UNIVERSITY OF OSLO
Department of Informatics

Conquering overlapping fragments in CVL

Master thesis

Anatoly Vasilevskiy

Spring 2013

Conquering overlapping fragments in CVL

Anatoly Vasilevskiy

30th April 2013

Abstract

Common Variability Language (CVL) is a generic variability modeling language. Fragment substitution is a fundamental CVL operation. The operation removes a set of model elements (placement fragment) and substitutes them with another set of elements (replacement fragment). Overlapping fragments represent a potential consistency challenge as the CVL execution may give unintended results for models where fragments intersect. Thus, we argue that there is a pragmatic need to handle overlapping placements. The need emerges, when several diagrams reference the same model. Hence, we can define a placement in one diagram while another placement in a different diagram references the same set of elements. It may indicate an error in the variability definition, but there are cases where we specify overlapping placements intentionally. In the thesis we carefully discuss such cases, 1) classify overlapping fragments, 2) find criteria to detect the different overlaps, and 3) suggest appropriate solutions via transformations.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Contribution and artifacts . . . 3
- 1.3 Thesis structure and Methods . . . 3

I Background **5**

2 Variability modeling realm **7**
- 2.1 Software Product Lines . . . 7
- 2.2 General-Purpose Languages . . . 8
- 2.3 Feature models . . . 9
- 2.4 Decision models . . . 10
- 2.5 Domain-Specific Languages . . . 11
- 2.6 Common Variability Language . . . 12
  - 2.6.1 Evolution . . . 12
  - 2.6.2 Overview . . . 13
  - 2.6.3 Basic concepts . . . 14
  - 2.6.4 MoSiS CVL tool . . . 16

3 Graph rewriting techniques **19**
- 3.1 Motivation and overview . . . 19
- 3.2 Algebraic approaches . . . 21
- 3.3 Tooling . . . 25
- 3.4 Graph transformations in the thesis . . . 27

II Contribution **29**

4 Problem statement and proposed solutions **31**
- 4.1 Substitution fragments in UML . . . 31
  - 4.1.1 Context . . . 31
  - 4.1.2 Neighboring placements problem . . . 32
- 4.2 Approach . . . 34
  - 4.2.1 Challenges . . . 34
  - 4.2.2 Node DSL and problem generalization . . . 36
  - 4.2.3 Graph rewriting to the neighboring placements problem . . . 38

4.3 Overlapping placements 42
4.3.1 Definitions and concepts 42
4.3.2 Kinds of overlaps 43
4.4 Adjacent placements 46
4.4.1 Overview 46
4.4.2 Solution 48
4.5 Kinds of crossing placements 50
4.5.1 Crossing placements 50
4.5.2 Crossing and adjacent placements 53
4.5.3 Contained placements 55
4.6 Summary: overlapping placements 56

5 Implementation: CVL fragment substitution engine 59
5.1 Design goals and requirements 59
5.2 Technologies 61
5.2.1 Eclipse Modeling Framework 61
5.2.2 Programming language 61
5.2.3 Maven 61
5.2.4 Spring 62
5.2.5 JUnit 63
5.3 Fragment substitution engine 63
5.3.1 Engine architecture 63
5.3.2 Fragment substitution module 65
5.3.3 Adjacent resolution module 67
5.3.4 Logging 71
5.3.5 Errors 72
5.3.6 Tests 73
5.3.7 Discussion 74

6 Experiments and validation 75
6.1 Neighboring placements problem 75
6.2 Evaluation 76
6.2.1 Strategies 76
6.2.2 Results 78

7 Related works 83
7.1 MoSiS CVL 83
7.2 Confluence of graph transformations 84
7.3 Conflicts in product line engineering 86
7.4 Feature-oriented and Delta-oriented programming 87
7.5 Aspect-oriented Programming 88

III Conclusion 89

8 Results and Future work 91
8.1 Results 91
8.2 Discussion 92

8.3 Future works 93

A Test strategies 1,2,3,4 - measurements 95

List of Figures

1.1 Different views of the car component model 2

2.1 Feature model of the car components 9
2.2 Variability model in the MoSiS and OMG CVL tools - tree representation 13
2.3 The Common Variability Language architecture 14
2.4 Basic CVL concepts 14
2.5 Boundary elements - metamodel 15
2.6 Variability model - tree representation 16
2.7 Specifying of the placement fragment - selection 17
2.8 Placement and replacement fragments 17
2.9 Binding boundary elements - binding editor 18

3.1 Graph transformation theory concepts 20
3.2 Pushout squares 22
3.3 The double-pushout approach (DPO) 23
3.4 The DPO example 23
3.5 The single-pushout approach (SPO) 24
3.6 The SPO example 24
3.7 Transformation rules in Henshin 26
3.8 Henshin Sequential Unit 27

4.1 The SafetyDrive composite structure 31
4.2 The SafetyDrive test case definition 32
4.3 The simplified class, composite structure and sequence diagrams 33
4.4 The instance diagram and simplified UML metamodel . . . 33
4.5 Flow to obtain a valid OMG CVL model 35
4.6 Node DSL - metamodel 36
4.7 The inwards/outwards references of the substitution fragment in UML 37
4.8 Simulation in the Node language 37
4.9 The inwards/outwards references of the simplified model . 37
4.10 Two placement fragments 38
4.11 Substitution fragments, bindings and the substitution result 38
4.12 The CVL substitution in terms of DPO 39
4.13 Two substitutions in Henshin 40
4.14 Two substitutions in Henshin - the wildcard approach 41

4.15 Expected result of two substitutions 41
4.16 Inner Placement (IP) and Outer Placement (OP) 43
4.17 Overlapping kinds 45
4.18 Adjacent placements and expected resolution 46
4.19 Actual result of the resolution 47
4.20 Variability model modifications 48
4.21 The crossing fragments 50
4.22 The crossing fragments and their unionizations 51
4.23 The crossing and adjacent placements, unionization and resolution 54
4.24 Contained placements 56

5.1 The fragment substitution matamodel 60
5.2 Maven dependency graphs (different scopes) 62
5.3 Substitution engine packages 63
5.4 The fragment substitution operation - class diagram 65
5.5 Adjacent resolution module - class diagram 67
5.6 Logging - class diagram 71
5.7 Thrown exceptions - class diagram 72

6.1 The neighboring problem, placements - validation 75
6.2 The neighboring problem, replacements - validation 76
6.3 The test fragments - basic sample 76
6.4 Test strategies 1,2 - data visualization 79
6.5 Test strategies 3,4 - data visualization 81

7.1 Substitutions in MoSiS CVL and OMG CVL 83
7.2 The neighboring problem - graph rewriting system 85
7.3 The neighboring problem - joinability of the critical pairs . . 85

List of Tables

2.1 Decision model in the tabular notation 10

3.1 Available graph-based tools 25

4.1 The simple 2x2 table 44
4.2 The overlapping kinds - overview 57

5.1 The packages and their descriptions 64
5.2 The engine exceptions 72
5.3 The test plans 73

6.1 Empirical order of growth - substitution TS 2 79
6.2 Empirical order of growth - finder TS 4 81

A.1 Mean values and standard deviations - part 1 95
A.2 Mean values and standard deviations - part 2 97
A.3 Mean values and standard deviations - part 3 99
A.4 Mean values and standard deviations - part 4 101

List of Definitions

2.1 Definition (Software platform) 7
2.2 Definition (Mass customization) 7
2.3 Definition (Variability) 7
2.4 Definition (Variability modeling) 7
2.5 Definition (General-Purpose Language (GPL)) 8
2.6 Definition (Feature model (FM)) 9
2.7 Definition (Decision model (DM)) 10
2.8 Definition (Domain-specific language (DSL)) 11
2.9 Definition (Base model) 13
2.10 Definition (Variability model) 13
2.11 Definition (Resolution model) 13
2.12 Definition (Placement/Replacement fragment) 14
2.13 Definition (Boundaries) 15
2.14 Definition (Fragment substitution) 15
2.15 Definition (Binding) 16
2.16 Definition (Binding process) 16

3.1 Definition (Graph) 19
3.2 Definition (Directed graph) 19
3.3 Definition (Type graph) 20
3.4 Definition (Instance graph) 20
3.5 Definition (Rule) 21
3.6 Definition (Pre-condition) 21
3.7 Definition (Post-condition) 21
3.8 Definition (Algebraic approach) 22
3.9 Definition (Morphism) 22
3.10 Definition (Pushout) 22
3.11 Definition (Commutative diagram) 22

4.1 Definition (Placement Element internal (PEint)) 42
4.2 Definition (Placement Element external (PEext)) 42
4.3 Definition (Inner Placement (IP)) 43
4.4 Definition (Outer Placement (OP)) 43
4.5 Definition (Independent placements) 43
4.6 Definition (Parallel independent placements) 44
4.7 Definition (Adjacent relation) 45
4.8 Definition (Crossing relation) 45
4.9 Definition (Contained relation) 46

4.10 Definition (Adjacent boundary) 49
4.11 Definition (Adjacent pair) 49

7.1 Definition (Parallel independence) 84

Preface

I would like to thank University of Oslo and SINTEF for giving me an opportunity to conduct my studies in this wonderful country, Norway. I am very grateful to my supervisor Øystein Haugen for the guidelines over the recent years, his constructive, helpful feedbacks and ideas. I really appreciate the expertise of Roy Grønmo in the graph rewriting theory, Brice Morin in EMF and their practical advices. Finally, I express my enormous gratitude to my lovely sister Maria Vasilevskaya for making my move to Oslo possible. I am sincerely thankful to my parents Viktor and Elena as well as others for their support throughout the years.

Chapter 1

Introduction

1.1 Motivation

The adoption of an assembly line was a breakthrough in the industrial engineering at the end of the 19th century and beginning of the 20th century. Assembly lines have gained the most importance and played the major role in production of cheap and reliable products. A manufacture was able shortly to produce a wide variety of products building them on assembly lines. The term product line appeared later describing this product variety. A product is built stepwise on an assembly line. A new functionality is added as a conveyor moves the product from one stage to another. In the end of the 20th century, one may find first attempts to use the product line principles in the software domain. In other words, we can assemble numerous software products from different components specifying a product line. The variability modeling term has been coined to describe the product line definition in the software realm. Common Variability Language (CVL) is a mean to variability modeling.

A placement and replacement are artifacts to define variability on a base model in CVL. Defining variability on different levels or in different views of the base model can lead to possible overlaps between variability artifacts. Some of these overlaps are desirable and done on purpose, others may indicate that a variability model is not consistent and requires additional modifications. Thus, management of overlapping fragments is an important part of the CVL execution semantics. The management includes detection of overlapping fragments, their classification and mitigation.

Modern modeling languages may have quite large metamodels, e.g. UML [41]. An instance of the metamodel represents a model created for a particular task. A model in UML can have a complex structure even for relatively small tasks; therefore, such models are difficult to analyze and develop. Contemporary modeling tools address this challenge using different diagrams to represent a model and reflect various aspects of a system. Such diagrams may reference the same model and even some entities in different diagrams can reference the same model elements.

In Figure 1.1, we can find two simplified class diagrams ((a) and (b)) with the generalized car components and a variant of their relations. In the instance diagram (Figure 1.1 (c)) of the UML metamodel [41], we introduce all components presented in the diagrams (Figure 1.1 (a) and (b)) of the car model together with their associations. We use arrows to show associations between classes in the instance diagram (c) just for the sake of simplicity, but the UML metamodel is somewhat more complicated than that. There are two core general elements in a car, i.e. a Motor, Gearbox (Transmission). The controls class diagram (on the right (b)) shows HandControls, FootControls components w.r.t. a Gearbox. One

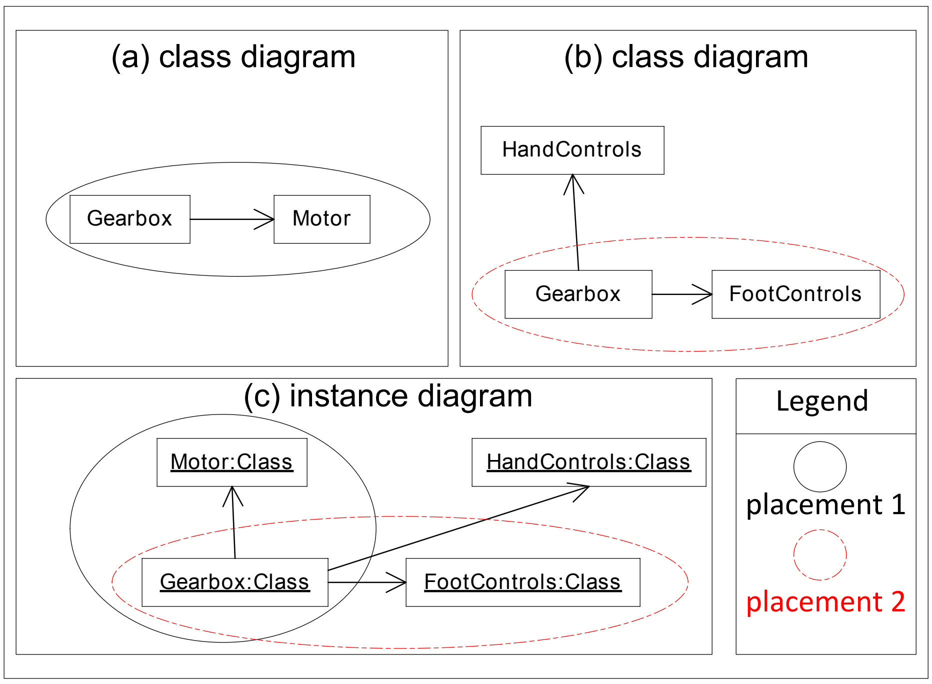


Figure 1.1: Different views of the car component model

may want to assemble a car with a certain functionality; thus, we replace the generalized gearbox and motor with something more concrete, for example, an automatic transmission and compatible motor. Hence, we define a placement in (a). Meanwhile, in the class diagram (b) we have to take care of the FootControls block, which depends on a particular gearbox kind. Thus, we select both and want to replace them with the automatic gearbox and two pedal foot control block. Since we have made the selections independently, the model (c) depicts an overlap on the gearbox component. The result of the substitution is well understood, i.e. we want the automatic gearbox with the corresponding motor and compatible two pedals control block. However, the current CVL implementation from SINTEF [22] fails to accomplish the task correctly due to the overlapping fragments in the model. The example shows a pragmatic need of detecting and tackling such situations.

1.2 Contribution and artifacts

The major artifacts and contribution of the thesis are:

- deduction and classification of the cases where the current CVL implementation does not succeed to derive a correct product;
- criteria to detect different kinds of overlapping fragments;
- mitigations for each of the classified cases;
- implementation of the substitution engine with the adjacent resolution functionality;
- validation of the implemented solution using an UML example together with other artificially created samples and its evaluation.

1.3 Thesis structure and Methods

The rest of the thesis is organized as follows. Chapters 2 and 3 give some background information. Chapter 2 covers the variability modeling realm describing basic techniques and principals. We pay special attention to Common Variability Language (CVL) highlighting its basic features and previous achievements. In Chapter 3, we introduce the graph transformation theory, graph based tools and put the graph rewriting theory into the context of our research. We discuss elaborately the problem from Section 1.1 (Motivation) and define solutions in Chapter 4. Further, we introduce an approach to the problem and describe whether the issue is solvable directly applying the graph transformation theory. Chapter 5 presents design decisions on the implementation of the substitution engine. We validate and evaluate the implemented engine in Chapter 6. Subsequently, Chapter 7 discusses some works related to our contributions. Finally, we conclude and explain future works in Chapter 8. Appendix A contains measurements made for the evaluation section.

Part I

Background

Chapter 2

Variability modeling realm

2.1 Software Product Lines

Software Product Lines (SPL) encompass a *systematic combination of mass customization and the use of a common platform for the development of software-intensive systems and software products* [45]. Where the term common platform is largely associated with a software platform. Pohl, Böckle and van der Linden in [45] give the precise definitions of the software platform and mass customization terms.

Definition 2.1 (Software platform) ***Software platform*** *is a set of software subsystems and interfaces that form a common structure from which a set of derivative products can be efficiently developed and produced [45].*

Definition 2.2 (Mass customization) ***Mass customization*** *is the large-scale production of goods tailored to individual customers' needs [45].*

Software becomes a major component of any computer based system. The amount of software for systems is growing and the amount of various software components for a particular system is growing even faster. The range of different possibilities for a particular system can be expressed through variability modeling. The variability term is not new by itself and basically means *the quality, state, or degree of being variable or changeable* [58] without any application to computer science. Variability defines mass customization with respect to SPL.

Definition 2.3 (Variability) ***Variability*** *is a flexibility that is a precondition for mass customization [45].*

We should outline that variability modeling in computer science closely associates itself with SPL, where

Definition 2.4 (Variability modeling) ***Variability modeling*** *attempts to describe more than one variant of a system for a (software) product line.*

The SPL term can refer to methods, tools and techniques at the same time. In other words, we can define SPL as a set of references to methods, tools and techniques for creating and maintaining a collection of similar software systems from a shared set of software assets [23].

SPL has been actively developing for the recent years. Generally, software is more flexible than hardware, meaning software allows introducing a new functionality that could not be easily achieved without software components. Hence, variability can be implemented in software more efficiently than in pure hardware. Thus, presently a strong need for adopting product line engineering can be observed in the software domain, especially, when size and complexity of systems exceed limits of what is feasible with traditional approaches [30].

A goal of any product line is a derivation of a product based on a wide range of components. This range implicitly shows a potential variety of the product line. The potential variety can be expressed over commonalities and variabilities of the product line. Therefore, lot's of research papers are dedicated to techniques, methods, tools which allow enabling of variability modeling in the most efficient manner. There is a set of techniques that can be exploited to define a variability in SPL [7].

2.2 General-Purpose Languages

Definition 2.5 (General-Purpose Language (GPL)) ***General-Purpose Language*** *(GPL) is a programing or modeling language designed for usage in a wide range of application domains.*

General-Purpose Languages (GPLs) provide means for variability modeling directly. We can highlight three main approaches to deal with variability in GPLs. However, the devision is quite synthetic because each of these approaches exploits core principles of GPLs, e.g. templates, virtual functions etc, but in different proportions. Nevertheless, each has its own methodology to apply the means of GPLs to achieve goals of variability modeling. Hence, the separation justifies itself:

- standard mechanism of GPL,
- annotation of GPLs,
- generative programming.

Standard mechanisms of GPL support variability via the principles of specialization, overriding and redefinition which can be expressed by virtual types/functions/classes, templates, inheritance, stereotypes. Bayer et al. [7] show how variability can be supported over the UML 2.0 means.

Annotation of GPLs is logically a step further towards improvement of expressing variability in GPLs. Generally, GPLs have not been developed to support and capture all aspects of variability, and the annotation approach is a way to incorporate variability having a certain separation of concerns through aspects.

Generative Programming (GP) is a more advanced paradigm that allows exploiting built-in GPLs means together with a new concept called

Active Libraries to enable efficient describing of variability [14]. Active Libraries are programming libraries, which take part in code generation actively. They provide abstractions, components, algorithms etc and can change, optimize them depending on a task.

Even though using of GPL is generally a pretty straightforward way to deal with variability, since we do not need to use other means or any external tools. It makes definition of variability available only to experts whom together with knowledge of the variability domain have technical understanding and skills to implement variability in a particular GPL. Using GPLs can improve the definition of variability for feature models, but we regress to low-level programming [60]. GPLs can incorporate variability mechanism into DSLs as well, but it causes the amalgamation of DSLs that leads to cluttering of a model in DSLs with variability specifications [21].

2.3 Feature models

A main concept of any software product line is a feature artifact. One may find a first usage of the feature term in the work of Kang et al. [31] as a mean to describe commonalities, and variabilities of software systems. We can define feature models as follows.

Definition 2.6 (Feature model (FM)) ***Feature model*** *(FM) represents the information of all possible products of a software product line in terms of features and relationships among them [8].*

Figure 2.1 shows the feature model that configures some car components. Feature modeling is an attempt to make the variability definition process

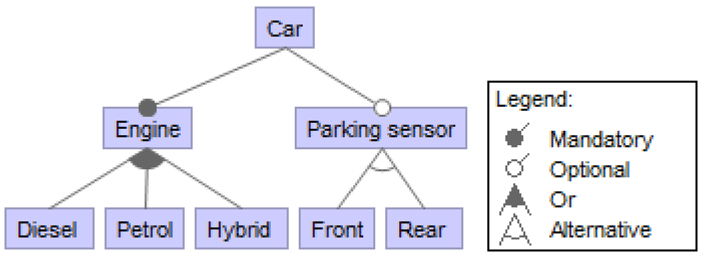


Figure 2.1: Feature model of the car components

available to stakeholders, i.e. experts in the variability domain. This approach provides abstractions of product components over their implementations. These abstractions form sets of configurations which encompasses variability of a product line. Feature models realize these configurations. A product derivation is achieved through the resolution of a feature model. A feature model looks as a tree, where nodes and leafs represent features, i.e. abstractions over product components. A major property of a feature model is finiteness. It means that possible derivations of a product is bounded by a corresponding feature model. In other words, all components/abstractions and derivation paths should be defined explicitly in a feature model. It significantly simplifies a validation process and testing of possible configurations. For example, Czarnecki and Wasowski [16] show how feature models can exploit existing logic-based tools, such as SAT solvers and Binary-Decision Diagram (BDD) libraries to validate their configurations.

There are several variability modeling approaches, which exploit the feature concept and recognize themselves as feature modeling, for example, the cardinality-based feature modeling approach by Czarnecki et al. [16]. Pohl et al. [45] describe the Orthogonal Variability Model (OVM) methodology, that prevents amalgamation of domain (product) artifacts with variability concepts. Common Variability Language (CVL) by Haugen et al. [12, 21, 25] makes use of FM having similar concepts and OVM defining variability orthogonally.

2.4 Decision models

Definition 2.7 (Decision model (DM)) ***Decision model*** *(DM) is a set of decisions that are adequate to distinguish among the members of an application engineering product family and to guide adaptation of application engineering work products [57].*

Decision models together with feature models have gained the most importance among existing approaches. But this approach to variability represents rather large class of developed approaches and exists nearly as long as feature-oriented modeling [48]. Table 2.1 shows a car decision model in the tabular notation. Czarnecki et al. [15] claim that the main

Table 2.1: Decision model in the tabular notation

decision name	description	type	range	cardinality/ constraint	visible/ re-leavent if
eng_-type	Which engine do you prefer?	Enum	Diesel | Petrol | Hybrid	1:1	
prk	Do you need parktronic?	Boolean	true | false		
prk_-type	Where do you need parktronic?	Enum	Back | Rear	1:2	prk= true

difference among DM and FM techniques is that DM approach exclusively focuses on variability modeling, while FM emphasizes both commonality and variability modeling. Existing DM techniques are mainly influenced by the Synthesis method [15, 57]. In decision modeling questions represent decisions. We need to answer all questions to derive a new product. In other words the derivation of a product is performed by making decisions at certain points. There are several main approaches in the realm of decision modeling [48]:

- Synthesis,
- Schmid and John,
- KobrA,
- DOPLER,
- PLUM,
- VManage.

The approaches differ from each other by the structure of a decision model; a set of choices possible for a decision, namely data type; dependency management among different decisions; relationships to artifacts; product derivation possibilities [48]. Each method defines a basic model and all these approaches share some commonalities with respect to the basic model structure.

2.5 Domain-Specific Languages

Definition 2.8 (Domain-specific language (DSL)) ***Domain-specific language*** *(DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain. [59]*

A domain-specific language operates with concepts from a domain. Experienced developers usually implement these concepts. Experts use these concepts to solve a problem acting as software developers during this process. Hence, DSL is a mean that intends to reduce conceptual gap between software developers and domain experts. It allows involving domain experts in the software development process that generally leads to significant improvements of the development process itself. DSL is a part of the infrastructure that together with other parts, i.e a domain framework, a domain-specific code generator allow creating models. Applications in DSL are automatically generated running on the top of the domain framework. The generated code needs not be edited or even looked at [32]. In short, the main benefit of applying the DSL approach is that DSLs can define the path all the way to realization. It means that we can use DSLs not only for the process of sketching, but as normal GPLs [7].

Using DSLs to describe the variability of a system can benefit the process of the SPL engineering [24]. A DSL is still a programming language regardless of abstractions it provides. It may incorporate the core GPL mechanisms, i.e. attributing, referencing, recursion, loops - which allow describing unbounded nature of variability. It should be admitted that this flexibility of DSLs opens the possibility to implement variability in many different ways. It leads to the complexity of supporting different tools and incorporating variability mechanisms in SPL. For example, Haugen et al. [21] outline two different ways of describing variability in systems, i.e. the first is a pure feature model, that does not depend on the implementation or design of systems, and the second one is to describe the variability model that relates to a base model. Haugen et al. [21] defines two different mechanisms with respect to the second possibility.

- Annotation of a base model by adding specific constructions to support variability. It causes an amalgamation of DSLs that leads to cluttering of the base model with variability specifications.
- Making separate, orthogonal models which are applied to the base model. It can be achieved by the definition of a separate language. However, variability is not clearly marked at this point.

Any DSLs can also been seen in general as a language for variability modeling, because it provides functionality to describe every system in a corresponding domain. Hence, we can include this option to the list above. Haugen et al. [21] draw our attention to the fact that the separate-language approach is preferable, since we do not overweight DSLs with concepts, which originate in another domain. DSLs are supposed to be lite, and operate with domain concepts only, otherwise they become overwhelmed with unnecessary details. However, there are some studies, which consciously choose the amalgamation approach to deal with variability, for example, the paper describes how variability can be woven into a metamodel [37].

2.6 Common Variability Language

2.6.1 Evolution

Common-Variability Language (CVL) is a domain-specific language for variability modeling. We can find basic principles, which laid the foundation for CVL in [7] by Bayer et al. in 2006, even though the term itself would be coined in later works. Further, Oldevik and Haugen [38] developed the variability approach to SPLs via transformations in MOFScript [40]. Finally, one may find the term CVL in the paper [21] of Haugen et al. together with the elaborated example of applying CVL to UML and Train Control Language (TCL) by Svendsen et al. [54].

SINTEF [22] has developed the first version of CVL within the MoSiS project ITEA 2 - ip06035 part of the Eureka framework [55]. The tool got its name after the MoSiS project - MoSiS CVL. MoSiS CVL has been successfully applied within several European projects, such as MoSiS and CESAR. Currently, a new CVL tool - OMG CVL is being developed within the VARIES project and the specification under a standardization process

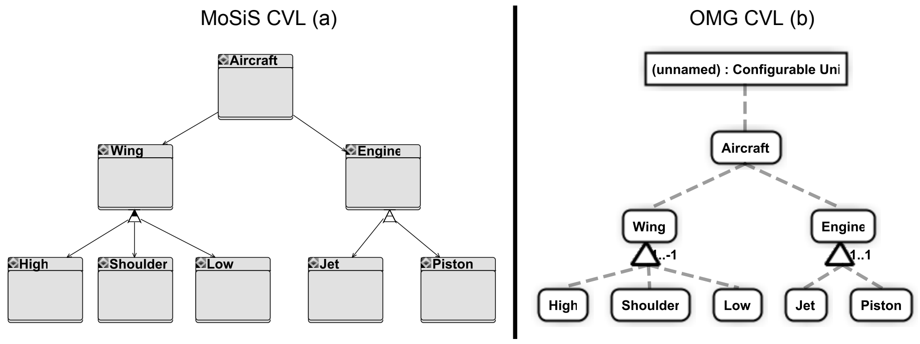


Figure 2.2: Variability model in the MoSiS and OMG CVL tools - tree representation

[12]. The metamotel of OMG CVL has some changes with respect to MoSiS CVL. For example, one may find the Configuration Unit [12] concept reflecting possible modularity, which may occur during a variability modeling. The OMG CVL metamodel enhances also means to references elements of a model on which variability is defined (a base model) etc. However, main concepts and principals remain unchanged, e.g. representing the variability model as a tree structure, see Figure 2.2.

2.6.2 Overview

CVL is specifically devoted to the variability domain, the language is formal and executable. There are several studies, which show how variability can be expressed in CVL. For instance, the authors in [2, 56] describe an example, where TCL is used to exemplify methodology of variability incorporation through CVL. CVL implements consolidated metamodel [12], and exploits three types of models: a base model, a variability model (VAR-model) and resolution model (RES-model). The CVL specification [12] gives the following definitions for the mentioned models.

Definition 2.9 (Base model) ***Base model*** *is a model on which variability is defined using CVL. The base model is not part of CVL and can be an instance of any metamodel defined via MOF [12].*

Definition 2.10 (Variability model) ***Variability model*** *is a collection of variation points, VSpecs, and constraints used to specify variability over a base model [12].*

Definition 2.11 (Resolution model) ***Resolution model*** *is a collection of VSpec resolutions resolving the VSpecs of a variability model [12].*

A CVL execution consumes these three models and results in a set of resolved models. The process is shown schematically in Figure 2.3 on the next page [12]. It should be admitted that this schema of the resolution process is DSL independent [56]. CVL applies one-way associations to a

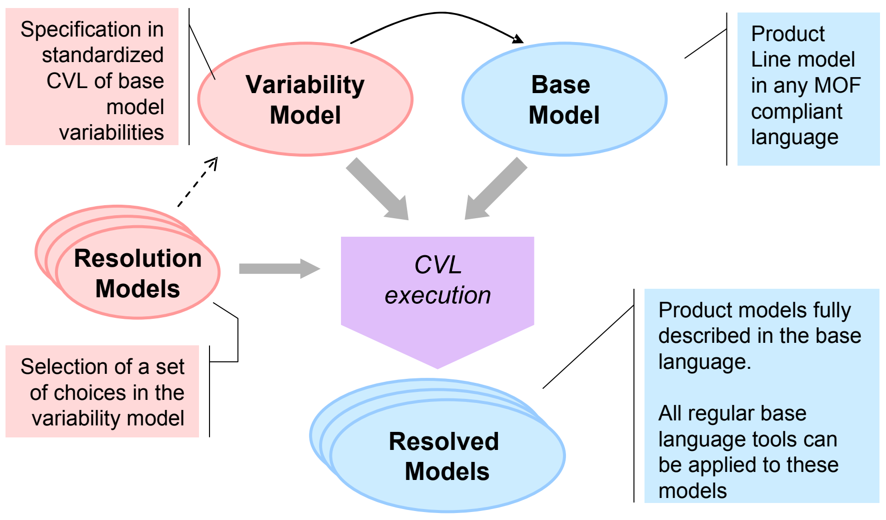

Specification in standardized CVL of base model variabilities
Variability Model
Base Model
Product Line model in any MOF compliant language
Resolution Models
CVL execution
Selection of a set of choices in the variability model
Product models fully described in the base language.
All regular base language tools can be applied to these models
Resolved Models

Figure 2.3: The Common Variability Language architecture

base model. And since CVL is separate and independent, no annotations or variability concepts are added into the base model or base language [2].

2.6.3 Basic concepts

The CVL specification [12] gives explicit definitions of all concepts involved in CVL, while we will try to focus on the basic principals and definitions. The fundamental concepts of CVL are placement and replacement fragments.

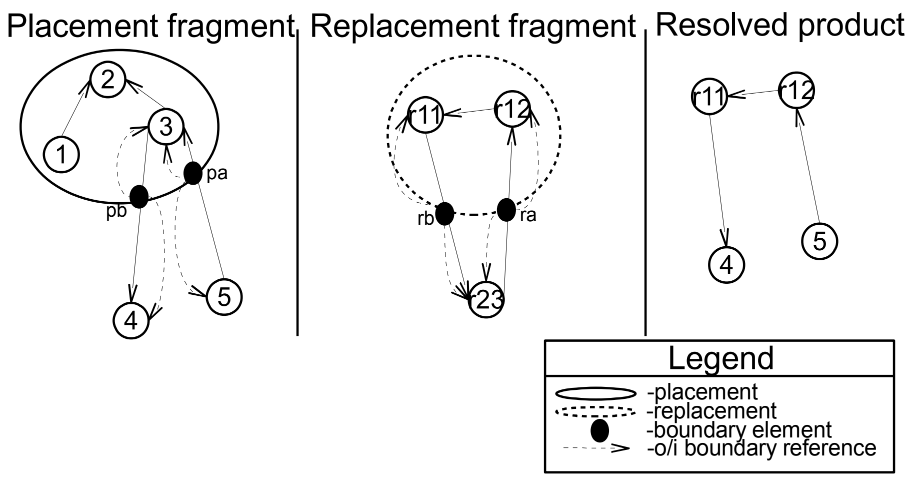


Figure 2.4: Basic CVL concepts

Definition 2.12 (Placement/Replacement fragment) ***Placement fragment*** *is a set of elements forming a conceptual 'hole' in a base model, which should be replaced by another fragment, namely* ***replacement fragment****.*

Figure 2.4 exemplifies a pair of the placement and replacement fragments. The placement in the figure is shown by the solid oval, while the dashed oval outlines the replacement. The elements inside ovals belong to the placement and replacement respectively. We should point out that placement and replacement fragments in CVL are defined via so called boundary elements.

Definition 2.13 (Boundaries) ***Boundaries*** *are elements, which fully define all references going in and out of a placement/replacement fragment [12].*

A variability expert may define a placement/replacement through a simple selection procedure on a model creating a set of modified elements. These elements inside placement/replacement may point to entities outside the given selection, these references are cut off creating boundaries. Figure 2.4 on the preceding page shows four boundary elements, i.e. two for the placement fragment (*pa*, *pb*) and two for the replacement fragment (*ra*, *rb*). The metamodel in Figure 2.5 depicts two kinds of placement boundaries, i.e. *ToPlacement*, *FromPlacement* and two kinds of replacement boundaries, i.e. *ToReplacement*, *FromReplacement*. An instance of *ObjectHandle* is a proxy

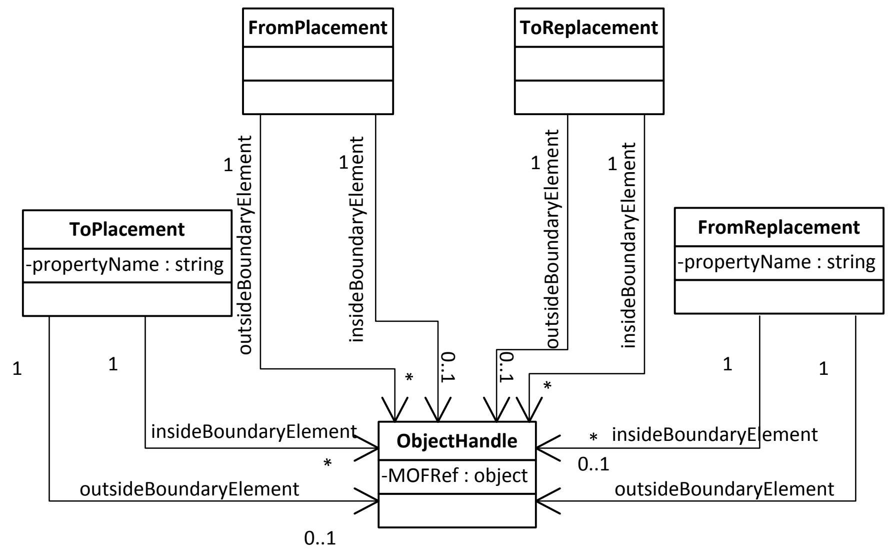


Figure 2.5: Boundary elements - metamodel

object that just references an object in a model. Each boundary element has two kinds of associations, which denotes references pointing inside and outside a placement/replacement, see Figure 2.5. We may find in Figure 2.4 on the facing page that *pa* is of the kind *ToPlacement* references the element *3* (inner neighboring element) and element *5* (outer neighboring element), *pb* is of the kind *FromPlacement* with references to *3* and *4*. The replacement fragment has *ra* of the kind *ToReplacement* and *rb* of the kind *FromReplacement*.

A product derivation in CVL is achieved by means of fragment substitution operations. Where the fragment substitution operation, we can define as follows.

Definition 2.14 (Fragment substitution) ***Fragment substitution*** *is an operation that substitutes a single object or model fragment (placement fragment) for another (replacement fragment).*

On the most right of Figure 2.4 on page 14 we can find the result of the placement substitution onto the replacement. As we can observe the operation removes the contents of the placement and copy the contents of the replacement onto the hole made by the removal of the placement elements.

The binding process is another key operation of the variability modeling in CVL.

Definition 2.15 (Binding) ***Binding*** *is a link that defines how a placement boundary element relates to another replacement boundary element.*

Definition 2.16 (Binding process) ***Binding process*** *is a process that specifies all bindings between a placement and corresponding replacement; thereby, it defines a substitution process.*

For our particular example in Figure 2.4 on page 14 by the binding of *pa* to *ra* and *pb* to *rb*, we specify that the element *5* should reference element *r12* and *r11* should point to *4* respectively in the derived product on the most right.

2.6.4 MoSiS CVL tool

The MoSiS CVL tool implements the first version of the CVL language and contains all basic concepts from the OMG CVL specification. An engineer defines a variability model in MoSiS CVL as a tree structure. Figure 2.6 depicts a variability model based on the example from Figure 2.4 on page 14. The figure shows a simple tree structure which may be seen

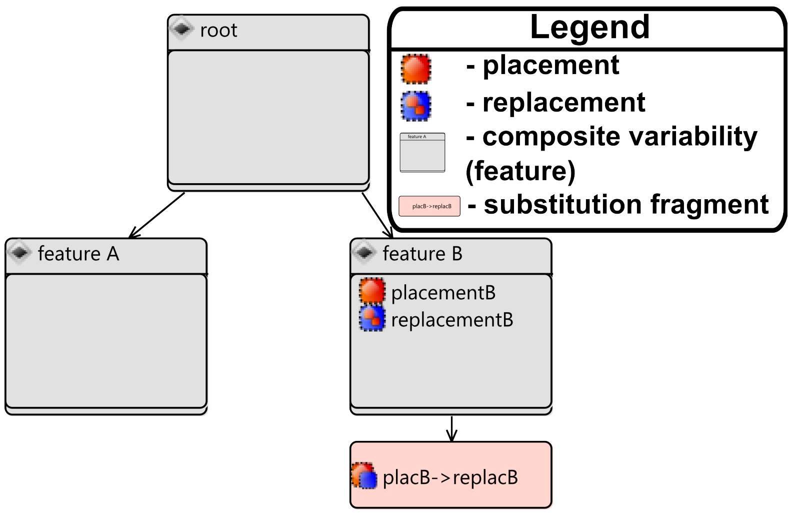


Figure 2.6: Variability model - tree representation

as a feature model, each composite variability element is a feature. A composite variability may define placement/replacement fragments and specify the substitution operation via the substitution fragment element. In the example from Figure 2.6 on the preceding page, the composite variability *feature B* defines the placement *placementB*, the replacement *replacementB* and the substitution fragment *placB->replacB*. In Figure 2.6 on the facing page, the substitution fragment specifies the substitution of the elements *1*, *2*, *3* (*placementB*) onto *r11*, *r21* (*replacementB*). We schematically show this process in Figure 2.4 on page 14. A variability engineer sets placement and replacement fragments via a simple selection procedure in a model. Figure 2.7 exemplifies the specification of the placement

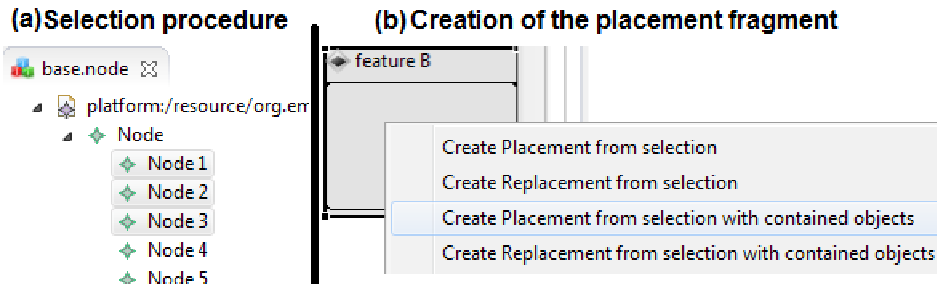

(a)Selection procedure
base.node
platform:/resource/org.em
Node
Node 1
Node 2
Node 3
Node 4
Node 5
(b)Creation of the placement fragment
feature B
Create Placement from selection
Create Replacement from selection
Create Placement from selection with contained objects
Create Replacement from selection with contained objects

Figure 2.7: Specifying of the placement fragment - selection

fragment through the selection procedure in the MoSiS CVL tool. We may see in Figure 2.7 (a) that one selects the elements (*1*, *2*, *3*) to substitute. These elements constitute the placement fragment which we create via a corresponding menu item, see Figure 2.7 (b). The MoSiS CVL tool automatically calculates all affected elements and creates boundaries on the links which reference elements outside/inside a placement/replacement. Further, we can observe the result of such calculation via highlighting of the icons which denote placement/replacement/substitution fragments. In Figure 2.8, one selects the icon *placementB* in the variability model and the tool shows the elements which comprise the placement fragment in the base model together with the affected elements (elements which do not belong to a placement, but have references pointing to elements inside/outside the placement). The final step is to specify how elements of

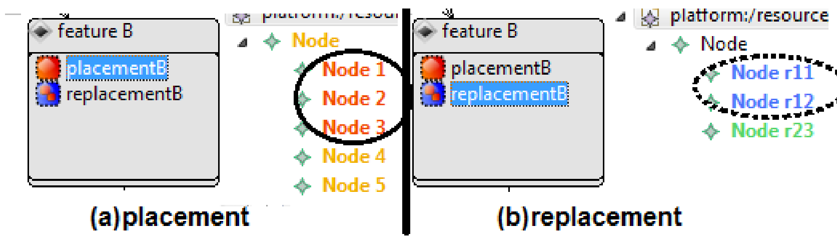
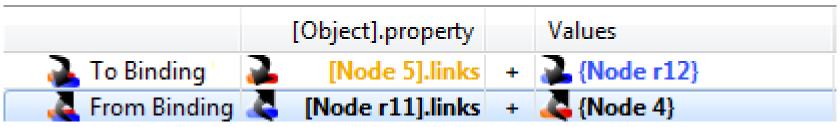


Figure 2.8: Placement and replacement fragments

the replacement fragment should substitute the elements of the placement yielding the resolved product in Figure 2.4 on page 14. The MoSiS CVL tool introduces the binding editor, which lists all calculated boundaries and provides means to define how a replacement fragment should be woven into a base model. Figure 2.9 shows the binding editor of the MoSiS CVL



	[Object].property		Values
To Binding	[Node 5].links	+	{Node r12}
From Binding	[Node r11].links	+	{Node 4}

Figure 2.9: Binding boundary elements - binding editor

tool. The boundaries of the placement (*pb*, *pa*) and replacement (*rb*, *ra*) reference the placement (*3*, *4*, *5*) and replacement (*r11*, *r12*, *r23*) elements respectively. One defines how these elements should reference each other in a final product. The binding process encompasses a specification of such references between elements of a placement and replacement. In Figure 2.9, we instruct the tool that *5* should reference *r12* and *r11* should point to *4* in the final model. Thus, a resolution process, which is driven by the given bindings, yields the resolved product in Figure 2.4 on page 14.

Chapter 3

Graph rewriting techniques

3.1 Motivation and overview

Variability modeling defines a product derivation from some basic structure. A product evolves through a process of modifications added to the basic structure, where a structure is a set of linked elements. Graphs have very basic and general definition; thus, we may always interpret any structure, e.g. a model or diagram as a graph.

Definition 3.1 (Graph) ***Graph*** *is a set of vertices and links between them which represent edges.*

Definition 3.2 (Directed graph) ***Directed graph*** *is a graph, where edges have direction.*

Different engineering fields use the graph representation as a powerful and convenient notation of formalization, studying and implementation of their task. Particularly, in object-orientation a runtime environment of an executed program is a set of communicating objects; therefore, an environment may be represented as a graph. In fact, any data structure is a graph. Changes of a model, data structure or runtime environment conforms to the corresponding changes in graphs. Graph transformation techniques may catch formally these changes which one may evaluate applying different mathematical instruments. Therefore, studying of the graph transformation theory, graphs and their applications have gained importance in different scientific and engineering realms.

The graph transformation approaches have emerged to overcome lack of expressiveness in traditional rewriting techniques, for example, Chomsky grammars [11], to transform non-linear structures like graphs. Heckel [26] claims that the first proposals, appearing in the late sixties and early seventies [43, 46], are concerned with rule based image recognition, translation of diagram languages, etc.

The graph theory distinguishes two kinds of graphs [11]:

- type graph,
- instance graph.

Definition 3.3 (Type graph) ***Type graph*** *represents concepts (classes) and their relations.*

Definition 3.4 (Instance graph) ***Instance graph*** *is an instance of a type graph.*

In other words, an entity in the given type graph is a class, while an entity in the corresponding instance graph is an object of the given class. We can consider a type graph as a model (for example, class diagram), while the corresponding instance graph is an instance of the model. A set of snapshots comprises an execution, where each snapshot represents an instance of the model at a particular moment in time. A snapshot is a set of objects; therefore, it may be seen as a graph. In the graph theory a structural changes of a graph one may describe via a *rule*. A rule is another fundamental concept of the graph transformation realm. It explicitly defines how a graph evolves. A transformation of a type graph to another type graph denotes a *model-to-model/out-place* transformation, while an *in-place* transformation represents the transformation of an instance graph [3]. Figure 3.1 shows the basic concepts of the graph theory. We introduce two

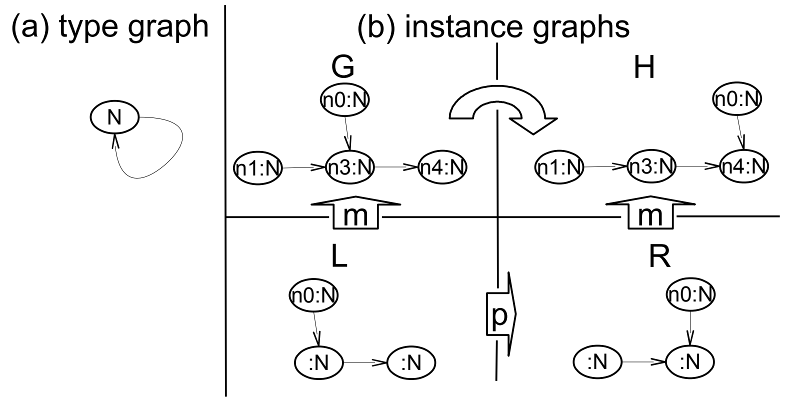


Figure 3.1: Graph transformation theory concepts

diagrams, i.e. (a) contains the type graph, while (b) contains four instance graphs of the type graph in (a). Our type graph contains one vertex N and the edge which starts and ends on the same vertex N. In our case, the diagram (a) can be seen as a class diagram, where the vertex N is a class of the kind N and the edge is an association. Two graph (G, H) in the diagram (b) are instances of the type graph. Graphs G, H contain four elements of the type N, two graphs are almost identical, but the vertex *n0* points to *n3* in G and references the vertex *n4* in H. Thus, we may say that the graphs G and H are the snapshots of the transformation process. Where we can use a rule to define this transformation. The graph L matches a subgraph in the graph G if we map *n0* in L to *n0* in G and two other vertices in L to *n3* and *n4* respectively. Thus, the graph L corresponds to a pre-condition triggering the changes of the subgraph in the graph G. The graph R matches a subgraph in the graph H applying the same mapping strategy. We may say that the graph R is a post-condition for the subgraph in the graph H. A pair of a pre-condition (graph L) and post-condition (graph R) comprises the rule *p*, see Figure 3.1 on the preceding page (b).

Definition 3.5 (Rule) ***Rule*** *defines a pre-condition and post-condition for the transformation.*

In Figure 3.1 on the facing page, the arrow with *p* inside schematically denotes the rule, while the arrows with *m* inside reflect the matches in the corresponding graphs.

Definition 3.6 (Pre-condition) ***Pre-condition*** *for transformation specifies a left-hand side (LHS) of the rule.*

Definition 3.7 (Post-condition) ***Post-condition*** *for transformation specifies a right-hand side (RHS) of the rule.*

Heckel [26] shows informally that a graph transformation from a pre-state G to a post-state H executes in three steps:

- search for a match of the left-hand side L in G,
- remove all nodes and edges in G from L \ R,
- paste a copy of R \ L, yielding the graph H.

In Figure 3.1 on the preceding page the match of LHS in G is a set of elements {*n0*, *n3*, *n4*} and corresponding edges. L \ R equals to the vertex *n0* and edge which starts on the vertex *n0* and ends on *n3*. Therefore, the vertex and edge are removed from G. R \ L comprises the vertex *n0* and edge which starts on *n0* and ends on *n4*. Further, we glue the corresponding vertex and edge on recently modified G yielding the graph H.

3.2 Algebraic approaches

Algebraic approaches seem to be the most popular techniques to graph transformations, but there are also others like Triple Graph Grammars (TGG) [52]. Schürr [51] introduces TGG as a new formalism to support an advanced declarative approach to graph transformations, which even allows defining context-sensitive transformations. The author uses the algebraic graph grammar to prove and explain the basic TGG concepts. However, the formalism can exploit any graph grammar approach as the base for TGG [51]. In the thesis, we present the following algebraic approaches:

- double-pushout (DPO) approach [18],
- single-pushout (SPO) approach [36].

and investigate the fragment substitution operation using such techniques to graph transformations.

Definition 3.8 (Algebraic approach) ***Algebraic approach*** *is an approach to graph transformations which defines how the graph B derives from the graph A using morphisms and pushouts.*

Where the term morphism we specify as follows.

Definition 3.9 (Morphism) ***Morphism*** *is a construction of the following structure:* $\boldsymbol{A} \overset{r}{\rightarrow} \boldsymbol{B}$, *where* $\boldsymbol{A}$ *is a source,* $\boldsymbol{B}$ *is a target and* $\boldsymbol{r}$ *is a function that defines the derivation from A to B.*

and the term pushout [1] is

Definition 3.10 (Pushout) *The pair ((k, g), D), where D is an object and k, g are two morphisms* $k : C \rightarrow D$, $g : B \rightarrow C$, *is a* ***pushout*** *of two morphisms* $f : A \rightarrow C$ *and* $h : A \rightarrow B$ *given that the diagram in Figure 3.2 (a) commutes.*

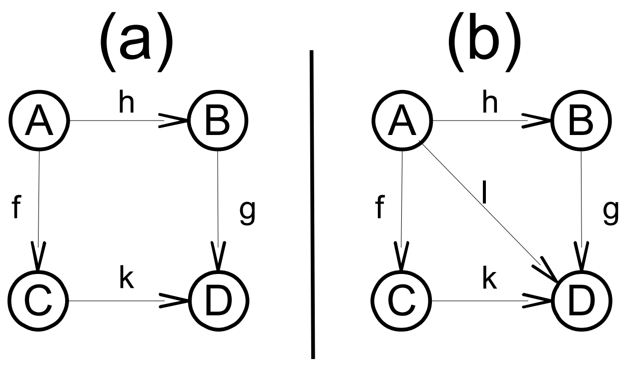


Figure 3.2: Pushout squares

Definition 3.11 (Commutative diagram) *A diagram is said to* ***commute*** *if any two paths between the same nodes compose to give the same morphism [4].*

In other words, if any two paths between the same vertices of a diagram yield the same result, then the diagram is commutative, i.e. commutes. Figure 3.2 (b) shows that the diagram in the same figure (a) is commutative. There are five morphisms in the figure (b), i.e. f : A → C, k : C → D, h : A → B, g : B → D, l : A → B, where f ∘ k results in l and h ∘ g yields l; thus, the diagram of the corresponding morphisms commutes (∘ denotes a composition of morphisms).

Formally, we can define the graph derivation in Figure 3.1 on page 20 (b) as d = $(G \overset{p,m}{\Longrightarrow} H)$ [13]. One may read this as a transformation from G to H applying the rule p at the match m. In DPO, a derivation of the graph H consists of two pushouts/gluing diagrams, which are schematically represented in Figure 3.3 on the next page. Four graphs define a transformation in DPO:

- L - left-hand side (LHS),
- R - right-hand side (RHS),

$$\begin{tikzcd}
L \arrow[d, "m"'] \arrow[dr, phantom, "(1)"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "d"] \arrow[dr, phantom, "(2)"] & R \arrow[d, "m^*"] \\
G & D \arrow[l, "l^*"] \arrow[r, "r^*"'] & H
\end{tikzcd}$$

Figure 3.3: The double-pushout approach (DPO)

- K - interface graph (part of the graph which should exist to apply the rule) [13],
- D - context graph (D = (G \ L) ∪ K) [13].

The first pushout in Figure 3.3 (1) represents a deletion of an occurrence of the graph L in G. Where the result of the deletion is the graph D. We represent this procedure as an inverse gluing operation. Meaning, that one should be able to yield the graph G by gluing L and D on the graph K having the graphs K, L and D. While we go other way around in practice, i.e. we yield the graph D having L, G and K. The second pushout in Figure 3.3 (2) inserts all elements of R in D which do not match any elements in K yielding the graph H [13, 18]. In other words, we perform additive operation of the graphs R and D, while graph K defines how we glue the graph R on the graph D resulting the graph H. Figure 3.4 shows how the diamond transforms into triangle using the DPO approach. The

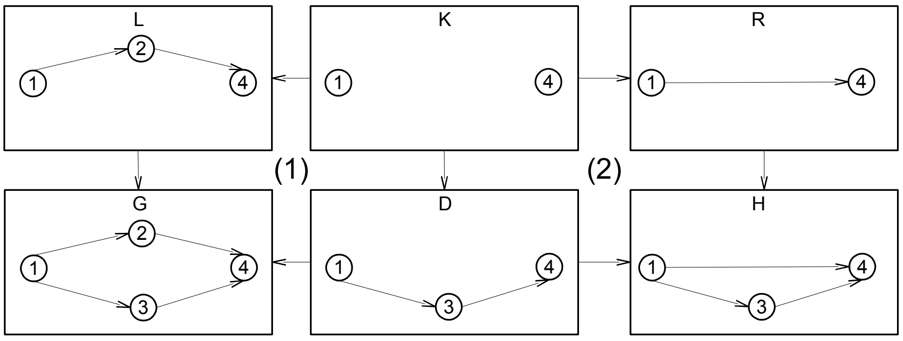


Figure 3.4: The DPO example

graph K contains two vertices *1, 4* defining the gluing points of the further transformation. The RHS R is glued on the graph D which is the graph G without the LHS L of the rule. It results in the triangle H. The DPO approach does not allow dangling edges. In order to preserve this invariant the approach has two *gluing conditions* which must hold [13]:

- dangling condition,

- identification condition.

The dangling condition claims if a rule defines deletion of vertices in G, then the rule should also specify a removal of all corresponding inwards and outwards edges. The identification condition requires that every vertex/edge of G that should be removed by application of a rule, matches a LHS only once. If these two conditions do not hold, a rule is not applied.

The SPO approach executes a derivation of a graph as a single pushout. Löe [36] describes the technique as follows: "... The single-pushout approach to graph transformation interprets a double-pushout transformation rule of the classical algebraic approach which consists of two total graph morphisms as a single partial morphism form the left- to the righ-hand side. ..." [36]. The SPO approach to graph transformations is different with respect to DPO, but Löe [36] stresses that SPO is a generalization of the DPO approach. Therefore, a DPO transformation can be obtained from a SPO by introducing an interface graph K. The

$$\begin{tikzcd}
L \arrow[r, "p"] \arrow[d, "m"'] \arrow[dr, phantom, "(1)"] & R \arrow[d, "m^*"] \\
G \arrow[r, "p^*"] & H
\end{tikzcd}$$

Figure 3.5: The single-pushout approach (SPO)

process of obtaining of the graph H from G schematically is shown in Figure 3.5. The diagram in Figure 3.5 has only one pushout. In SPO a transformation does not require two auxiliary graphs K and D which define transformation condition, context respectively and the transformation executes as a single transformation step. The SPO does not also impose any dangling conditions to hold. Therefore, the graph L in Figure 3.6 is a valid

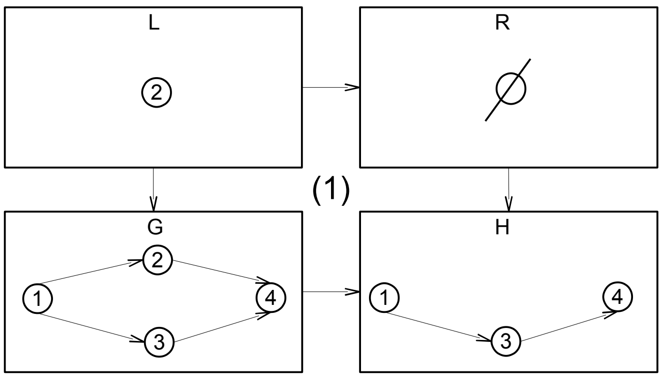


Figure 3.6: The SPO example

specification of LHS which matches the object 2 in G and the rule yields the correct transformation H. In other words, the SPO approach does not require a rule to specify all edges which start and end on a deleting vertex.

The basic difference between the DPO and SPO approaches is a way they handle dangling edges as it is shown in Figure 3.4 on page 23 and Figure 3.6 on the preceding page. The DPO approach does require to specify in a rule all inwards and outwards edges if a corresponding vertex has to be removed. While the SPO approach does not need to specify all deleting edges in a rule. The approach removes all dangling edges, i.e. in the single-pushout technique deletion has priority over preservation. On the contrary, preservation has priority over deletion in DPO [13]. The algebraic approaches have the well established theoretical bases. The pushout diagrams have formal mathematical representations and corresponding theorems with their proofs which ensure valid transformations if one follows the technique.

3.3 Tooling

The first attempts to use the graph rewriting techniques in the software engineering domain is the Progress [49, 50] approach [26]. Table 3.1 [61] gives a list of the available tools and their application areas.

Table 3.1: Available graph-based tools

Name	**Description**
Tools that are application domain neutral	
GrGen.NET	the graph rewrite generator, a graph transformation tool emitting C#-code or .NET-assemblies.
AGG	the attributed graph grammar system (Java).
GP (Graph Programs)	a programming language for computing on graphs by the directed application of graph transformation rules.
GMTE	the Graph Matching and Transformation Engine for graph matching and transformation. It is an implementation of an extension of Messmer's algorithm using C++.
Tools that solve software engineering tasks (mainly MDA) with graph rewriting	
GReAT	a model transformation language for model integrated modeling in the GME environment.
VIATRA	a transformation-based framework for validation and verification models in UML.
Gremlin	a graph-based programming language.
PROGRES	an integrated environment and very high level language for PROgrammed Graph REwriting Systems.
Fujaba	uses Story driven modelling, a graph rewrite language based on PROGRES.

Continued on the next page

Table 3.1 – *Continued from the previous page*

Name	**Description**
EMorF	graph rewriting systems based on EMF, supporting in-place model transformation and model to model transformation.
Henshin	graph rewriting systems based on EMF, supporting in-place model transformation and model to model transformation.
Mechanical engineering	
GraphSynth	an interpreter and UI environment for creating unrestricted graph grammars as well as testing and searching the resultant language variant.
booggie	integrates GrGen.NET with a port-based metamodel definition and an OpenGL graph visualization based on Tulip.
Artificial Intelligence/Natural Language Processing	
OpenCog	provides a basic pattern matcher (on hypergraphs) which is used to implement various AI algorithms.
RelEx	an English-language parser that employs graph rewriting to convert a link parse into a dependency parse.

As Table 3.1 on the previous page shows many domains employ the graph rewriting techniques and the tools use different platforms. We will give a short overview of Henshin [3] and EMorF [33], since EMF based tools are of our interest in the thesis.

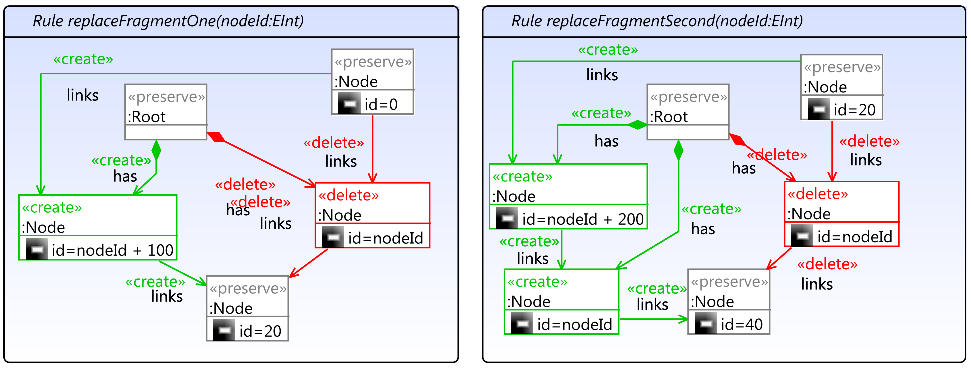


Figure 3.7: Transformation rules in Henshin

Hanshin is a declarative graph transformation language, which implements the DPO approach, with wide tool support. The language operates on EMF-based models and allows both out-place and in-place modifications. The language offers means for reasoning. Its basic concepts like rules are enhanced by powerful application conditions and attribute evaluation is flexible and implemented with Java and JavaScript [3]. The language is also enriched by control structures which allow defining the order of rule application in a modular manner [3]. The Henshin tool comprises an engine, several editors and means for reasoning by model checking [3]. Figure 3.7 on the facing page shows how Henshin defines transformation rules. The tool uses the unified editor, where one declaratively defines a transformation. The rules specify removal of the element of the type Node if *id* equals to *nodeId*, where *nodeId* is a parameter. Note, that a removed element should have one inward and one outward reference and should be contained. The elements in red and titled as 'delete' are elements to remove. We also substitute the removed element with another element of the same type, marked in green and with the caption 'create'. Note, that we simply specify the replacement operation. The operationes which we have specified is automatically translated into LHS and RHS, which we can observe and modify in another editor. In Figure 3.8, we introduce another

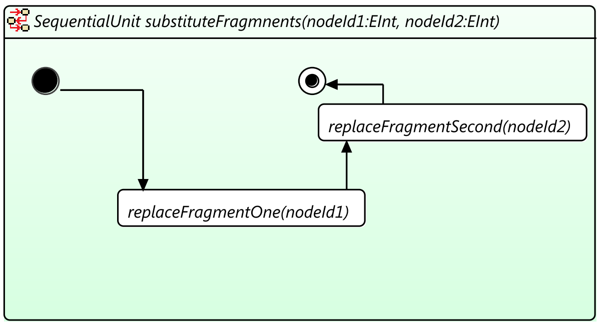

SequentialUnit substituteFragmnents(nodeId1:EInt, nodeId2:EInt)
replaceFragmentSecond(nodeId2)
replaceFragmentOne(nodeId1)

Figure 3.8: Henshin Sequential Unit

powerful concept of Henshin, i.e. sequential unit. It allows defining the rule execution order. In given example, we apply two rules which are executed sequentially. If one of these rules fails, then the whole operation is rolled back. The Henshin language provides six kinds of units, each of them has its own properties and features, Arendt et al. [3] discuss them in details.

EMorF is an open source graph transformation tool based on the EMF framework. It enables both in-place and out-place model transformations. The EMorF tool exploits the same approach as Henshin to define rules graphically in the declarative manner. In addition, it pays special attention to out-place/model-to-model transformations. Rules for model-to-model transformations are defined by triple graph grammars [33]. To increase expressiveness, EMorF incorporates also the OCL language to specify constraints and application conditions [33].

3.4 Graph transformations in the thesis

We will investigate whether the graph transformation techniques and tools based on the graph theory solve our problem, which we state later in the thesis, directly. Thus, we try to specify required operations in the Henshin graph transformation tool. It appears that the problem is general and not solvable straightforward using the graph rewriting approach. We will also show that a rewriting of rules, which may do the work in certain cases, contradicts to the philosophy of CVL. Therefore, it is not applicable and acceptable in the given context. We give also an overview of other graph based approaches to variability modeling in the concluding sections and show how they are different with respect to CVL.

Part II

Contribution

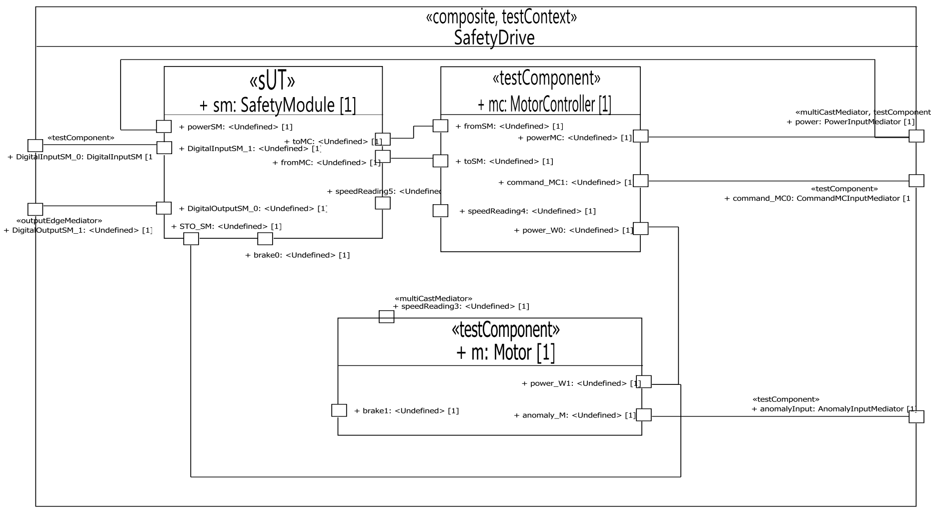
Chapter 4

Problem statement and proposed solutions

4.1 Substitution fragments in UML

4.1.1 Context

SINTEF and ABB have developed a software product line within the CESAR project [9]. The product line models and describes a variety of configurations for the SafetyDrive module. A safety drive is a part of machinery to control the speed of a motor in conveyor belts, hoisting machines and other means to move different goods, components, people in elevators etc. Figure 4.1 shows the composite structure of the safety drive module. We may see that the safety drive includes three components,



«composite, testContext»
SafetyDrive
«sUT»
+ sm: SafetyModule [1]
+ powerSM: <Undefined> [1]
+ DigitalInputSM_1: <Undefined> [1]
+ toMC: <Undefined> [1]
+ fromMC: <Undefined> [1]
+ speedReading5: <Undefined
+ DigitalOutputSM_0: <Undefined> [1]
+ STO_SM: <Undefined> [1]
+ brake0: <Undefined> [1]
«testComponent»
+ DigitalInputSM_0: DigitalInputSM [1
«outputEdgeMediator»
+ DigitalOutputSM_1: <Undefined> [1]
«testComponent»
+ mc: MotorController [1]
+ fromSM: <Undefined> [1]
+ powerMC: <Undefined> [1]
+ toSM: <Undefined> [1]
+ command_MC1: <Undefined> [1
+ speedReading4: <Undefined> [1]
+ power_W0: <Undefined> [1]
«multiCastMediator, testComponent»
+ power: PowerInputMediator [1]
«testComponent»
+ command_MC0: CommandMCInputMediator [1
«multiCastMediator»
+ speedReading3: <Undefined> [1]
«testComponent»
+ m: Motor [1]
+ power_W1: <Undefined> [1]
+ brake1: <Undefined> [1]
+ anomaly_M: <Undefined> [1]
«testComponent»
+ anomalyInput: AnomalyInputMediator [1]

Figure 4.1: The SafetyDrive composite structure

i.e. SafetyModule, Motor, and MotorController. We may find that the safety module has links to the motor and motor controller. It denotes communication paths between all three components. The safety module, motor, and motor controller are properties of the safety drive class in UML terminology.

One of the project goals was to incorporate quality assurance mechanisms, like testing, into the model. Thus, the given composite structure describes a test context of the safety drive. The elements of the composite structure are stereotyped with the UML Testing Profile. One may notice

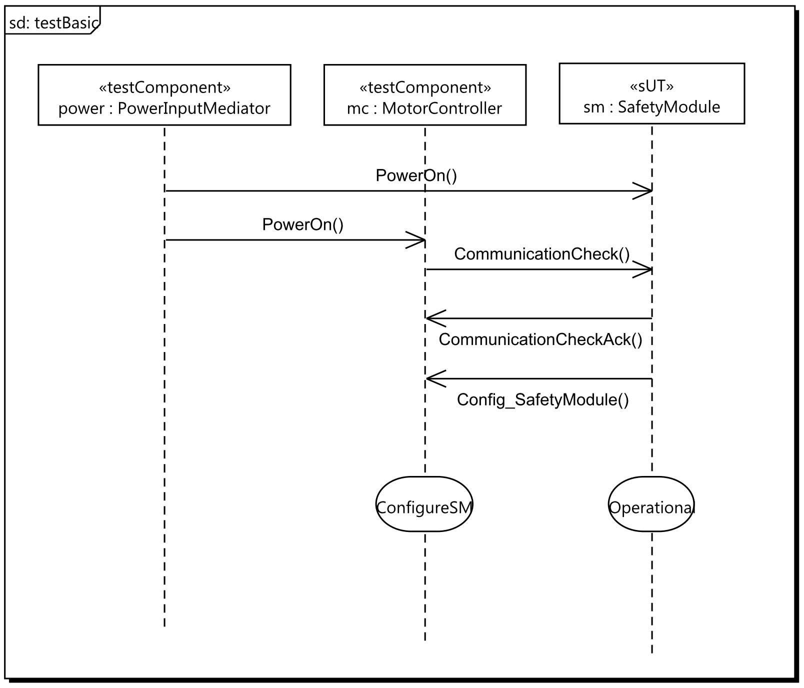


Figure 4.2: The SafetyDrive test case definition

that the safety module has the 'sUT' stereotype (system Under Test). It denotes that a focus of a test case is the safety module. We consider the safety module to be a black box. Figure 4.2 specifies a possible test scenario which is consistent with the given composite structure. The test case simply checks whether the safety module reaches the Operational state when we turn on the power.

4.1.2 Neighboring placements problem

Overview

We define variability on the base model where both the composite structure (see Figure 4.1 on the preceding page) and sequence diagram (see Figure 4.2) reference the same base model. One may want to test a safety module of another type, or simply another safety module which supports the same interface and communication protocol with the motor and motor controller. In order to achieve the goal, we need to perform several substitutions. Firstly, we need to substitute the SafetyModule property in the composite structure diagram in Figure 4.1 on the preceding

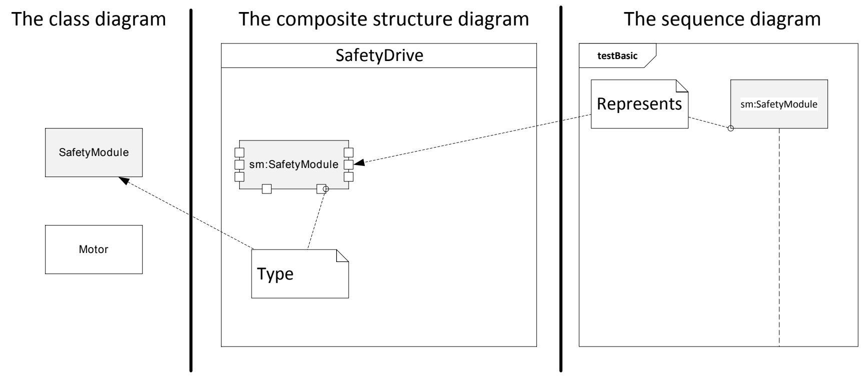


Figure 4.3: The simplified class, composite structure and sequence diagrams

page with a desired element. Secondly, we have to adjust the sequence diagram in Figure 4.2 on the preceding page, i.e. the lifeline currently typed as SafetyModule should reference to the correct element, in order to keep the model consistent. Finally, we need to replace a corresponding TypedElement. TypedElement is a generalization of the Property element in UML, i.e. a Property element in a composite structure may reference a Class, StateMachine etc. Figure 4.3 shows three sketches of the UML diagrams for the SafetyDrive example, where we have shaded elements to substitute. In Figure 4.4, we observe the simplified UML metamodel and

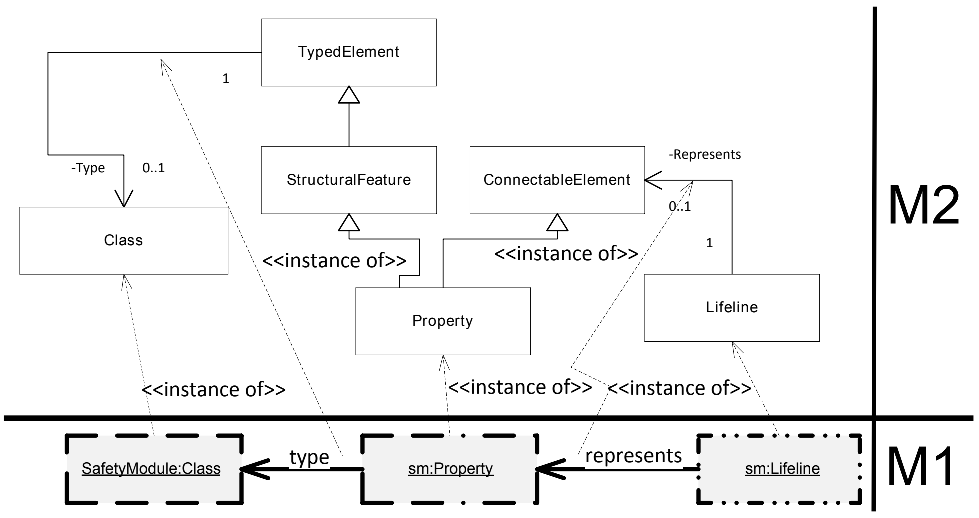


Figure 4.4: The instance diagram and simplified UML metamodel

model instances of the diagram elements from Figure 4.3. One should note that elements in different diagrams may reference each other via a model even though it may not be visible at first sight. The diagram in Figure 4.4 shows that the objects of the types Lifeline, Property and Class have links according to the UML metamodel.

Problem

We have shaded the elements in Figure 4.4 on the preceding page, which we have to substitute for the required effect. All three elements have different borders indicating different substitutions. The MoSiS CVL tool performs these three substitutions (where the substituted elements are adjacent and have reference between each other) yielding a final product which does not conform to the expectation. In particular, an execution breaks the relations between the lifeline, property and class.

An element in the UML diagram is a representation of the corresponding element in the model. Moreover, different diagrams may have representations of the same model elements. At the same time, diagrams (representing the same model elements) may not be tightly coupled between each over. Thus, changes made in one diagram may not be reflected in another diagram or may not be visible due to specifics of the diagram. In other words, in UML, different diagrams are views reflecting a different nature (behavioral, static etc) of the same model, where elements can cross-reference each other in the model. During modeling, we focus on a specific aspect of a system using different diagrams. It helps to keep the definition of a system clear, concise and feasible. In addition, it is a de facto standard in modeling. Thus, we can define substitution fragments (tailored to a specific functionality) in different diagrams (see for example Figure 4.3 on the previous page). It may appear that the model elements in the defined fragments are coupled with each other through links while the coupling is not visible on the diagram level (e.g. type, references in Figure 4.4 on the preceding page). Separate independent substitutions of such tightly coupled elements do not yield a correct model using the MoSiS CVL tool, i.e. the operations do not restore the links between the elements. Subsequently, it appears that graph transformation tools do not solve the problem either. Therefore, we argue that the problem is more general and of the high relevance and importance. We show this later in the thesis, discuss a reason of such behavior and propose a solution for the problem.

4.2 Approach

4.2.1 Challenges

Different metamodels

The OMG CVL metamodel has alterations with respect to MoSiS CVL. For example, the OMG CVL specification enhances means to describe placement and replacement fragments. Particularly, the mechanism that references elements of a base model has changed, see Figure 2.5 on page 15. OMG CVL has extra references for boundaries of the kind *FromPlacement*. In Figure 2.5 on page 15 the *insideBoundaryElement* association represents the reference pointing to an element inside a substitution fragment. It facilitates identifying elements of a placement fragment. Figure 2.5 on page 15 shows also the *outsideBoundaryElement* association for a boundary of the kind *ToReplacement*. The other associations one may find in the MoSiS CVL metamodel as well. Thus, models build in MoSiS CVL are not compatible with models in OMG CVL.

The OMG CVL tool is not available

We are currently developing the OMG CVL tool and various parts of the tool are on different stages of the development. Thus, we do not have a consistent framework to build models which are compatible with the OMG CVL metamodel. It complicates the work which the thesis outlines. On the one hand, a direct conversion from models built in MoSiS CVL to OMG CVL is not possible due to the differences in the metamodels. On the other hand, the MoSiS CVL tool is fully functional and can accomplish tasks automatically which are tedious and time consuming. For example, we

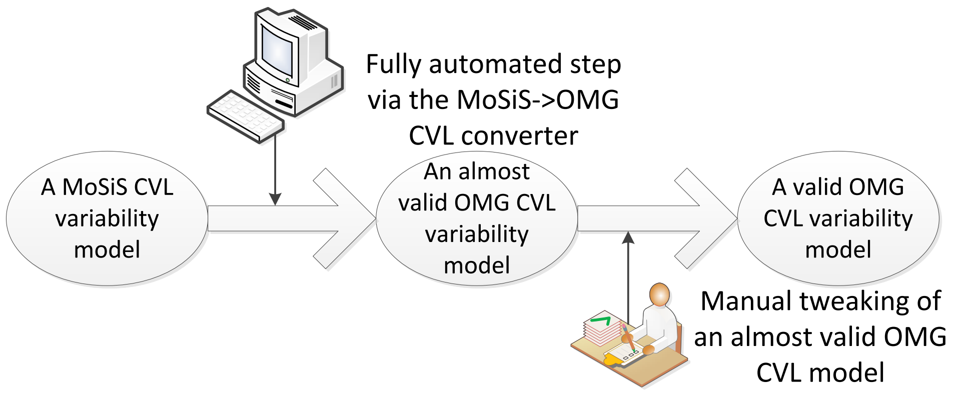

Fully automated step via the MoSiS->OMG CVL converter
A MoSiS CVL variability model
An almost valid OMG CVL variability model
A valid OMG CVL variability model
Manual tweaking of an almost valid OMG CVL model

Figure 4.5: Flow to obtain a valid OMG CVL model

can define substitution fragments and build variability models in MoSiS CVL in general. Thus, it is an advantage to utilize the MoSiS CVL tool comparing to if one does the same work manually. Even though the metamodels of MoSiS CVL and OMG CVL are different, we should admit that the partial conversion is feasible and the technique helps to come up with almost valid substitution fragments. Further, we may manually adjust the almost valid variability models to agree to the OMG CVL metamodel. Figure 4.5 depicts the elaborated process of obtaining a valid OMG CVL model.

Lack of samples

The MoSiS CVL tool has been actively used within several European projects; therefore, we have some product line definitions. However, the models were built to avoid cases where the MoSiS CVL is weak. Thus, we do not have a lot of practical examples with neighboring substitution fragments, but we could modify them to come across the problem of the interest. It complicates the process of validation and verification of proposed approaches to tackle the problem.

The large UML metamodel

Even relatively small product lines may look complicated. Model elements of a single substitution fragment may have intricate relations to other model elements. It depends on a language and complexity of its metamodel. We have used UML as a base language for several product lines including the SafetyDrive case. The UML metamodel is rather large (the UML metamodel has 255 metaclassses and each of them can have several associations to other classes); therefore, elements of substitution fragments defined on a UML-based base model have a lot of references to the rest model elements. Even though the MoSiS CVL tool takes care of the most complications, the elaborated process of obtaining a valid OMG CVL model involves a manual adjustment. Thus, a UML based example may complicate a process of attaining a valid instance of the OMG CVL metamodel. One may suggest to use a language with the effortless metamodel.

4.2.2 Node DSL and problem generalization

We introduce the Node DSL, in order to tackle challenges from Section 4.2.1. In Figure 4.6 one may see the metamodel of the Node language. We

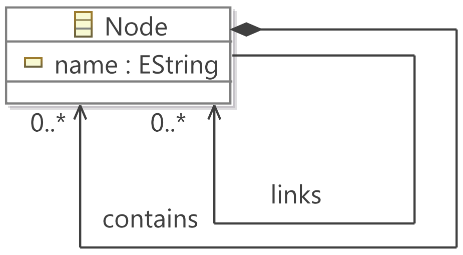
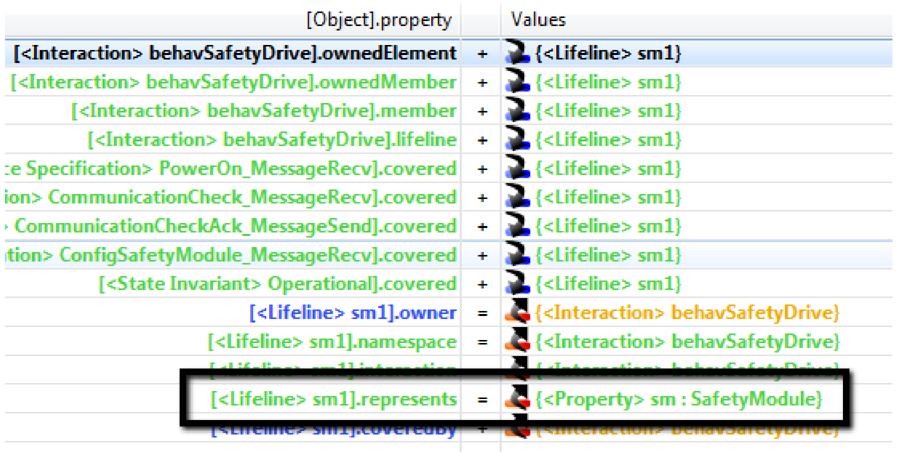


Figure 4.6: Node DSL - metamodel

may note that the metamodel of the language is extremely uncomplicated. It contains a single element of the kind Node and two references, i.e. the containment 'contains' and the association 'links'. Both references start and end on the element Node, meaning that every Node element may contain and/or link another Node. On the one hand, the suggested metamodel allows simulating all possible relations between elements. On the other hand, it simplifies and facilitates the elaborated process in Figure 4.5 on the preceding page of obtaining valid OMG CVL models.

Figure 4.7 on the next page shows the binding editor, where one may observe all outwards/inwards references (represented as the rows of the table) of the placement fragment for *sm:Lifeline*. Where we are interested in a single row in the black rectangle, the row denotes the 'represents' reference between *sm:Lifeline* and *sm:Property* in Figure 4.4 on page 33. While other rows just represent other links of *sm:Lifeline* reflecting the UML metamodel. Since the metamodels of MoSiS CVL and OMG CVL are



[Object].property		Values
[<Interaction> behavSafetyDrive].ownedElement	+	{<Lifeline> sm1}
[<Interaction> behavSafetyDrive].ownedMember	+	{<Lifeline> sm1}
[<Interaction> behavSafetyDrive].member	+	{<Lifeline> sm1}
[<Interaction> behavSafetyDrive].lifeline	+	{<Lifeline> sm1}
:e Specification> PowerOn_MessageRecv].covered	+	{<Lifeline> sm1}
ion> CommunicationCheck_MessageRecv].covered	+	{<Lifeline> sm1}
› CommunicationCheckAck_MessageSend].covered	+	{<Lifeline> sm1}
tion> ConfigSafetyModule_MessageRecv].covered	+	{<Lifeline> sm1}
[<State Invariant> Operational].covered	+	{<Lifeline> sm1}
[<Lifeline> sm1].owner	=	{<Interaction> behavSafetyDrive}
[<Lifeline> sm1].namespace	=	{<Interaction> behavSafetyDrive}
[<Lifeline> sm1].interaction	=	{<Interaction> behavSafetyDrive}
[<Lifeline> sm1].represents	=	{<Property> sm : SafetyModule}
[<Lifeline> sm1].coveredBy	+	{<Interaction> behavSafetyDrive}

Figure 4.7: The inwards/outwards references of the substitution fragment in UML

different, these references unnecessary complicate the process of derivation a valid OMG CVL model. We may remove this complication applying the Node language to imitate relations between elements. Figure 4.8 shows

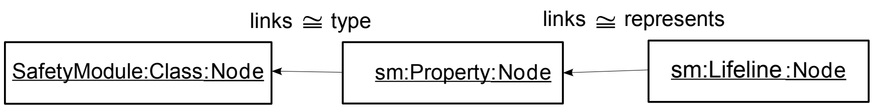
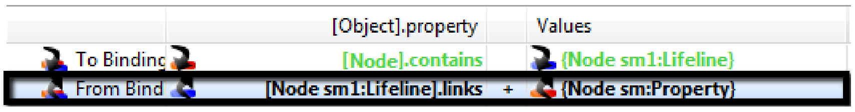

$$\begin{tikzcd}\boxed{\underline{\text{SafetyModule:Class:Node}}} & \boxed{\underline{\text{sm:Property:Node}}} \arrow[l, "\text{links} \cong \text{type}"'] & \boxed{\underline{\text{sm:Lifeline :Node}}} \arrow[l, "\text{links} \cong \text{represents}"']\end{tikzcd}$$

Figure 4.8: Simulation in the Node language

an instance of the Node metamodel that simulates the configuration from Figure 4.4 on page 33. Three nodes in Figure 4.8 represent *sm:Lifeline*, *sm:Property* and *SafetyModule:Class*. Two 'links' references simulate the 'represents' and 'type' associations between *sm:Lifeline*, *sm:Property* and *SafetyModule:Class* respectively. In other words, all nodes in Figure 4.8 simulate the neighboring relations between the elements of the placements from Figure 4.4 on page 33. Figure 4.9 shows the binding editor showing



	[Object].property		Values
To Binding	[Node].contains		{Node sm1:Lifeline}
From Bind	[Node sm1:Lifeline].links	+	{Node sm:Property}

Figure 4.9: The inwards/outwards references of the simplified model

all inwards/outwards references of the same kind substitution fragment, i.e. the substitution of *sm:Lifeline* (we replace *sm:Lifeline* with *sm1:Lifeline*), but we use the simplified model in the Node language (see Figure 4.8). One may notice that we reduce the amount of all links leading to/from the substitution fragment. At the same time, we preserve the reference of the interest (highlighted by the black rectangle). Thus, we reproduce and generalize the neighboring placement problem in the Node-based model which has more comprehensible metamodel. It simplifies the variability model and allows focusing on the problem area which is a link between neighboring elements. We put this problem on the front line with help of the Node language while it is hidden and not obvious in UML-based models due to complexity of the metamodel. In return, the Node DSL speeds up creation of simple highly tailored to a specific problem variability models. In addition, it enables more efficient verification and validation of the implemented engine since we can create a wide range of verification samples. Further, we will speak of models in the Node language if we do not mention opposite.

4.2.3 Graph rewriting to the neighboring placements problem

Applying the DPO approach to substitutions in CVL

CVL has means to define substitution fragments directly in a model. The MoSiS CVL tool utilizes such means via the simple selection procedure specifying placement and replacement fragments directly in a base model. Subsequently, an engineer defines how a replacement fragment is instrumented into the base model via the binding process. In Figure 4.10 one may see two placement fragments. There are two ovals, which schematically outline two placement fragments.

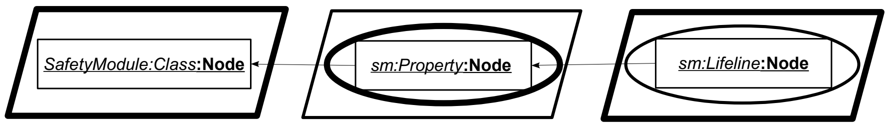


Figure 4.10: Two placement fragments

The oval with the thin line denotes the placement fragment for the *sm:Lifeline* element, i.e. *sm:Lifeline* should be replaced with another element. The second oval with the thick line specifies the placement fragment for *sm:Property*.

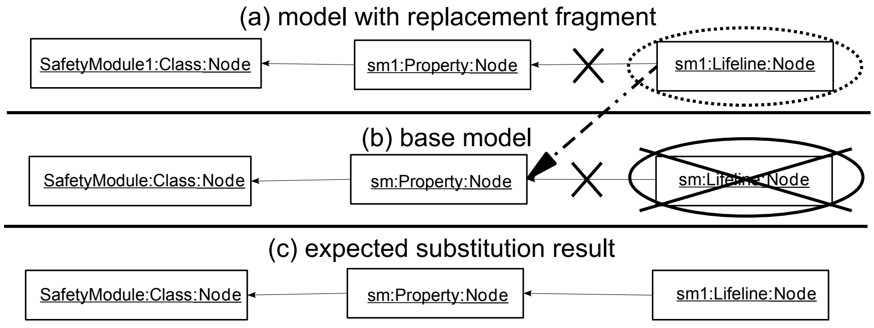


Figure 4.11: Substitution fragments, bindings and the substitution result

Two parallelograms with the thick line are the neighboring elements for the placement fragment *sm:Property* while the parallelogram with the thin line is the neighboring element for the placement *sm:Lifeline*. The MoSiS CVL tool calculates neighboring elements automatically. In order to specify a substitution, a variability engineer has to define how elements of a replacement fragment should reference placement neighboring elements in a base model. In the example from Figure 4.11 on the preceding page (a, b), one defines that the links from *sm1:Lifeline* to *sm1:Property* and *sm:Lifeline* to *sm:Property* should be cut off and the node *sm1:Lifeline* should reference the neighboring element *sm:Property* in the base model. We schematically show this process in Figure 4.11 on the facing page (a, b). The dashed arrow with the filled head represents the binding instructing to create a reference between *sm1:Lifeline* and *sm:Property* in the resolved product in Figure 4.11 on the preceding page (c). A variability engineer uses the binding editor in the MoSiS CVL tool to specify such references. The black rectangle in Figure 4.9 on page 37 highlights the row in the binding editor that specifies the binding between *sm1:Lifeline* and *sm:Property*. The substitution operation yields the product in Figure 4.11 on the preceding page (c) removing the placement fragment *sm:Lifeline*. The same procedure we should apply to the second placement fragment (*sm:Property*), i.e. we should bind the neighboring element *sm:Lifeline* in the base model to *sm1:Property* and *sm1:Property* should be bound to *SafetyModule:Class*.

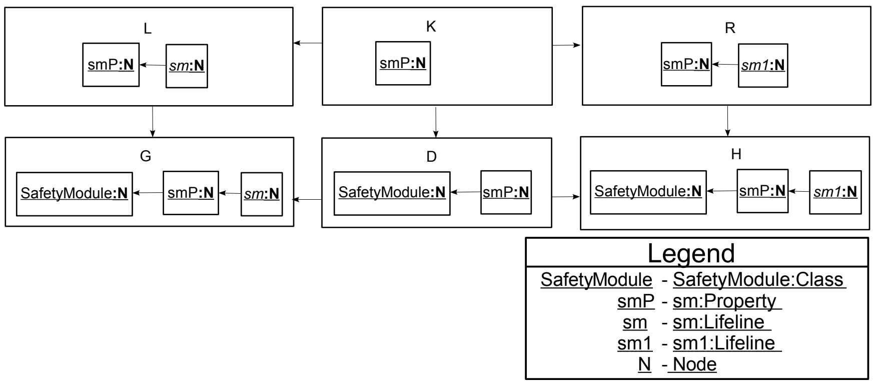


Figure 4.12: The CVL substitution in terms of DPO

To sum up neighboring elements of a placement define precisely a part of a model to be modified. In the example, *sm:Property* is a neighboring element for the placement *sm:Lifeline*. We should keep this element in the model and a new element (*sm1:Lifeline*) should be attached to the element, while the current one (*sm:Lifeline*) should be removed. Thus, one may see neighboring elements as *gluing points* (in terms of the graph transformation theory) which constitute the graph K in the DPO approach, see Figure 4.12. Placement neighboring elements of a base model together with elements of a placement constitute LHS (the graph L in Figure 4.12). While, placement neighboring elements with elements of a corresponding replacement form RHS, for example, the graph R in Figure 4.12. A derived product represents a target graph in DPO, e.g. the graph H in the figure. A source graph is a base model right before a substitution (the graph G in Figure 4.12). Therefore, one may express a substitution operation in CVL in terms of the graph rewriting theory.

Simulation approach

We may specify the substitutions in Figure 4.10 on page 38 using graph transformation techniques using the EMF based graph transformation tool Henshin. Figure 4.13 shows two rules and a sequence unit that defines the

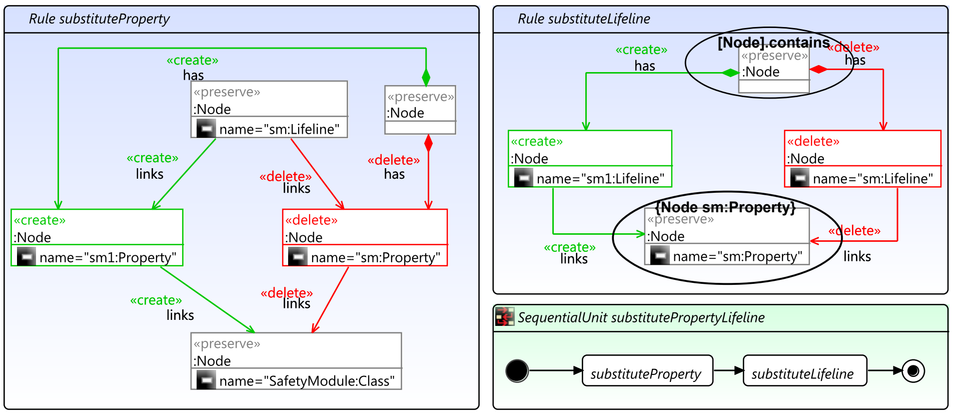


Figure 4.13: Two substitutions in Henshin

rule execution order. The first rule specifies the substitution of *sm:Property* while the second one defines the substitution of *sm:Lifeline*. One may notice that we use the same style as CVL to define the neighborings or gluing points in terms of the graph transformation theory. The rule *substituteLifeline* defines the substitution of the node *sm:Lifeline*, where we specify that the link from *sm:Lifeline* to *sm:Property* should be removed and the replacement *sm1:Lifeline* should reference the neighboring element *sm:Property*. In addition, one may notice the 'contains' relation which we remove for the placement element and create for the replacement element. It exactly matches the table in the binding editor in Figure 4.9 on page 37 where the first row instructs to create the 'contains' relation and the second row specifies that the replacement element *sm1:Lifeline* should reference *sm:Property*. Thus, two objects in the table with names *{Node sm:Propery}* and *[Node].contains* stand for the corresponding gluing points highlighted with the black ovals in Figure 4.13. An execution of the rules does not result in two substitutions in Figure 4.13. The rule *substituteProperty* disables the second substitution. The specified rules are not parallel independent; thus, we do not get the desired result, where both *SafetyModule:Property* and *SafetyModule:Lifeline* are replaced. Roughly speaking two transformations are independent if left-hand sides of both rules do not intersect [44]. In other words, if one transformation removes elements, which are used by another transformation, then the execution of both rules is not possible. Thus, the graph approach does not solve the issue of not establishing links between elements of the neighboring placements.

Wild card approach

There is another approach to define the transformations which we show in Figure 4.10 on page 38. Figure 4.14 on the facing page recaps the rules in Figure 4.13 on the preceding page with small modifications. In this

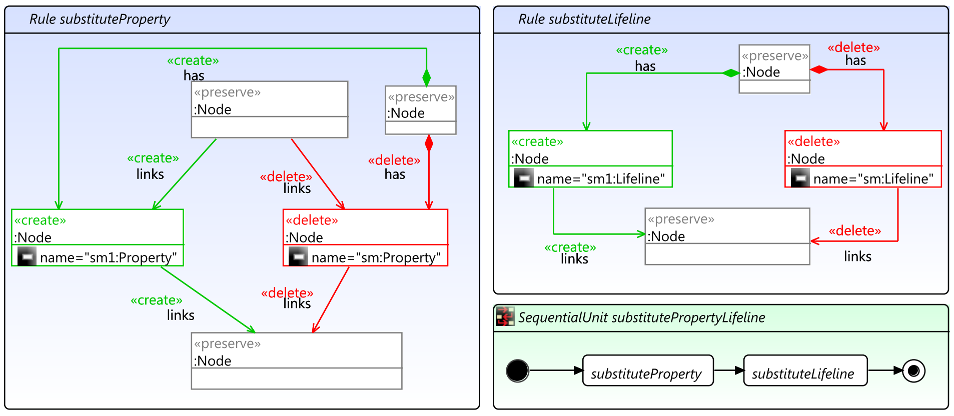


Figure 4.14: Two substitutions in Henshin - the wildcard approach

version of the rules in Henshin, we have removed the attribute 'name' for the gluing points. It indicates that the elements match any element of the type Node. Potentially, these elements can match any element in a model written in the Node language. We call such elements as wildcards. A wildcard element match any element. Thus, we do not impose any specific criteria on the gluing points except being of the type Node. In contrast to the simulation approach, the subsequent execution of the rules results in two transformations reaching the desired result, see Figure 4.15.

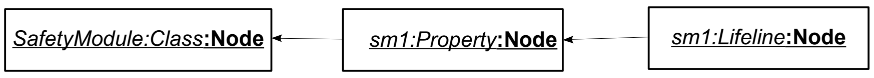


Figure 4.15: Expected result of two substitutions

However, the rules in this formulation do not simulate substitutions of the placement fragments in CVL. When we specify a substitution fragment in CVL we are exact and precise. Meaning, that we ask two kind of questions.

1. What do we want to substitute?

2. Where and how do we want to perform substitution?

An answer to the fist question is the selection procedure (in the MoSiS CVL tool) which defines definitive and precisely objects to substitute. While the binding process answers the second question, i.e. we specify where and how elements of a replacement should be injected in a model. It ensures that we perform the substitution operation for a given fragment only once. Each neighboring element matches exactly one element in a model, strictly speaking, the neighboring element is an element of the model itself. While the wild card approach specifies a meta-element which potentially may match many elements. It is somewhat different comparing to how CVL specification defines means to reference element of a model. Moreover, the wild card approach does not guaranty that a substitution performs only once. We can say even opposite that a transformation triggers every time the defined pattern matches. In addition, this approach does not allow imposing any restrictions on elements which could be a way around to strengthen a pattern and reduce matches. Thus, we do not consider the wild card approach as a solution for the neighboring placements problem.

4.3 Overlapping placements

4.3.1 Definitions and concepts

As we have already shown in the motivation example from Figure 1.1 on page 2 that we may define placement fragments such as they overlap. On the one hand, there are some cases, where a resolution result seems to be well understood despite overlaps between fragments. In other words, we do not see any pragmatic difficulties rather technical. On the other hand, there are situations, where a result is difficult to realize, and we observe some logical, structural conflicts. Thus, let us first classify possible relations between different placements. Therefore, we would like to introduce an alternative definition of a placement (replacement) fragment, which should help to exemplify all cases. We use models in the Node language in further discussions where a circle represents an instance of the Node class.

A placement fragment forms a conceptual 'hole' in a base model according to Definition 2.12 on page 14. Rephrasing, we may consider a placement as a set of chosen elements through an effortless selection procedure in a base model. The fragment substitution operation removes all elements of the placement during a resolution process. A replacement fragment later fills out the conceptual 'hole' caused by the deletion of the placement fragment. Any placement fragment is defined by means of boundary elements in CVL. Boundary elements reference selected elements together with elements which we do not explicitly select (see Figure 2.4 on page 14). Boundary references of the kind *outsideBoundaryElement* (see Figure 2.5 on page 15) point to elements outside a placement fragment. Boundary references outline also a set of affected elements (neighboring elements or gluing points), which we do not remove during a resolution process. In addition, we do not explicitly select these elements. Hence, we can conclude that a placement is a set of objects which is wider than the set of the explicitly selected objects.

Definition 4.1 (Placement Element internal (PEint)) ***Placement Element internal*** *(PEint) is the set of all elements referred by insideBoundaryElement references (IBEs) and all elements in the transitive closure of all references from IBEs, but cut off references at elements found through outsideBoundaryElement references.*

Definition 4.2 (Placement Element external (PEext)) ***Placement Element external*** *(PEext) is a set of all elements referred by outsideBoundaryElement references.*

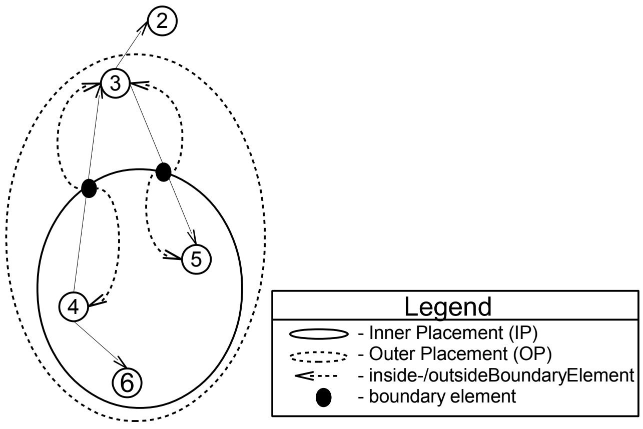


Figure 4.16: Inner Placement (IP) and Outer Placement (OP)

In Figure 4.16 PEint = {4, 5, 6} and PEext = {3}. Thus, we define two sets of elements (PEint, PEext), which are affected by a selection. The dashed arrows pointing to *3* are *outsideBoundaryElement* references, while the dashed arrows pointing to *4* and *5* are *insideBoundaryElement* references.

Definition 4.3 (Inner Placement (IP)) ***Inner Placement*** *(IP) is a set of all elements in PEint, i.e. IP $\equiv$ PEint.*

Definition 4.4 (Outer Placement (OP)) ***Outer Placement*** *(OP) is a set of all elements in PEext and Inner Placement (IP), i.e. PEext $\bigcup$ PEint.*

The oval in Figure 4.16 with the solid border outlines IP while the dashed bold line outlines OP.

4.3.2 Kinds of overlaps

A variability engineer defines a set of elements to substitute via an effortless selection in a base model. This selection constitutes PEint ($\equiv$IP) and a placement fragment in the CVL terminology. In other words, each placement represents a set of elements. Hence, the talk about relations between two placements can be discussed in terms of sets relations. There are two possible cases with respect to two different sets, i.e. two sets intersect or do not intersect. Formally, we may write the following.

1. $OP_1 \bigcap OP_2 = \varnothing$;
2. $OP_1 \bigcap OP_2 \neq \varnothing$.

If OPs of two placements do not intersect then the placements are independent.

Definition 4.5 (Independent placements) ***Independent placements*** *are placements, which do not share any common elements, i.e. their outer placements do not overlap.*

If two placements are independent then we may perform their resolutions in any order yielding the same result. In other words, the placements are confluent. Since independent placements do not bring any problems to the resolution process; therefore, the case where OPs do not intersect is not of our interest. Further, we speak of the case where two OPs intersect.

Let us have two placement where we define their OPs as follows:

- $\text{OP}_1 = \text{PEint}_1 \bigcup \text{PEext}_1$;
- $\text{OP}_2 = \text{PEint}_2 \bigcup \text{PEext}_2$.

Given that $\text{OP}_1 \bigcap \text{OP}_2 \neq \varnothing$, there are four possible intersection pairs between PEint_1, PEext_1 of OP_1 and PEint_2, PEext_2 of OP_2 according to a straightforward 2x2 table. Table 4.1 shows all possible combinations,

Table 4.1: The simple 2x2 table

	PEint_2	PEext_2
PEint_1	1	2
PEext_1	2	3

i.e. (PEint_1; PEint_2), (PEint_1; PEext_2), (PEext_1; PEint_2) and (PEext_1; PEext_2). Where two pairs (PEint_1; PEext_2) and (PEext_1; PEint_2) are symmetrical and of the same kind; therefore, we do not distinguish them. Other two pairs (PEint_1; PEint_2) and (PEext_1; PEext_2) are unique. We discard also the case (PEext_1; PEext_2), since CVL does not experience any problems tackling such configuration. We call such placements as parallel independent placements.

Definition 4.6 (Parallel independent placements) ***Parallel independent placements*** *are placements which just share their PEext.*

Parallel independent placements are confluent. Therefore, we just need to discuss the following two basic intersection kinds:

- $\text{PEint}_{1/2} \bigcap \text{PEint}_{2/1} \neq \varnothing$;
- $\text{PEint}_{1/2} \bigcap \text{PEext}_{2/1} \neq \varnothing$;

A non-empty intersection of these pairs indicates an overlap between OPs, where in some cases we may strengthen the statement by claiming overlap between corresponding IPs. We should also point out that an intersection between two different PEint implies an intersection of the corresponding PEint and PEext. While a pure intersection between PEint and PEext does not guarantee an intersection between the corresponding PEint-s. Thus, two basic intersection kinds emerge and look as follows:

- $\text{PEint}_{1/2} \bigcap \text{PEext}_{2/1} \neq \varnothing \wedge \text{PEint}_{1/2} \bigcap \text{PEint}_{2/1} = \varnothing \Rightarrow \text{IP}_{1/2} \bigcap \text{OP}_{2/1} \neq \varnothing \wedge \text{IP}_{1/2} \bigcap \text{IP}_{2/1} = \varnothing$;
- $\text{PEint}_{1/2} \bigcap \text{PEext}_{2/1} \neq \varnothing \wedge \text{PEint}_{1/2} \bigcap \text{PEint}_{2/1} \neq \varnothing \Rightarrow \text{IP}_{1/2} \bigcap \text{IP}_{2/1} \neq \varnothing \wedge \text{IP}_{1/2} \bigcap \text{OP}_{2/1} \neq \varnothing$;

These two cases lead to an incorrectly resolved model (MoSiS CVL [22]). There is also a special case of $\mathrm{PEint}_{1/2} \bigcap \mathrm{PEint}_{2/1} \neq \varnothing$, i.e.

- $\mathrm{PEint}_{1/2} \subseteq \mathrm{PEint}_{2/1} \Rightarrow \mathrm{IP}_{1/2} \subseteq \mathrm{IP}_{2/1}$.

Figure 4.17 depicts all three cases plus a cases which is hidden in an intersection between PEint-s and can not be expressed through a simple set relation.

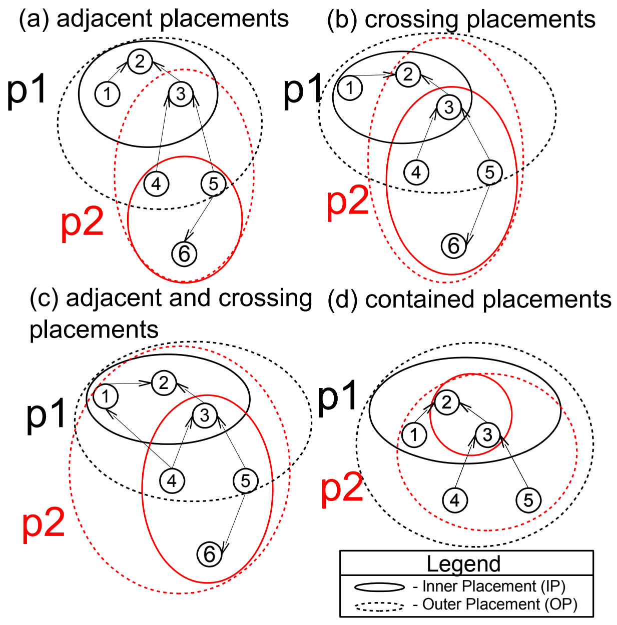


Figure 4.17: Overlapping kinds

Definition 4.7 (Adjacent relation) ***Adjacent relation*** *is a relation when two objects of different IPs have a direct reference between each over.*

Rephrasing, we need to consider two factors: two objects should belong to different IPs and objects should be directly coupled through a reference. In other words, a link relation between two elements of different IPs creates an adjacent relation between placements. Figure 4.17 presents two such cases:

- objects *4* and *5* of *p2* have the adjacent relation with object *3* in *p1* of the figure in the left top corner (a), i.e. *p1*, and *p2* are fragments of the kind adjacent placements;
- objects *1* and *4* have the reference, which defines the adjacent relation in the left bottom corner (c) of the figure.

Definition 4.8 (Crossing relation) ***Crossing relation*** *is a relation, when different IPs share objects.*

Figure 4.17 shows three cases, where two placements conform to the definition of crossing placements. The case in the left bottom corner (c) in Figure 4.17 satisfies both definitions 4.7, 4.8.

Definition 4.9 (Contained relation) ***Contained relation*** *is a crossing relation, when all elements of one placement belong to another placement.*

Two placements in Figure 4.17 on the preceding page (d) have the contained relation, since the objects of the placement *p2*, i.e. *2* and *3* are contained in *p1*. To sum up, there are three cases where two kinds, i.e. adjacent and crossing placements; crossing placements, are hidden in an intersection of PEint. Therefore, we have found four overlapping kinds overall.

1. $\mathrm{PEint}_{1/2} \bigcap \mathrm{PEext}_{2/1} \neq \varnothing \wedge \mathrm{PEint}_{1/2} \bigcap \mathrm{PEint}_{2/1} = \varnothing \Rightarrow \mathrm{IP}_{1/2} \bigcap \mathrm{OP}_{2/1} \neq \varnothing$ - *adjacent placements;*

2. $\mathrm{PEint}_{1/2} \bigcap \mathrm{PEext}_{2/1} \neq \varnothing \wedge \mathrm{PEint}_{1/2} \bigcap \mathrm{PEint}_{2/1} \neq \varnothing \Rightarrow$ $\mathrm{IP}_{1/2} \bigcap \mathrm{IP}_{2/1} \neq \varnothing$ - *adjacent and crossing placements* or *crossing placements;*

3. $\mathrm{PEint}_{1/2} \subseteq \mathrm{PEint}_{2/1} \Rightarrow \mathrm{IP}_{1/2} \subseteq \mathrm{IP}_{2/1}$ - *contained placements.*

We intend to discuss these cases further in the thesis and tackle during a resolution process.

4.4 Adjacent placements

4.4.1 Overview

Figure 4.18 shows the placements (a) with the adjacent relation together with the possible replacements (b) and the expected result (c). We

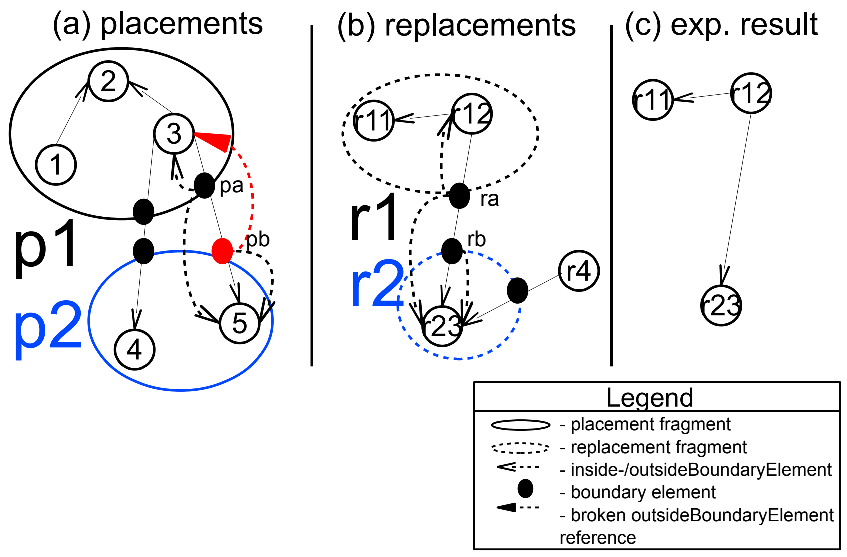


Figure 4.18: Adjacent placements and expected resolution

characterize a relation between placements of the kind adjacent by an overlap in PEext and PEint sets, where $\mathrm{PEext} \subseteq \mathrm{OP}_{1/2}$ while $\mathrm{PEint} \subseteq \mathrm{IP}_{2/1}$. Figure 4.18 also shows all boundary elements with its references. We define how the replacement fragments (b) substitute the placement fragments (a) via the binding process of the boundaries in Figure 4.18 on the preceding page. There are seven boundary elements overall, but for simplicity we just bind *pa* of the kind *FromPlacement* to *ra* of the kind *FromReplacement* and *pb* (toPlacement) to *rb* (toReplacement). The rest boundary elements are bound to NULL. It means the in a final product we do not want the presence of some references. In the example, a resolution result is

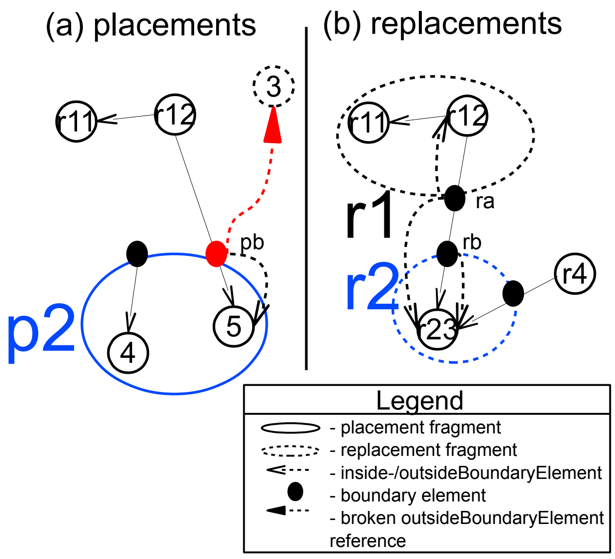


Figure 4.19: Actual result of the resolution

well understood, i.e. it conforms with our expectations and common sense. The resolution of the placements and corresponding replacements in Figure 4.18 on the facing page should result in the model that is shown in the same figure on the most right (c). However, an actual result of the MoSiS CVL tool [22] is different comparing to the expected result in Figure 4.18 on the preceding page (c), i.e. the element *r12* of the resolved product misses the link down to *r23*. The problem occurs because the base model changes during the substitution of *p1* onto *r1*. If we do not modify the variability model accordingly, then the boundary elements of *p2* may still point to the elements in *p1*. However, the elements of *p1* do not exist in the base model any more. We demonstrate the described case in Figure 4.19. Figure 4.18 on the preceding page shows that boundary *pb* of the kind *ToPlacement* has the *outsideBoundaryElement* reference (the arrow in red with the filled large head) which points to the object *3*. In addition, the boundary *pb* element defines *p2*; therefore, it does not take part in the resolution of *p1*. The resolution of *p1* comprises 1) the deletion of all elements in *p1* together with the object *3* and 2) copying of the elements of *r1* onto the placement fragment. Figure 4.19 shows the result of the *p1* resolution. Note, that the boundary element *pb* has the outside reference to the object *3*, which does not exist anymore in the model. The given configuration will lead to an incorrect resolution of the second placement *p2*. During a resolution of *p2*, the MoSiS CVL engine uses

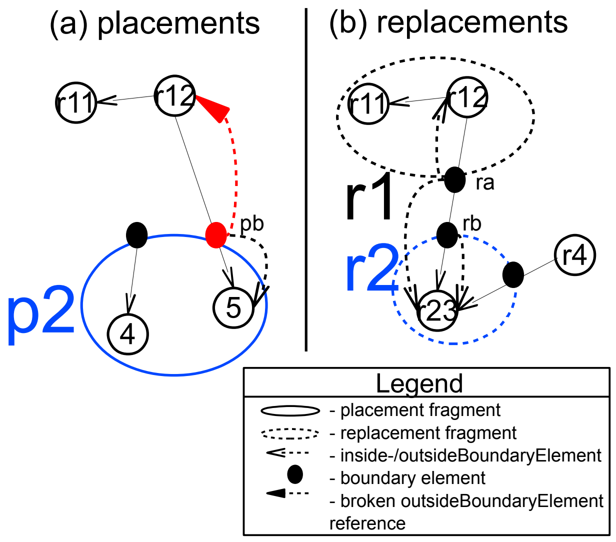


Figure 4.20: Variability model modifications

the *pb.outsideBoundaryElement* reference to reach the element (object *3*) and change its reference to point to *r23* using *rb.insideBoundaryElement*. While the placement *p2* is successfully removed, MoSiS CVL attempts to establish reference between the elements *3* and *r23*. Since the object *3* does not exist anymore, the reference is not created. But MoSiS CVL should try to create reference between *r12* and *r23* instead of *3* and *r23*. Hence, we get a model in Figure 4.18 on page 46 (c), but without the reference from *r12* down to *r23*.

4.4.2 Solution

A solution for the problem would be a modification of the variability model during an execution of MoSiS CVL. In the example, if *pb.outsideBoundaryElement* was pointing to the object *r12* then the resolution would end up with the proper result. Figure 4.20 shows required modifications. We can have this change by adding an extra step to the variability resolution process. An algorithm should check whether for a given placement and any other placement Criterion 4.1 holds. The criterion recaps the

Criterion 4.1 Adjacent placements

$\mathrm{OP}_{1/2} \bigcap \mathrm{IP}_{2/1} \neq \varnothing \wedge \mathrm{IP}_{1/2} \bigcap \mathrm{IP}_{2/1} = \varnothing$.

implication of $\mathrm{PEint}_{1/2} \bigcap \mathrm{PEext}_{2/1} \neq \varnothing \wedge \mathrm{PEint}_{1/2} \bigcap \mathrm{PEint}_{2/1} = \varnothing$ from Section 4.3.2 on page 43, but we strengthen the criterion by adding an extra condition, i.e. $\mathrm{IP}_{1/2} \bigcap \mathrm{IP}_{2/1} = \varnothing$. An overlap in IPs indicates a crossing relation between placements; thus, we rule it out. If Criterion 4.1 holds for any two placements then we can argue that the placements have the adjacent relation. Therefore, we need to find all boundaries which comprise the adjacent relation between the placements. We may do this by testing the *outsideBoundaryElement* references of the boundaries of the adjacent placements. Thus, we should test Criterion 4.2 against each binding of the adjacent placement. Given that two placements are adjacent (two place-

Criterion 4.2 Boundary elements: adjacent placements

```
Pl0,Pl1 - adjacent placements;
Boundaries - boundary elements of Pl0 and Pl0;
B - boundary element;
B.oBE - outsideBoundaryElement reference of a boundary
element;
getPEint(Pl0/Pl1) - function that returns PEint0/PEint1
```

$\mathrm{Pl}_0, \mathrm{Pl}_1, \forall$ B $\in$ Boundaries | B.oBE $\in$ getPEint($\mathrm{Pl}_0/\mathrm{Pl}_1$)

ments conform to Criterion 4.1 on the preceding page) and Criterion 4.2 for a boundary element holds, i.e. the *outsideBoundaryElement* reference of the boundary element (where the boundary element belongs to the first adjacent placement) points to an element of the second adjacent placement, then we should modify the *outsideBoundaryElement* reference of the corresponding boundary element. A boundary element, which conforms to Criterion 4.2, is an adjacent boundary, e.g. *pa* and *pb* are adjacent boundaries in Figure 4.18 on page 46 (a). We may say also that the boundary element *pa* belongs to the placement *p1* and cuts off the adjacent relation between elements *3* and *5*.

Definition 4.10 (Adjacent boundary) ***Adjacent boundary*** *is a boundary element which cuts off an adjacent relation between placement/replacement fragments.*

We say that the adjacent boundaries *pa* and *pa* constitute the adjacent pair.

Definition 4.11 (Adjacent pair) ***Adjacent pair*** *is a pair of adjacent boundaries which cut off the same adjacent relation.*

We should modify adjacent boundaries every time an adjacent placement is substituted. Thus, we keep a variability model consistent all the way through the resolution process. In the example from Figure 4.18 on page 46, *p1* represents Pl_0 while *p2* represents Pl_1. The set *Boundaries* contains all boundary elements from Pl_0 and Pl_1, i.e. *pa*, *pb* and two others without names though we do not consider them (bound to NULL). IP_0 is a set of *1, 2, 3* and OP_0 is *1, 2, 3, 4, 5*, while IP_1 equals to *4, 5*, and OP_1 contains *3, 4, 5*. $\mathrm{OP}_0 \bigcap \mathrm{IP}_1$ = *4, 5* and $\mathrm{OP}_0 \bigcap \mathrm{OP}_1 = \varnothing$; therefore, Criterion 4.1 on the facing page holds and the placements *p1* and *p2* are adjacent. Criterion 4.2 asserts *true* for both *pa* and *pb* because *pa.outsideBoundaryElement* points to *5*, which belongs to *p2*, and *pb.outsideBoundaryElement* references *3* of *p1*. Once we resolve the placement *p1* we should modify *pb.outsideBoundaryElement* such a way as *pb.outsideBoundaryElement* points to *r12*. In other words, given that the adjacent pair includes the boundaries *pa* and *pb* the following should always hold *pb.outsideBoundaryElement* == *pa.insideBoundaryElement*, where *pa.insideBoundaryElement* references *r12* after the substitution in our example.

4.5 Kinds of crossing placements

4.5.1 Crossing placements

Overview

An overlap of the kind crossing placements occurs, when same elements constitute two or more different placements. In other words, when PEint of one OP intersects PEint of another OP. Figure 4.21 shows a possible case. In the given example, two views (b, c) (each view has a placement) represent the same model (a). Thus, the model in Figure 4.21 illuminates an overlap of the kind crossing placements, i.e. the object *3* in views 1 (a) and 2 (b) is a representation of the object *3* in the model (a). Hence, we

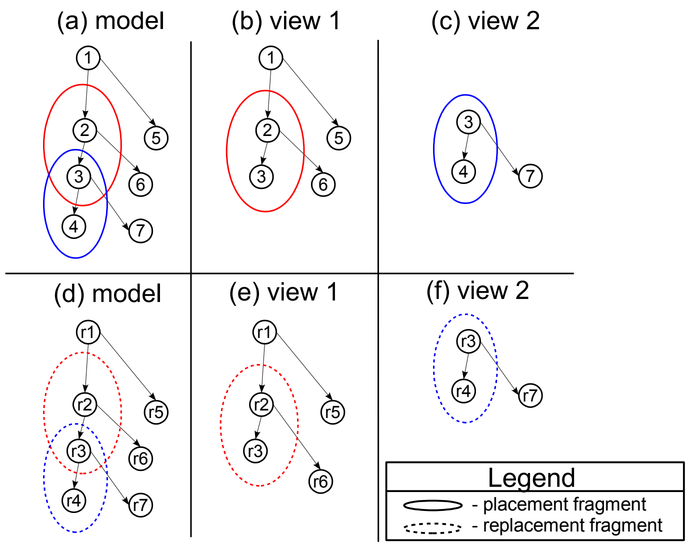


Figure 4.21: The crossing fragments

can observe that these two placements defined in the views overlap in the model. We should mention that it is also can happen to replacements, i.e their definitions can lead to an overlap of the crossing kind. Figure 4.21 shows the crossing replacements underneath the placements. An attempt to resolve these two placements one by one, will lead to broken references, an incorrect resolution process and a broken final model. Moreover, we can argue that the result depends on an order in which the placements are resolved. We would like briefly to describe possible challenges having an overlap of the kind crossing placements:

- there is a pragmatic need to define placements in a way when it causes an overlap (see our motivation example in Figure 1.1 on page 2);

- an overlap in placements can indicate that a user probably is trying to accomplish something inconsistent or illegal;
- the property of commutativity does not always hold, meaning that if we change the resolution order, the result is different;
- two placements can simultaneously have two kinds of overlaps, i.e. adjacent placements and crossing placements.

We argue that crossing placements (replacements) may be considered as a single placement (replacement). Thus, we introduce an unionization procedure as a solution for such kind of the relations between two placements. Since the overlap of the kind crossing can indicate both a pragmatic need and erroneous in a variability definition, we should be able to distinguish such cases. We believe that required information for the decision is already in a variability model. For example, in Figure 4.21 on the facing page the overlaps between placements and corresponding replacements seem to have the same look; therefore, the placement and corresponding replacements can be merged together. By testing a possibility to unionize crossing fragments, we 1) spot erroneous variability definitions and tackle cases where the unionization operation is possible, 2) reduce the overall amount of substitutions which way facilitate the derivation process and 3) widen the semantics of the placement/replacement definition which may enhance the variability specification process.

Solution

Figure 4.22 (a) shows a case, where we have the pure crossing relation between two placements. An overlap in IPs characterizes a case of the kind

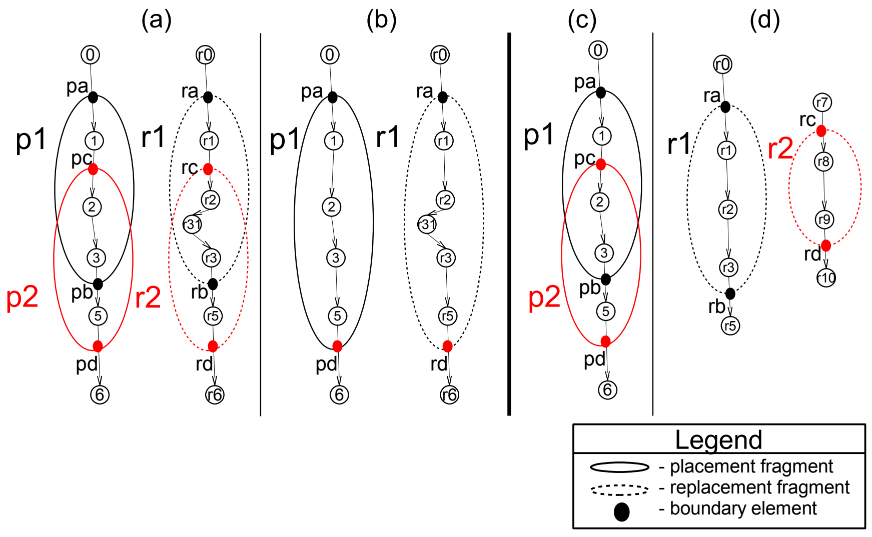


Figure 4.22: The crossing fragments and their unionizations

crossing placements, i.e.

- $IP_{1/2} \bigcap IP_{2/1} \neq \varnothing$.

We have mentioned that two different placements with the crossing relation between each other should be resolved as a single operation, meaning that we should attempt to unionize the given placements. In CVL boundary elements fully define placement and replacement fragments; therefore, we can manipulate with boundary elements in order to adjust fragments. Thus, the unionization of crossing fragments implies a deletion of some boundary elements which appear to be internal with respect to the unionized fragment. In other words, some boundary elements become meaningless because their *outsideBoundaryElement* references point to the elements inside the unionized placement. It contradicts to the definition of the *outsideBoundaryElement*, i.e. "The outsideBoundaryElement refers to the model elements on the outside of the placement fragment [12].". Criterion 4.3 should detect bindings to delete. In the example in Figure 4.22

Criterion 4.3 Boundary elements: crossing placements (initial)

```
Pl$_0$,Pl$_1$ - overlapping placements;
Pl$_{1/0}$.Boundaries - boundary elements of Pl$_{1/0}$;
B - boundary element;
B.oBE - outsideBoundaryElement reference of a boundary
element;
getPEint(Pl$_0$/Pl$_1$) - function that returns PEint$_0$/PEint$_1$
```

Pl_0, Pl_1, $\forall$ B $\in Pl_{1/0}$.Boundaries | getPEint(Pl_0) $\bigcap$ getPEint(Pl_1) $\neq \varnothing \wedge$ B.oBE $\in$ getPEint($Pl_{0/1}$)

on the previous page (a) we bind *pa* to *ra*, *pb* to *rb*, *pc* to *rc* and *pd* to *rd*. Using Criterion 4.3, we can identify that boundary elements *pb* and *pc* in Figure 4.22 on the previous page (a) ought to be removed and *rb* and *rc* accordingly. Thus, we have defined a new placement with just two boundary elements, i.e. *pa* and *pd* which are bound to *ra* and *rd* of the recently formed replacement (see the placement/replacement in Figure 4.22 on the preceding page (b)).

Let us consider another two placements and replacements in Figure 4.22 on the previous page (c, d). We bind all boundary elements as in the previous example, i.e. *pa* to *ra*, *pb* to *rb*, *pc* to *rc* and *pd* to *rd*. Criterion 4.3 asserts a removal of *pb* and *rb*, *pc* and *rc* which leaves the variability model in an inconsistent state. The replacements have dangling boundary elements (*ra* and *rd*) which do not define valid replacements. It happens because the replacements, i.e. corresponding boundary elements of the replacements (*rc*, *rb*), are not in the same relations as the placements are, meaning they do not overlap. The given configuration should indicate an error in a variability model, but should not be mixed up with the case in Figure 4.22 on the previous page (a). Thus, we need to develop a criterion to spot this situation, which can look as Criterion 4.4 on the facing page. The criterion highlights all bindings and corresponding boundary elements, which should be removed, but negative resolution of the criterion, when

Criterion 4.4 Boundary elements: crossing placements (advanced)

```
Pl$_0$,Pl$_1$ - overlapping placements;
Rl$_0$,Rl$_1$ - overlapping replacements;
Pl$_{1/0}$.Boundaries - boundary elements of Pl$_{1/0}$;
B$_{pl}$ - boundary element of a placement;
B$_{rl}$ - boundary element of a replacement;
B$_{pl/rl}$.oBE - outsideBoundaryElement reference of a boundary
element;
B$_{pl/rl}$.iBE - insideBoundaryElement reference of a boundary
element;
getPEint(Pl$_0$/Pl$_1$) - function that returns PEint$_0$/PEint$_1$
```

$\mathrm{Pl}_{0/1}, \mathrm{Pl}_{1/0}, \mathrm{Rl}_{0/1}, \mathrm{Rl}_{1/0}, \forall\, (\mathrm{B}_{pl},\mathrm{B}_{rl}) \in (\mathrm{Pl}_{1/0}.\mathrm{Boundaries},\mathrm{Rl}_{1/0}.\mathrm{Boundaries}) \mid \mathrm{B}_{pl}.\mathrm{oBE} \in \mathrm{getPEint}(\mathrm{Pl}_{0/1}) \Rightarrow \mathrm{B}_{rl}.\mathrm{iBE} \in (\mathrm{getPEint}(\mathrm{Rl}_{0/1}) \bigcap \mathrm{getPEint}(\mathrm{Rl}_{1/0})) \wedge \mathrm{B}_{rl}.\mathrm{oBE} \in \mathrm{getPEint}(\mathrm{Rl}_{0/1})$

the left side is true, indicates an error in a variability definition.

Criterion 4.4 asserts an error since the implication equals to *false* for (c) and (d) in Figure 4.22 on page 51. The overlap of Pl$_0$ (p1) and Pl$_1$ (p2) is *2,3*, both *outsideBoundaryElement* references of *pb* and *pc* point to elements in the corresponding opposite placement, i.e. *5* and *1* respectively in Figure 4.22 on page 51(c). Thus, the left hand side of the implication is *true* while the right hand side of the implication is *false* because the replacements do not overlap in Figure 4.22 on page 51(d). According to Criterion 4.4 we should remove *pc* and *rc*, *pb* and *rb* in Figure 4.22 on page 51(a) yielding (b).

4.5.2 Crossing and adjacent placements

Overview

Figure 4.23 on the next page shows an example which has the overlap of both kinds, i.e. crossing and adjacent placements. The boundary elements *pc* and *pb* define the overlap of the kind crossing placements while *pe* and *pf* constitute the adjacent case. The recently defined Criterion 4.4 for resolving crossing fragments assert an error for boundary elements *pe* (bound to *re*) and *pf* (bound to *rf*). It happens because the *insideBoundaryElement* references of *re* and *rf* do not point to any elements inside the overlap (*r2*, *r31*, *r3*). It deems the model in Figure 4.23 on the next page erroneous. We know also from Section 4.4 on page 46, that an overlap of the kind adjacent placements can be resolved, i.e a removal of *pe* and *pf* is not required. However, we can not use the approach to resolve the adjacent relation in the crossing placements case, because Criterion 4.1 on page 48 for adjacent placements does not hold due to the overlap of the placements. Thus, we can not apply either of the proposed criteria for the composition of the overlapping kinds. At the same time, we argue that the unionization should be still feasible in this cases.

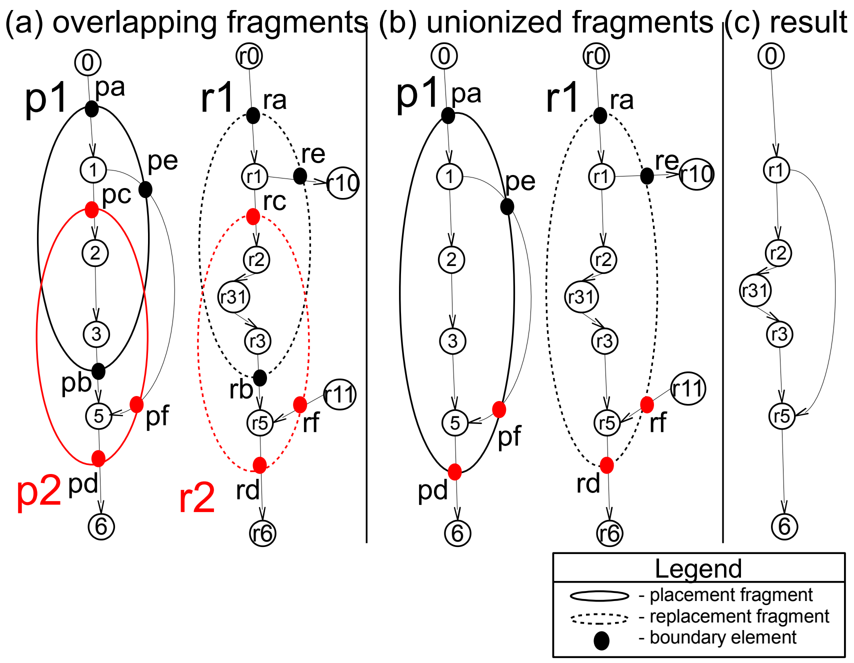


Figure 4.23: The crossing and adjacent placements, unionization and resolution

Solution

We need to develop a criterion which do not raise an error or instruct to remove *pe*, *pf*. There is a fundamental difference between *pc* and *pf*, which we can exploit. The reference *insideBoundaryElement* of *pc* points to element 2 which belongs to the intersection of *p1* and *p2*. While the *insideBoundaryElement* of *pf* references element 5 which is just internal w.r.t. *p2*. Thus, the criterion for the crossing and adjacent case should look as Criterion 4.5 on the facing page. The criterion is a modified version of the criterion from Section 4.5.1 on page 50, but we check that the *insideBoundaryElement* reference points to an element from the intersection of fragments. An algorithm should remove all bindings, which comply with the criterion. This should result in a brand new placement and replacement as in Figure 4.23 (b), which should contain elements of the initial fragments. The reference in Figure 4.23 (b) (which creates the adjacent relation between *p1* and *p2*), is just internal w.r.t. the recently formed placement. Hence, the adjacent relation is removed and the current model should not result in any inconsistency during a resolution, which we describe in Section 4.4 on page 46. The model on the most right (c) is the fully resolved model, see Figure 4.23. We should also note that an overlap of both kinds (crossing and adjacent) is a generalization of the pure crossing case; thus, Criterion 4.5 on the facing page is applicable for an overlap of the kind crossing placements.

Criterion 4.5 Boundary elements: adjacent and crossing placements

```
Pl0,Pl1 - overlapping placements;
Rl0,Rl1 - overlapping replacements;
Pl1/0.Boundaries - boundary elements of Pl1/0;
Bpl - boundary element of a placement;
Brl - boundary element of a replacement;
Bpl/rl.oBE - outsideBoundaryElement reference of a boundary
element;
Bpl/rl.iBE - insideBoundaryElement reference of a boundary
element;
getPEint(Pl0/Pl1) - function that returns PEint0/PEint1
```

$\mathrm{Pl}_{0/1}$, $\mathrm{Pl}_{1/0}$, $\mathrm{Rl}_{0/1}$, $\mathrm{Rl}_{1/0}$, $\forall\,(\mathrm{B}_{pl},\mathrm{B}_{rl}) \in (\mathrm{Pl}_{1/0}.\mathrm{Boundaries},\mathrm{Rl}_{1/0}.\mathrm{Boundaries})$ $\mid \mathrm{B}_{pl}.\mathrm{oBE} \in \mathrm{getPEint}(\mathrm{Pl}_{0/1}) \wedge \mathrm{B}_{pl}.\mathrm{iBE} \in (\mathrm{getPEint}(\mathrm{Pl}_{0/1}) \bigcap \mathrm{getPEint}(\mathrm{Pl}_{1/0}))$ $\Rightarrow \mathrm{B}_{rl}.\mathrm{iBE} \in (\mathrm{getPEint}(\mathrm{Rl}_{0/1}) \bigcap \mathrm{getPEint}(\mathrm{Rl}_{1/0})) \wedge \mathrm{B}_{rl}.\mathrm{oBE} \in \mathrm{getPEint}(\mathrm{Rl}_{0/1})$

4.5.3 Contained placements

Overview

Figure 4.24 on the next page (a) shows the possible replacement fragments with their boundary elements and references together with the placements. Two placements in Figure 4.24 on the following page (a) are in the contained relation. In order to spot two contained placements, we need to check that all elements of one placement are elements of another one, i.e. Criterion 4.6. We expect to get a model in Figure 4.24 on the following page

Criterion 4.6 Criterion - Containd placements

$\mathrm{IP}_{1/2} \subseteq \mathrm{IP}_{2/1}$

(b) after a resolution of both placement fragments. In the given example, we bind *p2a* to *r2a*, *p2b* to *r2b* and *p1a* to *r1a* correspondingly. Other boundary elements are bound to NULL for the sake of simplicity. We are trying to resolve *p2* and then *p1*. Figure 4.24 on the next page (c) highlights a broken reference once the *p2* placement fragment is resolved. Even though we have the broken reference, where the *insideBoundaryElement* of *p1a* points to element *3* (removed as a part of the *p2* resolution), it does not break the resolution of *p1*. The reference *insideBoundaryElement* of *p1a* does not take part in the resolution of *p1*. When we perform the resolution of *p1*, the *outsideBoundaryElement* reference of *p1a* is used to reach the element *5* and change its reference to *r5*. Thus, the binding process succeeds, but at the same time, the object *r2* (constitutes *p2* after the substitution) is not reachable through the *insideBoundaryElement* reference of *p1a*. It points to the element *3* which does not exist in the model anymore. However, in the given example there are other references which we may exploit to remove all elements of the placement *p1*, but by all means, we should adjust the reference to point to the correct element. Thus, we can use a similar

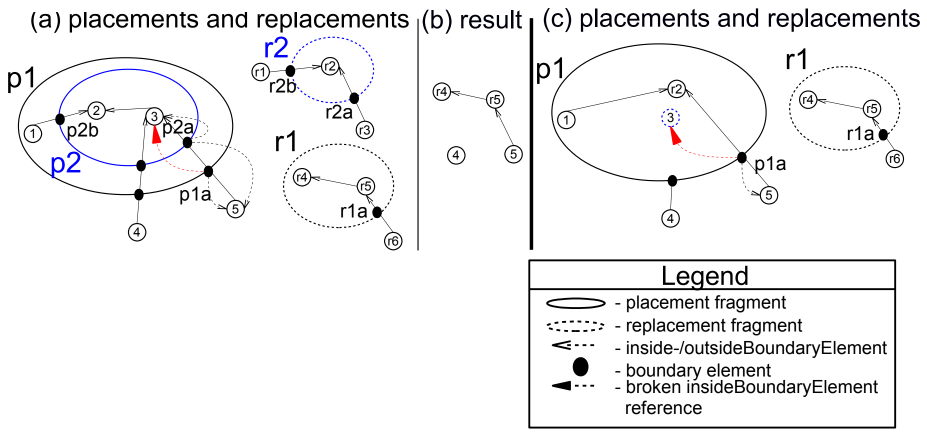

(a) placements and replacements
(b) result
(c) placements and replacements
p1
p2
r2
r1
p2b
p2a
p1a
r2b
r2a
r1a
Legend
- placement fragment
- replacement fragment
- inside-/outsideBoundaryElement
- boundary element
- broken insideBoundaryElement reference

Figure 4.24: Contained placements

algorithm from Section 4.4 on page 46. We show in Figure 4.24 (b) the final model. If we perform the resolution other way around (*p1* and then *p2*), we end up with the same final model. However, the resolution of *p2* is not executed since *p2* is removed together with the elements of *p1*.

Solution

We can conclude the following for fragments with the contained relation:

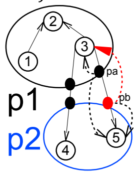
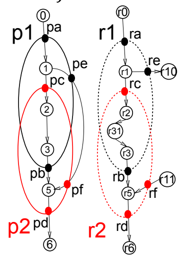
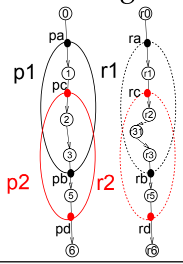
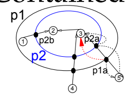
- if a resolution process begins with a containing placement then a contained one is meaningless;
- if a resolution process begins with a contained placement then we may have broken *insideBoundaryElement* references of a containing placement;
- the order of resolution process does not affect a final model.

Even though the MoSiS CVL tool implementation can tackle the contained placement case, we believe that it is a erroneous situation due to the ambiguity and pragmatical useless of the set up. Moreover, if we test Criterion 4.5 on the preceding page against the boundary elements of the contained placements, it asserts an error for the cases, even than replacements are in the crossing relation too.

4.6 Summary: overlapping placements

Table 4.2 on the next page summarizes all our findings in the previous sections.

Table 4.2: The overlapping kinds - overview

Overlapping kind	**Description**	**Suggested solution**
Adjacent 	Two placements have the adjacent relation if the placement do not share any common elements. i.e. $OP_{1/2} \bigcap IP_{2/1} \neq \varnothing \land IP_{1/2} \bigcap IP_{2/1} = \varnothing$	We should adjust boundaries which conform to Criterion 4.2 on page 49 - the criterion finds all boundaries where the *outsideBoundaryElement* reference points to any element of the another placement.
Crossing and adjacent  p1 p2 pa pb pc pd pe pf 0 1 2 3 5 6 r1 r2 ra rb rc rd re rf r0 r1 r2 r3 r5 r6 r10 r11 r31	The overlap of this kind is a generalization of the crossing relation, an intersection of IPs implies the overlap of this kind. $IP_{1/2} \bigcap IP_{2/1} \neq \varnothing$	We should try to unionize the placements. Replacements should have the crossing relation too to unionize placements, otherwise it is a failure in a variability model. Boundaries which conform to Criterion 4.5 on page 55 should be removed. If the criterion resolves to *false* when the left-hand side is *true* then it indicates a failure in the variability model.
Crossing  p1 p2 pa pb pc pd 0 1 2 3 5 6 r1 r2 ra rb rc rd r0 r1 r2 r3 r5 r6 r31	The crossing relation is a special case of the crossing and adjacent relation, thus an intersection of IPs indicates the overlap of this kind.	The crossing relation between two placements should be resolved through the unionization procedure. We use the same criterion and approach to tackle such placements as for the crossing and adjacent relation.
Contained 	The overlap of the kind happens when one placement is contained by another.	We consider this situation as a failure in a variability model.

From Table 4.2 one may notice that we apply the unionization procedure for crossing fragments while we adjust a broken reference for adjacent placements. We may notice also that in case of the crossing and adjacent relation, the unionization process yields a new placement; therefore, it removes the adjacent relation between two initial placements. One may argue that it is a solution for a pure adjacent relation between two placements. However, the unionization procedure requires that the corresponding replacements should have the same relation as the given placements. If the requirement does not hold we assert a failure in a variability model. Hence, the unionization limits a set of potentially valid variability definitions. Therefore, it is not an efficient way to unionize purely adjacent placements. In addition, we have shown that adjacent fragments can be handled by adjusting broken references. We do not have to take into account a relation between replacements.

Chapter 5

Implementation: CVL fragment substitution engine

5.1 Design goals and requirements

We set the following functional, non-functional and implementation requirements on the substitution fragment engine outlined as follows:

1. the substitution engine should take as a starting point the OMG CVL metamodel;
2. the substitution engine should utilize exception/logging capabilities to give a feedback on operations;
3. the substitution engine should detect fragments which are in the adjacent relation;
4. the substitution engine should be able to resolve the adjacent relation between fragments;
5. the substitution engine should be testable;
6. the substitution engine should be built on Eclipse Modeling Framework;
7. the substitution engine should utilize means and technologies to build and manage the engine;
8. the substitution engine should utilize development patterns and frameworks to achieve the goal of scalability, configurability;

The substitution engine is a part of the OMG CVL tool which SINTEF is currently developing. Thus, the substitution engine should fits in the OMG CVL infrastructure. Therefore, we implement the engine as a library that provides API and reports back on a substitution. The engine utilizes the following part of the OMG CVL metamodel, see Figure 5.1 on the next page.

The current version of the engine implements fragment substitution, detection and resolution of the adjacent relation. However, the suggested

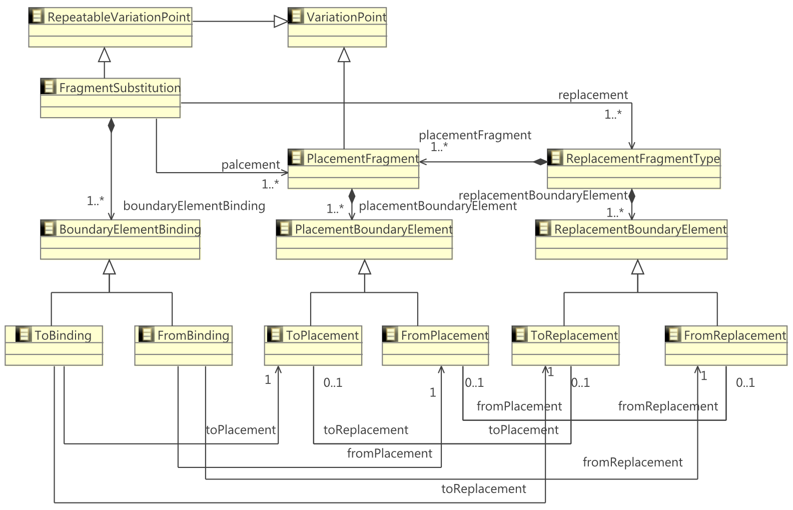


Figure 5.1: The fragment substitution matamodel

design supports the further enlargement of the substitution engine to implement the unionization approach to tackle the crossing relation between fragments.

We implement two basic methods to provide a feedback on an execution of the engine. Firstly, we integrate logging capabilities for the engine. Secondly, we introduce exceptions and an exception handling to respond on an incorrect execution of the engine. The combination of both these approaches improves the system response and facilitates a failure investigation.

We test the functionality of the substitution engine applying the unit testing approach. With help of the Node DSL and the elaborated approach we are able to build a moderate amount of small variability models. Thus, it enables creating test cases which are tailored to a specific functionality of the engine. We ensure also that a further development does not break or affect already created functionality.

The substitution engines uses Eclipse Modeling Framework (EMF) to implement substitution. EMF includes ECore is a mean for metamodeling where ECore is an EMF implementation of the EMOF standard issued by OMG. EMF provides a reach set of API which allows manipulating on models. Therefore, the engine is able to operate on any instance of a metamodel defined in ECore. It serves the main goal of the engine, i.e. substitute different fragments of a model.

We try to stick to modern, well supported frameworks and patterns which should help to achieve the goal of scalability and further support. We distinguish two kinds of frameworks: project management frameworks and development frameworks. A project management framework should enable efficient handling of possible third-party dependencies between different components of the engine, facilitate the compilation and building processes with further integration. While development frameworks should help to apply design patterns to the engine. We use Maven and Git to manage and build the engine and the Spring development framework to introduce design patterns, scalability and configurability.

5.2 Technologies

5.2.1 Eclipse Modeling Framework

Eclipse Modeling Framework (EMF) is an open-source Eclipse based framework that provides modeling and code-generation facilities. EMF includes the ECore metamodeling functionality which is an implementation of the OMG EMOF standard (a variation of the MOF specification). In addition, EMOF gives a straightforward framework for mapping MOF models to implementations such as XMI and uses UML 2.0 constrained class diagrams to define metamodels [42].

The OMG CVL metamodel is built using EMOF in UML 2.0. Thus, we are able to convert the OMG CVL metamodel to the ECore format and generate the partially implemented metamodel using the EMF code generating facilities. EMF framework includes continuously supported and well documented APIs as well as facilities to serialize models to the XMI format. Hence, EMF provides the off-the-shelf functionality enabling rapid development of the engine and can serve our main goal, namely the model manipulation goal.

5.2.2 Programming language

EMF is based on JVM platform and written in Java. Thus, we have decided to use Java as a base language for the OMG CVL substitution engine. EMF does not leave room for other possibilities; meanwhile, one may argue that the languages such Scala or MOFScript are available alternatives. MOFScript is a model to text transformation language based on EMF; thus, this language can be used for the engine. However, MOFScript is not currently supported by any vendor and the lack of tooling makes the development challenging. Scala is a general purpose language based on JVM which is actively developing now. Thus, one may use a full set of frameworks and libraries written in Java including EMF. However, Scala provides its own typing system that is compatible with Java, but anyway a function call of a Java library requires some type conversions. Thus, we may come across potential problems during implementation. In addition, we do not set a goal to utilize any functional programming capabilities and integrated design patterns of Scala. Therefore, we prefer using the native language of the EMF implementation to operate with API.

5.2.3 Maven

Maven is a project management tool which supports building, testing, reporting and documentation compilation from a single source. We use maven in the project for the dependency management, compilation and test execution. Even relatively small projects can have many dependencies to third party libraries. The substitution engine has over twenty third party dependencies. Maven automates the process of downloading required

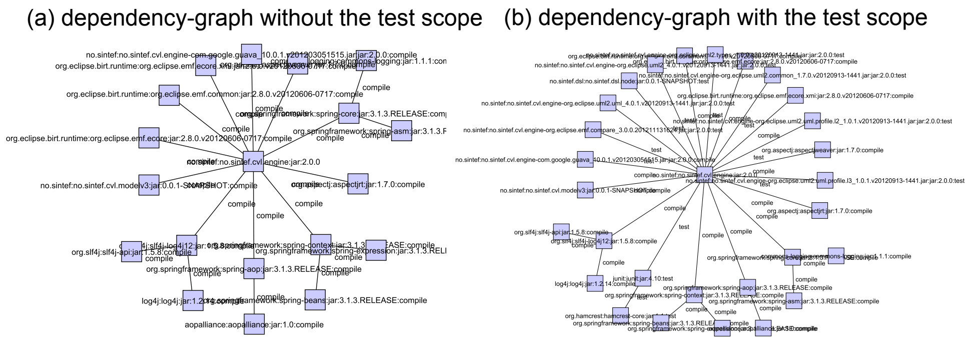

(a) dependency-graph without the test scope
(b) dependency-graph with the test scope

Figure 5.2: Maven dependency graphs (different scopes)

libraries, resolving of their dependencies and setting class paths for a project, where an engineer just needs to specify a library. In addition, Maven supports dependency scopes. We can specify a different set of libraries for different lifecycles of an application, e.g. compilation, testing, runtime. Figure 5.2 shows two dependency graphs for the substitution engine. The graph (a) is a dependency graph without the test scope, while the graph on the right (b) includes the test scope. One may notice that the graph on the left (a) has fewer nodes than the graph on the right (b). Each node is an external library where the node in the center is the substitution engine. The test scope includes libraries which we use only to test the engine and do not use during a normal execution of the engine. In other words, Maven allows excluding libraries in the final build which are not required for a normal execution. It would be problematical and not efficient to manage such dependencies manually. We also utilize the maven capabilities to run test cases and organize the failure reporting.

5.2.4 Spring

The Spring framework is an open source development framework build on the Java platform. The framework is scalable and can be used for a simple Java application as well as for developing enterprise applications. The Spring allows a straightforward way to integrate following popular design patterns, e.g. Inversion of Control (IoC) or Dependency injection (DI), Aspect-Oriented programming (AOP), Model View Control (MVC) etc.

Neither of these patterns are currently used by the substitution engine. However, we utilize the bean factory of the Spring framework to integrate logging functionality. For example, we may change easy a logging handler without changing the sources of the substitution engine using a configuration file of the Spring framework.

5.2.5 JUnit

JUnit is a testing framework for Java applications. We use JUnit to incorporate testing capabilities in the substitution engine. The framework introduces a test environment which includes advanced features to run test cases and analyze failures. It introduces also the test-driven development approach. The framework can be integrated with the Spring and Maven to ensure more efficient quality assurance of the substitution engine. For our substitution engine we have developed around 60 test cases.

5.3 Fragment substitution engine

5.3.1 Engine architecture

The overall project architecture of the engine is aligned to a simple Maven project. Therefore, one may find a typical folder structure:

- **src/main/jave** - source code folder of the engine;
- **src/main/resources** - non-code artifacts, which are related to the source, reside in this folder;
- **src/test/java** - test case folder of the engine;
- **src/test/resources** - non-code artifacts for test cases.

The suggested structure helps to stick to the dry principle, i.e. we do not mix together the engine and test-case sources. In addition, we separate the code and non-code artifacts. It is especially important due to the amount of test cases and their samples.

We try to support modularity of the engine. Thus, we put the code, which is responsible for different functional parts of the engine, in various packages. One may found the following packages in Figure 5.3. We do

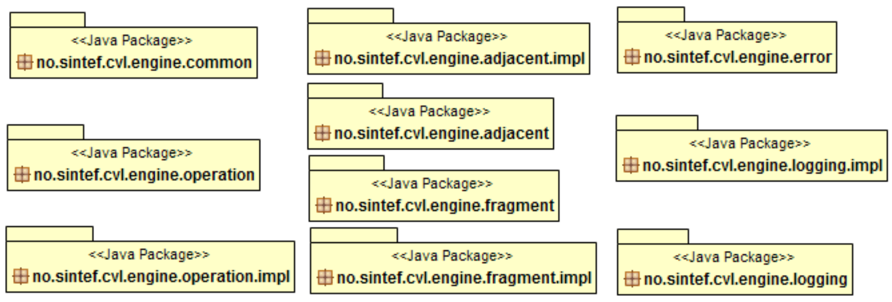

<<Java Package>>
no.sintef.cvl.engine.common
<<Java Package>>
no.sintef.cvl.engine.adjacent.impl
<<Java Package>>
no.sintef.cvl.engine.error
<<Java Package>>
no.sintef.cvl.engine.adjacent
<<Java Package>>
no.sintef.cvl.engine.operation
<<Java Package>>
no.sintef.cvl.engine.logging.impl
<<Java Package>>
no.sintef.cvl.engine.fragment
<<Java Package>>
no.sintef.cvl.engine.operation.impl
<<Java Package>>
no.sintef.cvl.engine.fragment.impl
<<Java Package>>
no.sintef.cvl.engine.logging

Figure 5.3: Substitution engine packages

not show dependencies and relationships between packages in order to keep the figure clean and readable. Table 5.1 on the following page lists all packages with short descriptions of the functionality encapsulated in a package.

Table 5.1: The packages and their descriptions

Name	Description
no.sintef.cvl.engine.adj-acent	The package contains interfaces outlining the functionality for finding and resolution of the adjacent relation between fragments. One may find the following interfaces Ad-jacentFinder, AdjacentFragment and Adja-centResolver.
no.sintef.cvl.engine.adj-acent.impl	The package contains implementation of the interfaces in no.sintef.cvl.engine.adjacent.
no.sintef.cvl.engine.co-mmon	The package contains common utilities for the substitution engine. We can list the following classes: • CVLFragmentCopier is an extension of the EcoreUtil.Copier. The class encap-sulates means to copy model elements. We use the copier to create copies of a replacement fragment. • SubstitutionContext is a context for the substitution engine, implements the singletone pattern and contains currently just a logging handler and creates the application context of the Spring framework. • The Utility class encapsulates small common utilities which are used across the substitution engine.
no.sintef.cvl.engine.err-or	The package contains all exceptions which the substitution engine throws.
no.sintef.cvl.engine.fra-gment	The package includes interfaces outlining the concepts of IP, OP, PEint and PEext. The in-terfaces describe holders which are wrappers for the basic OMG CVL concepts, e.g. place-ment, replacement, fragment substitution.
no.sintef.cvl.engine.fra-gment.impl	The package provides the imple-mentations for the interfaces in no.sintef.cvl.engine.fragment
no.sintef.cvl.engine.lo-gging	The package contains a basic logging inter-face.
no.sintef.cvl.engine.lo-gging.impl	The package contains implementations of the interface in no.sintef.cvl.engine.logging (de-fining how and where to output messages).

Continued on the next page

Table 5.1 – *Continued from the previous page*

Name	Description
no.sintef.cvl.engine.operation	The interfaces in the package outline the basic operations which the engine may perform. For example, it defines the Operation, Substitution interfaces.
no.sintef.cvl.engine.operation.impl	In the package, we implement the fragment substitution operation. The fragment substitution is the only and core operation available currently for the engine which substitutes a placement fragment with a replacement.

5.3.2 Fragment substitution module

Architecture overview

The class *FragmentSubOperation* implements a basic and core operation of the engine, namely the substitution operation. It removes elements of a placement fragment, copies a replacement fragment and insert elements of the replacement into the placement fragment. Figure 5.4 shows the *FragmentSubOperation* class and its main associations to another functional elements of the substitution engine. The figure shows that

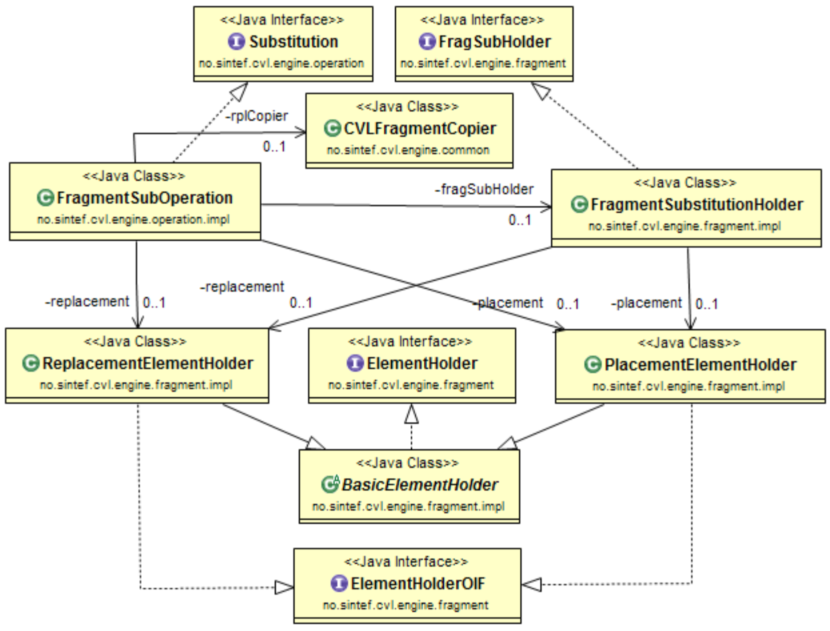


Figure 5.4: The fragment substitution operation - class diagram

the class *FragmentSubOperation* implements the Substitution interface which has a single public method **execute(boolean replace)** that runs the substitution operation. In order to perform a substitution, one should create an object of the type *FragmentSubOperation* and call the method *execute*, where the *replace* parameter defines whether we want to replace (replace = true) or augment (replace = false) the current placement fragment with the replacement fragment. *FragmentSubOperation* has associations to classes *FragmentSubstitutionHolder*, *PlacementElementHolder*, *ReplacementElementHolder* and *CVLFragmentCopier*.

FragmentSubstitutionHolder is a wrapper for the *FragmentSubstitution* class defined by the OMG CVL specification, see Figure 5.1 on page 60. The class is created in response to further needs to modify a substitution fragment. For example, the unionization procedure involves deletion of some bindings. In addition, the wrapper performs some pre-calculations which we use across the engine, e.g. a list of To-/FromBinding elements. It also keeps some additional information along an execution of the engine. For example, we need to store some references when we augment a placement fragment since the *insideBoundaryElement* reference of a boundary element may point to several elements in this case, but it is not possible according to the OMG CVL metamodel.

PlacementElementHolder and *ReplacementElementHolder* are wrappers for *PlacementFragment* and *ReplacementFragmentType* respectively, see Figure 5.1 on page 60. Both classes implement the same interfaces, namely *ElementHolder* and *ElementHolderOIF* which utilize the notions of PEint, PEext and IP, OP correspondingly and provide methods to calculate elements. Thus, we are able to implement the criteria to detect different relations between placement and replacement fragments.

Substitution operation

The implementation of the fragment substitution in *FragmentSubOperation* executes the operation in three steps:

1. copying - make a copy of the replacement fragment;
2. instrumenting - set references of the placement neighboring elements to the neighboring elements of the copied replacement.
3. deletion - remove all elements of the placement fragment (when we want to replace a placement fragment).

The class *CVLFragmentCopier* implements the copying capabilities of of the engine and extends default EMF means to copy elements of a model. Thus, the copier functionality allows creating copies of a replacement fragment achieving the first goal of the substitution operation, i.e. *copying*. We reach the second goal (*instrumenting*) via a loop over to/from boundary elements of a placement and corresponding replacement. Boundaries define neighboring elements of placement and replacement fragments. Thus, we can adjust corresponding references of the placement neighboring elements such as they point to elements of the copied replacement. We do not have to remove any elements of the placement explicitly, but we achieve the third step (*deletion*) by manipulating containment relations, i.e. elements, which are not contained by any element, are removed automatically.

5.3.3 Adjacent resolution module

Architecture overview

Figure 5.5 shows an architecture of the adjacent resolution module. The figure clearly shows that the adjacent resolution module includes three main classes, i.e. *AdjacentFragmentImpl*, *AdjacentFinderImpl* and *AdjacentResolverImpl* which implement respective interfaces. *AdjacentFragmentImpl* holds

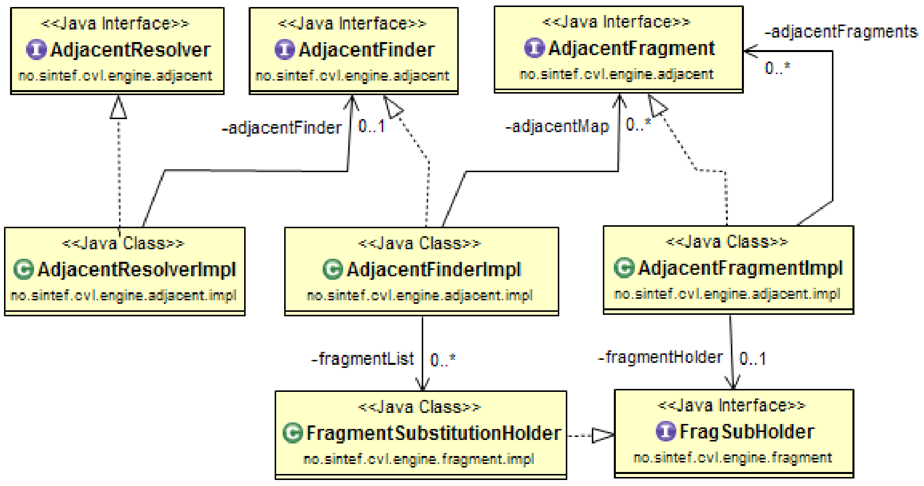


Figure 5.5: Adjacent resolution module - class diagram

fragments which are adjacent to a given substitution fragment. It encapsulates setter and getter methods to retrieve all required information (related to a particular fragment) for a further resolution. For example, one can find methods to set/retrieve adjacent fragments and their adjacent boundaries. *AdjacentFinderImpl* implements a logic to find the adjacent relation between fragments, wraps them into *AdjacentFragmentImpl* and calculates all required information for a further resolution. While the class *AdjacentResolverImpl* implements the adjacent resolution process.

Adjacent finder

AdjacentFinderImpl implements Criterion 4.1 on page 48 and Criterion 4.2 on page 49. The adjacent finder receives a list of substitution fragments to be processed and roughly speaking test each fragment in the list against each other to find out the adjacent relation. The function *isAdjacent* implements Criterion 4.1 on page 48; thus, we spot an existence of the adjacent relation between fragments. Listing 5.3-1 gives a pseudo code of the implementation for the *isAdjacent* method.

Listing 5.3-1: Pseudo code: implementation of Criterion 4.1 on page 48

```
private boolean isAdjacent(plcmnt, plcmnt1){
 HashSet<EObject> iP = plcmnt.getIPElmnts();
 HashSet<EObject> iP1 = plcmnt1.getIPElmnts();
 //intrs(iP, iP1) - intersection
 //isE − isEmpty()
 if(intrs(iP, iP1).isE){
  HashSet<EObject> oP = plcmnt.getOPElmnts();
  HashSet<EObject> oP1 = plcmnt1.getOPElmnts();
  if(!intrs(iP, oP1).isE || !intrs(iP1, oP).isE){
   return true;
  }
 }
 return false;
}
```

The function checks if IPs of two placements do not overlap, while corresponding IP and OP do overlap. We wrap adjacent fragments in *AdjacentFinderImpl* and create a map of fragments which are adjacent to a given one. The map is implemented as a simple list that stores adjacent placements. Further, we need to find boundaries which have to be modified. We call such boundaries adjacent, i.e. their *outsideBoundatyElement* references point to elements of adjacent placements. Listing 5.3-2 gives a pseudo code for the function which implements Criterion 4.2 on page 49.

Listing 5.3-2: Pseudo code: implementation of Criterion 4.2 on page 49

```
private HashMap getAdjacentBindings(fHolder, fHolder1){
 boundariesMap = new HashMap<FromBinding, ToBinding>();
 fromBindings = fHolder.getFromBinding();
 for(FromBinding fromBinding : fromBindings){
  fromPlacement = fromBinding.getFromPlacement();
  //iBElmnt − inside neighboring element
  //oBElmntS − outside neighboring elements
  iBElmnt = fromPlacement.getIBElmnt();
  oBElmntS = fromPlacement.getOBElmnt();
  toBindings = fHolder1.getToBindings();
  for(ToBinding toBinding : toBindings){
   toPlacement = toBinding.getToPlacement();
   //oBElmnt − outside neighboring element
   //iBElmntS − inside neighboring elements
   oBElmnt = toPlacement.getOBElmnt();
   iBElmntS = toPlacement.getIBElmnt();

   //diff() − symmetric difference
   //isE − isEmpty()
   if(iBElmnt==oBElmnt && diff(oBElmntS,iBElmntS).isE){
    boundariesMap.put(fromBinding, toBinding);
   }
  }
 }
 return boundariesMap;
}
```

The *if* statement of the second loop checks whether the *outsideBoundaryElement* reference of a boundary element points to an element of the adjacent placement by testing the first condition in the statement (*iBElmnt==oBElmnt*). At the same time, the second condition of the statement (*diff(oBElmntS,iBElmntS).isE*) tests whether two boundary elements cut the same references (such boundaries constitute the adjacent pair). Thus, we are able to adjust broken references later, because the following should always hold, i.e. the *insideBoundaryElement* and *outsideBoundaryElement* references of two adjacent boundaries in the adjacent pair should point to the same element.

Adjacent resolver

AdjacentResolverImpl encapsulates the logic of adjusting broken references in a variability model. Listing 5.3-3 shows a pseudo code of the implementation for the function that adjusts broken references.

Listing 5.3-3: Pseudo code: adjacent resolver

```
public void resolve(fragmentH) {
 //aFinder - adjacent finder
 aFrag = this.aFinder.getAdjacentMap().get(fragmentH);
 adjacentFragments = aFrag.getAdjacentFragments();

 for(AdjacentFragment aF : adjacentFragments){
  fragHolderAdjacent = aF.getFragmentHolder();

  //aBs - adjacent bindings
  //rev - reverse a map
  aBs = rev(aFrag.getAdjacentToBindings(aF));

  for(Map.Entry entry : aBs.entrySet()){
   //frB - from Binding
   //toB - to Binding
   frB = entry.getKey();
   toB = entry.getValue();

   //adjust broken outsideBoundaryElement refs
   //getIBElmnt - get inside neighboring element
   //getOBElmnt - get outside neighboring element
   iBElmts = toB.toPlacement().getIBElmnt();
   frB.getFromPlacement().getOBElmnt().clear();
   frB.getFromPlacement().getOBElmnt().add(iBElmts);
  }

  aBs = aFrag.getAdjacentFromBindings(aF);
  for(Map.Entry entry : aBs.entrySet()){
   //frB - from Binding
   //toB - to Binding
```

```
    frB = entry.getKey();
    toB = entry.getValue();

    //adjust broken outsideBoundaryElement refs
    //iBElmnts - inside neighboring element
    //oBElmnts - outside neighboring element
    iBElmnts = frB.getFromPlacement().getIBElmnt();
    oBElmnts = toB.getToPlacement().getOBElmnt();
    oBElmnts.clear();
    oBElmnts.add(iBElmnts);
   }
  }
}
```

In order to fix broken references, the algorithm has three loops, i.e. one outer loop and two inner loops. The outer loop iterates over adjacent fragments. Subsequently, two inner loops iterate over adjacent bindings. Firstly, we go over *fromBindings*, which are adjacent to *toBindings* of a current substitution fragment, and adjust *outsideBoundaryElement* references of *fromPlacement* boundaries. We adjust references in such a way as the *outsideBoundaryElement* reference of a *fromPlacement* boundary equals to the *insideBoundaryElement* reference of a corresponding adjacent *toPlacement* boundary. Secondly, we iterate over *toBindings*, which are adjacent to *fromBindings* of a current substitution fragment, adjusting *outsideBoundaryElement* references.

Listing 5.3-4 gives a pseudo code of the required operations to perform several substitutions. Firstly, we need two retrieve fragments from a variability model, wrap them in *FragmentSubstitutionHolder* and create a list of fragments. In the example, we have just two fragments. Secondly, we need to initialize an adjacent finder (it finds out whether there are any adjacent fragments in the list) and create an adjacent resolver. Finally, we iterate over the list with fragments and perform substitutions. Note, that each call of *execute* is followed by the *resolve* call of the adjacent resolver which adjusts references (if any). Therefore, we ensure that the variability model is consistent after a single substitution step.

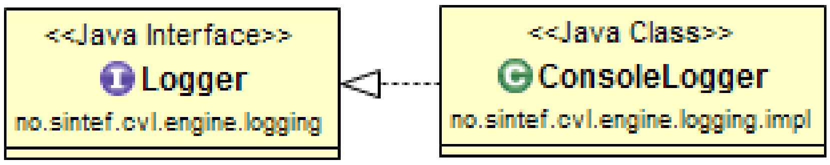
Listing 5.3-4: Pseudo code: two substitutions

```
...
//fragSub<N> - fragment from a variability model
fSH1 = new FragmentSubstitutionHolder(fragSub1);
fSH2 = new FragmentSubstitutionHolder(fragSub2);

fSHL = new BasicEList<FragmentSubstitutionHolder>();
fSHL.add(fSH1);
fSHL.add(fSH2);

aFinder = new AdjacentFinderImpl(fSHL);
adjacentResolver = new AdjacentResolverImpl(aFinder);
```

```
for(FragmentSubstitutionHolder fs: fSHL){
  fso = new FragmentSubOperation(fs);
  fso.execute(true);
  adjacentResolver.resolve(fs);
}
...
```

5.3.4 Logging

The engine implements the logging functionality using the Spring development framework. We utilize the bean factory of the Spring application context. Figure 5.6 shows that there is a single implementation of the *Logger* interface in the engine currently. *ConsoleLogger* is a simple logging handler



<<Java Interface>>
Logger
no.sintef.cvl.engine.logging
<<Java Class>>
ConsoleLogger
no.sintef.cvl.engine.logging.impl

Figure 5.6: Logging - class diagram

that prints out all messages in the standard output. We wrap the logger into a spring bean. Listing 5.3-5 shows a spring configuration file where we define the *defaultLogger* bean. Further, we can retrieve the logging bean from the spring application context using its *id*.

Listing 5.3-5: The simplified configuration file

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="..."
       xmlns:xsi="..."
       xmlns:aop="..."
       xsi:schemaLocation="...␣...">

 <bean id="defaultLogger"
  class="...logging.impl.ConsoleLogger"/>
  ...
</beans>
```

Listing 5.3-6 gives an example how a logging bean can be retrieved from the spring bean factory.

Listing 5.3-6: The logging bean

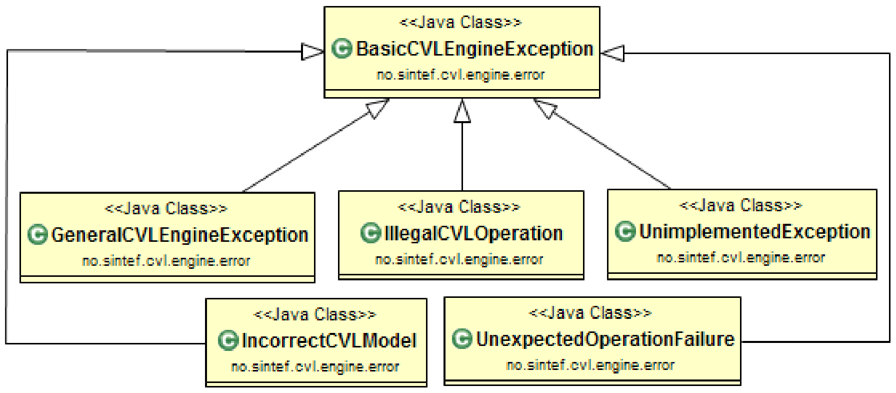
```
String configLocation = "META-INF/beans.xml";
...
context = new ClassPathXmlApplicationContext(
 new String[] {configLocation});
```

```
...
public Logger getLogger(){
 return (Logger) context.getBean("defaultLogger");
}
```

We store the logging bean in the application context which is available for any module of the engine. The application context of the engine implements the singletone pattern; therefore, it instantiates only once and stores required information for the engine. Thus, we can retrieve the logger handler on demand. Moreover, if we implement a new logger for the engine, we do not have to change any source of the application. We simply need to adjust the *bean.xml* file and recompile the engine.

5.3.5 Errors

Figure 5.7 depicts the architecture of all exceptions which the engine throws. Table 5.2 lists all exceptions and their purposes.



<<Java Class>>
BasicCVLEngineException
no.sintef.cvl.engine.error
<<Java Class>>
GeneralCVLEngineException
no.sintef.cvl.engine.error
<<Java Class>>
IllegalCVLOperation
no.sintef.cvl.engine.error
<<Java Class>>
UnimplementedException
no.sintef.cvl.engine.error
<<Java Class>>
IncorrectCVLModel
no.sintef.cvl.engine.error
<<Java Class>>
UnexpectedOperationFailure
no.sintef.cvl.engine.error

Figure 5.7: Thrown exceptions - class diagram

Table 5.2: The engine exceptions

Name	**Description**
BasicCVLEngineException	It is a superclass for all excptions the engine throws. We never throw it directly in the code.
GeneralCVLEngineExcept-ion	It is a general exception, we throw the exception for cases when it is difficult to identify a source of a failure.
IllegalCVLOperation	The exception is thrown when some operation can not been completed due to restrictions which are set by a metamodel. This type of an exception may imply in some cases that a variability model is incorrect.

Continued on the next page

Table 5.2 – *Continued from the previous page*

Name	**Description**
IncorrectCVLModel	The engine throws the exception when we may identify that a variability model is incorrect.
UnexpectedOperationFail-ure	The operation is thrown when an operation is not completed properly for some reasons.
UnimplementedException	The engine throws the exception when we try to execute operation which is not currently implemented. It should not be ever thrown in the final implementation.

5.3.6 Tests

We have implemented currently about sixty test cases to verify functionality of the engine. The vast majority of the test cases verify the substitution operation, but in general we can split all them in three main test plans. Each package implements a test plan. Table 5.3 lists all packages with a concise description for each of them.

Table 5.3: The test plans

Name	**Description**
no.sintef.cvl.engine.operat-ion	The package contains thirty five test cases verifying the pure substitution operation. The package contains negative test cases. The vast majority test samples are built using the Node language, however in some cases the metamodel of the Node language has been changed to test different edge conditions, e.g. for example when an association has finite multiplicity or presence/absence of a containment relation. We use UML for same test samples.
no.sintef.cvl.engine.adjace-nt	The package contains twenty two test cases which verify different adjacent configurations between placements. We use the Node language to build test samples, but there are some in UML.
no.sintef.cvl.engine.fragm-ent	The package defines two test cases. The test cases verify functionality of different holders checking calculated information such as IP, OP, PEint, PEext etc.

5.3.7 Discussion

The implementation shows that the different elements of the engine are fairly decoupled. Thus, it leaves room for further extension and refactoring. In addition, we have woven the Spring development framework and some design patterns, e.g. the singletone pattern. It should also ensure scalability and customizability of the engine.

We pay special attention to the adjacent module which implements the main contribution of the thesis. Generally speaking, a resolution of the adjacent relation is a three-fold process:

- finding adjacent relations;
- finding adjacent boundaries;
- adjusting adjacent boundaries.

We store explicitly all adjacent fragments, since there is no any implication or transitivity between adjacent fragments. Meaning, that the adjacent resolution operation does not raise a need to adjust boundaries of fragments in a transitive closure to a given fragment. In other words, if the fragment A is adjacent to the fragment B which is adjacent to C in return, then A is not adjacent to C through the transitive clause. Thus, it is enough to adjust boundaries which are explicitly adjacent and can be found in the adjacent founder. Adjacent placements should have adjacent boundaries since boundaries are only elements which define the relation between fragments. If it does not hold then there is a problem in a variability model or failure in the implementation. The final step is an adjusting of boundaries which should follow a single substitution operation in order to keep the consistence of a variability model all the way through the derivation process. The suggested methodology and implementation provide means to hold the invariant of keeping a variability model consistent.

Chapter 6

Experiments and validation

6.1 Neighboring placements problem

We validate the approach to the adjacent relation using the simplified example in the Node language which we use across the thesis. Figure 6.1 shows two placement fragments depicted by the ovals with the solid line. We define also two replacement fragments which are outlined by the ovals with the dashed line in Figure 6.2 on the following page. The dashed arrows are outside/inside boundary references, where the red arrow with the filled head is a reference to modify during the derivation process. In

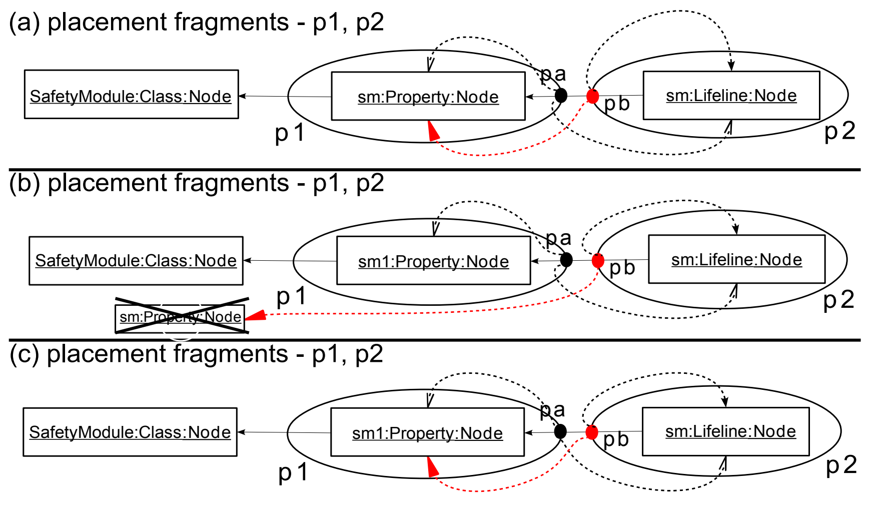


Figure 6.1: The neighboring problem, placements - validation

order to define the substitution of *p1* and *p2*, we bind *pa* to *ra* and *pb* to *rb*. Further, we substitute the placement *p1*. Figure 6.1 (a, b, c) depicts states which the base model goes through, i.e. (a) is an initial state, while (b) shows the broken reference after the substitution of *p1*, finally, (c) exemplifies the base model after the adjacent resolution process. Figure 6.1 clearly shows that an application of the developed algorithm leads to the desired result. Subsequently, the substitution of the placement *p2* yields the correct result where the link between the elements *sm1:Property* and

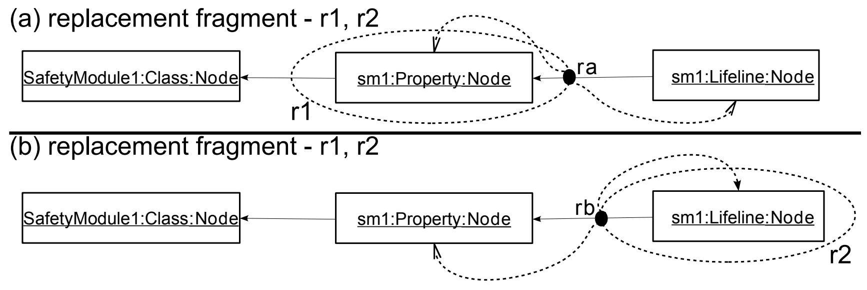


Figure 6.2: The neighboring problem, replacements - validation

sm1:Lifeline is established, because the *outsideBoundaryElement* reference of *pb* points to the required element after the adjacent resolution process. We also run the adjacent resolution algorithm against the SafetyDrive example in UML and get the expected substitutions.

6.2 Evaluation

6.2.1 Strategies

In this section, we try to evaluate the quality of the implemented adjacent resolution algorithm. EMF is a convenient environment for development in the modeling realm, where one has off-the-shelf means to operate on a model. A down side of EMF is that the framework is generally greedy with respect to consumed resources. Therefore, we do not impose any resource requirements on the engine so far while the execution time seems to be a dominating factor at this point. The engine is a part of the OMG CVL tool which is responsible for the substitution operation contributing to the overall execution process. At the same time, the implemented semantics

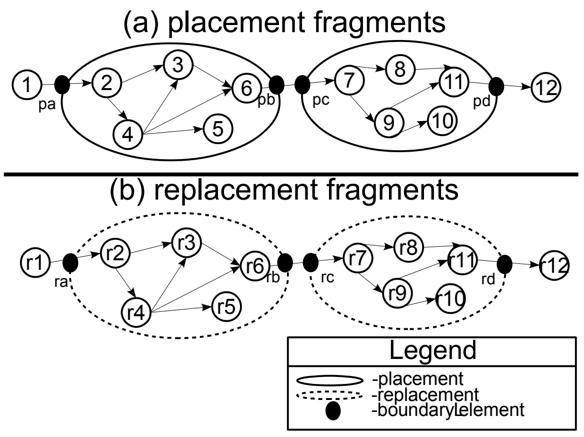


Figure 6.3: The test fragments - basic sample

is a relatively small part of the whole CVL execution semantics; therefore, we are interested in performing a single substitution as fast as possible. We try to find out factors which affect the resolution process in order to define strategies for further optimizations and improvements.

We define several test systems to measure time characteristics of the implemented resolution algorithm. Figure 6.3 on the preceding page shows a basic test sample for further experiments. It contains two placements with the adjacent relation which is defined by the link between two neighboring elements *6* and *7*. One may find two replacements fragments with elements we want to insert onto the placements. Each placement/replacement fragment defines two boundary elements which we bind to the boundaries of the corresponding replacement/placement fragment, i.e. *pa* is bound to *ra* and *pb* to *rb* etc. Further, we modify given fragments and test execution time for different operations in the engine. We measure the following durations:

- finder execution duration - time required to find all adjacent relations;
- substitution execution duration - time required to perform a single substitution;
- resolution execution duration - time required to resolve all adjacent relations for a single fragment;
- overall substitution duration - time required to execute the substitution and resolution of a single fragment; thus, it aggregates the execution durations of the substitution and resolution operations.

We conduct an experiment on a given test sample ten times, e.g. we measure the substitution duration for the placement *p1* in Figure 6.3 on the facing page ten times; further, we calculate a mathematical expectation to get an average value of the conducted measurements and finally, we compute a standard deviation to see how the measured values are different with respect to an average to get the feel of a possible error.

We present four test strategies, where experiments are designed in such a way that one can compare results. Measurements should give a picture on the performance of the engine and how different factors affect the derivation process. One may find all mean values and standard deviations in Tables A.1 on page 95, A.2 on page 97, A.3 on page 99, A.4 on page 101, but further, we work with the graphics which represent the data in the tables from Appendix A on page 95 for the convenience. All measurements are made on a machine with the following characteristics:

- **OS:** 32-bit Windows 7 Professional (SP1)
- **Processor:** Intel(R) Core(TM)2 Duo CPU P8600 @ 2.40GHz 2.40GHz
- **Installed memory (RAM):** 4.00GB (2.96GB usable)

Test strategy 1

We measure how the size of the placement/replacement affects the chosen durations. In the experiment, we increase the size of the placement/replacement gradually by step fifty from the default placement/replacement *p1/r1* up to a fragment which has one thousand elements. We increase the number of elements by simple copying of the element *5/r5*. Thus, we do not create any extra elements or references (which may create boundaries etc.) affecting measurements which is not desirable for the test strategy 1.

Test strategy 2

We would like to find out how the number of boundaries influences the speed of substitution and resolution. There are several options here: we can increase the number of adjacent boundaries (adjacent pairs), or boundaries which do not form the adjacent relation between fragments. For the test strategy 2 we copy boundaries and corresponding elements which do not take part in the adjacent relation, i.e. we copy *pa/ra*, and the element *2/r2* in the placement/replacement *p1/r1* in Figure 6.3 on page 76. We copy the boundary *pa/ra* gradually in a loop by step fifty up to one thousand boundaries.

Test strategy 3

We increase the number of adjacent boundaries (adjacent pairs) and corresponding elements. In the placement *p1* we copy the adjacent boundaries *pb* and *pc* gradually in a loop by step fifty up to one thousand adjacent pairs (which give two thousand adjacent boundaries overall, i.e. one thousand adjacent boundaries per each placement). For each test sample, we measure the finder, execution and resolution durations.

Test strategy 4

In the test strategy 4, we also increase the number of the adjacent boundaries (adjacent pairs) as in the test strategy 3, but here we copy an opposite adjacent fragment with all its references, such a way, we create a new adjacent relation between two placements. In the sample from Figure 6.3 on page 76, we copy the placement *p2*, thereby, the brand copied placement creates the adjacent relation with *p1*. The adjacent boundaries *pb* and *pc* are copied together with copying of the placement *p2*.

6.2.2 Results

Test strategies 1,2

Figure 6.4 on the facing page visualizes measurements made for the test strategies 1,2. The test strategies 1 and 2 are the same in the sense of the placement size. Both strategies replace the placement *p1* (see Figure 6.3 on page 76). The figure clearly shows that the substitution time is hardly affected by the size of the placements, which one may notice observing the growth of the *Single substitution - TS 1* line. Moreover, the calculated standard deviations show that the graph can be approximated by the linear function $f(x)=K*x + C$, where K equals to ~0.1 and C is roughly 400, meaning that the growth is low. In contrast, the growth is high for the test strategy 2

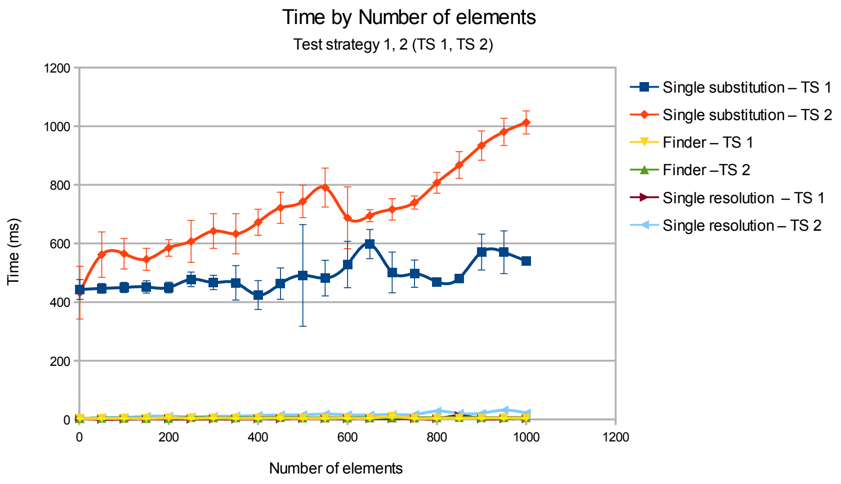


Figure 6.4: Test strategies 1,2 - data visualization

where we copy a non-adjacent boundary element together with an element of the placement. Almost all values of *Single substitution - TS 2* can be approximated by a linear function and one can make sure that the order of growth does not follow the power rule calculating empirical order using the approach by Sedgewick and Flajolet [53]. In other words, we may suggest that the order of growth for the algorithm follows the power rule, i.e. $\approx k*x^a$, where we can calculate the coefficient a using the following formula $a=log(t2/t1)/log(n2/n1)$ if a is a constant for different ranges of $n2$, $n1$ and corresponding $t2$, $t1$, then we have the power rule growth. Table 6.1 lists the calculated values for the coefficient a. One may observe the the coefficient a varies for the different ranges; thus, the order of growth for the *Single substitution - TS 2* line does not follow the power rule. Figure 6.4 also shows that the other durations do not increase with the size of the placement *p1*. The finder execution duration does not grow because we do not add any adjacent placements and boundaries. At the same time, one may observe the slight growth of the *Single resolution - TS 2* line. It seems to be logical since boundaries take part in a substitution; however, we may conclude that the number of non-adjacent boundaries does not affect the adjacent resolution process significantly.

Table 6.1: Empirical order of growth - substitution TS 2

Formula	**1/7/13/19**	**2/8/14/20**	**3/9/15**	**4/10/16**	**5/11/17**	**6/12/18**
			Points 1 - 6			
t=t2/t1	1.00	1.00	1.00	0.99	1.06	0.97
n=n2/n1	50	2	1.5	1.33	1.25	1.2
log(t)/log(n)	0.002	0.01	0.01	-0.01	0.26	-0.12
			Points 7 - 12			
t=t2/t1	0.99	0.91	1.09	1.06	0.98	1.09

Continued on the next page

Table 6.1 – *Continued from the previous page*

Formula	**1/7/13/19**	**2/8/14/20**	**3/9/15**	**4/10/16**	**5/11/17**	**6/12/18**
n=n2/n1	1.16	1.14	1.12	1.11	1.1	1.09
log(t)/log(n)	-0.01	-0.69	0.74	0.56	-0.20	1.05
Points 13 - 18						
t=t2/t1	1.13	0.83	0.99	0.94	1.02	1.18
n=n2/n1	1.08	1.07	1.07	1.06	1.06	1.05
log(t)/log(n)	1.55	-2.39	-0.11	-0.88	0.41	2.98
Points 19 - 20						
t=t2/t1	0.99	0.94				
n=n2/n1	1.05	1.05				
log(t)/log(n)	-0.01	-1.05				

Test strategies 3,4

Figure 6.5 on the facing page depicts mean values and standard deviations calculated for the test strategies 3 and 4. Both strategies check how adjacent boundaries influence the product derivation process. In each strategy, we create adjacent boundaries to *p1*, substitute the placement *p1* as well as resolve adjacent boundaries to *p1*. One may notice straightforward that adjacent boundaries affect the substitution process roughly the same way as non-adjacent boundaries do. Thus, we may conclude that it does not matter whether created boundaries are not adjacent at all or whether adjacent relations are created by a single fragment with many adjacent pairs or many adjacent fragments with a single adjacent pair (see Figure 6.4 on the previous page). Linear functions can approximate both graphics for the substitution duration. However, the other durations, i.e. the finder execution duration and the resolution execution duration (just for TS 4), are influenced differently with respect to TS 1, 2.

The difference in the resolution execution durations between test strategies 3, 4 is a catching eye feature on the line chart in Figure 6.5 on the facing page. We can observe that boundaries, which belong to a single adjacent placement (TS 3), almost do not affect the adjacent resolution process as well as non-adjacent boundaries from TS 3 (see Figure 6.4 on the previous page). At the same time, boundaries, which belong to different adjacent fragments, increase the resolution time linearly (TS 4). We should also pay attention to the graphics which show how the durations increase for the finder process. The growth is steep for TS 4 while the TS 3 finder execution duration increases gradually and represent a linear function. One may argue the graphic for the TS 4 finder execution duration may have the power rule growth, however, calculations based on the empirical data do not prove the assumption, see Table 6.2 on the facing page.

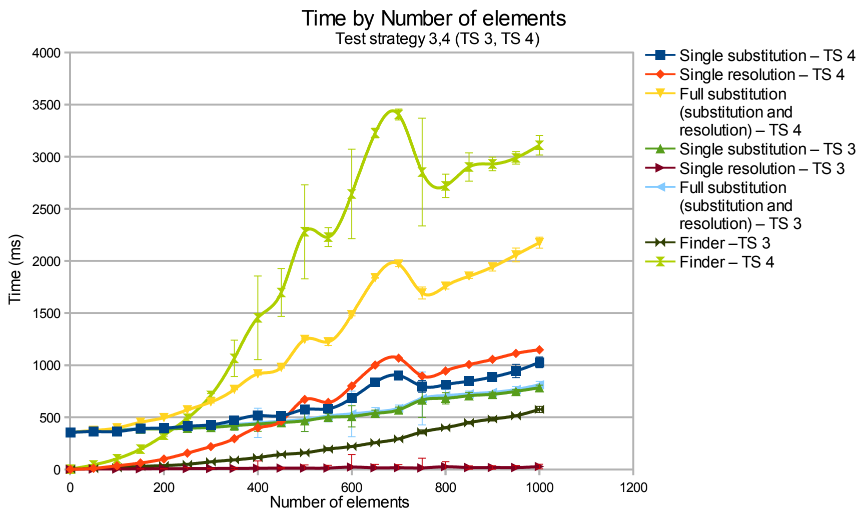


Figure 6.5: Test strategies 3,4 - data visualization

Table 6.2: Empirical order of growth - finder TS 4

Formula	**1/7/13/19**	**2/8/14/20**	**3/9/15**	**4/10/16**	**5/11/17**	**6/12/18**
Points 1 - 6						
t=t2/t1	14.4	2.37	1.89	1.68	1.49	1.44
n=n2/n1	50	2	1.5	1.33	1.25	1.2
log(t)/log(n)	0.68	1.24	1.57	1.82	1.80	2.02
Points 7 - 12						
t=t2/t1	1.50	1.36	1.16	1.34	0.97	1.18
n=n2/n1	1.16	1.14	1.12	1.11	1.1	1.09
log(t)/log(n)	2.63	2.33	1.30	2.80	-0.23	1.95
Points 13 - 18						
t=t2/t1	1.22	1.05	0.83	0.95	1.06	1.00
n=n2/n1	1.08	1.07	1.07	1.06	1.06	1.05
log(t)/log(n)	2.49	0.72	-2.56	-0.73	1.06	0.17
Points 19 - 20						
t=t2/t1	1.01	1.04				
n=n2/n1	1.05	1.05				
log(t)/log(n)	0.36	0.77				

Moreover, some points include substantial errors; therefore, one may try to approximate the graphic by a linear function which covers almost all value. At the same time, the graphic has some picks with minor deviations, which is not possible to explain by the given linear function. A nature of these picks is deemed unclear for us currently.

Chapter 7

Related works

7.1 MoSiS CVL

MoSiS CVL [22] is the first implementation of the CVL language. The tool has been developed within the MoSiS project ITEA 2 - ip06035 part of the Eureka framework [55]. Svendsen et al. [54] introduce the basic CVL concepts and methodology of applying CVL to any domain specific language on UML and Train Control Language (TCL) examples. We can implement this methodology directly using the MoSiS CVL tool. The CVL specification does not provide any conflict resolution semantics between fragments in CVL. Our thesis proposes such semantics. We should point out that Oldevik et al.[39] study conflicts and confluence of substitution fragments in CVL. They define a fragment as the triple *Eint*, *BEint*, *BEext*, where *Eint* represents elements internal to the fragment and not referenced by *insideBoundaryElement* references, *BEint* contains elements specified by *insideBoundaryElement* references, and *BEext* represents a set of elements referenced by *outsideBoundaryElement*. The authors claim that

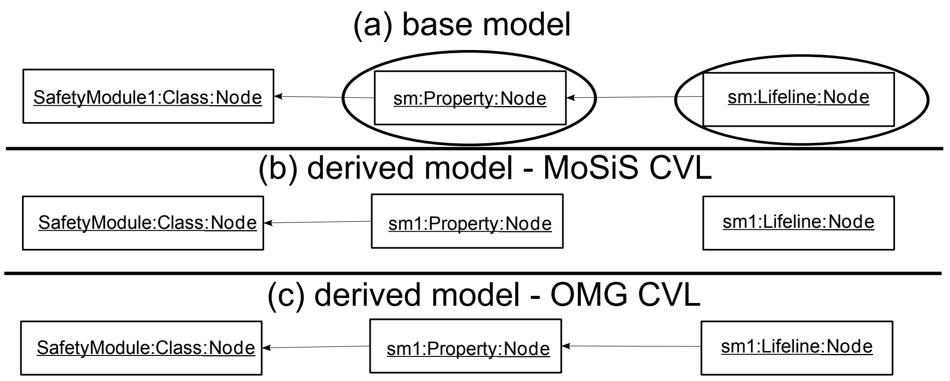


Figure 7.1: Substitutions in MoSiS CVL and OMG CVL

the confluence of substitution fragments follows only if an overlap exists in the *BEext* sets. In our proposed classification, we consider such fragments as *parallel independent*. *BEext* matches the definition of *PEext* while the concatenation of *Eint* and *BEint* is equivalently to *PEint* in the thesis. MoSiS

CVL implements partially the findings by Oldevik et al. Figure 7.1 on the previous page depicts the well established and used example across the thesis with two placements defining the substitutions of two neighboring elements, i.e. *sm:Property* and *sm:Lifeline* elements. These neighboring elements form the overlap between *BEint* of one placement and *BEext* of another as well as create the adjacent relation between the placements in the thesis terms. We call such placements *adjacent*. Oldevik et al.[39] state that this overlap should lead to an inconsistent model. In Figure 7.1 on the preceding page, we show the base model (a) and two derived products in the MoSiS CVL tool (b) and the implemented engine (c). The MoSiS CVL tool derives an incorrect product if substitution fragments have the adjacent relation. We can clearly see that the reference between the elements *sm1:Property* and *sm1:Lifeline* does not exist in the derived product. However, the engine presented in the thesis successfully resolves the adjacent relation (see Figure 7.1 on the previous page (c)) between the placements.

7.2 Confluence of graph transformations

Confluence of conflicting graph transformations plays a major role in the graph rewriting theory. Conflicts between transformations occur when the transformations share common elements, the graph rewriting theory calls such transformation non-parallel independent.

Definition 7.1 (Parallel independence) *Given two transformations $G \overset{p1(o1)}{\Longrightarrow} H1$ and $G \overset{p2(o2)}{\Longrightarrow} H2$, $G \overset{p1(o1)}{\Longrightarrow} H1$ is* ***(weakly) parallel independent*** *of $G \overset{p2(o2)}{\Longrightarrow} H2$ if the occurrence o1(L1) of the left-hand side of p1 is preserved by the application of p2 [27].*

Heckel, Küster and Taentzer [27] give theoretical bases for identifying a parallel independence between transformations in terms of the rewriting theory. If two transformations are parallel independent then the local Church-Rosser theorem states that the transformations can be performed in any order yielding the same result [19]. Thus, we can speak of confluence of the parallel independent transformations.

Confluence is also feasible for non-parallel independent transformations. In order to check confluence for non-parallel transformations, it is enough to test the confluence of all critical pairs. Where critical pairs are conflicting pairs built from the mutually conflicting transformations. If conflicting pairs (critical pairs) are confluent, that is, there are transformation sequences leading to a common successor graph [27] then the corresponding non-parallel independent transformations are locally confluent [27]. It is a recap on the Critical Pair Lemma [29] developed for string rewriting systems [34]. In other words, one may say a graph rewriting system is confluent if all its critical pairs are strongly joinable [44]. We say that critical pairs are strongly joinable if transformation sequences leading to a common reduct graph keeps all nodes which are preserved by the very first transformations yielding the critical pairs.

We have shown in Section 4.2.3 that the graph transformation tool Henshin does not solve the problem when one tries to substitute two neighboring elements independently. To be more precise one transformation disables another. Let us consider the following graph transformation system, see Figure 7.2. The figure exemplifies the neighboring problem (see Section

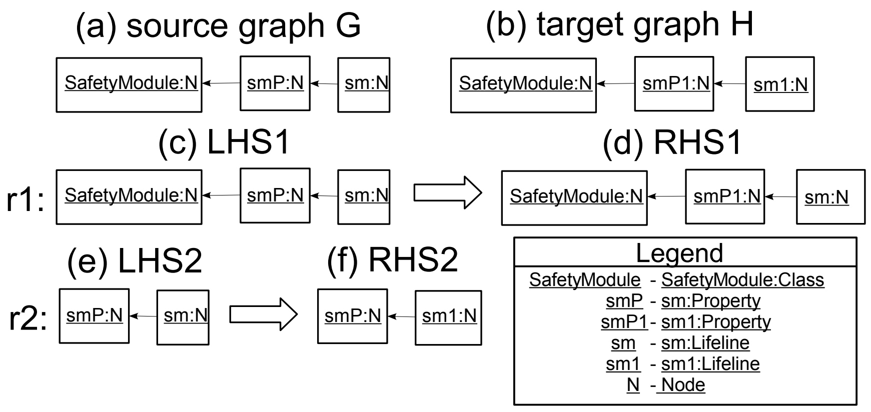


Figure 7.2: The neighboring problem - graph rewriting system

4.1.2) where the graph transformation system defines two transformations, which should substitute two neighboring elements. The system defines the source graph G (a) and the target graph H (b). The rule *r2* substitutes *sm:Lifeline* and the rule *r1* defines the substitution of *sm:Property*. The left-hand sides of both rules are in a conflict to each other, i.e. it is a delete-use conflict. The first rule removes elements which are used by the second one. We can check the confluence property of the given rewriting system by introducing the following critical pairs, see Figure 7.3. Figure 7.3 (d) shows

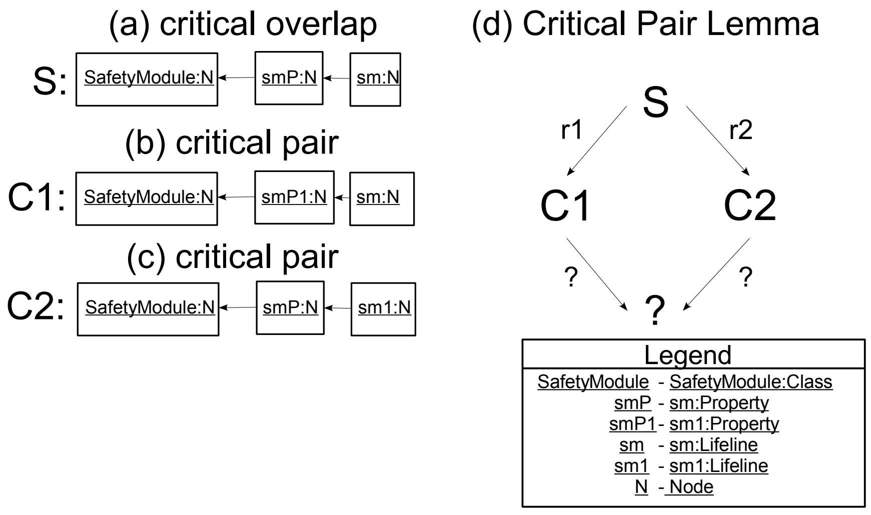


Figure 7.3: The neighboring problem - joinability of the critical pairs

that non of the rules *r1* and *r2* can reduce the critical pairs to the common graph. Thus, the critical pairs are not joinable; therefore, the graph transformation system in Figure 7.2 is not confluent. It implies that both transformations from Figure 7.2 on the previous page will never yield the graph H using the graph transformation approach. We have shown that both these transformations can be executed in CVL sticking to the proposed approach to resolution of the adjacent relation between fragments.

Triple Graph Grammars (TGG) [52] is another approach to graph transformations. The approach defines rule based declarative formalism to specify transformations. Rules are bidirectional in TGG, meaning that the separation between source and target graphs is rather synthetic, i.e. one may apply rules to a source graph deriving a target graph and the other way around. A rule is defined as a production that contains elements of source and target graphs together with an element that defines a bidirectional mapping between the elements of the source and target graphs. The approach has been mainly created to support the model-to-model/out-place transformation [51], but seems to be feasible for in-place transformations. TGG suggests an ordered application of rules as a mean to tackle conflicts which is somewhat different with respect to the algebraic graph transformation theory and CVL. For example in CVL, substitutions are independent operations and can be applied generally in the arbitrary order.

7.3 Conflicts in product line engineering

Oldevik et al.[39] analyze conflicts and confluence between substitution fragments in CVL. The authors discuss cases when multiple fragments of a variability model are in a conflict relation and suggest criteria to detect conflicts in a variability model. Oldevik et al. introduce a confluence checking procedure of fragment substitutions, discuss when confluence follows. The paper states that the suggested transformations can be mapped to the graph transformations and checked using the critical pair analyzes. We took the work of Oldevik et al. as a starting point for our research, classified overlaps, checked their confluence using graph transformation based tools and defined solutions to tackle overlapping fragments.

In [28] and [62] authors make a survey of challenges a user can come across during a configuration process and suggest a novel approach to fix conflicts in configurable software. Such system may use a constrain notion to define limitations on available options during a configuration process. These constraints may have a sophisticated nature in complex and highly configurable systems. Thus, it may bring additional challenges for a given particular configuration. In the thesis, we do not suggest any constrain based notation, but rather suggest the approach which adjusts a variability model to keep consistence all the way through a derivation process. If it is not possible, we consider a failure in a variability model.

Hyunsik Choi and Kyo Chul Kang [10] discuss complex feature models and tackling conflicts between different features during a configuration process. A main idea is that some frequently selected features in a model should be merged, making the feature model smaller. Configuration constraints should be defined on a group of features instead of targeting a particular feature. They argue that the suggested transformation process does not require any human interference. Thus, a configuration process of the transformed model should become conflict-free and easier than the configuration of the original one. The approach of Hyunsik Choi and Kyo Chul Kang is somehow similar to the unionization approach for crossing fragments described in the thesis. In addition, we show that an overlap of the kind adjacent fragments can be tackled by adjusting references in a variability model without the unionization procedure.

Diskin, Xiong and Czarnecki [17] state that development of software involves several models, each of them may capture different aspects of a system; therefore, models may have various metamodels. Such models are called heterogeneous. A need of merging heterogeneous models emerges when different teams develop such models independently. The authors employ consistency-checking-by-merging (CCM) idea to merge heterogeneous models. The idea involves two step procedure: specify overlap between models and merge them with further check of the result with respect to constraints in the global metamodel. The approach uses the graph rewriting and category theories to specify transformations. Therefore, it is not directly applicable when a graph transformation system is not confluent, e.g. when the neighboring problem (see Section 4.1.2) pops up.

7.4 Feature-oriented and Delta-oriented programming

Feature-oriented programming (FOP) [6] is an example of the Aspect-oriented Programming paradigm (AOP) introduced as the step-wise refinement approach by Batory et al. [5] to the development of complex systems. A core idea of the step-wise refinement approach is that a product may emerge by adding features incrementally to a simple base model. Batory et al. show that the approach can be applied to both code and non-code artifacts given that one defines the composition operation for each kind of artifacts.

Delta-oriented programming (DOP) [47] is an extension of the FOP paradigm and a novel programming language approach which operates with deltas to derive a product. A SPL definition contains a core module and set of delta modules in DOP. The core module represents a valid product where further products can be derived applying deltas [47]. Deltas allow removing elements from a product which is not generally allowed in the feature modeling. One may define a SPL on any language using the DOP paradigm. Deltas are specified as no class is added or removed in more than one delta module, and fields or methods (added, modified and renamed) are disjoint for every class, which are modified in more than one delta module. The approach proposes to resolve all conflicts between deltas by specifying the order of their resolution. The notion of deltas is somehow similar to fragments in CVL. However, one may define several fragments modifying the same elements in a model, which is a core distinction.

Moreover, we consider a substitution as an independent operation which CVL may apply potentially in the arbitrary order. Therefore, the ordering is not a solution to overlapping fragments at least within the current CVL semantics.

7.5 Aspect-oriented Programming

Aspect-oriented Programming (AOP) is an approach which allows weaving cross-cutting concerns into a program. One of the most successful implementations of the AOP concepts is AspectJ. Many frameworks (such as Spring) use the AspectJ terminology, which is a successor of the AOP concepts developed in Xerox PARC, in their implementations of AOP. AspectJ defines the *aspect* term as a modular that defines and contains cross-cutting concerns to cut across an application. A cross-cutting concern encompasses a *pointcut* and *advice*. A pointcut defines a pattern to match in the application, while an advice instructs how a cross-cutting concern should be inserted, e.g, before, after, within etc. A list of all available matches defined by a pointcut comprises *join points*.

Aspects are developed as separate units which can be applied independently. Therefore, AOP supports the separation of concerns principle. Lauret et al. [35] state that AOP suffers form a well-known composition issue i.e. several concerns are applied to the same join point. Conflicting concerns may cause an undesirable result during an execution. The problem is known as the aspect interference issue. Lauret et al. suggest inserting *executable assertions* to detect different kind of interference between aspects. The implementation uses the AIRIA resolver construct introduced in AspectJ [35]. As a solution to avoid undesirable interferences, the authors suggest ordering of conflicting advises. The notion of aspects is highly relevant to fragments in CVL which can be applied to the same model elements. However, the ordering of fragments to resolve conflicts is somewhat different with respect to CVL where substitution operations do not have any particular order.

Grønmo [20] presents an aspect language for UML 2.0 sequence diagrams. The author introduces the term *aspect diagram* to denote a sequence diagram-based aspect. Grønmo uses concrete syntax of the UML sequence diagram to weave in aspects. They state also that the sequence diagram is a special graph where graph elements are ordered. Grønmo adopts the graph transformation theory for the UML sequence diagram formalizing their aspect language. They study termination and confluence of the sequence diagram-based aspects system. The approach is limited to UML sequence diagrams due to their specifics. In our thesis, we try to generalize the problem and solution. In addition, we would like to tackle cases, i.e. the adjacent and crossing relations which are appeared to be non-confluent in terms of the graph transformation theory.

Part III

Conclusion

Chapter 8

Results and Future work

8.1 Results

CVL is a language to define software product lines. The language has the notion of fragments which allows specifying elements to substitute in a model. Modern modeling languages may have complex metamodels; therefore, the tools, which implement the corresponding metamodels, may use different diagrams to represent a model and facilitate the development process. Fragments defined in different diagrams may overlap in a model causing unintended results during a product derivation. A variability engineer may define overlaps intentionally, reflecting a pragmatic need to specify substitution fragments in different diagrams, or by accident where overlaps may indicate a failure in a variability model. In the thesis, we addressed this issue and proposed the appropriate solutions.

In Chapter 4, the thesis shows that the old CVL implementation struggles to obtain a product that conforms to the intentions of an engineer if a variability model has overlaps between placement fragments. We relate the problem to the graph transformation theory and show that a solution is not directly feasible applying techniques from that realm. Subsequently, we classify such overlaps and deduce the following kinds: the adjacent fragments, the crossing fragments, the crossing and adjacent fragments and contained fragments. We define the criteria to identify the mentioned relations between fragments and means to tackle the issue. The approach is based on modifying of the variability model, i.e. the unionization of crossing fragments or adjustment of boundaries for adjacent fragments.

We implement the findings in the substitution engine discussed in Chapter 5. The engine is able to perform the substitution operation and has the functionality to detect and solve the adjacent relation between placements. A basic idea of the approach to resolution of the adjacent relation is to modify the variability model in such a way that it is consistent all the way through the derivation process. The resolution process of the adjacent relation includes the following steps: detection of the adjacent relation, finding of adjacent boundaries, modifying of the adjacent boundaries. The engine should run the adjacent resolution procedure after a substitution operation to modify a variability model properly.

We use the SafetyDrive UML example to test that the substitution and resolution operations to adjacent fragments are feasible in the real world settings. Subsequently, we validate the adjacent resolution operation on the running example in the Node language which is the essence of the SafetyDrive UML case. In addition, we have also developed around sixty test cases and samples simulating different situations, which may occur between two adjacent placements. This validates that the engine correctly resolves the adjacent relation and can tackle various situations. The engine utilizes the modern means of development and design patterns which should increase scalability and customizability of the engine.

We have evaluated the execution time of the basic engine operations and found out how different factors affect the execution process in Chapter 6. The size of a fragment has a small impact on the duration of the substitution operation. The execution time linearly grows as the number of boundaries increases. We have also found the resolution operation is barely affected by the number of adjacent boundaries, which belong to a single adjacent placement; however, the growth is evident if we increase the amount of adjacent placements keeping the number of adjacent boundaries constant. The function, which represents the time dependence to find adjacent relations and boundaries, grows sharply with the increase of adjacent fragments.

8.2 Discussion

We use the set formalism to deduce kinds of overlaps. It helps us to emerge the exhaustive list of possible relations between fragments if we consider a fragment as a set of elements, i.e. independent fragments (fragments which do not intersect), crossing fragments (fragments which intersect) and contained fragments (one fragments is a subset of another). However, we have shown that relations between fragments in CVL is somewhat more complicated than just the well-known relations between sets due to possible references between elements of different fragments. Thus, we empirically define two overlapping kinds in addition: adjacent fragments, adjacent and crossing fragments. We have introduced our elaborated practical approach which shows that we have found all overlapping kinds. The approach exploits the set theory, the graph theory and combinatorics. In addition, we define our formalism giving basic definitions and providing the criteria to distinguish different overlapping kinds and boundaries. However, one may argue though that it is not obvious or clear enough that we have found all kinds of overlaps and the criteria are sufficient. We can express such arguments in favor of having a better math which may concisely and straightforward prove that 1) all possible overlapping kinds are found, 2) our criteria to detect overlapping fragments and classify boundaries cover all possible cases and sufficient enough.

In the thesis, we propose solutions to different overlapping kinds defining the overlapping resolution semantics in CVL. We use the real world example which is well established and covered in the thesis to specify the adjacent resolution semantics. Thus, we are confident in the proposed solution which is implemented and validated using the numerous test samples. While the resolution semantics for the crossing kinds is rather arbitrary at this point and based on the common sense, observations and expectations. In Section 4.5, we have described the approaches to tackle different crossing kinds. For example, we propose the unionization procedure for the *crossing placements*, *adjacent and crossing placements* kinds while the *contained placements* kind is deemed as a failure in a variability model. However, the unionization procedure is applicable to contained fragments as well. We currently miss also well established and understood examples with overlaps of such kinds from the real world. Moreover, we did not have enough time to implement and experiment with the proposed crossing resolution semantics.

8.3 Future works

The thesis introduced the substitution engine with the adjacent resolution functionality. We continue further testing of the engine to ensure that all operations work as expected. We need also to find other real world examples built on widely used modeling languages such as UML to verify the engine thoroughly.

In the thesis, we have proposed the solutions for all overlapping kinds. Meanwhile, the substitution engine currently implements resolution of the adjacent relation between fragments. Thus, one of the future goals is to realize the resolution functionality for other relations between fragments based on the proposed solutions.

The substitution engine is developed independently; thus, a further step is to integrate the engine with other parts of the OMG CVL tool and examine how it fits in the tool infrastructure.

One may observe from the evaluation section that the substitution does not take milliseconds and roughly can be measured by seconds, especially if we have fragments with lots of boundary elements. It indicates that the overall derivation process may be a time consuming operation. Therefore, the process may negatively affect the user tool experience. Some preprocessing can possibly reduce the derivation duration and improve the usability of the tool in return.

The evaluation section shows that all relations for TS 3, 4 in Figure 6.5 on page 81 depend linearly on the number of adjacent boundaries and placements. The substitution durations for TS 3 and TS 4 have almost the same growth. In fact, one can observe the slight difference only when the amount of adjacent boundaries is more than three hundred boundaries. However, the corresponding finder and resolution relations of TS 3 and TS 4 differ significantly. The finder and resolution functions for TS 4 grow faster than the respective graphs for TS 3. These dependencies affect the overall performance of the substitution engine. Therefore, we may need to improve the algorithms of finding adjacent relations and their resolutions in such case (when adjacent relations are created by different fragments) to benefit the overall performance of the engine. We have also noticed some picks of an unclear nature for TS 4. This may require further investigation.

In Section 8.2, we mention that there may be a math which may show concisely and straightforward that our formalism covers all possible cases. However, we have not investigated that due to time limitations. Thus, a possible extension of the work is to find or develop a mathematical theory which we can use to prove that the criteria are correct and sufficient. In addition, we may benefit from such formalism by improving the criteria to detect different relations and the product resolution process in return.

We have mentioned in the Section 8.2 that we have not had time to implement and experiment with the crossing resolution semantics. In addition, we currently do not have examples to validate the proposed approach to tackle crossing fragments. Further, we should try to find corresponding examples and verify the defined resolution semantics to crossing fragments. Subsequently, it may lead to modifications and improvements in the proposed crossing resolution semantics.

Appendix A

Test strategies 1,2,3,4 - measurements

The tables below contain the mean values and standard deviations for the measurements we have conducted for all test strategies (see Section 6.2 on page 76). For all experiments, we increase the number of elements inside a placement or amount of boundaries gradually by step fifty up to one thousand. For each step, we perform ten measurements and calculate the mean value and standard deviation. We fill the corresponding columns with such values in the tables below, where the very first row represents a step (the number of elements).

Table A.1: Mean values and standard deviations - part 1

Durations (ms)	**1**	**50**	**100**	**150**	**200**	**250**
Test strategy 1 - mean values						
Finder execution	3.2	3.3	3.6	3.1	5.1	3.6
Single substitution	443.1	446.7	450	451.8	449.9	477.7
Single resolution	3.1	0.7	1	1.7	1.4	1.2
Full substitution	446.2	447.4	451	453.5	451.3	478.9
Test strategy 1 - standard deviations						
Finder execution	1.39	0.48	0.51	0.56	5.95	1.89
Single substitution	34.04	16.59	17.32	21.02	18.25	24.72
Single resolution	5.50	0.48	0.47	0.82	0.51	0.42
Full substitution	34.59	16.58	17.25	20.84	18.55	24.86

Continued on the next page

Table A.1 – *Continued from the previous page*

Durations (ms)	**1**	**50**	**100**	**150**	**200**	**250**
Test strategy 2 - mean values						
Finder execution	3.3	4.7	4.5	4.2	4.3	6.8
Single substitution	432.3	561.9	565	545.9	584.9	607
Single resolution	0.6	6.4	7	10.6	11.2	8.9
Full substitution	432.9	568.3	572	556.5	596.1	615.9
Test strategy 2 - standard deviations						
Finder execution	0.67	2.75	0.97	0.91	1.05	2.14
Single substitution	89.85	77.33	52.06	37.39	28.29	71.73
Single resolution	0.51	0.69	1.69	5.35	2.39	1.85
Full substitution	89.99	77.39	53.15	41.15	28.94	72.07
Test strategy 3 - mean values						
Finder execution	2.8	8.1	16	28.5	36.6	49.2
Single substitution	353.9	367.5	372.6	386.8	388	395.8
Single resolution	0.8	3.8	4.7	6.1	7	8
Full substitution	354.7	371.3	377.3	392.9	395	403.8
Test strategy 3 - standard deviations						
Finder execution	0.42	0.87	0.66	4.52	0.69	0.63
Single substitution	3.87	3.86	5.01	29.99	4.29	3.11
Single resolution	0.42	1.03	0.67	0.31	0	0.66
Full substitution	4.00	4.59	5.29	29.96	4.29	3.04
Test strategy 4 - mean values						
Finder execution	3	43.2	102.4	194.3	328	490.9
Single substitution	354.1	363.4	363.2	390.7	396	414.6
Single resolution	0.8	10.3	34.5	61.6	99.6	156.5

Continued on the next page

Table A.1 – *Continued from the previous page*

Durations (ms)	**1**	**50**	**100**	**150**	**200**	**250**
Full substitution	354.9	373.7	397.7	452.3	495.6	571.1
Test strategy 4 - standard deviations						
Finder execution	0.66	0.91	2.54	2.75	8.62	5.25
Single substitution	5.15	2.54	3.29	25.84	6.48	10.57
Single resolution	0.42	0.48	0.97	1.64	2.01	8.94
Full substitution	5.15	2.49	3.12	25.78	6.11	18.89

Table A.2: Mean values and standard deviations - part 2

Durations (ms)	**300**	**350**	**400**	**450**	**500**
Test strategy 1 - mean values					
Finder execution	4.6	3.6	3.5	4.5	3.5
Single substitution	467	465.7	424.3	463.1	491.3
Single resolution	1.6	1.4	2.1	2.1	2.9
Full substitution	468.6	467.1	426.4	465.2	494.2
Test strategy 1 - standard deviations					
Finder execution	1.17	0.51	0.52	3.71	0.70
Single substitution	24.41	58.80	49.37	53.29	173.48
Single resolution	0.51	0.51	1.10	0.31	2.07
Full substitution	24.26	58.88	49.48	53.38	175.50
Test strategy 2 - mean values					
Finder execution	5.8	6	5.4	6.9	5.6
Single substitution	642.5	633.1	672.4	721.8	743.4
Single resolution	10.6	11.2	13.5	15.5	15.7

Continued on the next page

Table A.2 – *Continued from the previous page*

Durations (ms)	**300**	**350**	**400**	**450**	**500**
Full substitution	653.1	644.3	685.9	737.3	759.1
Test strategy 2 - standard deviations					
Finder execution	2.29	1.88	0.84	1.79	0.84
Single substitution	59.00	68.47	44.30	53.24	55.24
Single resolution	1.17	1.31	2.36	2.75	2.66
Full substitution	59.30	69.61	44.17	52.70	55.90
Test strategy 3 - mean values					
Finder execution	72.8	91.9	114	143	159.7
Single substitution	403.2	419.9	435.7	449.9	467.1
Single resolution	8.9	9.8	11.1	12.5	12.6
Full substitution	412.1	429.7	446.8	462.4	479.7
Test strategy 3 - standard deviations					
Finder execution	2.09	1.52	3.29	2.30	5.88
Single substitution	9.36	3.98	11.59	6.64	6.48
Single resolution	0.56	0.63	1.19	2.67	0.96
Full substitution	9.60	4.37	11.20	6.00	6.53
Test strategy 4 - mean values					
Finder execution	710	1065.7	1454.9	1697.4	2280.1
Single substitution	427.9	472.3	516.4	514.7	575.2
Single resolution	218.4	294.8	399.7	463.7	671.5
Full substitution	646.3	767.1	916.1	978.4	1246.7
Test strategy 4 - standard deviations					
Finder execution	17.15	174.82	400.98	229.21	450.64
Single substitution	4.86	44.69	70.52	36.33	102.66

Continued on the next page

Table A.2 – *Continued from the previous page*

Durations (ms)	**300**	**350**	**400**	**450**	**500**
Single resolution	2.41	9.53	71.14	6.58	30.70
Full substitution	5.65	52.55	140.35	38.69	117.40

Table A.3: Mean values and standard deviations - part 3

Durations (ms)	**550**	**600**	**650**	**700**	**750**
Test strategy 1 - mean values					
Finder execution	3.6	4.2	4.3	10.3	3.7
Single substitution	481.8	528.2	598.2	501.1	497
Single resolution	2.4	2.4	2.6	2.7	2.4
Full substitution	484.2	530.6	600.8	503.8	499.4
Test strategy 1 - standard deviations					
Finder execution	0.96	1.13	1.25	4.98	1.25
Single substitution	60.59	79.00	49.83	69.66	46.50
Single resolution	0.69	1.26	0.84	0.82	0.69
Full substitution	60.80	79.79	49.96	69.78	46.59
Test strategy 2 - mean values					
Finder execution	5.5	5.6	5.6	5.5	5.7
Single substitution	790.8	687.4	694.8	716.5	739.7
Single resolution	18.3	15.7	15.1	16.8	17
Full substitution	809.1	703.1	709.9	733.3	756.7
Test strategy 2 - standard deviations					
Finder execution	0.97	2.27	0.96	1.26	0.82
Single substitution	66.36	105.41	20.63	36.63	22.29

Continued on the next page

Table A.3 – *Continued from the previous page*

Durations (ms)	**550**	**600**	**650**	**700**	**750**
Single resolution	2.62	2.35	0.56	1.03	1.33
Full substitution	67.79	106.88	20.61	37.00	22.47
Test strategy 3 - mean values					
Finder execution	194.4	220	256.3	292.8	359.8
Single substitution	499.6	509.5	538.1	570.7	665.9
Single resolution	13.1	23	15.3	16.3	14.8
Full substitution	512.7	532.5	553.4	587	680.7
Test strategy 3 - standard deviations					
Finder execution	15.03	1.88	3.91	3.73	19.17
Single substitution	35.25	10.71	3.54	24.42	56.91
Single resolution	0.56	1.82	0.82	0.94	1.13
Full substitution	35.20	10.70	3.47	24.88	56.85
Test strategy 4 - mean values					
Finder execution	2229	2642.5	3226.8	3405.3	2852.6
Single substitution	581.7	683.2	836.9	902.8	797.4
Single resolution	642.6	800.5	1001.1	1067.9	895.8
Full substitution	1224.3	1483.7	1838	1970.7	1693.2
Test strategy 4 - standard deviations					
Finder execution	90.46	429.25	45.90	52.40	516.43
Single substitution	35.37	101.42	31.41	27.59	167.05
Single resolution	25.79	119.71	32.68	30.23	93.31
Full substitution	56.40	218.84	40.31	30.43	253.84

Table A.4: Mean values and standard deviations - part 4

Durations (ms)	**800**	**850**	**900**	**950**	**1000**
Test strategy 1 - mean values					
Finder execution	3.7	3.6	4.3	4.4	3.7
Single substitution	469.3	481.2	570.8	570.2	540.3
Single resolution	2.4	12.2	2.5	2.5	2.6
Full substitution	471.7	493.4	573.3	572.7	542.9
Test strategy 1 - standard deviations					
Finder execution	0.48	0.51	0.67	1.77	0.67
Single substitution	5.12	3.32	61.18	72.99	9.83
Single resolution	0.51	0.42	0.52	0.52	0.51
Full substitution	5.55	3.30	61.16	73.14	10.11
Test strategy 2 - mean values					
Finder execution	5.8	6.9	6.3	6.4	6.2
Single substitution	806.9	867.5	934.2	980.8	1012.8
Single resolution	28.3	20.8	21.2	31.8	22
Full substitution	835.2	888.3	955.4	1012.6	1034.8
Test strategy 2 - standard deviations					
Finder execution	0.91	1.44	0.67	0.84	0.42
Single substitution	35.34	45.59	49.98	46.38	39.22
Single resolution	1.41	2.57	1.61	1.75	1.88
Full substitution	36.34	47.14	51.00	47.61	39.71
Test strategy 3 - mean values					
Finder execution	400.2	448.1	483	515.3	575.5
Single substitution	682.7	706.8	720.7	748.5	784

Continued on the next page

Table A.4 – *Continued from the previous page*

Durations (ms)	**800**	**850**	**900**	**950**	**1000**
Single resolution	26	17.8	18.7	17.7	28.1
Full substitution	708.7	724.6	739.4	766.2	812.1
Test strategy 3 - standard deviations					
Finder execution	10.89	13.37	18.34	8.90	29.81
Single substitution	25.55	20.49	38.16	64.04	51.43
Single resolution	1.69	2.04	1.63	2.31	1.91
Full substitution	25.95	20.95	38.46	65.61	52.82
Test strategy 4 - mean values					
Finder execution	2719.9	2901.5	2930.3	2988.3	3109.6
Single substitution	812.4	846.6	886.6	944.4	1027.3
Single resolution	944.5	1007	1056	1113.2	1148
Full substitution	1756.9	1853.6	1942.6	2057.6	2175.3
Test strategy 4 - standard deviations					
Finder execution	111.80	135.75	65.28	60.08	93.77
Single substitution	54.04	32.43	20.42	19.27	24.22
Single resolution	48.08	16.04	16.65	18.82	21.77
Full substitution	56.85	27.80	8.15	28.66	33.22

Bibliography

[1] Jiří Adámek, Horst Herrlich, and George Strecker. *Abstract and concrete categories*. New York, NY, USA: Wiley-Interscience, 1990. ISBN: 0-471-60922-6 (cit. on p. 22).

[2] Øystein Haugen Andreas Svendsen, Xiaorui Zhang and Birger Møller-Pedersen. 'Towards Evolution of Generic Variability Models.' In: *Variability for You*. 2011. URL: vary2011.irisa.fr (cit. on pp. 13 sq.).

[3] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. 'Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations.' In: *Model Driven Engineering Languages and Systems*. Ed. by DorinaC. Petriu, Nicolas Rouquette, and Øystein Haugen. Vol. 6394. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 121–135. ISBN: 978-3-642-16144-5. DOI: 10.1007/978-3-642-16145-2_9 (cit. on pp. 20, 26 sq.).

[4] Michael Barr and Charles Wells. 'Toposes, triples and theories.' In: *Reprints in Theory and Applications of Categories*. Citeseer. 2005 (cit. on p. 22).

[5] D. Batory, J.N. Sarvela, and A. Rauschmayer. 'Scaling step-wise refinement.' In: *Software Engineering, IEEE Transactions on* 30.6 (June), pp. 355–371. ISSN: 0098-5589. DOI: 10.1109/TSE.2004.23 (cit. on p. 87).

[6] Don Batory. 'Feature-Oriented Programming and the AHEAD Tool Suite.' In: *Proceedings of the 26th International Conference on Software Engineering*. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 702–703. ISBN: 0-7695-2163-0. URL: http://dl.acm.org/citation.cfm?id=998675.999478 (cit. on p. 87).

[7] Joachim Bayer, Sebastien Gerard, Øystein Haugen, Jason Mansell, Birger Møller-Pedersen, Jon Oldevik, Patrick Tessier, Jean-Philippe Thibault, and Tanya Widen. 'Consolidated Product Line Variability Modeling.' In: *Software Product Lines*. Ed. by Timo Käköla and Juan Carlos Duenas. Springer Berlin Heidelberg, 2006, pp. 195–241. ISBN: 978-3-540-33253-4. DOI: 10.1007/978-3-540-33253-4_6 (cit. on pp. 8, 11 sq.).

[8] D Benavides, S Segura, and A Ruiz-Cortés. 'Automated analysis of feature models 20 years later: A literature review.' In: *Information Systems* 35.6 (2010), pp. 615–636 (cit. on p. 9).

[9] CESAR. *CESAR: Cost-efficient methods and processes for safety relevant embedded systems*. [Online; accessed 17-February-2013]. 2012. URL: http://www.cesarproject.eu/index.php?id=9 (cit. on p. 31).

[10] Hyunsik Choi and Kyo Chul Kang. 'Towards Scalable and Conflict-Free Configuration of Features.' In: *SPLC Workshops*. 2010, pp. 197–204 (cit. on p. 86).

[11] N. Chomsky. 'Three models for the description of language.' In: *Information Theory, IRE Transactions on* 2.3 (1956), pp. 113 –124. ISSN: 0096-1000. DOI: 10.1109/TIT.1956.1056813 (cit. on p. 19).

[12] *Common Variability Language (CVL)*. OMG. OMG document: ad/2012-08-05. 2012. URL: http://www.omgwiki.org/variability/lib/exe/fetch.php?id=start&cache=cache&media=cvl-revised-submission.pdf (cit. on pp. 10, 13 sqq., 52).

[13] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. 'Algebraic approaches to graph transformation, Part I: Basic concepts and double pushout approach.' In: *Handbook of graph grammars and computing by graph transformation* 1 (1997), pp. 163–245 (cit. on pp. 22 sq., 25).

[14] Krzysztof Czarnecki, Ulrich Eisenecker, Robert Glück, David Vandev oorde, and Todd Veldhuizen. 'Generative Programming and Active Libraries.' In: *Generic Programming*. Ed. by Mehdi Jazayeri, Rüdiger Loos, and David Musser. Vol. 1766. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2000, pp. 25–39. ISBN: 978-3-540-41090-4. DOI: 10.1007/3-540-39953-4_3 (cit. on p. 9).

[15] Krzysztof Czarnecki, Paul Gruenbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wasowski. 'Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches.' In: *Variability Modelling of Software-intensive Systems (VaMoS)*. ACM Press. Leipzig, Germany: ACM Press, 2012 (cit. on pp. 10 sq.).

[16] Krzysztof Czarnecki and Andrzej Wasowski. 'Feature Diagrams and Logics: There and Back Again.' In: *11th International Software Product Line Conference (SPLC 2007)*. IEEE, Sept. 2007, pp. 23–34. ISBN: 0-7695-2888-0. DOI: 10.1109/SPLINE.2007.24. URL: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4339252 (cit. on p. 10).

[17] Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. 'Specifying Overlaps of Heterogeneous Models for Global Consistency Checking.' In: *Models in Software Engineering*. Ed. by Juergen Dingel and Arnor Solberg. Vol. 6627. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 165–179. ISBN: 978-3-642-21209-3. DOI: 10.1007/978-3-642-21210-9_16 (cit. on p. 87).

[18] H Ehrig, M Pfender, and H J Schneider. 'Graph-grammars: An algebraic approach.' In: *Switching and Automata Theory, 1973. SWAT '08. IEEE Conference Record of 14th Annual Symposium on*. 1973, pp. 167–180. DOI: 10.1109/SWAT.1973.11 (cit. on pp. 21, 23).

[19] Hartmut Ehrig. 'Introduction to the algebraic theory of graph grammars (a survey).' In: *Graph-Grammars and Their Application to Computer Science and Biology*. Ed. by Volker Claus, Hartmut Ehrig, and Grzegorz Rozenberg. Vol. 73. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1979, pp. 1–69. ISBN: 978-3-540-09525-5. DOI: 10.1007/BFb0025714 (cit. on p. 84).

[20] Roy Grønmo. 'Using Concrete Syntax in Graph-based Model Transformations.' PhD thesis. University of Oslo, 2010 (cit. on p. 88).

[21] Ø Haugen, B Møller-Pedersen, J Oldevik, G K Olsen, and A Svendsen. 'Adding Standardized Variability to Domain Specific Languages.' In: *Software Product Line Conference, 2008. SPLC '08. 12th International*. 2008, pp. 139–148. DOI: 10.1109/SPLC.2008.25 (cit. on pp. 9 sq., 12).

[22] Øystein Haugen. *CVL Tool from SINTEF*. 2010. URL: http://www.omgwiki.org/variability/doku.php?id=cvl_tool_from_sintef (cit. on pp. 2, 12, 45, 47, 83).

[23] Øystein Haugen. *Welcome to the Common Variability Language Wiki*. 2010. URL: http://www.omgwiki.org/variability/doku.php?id=introduction_to_variability_modeling&rev=1248774285 (cit. on p. 8).

[24] Øystein Haugen, Birger Møller-Pedersen, and Jon Oldevik. 'Comparison of System Family Modeling Approaches.' In: *SPLC*. 2005, pp. 102–112 (cit. on p. 11).

[25] Øystein Haugen, Andrzej Wasowski, and Krzysztof Czarnecki. 'CVL: common variability language.' In: *Proceedings of the 16th International Software Product Line Conference - Volume 2*. SPLC '12. New York, NY, USA: ACM, 2012, pp. 266–267. ISBN: 978-1-4503-1095-6. DOI: 10.1145/2364412.2364462 (cit. on p. 10).

[26] Reiko Heckel. 'Graph Transformation in a Nutshell.' In: *Electronic Notes in Theoretical Computer Science* 148.1 (2006), pp. 187–198. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2005.12.018. URL: http://www.sciencedirect.com/science/article/pii/S157106610600048X (cit. on pp. 19, 21, 25).

[27] Reiko Heckel, Jochen Küster, and Gabriele Taentzer. 'Confluence of Typed Attributed Graph Transformation Systems.' In: *Graph Transformation*. Ed. by Andrea Corradini, Hartmut Ehrig, Hans Kreowski, and Grzegorz Rozenberg. Vol. 2505. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2002, pp. 161–176. ISBN: 978-3-540-44310-0. URL: http://dx.doi.org/10.1007/3-540-45832-8_14 (cit. on p. 84).

[28] Arnaud Hubaux, Yingfei Xiong, and Krzysztof Czarnecki. 'A user survey of configuration challenges in Linux and eCos.' In: *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*. VaMoS '12. New York, NY, USA: ACM, 2012, pp. 149–155. ISBN: 978-1-4503-1058-1. DOI: 10.1145/2110147.2110164 (cit. on p. 86).

[29] Gérard Huet. 'Confluent reductions: Abstract properties and applications to term rewriting systems: Abstract properties and applications to term rewriting systems.' In: *Journal of the ACM (JACM)* 27.4 (1980), pp. 797–821 (cit. on p. 84).

[30] Timo Käkölä and Juan Carlos Dueñas. *Software Product Lines: Research Issues in Engineering and Management*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006. ISBN: 3540332529 (cit. on p. 8).

[31] K C Kang, S G Cohen, J A Hess, W E Novak, and A S Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. rep. CMU/SEI-90-TR-21 ESD-90-TR-222. Pittsburgh, Pennsylvania 15213: Carnegie Mellon University, 1990. URL: http://wwwiti.cs.uni-magdeburg.de/iti_db/lehre/epmd/2007/bib/foda.pdf (cit. on p. 9).

[32] S Kelly and J P Tolvanen. *Domain-specific modeling: enabling full code generation*. Wiley-IEEE Computer Society Pr, 2008 (cit. on p. 11).

[33] Lilija Klassen and Robert Wagner. 'EMorF-A tool for model transformations.' In: *Electronic Communications of the EASST* 54 (2012) (cit. on pp. 26 sq.).

[34] Donald E Knuth and Peter B Bendix. 'Simple word problems in universal algebras.' In: *Computational problems in abstract algebra*. Vol. 263. 1970, p. 297 (cit. on p. 84).

[35] J. Lauret, H. Waeselynck, and J.-C. Fabre. 'Detection of Interferences in Aspect-Oriented Programs Using Executable Assertions.' In: *Software Reliability Engineering Workshops (ISSREW), 2012 IEEE 23rd International Symposium on*. Nov. Pp. 165–170. DOI: 10.1109/ISSREW.2012.34 (cit. on p. 88).

[36] Michael Löe. 'Algebraic approach to single-pushout graph transformation.' In: *Theoretical Computer Science* 109.1 (1993), pp. 181 –224. ISSN: 0304-3975. DOI: 10.1016/0304-3975(93)90068-5. URL: http://www.sciencedirect.com/science/article/pii/0304397593900685 (cit. on pp. 21, 24).

[37] Brice Morin, Gilles Perrouin, Philippe Lahire, Olivier Barais, Gilles Vanwormhoudt, and Jean-Marc Jezequel. 'Weaving Variability into Domain Metamodels.' In: *MODEL DRIVEN ENGINEERING LANGUAGES AND SYSTEMS, PROCEEDINGS*. Ed. by B Schurr, A and Selic. Vol. 5795. Lecture Notes in Computer Science. ACM; Springer; IEEE; Aerospace. HEIDELBERGER PLATZ 3, D-14197 BERLIN, GERMANY: SPRINGER-VERLAG BERLIN, 2009, pp. 690–705. ISBN: 978-3-642-04424-3 (cit. on p. 12).

[38] Jon Oldevik and Øystein Haugen. 'Higher-Order Transformations for Product Lines.' In: *Software Product Line Conference, International* 0 (2007), pp. 243–254 (cit. on p. 12).

[39] Jon Oldevik, Øystein Haugen, and Birger Møller-Pedersen. 'Confluence in Domain-Independent Product Line Transformations.' In: *Fundamental Approaches to Software Engineering*. Ed. by Marsha Chechik and Martin Wirsing. Vol. 5503. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2009, pp. 34–48. ISBN: 978-3-642-00592-3. DOI: 10.1007/978-3-642-00593-0_3 (cit. on pp. 83 sq., 86).

[40] Jon Oldevik, Tor Neple, Roy Grønmo, Jan Aagedal, and Arne-J. Berre. 'Toward Standardised Model to Text Transformations.' In: *Model Driven Architecture - Foundations and Applications*. Ed. by Alan Hartman and David Kreische. Vol. 3748. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 239–253. ISBN: 978-3-540-30026-7. DOI: 10.1007/11581741_18 (cit. on p. 12).

[41] OMG. *Unified Modeling Language: Superstructure*. version 2.0, formal/05-07-04. 2005. URL: http://www.omg.org/spec/UML/2.0/Superstructure/PDF (cit. on pp. 1 sq.).

[42] *OMG Meta Object Facility (MOF) Core Specification*. OMG. OMG Document Number: formal/2011-08-07. 2011. URL: http://www.omg.org/spec/MOF/2.4.1 (cit. on p. 61).

[43] John L. Pfaltz and Azriel Rosenfeld. 'Web grammars.' In: *Proceedings of the 1st international joint conference on Artificial intelligence*. IJCAI'69. Washington, DC: Morgan Kaufmann Publishers Inc., 1969, pp. 609–619. URL: http://dl.acm.org/citation.cfm?id=1624562.1624616 (cit. on p. 19).

[44] Detlef Plump. 'Confluence of graph transformation revisited.' In: *Processes, Terms and Cycles: Steps on the Road to Infinity* (2005), pp. 280–308 (cit. on pp. 40, 84).

[45] Klaus Pohl, Gönter Böckle, and Frank J van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. 1st. Springer Publishing Company, Incorporated, 2010. ISBN: 3642063640, 9783642063640 (cit. on pp. 7, 10).

[46] Terrence W. Pratt. 'Pair grammars, graph languages and string-to-graph translations.' In: *J. Comput. Syst. Sci.* 5.6 (Dec. 1971), pp. 560–595. ISSN: 0022-0000. DOI: 10.1016/S0022-0000(71)80016-8 (cit. on p. 19).

[47] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. 'Delta-Oriented Programming of Software Product Lines.' In: *Software Product Lines: Going Beyond*. Ed. by Jan Bosch and Jaejoon Lee. Vol. 6287. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 77–91. ISBN: 978-3-642-15578-9. DOI: 10.1007/978-3-642-15579-6_6 (cit. on p. 87).

[48] Klaus Schmid, Rick Rabiser, and Paul Grünbacher. 'A comparison of decision modeling approaches in product lines.' In: *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*. VaMoS '11. New York, NY, USA: ACM, 2011, pp. 119–126. ISBN: 978-1-4503-0570-9. DOI: 10.1145/1944892.1944907 (cit. on pp. 10 sq.).

[49] A. Schürr, A. J. Winter, and A. Zündorf. 'Handbook of graph grammars and computing by graph transformation.' In: ed. by H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 1999. Chap. The PROGRES approach: language and environment, pp. 487–550. ISBN: 981-02-4020-1. URL: http://dl.acm.org/citation.cfm?id=328523.328617 (cit. on p. 25).

[50] Andy Schürr. 'Programmed Graph Replacement Systems.' In: *In Rozenberg, G. (Ed.), Handbook on Graph Grammars: Foundations*. World Scientific, 1997, pp. 479–546 (cit. on p. 25).

[51] Andy Schürr. 'Specification of graph translators with triple graph grammars.' In: *Graph-Theoretic Concepts in Computer Science*. Ed. by ErnstW. Mayr, Gunther Schmidt, and Gottfried Tinhofer. Vol. 903. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1995, pp. 151–163. ISBN: 978-3-540-59071-2. DOI: 10.1007/3-540-59071-4_45 (cit. on pp. 21, 86).

[52] Andy Schürr and Felix Klar. '15 Years of Triple Graph Grammars.' In: *Graph Transformations*. Ed. by Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer. Vol. 5214. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 411–425. ISBN: 978-3-540-87404-1. DOI: 10.1007/978-3-540-87405-8_28 (cit. on pp. 21, 86).

[53] Robert Sedgewick and Philippe Flajolet. *An introduction to the analysis of algorithms*. Pearson Education India, 1996 (cit. on p. 79).

[54] Andreas Svendsen, Gøran K. Olsen, Jan Endresen, Thomas Moen, Erik Carlson, Kjell-Joar Alme, and Øystein Haugen. 'The Future of Train Signaling.' In: *Model Driven Engineering Languages and Systems*. Ed. by Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter. Vol. 5301. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 128–142. ISBN: 978-3-540-87874-2. DOI: 10.1007/978-3-540-87875-9_9 (cit. on pp. 12, 83).

[55] Andreas Svendsen, Xiaorui Zhang, Franck Fleurey, Øystein Haugen, Gøran K. Olsen, and Birger Møller-Pedersen. 'CVL Tool - Modeling Variability in SPLs.' In: *SPLC Workshops*. Ed. by Goetz Botterweck, Stan Jarzabek, Tomoji Kishi, Jaejoon Lee, and Steve Livengood. Lancaster University, 2010, p. 299. ISBN: 978-1-86220-274-0 (cit. on pp. 12, 83).

[56] Andreas Svendsen, Xiaorui Zhang, Roy Lind-Tviberg, Franck Fleurey, Øystein Haugen, Birger Møller-Pedersen, and Gøran K Olsen. 'Developing a Software Product Line for Train Control: A Case Study of CVL.' In: *SPLC*. 2010, pp. 106–120 (cit. on p. 13).

[57] Software Productivity Consortium Services Corporation Technical Report SPC-92019-CMC. *Reuse-Driven Software Processes Guidebook, Version 02.00.03*. 1993 (cit. on pp. 10 sq.).

[58] 'The American Heritage® Dictionary of the English Language, Fourth Edition.' In: (2013). URL: http://dictionary.reference.com/browse/influence (cit. on p. 7).

[59] A Van Deursen, P Klint, and J Visser. 'Domain-specific languages.' In: *Centrum voor Wiskunde en Informatika* (2000) (cit. on p. 11).

[60] M Voelter and E Visser. 'Product Line Engineering Using Domain-Specific Languages.' In: *Software Product Line Conference (SPLC), 2011 15th International*. 2011, pp. 70–79. DOI: 10.1109/SPLC.2011.25 (cit. on p. 9).

[61] Wikipedia. *Graph rewriting — Wikipedia, The Free Encyclopedia*. [Online; accessed 17-February-2013]. 2013. URL: http://en.wikipedia.org/w/index.php?title=Graph_rewriting&oldid=538577202 (cit. on p. 25).

[62] Yingfei Xiong, A Hubaux, S She, and K Czarnecki. 'Generating range fixes for software configuration.' In: *Software Engineering (ICSE), 2012 34th International Conference on*. June 2012, pp. 58–68. DOI: 10.1109/ICSE.2012.6227206 (cit. on p. 86).