Systems". In: *Proceedings of the IEEE International Conference on Software Quality, Reliability & Security.* QRS '15. Acceptance Rate (Full Paper): 20/91 = 22%. Vancouver, British Columbia, Canada: IEEE Computer Society, 2015, pp. 99–108. DOI: `10.1109/QRS.2015.24`.

[6] A. Colombo et al. "The Use of a Meta-Model to Support Multi-Project Process Measurement". In: *2008 15th Asia-Pacific Software Engineering Conference.* Dec. 2008, pp. 503–510. DOI: `10.1109/APSEC.2008.55`.

[7] Thomas Czogalik. "Datenflussmodellierung für Vertraulichkeitsanalysen in Palladio". Bachelor thesis. Karlsruher Institute for Technology, 2016.

[8] Arie van Deursen, Paul Klint, and Joost Visser. "Domain-specific Languages: An Annotated Bibliography". In: *SIGPLAN Not.* 35.6 (June 2000), pp. 26–36. ISSN: 0362-1340. DOI: `10.1145/352029.352035`. URL: `http://doi.acm.org/10.1145/352029.352035`.

[9] Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. "What is Needed for Managing Co-evolution in MDE?" In: *Proceedings of the 2Nd International Workshop on Model Comparison in Practice*. IWMCP '11. Zurich, Switzerland: ACM, 2011, pp. 30–38. ISBN: 978-1-4503-0668-3. DOI: 10.1145/2000410.2000416. URL: http://doi.acm.org/10.1145/2000410.2000416.

[10] Lajos Fejes, Gunnar Johannsen, and Gerd Strätz. "A graphical editor and process visualization system for man-machine interfaces of dynamic systems". In: *The Visual Computer* 10.1 (1993), pp. 1–18. ISSN: 1432-2315. DOI: 10.1007/BF01905527. URL: http://dx.doi.org/10.1007/BF01905527.

[11] The Ecipse Foundation. *Eclipse Graphiti Documentation*. Last visited on August, 20, 2016. URL: http://help.eclipse.org/mars/index.jsp?topic=/org.eclipse.graphiti.doc/javadoc/index.html%5C&help-doc.html.

[12] The Eclipse Foundation. *Graphiti Homepage*. Last Visited on March, 5, 2016. URL: https://eclipse.org/graphiti/.

[13] The Eclipse Foundation. *Sirius Homepage*. Version 3.1. Last visited on March, 5, 2016. URL: https://eclipse.org/sirius/.

[14] Martin Fowler. *Domain-specific languages*. 1. print. The Addison-Wesley signature seriesA Martin Fowler signature book. Upper Saddle River, NJ: Addison-Wesley, 2010 [erschienen] 2011. ISBN: 978-0-321-71294-3; 0-321-71294-3. URL: http://bvbr.bib-bvb.de:8991/F?func=service&doc_library=BVB01&doc_number=020690261&line_number=0001&func_code=DB_RECORDS&service_type=MEDIA.

[15] Erich Gamma, ed. *Design patterns : elements of reusable object-oriented software*. 37. print. Addison-Wesley professional computing series. Boston, Mass.: Addison-Wesley, 2009. ISBN: 0-201-63361-2; 978-0-201-63361-0 [Titel anhand dieser ISBN in Citavi-Projekt übernehmen].

[16] Object Management Group. *Object Constraint Language*. English. Version 2.4. Last visited on April, 21st, 2016. Feb. 2014. URL: http://www.omg.org/spec/OCL/2.4/.

[17] Object Management Group. *OMG Meta Object Facility (MOF) Core Specification*. English. Version 2.5. Last visited on April, 4th, 2016. June 2015. URL: http://www.omg.org/spec/MOF/2.5/.

[18] Object Management Group. *OMG Unified Modeling Language^{TM} (OMG UML)*. English. Version 2.5. Last visited on April, 4th, 2016. Mar. 2015. URL: http://www.omg.org/spec/UML/2.5/.

[19] The Open Group. "TOGAF". In: 2011. Chap. 34. Content Metamodel. URL: http://pubs.opengroup.org/architecture/togaf9-doc/arch/index.html.

[20] S. S. u. H. Mohani et al. "Smart grid system". In: *2016 SAI Computing Conference (SAI)*. July 2016, pp. 1278–1285. DOI: 10.1109/SAI.2016.7556144.

[21] Robert Heinrich. "Quality Modeling within Business Process Models". English. In: *Aligning Business Processes and Information Systems*. Springer Fachmedien Wiesbaden, 2014, pp. 59–77. ISBN: 978-3-658-06517-1. DOI: 10.1007/978-3-658-06518-8_4. URL: http://dx.doi.org/10.1007/978-3-658-06518-8_4.

[22] Robert Heinrich et al. "Integrating business process simulation and information system simulation for performance prediction". In: *Software & Systems Modeling* (2015), pp. 1–21. ISSN: 1619-1374. DOI: 10.1007/s10270-015-0457-1. URL: http://dx.doi.org/10.1007/s10270-015-0457-1.

[23] J. Henriksson et al. "Extending grammars and metamodels for reuse: the Reuseware approach". In: *IET Software* 2.3 (June 2008), pp. 165–184. ISSN: 1751-8806. DOI: 10.1049/iet-sen:20070060.

[24] Anders Hessellund and Andrzej Wąsowski. "Model Driven Engineering Languages and Systems: 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings". In: ed. by Krzysztof Czarnecki et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. Chap. Interfaces and Metainterfaces for Models and Metamodels, pp. 401–415. ISBN: 978-3-540-87875-9. DOI: 10.1007/978-3-540-87875-9_29. URL: http://dx.doi.org/10.1007/978-3-540-87875-9_29.

[25] "IEEE Standard Glossary of Software Engineering Terminology". In: *IEEE Std 610.12-1990* (Dec. 1990), pp. 1–84. DOI: 10.1109/IEEESTD.1990.101064.

[26] Yanbing Jiang et al. "«UML» 2004 — The Unified Modeling Language. Modeling Languages and Applications: 7th International Conference, Lisbon, Portugal, October 11-15, 2004. Proceedings". In: ed. by Thomas Baar et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. Chap. On the Classification of UML's Meta Model Extension Mechanism, pp. 54–68. ISBN: 978-3-540-30187-5. DOI: 10.1007/978-3-540-30187-5_5. URL: http://dx.doi.org/10.1007/978-3-540-30187-5_5.

[27] L. Jinghua and L. Yan. "Cross-fields Study of Modularity". In: *2006 IEEE International Conference on Management of Innovation and Technology.* Vol. 2. June 2006, pp. 595–599. DOI: 10.1109/ICMIT.2006.262288.

[28] Lennart C.L. Kats, Karl T. Kalleberg, and Eelco Visser. "Domain-Specific Languages for Composable Editor Plugins". In: *Electronic Notes in Theoretical Computer Science* 253.7 (2010). Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009), pp. 149–163. ISSN: 1571-0661. DOI: http://dx.doi.org/10.1016/j.entcs.2010.08.038. URL: http://www.sciencedirect.com/science/article/pii/S1571066110001179.

[29] Pierre Kelsen and Qin Ma. "Fundamental Approaches to Software Engineering: 13th International Conference, FASE 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings". In: ed. by David S. Rosenblum and Gabriele Taentzer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. Chap. A Modular Model Composition Technique, pp. 173–187. ISBN: 978-3-642-12029-9. DOI: 10.1007/978-3-642-12029-9_13. URL: http://dx.doi.org/10.1007/978-3-642-12029-9_13.

[30] M. Ko et al. "Extending UML Meta-model for Android Application". In: *Computer and Information Science (ICIS), 2012 IEEE/ACIS 11th International Conference on.* May 2012, pp. 669–674. DOI: 10.1109/ICIS.2012.48.

[31] Holger Krahn, Bernhard Rumpe, and Steven Völkel. “MontiCore: a framework for compositional development of domain specific languages”. In: *International Journal on Software Tools for Technology Transfer* 12.5 (2010), pp. 353–372. ISSN: 1433-2787. DOI: 10.1007/s10009-010-0142-1. URL: http://dx.doi.org/10.1007/s10009-010-0142-1.

[32] Max E. Kramer et al. “Extending the Palladio Component Model using Profiles and Stereotypes”. In: *Palladio Days 2012 Proceedings (appeared as technical report)*. Ed. by Steffen Becker et al. Karlsruhe Reports in Informatics ; 2012,21. Karlsruhe: KIT, Faculty of Informatics, 2012, pp. 7–15. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/documents/2350659.

[33] P. Kruchten, H. Obbink, and J. Stafford. “The Past, Present, and Future for Software Architecture”. In: *IEEE Software* 23.2 (Mar. 2006), pp. 22–30. ISSN: 0740-7459. DOI: 10.1109/MS.2006.59.

[34] Philip Langer et al. “EMF Profiles: A Lightweight Extension Approach for EMF Models”. In: *Journal of Object Technology* 11.1 (Apr. 2012), 8:1–29. ISSN: 1660-1769. DOI: 10.5381/jot.2012.11.1.a8. URL: http://www.jot.fm/contents/issue_2012_04/article8.html.

[35] Philip Langer et al. “From UML Profiles to EMF Profiles and Beyond”. In: STUFF MISSING. 2011.

[36] Sebastian Lehrig. “Applying Architectural Templates for Design-Time Scalability and Elasticity Analyses of SaaS Applications”. In: *Proceedings of the 2Nd International Workshop on Hot Topics in Cloud Service Scalability*. HotTopiCS ’14. Dublin, Ireland: ACM, 2014, 2:1–2:8. ISBN: 978-1-4503-3059-6. DOI: 10.1145/2649563.2649573. URL: http://doi.acm.org/10.1145/2649563.2649573.

[37] Bertrand Meyer. *Object-oriented software construction*. 2. ed. Upper Saddle River, NJ: Prentice Hall PTR, 1997. ISBN: 0-13-629155-4. URL: http://digitool.hbz-nrw.de:1801/webclient/DeliveryManager?pid=3853398.

[38] Brad A. Myers. “Taxonomies of visual programming and program visualization”. In: *Journal of Visual Languages and Computing* 1.1 (1990), pp. 97–123. ISSN: 1045-926X. DOI: http://dx.doi.org/10.1016/S1045-926X(05)80036-9. URL: http://www.sciencedirect.com/science/article/pii/S1045926X05800369.

[39] Larry O’Brien. *How Many Lines of Code in Windows?* last visited on August, 03, 2016. Dec. 2005. URL: http://www.knowing.net/index.php/2005/12/06/how-many-lines-of-code-in-windows/.

[40] MetaCase Oy. *The Graphical Metamodeling Example*. Version 2. 2008.

[41] Risto Pohjonen and Steven Kelly. “Interactive Television Applications using MetaEdit”. In: *Model-Driven Development Tool Implementers Forum*. 2007.

[42] Danny Poo, Derek Kiong, and Swarnalatha Ashok. “Object-Oriented Programming and Java”. In: London: Springer London, 2008. Chap. Modularity, pp. 103–117. ISBN: 978-1-84628-963-7. DOI: 10.1007/978-1-84628-963-7_8. URL: http://dx.doi.org/10.1007/978-1-84628-963-7_8.

[43] Christian Queinnec. *LISP in small pieces.* 1. publ. Includes bibliographical references. Cambridge [u.a.]: Cambridge University Press, 1996. ISBN: 0-521-56247-3; 0-521-54566-8.

[44] Ivar Refsdal. “Comparison of GMF and Graphiti based on experiences from the development of the PREDIQT tool”. Master’s Thesis. University of Oslo, 2011. URL: `http://hdl.handle.net/10852/9000`.

[45] B. R. Rimes. “A graphics-based system that supports the program understanding process”. In: *Systems Integration, 1990. Systems Integration ’90., Proceedings of the First International Conference on.* Apr. 1990, pp. 117–124. DOI: `10.1109/ICSI.1990.138671`.

[46] Davide Ruscio, Ralf Lämmel, and Alfonso Pierantonio. “Software Language Engineering: Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers”. In: ed. by Brian Malloy, Steffen Staab, and Mark Brand. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. Chap. Automated Co-evolution of GMF Editor Models, pp. 143–162. ISBN: 978-3-642-19440-5. DOI: `10.1007/978-3-642-19440-5_9`. URL: `http://dx.doi.org/10.1007/978-3-642-19440-5_9`.

[47] Melissa A. Schilling. “Toward a General Modular Systems Theory and Its Application to Interfirm Product Modularity”. In: *The Academy of Management Review* 25.2 (2000), pp. 312–334. ISSN: 03637425. URL: `http://www.jstor.org/stable/259016`.

[48] D. Schleicher et al. “Compliance scopes: Extending the BPMN 2.0 meta model to specify compliance requirements”. In: *2010 IEEE International Conference on Service-Oriented Computing and Applications (SOCA).* Dec. 2010, pp. 1–8. DOI: `10.1109/SOCA.2010.5707154`.

[49] Stephan Seifermann. “Architectural Data Flow Analysis”. In: *Proceedings of the 13th Working IEEE/IFIP Conference on Software Architecture.* WICSA’16. accepted, to appear. Venice, Italy: IEEE, 2016. URL: `http://sdqweb.ipd.kit.edu/publications/pdfs/seifermann2016c.pdf`.

[50] A. Selmeci and T. Orosz. “Modification free extension of standard software”. In: *Applied Machine Intelligence and Informatics (SAMI), 2014 IEEE 12th International Symposium on.* Jan. 2014, pp. 185–190. DOI: `10.1109/SAMI.2014.6822403`.

[51] B. Shneiderman. “Direct Manipulation: A Step Beyond Programming Languages”. In: *Computer* 16.8 (Aug. 1983), pp. 57–69. ISSN: 0018-9162. DOI: `10.1109/MC.1983.1654471`.

[52] David Canfield Smith. *Pygmalion : a computer program to model and stimulate creative thought.* Interdisciplinary systems research ; 40. Basel: Birkhaeuser, 1977. ISBN: 3-7643-0928-8.

[53] H. Stachowiak. *Allgemeine Modelltheorie.* Springer-Verlag, 1973. ISBN: 9783211811061. URL: `https://books.google.de/books?id=DK-EAAAAIAAJ`.

[54] Thomas Stahl and Markus Völter. *Model driven software development : technology, engineering, management.* Chichester [u.a.]: Wiley, 2006. ISBN: 0-470-02570-0; 978-0-470-02570-3. URL: http://swbplus.bsz-bw.de/bsz253002060cov.htm%20;%20http://bvbr.bib-bvb.de:8991/F?func=service%5C&doc%5C_library=BVB01%5C&doc%5C_number=015445749%5C&line%5C_number=0001%5C&func%5C_code=DB%5C_RECORDS%5C&service%5C_type=MEDIA.

[55] Dave Steinberg, ed. *EMF - Eclipse modeling framework.* 2. ed., revised and updated. The eclipse series. Boston, Mass.: Addison-Wesley, 2009. ISBN: 0-321-33188-5; 978-0-321-33188-5. URL: http://bvbr.bib-bvb.de:8991/F?func=service&doc_library=BVB01&doc_number=014933667&line_number=0001&func_code=DB_RECORDS&service_type=MEDIA.

[56] 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. 27–29, 2013, pp. 49–58. URL: http://www.kieker-palladio-days.org/.

[57] Michael Unterstein. *Relationale Datenbanken und SQL in Theorie und Praxis.* Ed. by Günter Matthiessen. 5. Aufl. 2012. eXamen.pressSpringerLink : Bücher. Berlin, Heidelberg: Springer, 2012. ISBN: 978-3-642-28986-6. URL: http://swbplus.bsz-bw.de/bsz376064560cov.htmhttp://dx.doi.org/10.1007/978-3-642-28986-6.

[58] Markus Voelter. "Generative and Transformational Techniques in Software Engineering IV: International Summer School, GTTSE 2011, Braga, Portugal, July 3-9, 2011. Revised Papers". In: ed. by Ralf Lämmel, João Saraiva, and Joost Visser. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. Chap. Language and IDE Modularization and Composition with MPS, pp. 383–430. ISBN: 978-3-642-35992-7. DOI: 10.1007/978-3-642-35992-7_11. URL: http://dx.doi.org/10.1007/978-3-642-35992-7_11.

[59] Ingo Weisemöller and Andy Schürr. "Model Driven Engineering Languages and Systems: 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings". In: ed. by Krzysztof Czarnecki et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. Chap. Formal Definition of MOF 2.0 Metamodel Components and Composition, pp. 386–400. ISBN: 978-3-540-87875-9. DOI: 10.1007/978-3-540-87875-9_28. URL: http://dx.doi.org/10.1007/978-3-540-87875-9_28.

[60] Srđan Živković and Dimitris Karagiannis. "Enterprise, Business-Process and Information Systems Modeling: 16th International Conference, BPMDS 2015, 20th International Conference, EMMSAD 2015, Held at CAiSE 2015, Stockholm, Sweden, June 8-9, 2015, Proceedings". In: ed. by Khaled Gaaloul et al. Cham: Springer International Publishing, 2015. Chap. Towards Metamodelling-In-The-Large: Interface-Based Composition for Modular Metamodel Development, pp. 413–428. ISBN: 978-3-319-19237-6. DOI: 10.1007/978-3-319-19237-6_26. URL: http://dx.doi.org/10.1007/978-3-319-19237-6_26.