TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Software and Multimedia Technolog, Software Technology Group

Master Thesis

Towards a Modular Product Line of Graphical Editors

Kevin Ivo Kassin
Born on: 14th August 1992 in Cottbus
Matriculation number: 3848351
Matriculation year: 2012

to achieve the academic degree

Master of Science (M.Sc.)

Supervisor
Dr.-Ing. Thomas Kühn
Supervising professor
Prof. Dr. rer. nat. habil. Uwe Aßmann

Submitted on: 7th August 2018

Statement of authorship

I hereby certify that I have authored this Master Thesis entitled *Towards a Modular Product Line of Graphical Editors* independently and without undue assistance from third parties. No other than the resources and references indicated in this thesis have been used. I have marked both literal and accordingly adopted quotations as such. There were no additional persons involved in the intellectual preparation of the present thesis. I am aware that violations of this declaration may lead to subsequent withdrawal of the degree.

Dresden, 7th August 2018

Kevin Ivo Kassin

Abstract

This thesis addresses designing Product Lines (PLs) of Graphical Editors (GEs). It provides a feasible top-down design approach specialized on such Graphical Editor Product Lines (GEPLs), which can be configured dynamically. Furthermore, the end product's features are implemented modular, which has numerous positive effects on the development and maintenance processes for the family. These effects reach from decreasing the complexity of big PLs, allowing to delegate split up development tasks onto multiple isolated working teams, easier debugging and flexibility to extend or specialize a family of products as well as being able to use functionalities developed by third-party vendors. While design methods avoiding monolithic architectures and implementations exist for many PL domains, there are none known for GEPLs. Yet, the domain of those offers many challenges as GEPLs are actually comprised of Software Product Lines (SPLs) and Language Product Lines (LPLs), which is a combination untackled by any modular design approach known to me. Additionally, products in the domain require to implement multiple distinct and specific concerns, leading to artifacts which differ significantly but have to be located and managed in a single component. Overall, this justifies the need for specialized design approaches for the GEPL domain. In regard to this need, this thesis gives an overview of the existing landscape of approaches to design PLs, analyzing solutions offered by other researchers. Furthermore, a requirement analysis for the GEPL domain is conducted. Its results are the foundation for the presentation of a top-down design approach for dynamically configurable GEPLs, which are implemented feature modularly. Finally, a case study documenting the development of such a family of GEs is providing a proof of its feasibility.

Acknowledgment

First of all, I want to thank my family and friends for supporting me in the turbulent time I worked on this thesis. Furthermore, my special thanks must go to Thomas Kühn, not only for contributing papers full of scientific knowledge, which is integral to this thesis but especially for being a patient and understanding supervisor since my bachelor thesis. I also want to thank Uwe Aßmann for being a teacher throughout my studies and offering me opportunities, which allow me to learn and develop. Finally, I want to thank Christian Deussen for our productive cooperation and his contribution to the case study in this thesis.

Contents

1 Introduction

1.1 Motivation

Product Lines (PLs) are widely used to reduce the financial costs and development time of software systems. These systems can be found in the form of classical software applications. In this domain many approaches for Software Product Lines (SPLs) [C. Kang et al., 1990, Kästner and Apel, 2013] exist. Furthermore, PLs can be used to develop systems in other domains. Examples of these are Language Product Lines (LPLs) in general [Kühn and Cazzola, 2016] and for specific purposes, e.g. LPLs for model transformations [Sánchez Cuadrado, 2012] or Model Driven Development (MDD) [Evans et al., 2003]. A brief overview of approaches classified by domain, modularity, configuration and design method can be found in Figure 1.1.

PLs usually achieve their goals by enabling the creation of new products in a fast and easy manner based on already existing artifacts and their composition. This works because PLs increase the reusability of artifacts possibly up to a point of *dynamic* **configuration** changes. This means that the products feature selection and thus the product variant can be changed at runtime.

In addition, **feature modularity** is achieved, if there is a way to remove or add features of a product easily. In this context *easily* means that this step can be automated and does not need to be executed by hand. To achieve this, there is a need for well-defined modules. They enable modularity by exactly describing what changes to specific artifacts of an application or language has to be done when adding or removing features from them. For example, Shaker *et al.* [Shaker et al., 2012] use behavior models to encapsulate a feature's own operations as well as its dependencies to other features.

Another interesting aspect of approaches to create SPLs and LPLs is their **design method**. Overall there are two distinct methods, the *top-down* and the *bottom-up* approach. Kühn *et al.* [Kühn and Cazzola, 2016] summarize both for LPLs in the following way: *Top-down* is defined, such as a feature model and compiler artifacts have to be created. Besides that, a mapping between the feature model and the artifacts is needed to configure the language compiler. The *bottom-up* approach generates a feature model and composition rules by analyzing individual components of a language. This results in a compiler for the LPL. In this thesis, these descriptions are generalized for SPLs by replacing the language compiler with the configured software application.

Apart from the previously mentioned domains in which PLs can be useful, the main focus of this thesis will be the domain of Graphical Editors (GEs). PLs in this domain, GEPLs, are not easy to create. Generic approaches for SPLs could be used, but are not optimized for this application. One challenge is that known frameworks for the development of graphical editors, such as *Graphical Editing Framework* (GEF)[1], Graphiti[2], Sirius [Viyović et al., 2014], DiaGen [Minas and Viehstaedt,

[1]URL: `https://www.eclipse.org/gef/`, last visited: 5.12.2017.

[2]URL: `https://www.eclipse.org/graphiti/`, last visited: 5.12.2017.

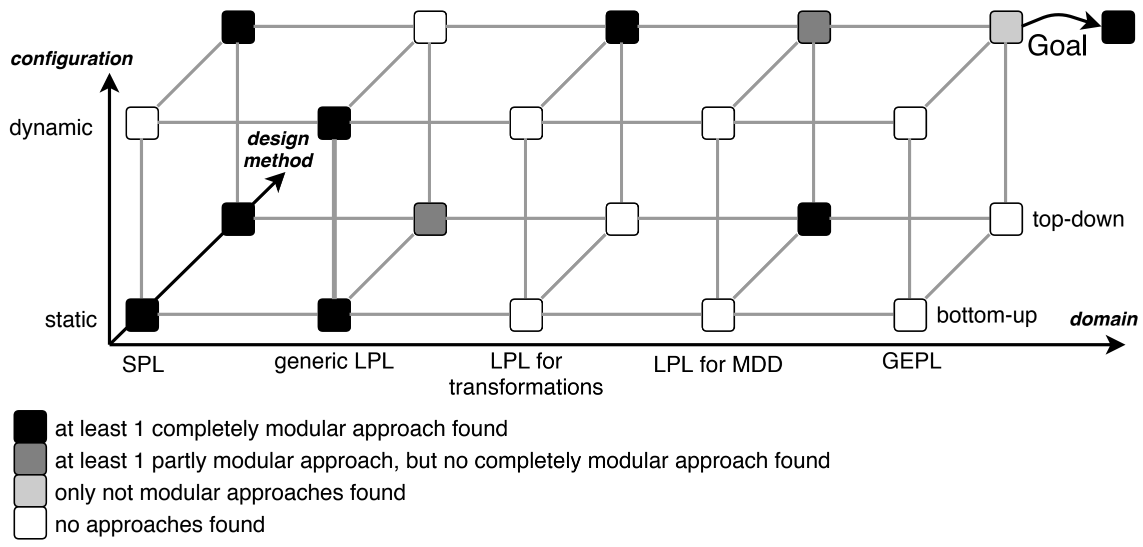

configuration
dynamic
static
design method
Goal
top-down
bottom-up
domain
SPL
generic LPL
LPL for transformations
LPL for MDD
GEPL
at least 1 completely modular approach found
at least 1 partly modular approach, but no completely modular approach found
only not modular approaches found
no approaches found

Figure 1.1: A brief overview of the paper coverage for approaches to create product lines in different domains with different properties.

1995] and MetaEdit+[3], do not support SPLs natively. In addition, as a GE is a specific and complex software application, there are many aspects of it that have to be changed when a feature is added to or removed from a GEPL. In a feature modular approach, these aspects would be encapsulated in a single feature module. Managing all these aspects for a feature is not trivial, which makes the creation of modular GEPLs challenging. Meanwhile, this challenge could be simplified if an approach dedicated to it is applied.

In the next two paragraphs, seven distinctive application concerns to encapsulate in a feature module will be explained. Firstly, there are a feature's *palette properties* and the *representation of its graphical objects*. While the palette entries' visibility and appearance are easy to manage, the graphical representation and the possible operations on it[4] are more complicated to capture. However, the *edit policies* and *model transformations* are hard to encapsulate too. Edit policies control which actions are executable with respect to the current configuration of the GEs. Model transformations are needed if the GE works with multiple metamodels. In [Kühn, 2017] such a situation is presented. One metamodel is tailored to be an intermediate model between a graphical representation and a domain model. To get to the domain model representation, a model transformation between the intermediate model and the domain model is needed.

Furthermore, the *feature model* needs to be modularized. This means there should be a safe automated method to add or remove features from a feature model. While an implementation of this might not be trivial there already exist solutions to this task. An example of such can be found in [Bagheri et al., 2011]. Finally, there are two more models to apply to changes to when adding or removing feature modules. These are the metamodels that are used to represent source and target models for the transformation. Just as for the feature model there already are solutions for such cases. Two examples are the approaches of Delta Ecore [Seidl et al., 2014] and the family of metamodel languages by Kühn *el al.* [Kühn et al., 2014]. Therefore this thesis will not focus on the modularization of the feature model, as well on the source and target metamodel.

These seven distinctive concerns of GEs can be realized by completely different approaches. For example, while the palette appearance and the graphical representation could be written in Java code, the edit policies might work with an own metamodel based on boolean statements. The feature model, as well as the source and target metamodels, are also defined in its own model

[3]URL: `http://www.metacase.com/products.html`, last visited: 18.12.2017.
[4]The CRUD operations (Create, Read, Update, Delete) [Truică et al., 2013] are a well known examples for such.

language. Finally, the model transformations could be coded in a specialized Domain Specific Language (DSL) for transformations.

An approach for the design of GEPLs can be found in [Kühn, 2017], as Kühn describes a family of role-based languages and a corresponding GE. You can find an implementation of the editor[5] as well. While it already is a dynamic GEPL, it lacks modularity. The reason for this is the monolithic implementation of the editors' visual representation and user interaction. The artifacts of the different application concerns are often scattered around the application and not encapsulated with respect to a specific feature.

1.2 Problem Definition

As one can see in Figure 1.1 the paper coverage of approaches to build SPLs and generic LPLs is quite good. For specific purposes of LPLs, the coverage is still acceptable. Meanwhile, there are no well known complete or partly modular approaches to create PLs in the domain of graphical representation languages and editors.

There are multiple application concerns that need to be changed when adding or removing features from a GE: A feature's *palette properties*, the *representation of its graphical objects*, its *edit policies*, *model transformation rules* for it and finally the changes to the GE's three *models*. In addition, a developer of a GEPL has to overcome the challenge to encapsulate this concerns using artifacts in different languages and metamodels for each aspect of a feature module. These special conditions cause a situation in which a modular approach tailored to the development of GEPLs is useful.

Therfore, this thesis will present such an approach and showcase it by modularizing an existing SPL of GEs. More specificly this thesis aims at describing a general development approach to establish modular PLs for graphical editors and their graphic representation languages. This GEPL should be dynamically configurable. Furthermore, the methodology presented in this thesis will focus on a top-down design method, but the possibility of a bottom-up adaption will be discussed too. One can see the properties of the new methodology in Figure 1.1 marked with the notation *Goal*. To showcase this approach, a modular reimplementation of the *Full-fledged Role Modeling Editor* (FRaMED) SPL [Kühn, 2017] will be used as a case study. To reduce the complexity of the reimplementation as much as possible, existing code should be reused. Furthermore, a part of this work, namely the implementation of the edit policies, will be performed by Christian Deussen.

In short, this thesis will make the following contributions to elaborate a modular design approach tailored to graphical editors:

- A survey on the literature about SPLs and LPLs
- A requirement analysis for the domain of GEPLs
- A presentation on a general top-down design methodology for dynamic and modular GEPLs
- An evaluation of that methodology
- A case study on that methodology by modularizing an existing family of graphical editors

1.3 Outline

The structure of this thesis will follow along with the steps to achieve the goal of it. Chapter 2 will survey the literature on SPL and LPL design to see if there are already compelling ways to create modular PLs and which techniques they use. Furthermore, the requirements for the domain of GEPLs will be analyzed in Chapter 3. Following in chapter 4 the top-down design methodology for dynamic and modular GEPLs will be proposed. An evaluation of this approach, using the

[5]URL: `https://github.com/leondart/FRaMED/tree/develop_branch`, last visited: 29.11.2017.

requirements found in chapter 3 can be found in this chapter (4.3) too. Chapter 5 will showcase the applicability of the methodology with a case study, reimplementing the FRaMED SPL. Finally, chapter 6 will conclude the thesis by summarizing its contributions and give an outlook for possible future works towards simplifying the development of GEPLs.

1.4 Terminology

In this section terms, concepts and their usage will be explained.

Software System Software systems are all groups of cooperating software artifacts to provide specific functionalities. This includes applications, transformation systems and compilers for languages, for example, these of DSLs.

Graphical Editor A GE, how it is specified in this thesis, is an application which enables the user to create models in a graphical manner. This means that the user can interact with a *palette* and a canvas to create *model objects with a graphical representation*. Via the *edit policies*, the GE can decide which user initiated interactions are allowed depending on its current state and the interaction context. Furthermore, it has to be able to *transform the user created models* based on an *abstract syntax model* instance to a different one depending on a *target metamodel*.

Graphical Editor Product Line A GEPL is a PL used to create multiple GEs with a different behavior and sets of features. Given the nature of GEs, this clearly identifies as an SPL using a *feature model*. In addition, according to this thesis' specification of GEs, a GEPL also is an LPL. The languages, this applies to, are defined by the abstract syntax model, target metamodel and the model transformation language.

Feature Modularity Pohl *et al.* describe feature modularity as a goal of the feature-oriented programming in a way that it should "[allow] to compose objects from individual features or abstract subclasses in a fully flexible and modular way" [Pohl et al., 2005]. Kästner *et al.* raise the question "how to arrange a code base such that features become explicit and composable" in this matter [Kästner et al., 2011]. This composable feature modules "eases system development and evolution because features can be developed in isolation, in parallel, and by third-party vendors" [Shaker et al., 2012]. Furthermore, Shaker *et al.* claim that "it is easier to understand a new feature in terms of its incremental changes to existing features". Feature Modularity of a software system must enable *System Extension* and *System Specialization*. To make this possible, there are two principles of feature modularity, established by Kästner *et al.* [Kästner et al., 2011], to follow. The first one is *locality and cohesion*, which states that artifacts addressing one feature are placed in a container unit in one location. Besides this, there is also the *encapsulation and information hiding* principle, which requires to differ between internal implementation details and external parts of components. In this thesis, the principle is interpreted that for every part of a component's artifact, it should be explicitly clear if it is publicly accessible to the outside of a feature module. In this thesis *modular* will be used synonymously to *feature modular*.

System Extension A software system is extended by adding feature modules to it. The new feature modules extend the functionality of the systems code base.

System Specialization Specialization of a software system is executed by removing its feature modules. That way the system loses specific functionalities and is less general usable.

2 Survey on Software and Language Product Line Design

The following chapter addresses the existing landscape of approaches to design SPLs and LPLs. Overall it presents a survey of 15 design methodologies, which are classified by four properties. The classification scheme is elaborated in the first Section 2.1 of this chapter. Following that, I will sum up the process of the approaches and reason about their classification. This procedure allows comparing the related work of this thesis. The overview of all surveyed papers can be found in Section 2.2. Furthermore, it results in a brief overview of the options to design SPLs and LPLs, which emphasizes the coverage of design approaches in respect to the classification properties. This means that gaps and uncovered areas in the spanned space of the classification can be found. The results of the survey are presented in the corresponding Section 2.3. Finally, the classification scheme of this survey will also be used to categorize the design approach presented in Chapter 4. A comparison of the new design methodology for PLs to the already established ones can be realized based upon the survey.

2.1 Classification Scheme

2.1.1 Domain

The first classification property addresses the domain in which a design approach is meant to be apllied in. An overview of the whole classification scheme and its visualization can be seen in the *Figure* 2.1. The domains present in this survey are plotted on the horizontal axis. While in the overview there is only a placeholder referenced, the concrete visualization of the survey, which can be found in the following sections, lists five domains. Some of those domains describe special cases of another one. However, it is not possible to define a clear order of more and more specific domains. In the following I will present all five concerned domains in this survey. The relations between them will be part of this elaboration.

Software Product Line An SPL defines a family of software systems. Such families can be software languages and applications. Software languages are further elaborated in the next paragraph. A software application can be defined as an executable program with a predetermined use case, which produces, visualizes or processes data and user inputs. The family members can be concrete applications depending on a chosen feature configuration. The features in the configuration are mapped to artifacts implementing the features' application aspects. The configuration, artifacts, and the mapping are used by a composition process. It combines, enables or disables multiple artifacts to create one executable application with a set of features corresponding to the current feature configuration. It is important to note that to approaches suitable for the creation of both,

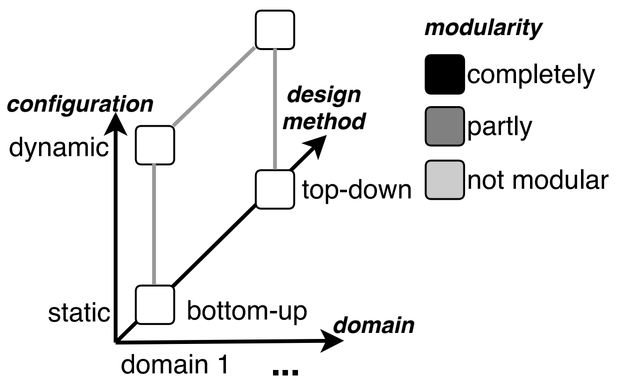


Figure 2.1: Overview on the classification scheme of the survey.

families of applications and languages, this domain will be assigned. This avoids labeling an approach as *SPL* and *generic LPL* at the same time, while it is still possible to mark down such that only allow designing families of software languages.

Generic LPL The family members of a Language Product Line are software languages without a specifically mentioned purpose. Usually, such languages are defined by either grammars or by metamodels [Kleppe, 2007] and offer syntax as well as semantic rules. Software languages have a broad variety of applications. They can be used to describe models, programs, artifacts with a markup representation and more [Kleppe, 2007]. Similar to the application kind of SPLs, parts of the software language are captured in single artifacts, which can be composed to a language compiler according to a feature configuration.

LPL for Transformations This domain describes specific cases of generic LPLs. It contains all those software languages with the specific purpose to implement model-to-model transformations. Such an implementation defines how elements of a source model are represented equally in a target model. Done for all possible model elements, this enables a controlled transition between two models. Multiple different representations of one subject are possible this way, while its only needed to initially create one representation of it.

LPL for MDD In this domain you can find software languages which are meant to be used for Model Driven Development processes. When developing an application model driven, there are multiple tasks to solve, which involves languages. Examples for such are the definition of Platform Independent Models (PIMs) and Platform Specific Models (PSMs), the transition of PIMs to PSMs, the verification of models and finally code generation [Parviainen et al., 2009].

GEPL A Graphical Editor Product Line is a family of GEs. Following the definition of a GEPL in the Section *Terminology* in the first chapter, it is a combination of an SPL and multiple LPLs. The editor implementation itself is an SPL, while the source and target models of the transformation need to be defined as an LPL. Usually, both PLs are dependent on the same feature configuration. For more details on the GEPL domain, see the requirement analysis of it in the corresponding Chapter 3.

2.1.2 Configuration

This classification property addresses when a Product Line can be configured to match an user's needs. In the visualization of the survey (*Figure* 2.1) it is drawn onto the vertical axis. There are two options for it. They differ in the fact of being able to alter the feature configuration during runtime or not. If this is not possible, the configuration is classified as *static*. A *dynamic* configuration allows to change the feature configuration while a PL's application is executed. By the nature of this definition, it is clear that a dynamic configuration editor can also easily be used to configure a PL outside of its application's runtime. The configuration editor needs to be executable independently from the rest of the application for this. Additionally, made changes has to be saved and taken into account when starting the PL's application.

2.1.3 Design Method

The classification property *Design Method* describes in which manner the design of a PL is executed and which parts of it have to be created manually. It is represented by the depth axis in *Figure* 2.1. Overall, there are three relevant aspects of PLs to look at in the context of the design method. Firstly, the feature model defining a structure of multiple PL features. Secondly, the artifacts implementing such defined features and lastly the mapping connecting both of the before mentioned parts of a PL. There are two distinct methods: The *top-down* and the *bottom-up* method. The following explanations for both is a generalization of the considerations by Kühn *et al.* [Kühn and Cazzola, 2016] about design methods for LPLs. Generalizing his elaboration in a reasonable manner allows us to apply them on design methods for more general PLs.

Using the top-down method a developer has to create all three mentioned parts of a PL manually. While this is a big effort, it also allows developing new PLs from scratch with a maximum of flexibility. This method follows the classic *waterfall model*[1] by starting at the conception of the applications' possible features. Beginning from the top of the waterfall model, one will reach the implementation of the development product by following the model down. Continuing with the bottom-up method, the opposite approach to design a PL is starting by analyzing an already build application or multiple artifacts. That way, only the features' implementations have to be provided, while the feature model, as well as the mapping between it and the artifacts, are generated automatically. Of course, the analysis does not come for free and mechanisms like a comparison process or annotations for artifacts are needed to support it. Furthermore it is not possible to develop an application as flexible and intuitive using the bottom-up design method. The reason for that is clear when realizing that the method starts from the bottom of the waterfall model, using a prebuild application or its parts to generate its conceptual parts in form of a feature model. While developing a PL with a bottom-up process from scratch is possible, implementing components without the conception a certain set of components can be perceived as not intuitive. Therefore implementation of a completely new PL using a bottom-up process will not be regarded in the further thesis.

2.1.4 Modularity

The last of the four properties is visualized by the gray shade of a point in the grid of the other three classification properties. It deals with the possibility of using the evaluated design approach to produce a feature modular PL. For such a PL it is possible to remove, add and replace features in an automated manner. There is no need to change the code of artifacts outside of the new feature's ones by hand. Adding a feature leads to an *extension* while removing one qualifies as a *specialization* of a PL. When replacing a feature's implementation the feature set of the PL does not change, only the artifacts implementing it. Additionally, this case is a combination of removing and adding features. Therefore, it follows from the abilities feature extension and specialization and will not be part of the definition of modularity levels directly. There are three indicators for modularity. Firstly the two principles of feature modularity, defined by Kästner [Kästner et al., 2011], should be fulfilled. This means that the artifacts should be collected in a comprising unit or location and their inner implementation should be differentiated from their external interface. Finally, it is important that dependencies between components can be derived and automatically handled. This enables system specialization since unresolvable dependencies can be avoided this way.

[1]One version of it can be seen in [Bullinger et al., 2003].

Overall I differ between three levels of feature modularity. The first one can be assigned to an approach which can be used to create completely modular PLs, capturing application extension and specialization. Secondly, there can also be partly modular PLs. Those can be either extended or specialized, but not both. Finally, the products of an approach might not have any of those abilities. While approaches able to create completely modular PLs are marked as black grid point, the gray shade get lighter towards the level of where approaches are not suitable to generate modular PLs.

2.2 Overview

Overall this survey evaluates 15 papers which address the design process of PLs. The papers are spread over five different domains which each capture the design of a specific type of PLs. All of the surveyed papers, listed in *Table* 2.1, are published between the years 2002 and 2017. In the mentioned table, the citation, title and authors can be found for each entry. Additionally, each of them also got an associated identifier. Finally, the four most right columns represent the classification of each surveyed paper. The classification for them is also visualized by the *Figure* 2.2. In the diagram, the evaluated work is referenced using their beforehand established identifiers. In the following, I will briefly discuss the process and the classification all 15 papers of this survey. By doing that, the content of *Table* 2.1 and the *Diagram* 2.2 is elaborated in the process.

Component-Based Product Line Engineering with UML

Firstly, I want to give an overview on the book *Component-based Product Line Engineering with UML* [Atkinson et al., 2002] by Atkinson *et al.*, corresponding to the entry with the *ID* 1 in the *Table* 2.1. It was released in 2002 and is thematically accompanied by two papers [Atkinson et al., 2000a, Atkinson et al., 2000b]. The book presents an approach with the goal to design SPLs using the *Unified Modeling Language* while focusing on a modular design of it. The approach is called *KobrA*(*Komponentenbasierte Anwendungsentwicklung*, translated *component-based application development*).

It elaborates how SPL components have plugs, which provide services, and socket, which define required services. If a socket and a plug are connected there is a contract model managing the interaction between the corresponding components. This establishes *contract relationships*. A *KobrA* component is described by five model types on two levels. On the *specification level*, a behavioral, structural and functional model can be found. The first two mentioned models are described by a UML state chart and class diagram, while the last model the functions via operation schemata. On the second level, the *realization level*, the structural model can be found again, together with an interaction and execution model. The interactions are defined by a UML collaboration model and a UML activity diagram describes the executions model of the component. Between all those models there can be relationships. Such are either *refinements*, f.e. between the two structural models on different levels, or *consistency relationships* between models on the same level.

To build an SPL with the approach, every component further defines a decision model to describe the feature choices on which it is dependent on. Overall a fix framework of components, described by multiple models, is created and used to instantiate members of the family. The instantiation is a composition of multiple components based upon the feature configuration and the components decision models. Dependencies between components are also handled by the framework. It organizes the components in a tree structure of *creation relationships*, meaning that the parent component in the tree creates its children elements. The components itself can be nested as multiple components plugged together can be treated as a new component. Unused plugs and sockets of the inner components are the interfaces of the overall component in such a case.

To classify the approach I will address every one of the properties in the order in which they are presented in *Table* 2.1. Is it clear to see that the book addresses the design of **SPLs** as it

ID	Citation	Title	Authors	Domain	Config-uration	Design Method	Modu-larity
1	[Atkinson et al., 2002]	Component-Based Product Line Engineering with the UML	Atkinson, Colin & Bayer, Joachim & Bunse, Christian *et al.*	SPL	dynamic	top-down	complete
2	[Svendsen et al., 2010]	Developing a Software Product Line for Train Control: A Case Study of CVL	Svendsen, Andreas & Zhang, Xiaorui & Lind-Tviberg, Roy *et al.*	SPL	static	top-down bottom-up	partly complete
3	[Font et al., 2015]	Building Software Product Lines from Conceptualized Model Patterns	Font, Jaime & Arcega, Lorena & Haugen, Øystein *et al.*	SPL	static	bottom-up	complete
4	[Mazo et al., 2015]	VariaMos: An Extensible Tool for Engineering (Dynamic) Product Lines	Mazo, Raúl & Muñoz-Fernández, Juan C. & Rincón, Luisa *et al.*	SPL	dynamic	top-down	none
5	[Seidl et al., 2014]	DeltaEcore – A Model-Based Delta Language Generation Framework	Seidl, Christoph & Schaefer, Ina & Aßmann, Uwe	SPL	static	top-down	partly
6	[Shaker et al., 2012]	A Feature-Oriented Requirements Modelling Language (FORML)	Shaker, Pourya & Atlee, Joanne M. & Wang, Shige	SPL	dynamic	top-down	complete
7	[Batory et al., 2002]	Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study	Batory, Don & Johnson, Clay & MacDonald, Bob *et al.*	SPL	static	top-down	complete
8	[Cuadrado and Molina, 2009]	A Model-Based Approach to Families of Embedded Domain-Specific Language	Cuadrado, Jesús Sánchez & Molina, Jesús Garcıá	generic LPL	static	top-down	partly
9	[Vacchi et al., 2014]	Automating Variability Model Inference for Component-Based Language Implementations	Vacchi, Edoardo & Cazzola, Walter & Combemale, Benoit *et al.*	generic LPL	dynamic	bottom-up	complete
10	[Kühn and Cazzola, 2016]	Apples and Oranges: Comparing Top-Down and Bottom-Up Language Product Lines	Kühn, Thomas & Cazzola, Walter	generic LPL	static	top-down bottom-up	partly complete
11	[Vacchi et al., 2013]	Variability Support in Domain-Specific Language Development	Vacchi, Edoardo & Cazzola, Walter & Pillay, Surresh *et al.*	generic LPL	dynamic	bottom-up	complete
12	[Sánchez Cuadrado, 2012]	Towards a Family of Model Transformation Languages	Cuadrado, Jesús Sánchez	LPL for transformations	dynamic	top-down	partly
13	[Voelter and Groher, 2007]	Handling Variability in Model Transformations and Generators	Voelter, Markus & Groher, Iris	LPL for transformations	dynamic	top-down	complete
14	[Evans et al., 2003]	Building Families of Languages for Model-Driven System Development	Evans, Andy & Maskeri, Girish & Sammut, Paul *et al.*	LPL for MDD	static dynamic	top-down top-down	complete partly
15	[Kühn, 2017]	A Family of Role-Based Languages	Kühn, Thomas	GEPL	dynamic	top-down	none

Table 2.1: List of the 15 surveyed papers

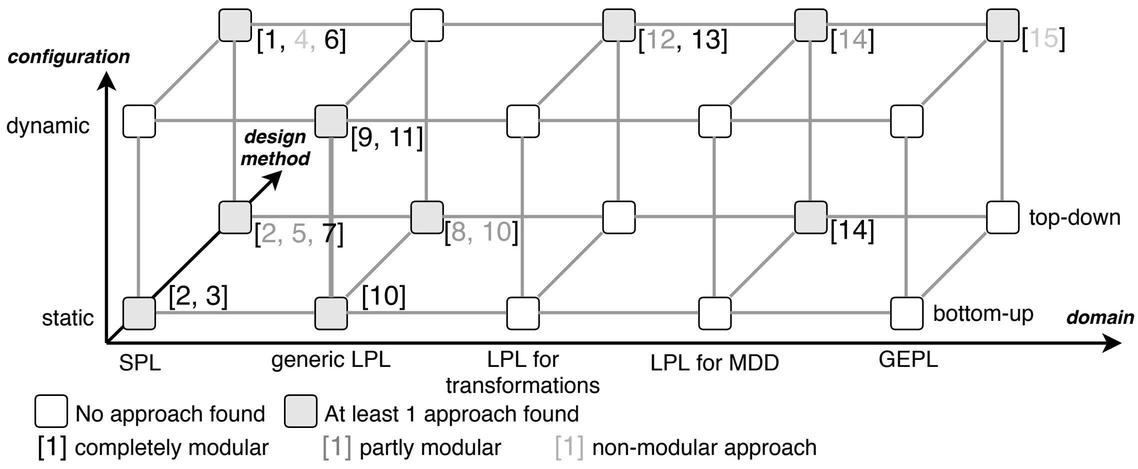


Figure 2.2: Classification of the 15 surveyed papers in a grid representation

does describe the definition of components which can create an executable application. It does further not limit the use case of the designed product. I classify the design methodology as one which can be configured **dynamically**. To change the configuration, the new configuration has to be checked against the decision models of all components in the families framework. This process decides which components are active in the following. Such calculations can easily be implemented dynamically and executed during runtime. As the components' decision models[2] with a belonging feature model has to be developed manually, this design method is **top-down**. Finally, it also follows the two principles of feature modularity by Kästner [Kästner et al., 2011].[3] It achieves locality as well as encapsulation by using components which offer plug and sockets. This way the inner implementation and outer interfaces are separated. In cooperation with the SPL framework's creation tree, all dependencies between components can be managed automatically. Therefore the approach is **completely modular**.

Developing a Software Product Line for Train Control: A Case Study of CVL

Svendsen *et al.* published the paper *Developing a Software Product Line for Train Control: A Case Study of CVL* [Svendsen et al., 2010] in 2010. It is referenced by the *ID* 2 in *Table* 2.1. The paper presents a case study on the usage of CVL (*Common Variability Language*) on a DSL to generate SPLs.

The DSL in the case study is used to describe train stations and signals. The description is used to generate code that controls the procedures and signals at those train stations. This represents MDD of fully executable artifacts with interlocked and reused code. CVL works with model artifacts, on the product realization layer, and features, on a different specification layer, to define variability for models. More precisely it contains how a model can vary from a given base model. Using CVL in the case study, parts of a train station can be replaced, added or removed in an automated process to generate new variations of such stations. The signal controlling code of similar products is carried over.

Both, a bottom-up and a top-down method are presented in the paper. In the bottom-up version of it, the commonalities and differences between multiple models are analyzed to detect atomic parts of the models. One of the compared models then is chosen as the base model. In contrast, the developer builds the atomic parts and a base model using the parts manually in the top-down approach. In the following, the base model is can be altered using the atomic parts with respect

[2]The decision models effectively represent the mapping of a component to the feature model.
[3]See the *Terminology* section 1.4 on feature modularity for an elaboration.

to the variability model defined with CVL. The generated models can be verified and validated automatically to conclude the process.

The whole process is classified as a design approach for **SPLs** as a code generation is based upon the set of different models. Therefore a family of executable applications is created by exploiting variability in models. As this is a model-based approach, I classify it as one with only **static** configuration. This classification applies to every approach which executes changes on models to create new products of a PL, except the described approach explicitly presents how this process is executed during runtime. However, none of the papers in this survey presented such an implementation. As already elaborated, Svendsen *et al.* used both design methods, **top-down** and **bottom-up**. Using the bottom-up design methods, the approach is **completely modular**. An SPL can automatically expand or specialize by adding or removing a new atomic model element. It just needs to be added or deleted from the compared models and the comparison process is repeated, given all compared models are still valid. Dependencies between atomic model elements are taken into account that way, while the principle of locality is ensured by the nature of atomic elements. Only **partly modular** is the design approach for the top-down method. While the locality can be realized here too, the dependencies between atomic model elements cannot be managed automatically. When removing a manually developed model element component, it can happen that another component, which is dependent on the removed one, cannot be used anymore. This dependency is not captured or avoided by Svendsen *et al.* in this case. Therefore only the SPL extension can be executed in a safe matter, but not the specialization.

Building Software Product Lines from Conceptualized Model Patterns

The paper *Building Software Product Lines from Conceptualized Model Patterns* [Font et al., 2015] with the *ID* 3 is written by Font *et al.* and published in 2015. It presents a bottom-up approach to design SPLs with a focus on model products. A rather unique property of the approach is the direct participation of the human in the component extraction process.

Similar to the method of the second approach by Svendsen *et al.*, a set of models is analyzed to find commonalities and variabilities of them. The result, amongst others, is a set of model patterns which vary from model to model and are part of the generated variability model in the following. Humans provide domain knowledge during this process by narrowing the input set of models and specifying which metamodel element's instances should always or never be part of the resulting model fragments. The SPL engineer can also check the automatically generated model patterns, exclude them from the resulting set or restart the process with an adjusted input set of models. Therefore the bottom-up process is not linear, but rather a loop. This way it can be ensured that the generated fragments are useful for the next step, which the model patterns are used to build new models out of them and creating a PL.

The domain classification of this approach is not unambiguous. It presents a way to design model product lines. While the design methodology of Svensen *et al.* [Svendsen et al., 2010] with the *ID* 2 specifies that edited models are used to generate code, this approach is more general about the use of it. I classify it as an approach associated with the **SPL** domain as models often are the base of LPLs and SPLs. This follows the classification idea presented in the Subsection 2.1.1. As an approach using model manipulation every time the feature configuration is changed, the configuration is classified as **static**. Besides this, the approach clearly uses a **bottom-up** design method, when comparing a set of models to determine variabilities of them. The bottom-up method also is part of the reason the approach is marked as **completely modular**. New model patterns can be easily taken into account by widening the input set of the whole process. When removing a pattern on which another one dependents, some models of the input set are no longer valid. Therefore the process can not start using the input models that contains the dependent pattern avoiding to create a fragment associated to the pattern. This way such dependencies are handled effectively. The approach also follows the principle of locality as a model pattern can be seen equivalent to the atomic model elements of the before mentioned approach of Svendsen *et. al* in this term.

VariaMos: An Extensible Tool for Engineering (Dynamic) Product Lines

Mazo *et al.* presents in *VariaMos: An Extensible Tool for Engineering (Dynamic) Product Lines* [Mazo et al., 2015], published in 2015, a tool assistant approach of using variability models to create dynamic PLs. His contribution is identified by the *ID* 4.

The paper of Mazo *et al.* describes an extensible tool that allows to define, verify, analyze, configure and simulate variability models. The language to represent such feature models can be created by using the tool too. More precisely this means that user can define an own DSL for variability models using the tool *VariaMos*. The instances of the variability models, the configurations, can be verified by validity checks taking feature model constraints into account. Furthermore, configuration can also be analyzed and simulated to iterate over the valid solutions of a partial configuration, propose alternatives and also visualize them. Overall it offers a broad support for feature modeling.

That said, it is also important to emphasize that *VariaMos* does only help to model decisions which components should be composed to build a member of a PL. It does not offer a solution on the composition itself. I still decided to take the tool into account for this survey as it follows the concept of feature-oriented PLs based upon [C. Kang et al., 1990], like all the other papers in this survey. Furthermore, three of the four classification properties can be determined similarly to the other surveyed papers.

Following the consideration about the domain classification in the Subsection 2.1.1, I assign the approach, which can be used to design families of languages and applications, to the **SPL** domain. By writing a fitting configuration editor it is easily possible to edit a configuration of a feature model created with *VariaMos* **dynamically**. As the tool encourages the user to define an own variability model language and feature model from scratch, I classify the *VariaMos* approach as a **top-down** design method. Not as easy to determine as the three classification properties before, is the modularity characteristic. It strongly dependends on the chosen composition concept or implementation. The examined paper does not define such a specification. Consequently, the *VariaMos* approach does not offer any modularity mechanism by itself and is marked with **no modularity** in the survey table. In combination with a modular design methodology using feature models this flaw could be avoided.

DeltaEcore – A Model-Based Delta Language Generation Framework

The paper *DeltaEcore – A Model-Based Delta Language Generation Framework* [Seidl et al., 2014] was published by Seidl *et al.* in 2014. Its *ID* reference in *Table* 2.1 is 5. It showcases a framework for *delta modeling*, a way to alter models dependent on a feature configuration.

Delta modeling allows transforming a valid model variant to another valid version of it in a feature configuration controlled manner. The changes applied to a model variant can add, remove, replace or modify elements of the source model variant. Some of these *delta operations* are generic, for example creating or referencing a model element, and others are language specific. Examples for language specific delta operations are set, unset or modifying specific properties of a model element. The generic operations are implemented on the metamodel associated with the model to change the variants of. Meanwhile, *delta dialects* can be defined to capture the language specific delta operations. In the following, these operations can be used to define *delta modules*. These modules usually define all the changes to a model variant for exactly one associated component. Depending on a feature configuration the component is selected or eliminated from the PL. Every time the status of the component changes, the model variant is altered according to the delta operations defined in its delta module.

The delta modeling approach using *DeltaEcore* can be categorized as a design methodology for **SPLs**. The paper presenting this approach describes how SPLs can be designed with by introducing variability into metamodels of languages, which often are the base of SPLs. As the approach using *DeltaEcore* is based upon model manipulation its classified as **static**. Furthermore, the examined design method is **top-down**, as the delta modules and feature model has to be implemented by hand. Finally, I evaluate the approach to only being **partly modular**. Because the dependencies between delta modules are can be formally defined but no mechanism to analyze them is mentioned, they can not be managed automatically when using this approach purely. Consequently, only system extensions are safe to execute, but no specializations.

A Feature-Oriented Requirements Modelling Language (FORML)

Associated with the *ID* 6, the paper *A Feature-Oriented Requirements Modelling Language (FORML)* [Shaker et al., 2012] was published by Shaker *et al.* in 2012. Shaker *et al.* describe a design methodology for SPLs using a world model in a certain state and extensible feature definitions.

The approach by Shaker *et al.* is based upon a world model. World model variants contain different concepts from the domain relevant to the developed SPL. It can further define domain specific constraints and has a world state associated to it. The world state is influenced by the instances of domain concepts, their properties, relationships and the feature configuration. A world model variant of an SPL and its state is constantly altered during the execution of the PL. Meanwhile, the world state decides how the whole SPL's behavior. How the world model can be changed is implemented in feature modules. Such modules use *feature-machines*, noted as UML activity diagrams, to define their behavior. More precisely, the feature-machines define *world-state transitions* with trigger events, guard conditions and an action, e.g. creating a new domain concept instance. This way the SPLs behavior is implemented.

Until now, I only talked about the general implementation of the SPL's behavior. To enable the composition of feature modules, each of them needs to implement a *feature-structure tree* (FST). This artifact saves nodes representing elements of the feature-machines in the tree structure and is needed in cooperation with the concept of feature enhancements. If a feature module `A` is dependent on a module `B`, `A` can enhance `B`. When doing this, the feature-machine of the feature module `A` is enhancing the feature-machine of `B`. This can be realized by using the specification of the FSTs of both modules and superimpose them. Effectively, common nodes are merged, while other nodes of feature module `A` are added to the tree of `B`. According to the new enhanced FST of module `B`, the same goes for its feature-machine, allowing module `A` to specialize and extend the behavior of the enhanced feature module. If a feature module or a feature enhancement is used, can be dependent on a feature configuration.

The presented approach can be categorized into the domain of **SPLs** as a significant focus of it is on the definition of behavior. This implies that the approach is meant to create families of applications. Furthermore, it offers a **dynamic** configuration. This can be reasoned as there are no limitations set by the superimposition of FSTs. The mechanism avoid altering the feature-machines themselves and instead only changes a tree structure are needed, which can easily be executed during runtime. As the feature modules, a corresponding feature model and a mapping between both has to be developed from scratch, the approach showcased by Shaker *et al.* uses a **top-down** design method. To address the modularity, I am convinced the evaluated design methodology is **completely modular**. For a system extension, a new feature module is added as a standalone feature or a feature enhancement, while the secondly mentioned mechanism can be used to handle dependencies between modules. When specializing an SPL by removing a feature module on which another module is dependent on, both modules are removed together if the dependent module is implemented as an enhancement. This ensures that it is possible to safely execute such an action, not leaving a faulty application.

Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study

The paper *Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study* [Batory et al., 2002] by Batory *et al.* is referenced by the *ID* 7 in the survey's table. It was published 2002 and presents a PL architecture to enable step-wise refinement of a PL, using *mixin-layers*.

Batory *et al.* showcase the *GenVoca Product Line Architecture*(PLA). It enables to add and remove components of an SPL, which are defined by feature modules, to refine the PL step-wise. The feature modules contain multiple classes that implement the behavior of the component. Furthermore, for each class in a *GenVoca component*, a state machine is defined in a *GenVoca* specific DSL. Consequently, the behavior implemented in the classes is also dependent on the their state. Similar to the design approach by Shaker *et al.* (*ID* 6), components can be defined as an extension to other components capturing dependencies between them. The extension of components is implemented by mixin-layer. This leads to inheritance structures between homonymous classes that exist in both components, meaning the class in the extended module is set as the supertype of the class in the extending component. By using such a mechanism the extending component can overwrite behavior of the refined component. Meanwhile, a state machine of an extending feature module can also refine the state machine of the extended component, but only by adding new states and transitions.

The examined design approach defines the behavior and the state of components' classes, which implies that it does address the **SPL** domain. However, the approach can only be configured **statically**. The reason for that is the mechanism of mixin-layers which is based upon inheritances, which can only be defined statically, especially as the approaches' case study is implemented in *Java*. The approach does not offer an implementation or concept of a process to derive the feature model or the distribution of components. Therefore I conclude that a **top-down** design method as applied. While the mechanism of mixin-layers limits the configuration of the design methodology, it enables the property of **complete modularity**. The extensions implemented by the mixin-layer mechanism allow to automatically handling dependencies between *GenVoca components*. In cooperation with the fact that the principles of locality and encapsulation are not violated by the component concept, this enables system extension and specialization.

A Model-Based Approach to Families of Embedded Domain-Specific Language

Identified with the *ID* 8, one can see the paper *A Model-Based Approach to Families of Embedded Domain-Specific Language* [Cuadrado and Molina, 2009] in the survey table. It was published 2009 by Cuadrado *et al.* The paper showcases an MDD approach for families of embedded DSLs, which are built by composing multiple language fragments together.

There are three distinct kinds of artifacts to implement for a language definition with this approach. Firstly, there is the abstract syntax model implemented as a metamodel. Secondly, artifiacts that are based upon the host language, an existing programming language, in which the DSLs to develop are embedded in. The underlying language is used to implement the DSL's behavior. The artifacts are marked up by specific keywords depending on a component each. The keywords, in cooperation with a final artifact, are used to generate the concrete syntax model of created languages. The mentioned final artifact is a mapping between the abstract syntax model and the beforehand mentioned keyword. Finally, executable code to implement the LPL is gained via model-to-model and model-to-code transformation based upon the concrete syntax model.

To allow the composition of LPL products, all the presented artifacts need to be composed themselves, often from fragments located in different components. The abstract syntax model of an LPL is built from multiple metamodel fragments either imported or merged together,[4] in respect to common metamodel elements. Equivalent to this, the same mechanism of import and extension

[4]This is also referenced as extension in the approach.

are used for the keywords, mappings and transformations. These loose and strict relationships allow uniting multiple fragments to feature dependent artifacts defining an LPL.

The approach by Cuadrado *et al.* is meant to design **generic LPLs**, which becomes clear when analyzing the MDD methodology. It is based on embedded languages[5] and created artifacts. Furthermore, the approach can only be configured **statically** as a change of the feature configuration leads to a composition of a new metamodel variant. This categorization is decided in accordance with the explanation on that topic in the classification of the paper with *ID* 2 by Svendsen *et al.* A **top-down** design method is used by the examined approach. The reasoning for this is the fact that no derivation algorithm for the feature model is explicitly presented. Cuadrado *et al.* describe a design methodology which only allows **partly modularity**. While system extension can be executed safely, specialization can be problematic. There is no way described to handle import dependencies between components. When removing a component, unresolvable imports could appear and break products of the LPLs.

Automating Variability Model Inference for Component-Based Language Implementations

In 2014, Vacchi *et al.* wrote the paper *Automating Variability Model Inference for Component-Based Language Implementations* [Vacchi et al., 2014], referenced by the *ID* 9. Vacchi *et al.* showcase a design approach to create families of grammar-based languages by assembling pluggable and composable units together dependent on automatically generated variability models.

Slices, which represent components in this design methodology, contain grammar production rules and its semantics. These language fragments constitute plugs and sockets of a slice. Sockets are required non-terminal symbol definitions, while plugs provide the definition of non-terminal symbols. Therefore any terminal symbols are plugs too and a series of connected plugs and sockets define one path from the starting symbol to a terminal symbol in a language's grammar tree. As these dependencies between slices can be analyzed straightforward, by checking which plugs are suitable to which sockets, a dependency graph can be generated easily. A far more complex process is used to derive a feature model from the set of slices organized in the dependency graph. In the first step of it, domain experts annotate slices with tags to give them a semantic, f.e. kind-of and part-of relationships. In the following, a distance metric is applied to the graph to generate a basic feature model. To generate the final version of it, some of the feature nodes are merged and labels are generated for them. Additionally, multiple sub-processes are applied to find mandatory relationships between slices and alternatives in the modeling choices. In the following, the generated feature model can be instantiated for a configuration dependent composition of languages defined by grammars.

I associate the design methodology by Vacchi *et al.* with the **generic LPL** domain as the paper does not imply a specialized usage of the composed languages. Meanwhile, the type of component description, grammar rules in the Backus-Naur form, locks the approach to the design of LPLs. The grammar composition of an LPL member is realized by connecting plugs and sockets. Consequently, as connecting and separating these can be executed during runtime, a **dynamic** configuration is possible. This process is further secured by relationship management using the dependency tree and generated feature model. By handling the dependencies in a changing configuration, the problem of creating optionless broken paths in a grammar tree can be avoided. The generation of the feature model based on a set of slices determines that the evaluated approach uses a **bottom-up** design method. As such a method, it also offer **complete modularity**. When extending or specialization an LPL, the set of its slices can be changed accordingly and checked for validity. If the set is still valid, as no unpluggable socket exist, the generation of the LPL's feature model can be repeated, thus creating a new PL.

[5] An example is the usage of keywords in the underlying programming language.

Apples and Oranges: Comparing Top-Down and Bottom-Up Language Product Lines

The *ID* 10 references the paper *Apples and Oranges: Comparing Top-Down and Bottom-Up Language Product Lines* [Kühn and Cazzola, 2016] by Kühn *et al.* The authors compare a top-down and a bottom-up design method to develop metamodel-based DSLs.

Common for both design methods is that a metamodel, defining the abstract syntax model of an LPL member, is altered according to a configuration derived from a feature model. The composition of the metamodel is implemented by delta modeling, explicitly the *DeltaEcore* framework, presented by the paper of Seidl *et al.*, referenced with the *ID* 5 in this survey. The overall feature model approach is adapted to the needs of LPLs. However, both design methods differ in the artifacts to implement, the way the feature model is created and the definition of a grammar.

Starting with the top-down design method, there are three phases to execute when developing an LPL in this manner. In the first one, PL engineers identify commonalities and differences of languages in the domain relevant to the developed LPL. From this manual analysis, one can derive a feature model as well as metamodel elements. In the second phase, those artifacts need to be related to each other, effectively creating a mapping from the features in the feature model to the metamodel fragments. The final step defines suitable grammar fragments and maps them to the features as well as the parts of the abstract syntax model. This way the concrete grammar of a language can be defined by artifacts distributed into different components. To conclude the considerations on the top-down design method, it is important to note that the examined paper does not present the process to compose the fragments of the grammar due to missing direct tool support. It is still presented as a feasible task, which is taken into account for the classification later.

In contrast, an LPL product defined by the bottom-up method is not defined by a specific grammar but is implemented as an embedded language. Therefore it uses the grammar of its host language. As a bottom-up approach, it further does not require to define the feature model by hand. Instead, it is derived from analyzing the components of the host language as well as the manually created metamodel fragments of the languages to develop. To enable this process, tags related to features are added to the metamodel fragments. These tags also allow managing configurations, e.g. calculating viable variants and alternatives. The feature model generation is executed in two steps. Firstly, an initial feature tree is created, which only contains the features, but no relationship between them. These mandatory, alternative or required dependencies are determined and added to finalize the feature model in a second step.

Analyzing the kind of artifacts, like metamodels or grammars, and the embedded language nature of the bottom-up design method, I assign the **generic LPL** domain to this approach. Furthermore, the design methodology only allows a **static** configuration. The reason for this is the use of *DeltaEcore*, which itself is classified as limited to the configuration to be done statically in this survey. Only the case study by Svendsen *et al.*, which is referenced by the *ID* 2, and this approach apply both design methods, **top-down** and **bottom-up**. The modularity classification differs between both methods. The top-down method only allows **partly modularity**, enabling a save extension but no specialization. The problem when removing a module, is that dependencies between components are not captured or derived into a model and automatically handled. That problem is avoided by the **completely modular** bottom-up design method as these dependencies are managed during the feature model generation, which is always executed when adding or removing components of the developed LPL.

Variability Support in Domain-Specific Language Development

Another surveyed paper by Vacchi *et al.* is called *Variability Support in Domain-Specific Language Development* [Vacchi et al., 2013] and referenced by the *ID* 11. Its goal is allowing to create families of DSLs using variability models, while also avoiding a monolithic implementation to enable reusability of its artifacts.

Similar to the other surveyed paper by Vacchi (*ID* 9), this approach also uses grammars to define DSLs in a bottom-up design method. I split the associated process is into two steps. The first one comprises the generation of a dependency graph and variability model based upon a set of slices, which are components containing a number of grammar rules and references to its semantic actions. The dependency graph is calculated the same way as it is done in the paper identified by *ID* 9. In the following a feature tree is constructed, containing a feature for each of the slices. The dependencies of the slices are modeled in it by using structural parent-child relationships or feature model constraints, like implication or equivalences. At the end of the first step, a domain expert looks over the generated feature model to eliminate redundantly defined features and detecting missing functionalities, to add later. In the second step, language-specific interpreters can be generated automatically. This is realized by creating a language descriptor, which inter alia lists the slices to integrate into a language, based upon a configuration. For the interpreter, a grammar is composed by using all rules defined in slices, which are listed by the language compiler. The dependency graph as well as the feature model relationships allow a validity check of language descriptors and enable the composition.

By working with grammar artifacts and not specifying a certain use case for its products, this design approach by Vacchi *et al.* can be rated among the **generic LPL** domain. Besides this, the design methodology can be configured **dynamically**. While this paper does not present the concept of plugs and sockets as prominent as the one identified with the *ID* 9 did, it becomes clear that slices are structured and build very similarly in both approaches. As I could not find any differing limiting factor for a dynamic configuration in this approach, my decision follows the argumentation of the former classified paper with the *ID* 9 by Vacchi. From the fact that the generation of the variability model is integral to the overall design process, I conclude that a **bottom-up** design method is used by the examined approach. Finally, I classify the **modularity** of the design methodology as **complete**. Locality as well as encapsulation are achieved by the concepts of slices with their automatic detection and handling of dependencies. When changing the set of available slices, only a simple validity check for unresolvable dependencies and rebuilding the feature model is needed.

Towards a Family of Model Transformation Languages

With *Towards a Family of Model Transformation Languages* [Sánchez Cuadrado, 2012], Cuadrado contributes a second paper to this survey. It was published in 2012 and is referenced by the *ID* 12. In this approach, multiple simple languages, which focus each on a specific transformation task, can be composed to a family of rule-based model transformation languages.

Cuadrado addresses two aspects. Firstly, there are the main design decisions which are common to all languages in the LPL to specify and realize. The second aspect describes how interoperability and a composition of the languages can be achieved. A tool to reach the goals of both aspects is a common intermediate language for model transformations, which uses lower-level mechanisms to allow it being compatible with as many transformation languages as possible. For all component languages, a transformation to this intermediate language has to be defined. In particular, the intermediate language can be used to implement general design decisions, which are relevant to all component languages as well as enable interoperability between the different languages. The composition of the component languages is realized by allowing one language access to functionalities of other ones and building chains of transformation rules.

To make this possible the relationships types between the components of an LPL for transformations have to be found and taken into account by the common intermediate language. If a transformation rule or pattern needs values saved or calculated by artifacts of another component language, the keyword *providing* makes clear that a specific value should be accessible for all other component languages too. The accessibility of it is controlled by the intermediate language in the following. Furthermore, the intermediate language also act as an tracing model, creating a level of indirection, to resolve references between different component languages. It can also handle model elements, which are extended by an attribute only during the transformation. Of course, the mechanism for those decorated elements need to be implemented for the component languages, which want to create and query the elements. Finally, the transformation has to be configured using potentially multiple small transformation languages at once. Transformation rule chains enable this, while also avoiding to merge rules of different languages together. Instead the transformation rules are executed successively in a manner that the resulting model element of the first rule is the input for the second one and so on. The rules in these chains and their order are defined in configuration files.

The common intermediate language is specialized for model-to-model transformations. Therefore Cuadrado's design methodology is limited to the domain of **LPLs for transformations**. The approach can be configured **dynamically**, as only the configuration files determining the transformation rule chains and the relevant keywords have to be changed when altering the configuration. It is feasible that both tasks can be executed during runtime. Cuadrado presents a **top-down** design method as no processes to automatically generate a feature model based upon a set of component languages is mentioned. Lastly, its design approach is only **partly modular** as it is not always possible to specialize an LPL with it safely. The problem is that relationships between component languages are not derived and managed automatically. Therefore it can happen that a component language was dependent on a removed one, which leads to unresolvable references or value queries.

Handling Variability in Model Transformations and Generators

The paper *Handling Variability in Model Transformations and Generators* [Voelter and Groher, 2007] by Voelter *et al.* is identified by the *ID* 13 and published in 2007. Overall the paper discusses how SPLs can be created using an MDD approach by defining the variability in the model transformations and code generators. However, I will focus only on the question how model-to-model transformations can be made configurable.

Voelter *et al.* address aspects, which lead to cross-cutting code among several components of model-driven developed SPLs, with MDD-AO-PLE (*Model Driven Development Aspect-oriented Product Line Engineering*). These aspects itself are also encapsulated in components and often cross-cut multiple transformation rules. The examined approach addresses the cooperation between those two types of components, the aspect components and the cross-cut components. Voelter *et al.* describe the transition between the problem space model, which contains requirements, features and a configuration to a suitable solution space. In particular, the problem space is defined by requirement tables and a feature model. An instantiation of the feature model is used to decide which aspects should be applied to a set of model transformation rules. This creates the solutions space. To implement the aspects, interceptors are used. They allow executing code before and after transformation rules. However, these interceptors do not change the code in the transformation rules but represent a loosely related artifact called before and after a rules execution. This offers the advantage of being able to easily change between using and not using an interceptor, depending on a feature model.

The approach is classified as belonging to the **LPL for transformations** domain as I explicitly focused on the part of that discusses how to enable cross-cutting variability for a transformation languages. Furthermore, the separation of the aspects implemented as interceptors and the transformation rules allow a **dynamic** configuration. This avoids building or rewriting transformation rules directly when configuring the LPL. Only an analysis of a configuration and decision if a specific interceptor should be used or not is needed, which can be executed during runtime. Voelter *et al.* further uses a **top-down** design method as the developer needs to implement the feature model as well as the mappings between aspects and the feature manually. Finally, the evaluated approach also offers **complete modularity**. Using it, locality and encapsulation can be achieved and is even emphasized by the idea of treating aspects outside of the cross-cut components. The use of interceptors also directly handles dependencies between the cross-cut components and aspect components. The cross-cut components are never dependent on aspects. Meanwhile, removing a cross-cut component associated with an aspect only leads to a smaller set of cross-cut transformation rules of the aspect and does not create unresolvable dependencies. Relationships between aspects can be managed the same way as it is feasible to apply interceptors on interceptors.

Building Families of Languages for Model-Driven System Development

Evans published the paper *Building Families of Languages for Model-Driven System Development* [Evans et al., 2003], referenced with the *ID* 14, in 2014. The paper collects and describes two architectures and a number of implementation techniques for families of languages meant to be used for MDD, to allow a maximum of flexibility in such processes.

Evans *et al.* present two architectural possibilities when creating LPLs. The first architecture follows the classical PL formula of offering variable extensions to a fixed core of language elements. In contrast, an integration framework enables the cooperation of multiple fully implemented languages via mapping. The process executed using the mapping is called *language integration*. Additionally, the paper provides an overview of seven implementation techniques in the same domain. It is assumed that the languages of the developed LPL are defined by metamodels, which can be modified if needed.

The first technique uses *stereotypes*[6] to allow a lightweight extension and extends the metamodel elements by attributes and operations, e.g. to restrict the state space of an element. This does not enable to add new metamodel elements to the model. However, the next technique, *meta-metamodel instantiation* makes this possible. The heavyweight extension uses a new instance of the metamodel which can be expanded on a developer's wish. The original metamodel is elevated to a meta-metamodel at this point. The third presented technique adds sub-elements to an existing element of a metamodel by using *abstract class hierarchies*. Thereby, the sub-elements inherit from the extended element. If a metamodel uses packages, the following technique can be useful. *Package extension* allows one package to extend another one by adding new elements to it. *Templates* can also be used to manipulate metamodels. Such define patterns of metamodel elements and their relationships as reusable packages. The first five mentioned techniques are applicable for the PL architecture of language families. In contrast, the sixth one, *explicit mapping*, is meant to implement integration frameworks as the mapping allows that one language can be viewed as an instance of another one. This effectively enables cooperation between otherwise fully independent languages. I categorize the final technique of *feature models* as cross-cutting. They can be used to decide which changes to a metamodel should be executed or which languages cooperate in an integration framework.

As Evans *et al.* do not present a completely elaborated design approach but rather a collection of architectures and techniques, the classification is not as unambiguous as one of the other design methodologies. However, I will classify the single architecture choices and techniques to gain a representative evaluation of the paper by Evans *et al.* The choice of the architecture is determining on the modularity of an LPL. Locality and encapsulation can be achieved by both as the paper mention a strict differentiation between the intra-model architecture, concerning a language-specific component organization, and the inter-model architecture, which addresses the organization of all available components in an LPL's framework. Meanwhile, the automatic handling of dependencies between components is the difference between the PL and integration framework architecture. The first architecture allows complete modularity as components can be implemented as extensions

[6]This technique is equivalent to the eponymous concept in the UML. [Rumbaugh et al., 1999]

to other components. This ensures the deletion of such extension when removing the extended component and avoids unresolvable dependencies when changing the set of available components. While the PL architecture makes system extension and specialization possible, the use of an integration framework only enables specialization. When adding a new language to integrate into the framework a mapping between the new one and all the already integrated ones has to be provided. To realize this, either the new component needs to know all the already integrated languages or already existing components would need to know there is a new language to integrate. Removing a language from the framework is not a problem as there are no dependencies between the autonomous implemented languages.

The choice of the used techniques determines if the configuration can be executed during runtime or not. The techniques that address altering the metamodel of a language do only allow a static configuration. Such techniques are stereotypes, meta-metamodel instantiation, abstract class hierarchies, package extensions and templates. In contrast, the configuration a language family can be altered during runtime when using explicit mappings. Overall I split the classification of the paper in terms of configuration and modularity. But at first I will address the domain of it. Evans *et al.* emphasize that the presented techniques are targeted to allow flexibility for languages used in MDD processes. Therefore it is associated to the **LPL for MDD** domain. Furthermore, the paper does not present processes to generate the feature model in an automated manner and is classified as a **top-down** design method. The configuration and modularity evaluation of this survey entry is coupled as depending on the used architecture and techniques a family of languages does either provide **static configuration and complete modularity** or **dynamic configuration and partly modularity**. The first classification can be achieved by applying the PL architecture with techniques manipulating a metamodel, while the second one is relevant to integration frameworks using explicit mappings.

A Family of Role-Based Languages

The last entry of the survey, referenced by the *ID* 15, is the thesis *A Family of Role-Based Languages* [Kühn, 2017] by Kühn. Kühn showcases a family of Role-based Modeling Languages and suitable GEs, which further contains an LPL of transformation languages.

I will focus on the family of GEs, which is also the base of the case study in this thesis[7]. The family of role-based languages is also part of the case study and is elaborated in 5.1.1. As it is heavily based upon delta modeling using *DeltaEcore* [Seidl et al., 2014], which is already surveyed and identified by *ID* 5, the family of RMLs is not part of this survey. It provides the definition of the feature model used for the GEPL, of which configurations can be altered dynamically. The GEPL can be split up into two parts. The configuration changes are relevant to an SPL and an LPL.

The SPL aspect of it addresses edit concerns, just as the graphical representation of model elements and the access to edit operations. The way these concerns are made feature dependent involves noticing the artifacts that are responsible for defining the appearance of model elements, the GE's palette and context menus every time the configuration is changed. For the artifacts addressing the visual appearance of model elements, this could lead to show or hide specific graphical objects, while it also might be needed to recalculate the visible entries associated to GE functions in the palette and context menus. The LPL aspect applies to a model-to-model transformation language. The transformation is implemented by rules which offer *guard* expression to decide if a rule is applied to transform a specific model element. Checks for chosen features in the current configuration can be either implemented in the guard expression or in the transformation rule's actions, which define how a model element is transformed.

Kühn's design approach can be associated with the domain of **GEPLs**. The product of is a family of GEs and has definitive characteristics distinguishing it from a generic SPL. A GEPL involves

[7]See 5.1.2 for reference.

aspects of cooperating SPLs and LPLs. The GEPL can be configured **dynamically** as all feature dependent parts of it implement configuration checks during runtime. The approach further uses a **top-down** design method, meaning that a developer has to define the feature model as well as the mapping between its features and the implementing components. The modularity is the biggest weak point of the examined design methodology. It offers **no modularity** as code implementing one feature is spread around different artifacts and not collected in a single comprising component. This monolithic code base allows no automatic system extension or specialization.

2.3 Discussion

The survey unveils a glimpse into the wide variety of conceptual approaches, tools and frameworks to design PLs. It contains a spectrum described by general design concepts, like survey entry with the *ID* 1, a collection of multiple PL techniques in paper 14, a tool showcase in paper 4, a technical report identified by *ID* 6, and industry-oriented papers, like entries 2 and 7. The surveyed papers describe possibilities to design families of different products, applications or languages, and often offer an associated case study along. Further into the design approaches, components can be of different natures, e.g. single independent features, extending dependent features, cross-cutting aspects, parts of language grammar definition or whole independent languages. However, albeit the variety there can also be found common technologies between the approaches, which I want to give a brief overview on in the following.

A set of recurring languages and frameworks to be found in multiple surveyed papers comprises CVL, EMF and Neverlang. The *Common Variability Language*(CVL) [Fleurey et al., 2012] is a domain-independent language to specify and resolve variability [Vacchi et al., 2014]. Consequently, it is used in the survey entries 2, 3, 9 and 11 to define variability models as a base of the developed PLs. Other approaches either do not address the creation or management of the variability model, generate them automatically or use other languages for this task. The *Eclipse Modeling Framework* [Steinberg et al., 2009] can be used to define and manage generic models. Its broad service is in usage to define metamodel in the surveyed papers, e.g. the ones referenced by the *IDs* 2, 3, 5, 8 and 15. Finally, *Neverlang* [Vacchi and Cazzola, 2015] is a framework which can be used to define grammar-based language modules containing information about parsing and type checking in respect to an underlying language. Features and associated tags for the features' grammar position and classification are also part of the modules [Vacchi and Cazzola, 2015]. The *Neverlang* framework is applied by approaches described in the papers 9, 10 and 11.

Before I present and evaluate the results of the classification, I want to address the, in my opinion, two most relevant sources of errors. Firstly, the survey only comprises 15 papers. This number is not high enough to allow gaining a fully representative result for the overall PL domain. However, I consider such a survey feasible only in a focused and dedicated work towards it. I derive my results in respect to the limited amount of surveyed papers. Another risk for the significance of this survey is the fact that a classification like this always is subject to interpretation. The exact assignments of properties to design approaches cannot be completely formalized and may differ between various researchers. To handle this, I put a focus on the argumentation why I classified each of the presented papers as I did. An example of a disputed issue could be the general association of design approaches manipulating metamodels with the limitation of static configuration. To defend this survey consistent decision, I would argue that altering a metamodel during runtime is a far more complex process than other dynamic variability mechanisms, like dynamic changes of connection between plugs and sockets or dynamic feature configuration queries in code. Additionally, as the metamodels in a PL are most likely to be a high-level base artifacts, altering them can lead to changes in a whole product which are also harder to manage and keep in control. This further adds up to the effort of metamodel changes. Therefore, together with the fact that none the surveyed papers using metamodel manipulation to introduce variability, addresses the topic of static or dynamic configuration, I came to the decision for the static configuration classification on such

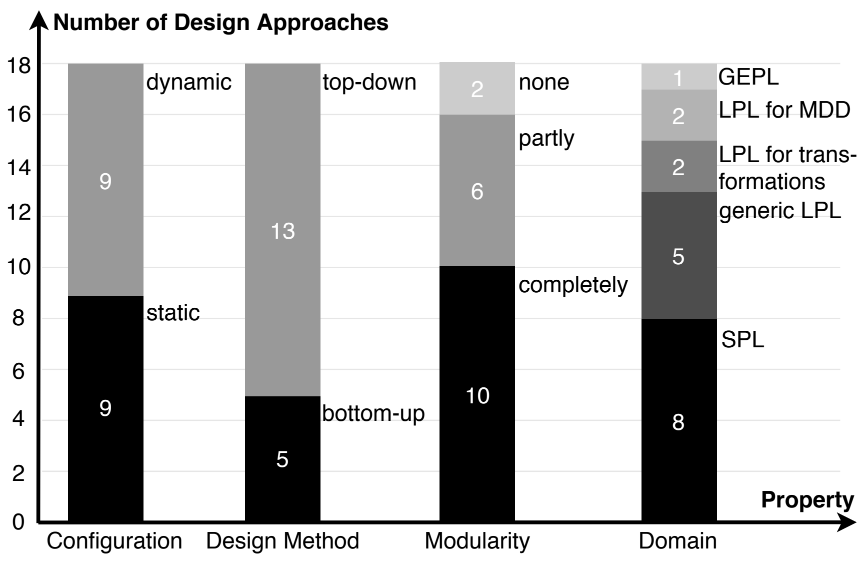


Figure 2.3: Number of design approaches for each of the four classification properties.

approaches. While this is only one example, there might be more parts of the classification in this survey, in which different properties can be assigned and argued for.

2.3.1 Evaluation

Overall the 15 surveyed papers present 18 different classified approaches. The papers referenced by the *IDs* 2, 10 and 14 provide each two different design approaches. These are differentiated by the design method for paper 2 and 10. The two methodologies presented in paper 14 apply various PL architectures and techniques, leading to different configuration and modularity properties. *Figure* 2.3 displays the distribution of specific classification properties for the 18 approaches. One can see that there is the same number of design methodologies allowing dynamic configuration as such that only enable the creation of PLs to be configured statically. The 50% share of only statically configurable design approaches can be explained as there are many papers presenting either model driven design methodologies or discuss metamodel-based LPLs. Such survey entries often introduce variability by altering metamodels. The processes of altering metamodels according to a feature model are evaluated as not being feasible to execute at runtime of the associated PL in this study. Meanwhile the papers presenting such solutions do not address these issue and are therefore classified as only offering static configurable PLs.

Analyzing the distribution of approaches in respect to their design method, it can be stated that significantly fewer methodologies use bottom-up processes. The five to 13 ratio between approaches applying bottom-up and top-down methods, can be reasoned by the complexity of generation and transformation implementations for feature models and the associated mappings. Processes to generate feature models often do involve deriving multiple other models and analyzing or transforming them. While it the calculation of features in a variability model is relatively straightforward, gaining knowledge about their relationships can be quite hard. The surveyed paper 9 by Vacchi *et al.* show the effort to implement such processes quite good, as it mentions several steps and sub-steps to derive a feature model based on a set of language fragments, including the use of annotations, distance metrics, merging as well as labeling procedures and heuristics. While the effort put into the developing of such solutions can be worth as it renders defining feature models obsolete, it can still be perceived a starting hurdle. Another reason for the lower number of design approaches which use a bottom-up method is the more general procedure of the top-down process. To use a bottom-up method, existing components or full products need to be accessible, which is not always the case. When applying a top-down process a new PL can be created from scratch, which offers flexibility in the development.[8] Overall the top-down design method is applicable to develop PLs in more situation.

Continuing with the next classification property, ten of the evaluated design approaches allow complete modularity. Six of them either fail to enable system extensions or specializations, while the last two do offer no modularity at all. Often the partly modular design approaches are not able to derive and handle dependencies between components automatically. In such a case, it is not ensured that a component can be removed safely from the PL. The removal might break the product family as it creates unresolvable dependencies if a feature module was deleted on which other ones depend. By automatically handling dependencies, via feature extension or the usage of a dependency graph, for example, this problem can be avoided. In one case of a partly modular design approach, namely the one based on integration frameworks presented in paper 14, the system extension cannot be executed in an automated manner. Left to reason about are the two design approaches which offer no modularity. On one hand, Paper 4 does not address modularity and component design at all as it describes a tool limited to creating, analyzing and managing feature models and the languages to define these. On the other hand, paper 15 presents a GEPL, which components' implementations are comprised of spread around code. Therefore it is not possible to add or remove features in an automated way. Only in the last mentioned paper, the feature modularity principle of locality is violated, while the principle of encapsulation is not respected by some more approaches that are classified as not or only partly modular.

The fourth bar on the right in *Figure* 2.3 showcases the distribution of classified approaches on the PL domains. Naturally, for the more general domains, like SPLs and LPLs, more design approaches are found. When the domain is more specific, LPLs for special use cases, for example, or leads to the use of combinations of SPLs and LPLs, like GEPLs, the number of design methodologies is significantly reduced. This can be seen clear as there are eight approaches addressing the design of SPLs and five of them for generic LPLs. Meanwhile, there are only five methodologies meant to be used in the remaining three domains, distributing two on LPLs for transformations, two on LPLs for MDD and finally one on the GEPL domain.

2.3.2 Results

To conclude the survey, I want to analyze the coverage of design approaches, especially dynamic and modular ones, for the different domains. The distributions of those for different classification properties were evaluated isolated from each other in the subsection before. In contrast, *Figure* 2.4 combines all four properties to give an overview of the coverage to analyze. In it, the grey shades of the grid nodes still represent the modularity, but summarize all found approaches with the same domain, configuration and design method assigned. The classification grid shows that there is a good coverage of modular design approaches for the domains of SPLs of LPLs. For SPLs three of the associated nodes are drawn black, meaning there exists at least one modular approach. Only design methodologies for dynamic configurable SPLs based on a bottom-up process are missing in this survey. In comparison, such an approach is part of the survey for LPLs, alongside static top-down and bottom-up methods. However, the top-down design methodologies only offer partly modularity. This means developing an LPL with a generic process starting with its conception does not enable complete modularity without extra effort besides the used approach presented in this survey. While this can be evaluated as a worse coverage in respect to modular approaches, the generic LPL domain still offers a wide variety of design processes.

This isn't the case for more specific domains. In the third domain of LPLs for transformations effectively only top-down methods exist. To keep the figure clear, only the node representing top-down methodologies for dynamically configurable PLs is marked as occupied in *Figure* 2.4. However, as it is a fairly simple technical task to implement a static configuration for dynamical

[8]Note that over time the additonal implementation effort for top-down components can be potentially bigger than implementing a bottom-up process.

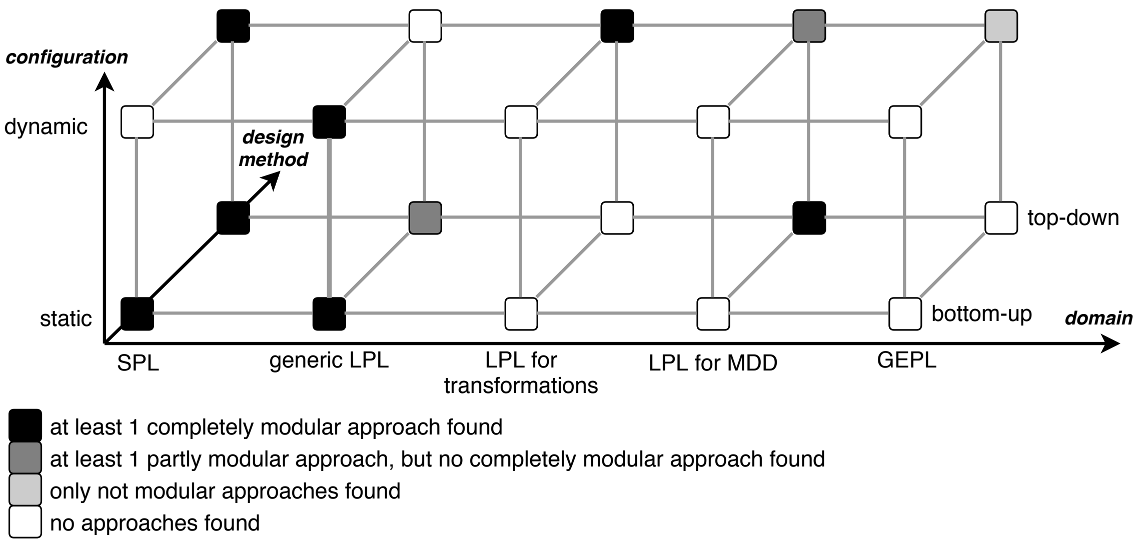

configuration
dynamic
static
design method
top-down
bottom-up
domain
SPL
generic LPL
LPL for transformations
LPL for MDD
GEPL
at least 1 completely modular approach found
at least 1 partly modular approach, but no completely modular approach found
only not modular approaches found
no approaches found

Figure 2.4: Coverage of the surveyed paper for different domains and other properties.

configurable PLs, the existence of static top-down methods in the currently addressed domain is implied. For good news, it is possible to design LPLs for transformations in a modular manner. Unfortunately, this survey did not find a bottom-up process to define such LPLs. The same goes for the next domain addressing LPLs for MDD. Only approaches of a top-down nature are part of this overview on design methodologies. It is even worse compared the domain mentioned before, as there is no dynamic modular approach found. Instead it only offers partly modularity, limiting the range of available design methodologies even more.

Finally, for the domain of GEPLs, only one top-down approach can be found, which is not modular. This is a problem as GEs are applications with a high complexity. An LPL for transformations or MDD process could also be designed by using approaches for a generic LPLs. More specialized design methodologies are usually seen as more useful but are not needed in any case for LPLs. This need is bigger for GEPLs as they are a combination of SPLs and LPLs, which immediately leads to a more complex situation. Furthermore, GEs are comprised of multiple concerns, which should be implemented in different artifacts as much as possible, which leads to the problem of managing a lot of artifacts of different kinds in one component. This circumstance adds more complexity and renders a specialized modular design approach for the GEPL domain even more useful. As this need is the base of the thesis, the following chapters are dedicated to defining concrete requirements for PLs in the domain (Chapter 3), presenting a dynamic modular top-down design approach (Chapter 4) and finally also implementing a case study according to the beforehand established design methodology (Chapter 5).

3 Requirements of Graphical Editor Product Lines

This chapter lists and presents the requirements of GEPLs. Besides the general tasks of a GE and the SPL aspect of it, a modular development has also to be addressed by the requirements. The list of requirements will be used to evaluate the general design approach (Chapter 4) and the case study (Chapter 5) presented in this thesis. Therefore, they have to be fitting for an analysis of the concrete GEPL implementation of the case study. Meanwhile, they should be defined general enough to estimate if the approach to the development of GEPLs is useful. This chapter will be structured by differentiating between functional and non-functional requirements. Furthermore, the functional requirements are classified by the application concerns they belong to. According to that the application concerns are presented in the *Functional Requirements* section and further elaborated by the requirements and the textual description accompanying them. Finally, a distinction between the user and developer oriented non-functional requirements is made.

After all, the requirements of the GEPL domain follow the considerations made by Glinz [Glinz, 2007]. He presents a taxonomy of requirements, which divides them, as already established, into functional and non-functional ones. Furthermore, the taxonomy organizes non-functional requirement in the categories of performance, special quality and constraint requirements. Chung *et al.* [Chung and do Prado Leite, 2009] published another helpful analysis of requirements. He provides a variety of classification scheme for such, including lists and structures of non-functional requirements derived from science and industry. Both the papers of Glinz and Chung *et al.* are used as a foundation for the elaboration of the non-functional requirements in this chapter.

3.1 Functional Requirements

In the following section, the functions of a GEPL will be broken down. They are structured by the seven concerns they belong to as artifacts of every such aspects have their own distinct tasks. The hierarchy of the concerns can be seen in *Figure* 3.1. There are *Edit* and *Language Family Concerns*. Roughly speaking, the *Edit Concerns* define how elements are drawn during their whole life cycle (*Graphical Representation*), how the palette is visualized (*Palette Properties*) and which operations can be executed in which situation (*Edit Policies*). Overall these are tasks common to all GEs. To create a PL of GEs the artifacts of *Language Family Concerns* are needed. First and foremost, the *Feature Model* is the foundation of a feature-oriented SPL. The *Abstract Syntax Model*, an intermediate representation between the syntax model of the GE and the *Target Metamodel*, is also part of the *Language Family Concerns*. The lastly mentioned *Target Metamodel* itself is a concern, offering a comprehensive model representation of a domain. The same is the case for the *Model Transformation* between models of the *Abstract Syntax Model* and the *Target Model*. This coarse

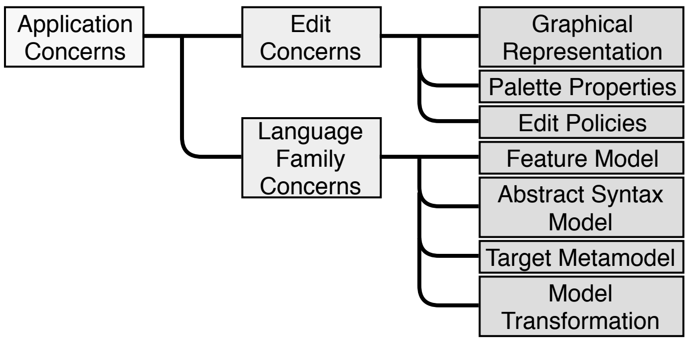


Figure 3.1: The applications concerns of a GE.

description of the concern's tasks is more elaborated by the following functional requirements. It is important to note that the functional specifications are partly derived and generalized from the FRaMED Tool described in [Kühn, 2017] and [Kühn et al., 2016].

3.1.1 Edit Concerns

Graphical Representation

Overall the artifacts of the *Graphical Representation* concern have four big remits. All these are captured by different requirements listed in *Table* 3.1. The first field of tasks, defined by the functional requirements `FR01` to `FR03`, gives the concern's name. These functions describe how the visual appearance of a model element's graphical object is controlled. While `FR01` manages this during the creation, `FR02` and its sub-requirements are responsible during the lifetime of a model element. Some operations on model elements change the graphical representation of such. A minimal set of user-initiated operations is defined by the requirements `FR02.1` to `FR02.4`, including moving and resizing. Textual attributes of model elements, e.g. the name or cardinalities, can be subject to change too. Finally, it should be possible to alternate the source or target anchors of relations. These four types of actions changing a model element's visual appearance have to be implemented in the *Graphical Representation* concern's artifacts. `FR03` states that the same artifacts also have to address how a model element's graphical object is removed completely.

A second model needs to be manipulated, according to the changes of the *Concrete Syntax Model*, which is the GE framework's model capturing the visual appearance of a diagram. This second model is the domain specific *Abstract Syntax Model*. Why it is needed and its requirements are listed in the paragraphs about the *Abstract Syntax Model* in the section 3.1.2. In contrast, the model previously mentioned for the visual representation is offered by and dependent on the chosen GE framework. However, a co-evolution between the two models needs to be implemented. This is captured by the requirements `FR04` to `FR06`. They define how the *Abstract Syntax Model* is alternated when creating (`FR04`) and deleting (`FR06`) model elements. Furthermore, there is a subset of the operations presented in the requirements `FR02.1 to FR02.4`, that also implicates that the *Abstract Syntax Model* is subject to changes. Such are on one hand the change of textual attributes. On the other hand, manipulating the source and target of a relation also propagates to the domain model elements of the *Abstract Syntax Model*. These operations are defined by the requirement `FR05`.

The seventh top-level requirement defines the functionality of offering multiple zoomed in views on one diagram. This makes the most sense for model elements that group other elements together. Such a grouping model element could be presented by a simple frame around all inner elements. However, this visual solution can become too large, as the grouping element can have any number of children. Furthermore, it is difficult to use such a presentation for nested grouping model elements. Consequently, a feature which enables the user to *step in* and *out* of grouping model elements is

ID	Description
FR01	The artifacts of the *Graphical Representation* concern shall define the visual appearance of model elements on their creation.
FR02	The artifacts of the *Graphical Representation* concern shall define how the visual appearance of model elements is changed when executing user-initiated operations on them.
FR02.1	Move: Changing the position of a model element's graphical object.
FR02.2	Resize: Changing the size of a model element's graphical object.
FR02.3	Textual edit: Edit textual parts of a model element's graphical object.
FR02.4	Reconnect: Changing the source or target anchors of a model element's graphical object.
FR03	The artifacts of the *Graphical Representation* concern shall remove the visual appearance of model elements on their deletion.
FR04	The artifacts of the *Graphical Representation* concern shall define how the *Abstract Syntax Model* is changed on the creation of a model element.
FR05	The artifacts of the *Graphical Representation* concern shall define how the *Abstract Syntax Model* is changed when executing user-initiated operations on model elements.
FR05.1	Textual edit: Edit a purely textual attribute of a model element.
FR05.2	Reconnect: Changing the referenced source or target element of a model element.
FR06	The artifacts of the *Graphical Representation* concern shall define how the *Abstract Syntax Model* is changed when deleting a model element.
FR07	The artifacts of the *Graphical Representation* concern shall enable to zoom into views on multiple levels on the edited diagram.
FR08	The artifacts of the *Graphical Representation* concern shall implement sanity checks when creating a model element.
FR09	The artifacts of the *Graphical Representation* concern shall implement sanity checks when executing user-initiated operations on model elements.
FR09.1	Move: Changing the position of a model element's graphical object.
FR09.2	Resize: Changing the size of a model element's graphical object.
FR09.3	Textual edit: Edit textual parts of a model element's graphical object.
FR09.4	Reconnect: Changing the source or target anchors of a model element's graphical object.
FR10	The artifacts of the *Graphical Representation* concern shall implement sanity checks when deleting a model element.

Table 3.1: Functional requirements of GEPLs associated with the *Graphical Representation* concern.

useful. When stepping into a model element, all elements outside of it are not visible anymore. Only the child elements of the grouping element are shown, which reduces the potential number of presented model elements drastically. Of course it is also possible to zoom in on a fitting model element inside another grouping element, thus solving the problem of presenting nested ones.

Finally, the requirements `FR08` to `FR10` describe that the artifacts of the *Graphical Representation* concern should implement simple sanity checks for all operations to edit a model element during its whole life cycle. Similar to the life cycle referenced for the feature `FR01` to `FR03`, this means that sanity checks are needed during the creation of an element, its lifetime and its deletion. The checks during the creation are defined by `FR08`, while `FR10` does the same for removing an element. During the lifetime of such model elements, the operations already described for the requirements `FR02.1` to `FR02.4` can be executed. `FR09.1` to `FR09.4` establish that for the same operations sanity checks are needed, to determine if they can be executed. These simple sanity checks are meant to look for missing information or inconsistencies in the model to edit, which make it impossible to execute a specific operation usefully. It is important to note that this excludes analyzing the current feature configuration of the GEPL, as such checks are part of the *Edit Policy* concern.

ID	Description
FR11	The artifacts of the *Palette Properties* concern shall define if GE features, to be shown in the palette, are visible depending on the current editor state.
FR11.1	The editor state to consider in this case contains the current feature configuration.
FR11.2	The editor state to consider in this case contains the current editor view.
FR12	The artifacts of the *Palette Properties* concern shall define if GE features, to be shown in a context menu, are visible depending on the current editor state.
FR12.1	The editor state to consider in this case contains the current feature configuration.
FR12.2	The editor state to consider in this case contains the current editor view.
FR12.3	The editor state to consider in this case contains the information on which kind of element the context menu is opened on.
FR13	The artifacts of the *Palette Properties* concern shall define the appearance of the GE features to be shown in the palette.
FR13.1	They define the presented name of a GE feature to be shown in the palette.
FR13.2	They define the presented icon associated with a GE feature to be shown in the palette.
FR13.3	They define in which palette category a GE feature is located in.
FR14	The artifacts of the *Palette Properties* concern shall define the presented name of the GE features to be shown in a context menu.

Table 3.2: Functional requirements of GEPLs associated with the *Palette Properties* concern.

Palette Properties

The *Palette Property* concern's artifacts are responsible for the visibility and appearance of feature entries in the GEPL's palette and context menus. The visibility calculations, depending on the current GE state, are represented by the requirements FR11 and FR12 in *Table* 3.2, as well as their sub-requirements. Both top-level requirements are differed by the type of feature that is addressed. FR11 is concerned with features that are shown in the palette of a GE. Meanwhile, the other mentioned requirement describes that the artifacts of the *Palette Property* concern control the visibility of features meant to be shown in a context menu. The context menu is usually opened by a right click on a specific model element. Back to requirement FR11, its sub-requirements describe what parts of the editor state are to analyze to determine if a feature is visible in the palette or not. Firstly, there is the configuration of the editor (FR11.1). Secondly, the current view of the GE is also to consider, according to FR11.2. This is coupled with the requirement FR07 of the *Graphical Representation* concern. Hence, it is important to know if the view of the GE is currently zoomed in, an if, in which kind of model element the user stepped in. These two parts of the editor state are also relevant to decide if a context menu's feature should be visible in the current situation. Accordingly, the requirements FR12.1 and FR12.2 are defined here equivalent to the sub-requirement of FR11. However, there is one more influence on the previously mentioned decision. Integral for the calculation of a feature's visibility in a context menu is, on which kind of model element the menu is opened on.

The appearance of features also has to be defined by this concern's artifacts. On one hand, for features meant to be shown in context menus this is fairly easy as only the externally presented name of that feature has to be determined. Requirement FR14 states that. On the other hand, the appearance of features in the palette is specified by two more attributes. The sub-requirements of FR13 capture these. Equally to FR14 the presented name of a GE feature, that is part of the palette, is defined according to FR13.1. Additionally, the specification of the appearance also contains a reference to an icon representing the feature figurative (FR13.2) and the information to which palette category the GE feature belongs to (FR13.3). These palette categories group similar features together and offer a structured look of the palette.

ID	Description
FR15	The artifacts of the *Edit Policy* concern shall define if a model element can be created depending on the current editor state.
FR15.1	The editor state to consider in this case contains the kind of model element to create.
FR15.2	The editor state to consider in this case contains the current feature configuration.
FR15.3	The editor state to consider in this case contains the current editor view.
FR16	The artifacts of the *Edit Policy* concern shall define if user-initiated operations can be executed on a model element depending on the current editor state.
FR16.1	This should be ensured for all basic user-initiated operations.
FR16.1.1	Move: Changing the position of a model element's graphical object.
FR16.1.2	Resize: Changing the size of a model element's graphical object..
FR16.1.3	Textual edit: Edit textual parts of a model element's graphical object.
FR16.1.4	Reconnect: Changing the source or target anchors of a model element's graphical object.
FR16.2	The editor state to consider in this case contains multiple influences.
FR16.2.1	The editor state to consider in this case contains the kind of operation to execute.
FR16.2.2	The editor state to consider in this case contains the kind of model element to edit.
FR16.2.3	The editor state to consider in this case contains the current feature configuration.
FR16.2.4	The editor state to consider in this case contains the current editor view.
FR17	The artifacts of the *Edit Policy* concern shall define if a model element can be deleted depending on the current editor state.
FR17.1	The editor state to consider in this case contains the kind of model element to delete.
FR17.2	The editor state to consider in this case contains the current feature configuration.
FR17.3	The editor state to consider in this case contains the current editor view.

Table 3.3: Functional requirements of GEPLs associated with the *Edit Policy* concern.

Edit Policies

The *Table* 3.3 lists all requirements of the *Edit Policy* concern. Edit policies are used to calculate if GE operations for a model element can be executed. This happens during the whole life cycle of an element: At creation (FR15), during its lifetime (FR16) and when it is going to be removed from a model (FR17). Firstly, I want to discuss the requirements FR15 and FR17. The function associated to first requirement mentioned determines if a model element can be created in a model, dependent on the model element's type, the current feature configuration and the editor view. These influences on the calculation are captured by the sub-requirements FR15.1 to FR15.3. The editor view dependents on the function defined in requirement FR07 of the *Graphical Representation* concern. It differs between a top-level view and multiple views in which the user zoomed into different model elements. Very similar is the requirement FR17 defined, as its sub-requirements reference the same influences to check if a model element can be deleted.

The specification of the requirement FR16 is more complicated. It captures the decision typical for edit policies when the user wants to execute an operation on a model element after its creation and before its deletion. Its sub-requirements can be separated into two categories. Firstly there is a minimal set of operations, for which the edit policies need to decide on their execution (FR16.1). As these operations already played an important role for other requirements, like FR02.1 to FR02.4, one can look up their definition in the paragraphs about the *Graphical Representation* concern's requirements. FR16.2 represents the second category, defining the influential factors for the decision made by the edit policies. The parts of the editor state defined by FR16.2.2 to FR16.2.4 are identical to the already presented ones of FR15.1 to FR15.3. However, the requirement FR16.2.1

ID	Description
FR18	The artifacts of the *Feature Model* concern shall define a structured and constrained feature model.
FR18.1	The feature model contains multiple named features.
FR18.2	The feature model collects its features in a structured manner.
FR18.3	The feature model allows feature constraints between features.
FR18.3.1	Implication: If A implies B, B is always selected if A is selected.
FR18.3.2	Equivalence: If A equals B, A can only be selected together with B, and vise versa.
FR19	The artifacts of the *Feature Model* concern shall implement the validity check for configurations derived from its feature model.
FR19.1	They shall calculate the validity of configurations depending on the structure of the feature model.
FR19.2	They shall calculate the validity of configurations depending on the feature constraints.
FR20	The artifacts of the *Feature Model* concern shall calculate automatic selections and eliminations of features in a configuration derived from the feature model.
FR20.1	They shall calculate the automatic selections and eliminations dependent on the structure of the feature model.
FR20.2	They shall calculate the automatic selections and eliminations dependent on the feature constraints.
FR21	The artifacts of the *Feature Model* concern shall define a standard configuration.
FR22	The artifacts of the *Feature Model* concern shall implement a configuration editor.
FR22.1	The configuration editor allows dynamic configuration changes.
FR22.2	The configuration editor shows the structure of the feature model.
FR22.3	The configuration editor shows the feature constraints of the feature model.
FR22.4	The configuration editor shows the status of features in the current configuration.
FR22.5	The configuration editor shows which features' statuses are locked due to the feature model's structure and feature constraints.
FR22.6	The configuration editor shows which automatic selections and eliminations were executed due to the feature model's structure and feature constraints.
FR23	The artifacts of the *Feature Model* concern shall signal configuration changes to artifacts of other concerns.

Table 3.4: Functional requirements of GEPLs associated with the *Feature Model* concern.

states how now also the kind of operation is important to consider when checking if the operation can be executed.

3.1.2 Language Family Concerns

Feature Model

While the feature model is an important foundation of a GEPL, its definition is not the only task of the *Feature Model* concern's artifacts. All remits of them are listed in *Table* 3.4. However, the first top-level requirement FR18 describes how a proper and comprehensive feature model is defined. It collects multiple named features in manageable structure (FR18.1 and FR18.2), for example, a tree. Furthermore, the feature model should also allow defining feature constraints, like implications and equivalences between features, which is captured in the FR18.3 and its sub-requirements. Now moving on from the pure feature model definition, there are multiple requirements addressing the management of the configuration derived from it. There are basic tasks, like checking if a configuration is valid (FR19), offering a standard configuration (FR21) or signaling configuration changes to other artifacts of the GEPL implementation if needed (FR23). According to the sub-requirements of FR19, the validity is decided by taking the structure and the feature constraints of the associated feature model into account. The same is the case for the GEPL feature stated by FR20. It captures the calculation of the automatic selection and elimination of features, which can be needed if the status of a specific other feature is changed. A simple example is the automatic selection of a feature if an equivalent feature was just selected.

Finally, I want to address the requirement FR22. Its concerned about a way to edit the configuration. Therefore, it describes the implementation of a sub-editor, which is a part of the overall GEPL. A basic characteristic is defined by FR22.1. It states that the configuration editor is dynamic, meaning that its possible to change a diagrams configuration during runtime using it. The other sub-requirements FR22.2 to FR22.6 establish which information the configuration editor publishes to the user. Firstly, it shows the structure (FR22.2) and feature constraints (FR22.3) of the underlying feature model. Furthermore, the status, selected or eliminated, of the configurations features is displayed (FR22.4). Not only the basic status is interesting for the user, also if the status of a feature is locked at the moment. This is stated by FR22.5 and can happen if the structure or the feature constraints are specified in a certain way. For example, a feature with sub-features cannot be eliminated, if at least one of its children is still selected. Concluding, FR22.6 establishes that automatic selections and eliminations are transparent to the user of the configuration editor. Therefore, after each configuration change, automatic consequences for other features of the same configuration need to be calculated, executed and immediately presented to the user.

Abstract Syntax Model

The *Abstract Syntax Model* needs to have two characteristics. Both are subsumed by the requirements in *Table* 3.5. Firstly, the *Abstract Syntax Model* is a model of the domain, the associated GEPL is meant to be used for (FR24). This means that it needs to capture all relevant domain specific concepts, their attributes and relationships with each other. Besides that, the *Abstract Syntax Model* also functions as intermediate representation, according to the requirement FR25. It mediates between the *Concrete Syntax Model*, which captures the structure and graphical information of a diagram to visualize it, and the *Target Metamodel*. The lastly mentioned is a comprehensive domain model, only concerned about the structure of a diagram, not its graphical representation. To be suited as such an intermediate representation, the *Abstract Syntax Model* needs to have special properties to allow co-operation with both of the other mentioned models. On one hand, its co-evolved with the *Concrete Syntax Model*. Details on the co-evolution are listed in the discussion of the *Graphical Representation* concern's requirements. To enable such a co-evolution in an efficient manner it should share commonalities to the concrete variant, for

ID	Description
FR24	The artifacts of the *Abstract Syntax Model* concern shall define a metamodel suitable to the domain, the GEPL is tailored to.
FR25	The artifacts of the *Abstract Syntax Model* concern shall define a metamodel suitable as intermediate representation between the *Concrete Syntax Model* and the *Target Metamodel*.
FR25.1	They shall define the metamodel to derive models from, which can be co-evolved with the *Concrete Syntax Model*.
FR25.2	They shall define the metamodel in order to not contain useless information to the model transformation between the *Abstract Syntax Model* and the *Target Metamodel*.
FR25.3	They shall define the metamodel to derive models from, which are suitable as a source model of the model transformation between the *Abstract Syntax Model* and the *Target Metamodel*.

Table 3.5: Functional requirements of GEPLs associated with the *Abstract Syntax Model* concern.

ID	Description
FR26	The artifacts of the *Target Metamodel* concern shall define a comprehensive and reusable metamodel to the domain, the GEPL is tailored to.
FR27	The artifacts of the *Target Metamodel* concern shall define a metamodel to derive models from, which are suitable as a target model of the model transformation between the *Abstract Syntax Model* and the *Target Metamodel*.

Table 3.6: Functional requirements of GEPLs associated with the *Target Metamodel* concern.

example, similar structural properties of its elements. This is stated by the requirement FR25.1. On the other hand, it should also take the *Target Metamodel* into account. To represent that, the two sub-requirements FR25.2 and FR25.3 exist. The *Abstract Syntax Model* and *Target Metamodel* are related by a model transformation. Consequently, the *Abstract Syntax Model* is meant to capture only the needed information for the model transformation. This usually excludes visual properties, like positions and sizes of model elements. Additionally, the intermediate models need to be suitable as the source model of the model transformation. Equivalent to the already established co-evolution, it useful to design the metamodel of the *Abstract Syntax Model* with regard to the *Target Metamodel*. This is useful as it eases the implementation of the model transformation.

Target Metamodel

The *Target Metamodel* concern is named as it is, because it defines the metamodel of the target representation for the model transformation, already established in the requirement discussion of the *Abstract Syntax Model* concern. One can easily derive the requirement FR27, listed in *Table* 3.6, from that fact. It states that the transformation and the *Target Metamodel* need to be compatible with each other. Usually, the transformation is implemented to suit the *Target Metamodel*. But there is also a second requirement that should be fulfilled by a *Target Metamodel*. FR26 describes how it has to be a comprehensive model definition of the domain the GEPL is meant for. Equivalent to the *Abstract Syntax Model*, this means that all relevant domain concepts and their relations are part of it. Furthermore, it makes sense to demand reusability for the models derived from it as the result of the transformation also is the final product of the modeling process in the GEPL.

Model Transformation

The requirements of the artifacts responsible to implement the *Model Transformation* can be seen in *Table* 3.7. Firstly, the task to transform model instances of a specific metamodel to another one is captured by the requirement FR28 and its sub-requirements. These metamodels are specified to be the *Abstract Syntax Model* and the *Target Metamodel*. How the transformation is implemented strongly depends on the concrete definitions of these metamodels. However, to realize such a *Model Transformation*, there needs to be a specification on how every model element is handled. This includes decisions if and how a model element of the *Abstract Syntax Model* should be transformed. The check if a model element is meant to be transformed is dependent on the current feature configuration of the GE diagram. Therefore the transformation is feature dependent. Besides that, it is also important to implement how exactly a model element is transformed. Defining how an equivalent element of the *Target Metamodel* to one of the *Abstract Syntax Model* is represented, is not a trivial task. It not only depends on the kind of model element to transform. Instead, there are additional influences like the location of the element in the model structure. The same model element might be transformed in different ways if it is part of another element, that groups such together. Furthermore, one model element's transformation can lead to other elements to be transformed as well. This needs to be addressed by the artifacts of the *Model Transformation* concern too. Finally, the last functional requirement FR29 states that the transformation is executed on every save of the currently edited diagram.

ID	Description
FR28	The artifacts of the *Model Transformation* concern shall define how instances of the *Abstract Syntax Model* are transformed into an instance of the *Target Metamodel*.
FR28.1	For every kind of model element, they shall define how the model element is transformed depending on its structural location and relation to other model elements.
FR28.2	For every kind of model element, they shall define if the model element is transformed depending on the feature configuration of the currently edited diagram.
FR29	The transformation is triggered everytime a diagram is saved.

Table 3.7: Functional requirements of GEPLs associated with the *Model Transformation* concern.

3.2 Non-Functional Requirements

This section lists and discusses a selection of non-functional requirements relevant to a GEPL. Additionally, the modularity of such an SPL is demanded to enable checking against the goal of this thesis. The requirements presented in the following are divided into user and developer oriented ones. Furthermore, it is important to note that all of the listed non-functional requirements will be analyzed argumentative, not by measurements. On one hand, this decision is made as empiric surveys on the usability or exact measurements on the performance are not integral to the goal of this thesis. Instead, the focus should be on the modular aspect of GEPLs. On the other hand, some non-functional requirements, like the performance, can not be ignored completely.

3.2.1 User Requirements

Usability

The first presented non-functional requirements in *Table* 3.8 address the usability of the GEs. The aspect of usability is integral to any application meant to be directly used by humans. Therefore its mentioned in multiple books, which classify non-functional requirements [Boehm et al., 1978, Grady and Caswell, 1987]. Furthermore, it is also part of the classification scheme by Roman, coined under the term *Interface Requirements* [Roman, 1985], and Glinz's requirement taxonomy [Glinz, 2007]. In the following, I will present four requirements to analyze the usability of an application tailored to the domain of GEPLs.

To ease the usage of a GEPL, its Graphical User Interface (GUI) should be intuitive and transparent to the user. Additionally, the GUI should be clearly designed and the implementation of the GEPL has to be tolerant to unreasonable user inputs. The requirement `NFR1` addresses the *intuitiveness* of the GE's GUI. The sub-requirements `NFR1.1` to `NFR1.3` elaborate what this means exactly. A user new to the application should intuitively understand what a certain GE function does and how to access it. These function can be GE features present in the palette or context menus, but also basic functions like saving a diagram. By saving a diagram the model transformation is triggered, which shows that there are also indirect interactions to access the GE's functionalities. Visible icons and names of GE functions should be unambiguous to make clear what a certain interaction does. Similar, dialogues should be written in a manner that explains what exact inputs the GEPL expects in a specific situation. Related to the intuitiveness is the *transparency* requirement `NFR3`. On one hand, its sub-requirements describe how the user should always be informed when specific GE actions are executed. The sub-requirements of `NFR3.1` elaborate for which actions this is the case. Basically, everytime a change to the diagram or feature configuration is executed, as well if the diagram is saved, the user should be clear about this. If one change on the edited diagram triggers another one, both should be executed instantly and made visible to the user. The same goes for automatic selections and eliminations of features when altering the configuration. On the other hand, the current state of the GEPL should also be presented to the user during any moment (`NFR3.2`). This states includes the current configuration and editor view. This is espe-

ID	Description
NFR1	Intuitiveness: The GUI of the GEPL should be intuitively understood by the user.
NFR1.1	The user intuitively knows what the GE functions do.
NFR1.2	The user intuitively knows how to access the GE functions.
NFR1.3	Dialogues are written clearly.
NFR2	Clarity: The GUI elements of the GEPL should be easily comprehensible.
NFR2.1	The palette does not necessarily show all available features at once.
NFR2.2	The diagram does not necessarily show all model elements at once.
NFR3	Transparency
NFR3.1	It should always be clear to the user when and what action of the GEPL is executed.
NFR3.1.1	That applies to operations that change the edited diagram.
NFR3.1.2	That applies to changes to the current feature configuration.
NFR3.1.3	That applies when the edited diagram is saved.
NFR3.2	It should always be clear to the user in what state the member of the GEPL family is.
NFR3.2.1	The visible state contains the current feature configuration.
NFR3.2.2	The visible state contains the current editor view.
NFR3.2.1	The visible state contains if and on which element a context menu is opened on.
NFR3.2.1	The visible state contains if there are unsaved changes to the edited diagram.
NFR4	Fault Tolerance: The GE can handle unreasonable user input without compromising its state.
NFR5	Performance
NFR5.1	Time consumption: GE actions only take a reasonable time depending on the frequency of their execution.
NFR5.2	Memory space consumption: The diagrams representations should be saved in an memory space efficient manner.

Table 3.8: User oriented non-functional requirements of GEPLs.

cially important as those define which features can be accessed. If a function present in a context menu is requested, it should be clear on which element the menu is opened on. This decides which functions can be accessed via the context menu. Finally, the information on unsaved changes to the current diagram should belong to the public state of the GEPL too.

Before, I skipped the requirement NFR2. It defines a request for the possibility to hide and show available features in the palette as well as model elements in the diagram depending on the user's demand. This is useful to keep track of the palette and especially big diagrams. A user often does not need to see every available feature in the palette. Consequently, making all visible represents a possible distraction for the user. Giving the user the possibility to hide specific features for the time being avoids that problem. Hiding model elements of a diagram is strongly linked to the functional requirement FR07 in *Table* 3.1. Zooming in and out of model elements allows to not have all model elements visible at once, making effective work with diagrams of many elements possible. To conclude the list of requirements associated with the usability of a GEPL, I need to mention the aspect of fault tolerance (NFR4). This requirement describes how a GE does react to unreasonable user inputs. Examples for such could be invalid edits of the diagram, like giving elements invalid or identical name when that is not allowed. In such a case the state of the GE should not be compromised.

Performance

Table 3.8 also defines performance requirements. That kind of requirement is described by multiple classification schemes [Roman, 1985, Grady and Caswell, 1987, Glinz, 2007]. However, all of those fan out the sub-requirements to a broad variety. Roman [Roman, 1985] includes security and reliability aspects to it. Meanwhile, Grady and Caswell [Grady and Caswell, 1987] also classify resource consumption and efficiency as performance requirements. While this makes sense for some domains, I only want to look at the time and space bounds of the application for two reasons. Firstly all three classification schemes commute in this consideration. Secondly, in my opinion, these are the most important requirements of the performance aspect for the use case of this thesis. That's the case as data security is most likely irrelevant to most users of a GE. Resource consumption and efficiency are only relevant for a GEPL applied to create extreme sized diagrams. I revisit this topic in the conclusion of this thesis (Section 6.3). The reliability aspect, as it is described in [Roman, 1985], contains the availability and integrity of data. While the data integrity is ensured by the correct implementation of functional requirements, the availability does not play a role as the concept of downtime does not fit into the domain of modular GEPL as I see it.

The requirement `NFR5` and its sub-requirements define the performance requests on a GEPL. Firstly, there is `NFR5.1` which addresses the time boundaries of GE actions. Each of those should only take a limited time. That limit depends on the frequency of the action's execution. GE actions that are only executed once or at most infrequently, like creating a whole new diagram, can be more time consuming than other actions. Saving diagrams changes, calculating if a configuration is valid or how the palette looks after a configuration change are executed more frequently. Therefore their execution time boundary is much shorter. Finally, there are GE actions executed multiple times per second, for example, checks depending on the current mouse position. Such actions need to have an extremely short execution time. However, there is also the second sub-requirement `NFR5.2`, which requests that the representations of the created, edited and transformed diagram is as small as possible.

3.2.2 Development Requirements

The development requirements discussed in this subsection address the goal of creating GEPLs which are as flexible in its offered features as possible. This flexibility can be achieved by implementing it in a modular manner. Therefore all three top-level requirements presented in *Table* 3.9 are connected to that objective. Other developer oriented requirements, like testability or portability to different devices, are discussed in the future work Section 6.3.

Platform Independence The first requirement `NFR6` states that the implementation of all artifacts of the GEPL should not be dependent on the operating system it is running on. On one hand, this can lead to a reduction of time and financial costs in the development compared to a process in which a GEPL's implementations differ between operating systems. On the other hand, it is useful as it allows to employ the same feature modules on every system equally, effectively maximizing the reuse potential of feature modules.

Reconfigurability The following requirement `NFR7` and its sub-requirements describe how a modular GEPL's feature set can be expanded (`NFR7.1`) or specialized (`NFR7.2`) in an automated manner. Finally, a feature's implementation can be exchanged, which does not change the feature set's size or the feature's purpose. This is defined by the final sub-requirement `NFR7.3`. All three sub-requirements specify that the described reconfiguration is executed automated. This means that no code has to be changed by hand in the application core or modules to add, remove or replace a feature of the GEPL. Instead, a developer only have put a feature module's artifacts in a beforehand specified location. The GEPL's core implementation then finds the new module and integrates its artifacts automatically. Furthermore, it is important to note that the process of reconfiguration, according to the requirements listed here, happens statically. An expansion of the reconfiguration to be executable during runtime will be addressed in the future work section 6.3.

Modularity To allow such a reconfiguration process in general, modular feature implementations are needed. Furthermore, the main goal of this thesis is to investigate how modularity can be realized for complex applications like a family of GEs. Therefore, the last non-functional top

ID	Description
NFR6	Platform Independence: The GEPL implementation shall be independent of the operating system running it.
NFR7	Reconfigurability: The feature set of the GEPL shall be changeable statically in an automated manner.
NFR7.1	The feature set of the GEPL can be extended statically in an automated manner.
NFR7.2	The feature set of the GEPL can be specialized statically in an automated manner.
NFR7.3	Features' implementations can be easily exchanged statically in an automated manner.
NFR8	Feature Modularity: The artifacts of the GEPL shall be implemented and structured in a modular manner.
NFR8.1	Location and cohesion: The artifacts of one feature are located and grouped in one structural unit.
NFR8.2	Information hiding and encapsulation
NFR8.2.1	The feature modules do not publish its internal attributes and operations to artifacts outside of it.
NFR8.2.2	The feature modules do offer interfaces as own artifacts to publish selected attributes and operation to artifacts outside of it.
NFR8.3	The application core's and feature modules' artifacts shall not reference another features module's internal artifacts directly.

Table 3.9: Developer oriented non-functional requirements of GEPLs.

level requirement NFR8 has a high priority. It states that a GEPL's implementation should ensure that it follows two principles. The requirements NFR8.1 and NFR8.2 are derived from Kästner *et al.* [Kästner et al., 2011]. They formulated two principles to achieve feature modularity. Firstly, there is the principle of *location and cohesion* (NFR8.1), which specifies that one feature's artifacts should be grouped together in a way they build a structural unit. This avoids that associated artifacts are scattered around and ease to find the artifacts to remove or replace when reconfiguring the feature set of a GEPL. NFR8.2 addresses how to handle a feature module's implementation details. Kästner *et al.* described that it is important to differ between the internal implementation and external interfaces of a module. This principle of *information hiding and encapsulation* further states that a feature module can not just make all their internal attributes and operations public as this violates the idea of enclosed modules. That fact is presented by NFR8.2.1. However, such public information would be needed to allow the feature module co-operating with the application core and other feature modules. Therefore, each feature module has interfaces to defines which parts of its artifacts can be accessed from outside of it. Firstly, this practice limits the access to a module's internal information significantly. Secondly, it also eases changing the implementation and especially identifiers of internal attributes and operations. If an identifier referenced by a feature module's interface is about to get changed, the identifier only needs to be updated in the interface. This is acceptable as a code change only leads to altering the implementation in the same feature module. A solution not using references would require to update all references to the changed identifier in other modules code, which decreases the maintainability and creates hard to manage dependencies between feature modules. Finally, these dependencies are also addressed by the requirement NFR8.3. It explicitly forbids that artifacts directly reference internal attributes and operations of modules other than the one they belong to. The same is the case for artifacts of the application core. By defining such a constraint, fixed dependencies between the core and modules as well as between different feature modules can be avoided.

4 Design of Graphical Editor Product Lines

In this chapter, I will present a top-down design approach for dynamically configurable and modular GEPLs. The design approach is meant to be as general as possible, while also following the description of GEPLs defined by the requirements in Chapter 3. Therefore concrete implementation details, e.g. the choice of frameworks, will be omitted. In the end, this chapter will provide a generally applicable and implementation independent guideline to the design of GEPLs. This includes how all relevant functionalities of products in such a PL are implemented while also respecting the domain's non-functional requirements, especially the developer-oriented ones. Two of those specify the modularity of a GEPL created by the presented methodology. The focus of it is put on these modularity and reconfiguration requirements. Section 4.1 addresses the property of modularity and the other three characteristics, originating from the survey in Chapter 2. Following in Section 4.2 is the elaboration on the design approach itself, structured by the concerns of GEs, which are established in *Figure* 3.1. Finally, Section 4.3 reflects on the presented approach, evaluating if all requirements of the GEPL domain can be met by a product of the methodology and summarizing by which concepts requirements can be fulfilled.

4.1 Characteristics

The characteristics discussed in this section are following the classification scheme described in the *Survey* (Chapter 2). Therefore, I will omit to explain their complete nature and possible alternatives. Instead, I will reason on why the properties of *dynamic configuration*, *top-down design method* and *complete modularity* are chosen. The black marked grid point in *Figure* 4.1 visualizes how the presented design approach is classified in the representation established in the *Survey* chapter 2.

GEPL Domain

However, I want to start addressing the yet unmentioned characteristic. It describes in which domain a design approach is meant to be used. Regarding this, it is important to argue why there is the need for a specific GEPL domain and why it might be a problem to create those PLs using more general approaches, such for SPLs for example. There are two main arguments for that. Firstly, multiple general design methodologies need to be combined to create a GEPL following the requirements established in this thesis. Such GEPLs apply an SPL as well as potentially multiple LPLs. As those are dependent on the same feature model, all PLs that are part of a GEPL need to be compatible toward the identical variability model. Beside potential compatibility problems, the definition of multiple PLs leads to a certain degree of complexity overall. Additionally, there is

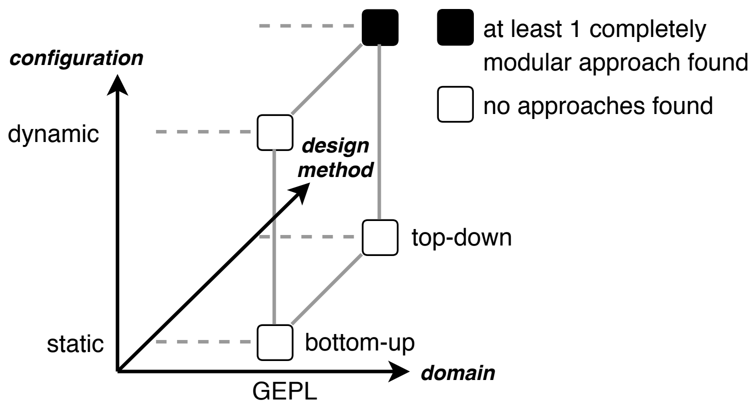


Figure 4.1: An extract of the survey classification including the presented design method.

also the basic complexity of GEs, which comprise multiple concerns[1]. Those concerns should be separated as much as possible and are often implemented in distinguished kinds of artifacts. While an integration of various artifact types is not trivial for a single GE, the challenge is even harder when combining such a situation with the modular PL aspect. In this context, the distinguished artifacts also need to located in a single feature module, still allowing access to them from different points in the overall GEPL. Overall this leads to a high complexity of GEPLs which justifies the need for specialized design approaches in this domain.

Dynamic Configuration

Being able to alter the configuration during the runtime of a GEPL has several advantages. In general, a dynamically configurable PL offers a more flexible way to select a specific product with a desired set of features. The usability, in particular, the comfort and user-friendliness, is increased using a dynamic configuration in comparison to a static one. It avoids restarting the GEPL products everytime a change on the feature model is needed. Besides the general advantage, there are domain specific situations which can profit from allowing to change the feature configuration during runtime. Depending on the definition of GEs, it probably should be possible to edit multiple diagrams at the same time. They might also be distinguished by their configurations, meaning that an isolated feature configuration for each edited diagram exists. To execute such a configuration process for multiple diagrams statically is inconvenient and better realized in a configuration editor associated to the corresponding diagrams' control elements.

Top-Down Design Method

To choose a top-down design method for the creation of GEPLs offers multiple advantages. It is a more generally applicable approach in comparison to a bottom-up process as it can be used to develop PLs from scratch intuitively. On one hand, this offers independence from existing artifacts or full applications and languages. On the other hand, the proceeding allows to develop the feature model and the implementing artifacts in a flexible manner as no constraints of legacy artifacts has to be taken into account. The advantages over a bottom-up process of flexibility and more generic applicability do not come for free. Deriving the feature model and associated mapping from a set of artifacts or a complete product, as bottom-up processes allow, eliminates a lot of work when creating PLs. By using a top-down approach this extra effort needs to be put in too. Overall this constitutes a situation in which implementation effort needs to be weighted against flexibility and universality.

[1]See *Figure* 3.1 in the *Requirements* chapter 3 for more information.

Not yet mentioned in this situation is the initial effort to automatize the variability model generation. To realize such a process, an implementation analyzing a set of components or whole products needs to be provided by former development projects or a third party. While the possibility to use third-party implementations might seem attractive as it reduces effort significantly, it further decreases the development flexibility when being dependent on legacy software which also needs to be compatible with the components structures. Besides that way to avoid the initial effort using a bottom-up process, other needed steps cannot be skipped as easily. Most probably the component's implementation has to be annotated with keywords or tags, e.g. to allow an automatic process taking the semantic of language structure into account. While the initial effort can be compensated by the reuse of the automated processes and annotated components over time, this point of return might not be reached for specific development situations.

When weighing the advantages and disadvantages of a top-down design method against each other, also taking the initial effort of using a bottom-up process into account, I would argue that the flexibility and universality is worth the extra work to put into the manual definition of a feature model as well as mappings between it and the feature modules. I would especially recommend to avoid using the alternative bottom-up method if there is no suitable existing implementation for the feature model generation and the reuse potential of a manually developed bottom-up approach is low. Therefore, a top-down method is applied in the design approach presented. However, an adaption to a bottom-up approach is discussed in the future work Section 6.3 of this thesis.

Complete Modularity

A complete modular PL allows the adaption of it to new requirements or changes in the addressed domain. Additionally, the transition to a domain requiring similar functionalities is possible. The potential reuse of existent modules, either self-developed or gathered from a component market, is another advantage of developing PLs with respect to feature modularity. The reuse can lead to saving implementation effort, while also allowing to use well-tested components. Finally, a clear structure of as isolated as possible components has advantages in respect to the initial development and the maintainability of a software system. Components which encapsulates a maximum of its implementation and only publish as much detail of it as needed, do allow to split the development of feature modules to different teams and developers. They can work isolated on their part of the development process, as long as the interfaces between the components are clearly defined. When maintaining a modular system, debugging is furthermore eased as the location of code implementing certain functionalities and the possible co-operations are well defined by the component's structure and interfaces. If non-essential parts of a software system have to be debugged, it is even an option to remove a responsible module from the system to avoid damage inflicted by the bug to fix. If there is already an alternate module providing the same functionality ready to use, the faulty component can be replaced too. Overall complete modularity offers a lot of advantages that are a reason to include the property for the upcoming design approach.

4.2 Design Approach

In the following, I will present a design methodology for GEPLs. The design approach will apply a top-down process to create completely modular families of GEs which are based upon feature models and can be configured dynamically. According to the definition established in this thesis, a GE is comprised of seven concerns, which are discussed in detail by the Chapter 3. For each of those concerns, I will shortly summarize the tasks of their artifacts, how these artifacts look in general and in which manner they can be structured modular while still providing their functionality. If a concern influences the classification of the presented design approach with respect to the properties of the top-down method and dynamic configuration, considerations towards it will follow. Finally, the feasibility of the design method will be discussed, but only as a brief summary as a far more elaborated feasibility proof is given by the case study in Chapter 5.

4.2.1 Edit Concerns

Graphical Representation

Tasks As a foundation, in this approach, each feature module defines the implementation of one model element type. The artifacts of the first concern are responsible for the representation of the model elements in the concrete and abstract syntax model. The concrete syntax model saves the graphical representation of model elements, while the abstract version of it only stores their structure and properties. Every time a model element is created, deleted or edited, for example, moved or resized, at least one of those two models is subject to change. This constitutes a co-evolution between the models if both are altered at the same time. In such a situation, the implementing artifacts of this concern define how the graphical, as well as the model element's representation in the abstract syntax model, is manipulated. Additionally, sanity checks when changing the concrete or abstract syntax model, have to be provided. They check for mandatory but missing information or other problems which could lead to an erroneous execution of the GE action. Finally, the mentioned artifacts need to implement a function to zoom in and out on levels of an edited diagram, for example by zooming into the inner elements of a model element grouping other ones together.

Solutions Before I address the solutions for the tasks, I want to note that 4.2 visualizes the architecture of a GEPL created with the presented design methodology. The artifacts controlling the representation of model elements in the concrete and abstract syntax model need to provide hooks, which are addressed when the user initiates a GE action to edit a diagram. This means there needs to be a mapping between user inputs realized in the GUI to the mentioned hooks. The operations, referenced by hooks, implement the changes to a diagram's concrete syntax model instance, effectively defining the graphical representation of a model element. It is depending on the edit the user initiated and the feature configuration. Additionally, such operations also implement altering the abstract syntax model instance of a diagram. It makes sense to separate the modification of each of the two models in an own operation, even it is triggered by the same user inputs. This helps to keep the structure of model altering operations clear and allows that one operation prepares and calls the second one. In an example situation, creating a new diagram element firstly triggers an operation to add a representation of it to the abstract syntax model. It prepares all the visible information associated with it and calls a second method to add a corresponding visual object to the concrete syntax model. Overall, the edits on the concrete and abstract syntax model are represented by the *edit* and *co-evolution* relations in the *Figure* 4.2.

The sanity checks for model altering GE actions can also be implemented by the before mentioned artifacts. These checks can be implemented as part of the GE action's operations itself. Alternatively, the hooks, referencing the operations, refer to other methods to execute the sanity checks before triggering them. In the first case, the operations that change the syntax models call the methods, implementing the sanity checks, directly. The second option leads to a situation in which this is not needed, as the mentioned hooks manage to execute the sanity checks. Either way, the *UI* artifacts in *Figure* 4.2 are responsible for the implementation of the sanity checks. Finally, the diagram zoom functionality has to be taken into account. In contrast to the actions mentioned before, it is implemented as a GE action that does not change a model instance. Instead, just the visible sector of a diagram is changed when zooming into or out of a model element for example. To realize this, the user-initiated GE action uses a hook referencing implementing operations. These methods define how to choose the correct fragments of the concrete syntax model instance to show. When zooming in, the chosen fragment is contained in the visible diagram part before the zoom. When the reverse GE action is triggered, the view of the editor returns to original diagram fragment again.

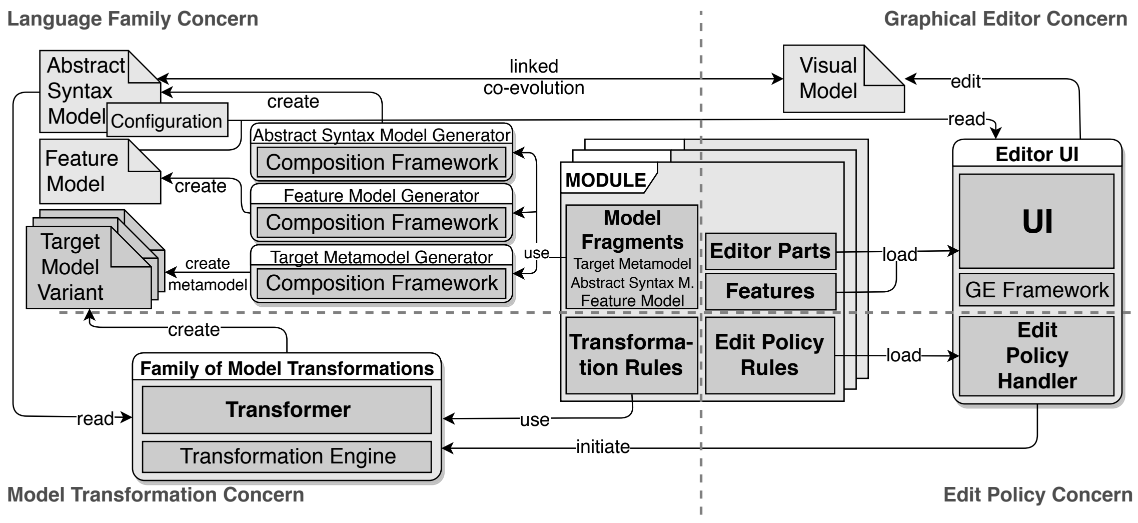


Figure 4.2: The architecture of a completely modular implemented GEPL with the presented approach.

Modularization Following the same criteria for modularity as used in the survey (Chapter 2), it depends on locality, encapsulation and handling of dependencies between feature modules. The first two criteria originate from the principles for feature modularity by Kästner *et al.* [Kästner et al., 2011]. Locality states that all artifacts associated with one feature module are located in a common container. Differing between internal implementation details in contrast to externally published attributes and operations of a component's artifacts ensures the encapsulation of it. Dependencies between non-essential components, which arise when artifacts in one feature module use artifacts' implementations of another one, have to be handled also. Doing this, direct and hard-coded references have to be avoided as otherwise, system specialization would not be possible in a safe manner.

This is part of a constraint set. The constraints describe which artifacts can directly reference other artifacts depending on their *modularity levels*. A visual representation of the set can be seen in *Figure* 4.3. A modularity level describes the location of an artifact. Firstly, an artifact can be located in the *application core*, which is built monolithic and offers no modularity at all. The second modularity level is comprised of *core feature modules*. Artifacts of those are structured in feature modules but offer such basic functionalities, the components are considered essential. System specialization is not intended on them, but their implementation should be replaceable by another module providing the same features and offers the same public interface. The highest degree of modularity is required for generic *feature modules*. These implement non-essential GE functions and are meant to be removable and addable without breaking the GEPL, e.g. by creating unresolvable dependencies. *Figure* 4.3 shows how the modularity degrees are reflecting in the allowed direct references to artifacts on the three levels. As the application core is seen as a fixed code base it can be referenced directly from anywhere. Directly referencing a core module is only accepted, which means that it should be avoided as much as possible, but if it is needed it does not hurt the overall modularity of the PL significantly. This includes references from one core module to another one. In contrast, all direct references to non-essential feature modules are forbidden, also those between different components on this level.

After all those general considerations on modularity in this approach, I want to continue with concern specific explanations. Overall, I presented two types of artifacts in the paragraphs addressing the solution to the concern's challenges. The artifacts of the first type implement GE actions that alter syntax models, while the other ones implement the diagram zoom function. Locality for all those artifacts is not a problem, but to achieve encapsulation and dependency handling is more challenging. A third kind of artifacts, namely *References*, need to be created. They act as

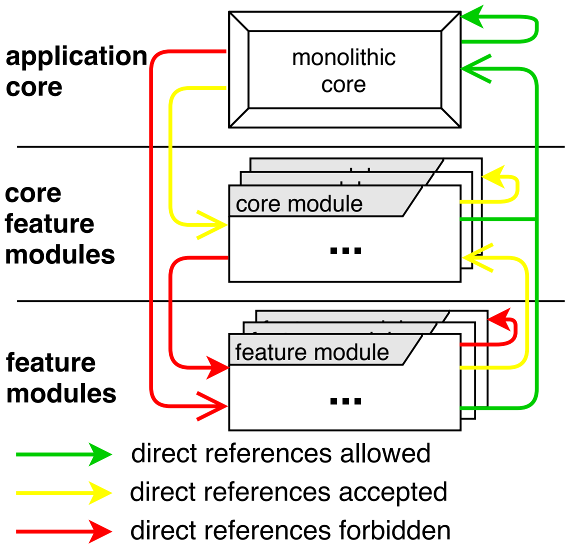


Figure 4.3: Visualization on which artifacts can directly reference other artifacts based upon their modularity levels.

the interface of a component, publishing selected implementation parts, for example, the diagram zoom function or attributes and operations of classes, which define behavior to alter syntax models. They create a level of indirection offering public names of functions and attributes for the internal implementations. Changing the internal identifiers of such therefore only lead to adjustments inside one component, in particular, the implementation itself and the *Reference*. Furthermore, the *References* control which implementation details are public, which allows a high degree of encapsulation while also allowing cooperation between feature modules.

However, the solution as described yet does not solve the problem of components directly referencing each other. In particular, the *References*, which are the target of those relations, are meant to be located in the module they are representing. To solve it, *References* cannot be named directly by an artifact, which is not part of the component, the *Reference* is located. Instead, they have to be found and made usable during runtime. The search process can be realized by using meta information on them, e.g. their supertype, annotations or the existence of a specific attribute. Following these concepts, dynamic instantiation might be needed to be able accessing the *References*. When removing a non-essential module, the implementation as well as the *References* to it disappear. As dependent components do not refer to the *References* directly but apply dynamic loading, such components are not broken by an unresolvable dependency. The dependent component can still work if the dependency that existed beforehand is optional or alternative implementations are used. The process of dynamic loading is represented in the resulting architecture for GEPLs, in particular, by the *load* relation in the *Graphical Editor Concern* quadrant. Overall all modularity criteria are fulfilled.

Relations to Other Properties This concern not only relates to the property of complete modularity, it also plays a role when analyzing the dynamic configuration of the approach. The reason is that the visual appearance of specific model elements can be dependent on the feature configuration, a diagram is associated with. Eliminating a feature in the configuration could lead to hiding some part of it for example. This distinguishes from the already established GE action as the there happen no diagram edits, but only hides irrelevant information to the user, which still exists and can be shown again when needed. The goal to achieve a dynamic configuration now leads to the need of calculating the diagram representation depending on a configuration when not editing a model element. To make this possible, artifacts for this purpose are implemented as a listener for configuration changes. When getting notified about a configuration change, these artifacts analyze the changes and can draw conclusions based on it. For this function, the artifacts of this concern read the configuration of a diagram, which is shown in *Figure* 4.2 by the *read* relation pointing to the *Editor UI*. This functionality is emphasized in this paragraph as it is integral to enable the dynamic configuration.

Feasibility Finally, I want to give a brief discussion on the feasibility of the proposed solutions for challenges related to this concern. The GE actions to change the syntax models, including the sanity checks, seem like rather complex processes as they connect the front end of the GE to its back-end application. A user input perceived by the GUI has to be mapped to hooks which execute sanity checks and eventually apply model altering operations. Fortunately, this functionality is often implemented by GE frameworks and does not need to be implemented manually. The definition of behavior that manipulates models can be treated similarly. While it is possible to implement such behavior by using metamodel dependent languages directly, it is also possible to apply an indirection on it. This indirection is enabled by operations which are provided by the GE framework. The use of a framework is regarded in the resulting GEPL architecture as it contains such an element. Furthermore, it is a trivial task to select suitable diagram fragment for the zoom function as long the concrete syntax model, usually defined by the GE framework, allows nested diagrams. In such a case, it is possible to encapsulate diagram fragments as sub-diagrams. When zooming into a diagram, the inner sub-diagram can be easily selected as newly visible sector of the diagram to edit.

The artifacts implementation according to the modularity requirements is feasible too. An implementation language defining the artifacts of this concern need to allow dynamic search and use of *References*. Both can be realized relatively easy as the analysis of meta information on artifacts is trivial, and structural reflection, which is the base for dynamic instantiation, is included in many programming languages [Malenfant et al., 1996]. The third criterion of locality is also trivial since there are many ways to access artifacts in component containers, reaching from *References* to the definition of extension points. Finally, for the implementation of listeners to feature configuration changes, the well-known observer pattern can be used.

Palette Properties

Tasks This concern's artifacts control the accessibility to GE actions by defining how the palette and the context menus look like. For each of these ways to access GE actions, the visibility and visual representation for possible entries are managed. The visibility of palette entries is dependent on the current configuration and the editor view, which can be manipulated by zooming into and out of diagram fragments. Meanwhile the visibility of entries in context menus dependents on the same editor state parts, but additionally also on the model element a0 context menu is opened. Therefore the definition and calculation of the visibility for context menu entries is more complex than the same processes for palette entries. However, this turns around for visual appearance of GE action's entries. The definition of context menu entries' visual appearance only contains a representing name, while palette entries have a name, an icon and can be associated to a specific palette category if the concern's artifacts deal with the tasks of building the palette as a multileveled structure.

Solutions Just like the first concern, this one is also represented by the *Graphical Editor Concern* quadrant. All its solution can be found in the corresponding module part and the *UI* package. I want to start with the realization of decisions on the visibility and appearance of palette entries. The palette, in general, is created by the application core. This can include defining a structure using potentially nested categories, which have to be clearly identifiable by their names or in another referable way. Furthermore, the application core artifact building the palette might also add the GE actions' entries to the palette. If this is the case, it needs to store a list of all implementing artifacts of the potential GE actions, gathered by dynamic loading. When creating or updating the palette, it iterates on this list, deciding for each GE action to be included in the palette or not. The alternative would be that the implementing artifacts of GE actions can access the palette and add a corresponding entry to it. Both of those variants are acceptable and can be used modular, as explained later. Similar to the build process of the palette, the context menu can be created dynamically.

To enable the implementing artifacts of GE actions to store if and how the actions can be added to the palette, they define an object following the concept of a *FeaturePaletteDescriptor*. It collects information to decide if a GE action is accessible and how it is visually presented in one compact object. The visibility check for entries is implemented via a method analyzing the current configuration as well as the editor view and returning a boolean value as a result. Meanwhile, it can also save the other important information, like the feature's representing name, a reference to an icon file and the category of the palette it belongs to. In a similar way, the visibility and name of a context menu entry for a GE action can be defined. As this information is not enough to justify an own packaging object, like the *FeaturePaletteDescriptor* is, the approach intends to implement the visibility as an isolated operation and the showed name the context menu as a single attribute. It is important to note that this process is based upon the fact that a GE action cannot be part of the palette and a context menu. If this is possible, it makes sense to generalize the concept of the *FeaturePaletteDescriptor*, ignoring the icon reference when making a GE action accessible in the context menu.

In the case of an application core artifact iterating over a list of implementations of GE actions to build the palette or context menu, the core artifact accesses the information towards the palette and context menu properties, provided by the listed elements. When the action implementing artifacts add themselves, this information is only needed locally in the feature module.

Modularization The concerns artifacts to structure in a modular manner are the same the ones described in the first concern of *Graphical Representation*. The reason for this is the definition of the palette and context menu properties of GE actions in their implementing artifacts, which are also subject to the first concern. Therefore, the argumentation for the locality principle in that concern can be followed. To discuss the encapsulation and the dependency handling, a case distinction is needed. In the first case, in which application core artifacts add the entries to the palette and context menus, the properties of visibility and appearance are treated explicitly as public knowledge on a feature module. This can either be realized by making its definition in the implementing artifacts of GE actions public, use a *getter* method for it or create a specific *Reference*[2] for them. While I would recommend using the second or third variant for the cleanliness of code, all three variants can be used and follow the principle of encapsulation. At least as long as publishing the palette and context menu properties is defined explicitly, for example, by a keyword of the programming language. Additionally, the dependency of an application core artifact using a *Reference* or GE action implementing artifacts in a feature module decreases the degree of modularity not. Such dependencies do not involve direct references. Instead, the application core artifacts' stored lists of GE actions' implementations are gathered by the use of dynamic loading. The second case states that GE actions add themselves to the palette. This is perfect for the encapsulation as their palette and context menu properties are completely hidden inside their feature module. Furthermore, direct references of application core parts are always allowed. Therefore, the second case creates no problems related to dependencies between artifacts.

Feasibility Similar to the feasibility discussion of the first concern, the source code of implementing the general palette and context menu GUI is fortunately usually part of GE frameworks or platforms. Consequently, the GEPL developer only curates the content of them. Defining the visibility and visual appearance of GE actions is trivial as it only uses methods, which return a boolean value, and simple attributes. When selecting the solution of an application core artifact, which adds entries to the palette and context menu, the used programming language needs to allow dynamic searches and use of artifacts, for example by structural reflection. In the other solution in which the GE actions add their entries to the palette and context menu on their own, this is not needed but require that these implementing artifacts of the GE actions have access to the palette building process which can be a problem depending on the used GE framework.

[2]Hereby the concept established in the in consideration of the first concern is meant.

Edit Policies

Tasks Edit Policies define if GE actions that create, manipulate or delete model element, can be executed. This decision is dependent on the chosen feature configuration, the editor view, the kind of operation and the type of model element, which is intended to be edited. Besides creating and deleting elements, the concrete and abstract syntax model can also be changed by moving and resizing elements, reconnecting relations as well as editing textual properties. This is only a minimal set of operations on model elements and can be expanded.

Solutions The definition of edit policies is possible in any suitable representation of rules analyzing input data, e.g. a set of chosen features, and allowing an assignment of the rules to tuples of a GE action as well as a type of model element. The analysis can be based on boolean checks for the status of feature in a configuration for example. Furthermore, the chosen edit policy representation has to be defined in a way that satisfiability and consistency checks on a set of policies can be implemented. Finally, its also important to allow the existence of multiple edit policies for one tuple of a GE action and model element type. When combined with the ability to ensure that one policy can not invalidate another one, except the PL developer forces it, this is a foundation for modularity.

Yet unmentioned is the execution aspect of the edit policies. Fortunately, there already are reusable hooks for them. Edit policies are triggered alongside the sanity checks, that are part of the *Graphical Representation* concern. This is a suitable solution as the sanity check, equally to the edit policy ones, are always executed before a GE action, which changes the concrete or abstract syntax model, is about to get applied. The execution itself is managed by a handler, collecting all available edit policies by searching them dynamically. Like presented in the *Figure* 4.2, the handler is a direct part of the *Editor UI*, which makes sense as the **UI!** hooks are used. Furthermore the location of the edit policy rules and the dynamic searching for them is visualized in the architecture. After calculating if a GE action, to execute, corresponds to the tuple of one or more edit policies, the handler combines the boolean statements of all found edit policies to one overall check. The sub-statements can either be connected by an *and* or an *or* conjunction.

Modularization The two feature modularity principles of locality and encapsulation can easily be achieved for the artifacts of this concern. Locality is enabled by the handler dynamically searching for edit policy definitions, which allows to place them in the feature modules they belong to. However, the edit policy definitions are seen as external knowledge about the model element type defined by the feature modules, in particular in which ways they can be edited in a specific situation. Therefore dynamically loading such information does not violate the principle of encapsulation.

The fulfillment of the third criterion is more complex to explain. In the case of edit policies, dependencies are avoided as rules do not reference each other and the application core artifact uses dynamic searching to collect the rules. However, in specific situations, dependencies are created by the relation between the model element type to edit and another one. For example, when the decision, if it is possible to edit a model element of a specific type, is depending on its parent element. In such a situation, there need to be multiple edit policies implemented, one for every possible parent element with a different calculation. On the first glance, this is not a problem as the handler that manages all edit polices combines the rules addressing the same actions and kind of model elements. But looking deeper, a dependency between the model element to edit and the parent element exists. In the situation, removing the feature module, the parent element belongs to, while not deleting the edit policy depending on it, can break the whole GEPL. To avoid this, there are two solutions. Firstly, the GEPL developer can locate the dependent edit policy definition in the feature module, that addresses the implementation of the parent element. Secondly, a more clean solution would be to create a new crossover component encapsulating the dependency. This crossover component can be defined as a sub-module of the parent element's feature module. In both cases, the edit policy is deleted along with the feature module it is dependent on. Applying these solutions, all criteria for modularity can be fulfilled.

Feasibility The foundation for the presented edit policy approach is a suitable language to define the policies. If there is not an existing solution, creating a DSL is an effortful but feasible way to create such a representation. The definition of the concrete edit policies is fairly easy, as the defining the GE action, the model element type and a boolean expression on input variables are trivial given suitable language structures are defined. As the application core artifact handling all edit policies, dynamically searching for those, the programming language in which it is written has to allow that. Checks for the satisfiability on boolean expressions and the consistency of models are well known and solved problems. Finally, the security mechanism, that one rule with the same referenced GE action and model element type can not invalidate another one, can be implemented by using *or* conjunctions during the combination of their boolean expressions.

4.2.2 Language Family Concerns

Feature Model

Tasks This concerns artifacts define a variability model in form of a structured feature model containing constraints, which define relations between features. A minimal set of those relations contains implications and equivalences but could be extended easily by exclusion for example. For the derived configurations of the feature model, a validity check and the calculation of automatic selections and eliminations of features have to be implemented. Such selection and eliminations can be implied by the structure and constraints of the associated feature model of a configuration. Furthermore, a dynamic configuration editor visualizing the structure, the relevant constraints, the precise status of a feature and the results of beforehand mentioned calculations. When using the configuration editor, changes to the configuration have to be signaled to artifacts depending on it. Finally, a standard configuration has to be defined, to be used when a new diagram is created.

Solutions The representation of feature models can be defined by using a specialized metamodel based language. The metamodel has to cover the features itself and a way to define the structure as well as non-structural constraints between them. The implementation of the decisions on validity as well as automatic selections and eliminations strongly depends on the chosen metamodel. Therefore, it makes sense to develop them alongside it. Besides the feature model, the configuration editor is another big artifacts to create. If there are multiple parallel configurations in the GEPL possible as every diagram has an own configuration attached, the associated editor is most useful to be implemented as a sub-editor visually related to the diagram it belongs to. In such a case, a diagram would be opened in multipage editor allowing access the sub editors to alter the diagram itself on one page and the configuration using a second editor page. If the GEPL is limited to editing one diagram at once this effort is not needed and the configuration editor can be implemented separated from the actual diagram editor.

Addressing the configuration editors content, the associated feature model has to be analyzed to generate a visual tree structure. The tree is comprised of the feature names and statuses, indicating if a feature is selected, eliminated or even locked. If it is not locked the feature's status can be changed by the user, leading to calculating the validity and automatic consequences for other features' statuses. The results of such calculations are directly shown to the user, via a status bar and the changed status indicator of automatically selected and eliminated features in the edited configuration. The final step when altering a configuration is to notice listeners, which are described in the discussion of the *Graphical Representation* concern. The initial statuses of the configuration's features are defined in an artifact which is suited to save a set of chosen features. The by this way defined standard configuration is read and reproduced for any newly created diagram, which has an own isolated feature configuration attached.

Modularization The artifacts presented in the *Solution* paragraphs are the listeners to configuration changes, the dynamic configuration editor and the feature model. For the modularization, only the feature model is interesting. While the mentioned listeners are already addressed in the *Graphical Representation* concern, the configuration editor is a part of the application core. How the feature model is modularized is transparent to both. To realize a modularized feature model, it has to be split up into multiple parts, which are located in the components they belong too. These feature model fragment can be merged together to generate one feature model, which contains a representation for all implemented features. In the resulting architecture of a GEPL created with this design approach, the idea can be seen as a module contains model fragment, which have a *use* relation to a *Feature Model Generator*.

In the following, I will present one solution for such a process, which is based on Bagheri *et al.* [Bagheri et al., 2011]. Alongside side to their feature model fragment, feature modules contain artifacts to define two sets feature model constraints. The first set contains feature module cross-cutting constraints, meaning constraints that involve features that are implemented in a different component. The second set defines feature model constraints used to control the merging process for the variability model by referencing all features in the feature model fragments to take into account. The whole procedure, triggered on the startup of the GEPL, begins with combining all rules of the secondly mentioned set of constraints. The resulting rule set is used to merge all fragments together with respect to overlapping parts and feature structures. Finally, the constraints have to be added to the generated feature model, taking the fragment internal constraints as well as the module cross-cutting ones into account.

When generating the feature model this way, the standard configuration has to be treated the same. For that, every feature module defines a part of the standard configuration according to its feature model fragment. These fragments can be combined to a standard configuration, which has to be validated. When components differ in the status of the same feature, implementations to decide on dominant definitions has to be provided. If a created standard configuration is not valid, the before mentioned algorithm can also search for possible alternatives. It is important to note that all artifacts, including the feature model fragments, the sets of constraints and the standard configuration parts, are searched for dynamically when building the feature model and standard configuration.

To evaluate the modularity criteria, I want to start with the principle of locality. The feature model fragments and the other needed artifacts can be located in the components the belong to, which is enabled by the dynamic searching for them. The encapsulation principle is respected as all those artifacts are explicitly seen as part of the public interface of a component and are never used internally by the defining component. The considerations on the dependency handling are split up into two aspects. The first one addresses references to feature model fragments and standard configuration part artifacts in the components. As the process using them is part of the application core, these references cannot be based on direct identifying. Fortunately, this can be avoided by identifying them via a file extension and dynamically searching them. The second aspect to look at is comprised of dependencies that are constituted by module cross-cutting feature model constraints. A feature model fragment `A` is dependent on another one, called `B`, e.g. if a feature in `A` is implied by a feature in the fragment `B`. I such a case, the implication constraint is located in the same component as the fragment `B`. This way, removing the fragment `B` containing component, the constraint is also not part of the feature model anymore. For equivalent features both components define each one of the two implications, an equivalency is comprised of. In such a case, either the components which contain the dependencies can only be deployed together for the GEPL or other solutions are used where constraints referencing not existent features are ignored. Overall all the presented solutions handle dependencies between components introduced by this concern's artifacts. Consequently, all the criteria for modularity are fulfilled.

Relations to Other Properties By its nature, this concern relates to the classification property of dynamic configuration as its artifacts implement a suitable editor specialized for the purpose of editing the configuration during runtime.

Feasibility While tasks like defining a feature model and managing its configuration, including validity and other calculations as well as applying a standard configuration, are not trivial, there are specialized frameworks and tools for such purposes. The configuration editor can be realized using a dialog showing text labels and checkboxes in a tree structure. The status bar, publishing if the chosen configuration is valid, can also be implemented using a text label. All those elements are well known and can easily be combined together. The visualization of automatic selections and eliminations can be made depending on the framework or the tool managing the configuration. Furthermore, the integration of multiple sub-editors into one is possible by implementing control structures as tabs or pages for the overall editor. In the discussion on the *Graphical Representation* concern, I already addressed the feasibility of the listener approach. Finally, the feasibility discussion of this concern's way to allow modularity can be split up. Firstly, there is the process to build the feature model. Regarding this, Bagheri *et al.* [Bagheri et al., 2011], who proposed the process also executed and evaluated multiple case studies for it, proving its feasibility. However, Bagheri *et al.* do not address generating the standard configuration. Fortunately, it is not as complex as creating the feature model modular. Deciding on the dominance of status definitions for features in the configuration as well as deducing possible alternatives by choosing different dominant definitions in various combinations allows finding a valid standard configuration easily.

Abstract Syntax Model

Tasks The abstract syntax model is a representation of an edited diagram which mediates between the concrete syntax model, which is focused on the structural graphic representation of model elements, and the target metamodel. In contrast, the last-mentioned model addresses the comprehensive and reusable domain representation of model elements. Consequently, this concern's artifacts define an intermediate model, which is a suitable domain model, as well as caters to store the structure of model elements. The defined model differs from the concrete syntax model as it does not care for the purely visual properties, like position and size of elements, which are ignored when co-evolving the concrete and the abstract syntax model. Furthermore, it is also different from the target metamodel as it does not put a focus on reusability as much. The definition of the target metamodel, described in the next concern, takes such aspects into account. To get from the abstract syntax to the target metamodel, a model transformation is used, which is also part of an own concern. Overall this means that the abstract syntax model needs to be defined as a suitable partner model in the mentioned co-evolution as well as a source model of the model transformation.

Solutions As the definition of the here discussed intermediate representation is depending on the concrete syntax model[3] and the target metamodel[4], it is hard to give a useful but not too specific guideline for the creation of the abstract syntax model. Too specific guidelines would limit the use of the presented approach. However, it is possible to say the following at least. The definition of the abstract syntax metamodel should recreate the domain concepts in a clear and lightweight manner, e.g. as a type attribute instead of own metamodel elements and big inheritance structures. This way it is possible to make the co-evolution easier, while still deploying a suitable domain model. Using the transformation, the lightweight realization of domain concepts can be turned in a more heavyweight one, based upon metamodel elements. It further eases the co-evolution if the abstract syntax model caters to the nature of the concrete one. This can be achieved by putting a focus on a basic style for model elements with inner and associated elements, which have relations to other model elements. This is a typical definition style for graphical models as the concrete syntax model is one. Finally, I want to note that the intermediate model can be used to bridge big differences, for example, in the structure of a diagrams' representation, between the concrete syntax model and the target metamodel.

Modularization Similar to the considerations on the feature model, to allow a modular definition of the abstract syntax model, it has to be split up into parts which can be distributed among feature modules. Therefore, the modularization solution is represented similarly in the often ref-

[3]It is usually provided by the used GE framework.

[4]The target metamodel is most likely to be developed by or in cooperation with domain experts.

erenced architecture (*Figure* 4.2). While different approaches can be applied in such a situation, I propose the use of delta modeling. This solution states that a model can be modified according to the definition of delta modules. The modifications can add and remove model elements, extend elements by attributes or modify these attributes for example. If such delta modules are located in components, a feature module can define its own abstract syntax model fragment. It is added to the corresponding metamodel when the component is included in the GEPL. In the case of the abstract syntax model, this process is not dependent on a configuration, but instead can be executed once at the start of the PL. This is the case as the intermediate representation of a diagram is not feature dependent. The configuration plays a role in the possibility to edit the diagram overall and during the model transformation, introducing variability into the produced instance of the target metamodel. Therefore the model transformation is and target metamodel can be feature-dependent.

The process to built the abstract syntax model is meant to be like the following. Beforehand the generally allowed delta operations as well as the ones specifically tailored to the abstract syntax model have to be defined. Furthermore, a fixed core of the intermediate model has to be provided. An application core artifact searches dynamically for delta modules, responsible for the abstract syntax model, when starting the GEPL. Those delta modules are sets of delta operations, which describe how the core and by other delta modules added metamodel elements can be extended as well as modified. By doing this, the abstract syntax model is composed depending on the available feature modules. If one delta module modifies or extends the model element of another component, dependencies occur. To handle these, they have to be defined explicitly as a part of the dependent delta module. Firstly, this allows calculating an order to apply delta modules, presumed there are no cyclic dependencies. Secondly, the explicit notation can be used to detect unresolvable dependencies in a delta module, which leads to ignoring the associated component for the whole GEPL. This makes sense in situations where a component was removed on which another one depends on. Its assumed the dependent component cannot work without the removed one. If the dependency is defined in delta module explicitly, such a problem can be detected and handled by not including the dependent component overall.

As the definition of possible delta operations, a abstract syntax metamodel core and the artifact executing the process to compose the metamodel are part of the application core, these artifacts are not further regarded for the modularity evaluation. Remaining only with the delta modules to consider, the dynamic searching allows it to locate them in the components they belong to, but be still accessible. Therefore the principle of locality can be followed. Encapsulation of the components is also ensured as the delta modules do not have a purpose inside the feature module and can be considered as marked for external usage explicitly. When handling dependencies between component, on one hand, direct references are avoided by applying direct searches for delta modules. On the other hand, dependencies are explicitly defined in the dependent delta modules allowing to detect unresolvable ones and avoid including them into the GEPL.

Feasibility Addressing the feasibility of the presented solution, to define metamodels is a widespread task which is supported by many frameworks and tools. The given guidelines are viable and can be combined as the usage of an attribute for domain concepts does not limit the definition of model elements in the style described by the guideline. The delta modeling approach can be implemented by using specialized frameworks for it. Such a framework usually provides generally applicable delta operations, allows to define specialized ones and applies the changes specified in the delta modules. If it does not provide a way to calculate the order of execution for the delta modules, such a function can be implemented manually. A dependency graph, which is built by analyzing all delta modules and searching for defined dependencies, is helpful to achieve this. Via the graph, cyclic dependencies can be found and an order can be calculated if no cycles are found. Furthermore, unresolvable dependencies can be detected to avoid breaking the PL.

Target Metamodel

Tasks Similar to the abstract syntax model, the target metamodel is a comprehensive domain model. Meanwhile, the focus of the target metamodel is not mediate between models, but be a reusable end product of the work with the GEPL. It is created by a transformation, starting out from the abstract syntax model. Therefore it has to be defined in a way that it is possible to derive model instances from, which are suitable target models of the mentioned transformation.

Solutions I want to begin with the task of defining the target metamodel in a manner that one can derive reusable end products of it when working with the GEPL. To make that possible, *accessibility* and *compatibility* should be kept in mind when design decisions are made for the metamodel. Firstly, a visualized, clear and logical structure of the metamodel increases the accessibility significantly as a user can understand the background of the created model instances easier. The model instances should further be visualizable outside of the GEPL it was generated in. This property would enable readability of the produced metamodel instance without being dependent on the concrete syntax model and the associated GE framework. After the user understands the created model, compatibility to framework and tools allows to effectively reuse it.

Furthermore, the target metamodel has to be a comprehensive domain model. As domain modeling is a large scientific field, a discussion on how to define a useful domain model would go far beyond the scope of this thesis. However, a good foundation for a successful definition of the target metamodel is the cooperation of a PL engineer with domain experts, to profit from the domain knowledge while working software solution oriented. More specific to the GEPL domain, it can be helpful to differ between the involvement of domain concepts in the abstract syntax model and target metamodel. While the first one can rely on a lightweight solution, for example, a simple type attribute, I would recommend using a more flexible solution for the target metamodel. Such flexibility can be achieved by defining domain concepts as metamodel elements, which can have own attributes, relations and inheritance structures. By developing the model-to-model transformation with respect to the abstract syntax model and target metamodel, it can be ensured that both the source and target models instances are suitable.

Modularization To allow modularity for the target metamodel, it can be composed out of fragments defined in the components of the GEPL. This means that these delta modules define how a model is changed, using delta operations, and on which other delta modules they depend on. As the target metamodel describes the end representation of a feature configuration dependent model transformation, it can make sense to compose the metamodel also dependent on a configuration. If this is not needed, the composition can be implemented in exactly the same way as it is shown for the abstract syntax model. Independently of the exact way to modularize the target metamodel generation, the general process can be seen in *Figure* 4.2. The architecture shows how model fragments are *used* by a generator to *create the metamodel* for the *Target Model Variants* of the model transformation. In the following, I will only present the newly needed implementation for a feature-dependent model composition.

Overall there two options to implement this. The first one states, that there is one metamodel used for multiple opened diagrams. Before every transformation start, the current relevant configuration is analyzed and delta modules, found by dynamic searching, are executed on a fixed metamodel core with respect to the configuration. Concluded, the target metamodel is suited to the configuration, the model transformation uses too. When transforming any diagram with a different feature configuration the process is repeated. Just as described for the abstract syntax model, the execution order can be decided using a dependency graph. The second possible solution proposes that every diagram has an own copy of the target metamodel associated, which is always adjusted to the diagram's configuration. When altering the feature configuration, the delta modules, which reference a status changed feature, are executed on the target metamodel copy belonging to that diagram. When transforming the diagram, its associated target metamodel is used to derive the transformation target from.

The advantage of this first option is, that only one metamodel has to be saved and managed, while rebuilding the target metamodel every time a transformation is started with a slightly altered configuration can lead to a high calculation load resulting in bad response times of the GEPL. Unfortunately, this can not be avoided when only one target metamodel should be used for multiple opened diagrams with isolated configurations. Otherwise, configuration changes made across multiple edited diagrams would be propagated to the one target metamodel artifact, such as all those configuration changes have to be tracked and reversed regularly, resulting in an ineffective and error-prone solution. Meanwhile, this problem can be avoided by the second option completely. As delta modules are applied on the target metamodels everytime a configuration is altered, only small and needed changes to the metamodel happen at once. This is a big advantage, but there is also a disadvantage. Managing and persisting multiple metamodel copies for each created diagram is not desired as there is a higher effort to maintain the metamodel copies, in contrast to the first option using only one target metamodel artifact.

Locality, the first criterion for feature modularity, can be achieved as delta modules are searched for by the application core artifact controlling the process to change the target metamodel during runtime. The delta modules for the metamodel are only used outside of the defining feature module, allowing them to be explicitly seen as part of the external interface of a component. This fact combined with the dependency management, based on explicitly defined relations and a dependency graph, presented in the *Abstract Syntax Model* concern, leads to fulfilling all criteria for modularity of this concern's artifacts.

Feasibility Frameworks can be applied to define the metamodel and visualize it as well as the end product models created by the GEPL. Visualization of the metamodel is usually provided by the framework used to define it. The graphical representation of the target metamodel instances independent of the GEPL's concrete syntax model requires a separate tool which can be implemented significantly less burdensome in comparison to the GEPL. Reasons for this are the absence of the PL aspect, as a complete version of target metamodel can be the applied for it, and the read-only limitation on the diagram's end product. Compatibility of the created target metamodel instances to supporting tools can be encouraged by choosing a common base for the definition of models, which is also widely spread in existing tools. Basic consideration of the feasibility of delta modeling can be found in the corresponding paragraph of the *Abstract Syntax Model* concern. Making the delta module's operations dependent on a feature configuration is a feasible task, as the checks regarding this can be either implemented in the delta modules itself or in the application core artifact controlling the process they are meant to be used for. While the language applied to define the delta modules might not offer a way to analyze a configuration, the second mentioned artifact is most likely to be implemented in a programming language capable to do so.

Model Transformation

Tasks The artifacts of this concern enable a model-to-model transformation which generates a target metamodel based representation of a diagram. It is triggered when saving a diagram and starts out from the diagram's abstract syntax model. Additionally, the relevant feature configuration plays a role as it decides if a model element is transformed or not. Overall such a proceeding makes it possible to introduce variable end products of the GEPL, based on a feature independent intermediate representation.

Solutions To define if and how model elements of the abstract syntax model are transformed to their equivalent representations in the target metamodel, I propose the use of transformation rules. Such a rule defines which element of the source model is addressed by the rule and from which target metamodel element its transformed representation is derived of. Furthermore, it can specify a boolean expressions, checking if the rule should be executed or not. This can be dependent on sanity checks, the model element's structural environment, relations to other elements and finally also the chosen feature configuration. The third part of a transformation rule describes its actions itself, e.g how attributes of the abstract syntax model element are handled.

The whole transformation process is triggered when saving a diagram. When this happens, an application core artifact executes a base rule which transforms the overall abstract syntax model by delegating the transformation tasks for specific model elements to the suitable transformation rules. The specific model elements are defined as children of the top level model element of the abstract syntax model, so the base rule can iterate over them. For model elements that group other elements, the same procedure is applied to transform their children elements too. Finally, the same pool of transformation rules is used in situations where a relation does reference its start and end targets. This way the referenced model elements can be transformed too. However, before a rule is executed, its boolean expression is evaluated, to ensure the transformation rule's action should be executed or not, e.g. as a certain feature in the configuration is not selected. In the resulting architecture for GEPLs (*Figure* 4.2), the transformation process can be seen. The initiating application core artifact, the *Transformer*, has three relations. He *uses* the transformation rules and *reads* the source model instance, with a configuration attached, to *create* a *Target Model Variant*.

Modularization To modularize the above-described solution, the model element specific transformation rules are located in the components in which the model element they transform is addressed. The base rule as well as the transformation starting artifacts are part of the application core and therefore not relevant for this modularization. When starting the transformation of an abstract syntax model instance, the mentioned application core artifact searches dynamically for all transformation rules in the components. In the beforehand mentioned architecture, this is represented by the *use* relation in the *Model Transformation Concern* quadrant. The found rules are made accessible and executable for the base rule as well as for each other. This means that the base rule and each transformation rule from a feature module can trigger the execution of each other components' rules. If this is done, the process of the transformation described in the solution paragraphs can be used, while also enabling modular structures of this concern's artifacts.

To prove the modularity, I will follow the established criteria for it. As all but one of the transformation rules are dynamically found and made accessible for all rules, they can be placed in their corresponding component. Therefore, the principle of locality can be followed. Similar to the model fragments of feature, abstract syntax and target metamodel, the transformation rules are explicitly meant to be used outside of the component and offer no internal service. Therefore, they do not blur the line between internal implementation details and the external interface of a feature module, leading to not violating the encapsulation principle by Kästner [Kästner et al., 2011].

Finally, the dependencies of components, introduced by relations of this concern's artifacts, are either avoided or handled. Fixed direct references between the application core artifacts initiating the transformation and the components' rules can be avoided by dynamic searching them. However, they still have to be made accessible and executable for the base rule, which can be managed by creating those references according to the list of found transformation rules during runtime. The dependencies between different transformation rules in components can be handled in two ways. Firstly, they have to be defined explicitly in the dependent rule. This way the dependency graph solution, presented in the consideration of the *Abstract Syntax Model* concern, can be applied to avoid breaking the GEPL when removing feature modules. Secondly, rule inheritance is a useful tool to handle dependencies. They allow to split up rules specialized for one kind of abstract syntax model element. By doing this, situations, in which such model elements are transformed differently depending on their parent element, for example, can be implemented in multiple rules which can be located in different components. The rules can be placed in the feature module that addresses the parent element the rule depends on. By doing this, it can be ensured that the rule is not taken into account anymore, when removing the component it is dependent on. For a cleaner solution, the rule can also be located in own feature module, which is a sub-component of the parent element's one. Overall all criteria for feature modularity in regard to this concern is fulfilled.

Feasibility The general process of transformation execution using the base rule and model element specific rules is supported by transformation engines. They implement the syntax and semantics of a language to define transformation rules, auxiliary variables as well as methods. As the base rule is an application core artifact, working on the integrated abstract syntax model elements, it might seem strange that it can trigger the transformation of any model element. This is only possible as it does not take their structure or attributes into account. Instead it just strictly iterates over all children of the top-level element of an abstract syntax model instance and treats every model element the same. The model element specific treatment is defined in the transformation rules, one can find in the components. The ability to dynamically search transformation rules is dependent on the programming language implementing the application core artifact that initiates the transformation. Meanwhile the generation of references in the base rule strongly dependent on the choice of the transformation engine. It is possible that the transformation language does not allow to add references to the base rules. However, there is the possibility of using string manipulation to generate a new artifact, which is comprised of the normal code specified by the base rule and the added references. Finally, the feasibility of the dependency graph solution is discussed in *Abstract Syntax Model* concern's paragraphs, while the rule inheritance solution can only be applied when the transformation language takes such a concept into account.

4.3 Discussion

4.3.1 Techniques

In the following, I will summarize the design approach by giving a brief overview of the techniques applied in it.

Code Reuse This paragraph does not address the reuse of components enabled by feature modularity. Instead, it focuses on the *use of existing solutions in form of frameworks, tools and DSLs*. The design approach mentions such reusable implementations as a factor for feasibility in all seven concerns. Starting with the GE framework, there can also be transformation and edit policy languages. Modeling frameworks can be used for the definition as well as the composition of metamodels, while the feature model and its configurations can be managed by frameworks or tools. Using an existing solution for a part of the problem to solve can potentially decrease the solving effort significantly, but also introduce new limitations too. Furthermore, a dependency on the framework, tool or DSL can be problematic. On the whole, reusing code in the form of frameworks and similar means is an important part of this design approach for GEPLs when the choice of the framework, tool or DSL is well deliberated.

Working with Multiple Models The elaborated design methodology is based on three different representations for one diagram. They are quite different, as the concrete syntax model, saves the visual properties of a diagram and can usually be seen as fixed by the GE framework. In contrast, the target metamodel is a domain specific end product of the editing process, meant to be reused. Consequently, the conceptual, structural and content-related differences between the models are significant. Therefore, the first technique, I want to present in this paragraph, plays a role. It states that the use of an *intermediate model* makes it possible to effectively bridge between the two distinguished diagram representations. The abstract syntax model achieves this by following the style of the concrete version, putting a focus on model elements with inner and related elements, while also taking the domain concepts into account. When working with three different diagram representations, it must a possible to create one of another. To realize that, two techniques are applied. Firstly, *co-evolution* which alters multiple models in parallel in an equivalent way. As the co-evolution is triggered every time a diagram is edited, it is characterized by many small calculations, which should not be too complex. More powerful is the approach of *model transformation*, which is initiated not as often as the co-evolution. By using a model transformation, one accepts that instead of many small calculations, one costly process is used. However, for both approaches, there are purposes in the presented design approach for GEPLs.

Techniques for Modularity The foundation for feature modularity is *splitting up an application* into multiple distinguished features in a systematic way. For a GE creating models, this can be done by handling each model element in an, as much as possible isolated, feature module. By the nature of the situation, this distribution of artifacts usually is equivalent to the definitions in the feature model. One step further, it also makes sense to *split up one artifact* associated to a feature and locate the resulting artifacts in different feature modules. This allows implementing features depending on other ones. An example of such a situation is comprised of a sub-feature module defining its behavior depending on its parent module. The same can also be used for components which are not in a structural relation. In this case, the next technique of *crossover feature modules* between two components makes sense. In them, the relation of the two modules is encapsulated. This is also reflected in the facts that it is dependent on both related component, meaning that removing one related component leads to also not including the crossover feature module to the feature set of the GEPL.

To manage those dependencies between components, another technique mentioned by the design approach is applicable. By *defining dependencies explicitly*, it is possible to derive a *dependency graph* automatically, which helps to detect unresolvable dependencies and avoid including components with those dependencies into the runtime of a GE, as they would break it. Furthermore, it is a foundation for the next technique of *composing models*. The presented design approach describes merging feature model fragments and applying delta modules for the abstract syntax model and target metamodel to realize the composition.

Finally, two recurring techniques are *dynamic searching* for artifacts by supertypes, tags or file extension and *Reference* classes. The first technique enables locality, avoids dependencies by direct references and allows to implement centrally controlled processes, like the generation of the palette, in a decentralized manner. The *References* act as interfaces for a component to differ between internal implementation details and selected external knowledge on the component's behavior.

4.3.2 Evaluation

In this subsection, I will evaluate the presented design approach regarding two aspects. The first one addresses how well the design methodology takes the GEPL domain's requirements into account. This allows assessing if the approach can be used to solve the problems in the domain. Following this, I will briefly discuss the overall feasibility of it.

Accordance with Requirements

The following paragraphs will not provide a complete analysis on how every single requirement is met as such a proceeding is not beneficially applicable for a general design approach independent of an implementation. Instead, I will discuss how the given requirements influence the design decisions in a general manner. The *Tasks* and *Solutions* paragraphs in every concern's section follow the functional requirements for the corresponding concern. Therefore the elaborated solutions in the secondly mentioned paragraphs, often involving multiple options to choose from, do provide ways to implement GEPLs fulfilling the given functional requirements.

When addressing the non-functional requirements, it is noteworthy that four of the eight top-level requirements strongly depend on the implementation details, that are not part of the design approach. The requirements *Intuitiveness, Transparency, Performance* and *Platform Independence* are these four. Intuitiveness and transparency are properties of the GUI, which is only described in terms of functionality, e.g. for the configuration editor. These specifications do not contain information if dialogs, GE function names, status bars or other GUI elements conform to the mentioned requirements. However, the approach does not disable implementations of such elements in any way. The requirement of *Platform Independence* is strongly dependent on the chosen technology base, frameworks and tools. By the nature of a general approach, such choices are not included. Finally, implementation details, which should not be specified by a design approach like this, are also extremely important to assess the requirements addressing the performance. Similar to the conclusion on the other two non-functional requirements yet mentioned, the design approach on GEPLs does not limit the choice of the technological foundation, frameworks and other implementation details. Therefore platform independence is not hindered and performance limiting bottlenecks can be avoided.

The remaining non-functional requirements deal with the clarity of the GUI, fault tolerance, reconfiguration and feature modularity. Clarity, how it is defined by the requirement analysis in this thesis, can be achieved by functions described in the design approach. Therefore, it can be discussed here in contrast to the other GUI properties. In particular, it defines how not all palette entries and model elements should be visible all the time, if not wanted at least. For palette entries, the concept of categories offers a simple solution, as long as entries can be hidden by folding up a category. Via the zoom in and out function the user controls which model elements are visible at any time. Continuing, the next requirement of fault tolerance is taken into account by the sanity checks and edit policies which are executed on user inputs. They avoid triggering GE actions compromising a GEPL state. The last two requirements *Reconfiguration* and *Feature Modularity* are part of the design approaches elaboration in this chapter directly. The *Modularity* paragraphs in it describe how the beforehand presented artifacts and solutions can be implemented in a modular manner. Additionally, they evaluate the feature modularity using criteria following the *Reconfiguration* and *Feature Modularity* requirements. The criteria contain the principles by Kästner *et al.* [Kästner et al., 2011] and avoiding direct references between certain artifacts. Furthermore, dependencies between components have to be handled automatically, which is the foundation for system specialization, a part of the *Reconfiguration* requirement. As the evaluation gives that all criteria for modularity are fulfilled for all seven GE concerns, the same goes for the mentioned requirements.

Of the overall 37 top-level requirement, 33 are taken into account by the presented design approach. Solely four non-functional requirements are not directly addressed, but also not blocked by the design methodology.

Feasibility

In the elaboration of the design approach, one can find the *Feasibility* paragraphs, which discuss if the presented solutions and modularizations can be realized. However, this is only the isolated view on the feasibility. It is still due to assess if the artifacts presented in the design methodology can cooperate which each other in a modular manner. To realize that, the cooperation between artifacts of GE concerns has to be analyzed, to be able checking every found collaboration for feasibility. *Figure* 4.4 illustrates them by connections between the corresponding concerns.

Starting out with the centralized concern of the *Feature Model*. The reason for its collaborations is the feature dependency of artifacts belonging to the other concerns. Assuming every diagram has an own configuration attached, it is easily possible to read it from the diagram. This way the artifacts adding palette as well as context menu entries, executing edit policies, the model transformation and the composition of the target metamodel can access it. Meanwhile, the cooperation with the *Graphical Representation* concern is solved by the already established observer pattern. From a modularization standpoint composing the feature model at the start of the GE is transparent to the mentioned artifacts and the *Graphical Representation* concern. The artifacts addressing the graphical representation also have many other collaborations. Inter alia, they define their visibility and visual appearance in the palette as well as the context menu. When adding palette entries this information might be accessed by a *getter* method. Furthermore, the mentioned artifacts co-evolve the abstract syntax model alongside the concrete version. This is realized by code in the *Graphical Representation* artifacts altering an abstract syntax model instance, which was created with its associated diagram. Finally, the artifacts of the *Graphical Representation* concern also offer the hooks

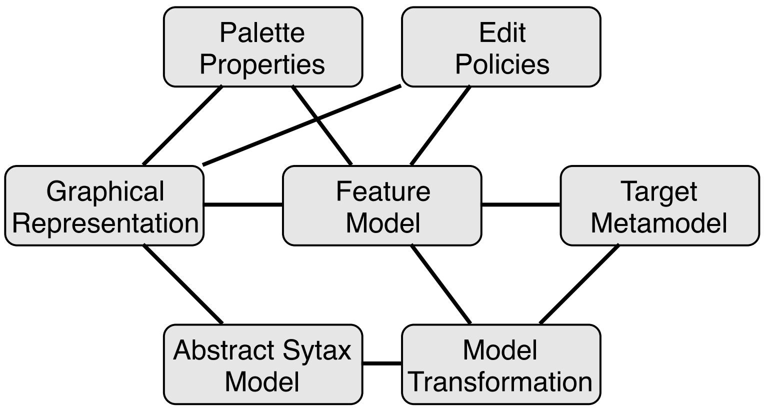


Figure 4.4: Cooperation between artifacts in regard to the concerns they belong to.

to trigger edit policy evaluations. To start this process, the handler of all edit policies is referenced in the artifacts dealing with the graphical representation. For all these collaborations the location of the artifacts implementing the graphical representation and edit policies is not relevant, which allows modularity. The last two collaborations are very similar. They describe how the *Abstract Syntax Model* and *Target Metamodel* concerns are connected to the model transformation, as they define the source and target model representation of it. The collaboration is implemented by the transformation rules which reference both metamodels, while the composition process of both is transparent to the rules.

These considerations allow to assess the design approach as feasible as the concern internal solution as well as the cooperation between artifacts of different concerns seems to be feasible. In combination with the results of the investigation if requirements of the GEPL domain are taken into account, it can be evaluated as successful. Overall, I conclude that the top-down design methodology is a feasible way to create dynamically configurable GEPLs in a modular way. Meanwhile, the case study in Chapter 5 can reinforce this statement.

5 Case study: Modularization of a Family of Graphical Editors

In the following, a case study will be presented to show the usability of the formerly introduced design methodology for modularized GEPLs. The case study will be executed by modularizing the FRaMED SPL, an existing family of GEs. A brief explanation of the existing GEPL can be found in the *Background* section (5.1). In the same section, other relevant technologies used by FRaMED are concerned, including a family of Role-based Modeling Languages (RMLs). The next section (5.2) presents the implementation of the case study, structured by the already established concerns of GEs. Finally, the realization of the modularization will be discussed by checking it against the requirements defined in *Chapter* 3. Moreover, limitations of the modularization, as well as possible solutions to those, will be highlighted.

5.1 Background

This section presents the conceptual and technological base of the case study. It can be split into three parts. Firstly there is the *Compartment Role Object Model*, a family of Role-based Modeling Languages. Following is the summary of the existing not modularized GEPL of FRaMED. Furthermore, some other technologies, mostly frameworks, are briefly introduced. To minimize the effort of the modularization the technological base should be reused as much as possible. Of course, this can be a challenge when modularizing, since nearly none of the reused technologies are optimized to be modularized. Consequently, the capability to be used in a modularized application will be discussed for each of the reused technologies here too.

5.1.1 *Compartment Role Object Model*

One big part of the conceptual *and* technological base of the FRaMED SPL is the aspect of role-based modeling. Therefore, an introduction into the conceptual view of Role-based Modeling Languages as well as the technological background of the *Compartment Role Object Model* is following. The CROM is a concrete implementation of a family of RMLs.

Role-based Modeling Languages

Roles are a concept going back as far as 1973 [Bachman, 1973]. Later, Bachman *et al.* show the use of roles for data models [Bachman and Daya, 1977], an early mark of role-based modeling. They used roles to capture the context-dependent as well as the collaborative behavior of objects. These characteristics are still known for roles today. In fact, the role concept is often seen as

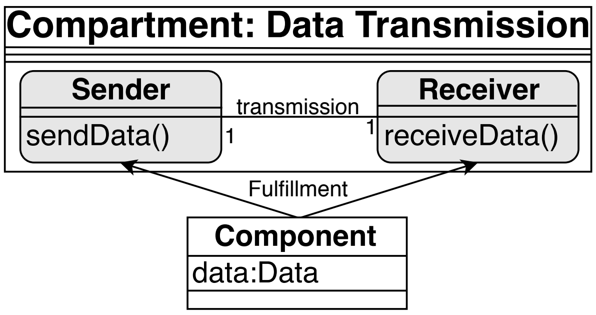


Figure 5.1: Example for context and roles: Data Transmission using a one-to-one sender and receiver relationship

a way to "tame the increased complexity and context-dependence of current context-adaptive, distributed software system" [Kühn, 2017] today. The application of roles can also be found in long known modeling languages, e.g. the Entity-Relationship Model (ER) [Chen, 1976] and the *Unified Modeling Language* (UML) [Rumbaugh et al., 1999]. The usage of roles in this domain modeling languages can be very complex and might lead to counter-intuitive modeling [Kühn et al., 2014]. Therefore the concept of roles is not used to its full potential. Consequently, more specialized RMLs are needed to enable this.

To develop such an RML, one need be clear about the characteristics of roles. This is a challenge since there is no common agreement about the abilities and limitations of roles [Kühn, 2017]. One solution to this problem, a family of RMLs, is implemented by CROM (5.1.1). A common base for a definition are the three natures of roles. Roles have a behavioral, a relational and a context-dependent nature.

Behavioral Nature Objects, that play a certain role, can execute own as well as a role's behavior, using attributes and operations of the object and the played role. Furthermore, there is no limit on how many roles of different kinds an object can play at the same time. Meanwhile, a role can also be played by multiple objects of different classes. Which exact roles an object can play is defined by *fill-relations*. These are drawn between classes and role types to signify that a class instance can play a role of that types.

Relational Nature Roles can be part of relationships. This enables objects to acquire relationship specific properties by playing a role. Just as the *behavioral nature* this is an integral part to ease the separation of concerns for classes. The relational and behavioral properties for different interactions can be distributed by implementing them in different roles. This works since objects can play as many roles as they want at once.

Context-dependent Nature Roles and relationships are context-dependent. The context offers a definitional boundary for both. As this case study is based on [Kühn, 2017], the context will be referenced by the term *Compartment* from now on. This makes sense as the term *context* is heavily overloaded. Furthermore, *Compartments* are special since they are not only contextual boundaries but also implement attributes and operations. Additionally, they can play roles, defined either in their own or in a foreign context.

In the following, I will provide an example to illustrate the meaning of these natures. A diagram of it can be seen in Figure 5.1. It shows a simplified version of a one-to-one data transmission. In this example, the objects of a class `Component` can play two roles: `Sender` and `Receiver`. The class provides the data to transfer, but not the operations to send or receive such data. The behavioral as well as relational properties for the transmission are implemented in the already mentioned roles and the relationship `transmission`. The roles are located in a Compartment `Data Transmission`.

The *behavioral nature* is represented by the `Fulfillment`-relations drawn from the class `Component` to the roles. When the `Component` plays one of the two roles, it can execute the `send` or `receive` operation implemented in those. This way, the concern of creating and managing data is separated by the transmission concern, as the first mentioned concern is implemented in the class. Meanwhile, the transmission is encapsulated in the Compartment and its roles. Furthermore, the concerns of a sender and of a receiver are distinct too. Without roles, all those concerns would be concentrated in the single class of the component. However, one object can also play both roles at the same time, meaning being a sender *and* receiver. This is useful as one object could be in more than one data transmission at once. These different data transmissions are symbolized by the relationship `transmission`, which spans from exactly one sender to one receiver. Finally, the meaning of the *Context-dependent nature* can be explained by imaging different types of data transmission. There could be different security levels using different data transfer protocols for example. These distinct versions of a data transmission could be captured in different compartments, in which the sender and receiver roles implement their operations differently. It could be also possible that in another data transmission context not only one-to-one transfers are allowed. This would be modeled by using other relationship cardinalities than `1`.

Beside the natures, there are two more characteristics of roles to be addressed. The first one is the fact that they can be *constrained*. This includes role constraints as well as related constraints like these of relationships and other elements in role models. *Role constraints*, a range from one to infinity($*$) for example, are occurrence constraints. That means that if a role in this context is played, it has to be played by a number of objects in that range. On the other hand, the *relationship constraints* specify how many role players take part in a relationship, at both ends of the relationship. Concluding, the roles are also *anti-rigid*. Rigid instances of an class belong to this class as long as they exist. They can not change their affiliation to it without being reinstantiated. In contrast, playing a role does not create such an affiliation. An object can change the roles it is playing, but the role does not cease to exist. Therefore objects are rigid, while roles can be classified as anti-rigid.

However, these natures and characteristics are coarse-grained and not directly usable for a definition of an RML. Fortunately, there are papers that addressed that problem. Steimann elaborated features of roles [Steimann, 2000] listed as the first 15 features in Figure 5.2. This features can be associated with the *behavioral* and *relational nature*. The features 1, 10, 11 and 13 capture the aspect that objects acquire properties and behavior when playing roles. Defining how roles can be fulfilled in general are the features 3, 4, 5, 7, 8 and 9. However, the features 6 and 12 show that roles can be constrained. While the former mentioned features are part of the *behavioral* nature of roles, the features 2 and 13 can be seen as *relational* features. The 14th and 15th feature represent a contradiction. This is a problem of perspective. From a conceptual view, an object and a played role should be indistinguishable. Technical, this would be a problem as an execution engine needs to be able to separate the identity of a player and its filled roles [Kühn, 2017]. Herrmann proposes the idea to use different operations to check equality, to solve the problem [Herrmann, 2007]. A more elaborated discussion about the topic can be found in the formerly mentioned paper by Kühn.

The extended list of features by Kühn also represent the *context-dependent nature* of roles. This is realized by the features 19 to 27. The features 19 and 21 describe how roles have to be part of at least one context, while other features mainly talk about the abilities of Compartments. Meanwhile, another focus is put on constraints in role models. The features 16 to 18 describe constraints affiliated in the *relational nature*, but also introduce the need of role group constraints. Lastly mentioned ones are used to group roles together in a way such as *fill-relations* can target these collections of roles. Additional, role group constraints can limit how many roles in the targeted group have to be played when fulfilling the group. Concluding, this list of RML features is suitable to describe the fine-grained characteristics of roles. It is further used as a base to construct a feature model for a family of RMLs, which is presented in the following as part of CROM.

1. Roles have properties and behaviors
2. Roles depend on relationships
3. Objects may play different roles simultaneously
4. Objects may play the same role (type) several times
5. Objects may acquire and abandon roles dynamically
6. The sequence of role acquisition and removal may be restricted
7. Unrelated objects can play the same role
8. Roles can play roles
9. Roles can be transferred between objects
10. The state of an object can be role-specific
11. Features of an object can be role-specific
12. Roles restrict access
13. Different roles may share structure and behavior
14. An object and its roles share identity
15. An object and its roles have different identities

16. Relationships between roles can be constrained
17. There may be constraints between relationships
18. Roles can be grouped and constrained together
19. Roles depend on compartments
20. Compartments have properties and behaviors
21. A role can be part of several compartments
22. Compartments may play roles like objects
23. Compartments may play roles which are part of themselves
24. Compartments can contain other compartments
25. Different compartments may share structure and behavior
26. Compartments have their own identity
27. The number of roles occurring in a compartment can be constrained

Figure 5.2: The classifying features by Steimann (features 1 to 15) [Steimann, 2000] extended by Kühn (features 16 to 27) [Kühn, 2017]

CROM: A Role-based Modeling Language Family

The *Compartment Role Object Model* (CROM) [Kühn et al., 2014] [Kühn, 2017, pp.140] is a metamodel family for role modeling. Via feature modeling is it possible to create metamodels for RMLs depending on the user's needs. This need is connected to the fact that there is no common definition of roles, what they exactly are and can [Kühn, 2017]. On one hand, a user adaptive way to create RML metamodels is a comfortable way to create specialized RMLs. On the other hand, this way the changeability of an application based on role-based languages can be raised. Using previous RMLs, e.g. Lodwig [Steimann, 2000], Generic Role Model [Dahchour et al., 2002], E-Cargo Model [Zhu and Zhou, 2006] or the Helena Approach [Hennicker and Klarl, 2014],[1] often meant that not all of the needed features of roles were available. In this case, mixing compatible RMLs to saturate the need for features would be a solution. But as Kühn *et al.* describe in [Kühn et al., 2014], it is not feasible to do that, since most approaches of RMLs do not define and publish their metamodels. It might also be the case, that not all available characteristics of roles for the user's application are needed. In this case a used RML is more complicated than needed. Additionally, not wanted features might have to be disabled in a way not needed normally. Consequently, a family of configurable metamodels for RMLs is useful to make the usage of those easier and more customizable.

Now, as the concept of CROM is cleared up, I want to address the composition the CROM metamodel. As already said, CROM is a feature-oriented family that composes metamodels according to a user's configuration. This composition is executed by using delta modeling. The architecture of CROM and especially the needed artifacts for the whole process can be seen in Figure 5.3. The `Metamodel Generator` uses `Delta Modules` which encapsulate the changes to the metamodel

[1]A list of more RMLs can be found in [Kühn et al., 2014].

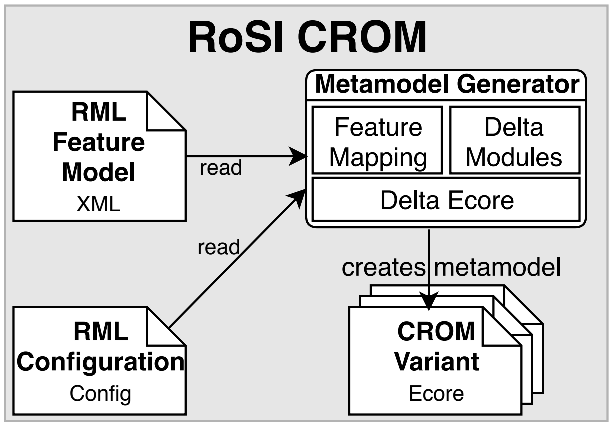


Figure 5.3: Architecture of the *Compartment Role Object Model*, extracted from [Kühn, 2017].

depending on a limited amount of feature choices per module. These feature choices can be found in the `RML Configuration` and the `RML Feature Model`. A `Feature Mapping` defines the connections between the `Delta Modules` and the features. The underlying framework for managing the delta modeling is *DeltaEcore*[2] [Seidl et al., 2014]. When checking if certain features are chosen or not, the framework finds `Delta Modules` to execute by using the already mentioned mapping. Executing such a module results in changes to the metamodel, effectively creating a new `CROM Variant`.

The used feature model is created, configured and validated using *Feature IDE*[3] [Thüm et al., 2014]. More information about *FeatureIDE* can be found later in this section (5.1.3). The mentioned feature model is derived from the list of classifying features of roles. The role model can be seen in Figure 5.4. In the following, I will reason about the concrete features of the model. The status of abstract features is not a user's choice but is derived by rules depending on concrete features. As the focus should be on the user-adaptive creation of metamodels, the abstract features will be only mentioned if needed in the explanation of a concrete one. As some of the features in the conceptual feature list for RMLs are mandatory,[4] not important to a technical implementation [5] or do represent not existent limitations,[6] such features do need to be present in the effective feature model of CROM.

However, the features 1, 2, 3 and 5 of the feature model are associated with the *behavioral nature* of roles. While the features 1 and 2 depict that roles can own attributes and operation, the 3rd feature introduces role inheritance. Inheritance relations of role types are added to the feature model as a logical extension of class inheritance. Feature 5 controls if roles can fulfill other roles. A somewhat similar task is given to the features 9 to 11. These introduce `Role Constraints` which limit roles have to be played, or cannot be played, together. With the features 12 and 13, there are two more constraints of the *behavioral nature*. `Group Constraints` as well as `Occurrence Constraints` of roles were already addressed in general considerations on RMLs. The *relational nature* of roles is represented by the features 8 and 14 to 17. In fact, the 8th and 14th feature are linked to together and correspond to the existence of relationships in the created RML metamodel. The other mentioned features of this nature are constraints. While the status of feature 15 decides on the usage of relationship cardinalities, the existence of constraints on one relationship (feature 16) and between two relationships (feature 17) can be configured too. `Intra Relationship Constraints`, on one hand, characterize a relationship, as they can be reflexive or acyclic for example. On the other hand, there are `Inter Relationship Constraints`, which

[2]URL: `http://deltaecore.org`, last visited: 14.05.18.
[3]URL: `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide`, last visited: 14.05.18.
[4]Features 5, 10 and 11 in Figure 5.2.
[5]E.g. feature 14 and 15 in Figure 5.2.
[6]The features 3, 4, 7, 9, 13 and 25 in Figure 5.2 for example.

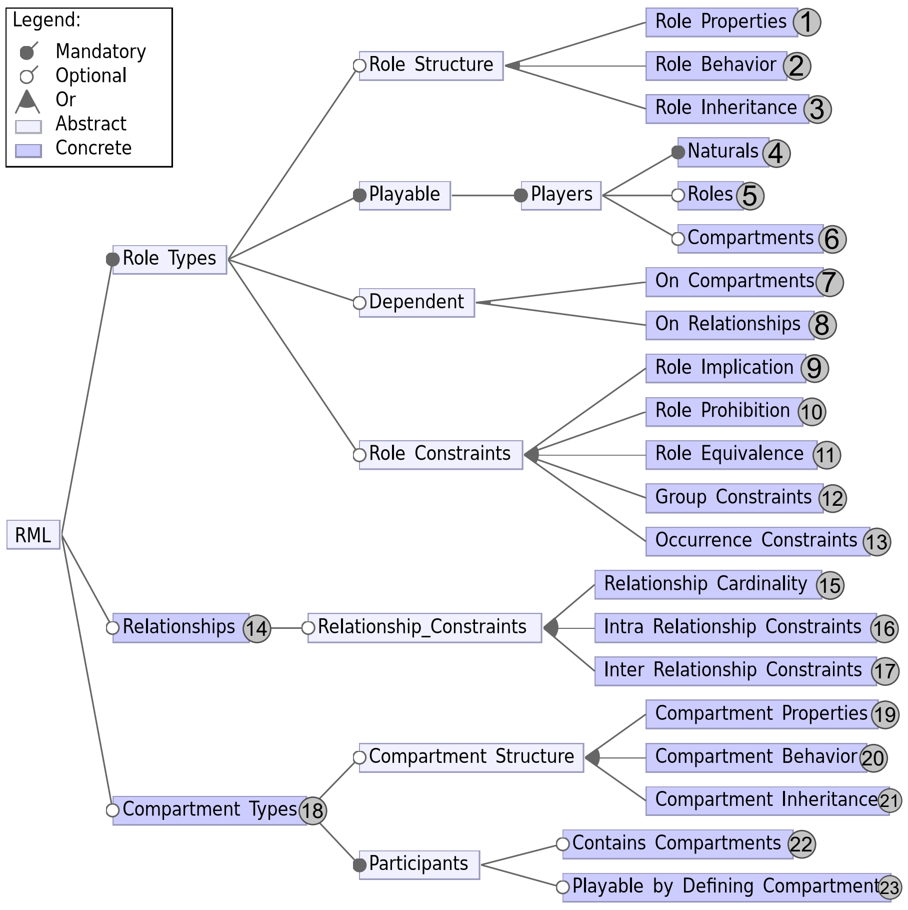


Figure 5.4: The effective feature model of CROM, extracted from [Kühn, 2017]

determine if two relationships can or have to exist at the same time when one of them is relevant to a role.

Most of the remaining features are associated with the *context-dependent nature* of roles. Firstly, there are the two linked features 7 and 18 deciding on the existence of `Compartment Types`. If these features are not chosen, roles are still context-dependent. In this case, the context is represented by a `Role Model` element. There can only be ons of those. Furthermore, such an element does not have properties, behavior, inheritance structures and cannot play roles. If `Compartment Types` exist, features decide if they can play roles in general (feature 6) or even roles defined in their own context (feature 23). Other features of this nature control the abilities of `Compartment Types`. If they have attributes and operations (features 19 and 20) or can contain inner `Compartment Types` themselves for example. Finally, the mandatory feature 4 is left to address. `Natural Types` represent classes that always can play roles. As CROM is based on class diagrams these cannot be disabled.

Concluding, CROM is an implementation of a RML family, enabling the comfortable configuration of an user-adaptive RML metamodel. As such it is also a suitable base of FRaMED (5.1.2), an editor using its feature model, derived from a sophisticated analysis of RMLs.

5.1.2 *Full-fledged Role Modeling Editor*

FRaMED SPL

In the following, I will address the FRaMED SPL [Kühn, 2017, pp.147], in particular, the current monolithic version of FRaMED provided under the heading *FRaMED 2*. The FRaMED SPL is a feature-oriented dynamic GEPL. More specifically, it is a family of user-adaptive Graphical Editors for role-based modeling, based on CROM (5.1.1). While most technological choices were subject to changes during the tools development history, the CROM is a constant of its foundation. I will follow the structure of the concerns[7] of the FRaMED GEPL and present how these are realized in its general approach. Addressing this, it is also interesting to look for dependencies in the editor SPL, created by specific choices of technologies, like frameworks, models and engines.

Beginning with the *edit concerns*, the graphical representation of model objects and the *palette properties* are managed by a framework for GEs. The choice of this framework is a critical decision, as it is a big factor in how much effort is needed to create and maintain the application. Furthermore, a change of the framework choice results in the need for reimplementing the FRaMED SPL. FRaMED is a family of editors in the *Eclipse* enviroment[8]. Consequently, while the Eclipse platform offers multiple options, the number of suitable frameworks is limited. Some of those are mentioned in 5.1.3. The third *edit concern* of *edit policies* is implemented by using a rule-based DSL. This way the solution might be harder to engineer initially, but also is more specialized toward its purpose. This way, it is possible to enable a comfortable way to define edit policies, compensating the initial effort. Additionally, it mostly avoids dependencies to frameworks that might be changing in an undesirable way.

The CROM comes into play when the *model concerns* are addressed. Firstly, there is the *feature model* of FRaMED. It is identical to the effective feature model for RMLs defined by the CROM. Furthermore, just like the feature model of CROM, it is also created and managed by using the framework *Feature IDE*. The feature model is the foundation for the dynamic configuration of the GEPL. To enable the user to change the configuration during runtime, an additional sub-editor of the GE is needed. Besides that, the consequences of a configuration change have to be calculated and executed. It is important to note that these consequences are crosscutting the concerns, as a change in the configuration can affect specific artifacts of all *edit concerns* for example.

However, the CROM also represents the *target model* of the *model transformation*. The source of the transformation, the *abstract syntax model*, is another concern. A concrete syntax model is part of the already mentioned GE framework and saves the structural and graphical information about a diagram. Meanwhile, the abstract version of it lacks the latter type of information. Therefore, it reduces the concrete syntax model by its visual details, like sizes or positions of model elements. Finally, it also acts as an intermediate representation between the concrete syntax model and the CROM. This intermediate representation is not dependent on the configuration. Rather, it has a feature configuration attached to it. Consequently, the *model transformation* converts an instance of the abstract syntax model and an associated configuration to a feature dependent model instance of CROM. The transformation is executed by an engine tailored for that purpose. Similar to the decision of the GE framework, the choice of the transformation engine can be critical and there are multiple options in the *Eclipse* environment. Some of the options are mentioned in 5.1.3, including an explanation of the one chosen.

FRaMED 2

Subsequently, I will briefly talk about the current version of FRaMED, as it is the base of the case study presented here. The FRaMED SPL is in development since 2015. Consequently, there were multiple big enhancement and changes to the family of GEs. Starting with the first version [Kühn,

[7]The concerns of GEPLs are listed in Section 3.1.
[8]URL: `https://www.eclipse.org/`, last visited: 14.05.18.

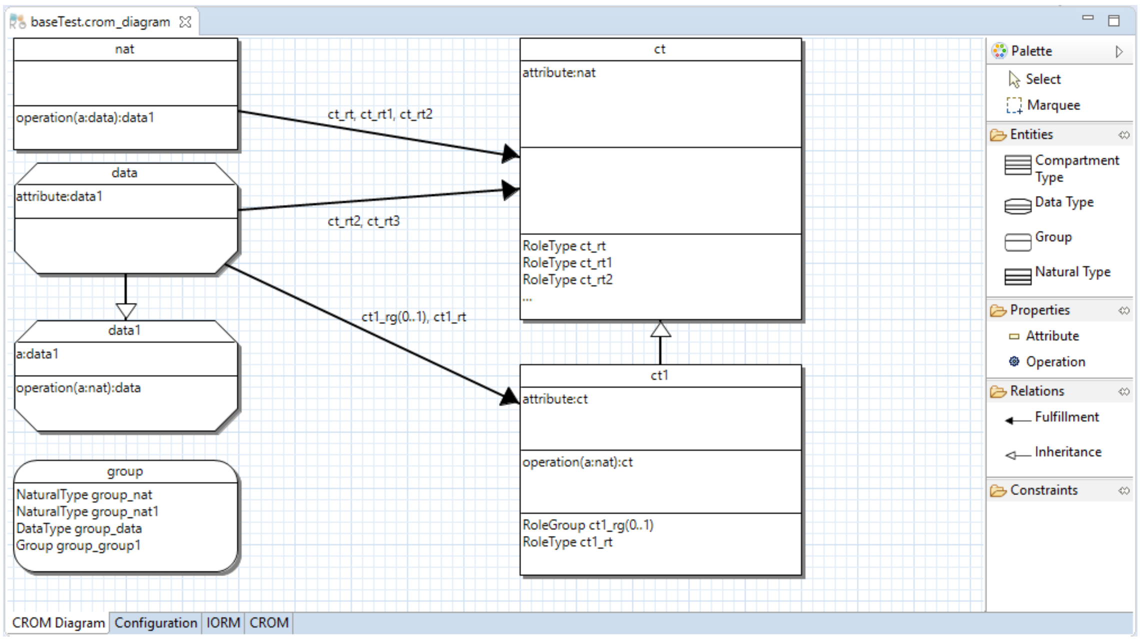


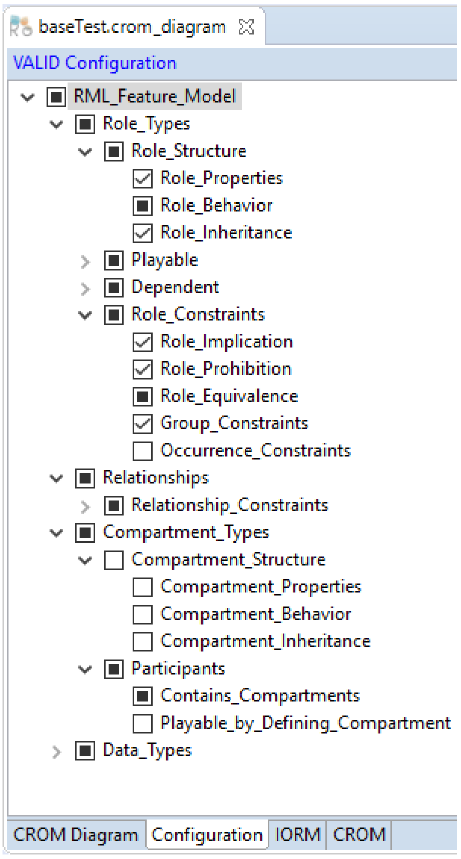
Figure 5.5: The second version of the *Full-fledged Role Modeling Editor*.

2017][9], these developments resulted in FRaMED 2[10]. The following will discuss the monolithic build of the second version of FRaMED. Frameworks and engines mentioned will be explained more precisely in 5.1.3.

The biggest change in the development of FRaMED 2 in comparison to its predecessor is the used GE framework. The second installment uses *Graphiti*[11] as a foundation to develop the GE concerns. The framework is used to manage the graphical representation of model elements. This is defined in the concrete syntax model of a diagram. Consequently, the corresponding metamodel is part of *Graphiti*. Additionally, the palettes entries' visibility and appearance are controlled by it. Corresponding to the change of the GE framework, a new intermediate representation was introduced in FRaMED 2. The *Intermediate Object Role Model* (IORM) operates as the abstract syntax model, reducing *Graphiti*s syntax model by its visual data, saving only structural information. Sorting out the visual information is a way to make the model transformation easier to implement and execute. The model transformation is executed using the *Epsilon Transformation Language* (ETL)[12]. The transformation engine is used to convert an instance of feature independent IORM to a CROM instance reliant to the configuration. As FRaMED 2 also uses the feature model of CROM, *Feature IDE* [Thüm et al., 2014] is used to work with its feature model.

Now that the used frameworks and engines are mentioned, it is time to document its development state before the execution of the case study. FRaMED 2 implements the envisioned dynamic GEPL partly. The dynamic configuration with its associated sub-editor, which can be seen in *Figure* 5.6, is widely reused from its predecessor. The same applies to the dialogues to create a whole new diagram or to edit specific elements. Furthermore, the palette and the model transformation are fully implemented feature dependent. While most of the parts concerning the graphical representation are implemented reliant on the configuration as well, there are features ignored in the discussed version of FRaMED. Examples for that are occurrence constraints of roles and relationship cardinalities. These should be invisible and not editable when the corresponding features are disabled. The implementation of feature-dependent *Edit Concerns* are mostly complete but lacks in this situation. Meanwhile, the complete feature of role groups is not available in this development state. It is a

[9]URL: `https://github.com/leondart/FRaMED`, last visited: 15.05.18.
[10]URL: `https://github.com/Eden-06/FRaMED-2.0`, last visited: 29.07.18.
[11]URL: `https://www.eclipse.org/graphiti/`, last visited: 15.05.18
[12]URL: `https://www.eclipse.org/epsilon/doc/etl/`, last visited: 17.05.18.



baseTest.crom_diagram

VALID Configuration

```
v ■ RML_Feature_Model
    v ■ Role_Types
        v ■ Role_Structure
              ☑ Role_Properties
              ■ Role_Behavior
              ☑ Role_Inheritance
        > ■ Playable
        > ■ Dependent
        v ■ Role_Constraints
              ☑ Role_Implication
              ☑ Role_Prohibition
              ■ Role_Equivalence
              ☑ Group_Constraints
              ☐ Occurrence_Constraints
    v ■ Relationships
        > ■ Relationship_Constraints
    v ■ Compartment_Types
        v ☐ Compartment_Structure
              ☐ Compartment_Properties
              ☐ Compartment_Behavior
              ☐ Compartment_Inheritance
        v ■ Participants
              ■ Contains_Compartments
              ☐ Playable_by_Defining_Compartment
    > ■ Data_Types
```

Figure 5.6: FRaMED 2: The dynamic configuration sub-editor.

complex feature with influence in all concerns. That is why the feature is used to investigate the ability to extend the modularized application by completely new features. Finally, the edit policies are not implemented yet.

In summary FRaMED 2 is suited as the foundation of this case study. There is a clearly defined objective of the application. Furthermore, the technological and conceptual base of the editor is well sophisticated. It has a fairly big number of monolithic build features to modularize. Additionally, some envisioned features are not implemented completely yet or need to be implemented from scratch. Therefore a case study based on FRaMED 2 can be used to explore if a design method for dynamic GEPLs can be applied to modularize families of GEs. However, the ability to remove and add complete modules from the application can be investigated also.

5.1.3 Reusable Technology

In this subsection, I will discuss the technological base of FRaMED 2. As CROM is a complex case for such a foundation, reaching deep into the concept of RMLs, it is reviewed in its own Subsection 5.1.1.

Eclipse

FRaMED is written in the *Eclipse* environment. The ecosystem offers comprehensive support for the development of model-based GEs. On one hand, there is the *Eclipse Modeling Framework* (EMF) [Steinberg et al., 2009], usable for all kinds of model driven applications. It is highly supported by other frameworks in the environment, e.g. for validation, versioning and visualization. That way, the usage of EMF is flexible, as its function can be extended by companion plug-ins.

On the other hand, the *Eclipse* GEF[13] and its derived frameworks provide multiple solutions to develop GEs. Either pure GEF or *Sirius* [Viyović et al., 2014],[14] *Graphiti*[15] and *Graphical Modeling Framework* (GMF) *Tooling*[16] can be used. While all frameworks are based on GEF, the last two mentioned ones are also part of the *Graphic Modeling Project* (GMP)[17]. The range of the solutions is broad, as GMF*Tooling* uses an MDD approach and *Graphiti* mostly relies on manually written code managed by the framework. Meanwhile using Sirius, GEs are mostly designed by building a configuration model without using code generation at all.

Frameworks for more specific tasks of this case study are also available in the *Eclipse* ecosystem. To execute the model transformation ETL [Kolovos et al., 2010][18], *Optimus*[19] or *EMorF*[20] can be used. All three engines apply different approaches as *Optimus* defines transformation in *Java* code, while the other two use specific DSLs. ETL is a rule-based language to implement model-to-model transformations. Using *EMorF*, the transformations are defined graphically. Furthermore, this engine also uses an MDD approach. Lastly, a framework to create, validate and manage feature models during runtime is needed. Two options to do this are EMF *Feature Model*[21] and *FeatureIDE* [Thüm et al., 2014].[22] Both allow the graphical definition and validation of feature models.

Overall this overview shows that the development of a GEPL, like FRaMED, on the *Eclipse* platform offers a broad range of approaches by different frameworks for all relevant tasks. This means that a balance of flexibility and efficiency in the development of such SPLs exists. Therefore, I conclude that Eclipse is a suitable environment for the FRaMED SPL and can be seen as a strong enabling factor for it.

Graphiti

Graphiti is a framework to create GEs. As such, it is involved in multiple concerns, namely all edit concerns. Its architecture dictates how the graphical representation of objects and the palette properties are defined. Furthermore, it offers the hooks used by the edit policies to ensure which actions can or cannot be executed. A loose bond to the feature model concern exists as the diagram editor created and managed by *Graphiti* has to be compatible with the dynamic configuration. Therefore the GE framework is deeply connected to many aspects of FRaMED's implementation and its approach has a significant influence on it.

Graphiti's architecture is based on providers and features. Both can be seen in Figure 5.7. In the implementation of this case study, four providers were used. The `DiagramTypeProvider` defines the properties of its kind of diagrams. It saves the diagram type's name and acts as parent element for the other providers. That means that it has, inter alia, a `FeatureProvider` attached. The attached provider collects all features of the GE, which define the graphical representation of elements as well as the operation on these. There are two types of these features: *planned* and *custom* features. The first type of features have planned behavior, such as the examples visible in Figure 5.7: *Add*, *Create* and *Delete* Features. These examples roughly describe the life cycle of elements in the editor, leaving out operations like moving, layouting or resizing a graphical object. The functionality of custom features is not predetermined by the framework. There are two more providers managed by the Diagram Type Agent. The `ToolBehaviorProvider` collects code about the palette, context menus and actions on a double click. All the calculations of the

[13]URL: `https://www.eclipse.org/gef/`, last visited: 18.05.18.
[14]URL: `https://www.eclipse.org/sirius/`, last visited: 18.05.18.
[15]URL: `https://www.eclipse.org/graphiti/`, last visited: 18.05.18.
[16]URL: `http://www.eclipse.org/gmf-tooling/`, last visited: 18.05.18
[17]URL: `http://www.eclipse.org/modeling/gmp/`, last visited: 18.05.18.
[18]URL: `https://www.eclipse.org/epsilon/doc/etl/`, last visited: 18.05.18.
[19]URL: `https://github.com/awltech/eclipse-optimus`, last visited: 18.05.18.
[20]URL: `https://marketplace.eclipse.org/content/emorf`, last visited: 18.05.18.
[21]URL: `https://projects.eclipse.org/projects/modeling.emft.featuremodel`, last visited: 18.05.18.
[22]URL: `https://featureide.github.io/`, last visited: 18.05.18.

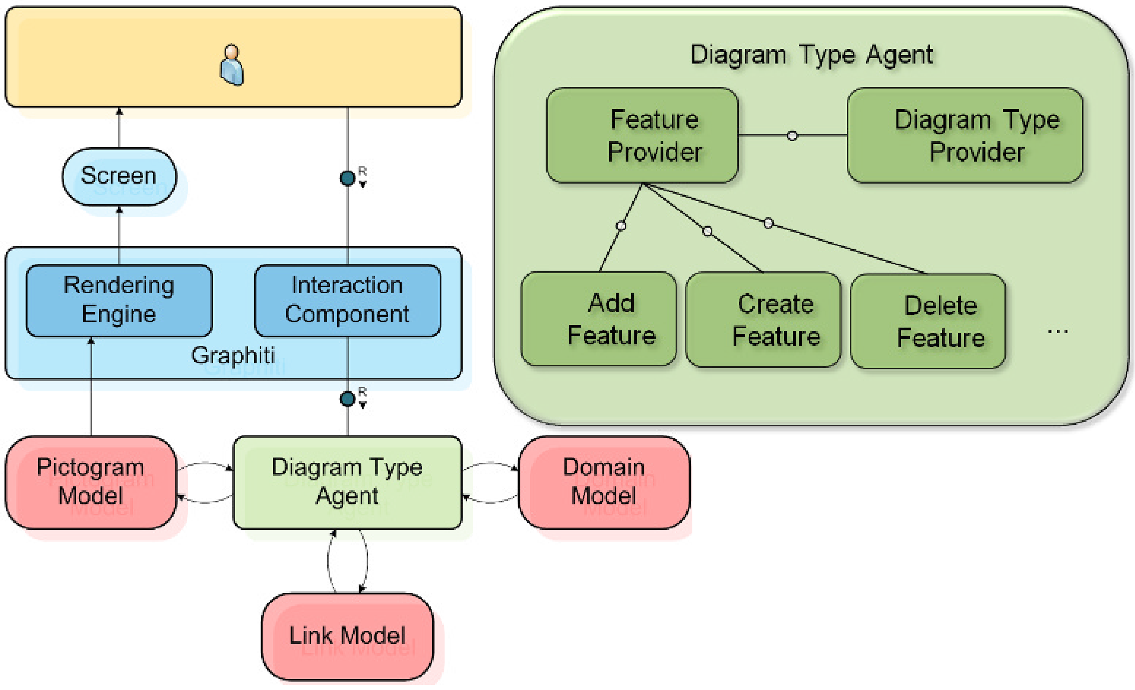


Figure 5.7: *Graphiti*: Overall architecture and cooperation of providers, extracted from *Graphiti* documentation[23].

GE's possible user interactions are context sensitive and executed during runtime. Meanwhile the `ImageProvider` connects icon resource paths to an identifier, which is referenced in the features. This way a feature does not need to know about exact file paths for its palette icons.

Another big part of *Graphiti*s architecture is the interaction between its models. It is presented in Figure 5.8 and described deeply in the *Eclipse Magazine*.[24] At first, there is the `Pictogram Model`, which defines the visual representation of a diagram to display. It is split into two levels, as one level addresses the hierarchy of elements and the second is using `Graphical Algorithms` to save position coordinates and sizes of elements. The `Graphical Algorithms` are contained in the `Shapes` of the hierarchy level. The metamodel of the `Pictogram Model` is offered by Graphiti and cannot be changed. However, the `Domain Model` is not prescribed by the framework. An instance of it has to be created by the developer of the GE and edited according to the visual model via co-evolution. This means that often an interaction with the editor leads to consistent changes in the `Pictogram` and `Domain Model`. This can be a neuralgic point as it is a big problem if the changes are not executed equivalently. Fortunately, not all edits involve the co-evolution. E.g. moving, resizing or layouting a graphical object are purely visual changes that do need to be propagated to the structural `Domain Model`. Finally, the `Link Model` exists to enable the co-evolution for edits after the a diagram element in both models are created. It links the representing elements on the hierarchical level of the `Pictogram Model` to their corresponding domain objects. The links are set during creation of an diagram element and disappear as these are deleted again.

In the subsection 5.1.2, I talked about the concept of a concrete and an abstract syntax model. The `Pictogram` and `Domain Model` act as an example of these. The first one mentioned represents the concrete syntax model. Meanwhile, the `Domain Model` strips its visual data and is the abstract syntax model. As mentioned in 5.1.2, while the concrete syntax metamodel is given by *Graphiti*, as `Domain Model` the IORM is introduced in FRaMED 2.

[23]URL: `https://www.eclipse.org/graphiti/documentation/`, last visited: 30.05.18.

[24]URL: `https://www.eclipse.org/graphiti/documentation/files/EclipseMagazineGraphiti.pdf`, last visited: 22.05.18.

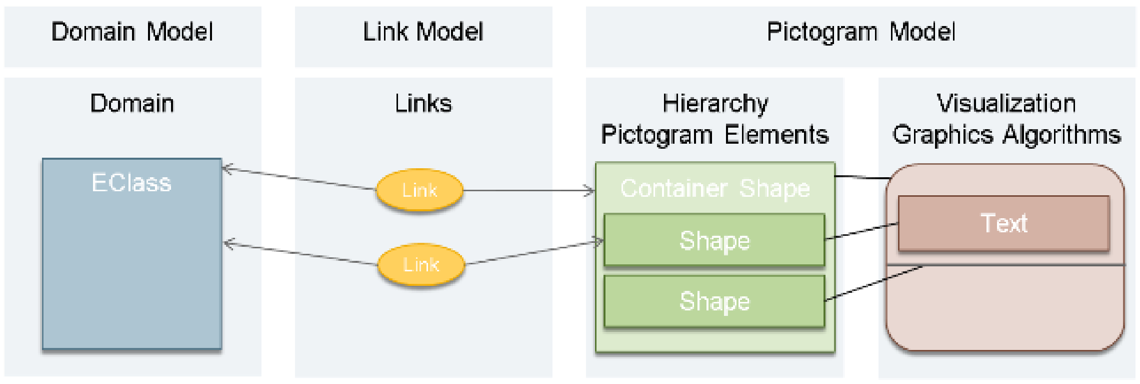


Figure 5.8: *Graphiti*: The interaction of models, extracted from *Graphiti* documentation[27].

Epsilon Transformation Language

The *Epsilon Transformation Language*[25] is a language and engine for model-to-model transformations. It is part of the *Epsilon* framework,[26] a family of languages for code generation as well as model validation, comparison, merging and more. ETL is a rule-based engine and implements an own DSL to define the transformation rules. In this case study, one base rule, which dispatches parts of the transformation to other more specific rules, is executed to start a model transformation. To enable this, the base rule imports other rules and calls them according to their guard checks. In order to explain this, I will present the structure of the ETL rules, visible in *Figure* 5.9.

A rule transforms a specific element of the source model to one of the target model, naming them `s` and `t`. These variables can be used in the rule to access these elements. The keyword `extends` signalizes a rule inheritance, which will be explained later. The extended super rule needs to be imported, which happens in the header of the transformation rule. In the rule, one can see the `guard`, which decides if the rule should be executed depending on a boolean expression, e.g. checking for properties of the source model element `s` to transform. Following are the actions of the rule. These most likely include manipulating the target model element `t`, for example setting its name the same as `s`. If the guard evaluates to `true` there are also the optional `pre` and `post` `operations` to be executed before and after the rule.

Returning to the question of how the transformation proceeds, the base rule operates on the top-level source model element and iterates over its children. For each child, the transformation can be triggered. The `guard` expressions of the specific rules decide which of them is responsible. As the specific rules can also start transformations for the children of their `s`, the elements of the model are discovered and converted to the target representation hierarchical. However, this is only the basic approach and there are more specific features to explore. Firstly, I want to address the already mentioned concept of rule inheritance. In the case a rule extends another rule, it will be only executed if its own as well as the guard of the super rule is evaluated to true. Furthermore, both work on the same source and target model element. The sub rule's types of `s` and `t` have to extend their corresponding types used for the super rule. The rule inheritance can be used to split up big monolithic rules, which proofed itself useful for the modularization in this case study. A more precise description of it can be found in the paragraphs of 5.2.2. The same applies to another feature of ETL: Extended properties. These properties can be attached to model elements, which allows the developer to propagate information between rules. In practice, without using extended properties, the feature configuration is only known to the base rule. By attaching the configuration to the children of the top-level source element in the base rule, the user's feature choice can be used in every other transformation rule too. This is a foundation for the dynamic configurable

[27]URL: `https://www.eclipse.org/graphiti/documentation/`, last visited: 30.05.18.
[25]URL: `https://www.eclipse.org/epsilon/doc/etl/`, last visited: 18.05.18.
[26]URL: `https://www.eclipse.org/epsilon/doc/book/`, last visited: 17.05.18.

```
import "SuperRule.etl";
pre ExampleRule { <actions> }
rule ExampleRule
 transform s : source!SourceElement
 to t : target!TargetElement
 extends SuperRule {
  guard : <boolean expression>
  <actions> }
post ExampleRule { <actions> }
```

Figure 5.9: ETL: Structure of a transformation rule.

model-to-model transformation. Further explanation of an interaction between rule inheritance and extended properties is elaborated in the *Realization* Section (5.2.2).

FeatureIDE

FeatureIDE is an extensible framework for feature-oriented SPLs [Thüm et al., 2014]. According to Thüm *et al.* the frameworks helps to avoid using multiple command line based tools when creating an SPL. It offers the ability to use one framework for multiple phases of the Feature-oriented Software Development (FOSD). These four phases are the following: domain analysis, requirement analysis, domain implementation and software generation. Furthermore, it is presented as a way to connect various SPL implementation techniques, e.g. feature-, delta- or aspect-oriented development. To achieve this, the similarities between the techniques are exploited by offering reusable artifacts. This is best possible for the domain and requirement analysis since these steps are nearly the same in all techniques. The extensible nature of *FeatureIDE* is visible when looking at the remaining two phases. For the domain implementation and software generation, the framework uses tools, which are either integrated or independently installed plugins [Thüm et al., 2014]. This enables to apply *FeatureIDE* for different SPL implementation techniques, depending on the tools.

However, in this case study, the framework is only used for two of the four phases. During the domain analysis, *FeatureIDE* is used to create the feature model. It offers a graphical editor to execute this step, allowing an intuitive way to do so. Additionally, the framework is used to validate and manage configurations with respect to the constraints of the feature model. On one hand, this includes the calculation if a configuration is valid or not. On the other hand, one configuration change initiated by the user can lead to multiple automatic selections and eliminations of other dependent features. The dependencies can be led back to the structure or the feature constraints of the model. The two last mentioned tasks of *FeatureIDE*, both associated with the requirement analysis, are sensitive to the scalability of the feature model usage. It is not feasible to develop validation checks as well as the automatic selection and elimination logic by hand for every new feature model. Instead, a framework for feature models should use the defined constraints of such a model to offer effective solutions for both tasks in a developer-friendly way. *FeatureIDE* meets this requirement.

5.2 Realization

In this section, I will discuss how the case study is realized by providing concrete implementation details depending on the *Eclipse* environment, the already presented frameworks and engines. The following will be structured according to the concerns, elaborated in *Chapter* 3. For every concern, three kinds of information will be given. Firstly, the tasks of the GE concern's artifacts will be addressed. Following is an explanation about the general implementation of the solution to the formerly presented tasks. Finally, I will show how the GEPL was modularized in respect to the discussed part of the editor.

Graphiti Feature	Methods	Pictogram model	Domain model	Impact
Create	`canCreate` `create`	✗	✓	Creates a model element in the domain model. This operation is usually coupled with the *add* feature.
Add	`canAdd` `add`	✓	✗	Creates a graphical object in the pictogram model.
DirectEditing	`canDirectEdit` `getInitialValue` `checkValueValid` `setValue`	✗	✓	Edits a value, e.g. the name, of a model element in a direct manner. The edit does not open a new dialog. The visual change of the value is executed by the *update* operation.
Layout	`canLayout` `layout`	✓	✗	Adjusts the layout of a graphical object. This often means inner elements, like lines or texts are moved to their correct positions.
Update	`canUpdate` `updateNeeded` `update`	✓	✗	Changes shown values, e.g. the name text, of a graphical object, to be up to date with the linked domain model element.
Move	`canMove` `move`	✓	✗	Changes the position of a graphical object on the canvas. A moved object can not change its parent container in the current implementation.
Resize	`canResize` `resize`	✓	✗	Changes the size of a graphical object. This operation is usually coupled with the *layout* operation.
Delete	`canDelete` `delete`	✓	✓	Removes a graphical object and its linked domain model element.

Table 5.1: *Graphiti*: Overview of the possible operations on *Graphiti* shapes.

5.2.1 Edit Concerns

Graphical Representation

The artifacts of this concern define the visual appearance of model elements on creation and edits afterward. This also means that they implement how operations, also called *Graphiti* features, change graphical objects. These operations can be moving or resizing such an object for example. Other operations, like deleting a model element, also involves the co-evolution between the concrete and abstract syntax model.[28] An overview of the used the *Graphiti* features can be seen in the *Tables* 5.1 and 5.2. An elaborated explanation of the table can be found in the next paragraph. However, another task is yet unmentioned. If an operation on a graphical object can be executed is also partly checked by artifacts associated with this concern. Partly means that only sanity checks are executed as feature dependent decisions are defined by rules of the edit policy concern. The sanity checks, e.g. verify that the edited objects are of the correct type or have all needed child element defined. Furthermore, it usually looks up if the link between the domain object and its graphical representation exists. These checks ensure that the execution of the operation makes sense.

The key understand to how *Graphiti* defines the visual appearance of elements, is to understand the feature architecture of the framework. The features yet mentioned are planned operations that can be executed on model elements. Unplanned features will be discussed in later paragraphs. The planned features, presented in *Table* 5.1, work on *Graphiti* shapes, for example, a *Natural, Role* or *Compartment Type*. Meanwhile, *Table* 5.2 gives an overview of such features for connections, e.g. relationships or fulfillment relations. Both tables are based on the same information as both list

[28]See the Graphiti section in 5.1.3 for reference.

Graphiti Feature	Methods	pictogram model	domain model	Impact
Create	`canStartConnection` `canCreate` `create`	✗	✓	Creates a model element in the domain model. This operation is usually coupled with the *add* feature.
Add	`canAdd` `add`	✓	✗	Creates a graphical object in the pictogram model.
Reconnect	`canReconnect` `postReconnect`	✓	✓	Reconnects an existing connection to a new source or target element.
Delete	`canDelete` `delete`	✓	✓	Removes a graphical object and its linked domain model element.

Table 5.2: *Graphiti*: Overview of the possible operations on *Graphiti* connections.

the methods to implement and the impact of the feature. The influence of features is described textually while also signaling which models, the concrete and abstract syntax model, is edited. For some operations, both models can be subject to change.

Firstly, it can be seen that the before mentioned sanity checks can be easily implemented for all operations using the methods whose names begin with the word *can*. How the other tasks are solved can be seen in the table too. At creation, the *Add* feature is used to define the graphical representation of *Graphiti* shapes and connection. After the creation time, a shape's appearance can be directly changed by the *Move* and *Resize* features. The *DirectEditing, Layout* and *Update* operations lead indirectly to the same situation. This can be explained as the *DirectEditing* feature only edits the domain model while being coupled to the *Update* operation, which causes the corresponding change in the pictogram model. Like the *Update* feature is only applied when another operation changed a model, the *Layout* operation is usually used when a *Graphiti* shape was moved or resized. For connections, the *Reconnect* feature serves a similar purpose as moving a shape around. Concluding a life cycle of a connection or shape, the *Delete* operation manipulates both mentioned models by removing a specific model element.

Finally, the co-evolution is represented in the discussed tables too. A yet unmentioned *Graphiti* feature is the one that creates a new model element in the domain model. The *Create* feature is coupled with the already mentioned one that adds corresponding graphical objects to the pictogram model. A similar connection can be found between the *DirectEditing* and *Update* operation. Linking two features loosely together by triggering each other leads to both models being manipulated in a concurring way. This can also be achieved if one feature changes both models at once, like the *Reconnect* and *Delete* operations. Besides the planned features there are also custom features. Their purpose is completely open to being implemented by the developer of the GE. Similar to planned features the interface of them also offers a method to check if the feature can be executed. In conclusion to the last two paragraphs, a developer using *Graphiti* to create a GE implements the needed planned and unplanned features. The framework manages user-initiated and intern triggers to these features. This way, the development tasks are easier to solve than implementating everything completely from scratch, while still offering enough flexibility.

Now that I addressed the atomic units of a GE solution based on *Graphiti*, I will discuss how such are structured together. Planned features are collected in `IPattern`, for *Graphiti* shapes, and `IConnectionPattern` else. These interfaces are the foundation for classes that represent all planned operations on one specific diagram element. Often a class implementing an `IPattern` addresses all features presented in *Table* 5.1. Of course, this classes can also be extended to fulfill needs of a specific implementation. This will be heavily used for the modularization in this case study. One structural level higher the `FeatureProvider` plays an important role. It collects all the *Graphiti* Pattern as well as the custom features in one place. Its list of referenced `IPattern`,

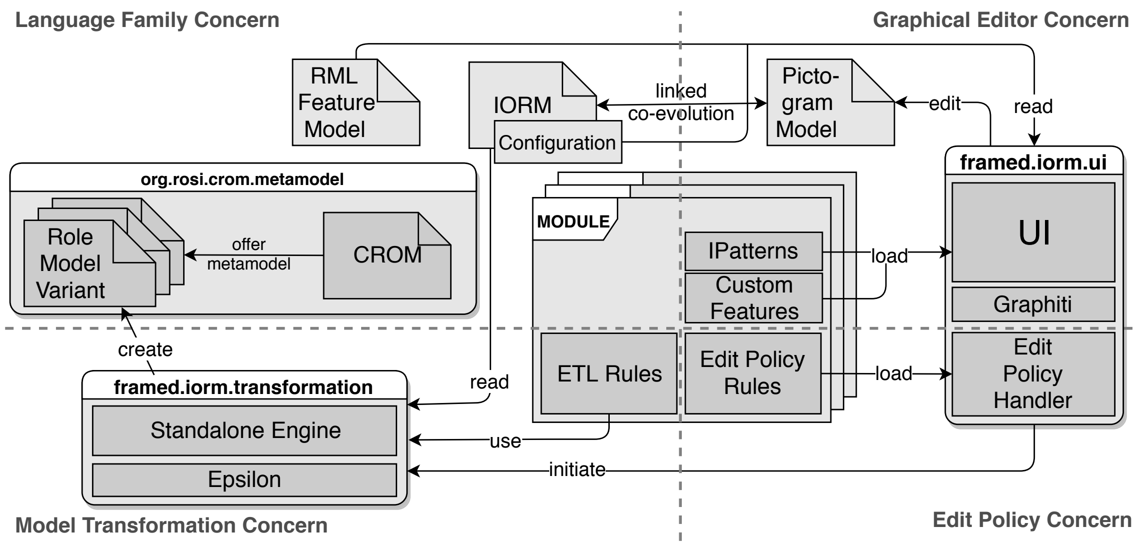


Figure 5.10: The modularized architecture of FRaMED 2.

IConnectionPattern and custom features classes, results in a, by *Graphiti* organized, set of representation and interaction information on model elements of a diagram type.

However, the question of how all these artifacts can be modularized is still open. This is a challenge since the FeatureProvider needs to reference the objects of them all. In a non-modularized application, the objects can be instantiated and referenced in fix code. This means that the FeatureProvider already knows during compilation time which *Graphiti* pattern and custom features are to add, as is it hard-coded. This procedure makes it impossible to easily extend and specialize the application in a modularized manner. The solution to this key problem is dynamic loading and instantiating the needed artifacts during runtime. This way the FeatureProivder does not need to explicitly know about new or removed artifacts relevant to it. Instead, all classes in specific places are searched. If a class extends a specific supertype, it will be instantiated and added to the FeatureProvider. This modularizable procedure can be seen in Figure 5.10. Right now, I will put the focus on the second quadrant named *Graphical Editor Concern* as all concerns will be discussed in the rest of chapter bit by bit.

One can see the project framed.iorm.ui which includes the UI and the Graphiti framework among other things. This is not specific to the modularized SPL. The same goes for the processes in which the project's artifacts read the feature models specific configuration and edit the pictogram model. Cross-cutting the graphical representation and language family concerns is the linking and co-evolution of the abstract and concrete syntax model. The implementation of the FeatureProvider is part of the UI. At this point, the commonalities between the monolithic and modularized versions of FRaMED end. In the monolithic version of the GEPL, the *Graphiti* pattern and custom features would also be located in the UI. Instead, these artifacts are encapsulated in concern cross-cutting modules. The modules are in the center of the architectural diagram and contain artifacts of all implementation aspects. The formerly mentioned patterns and custom features are such artifacts for the concern of the graphical representation.

The graphic further shows how the module specific classes are fetched dynamically by the UI, more precisely by the FeatureProvider, which is symbolized by the load relation. How this is done in code can be seen *Figure* 5.11. It presents a simplified version of the solution to dynamically find and instantiate *Graphiti* patterns in the FeatureProvider. This happens in the constructor of the class. By filtering all java classes in modules for specific concrete sub-classes, all *Graphiti* patterns can be found. The searched classes have given supertypes, which are also part of the UI project. These supertypes are FRaMED specific implementations of the *Graphiti* pattern interfaces IPattern and IConnectionPattern. For the sub-tasks yet mentioned, the auxiliary class UIUtil, which offers

```java
public FeatureProvider(...) {
 List<Class<?>> classes = UIUtil.findModuleJavaClasses();
 for(Class<?> c : classes) {
  if(!Modifier.isAbstract(c.getModifiers())) {
   if(UIUtil.getSuperClasses(c).contains(FRaMEDShapePattern) ||
      UIUtil.getSuperClasses(c).contains(FRaMEDConnectionPattern)) {
     Object object = c.newInstance();
     if(object instanceof FRaMEDShapePattern)
      addPattern((FRaMEDShapePattern) object);
     if(object instanceof FRaMEDConnectionPattern)
      addConnectionPattern((FRaMEDConnectionPattern) object);
}}}}
```

Figure 5.11: Simplified source code of instantiating and adding *Graphiti* patterns dynamically to the FeatureProvider.

solutions for various basic problems, is used multiple times in the constructor. Finally, the found sub-classes are instantiated and added to the feature provider, which differentiates between the shape and connection patterns. In a similar manner the FeatureProvider also dynamically loads custom features. This process is implemented in an own method getCustomFeatures, which checks against a superclass for all custom features.

To conclude these paragraphs, I want to address how modules and their interactions are realized in respect to the concern of the graphical representation. In this case study, modules are packages inside the project framed.iorm.ui. This makes the dependency management easy as dependency circles between the modules and the UI can be avoided without using plug-in extension points. Consequently, this option was chosen to keep the implementation effort small. However, this is just a question of the implementation, not a conceptual challenge. It is possible to offer these modules as *Eclipse* projects or plug-ins. In such a case, extension points would solve dependency problems. They could also be encapsulated in jar files. When put into the correct folder, the GEPL would find and account them. Finally, I want to talk about the interactions between modules. To understand why these are problematic, it is important to know the differences between the core application, core modules and generic feature modules. The core application is essential to the family of GEs and is structured monolithic. The FeatureProvider is part of the application core. Other than that, there are modules, distinguished by the roles they play. There are core modules, which encapsulate core functions in a modularized manner, and generic modules. The second kind of modules is expected to be changed regularly. It has to be easy to remove or add them to the SPL to achieve feature modularity. I would recommend going for the same goals for core modules, but as these are a way more stable parts of the application, feature modularity is less important for those. In general, the modules can always reference functions of the monolithic core application. To offer modularity, the core application cannot directly reference artifacts in modules. I already mentioned an example for this, when I presented how the FeatureProvider uses *Graphiti* patterns without referencing them explicitly. Furthermore, modules cannot easily reference other modules' attributes and operations. This can be seen in accordance with the feature modularity principle of information hiding and encapsulation by Käster [Kästner et al., 2011].[29]

To differ between the internal implementation and the external interface of modules, *References* are used in this case study. The concrete *References* are implemented in the modules offering selected internal attributes and operations to the public. That way, most of the modules' implementation details are hidden, while some needed information can be accessed outside of the module. To get a *Reference*, once again, dynamic class loading is used. Similar to the way the FeatureProvider executes this step, an abstract supertype is used to identify the wanted *References*. Of course, the abstract *Reference* class needs to be implemented in the core application to

[29]The two principles are discussed in the term definition of *feature modularity* (1.4).

```
/*package: org.framed.iorm.ui.references*/
public abstract class AbstractGroupingFeatureReference {
 protected Type modelType;
 protected String DIAGRAM_KIND,
                  SHAPE_ID_NAME,
                  SHAPE_ID_TYPEBODY;
 //getter methods for the attributes
}
/*package: org.framed.iorm.ui.modules.compartment*/
public class CompartmentGroupingFeatureReference
     extends AbstractGroupingFeatureReference {
 Literals literals = new Literals();
 public CompartmentGroupingFeatureReference() {
  modeltype = Type.COMPARTMENTTYPE;
  DIAGRAM_KIND = literals.DIAGRAM_KIND;
  SHAPE_ID_NAME = literals.SHAPE_ID_COMPARTMENTTYPE_NAME;
  SHAPE_ID_TYPEBODY = literals.SHAPE_ID_COMPARTMENTTYPE_TYPEBODY;
}}
/*package: org.framed.iorm.ui.coremodules.customfeatures*/
public class StepInFeature extends FRaMEDCustomFeature {
 public void execute(ICustomContext context) {
  AbstractGroupingFeatureReference grRef =
   UIUtil.getGroupingReferenceForType(...);
  if(grRef == null) return;
  Diagram diagramToStepIn = UIUtil.getDiagramForGroupingShape(...,
   grRef.getShapeIdTypebody(), grRef.getShapeIdName(), grRef.getDiagramKind());
}}
```

Figure 5.12: Simplified source code demonstrating the interaction between *Compartment Types* and the *Step In* custom feature.

enable all modules and core artifacts access the concrete *References*. This can clearly be seen as a limitation of the modularization, which is addressed in the corresponding subsection 5.3.2. However, the abstract class defines which parts of the internal implementation is exposed. Meanwhile, the *References* in the modules offer concrete values and operations to fulfill the definition.

As a concluding example, I want to present the interaction between the modules of *Compartment Types* and the *Step In* custom feature, which is encapsulated in a core module. As a *Compartment Type* groups other model elements together, it is considered a grouping feature. To access the inner elements of a *Compartment Type* the editor user steps into the *Compartment*, zooming one layer deeper into the role model. Simplified source code for the example can be seen in Figure 5.12. One can see how the `AbstractGroupingReference` declares which data should be exposed by the *Reference*. In this case it publishes three `Strings`, one of such defining which kind of diagram a feature, that groups other elements, uses to do this. The other two attributes enable identifying specific shapes that are part of the graphical representation of a grouping feature's model element. The purpose of the attribute `modeltype` will be cleared up later. The abstract *Reference* is located in a core application package. The concrete `CompartmentGroupingFeatureReference` is part of the *Compartment Type* feature module and defines the values of the variables declared by the abstract *Reference*. It uses a module specific auxiliary class *Literals* for that purpose. Lastly, the *Step In* custom feature uses the concrete *References* when it is executed. By using the formerly mentioned auxiliary class `UIUtil`, it gets the correct concrete *References*. It searches all available `GroupingFeatureReferences` and checks which one of those is implemented for the same type as the element to step it. In this example, the type would be a `CompartmentType`. For that purpose, the `modeltype` attribute is part of the *References*.

Palette Properties

This concern addresses which planned and unplanned features[30] are accessible to the user in a specific editor state. The state is dependent on the configuration, the current view of the editor and what kind of element is right-clicked. The view differs whether you are stepped in a *Compartment Type* or not. For planned features, it is decided if they are shown in the palette, only dependent on the feature configuration and the editor view. But not only visibility is implemented for planned features by artifacts of this concern. The appearance in the palette is taken into account too. It is defined by a feature name, icon and palette category. Custom features do not appear in the palette, but a context menu when right-clicking a model element. It is this concerns' artifacts' task to ensure that the correct custom features are available in the context menu, depending on the clicked element and the feature configuration.

The logic to determine if and how a feature is shown in the palette or context menu is written in *Java* code and managed by *Graphiti*. The framework offers two providers for this purpose. On one hand, there is the *ImageProvider* linking file paths of images to identifiers. By that, the implementations of the interface `IPattern` and `IConnectionPattern` do not need to handle potentially impractical and long file paths in their code to specify the icon of a palette entry. The identifiers can then be referenced by the planned features' *Graphiti* patterns. This is realized by letting the operation `getCreateImageId` returning the identifier of the wanted icon. In the following, *Graphiti* takes this into account when building the palette. On the other hand, the `ToolBehaviorProvider` decides when a feature is visible in the palette and context menu. Two operations enable that. Firstly, there is `getPalette` which returns an array of `IPaletteCompartmentEntry` objects. These are categories to be shown in the palette. Nested categories are possible but not used in this case study. These categories contain the actual palette entries. This means that the logic in the `getPalette` method also needs to be able to build the correct structure of the palette. While deciding which features are shown in the palette, it decides in which category. In the monolithic version of the GEPL, the whole logic for this tasks is hard-coded in the `getPalette` function. The operation `getContextMenu` is the second operation of the `ToolBehaviorProvider` to address. It decides which custom feature is accessible via the context menu, analyzing which model element was clicked at. Again, in the not modularized family of GEs the logic to calculate this, uses direct references to every unplanned feature.

That said, I will follow up by presenting the modularized approach to the responsibilities of this concern. Starting with the building process of the palette, I want to address the `ImageProvider`. It is not possible to use fixed `String` variables for the icon identifiers and file paths in the provider class in a modularized FRaMED. Instead, these attributes are defined in the feature's *Graphiti* patterns, the icons belongs to. Of course, the `ImageProvider` needs to get them without directly referencing the *Graphiti* pattern classes. This is avoided by using dynamic loading and instantiating of the pattern classes, similar to the same process in the `FeatureProvider`. The icon file path and identifier are fetched from these objects in the following. Besides the `ImageProvider`, the `ToolBehaviorProvider` needs to be implemented modularly. Again, it is not possible to hard-code the logic for all existing features in the provider class, since that is a monolithic implementation. All needed information about features' palette visibility and appearance need to be defined in the implementations of the `IPattern` and `IConnectionPattern`. Before I explain how this can be managed in a compact manner, I want to briefly present three specific artifacts. They are enumerations which represent possible values used in the building process of the palette.

`PaletteView` This enumeration describes a part of the editor state. It dependents on the fact if the editor is stepped in a *Compartment Type* or not. Consequently, there are two possible values. These two differ as the palette is fundamentally different between both cases. E.g. the feature of *Role Types* and the relation as well as the constraint features belonging to them are only shown if the user stepped into a *Compartment Type*.

[30]These are explained in the previous subsection.

```
/*package: org.framed.iorm.ui.palette*/
public class FeaturePaletteDescriptor {
 public PaletteCategory paletteCategory = PaletteCategory.NONE;
 public ViewVisibility viewVisibility = ViewVisibility.NO_VIEW;
 public FeaturePaletteDescriptor(PaletteCategory pC, ViewVisibility vV) {
  paletteCategory = pC;
  viewVisibility = vV;
 }
 public boolean featureExpression(List<String> features, PaletteView pV) {
  return true;
}}
/*package: org.framed.iorm.ui.modules.compartment*/
public class CompartmentTypePattern extends FRaMEDShapePattern ... {
 private FeaturePaletteDescriptor spec_FPD = new FeaturePaletteDescriptor(
  PaletteCategory.ENTITIES_CATEGORY,
  ViewVisibility.ALL_VIEWS) {
   public boolean featureExpression(List<String> framedFeatureNames,
       PaletteView pV) {
    switch(pV) {
     case NON_COMPARTMENT_VIEW: return framedFeatureNames.contains(
      "Compartment_Types");
     case COMPARTMENT_VIEW: return framedFeatureNames.contains(
      "Contains_Compartments");
     default: return false;
}}};
```

Figure 5.13: Simplified source code of the `FeaturePaletteDescriptor` and its use in *Graphiti* pattern.

`ViewVisibility` This artifact is used to define in which view a palette entry is visible. There are four values to consider. Firstly, there are the two values of `PaletteView`. Additional to that, there are also features that are not shown in one palette view, but both or none. If the configuration allows it, *Compartment Types* can be nested. Therefore, the feature to create a *Compartment Type* can be visible in both views. But there are also features that should never be manually accessible to the user. Such is the `Model` feature. It is used to create the top-level model element in the domain model when creating a new diagram. After this point, it should not be possible to execute it again for an existing diagram.

`PaletteCategory` In this enumeration one can find values for the four palette categories *entities, properties, relations* and *constraints*. The values are used to define in which category a feature belongs to. Additionally, there is a fifth value symbolizing that it is never shown in the palette.

There are four palette properties: The icon, the visibility depending on the view, the visibility depending on the feature configuration and the palette category in which a planned feature is shown in. I already cleared up how the `ImageProvider` manages the first property. The last three properties are managed by the `ToolBehaviorProvider` and together they form the `FeaturePaletteDescriptor` (`FPD`). The `FPD` is an attribute of the *Graphiti* patterns. The standard implementation as well as a concrete instantiation of it can be seen in *Figure* 5.13. An `FPD` uses two of the three previously presented enumerations: `PaletteCategory` and `ViewVisibility`. These values are used to manage the second and fourth palette properties mentioned at the start of this paragraph. What is still missing is the part of the `FPD` that decides whether a feature is visible depending on the current configuration. To do that, the method `featureExpression` exists. It takes a list of all chosen features and the current palette view as parameters. A logical expression using the two parameters returns a `boolean` value in the following, deciding the visibility of a feature in the palette. The standard implementation defines the palette properties of a feature

```
1  /*package: org.framed.iorm.ui.providers*/
2  public class ToolBehaviorProvider ... {
3   void addShapeFeature(FRaMEDShapePattern pattern, List<String> features) {
4    FeaturePaletteDescriptor fpd = pattern.getFeaturePaletteDescriptor();
5    if((fpd.viewVisibility == ViewVisibility.ALL_VIEWS) ||
6       (paletteView == PaletteView.NON_COMPARTMENT_VIEW &&
7        fpd.viewVisibility == ViewVisibility.NON_COMPARTMENT_VIEW) ||
8       (paletteView == PaletteView.COMPARTMENT_VIEW &&
9        fpd.viewVisibility == ViewVisibility.COMPARTMENT_VIEW)) {
10     if(fpd.featureExpression(features, paletteView)) {
11      switch(fpd.paletteCategory) {
12       case ENTITIES_CATEGORY: entityCategory.addToolEntry(...); break;
13       case PROPERTIES_CATEGORY: propertiesCategory.addToolEntry(...); break;
14       case RELATIONS_CATEGORY: relationsCategory.addToolEntry(...); break;
15       case CONSTRAINTS_CATEGORY: constraintsCategory.addToolEntry(...); break;
16       default: break;
17 }}}}
```

Figure 5.14: Simplified source code on how the `ToolBehaviorProvider` analyses palette properties defined in the `FPD`.

that is never shown in the palette. That is the consequence of the fact, its value of the palette category and view signal that it is not visible in any category and view. If a feature should be visible under specific circumstances, its *Graphiti* pattern class needs to instantiate its `FPD` explicitly describing such circumstances. One can see that in the bottom part of the Listing 5.13 as the `CompartmentTypePattern` modifies its `FPD` to fit the feature's palette properties. Firstly, it sets the values of the enumerations in such way that the feature of *Compartment Types* is accessible in all views in the palette category of the entities. Of course only as long as the operation `featureExpression` returns `true`. The implementation of this method checks for two different features, depending on the palette view in which the editor works. Outside of a *Compartment Type*, it checks if the features with the same name is chosen by the user. If the editor's view is stepped in a *Compartment Type* diagram, it decides the features visibility depending on a different configuration part. It checks if the user wants to allow that *Compartment Types* can be created in other ones. Of course, this indirectly contains a test if *Compartment Types* are allowed in general.

Until now, I addressed how the `FPD` defines palette properties but did leave out how the `ToolBehaviorProvider` analyses them. To do this, I want to present a simplified version of the palette building process in *Figure* 5.14. The listing shows how the `ToolBehaviorProvider` decides if and where a feature associated with a *Graphiti* shape is added to the palette. The presented operation `addShapeFeature` is called by the previous mentioned `getPalette` method. After fetching the `FPD` objects of the feature pattern, the visibility dependent on the palette view, which is checked in the lines 5 to 9. If a feature's `FPD` object specifies visibility in either both or only the current view, represented by the `paletteView` variable, the feature expression of it evaluated in the following. By doing this in line 10, the `ToolBehaviorProvider` does decide if the concerned feature is meant to be present in the palette by using the feature configuration as a parameter. Finally, if this check was successful too, the `ToolBehvaiorProvider` assigns the feature's palette entry to a category by analyzing the corresponding `FPD` attribute `paletteCategory`.

Lastly, I want to talk about custom features and how their accessibility to the user is controlled. The *Figure* 5.15 presents source code for that purpose. The example shows the feature to edit the name and cardinalities of *Relationships*. Like every custom feature class, `EditRelationshipFeature` implements an operation called `contextMenuExpression`. It defines if a custom feature is present in the context menu when right-clicking a specific model element. The visual and domain representation are parameters to this method. The visual element is checked to be a *Graphiti* connection or a decorator of such. The connection decorators of *Relationships* are its name and cardinali-

```
 1 /*package: org.framed.iorm.ui.relationship*/
 2 public class EditRelationshipFeature ... {
 3  public boolean contextMenuExpression(PictogramElement pe, EObject bo) {
 4   if(pe instanceof FreeFormConnection ||
 5      pe instanceof ConnectionDecorator) {
 6    if(bo instanceof Relation) {
 7     Relation relation = (Relation) businessObject;
 8      if(relation.getType() == Type.RELATIONSHIP)
 9       return true;
10   }}
11  return false;
12 }}
13 /*package: org.framed.iorm.ui.providers*/
14 public class ToolBehaviorProvider ... {
15  public IContextMenuEntry[] getContextMenu(ICustomContext customContext) {
16   List<IContextMenuEntry> possibleMenuEntries = ...,
17                           contextMenuEntries = new ArrayList<IContextMenuEntry>();
18   PictogramElement pe = customContext.getPictogramElements()[0];
19   EObject bo = UIUtil.getBusinessObjectIfExactlyOne(pictogramElement);
20   for(IContextMenuEntry contextMenuEntry : possibleMenuEntries) {
21     if(contextMenuEntry.getFeature().contextMenuExpression(pe, bo))
22      contextMenuEntries.add(contextMenuEntry);
23   }
24   return contextMenuEntries....
25 }}
```

Figure 5.15: Simplified source code presenting the build process of the context menu.

ties. This part, in the lines 4 and 5, ensures that the custom feature is only accessible if such elements are clicked at. Following in line 6 to 8, a check on the domain model element is performed. Overall it looks up if the business object is of the IORM specific type `RELATIONSHIP`. Only if both checks are successful, the operation returns `true`. The return value is used by the `ToolBehaviorProvider` to build the context menu containing the correct custom features. It does this by iterating over all possible menu entries of custom features. For each of those, the provider calls the `contextMenuExpression` operation. If it is evaluated to `true`, the corresponding feature is added to a list of context menu entries.

Edit Policies

The conception and implementation of the artifacts in this concern was provided by Christian Deussen. In the following, I will describe his work in my own words. The concern of edit policies controls which operations are allowed to execute under specific circumstances. Other than the pure sanity checks which are directly implemented in the *Graphiti* patterns, the edit policies are dependent on the configuration of the editor. A complete list of influences for this decision comprises the operation to execute, the model element it should be applied to, the current editor view[31] and the before mentioned feature configuration.

To define rules which take all these influences into account, a DSL is used. The metamodel of it can be seen in *Figure* 5.16. An edit policy model has multiple policies, while one policy is responsible for exactly one kind of operation, called `Action`, on one specific type of model element. A policy itself is split into two rules: A left side, the *feature rule*, and the *constraint rule*, which marks the right side of a policy. As one can see in the metamodel diagram both of those rules are represented by an element called `AbstractRules`. It also allows using unary operators, namely `not`, and binary conjunctions. The last mentioned category contains *and* as well as *or*. *Feature rules*, marked as `IsFeature` in the metamodel, check if specific features are selected in a given

[31]The editor view is equivalent to the palette view.

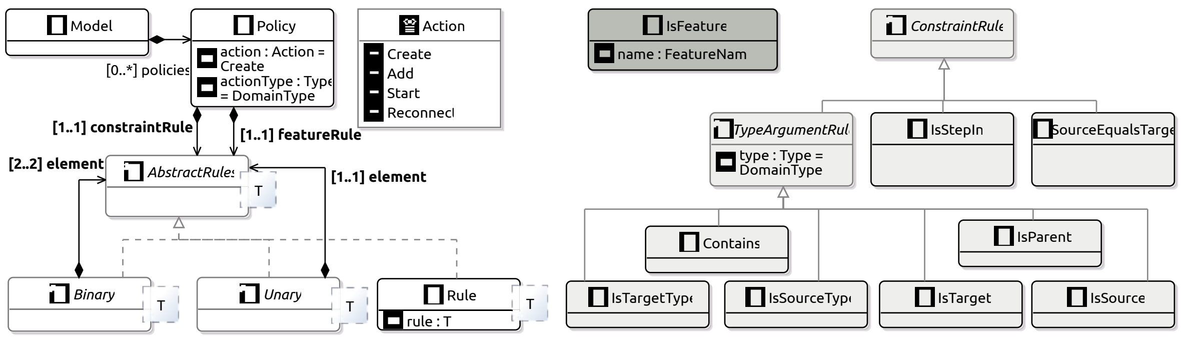


Figure 5.16: The meta model used to define edit policy models.

```
Action  Element  {
  Feature_1  and  not  Feature_2  when  isSource(Type_1)  or  isParent(Type_2) }
```

Figure 5.17: Example of the general structure of an edit policy rule.

configuration. Besides that, checks for types, source, target elements and more are executed by the `ConstraintRules`. *Figure* 5.17 illustrates how a generalized edit policy can look like. The edit policy begins by declaring for which kind of operation on what model elements it is responsible. The inner of the policy is split by the keyword `when`. On the left side, there are two features conjunct. This part of the policy is the *feature rule*. However, the *constraint rule*, to the right of the splitting keyword, checks for the types of the source element, given it is a rule for a connection, and the elements parent container. Now I will follow up presenting four artifacts to explain how these rules are used in *Java* code.

`ConstraintRuleVisitor` **and** `FeatureRuleVisitor` These classes implement the evaluation respectively of the left and right side of policies. This includes the actual checks, e.g. for features and types while dealing with unary and binary operators.

`EditPolicyHandler` This artifact offers handles to evaluate all policies in the edit policy models of one diagram. The evaluation itself is propagated to `RuleVisitor` classes. The `EditPolicyHandler` has references to his associated diagram and all found models. The handles are called by the `Graphiti` patterns in the operations starting with *can*. This way the patterns implement own sanity checks and use the edit policies as well.

`EditPolicyService` This class offers foundational services in the context of edit policies, as it contains the logic to find all edit policy model and manages the `EditPolicyHandlers`. Dynamical searches make the edit policy models accessible to the `EditPolicyHandler` and are especially important for the modularized version of FRaMED 2. Finally, the `EditPolicyService` also checks for the satisfiability and consistency between edit policy models.

It is not yet cleared up completely is how the artifacts of this concern are structured and used modularly. The third quadrant of *Figure* 5.10 illustrates how it is done. One can see that the `EditPolicyHandler` is part of the `UI` project. In fact, all four introduced artifacts are. This makes sense as the handler class and parts of the `EditPolicyService` are tailored to be used in *Graphiti*. E.g. the handles to evaluate policy models, as well as the service managing one `EditPolicyHandler` per diagram, are implemented according to the processes in *Graphiti*. Meanwhile big parts of the metamodel for the edit policies are designed more generic. Besides the `Action` enumeration, it could be easily reused for other GEPLs for role models. Therefore, it is located in an own project. However, the architecture also shows how every module has its own edit policy model. This creates two challenges: finding these models in the modules and negotiating between

```
1  /*package: org.framed.iorm.ui.coremodules.naturaltype*/
2  Create Fulfillment {
3     true when
4     isStepOut() and isSource(NaturalType) and isTarget(CompartmentType) }
5  /*package: org.framed.iorm.ui.modules.datatype*/
6  ... Dates when
7     isStepOut() and isSource(DataType) and isTarget(CompartmentType) }
8  /*package: org.framed.iorm.ui.modules.roletype*/
9  ... Roles and Contains_Compartment when
10    isStepIn() and isSource(RoleType) and isTarget(CompartmentType) }
11 /*package: org.framed.iorm.ui.modules.compartmenttype*/
12 ... Compartments when
13    isSource(Compartment) and not SourceEqualsTarget() and
14    isTarget(CompartmentType) }
15 ... Compartments and Playable_by_Defining_Compartment when
16    isSource(Compartment) and SourceEqualsTarget() }
```

Figure 5.18: Example of an edit policy rule for creating *Fulfillments*.

them. Firstly the `EditPolicyService` searches all edit policy models which are encapsulated in their own files. These files can be found by checking their extension *.editpolicy*. Negotiating edit policies is needed if two or more of them are defined for the same operation on the same kind of model element. This is important in a modularized version of the application as it is essential to split up the logic behind the decisions defined in edit policies. Furthermore, edit policies might be implemented, without being able to know which other ones are present. By default, they are all just connected together by an *or* conjunction. This ensures that an added policy cannot invalidate another already implemented one. If a developer wishes that *and* conjunctions are used to combine the policies, the keyword `overwrite` can be used when defining the edit policy.

Figure 5.18 shows edit policies concerning the creation of *Fulfillments*. The policies are split up and located in multiple modules. In which exactly is dependent on the source element of the *Fulfillment*. The first rule in the lines 2 to 4 is responsible to decide if the fulfillment can be created when the source element is of the type *Natural Type*. This is implemented by specifying `isSource(NaturalType)` in the rule. A similar check is used for the target element. In this version of FRaMED 2, it can only be a *Compartment Type*, which is common for all five policies. Finally, the first rule also checks what the editor view is. Since *Natural Types* do not exist in *Compartments*, the view expected in the addressed rule is out of such (`isStepOut()`). One can extract that the right side rule of a policy describes the situation in which the left sides feature rule is used. In the case of this rule, there are no features specified as *Natural Types* can always be sources of *Fulfillments*. That is different for *Data Types*. The feature model of CROM contains the feature *Dates*, which describes if such elements can play roles. Therefore, while the right side of the policy in the lines 6 and 7 is similar to the mentioned before but the feature rule now checks if the *Dates* feature is selected by the user. The same goes for the *Role Types*. Here the corresponding feature is called *Roles*. It is worth to notice that, since *Role Types* are always bound in the context of a *Compartment Type*, the edit policy for it uses `isStepIn()`. The last two presented policies are located in the *Compartment Type* module. While both concern *Fulfillments* with a *Compartment* as source element, they differ in the fact if the target and source elements are the same. This is checked by the expression `SourceEqualsTarget()`. The first policy in the lines 12 to 14 uses the negated evaluation of it. In this case, the left-hand rule only needs to look up if the feature *Compartments* is activated. This feature is equivalent to *Dates* and *Roles*. In the lines 15 and 16 a policy is shown that addresses *Fulfillments* from and to the same element. Consequently, it also checks if the current configuration has the feature *Playable_by_Defining_Compartments* chosen. Following its name, it decides if a *Compartment* can play a role in its own context.

Concluding, I want to summarize why an own DSL is used to define edit policies, in contrast to *Java* code for example. Firstly, unrelated policies need to be located in separated artifacts, which is important for the modularization of FRaMED 2. That is already the case applying the DSL approach. Blatantly, this could also be achieved by developing specialized *Java* classes responsible for implementing edit policies. In such a situation the separation of concern would not be this obvious. However, the usage of simple *Java* code would create other challenges. Checking edit policies for satisfiability and especially consistency on a set of policies is not always trivial. Using the edit policy models this task is easier to perform. Finally, the structure of these models ensures that one rule can not invalidate another one, concerning the same operation on the same model element. Using the models to define edit policies, similar ones are connected by an *or* conjunction, as long as the user does not explicitly want to use an *and* operation. If only *or* conjunctions are used, the previously mentioned property can be ensured. This is important as a rule in one module does not know about the implementation details of other rules, again following the principle of information hiding and encapsulation [Kästner et al., 2011].

5.2.2 Language Family Concerns

Feature Model

The feature model is the foundation of the SPL aspect as it offers a tree structure of features while also defining relationships between them. Furthermore, there are configurations associated with it, which need to be managed, including checks for validity and logic that calculates automatic selections and eliminations of features. These automated feature choices can occur if features are related to each other, either by the model's structure or constraints. Such constraints can be that one feature implicates or excludes another one. Twos feature can also be equivalent, meaning both can only be chosen together. This concern also addresses the definition of the standard configuration, which is used when a new diagram is created. Another important task is the implementation of a configuration editor page. Such an editor tab should create a user-friendly representation of the feature model. Especially, the structure and the status of features should be transparent to the user. The status does not only include if a feature is chosen or not. There are also locked features, which cannot be eliminated by the user as the structure or constraints of the feature model do not allow it. This needs to be visible to the user, just as automated selections and eliminations should be updated instantly after a user interaction with the configuration editor tab. Finally, changes to the configurations have to be signaled to other parts of the GEPL.

The definition of FRaMED's feature model is elaborated in 5.1.1. It is realized using *FeatureIDE*, creating an *XML* file capturing the feature model's structure and dependencies between features. In the same project, in which this artifact is located, one can find also a file specifying the standard configuration, which is read and inserted into the abstract syntax model of a new diagram, created with the family of GEs. The editor to change the configuration[32] is built by using *Java* code reading *FeatureIDE*'s model definition. It follows up creating a collapsible tree view of the features. *FeatureIDE* is also used to perform a validation check for the current configuration, showing the result on the top of the editor page. The same can be said about the calculations of automatically selected and eliminated as well as locked features. As *FeatureIDE* offers implementations for these decisions, development effort and error-prone implementations can be avoided. The framework executes these operations by analyzing the features' dependencies on each other. It further differentiates between features that were set manually or automatic.

Lastly, I want to explain how the whole GEPL learns of a configuration change. Every time a user changes the status of a feature in the configuration editor, a command is executed. The object of `ChangeConfigurationCommand` itself calls the *Graphiti* custom feature `ChangeConfigurationFeature`. The custom feature actually changes the configuration attached to the abstract syntax model. This happens by reading the current configuration, using *FeatureIDE*, and setting it as the new

[32]The editor tab is visible in *Figure* 5.4.

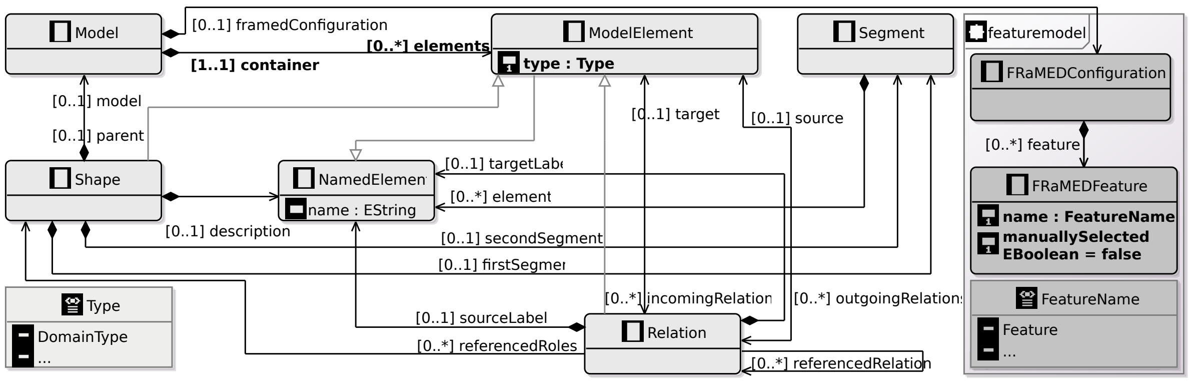


Figure 5.19: The metamodel of *Intermediate Object Role Model*.

configuration of the IORM instance. Consecutively, the palette is refreshed and *Graphiti* patterns of specific model elements are notified about that change. An example of such is the `RelationshipCardinalityPattern`. It does set the cardinality elements of relationships visible or invisible, depending on the status of the corresponding feature. By registering in the `ChangeConfigurationFeature`, the *Graphiti* pattern gets notified about every configuration change. However, the `ChangeConfigurationCommand` is executed as an intermediate step to ensure its possible to undo and redo the configuration changes.

It is important to note, that I talk about a possible FRaMED 2 specific implementation of a modular feature model in the modularization part of the *Future Work* Section 6.3.3. The presented solution there is not realized in the case study, as implementing existing solutions is not that interesting for the contributions of this thesis. Instead, the explanation should show that it is feasible to use a modular feature model. By doing that, I work towards the goal of presenting how a fully modularized *GEPL* can be implemented.

Abstract Syntax Model

The abstract syntax model acts as an intermediate representation between the concrete syntax model, offered by *Graphiti*, and CROM. It has to be suited as the source model of the model transformation and reduce the *Graphitis* pictogram model by its visual information, only saving the structural representation of a role model. Furthermore, it needs to be accessible and fitting to be co-evolved alongside the pictogram model.

The *Intermediate Object Role Model* is implemented by an *Ecore* metamodel. A simplified diagram of it can be seen in *Figure* 5.19. One can see that a model can have multiple model elements. These have a name and can be either a `Shape`, like a *Natural Type*, or a `Relation`. The last-mentioned kind of model elements can be *Inheritances* or *Relationships* for example. The Enumeration `Type` lists such options. `Shapes` reference two segments, which are used to save operations and attributes. Additionally, they have a description which is used to represent occurrence constraints of roles. Meanwhile, *Relations* might reference their target and source places with corresponding labels to save cardinalities. Finally, they can also reference *Role Types*, *Role Groups* and other `Relations`. The ability to reference the first two types is important for *Fulfillment* relations. Returning to the `Model`, the reference to the feature configuration catches the eye. The configuration is part of its own package, which also contains the implementation of a feature itself and an enumeration of possible feature names. A `FRaMEDConfiguration` owns any number of features. These save the selected features choices.

Overall the IORM is a mix between the pictogram model of *Graphiti* and CROM. `Shapes` in the IORM reference two segments for properties and operations, just like the visual representation of *Graphiti* does for *Role* or *Natural Types*. It also differentiates between `Shapes` and `Relations`, equivalent to *Graphitis* separation of shape and connection pattern. Meanwhile, the IORM also shares similarities with the CROM. Hierarchies of diagrams in the concrete syntax model are often broken up to ensure the correct elements are shown at any time. This means that they are not the same as the hierarchies of the business objects associated with the diagrams. However, the IORM shares the hierarchical structure of the CROM. These properties make it easier to execute the co-evolution between the concrete and abstract syntax model, while also allowing for a more simple implementation of the model-to-model transformation.

Just like a modularization of the feature model, implementing a modular way to extend and specialize the abstract syntax model does not create added value to the contributions of this thesis. Furthermore, the solution, using *DeltaEcore* [Seidl et al., 2014], would be similar to the already implemented *DeltaEcore* application to generate metamodels of CROM. However, for the sake of completeness, a brief description of how a modularization of IORM would look like can be found in 6.3.3.

Target Metamodel

In this case study, the target metamodel needs to define a comprehensive and reusable representation of role models. As a model-to-model transformation creates the concrete roles model, the metamodel of them needs to be suitable to this transformation.

The metamodel of transformation targets is the *Compartment Role Object Model*. For a more elaborated discussion of it, see Subsection 5.1.1 or [Kühn et al., 2014]. As CROM is family of metamodels, there is not only one metamodel, but one for every valid configuration. Therefore, I don't want to show a metamodel diagram of only one variant, but rather describe the general structure of all metamodels of CROM. For almost every feature in the feature model associated with CROM, there are corresponding model elements. As the feature model was created to comprehend all concepts of RMLs, a model properly build according to the feature model also achieves those goals. In the CROM, an element `Model` can have any number of children. These are of the type `ModelElement` and can be separated into four categories. Firstly, there are type elements, e.g. operations and attributes. Additionally, there are rigid types, like *Natural* or *Compartment Types*, and their opposite. *Role Types* are anti-rigid. The concept of rigidity in this context is discussed in 5.1.1. Finally, the model elements can also be relations. *Inheritances* and *Role Constraints* are examples of these. A hierarchy of elements can be modeled as such can also have a `Model`. Grouping model elements, like *Compartment Types* and *Role Groups*, can manage own children elements that way.

The modularization of CROM in the FRaMED SPL would be fairly simple. For that purpose, the already implemented metamodel generation can be applied. However, it is not yet fully integrated into FRaMED 2, because there was no need for the metamodel generation until now. Furthermore, the integration is not an integral part of the thesis. I address the modular implementation of the target metamodel in the future work paragraphs (Subsection 6.3.3).

Model Transformation

The artifacts of the last concern define how the transformation from the abstract syntax to the target metamodel is implemented. In this case study, an instance of the IORM is transformed into a variant of CROM. The execution of the transformation is feature dependent on the by the IORM referenced configuration.

The Transformation for a specific diagram is started every time it is saved. In the following, the source model instance is read by the transformation engine. By using a set of rules, a role model in the CROM representation is created. The rules are implemented in the *Epsilon Transformation*

```
 1 public class TransformationExecutor extends EpsilonStandalone {
 2  copiedAndGeneratedFiles = new ArrayList<File>();
 3  public TransformationExecutor() {
 4   List<String> importNames = new ArrayList<String>();
 5   Enumeration<URL> moduleFileEnumeration = UIBundle.findEntries(..., "*.etl");
 6   for(URL url : moduleFileURLs) {
 7    if(!packageMarkedAsNotUsed(url.toString(), "modules/") &&
 8       !packageETLFilesMarkedAsNotUsed(url.toString(), "modules/")) {
 9     Path path = //copy *.etl file and return path
10     copiedAndGeneratedFiles.add(new File(path.toString()));
11     importNames.add(etlFile.getName());
12   }}
13   generateORM2CROMWithImports(importNames, epsilonFolder);
14  }
15  boolean packageMarkedAsNotUsed(String url, String sourceFolder)
16  { /*return true, if package name starts with underscore*/ }
17  boolean packageETLFilesMarkedAsNotUsed(String url, String sourceFolder)
18  { /*return true, if etl file name starts with underscore*/ }
19 }
```

Figure 5.20: Simplified source code of the process to find ETL files in modules.

Language. The Language is already presented in the *Background* Section 5.1.3. Therefore, I only want to give a brief overview on the rule structure in this paragraph. ETL defines which kind of source model element can be transformed to which type of target model element. A guard expression is used to calculate if a rule can be executed under specific circumstances. Furthermore, it's possible that a rule extends another one. Finally, there are two operations, which are executed before and after a rule respectively. Almost all model elements are associated with specific rules for the transformation, which can be seen as an important foundation for the modularization. Exceptions are elements like operations and attributes which are concerned in the elements they belong to. Beside these model specific rules, there is a base rule starting the transformation. It works on the top-level source model element and iterates over all children of this element. The transformation engine chooses the correct imported rules to execute on the children elements according to the source elements and guard expressions of the rules.

Finally, I want to address how the transformation is modularized. In quadrant 3 of the architecture diagram 5.10, One can see the how the transformation is located in its own project `framed.iorm.transformation`. It contains the transformation engine itself as well as the *Epsilon* language family, including ETL. The incoming `initiate` and `read` relations describe how artifacts of the `UI` project trigger the transformation, which fetches the instance of the IORM instance to transform. The outgoing `create` relation indicate that the transformation produces and role model variant as *Compartment Role Object Model* instance. The architectural elements discussed by now are not exclusive to the modular implementation. A modular specific detail is the location of the ETL rules. All but one rule are located in their respective model element's modules. The transformation engine, more specific the class `TransformationExecutor`, fetches the modules' rules and uses them for transformation itself as well as to build the ETL base rule.

Listing 5.20 points out how to access ETL rules implemented in modules. ETL can't import files across projects. As the modules and the `TransformationExecutor` are located in different projects, the modularized usage is not as easy as the dynamic loading and instantiating of *Java* classes in other concerns. Instead, the files defining the rules need to be copied to the `transformation` project. The code to do that can be seen in the lines 9 to 11. However, the two lines before check if a package or an ETL file is marked to not be used, using two operations, which are implemented in the same class. Changing the name of a single ETL file or a whole package to begin with an underscore will result in it being ignored by the `TransformationExecutor`. This is meant for presentation and debugging purposes. The code line 11 and 13 are used to enable

```
1  public class TransformationExecutor extends EpsilonStandalone {
2   copiedAndGeneratedFiles = new ArrayList<File>();
3   private String generateORM2CROMWithImports(List<String> importNames, ...) {
4    List<URL> ORM2CROMUrls =
5     Collections.list(TransformationBundle.findEntries(...);
6    fileText = /*fetching content of base rule without base imports*/;
7    for(String s : importNames)
8     fileText = fileText.replace(importMarker, importMarker + "import" + s);
9    generatedORM2CROM.createNewFile();
10   copiedAndGeneratedFiles.add(generatedORM2CROM);
11   /*write content of fileText into new generated ORM2CROM*/
12   return generatedFile;
13 }}
```

Figure 5.21: Simplified source code of the process to generate the base rule for the model transformation.

the build process of the base rule. While the general code of the base rule is fixed, the sub-rules have to be imported. Of course, the imports cannot be predetermined and have to be created at runtime. Therefore, the names of the files to import are collected in line 11 and the list of them is used for the call in line 13. The implementation of the called operation is presented in *Listing* 5.21. By iterating over the names of the copied rules, imports for them are created in line 7 and 8. These import statements are added to the general code of the base rule and a new file *generatedBaseRule.etl* is created. It is important to note that only the imports have to be written, because the guard expressions and specified source element of the imported rules are used to determine which one is to use when. This way references in the base rule's action itself can be avoided.

There are two important features of the ETL yet unexplained. Firstly, I want address rule inheritance, which is a foundation for the modularization of the model transformation. It allows distributing similar ETL rule across different packages. In *Figure* 5.22 one can see what exactly is meant by that. The first rule `Inheritance` defines how *Inheritances* are transformed in general. However, it is not useful to execute only this rule as it does not specify which kind of *Inheritance* is executed. Therefore the rule is marked as abstract. An abstract rule is only executed if a concrete sub-rule of it is executed also. In the example, such a sub-rule is `CompartmentInheritance`. In the sub-rule, the target model type is `CompartmentInheritance` instead of the general CROM type of *Inheritances*. According to that, there are checks for the type of the source, target element and the corresponding feature `Compartment_Inheritance`. Similar rules are also implemented for *Role Types* and *Natural Types*. The rule inheritance allows implementing the general transformation of *Inheritances* located in an own module, while the sub-types of *Inheritances* can be implemented in the modules associated with them. Consequently, this avoids duplicate code and allows for modularization. One can easily imagine implementing the transformation of a new type of *Inheritance* for new kind of model element added to the GE, only creating a new sub-rule of `Inheritance`.

Besides rule inheritance, the ETL feature of extended properties is heavily used in the modular implementation of FRaMED 2. The properties of model elements can be extended by using the notation $\sim$. As the source model elements are used across multiple rules, information can be propagated from one to another using extended properties. This is most important for two purposes in this case study. Both are present in the ETL code listed in Figure 5.23. Firstly, in the base rule `ORM2CROM` the current feature configuration is read and saved in a map, relating its name to its status. This collection is attached to every source model element to transform. This enables access to the GE's configuration in every ETL rule, not only the base rule. Line 5 shows how the model elements are extended by the configuration, while the 19th and 25th lines demonstrate how the same property is used. The guard expressions check for specific features of the configuration in these lines of code.

```
/*package: framed.iorm.ui.coremodules.inheritance*/
@abstract
rule Inheritance
 transform s : source!Relation
 to t : target!Inheritance {
  guard : s.getType==(source!Type#Inheritance)
  //general implementation for inheritances
}}
/*package: framed.iorm.ui.modules.compartment.inheritance*/
rule CompartmentInheritance
 transform s : source!Relation
 to t : target!CompartmentInheritance
 extends Inheritance {
  guard : s.getSource.getType==(source!Type#CompartmentType) and
          s.getTarget.getType==(source!Type#CompartmentType) and
          s.~features.get("Compartment_Inheritance")
}
/*package: framed.iorm.ui.modules.roletype.inheritance*/
rule RoleTypeInheritance ...
/*package: framed.iorm.ui.coremodules.naturaltype*/
rule NaturalTypeInheritance ...
```

Figure 5.22: Code of ETL rules presenting the application of rule inheritance.

Secondly, extended properties also allow creating rule inheritance structure, in which two non-exclusive rules can inherit from the same super-rule. Exclusive rules cannot be executed for the same model element during the same transformation. If two rules are not exclusive to each other, this property cannot be ensured, which can lead to problems in cooperation with rule inheritance. In ETL the rule execution is implemented in a way that for every concrete sub-rule its super-rule is executed once creating a model element of the target model. Now, if two sub-rules of the same super-rule are executed together, the super-rule is applied twice, creating two different target model elements. As both sub-rules are meant to work on the same target element, this is unwanted behavior and has to be avoided. This is managed by attaching the target element to its source element. When transforming a specific element for the first time during one model transformation, the resulting target model element is saved as an extended property of its source model element. In the following, all actions concerning the transformed element are applied to the referenced element in the extended property. As these properties are rule-overlapping, this can be used to ensure that multiple concrete sub-rules work on the same referenced target model element.

Code presenting the implementation of this can be seen in Listing 5.23. Both the rules `RoleTypeWithOccurrenceConstraints` and `RoleTypeWithAttsAndOps` inherit from the general ETL rule for *Role Types*. The sub-rules have to be executed together for every role if the features for attributes, operations and occurrence constraints are activated for such. Therefore, there is no exclusion between those two rules. In line 11, the general rule `RoleType` checks if the role to transform was already transformed before. Only if not, it applies the general actions to the role and sets the reference `~transformed` to the newly transformed element. Afterward, the sub-rules are executed, working on the model element referenced by `~transformed`. Consequently, they do not use the automatically generated target model element `t`, defined in the rule headers. However, since the element behind `t` is already created, but sometimes not wanted, it has to be deleted eventually. All three rules addressing *Role Types* check if their target model element `t` is the same as the one referenced in `~transformed`. If they are not identical, the element `t` is deleted, which can be seen in the code lines 15, 21 and 27.

```
rule ORM2CROM
 transform s : source!Model to t : target!Model {
  guard : s.parent == null
  //create feature map reading current configuration
  for(e in s.elements) e.~features = featureMap;
}
@abstract
rule RoleType
 transform s : source!Shape to t : target!RoleType {
  guard : ...
  if(s.~transformed == null) {
   s.~transformed = t;
   //general implementation for role types
  }
  if(not(s.~transformed == t)) { delete(t); }
}
rule RoleTypeWithOccurrenceConstraints
 transform s : source!Shape to t : target!RoleType extends RoleType {
  guard : s.~features.get("Occurrence_Constraints")
  //transformation of occurrence constraint using s.~transformed, not t
  if(not(s.~transformed == t)) delete(t);
}
rule RoleTypeWithAttsAndOps
 transform s : source!Shape to t : target!RoleType extends RoleType {
  guard : s.~features.get("Role_Behavior")
  //transformation of attributes/ operations using s.~transformed, not t
  if(not(s.~transformed == t)) delete(t);
}
```

Figure 5.23: Code of ETL rules presenting the application of extended properties.

5.3 Discussion

In this section, I want to evaluate the case study towards the goal of creating a modular family of GEs. To do that, I will discuss to what extent and how the end product of the case study fulfills the requirements defined in Chapter 3. Following is a consideration of the limitations of the modularization executed in the case study. Finally, I will draw the conclusions of the case study.

5.3.1 Requirements

Functional Requirements

Overall, I can state that the family of applications, developed in the case study, fulfills all functional requirements previously defined. To defend this statement, I will present how this is achieved. Firstly, there are the *Tables* 5.3 and 5.4 summarizing all functional requirement in short. In this tables, one can also see which artifacts implement functions described by specific requirements. For a more detailed list of requirements, see the corresponding Chapter 3. Additionally, I will address how the listed artifacts realize such implementations.

Edit Concerns The *Table* 5.3 addresses the functional requirements associated with the *Edit Concerns*. The first one of such is the *Graphical Representation* concern. Most of its requirements are fulfilled by *Graphiti* features. These can be part of *Graphiti* patterns, single custom features or specialized classes offered by *Graphiti*. Such specialized classes are `FRaMEDReconnectFeature` and `FRaMEDDeleteConnectionFeature`. These are needed as *Graphiti* patterns for visual objects of relation model elements do not allow to specify delete and reconnect features for them. Instead, the logic for these is normally centralized in the previously mentioned classes. That is a problem as it hinders the modularity of the GEPL. It cannot be avoided to use them at all, but its possible

Concern	IDs	Requirement subject	Artifacts
Graphical Representation	FR01 to FR03	Visual appearance of model elements during their whole life cycle	*Graphiti* Pattern, *Graphiti* Custom Features, `FRaMEDReconnectFeature`, `FRaMEDDeleteConnectionFeature`
	FR04 to FR06	Represention of model elements in the *Abstract Syntax Model* during their whole life cycle	*Graphiti* Pattern, *Graphiti* Custom Features, `FRaMEDReconnectFeature`, `FRaMEDDeleteConnectionFeature`
	FR07	Zoom into multiple views of a diagram	`StepInFeature`, `StepInNewTabFeature`, `StepOutFeature`
	FR08 to FR10	Sanity checks for operation's executions on model elements during their whole life cycle	*Graphiti* Pattern, *Graphiti* Custom Features, `FRaMEDReconnectFeature`, `FRaMEDDeleteConnectionFeature`
Palette Properties	FR11	Decision if GE features are visible in the palette	`FeaturePaletteDescriptor`, *Graphiti* Pattern
	FR12	Decision if GE features are visible in the context menu	*Graphiti* Custom Feature
	FR13	Appearance of GE features in the palette	`FeaturePaletteDescriptor`, *Graphiti* Pattern
	FR14	Appearance of GE features in the context menu	*Graphiti* Custom Feature
Edit Policies	FR15 to FR17	Editor state dependent checks for operation's executions on model elements during their whole life cycle	`.editpolicy` files, *Graphiti* Pattern, `FRaMEDReconnectFeature`, `FRaMEDDeleteConnectionFeature`

Table 5.3: Summarized and collapsed table of the functional requirements associated with *Edit Concerns*.

the implement the delete and reconnect features in a non-centralized manner anyway. To do that, I expanded the *Graphiti* connection patterns by functions dedicated to those two operations. The logic to delete and reconnect relations can now be implemented in the *Graphiti* patterns. However, the centralized classes `FRaMEDReconnectFeature` and `FRaMEDDeleteConnectionFeature` still need to learn about these implementations. Therefore they dynamically load all *Graphiti* connection patterns' classes and call the corresponding methods from them.

The already mentioned artifacts address multiple tasks of the *Graphical Representation* concern. Firstly, the implementation of *Graphiti* features define the visual appearance of model elements when creating, moving, resizing, text editing, reconnecting or deleting them. This is a minimal set of operations defined by the requirements `FR01` to `FR03`. As one can see in the *Tables* 5.1 and 5.2 of the *Realization* Section, *Graphiti* offers methods for all of such basic operations on model elements. However, *Graphiti* also establishes some additional operations which are not directly triggered by the user, like `Update` and `Layout`. These are called either internally by the *Graphiti* framework or as a consequence of a user-initiated operation. The explanation on how the requirements `FR04` to `FR06` are fulfilled is similar. These requirements represent the evolution of the *Abstract Syntax Model* according to some of the same operations. The set of operations which trigger changes of the *Abstract Syntax Model* is a subset of the GE actions altering the *Concrete Syntax Model*, which saves visual information of the model elements. Therefore, it is clear that *Graphiti* offers the operations to implement the described co-evolution between the *Concrete* and *Abstract Syntax Model*. However, in the concrete implementation of FRaMED 2 the changes on both models are often distributed in two different operations. A prime example is the situation of creating a model element. While the element's domain representation in the *Abstract Syntax Model* is generated by *Graphiti*'s `Create` operation, its graphical representation is created by the `Add` method. Only the `Create` operation is user initiated. It also calls the add method. Concluding on the first six functional requirements, *Graphiti* offers all needed operations to manipulate both mentioned models. It uses even more methods to split up the logic editing one of the two models at once.

The same artifacts are also responsible to define sanity checks for all user-initiated operations on model elements. This is requested by the requirements `FR08` to `FR10` and can be easily implemented as *Graphiti* always calls a method checking if an operation on a model element can be executed in the current situation. Consequently, the sanity checks can be coded in these methods, which names start with *can*. Alongside, the edit policy checks are also called at this point. Finally, the remaining not mentioned requirement `FR07` of the *Graphical Representation* concern states how it shall be possible to zoom into multiple views of the same diagram. Especially in the realization of FRaMED 2, this is an important feature as roles have the context-dependent nature. Such contexts are represented as *Compartment Types* in the CROM, which often leads to big and nested grouping structures of model elements. Without the ability to zoom into each context, the GE would be much harder to use effectively. There are three artifacts fulfilling this requirement. The *Graphiti* custom features `StepInFeature` and `StepInNewTabFeature` can be executed on any grouping model element, such as *Groups* and *Compartment Types*. On execution, the part of the GE's diagram belonging to the grouping model element is opened in a new multipage editor. This way the new zoomed in view is made available by the GEPL. If the `StepInNewTabFeature` was used then both, the old and new view, will be left open. Otherwise, the `StepInFeature` closes the old view on the diagram. A third custom feature, the `StepOutFeature`, can be accessed in every zoomed in view to go one view level higher.

The second *Edit Concern* in Table 5.3 addresses the *Palette Properties*. *Graphiti* features are either shown in the palette or the context menu of FRaMED 2. The requirements `FR11` and `FR13` are responsible for the visibility and visual appearance of GE features implemented by *Graphiti* patterns. Such features are present in the editor's palette. Both attributes are partly defined by the `FeaturePaletteDescriptor`. As already mentioned in the *Realization* Section of this chapter, it saves in which palette category and under which editor state a feature is accessible in the palette. The wanted state can be specified by a method checking for certain features in the current configuration and an enumeration value containing possible GE views. The `FeaturePaletteDescriptor` is defined in the *Graphiti* pattern of the feature it belongs to. The pattern itself also defines the palette icon and name of the GE feature. With all those information collected in module artifacts, features implemented by *Graphiti* patterns can be presented correctly in the editor's palette. Yet unmentioned are the *Graphiti* custom features. These can only be accessed by using a context menu. The requirements `FR12` and `FR14` are responsible for their visibility and appearance. Both attributes are defined in the custom feature classes itself. The appearance of such a feature only contains its name, which is easily set up by a custom feature's class. However, the decision if a custom feature should be part of the context menu depends on the current GE state. To check on this, custom features simply implement an operation defining its wanted state, for example, on which model element the context menu is opened on. When building the context menu, this operation is called on the dynamically loaded custom features.

Finally, the last three functional requirements associated with *Edit Concerns* belong to the *Edit Policies*. `FR15` to `FR17` overall request to check the editor state for every execution of an operation on model elements. Depending on the state, it has to be calculated if the operation should be executable in the current situation or not. The hooks for these checks are the operations, which names start with *can*. These exist in the *Graphiti* patterns, `FRaMEDReconnectFeature` and `FRaMEDDeleteConnectionFeature`. However, the implementation of the checks itself can be found in its own files. These files define edit policies containing multiple rules to analyze the editor state against. For a more elaborated explanation on the edit policies, see the *Realization* section 5.2.

Concern	IDs	Requirement subject	Artifacts
Feature Model	FR18	Definition of a structured and constrained feature model	*FeatureIDE* feature model
	FR19	Validity check for configurations	`FRaMEDFeatureEditor`
	FR20	Calculations of automatic selections and eliminations in configurations	`FRaMEDFeatureEditor`
	FR21	Definition of a standard configuration	Editor diagram file
	FR22	A dynamic configuration editor	`FRaMEDFeatureEditor`
	FR23	Signaling configuration changes to artifacts of other concerns	`ChangeConfigurationFeature`
Abstract Syntax Model	FR24	A metamodel suitable to the domain, the GEPL is tailored to	*Intermediate Object Role Model*
	FR25	A metamodel suitable as intermediate representation between the *Concrete Syntax Model* and the *Target Metamodel*	*Intermediate Object Role Model*
Target Metamodel	FR26	A comprehensive and reusable metamodel to the domain, the GEPL is tailored to	*Compartment Role Object Model*
	FR27	A metamodel to derive models from, which are suitable as target model of the model transformation between the *Abstract Syntax Model* and the *Target Metamodel*	*Compartment Role Object Model*
Model Transformation	FR28	Definition of how instances of the *Abstract Syntax Model* are transformed to an instance of the *Target Metamodel*	ETL rules, `TransformationExecutor`
	FR29	Triggering the transformation every time a diagram is saved	`MultipageEditor`

Table 5.4: Summarized and collapsed table of the functional requirements associated with *Language Family Concerns*.

Language Family Concerns *Table* 5.4 summarizes all functional requirement belonging to the *Language Family Concerns*. The first of the four *Language Family Concerns* addresses the *Feature Model* and its derived configurations. The feature model itself (`FR18`) is created by using the framework *FeatureIDE*. It allows creating constrained feature models in a tree structure comfortable. I already dealt with the framework in the section 5.1.3. It also helps to implement the requirements `FR19` and `FR20`. On one hand, these requirements define that configuration can be checked for their validity. On the other hand, there are selections and eliminations automatically following user-initiated configuration changes. These have to be calculated. Fortunately, *FeatureIDE* already implemented such functions. Therefore the `FRaMEDFeatureEditor`[33] just needs to take over the results of the *FeatureIDE* implementations and show the validity and automatic configuration changes to the user.

Continuing with requirement `FR21`, there needs to be a defined standard configuration as such is used to initiate a new GE diagram. In FRaMED 2, the standard configuration is saved in a fixed diagram file only containing the configuration, but no model elements. This file can easily be read during the creation of a new diagram. `FR22` is implemented by the already mentioned artifact `FRaMEDFeatureEditor`. Overall, it implements a dynamic editor for configurations, which can handle structured and constrained feature models. In relation to requirements `FR19` and `FR20`, the `FRaMEDFeatureEditor` shows if a configuration was evaluated as valid or not in a text label on the top of the configuration editor. It also updates the shown feature configuration imme-

[33]This is the dynamic configuration editor described by requirement `FR22`.

diately on automated selection or eliminations. Finally, the configuration editor differs between free and locked selections of features in the configuration visually. Therefore, I can conclude that the `FRaMEDFeatureEditor` fulfills all sub-requirement of `FR22` listed in the *Requirement* Chapter. Furthermore, configuration changes have to be signaled to dependent artifacts (`FR23`). To enable that, some *Graphiti* patterns implement the interface `ChangeConfigurationListener` and register themselves to be listeners at the custom feature `ChangeConfigurationFeature`. It executes the configuration changes and notifies the listeners by calling a specific method on them. By calling this method, it also propagates the configuration changes to the listeners.

The next concern is about the *Abstract Syntax Model*. The *Intermediate Object Role Model*, implemented by an *Ecore* metamodel, fulfills both requirements `FR24` and `FR25` on such a model. It is a fitting domain model as it captures all relevant concepts of RMLs. IORM shapes and relations have a type, which represents a specific concept from the domain. Such can be *Compartment Type, Role Type, Inheritances* or *Relationships* for example. By using segments of shapes, attributes and operations can be defined. Other properties of model elements can be used to represent occurrence constraints or reference other model elements as source or target placeholders. How the metamodel of the IORM exactly looks like can be seen in the *Realization* section of this chapter. From analyzing the metamodel, I conclude that the IORM is an accurate representation of the RML domain, while not containing useless visual information, like position or size of elements. Besides that, the model also needs to be suitable as an intermediate representation of the GEPL. On one hand, it needs to be possible to execute the co-evolution between it and the concrete syntax model. On the other hand, it should also allow to derive suitable source models for the model-to-model transformation to the CROM from it. However, I already discussed how the definition of the intermediate representation share common ground with both, the concrete syntax model and the CROM, in the paragraphs about the IORM in Section 5.2. By following this argumentation, I consider that the IORM is suited to play the role of an intermediate representation in FRaMED 2.

Similar to the abstract syntax model, the *Target Metamodel* also should be a suitable domain model. It is important to note, that the demands on the target metamodel are higher. This can be explained as instances of it are the end result of the edited diagrams. As such, it is logical to use them as software artifacts when developing an application using role models. Therefore, the reuse potential is an important factor when evaluating the target metamodel. These demands are defined by the requirement `FR26`. The implementation of the target metamodel, the *Compartment Role Object Model*, offers the definition of a fitting domain model, while also putting a focus on its reuse. I conclude the first statement, as Kühn *et al.* [Kühn et al., 2014] developed CROM by analyzing 26 classifying features of RMLs. In later works one more feature was added to the list. I discussed the 27 classifying features in the Subsection 5.1.1, as they were used to derive a comprehensive domain and feature model for RMLs from. Following this procedure, it is clear to see that the metamodel, associated with the complete feature model of the CROM family, defines a suitable domain model for RMLs. Furthermore, the instances of the CROM can also be used for purposes outside of FRaMED 2. One example for such usage is the offered code generation based on instances of the CROM. An implementation for the code generation to SCROLL [Leuthäuser and Aßmann, 2015] as well as RSQL [Jäkel et al., 2014] can be found.[34] However, there is also the requirement `FR27` which states that the instances of CROM have to be a suitable as target model to the model transformation of FRaMED 2. This is the case as the implementation of the model transformation is tailored to the CROM as target metamodel.

The last functional requirements are associated with the *Model Transformation* concern. `FR28` describes the model transformation itself. It states that artifacts of the concern implement if and how the model elements in the instance of the abstract syntax model are transformed into elements of a target metamodel instance. The decision if a model element is transformed is dependent on the current feature configuration. This requirement is implemented by the transformation rules, written in the *Epsilon Transformation Language*, and the `TransformationExecutor`. The

[34]URL: `https://github.com/Eden-06/CROM`, lastly visited: 02.07.2018.

latter one dynamically finds as well as copies the transformation rules and initiates the model transformation using them, starting from the base rule of the transformation. The rules often check for model element types and features in their guards. Firstly, this way it can be ensured, the correct rules are applied to the correct elements. Secondly, it allows the transformation to be feature dependent. The rules further implement what model element of the target metamodel is created and how its properties and relations are defined. Finally, the requirement `FR29` states that the transformation is applied on a diagram every time the user saves it. Since the save operation is part of the `MultipageEditor` class, the same class triggers the transformation by using the `TransformationExecutor`.

Non-Functional Requirements

Similar to the tables that are part of the evaluation of the functional requirements, I summarized all non-functional requirements in the *Tables* 5.5 and 5.6. For those requirements it is possible to, I also listed artifacts that offer solutions for them.

User-oriented requirements The first of those tables (5.5) deals with user-oriented requirements. Overall there are five top-level requirements of this kind. The first one addresses the intuitiveness of the GE's GUI. This requirement is split up into three properties of the GUI. Firstly, it should be clear how to access functions of the GE. In FRaMED 2, the user reaches all functions by well-known interaction points. There is the palette, the context menu, double clicks and finally tab and tool bars. The palette allows access to planned *Graphiti* features, while the context menu and double-clicking on a model element enable to use *Graphiti* custom features. Additionally, there are two tab bars, one to change between different `MultipageEditors` and the other one to open different sub-editors for one diagram. Examples for the sub-editors are the configuration and diagram editor itself. The toolbar allows the user to save the current model or get access to the wizards to create new diagrams. The second property, that ensures an intuitive usage of the GEPL, is that the user directly knows what a certain function does. This is achieved by a clear visual appearance of the GE functions. For GE features that are part of the palette, the visual appearance is defined by its shown name and icon. Meanwhile, custom features are only represented by their name. Overall, the presented names of features should be unambiguous. For FRaMED 2, this is the case as the names of features mostly follow the RML domain concept's names. For other functions, like saving a diagram or triggering the transformation, this is trivial as the icon for the first-mentioned function is well-known and the transformation is always triggered by saving a diagram. Additionally, dialogues, e.g. to edit model elements or create a new diagram, are also clearly written and user-friendly in this case study's product. An example of that is the `EditFulfillmentDialog` which allows editing the referenced roles of a *Fulfillment* relation by choosing possible options from a prepared list. This solution avoids that the user needs to type the names of role types by hand when editing a *Fulfillment*.

Continuing with the next requirement, `NFR2` describes how clarity is an important requirement of the GE's GUI. It can be achieved by either allowing the user to hide GUI elements by choice or never show all of them at once. In FRaMED 2, this concept is implemented on two levels. On one hand, it is possible to hide feature entries in the palette. By this, I do not mean currently unavailable features as the GEPL's configuration rules them out. Instead, the user also has the possibility to hide available feature entries by collapsing palette categories. This is an ability offered by the `FRaMEDDiagramEditor` based on an implementation by *Graphiti*. On the other hand, there is the conceptual idea that not all model elements should be visible for big and nested diagrams at once. In the domain of RMLs, one cannot avoid grouping model element into another. The reason for that is the contextual nature of roles. However, the solution for this the possibility is to zoom in and out to multiple editor views. For such an implementation, see the discussion of the functional requirement `FR07` in this subsection.

The third non-functional requirement `NFR3` is about the transparency of the GE's actions and state. The requirement concretely lists three kinds of GE actions. The first two describe GEPL

IDs	Requirement subject	Artifacts
NFR1	Intuitiveness of how to access GE functions, what they do and intuitiveness of dialogues	MultipageEditor, FRaMEDDiagramEditor, FRaMEDFeatureEditor, Wizards and Dialogues[a]
NFR2	Clarity by limiting the amount of visible palette entries and model elements	FRaMEDDiagramEditor, *Step* features[b]
NFR3	Transparency	
NFR3.1	Its clear to the user when the diagram or feature configuration is changed and changes are saved.	MultipageEditor, FRaMEDDiagramEditor, FRaMEDFeatureEditor
NFR3.2	The feature configuration, editor view, a context menu's model element and if there are unsaved changes are always visible to the user	MultipageEditor, FRaMEDDiagramEditor, FRaMEDFeatureEditor
NFR4	Fault tolerance: The GE can handle unreasonable user input without compromising its state.	*Edit Policies*,*Graphiti* Pattern, *Graphiti* Custom Features, FRaMEDReconnectFeature, FRaMEDDeleteConnectionFeature, EditFulfillmentDialog
NFR5	Performance: Time and Memory space consumption	

[a]Examples for such are RoleModelWizard, EditRelationshipDialog and EditFulfillmentDialog.
[b]Explicitly these are StepInFeature, StepInNewTabFeature and StepOutFeature.

Table 5.5: Summarized and collapsed table of the user-oriented non-functional requirements.

features that execute changes to the current diagram and configuration. This means altering the concrete and abstract syntax model as well as the feature configuration always triggers an immediate update of the GUI, according to the change. For altering the diagram, this is part of *Graphiti*'s implementation on which the FRaMEDDiagramEditor is based on. However, the FRaMEDFeatureEditor is implemented in such a way that it updates its tree representation of the feature configuration every time the *Graphiti* custom feature ChangeConfiguration is executed. The third action listed by the sub-requirement NFR3.1 is saving the diagram. When it is applied, it is shown by the MultipageEditor's commonly known save icon. Furthermore, this last part of the requirement also means that the editor should avoid saving the diagram by itself or at least warn the user before doing so. This is relevant as saving the diagram is needed to allow the user to zoom in and out of a model element, using the *step* features. In the current implementation, these features are not accessible when the diagram's models contain unsaved changes. I justify this decision as it preserves transparency to the user. Unfortunately, it does also compromise the intuitiveness, as it might not be directly clear to the user why it is not possible to zoom in and out of an element. Overall, I evaluate that the requirement NFR3 does profit more by this decision that the requirement NFR1 is compromised by it. Not only the GE action should be transparent but also its state. The visible state contains information on which element a context menu is opened, which is implemented by the framework *Graphiti* in its DiagramEditor class. Additionally, the editor view and if there are unsaved changes of a diagram, is presented by the MultipageEditor. To publish the editor view, the artifact shows the name of the element in which the user zoomed in as the MultipageEditor's name. If the user works on the top-level of a diagram, the role models name is chosen instead. The information if a diagram contains unsaved changes is displayed in the *Eclipse* own status bar on the bottom of the application. Finally, the last part of the visible state is the current feature configuration. It can always be accessed by opening the FRaMEDFeatureEditor.

NFR4 addresses the fault tolerance of the GEPL, meaning the ability to handle unreasonable user input. Such situations should not lead to failures and inconsistent states of the editor. I want to talk about three ways to avoid such problems in FRaMED 2. Firstly, there are the *Edit Policies* which do not only forbid to apply *Graphiti* features if the configuration contradicts it. It furthermore also checks for specific types, for example, it only allows to draw an *Inheritance* between two model elements of the same type. Therefore, there can not be illogical inheritance relations between a *Natural* and a *Role Type*. See the considerations on the functional requirements FR15 to FR17 for more information. Similar to the edit policies, there are the sanity checks implemented by all *Graphiti* patterns and features. They avoid executing operations when integral information is missing. In a situation like that the execution could easily lead to a compromised diagram or editor state. The associated functional requirements are FR8 to FR10. The third way to avoid situations in which unreasonable user input leads to failures is implemented by the EditFulfillmentDialog. As already mentioned it prepares a list of playable roles from which the user can choose from when editing a *Fulfillment*. On one side that is a user-friendly feature, on the other side, this also prevents illogical role references of *Fulfillments*.

The final user-oriented requirement NFR5 defines performance requests. While it is easy to explain how the requirement addressing the memory space consumption is achieved, it is more difficult to discuss the time consumption. The first mentioned performance aspect is about the size of the produced diagram artifacts. These artifacts are two files. Firstly a file containing the instances of the concrete and abstract syntax model as well as the feature configuration. A second file saves the transformation result, namely an instance of the CROM. These are pure text files based on the *Extensible Markup Language* (XML). Even for big diagrams, these files usually don't exceed a size of 1 MB.

To discuss the time aspect of the GE's performance, I want to talk about possible bottlenecks of the application and how they are handled. The not summarized requirement NFR5 presented in the *Requirement* Chapter 3 states that every GE action should only take a reasonable time depending on its execution frequency. I want to start with actions that are only triggered infrequently or even just once per diagram. The basic example for such an operation is the creation of a diagram. When the user creates a new diagram only one file is generated, namely one with the extension *crom_diagram*. The RoleModelWizard generates a concrete syntax model instance, while the *Graphiti* pattern ModelPattern is used to create the abstract syntax model instance and the feature configuration of a diagram. The initial configuration depends on the artifact described in the functional requirement FR21. All these generated structures are small and predetermined. This means that there are no big calculations to execute. Therefore, I would argue that the creation of a diagram in FRaMED 2 is not a performance problem. The second file of a diagram is created by the transformation, which will be discussed later in the next paragraph. The next possible bottleneck are the dynamic loading processes. To avoid the problem, the implementation of FRaMED 2 minimizes how often classes are loaded. The FeatureProvider only loads the *Graphiti* patterns on its instantiation and saves the references in a list, which is public for other artifacts. By doing that, other artifacts don't need to dynamically search and load *Graphiti* patterns. The same goes for the ToolBehaviorProvider which does the same for *Graphiti* custom features. Besides the patterns and custom features, the *References*, which publish selected implementation details of a module, need to be loaded dynamically too. This is often the case for *Graphiti* patterns which implement features that are dependent on other ones. In such a case, the pattern itself searches for the references only once when instantiated. Overall, the careful handling of this type of GE action, allows to eliminate it as a bottleneck.

Yet unmentioned are more frequent actions. Firstly, there are operations moderately often executed. Examples for such are building the palette as well as context menu and editing a single model element or the feature configuration of a diagram. All of these are fairly easy operations to execute. When building the palette or context menu, only simple checks on the feature configuration and editor view are done for every fitting *Graphiti* feature. Changes in the configuration, also including automatic selections and eliminations, only influence a small number of configuration features at once. Editing a model element can be a bottleneck potentially as many other edits might be the consequence. However, the current implementation of the GEPL is oriented to avoid this problem as good as possible. In only a few cases this can not be prevented completely, for example, when deleting a model element that groups other elements together. When deleting the grouping element, all inner model elements have to be deleted too. In the case of nested grouping elements, there same goes for the inner grouping elements. Meanwhile, most operations on model elements do not lead to other changes in the diagram. Consequently, this is not necessarily a big negative impact on the performance. Saving a diagram does also belong to the more frequent GE actions. It leads to synchronizing the configuration of the feature editor with the configuration in the diagram and triggers the transformation. While synchronization is an easy comparison task which can not really be seen as dangerous to the performance, the transformation could.

However, the usage of a specialized transformation language helps to counter that. Besides the claim, that an engine such dedicated usually indicates an effective implementation of its task, FRaMED's concrete realization is also time-saving. Its systematic process of following the structures of model element deeper and deeper is implemented in respect to never transform a model element twice with the same rule. This is avoided by using the `equivalent` operation by the *Epsilon Transformation Language*. Once an element is transformed for the first time, its target representation can be accessed via the operation, which does not trigger the transformation for it again. This can be used for the most cases, but not in the case of inheritance of 2 non-exclusive sub-rules. In this case the ETL feature of extended properties is used.

Finally, it is important to explain why the modularization is not a performance problem of the model-to-model transformation. It leads to two effects. Firstly, the transformation rule files has to be found, copied and referenced by a string manipulation in the base rule. All those operations can be executed in a quick manner as the files are small and the string manipulation is limited to the head of the base rule. Secondly, as the big rules of the not modularized implementation of FRaMED are split up to enable modularity following the feature model structure, more rules are executed per transformation process. While this creates some overhead of more rules to trigger, the intrinsic operations of the rules take the same time to execute. Consequently, I conclude that the transformation also fulfills the performance requirement. Lastly, there are GE actions executed multiple times per second. Such are sanity and edit policies checks. Those are triggered on moving the mouse for example. This is no problem as these only contain small checks for a few features, types or missing information.

Developer-oriented requirements The fulfillment of the first developer oriented-requirement `NFR6`, which addresses the operating system independence, results from the choice of implementing it on the *Eclipse* platform. The platform runs FRaMED 2 on a *Java* virtual machine. That is why, a operating system able to run *Eclipse*, can also execute FRaMED 2. The following two requirements are strongly connected to each other. `NFR7` describes that is should be possible to add, remove and exchange GE features statically. This reconfiguration process should be executed in an automated manner, meaning to add a feature, the developer of a GEPL just places module artifacts, that implement the feature, in a specific folder or structure. To remove the feature again, it is enough to simply delete the module from that folder again. This is implemented in FRaMED 2 as it is easily possible to remove modules, for example, *Relationship Constraints*, by marking the packages `relationship.inter_relationship_constraints` and `relationship.intra_relationship_constraints` to not be used. This is done by renaming a package to start with an underscore. Alternatively, module packages can also be deleted completely. In both cases, the GEPL still works as before with the exception of *Relationship Constraints* not being part of the palette or accessible in any other way.

This property is enabled by the fulfillment of the last non-functional requirement `NFR8`. It defines three principles for feature modularity. The first two are a part of Kästner's work [Kästner et al., 2011]. `NFR8.1` states that artifacts associated to one feature are grouped together in fea-

IDs	Requirement subject	Artifacts
NFR6	Platform Independence: The GEPL implementation shall be independent of the operating system running it.	
NFR7	Reconfigurability: The feature set of the GEPL shall be changeable statically in an automated manner.	
NFR8	Feature Modularity	
NFR8.1	Location and cohesion: The artifacts of one feature are located and grouped in one structural unit.	Module packages
NFR8.2	Information hiding and encapsulation: The feature modules differ between internal implementation and external interfaces.	*References*[a]
NFR8.3	The application core's and feature modules' artifacts shall not reference another features module's internal artifacts directly.	*References*[a]

[a]Examples for such are `AbstractGroupingFeatureReference` and `AbstractAttributeAndOperationReference`.

Table 5.6: Summarized and collapsed table of the developer-oriented non-functional requirements.

ture modules. The implementation of FRaMED 2 uses packages as a structure to realize such modules. Access to the modules artifacts is enabled by dynamically loading. The second principle by Kästner (`NFR8.2`) talks about the different treatment of a module's internal implementation and external interfaces. While internal details should be hidden to artifacts outside of it, external interfaces can be used to grant limited access to specific implementations of a module. These interfaces are called *References* in this case study's product. These *References* are classes that can be dynamically loaded when an artifact is dependent on a module's feature implementation. By dynamically loading them, using an abstract supertype in the application core, direct references can be avoided. This way its still possible to easily remove modules which contain *References*. Related to the second feature modularity requirement is `NFR8.3`. It forbids that an artifact references another module's internal classes, operations or attributes. If this would be allowed everytime a feature module would be removed, references to it have to be changed by hand. FRaMED 2 was implemented in respect to these limitations of access. However, the requirement `NFR8.3` mentions the application core. While its artifacts can not reference the module's implementation, it does work the other way around. This means the module's classes can always use the application core as it is seen as a fixed part of the GEPL.

Overall, I evaluate that all functional and non-functional requirements are fulfilled by FRaMED, when including the presented modularization solutions for the *Feature Model, Abstract Syntax Model* and *Target Metamodel* concerns. While this also includes the requirements addressing the feature modularity, there are limitations of this case study's modularization. While I consider them to not violating the modularity requirements, they are important to talk about and especially provide possible solutions for them.

5.3.2 Limitations of the Modularization

In the following, I will give a description and solution for limitations of the modularization in the case study. These limitations can be the consequences of this thesis' focus, the GE framework choice, or the decision to avoid implementation effort which is out of the scope for this work. The following discussion on them will be structured by the way the limitations violate the criteria for feature modularity. Independent of those, it is important to note again that the three of the *Language Family Concerns* are not modularized as there are known feasible solutions for the modular design of those. One possible solution for each is presented in the *Future work* Section 6.3.

Locality: Occurrence Constraints The first problem of the modularization is a limitation on the principle of locality. It is a direct consequence of using *Graphiti* as GE framework. When creating, adding and direct editing occurrence constraints of a *Role Type*, code implemented in the *Graphiti* pattern of the *Role Type* is applied. In contrast, the principle of location states that the code addressing occurrence constraints should be placed in an own feature module as they are associated to a distinct feature in the variability model of CROM. While the problem could be solved for the adding and direct editing implementations by using inheritance structures for the *Graphiti* patterns, this is not a complete solution as there can only be one `create` operation for features addressing a distinct model element. The reasons is that there is a palette entry for each `create` operation. The occurrence constraints are not a feature with such a palette entry, but represent an attribute of another model element. This is a *Graphiti* specific limitation, causing that the code to create occurrence constraints cannot be implemented in a `create` operation of *Graphiti*.

Instead, a complete solution implements the code for all three operations on occurrence constraints in an own artifact which is used by the *Graphiti* pattern for *RoleTypes*. In particular, the last mentioned pattern searches dynamically for a *Reference* class in the occurrence constraint module. This *Reference* introduces a level of indirection to enable encapsulation of the modules' implementation and offers, inter alia, a method interface leading to the code creating an occurrence constraint. When the `RoleTypePattern` does create, add or direct edit a model element, the *Reference* is used to access the needed code addressing occurrence constraints. Unfortunately, this means that the *RoleType* module still needs to know that there can be a *Reference* for occurrence constraints, but this constitutes a very loose connection as it is realized by dynamically loaded *References*. This has the advantage that the feature modules for occurrence constraints can be removed, without breaking the `RoleTypePattern`. The same problem occurs for other GE features implementing model elements with occurrence constraint, like relationships and role groups, and can be solved equivalent.

Encapsulation: Attributes and Operations The next limitation emerges when transforming model elements with attributes and operations. As attributes and operations are modeled as isolated features in the CROM variability model, they have to be implemented independent of the model elements they belong to. However, as properties and behavior are always coupled in the feature model, they can be both addressed in one feature module. Therefore, the *Epsilon* operations helping the ETL rules transforming model elements with attributes and operations are located in such a feature module, isolated from the transformation rule using them. The problem is that these rules access the *Epsilon* operations by a fixed hard-coded reference by name. While the feature module addressing attributes and operations, as a core module, is not meant to be removed, it should be able to interchange its implementation, which can involve name changes of such *Epsilon* operations. In such a case, altering the internal implementation details leads to the need for changing other modules implementation, effectively violating the principle of encapsulation.

The easy solution for this is introducing a level of indirection by creating an interface in the form of an ETL file, equivalent to the *Reference* classes used in the *Graphical Representation* concern. The ETL interface offers fixed named operations, transformation rules can reference. By delegating the calls to the corresponding implementations in the original *Epsilon* operations, the functionality behind the referenced operations can be accessed. When the internal names of the original *Epsilon* operations change, only the ETL interface has to be adjusted.

Encapsulation: Features' Names Another modularity limitation associated with encapsulation, is based upon the feature expression used in the `FeaturePaletteDescriptor`, in *Graphiti* custom features and the transformation rules. They all do explicitly reference features' names. Such cannot be foreseen by a module for a feature implemented by another component, using the approach to compose a feature model presented in Section 6.3. Furthermore, from an encapsulation standpoint, the internal name of a component's feature should be replaceable as wanted without creating the need to change artifacts in other modules.

This can be avoided by not referencing features by their names in the feature model, but by more stable tags. Every definition of a feature model fragment contains the tags for its features. When checking for a feature's status in a configuration, a tag is used to access the feature behind it. Firstly, this allows changing the names of the features behind the tag without affecting more than one component. Secondly, situations in which there is no feature found for a tag can be handled flexibly. It can be treated as an evaluation to false. Alternatively, a warning or even an error can be thrown. To avoid breaking the GEPL, the usage the tags can further be involved in the generation of a dependency graph, which is discussed in the general approach for the *Abstract Syntax Model, Target Metamodel* and *Model Transformation* concerns.

Dependencies: *Role Group* Elements The next limitation is caused by too deep connections between artifacts. It addresses the dependencies of model elements, which can be in a *Role Group*, to the corresponding *Role Group* feature module. *Role Types* in such groups has to be treated differently than *Role Type* outside of such, both in their graphical representation as well as their transformation. This can easily be handled by implementing two *Graphiti* patterns for *Role Types*, where one applies if the model element addressed is in a *Role Group* or if the parent element is null. The pattern that implements *Role Types* in groups is located in the feature module of the *Role Groups*, which itself is a sub-module of the *Role Type* component. Overall by this solution, each of those components can be removed without breaking FRaMED 2. After this illustrating example, I can address the actual problem. The same dependency exists for the transformation of relationships in *Role Groups*. In contrast to *Role Types*, the feature module of relationships is not a super-module to the *Role Group*'s one. Therefore, deleting the *Relationship* feature module leads to a situation in which relationships in a *Role Group* can still be transformed but those outside of one not.

To solve that problem, the transformation rules addressing relationships in a *Role Group* are encapsulated in an own feature module. It has to be dependent on two other components, namely the ones that are associated with the relationships and the *Role Groups*. To realize that, the dependency graph solution presented in the general approach for the *Abstract Syntax Model, Target Metamodel* and *Model Transformation* concerns can be used.

System Extension: *References* Using the current implementation of FRaMED 2, a developer of a new component is limited to the given set of abstract supertypes for *Reference* classes. They are defined in the `UI` package, which represents a big part of the application core. As the supertype of references is used as metadata to search them dynamically, this limitation cannot be ignored. Not being able to remove unneeded *References* automatically is not a big problem, seeing that six of seven references are associated to core modules in the current state of the GEPL. However, the extensibility for those artifacts is of big interest.

Changing the metadata based upon which the *References* are searched is a way to solve the problem. By getting rid of any supertypes and using tag annotations on the concrete *Reference* classes, they can be referenced in a more free manner. Presuming cooperating feature modules settle on the use of the same tags, the rest of the work with the *References* stays the same. If a non-essential component, containing a *Reference*, is removed from the GEPL, this is not a problem as finding no *References* associated with a certain tag does not have to lead to an error.

System Extension: Palette Categories To allow components to add their own palette categories to FRaMED, the current definition of categories in an enumeration fixed at compile time has to be omitted. Instead, each feature to be shown in the palette can use its *Graphiti* pattern to define the name of the palette category it belongs to, e.g. in the `FeaturePaletteDescriptor`. When the `ToolBehaviorProvider` builds the palette, it generates a palette category every time a category name is found for the first time. Concluding, the feature entries are added to their corresponding created categories.

System Extension: Palette Views Palette views correspond to the editor view and capture if the user stepped into a *CompartmentType* or not. They are used in the `FeaturePaletteDescriptor`. The current situation of a hard-coded enumeration defining palette views, has to be changed for a solution. To be able to add elements to the set of palette views, the `GroupingFeatureReferences` have to be extended. Such *References* are used in modules of model elements that the user can step in, which would change the editor view. By allowing the *References* to specify the editor view by an identifier corresponding to their model element, the set of views can be extended. When a `FeaturePaletteDescriptor` references palette views, it uses the identifier for those. If no palette view can be found for a given identifier, searching in the `GroupingFeatureReferences`, this situation can be handled differently depending on how essential an associated feature module is.

5.3.3 Results

Overall, I evaluate the case study as a success, in terms of proving the feasibility of the elaborated design approach in Chapter 4. It follows the proposed top-down design approach for a dynamically configurable family of GEs, which avoids a monolithic implementation. Furthermore, it achieves fulfilling all defined requirements of the GEPL domain. To prove both parts of this statement, I want to start giving a brief overview on how the implementation described in the case study matches up with the presented design approach. It will be structured alongside the concerns of a GE and list mainly the commonalities between the design approach and the documented implementation in this chapter.

Accordance with the Design Approach

Graphical Representation The hooks of user initiated GE actions and the associated mapping are both provided by *Graphiti*. The GE actions are implemented by *Graphiti* features, which allow implementing how to alter the concrete and abstract syntax model as well as the sanity checks in their operations starting with *can*. For most of the GE actions, the model modifications for each model is defined separately, e.g. by the operation pairs of `create/` add and `directEdit/` `update`. The zoom functionality is realized by the `StepInFeature`, `StepInNewTabFeature` and `StepOutFeature`, which are all three implemented as *Graphiti* custom features. They search the suitable diagram fragments in a flat structure. The concern furthermore applies the *Reference* concept to differ between internal and external parts of a feature module as well as it allows dynamically loading of such *References* based upon supertypes.

Palette Properties In accordance to the presented design approach the `ToolBehaviorProvider` is an application core artifact, building the palette structure and adding its entries as well as calculating the context menu. To do this, it loads all planned *Graphiti* features and custom features dynamically based upon their supertypes and iterates over them. Depending on the analysis of their `FeaturePaletteDescriptor`, which is made accessible by a *getter* method, planned *Graphiti* features' entries are added to the palette with a specific icon and name. The *Graphiti* custom features offer their `contextMenuExpression` to calculate if they are part of a certain context menu depending on factors, described in the design approach.

Edit Policies The suitable representation specified in the matched design methodology is a, for this purpose developed, DSL, which can analyze model types, parent elements and a feature configuration for example. This analysis step's result is a boolean expression to decide if a GE action can be executed. The conjunction of the edit policies using the logical *or* is important as it does not allow any new added rule to invalidate another one. The edit policy language further fits the provided description of it, as it allows satisfiability and consistency checks on a set of policies. If policies for a model element depend on its parent element, the solution locating them in the associated feature module of the parent element is chosen, for example for role constraints in *Role Groups*.

Feature Model As proposed by the design approach a framework, namely *FeatureIDE*, is used to define a variability model and manage its configurations. This includes validating and calculating automatic selections and eliminations. The configuration editor is implemented as *Eclipse* `EditorPart`, comprised of a tree structure with checkboxes associated with a selection listener. FRaMED 2 follows the approaches' description of the diagram specific configurations. Listeners for configuration changes, artifacts implementing occurrence constraints, for example, are noticed via the observer pattern. Furthermore, the GEPL also defines a standard configuration. Finally, the description of an implementation for a modular feature model (6.3.3) follows the proposed idea of using the approach of Bagheri *et al.* [Bagheri et al., 2011].

Abstract Syntax Model The IORM is defined using a modeling framework, in particular, the EMF. It follows the style of model elements with inner and associated elements, suitable to the concrete syntax model provided by *Graphiti*, while also using a lightweight definition of domain concepts. Furthermore, the IORM bridges the differences of flat diagram structures in the concrete syntax model to hierarchical model element structures. The considerations on future work, in the Section 6.3, state that the abstract syntax model can be built on the startup of the GEPL by using a delta modeling approach, namely *DeltaEcore*.

Target Metamodel The CROM is an implementation for the target metamodel which takes accessibility, comprehensibility and reusability into account. It offers an own notation, which can be created with a associated family of RMLs, to address accessibility. It further is comprehensive as it was created with the help of an extensive analysis of the RML domain and uses the advantages of a heavyweight definition of domain concepts. Its reusability is increased by approaches like formalCROM[35] which is a foundation to generate role base programming code based upon CROM instances. In Section 6.3, an approach of using the existing metamodel family for the creation of the target metamodel is discussed. Fittingly to the design methodology, the family applies delta modeling.

Model Transformation Accordingly to the matched design approach, the ETL is a rule-based transformation engine, which allows defining a rule's source and target elements, guard expression, actions and rule inheritance structures. The guard expression is checked automatically before every rule execution and can be used to check for chosen features among other circumstances. Rule inheritance enables splitting up rules and locating them in modules they depend on, for example relationships in *RoleGroups*. When saving a diagram, the `TransformationExecuter`, an application core artifact, does generate a base rule with references to all other transformation rules, which are searched for and copied dynamically. Due to ETL, every rule can now access every other rule.

Accordance with the Requirements

After showcasing to what high degree the case studies' implementation matches up with the beforehand presented design approach, I want to summarize how successful the documented implementation is in regards to the requirements of the GEPL domain. FRaMED 2 in its current state, clearly fulfills all functional requirements. An elaboration on this can be read in this section (5.3.1). It is not that easy for the non-functional requirements as I omit measurements for them in this thesis, because this would be beyond the scope and is not target oriented. Instead, I extensively discussed how the non-functional requirements are taken into account when developing and modularizing FRaMED 2. Considerations on this can also be found in 5.3.1. Therefore I will skip the user-oriented non-functional requirements as well as the demand for platform independence. I consider all these already mentioned non-functional requirement as addressed by the case study's product in a sufficient degree.

[35]URL: `https://github.com/Eden-06/formalCROM`, last visited 29.07.18.

Instead, I want to focus on the modularity of the implemented GEPL. Overall, four of the seven GE concerns are realized in such a manner, that allows modular system extension and specialization. These results alone do not justify to evaluate the requirements of reconfiguration and modularity as fulfilled. The remaining monolithic implemented concerns are comprised of fixed a feature model, abstract syntax model and target metamodel. This leads to the problem that new components have to reference features, and model elements in the existing representation, which effectively shuts down the system extension. However, at the moment system specialization is possible, with the limitation that mentioned models will not shrink with the set of feature modules. When taking the proposed and elaborated solutions in the *Future Work* Section 6.3 into account, system extension and specialization is enabled. They do further allow to fulfill all criteria for feature modularity, namely locality, encapsulation and dependency handling. Therefore, I consider the last two requirements as met too.

Overall I showcased a modular implementation for a dynamically configurable GEPL, following a top-down design approach. As the development and modularization process of the GE family matches up with the design approach closely and the GEPL fulfills all requirements defined for its domain, I conclude that the applied design methodology is feasible in an application scale.

6 Conclusion

In this chapter, I will conclude the thesis by summarizing the results of the of previous chapters in regard to the presented design approach. Following it, all the contributions of this thesis will be listed and their meaning to the work will be briefly described. Finally, I present future work ideas, which result from this thesis.

6.1 Summary

This thesis successfully presents a top-down design approach for dynamically configurable and modular GEPLs. The success of the design approach is dependent on how good it is fulfilling the desired properties and if the methodology is feasible. Overall the design approach covers the criteria by describing in detail how the functionalities of a dynamic GEPL are implemented, how the artifacts realizing that are modularized and how the feasibility is ensured.

6.1.1 Desired Properties

In the following, the particular desired properties will be listed. Additionally, an analysis, if the properties are taken into account by the design approach, will be provided.

Top-Down Design Method While requiring the developer to provide the feature model and associated mappings, a top-down design process offers a maximum of development flexibility with a minimum of initial implementation effort for the mentioned process. This comes with the cost of high development effort for every component. The design approach in this thesis clearly deploys a top-down design method as it composes a feature model based on manually implemented fragments of it.

Dynamical Configuration Being able to change the configurations of the GEPL during runtime, allows for a high degree of flexibility towards the user's specific domain interpretation. The design approach mentions a dynamic configuration editor on which up to five of the seven GE concerns are dependent. Artifacts of those concerns are designed to analyze the feature configuration during runtime and adjust their behavior or composition accordingly, which is an explicit part of the design method.

Complete Modularity A completely feature modular PL is implemented by a set of clearly distinct feature modules and allows to add and remove those components from the mentioned set. This property can only be ensured by following the principles of locality and encapsulation as well as explicitly defining and handling dependencies between feature modules. By describing modularity mechanism for all artifacts in the components of a GEPL, modularity can be realized as an explicit part of the design approach. In a discussion, all concerns and their artifacts are examined on fulfilling the given criteria for feature modularity. This assessment results in the statement that all artifacts in components are treated in a way enabling complete modularity for a GEPL designed by the presented approach.

GEPL Domain This domain can be split up into seven distinct concerns and a domain analysis of it has identified 29 functional and eight non-functional top-level requirements, which can further be refined by multiple sub-requirements. An analysis of the design approach in regards to the requirements, which are taken into account by the design decisions of the approach, reveals that this is the case for 33 requirements. Solely four non-functional requirements are not taken into account, while also not being blocked. Furthermore, those requirements are strongly dependent on implementation details, which are not part of the design approach.

6.1.2 Feasibility

The feasibility of the presented design approach is proven by an extensive discussion to assess it and partly proven by a case study. The assessment is comprised of concern internal feasibility considerations and reflections on the practicability of concern cooperation. Firstly, all internal solution parts for a concern's tasks and modularization mechanisms are checked for feasibility. Secondly, on the practicability of collaborations between artifacts in different concerns is reasoned. By considering existing solutions to reuse, trivial tasks and challenges to implement manually, the feasibility can be assessed. Overall the discussion concludes that the design approach is completely feasible.

The second part of the feasibility prove is represented by a case study. In it, all seven concerns are implemented according to the tasks and solutions described by the design approach. However, only four of the seven concerns are also modularized. While seen as feasible, the modularization is not realized for the concerns defining models. As there existing solutions for modularization of those, which implement own case studies, the scientific value of implementing them is low. Instead, possible implementations are described later in this chapter.

Overall, the design approach, presented in this thesis, has the desired properties, assessed by extensive investigation on each property and its role in the methodology. Furthermore, the approach is also feasible, which is examined in detail on every part of the presented solutions and modularization mechanisms. Additionally, a case study proves the practicability for four of seven concerns, while not stating that it is impossible to realize the approach for the other three concerns. Therefore, I conclude that this thesis reached its goal to provide a feasible top-down design approach for modular GEPLs, which can be configured dynamically.

6.2 Contributions

The main contribution of this thesis is an extensive description on a top-down design methodology for Graphical Editor Product Lines, which offers dynamic configuration and a feature modular implementation. In order to provide this main contribution, I conducted a survey on the existing landscape of design approaches for SPLs and LPLs. It features 15 papers, which offer a wide variety of different served domains and abilities. The results of the survey show that the coverage of specialized design approaches for GEPL is limited to non-modular ones. In combination with the complexity of the GEPL domain, the outcome justifies the need for the main contribution of this thesis. The next step was a domain analysis for families of GEs. This analysis provides a set of requirements to evaluate families in the domain against. A case study, implementing an executable GEPL, following the newly developed design approach, substantiates the feasibility of the main contributions methodology. The study further presents an example of the design approaches' usage and illustrates how the concepts of the methodology can be implemented in a concrete case. The case study's code is available online.[1]

[1]URL: `https://github.com/Eden-06/FRaMED-2.0`, last visited: 29.07.18.

The main contribution is characterized as a design approach for the GEPL domain, which is ensured as the requirements of this domain are taken into account heavily. Further, a top-down design method is used. Additionally, the resulting GEPL provides a dynamic configuration and a modular implementation. While there are dedicated concepts in the design approach to implement the dynamic configuration, a large focus is put on the feature modularity. Multiple artifacts and processes are meant to allow system extension and specialization is an automated manner. The comprehensive presentation of the main contribution is comprised of detailed considerations on the concerns, their artifacts, tasks, solutions and modularity mechanisms. Relying on a discussion and the case study, the feasibility of the design approach is assessed as well as proven by example.

6.3 Future Work

This section presents possible extensions of the design methodology, the GEPL domain analysis and the case study. The design approach presented in this thesis is based on a top-down design method, but it would be also an option to use it in combination with a bottom-up process. This option is discussed in Subsection 6.3.1. Furthermore, future work on the set of requirements for the domain of GEPLs is proposed (Subsection 6.3.2). Finally, three GE concerns are not implemented modularly in the case study. This is addressed by describing possible solutions for these concerns in Subsection 6.3.3, enabling full feature modularity of the case study. For the approaches to modularize the concerns' implementations, it applies that the presented solution is seen in the context of the case study and meant to work with the already existing artifacts.

6.3.1 Bottom-Up Design Method

When implementing a bottom-up process for a GEPL, the goal is to allow generating a feature model without being dependent on manually developed feature model fragments. Instead, an automated analysis of the GEPL components leads to a suitable feature model and the mapping between the components and the features. To realize such a process the components or artifacts of them have to be annotated manually to allow the analyzing algorithm to create features for them. Besides the annotations, a dependency graph is needed for the generation of the variability model. Fortunately, the dependency graph can be automatically calculated and is also a part of the top-down design approach. In the following, the feature model generation creates features for every annotated component, places them in a tree structure and adds feature constraints to them. The structure as well as the constraints can be derived from the dependency graph. As by-product the mapping between components and features its created too.

To assess if it makes sense to offer a bottom-up design method, I want to evaluate the advantages and disadvantages of it in comparison to the existing top-down approach. A clear advantage is a reduced effort to implement new components. They do not need to contain a feature model fragment. While the bottom-up process is potentially quite complex, it replaces the also complex feature model composition using the fragments. As the dependency graph is used to create a feature model, the GEPL is based on, unresolvable dependencies which could break the family of GEs can be detected and handled automatically. Given the bottom-up process is transparent to the user and developer, involving no human influence on controlling it, the feature model generation undertakes tasks of the developer, while not limiting the user's experience with the GEPL. However, a bottom-up design method also has disadvantages. Firstly, there is the immense initial effort to implement a bottom-up process. Furthermore, the process might involve human influence [Font et al., 2015, Vacchi et al., 2013], even after the annotation of components. Finally, the development process could be perceived as not intuitive as annotations are used to guide the components mapping to a feature, instead of more a informative and clear feature model fragment.

Overall, I would recommend the use of a bottom-up design method as an alternative to a top-down one if it can be foreseen that the GEPL is meant to be used in a variable domain. In such a domain, new components need to be created often and implementation of existing ones often change. In such circumstances the reduced effort to realize new components has the most positive effect. In regard to this, the high initial implementation effort for the process behind the variability model generation can be compensated, especially as it replaces a feature model composition in the top-down approach.

6.3.2 Requirements

In the following, I will propose additional requirements, to consider as part of the GEPL domain analysis for further works on the topic. Firstly, there is the *portability*, which addresses the usage of the GE on different terminals. They usually differ in screen sizes and formats, which has to be taken into account by an adaptive GUI , e.g. by using suitable palette and model elements. Furthermore, a performance requirement concerning the *resource efficiency* can be useful in regards to the scalability of the GEPL. Another requirement is the *dynamic reconfiguration*, which states that it should be possible to extend and specialize the GEPL during runtime. Finally, *testability* is an integral part of every software system. It is not addressed in this thesis, as it is too complex, especially in combination with the demand on modularity. While portability and dynamic reconfiguration are out of scope, the resource efficiency is too hard to measure under the circumstances of the thesis.

Additionally, I want to briefly talk about the evaluation processes for existing requirements, which could be relevant for further works on the GEPL domain. The evaluation step is easy for functional requirements and the modularity requirements are traceable by checks on the reconfigurability and specific principles. For other non-functional requirements, this is not applicable. Usability requirements, like transparency and clarity, can be best evaluated in an empiric study, while performance demands should be measured. However, both requirements do not describe a focus of this thesis.

6.3.3 Modularization

Feature model

A modular feature model allows it to extend an SPL by features, that are not yet referenced in the feature model of the PL. A new module would define its own feature to be integrated into the feature model, while also implementing it. If a feature module is removed, the part of the feature model associated to the deleted module also disappears. In the following, this extension and specialization as well as the generation of a fitting standard configuration will be elaborated. All other, in the case study mentioned, tasks of the *Feature Model* concern are compatible with a modularized variability model. The building process of the feature model is transparent to the configuration editor presenting an instance of it.

The solution for feature model modularity, I want to present, is published by Bagheri *et al.* [Bagheri et al., 2011]. The authors want to enable that multiple teams of experts in their domain can develop one application simultaneously, using only the feature model belonging to their domain. To make that possible the overall feature model of an application needs to be split up into multiple small fractions of it. Consequently, a merge of two or more feature models must be possible too. There can be overlapping parts of the feature model in these subsets of features. Each part of the entire feature model is called a *feature model module*. As Bagheri *et al.* say, dependencies between two features in different feature model modules are captured by *module bridges*. These match the feature constraints in a feature model defined using *FeatureIDE*. On one hand, these are defined for every feature model module, on the other hand, they are also part of the *sentinel*. The feature model sentinel is a collection of module bridges, that are used to define how the feature models fragments are composed overall. The sentinel references the feature model modules to merge. The composition is executed in respect to the module bridges in the sentinel and the feature model modules. Finally, Bagheri *et al.* also describe how the validation of feature models can be automated using this approach.

Bagheri's solution already has many similarities to the feature model development and management using *FeatureIDE*. The module bridges, for example, could easily be implemented using *FeatureIDEs* model constraints. Therefore, I conclude that his approach can be combined with the framework. In the FRaMED 2 specific case, every module would contain three artifacts. Firstly, there is the feature model fragment, a modules specific part of the overall feature model. Besides that, a developer might need to define a part of the sentinel's definition. Lastly, every module has to contain a file defining the standard configuration of its own feature model part. At the start of the GEPL, the building process of the used feature model would be executed. The sentinel is generated at first. To do that all files that define a part of the sentinel are searched, for example by a specific file extension. These files offer references to the feature model modules to merge together. By putting all these references into the specifically formed module bridge of the sentinel, a feature model containing all features in the feature model modules can be generated. Finally, the overall standard configuration can be calculated by merging all standard configurations for the small split up feature models together. Rules, which definition dominate another one, can be implemented. The generated configuration can be checked for validity and alternative domination rules can be applied if needed, creating different standard configurations.

Abstract Syntax Model

To allow components extending the abstract syntax model, a delta modeling approach is useful. In it, delta modules define changes to a model when applied. Consequently, it possible to encapsulate the abstract syntax model fragments in such delta modules. By placing them in different components, each of those can define their own parts of the abstract syntax model, using operations that add, or modify metamodel elements. Now when starting the GEPL, in all components which are included in the GEPL, a composing artifact searches for delta modules and executes them on the core of the abstract syntax model. This can happen at startup of the GEPL as the metamodel of the abstract syntax model is not depending on a feature configuration and the same for all editor diagrams, which I consider as static. This does not violate the GEPL's property of being dynamically configurable, because the composition is not feature-dependent.

For the FRaMED 2 specific solution, I propose the usage of *DeltaEcore* [Seidl et al., 2014]. It enables delta modeling for models in an EMF [Steinberg et al., 2009] representation. Beside the fixed core of the IORM and the delta modules, defined in a language provided by *DeltaEcore*, it can be required to define operations only possible on the IORM. Finally, as the delta modules indicate their dependencies between them with the help of the `requires` keyword, a dependency graph on the level of components can be derived. An example of a *DeltaEcore* rule can be seen in *Listing* 6.1. Assuming all this is provided, the process of composing the metamodel of IORM can be executed the already described way. An artifact, triggered by an *Eclipse* application life cycle hook, searches for all delta modules to include via file extension and calculates the dependency graph at the start of the GEPL. The calculation of the dependency graph is a simple task, realizable by analyzing all explicitly defined dependencies in the delta modules. The dependency graph can detect cycle and unresolvable dependencies. If no problems were found it is used to calculate the order delta modules are executed on the IORM core, always ensuring a dependent module is applied after the one it depends on. In the case of found problematic dependencies, the set of feature modules can be automatically fixed, removing certain ones, which can be calculated based on the dependency graph too.

```
feature delta "genericFeature" {
        requires iorm.ecore, requiredFeature.decore, referencedFeature.decore
        EClass genericFeatureEClass = new EClass(name : "genericFeature");
        addEClass(genericFeatureEClass, <iorm>);
        addESuperType(<requiredFeature>, genericFeatureEClass);
        EReference referencedFeatureEReference = new EReference(...);
        addEReference(referencedFeatureEReference, genericFeatureEClass);
}
```

Figure 6.1: Template for a delta module implemented with *DeltaEcore*.

Target Metamodel

The design approach for GEPLs, presented in this thesis, addresses the feature configuration dependent composition of the target metamodel during runtime. However, the approach also stated that the target metamodel does not need to be composed in respect to a configuration in any case. For the FRaMED 2 specific case, I consider the second option as more useful. In the situation given by FRaMED 2, it makes more sense to build the metamodel of CROM only at the start of the GEPL. The exact same process as described for the abstract syntax model in this subsection can be used. It is worth noting that there already exist a CROM core and delta modules associated with the model as CROM is a family of RMLs. However, this reusability is the only new aspect to the modularization of it. Therefore, I will not further discuss the implementation, but why a configuration independent and static composition already fits the requirements in this case.

Following the description of the general solution for the dynamic composition of the target metamodel, there is always a significant problem creating a complex solution. Either there is only one metamodel for multiple diagrams with different configurations or multiple metamodels have to be managed, one for each diagram in the GE. The first solution leads to a lot of execution effort as the composition needs to be executed, completely starting from the core of the target metamodel, every time the model transformation is executed with a different configuration. Meanwhile, needing to create, compose, alter and delete single metamodel copies is needed for the second option, which is undesired. While these disadvantages of the dynamic and configuration dependent solutions cannot be ignored, the need for such a solution would justify implementing them nevertheless. When analyzing the need, it is clear that these ways to compose CROM would only increase the configuration dependence of artifacts as well as the conformity between the transformation and the target metamodel. The conformity means that the CROM only does include metamodel elements of which instances can be created by the transformation. However, this is not a priority of the design method and the case study. In contrast, the modularity is such a priority and can be achieved by a less complex feature independent and static implementation.

List of Figures

List of Tables

Abbreviations

CROM	*Compartment Role Object Model*
CRUD	Create, Read, Update, Delete
DSL	Domain Specific Language
EMF	*Eclipse Modeling Framework*
ER	Entity-Relationship Model
ETL	*Epsilon Transformation Language*
FOSD	Feature-oriented Software Development
FRaMED	*Full-fledged Role Modeling Editor*
GE	Graphical Editor
GEF	*Graphical Editing Framework*
GEPL	Graphical Editor Product Line
GMF	*Graphical Modeling Framework*
GMP	*Graphic Modeling Project*
GUI	Graphical User Interface
IORM	*Intermediate Object Role Model*
LPL	Language Product Line
MDD	Model Driven Development
PL	Product Line
PIM	Platform Independent Model
PSM	Platform Specific Model
RML	Role-based Modeling Language
SPL	Software Product Line
UML	*Unified Modeling Language*
XML	*Extensible Markup Language*

Bibliography

[Atkinson et al., 2002] Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wüst, J., and Zettel, J. (2002). *Component-based Product Line Engineering with UML*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Atkinson et al., 2000a] Atkinson, C., Bayer, J., Laitenberger, O., and Zettel, J. (2000a). Component-based software engineering: The kobra approach. In *ICSE Software Product Line Workshop*.

[Atkinson et al., 2000b] Atkinson, C., Bayer, J., and Muthig, D. (2000b). *Component-based Product Line Development: The KobrA Approach*, pages 289–309. Springer, Boston, MA, USA.

[Bachman, 1973] Bachman, C. W. (1973). The programmer as navigator. *Commun. ACM*, 16(11):635–658.

[Bachman and Daya, 1977] Bachman, C. W. and Daya, M. (1977). The role concept in data models. In *Proceedings of the Third International Conference on Very Large Data Bases - Volume 3*, VLDB '77, pages 464–476. VLDB Endowment.

[Bagheri et al., 2011] Bagheri, E., Ensan, F., Gasevic, D., and Boskovic, M. (2011). Modular Feature Models: Representation and Configuration. *Journal of Research and Practice in Information Technology*, 43:109–140.

[Batory et al., 2002] Batory, D., Johnson, C., MacDonald, B., and von Heeder, D. (2002). Achieving extensibility through product-lines and domain-specific languages: A case study. *ACM Trans. Softw. Eng. Methodol.*, 11(2):191–214.

[Boehm et al., 1978] Boehm, B., Brown, J., Kaspar, H., Lipow, M., MacLeod, and G.J., Merritt, M. (1978). *Characteristics of Software Quality*. Amsterdam.

[Bullinger et al., 2003] Bullinger, H.-J., Fähnrich, K.-P., and Meiren, T. (2003). Service engineering—methodical development of new service products. *International Journal of Production Economics*, 85(3):275 – 287. Structuring and Planning Operations.

[C. Kang et al., 1990] C. Kang, K., Cohen, S., A. Hess, J., Novak, W., and Spencer Peterson, A. (1990). Feature-Oriented Domain Analysis (FODA) feasibility study. Technical report, Carnegie Mellon University, Software Engineering Institute.

[Chen, 1976] Chen, P. P.-S. (1976). The entity-relationship model&mdash;toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36.

[Chung and do Prado Leite, 2009] Chung, L. and do Prado Leite, J. C. S. (2009). *On Non-Functional Requirements in Software Engineering*, pages 363–379. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Cuadrado and Molina, 2009] Cuadrado, J. S. and Molina, J. G. (2009). A model-based approach to families of embedded domain-specific languages. *IEEE Transactions on Software Engineering*, 35(6):825–840.

[Dahchour et al., 2002] Dahchour, M., Pirotte, A., and Zimányi, E. (2002). A generic role model for dynamic objects. In Pidduck, A. B., Ozsu, M. T., Mylopoulos, J., and Woo, C. C., editors, *Advanced Information Systems Engineering*, pages 643–658, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Evans et al., 2003] Evans, A., Maskeri, G., Sammut, P., and Willans, J. (2003). Building families of languages for model-driven system development. *Workshop in Software Model Engineering*.

[Fleurey et al., 2012] Fleurey, F., Haugen, Ø., Møller-Pedersen, B., Svendsen, A., and Zhang, X. (2012). Standardizing variability – challenges and solutions. In Ober, I. and Ober, I., editors, *SDL 2011: Integrating System and Software Modeling*, pages 233–246, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Font et al., 2015] Font, J., Arcega, L., Haugen, O., and Cetina, C. (2015). Building software product lines from conceptualized model patterns. In *Proceedings of the 19th International Conference on Software Product Line*, SPLC '15, pages 46–55, New York, NY, USA. ACM.

[Glinz, 2007] Glinz, M. (2007). On non-functional requirements. In *15th IEEE International Requirements Engineering Conference (RE 2007)*, pages 21–26.

[Grady and Caswell, 1987] Grady, R. B. and Caswell, D. L. (1987). *Software Metrics: Establishing a Company-Wide Program*.

[Hennicker and Klarl, 2014] Hennicker, R. and Klarl, A. (2014). *Foundations for Ensemble Modeling - The Helena Approach*, pages 359–381. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Herrmann, 2007] Herrmann, S. (2007). A precise model for contextual roles: The programming language objectteams/java. *Appl. Ontol.*, 2(2):181–207.

[Jäkel et al., 2014] Jäkel, T., Kühn, T., Voigt, H., and Lehner, W. (2014). Rsql - a query language for dynamic data types. In *Proceedings of the 18th International Database Engineering &#38; Applications Symposium*, IDEAS '14, pages 185–194, New York, NY, USA. ACM.

[Kästner and Apel, 2013] Kästner, C. and Apel, S. (2013). *Feature-Oriented Software Development*, pages 346–382. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Kästner et al., 2011] Kästner, C., Apel, S., and Ostermann, K. (2011). The Road to Feature Modularity? In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, SPLC '11, pages 5:1–5:8, New York, NY, USA. ACM.

[Kleppe, 2007] Kleppe, A. (2007). A language description is more than a metamodel. In *Fourth International Workshop on Software Language Engineeringl*.

[Kolovos et al., 2010] Kolovos, D., Rose, L., Paige, R., and Garcıa-Domınguez, A. (2010). *The Epsilon Book*.

[Kühn, 2017] Kühn, T. (2017). *A Family of Role-Based Languages*. PhD thesis, Technische Universität Dresden, Faculty for Computer Science, Software Technology Group, Dresden, Germany.

[Kühn et al., 2016] Kühn, T., Bierzynski, K., Richly, S., and Aßmann, U. (2016). Framed: Full-fledge role modeling editor (tool demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2016, pages 132–136, New York, NY, USA. ACM.

[Kühn and Cazzola, 2016] Kühn, T. and Cazzola, W. (2016). Apples and Oranges: Comparing Top-down and Bottom-up Language Product Lines. In *Proceedings of the 20th International Systems and Software Product Line Conference*, SPLC '16, pages 50–59, New York, NY, USA. ACM.

[Kühn et al., 2014] Kühn, T., Leuthäuser, M., Götz, S., Seidl, C., and Aßmann, U. (2014). A Metamodel Family for Role-Based Modeling and Programming Languages. *Software Language Engineering, Volume 8706 of Lecture Notes in Computer Science*, page 141–160.

[Leuthäuser and Aßmann, 2015] Leuthäuser, M. and Aßmann, U. (2015). Enabling view-based programming with scroll: Using roles and dynamic dispatch for etablishing view-based programming. In *Proceedings of the 2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*, MORSE/VAO '15, pages 25–33, New York, NY, USA. ACM.

[Malenfant et al., 1996] Malenfant, J., Jacques, M., and Demers, F. N. (1996). A tutorial on behavioral reflection and its implementation. In *Proceedings of the Reflection*, volume 96, pages 1–20.

[Mazo et al., 2015] Mazo, R., Muñoz Fernández, J. C., Rincón, L., Salinesi, C., and Tamura, G. (2015). Variamos: An extensible tool for engineering (dynamic) product lines. In *Proceedings of the 19th International Conference on Software Product Line*, SPLC '15, pages 374–379, New York, NY, USA. ACM.

[Minas and Viehstaedt, 1995] Minas, M. and Viehstaedt, G. (1995). DiaGen: a generator for diagram editors providing direct manipulation and execution of diagrams. In *Proceedings of Symposium on Visual Languages*, pages 203–210.

[Parviainen et al., 2009] Parviainen, P., Takalo, J., Teppola, S., and Tihinen, M. (2009). *Model-Driven Development.*

[Pohl et al., 2005] Pohl, K., Böckle, G., and van Der Linden, F. J. (2005). *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media.

[Roman, 1985] Roman, G. C. (1985). A taxonomy of current issues in requirements engineering. *Computer*, 18(4):14–23.

[Rumbaugh et al., 1999] Rumbaugh, J., Jacobson, I., and Booch, G., editors (1999). *The Unified Modeling Language Reference Manual*. Addison-Wesley Longman Ltd., Essex, UK, UK.

[Sánchez Cuadrado, 2012] Sánchez Cuadrado, J. (2012). *Towards a Family of Model Transformation Languages*, pages 176–191. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Seidl et al., 2014] Seidl, C., Schaefer, I., and Aßman, U. (2014). DeltaEcore - A Model-Based Delta Language Generation Framework. In *Lecture Notes in Informatics (LNI), Proceedings - Series of the Gesellschaft fur Informatik (GI)*, pages 81–96.

[Shaker et al., 2012] Shaker, P., Atlee, J. M., and Wang, S. (2012). A Feature-Oriented Requirements Modelling Language. In *2012 20th IEEE International Requirements Engineering Conference (RE)*, pages 151–160.

[Steimann, 2000] Steimann, F. (2000). On the representation of roles in object-oriented and conceptual modelling. *Data & Knowledge Engineering*, 35(1):83 –106.

[Steinberg et al., 2009] Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2009). *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition.

[Svendsen et al., 2010] Svendsen, A., Zhang, X., Lind-Tviberg, R., Fleurey, F., Haugen, O., Møller-Pedersen, B., and Olsen, G. K. (2010). Developing a software product line for train control: A case study of cvl. In *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond*, SPLC'10, pages 106–120, Berlin, Heidelberg. Springer-Verlag.

[Thüm et al., 2014] Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., and Leich, T. (2014). Featureide: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79:70 – 85. Experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010).

[Truică et al., 2013] Truică, C.-O., Boicea, A., and Trifan, I. (2013). CRUD Operations in MongoDB. In *International Conference on Advanced Computer Science and Electronics Information (ICACSEI 2013)*, pages 347–350.

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

[Vacchi et al., 2014] Vacchi, E., Cazzola, W., Combemale, B., and Acher, M. (2014). Automating variability model inference for component-based language implementations. In *Proceedings of the 18th International Software Product Line Conference - Volume 1*, SPLC '14, pages 167–176, New York, NY, USA. ACM.

[Vacchi et al., 2013] Vacchi, E., Cazzola, W., Pillay, S., and Combemale, B. (2013). Variability support in domain-specific language development. In Erwig, M., Paige, R. F., and Van Wyk, E., editors, *Software Language Engineering*, pages 76–95, Cham. Springer International Publishing.

[Viyović et al., 2014] Viyović, V., Maksimović, M., and Perisić, B. (2014). Sirius: A rapid development of DSM graphical editor. In *2014 18th International Conference on Intelligent Engineering Systems (INES)*, pages 233–238. IEEE.

[Voelter and Groher, 2007] Voelter, M. and Groher, I. (2007). Handling variability in model transformations and generators. In *In Companion to the Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2007*, New York, NY, USA. ACM.

[Zhu and Zhou, 2006] Zhu, H. and Zhou, M. (2006). Role-based collaboration and its kernel mechanisms. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 36(4):578–589.