A Reference Structure for Modular Model-based Analyses

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Sandro Giovanni Koch

aus Lindenberg i. Allgäu

Tag der mündlichen Prüfung: 07. Juli 2023
Erster Gutachter: Prof. Dr. Ralf H. Reussner
Zweiter Gutachter: Prof. Dr. Bernhard Rumpe

Danksagung

Es ist geschafft! Mit großer Freude und Dankbarkeit möchte ich meine Anerkennung aussprechen. Mein aufrichtiger Dank gilt Ralf, dessen unermüdliche Unterstützung, Leitung und Expertise maßgeblich dazu beigetragen haben, dass ich meine Dissertation erfolgreich abschließen konnte. Unsere inspirierenden Gespräche und sein Engagement haben meine akademische, aber auch meine persönliche Reise bereichert. Ebenfalls möchte ich meine aufrichtige Dankbarkeit gegenüber Robert ausdrücken, der mir in zahlreichen Gesprächen und Publikationsprojekten mit Rat und Tat zur Seite stand.

Ein besonderer Dank geht an meine Lebensgefährtin Miri, die bedingungslos an meiner Seite stand, mich ermutigte, wenn die Hürden schier unüberwindbar schienen, und immer Verständnis für die unzähligen Stunden hatte, die ich dieser Dissertation gewidmet habe. Ihre Geduld und Nachsicht waren eine unverzichtbare Stütze, die mich zuversichtlich voranschreiten ließ.

Auch meinen Eltern und meiner Schwester möchte ich von ganzem Herzen danken. Ihre bedingungslose Unterstützung und grenzenlose Liebe haben mich durch Höhen und Tiefen getragen. Ohne ihre stetige Ermutigung und Zuversicht hätte ich diesen Meilenstein nicht erreichen können.

Ein herzlicher Dank geht auch an meinen Freund Marco, dessen motivierende Worte und fester Glaube an mich mich dazu inspirierten, meinen Weg konsequent zu verfolgen.

Auch meinen geschätzten Kollegen möchte ich meinen Dank aussprechen. Die vielen Gespräche über Gott und die Welt, die moralische Unterstützung und der fachliche Austausch haben meinen akademischen Weg bereichert und mir geholfen, neue Perspektiven einzunehmen. Kiana und Axel möchte ich für ihre Unterstützung, auch über die Zeit am Lehrstuhl hinaus, danken. Dominik und Stephan bin ich dankbar dafür, dass sie meinen kulinarischen Horizont erweitert und mir meine Grenzen aufgezeigt haben. Danke auch an Heiko für die offene Tür und die vielen Gespräche in der Mittagspause und nach Feierabend. Mein besonderer Dank gilt Jan, Frederik, Yves, Tobias, Dominik, Maximilian und Nicolas, die mit mir zusammen die magischen Welten von Ikoria bis Zendikar erkundeten und so die langen Stunden erträglicher gemacht haben.

Abschließend möchte ich all jenen danken, die mich auf diesem Weg begleitet haben, sei es durch persönliche Begegnungen, aufmunternde Worte oder positive Gedanken. Ihr alle habt dazu beigetragen, dass diese Dissertation zu einem Erfolg wurde.

Abstract

Context: In this thesis, we investigated the evolvability, understandability, and reusability of model-based analyses. For this purpose, we studied the co-dependency between models and analyses, particularly the structure and interdependence of artefacts and the feature-based decomposition and composition of model-based analyses.

Challenges: Software developers use models of software systems to determine the evolvability and reusability of an architectural design. These models enable them to analyse the software architecture before writing the first line of code. However, due to evolutionary changes, model-based analyses are also prone to the deterioration of evolvability, understandability, and reusability. These problems can be traced back to the co-evolution of the modelling language and the analysis. The purpose of an analysis is a systematic examination or study of specific properties of a system under study. For example, suppose software developers want to analyse new properties of a software system. In that case, they must adapt features of the modelling language and the corresponding model-based analyses before they can analyse new properties. Features in the context of the modelling language are, for example, quality properties such as performance or reliability. Features in a model-based analysis are, for example, an analysis technique that analyses such a quality property. Such changes lead to the increased complexity of the model-based analyses and, as a result, to difficult-to-maintain model-based analyses. This increasing complexity reduces the understandability of the model-based analyses. As a result, development cycles lengthen, and software developers need more time to adapt the software system to changing requirements.

State-of-the-Art: Current approaches allow the coupling of analyses on one system or across distributed systems. These approaches provide the technical structure for coupling simulations rather than a structure and process for how components can be (de)composed. Another challenge in composing analyses is the behavioural aspect, in which the analysis component influences what. By synchronising each participating simulation, the decomposition of simulations increases the need for communication. State-of-the-art approaches reduce communication overhead; however, decomposition and composition are left to the user. There are also several approaches to modelling variability that use feature diagrams to be able to use product line techniques. However, these approaches must provide a process for identifying and structuring analysis features.

Contributions: This thesis aims to improve model-based analysis's evolvability, understandability, and reusability. For this purpose, we take the reference architecture for domain-specific modelling languages as a basis and investigate the transferability of the structure of the reference architecture to model-based analyses. The layered reference architecture maps the dependencies of the analysis features and components by assigning them to specific layers. We developed three processes for applying the reference architecture: (i) refactoring an existing model-based analysis, (ii) designing a new model-based analysis, and (iii) extending an existing model-based analysis. In addition to the reference architecture for model-based analyses, we have identified recurring structures that lead to problems in evolvability, understandability and reusability; in the literature, these recurring structures are also called bad smells. In particular, we have investigated the co-dependency of Domain-specific Modelling Languages (DSMLs) and model-based analyses that lead to these recurring problems. So far, bad smells for DSMLs and source code have been considered separately, although they are co-dependent. We examined established model-based analyses and identified and specified thirteen bad smells. In addition to specifying the bad smells, we provide a process for automatically identifying them and strategies for refactoring them so that developers can avoid or fix them. We also developed a modelling language for specifying this thesis's structure and behaviour of analysis components. Simulations are analyses to investigate a system when experimenting with the existing system is too time-consuming, costly, dangerous or simply impossible because the system does not exist (yet). Developers can use the specification to compare simulation components and thus identify identical components. Finding similar simulation components allows developers to reuse existing components and reduce the effort required to develop new components.

Validation: We evaluated our first contribution, the reference architecture for model-based analyses, by applying it to four model-based analyses. We chose a scenario-based evaluation that derives historical change scenarios from the repositories of the model-based analyses. In the evaluation, we can show that evolvability and understandability improve by determining the complexity, coupling, and cohesion. The metrics we used originate from information theory but were already used to evaluate the reference architecture for DSMLs. We evaluated our second contribution, the bad smells that emerge due to the co-dependency of model-based analyses and their corresponding DSMLs, by searching four model-based analyses for occurrences of our bad smells and fixing the found bad smells. We also chose a scenario-based evaluation that derives historical change scenarios from the repositories of the model-based analyses. We can show that bad smells negatively affect evolvability and understandability by determining the complexity, coupling, and cohesion before and after the refactoring. We evaluated our third contribution, the approach to specify and find components of model-based analyses, by specifying components of two model-based analyses and by applying our search algorithm to find analysis components with similar structure and behaviour. The evaluation results show that we can find similar analysis components and, as a result, that our approach can ease the search for analysis components with similar structure and behaviour. Thus, it can ease the reuse of such components.

Benefits: The contributions of our work support architects and developers in their day-to-day work to develop maintainable and reusable model-based analyses. For this purpose, we provide a reference architecture that aligns model-based analysis with the domain-specific modelling language, thus facilitating co-evolution. In addition to the reference architecture, we provide refactoring operations that allow architects and developers to align an existing model-based analysis with the reference architecture. In addition to this technical aspect, we have identified three processes that enable architects and developers to develop a new model-based analysis, modularise an existing model-based analysis and extend an existing model-based analysis. Of course, this is done so that the results conform to the reference architecture. In addition, our specification allows developers to compare existing simulation components and reuse them as needed. This avoids the need for developers to re-implement components.

Zusammenfassung

Kontext: In dieser Arbeit haben wir die Evolvierbarkeit, Verständlichkeit und Wiederverwendbarkeit von modellbasierten Analysen untersucht. Darum untersuchten wir die Wechselbeziehungen zwischen Modellen und Analysen, insbesondere die Struktur und Abhängigkeiten von Artefakten und die Dekomposition und Komposition von modellbasierten Analysen.

Herausforderungen: Softwareentwickler verwenden Modelle von Softwaresystemen, um die Evolvierbarkeit und Wiederverwendbarkeit eines Architekturentwurfs zu bestimmen. Diese Modelle ermöglichen die Softwarearchitektur zu analysieren, bevor die erste Zeile Code geschreiben wird. Aufgrund evolutionärer Veränderungen sind modellbasierte Analysen jedoch auch anfällig für eine Verschlechterung der Evolvierbarkeit, Verständlichkeit und Wiederverwendbarkeit. Diese Probleme lassen sich auf die Ko-Evolution von Modellierungssprache und Analyse zurückführen. Der Zweck einer Analyse ist die systematische Untersuchung bestimmter Eigenschaften eines zu untersuchenden Systems. Nehmen wir zum Beispiel an, dass Softwareentwickler neue Eigenschaften eines Softwaresystems analysieren wollen. In diesem Fall müssen sie Merkmale der Modellierungssprache und die entsprechenden modellbasierten Analysen anpassen, bevor sie neue Eigenschaften analysieren können. Merkmale in einer modellbasierten Analyse sind z. B. eine Analysetechnik, die eine solche Qualitätseigenschaft analysiert. Solche Änderungen führen zu einer erhöhten Komplexität der modellbasierten Analysen und damit zu schwer zu pflegenden modellbasierten Analysen. Diese steigende Komplexität verringert die Verständlichkeit der modellbasierten Analysen. Infolgedessen verlängern sich die Entwicklungszyklen, und die Softwareentwickler benötigen mehr Zeit, um das Softwaresystem an veränderte Anforderungen anzupassen.

Stand der Technik: Derzeitige Ansätze ermöglichen die Kopplung von Analysen auf einem System oder über verteilte Systeme hinweg. Diese Ansätze bieten die technische Struktur für die Kopplung von Simulationen, nicht aber eine Struktur wie Komponenten (de)komponiert werden können. Eine weitere Herausforderung beim Komponieren von Analysen ist der Verhaltensaspekt, der sich darin äußert, wie sich die Analysekomponenten gegenseitig beeinflussen. Durch die Synchronisierung jeder beteiligten Simulation erhöht die Modularisierung von Simulationen den Kommunikationsbedarf. Derzeitige Ansätze erlauben es, den Kommunikationsaufwand zu reduzieren; allerdings werden bei diesen Ansätzen die Dekomposition und Komposition dem Benutzer überlassen.

Beiträge: Ziel dieser Arbeit ist es, die Evolvierbarkeit, Verständlichkeit und Wiederverwendbarkeit von modellbasierten Analysen zu verbessern. Zu diesem Zweck wird die

Referenzarchitektur für domänenspezifische Modellierungssprachen als Grundlage genommen und die Übertragbarkeit der Struktur der Referenzarchitektur auf modellbasierte Analysen untersucht. Die geschichtete Referenzarchitektur bildet die Abhängigkeiten der Analysefunktionen und Analysekomponenten ab, indem sie diese bestimmten Schichten zuordnet. Wir haben drei Prozesse für die Anwendung der Referenzarchitektur entwickelt: (i) Refactoring einer bestehenden modellbasierten Analyse, (ii) Entwurf einer neuen modellbasierten Analyse und (iii) Erweiterung einer bestehenden modellbasierten Analyse. Zusätzlich zur Referenzarchitektur für modellbasierte Analysen haben wir wiederkehrende Strukturen identifiziert, die zu Problemen bei der Evolvierbarkeit, Verständlichkeit und Wiederverwendbarkeit führen; in der Literatur werden diese wiederkehrenden Strukturen auch als Bad Smells bezeichnet. Wir haben etablierte modellbasierte Analysen untersucht und dreizehn Bad Smells identifiziert und spezifiziert. Neben der Spezifizierung der Bad Smells bieten wir einen Prozess zur automatischen Identifizierung dieser Bad Smells und Strategien für deren Refactoring, damit Entwickler diese Bad Smells vermeiden oder beheben können. In dieser Arbeit haben wir auch eine Modellierungssprache zur Spezifikation der Struktur und des Verhaltens von Simulationskomponenten entwickelt. Simulationen sind Analysen, um ein System zu untersuchen, wenn das Experimentieren mit dem bestehenden System zu zeitaufwändig, zu teuer, zu gefährlich oder einfach unmöglich ist, weil das System (noch) nicht existiert. Entwickler können die Spezifikation nutzen, um Simulationskomponenten zu vergleichen und so identische Komponenten zu identifizieren.

Validierung: Die Referenzarchitektur für modellbasierte Analysen, haben wir evaluiert, indem wir vier modellbasierte Analysen in die Referenzarchitektur überführt haben. Wir haben eine szenariobasierte Evaluierung gewählt, die historische Änderungsszenarien aus den Repositories der modellbasierten Analysen ableitet. In der Auswertung können wir zeigen, dass sich die Evolvierbarkeit und Verständlichkeit durch die Bestimmung der Komplexität, der Kopplung und der Kohäsion verbessert. Die von uns verwendeten Metriken stammen aus der Informationstheorie, wurden aber bereits zur Bewertung der Referenzarchitektur für DSMLs verwendet. Die Bad Smells, die durch die Co-Abhängigkeit von modellbasierten Analysen und ihren entsprechenden DSMLs entstehen, haben wir evaluiert, indem wir vier modellbasierte Analysen nach dem Auftreten unserer schlechten Gerüche durchsucht und dann die gefundenen Bad Smells behoben haben. Wir haben auch eine szenariobasierte Auswertung gewählt, die historische Änderungsszenarien aus den Repositories der modellbasierten Analysen ableitet. Wir können zeigen, dass die Bad Smells die Evolvierbarkeit und Verständlichkeit negativ beeinflussen, indem wir die Komplexität, Kopplung und Kohäsion vor und nach der Refaktorisierung bestimmen. Den Ansatz zum Spezifizieren und Finden von Komponenten modellbasierter Analysen haben wir evaluiert, indem wir Komponenten von zwei modellbasierten Analysen spezifizieren und unseren Suchalgorithmus verwenden, um ähnliche Analysekomponenten zu finden. Die Ergebnisse der Evaluierung zeigen, dass wir in der Lage sind, ähnliche Analysekomponenten zu finden und dass unser Ansatz die Suche nach Analysekomponenten mit ähnlicher Struktur und ähnlichem Verhalten und damit die Wiederverwendung solcher Komponenten ermöglicht.

Nutzen: Die Beiträge unserer Arbeit unterstützen Architekten und Entwickler bei ihrer täglichen Arbeit, um wartbare und wiederverwendbare modellbasierte Analysen zu entwickeln. Zu diesem Zweck stellen wir eine Referenzarchitektur bereit, die die modellbasierte Analyse und die domänenspezifische Modellierungssprache aufeinander abstimmt und so die Koevolution erleichtert. Zusätzlich zur Referenzarchitektur bieten wir auch Refaktorisierungsoperationen an, die es Architekten und Entwicklern ermöglichen, eine bestehende modellbasierte Analyse an die Referenzarchitektur anzupassen. Zusätzlich zu diesem technischen Aspekt haben wir drei Prozesse identifiziert, die es Architekten und Entwicklern ermöglichen, eine neue modellbasierte Analyse zu entwickeln, eine bestehende modellbasierte Analyse zu modularisieren und eine bestehende modellbasierte Analyse zu erweitern. Dies geschieht natürlich so, dass die Ergebnisse mit der Referenzarchitektur konform sind. Darüber hinaus ermöglicht unsere Spezifikation den Entwicklern, bestehende Simulationskomponenten zu vergleichen und sie bei Bedarf wiederzuverwenden. Dies erspart den Entwicklern die Neuimplementierung von Komponenten.

Contents Overview

Contents

List of Figures

List of Tables

List of Listings

Acronyms

API Application Programming Interface. 87

aPS automated Production System. 163, 165, 180, 182, 183, 202

AS Automated System. 163

AST Abstract Syntax Tree. 38

BERT Bidirectional Encoder Representations from Transformers. 245, 246

BPMN Business Process Modeling Notation. 178, 179

BPMN2 Business Process Modeling Notation 2. 71, 107, 112, 114, 118, 119, 157, 161, 163, 176, 178

CLI Command Line Interface. 87, 144, 147, 149

CoDES Composable Discrete-Event Scalable Simulation. 242, 251

CPU Central Processing Unit. 160

CuBERT Code Understanding BERT. 245

DCRA Duplicated Code Refactoring Advisor. 247, 248

DES Discrete-event Simulation. 130–133, 149–151, 223, 224, 226, 227, 230, 234, 235, 249, 262

DEVS Discrete Event System Specification. 242, 249, 251

DI Dependency Injection. 245

DIS Distributed Interactive Simulation. 5, 241, 250

DSL Domain-specific Language. 35, 38, 80, 86, 127, 130, 133, 139, 140, 144, 234, 243, 244, 261, 262

DSML Domain-specific Modelling Language. xxiii, xxiv, 3–9, 11–17, 21, 22, 24, 25, 29–32, 41–50, 52–55, 57–63, 69, 70, 73, 74, 77, 78, 80, 84, 86–89, 93–95, 97, 99–108, 110–112, 114–125, 130, 157–159, 161–168, 170, 176, 179, 180, 183, 185, 186, 192, 194, 197, 199–203, 206, 207, 211, 218, 219, 221, 223, 224, 230, 231, 233, 239, 240, 243–249, 251–257, 259–262

EBNF Extended Backus–Naur Form. 37, 133

EMF Eclipse Modelling Framework. 31, 37, 111, 125, 144, 145, 243

EMOF Essential Meta-Object Facility. 16, 17, 37, 256, 259

EPC Event-driven Process Chains. 161

ETI Electronic Tool Integration. 241, 250

FMI Functional Mock-up Interface. 5, 241, 250

FoAA Field of Activity Annotations. 165

FOM Federate Object Model. 250

FST Feature Structure Tree. 23, 24, 67

GQM Goal Question Metric. 35–37, 168

HDD Hard Disk Drive. 160

HLA High-Level Architecture. 241, 250

IDE Integrated Development Environment. 25, 26, 63, 99–101, 115, 116, 125, 244

IEC International Electrotechnical Commission. 20, 169, 198

ISO International Organization for Standardization. 20, 161, 169, 198

KAMP Karlsruhe Architecture Maintainability Prediction. 110, 111, 163

KAMP4aPS Karlsruhe Architecture Maintainability Prediction for Automated Production Systems. 157, 158, 163, 165, 180, 182–184, 186, 188, 190, 195, 201, 205, 208, 212, 214, 216, 255, 258

LOP Language Oriented Programming. 244

MMRUC3 Move Method Refactoring Using Coupling, Cohesion, and Contextual Similarity. 246

MOF Meta-Object Facility. 37

MOOS Measure Of Software Similarity. 248

MPM Multi-Paradigm Modelling. 242

MPS Meta Programming System. 244

NLP Natural Language Processing. 245

NP Nondeterministic Polynomial Time. 33, 34

NTM Nondeterministic Turing Machine. 34

OMG Object Management Group. 161

ONTOCEAN Ontology for Code smell Analysis. 246, 247

OOP Object-Oriented Programming. 24, 42, 58

OSORE Ontology for Software Refactoring. 246, 247

P Polynomial Time. 33

PCM Palladio Component Model. 46, 74, 97, 101, 106, 110, 157–159, 171, 173–175, 201, 220, 227

PLC Programmable Logic Controller. 163

PPU Pick and Place Unit. 163

RD Resource Demand. 228

RESYS Refactoring Recommender System. 246, 247

SEFF Service Effect Specification. 160, 174, 228

SMT Satisfiable Modulo Theories. 32, 33, 128, 130, 137–143, 146, 147, 225, 250

SVN Apache Subversion. 176

SysML Systems Modelling Language. 11, 21

UI User Interface. 87, 146

UML Unified Modelling Language. 11, 21, 244, 247

"The choice of model affects the flexibility and reusability of the resulting system."

– Martin Fowler, *Analysis Patterns*

Part I.

Prologue

1. Introduction

In this thesis, we investigate the co-dependency of Domain-specific Modelling Languages (DSMLs) and model-based analyses, significantly how the structure of the DSML can affect the evolvability, understandability, and reusability of its corresponding model-based analyses. Co-dependency refers to the relationship between model-based analyses and DSMLs, where a change in the DSML affects the functionality or behaviour of the model-based analysis. Model-based analyses and DSMLs are tightly coupled, meaning they rely heavily on one another and cannot function independently. We present a reference architecture for model-based analyses that provides a guideline for (i) decomposing an already existing model-based analysis, (ii) composing a model-based analysis, and (iii) developing a model-based analysis from scratch. Also, we present newly identified and specified bad smells that arise from the co-dependency of DSMLs and model-based analyses. Furthermore, we present an approach to specify components of model-based analyses regarding their structure and behaviour to use these specifications to improve the reusability of model-based analysis components. This chapter is structured as follows: In Section 1.1, we motivate why considering the co-dependency of DSMLs and model-based analyses is a relevant subject for improving the evolvability, understandability, and reusability of model-based analyses. In Section 1.2, we present problems when considering the co-dependency of DSMLs and model-based analyses. How we answered our research questions is presented in Section 1.3 in the form of our contributions. We present the outline of this thesis in Section 1.4.

1.1 Deteriorating Evolvability, Understandability, and Reusability of Model-based Analyses

The internal quality of a software system affects its evolvability, understandability, and reusability [ISO10]. Historically grown software systems are prime examples where changes made during the software system's lifetime successively reduce its internal quality. Reduced evolvability, understandability, and reusability of a software system mean longer development cycles and delayed implementation of necessary changes, like innovations, regulations, or to a lesser degree, trends. To counteract the deterioration of internal software quality, software developers have access to analyses that allows them to investigate the internal software quality before implementing changes. Modelling the software system with the planned changes allows for analysing the effect on the internal quality, which leads to a better overall quality of the software system. Adverse effects on the internal software quality, like increased complexity or security breaches, can be predicted and, if determined as harmful, also be avoided.

Analyses are built to answer questions about specific properties of a system under study. Such an analysis usually does not take the real system as input; instead, it reasons about a model of the system [Tal+21b]. Such analyses are called model-based analyses; in the context of this thesis, do these analyses derive and communicate insights on the quality of a software system by using modelling languages and models of software systems [ZMK18]. Model-based analyses are also software systems; ergo, they are prone to the deterioration of their internal quality. Besides the model-based analyses, their input models deteriorate over time [HSR19]. Due to changed or new requirements, the DSMLs of these input models have to evolve over time; for example, when new properties of the system under study are added to the DSML. If the model-based analyses that work with the DSML are not adapted to support the changed DSML, they become less relevant for the analysis user because it successively supports fewer features.

How to improve evolvability and reusability is well-researched for software systems. The Gang of Four (E. Gamma, R. Helm, Ralph E. Johnson, and J. Vlissides) published design patterns (reoccurring structures) for object-oriented software, where they present reusable design patterns for object-oriented code [Gam+95] and Neill et al. [NLD11] present patterns that negatively affect the quality of a software system. Besides design patterns and anti-patterns, bad smells in object-oriented software can help software developers find occurrences in their source code that can lead to a deterioration of the internal quality of a software system.

Heinrich et al. [HSR19] investigated the evolvability and reusability of DSMLs. The outcome of their research is a reference architecture for DSMLs that, on the one hand, restricts the developer of DSMLs regarding their possible design decisions; on the other hand, does their reference architecture improve the evolvability and reusability, if the DSML follows the rules of their reference architecture. Strittmatter et al. [Str+16] derived bad smells for DSMLs from the aforementioned bad smells for object-oriented software.

However, to the best of our knowledge, no approach considers the co-dependency of DSMLs and their corresponding model-based analyses. Suppose software developers want to analyse new properties of a software system. In that case, they have to adapt features of the modelling language and the corresponding model-based analyses before they can analyse these new properties. Features in the context of the modelling language are, for example, quality properties such as performance or reliability. Features in the context of a model-based analysis are, for example, an analysis technique that analyses such a quality property. Such changes lead to complex and difficult-to-maintain model-based analyses. This increasing complexity reduces the understandability of the model-based analyses.

1.2 Problem Statements

In this section, we present the problems we identified that affect the evolvability, understandability, and reusability of model-based analyses. We identified three major problems:

Problem Statement 1

The evolvability, understandability, and reusability of historically grown model-based analyses suffer from increasing complexity.

The first problem we identified is the deterioration of the evolvability, understandability, and reusability of historically grown model-based analyses. Model-based analyses and their corresponding DSML change during their lifetime, for example, due to changing requirements or legal constraints. These changes lead to reduced software quality, negatively affecting the maintainability, especially the evolvability, understandability, and reusability of the model-based analyses. We consider difficult-to-maintain software to be a concern since it can lead to a range of issues. Increased development time and costs: When code is challenging to comprehend or modify, it takes developers longer to make changes, resulting in higher project costs. Reduced software reliability: Code that is difficult to maintain is more likely to contain bugs, which can lead to decreased software reliability. Limited scalability: If the code is difficult to comprehend and modify, scaling the software to meet the users' needs can be challenging. The difficulty for new developers: New developers may need help understanding the codebase, which limits their capacity to contribute. Overall, difficult-to-maintain software makes it more difficult to change and adapt, resulting in lower quality, higher costs, and a reduced ability to meet user needs. Heinrich et al. [HSR19] and Strittmatter [Str20] have shown that changes negatively affect the evolvability, understandability, and reusability of DSMLs. They provide a reference architecture for DSMLs that helps to improve these properties of the DSMLs. However, their approach did only focus on DSMLs, the effect on the software that utilises the DSMLs was ignored. Approaches like Functional Mock-up Interface (FMI) [Blo+12] or Distributed Interactive Simulation (DIS) [IEE95] focus on improving the reusability of analyses, but they only provide solutions for the analysis and not their corresponding DSML.

Problem Statement 2

The dependency of model-based analyses on their corresponding DSML results in a deterioration of their evolvability, understandability, and reusability.

The second problem we identified emerges because of the co-dependency of model-based analyses and their associated DSML. The model-based analyses and the DSML, rely on each other to function correctly. The following points are potential problems associated with co-dependency in model-based analyses and DSMLs. Lack of autonomy: Model-based analyses and DSMLs are co-dependent, and the model-based analysis cannot function independently. The lack of autonomy limits the reuse of the model-based analysis in a different context, making it difficult to replace or update the DSML without affecting the model-based analysis. Difficulty in identifying errors or bugs: Errors or bugs in model-based analyses and DSMLs can be hard to identify, especially when it needs to be well documented. Difficulty in resolving errors or bugs: Once identified, errors or bugs in model-based analyses and DSMLs can be challenging to resolve as a change might have a cascading effect, which results in more changes than initially anticipated. Increased complexity: Co-dependent model-based analyses and DSMLs can increase the overall complexity of the software ecosystem and make it more difficult to understand how they work together. Increased maintenance cost: Co-dependent model-based analyses and DSMLs systems require more maintenance and testing, which can increase the cost of development and operations. Co-dependency in model-based analyses and DSMLs can create a number of problems and make it difficult to evolve, understand and reuse the model-based analyses and DSMLs. It is essential to be aware of the co-dependency and to identify common patterns that indicate problems that can negatively affect the evolvability, understandability, or reusability of model-based analyses and DSMLs.

Problem Statement 3

Increasing complexity makes model-based analyses more challenging to understand and, as a result, to maintain, extend, or reuse.

The third problem we identified comes from the complexity of analysis components and the effort required to identify reusable, already existing analysis components. It can be challenging to find reusable analysis components for several reasons. Lack of standardisation: There are many different ways to write software, and components built for one analysis may need to be compatible with another. No standardisation can make it difficult to find components that can be easily integrated into a new analysis. Discoverability: With the vast amount of software available, finding the necessary component is challenging. There are many ways to discover reusable software components, such as searching online or browsing through open-source repositories, but finding the right one can still be time-consuming. Reusable software components often require ongoing maintenance to ensure they continue to work with the latest versions of other analyses and to fix any bugs. Reusable analysis components require proper documentation to use them effectively. With proper documentation, it can be easier to understand how to use an analysis component or configure it to work with the needs of a specific project.

1.3 Contributions

In this section, we present an overview of the three contributions of this thesis. This thesis aims to improve the evolvability, understandability, and reusability of model-based analyses. The three contributions of this thesis do support architects and developers in developing maintainable and reusable model-based analyses. The overall research goal of this thesis aligns with the presented problems in Section 1.2. Although there are approaches to improve the evolvability, understandability, and reusability of DSMLs and object-oriented software, respectively, we aim to improve these attributes by considering the co-dependency of DSMLs and model-based analyses. Thus, we formulate the following overall research goal for this thesis:

Overall Research Goal

We aim to improve the evolvability, understandability, and reusability of model-based analyses.

To reach our overall research goal, we provide three contributions. The first contribution extends the reference architecture for DSMLs by the domain of model-based analysis. Furthermore, our second contribution provides reoccurring patterns that negatively affect the evolvability, understandability, and reusability of model-based analyses. Our third and last contribution is the specification of analysis components to improve the reusability of model-based analyses.

Contribution 1

We propose a reference architecture for model-based analyses with accompanying processes to (i) modularise an existing model-based analysis, (ii) develop a model-based analysis from scratch, and (iii) extend an already existing model-based analysis.

To address Problem Statement 1, we take the reference architecture for domain-specific modelling languages [HSR19] as a basis and investigate the transferability of the structure of the reference architecture to model-based analyses. We introduce a five-layer architecture that uses four layers of the reference architecture for domain-specific modelling languages (basic features, domain-specific features, quality features and analysis configuration) and extends them with an experiment layer. The layered reference architecture maps the dependencies of the analysis features and components by assigning them to specific layers. We developed three processes for applying the reference architecture: (i) refactoring an existing model-based analysis, (ii) designing a new model-based analysis from scratch, and (iii) extending an existing model-based analysis. We refactored four representative model-based analyses and used them as case studies. After the refactoring, we compared the modular model-based analyses with the original monolithic model-based analyses regarding metrics complexity, coupling, and cohesion.

Contribution 2

We provide a set of bad smells for model-based analyses that emerge because of the co-dependency of model-based analyses and their corresponding DSML. We also provide identification and refactoring strategies to identify and fix bad smells.

To address Problem Statement 2, we have identified recurring structures that lead to problems in evolutionary capability, comprehensibility and reusability; in the literature, these recurring structures are also called bad smells. In particular, we have investigated the co-dependency of DSMLs and model-based analyses that lead to these recurring problems.

So far, bad smells for DSMLs and source code have been considered separately, although they are co-dependent. Model-based analyses are based on the DSML they analyse, they require the DSML as input for the analysis, and a change in the DSML leads to a change in the corresponding model-based analyses. We examined established model-based analyses and identified and specified thirteen bad smells. In addition to specifying the bad smells, we provide a process for automatically identifying these bad smells and strategies for refactoring them so that developers can avoid or fix them. To evaluate this contribution, we searched established model-based analyses for our bad smells, so we could show that the bad smells we specified occur in real-world systems. We also investigated how fixing the bad smells affects the evolvability, understandability and reusability of the model-based analyses we studied. After refactoring, we compared the modular model-based analyses with the original, unmodified model-based analyses concerning the metrics complexity, coupling and cohesion.

Contribution 3

We provide a DSML for the specification of analysis components that allow the analysis developer to find analysis components that are similar regarding their structure and behaviour.

To address Problem Statement 3, we developed a modelling language for specifying the structure and behaviour of simulation features. Simulations are analyses to investigate a system when experimenting with the real-world system is too time-consuming, costly, dangerous or simply impossible because the system does not exist (yet). Developers can use the model to compare simulation features and thus identify identical features. Finding similar simulation features allows developers to reuse existing features and reduce the effort required to develop new features. To evaluate the approach, we specified features of two open-source simulations and compared them. By modelling these existing simulations, we investigated the applicability of the approach. In addition to modelling the structure and behaviour of simulation features, we also evaluated the accuracy of identifying similar simulation features. We used our approach to compare the specified features with the specified features of the case studies to determine the precision and recall of the approach.

1.4 Thesis Outline

This chapter presented the problems and challenges that arise through the co-dependency of DSMLs and model-based analyses. Based on the presented problems, we defined our overall research goal. Furthermore, this chapter presents an overview of our contributions. The remainder of this thesis is organised as follows:

Chapter 2: In the second chapter, we present the terms and definitions that are used throughout this thesis. Terms and definitions that are dedicated to only one chapter are placed in the respective chapter. We also present the foundation for our contributions which mainly consists of the reference architecture for DSMLs by Heinrich et al. [HSR19] and the validity types by Runeson et al. [Run+12].

Chapter 3: In the third chapter, we present our reference architecture for model-based analyses. Besides our modularisation concepts for model-based analysis we also present the processes for the decomposition and composition of model-based analyses. Furthermore, we present refactoring operations to adapt an already existing model-based analyses and a concrete instantiation of our reference architecture in the context of quality analyses.

Chapter 4: In the fourth chapter, we present the bad smells in model-based analyses that arise through the co-dependency of DSMLs and their corresponding model-based analyses. We categorise the bad smells we derived from bad smells in object orientation and bad smells in DSMLs. Furthermore, we present strategies to identify our bad smells in model-based analyses.

Chapter 5: In the fifth chapter, we present our approach to specify and reuse model-based analyses components. First, we present our approach to specifying analysis components regarding their structure and behaviour. Then, we present our approach to compare and identify analysis components.

Chapter 6: In the sixth chapter, we present the four case studies we use throughout this thesis. We discuss our selection criteria for the case studies and then briefly overview each case study. Not only do we present the model-based analyses, but we also present the four DSMLs that the model-based analyses use.

Chapter 7: In the seventh chapter, we present the evaluation of our reference architecture for model-based analyses. First, we present our research goals and metrics for this contribution. Then, we present our evaluation design by presenting the evolution scenarios and the details of the refactorings of the four case studies. After the refactorings, we present the results of the evaluation. We close this chapter by discussing the threats to validity and the conclusion.

Chapter 8: In the eighth chapter, we present the evaluation of our bad smells for model-based analyses. First, we present our research goals and metrics for this contribution. Then, we present our evaluation design by presenting the evolution scenarios, the analysis details, and the refactorings of the four case studies. After the refactorings, we present the results of the evaluation. We close this chapter by discussing the threats to validity and the conclusion.

Chapter 9: In the ninth chapter, we present the evaluation of our specification and reuse approach for model-based analysis components. First, we present our research goals and metrics for this contribution. Then, we present the results of the evaluation. We close this chapter by discussing the threats to validity and the conclusion.

Chapter 10: In the tenth chapter, we present the related work to differentiate our work from the state-of-the-art. For our first contribution, the reference architecture for model-based analyses, we present related work concerning the decomposition and composition of analyses. Then, we present related work that integrates DSMLs and model-based analyses. For our second contribution, the bad smells in model-based analyses, we present related work concerning detecting and refactoring bad smells. For our third contribution, the specification and reuse of model-based analysis components, we present related work comparing source code and the specification and reuse of analysis components.

Chapter 11: In the eleventh chapter, we summarise our contributions and evaluation results. We discuss our results in the context of our research goal and conclude this thesis. Additionally, we discuss possible future work.

2. Foundation

In this chapter, we present the foundations of this thesis. First, in Section 2.1, we introduce terms and definitions that are used throughout the whole thesis. We introduce DSMLs in Section 2.1.1 and model-based analyses in Section 2.1.2. The roles we use to describe the target audience of our approaches are presented in Section 2.1.3; we distinguish the role of the developer (cf. Section 2.1.3.1) and the role of the user (cf. Section 2.1.3.2). Besides the terms and definitions, we also present the foundational concepts in Section 2.2 on which this thesis is built on. We then provide dedicated foundation sections for our contributions in Section 2.3, Section 2.4, and Section 2.5. In Section 2.6, we present the foundation for our evaluation. The validity types by Runeson et al. [Run+12] that we used for every evaluation in this thesis are presented in Section 2.6.1 and the principles of the Goal Question Metric Approach are presented in Section 2.6.2. Finally, we present the technical foundation in Section 2.7.

2.1 Terms and Definitions

In this section, we present the terms and definitions that we use throughout all contributions. First, we introduce the term *domain-specific modelling language* in Section 2.1.1. In Section 2.1.2, we introduce the term *model-based analysis* and what an *analysis* and an *analysis model* is. The roles we use throughout this thesis are introduced in Section 2.1.3.

2.1.1 Domain-specific Modelling Language

Compared to general-purpose modelling languages like Unified Modelling Language (UML), DSMLs is explicitly tailored to the needs of particular application domains. They are typically less expressive, concentrating instead on the essential ideas associated with the pertinent domain. When compared to general-purpose modelling languages, these specialised languages make it possible to express domain models in a manner that is both more succinct and precise. Professionals specialising in a particular field or domain can use established and widely recognised modelling languages such as UML [Rum17] and Systems Modelling Language (SysML) [FMS14] to represent and design complex systems. Alternatively, they may develop their DSML to suit their needs and requirements. UML is a general-purpose modelling language that has become the industry standard for modelling software-intensive systems. SysML is an extension of UML specifically designed to support the modelling and analysis of complex systems, including hardware and software. On the other hand, DSMLs are modelling languages custom-built for a particular domain or problem space, such as medical devices or financial systems. DSMLs enable domain experts to model and design systems using concepts and abstractions specific to their domain, leading to more efficient and effective design and analysis. The decision to use an established modelling language or develop a DSML depends on the complexity and specificity of the domain in question and the expertise and resources available to the domain experts. Because it is simpler for domain experts to learn from and comprehend a DSML than a general-purpose language, it is possible that using a DSML may improve communication with domain experts, ultimately leading to higher productivity [SVC06]. A modelling language is constructed from building blocks, including explicit syntax and corresponding semantics. The words and the structure of the language, often known as its "grammar"are referred to by the term *syntax*. For example, denotational semantics can be realised through the mathematically sound definition of a semantic mapping from well-formed models to an appropriate and well-understood semantic domain [HR04]. Each modelling language has its semantic domain within which it operates. As an illustration, statecharts make use of I / O-relations. The syntax of DSML can be textual or graphical (including diagrams, for example). Even though diagrams can help gain a general understanding of a concept, this understanding can quickly get clouded by confusion, making it difficult to traverse. Textual languages make acquiring an overall picture of the model more challenging. However, they have the advantage of being compatible with well-known development tools like copy and paste, syntax highlighting, and auto-completion. There are two possible approaches to defining a textual domain-specific language, and they are as follows: Grammar-based: Define a grammar to specify the language and a metamodel will be generated based on the grammar definition provided. Mapping-based: After independently constructing the metamodel and the concrete syntax, the next step is to define a mapping between the two.

2.1.2 Model-based Analyses

A *model-based analysis* is a type of analysis that uses models for reasoning about a system and for communicating the results [ZMK18]; it is a tool for the engineer to understand a problem better [Fow96]. A model-based analysis provides a detailed examination of a model of a system under study. According to Talcott et al. [Tal+21a], the purpose of model-based analyses can be: gaining structural, behavioural, or quality information about a system. In the context of this thesis, these models are developed according to a DSML; thus, the model-based analysis can analyse different instances of a DSML. A model-based analysis uses models for examining the structure, behaviour and quality of a process or system and models are used for communicating the results. The system is modelled with a DSML containing the information about the system required for the analysis. For example, if the DSML specifies the architecture of software systems, the engineer can model a software system with different architectures to find the software system with the best performance. A benefit of model-based analyses is that if the system is too complex, too expensive or does not exist yet, an analysis can provide insights before the system is implemented [Law15]. Instances of the DSML serve as input for the analysis. We

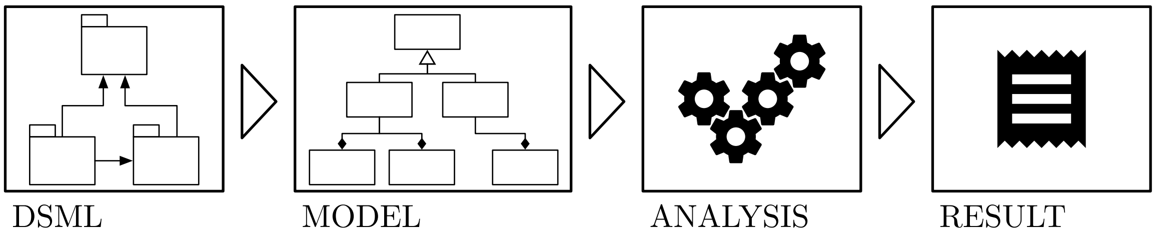


Figure 2.1.: The Context of DSMLs, Models, Analyses, and Results

distinguish between analysis models that represent the “mental model” that describes how the analysis solves the problem [Fow96] and analysis models that represent the system the analysis reasons about [HSR19].

In Figure 2.1, we present the context of DSMLs, models, analyses, and the analysis results. The DSML allows the engineer to model the system to be analysed. However, the engineer only models the system with some possible details. For example, if the performance of software architecture is analysed, the model does not contain details of the algorithm [Reu+16]. A model is always a reduction of the thing it represents [Sta73]. The DSML prevents the engineer from modelling every detail of the system; it allows only to model of the elements required by the analysis. The analysis takes an instance of the DSML, a model, and produces the analysis results. An analysis can analyse different properties of the model; depending on the properties, the analysis results are different. For example, in addition to the aforementioned performance analysis, the same model can be analysed regarding other properties, like reliability, performability, or security. The performance analysis provides data containing the system’s throughput, whereas the security analysis can provide a list of security breaches. We assume that the results of the analysis also follow a DSML; this DSML can be part of the DSML that specifies the input model; however, depending on the analysis and the type of results it creates, the result DSML can be independent of the input DSML. For example, if the results are required for another model-based analysis that requires a different DSML instance as input [KR19], then the results can be transformed according to the desired DSML. The analysis contains algorithms that interpret the input model, and analysis routines are called depending on the model type. These routines contain the analysis algorithms that investigate properties, like performance or reliability, of the system under study.

2.1.3 Roles

In this thesis, we distinguish two main roles involved in the development process of model-based analyses. In Section 2.1.3.1, we present the *developer* role and all sub-roles associated with it. We present the *user* role and all sub-roles associated with it in Section 2.1.3.1.

2.1.3.1 Developer Role

Heinrich et al. [HSR19] defined the roles that specify and evolve a DSML. In this thesis, these role specifications are extended by adding roles for specifying and developing a model-based analysis.

Based on their interaction with analyses, we mainly distinguish between key developer roles: *analysis developer* and *tool developer*. The analysis developer is accountable for defining, developing, implementing, and maintaining the various components of an analysis, such as modelling analysis requirements, debugging issues, implementing analysis components, and adding new analysis features to the analysis specification. On the other hand, the tool developer is responsible for creating and maintaining tools that utilise the analysis. This includes writing and modifying code to initiate the execution of the analysis and utilise its results. In designing tools for orchestrating analyses, the tool developer must possess knowledge of the various orchestration strategies. A choice of six orchestration procedures is available to the tool developer [Hei+21a]. For future references, the term *developer* will collectively refer to both roles.

Within the context of our approach, the role of the analysis developer is subdivided even further into those of the *analysis architect* and the *analysis component developer*. Analysis component dependencies, the feature model, and the feature definition are all the responsibility of the analysis architect. For instance, they are responsible for the creation of the analysis specification, the validation of the analysis, the specification of new features, and the modification of feature specifications in accordance with the requirements currently in place or that may change in the future. It is the responsibility of the analysis component developer to implement analysis components. In doing so, they are responsible for implementing the features specified by the analysis architect as distinct components. Both roles work together while creating or changing analysis components.

2.1.3.2 User Role

In the thesis, we also discuss the role of the *user*, which refers to the person conducting the model-based analysis. An analysis is carried out by the user with the use of tools that function on DSML instances to detail the input and output of the model-based analysis and start the process of carrying out the analysis. Because of this, we will refer to people in this position as *tool users*. In the context of "DSMLs generate and alter models using editors", the term "tool users"refers to those who use these editors. Within the scope of model-based analysis, they conduct analyses utilising these models. When we speak of abstractions and properties, we refer to specific groups of abstractions and properties that are frequently employed together and have a central concept, also known as a concern. Analysis of software structure and behaviour and studies into software performance are just a few examples of activities that fall under this category.

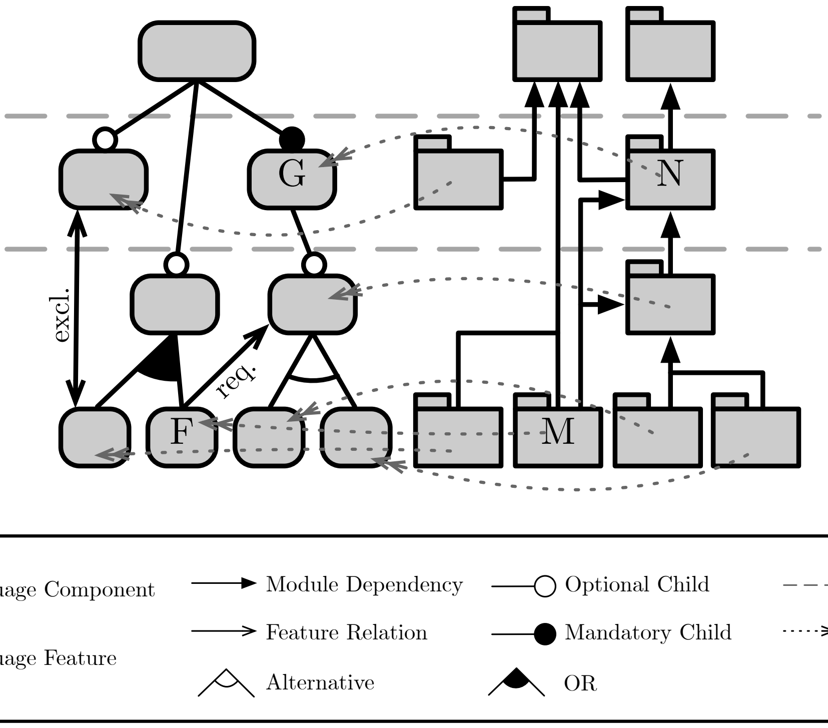


Figure 2.2.: Reference Architecture for DSMLs– Concept [HSR19]

2.2 Foundational Concepts

In this section, we present the concepts this thesis is based on. In Section 2.2.1, we introduce the reference architecture for metamodels by Heinrich et al. [HSR19] that inspired our reference architecture for model-based analyses and thus, also influences the bad smells for model-based analyses. In Section 2.2.2, we present the metrics we used to evaluate our contributions. Foundational concepts we used for only a single contribution will be placed in dedicated sections.

2.2.1 A Reference Architecture for Metamodels

To improve the evolvability and reusability of metamodels, Heinrich et al. developed a reference architecture that provides a layered structure and a set of dependency rules [HSR19]. Figure 2.2 depicts the concepts of the reference architecture for DSMLs. In this section, we present these concepts. They also provide a modularisation concept for metamodels that allows the developer of a metamodel to modularise an existing, monolithic metamodel. Besides the modularisation concepts, Heinrich et al. also provide guidelines on applying their reference architecture for metamodels to existing metamodels and developing one that complies with their reference architecture. They provide a systematic way of creating, extending and reusing metamodels or parts of these metamodels. In their work, they transfer modularisation concepts from object-oriented design and the idea of a reference architecture to metamodels for quality modelling. They gathered the requirements for the reference architecture from a historically grown metamodel. Their reference architecture supports instance compatibility and non-intrusive, independent extension of metamodels.

2.2.1.1 Language Features

A metamodel is equivalent to a DSML, composed of language features. Language features allow the specification of a language on a conceptual level. They differentiate between atomic and composed language features, where atomic language features are an abstraction of the subject that is modelled. A composed language feature consists of atomic language features and other composed language features. The dependencies of the language features are derived from the subject it represents.

2.2.1.2 Feature Models in the Context of DSMLs

To express the structure of a DSML, Heinrich et al. [HSR19] employ feature models commonly used in the product line community. The feature models are used to restrict the dependencies of the DSML. The dependencies of the feature model must be derived from the dependencies of the language features. Restrictions like the prohibition of cycles and strict parent feature relations give a framework for the DSML developer. Instead of merely a graph of language features and the dependencies between them, a feature model organises the language's characteristics into a hierarchical structure. The developer that works with the DSML, such as analysis developers, have an easier time selecting the features they want to employ since they may begin their search at the top level of the feature hierarchy and proceed only to the branches that are pertinent to their needs.

2.2.1.3 Modules and Dependencies

According to Heinrich et al. [HSR19], language features must be implemented. The implementation happens in the metamodel modules; these modules are containers for packages and classes with a dependency structure that follows the dependency structure of the language features. However, the dependencies of the metamodel modules are more concrete, where a language feature has two kinds of dependencies, optional and mandatory; the dependencies of metamodel modules follow the dependencies that exist in metamodel modelling. A metamodel divided into metamodel modules still counts as a metamodel.

2.2.1.4 Extends Relation

Another contribution of Heinrich et al. [HSR19] is the definition of an *extends* relation that defines the dependencies between metamodel modules. They extended the Essential Meta-Object Facility (EMOF) standard, as it cannot restructure metamodel module dependencies. Without these *extends* relations, the EMOF extension would have violated the reference architecture for DSMLs. EMOF cannot add new class properties without hampering the reusability of the DSML.

2.2.1.5 Layers

The reference architecture for DSMLs also uses layers to group language features and metamodel modules. Language features and their corresponding metamodel modules must be located on the same layer and only be placed on one layer. Although Heinrich et al. [HSR19] do not set the number of layers for DSMLs, they propose a four-layered reference architecture tailored to DSMLs for quality modelling and analysis.

2.2.1.6 Layers in Metamodels for Quality Modelling and Analysis

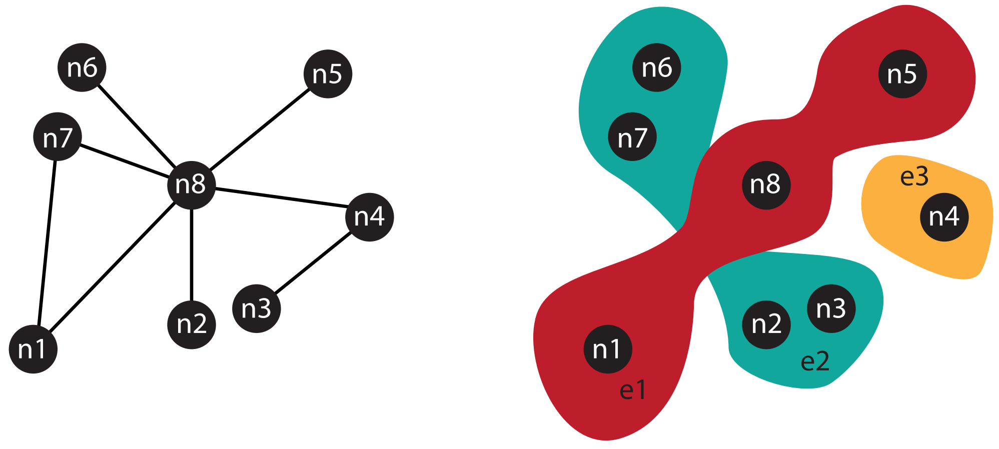
For the layering of a DSML that allows modelling and analysis of quality attributes, Heinrich et al. [HSR19] propose a layered architecture with four layers.

Paradigm Layer: On the paradigm (π) layer is the fundamentals of the DSML located. It contains structural and behavioural patterns that occur throughout the DSML. Especially foundations independent of the domain are located on the π layer. The idea is that the π layer contains only abstract concepts and, thus, can only be used with another layer.

Domain Layer: The domain (Δ) layer follows the π layer; features on the Δ layer assign domain-specific semantics to the features on the π layer. For example, on the π layer, concepts like classes and relations are defined. On the Δ layer, classes of software systems extend the notion of classes, and relations become more specialised by introducing inheritance relations. On the Δ layer, the developer can specify multiple domains. If a developer is solely interested in software, then the metamodel module for software components is all that is included in the Δ layer. A DSML may consist only of the π and the Δ layer for designing and documenting a system not concerned with quality. However, language features that can be used for modelling or analysing quality properties are not placed on the Δ layer; instead, they are a component of the following layers.

Quality Layer: The quality (Ω) layer follows the Δ layer; on it can, the developer defines the quality properties that can be modelled with the DSML. These quality properties are built on domain-specific language features. For example, the developer can add quality properties for each domain feature that specify a language feature's performance or reliability. The Ω layer is specific for DSMLs that model the quality of a system.

Analysis Layer: The analysis (Σ) layer follows the Ω layer; features on the Σ layer are required by model-based analyses that use the DSML for analysis. If, for example, an analysis needs attributes that are referenced on other layers, this information is located on the Σ layer. The value of the attribute is altered over different analysis executions. The attribute is specified in a module found in one of the more generic layers. Model-based analyses that use the DSML can share the features on the Σ layer.



(a) Regular Graph **(b)** Hypergraph

Figure 2.3.: Two Graphs with the Same Nodes (n1..n8) but the Regular graph has Edges (black lines) and the hypergraph has Hyperedges (e1..e3)

2.2.2 Hypergraph Metrics

In this section, we present the hypergraph metrics we use to evaluate our case studies. The hypergraph metrics are based on the work of Allen et al. [All02; AGG07]. These metrics use graph and hypergraph abstractions of software systems to determine the information entropy of a software system. The hypergraph metrics calculate a software system's complexity, coupling, and cohesion to determine the information entropy of a system. In contrast to metrics that, for example, count the number of incoming and outgoing calls to determine cohesion and coupling, the hypergraph metrics also consider the interconnection of the software system. The higher a software system is interconnected, the more complex the system is. Thus, the hypergraph metrics allow us also to consider the interconnection to determine the complexity and, thus, the evolvability and understandability of a software system.

The difference between a graph and a hypergraph is that graphs consist of nodes and edges, and hypergraphs consist of nodes and hyperedges. An edge connects two graphs. In contrast to an edge, a hyperedge can connect more than two edges. Another benefit of hyperedges, in contrast to regular graphs, is that they can model the set-use relationships of public attributes [HSR19]. Figure 2.3 depicts the difference between graphs and hypergraphs. Figure 2.3a shows a regular graph with eight nodes depicted as black circles. The edges, depicted as black lines, create pairs of nodes (e. g., n1 and n7 or n1 and n8). Figure 2.3b shows a hypergraph with eight nodes depicted as black circles. The hypergraph contains three hyperedges. The first hyperedge *e1* connects the nodes *n1, n8,* and *n5*. The second hyperedge *e2* connects the nodes *n2, n3, n6,* and *n7*. The third hyperedge contains a single node, *n4*.

In our evaluation, we use the approach by Jung [Jun16] to extract hyperedges and hyperedge modules. Jung's approach is based on the approaches by Schütt [Sch77] and Allen et al. [AGG07], which describe how to extract hypergraphs from software systems. Jung's approach separates a hypergraph into modules. We denote this modular hypergraph *H*. A hypergraph module represents a set of nodes; each node can only be contained in one module. According to Strittmatter [Str20], we distinguish between hyperedges that do or do not cross module boundaries. A hyperedge that does not cross module boundaries is called intra-module hyperedge. A hyperedge that crosses module boundaries is called inter-module hyperedge.

$$Size\ (H) = \sum_{i=n}^{n}(-log_2 p_L(i)) \tag{I}$$

$$Complexity\ (G) = \left(\sum_{i=j}^{n} Size(G_j)\right) - Size(G) \tag{II}$$

We use the size metric of Allen et al. [AGG07] to determine the complexity of a software system. Equation (I) shows how to calculate the *Size* of a modular hypergraph *H*. The sum is calculated over the probability pattern $(p_L(i))$ for all nodes i in the hypergraph *H*. We must calculate the size of the hypergraph, and therefore, we must establish a pattern for each hypergraph. We fill a vector with ones and zeros to represent the pattern. Each entry represents the relation of hyperedges and its nodes. A one means the hyperedge is connected, and a zero means the hyperedge is not connected to the node. Identical patterns are grouped, and the number of occurrences is saved. The probability of each pattern p is calculated by calculating the ratio of the number of occurrences and the number of nodes in H [AGG07].

Equation (II) shows how to calculate the complexity of a system. The *complexity* function takes a modular hypergraph *G* as input. The size is calculated for each modular hypergraph G_j in *G*. The hypergraph G_j is a hypergraph that contains a node j and all nodes connected to j by hyperedges. After the size for each modular hypergraph is calculated, the size metric finally gets applied to the whole modular hypergraph *G*. According to Allen et al. [AGG07] is the coupling of a modular hypergraph defined as the complexity of *G*, whereas *G* is reduced by the intra-module hyperedges. We used Jung's [Jun16] approach to determine the modular hypergraph H^* that contains only inter-module hyperedges. Based on the hypergraph H^* we calculated the complexity of the system.

$$Cohesion\ (MG) = \frac{Complexity\ (MG^o)}{Complexity\ (MG^{(n)})} \tag{III}$$

Equation (III) shows how to calculate the cohesion of a system. According to Allen [All02], cohesion, in terms of a hypergraph, is the ratio of the intra-module graph MG^o and the complexity of the whole graph $MG^{(n)}$. We applied the cohesion metric to a regular graph. We follow Jung's method [Jun16] to calculate cohesion as with the previous metrics. The modular hypergraph H must be mapped to a regular graph MG, which replaces each hyperedge with a set of edges connecting all nodes previously connected by the hyperedge. The result MG gets stripped of all inter-module edges; the result is a graph MG^o with only intra-module edges. MG also is used to create the complete graph $MG^{(n)}$.

Sub-graph Extraction: Evolvability cannot be seen as the absolute property of a whole software system; therefore, it should always be considered in a concrete evolution scenario [Ros+15]. We implemented a scenario-based evaluation to create such evolution scenarios by calculating the metrics for parts of the software system relevant to the evolution scenario. We could avoid applying the metrics to the case studies as a whole. Each evolution scenario of a case study represents a sub-graph. For each case study, we extract the relevant sub-graph per evolution scenario. When implementing the change of an evolution scenario, the sub-graph represents the part of the software system that the developer must inspect. Classes affected by the evolution scenario are *affected classes*. We construct a sub-graph consisting of dependent classes by basing its composition on the impacted classes and the nature of the change. For instance, if a method signature were to change, the classes associated with that method would be added to the sub-graph.

2.3 Foundational Concepts for the Decomposition and Composition of Model-based Analyses

In this section, we present supplemental concepts required to follow the contents of our first contribution, the decomposition and composition of model-based analyses.

2.3.1 Quality Property

Quality property is a term defined in International Organization for Standardization (ISO) / International Electrotechnical Commission (IEC) 25010 quality models [ISO10]. Examples of quality properties include performance, reliability, and maintainability. Part of the evolvability of a software system is its maintainability and extensibility. According to the software evolvability model by Breivold et al. [BCE08], the model consists of sub-characteristics analysability, integrity, changeability, extensibility, portability, and testability. Breivold et al. [BCE08] presented the software evolvability model. This model comprises the sub-characteristics of analysability, integrity, changeability, extensibility, portability, and testability. According to the ISO / IEC 25010 software quality model [ISO10], the characteristic of maintainability and portability map to the sub-characteristics of the software evolvability model by Breivold et al. [BCE08]. The sub-characteristics analysability, changeability, stability, and testability are part of the maintainability characteristic of ISO / IEC 25010 and the sub-characteristics of adaptability, installability, co-existence, and replaceability are part of the portability characteristic of ISO / IEC 25010.

2.3.2 Modelling Language

One purpose of modelling languages is to design processes and systems. According to Holldobler [HRW18], a modelling language is developed and used to define models to design and analyse systems effectively. They can be used to reason about the processes and systems they represent. As a result, recurrent domain knowledge is captured in language features and patterns, which are then used to construct instances of the modelling language (i. e., models). Thus, they are called Domain-specific Modelling Languages (DSMLs). DSMLs can be subdivided into grammar-based languages and metamodel-based languages [HSR19]. Modelling languages are developed and used to design and reason over processes and systems efficiently and effectively. Recurrent domain knowledge is captured in the form of language features and patterns, which are then used to construct models. As a result, they are referred to as DSMLs. Grammar-based languages and metamodel-based languages are the two types of DSMLs. Recurrent domain knowledge is captured in the form of language features and patterns, which are then used to construct models. In our research, we are interested in metamodel-based DSMLs as our previous reference architecture is tailored to DSMLs. By transferring our knowledge to model-based analyses, we expect to improve their evolvability and reusability. Modellers can use standardised languages, such as UML [Rum17] or SysML [FMS14] or they can design their own DSMLs [Com+18]. Language workbenches [SVC06; HKR21] enable describing extensible languages to capture reoccurring domain knowledge. A *modelling language feature* is an abstraction of a thing to be modelled [HSR19]. It is necessary to have a clear syntax and a corresponding set of meanings, which together form the language's explicit syntax and associated semantics, to create a modelling language. The *syntax* defines the language's words and structure, while the semantics explains what the model means. For instance, denotational semantics is usually achieved through a mathematically valid definition of semantic mapping from well-formed models to a suitable and well-understood semantic domain [HR04]. State charts use I / O relations to define their semantic domain, while other languages may use other approaches. The FOCUS approach [BS01; RR11] is an example of a mathematical system model used for integrated semantics, which enables embedding other semantic domains, such as SysML semantics.

2.3.3 Feature Model

A *feature model* [CE00] is a formal representation of the variability and interdependencies among the features of a subject. The feature model determines a subset of the relevant features to a particular scenario. This subset is determined by constructing a feature graph, which is represented as a tree structure. Figure 2.4 shows tree structure and the relations of features in a feature graph. The tree structure is formed through parent-child relationships between features, where each feature, excluding the root node, has one parent, and the root node has only child relationships. The relationships between parent and child features can be mandatory, optional or part of an alternative set or an OR set [CE00]. Any mandatory child features must also be selected if a parent feature is selected. Optional child features can be chosen but are not required. In an alternative set, only one feature can be selected, and at

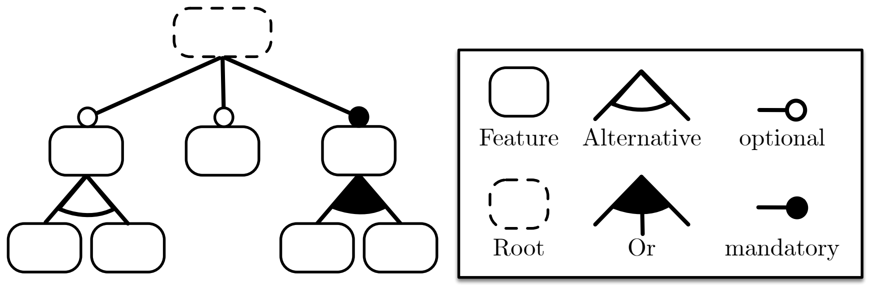


Figure 2.4.: Concept of Feature Models

least one must be selected from an OR set of features. Language features are implemented by modelling language components [HSR19]. A *modelling language component* describes language constituents, e. g., through metamodels or grammars, has explicit interfaces and composition operators [But+19; HSR19] for other modelling language components, and has an individual, composable semantics. Analysis features are implemented by *analysis component*s containing the analysis algorithms realised in source code. These analysis components are executable on the required language features, have explicit interfaces, and can be combined with other analysis components using composition operators.

We utilise feature models to express the features of DSMLs and model-based analyses. Based on a feature model, subsets of the given features are selected to specify which modelling language or analysis features are of current interest in analysis composition and tool development [Str20]. In the context of our research, we allow the specification of a set of features that contains only one feature. That is uncommon for most feature models, but the benefit is that the feature set can be augmented with other features without modifying the type of child relation. A feature selection is a subset of the features in the feature model that adheres to the feature relations' requirements [HSR19]. *Requires* and *excludes* relationships between features are also possible. Relationships must be directional, and mutual relationships are excluded.

2.3.4 Analysis Composition

Analysis composition combines sub-analyses into one complete analysis, where the individual sub-analyses adhere to their sub-language. The individual sub-results adhere to their sub-analyses, and an appropriate *orchestration* of sub-analyses defines the order and use of sub-results into a complete result. Analysis helps answer a question about a system that might be too complex or not accessible; thus, analyses work with models (i. e., based on a DSML) of the system under study. The models represent relevant aspects of the system under study that help to answer the questions the analysis is determined to answer. Heinrich et al. [Hei+21b] define an analysis as follows:

$$M, C \vdash_T Q \rightsquigarrow A$$

M, *C*, *Q*, and *A* are models of the system, the context, the question, and the answer domains, respectively; *T* is an analysis technique used in the analysis. The definition of analyses can also be mapped on sub-analyses. A sub-analysis answers a sub-question *Q* using an analysis technique *T*; to answer the sub-question, it analyses a sub-language *M* under a context *C*. Sub-analyses must be combined to create an analysis that can reason about multiple facets of a system (i. e., sub-language). This composition of sub-analyses has many facets. Talcott et al. [Tal+21a] distinguish three general forms of composition that we will apply: Model composition (white-box composition) is the analysis input model, realised by language integration. The composition of the analysis results by orchestrating encapsulated analyses is called result composition (black-box composition). The composition of the analysis techniques by orchestrating the steps of two or more analysis algorithms is called analysis composition (grey-box composition). In [Tal+21a], they mathematically define the concept of analysis composition based on these three types of composition. Sub-analyses can be selected from the feature model visualised in Figure 3.2 to configure and extend a model-based analysis. Their associated analysis techniques are composed by selecting sub-analysis from the feature model.

2.3.5 Feature Composition

Apel et al. [Ape+08] introduce the concept of a Feature Structure Tree (FST) for the feature composition. An FST models a feature's structural elements (source code artefacts), e. g., classes, fields, or methods hierarchically. The concept of FST corresponds to our feature and component notation; however, as depicted in Figure 3.2, two separate graphs represent our notation of features and components. Nevertheless, our analysis feature and analysis component notation can be transformed into the FST notation, see Figure 3.18.

For the transformation process, it is imperative to have both the feature graph and the component graph. The existing structure of the feature graph can be utilised without modifications. The analysis components are represented as double-lined rectangles with rounded corners, which are then added to the feature graph as leaf nodes.

The implements dependency between an analysis component and an analysis feature is transformed into a double-lined dependency, while the dotted implements arrow between components becomes a dashed arrow. Each analysis component is added as a terminal node to the feature graph, with a terminal node being defined as the last node in the graph with no child nodes.

Dependencies on terminal nodes are represented as implements dependencies, and dotted arrows between terminal nodes represent analysis component dependencies. As a result, all the operations outlined in [Ape+08] can be applied to this model.

The following paragraphs provide an overview of the operators of Apel et al. [Ape+08] and our extension for features and feature composition. A feature composition is defined as:

$$f \bullet g$$

This rather abstract operator is divided into two sets, the introduction set I and the modifications set M, as well as three operations:

$$\oplus : I \times I \rightarrow I, \odot : M \times M \rightarrow M$$

and

$$\odot : M \times I \rightarrow I$$

An *atomic introduction* expresses basic features and their composition. An atomic introduction is a part of implementing a basic feature, such as a method, field, class, or package. Introductions are the basic units of difference between two basic features. The superimposition of all paths/atomic introductions in its FST is a basic feature. Apel et al. [Ape+08] model FST superimposition using the introduction sum. The introduction sum operator $\oplus$ is an operation over the set I, adding two atomic introductions; the result is a non-atomic introduction. Introduction sum $\oplus$ over the set of I of introductions forms a non-commutative idempotent monoid $(I, \oplus, \xi)$. The introduction sum is associative and non-commutative. The identity is defined as follows:

$$\xi \oplus i \ = \ i \oplus \xi \ = \ i - \xi \text{ is an empty FST}$$

The idempotence says that only the rightmost occurrence is effective in a sum; duplicates have no effect:

$$i \ \oplus \ j \ \oplus \ i \ = \ j \ \oplus \ i$$

For $j = \xi$ follows $i \ \oplus \ j \ = \ i$.

2.3.6 Analysis Decomposition

We define *analysis decomposition* as separating one analysis into individual sub-analyses, where a model that adheres to the language of the complete analysis can be projected into individual sub-analyses. Decomposing an analysis result in: $M, C \vdash_T Q \rightsquigarrow A$, per extracted sub-analysis. In software engineering exist, different approaches to determining modules; in the context of analyses, we call them sub-analysis. The purpose of decomposing software is to make each module more manageable. One core principle of programming software is “Divide and Conquer”, separating software into functions [CKM22], classes [NKB00], or frameworks [CLZ04] to separate concerns [CG20], or even hardware/software interfaces [DMV20]. For example, in a modularised software system, developers do not have to understand each line of source code when they extend or maintain software. Instead, they must know the module where the change must occur. In Object-Oriented Programming (OOP), for example, classes are used to represent concepts of the software and to make the software more manageable. In aspect-oriented programming, the software is decomposed according to the aspects the software covers. For the development of DSMLs, Heinrich et al. [HSR19] provide a decomposition concept for metamodels. However, to the best of our knowledge, there is no approach that considers the modular structure of a DSML and derives a decomposition concept for its corresponding model-based analyses. To fill that gap, in the following section, we will present our decomposition concept for model-based analyses while considering the modular structure of their corresponding DSML.

2.4 Bad Smells in Different Domains

In this section, we present supplemental concepts required to follow the contents of our second contribution, bad smells in model-based analyses. Therefore, we explain the bad smells of different domains. We use these types of bad smells to derive bad smells for the domain of model-based analyses. As bad smells reduce the evolvability and reusability of software, our first goal is it, to identify and refactor bad smells in the domain of model-based analyses.

According to Martin Fowler and Kent Beck are bad smells structures in the code that suggest the possibility of refactoring [Fow99]. Bad smells are created from the experience of developers that have looked at lots of code.

First, we explain bad smells on the code level, i. e. in object-oriented software. Second, we explain bad smells in metamodels and DSMLs. Moreover, finally, we discuss why model-based analyses still have bad smells that are not covered by either object-oriented bad smells nor DSML bad smells.

2.4.1 Bad Smells in Object-oriented Software

In this section, we focus on bad smells on the code level, code written in an object-oriented programming language in particular. The term *bad smell* was shaped by Kent Back in the late 20th century. Kent Beck and Martin Fowler initially identified 21 bad smells in 1999 [Fow99]. Since then, in 2018, Martin Fowler published a new revision of its book "Refactoring", and in it, they changed the names of some bad smells and added one additional bad smell [Fow18]. We will refer to the new version of the bad smells. Although not all bad smells are required to understand the bad smells of model-based analysis, we want to give a brief overview of these 22 bad smells.

Mysterious Name: According to Fowler, it is fun to puzzle over words in detective fiction rather than in code. The code may seem boring and straightforward, with no suspense and immersion. Developers should put much care into naming functions, modules, variables, and classes to express what they do and how to utilise them. Naming is one of programming's two hardest things [Fow18]. It is worth renaming things, even if it has the slightest chance of improving the readability of the code. With modern Integrated Development Environments (IDEs), renaming classes, methods, variables, or modules requires almost no effort besides finding a suitable name.

Duplicated Code: The same code structure in multiple places makes software difficult to maintain. Unifying the code and consolidating it in one place will improve the maintainability of the software, as there is only one place where a change can occur. Duplicate copies require careful reading to spot differences. Modern IDEs can spot identical code; however, when the code is similar but not identical, they cannot identify it. The developer must discover each duplicate code to change it. Having identical expressions in two class methods is a sign of duplicated code. Fixing this smell requires extracting the code from both places into one method. If the code is similar but not identical, using different statements allows one to organise it.

Long Function: Fowler et al. [Fow18] found that software with short functions lasts the longest. Short functions have no computation, and the software is an infinite delegation of function calls. However, short functions are easier to explain, to share, and as they are easier to understand, they are also easier to select. Longer functions are harder to grasp, and subroutines of older languages create overhead that discourages short functions. Modern languages have eliminated in-process call overhead. Good naming makes little functions easier to understand, and if a function's name is easy to understand, its body is usually unnecessary. Functions should be decomposed more aggressively; for example, instead of writing a comment, write a function [Fow18]. This function contains the code the developer wanted to comment on but is titled after its purpose, not its function. The key is the semantic difference between what a method does and how it accomplishes it.

Long Parameter List: A method call with a long parameter list indicates bad programming. It indicates that there could be a problem with the implementation. There is no definitive guideline for how many criteria are excessive. Usually, anything over three or four is too many. However, functions require parameters if they have no access to global data. Usually, there is a reason why the symptoms mentioned exist. A method may require more information. The developer may have attempted to create a generic function to handle many scenarios.

Global Data: Since the first days of programming software, we have been told of the horrors of global data. The difficulty with global data is that it may be edited from anywhere in the code base, and there are no means to detect which portion of code touched it. This leads to problems that occur in a running system. Global variables are the most apparent form of global data; however, class variables and singletons also cause problems. Small amounts of global data are manageable, but the more global data exist, the more complicated it becomes.

Mutable Data: Data changes can frequently result in unexpected results and complex issues. When updating some data in one place, developers may need to realise that another software component expects something different. This can result in a failure that is especially difficult to detect if it only occurs in rare circumstances. As a result, a whole programming paradigm is predicated on the idea that data should never change and that altering a data structure should always provide a new copy of the structure with the change, keeping the old data untouched. This paradigm was also introduced in Java with lambdas and streams.

Divergent Change: Software changes over time [Leh80]; thus, the design of the software should allow developers to implement changes with as less effort as possible. Locating the spot to change should also be easy. If the developer cannot do so, it is a sign of the Divergent Change smell. Divergent change happens when one module is frequently altered differently for various reasons. Separating concerns by creating separate modules for each concern mitigates the Divergent Change smell. As a result, when one context changes, the developer needs to grasp only that context and ignore the rest.

Shotgun Surgery: The Shotgun Surgery smell is comparable to but not the same as the Divergent Change smell. Small changes to different classes occur every time the developer changes something in the code, a sign of the Shotgun Surgery smell. When changes are dispersed, they are challenging to locate, and it is simple to overlook a crucial change. Overlooked changes can lead to unforeseen behaviour and even more changes.

Feature Envy: When decomposing and modularising software, the goal is to break the code into modules to enhance interaction inside a module (cohesion) while minimising dependencies between modules (coupling). An example of Feature Envy arises when a function in one module has more dependencies and calls on functions or data in another module than on functions or data in its module. Fowler et al. [Fow18] bring the example of a function calling numerous getters of class to compute some value.

Data Clumps: The data that is semantically related tend to clump together, e. g. in fields of classes or as parameters of functions. These data collections commonly comprise a data clump that is tightly related and often interdependent. As a result, they are frequently utilised together as a group.

Primitive Obsession: Programming languages like Java utilise a set of primitive types that are frequently used, for example, integers, floating point numbers, and strings. According to Fowler et al. [Fow18], programmers must be more open to constructing their own types beneficial in their domain, such as money, coordinates, or ranges. As a result, they implement calculations that treat monetary amounts as simple numbers or calculations of physical quantities that ignore units (increasing inches to millimetres). A correct type typically includes consistent display logic for when it has to be presented in a user interface, if nothing else.

Repeated Switches: Switch statements have fallen out of favour with the developers of object-oriented software. Each switch statement could represent a not implemented polymorphism, and in the first version of “Refactoring” by Fowler et al. [Fow99], they had a bad smell called *Switch Statements*. Due to the advancement of switch statements, they are okay; however, the issue with duplicate switches is that when a developer adds a clause, they must find and update all the switches.

Loops: Loops are part of almost every modern C-like programming language, including Java, C#, or Go. They can be very complex and, thus, hard to comprehend and maintain. However, modern concepts like streams and pipelines allow omitting loops entirely. Pipeline operations like filter and map let the developer easily determine which items are included in the processing and what happens to them.

Lazy Element: Introducing structure to a program element to allow variability or reusability where there is no need to. It could be a function with the same name as its body code or a class that is effectively just one simple function [Fow18]. This is when a function is overengineered with the purpose for later use that never happens. The result is an unnecessarily complex structure and, thus, more difficult to maintain software systems.

Speculative Generality: Speculative Generality is similar to the Lazy Element smell. Instead of a single element, it covers the concept of planned but never used elements and functionality of a system. The added code makes it more difficult for the developer to understand the code, especially when they cannot differentiate between code used and code just there for decorative purposes.

Temporary Field: Setting values of fields inconsistently leads to misunderstandings and code that is difficult to understand. If a field is only sometimes set, it indicates that a class covers multiple concerns.

Message Chains: This smell occurs when, in order to retrieve an object, a chain of other objects is called. This cascading effect leads to a sequence of method calls prone to errors when one part of the chain must be changed. It is also hard for the developer to understand, as they must follow the chain of calls to determine which objects are part of the chain and where the requested object originates.

Middle Man: The Middle Man smell is related to the Message Chains smell, where functionality is delegated too often. This makes it difficult for the developer to locate certain functionality in the software. In the worst case, when the delegating entity is changed, it can also affect the delegated.

Insider Trading: Insider Trading occurs when the modules of a software need to exchange data frequently. This unnecessarily increases the coupling between the modules, and thus, it negatively affects the maintainability of the software system. Removing data exchange between modules is not feasible, as it would create one monolithic module; however, the goal is to reduce the exchange to a minimum.

Large Class: A class that combines multiple concerns tends to be very large. Such a class has too many fields and methods. When a class has too many fields, duplicating code is unavoidable. A class may only use some of its fields all of the time.

Alternative Classes with Different Interfaces: Classes that should be interchangeable must have the same interfaces; otherwise, they are unsuitable as a substitute.

Data Class: Such classes can store and provide data but lack behaviour. Data classes indicate behaviour in the wrong place, so relocating it from the client to the data class can help. One exception is a record used as a function result.

Refused Bequest: Subclasses are intended to inherit the methods and fields of their parents. If they do not use these inherited methods and fields, this is the Refused Bequest. If a subclass reuses behaviour but does not support the superclass's interface, it smells like a denied bequest. Denying implementations does not bother us, but refusing interfaces does.

2.4.2 Bad Smells in Domain-specific Modelling Languages

In this section, we present the bad smells in DSML defined by Hahn and Strittmatter [Hah17; Str20]. They grouped the bad smells according to the methodology by Ganesh [GSS13]. For our contribution, we use the same methodology; therefore, we will present it in more detail in Section 4.2. In addition to the 22 bad smells in object-oriented software, they identified 19 bad smells specific for the domain of DSMLs. In the following section, we briefly overview these 19 bad smells for DSMLs.

Missing Class: When a class contains more than one concept, some attributes and references should be part of another class. It makes it difficult for the modeller to identify the concept of a class; hence it impedes the understandability and changeability of the affected class. The Missing Class bad smell results from a mistake; however, moving the affected attributes and references to a new class fixes it.

Dead Classifier: If the modeller cannot create an instance of a classifier, the classifiers are unusable. Strittmatter differentiates two types of classifiers: the first are classes, and the second and enums. The class misses incoming dependencies, and the enum is not used as an attribute in a class. These dead classifiers make the DSML more difficult to understand and are a proving ground for errors. Fixing this bad smell requires the language developer to delete the affected classes and enums.

Inconsistent Abstraction: If a more package or DSML file depends on a more specialised package or DSML file, the dependencies cross a boundary, resulting in an inconsistent abstraction. These inconsistent abstractions can have negative effects on the maintainability of the DSML, as changes to a more specific package or DSML can affect the more generic one. Performing the dependency inversion refactoring presented by Heinrich et al. [HSR19] fixes this bad smell.

Language Feature Scattering: If classes that are part of a feature are scattered over different packages, especially cross-cutting features, the understandability and maintainability of the DSML are hampered. This bad smell results from changes to the DSML that occur over time. Refactoring this bad smell is to move the affected classes into a common feature.

God Class: A class containing too many properties, such as attributes and references, is hard to understand. It is also a symptom of the amalgamation of multiple concepts in one class. Setting a universal threshold for the size of a class that can be applied to any DSML is impossible; the generation of false positives is inevitable. The developer has to decide whether the God Class is predestined for refactoring. To fix a God Class, the developer separates the properties in accordance with the concepts and moves these properties to new classes accordingly.

Blob Package: If a package contains classes of different language features, it requires more effort to understand the package. Due to the unnecessarily high number of language features in such a package, the developer has to differentiate the different features. Also, they must understand which class belongs to which feature. The result is a package that is difficult to understand and maintain. To fix this bad smell, the developer groups the classes according to their feature and moves them into new packages representing precisely one feature.

Metamodel Monolith: If a DSML file consolidates different language features, it requires more effort to understand the DSML file. Due to the unnecessarily high number of language features in such a DSML file, the developer has to differentiate the different features. Also, they must understand which class belongs to which feature. The result is a DSML file that is difficult to understand and maintain. To fix this bad smell, the developer groups the classes according to their feature and moves them into new DSML files that represent exactly one feature.

Missing Hierarchy: Type information enclosed in a class using attributes and basic data types prevents the developer from adding features selectively. Thus, features are added to more basic classes and the complexity of the DSML increases. Instead of using basic data types, the developer has to introduce specialised types representing the desired information.

Instance Data Modelled by Inheritance: When a type models data that changes during the lifetime of an instantiated object, it models state information instead of type information. Such a structure is achieved by inheritance that models non-type information. The code that works with instances of the DSML is getting more complex, as it must create new objects to handle changes. Also, due to the increased inheritance depth, the DSML is more complex than necessary. To fix this bad smell, the developer must replace the inheritance with one or more attributes.

Redundancies in Hierarchy: If a subclass contains properties identical to the properties of its parent or sibling class, it is a case of redundancies in hierarchy. Sibling classes share the same superclass. Changes to one class affect all other classes with the same properties; these classes can only evolve together. To fix this bad smell, the developer must move the affected properties to the common superclass and remove the properties in all subclasses.

Wide Hierarchy: If a class has an exceeding number of subclasses, the inheritance hierarchy is difficult to comprehend for the developer. Also, the tools that use the DSML become more complex, as no intermediate superclass exists; thus, the type safety is lost [Str20]. To fix this bad smell, the developer must introduce intermediate superclasses to ensure type safety.

Speculative Hierarchy: An abstract class is created to allow multiple classes to inherit common properties; however, if only one class inherits from the abstract class, it introduces unnecessary complexity to the DSML. The abstract class is generated with the expectation that, in the future, more than one class will inherit from it. However, if no class inherits from the abstract class, the inheritance relationship has been introduced unnecessarily and only complicates the DSML. The developer moves the properties from the abstract class to its only subclass to fix this bad smell.

Deep Hierarchy: Theoretically, the chain of classes formed by inheritance relations can be arbitrarily deep. In reality, the deeper the chain is, the more complex it becomes; thus, it becomes more and more complex for the developer to comprehend such a structure in the DSML. Combined with other smells like Speculative Hierarchy or Instance Data Modelled by Inheritance, the fixes for these smells help to deal with the Deep Hierarchy smell. Otherwise, if these other bad smells do not occur, the developer can merge classes in the chain to reduce the depth.

Multipath Hierarchy: If a class inherits via different paths from one class, it either contains unnecessary dependencies or the concept is so complex that it requires such a structure. If the dependency structure contains unnecessary dependencies, it makes the DSML unnecessarily complex. Removing one path is not easy if none of the paths is a direct dependency because the inherited properties of the classes in between would get lost. As a result, the developer can delete all direct dependency paths. The remaining paths must be analysed to determine whether they contain unnecessary dependencies, i. e., inherited properties are not used.

Concrete Abstract Class: Classes in a DSML that are concrete but are never instantiated should be abstract. Also, the supertype of multiple types and the concrete supertype of abstract classes should be abstract. Such classes make it difficult for the tool user to create instances of the DSML, as the pool of classes they can instantiate is bigger than it should be. The developer declares the affected class as abstract to fix this bad smell. They must remember that instances that use the affected classes are not invalidated. To circumvent the problem, the developer can provide a transformation that fixes the instances if needed.

Dependency Cycle: Dependency cycles can occur between different entities of an DSML. Strittmatter [Str20] differentiates three levels: cycles between classes, packages, and DSML files. They all have in common that they make it difficult for the developer to understand the DSML. They also worsen the maintainability of the DSML, as the effort required to implement a change is unpredictable. Heinrich et al. introduce class and package level refactorings to fix Dependency Cycles in DSMLs [HSR19] to fix this bad smell.

Container Relation: The Container Relation smell is exclusively present for DSMLs that were created with the Eclipse Modelling Framework (EMF). In the context of this dissertation, we do not exclusively focus on EMF-based DSMLs; however, the foundational work on which our approaches are based uses the EMF-based DSMLs. Therefore, we include the description of the Container Relation smell. In EMF, if a dependency is a containment dependency is already defined by the framework; however, it is still possible to create an explicit containment dependency. According to Strittmatter [Str20], adding such a dependency adds unnecessary complexity to the DSML, especially when a class has multiple containment dependencies. The tool developer must handle these additional dependencies when the container is deleted; the dependency remains. Either the tool user has to remove the remaining dependency, or the tool must implement a routine that removes such pointless dependencies. If the user has to deal with it, it is an inconvenience for them, but if the handling is integrated into the tool, each change to the dependency structure will result in changes. To fix this bad smell, the developer must delete the containment dependency.

Obligatory Container Relation: The Obligatory Container Relation is when a type is contained in multiple containers, but one of these dependencies has a multiplicity of one. As a result, the type is always contained in the container with a multiplicity of one. The other dependencies are, therefore, useless; they only increase the complexity of the DSML. To fix this bad smell, the developer removes the dependency with the multiplicity with one or all other dependencies.

Specialised Relation: Suppose the relation structure is redundant; for example, when on the abstract level, the same relation exists as on a more concrete level. In that case, it impedes the understandability and maintainability of the DSML. Due to the additional relations, it is difficult for the developer to determine whether they are legit or redundant. If the relation on either level changes, all other redundant relations must also be changed. To fix this bad smell, the developer removes the specialised relation so that the most generic remains.

2.5 Foundational Concepts for the Reuse of Model-based Analysis Components

In this section, we present supplemental concepts required to follow the contents of our third contribution, the reuse of model-based analysis components.

2.5.1 Satisfiable Modulo Theories

The Satisfiable Modulo Theories (SMT) problem is a decision problem concerned with the satisfiability of first-order logic formulas with respect to a background theory. Examples of possible theories are the theory of integers or the theory of real numbers. An SMT-Solver is a software that solves this problem for as many instances as possible [DB11]. The SMT problem is NP-hard, and as a result, it can be undecidable; the high computational complexity of the SMT-problem forces most solvers to focus on classes of SMT-instances that appear in practice [MB09]. As a result, researchers try to find decidable classes of theories to extend the number of decidable SMT problems.

A formula P is *satisfiable* if values are assigned to its function symbols so that P evaluates to true. P is *valid* if P evaluates to true for all possible assignments of function symbols. The relationship between validity and satisfiability can be expressed as follows:

$$P \text{ is valid } \iff \neg P \text{ is not satisfiable}$$

or alternatively:

$$P \text{ is not valid } \iff \neg P \text{ is satisfiable}$$

An SMT-Solver will only check a formula P for satisfiability and output a valid assignment of values to function symbols if the solver can find such an assignment. With this connection between satisfiability and validity, the solver can determine if a formula P is valid by determining if $\neg P$ is (not) satisfiable.

SMT-LIB: The SMT-LIB standard [BFT17] defines a language to specify instances of the SMT-problem. Based on the selected background theory, variables can be declared as function symbols and conditions on these variables. Listing 2.1 shows the declaration of an integer variable z and a condition that requires that $z^2 = 25$.

```
(declare-fun z () Int)
(assert (= (* z z) 25))
(check-sat)
```

Listing 2.1: SMT Declaration Example

The condition is satisfiable and any SMT-solver should find a valid assignment of z (e. g., $z = 5$). The condition can be negated and rechecked for satisfiability to check the formula for validity, as shown in Listing 2.2.

```
(declare-fun z () Int)
(assert (not (= (* z z) 25)))
(check-sat)
```

Listing 2.2: SMT Validity Example

The result will show that the second formula is satisfiable (e. g., for $z = 15$), i. e. the solver found a counterexample to the validity of the original formula, implying that the original formula is not valid.

2.5.2 P versus NP

In theoretical computer science, the P vs NP (Polynomial Time (P) vs Nondeterministic Polynomial Time (NP) problem is an unsolved complexity theory problem. The question is whether the set of problems that can be solved quickly (*P*) and the set of problems that can be checked for correctness fast (*NP*) are identical. Quickly solvable or checkable implies the existence of an algorithm that solves the problem; the computational effort (number of computational steps) is bounded by a polynomial function depending on the amount of the input. The input size is the number of elements fed into the algorithm. For example, when sorting index cards, the size is the number of index cards.

For all problems that can be solved quickly, one can also quickly check the correctness of a solution. This needs to be clarified in the opposite direction: For some problems, an algorithm exists that can quickly check a proposed solution, but neither could an algorithm be found that also quickly finds a correct solution nor could the impossibility of such an algorithm be proven. Thus, the question is unsolved. If one were to find an algorithm for all quickly testable problems *NP* that also solves them quickly, then $P = NP$ would apply.

If it could be shown for at least one problem from *NP* that it cannot be solved quickly in principle, $P \neq NP$ would be proved.

2.5.3 Nondeterministic Polynomial Time

NP is a complexity class for decision problems where there is evidence for “yes” answers that can be verified efficiently (in polynomial time). However, it can sometimes take time to find such proof. So an alternative description of NP is the class of all decision problems that can be solved by a non-deterministic Nondeterministic Turing Machine (NTM) with respect to the input length in polynomial time. Here, an instance is answered “yes” if at least one of the possible computations of the non-deterministic Turing machine answers with “yes” [KT06].

NP-complete problems probably cannot be solved efficiently. All known deterministic algorithms for these problems require exponential computational effort, and it is a strong hypothesis that there are no more efficient algorithms. Confirming or disproving this hypothesis is the P-NP problem, one of computer science’s most important open problems. The best-known NP-complete problem is the travelling salesman problem.

2.5.4 Graph Isomorphism

When two graphs are isomorph, they have a bijection between their vertex sets that preserves the adjacency between them [McK+81]. An *automorphism* is an isomorphism that exists between a graph and itself [MP14]. Figure 2.5 shows two graphs that are at

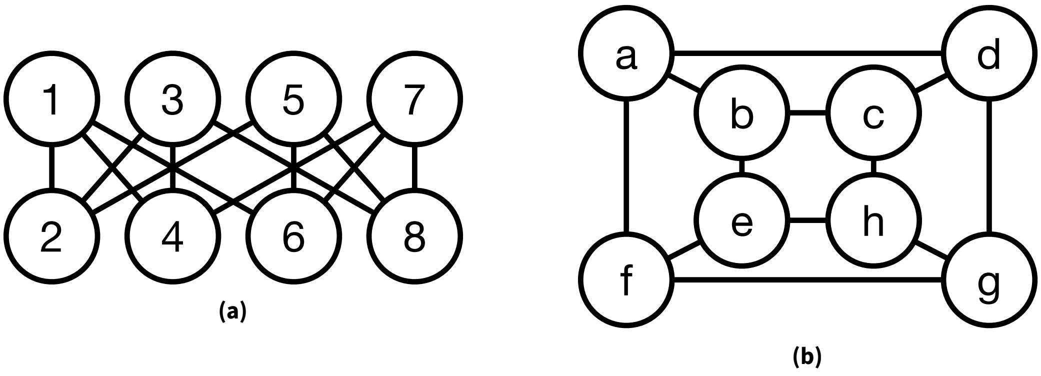


Figure 2.5.: Two Graphs with the Same Number of Nodes and Edges but Different Node Names

first glance not identical; however, the graph isomorphism analysis allows us to determine whether these two graphs are not identical. A bijection between their vertex sets and preserving the adjacency of the vertex sets shows that the two graphs are structurally identical. The result is $1 \rightarrow a, 2 \rightarrow b, \ldots, 8 \rightarrow h$. The nodes’ and edges’ names and identifiers are not taken into account in the graph isomorphism analysis.

The collection of all automorphisms of a graph G is called the automorphism group Aut(G) [MP14]. The graph isomorphism problem is concerned with determining if two given graphs are isomorph i. e. if they are structurally identical. The nodes' and edges' names and identifiers are not considered when determining if two graphs are isomorph. The graph isomorphism problem is known to be one of the NP problems, but it needs to be clarified whether it is NP-complete. From 1984 until 2015, the fastest running time was set at $e^{O(\sqrt{(n \cdot \log n)})}$ by Babai et al. [BKL83]. Since 2015 the fastest running time is set at $e^{(\log n)^{O(1)}}$ also by Babai [Bab16].

2.5.5 Domain-specific Language

A Domain-specific Language (DSL) is a formal language designed and implemented to ease the interaction between humans and computers for a specific domain. Due to DSLs, key aspects of a domain can be formally expressed and modelled [SVC06]. A DSL possesses a metamodel, including its static semantics and corresponding concrete syntax. When designing a DSL for a domain, the goal is to have a high degree of problem specificity: The DSL should focus on the problems of the domain, and it should exclude anything that is not part of the domain. This makes the DSL usable by domain experts without special knowledge about general-purpose programming languages. The opposite of a DSL is a general-purpose programming language, such as Java or C++, or a universally applicable modelling language, such as UML. The benefits of a DSL are that the domain experts can focus on the problem at hand without worrying about the syntactic specificities of a general-purpose language. Due to the reduced complexity of a DSL, the effort to learn a DSL is less than learning a general-purpose language. One drawback is that the semantics of a DSL must be well documented [SVC06]. Another challenge is that the semantics of the DSL must be intuitively clear to the modeler [SVC06]. DSL adopts concepts from the problem space so that a domain expert will recognise its "domain language". According to Stahl et al. [SVC06], the semantics of a DSL are relevant when the modeller must know the semantics of the language entities so that they can create reasonable models.

2.6 Foundation of the Evaluation

In this section, we present the foundation for the evaluation we used throughout this thesis. We introduce the validity types that Runeson et al. [Run+12] defined that we used throughout every evaluation in this thesis, and then we introduce the Goal Question Metric (GQM) approach by Basili et al. [BCR94].

2.6.1 Validity Types

We use case studies to evaluate our contributions. To determine the validity of our approach, we use the four types of validity introduced by Runeson et al. [Run+12]. They distinguish four types of validity for case study research in the area of software engineering: *Internal, external, construct, and conclusion validity.* The better the case studies and the evaluation addresses the validities, the more weight the conclusions we can draw from the results.

Internal Validity: In the case study-driven evaluation, the conductor of the experiment should be able to link the effects observed in the case study to a cause. The internal validity type deals with this circumstance. It concerns the cause and effect and whether the observed effects can be linked to a cause. Regarding cause and effect, the conductor must also consider possible side effects. In the best case, the effect is traceable to a specific cause. This validity type is compromised when effects in the observed case study have unknown causes.

External Validity: Case studies represent a type of system that, ideally, allows us to conclude these types of systems in general. The external validity type deals with the ability to generalise the conclusions drawn from the case study. External validity is compromised if the case study sample is not sufficiently diverse to support valid conclusions for the supposed scope to which the conclusions should apply.

Construct Validity: The case study and the evaluation must be constructed to measure the desired information. In the case of this thesis, we have to ensure that the metrics we use to measure our case studies analyse the desired property. This validity type is compromised when the metrics are inappropriate for the desired case.

Conclusion Validity: Science is based on the repeatability of experiments, the reproducibility of results and especially the unambiguousness of the conclusions drawn. Therefore, the aim is that other researchers can conduct the evaluation and that they also obtain the same results. The results should not leave room for interpretation. This validity type is compromised, for instance, when the data and the tools are unavailable or when the process of how the evaluation is conducted is unknown.

2.6.2 Goal Question Metric Approach

With software evaluation in mind, the GQM was developed by Basili et al. [BCR94]. The idea of the GQM approach is to derive research questions and metrics from the goals we want to achieve. In 2008, Koziolek [Koz08] stated that the GQM approach could also be applied to evaluate approaches in other engineering domains. In contrast to bottom-up processes, where the metrics are selected without a concrete goal, Basili et al. [BCR94] proposes a top-down process, where defining the goal is the first step of the evaluation. Defining the goal first prevents the scientist from using metrics that measure irrelevant attributes in their evaluation. The process of the GQM is as follows: First, the goals of the topic that should be examined are specified in detail. For example, we are implementing an approach to compare software components to find software artefacts that could be reused in another software project. There may be one or more questions that are defined for a goal. For example, does the compare algorithm find all potentially fitting software artefacts that could be reused? There may be one or more metrics that are defined for a query. In our example, the metrics precision, recall and F1 are suited to determine whether all software artefacts are identified, no software artefact is wrongly identified, and no software artefact was forgotten. Following the measurement, the GQM strategy will undergo a bottom-up evaluation. The questions can be answered by examining the metrics in question. How the questions were answered enables us to draw judgments regarding their objective.

For each contribution, we provide a plan according to the GQM approach. First, we define the goals we want to achieve by evaluating our contributions. Therefore, to identify whether we have achieved our goals, we derive questions we answer. To answer the evaluation questions, we define metrics whose results we use to get a quantitative measure.

2.7 Technical Foundation

In this section, we present the technical foundation, which includes the third-party tools we use throughout this thesis.

2.7.1 Eclipse Modelling Framework

The EMF [Ste+09] is a framework for model-driven software development. It provides various tools and frameworks to help develop metamodels and domain-specific languages. EMF utilises the Ecore metamodel description approach to describe metamodels. Ecore includes fundamental object-oriented modelling concepts such as packages, classes, references and attributes. Ecore is based on a subset of the Meta-Object Facility (MOF) metamodelling standard. EMF had a direct influence on the formulation of the EMOF standard. As a result, Ecore serves as a reference implementation of EMOF. The EMF also contains tools for generating Java classes, APIs, and graphical editors for model generation and manipulation. When designing UI elements such as editors, it is possible to rely on already provided functionality thanks to the integration with the Eclipse framework. These UI elements offer only a bare minimum of functionality, but this is often all required. Other Eclipse projects, such as Sirius, can be utilised if more complex editors are necessary.

2.7.2 Xtext

Xtext is a framework for creating domain-specific languages that integrate with the EMF [Bet16]. It uses a grammar-based approach to generate a parser, linker and editor from a specified grammar; however, Xtext can also integrate existing metamodels as part of the abstract syntax. The language can then be extended by custom scoping, validation and code generation. Xtext uses the Extended Backus–Naur Form (EBNF) syntax notation to describe the terminal rules of the grammar. More details regarding Xtext and its grammar language can be found at [ES21].

```
(nodes)-[:relation]->(anotherNodes)
```

Listing 2.3: Cypher Syntax

2.7.3 Xtend

Xtend is a general-purpose programming language that compiles Java code, making it interoperable with Java [Bet16]. Because of its template engine, Xtend is often used to implement code generation for DSLs defined with Xtext. The functional aspects of Xtend, such as Lambda-expressions and graph-transformation tools such as the create-methods, make it suitable as a model transformation language.

2.7.4 Neo4J

Neo4J is a graph database that is implemented in Java [1]. It is developed as Open Source software, and it has been available since 2010. Neo4j is an embedded, disk-based, transactional database engine that stores data structured in graphs instead of tables. In Neo4j, information is stored as an edge, a node or an attribute. A node can have any number of attributes, and nodes and edges can have a label. The database uses schemas for indexing, available via the query language Cypher. Cypher is a graph query language that allows the user to retrieve data from the graph. The query language is comparable to SQL in relational databases. The Cypher language has a syntax that visualises nodes and edges.

Listing 2.3 shows the syntax of the Cypher query language. Nodes are enclosed in rounded brackets, and relations between nodes are represented as arrows. The type of relation can be specified by using square brackets.

2.7.5 Spoon

Spoon is an open-source library that allows the user to analyse, rewrite, and transform Java source code. One benefit of Spoon is that the source code must not compile in order for Spoon to analyse it. Especially when the developer wants to analyse only parts of an model-based analysis, they are not required to provide compilable source code. Spoon uses a metamodel to represent the java source code, and its instances resemble an Abstract Syntax Tree (AST). However, it is reduced in its complexity compared to the metamodel of Sun's compiler (`javac`). Other than a compiler-based AST (such as that provided by javac), the Spoon metamodel of Java is intended to be easily understood by ordinary Java developers, allowing them to build their own programme analyses and transformations. The Spoon metamodel provides all information needed to generate compilable and executable Java programmes (hence contains annotations, generics, and method bodies) [Paw+15].

[1] https://www.neo4j.com

Part II.

Improving Evolvability and Reusability of Model-based Analyses

3. Decomposition and Composition of Model-based Analyses

Changes made to the software, especially when done under time and financial constraints, have the potential to reduce software quality significantly. Developers can utilise model-based analyses to examine the impact on the quality of foreseen changes in software systems before performing changes. Using model-based analyses can prevent bad repercussions on software quality, such as performance drops, reduced reliability, or security breaches. One type of these analyses is known as *model-based analysis*; such analyses derive and communicate information on the quality of a software system by using modelling languages and models of software systems [ZMK18]. In addition to the system, a model-based analysis examines, the model-based analyses themselves are also prone to changes over time. As a result, historically grown model-based analyses suffer from increasing complexity and deterioration of internal software quality. If the model-based analysis is not adapted to the changed requirements, it becomes less relevant for the users.

Analysis developers must also adapt model-based analyses to changes of their corresponding DSML, see Section 2.1.3. A corresponding DSML is the metamodel on which the input models of the model-based analyses are based on. DSMLs evolve over time, and therefore, they are also prone to decline quality, even if they evolve more slowly than software systems [HSR19]. New features that are added to the DSML ideally do not affect the corresponding analysis. When the changes are made in a non-intrusive way, for example, when the DSML is developed according to the reference architecture for DSMLs by Heinrich et al. [HSR19]. A non-intrusive change can be a new language feature that inherits properties from an existing language feature; for example, an analysis developer adds to a performance simulation of software systems the TCP/IP stack simulation to simulate remote calls. If a corresponding model-based analysis needs no use of the newly added feature, an analysis developer must adapt the model-based analysis to the feature of its corresponding DSML.

Due to changes to the DSML and the resulting, inevitable change of the model-based analysis, model-based analyses become more complex over time. Such historically grown model-based analyses are hard to evolve and to reuse [KHR22a]. Ideally, analysis developers have no issues understanding the source code of the model-based analysis to implement or change a feature without introducing the technical debt. The more complex the code base of an model-based analysis is, the more time needs an analysis developer to understand the code base and the more time a developer need for the implementation.

In addition to a general deterioration in quality brought on by evolution, model-based analyses are notoriously difficult to reuse because they are frequently bound to a particular domain and DSML. Our goal is to reduce the complexity and to improve the understandability and reusability of model-based analyses. Therefore, instead of having two different architectural concepts, one for the model-based analysis and one for the DSML, we investigate whether we can apply the same architecture to the model-based analysis and its associated DSML.

Since the beginning of software development, developers have aimed to divide software into smaller pieces to make the software more manageable. *Divide and Conquer* is the strategy for decomposing software into smaller entities to reduce the complexity of the overall system. In OOP, classes are such entities to encapsulate concerns. Heinrich et al., for example, propose a reference architecture suited to DSMLs in order to enhance the evolvability and reusability of DSMLs [HSR19]. In software engineering, reference architectures provide a general architecture for applications in a certain domain. For example, the Java EE architecture is a layered reference design for developing Java applications, according to [Som18]. In software engineering, developers can choose from a myriad of architectural patterns that propose how to decompose and modularise a software system [Ric15].

Metamodel design and object-oriented design have many similarities. Composition, acyclic dependencies, dependency inversion, and layering are all principles that can be transferred from DSMLs to model-based analysis [HSR19]. Although model-based analyses are also software systems and approaches that improve the quality of software systems (e. g., separation of concerns, fixing bad smells) are also applicable to model-based analyses, there are no solutions that are tailored to the co-evolution of model-based analyses and their corresponding DSMLs to the best of our knowledge. However, to the best of our knowledge, no approach considers the co-dependency of model-based analyses and their corresponding DSML.

When discussing reference architectures for model-based analyses, we use the term reference architecture for convenience. Our research is aimed at the relationships between a DSML and the model-based analyses that go with it. We apply our experience of DSML structuring, development, and refactoring to model-based analyses. Our reference architecture enables independent development of analysis features and serves as a template solution for methodically extending and reusing components of model-based analyses. Features that have a representation in a DSML and a matching model-based analysis can evolve and be utilised and reused together as a result of our research. In order to accomplish this goal, the reference architecture establishes a framework for model-based analyses that take into account the corresponding DSML. Therefore, we investigate how the evolvability, understandability and reusability of model-based analyses are affected when the structure of the associated DSML (i. e., layers, constraints of dependencies, the introduction of features) is transferred to the structure of model-based analyses.

In this chapter, we present our first contribution: a reference architecture for model-based analyses that takes the structure of the corresponding DSML into account to improve the evolvability and reusability of model-based analyses: (i) decompose an already existing model-based analysis, (ii) compose a model-based analysis, and (iii) develop a model-based analysis from scratch.

The chapter is structured as follows:

- After presenting the hypothesis and research questions in Section 3.1, we use an application scenario to derive requirements for our reference architecture, which we then present in Section 3.2.
- Section 3.3 provides our decomposition approach for the reference architecture. To create these concepts, we have applied insights gained from the reference architecture for DSMLs to model-based analyses and made adjustments where necessary. To this end, we introduce the reference architecture tailored to model-based analyses. Likewise, we present a concrete instantiation of our reference architecture with layers tailored to model-based quality analyses and a set of features that are required by model-based quality analyses.
- In Section 3.4, we introduce our composition concept for model-based analyses.
- In Section 3.5, we provide guidelines for implementing the reference architecture. We present three processes on how to apply our approach, considering three application scenarios: (i) refactoring an already existing model-based analysis, (ii) developing a model-based analysis from scratch, and (iii) extending a model-based analysis.
- In Section 3.6, we present the technical foundation that we developed to decompose and compose model-based analyses.

3.1 Hypothesis and Research Questions

In this section, we present research questions for the first contribution of this thesis. Heinrich et al. [HSR19] proposed a reference architecture for DSMLs that separates a DSML into language features and then distributes these language features on different layers. Each layer contains a set of features of the DSML. The reference architecture for DSMLs provides a clear structure and extension mechanisms for DSMLs. To refactor existing, monolithic DSMLs, they also provide refactoring operations on class and package level. Due to the reference architecture for DSMLs, the refactoring operations and processes to modularise monolithic DSMLs, they have shown that their approach improves the evolvability and reusability of DSMLs. Unfortunately, their approach only considers DSMLs, software that uses these DSMLs do not gain any advantage regarding evolvability or reusability. On a conceptual level, the DSML and its corresponding model-based analyses, are separated. Although the DSML is modularised and separated into layers, its corresponding model-based analyses can still be monolithic or follow a totally different architectural pattern. As a result, analysis developers must understand the semantics of the DSML and, additionally, must invest their time to understand the model-based analysis and how it is using the DSML. We raised the question: “What if the DSML and its corresponding model-based analyses follow the same architectural pattern?”. To the best of our knowledge, there have been no studies to date that have investigated whether the evolvability, understandability, and reusability of model-based analyses improves, when the same architectural pattern that is used by the DSML is also applied to the model-based analysis. In this thesis, we will focus on the pattern of the reference architecture for DSMLs by Heinrich et al. [HSR19]. Thus, we derive the following hypothesis for transferring concepts of DSML development to model-based analysis development:

Hypothesis 1

The evolvability, understandability, and reusability of a model-based analysis will improve when transferring the concepts of the reference architecture for DSMLs to model-based analysis.

The results of [HSR19] show that the reference architecture improves the evolvability of DSMLs and it improves the need-specific use and reuse of language features. Ideally, introducing a layered structure to a model-based analysis that resembles the structure and the semantics of the corresponding DSML has the same positive effect on model-based analyses.

We formulate the following research questions to determine whether the hypothesis 1 is correct.

Research Question 3.1

Does using an isomorphic structure, which corresponds to the reference architecture for modelling languages, improve the evolvability of model-based analyses?

The ability of a model-based analysis to evolve determines whether a model-based analysis will stand the test of time. If the developers cannot implement new features or change features to adapt the model-based analysis to changed requirements, the model-based analysis will lose relevance for the user. As a result, the model-based analysis falls out of users' favour and is no longer used. Using the same concepts in both DSMLs and model-based analysis, allows developers to transfer their knowledge about the structure and the semantics of the DSML to their corresponding model-based analyses and vice versa. Even when the analysis developers cannot transfer their understanding of the DSML to the model-based analysis, for example, because they have no knowledge about the DSML, having a modular model-based analysis should be better evolvable than its monolithic counterpart. When each language feature in the DSML also has a feature representation in the model-based analysis, developers can locate the spots to apply the changes in the model-based analysis by consulting the DSML.

Research Question 3.2

Does using an isomorphic structure, which corresponds to the reference architecture for modelling languages, improve the understandability of model-based analyses?

Understanding a model-based analysis and determining the use and semantics of language features in the analysis is a non-trivial task. For the developer of the model-based analysis it can be unclear whether a language feature is used in the model, whether the language feature is used correctly in the model-based analysis, how it will affect the analysis result, or if it affects the result at all [Hei+21b]. When the features of a DSML and a corresponding model-based analysis are identical (i. e., have the same structure and semantics), the developer can locate a analysis feature by investigating the features of the DSML. Having an identical feature structure, analysis developers can identify whether a language feature is used in the model-based analysis. When it is easier to identify the usage, the analysis developers can focus on understanding how the language feature is used and how it affects the analysis result. Even if the analysis developer has no knowledge about the DSML, they can use the DSML as starting point to understand the structure of the system the model-based analysis reasons about. We also assume that using a modular and layered architecture with a fixed set of rules and constraints for model-based analyses will improve the understandability, reducing the complexity of a model-based analysis.

Research Question 3.3

Does using an isomorphic structure, which corresponds to the reference architecture for modelling languages, improve the reusability of model-based analyses?

The monolithic structure of a model-based analysis does impede the reuse of analysis features in other model-based analyses, according to Heinrich et al. [Hei+21b]. When the use of language features in a model-based analysis is not separated, i. e., one component of the model-based analysis has dependencies on all language features, the benefit of having feature configurations is irrelevant, because all language features must be used no matter the configuration. The modular structure of a DSML developed according to the reference architecture of Heinrich et al. allows the reuse of dedicated features [HSR19]. Transferring the modular structure of a modular DSML to a monolithic model-based analysis introduces a modular feature structure to the model-based analysis. Thus, by introducing a modular structure to a model-based analysis, the reusability of analysis features should be improved.

3.2 Requirements for the Reference Architecture for Model-based Analyses

Based on our research questions, we derive requirements for a reference architecture for model-based analyses that supports evolvability, understandability, and reusability. Before we can derive these requirements, we introduce an application scenario based on an illustrative example model-based analysis. We use this application scenario and the research questions to derive the requirements for reference architecture for model-based analyses.

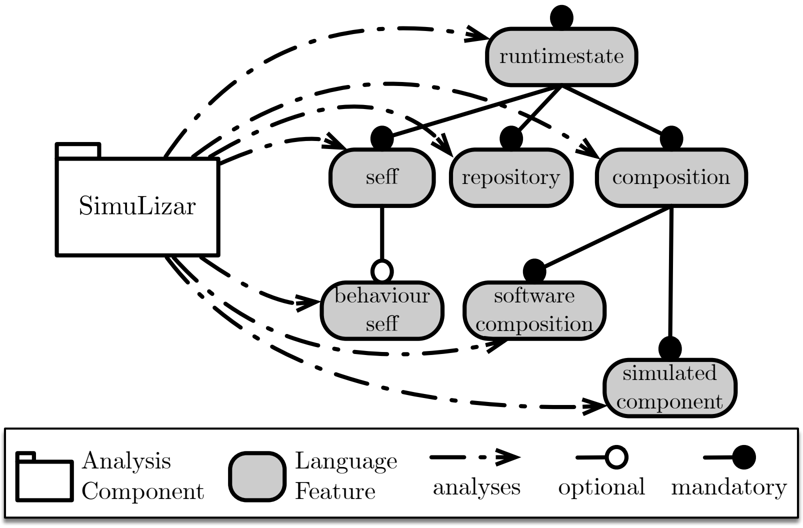


Figure 3.1.: Illustrative Example of the Dependency Structure of SimuLizar and the Palladio Component Model (PCM)

We use the *Palladio Simulator* as an illustrative example in our application scenario. As part of the *Palladio Approach* [Reu+16], the Palladio Simulator allows software architects to analyse a software architecture based on a model. The Palladio Simulator allows analysing such an architectural model regarding different quality properties, such as performance [BKR09], reliability [Bro+12], maintainability [Ros+17], and security [Sei+22; WHR22]. The central component of the Palladio Approach is the Palladio Component Model (PCM). The PCM is a DSML that allows the software architect to specify and to document software architectures. For our application scenario, we focus on the performance analysis part of the Palladio Simulator. The performance simulator of the Palladio Simulator is called SimuLizar. SimuLizar does interpret the PCM to derive performance information about the system under study; however, due to the size and deprecated language features of the PCM, SimuLizar does not support all features of the PCM. SimuLizar is a historically grown model-based analysis, with the typical deterioration of the internal quality over time. In our chapter about the case studies (Chapter 6), we present more details regarding SimuLizar and other historically grown model-based analyses. In this section, we focus on the shortcomings of SimuLizar that influence its evolvability, understandability, and reusability.

Project Structure Erosion: The model-based analysis SimuLizar has been continuously extended and maintained since 2013. The continuous development resulted in software corrosion [Par79]. Over time, more and more features were added to SimuLizar. Figure 3.1 illustrates the dependency structure of SimuLizar and the PCM. All dependencies on the PCM are bundled in one component of SimuLizar, depicted as a white rectangle. A simplified version of the features of the PCM is depicted as grey rectangles with rounded corners. In its first version, SimuLizar was only capable of performing design-time performance analysis for self-adapting systems.

Uncontrolled Dependency Growth: The features with dependencies on language features were added to the same component of SimuLizar, where all dependencies on the language features are located, resulting in a model-based analysis that has to ship every feature, depends on a language feature, even if the language feature is not required. The increasing amount of features resulted in a growing, monolithic structure that led to uncontrolled dependency growth.

Feature Drift: This monolithic structure deteriorated evolvability, understandability, and reusability. As a direct consequence, evolvability decreased, and *feature drift* occurred. The term “feature drift” refers to the process by which developers add unnecessary features to a system for the end user. Due to feature drift, users will need help determining whether the simulated DSML features impact the analysis outcome.

We use these shortcomings (i. e., project structure erosion, uncontrolled growth of dependencies, and feature drift) to derive the following requirements for the reference architecture for model-based analyses:

- **Requirement R1** (Improved Evolvability): The first requirement is that a model-based analysis that is developed according to our reference architecture for model-based analysis has better evolvability than a model-based analysis that is not developed according to our reference architecture. We assume that when the analysis architect applies the reference architecture to a monolithic model-based analysis, the evolvability of the model-based analysis improves. We determine the evolvability of a model-based analysis as *good* when the model-based analysis has a low complexity, a low coupling, and a high cohesion (cf. Section 7.2).
- **Requirement R2** (Non-intrusive Extension): When the analysis architect extends features of a model-based analysis, and they have to make changes to the existing analysis features, other analysis features might be affected by such an extension. Such intrusive extensions hamper the evolvability of model-based analysis, because the effort to add an extension needs to be clarified. Also, due to the dependencies of a more general feature on an extension, the feature can not be reused without the extension. To avoid such intrusive and unnecessary changes to existing analysis features, analysis developers should not alter the components they want to extend. Therefore, we require that the reference architecture for model-based analysis must guarantee that components of the remaining model-based analysis have no dependencies on the extension. Extensions that do not alter the analysis features they extend are non-intrusive. Non-intrusive extensions avoid the detrimental effects of such dependencies.
- **Requirement R3** (Consistent Dependencies): According to our first hypothesis, we assume that the analysis architect uses the structure of the DSML to derive the structure of model-based analysis. If the features of the DSML and the features of the model-based analysis are *not aligned*, changes to the DSML can have unpredictable effects on the model-based analysis. Features are not aligned when features and feature dependencies of the DSML are different from the features and feature dependencies of the analysis. In the end, the DSML and the model-based analysis are

inconsistent. Such inconsistencies occur when the dependencies of the DSML and its corresponding model-based analysis are not aligned. These inconsistencies make it difficult for the analysis developer to determine the effort required to adapt the model-based analysis to changes made in the DSML. It is impossible to predict the development time of a feature, leading to a delayed feature release and the delay of additional features. As a result, we require that the feature structure of reference architecture for model-based analysis is consistent with the feature structure of its corresponding DSML.

- **Requirement R4** (Need-specific Reuse): Due to the monolithic nature of SimuLizar and model-based analysis in general, it is hard for analysis developers to reuse features of one model-based analysis in another. Also, analysis developers need to understand a model-based analysis and its dependency structure as a whole to extract components. It is especially tedious when the analysis developer must understand all available features, although they do not want to reuse all these features of the model-based analysis. These unnecessary dependencies can confuse and require additional effort to comprehend by the analysis developer. Thus, we require that the reference architecture for model-based analyses allows the analysis architect to select analysis features for reuse without affecting other analysis features.
- **Requirement R5** (Need-specific Use): When developing tools for composing a model-based analysis, the developer of those tools needs to have a solid understanding of the DSML as well as the model-based analysis that is based on DSML. The complexity of creating such a model-based analysis presents a barrier to creating useful tools. Additionally, the final model-based analysis may incorporate features that are not used by the analysis that the tool user is performing. To address this shortcoming, we require that the reference architecture allow the tool developer to selectively use components of the model-based analysis based on their demands.

3.3 Decomposition of Model-based Analyses

For decomposing a monolithic model-based analysis, we assume that the models used by the analysis follow a DSML. We also assume that the DSML is already modularised according to the reference architecture for DSMLs by Heinrich et al. [HSR19]. This assumption results from our Hypothesis 1 that the concepts of the modular DSML can be transferred to the model-based analysis. In this section, we present our approach for decomposing model-based analyses. In Section 3.3.1, we present concepts for the modularisation of model-based analyses. To realise our modularisation concepts, we divide the decomposition of model-based analyses into two parts. First, we introduce an instantiation of our reference architecture for model-based analyses using the domain of model-based quality analyses as an example. In the example, we provide a set of five layers in our reference architecture (cf. Section 3.3.2). Second, we provide tools for the analysis developer to refactor a monolithic model-based analysis into a modular model-based analysis (cf. Section 3.3.3). We provide a set of refactoring operations to apply our reference architecture to existing model-based analyses. We differentiate between refactorings on the class level (cf. Section 3.3.3.1) and refactorings on the component level (cf. Section 3.3.3.2).

3.3.1 Modularisation Concepts for Model-based Analyses

We utilise feature models to represent features of model-based analyses. The feature structure we introduced in this thesis is based on the layered reference architecture for metamodels presented by Heinrich et al. [HSR19]. They differentiate language features and language components (formerly defined as language module). Each language feature and language component is placed at exactly one layer. In the context of the reference architecture for DSMLs, a layer refers to a logical grouping of related language features and language components. Language features and their corresponding language components must be located on the same layer and only be placed on one layer. Although Heinrich et al. [HSR19] do not set the number of layers for DSMLs, they propose a four-layered reference architecture that is tailored to DSMLs for quality modelling and analysis.

We show the concepts of our extended layered reference architecture for model-based analyses in Figure 3.2. As long as the DSML follows the reference architecture of Heinrich et al., the following modularisation concepts can be applied to the model-based analysis that uses the DSML for analysis. In our reference architecture, we distinguish features of languages and model-based analyses, and components of languages and features. As in the work of Heinrich et al., we place each feature and component at one layer. In the following sections, we present our extended reference architecture in more detail and discuss the reasoning behind our design decisions. In the remainder of this thesis, we refer to the extended reference architecture as reference architecture for model-based analyses.

3.3.1.1 Use of Feature Models

We use feature models to structure the features of model-based analyses. Each node in the feature graph represents either a feature of the DSML or a feature of the model-based analysis. The analysis architect defines the feature dependencies in the feature model. This dependency structure already restricts the dependencies at the feature level. The dependency structure limits the possibility of the dependencies of the model-based analysis; as a result, the complexity for the analysis component developer is limited. Additionally, it provides guidance for the analysis component developer.

In contrast to feature modelling from, for example, the product line community, our concept allows multiple root features to group other features. Multiple root features are required to represent the concept of atomic analysis features and composed analysis features. We also ban cycles in our feature models. Cycles can prevent the analysis developers from selecting features individually, and cycles also prevent a straightforward transition through the graph. The straightforward transition allows selecting features that

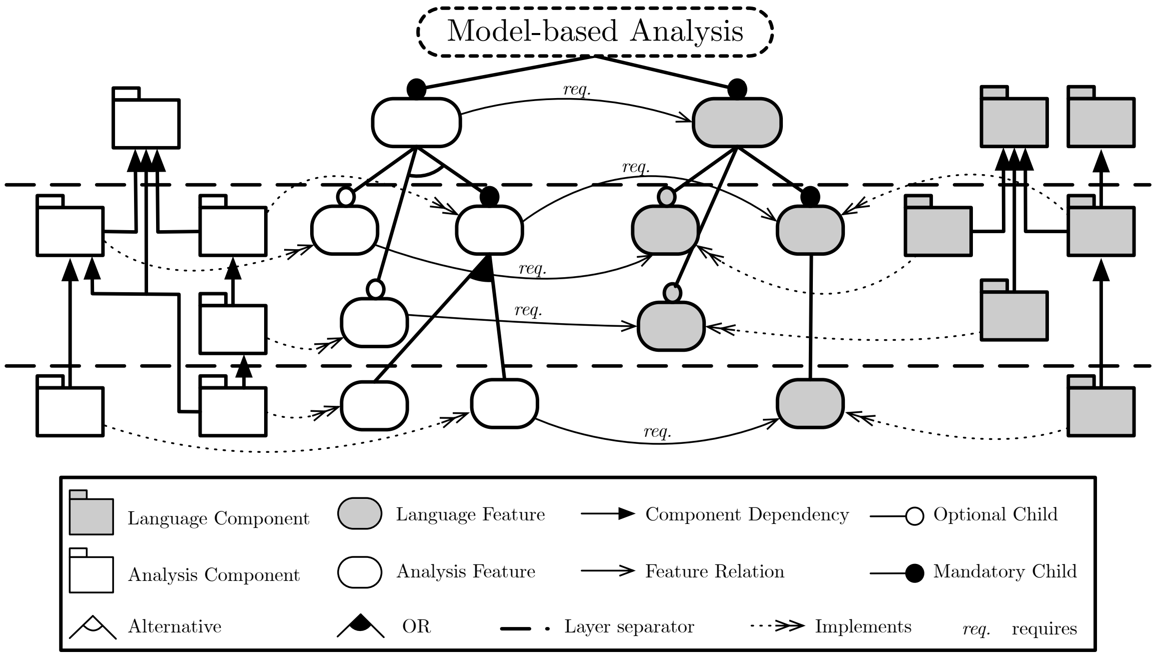


Figure 3.2.: Structure and Relations of the Reference Architecture for Model-based Analyses and DSMLs

are merely based on their feature dependencies (mandatory / optional) to more specialised features (cf. Section 3.3.1.4).

3.3.1.2 Language Feature and Analysis Feature

Before we go into detail regarding our modularisation concepts for model-based analyses, we present the language feature and language component definition by Heinrich et al. [HSR19]. Heinrich et al. modularise a DSML by separating it into distinct language features with a defined parent-child relation. The features represent the conceptual level of the DSML and the model-based analysis. A language feature is the expression of a concept without any detail regarding its technical realisation. In Figure 3.2, we depict the language features as grey rectangles with rounded corners. The language architect creates the language feature graph as they work on the conceptual level. The language architect can add atomic language features or composed language features to the language feature model. An atomic language feature models one abstraction, and a composed language feature is comprised of atomic and other composed language features. When a language feature represents the concerns of an analysis user, it is called a *user language feature*.

We map the modularisation concepts of Heinrich et al. to model-based analyses. In this subsection, we extend the notion of language features by the notion of analysis features. An analysis feature is a concept of the model-based analysis, e. g. an abstraction of a system's property to be analysed. In Figure 3.2, we depict the analysis features as white rectangles with rounded corners. The analysis architect creates the analysis feature graph. Regarding the creation of the feature graph, the analysis architect role is identical to the role of the language architect. In comparison to the language feature denotation, we define *atomic analysis features*, *composed analysis features*, and *user analysis features*. An atomic analysis feature represents a concept of a system that the model-based analysis can analyse, and a composed analysis feature comprises of atomic analysis features and composed analysis features. Composed analysis features are required to cluster analysis features that represent a broader concept and are always used together. A user analysis feature is a special type of the generic analysis feature, as it represents concepts regarding the analysis user. An analysis feature has a dependency on a language feature if it requires the language feature for the analysis.

For the sake of comprehensibility, if we reference language features and analysis features, respectively, we will call them *feature*. Dependencies between features in the same feature graph, either the language feature graph or the analysis feature graph, are called *feature dependencies*. The feature dependencies of the language feature graph determine the feature dependencies in the analysis feature graph. The dependencies of an atomic analysis feature are derived from the language features it analyses i. e. requires, and the dependencies of a composed analysis feature are derived from the dependencies of the analysis features it contains. We distinguish first- and second-class atomic features. An atomic feature contained in a root feature is a first-class feature, and a second-class feature is only contained transitively.

The two feature graphs are connected by the root feature that frames the model-based analysis. This root feature is depicted as a white rectangle with rounded corners and a dashed line as a contour. The model-based analysis represents all possible configurations that can be derived from the language feature and analysis feature graph.

3.3.1.3 Language Component and Analysis Component

Heinrich et al. [HSR19] introduced the concept of language components. They require that each language feature is implemented by a language component; thus each language component has a *implements* dependency on language features. A language component contains packages and classifiers. Language components can have dependencies on other language components and follow a set of restrictions (cf. metamodel module in [HSR19]). In Figure 3.2 language components are depicted as grey rectangles.

We use the concept of language components to derive our concept of analysis components. We require that each analysis feature is implemented by an analysis component; however, in contrast to language components, an *analysis component* is a container that contains classes, packages and especially analysis algorithms. An analysis component is realised in source code; it also can have dependencies on other analysis components. The analysis algorithms analyse language features; ergo, analysis features have requires dependencies on language features. The dependencies are of a transitive nature, as only analysis features have explicit dependencies on language features. In Figure 3.2 analysis components are depicted as white rectangles.

For the sake of comprehensibility, if we reference language components and analysis components, respectively, we will call them *component.* The analysis component developer implements analysis components. We derive the dependencies on other analysis components from the dependency structure of the analysis features. Thus, the dependency structure must conform to the dependency structure of the corresponding features. We do not require an identical dependency structure; the dependencies can conform directly or transitively. Furthermore, components are not allowed to form dependency cycles (cf. acyclic dependencies' principle [Mar03]), as when components of a cycle are used or changed, all other components in the cycle are affected. This hampers those components that can be reused individually.

3.3.1.4 Layering

In the context of our reference architecture for model-based analyses, we define a *layer* as a logical grouping of features and components. A feature and its corresponding components are located at the same layer, and features and components are only located at exactly one layer. The number of layers depends entirely on the DSML and the model-based analysis. Heinrich et al. recommend a maximum of four layers for DSMLs [HSR19]. We add an additional layer to the architecture of model-based analyses (cf. Section 3.3.2). According to the layered architecture pattern [RF20], features and components are placed in accordance with their dependencies: Feature-required and -parent and component dependencies must point in the same direction. To be more precise, the dependencies are only allowed to point to the same or a more basic layer (the more abstract, the more basic the layer). The more basic the layer is, the more on top it is located in our visual notation. More abstract classes can be seen at the top, while inheritance / generalisation lines point up.

3.3.1.5 Relation Between Modularisation Concepts

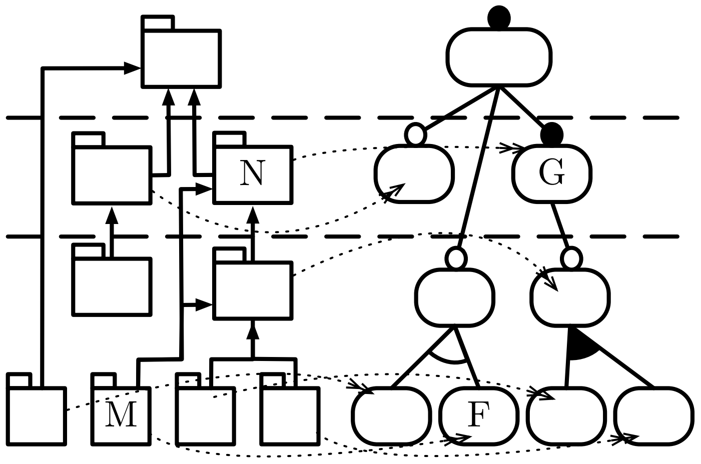


Figure 3.3.: Feature and Component Relation [HSR19]. Legend see Figure 3.2

Figure 3.3 depicts the relation of features and components. It represents no actual model-based analysis; however, it serves as an example of an arbitrary model-based analysis. The left graph represents the analysis component structure, and the right graph represents the analysis feature structure. Each analysis component and analysis feature is located at one layer; the notation of elements conforms to the legend shown in Figure 3.2.

Dependencies between analysis components are when classes, fields, or methods of a analysis component *M* depend on classes, fields or methods of another analysis component *N*. Dependencies can be, for example, the extension of a class, an aggregation of a class, or a plain reference.

Regarding component-oriented design, Reussner et al. [Reu+16] differentiate requires and provides roles. On the component level, we interpret these relations as dependencies. When deriving dependencies from analyses, dependencies manifest as connections between input and output [Hei+21a]. In addition to the structural input and output connection dependencies, we derive *behavioural* dependencies between analysis components. By adding events, analysis components can be coupled at a behavioural level [Koc+22]. We model the behavioural dependencies at the component level, so our approach does not require additional syntax. We focus on the presence and direction of the dependencies; thus, the precise nature of the dependencies between analysis components is irrelevant at the level of analysis components and their dependencies.

In the feature graph, the analysis features *F* and *G* are the corresponding analysis features of the analysis components *M* and *N*. If there is a path in the feature graph from *G* to *F*, then the dependency from *N* to *M* is considered *supported*. Each analysis component dependency must be supported; otherwise, the analysis features and analysis components are not individually reusable. For example, if the dependency of *M* to *N* is not supported, these components are unrelated semantically. This means that the analysis feature *G* must be used when the analysis feature *F* gets selected. If each analysis feature dependency is supported, the analysis feature and analysis component models are *conformal*.

3.3.2 Layers in Model-based Quality Analyses

The layering concept shown in Figure 3.2 does represent the general idea of having layers in model-based analyses. The analysis architect can choose an arbitrary number of layers with an arbitrary semantics. To illustrate an instantiation of our reference architecture for model-based analyses, we propose the layering system of our reference architecture for model-based quality analyses. The layering system is based on the modularisation concepts presented in Section 3.3.1. The first four layers of our architecture use the same four layers as in the reference architecture for quality metamodels. We add a fifth layer; as a result, our reference architecture for model-based quality analysis corresponds to a five-layered architecture. For other types of model-based analyses, the number and purpose of the layers can differ; the layers are derived from the associated DSML.

Within our reference architecture for model-based analyses, we define a layer as a logical grouping of features and components. A layer consists of features and their corresponding components, and each feature and component is assigned to one layer only. The layering restricts feature-required and -parent dependencies and component dependencies; they must all point in the same direction. A layer represents an abstract grouping; a layer can contain language features, language components, analysis features, and analysis components equally. The layering only affects the dependency direction and grouping of classes; the layering does not affect the behaviour of the analysis. Therefore, we can reuse the four layers of the reference architecture for quality metamodels.

Furthermore, for each language feature, we have a corresponding analysis feature. Features represent the conceptual level of the DSML and the model-based analysis. They are expressed as a concept without technical detail. Analysis features have a dependency on language features if they require them for analysis. The feature dependencies of the language feature graph determine feature dependencies in the analysis feature graph. Due to the conceptual nature of a feature, we can have the same structure of features in the model-based analysis as in the DSML.

In addition to the layers paradigm (3.3.2.1), domain (3.3.2.2), quality (3.3.2.3), and analysis (3.3.2.4) which are identical to the layers in the reference architecture for DSMLs, we propose the experiment (3.3.2.5) as fifth layer of our reference architecture for model-based quality analyses. Figure 3.4 depicts the five layers for model-based quality analyses.

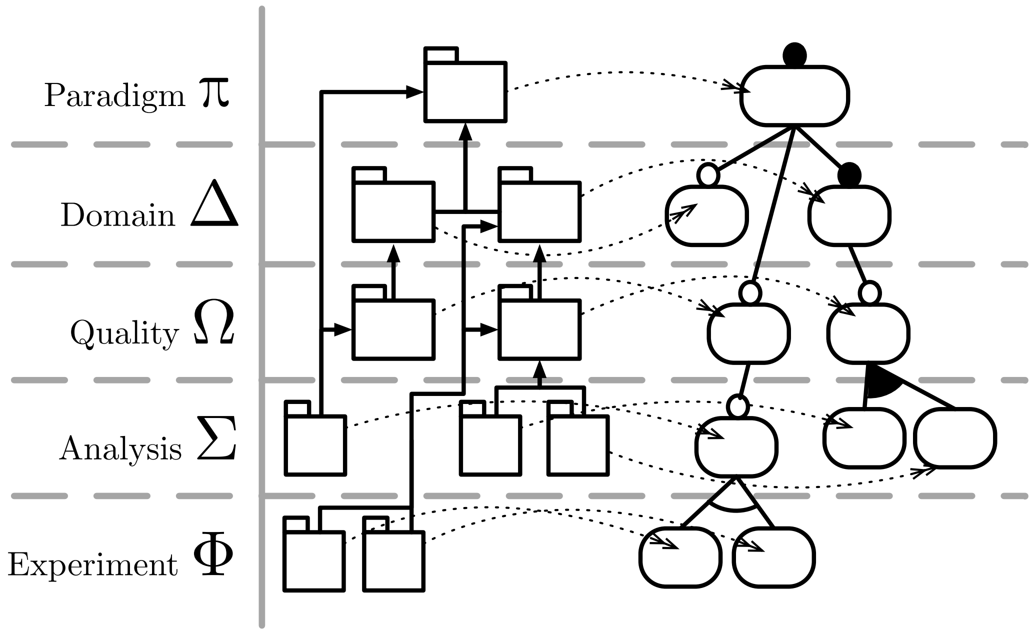


Figure 3.4.: Layering Structure for the Reference Architecture for Model-based Analyses. Legend see Figure 3.2

3.3.2.1 Paradigm Layer

The π paradigm layer is the most abstract layer in our reference architecture for model-based quality analyses. The layer contains the building blocks of the model-based analysis, and it contains defining analysis features for reoccurring patterns of structure and behaviour. It does not contain any dynamic semantics; for example, in a performance analysis, state or interfaces are specified in π without specifying their usage. Features on the π layer can reference the features on the π layer on the DSML architecture. These references are used for the analysis of the π features. In addition to the analysis of the π features, the π layer also contains basic features that are only relevant for the model-based analysis. Due to the missing dynamic semantics, the π layer is not intended to be used without another layer.

Dependencies on other layers are not allowed, as they would contradict the layering principle and dependencies on more abstract layers are not possible, as π represents the most abstract layer. The π layer can only have dependencies on features on the same layer. It contains mostly abstract classes or interfaces, as the more concrete layers are intended to add the dynamic semantics.

The analysis components for the π layer that the analysis developer develops must be so generic that the analysis developer can reuse analysis components for different domains. For example, model-based analyses can share features like the specification of users and states. They are so generic that a user merely has a name, and a state has a field indicating that the state is active. These features are so generic that one analysis uses these features to analyse the behaviour of a user of a software system, and the other analysis uses these features to analyse users in a hardware environment, for example, a production plant. However, the analysis architect specifies the specialisation of a feature on a more specialised layer, for instance, the domain layer Δ.

3.3.2.2 Domain Layer

The domain layer Δ is the first extension layer. It is placed after the π layer. The Δ layer extends the π layer by extending its abstract components. How a component is extended depends on the underlying programming language. In Java for example, the analysis developer extends the abstract classes and implements the interfaces of the π layer. Thus, the analysis developer adds structure and behaviour to the model-based analysis.

The added structure and behaviour introduce domain-specific semantics to the model-based analysis. For example, suppose the analysis developer wants to add the two domains of software systems and hardware systems (e. g., production plants). In that case, they can extend the features introduced in Section 3.3.2.1. The analysis developer adds a software user, a software state, a hardware user, and a hardware-state features. This way, the model-based analysis is able to analyse software and hardware systems.

The benefit of separating the domain-specific features into hardware- and software-specific features is that if the analysis developer does not need to analyse hardware systems, the hardware-specific features can be ignored without affecting the software-specific features. However, the domain layer is explicitly not restricted to one domain.

Although the Δ layer can contain atomic features, the analysis architect must consider whether they should place such a feature at the π layer. Alternatively, they could extract the fundamentals and place them at the π layer. The remainder remains at the Δ layer.

An model-based analysis can consist of only the π and Δ layer. The analysis user runs it to reason about structural and behavioural properties. However, the analysis developer can extend the analysis by adding features for modelling or analysing quality properties. We recommend having π and Δ in each model-based analysis. This allows the analysis developer to reuse general features of a model-based analysis (π) and domain-specific features (Δ) for different kinds of specialisations a model-based analysis can have. The next layer illustrates the analysis of quality attributes, and it is such a specialisation a model-based analysis can have.

3.3.2.3 Quality Layer

The quality layer Ω extends the model-based analysis by analysing quality properties. Quality properties are, for example, the performance or reliability of a system. The security and safety of a system are also quality properties. The Ω layer contains the main reason why a model-based analysis exists, as it is crucial to determine the quality of a system before spending resources implementing such a system. Thus, the analysis developer adds the analysis that determines the quality of a modelled system to the Ω layer.

The added quality analysis features extend the model-based analysis by quality-specific semantics. For example, if the analysis developer wants the user to be able to analyse the performance and reliability of a system, they can utilise the domain-specific features. Before implementing the analysis components, the analysis developer must determine whether the quality analysis can be applied to each domain or whether the quality analysis is specific to a certain domain. For example, considering the performance analysis of software and hardware systems, the concept of queues can be applied to both domains. Thus, the analysis developer extracts a common feature that the performance analysis of both domains can use, and then they can develop the individual components. They place the individual components also on the Δ layer.

The benefit of separating the quality-specific features is that if the analysis developer only needs to analyse a certain quality attribute, the remaining quality analysis features can be ignored. This flexibility allows the Ω layer to be explicitly not restricted to one quality property.

3.3.2.4 Analysis Layer

The analysis layer Σ extends the model-based analysis by analysis features that are relevant for a concrete analysis execution. Part of the Σ layer is the specification of configuration data of the model-based analysis. Additionally, this layer holds the information of the runtime state and the specification of model-based analysis output. For example, if a reference to a model attribute is needed for a reachability analysis, the value range and the reference to the attribute are located on Σ.

The features on the Σ layer utilise the features of the Ω layer. For example, the performance and reliability analysis could be specified separately; however, if the analysis developer needs to combine both features to perform a performability analysis. The analysis developer defines a feature that interprets the runtime state and the output of these two features to determine the performability of the analysed system.

The features located at the Σ layer represent an experiment that the analysis could run. Such an experiment is represented on a structural level. The experiment runs are located at the Φ layer.

3.3.2.5 Experiment Layer

The experiment layer Φ extends the model-based analysis by analysis features that represent the experiment runs. If, for example, the model-based analysis provides setting seed values for an experiment, the analysis developer places the seed value feature on the Φ layer. More examples of features at the Φ layer are setting the initial states, termination conditions or the sequence of actions; other features of the analysis are invoked. Especially the sequence of actions is relevant when setting up more than one experiment runs. The Φ layer allows creating of reproducible experiments that contain a sequence of analysis runs.

3.3.3 Refactoring Operations for Modularising Model-based Analyses

In this section, we present the refactoring operations we provide for analysis developers. The refactorings were first published in our technical report [KHR22a]. These refactoring operations are meant to apply our reference architecture to an already existing model-based analysis. The model-based analysis and its corresponding DSML must fulfil two preconditions. The first and most important precondition is that the DSML already conforms to the reference architecture for metamodels [HSR19]. This means that the DSML is already separated into layers, the language features and language components are cycle free, and the dependencies point only to a more generic layer. The second precondition is that the model-based analysis uses the DSML or instances of the DSML to reason about a system.

We divide the refactoring operations into refactorings on the analysis class level and refactorings on the analysis component level. The analysis class level refactorings consist of operations that split or merge classes, fix dependency cycles and change the dependency direction of classes. The analysis component level refactorings consist of operations that split (horizontally or vertically) or merge components, and that extract features and components, if needed.

The refactorings we present are based on the DSML refactorings of Heinrich et al. [HSR19] and OOP refactorings [Fow18]. The DSML structure (i. e., number of layers, features and dependencies between features) serves as a template for the model-based analysis design (see Figure 3.2). We apply the refactorings presented in this section to transform the monolithic model-based analysis into the modular structure of our reference architecture.

The figures in this section heavily rely on the legend depicted in Figure 3.5. Rectangles that resemble a folder symbol represent components, and regular rectangles represent classes. In order to distinguish between language and analysis elements, the language elements are coloured grey, and the analysis elements are coloured white. If an element represents both, it is filled half grey and half white.

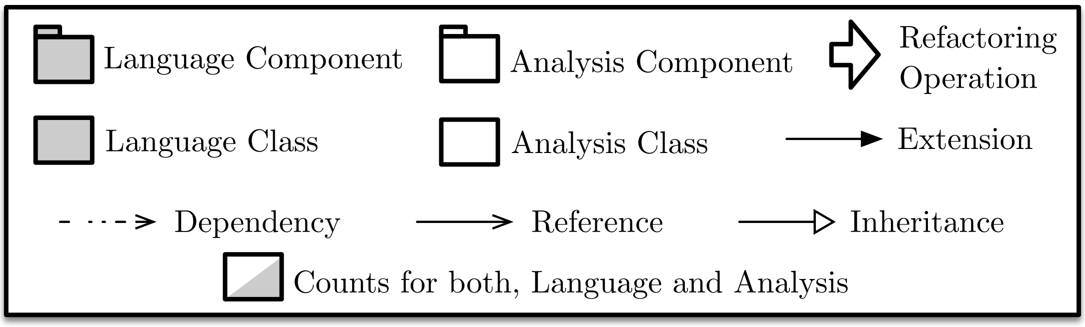


Figure 3.5.: Legend for the Notational Elements Used to Depict the Refactoring Operations

3.3.3.1 Analysis Class Refactorings

The analysis class refactorings are the foundation to modularise an existing, monolithic model-based analysis. These refactorings provide a toolset for the analysis developer to adapt the structure of a model-based analysis to the structure of its corresponding DSML. It is not always necessary for the analysis developer to use all refactorings to accomplish a modularised model-based analysis. The refactorings are, per definition, not intended to change the behaviour of the refactored system [Fow18]. We distinguish four refactoring operations on the class level: *class split*, *class merge*, *breaking of dependency cycles*, and *dependency inversion.*

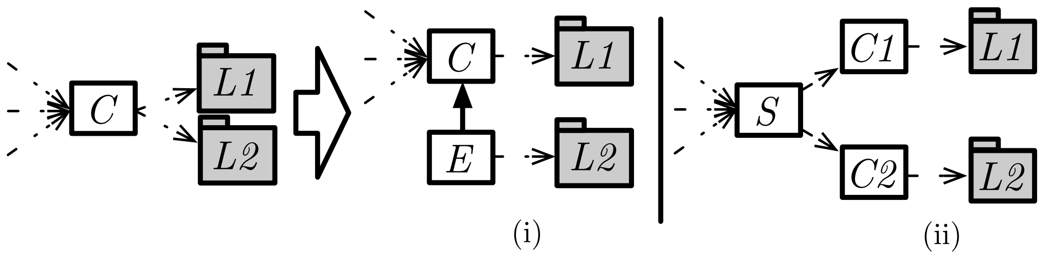


Figure 3.6.: Class Split [KHR22a]

Class Split Splitting a class is a common refactoring operation in software development. The class split refactoring operation is where class elements, such as attributes and methods, are extracted and transferred into one or more new classes [Fow18]. In language- and object-oriented design, the goal of the class split refactoring is to separate different concerns into separate classes to improve the comprehensibility of individual classes. The refactoring operation class split is shown in Figure 3.6. We require to split a class when it has dependencies on different language components. The analysis class C has dependencies on the two language components L1 and L2. Our approach assumes that the underlying language is already modularised and partitioned. Therefore, if possible, a class should be split when it has more than one language component as a dependency. Another problem is that after the refactoring, we must be able to distinguish whether the language components are placed on one layer or distributed over several layers in the architecture. When a class is split according to the structure of the language components, the refactored classes must be distributed according to their dependencies in the same architecture layers. The analysis developer can choose from two class splits. The first class split is shown in Figure 3.6 (i), the analysis class C, that needs to be split up, is extended by a new analysis class E. Also, E takes properties of C; for this, the required properties are factored out from C to E. Incoming dependencies remain on C.

From a purely syntactical view, attributes, methods, references, containments, and inheritance can be factored out on the class level without complications. In the case of model-based analyses, it is often impossible to split a class according to the language structure. An analysis feature might need different language features to perform an analysis. However, the structure of the DSML requires that the analysis feature has no dependency on the language feature i. e., has no knowledge about the language feature. However, given the structure of the language, it is not always possible to separate a class as demanded by the reference architecture of the metamodel. This can occur if, for example, language components from different layers are used with dependencies on each other. Besides the elements that can be cleanly separated from a class and the components that do not have dependencies on the language component, we propose encapsulating the inseparable elements in a class and then placing them in the most specific layer. As it is shown in Figure 3.6 (ii), a specialisation analysis class S is introduced, which uses the second split class refactoring as shown in Figure 3.6 (i), but the incoming dependencies are shifted to S. In the worst-case scenario, the classes cannot be fully split so that S holds dependencies of L1 and L2.

Class Merge Like the class split, the class merge is also a common refactoring operation that originates in object-oriented design [Fow18]. A class merge transfers attributes and methods of a class to another class. In the context of our reference architecture for model-based analyses, a class merge is required when two classes have dependencies on the same language component.

The class merge is intended to combine concerns that are distributed across different classes. The class merge refactoring is shown in Figure 3.7. We differentiate between a clean class merge Figure 3.7 (i) and a class merge with a rest Figure 3.7 (ii). When a

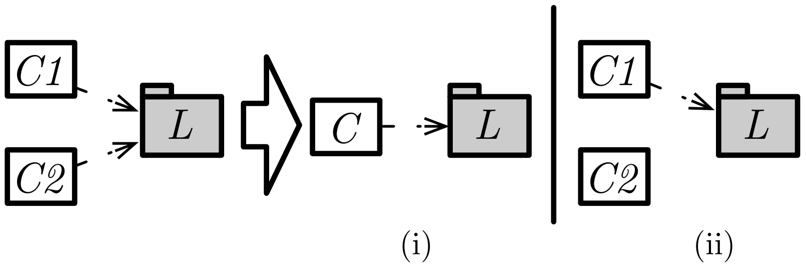


Figure 3.7.: Class Merge [KHR22a]

language component of the DSML has scattered dependencies, i. e., types of a language component are referenced by multiple classes and levels, the class merge can be used to merge these dependencies. A class merge is performed by extracting attributes and methods of one class and then inserting them into another class. The result is an extended target class with attributes and behaviour of the source class. Figure 3.7 (i) shows the class merge, both classes can be combined to one class when both classes depend on only language classes of the same language component. C1 and C2 have dependencies on the same language component L; the merge combines C1 and C2 into one new class, C which, as a result, shares the dependencies on the desired language component *L*.

When the classes that should be merged also depend on language classes from different language components, only the attributes and methods are allowed to be moved when they depend on language classes of the same language component L. Figure 3.7 (i) shows the class merge with a rest. The attributes and methods of C2 that had dependencies on the language component L were moved to C1. The remaining attributes and methods of C2 that have dependencies on other language components were not moved. If the classes C1 and C2 were merged like shown in Figure 3.7 (i), we would have produced a scenario that requires a class split as shown in Figure 3.6.

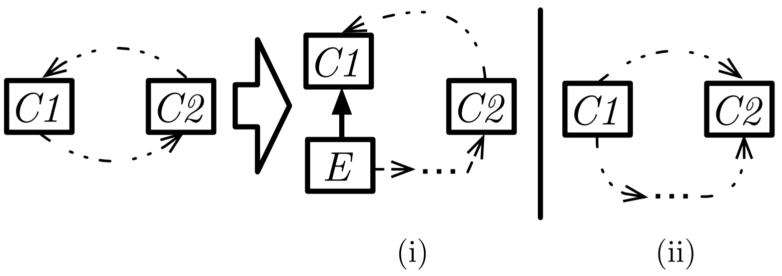


Figure 3.8.: Breaking Dependency Cycles between Classes of the Model-based Analysis [KHR22a]

Breaking Dependency Cycles As stated in Section 3.3.1.1, a model-based analysis that is modelled according to our reference architecture must be cycle free. If the bad smell *cyclic dependencies*, known from object-oriented design, occurs, the following refactoring operations show how developers can break such cycles. Dependency cycles prevent easy extension of software systems [Par79], and according to Fowler [Fow01], dependency

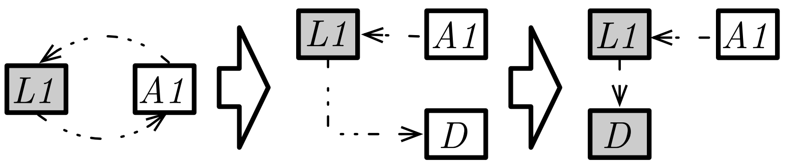


Figure 3.9.: Breaking Dependency Cycles between Classes of the Model-based Analysis and its associated DSML

cycles make a system harder to understand, thus, harder to maintain. How to refactor dependency cycles between classes of the model-based analysis is shown in Figure 3.8. We assume that the DSML does not contain any dependency cycles [HSR19], and if our reference architecture for model-based analyses is applied, the model-based analysis should also not contain any dependency cycles. Due to the co-dependency of model-based analyses and DSMLs, we distinguish two types of dependency cycles. The first type occurs between classes of the model-based analysis (see Figure 3.8). The second type occurs between classes of the model-based analysis and the DSML (see Figure 3.9).

First, we explain how dependency cycles between analysis classes can be broken up. As in language- and object-oriented design, we distinguish two refactoring operations to break dependency cycles. On the one hand, the previously presented class split can be used; on the other hand, dependency inversion is also a valid option to break dependency cycles. The initial state is that `C1` and `C2` depend on each other. The outgoing dependency of `C1` is factored out into `E` if they contributed to the cycle. As a result, `C1` is split, and `C1` has no dependency on `E`; thus, the cycle no longer exists see Figure 3.8 (i). The dependency inversion is described in the following section. Dependency inversion is one technique to tackle dependency cycles, as exemplified in Figure 3.8 (ii).

The language must have no dependencies on the analysis; otherwise, changes in the analysis can trigger changes in the language. One DSML can be used by multiple model-based analyses; as a result, a change in one model-based analysis could have a cascading change effect on all remaining model-based analyses that are associated with the DSML. Fixing a dependency cycle between language and analysis classes is depicted in Figure 3.9. To break up dependency cycles between language (*L1*) and analysis classes (*A1*), the analysis developer and the language developer must split the analysis class *A1* to separate the elements the language class depends on (*D*) from the remainder of the class (*A1*). If possible, the elements of *D* can be added to a language feature, alternatively, *D* becomes part of the generated language component.

Dependency Inversion According to Martin [Mar03], abstractions (`A`) must not depend on specifics (`S`); instead, specifics must depend on abstractions. This statement is known as the dependency inversion principle. It originated in the object-oriented design and was later adapted by Heinrich et al. to suit the design of DSMLs [HSR19]. To tackle the problem when dependencies violate the constraints of our reference architecture, we present a refactoring solution that transfers the reference architecture for DSMLs to model-based analyses. The general refactoring for dependency inversion is illustrated in Figure 3.10,

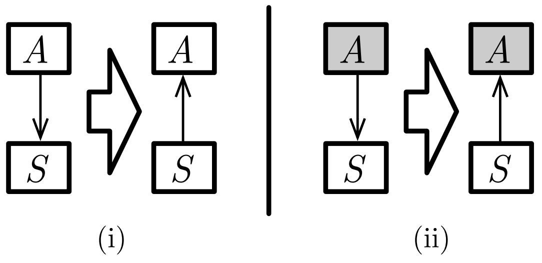


Figure 3.10.: Dependency Inversion [KHR22a]

dependency inversion by inheritance is shown in Figure 3.11, dependency inversion by reference is shown in Figure 3.12, and dependency inversion by bidirectional reference and containment is shown in Figure 3.13.

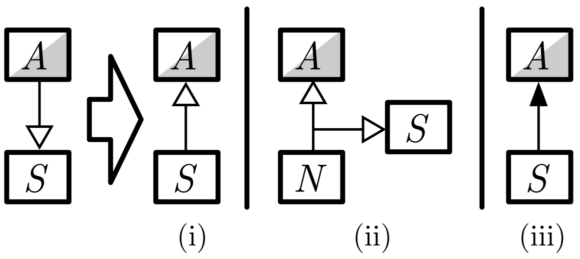


Figure 3.11.: Dependency Inversion – Inheritance [KHR22a]

In Figure 3.11, we present the dependency inversion by inheritance refactoring. If `S` is a specialisation of `A`, the inheritance is in the wrong direction and must be inverted; especially when *A* is a language class. Occurrences of `S` and `A` in the analysis code must be switched Figure 3.11 (i). Inverting the inheritance means that concepts that are relevant for *A* are moved from *S* to *A*. The inheritance can be removed entirely if a feature is implemented by both `A` and `S`, as both implement the same functionality. Alternatively, a subclass can be introduced to invert the inheritance. The new subclass (`N`) of `A` and `S` is introduced Figure 3.11 (ii). Dependencies must be redirected to either `A`, `S`, or `N`; for this, incoming dependencies of `A` and `S` are used. If `S` is not a specialisation of `A` but if it is part of a first-class analysis component, the inheritance is removed and replaced by a reference from `S` to `A` Figure 3.11 (iii). All these refactorings have in common that when *A* is a language feature, the language model must be changed due to the problematic dependency from the DSML to one analysis.

In Figure 3.12, we present an approach to invert a dependency by refactoring a reference. A reference can be inverted using a class split, this type is presented in Figure 3.12 (i). A new class `E` is introduced when inverting the reference. `E` replaces the reference from `A` to `S`. This option should be chosen if `S` is part of a first-class analysis component (i. e., an instance of `S` does not depend on an instance of `A`). This refactoring is applicable for refactoring language and analysis classes, as it fixes the dependency direction for both. If `S` is part of a second-class analysis component, which extends the functionality of `A` but is no further extended, a simple extends relation, in the case of an object-oriented

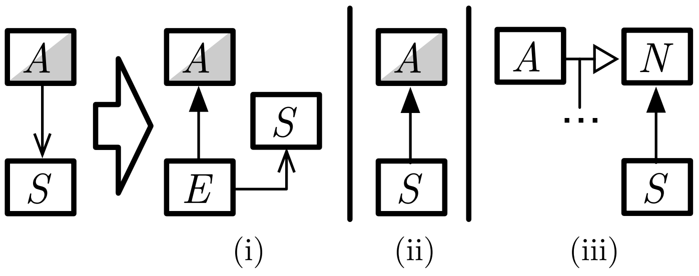


Figure 3.12.: Dependency Inversion – Reference [KHR22a]

language inheritance, can be implemented Figure 3.12 (ii). However, if S needs to be further specialised, introducing a common superclass N is advised Figure 3.12 (iii). The third refactoring is only applicable for analysis classes, as *A* would still depend on a class of a model-based analysis. When *A* and *N* are language classes, we recommend consulting the refactoring operations for DSMLs proposed by Heinrich et al. [HSR19].

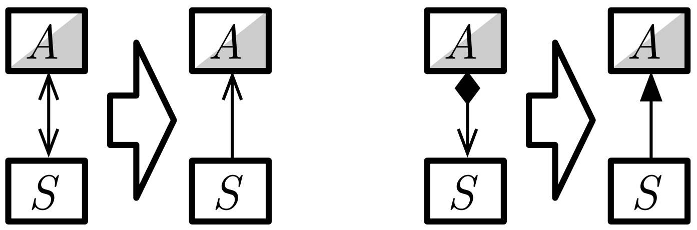


Figure 3.13.: Dependency Inversion – Bidirectional Reference and Containment [KHR22a]

A bidirectional reference between two classes A and S is the simplest form of a dependency cycle (see Figure 3.8, and a special case of Figure 3.12). The bidirectional nature implies a redundant reference, which is usually detected by an IDE like IntelliJ or Eclipse. To remove one redundant reference, the parts in *S* that are referenced by *A* can be moved from *S* to *A*, see Figure 3.13 (i). Containment references can be removed by extracting an extension class S representing the desired feature. That way, features can be strictly separated, see Figure 3.13 (ii). In both cases, if a language class depends on an analysis class, the developer must consider the possible cascading changes that affect all associated model-based analyses when modifying the DSML.

3.3.3.2 Analysis Component Refactorings

In addition to our refactoring operations on the class level, we present refactoring operations that target analysis components instead of classes. What analysis developers need for the class-level refactorings can be transferred to the component level. Instead of performing single, small refactorings on the class level, when an analysis developer has to adjust whole components, the refactoring can get tedious, depending on the number of classes in a component. Therefore, we present five additional refactoring operations on the component level: *horizontal split*, *vertical split*, *merge*, *extension extraction*, and *feature support extraction.*

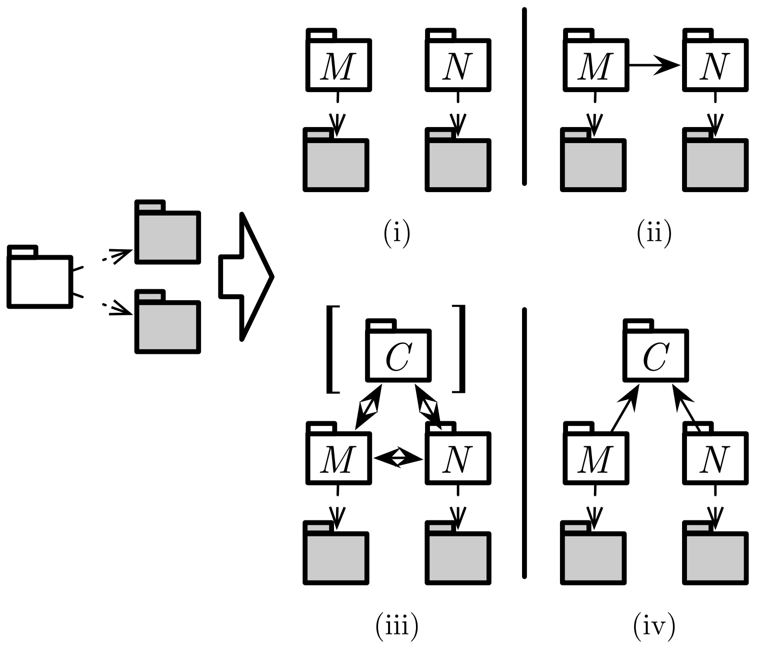


Figure 3.14.: Horizontal Split [KHR22a]

Horizontal Split An analysis component must be split horizontally by the analysis architect if parts of an analysis component can be used independently of each other (cf. Single Responsibility Principle [Mar+03]). An initial indicator to split an analysis component is when an analysis component has dependencies on multiple language components. Figure 3.14 (i) shows the potential best-case outcome; the components are unrelated. In Figure 3.14 (ii), one of the analysis components is dependent on the other. In Figure 3.14 (iii), the potential worst case is shown. The new components M and N may still share the original component's common part C. The parenthesis around C indicates that this component does not necessarily exist. All the analysis components may be mutually dependent. The dependencies of M and N must be adjusted according to the dependencies of the analysis feature they implement. The adjustment of the dependencies must be made by the analysis architect and the analysis component developer in Figure 3.14 (iv) the components M and N dependent on a common component C. The common analysis component C also indicates an additional feature, which is an addition to the analysis feature graph.

Vertical Split The vertical split is illustrated in Figure 3.15. The analysis architect performs this refactoring if the layer to which an analysis component could be assigned is unclear. An indicator to vertically split an analysis component is when said component has dependencies on language components on different layers. A horizontal split is recommended if the language components are on the same layer. However, if the language

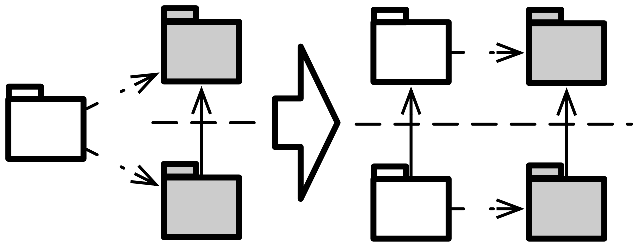


Figure 3.15.: Vertical Split [KHR22a]

components are located on different layers, the analysis architect divides the analysis component so that each resulting analysis component can be assigned to one layer. The analysis component developer must split classes if necessary. After the refactoring, each resulting analysis component is assigned to its layer by the analysis architect. The resulting architecture could have dependencies that point from an abstract to a more specific layer. If this is the case, the analysis component developer must perform dependency inversion.

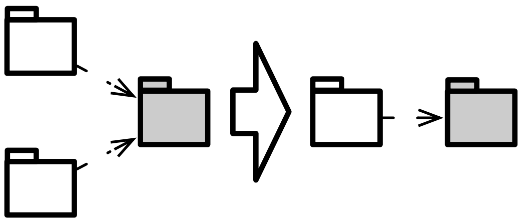


Figure 3.16.: Merge [KHR22a]

Merge Figure 3.16 shows the component merge refactoring. A merge refactoring could be advisable when more than one analysis component depends on the same language component and if the analysis components are located on the same layer. The analysis developer checks whether the dependent language features have a mandatory feature relation or if the analysis components form a dependency cycle. If one of these constructs can be found in the architecture, the analysis architect should consider merging those features and their analysis components. There may be various dependency constellations between the merged analysis components, like one-directional or bidirectional. Merging analysis components could also be advisable if they have no dependencies but classes of an analysis component that are abstract function as ubiquitous superclasses.

Extension Extraction The analysis architect uses the extension extraction refactoring if an analysis component contains content that does not belong to the feature it implements. An indicator for refactoring is if the optional content cannot be used independently. The extension extraction refactoring is depicted in Figure 3.17 – the analysis architect factors out the optional content of `M` into a new analysis component `C`. The remainder of `M` is denoted as `M'`. The classes of component `M` must be split if they should be located in `N` but contain optional properties that belong to `M'`. The analysis component developer also does this refactoring. If a class has dependencies on multiple language components, which cannot

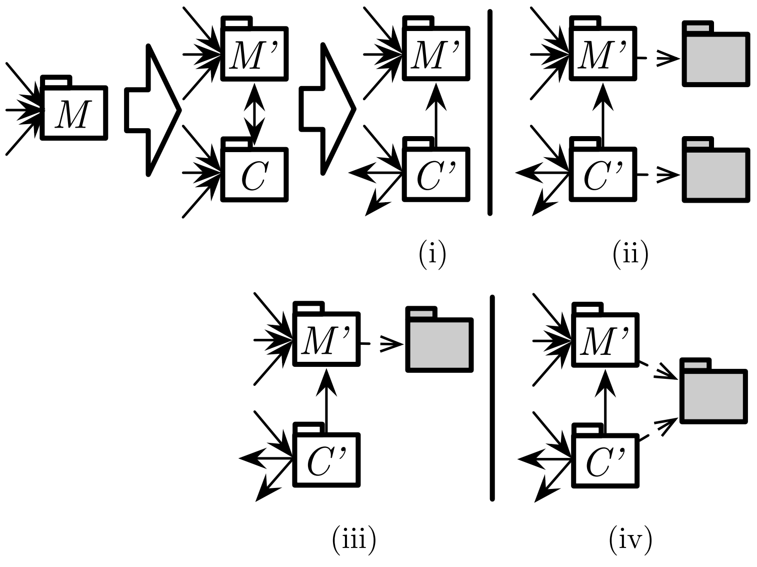


Figure 3.17.: Extension Extraction [KHR22a]

be factored out, the class must be put in the most specialised analysis component. The following step reverses all dependencies from elements of `M'` to `C`. Incoming dependencies on `C` must be considered for dependency inversion. The result of the dependency inversion is shown as outgoing dependencies of `C'`. The refactoring can be performed if the analysis components have no dependencies on any language component. However, if `M` has dependencies on multiple language components, each dependency should be refactored into one dedicated analysis component (see Figure 3.17 (ii)). If the optional content of `N'` represents a dedicated analysis component that has no representation in the language, `N'` must be refactored into a dedicated analysis component with no dependencies on the language (see Figure 3.17 (iii)). If it is reasonable to separate optional content, `C'` but the dependencies on one language component cannot be separated Figure 3.17 (iv) must be applied.

3.4 Composition of Model-based Analyses

In this section, we present composition operators that allow the analysis architect to compose model-based analysis features and components. The composition operators were first published in our technical report [KHR22a]. We developed the composition operators presented in this section to work with our reference architecture for model-based analyses. As a result, a model-based analysis must follow our reference architecture so that the composition operators in this section can be applied to it. After modularising an existing model-based analysis with the refactoring operations presented in Section 3.3.3, the model-based analysis is separated into features. These features can be used to create or extend other model-based analyses by composing them with each other. The concepts presented in this chapter are heavily inspired by feature composition in general and the feature composition by Apel et al. [Ape+08] in particular (cf. Section 2.3, Feature Composition).

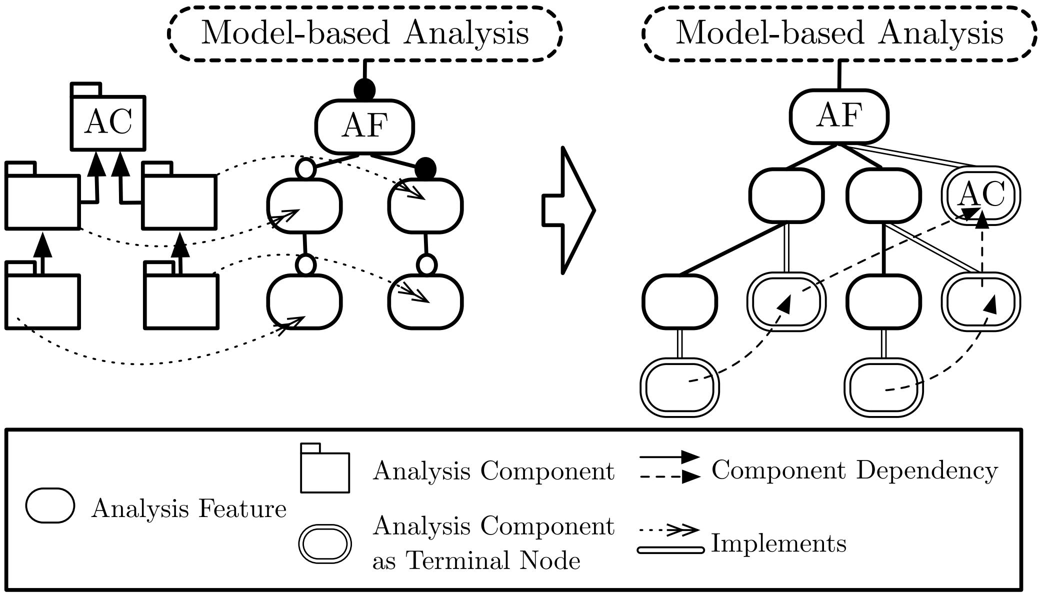


Figure 3.18.: Transformation of Our Modularisation Concept to a Feature Structure Tree [KHR22a]

The composition operators that we will present operate on the structure of both the analysis feature and analysis component, as depicted in Figure 3.18. Besides the introduction sum of Apel et al. [Ape+08], we define another operation, the *Introduction Diff*. The introduction diff is needed to remove nodes from an FST. The introduction diff operator $\ominus$ is also an operation over the set of I, which removes atomic introductions; the result is again a non-atomic introduction or an empty FST (ξ). Introduction diff $\ominus$ over the set of I of introductions forms a non-commutative idempotent monoid $(I, \ominus, \xi)$. The introduction diff is non-associative and non-commutative. By means of non-associativity, without the order of operations, the calculation is performed from left to right:

$$(f \ominus g) \ominus h \neq f \ominus (g \ominus h)$$

For $j = \xi$ follows $i \ \ominus \ j \ = \ i$.

A modification specifies how a feature affects another feature during composition and how features are composed. Modifications consist of queries and changes. The queries are for fetching the affected components to apply a set of changes to achieve the desired composition. Their associated analysis techniques are composed by selecting sub-analysis from the feature model. A modification m has a query q, which selects a set of atomic introductions, and a change c, which will be applied to the query result $m \ = \ (q, c)$. The modification application operator $\odot$ is applied to the set M of modifications and the set I of introductions. A modification can return either the input introduction or a changed introduction. The modification is associative and non-commutative. The identity is defined as follows:

$\zeta \odot m \ = \ m \odot \zeta - \zeta$ is a class of empty modifications.

Modifications are not idempotent.

We identified six modification operations to aid the development of model-based analysis. Each operation utilises the previously introduced composition operators. We distinguish between two atomic modification operations (*add* and *delete*) and four composite modification operations (*move*, *replace*, *merge*, and *split*). The analysis developer proposes possible change operations, which the analysis architect can then utilise.

Add a node to the graph: The *add operation* $m_{add} \odot (f_1 \oplus f_2)$ combines two features, f_1 and f_2. The modification operation m_{add} fetches the added features and their parent nodes. If the added features are components, the modification operation determines which extension dependency has to be used. The modification can be detailed, e. g. which extension mechanism to use on which class, or it can be generic. Additionally, the modification does not need to be automated; manual steps can also be part of such a modification.

Delete a node of the graph: The *delete operation* $m_{delete} \odot (f_1 \ominus f_2)$ removes feature f_2 from feature f_1. The m_{delete} fetches all child features of f_2 and removes them from the graph. Also, the m_{delete} modification operation removes dependencies concerning the deleted features. This ensures that the graph no longer contains features representing a specialisation of a feature that no longer exists. Furthermore, if extends dependencies are no longer viable, components will be modified.

Move a node of the graph: The *move operation* $m_{move} \odot ((f_1 \ominus f_2) \oplus f_2)$ moves the existing feature f_2. The *move operation* is a composite modification operation that first deletes the features f_2 that must be moved. Then, the feature f_2 is added to the new destination in the graph. The modification operation m_{move} adapts the dependencies of the features and components affected by the move operation.

Replace a node of the graph: The *replace operation* $m_{replace} \odot ((f_1 \ominus f_2) \oplus f_3)$ replaces the feature f_2 with feature f_3. The *replace operation* is also a composite modification operation, as the *move operation.* After removing the feature f_2, the feature f_3 replaces the feature f_2 by adding it to the tree. The modification operation $m_{replace}$ adapts the dependencies in the component nodes in the graph. If the new node replaces a feature without replacing the dependent component, $m_{replace}$ must ensure the compatibility between the feature and component node.

Merge two nodes of the graph: The *merge operation* $m_{merge} \odot (f_1 \ominus f_2)$ merges the features f_1 and f_2. The *merge operation* also is a composite modification operation; it utilises the *delete operation* to remove the feature f_2 from the graph. The m_{merge} modification operation moves the internals of feature f_2 to feature f_1, and all dependencies from the feature f_2 are moved to the feature f_1.

Split a node: The *split operation* $m_{split} \odot (f_1 \oplus f_2)$ splits feature f_1 in the features f_1 and f_2. The *split operation* also is a composite modification operation; it adds a new feature f_2 to the graph that inherits internals and dependencies of the feature f_1. The m_{split} modification operation moves this internals of feature f_1 to the new feature f_2, with dependencies getting also modified.

3.5 Application Process

In this section, we present processes to apply our reference architecture for model-based analyses in different scenarios. We differentiate three scenarios in the development of an model-based analysis where an analysis developer can apply our reference architecture where we consider the state of the DSML and the state of the model-based analysis together. The first scenario is the modularisation of an already existing model-based analysis (i). The second scenario is the development of an model-based analysis from scratch (ii). The third and last scenario is the extension of an model-based analysis (iii).

The steps differ regarding the development stage of the DSML and the model-based analysis. In the first scenario (i), the analysis developers already have access to implemented analysis components that analyse the DSML. We assume that the DSML is already modularised according to the reference architecture for metamodels; ergo, we assume that the feature model already exists. Thus, the analysis developers must modularise the analysis components according to the feature model. The refactoring operations for modularising a model-based analysis are tailored to object-oriented software; therefore, the process for the first scenario is applicable as long as the DSML follows the reference architecture for DSMLs and the model-based analysis is implemented in an object-oriented programming language.

In the second and third scenarios, the analysis developer creates the DSML and analysis feature model before implementing (ii) or extending (iii) the analysis components. The composition operators for implementing dedicated exchangeable and extendable analysis components rely on the inheritance principle of object-oriented software. As a result, these processes also require that the model-based analysis will be (scenario (ii)) or is (scenario (iii)) implemented in an object-oriented language. These scenarios extend the application processes for DSML modularisation and composition [HSR19].

Although the processes restrict the analysis developers in their freedom on how to design, implement, and extend model-based analyses, the processes also have their benefits. The processes provide a structure for the analysis developers that they can follow, which unifies the design, development, and extension process [HSR19]. Due to the processes, the evolvability and reusability of the model-based analysis are improved.

This section is structured as follows: First, we present the process to modularise an existing model-based analysis in Section 3.5.1. Second, we present the process to develop a model-based analysis from scratch in Section 3.5.2. Finally, we present the process to extend a model-based analysis in Section 3.5.3.

3.5.1 Modularisation of an Existing Model-based Analysis

The following applies to all processes; the analysis requires a corresponding DSML modularised according to the reference architecture for metamodels. We also assume that a feature model of the language exists. We start with the modularisation of an already existing, monolithic, model-based analysis. The analysis developers' goal is to adapt the monolithic model-based analysis to our reference architecture. The steps in these processes are not performed purely sequentially; therefore, we will discuss where it is advisable to repeat certain steps.

To create these processes, we modularised the software performance analysis SimuLizar and extracted our findings. We published a technical report in [KHR22a] for more details regarding the refactoring. SimuLizar also serves as a case study in this thesis (cf. Section 6.2). The analysis SimuLizar is not the only model-based analysis to which we applied our reference architecture; we selected case studies from different domains to have a variety of model-based analyses (cf. Section 6.1). Thus, we can state that our process is applicable to a variety of object-oriented model-based analyses.

To determine the modularisation potential, we define the following criteria: The first question an analysis developer has to answer is: Does the model-based analysis resemble the structure of the corresponding DSML? They can answer it by identifying features of the DSML in the model-based analysis, and the dependencies of these features can be identical (i. e., direction, type, and connection of features). The analysis developer requires information about the features so that they can create the feature model of the model-based analysis. Ideally, they can place the features already on a layer in the reference architecture. The analysis developer can utilise our refactoring operation of Section 3.3.3 and the composition operators of Section 3.4 to fix the dependency structure, breaking up cycles and placing the analysis features and analysis components on the right layer. The more the features and dependencies resemble the structure of the DSML, the less effort has the analysis developer to adapt the model-based analysis to the reference architecture.

Figure 3.19 depicts the steps of modularising a monolithic, model-based analysis. The process contains eight steps in total, one prerequisite step and seven steps for the modularisation of model-based analyses. In the remainder of this section, we present the details of each process step. For each step, we present the roles that are required for the step, a detailed description, and the results.

3.5.1.1 Prerequisite: Modular DSML

Roles Involved: Language Architect, Language Component Developer

Each model-based analysis has a corresponding DSML that it can analyse. Before the analysis developer can modularise an existing model-based analysis according to our reference architecture, we require that the DSML follows the reference architecture of Heinrich et al. [HSR19]. If the DSML is not modularised, the language architect must refactor the DSML according to the reference architecture for metamodels. We recommend refactoring the DSML before modularising the model-based analysis, as it requires less effort to change a metamodel than to change the code-base of a model-based analysis.

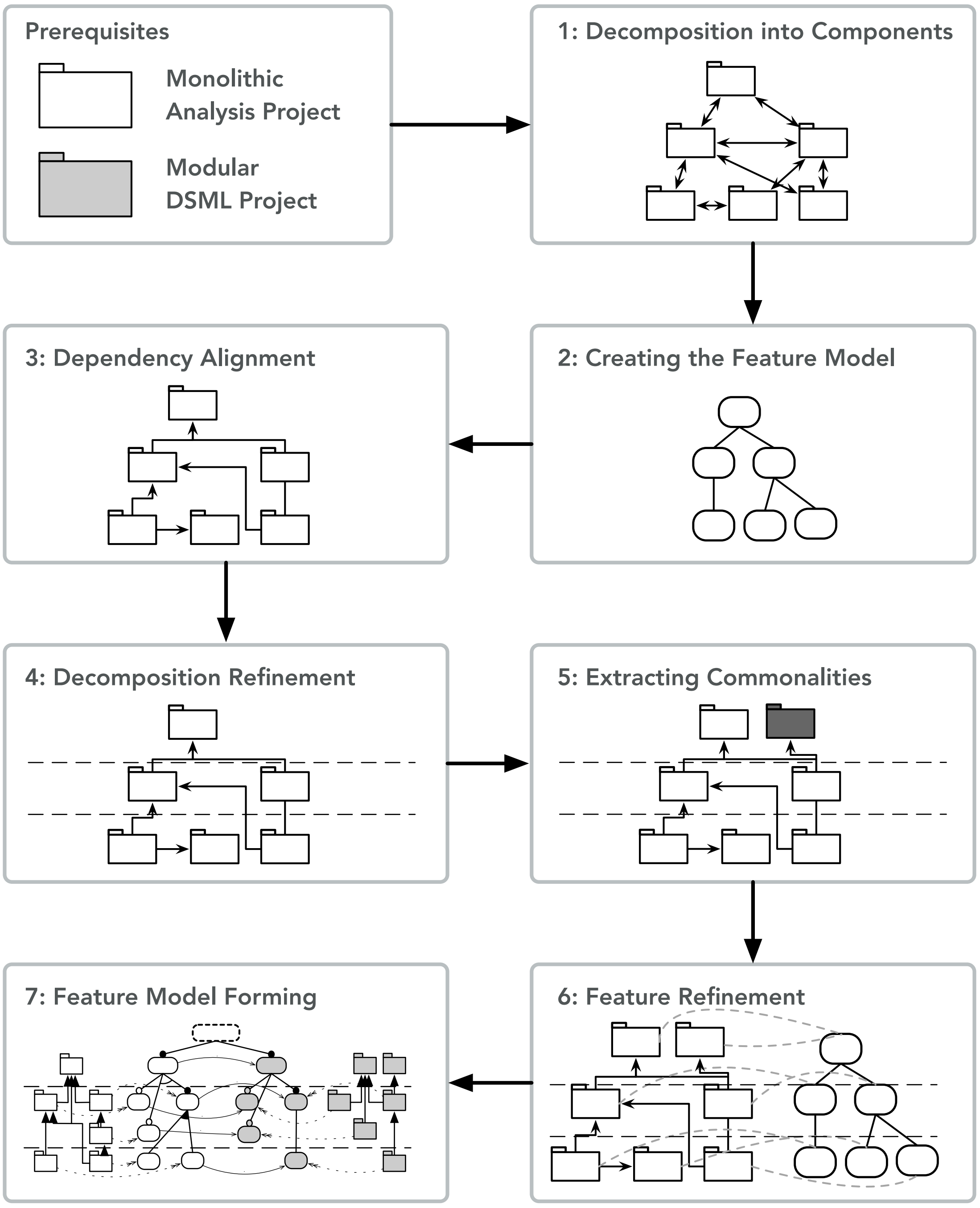

Prerequisites
Monolithic Analysis Project
Modular DSML Project
1: Decomposition into Components
2: Creating the Feature Model
3: Dependency Alignment
4: Decomposition Refinement
5: Extracting Commonalities
6: Feature Refinement
7: Feature Model Forming

Figure 3.19.: Modularisation – Process Overview

3.5.1.2 Step One: Decomposition into Layers

Roles Involved: Analysis Architect, Analysis Component Developer, Language Architect, Language Component Developer

Figure 3.20 depicts the detailed process of this step. For the visualisation, we use the Business Process Modeling Notation 2 (BPMN2). The analysis architect checks the doc-

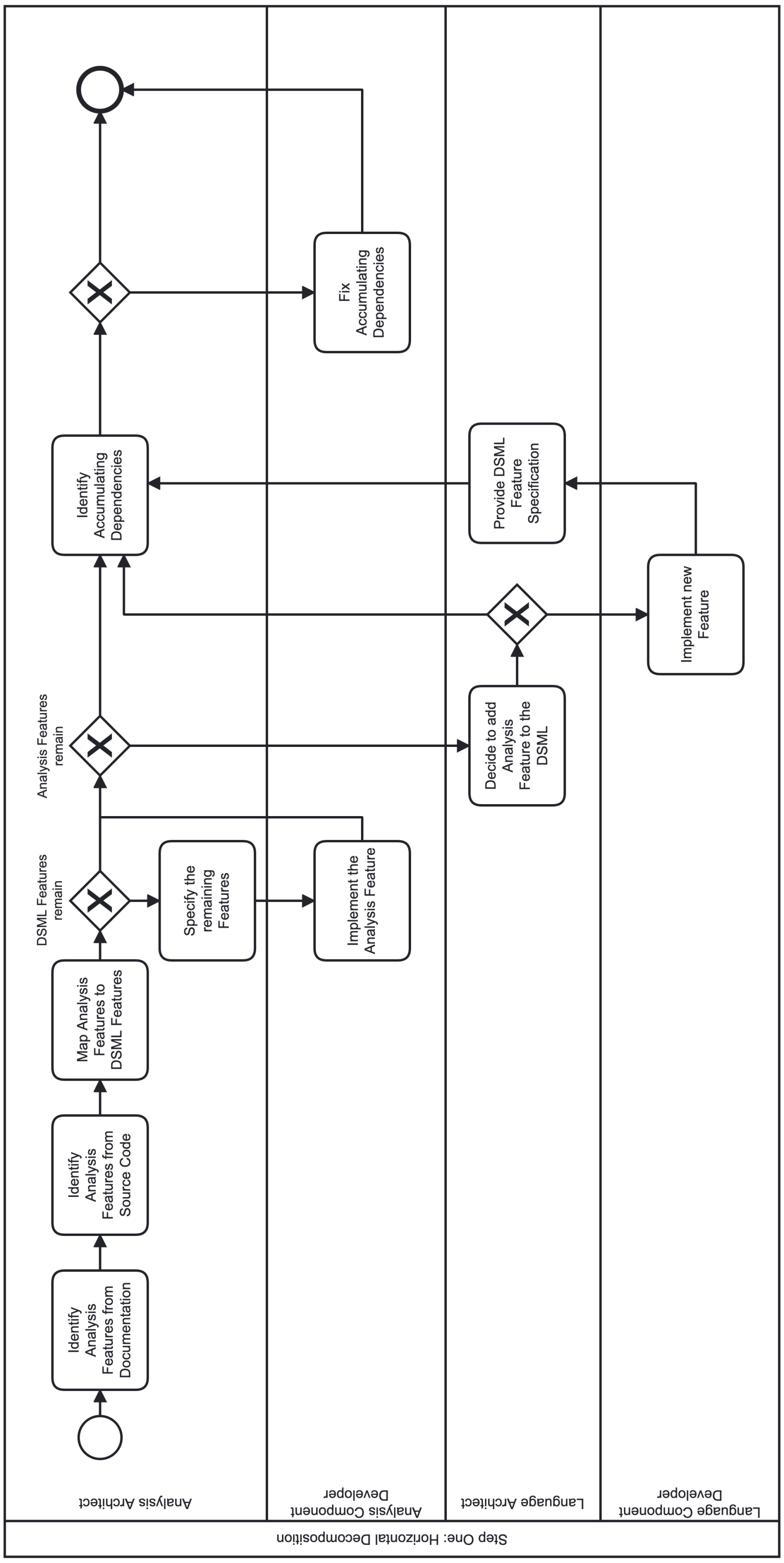

Step One: Horizontal Decomposition
Analysis Architect
Analysis Component Developer
Language Architect
Language Component Developer
Identify Analysis Features from Documentation
Identify Analysis Features from Source Code
Map Analysis Features to DSML Features
DSML Features remain
Specify the remaining Features
Implement the Analysis Feature
Analysis Features remain
Decide to add Analysis Feature to the DSML
Implement new Feature
Provide DSML Feature Specification
Identify Accumulating Dependencies
Fix Accumulating Dependencies

Figure 3.20.: Modularisation Step One: Decomposition into Layers

umentation and the source code of the model-based analysis to identify its features. To identify the features of the model-based analysis, they can consider the package or module structure of the source code. Then they identify the features that correspond to the features of the DSML. If the DSML has features that the model-based analysis is missing, the analysis architect can decide whether to add the feature to the model-based analysis or to leave the feature out. When they decide to add the missing feature, the analysis architect specifies the feature so that the analysis developer can implement the corresponding analysis component. If the model-based analysis has features that are not part of the DSML, the analysis architect decides, together with the language architect, to add the feature to the DSML. When they decide to add the feature to the DSML, the language component developer must implement the language component. The result is a set of analysis features and analysis components.

The next step for the analysis architect is to identify the problem of the accumulation of dependencies to identify the analysis components that must be split up. This problem occurs when an analysis component has dependencies on multiple analysis features of different layers. The analysis component developer can split these analysis components up by using the *horizontal split refactoring* (cf. Section 3.3.3.2). The analysis component developer performs the horizontal split refactoring until all accumulation of dependencies is gone, or the remaining analysis components cannot be split up further. Finally, the analysis component developer assigns the analysis components to their designated layer.

In SimuLizar, for example, the *RDSEFFSwitch* class, one of the biggest accumulation of dependencies, were in the interpreter component of SimuLizar. We split the *RDSEFFSwitch* into separate classes that we then could place on π, Δ, and Ω.

Result: As the analysis component developer did not fix any other problems, the result is a set of highly interconnected analysis components that are placed at their designated layer. The dependencies can point in the wrong direction (i. e., from a generic to a more specific layer), and the dependencies can form cycles. These errors will be addressed in the following steps.

3.5.1.3 Step Two: Creating the Feature Model

Roles Involved: Analysis Architect

In the second step, the analysis architect creates the feature model for the model-based analysis. First, they populate the feature model with the features that are also present in the feature model of the DSML; however, they exclude the features discarded in *Step One: Decomposition into Layers* Section 3.5.1.2. Then, they check which analysis component does not have a corresponding analysis feature. For each analysis component without an analysis feature, they create a feature in the feature model; however, they ignore the dependencies of the analysis components. As stated in Section 3.5.1.2, the dependencies of the analysis components may not conform to our reference architecture. Thus, the analysis architect models the feature dependencies according to our reference architecture.

In our case study SimuLizar, the analysis contains an analysis component that supports the reconfiguration of an architecture model [KHR22a]. We declared for this analysis component a new feature, which does not have a representation in the PCM. We place this new feature in accordance with the constraints of our reference architecture and its dependencies.

Result: The result is a feature model of the model-based analysis corresponding to the feature model of the DSML. The feature model also contains new features that are exclusive to the model-based analysis. The dependencies of the analysis features conform to our reference architecture; however, the dependencies of the analysis components can violate the constraints of our reference architecture (i. e., dependency cycles or wrong dependency direction).

3.5.1.4 Step Three: Dependency Alignment

Roles Involved: Analysis Architect, Analysis Component Developer

Figure 3.21 depicts the process of this step. In the third step, the analysis architect checks the dependencies of the analysis components. They identify where the dependencies are aligned, which means they check whether a dependency at the analysis feature level is present at the analysis component level. They also check whether a dependency is missing at the analysis component level; i. e., the feature model has a dependency that is missing at the analysis component level. The analysis architect also checks the direction of the dependencies, whether they are aligned with the analysis feature dependencies, and whether they conform to the constraints of our reference architecture. A good starting point for checking the dependencies is at the most specific layer of the model-based analysis, as the most specific analysis components have the least number of incoming dependencies.

Aligning the dependencies requires that the analysis architect and the analysis component developer looping through the following steps until the dependencies at the analysis component level are aligned and conform to our reference architecture. The steps are for one pair of analysis components and one dependency between these analysis components:

1. The analysis architect checks whether the two classes that are responsible for the dependency are placed at the layers so that the dependency between these classes points from the specialised to the more generic layer. If that is not the case, they move the class at the more generic layer into the analysis component at the more specific layer.
2. If they encounter a class that does not belong in one of the analysis components, they determine an analysis component to which the class belongs. When they cannot find a fitting analysis component, they move the class to a new analysis component. This new analysis component can also host other classes that are extracted during this process.

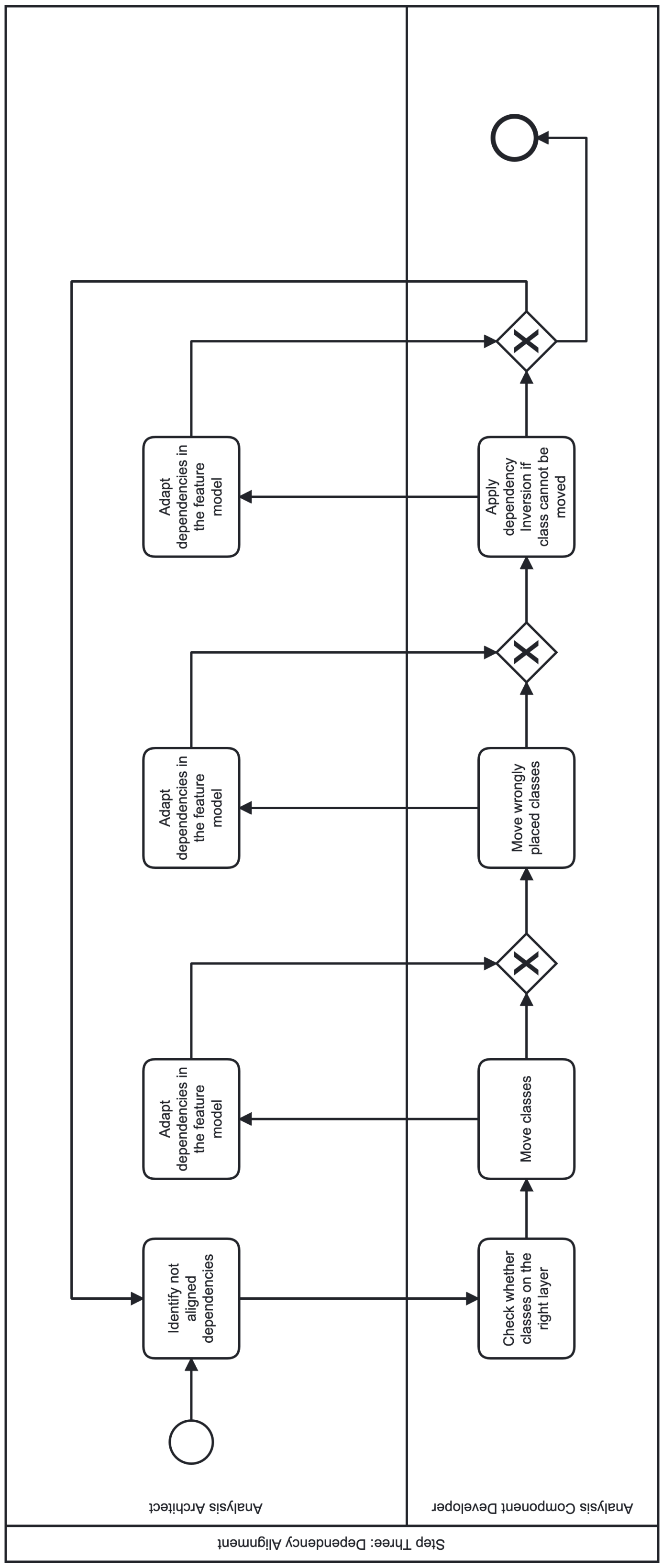

Step Three: Dependency Alignment
Analysis Architect
Analysis Component Developer
Identify not aligned dependencies
Check whether classes on the right layer
Move classes
Adapt dependencies in the feature model
Move wrongly placed classes
Adapt dependencies in the feature model
Apply dependency Inversion if class cannot be moved
Adapt dependencies in the feature model

Figure 3.21.: Modularisation Step Three: Dependency Alignment

3. If they cannot move a class due to various reasons, they perform the *dependency inversion* refactoring (cf. Figure 3.10) of the dependency.
4. For a missing dependency, the analysis architect can move a class that does not belong to another analysis component but has the dependency to the desired analysis component. They omit this step if there is no available class until a class with the desired dependency is identified.

Each step requires the analysis architect to update the feature model accordingly.

In the context of the model-based analysis SimuLizar, we had to split the *RDSEFFSwitch* class in *Step One*. This split, however, created a cyclic dependency of three classes. To break the cycle, we introduced a builder class that inverted the dependency at one class; thus, we could break the cycle.

Result: The result is a modular model-based analysis that is free of dependency cycles and wrong dependency directions. Thus, all dependencies, either between analysis components on different layers or analysis components on the same layer, are aligned with the dependencies in the feature model.

3.5.1.5 Step Four: Vertical Decomposition

Roles Involved: Analysis Architect, Analysis Component Developer

Although the refactored model-based analysis is modular after *Step Three*, the analysis architect cannot reuse the analysis features of the model-based analysis for other quality properties or domains. If the model-based analysis still contains the accumulation of dependencies that are located on exactly one layer, the analysis features and the analysis component that form the blob are not reusable individually. Therefore, the analysis component developer splits the affected analysis components to remove the accumulation of dependencies. *Step Four* is done to improve the reusability of the model-based analysis. In this step, the analysis architect also assigns analysis of the analysis components to the same layer as their corresponding analysis feature.

In the context of the model-based analysis SimuLizar, the components resulting from the *RDSEFFSwitch* split still contained the accumulation of dependencies. Therefore, we had to split the components further to resolve this problem.

Result: The result of this step is a model-based analysis that contains features that an analysis architect can reuse for different domains and for the analysis of different quality properties.

3.5.1.6 Step Five: Extracting Commonalities

Roles Involved: Analysis Architect, Analysis Component Developer

In the fifth step, the analysis architect refines the π layer. Until this step, the paradigm layer only contains features that are present in both the language feature model and the analysis feature model. The remaining analysis features in the Δ layer can also contain analysis-specific features that are so generic that they are relevant for other model-based analyses, regardless of their domain. Therefore, the analysis architect identifies analysis features that are contained in the layers above π but are fundamental for model-based analyses.

If the analysis architect must refactor an analysis component (e. g., move classes out of the component) and the affected classes are not abstract, they create a new abstract class. That abstract class contains the fundamentals of the analysis component. The analysis component developer moves the remaining parts (attributes, methods) into the concrete class. This concrete class inherits from the new abstract class. By performing these steps, the domain-specific parts of a analysis component remain on the Δ layer. As before, after each performed refactoring, the analysis architect updates the analysis feature model.

In the context of the model-based analysis SimuLizar, we did not have to introduce more generic components to the π layer. The reason is a tailored DSML that also considers the analysis-specific features.

Result: This step results in a more tailored π layer and a more reusable Δ layer, as the generic features are decoupled from the domain features. However, analysis components and classes could be too fine-grained. An indicator for too fine-grained analysis components is the high number of classes that are contained in an analysis component or that the size of some classes is small. We do not provide a counting metric to determine small analysis components or classes; however, in the following step, we provide an indicator to identify analysis components or classes that might not be ideally sized.

3.5.1.7 Step Six: Feature Refinement

Roles Involved: Analysis Architect, Analysis Component Developer

Until this step, the focus of the analysis architect was to modularise the model-based analysis and to align the DSML and analysis feature models. The analysis architect identifies too fine-grained analysis components and classes in this step. Therefore, they search for scattered dependencies. The scattering of dependencies occurs when a feature has incoming dependencies of different analysis components. Often, the analysis architect cannot avoid that an analysis feature has incoming dependencies of multiple analysis components. Suppose these components, however, are located on the same layer, and they also do not depend on multiple features. In that case, the analysis component developer can perform the refactoring *class merge* to reduce the number of scattered dependencies.

In the context of the model-based analysis SimuLizar, the merge of classes or analysis components was not required.

Result: This step results in a more concise model-based analysis with analysis components with a specific purpose.

3.5.1.8 Step Seven: Feature Model Forming

Roles Involved: Analysis Architect, Analysis Component Developer, Language Architect, Language Component Developer

After creating and refactoring the feature model, the feature model must be created by the analysis architect. The initial step is to create a root feature. Then, the steps *Step Six: Feature Grouping* (Section 3.5.2.7), *Step Seven: Parent Feature Identification* (Section 3.5.2.8), and *Step Eight: Child Feature Type Determination* (Section 3.5.2.9) from the following Section 3.5.2 are performed by the analysis architect.

Result: After finishing this step, the result is a modular model-based analysis that conforms to our reference architecture for model-based analyses.

3.5.2 Developing a Model-based Analysis from Scratch

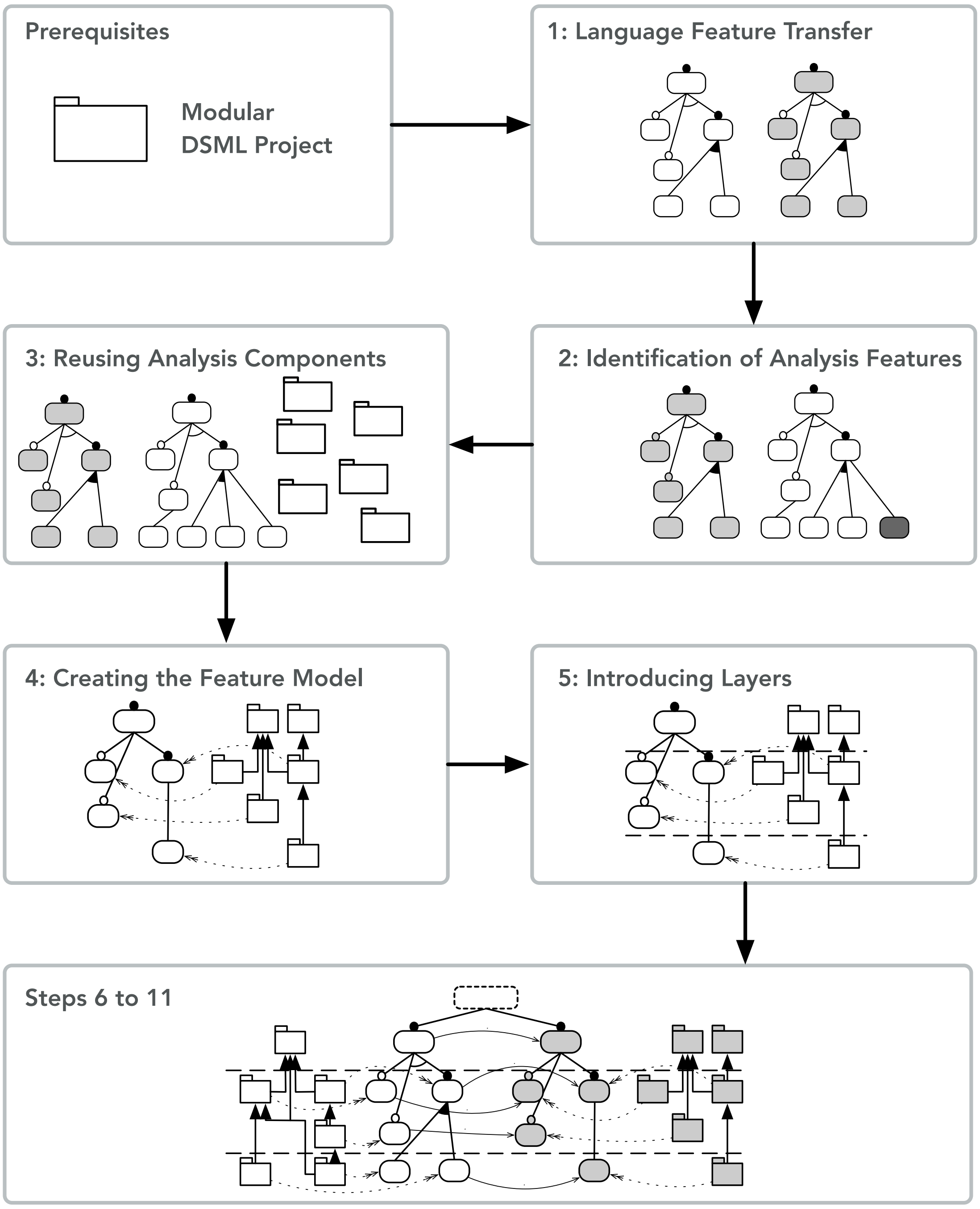
In this section, we present the application process to create a new model-based analysis. The following steps are intended to be executed iteratively. To express the variability, we use feature models of the model model-based analyses (cf. Section 3.3, Apel et al. [AK09], and Czarnecki et al. [Cza+12]). In some cases, it can be beneficial for the analysis architect to backtrack to a previous step, for example, if they overlooked a feature or they identified a feature that they can now split.

Figure 3.22 depicts steps in the process of developing a model-based analysis from scratch. The process contains eleven steps in total; steps six to eleven are combined because they would depict only small variations of the picture shown in step five. In the remainder of this section, we present the details of each process step. For each step, we present the roles that are required for the step, a detailed description, and the results after performing a step.

3.5.2.1 Step One: Language Feature Transfer

Roles Involved: Analysis Architect

For this process to work, we assume that for the model-based analysis, we want to develop, a corresponding DSML already exists. Furthermore, we require that the DSML corresponds to the reference architecture for metamodels (cf. Heinrich et al. [HSR19]). If a DSML exists, but the DSML is not modularised, we advise the language architect and language developer to modularise the DSML before proceeding with this process.



Prerequisites
Modular DSML Project
1: Language Feature Transfer
2: Identification of Analysis Features
3: Reusing Analysis Components
4: Creating the Feature Model
5: Introducing Layers
Steps 6 to 11

Figure 3.22.: New Model-based Analysis – Process Overview

In the first step, the analysis architect creates the analysis feature model. First, they add the features to the feature model that also exist in the language feature model. They also adopt the layers and feature dependencies from the language feature model.

Result: The result is an analysis feature model that mirrors the language feature model.

3.5.2.2 Step Two: Identification of Analysis Features

Roles Involved: Analysis Architect, Analysis User

In the second step, the analysis architect identifies features that exist exclusively in the model-based analysis. Therefore, they work together with the analysis user, or if no analysis user is available, the analysis architect estimates their concerns. The analysis architect should elicitate the requirements after the feature model of the DSML is completed to avoid unnecessary changes to the analysis feature model. If it is not avoidable for the analysis architect because the DSML and model-based analysis development start simultaneously, the potential effort to implement the changes are limited. In this starting phase, no code is written; thus, the analysis components do not exist. If this step is visited in a later development stage, the effort to implement the changes is higher, as the analysis components are surely affected by the changes.

Result: The result of this step is an analysis feature model that contains analysis-specific features.

3.5.2.3 Step Three: Reuse of Analysis Components

Roles Involved: Analysis Architect

In the third step, the analysis architect identifies already existing analysis components. Therefore, they can utilise our approach to identify already existing analysis components by comparing the structure and behaviour of analysis components (cf. Chapter 5). We do not distinguish between in-house analysis components or publicly available analysis components.

The analysis architect selects an analysis feature and searches the available analysis components for a component that matches the feature specification. If the feature was specified with our DSL to specify the structure and behaviour of analysis features (Chapter 5), the analysis architect can use our toolchain [KR22] to find matching analysis components automatically. The analysis architect assigns the found analysis components to their corresponding analysis feature in the analysis feature model.

Result: The result of this step is an extended feature model that contains reusable analysis components.

3.5.2.4 Step Four: Creating the Feature Model

Roles Involved: Analysis Architect

In the fourth step, the analysis architect creates the feature model. Therefore, they create the root node of the feature model and then label it. Usually, the root node gets labelled after the name of the model-based analysis. In the case of SimuLizar, we called the root node simply SimuLizar. Then, the analysis architect adds the identified features from *Step*

Two: Analysis Feature Identification (Section 3.5.2.1) to the feature model. After they added these features, they added the analysis-specific features identified in *Step Two: Analysis Feature Identification* (Section 3.5.2.2).

Up to this point, the analysis architect added only features to the feature model without considering the relationship between these features. To add the relationship between two features A and B, we define the following rules:

Application of Requires Relation

- if a reused analysis component that implements A has a dependency on a reused analysis feature that implements B
- if analysis feature A is an extension of analysis feature B
- if analysis feature A is dependent on the content of analysis feature B

Application of Excludes Relation

- if analysis feature A prohibits analysis feature B or vice versa

As we prohibit cycles of requires-relations, the analysis architect has to break these cycles up (e. g., by using the dependency inversion).

Result: The result of this step is a feature model of the model-based analysis.

3.5.2.5 Step Five: Introducing Layers

Roles Involved: Analysis Architect

The analysis architect introduces layers to the feature model in the fifth step. Therefore, they assign each feature to a single layer by sticking to the constraints of our reference architecture. In this step, they omit the π layer (cf. *Step Six: Paradigm Extraction*). To distribute the features, the analysis architect must follow these steps:

1. The analysis architect assigns analysis features that also have representations in the language to the same layer.
2. To extract features that are not relevant to the current layer, they create a new analysis feature containing irrelevant parts. They assign to the current layer the original analysis feature. The architect handles the newly generated, unassigned analysis feature when the following layer is modularised. They declare a requires-relation from the new analysis feature to the original one.
3. To conform to the layering, and the specified dependency direction of the reference architecture, they must reverse the feature dependencies of the more basic layers to the analysis features of the current layer.

Result: This step results in a feature model where the features are separated into layers while the π layer is still empty.

3.5.2.6 Step Six: Extracting the Paradigm Layer

Roles Involved: Analysis Architect

In the sixth step, the analysis architect fills the remaining empty π layer. Therefore, they have to decide whether a analysis feature represents a fundamental feature of model-based analyses. They also can create new analysis features and place them on the π layer. For each feature on the π layer, they must add requires relations from features of the Δ layer if they depend on the new analysis features.

Result: This step results in a feature model where all layers are populated.

3.5.2.7 Step Seven: Grouping of Features

Roles Involved: Analysis Architect

In the seventh step, the analysis architect groups analysis features that share concepts or properties. For example, analysis features can share types, structural abstractions or behaviour. Therefore, the analysis architect does model a logical grouping of such features. A hard restriction is that the analysis architect can only group features that are located on the same layer. If they want to group analysis features of different layers, they first have to move the analysis features to the same layer.

The analysis architect groups features by introducing a new feature. This new feature is then declared as a parent for each analysis feature in the group. The analysis architect also adds alternative and OR conditions to the grouping. In this step, they also add excludes relations to the features. For each excludes relation, they must ensure that an alternative feature selection exists.

Result: The result of this step is a partially connected feature model that contains alternative and OR sets. The feature model also contains excludes relations.

3.5.2.8 Step Eight: Parent Feature Identification

Roles Involved: Analysis Architect

In the eighth step, the analysis architect identifies the analysis features on the π layer that directly relates to the feature model's root node. The indicators for the analysis architect to identify such analysis features are:

- A composed analysis feature contains atomic analysis features that are fundamental to the analysis. In the context of the analysis SimuLizar, such a feature would be a simulation-type feature.
- A composed analysis feature contains atomic analysis features that are shared by analyses. In the context of the analysis SimuLizar, such a feature would be the definition of a time feature that is used throughout the analysis.

- An analysis feature has no outgoing feature dependencies

The analysis architect then identifies the parent features for the remaining analysis features. An incoming requires-relation can identify the parent. When one feature extends another, they define a parent relationship from the extending feature to the extended feature. The parent relationship always replaces an existing dependent relationship between the two features. A parent relation, such as the requires-relations, must not refer to a more specific layer.

Result: The result of this step is that the analysis features of the π layer are now connected.

3.5.2.9 Step Nine: Adding the Remaining Dependencies

Roles Involved: Analysis Architect, Analysis Component Developer, Language Architect, Language Component Developer

In the ninth step, the analysis architect connects the remaining analysis features in the feature model by determining the child features in the feature model. In *Step Seven: Feature Grouping* (Section 3.5.2.7), the analysis architect already connected some analysis features. For the remaining features, the analysis architect determines the parent features. They mark each parent feature as mandatory. The root analysis feature is the only feature in the feature model that has no parent feature. Dependencies that cross the π layer must be OR sets; otherwise, the analysis features of the π layer could be selected without a analysis feature of the next layer. As each analysis feature of the π layer is abstract or an interface, the analysis would not be usable. The OR set ensures that each analysis feature of the π layer is selected with at least one child of the next layer. Dependencies that cross other boundaries must be optional; otherwise, the next layer is mandatory.

Result: The result of this step is a fully interconnected feature model.

3.5.2.10 Step Ten: Implementing the Features

Roles Involved: Analysis Architect, Analysis Component Developer

In this step, the analysis component developer implements the analysis components according to the feature model that the analysis architect developed in the previous steps. Not every analysis feature has a corresponding analysis component; for example, the root node and the parent node created to group features are merely containers to host the model-based analysis (root node) or a logical group of analysis features.

The analysis component developer can add new dependencies while implementing the components. When adding new dependencies, the analysis component developer must adhere to the constraints of our reference architecture. The analysis architect then must update the feature model accordingly.

If the analysis component developer notices that an analysis component requires multiple language features for an analysis, they create an analysis component for each language feature. The analysis component developer then introduces an indirection. They create an additional analysis component that references the other analysis components that depend on the language feature.

Result: The result of this step is a analysis component model with corresponding analysis code.

3.5.2.11 Step Eleven: Revision and Refinement

Roles Involved: Analysis Architect, Analysis Component Developer

In this final step, the analysis architect has a modular model-based analysis that corresponds to our reference architecture. Until this step, the analysis architect and the analysis component developer aligned the structure of the model-based analysis to its corresponding DSML. To finalise the modularisation of the model-based analysis, they can still make changes to the analysis feature model. These changes are mostly done to refine the model and the code. The analysis architect can use the insight gained by the analysis component developer during the implementation of the analysis components for further improvement.

Result: After finishing the last iteration, the result is a newly developed modular model-based analysis that conforms to our reference architecture for model-based analyses.

3.5.3 Extending a Model-based Analysis

In this section, we present the application process to extend an already existing model-based analysis. Therefore, the model-based analysis that is extended must conform to our reference architecture. The steps are intended to be executed sequentially; however, if necessary, the analysis architect can backtrack no previously executed steps.

Figure 3.23 depicts the steps of developing a model-based analysis from scratch. The process contains five steps in total. In the remainder of this section, we present the details of each process step. For each step, we present the roles that are required for the step, a detailed description, and the results.

3.5.3.1 Step One: Identification of Analysis Features

Roles Involved: Analysis Architect, Analysis User

In the first step, the analysis architect investigates the goal of the planned extension and identifies analysis features that are required by the extension but are not yet part of the model-based analysis. They can use the concerns of the analysis user to identify new analysis features for the model-based analysis. Alternatively, if the DSML has new features,

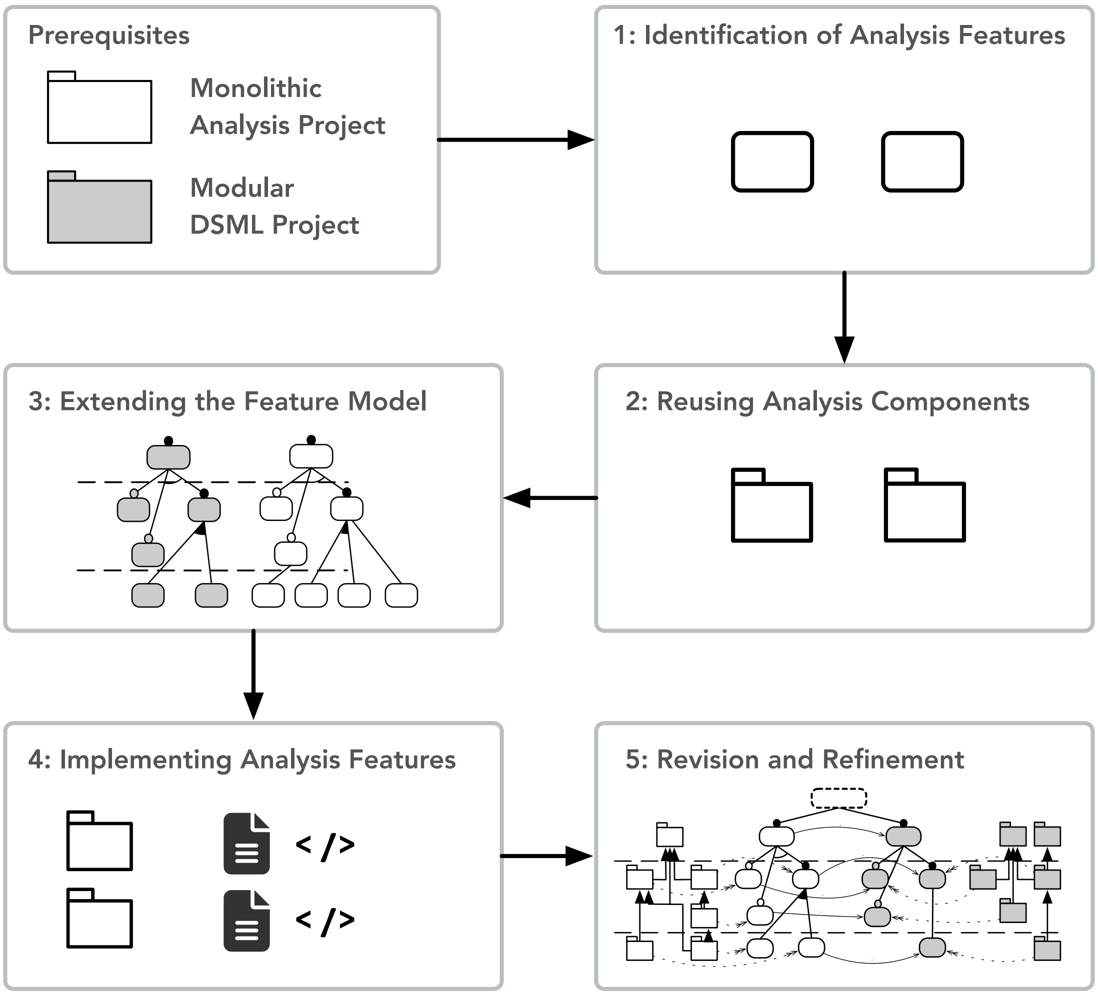

Prerequisites
Monolithic Analysis Project
Modular DSML Project
1: Identification of Analysis Features
2: Reusing Analysis Components
3: Extending the Feature Model
4: Implementing Analysis Features
</>
</>
5: Revision and Refinement

Figure 3.23.: Extending a Model-based Analysis – Process Overview

the analysis architect decides whether to add the new language features to the model-based analysis.

Result: This step results in a set of new analysis features that are required for a planned extension of the model-based analysis.

3.5.3.2 Step Two: Reusing Analysis Components

Roles Involved: Analysis Architect, Analysis Component Developer, Language Architect, Language Component Developer

In the second step, the analysis architect identifies already existing analysis components to implement the new analysis features. Therefore, they can utilise our approach to identify already existing analysis components by comparing the structure and behaviour of analysis components (cf. Chapter 5 and [Koc+22; KR22]). We do not distinguish between in-house analysis components or publicly available analysis components.

The analysis architect selects one of the new analysis feature and searches the available analysis components for a component that matches the feature specification. If the feature was specified with our DSL to specify the structure and behaviour of analysis features (Chapter 5), the analysis architect can use our toolchain [KR22] to automatically find matching analysis components. The analysis architect assigns the found analysis components to their corresponding analysis feature in the analysis feature model.

Result: The result of this step is an extended feature model that contains reusable analysis components.

3.5.3.3 Step Three: Extending the Feature Model

Roles Involved: Analysis Architect

In the third step, the analysis architect extends the analysis feature model by the newly identified features. If the feature corresponds to a language feature, they name the new analysis feature accordingly. Also, the analysis architect places the new analysis feature in the same layer with the same dependencies as their counterpart. New analysis features with no representation in the DSML must be placed as described in Section 3.5.2.5 *Step Five: Layering*. The dependencies are modelled following Section 3.5.2.8 *Step Eight: Parent Feature Identification* and Section 3.5.2.9 *Step Nine: Adding the Remaining Dependencies*.

Result: The result of this step is an extended feature model with its necessary dependencies.

3.5.3.4 Step Four: Implementing the Remaining Analysis Features

Roles Involved: Analysis Architect, Analysis Component Developer

In the fourth step, the analysis component developer implements the new analysis components according to the feature model that the analysis architect extended in the previous steps (cf. *Step Ten: Feature Implementation* Section 3.5.2.10).

The analysis component developer can add new dependencies while implementing the components. When adding new dependencies, the analysis component developer must adhere to the constraints of our reference architecture. The analysis architect then must update the feature model accordingly.

Result: The result of this step is an analysis component model with corresponding analysis code.

3.5.3.5 Step Five: Revision and Refinement

Roles Involved: Analysis Architect, Analysis Component Developer

In this step, the analysis architect can make changes to the analysis feature model. These changes are mostly done to refine the model and to use the insight gained by the analysis component developer during the implementation of the analysis components. They can return to the first step to iterate over the steps. Especially when, during the development of the extension, the requirements change due to new insights (e. g., better performance or new language features). Furthermore, to refine the extension, the analysis architect and the analysis component developer can exchange or modify reused analysis components to adapt to the changed requirements.

Result: After finishing the last iteration, the result is a modular model-based analysis with an extension that conforms to our reference architecture for model-based analyses.

3.6 Technical Contribution for the Analysis and the Refactoring of Model-based Analyses

In this section, we present our tooling that aids the analysis architect and the analysis component developer to analyse and refactor model-based analyses. Our tooling is separated into an Application Programming Interface (API) that provides interfaces for analysing and refactoring model-based analyses and a User Interface (UI) reference implementation that allows the analysis architect and the analysis component developer to access the analysis and refactoring capabilities of our tooling. We implemented the UI as a Command Line Interface (CLI). We named our tool Refactor Lizar and the reference implementation Refactor Lizar CLI. First, we present the analysis part of Refactor Lizar in Section 3.6.1. Second, we present the refactoring capabilities of our tool in Section 3.6.2. The reference implementation of Refactor Lizar can be found on GitHub [KWc].

3.6.1 Analysis Library – Refactor Lizar

We implemented the analysis part of Refactor Lizar as a Java library. The library is available on GitHub [KWb] or Maven Central [KWa]. In this section about Refactor Lizar, we will focus on the analysis capabilities that the analysis architect can utilise to identify the parts of the model-based analysis that are not conforming to our reference architecture. We differentiate between issues that result from the feature and component structure of the model-based analysis and its corresponding DSML, and issues that result from constraints of our reference architecture. First, we present in Section 3.6.1.1 the identification and refactoring of the accumulation of dependencies of one class on multiple analysis features.

3.6.1.1 Accumulation of Dependencies Detection

The accumulation of dependencies occurs when multiple language features are used in one analysis component or one analysis class. In our previous work [Hei+21b], we provide insights into the Palladio Simulator, where we derived problems that occurred during the development of the Palladio Simulator. We identified the accumulation of dependencies as one of the problems that occurred during the development of the Palladio Simulator. In order to find accumulated of dependencies, the analysis architect must provide the path to the DSML and the path to the model-based analysis code. The model-based analysis must be written in Java 17 or older in order for Refactor Lizar to be able to analyse the model-based analysis. The analysis result is an analysis report that contains a list of the affected language types and affected analysis components. If the analysis architect needs a more detailed report, Refactor Lizar can also provide the affected classes in its report. If these analysis components are located on multiple layers, the analysis architect must merge these analysis components and place them on the same layer as the language feature.

In the following Chapter 4, we analyse reoccurring patterns that negatively affect the evolvability and reusability of model-based analyses. More details regarding the accumulation of dependencies can be found in Section 4.2.4.4 – Rebellious Modularity.

3.6.1.2 Detection of Scattered of Dependencies

The accumulation of dependencies occurs when a type of a language feature is used in multiple analysis components. In our previous work [Hei+21b], we provide insights into the Palladio Simulator, where we derived problems that occurred during the development of the Palladio Simulator. We identified the scattering of dependencies as one of the problems that occurred during the development of the Palladio Simulator. In order to find a scattering of dependencies, the analysis architect must provide the path to the DSML, and the path to the model-based analysis code. The model-based analysis must be written in Java 17 or older in order for Refactor Lizar to be able to analyse the model-based analysis. The analysis result is an analysis report that contains a list of the affected language types and affected analysis components. If the analysis architect needs a more detailed report, Refactor Lizar can also provide the affected classes in its report. If these analysis components are located on multiple layers, the analysis architect must merge these analysis components and place them on the same layer as the language feature.

In the following Chapter 4, we analyse reoccurring patterns that negatively affect the evolvability and reusability of model-based analyses. More details regarding the scattering of dependencies can be found in Section 4.2.4.2 – Degraded Modularity.

3.6.1.3 Layer Violation Detection

A layer violation occurs when dependencies of an analysis component point from a generic to a more specific analysis component or if a dependency skips a layer. Another type of layer violation is when an analysis feature is located on a different layer as a corresponding analysis component. In order to fix this problem, the analysis architect must either invert the dependency (if it points in the wrong direction) or introduce a new analysis component in between the skip. If the location of an analysis feature and an analysis component are not the same, the analysis architect fix this smell by moving either the analysis feature of the analysis component to the right layer. The layer identification of analysis components requires further annotation by the analysis developer, while the layer identification of referenced DSML types is made automatically. To help the analysis architect identify layer violation occurrences, Refactor Lizar provides an interface to identify this bad smell automatically.

In order to find occurrences of a layer violation, the analysis architect must provide the path to the model-based analysis code. The model-based analysis must be written in Java 17 or older in order for Refactor Lizar to be able to analyse the model-based analysis. The analysis result is an analysis report that contains a list of affected analysis components. If the analysis architect needs a more detailed report, Refactor Lizar can also provide the affected classes in its report.

In the following Chapter 4, we analyse reoccurring patterns that negatively affect the evolvability and reusability of model-based analyses. More details regarding layer violations can be found in Section 4.2.3.

3.6.1.4 Dependency Cycle Detection

A dependency cycle occurs when the dependencies of components or classes form a loop. Such a dependency cycle hampers the extensibility or changeability of the affected elements. To fix a dependency cycle, the analysis architect has to invert the dependency of one or more classes of the cycle (cf. Section 3.3.3.1).

In order to find occurrences of dependency cycles, the analysis architect must provide the path to the model-based analysis code. The model-based analysis must be written in Java 17 or older in order for Refactor Lizar to be able to analyse the model-based analysis. The analysis result is an analysis report that contains a list of the affected analysis components. If the analysis architect needs a more detailed report, Refactor Lizar can also provide the affected classes in its report.

In the following Chapter 4, we analyse reoccurring patterns that negatively affect the evolvability and reusability of model-based analyses. More details regarding dependency cycles can be found in Section 4.2.4.5, where we investigate cycles that are formed between a model-based analysis and a corresponding DSML.

3.6.1.5 Complexity, Coupling, and Cohesion Analysis

For the evaluation of our reference architecture we implemented the metric calculation as part of Refactor Lizar. Details regarding the metrics' evaluation and selection can be found in Chapter 7.

The metrics calculation is inspired by an implementation of Jung [Jun16]. Before we can analyse a model-based analysis regarding these metrics, the analysis requires the path to the source code that will be analysed. It also needs a set of classes part of the source code the user wants to analyse. The set of classes allows for performing multiple analyses under the same path. Suppose the user wants to analyse all classes or a set of classes with certain parameters in the path's folders and sub-folders. In that case, we allow regular expressions to specify a desired subset. Furthermore, the analysis also allows to specify data classes in the same way as those to analyse. Data classes dilute the result, although they contain no behaviour; therefore, we allow omitting data classes when calculating the metrics. We store this information in two separate text files.

To start the analysis, Refactor Lizar needs the path to files with the observed system (classes to analyse) and to the file with the data classes. Refactor Lizar calculates the observed system's cohesion, coupling, and complexity using the metrics introduced by Allen et al. [All02]. The results are the metrics' values cohesion, coupling, and complexity.

3.6.2 Refactoring Library

We implemented the refactoring part of Refactor Lizar as a Java library. The library is available on GitHub [KWb] or on Maven Central [KWa]. In this section about Refactor Lizar, we will focus on the refactoring features that the analysis architect needs to adapt a model-based analysis so that it conforms to our reference architecture.

The following refactoring operations are supported by Refactor Lizar:

- Move type members of classes
- Introduce inheritance
- Adapt interface extension
- Change the visibility of members
- Change the visibility of methods
- Introduce new types

Refactor Lizar utilises these refactorings to implement refactorings presented in Section 3.3.3. Refactor Lizar supports the following refactorings:

- Class Split (cf. Section 3.3.3.1)
- Class Merge (cf. Section 3.3.3.1)

- Breaking Dependency Cycles (i) (cf. Figure 3.8)
- Dependency Inversion (cf. Section 3.3.3.1)
- Horizontal Split (cf. Section 3.3.3.2)
- Vertical Split (cf. Section 3.3.3.2)
- Component Merge (cf. Section 3.3.3.2)
- Extension Extraction (cf. Figure 3.17)

4. Bad Smells in Model-based Analyses

In software engineering, developing software that can evolve over time is crucial. Evolvability is determined by the effort required by developers to implement changes like extending a feature or adding new features to a software system in a reasonable time frame. In the previous chapter, we introduced our reference architecture for model-based analyses to improve the evolvability and reusability of model-based analyses. However, such a reference architecture is not the philosophers' stone to solve all evolvability and reusability issues of model-based software systems. The reference architecture gives the analysis developer a structure which can guide them through the development and extension process; however, the reference architecture does not prevent issues unrelated to the architecture of a model-based analysis. When the analysis developers, for example, use primitive types instead of dedicated types (cf. Missing Abstraction in Section 4.2.1.2), or when they implement analysis behaviour multiple times (cf. Duplicated Abstraction in Section 4.2.1.1). Such problems negatively affect the evolvability and reusability of model-based analysis.

In this chapter, we focus on the improvement of the evolvability and reusability of model-based analyses that arise due to the co-evolution of model-based analyses and their corresponding DSMLs. Model-based analyses change over time due to new or changing requirements, and developers must adapt them to meet these requirements. The ability to adapt to such changing requirements is hampered by problematic structures in the source code and the architecture. Such problematic structures are called *bad smells*. Bad smells impede the evolvability and reusability of software systems. The term *bad smell* was introduced by Martin Fowler and Kent Beck in the late 90s [Fow99]. According to Fowler et al. [Fow99], bad smells are structures that have the potential for refactoring. Refactorings change the structure of the source code (i. e., move attributes or methods), but they should not change the behaviour of the software. Bad smells are not a theoretical construct; they are derived from the experience of developers that have gathered experience by creating and refactoring source code [Fow18].

Fowler and Beck initially defined 21 bad smells that can occur in object-oriented software. Strittmatter identified bad smells for DSMLs by using object-oriented bad smells as baseline [Str20]. He identified 13 bad smells for DSMLs. Both Fowler and Strittmatter provide strategies on how to fix bad smells.

So far, bad smells for DSMLs and source code have been considered separately, although they are co-dependent. Model-based analyses are based on the DSML they analyse; they need the DSML as analysis input, and changing the DSML results in changes of their corresponding model-based analyses. It is unclear whether such bad smells exist in the domain of model-based analyses or which bad smells originate from the co-dependency of DSMLs and their corresponding model-based analyses. Further, as there are no dedicated bad smells for model-based analyses and their corresponding DSML, we do not know how these dedicated bad smells influence the evolvability and reusability of model-based analyses. If, for example, a DSML and its corresponding model-based analysis form a cycle, the impact of a change to either the DSML or the model-based analysis is unpredictable. Such structures can result in huge costs and the potential failure of a project, as changes to either the DSML or the model-based analysis can result in unforeseen costs and effort.

In this chapter, we investigate the bad smells that are unique for DSMLs and their corresponding model-based analyses. First, we present our hypothesis and research questions in Section 4.1. We present the bad smells of model-based analyses in Section 4.2. Our approaches to identify bad smells in model-based analyses are presented in Section 4.3. In Chapter 8, we present the evaluation of the bad smells; in Section 10.3, we present the related work to our approach. We present the conclusion and future work in Section 11.2.

4.1 Hypothesis and Research Questions

In this section, we present the hypothesis and research questions for the second contribution of this thesis. Detecting and fixing bad smells in object-oriented software [Fow18] or domain-specific modelling language [Str20] helps to improve the internal quality of the source code or the DSML. Such an internal quality improvement reduces the software's complexity and the DSML. Reducing the complexity allows the analysis developer to comprehend the model-based analysis code faster, thus positively influencing the evolvability and reusability. Recognising what fixing bad smells means for software systems and DSMLs, respectively, we derive the following hypothesis for bad smells in the domain of model-based analyses:

Hypothesis 2

The evolvability and reusability of a model-based analysis will improve when fixing bad smells that originate from the co-dependency of model-based analysis and their corresponding DSML.

In order to determine whether our hypothesis 2 is correct, we must answer the following research questions.

Research Question 4.1

Which bad smells arise from the co-dependency of model-based analysis and their corresponding DSML?

To the best of our knowledge, the co-dependency of model-based analyses and DSMLs regarding potential bad smells is unexplored. Therefore, it is unknown whether bad smells even exist for model-based analyses. Before further investigation, we must identify bad smells that exist only in the domain of model-based analyses. Having dedicated bad smells enables analysis developers to improve the internal quality of their model-based analysis by identifying common problems obtained from our empirical analysis. These bad smells can also affect the corresponding DSML, which will positively affect features of the DSML. If, for example, due to refactorings, analysis features are identified that are not used in the model-based analysis although dependencies on the corresponding language feature exist, both features have the potential to be deleted by the language architect and analysis architect respectively.

Research Question 4.2

How to refactor bad smells of model-based analyses and their corresponding DSML without affecting the behaviour of the analysis?

Refactoring operations are intended to change the structure of the software, but the behaviour has to remain the same. For example, changing the name of a field does not affect the behaviour of the code. Identifying bad smells in the domain of model-based analyses is only the first step to improving their evolvability and maintainability. If an analysis developer cannot refactor the bad smells without affecting the behaviour of the model-based analysis, it would require more effort to fix the bad smells. Therefore, we develop refactoring operations that analysis developers can use to fix bad smells in their model-based analyses without altering their behaviour.

Research Question 4.3

How do bad smells of model-based analyses and their corresponding DSML influence the evolvability and reusability of model-based analyses?

After bad smells in model-based analyses are identified and refactored, the effect on evolvability and reusability is unclear. Fixing bad smells is intended to improve the internal quality of a software system. Having a system that is easier to evolve and reuse increases the chance that such a system will longer be maintained and, thus, can longer exist in the market. Therefore, we analyse model-based analyses to find and fix the bad smells we identified when answering research question 4.1. We use the results to determine the effect of the refactorings on the evolvability and reusability of model-based analyses.

4.2 Bad Smells in Model-based Analyses

In this section, we explain the bad smells we derived from bad smells in object-oriented software and from bad smells in DSMLs. We use the classification of structural design smells by Ganesh et al. [GSS13] to distinguish four types of bad smells: *Abstraction*, *Encapsulation*, *Hierarchy*, and *Modularity*. Figure 4.1 shows the classes and the bad smells

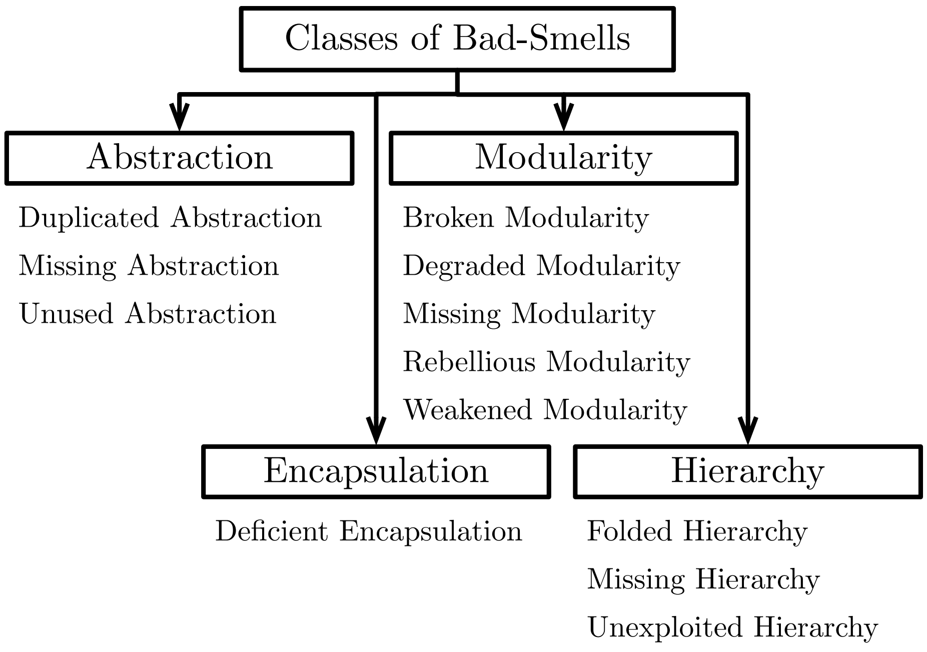


Figure 4.1.: Classification of Bad-Smells in Model-based Analyses

we identified for model-based analyses. The naming scheme of our bad smells is oriented on the four types by Ganesh et al. [GSS13]. They introduce four categories of bad smells:

Abstraction: "An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries relative to the viewer's perspective."

Encapsulation: "Encapsulation is the process of compartmentalising the elements of an abstraction that constitute its structure and behaviour; encapsulation serves to separate the contractual interface of an abstraction and its implementation."

Modularity: "Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules."

Hierarchy: "Hierarchy is a ranking or ordering of abstractions."

In the following subsections, we present the 12 bad smells we identified and specified by analysing existing model-based analyses. We named the bad smells following the naming guidelines by Ganesh et al. [GSS13]. For each bad smell, we developed a process for the analysis developer how to identify and refactor them. We explain the negative effects on the evolvability and reusability of each bad smell and what are the circumstances in which the bad smells are created.

We start with the class of abstraction bad smells in Section 4.2.1. Then, we present the bad smells associated with the encapsulation class in Section 4.2.2 In Section 4.2.3, we present the bad smells associated with the hierarchy class. Finally, in Section 4.2.4, we present the bad smells associated with the modularity class.

4.2.1 Abstraction

An abstraction is a representation of the basic characteristic of an object that sets it apart from all other types of objects and, as a result, gives clearly defined conceptual boundaries relative to the observer's viewpoint.

4.2.1.1 Duplicated Abstraction

We derive the *Duplicate Abstraction* smell from the object-oriented *Duplicated Code* bad smell (cf. Section 2.4.1). The bad smell also looks for duplicated structures; however, unlike the duplicate source code smell, the duplicated abstraction focuses on the dependency structure of the model-based analysis and its corresponding DSML. Figure 4.2 depicts

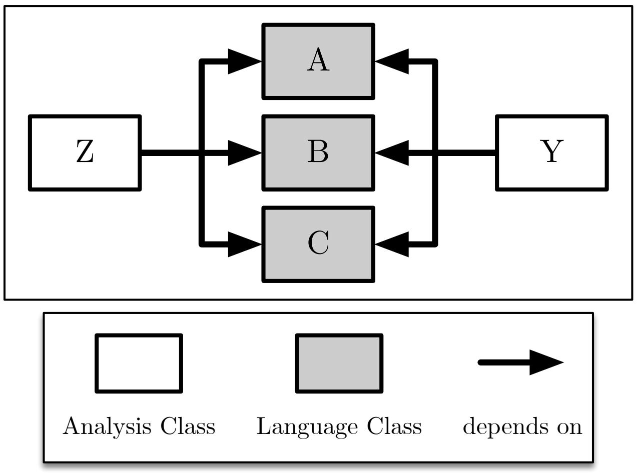


Figure 4.2.: Duplicated Abstraction

the duplicated abstraction smell. The dependency structure of analysis class *Z* and *Y* are identical, which means that the analysis and language classes have the same dependency structure. The analysis classes *Z* and *Y* depend on the language classes *A*, *B*, and *C*. Both classes are part of the same model-based analysis.

Effect: This bad smell indicates that the analysis class that depends on a language type is implemented multiple times. The analysis class is part of an analysis component; the analysis component implements an analysis feature (cf. feature in Chapter 2). In the event of a change to either the language classes or to the implementation of the analysis class, the analysis developer must modify multiple occurrences of such an analysis class, as they presumably implement / analyse the same feature. If the analysis developer does not know about the other occurrences of the analysis class, the model-based analysis will have different behaviour, depending on the invoked analysis class. For example, the tool user models a software system with the PCM (cf. Chapter 2) and the resource demand of a component is calculated in two different analysis classes. If the invocation of the analysis classes depends on factors that are not clear to the tool user, they cannot rely on the analysis result. Another drawback is that the effort required to make changes is *n*-times the number of duplicated analysis classes. If not all duplicated classes are changed during a change, the *Duplicated Abstraction* smell leads to divergent behaviour and, thus, to bugs that the analysis developer must fix. Even if all occurrences of the identical analysis classes are changed, the effort required to implement a modification is more than if the analysis class exists only once.

Causes: When multiple analysis classes implement the same concern, the *Duplicated Abstraction* bad smell arises. When multiple analysis algorithms are merged into one class, a single algorithm might be unusable independently of the remainder of the analysis class. As a result, separating the algorithms of the analysis class requires more effort than implementing the required algorithm in a new class. The more analysis classes implement the same algorithms, the harder it becomes for the analysis developer to comprehend the model-based analysis. Thus, the analysis developers have difficulty knowing each implemented feature of the model-based analysis. This knowledge gap leads to the multiple implementation of an analysis feature.

Identification: As shown in Figure 4.2, the *Duplicated Abstraction* smell occurs if multiple classes in a model-based analysis have dependencies on the same set of language classes. For each analysis class, the analysis developer collects its dependencies and removes all dependencies that point to language classes. Given the set of dependencies, the analysis developer searches for classes that depend on the same language classes. In our analysis, we set the threshold of common dependencies to two so that if classes have only two or fewer dependencies in common, we do not count them as *Duplicated Abstraction* smell. If multiple analysis classes depend on the same language class is no absolute indicator for the *Duplicated Abstraction* smell, as the analysis classes could also analyse different quality properties. Therefore, the analysis developer must decide whether the identified similarity in the dependency structure holds true for the *Duplicated Abstraction* smell. If two analysis classes have identical dependencies, we determine it as the occurrence of the *Duplicated Abstraction* smell.

Refactoring: The analysis developer can refactor the bad smell *Duplicated Abstraction* by moving the dependencies into one class (cf. Refactorings in Section 3.3.3). The simplest way to move all dependencies is to merge all affected classes into one class. Alternatively, they can split the analysis classes so that the dependencies on the language types are clearly separated. After splitting the class, each dependency is located in a dedicated class. The analysis developer has to consider whether to merge the classes with the dependencies or split the classes. The analysis developer has to decide how to refactor this bad smell.

Limitations: Suppose the analysis developer merges the analysis classes. In that case, the result can produce the bad smell *Rebellious Modularity* (cf. Section 4.2.4.4) or if they split classes so that each has a dependency on one language type they produce the bad smell *Degraded Modularity* (cf. Section 4.2.4.2). An automated refactoring or a clear identification of the bad smell *Duplicated Abstraction* is not possible. The analysis developer has to decide which occurrence to refactor, and they have to remember that the refactoring can result in other bad smells. However, finding occurrences of the *Duplicated Abstraction* smell can also help the analysis developer find duplicated code and behaviour within one model-based analysis. Even if they do not refactor the *Duplicated Abstraction* smell, the analysis developer can find classes that represent similar behaviour; thus, they are less likely to miss classes during a change of the behaviour if it occurs in multiple classes.

4.2.1.2 Missing Abstraction

The DSML introduces types to the model that the model-based analysis then can use to perform an analysis; however, when the language developers and analysis developers do not work together to coordinate the development of the DSML and the model-based analysis, the types of the DSML become incomplete. We determine a type as incomplete when the analysis developer requires the type to have more attributes than it actually has. A hypothetical example of an incomplete type is a delay type in a simulation that does only contain the delay time in milliseconds. When the model-based analysis needs the delay in another unit than milliseconds, the type could contain a transformation from milliseconds into the desired unit. When the delay type does not contain such a transformation, every time the delay type occurs, the analysis developer needs to transform the type manually if the unit of the delay type changes, every occurrence of the transformation must be adapted by the analysis developer. Suppose analysis developers cannot introduce new types to the DSML but must represent certain concepts. In that case, they can either introduce new types in the context of the model-based analysis or use primitive types to represent concepts of the analysis. The *Missing Abstraction* smell deals with these primitive types. For example, in the Palladio Simulator (cf. Chapter 2), the *delay* in the simulation is represented by a primitive type (i. e., integer) instead of a type in the DSML.

```
scheduleDelay(Event event, int delay, int iteration, int participants) {
  while(isValid(delay))
    addParticipants(participants);
    waitDelay(delay);
    iteration++;
    event.fire();
}
```

Listing 4.1: Using a Primitive Type (delay)

Listing 4.1 shows a simple example referring to the Palladio Simulator example. The second parameter of the method *scheduleDelay* has the primitive type integer. The delay is used in the *isValid* and the *waitDelay* methods.

Effect: If a concept is represented by a primitive type (e. g., integer, double, or string), the comprehensibility of the source code gets worse. For example, when an analysis developer must invoke a method with multiple parameters of the same type, they have to consult the documentation or, when possible, the method itself to determine which parameter represents what. Access to the invoked method is only possible when the source code is available. Modern IDEs assist the analysis developer in identifying the correct parameter; however, an IDE also needs access to the source code to aid the analysis developer. In the Palladio Simulator example, the analysis developer can introduce bugs by accidentally setting the parameters in the wrong order if a method requires the delay and some other integer parameters. Modern IDEs provide some assistance to avoid such errors, but current tools cannot distinguish the parameters by their semantics. As a result of the *Missing Abstraction* smell, analysis developers require more effort to understand the code base of a model-based analysis. Changing concepts represented by primitive types are elaborate when the concepts are used throughout the model-based analysis. The analysis developer has to trace each occurrence of a variable with the primitive type. Even with the aid of modern IDEs, the analysis developer must change each occurrence manually.

Causes: Introducing new types, especially at the beginning of the development, is reasonable to avoid introducing dedicated types. If it is unclear whether a type is used in a broader scope or has just a single use, the effort to introduce new types is not justifiable. Therefore, it is not uncommon to use primitive types in the beginning. Changing a primitive type, for example, from an integer to a floating point, in the context of one class or even one method requires low effort; adapting a dedicated type, on the other hand, is more complex. The analysis developer must coordinate with the language developer the extension of the DSML. However, the technical debt increases when such a concept is represented by a primitive type and the concept is used on multiple occasions in the model-based analysis. Using a primitive type is quick to implement, with the cost that method signatures can become too complex.

Identification: To identify the *Missing Abstraction* smell, the occurrence of a primitive type is the first indicator. Such a primitive type occurs in a method signature and the method itself. The identification is based on the primitive types the language supports; therefore, the primitive types of the language must be identified. Marking every occurrence of a primitive type would create too many false positives, as an auxiliary variable would also be part of the result. Therefore, we need a more elaborate process to identify the *Missing Abstraction* smell. An indicator for the *Missing Abstraction* smell is when a variable can be traced through different methods / analysis classes, and it is advisable to use a dedicated type. Alternatively, variables that have the same name but occur in different classes and methods are also an indicator of this bad smell.

Refactoring: Fixing this bad smell requires introducing a dedicated type that represents the concept that is represented by a primitive type. In the context of the Palladio Simulator, the analysis developer creates a class that represents the concept of a delay. The class must contain a constructor that requires the primitive type as an input. If the concept is also part of the DSML, the language architect also introduces the new type to the DSML, either as part of a feature or, if the concept is more complex, as a new feature. If necessary, the analysis developer adds behaviour to the new class. For example, determining the delay regarding the resource (i. e., CPU or HDD) which causes the delay. After the new type is created, all occurrences of the primitive type must be replaced by the new type.

Limitations: Every occurrence of a primitive type can indicate a *Missing Abstraction* smell. To identify the *Missing Abstraction* smell, manual interaction by the analysis developer is required. The analysis developer must identify whether a primitive type that is passed through classes and methods represents a concept. Also, just because variables share the same name does not mean they represent the same concept. Therefore, we need the analysis developer to decide whether the variables that share the same name represent the same concept.

4.2.1.3 Unused Abstraction

In the development of DSMLs and corresponding model-based analyses, especially in historically-grown systems, some language types are not or no longer used. The *Unused Abstraction* smell corresponds to the dead code bad smell. The dead code bad smell from object-oriented design is a widespread problem in object-oriented software [Cai+21]. We assume each available type in a DSML impacts the analysis result when modelled. This assumption means the tool user has the guarantee to model all types with the assumption that the modelled elements impact the analysis outcome. If the assumption does not hold, in the best case, the modelled types do not affect the result or the interpretation of the results. However, in the worst case, the tool user interprets the results involving unused types. As a result, they can conclude that the modelled types affect the result, which can lead to wrong conclusions.

Effect: The negative effect on the assumptions and the conclusions is not the only effect the *Unused Abstraction* smell can have. A single unused language type can drastically impact the interpretation of the results and the complexity of the created models. The tool user has no idea that a certain type has no effect on the analysis, so the model they create can contain unnecessary model elements. Also, the evolvability or reusability of a model-based analysis is affected, as the impression can arise that the type must be used in the analysis or that the type is somewhere used. Furthermore, when the number of unused types grows, the features of a DSML contain more types than necessary, which makes the features convoluted. As a result, the analysis developer can assume usage of a type in the model-based analysis, which can increase the effort required to fix bugs or extend the model-based analysis.

Causes: During the lifetime of a DSML, due to changing requirements, features are added or removed. Ideally, the DSML and its corresponding model-based analysis evolve together. However, when features are no longer needed, they are no longer maintained and are eventually removed from the latest version of the model-based analysis. Due to the size difference of DSMLs and model-based analyses, DSMLs are usually smaller; changing a DSML compared to a corresponding model-based analysis requires less effort. The complexity of a DSML does increase when a few types are added; the PCM, for example, has 209 types [HSR19]. Compared to the 200k lines of code of its corresponding model-based analysis, changing the DSML requires comparably less effort. On the other hand, the corresponding model-based analysis feature requires more maintenance effort. Nevertheless, removing unused types in a DSML will improve its complexity. In the context of the Palladio, for example, the PCM and its analyses have many features (cf. Chapter 6 and [HSR19]). These features depend on a certain Java version and a certain version of the Eclipse IDE. At some point, some features were no longer needed or required too much effort to keep them up-to-date. As a result, these features were no longer added to the newest version of the Palladio Simulator. If the features added new types to the PCM, they remained in the DSML, eventually resulting in unused types.

Identification: In order to identify occurrences of the *Unused Abstraction* smell, the analysis developer must analyse the dependencies of the model-based analysis on the DSML.

Each language type without an incoming dependency from the model-based analysis can be considered unused. If multiple model-based analyses are associated with one DSML, we recommend using our reference architecture for model-based analyses (cf. Chapter 3). According to our reference architecture, the model-based analysis is modularised according to the language features of the DSML. Due to our feature-based approach, each model-based analysis corresponds to one feature configuration. A feature configuration is a valid subset of all available features (cf. Section 3.3.1). If we want to find occurrences of the *Unused Abstraction* smell considering all model-based analyses, we analyse whether all types of a feature configuration are used in the model-based analysis. This is necessary because if we use all features instead of a feature configuration, all features not part of the configuration automatically correspond to the *Unused Abstraction* smell, as they are not used in the configuration.

Refactoring: If a language type has no incoming dependencies, the language architect can remove the language type from the DSML. If all types of a language feature are not used by the model-based analysis, the language architect can also consider removing the whole feature. However, if the DSML is developed according to the reference architecture for DSMLs by Heinrich et al. [HSR19], the language feature can remain in the feature structure of the DSML. The modular structure and the extension mechanisms of the reference architecture for DSMLs allow removing features for a certain configuration. This is relevant when multiple model-based analyses use the DSML, but not each model-based analysis requires the feature. Removing types from the DSML affects the model instances; the tool user must change their models if the unused types are used in an instance.

Limitations: The refactoring of the *Unused Abstraction* smell, i. e., removing the unused language type, can have negative effects if the DSML has a monolithic structure. If the type is part of a cycle or other types have dependencies on the unused type, further changes can be required. Also, deleting a language type could render the models and tools like editors of the DSML useless. Therefore, we recommend applying the reference architecture for DSMLs before making any deletions. When the reference architecture for DSMLs and the reference architecture for model-based analyses are applied, the DSML and the model-based analysis are cycle free, and as a result, implementing changes is more predictable.

4.2.2 Encapsulation

The process of encapsulation involves modularising the components of an abstraction that are responsible for the behaviour and structure. The primary role of the encapsulation is to maintain the separation between an abstraction's interface and its implementation.

4.2.2.1 Deficient Encapsulation

Historically grown DSMLs and model-based analyses will change during their lifespan; otherwise, their quality declines [Leh80]. Adding new features is one cause for such changes. Existing features or types could sometimes be merged as they represent the same concept. For example, the network communication part of a software and hardware simulation was initially implemented as a static delay. In order to add more functionality, the ISO/OSI stack gets added to the DSML, and the model-based analysis. The static delay

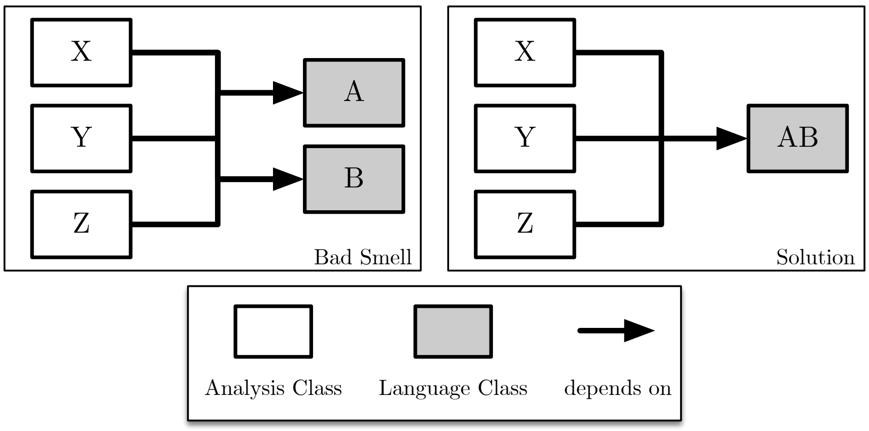


Figure 4.3.: Deficient Encapsulation

is no longer used in the analysis; even if a static delay is modelled, the analysis maps it onto the ISO / OSI stack analysis. As a result, both features (i. e., static delay for network communication and the ISO / OSI stack) are always used together. When language types are always used together, we determine it as *Deficient Encapsulation* smell. Figure 4.3 depicts the bad smell and its solution. For the bad smell example on the left side, the analysis classes *X*, *Y*, and *Z* depend on the same language classes *A* and *B*. These same language classes The solution in Figure 4.3 depicts the same analysis classes *X*, *Y*, and *Z*, however, now they depend on a single, new language class *AB*. In the refactoring part of this section, we describe how to achieve the solution to the *Deficient Encapsulation* smell.

Effect: The *Deficient Encapsulation* smell indicates an unnecessarily high number of language types in the DSML size is bigger than it needs to be. The size of the DSML and the model-based analysis increases, and the number of types that should be used together is also increased. The size of the DSML is bigger because of unnecessary language types, and the size of the model-based analysis is bigger because of the additional dependencies on the language types. Thus, the complexity of both the DSML and the model-based analysis also bigger, as the analysis developer must know the types, their interaction with each other, and their dependencies. If the analysis developer lacks this knowledge, the potential for errors and bugs increases, as they might use types other than the language developer intended. In the example of Figure 4.3, the analysis classes *A* and *B* are interchangeable. Depending on which language type is present in the model, the analysis developer has to cover all cases, resulting in duplicated analysis code. Due to the increasing complexity, the effort to implement or change new features increases.

Causes: During the lifetime of a model-based analysis, due to changing requirements, new language types are added to the DSML. These language types can require that they are only used together with other, maybe already existing, language types. Another cause is that the language architect anticipates that the language types are used individually, but the model-based analysis was no use-case for the individual language types. Furthermore, to avoid the *Missing Abstraction* smell (cf. Section 4.2.1.2), the analysis developer is encouraged to introduce more language types to the model-based analysis to avoid using primitive types. The result is a set of language types that are always used together. Another possible cause is that the use of language types in the model-based analysis is unclear. This can happen when the DSML is developed independently of the model-based analysis, for example, by another team or company.

Identification: A first indicator for the *Deficient Encapsulation* smell is that analysis classes that depend on one type *A* also always depend on type *B*. Therefore, to identify such an occurrence, we create pairs of each type and compare these pairs to the total number of occurrences of one type. For example, when type *A* occurs five times and the pair of type *A* and *B* also occurs five times, we have a positive match. To further investigate the usage of both types, we analyse each class that contains the pair of type *A* and *B*. We check each method signature for the usage of both types. If the result is that both types always occur in pairs of the signature, we determine it as *Deficient Encapsulation*.

Refactoring: In order to fix this bad smell, the language developer introduces a new language type; in the case of our example depicted in Figure 4.3, they create the language class *AB*. After creating the language class *AB*, the language developer can choose whether they move all characteristics (i. e., attributes, relations) of the affected types *A* and *B* to the new language type *AB*. Alternatively, they move only the attributes and relations to the new language class that creates dependencies. Changing the DSML could invalidate corresponding model-based analysis or editors that are based on the DSML. Therefore, instead of altering the DSML, the analysis developer can introduce a wrapper in the model-based analysis that encapsulates both types that form the bad smell. The result is that the analysis classes *X*, *Y*, and *Z* only depend on one language class *AB*.

Limitations: If adapting dependent model-based analyses and editors requires too much effort, fixing the *Deficient Encapsulation* smell is a trade-off decision. The language architect and the analysis architect must decide whether or not fixing it is worth increasing the technical debt. The more the doubled language types are used, the more complex it becomes to distinguish the language types. The example in Figure 4.7 shows only two redundant language types; however, if the number of these redundant types grows, the more complex the DSML and the model-based analysis becomes. Furthermore, if changes to the language would break existing model-based analyses or another tooling like editors, it also affects the decision-making regarding changing the DSML.

4.2.3 Hierarchy

A ranking or ordering of abstractions is what we mean when we talk about hierarchies. The bad smells *Folded Hierarchy* and *Unexploited Hierarchy* can be detected when the model-based analysis is developed according to our reference architecture for model-based analyses. In Chapter 3, we introduce our reference architecture for model-based analyses, and in Section 3.3.1.4, and Section 3.3.2 we explain the layering of our reference architecture for model-based analyses.

4.2.3.1 Folded Hierarchy

According to Heinrich et al. [HSR19] and our reference architecture for model-based analyses, language features, analysis features, language components, and analysis components are separated into layers, and each feature and each component is located on a single layer. More details regarding layering of model-based analyses are presented in Section 3.3.2. An analysis component can have dependencies on a language component of the same layer or of a more generic layer. Both are possible; we distinguish strict layering (i. e., dependencies

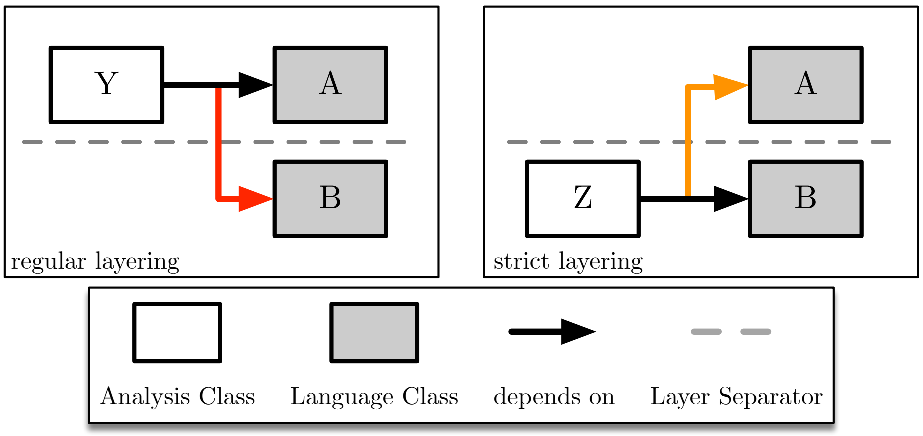


Figure 4.4.: Folded Hierarchy

are allowed only on language components of the same layer) and regular layering (i. e., dependencies on language component of the same or a more generic layer). Figure 4.4 depicts violations of the strict (orange, *Z* to *A*) and the regular (red, *Y* to *B*) layering. The analysis class *Y* and the language class *A* are located on a more generic layer. The analysis class *Z* and the language class *B* are on a more specific layer. The dependency from *Z* to *A* is a violation when strict layering is applied. If we apply regular layering, this dependency is allowed. The dependency from *Y* to *B* is not allowed in both.

Effect: We differentiate the effects of strict and regular layering. Strict layering has a clear dependency structure that allows the analysis developer to place and locate the usage of language classes in the model-based analysis. When ignoring the rule of strict layering, the model-based analysis is harder to comprehend, as its structure diverges from the structure of the DSML. Violating the rule that the dependency of a generic on a specific layer is forbidden, the more generic layer is no longer independent of, the more specific layer. Ergo, the layers cannot be reused independently of each other. The *Folded Hierarchy* smell enables dependency cycles between layers when the *Weakened Modularity* smell (cf. Section 4.2.4.5) is also present. As a result, it impedes the maintainability of the model-based analysis (cf. Section 3.3).

Causes: If the DSML and the model-based analysis are developed independently, the language architect cannot anticipate how language classes are used. As a result, they can introduce specialised types that might be needed by an analysis class on a more generic layer. If the DSML does not add such a generic type, the analysis developer cannot comply with the regular layering. Then, the specialised analysis class must depend on the language type on the more specialised layer. Violations of either strict or regular layering can originate in the lack of understanding on the part of the analysis developer regarding the DSML. In the modular version of the PCM [SHR18], some language types on different layers have the same name due to inconsistent naming. This resulted in occurrences of the *Folded Hierarchy* smell, and the analysis developer has to guess which type to use.

Identification: In order to identify violations of the *Folded Hierarchy* smell, each incoming dependency of a language class is analysed. Only incoming dependencies originating from analysis classes are relevant; thus, all other incoming dependencies are discarded. If the dependency occurs between classes on the same layer, no violation is detected, and the dependency is discarded. However, suppose the dependency occurs from an analysis class located on a layer that is more specific on a language class located on a more generic layer. In that case, the strict layering is violated (cf. Figure 4.4, orange dependency). These occurrences are categorised and collected as dependencies that violate strict layering. If the violations of the strict layering are irrelevant, these dependencies also get discarded. The category of layering violations that occur from an analysis class located on a more generic layer, on a language class that is more specific, is always accounted to the *Folded Hierarchy* smell (cf. Figure 4.4, red dependency).

Refactoring: In order to refactor the *Folded Hierarchy* smell, the analysis developer collects the dependencies of the analysis classes on the language classes that break the regular and the strict layering rule. For each violation of the layering rules, the analysis developer must adapt the dependencies so that they no longer violate the layering rules. If, for example, the red dependency in Figure 4.4 is the only dependency of *Y*, the analysis developer can move the analysis class *Y* to the same layer as the language class *B*. If an analysis class has dependencies on multiple layers, the affected fields and methods with dependencies on the same layer are extracted into a new analysis class. The dependencies of the analysis class *Z* in Figure 4.4 prevent that the analysis developer can move *Z* on the same layer as language class *A*, as they would create another layer violation. Therefore, the analysis developer creates a new analysis class on the same layer as the language class *A*. Then, they identify the attributes and methods in the analysis class *Z* that depend on the language class *A*. After identifying the attributes and methods, they move them into the newly created analysis class. However, if the analysis class cannot be split, and it is a strict layer violation, the language architect has the option to introduce a new language type on the required layer. In our example, the language architect introduces a language class *C* on the same layer as *B*. Then, *C* inherits from language class *A* and the orange dependency is replaced by a dependency on *C*. Figure 4.5 depicts the refactoring by inheritance; introducing a new language class *C* to fix the dependency error.

Limitations: Strict layering is not always possible, for example, when an analysis algorithm requires a language type of a more generic layer. In the refactoring section, we

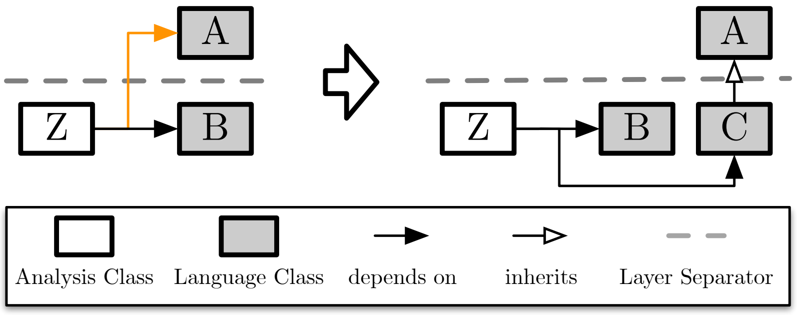


Figure 4.5.: Folded Hierarchy – Refactoring by Inheritance

proposed that the language architect introduces a new type on the required layer. In our example, the language architect had to introduce a new language class *C* on the same layer as *B*. Then, they made language class *C* inhering from language class *A* and replaced the orange dependency (*Z* on *A*) with a dependency on *C*. However, if the DSML is not changeable, for example, when it implements a standard (cf. the BPMN2 metamodel [HSR19]), introducing new types is impossible.

4.2.3.2 Missing Hierarchy

This bad smell originates from the object-oriented bad smell *Switch Statements*. Switch statements indicate a lack of polymorphism in the object-oriented design. However, the excessive use of language types in switch statements indicates that polymorphism is missing in the DSML design. The switch statements in the model-based analysis indicate missing subtypes in the DSML.

Effect: Maintaining switch statements requires more effort than the maintenance of a type hierarchy. Each new case must be changed in the switch statements, resulting in more changes as switch statements exist. Thus, the model-based analysis is harder to evolve and reuse. Switch statements also mask possible polymorphism, which can also affect the time to develop new features or maintain existing features (cf. Repeated Switches in [Fow18]). Furthermore, differentiating language types based on their attributes is hard to follow, as the analysis developer has to understand the effect the state of the attribute has on the analysis. Therefore, such constructs are prone to errors, and changes to the semantics of the attributes can affect each switch statement that handles such an attribute.

Causes: If the DSML does not provide subtypes, switch statements are a fast solution to differentiate states of a language type. Needing a fast solution also can result in implementing switch statements instead of changing the DSML. As changing the DSML could affect other dependent model-based analyses and editors. Such changes would initially require more effort than using a switch statement. Furthermore, changing the DSML would create inconsistencies; for example, when it implements a standard (cf. the BPMN2 metamodel [HSR19]), introducing new types to the DSML, would make it inconsistent to the standard. Diverging from a DSML standard would render the tooling incompatible with models that are developed with another language that follows the standard. So, if an analysis developer wants to use the DSML that complies with the standard, they must use helper constructs like the switch statement to cope with this disadvantage of the DSML.

Identification: In order to identify the *Missing Hierarchy* smell, each analysis class in the model-based analysis is analysed. If the analysis class that contains a switch statement contains a dependency on a language type, the analysis class is marked for further analysis. If the switch statement has a language type as a condition, we determine it as an occurrence of the *Missing Hierarchy* smell. According to Fowler et al. [Fow18], in their revision of the object-oriented bad smells, only some occurrences of a switch statement are problematic. However, if in the model-based analysis the analysis developer has two distinct cases of a language type, the switch statement makes the analysis code convoluted; therefore, we advocate marking every switch statement that meets our conditions for the *Missing Hierarchy* smell.

Refactoring: We define two approaches to refactor the *Missing Hierarchy* smell. The first approach introduces subtypes and uses dynamic dispatch to replace the switch statements. The method call replaces the switch statement itself, and each case is implemented as a method that replaces the case. The second approach is implementing the visitor pattern to replace the switch statement. If the states of the language types are unrelated and new operations are needed frequently, it is highly inconvenient for developers to implement a new subclass for each new operation.

Limitations: As mentioned in the *Causes* section of this smell, it is not always possible to introduce new types to the DSML. Either because the DSML is not open source and, therefore, cannot be changed or the DSML is implemented according to a standard and changing it would require changing either the standard or models that follow the standard cannot be analysed.

4.2.3.3 Unexploited Hierarchy

We assume that the model-based analysis is developed according to our reference architecture for model-based analyses model-based analysis (cf. Chapter 3). The model-based analysis has distinct layers, and each analysis component of the model-based analysis is located in one of these layers. According to our reference architecture for model-based analyses, dependencies between analysis components are allowed when the analysis components are located on the same layer. A dependency from an analysis component to an analysis component on a more generic layer is also allowed. A dependency from an analysis component to an analysis component on a more specific layer is forbidden. If an analysis component has a dependency on a more generic layer that is not adjacent, it is only a problem when the layering is strict. The left side in Figure 4.6 depicts a valid dependency structure with two layers. One class is located in a generic (top layer), and two classes are located in a specialised layer (bottom layer). The right side depicts a three-layered architecture. The orange arrow (*A* to *B*) indicates a layer violation because a layer in between is omitted. It is only a problem when the layered architecture is intended

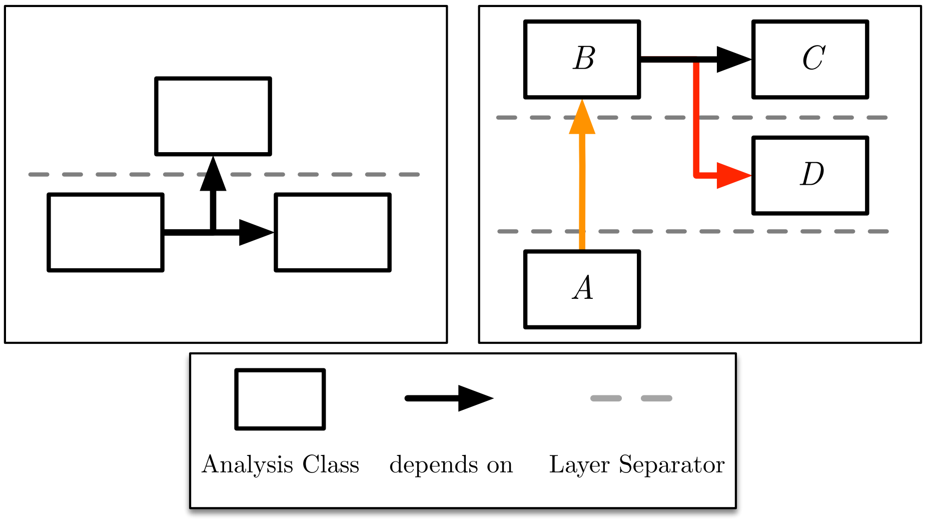


Figure 4.6.: Unexploited Hierarchy

to be strict. The red arrow (*B* to *D*) indicates a layer violation because a generic class depends on a more specialised class.

Effect: We differentiate the effects of strict and normal layering. Strict layering has a clear dependency structure that allows the analysis developer to exchange components within a layer without the need to modify components on a more generic layer. When ignoring the rule of strict layering, the model-based analysis is harder to comprehend. When a dependency skips a layer, for example, to improve the performance, it increases the coupling of the model-based analysis. When an analysis class depends on an analysis class on a more specific layer, the more generic layer is no longer independent of, the more specific layer. Ergo, the layers cannot be reused independently of each other. It also enables dependency cycles between layers, which impedes the maintainability of the model-based analysis (cf. Section 3.3).

Causes: A lack of understanding of the layers of the model-based analysis by the analysis developer can result in component misplacement in the architecture. Or, if analysis developers want to improve the performance, they let dependencies skip layers and, thus, violate the strict layering. Another case that generates layer violations is when analysis developers remove a component, which can result in a skipped layer. The removed component is part of a chain of dependencies, for example, the second class in the middle of the chain. When the second class is removed, the third class that depended on the second one now needs a replacement. A new class is introduced on the same layer as the second class, which would have no effect, or the second class is replaced by a dependency from the third to the first class.

Identification: Identifying if an analysis component is placed in the right layer is only possible when a mapping to a layer is documented. For example, annotating each component with its designated layer or creating a map that documents the components and their layers. With this mapping, each dependency has to be analysed. Dependencies within a layer are discarded. Then, starting from the most generic layer, each remaining dependency is checked. The most generic layer should have no remaining dependencies; however, if a dependency remains, it automatically violates the layering rules, as the most generic layer must not have dependencies on other layers. For the remaining layers, each dependency is analysed to determine whether it points only to a more generic layer and, if necessary if it points only to an adjacent layer. Checking whether the dependencies point to a more generic layer is unnecessary for the most generic layer. These dependencies are not part of the result, and the remaining dependencies are violations of the *Unexploited Hierarchy* smell.

Refactoring: The refactoring of a model-based analysis, especially introducing layers and fixing layer violations, is presented in Section 3.3.3. When a dependency of an analysis class points to a more specific analysis class, we define the dependency inversion refactoring to fix it. The dependency inversion refactoring is divided in *Dependency Inversion by Inheritance* (cf. Figure 3.11), *Dependency Inversion by Reference* (cf. Figure 3.12), *Dependency Inversion by Bidirectional Reference* and *by Containment* (cf. Figure 3.13). In this chapter, we will not provide further details regarding the refactoring of model-based analysis; please go to Chapter 3 for more details.

Limitations: One disadvantage of a layered approach is that the code required to route and manipulate data across a layer might slow down the performance of the model-based analysis. This is especially noticeable in portions of the model-based analysis where the data would be better suited structurally in deeper layers than in the layers they are authorised to access. Reports offering aggregated data, such as totals or averages, are prominent instances of this, as data aggregated on a generic layer could be moved up several layers without any modification before it is printed. If the performance of the model-based analysis is crucial, the benefits of the layered approach are neglected in the aforementioned application scenarios in favour of faster execution.

4.2.4 Modularity

The capability of a system to be decomposed into a collection of self-contained and loosely coupled modules is referred to as its modularity. The loosely coupled modules of a model-based analysis allows the developer to change individual modules without changing other modules. Also, a clear dependency structure of the modules, like the reference architecture for model-based analysis improves the evolvability and reusability of model-based analyses (cf. Chapter 3).

4.2.4.1 Broken Modularity

The DSML should have no dependencies on the tool in which it is used, as a DSML can be used by different model-based analyses. Each of these model-based analyses could analyse different quality properties; see, for example, the Karlsruhe Architecture Maintainability Prediction (KAMP) methodology [HBK18]. The Palladio-Simulator utilises the PCM to analyse the performance of software systems based on an architectural model. The PCM is also utilised by the KAMP approach to analyse the impact of changes in the domain of software systems and business processes. To avoid that changes in the model-based analysis affecting the DSML, we forbid dependencies from the DSML to its corresponding model-based analyses. If a dependency of the DSML to the model-based analysis occurs, we call it the *Broken Modularity* smell.

Effect: Suppose the DSML would know the model-based analyses, i. e. has dependencies on its tooling, all model-based analyses that use the DSML have to deal with these unnecessary dependencies. Changes to one model-based analysis can result in changes of the DSML, which in return can result in changes to other corresponding model-based analyses. Such dependencies increase the coupling of the DSML and its corresponding model-based analyses, and as a result, they are more difficult to maintain, evolve or reuse. It also allows dependency cycles between the DSML and the model-based analysis, resulting in even more maintenance, evolvability, and reusability difficulties.

Causes: Adding features that should be located in the model-based analysis instead of the DSML can happen if the language architect adds analysis concerns of the DSML. For example, when the language architect adds the type of simulation (continuous, discrete, event-based; cf. Chapter 2) to the DSML, the model instances contain this simulation-specific information in their model. Simulations are a subset of analyses that examine a system over time. They are used when experimenting with the real system is too time-consuming, costly, dangerous or simply impossible because the system does not exist (yet). Analyses that use no simulation approaches have to deal with model elements representing the kind of simulation the modeller intended. In the best case, the elements can be ignored by the analysis. However, if the analysis developer has no idea how to interpret model elements that are not part of the DSML, they could use such elements in their analysis. Even if during the analysis only the presence of the unused model elements is checked, a change to the DSML that modifies these elements would also require a change in the model-based analysis. A change that could have been avoided if the elements were at the right place (i. e. part of the model-based analysis instead of the DSML) from the beginning.

Identification: In order to detect the *Broken Modularity* smell, the outgoing dependencies of the DSML must be analysed. Therefore, we collect all dependencies of the DSML and discard all incoming dependencies, as tools like editors and model-based analyses require the DSML either to model or to analyse model instances. Also, we discard all internal dependencies of the DSML, for example, a dependency that points to a type of the DSML. The dependencies that remain are dependencies on external tools and libraries. The last step is to remove all dependencies on libraries that are required for the DSML; for example, if the DSML was created with the EMF, we discard these dependencies. Dependencies on the standard Java library or other DSMLs are also required and, thus, discarded. What remains are the outgoing dependencies on tools like editors and model-based analysis. We regard the remaining dependencies as the *Broken Modularity* smell.

Refactoring: Fixing this smell requires the language architect and the analysis architect to work together. The knowledge of both, about the domain of the DSML and the model-based analysis, is required to identify and fix wrongly placed language features in the DSML. For each outgoing dependency that is marked as *Broken Modularity* smell, the architects must decide whether the whole feature or only the types that create the dependency must be moved to the model-based analysis. If the types of the DSML can be moved to the model-based analysis, the language types are removed from the feature of DSML and added to the corresponding feature in the model-based analysis. For the case that the model-based analysis does not have such a feature, the analysis architect introduces a corresponding feature in the model-based analysis. Alternatively, if multiple language types must be moved to the model-based analysis, it could be sufficient to move the whole feature to the model-based analysis. For example, if the simulation types are part of a feature with the same name. The analysis architect creates a feature with the same name, and the analysis component developer implements the corresponding components (cf. Chapter 3).

Limitations: Identification of the *Broken Modularity* smell is only possible when the source code of the DSML is accessible, and the same is true for changes that affect the DSML. Even if the DSML is accessible, if the DSML implements a standard, for example, the BPMN2 standard, the language architect might want to avoid changing the DSML. Changing the DSML means that it no longer conforms to the standard. Ergo, model-based analyses that expect a model that conforms to the standard can no longer use instances of the changed DSML. The same goes for each tool that is based on the standard DSML. Another problem when fixing the *Broken Modularity* smell is that the developers must change each corresponding tool of the DSML. In the long term, we recommend fixing this bad smell, as it creates unpredictable changes to the DSML and its corresponding tooling; however, the effort that comes with fixing the bad smell should be considered and planned for each corresponding tool. Otherwise, each tooling is only usable once the smell is fixed.

4.2.4.2 Degraded Modularity

If a language component is used by multiple analysis components, the analysis developer has to make many changes to many different analysis components when the language component is modified. The *Degraded Modularity* is inspired by the Shotgun Surgery smell [Fow18]. A single responsibility, in the case of the DSML, a language feature, is split up among analysis components. Figure 4.7 depicts the *Degraded Modularity* smell. It shows three analysis components that depend on one language component, such a dependency results from dependencies of analysis classes that are part of an analysis component. An analysis component can have multiple classes that depend on a language component.

Effect: Due to the scattering of language components all over analysis components, the code is difficult to comprehend. The three analysis components shown in Figure 4.7 could implement the same analysis algorithm of the language component, or each analysis component implements a different variant of the same analysis algorithm, or each analysis component implements a different analysis algorithm. In order to understand how these analysis components work together or how they work, the analysis developer has to consult the documentation of each analysis component, which makes it time-consuming and difficult to comprehend. Another effect is that changes to one language component

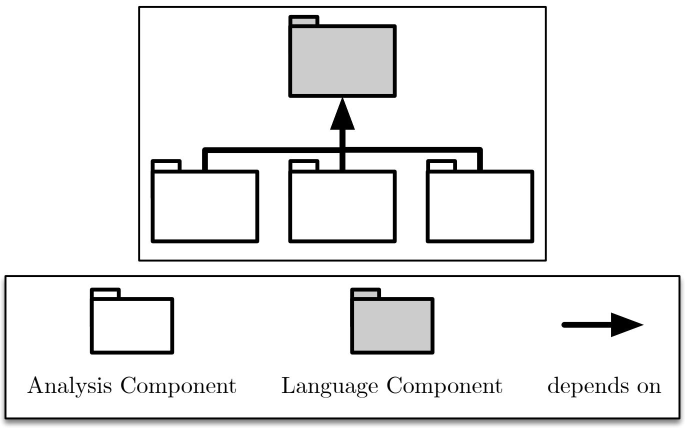


Figure 4.7.: Degraded Modularity

can require that the analysis developer must change multiple analysis components; hence, the similarity to the bad smell *Shotgun Surgery* (cf. Chapter 2). The coupling between the analysis components and the language component is increased, and if the analysis components also depend on each other, reusing a single analysis components is difficult. Furthermore, the increased coupling hampers the evolvability of the model-based analysis, as changes to one of the analysis components can lead to changes in the remaining analysis components.

Causes: During the lifetime of a model-based analysis, analysis developers add and change analysis components of the analysis. A language component that was used by a single analysis component could then add to other analysis components. Such errors happen easily without restrictions that give a warning or prevent the analysis developer from adding the dependency. If the analysis developer must implement a new analysis, they must know every analysis component and their intent. Only then can they determine whether the new analysis must be part of an already existing analysis component or whether they must implement a new one. Even if the analysis developer is aware that an analysis component already exists, they could decide to implement a new analysis component to avoid understanding the existing code and implement the analysis code faster. Another cause for the *Degraded Modularity* smell is cross-cutting concerns of language components. If a language component is required in a variety of analysis components, it is better to split the language component up instead of consolidating each dependency in one language component. This is only possible when the analysis components depend on different classes of the language component. If the analysis component represents multiple concerns, it is more difficult to comprehend, evolve, and reuse due to its many dependencies on other language components and other analysis components. It is difficult to reuse an analysis component that implements different concerns. If an analysis component implements multiple concerns, for example, if an analysis component determines a system's performance and reliability analysis, reusing the performance part requires reusing the reliability part also. The analysis component needs information about the system's performance part (e. g. , throughput, processing time) and about the reliability (e. g. , mean time to failure). These requirements create either dependencies on other analysis components or, if the analysis developer decides to contain them all in one analysis component, the analysis component gets too complex.

Identification: To identify whether a language component is used in multiple analysis components, we analyse all incoming dependencies of a language component. A dependency is an incoming dependency when it points from an analysis component to the analysed language component. To identify incoming dependencies, we determine an language component and search all analysis components for dependencies on the respective language component. If the language component has incoming dependencies of multiple analysis components, we determine it as *Degraded Modularity* smell. For example, if the DSML and the model-based analysis are layered according to our reference architecture for model-based analyses (cf. Chapter 3), we analyse and compare only the incoming dependencies of one layer at a time. Due to the modularisation of the analysis, language components are needed on different layers; therefore, we count only incoming dependencies of the same layer. Ergo, in a layered architecture, a language component can have incoming dependencies from different layers. If an incoming dependency originates from a more specialised layer, it corresponds to the *Folded Hierarchy* smell (cf. Section 4.2.3.1).

Refactoring: If the affected analysis components do not depend on other language components, merging these components will fix this bad smell, as all dependencies on the language component now come from a single analysis component. However, if these components represent different concerns (i. e., have dependencies on other language components), merging these components would create the bad smell *Rebellious Modularity* (cf. Section 4.2.4.4). Alternatively, the analysis developer can move the classes of the analysis component that depend on the language component. These classes can either be moved to a new analysis component or to an analysis component with dependencies on the language component. However, even a class could contain dependencies on multiple language components; therefore, the analysis developer can move the affected fields and methods to fix this bad smell. Another way to fix the *Degraded Modularity* smell is to split the language component up. Heinrich et al. [HSR19] introduced refactoring operations for splitting a language class or language component up. The goal is to group the language types by incoming dependencies from their corresponding model-based analysis. The result in the case of our example shown in Figure 4.7, the number of language components is increased from one to three, and each analysis component depends on one, different language component.

Limitations: Identifying the *Degraded Modularity* smell does not always require access to the DSML. If the DSML implements a standard, for example, the BPMN2 standard, the language architect might want to avoid changing the DSML. Therefore, they can choose to merge the affected analysis components; however, they must be aware that merging the analysis components can create the *Rebellious Modularity* smell (cf. Section 4.2.4.4). If the language architect must change the DSML and the DSML is developed according to a standard, changing the DSML means it no longer conforms to the standard. Ergo, model-based analyses that expect a model conforming to the standard can no longer use instances of the changed DSML. The same goes for each tool that is based on the standard

DSML. Another problem when fixing the *Degraded Modularity* smell is that the developers must change each corresponding tool of the DSML.

4.2.4.3 Missing Modularity

This bad smell heavily refers to our reference architecture for model-based analyses (cf. Chapter 3). Our reference architecture requires a layered structure of the model-based analysis, similar to the reference architecture for DSMLs [HSR19]. The layering helps the analysis developer to locate analysis components and reuse components. The layering also helps to avoid cycles of the analysis components (cf. Section 4.2.3.3) and cycles of analysis components and language components (cf. Section 4.2.3.1). The *Missing Modularity* smell is only applicable when the model-based analysis should have layers because then when the model-based analysis does not contain any or just one layer, we determine it as *Missing Modularity* smell. Having no layer or one layer is the same, cf. Section 3.3.2.

Effect: Due to the missing layers, it is harder to identify the role of an analysis component in the context of the model-based analysis. The model-based analysis is missing a clear structure that allows the analysis developer to locate concerns and, if necessary, make changes or add new features without understanding each analysis component in the analysis. If, for example, an analysis component is located on the domain layer (cf. Section 3.3.2.2) of the model-based analysis, the analysis component represents some form of domain-relevant information. If the analysis developer wants to add new domain-specific behaviour, they only have to consider the analysis components on the respective layer for the desired extension. However, if such a structure is missing, the analysis developer has to search the source code of the analysis component to determine its concern in the model-based analysis. Also, the layered structure provides a clear dependency structure: only cross-layer dependencies from a layer to a more generic layer are allowed. This allows layers to be exchanged without affecting more generic layers.

Causes: Missing layers in an existing model-based analysis can have multiple causes. One is that the DSML is already modularised according to a layered architecture like the reference architecture for DSMLs by Heinrich et al. [HBK18]. In Chapter 3, we investigate the effects of modularising a model-based analysis according to the layered structure of the DSML. However, the *Missing Modularity* smell is more a suggestion than a clear smell when the DSML is already layered, but its corresponding model-based analysis follows none or another architecture pattern. Therefore, we consider this smell as not essential for the quality of the model-based analysis.

Identification: The model-based analysis either has layers or not; there is little to do to identify the *Missing Modularity* smell. The analysis developer must ensure the layers are identifiable for the other analysis developers working on the model-based analysis. How to define layers depends on the programming language and, under some circumstances, also on the IDE. In Java and Eclipse, for example, the layers could be organised by projects, where each project represents a layer, and each package represents an analysis component of the layer. However, reusing single packages is not provided by the Java programming language; thus, we recommend using a project to represent an analysis component and group the analysis components that are located on the same layer in working groups of the Eclipse IDE.

Refactoring: Ideally, the model-based analysis corresponds to a DSML that is already modularised according to the reference architecture for DSMLs. If that is the case, we explain the refactoring operations in Section 3.3.3 and Section 3.5.1 and the process to modularise a model-based analysis. However, if the DSML does not correspond to the reference architecture for DSMLs, we recommend refactoring the DSML first; see [HSR19].

Limitations: The *Missing Modularity* smell is only applicable if the model-based analysis is intended to have layers. If the layers are not identifiable due to language restrictions or missing documentation, the smell can indicate such problems. However, if a layered structure is already applied, the smell cannot detect problems in the layering itself. To find problems regarding the layering of the model-based analysis visit the smells *Folded Hierarchy* in Section 4.2.3.1 and *Unexploited Hierarchy* in Section 4.2.3.3.

4.2.4.4 Rebellious Modularity

A language component can contain multiple concerns. Even if the DSML is modularised according to the reference architecture for DSMLs [HSR19], the language architect might not be able to anticipate how the concerns are structured. As a result, the language architect can model the concerns very fine-grained, which means that the concern is separated into multiple language components. For example, instead of having a control flow language component in a DSML for software architectures, the language architect separates the control flow language component into multiple language components. However, the corresponding model-based analysis would only need a level of detail in the model. The indicator for a concern that is separated over different language components is multiple outgoing dependencies from one analysis component to different language components. Outgoing dependencies are dependencies from an analysis component to a language component. The dependencies result from classes of the analysis component that either extend or use classes from the language component. Figure 4.8 depicts the *Rebellious Modularity* smell, it shows one analysis component that depends on three language components.

Effect: The analysis component affected by the *Rebellious Modularity* contains many lines of code across multiple classes, similar to a god class (cf. Chapter 2); it either contains many classes or complex classes that cover different concerns. This analysis component cover different concerns; hence the dependencies on multiple language components. Due to the high number of classes or the large classes, the analysis component is complex and difficult to comprehend for the analysis developer. Also, reusing a subset of the language components with the analysis component is difficult, as all language components are required by the analysis component. As a result, due to the complexity of the analysis component, the evolvability is hampered, and the reusability is also problematic when some required language components cannot be used.

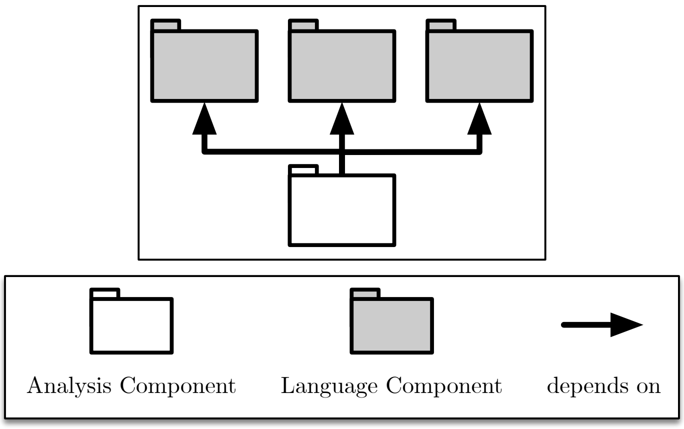


Figure 4.8.: Rebellious Modularity

Causes: The causes for the *Rebellious Modularity* smell on the model-based analysis can have two origins: *(I)* When the analysis component represents multiple concerns, it contains many classes that depend on different language components. Extending the analysis component by adding dependencies on language components requires less effort than splitting the analysis component or creating a new analysis component. *(II)* When the language components represent only a fraction of a concern, the language architect might not understand how the DSML is used. Alternatively, during the development of the DSML, the intended use of the language components shifted, and the fine-grained modular structure remained. For example, first, the language architect intended to use the aforementioned detailed control flow structure. However, while developing the corresponding model-based analysis, they discovered that the increased effort in modelling such a fine-grained structure does not benefit the analysis result. Therefore, they decided not to use the detailed elements of the DSML in the model-based analysis, but the possibility to model such details was not removed from the DSML.

Identification: Each analysis component is analysed regarding its outgoing dependencies. Outgoing dependencies are the dependencies from an analysis component to a language component. We disregard the dependencies between analysis components; as for the *Rebellious Modularity*, only the dependencies between analysis components and language components are relevant. If an analysis component depends on any number of language components above a threshold *n*, we determine it as *Rebellious Modularity*. The threshold *n* allow the analysis developer to define how many dependencies are allowed before they count as *Rebellious Modularity* smell.

Refactoring: If the cause for the bad smell is (I) that an analysis component represents multiple concerns, we split the analysis component into multiple components. Regarding Figure 4.8, we create two additional components, each depending on one language component respectively. To split an analysis component, the classes which contain undesired dependencies (i. e., dependencies on another language component than the required one) are moved to another or a new analysis component. If the classes contain dependencies on multiple language components, we split them. More details regarding the refactoring can be found in Section 3.3.3. If the cause for the bad smell is (II), the language components represent the same concern; we merge the language components. The refactoring operations of merging analysis components and analysis classes are presented in Section 3.3.3.

Limitations: Identifying the *Rebellious Modularity* smell does not always require access to the DSML components. If the DSML implements a standard, for example, the BPMN2 standard, the language architect might want to avoid changing the DSML. If the language architect must change the DSML and the DSML is developed according to a standard, changing the DSML means that it no longer conforms to the standard. Ergo, model-based analyses that expect a model conforming to the standard can no longer use instances of the changed DSML.

4.2.4.5 Weakened Modularity

When language components depend on analysis components, then this corresponds to the bad smell *Broken Modularity* (cf.Section 4.2.4.1). However, when dependencies from language components on analysis components exist, they can form dependency cycles between the DSML and its corresponding model-based analysis. Figure 4.9 depicts such a

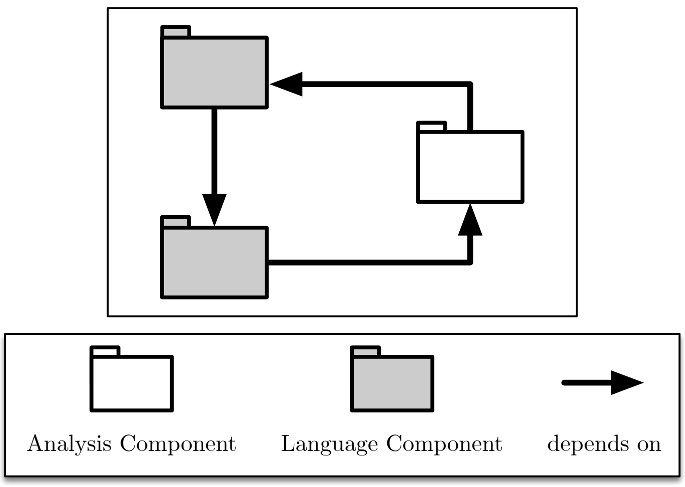


Figure 4.9.: Weakened Modularity

dependency cycle. The depicted cycle contains three components; however, the smallest cycle can exist between a language component and one analysis component. An analysis developer can easily detect and fix such examples, but if the cycles are complex, e. g. containing a dozen components, it is difficult for the analysis developer to detect them.

Effect: Dependency cycles between language components and analysis components have the same negative effect as dependency cycles in DSMLs [Str20] or in object-oriented software [Fow18]. If an analysis developer changes an analysis component in such a dependency cycle, it is most likely that they must also adapt the remaining components in the cycle. Maintenance steps become unpredictable, and in the worst case, they are a great risk because the effort required to realise a change cannot be estimated. The more complex the cycles in the model-based analysis, the higher the risk of changing existing code or adding new features. If the DSML is used by multiple model-based analyses, changes can affect not only one model-based analysis but all model-based analysis that depend on the DSML.

Causes: An analysis developer either introduces cycles into a system on purpose, without knowing the dire consequences or by accident. Ignoring the consequences of dependency cycles are inexcusable but unavoidable if done deliberately. Introducing dependency cycles by accident is only avoidable if the cycles are small enough so the analysis developer can detect them manually. Adding a language component as a dependency to an analysis component can form a cycle when somewhere exists a dependency from a language component on an analysis component.

Identification: In order to detect the *Weakened Modularity* smell, the analysis developer can transform the DSML and model-based analysis elements (i. e., classes, methods, attributes, packages, and DSML files) into a directed graph. They then apply a cycle detection algorithm like Floyd's cycle detection algorithm [Flo67] to that graph. Alternatively, the analysis developers can use a graph database like Neo4J [1] and its *apoc* extension [2] to detect cycles. For example, a graph can be created according to Strittmatter [Str20] when packages are transformed into nodes and dependencies between packages are transformed into edges. This transformation into a graph is applicable for both the DSML and the model-based analysis.

Refactoring: Fixing this smell requires understanding the DSML and the model-based analysis. If the types of the DSML can be moved to the model-based analysis, the language types are removed and added to the model-based analysis. This applies when the analysis developer finds in addition to the cycle also the *Broken Modularity* smell (cf. Section 4.2.4.1). Then, they can perform the refactoring described in the refactoring paragraph of the *Broken Modularity* smell. Alternatively, if multiple language types must be moved to the model-based analysis, it could be sufficient to move the whole feature. In order to move a whole feature, the refactoring of the *Broken Modularity* smell is applied to each class that corresponds to the affected feature. Also, dependency inversion can be performed to fix this bad smell (cf. Figure 3.8). Therefore, the language architect must inverse the dependency from the language component on the analysis component.

Limitations: Identification of the *Weakened Modularity* smell is only possible when the source code of the DSML is accessible, and the same is true for changes that affect the DSML. Even if the DSML is accessible, if the DSML implements a standard, for example, the BPMN2 standard, the language architect might want to avoid changing the DSML. Changing the DSML means that it no longer conforms to the standard. Ergo, model-based analyses that expect a model that conforms to the standard can no longer use instances of the changed DSML. The same goes for each tool that is based on the standard DSML. Another problem when fixing the *Weakened Modularity* smell is that the developers must change each corresponding tool of the DSML. In the long term, we recommend fixing this

[1] https://neo4j.com/

[2] https://neo4j.com/labs/apoc/4.1/overview/apoc.nodes/apoc.nodes.cycles/

bad smell, as it creates unpredictable changes to the DSML and its corresponding tooling; however, the effort that comes with fixing the bad smell should be considered and planned for each corresponding tool. Otherwise, each tooling is unusable until it is fixed.

4.3 Automatically Identify Bad Smells in Model-based Analyses

In Section 4.2 we introduced the twelve bad smells we identified; furthermore, we explained for each bad smell how to identify them. In this section, we show which bad smells we can automatically identify and, especially, how we are able to identify them. In order to identify the following bad smells, the model-based analysis must have an associated DSML that conforms to the reference architecture for DSMLs by Heinrich et al. [HSR19].

4.3.1 Identification of Abstraction Smells

In this section, we present how we identify bad smells related to the abstraction category presented in Section 4.2.1.

4.3.1.1 Duplicated Abstraction

We are able to identify the *Duplicated Abstraction* smell automatically. We transform the dependencies of analysis classes on language classes into a graph notation. To be able to create the graphs, the developer who performs the analysis must provide the path to the DSML and the path to the model-based analysis. Each analysis class and language class is represented as a node in the graph. If an analysis class uses a language class, these dependencies are represented as edges between these nodes. Our tool identifies each analysis class that shares a certain number of dependencies on language classes. The developer who performs the analysis can set the threshold of shared dependencies according to their needs. Listing 4.2 shows the sequencing when identifying the *Duplicated Abstraction* smell.

4.3.1.2 Missing Abstraction

We can automatically identify the *Missing Abstraction* smell. We collect all methods of the analysis classes and identify the methods with a primitive type in the signature. The extract the methods, the developer who performs the analysis must provide the path to the analysis. Our tool identifies each method that has a primitive type in its signature. The developer who performs the analysis must determine whether the found signature should be contemplated for further investigation. Listing 4.3 shows the sequencing when identifying the *Missing Abstraction* smell.

```
def duplicatedAbstractionIdentification(threshold):
   analysisClasses.foreach(class ->
      var dependenciesOnLanguageClasses =
         getDependenciesOnLanguageClasses(class)
      var sameDependencies =
         compareToDependenciesOnRemainingAnalysisClasses
            (dependenciesOnLanguageClasses)
      var sameDependencies = filterDependencies(threshold)
   )
   return sameDependencies

def compareToDependenciesOnRemainingAnalysisClasses
   (dependenciesOnLanguageClasses):
   var dependencies
   analysisClasses.foreach(
      dependencies.
         add(sameDependencies(
            dependenciesOnLanguageClasses,
            analysisClass))
   )
   return dependencies
```

Listing 4.2: Identification of the Duplicated Abstraction Smell

```
def missingAbstractionIdentification(path):
   var classes = getAllAnalysisClasses(path)
   classes.foreach(class ->
      methods = class.getMethods()
      methods = methods
         .filter(it::hasPrimitiveParameter)
   )
```

Listing 4.3: Identification of the Missing Abstraction Smell

4.3.1.3 Unused Abstraction

In order to identify the *Unused Abstraction* smell automatically, we transform the dependencies of analysis classes on language classes into a graph notation. To be able to create the graphs, the developer who performs the analysis must provide a link to the DSML and the model-based analysis. Each analysis class and language class is represented as a node in the graph. If an analysis class uses a language class, these dependencies are represented as edges between these nodes. We then search for each language class node that is not connected to an analysis class node and provide a list of all identified nodes as a result of the analysis.

```
def deficientEncapsulationIdentification(languagePath, analysisPath):
   var parameterTuples
   var methods = getAllMethodsFromAnalysis(analysisPath)
   filterParameters(methods)
   methods.foreach(method ->
      parameterTuples.add(createTuples(method))
   )
   removeIdenticalPairs(tuples) // {a,a} or {b,b}
   countTupleOccurrences(tuples)
```

Listing 4.4: Identification of the Deficient Encapsulation Smell

4.3.2 Identification of the Encapsulation Smell

In this section, we present how we identify the bad smell related to the encapsulation category presented in Section 4.2.2.

4.3.2.1 Deficient Encapsulation

We can partially identify the *Deficient Encapsulation* smell automatically. The developer who performs the analysis must provide the path to the DSML and the path to the model-based analysis. We collect all types of the language and all methods of the model-based analysis. The methods with only one parameter are discarded. For each parameter pair, we generate a tuple, but we omit tuples that contain the same parameter. Then, we count the occurrence of each tuple. The resulting tuples and the number of occurrences are documented as *Deficient Encapsulation* smell. The developer who performs the analysis has to decide whether these tuples of language types can be merged. Our tool identifies each parameter tuple that occurs in the methods of the model-based analysis and counts the number of occurrences. Listing 4.4 shows the sequencing when identifying the *Deficient Encapsulation* smell.

4.3.3 Identification of Hierarchy Smells

In this section, we present how we identify bad smells related to the hierarchy category presented in Section 4.2.3.

4.3.3.1 Folded Hierarchy

We can automatically detect the *Folded Hierarchy* smell when the layer information is available in the DSML and the model-based analysis that gets analysed. The developer who performs the analysis must provide the path to the DSML and the model-based analysis. Furthermore, they must provide the layers that are used by the DSML and model-based analysis. We transform the language and analysis classes' dependencies into a graph notation. Each class is represented as a node in the graph. If an analysis class references a language class, these dependencies es represented as an edge between these nodes. Each class node contains information on which layer the class is located and which order number the layer has. The more generic a layer is the smaller its number. We distinguish between two violations of the layering. The first violation is when an analysis class (source) depends on a language class (target) on a more specific layer. To detect this violation, we compare the order number of the classes. If the order number of the source class is less than the order number of the destination class, the aforementioned layering rule is violated. The second violation is when an analysis class (source) depends on a language class (target) on a more generic layer. To detect this violation, we compare the order number of the classes. If the difference between the order numbers, source minus destination, is greater than zero, the aforementioned layering rule is violated.

4.3.3.2 Missing Hierarchy

We can automatically detect the conditions that lead to the *Missing Hierarchy* smell. The developer who searches for the *Missing Hierarchy* smell must provide the path to the DSML and the model-based analysis. First, we search the analysis classes and extract all references on the DSML. Then, we filter the reference and remove all references that do not originate from a switch statement. The developer must analyse the remaining references to determine whether they contribute to the *Missing Hierarchy* smell.

4.3.3.3 Unexploited Hierarchy

We can automatically detect the *Unexploited Hierarchy* smell when the layer information is available in the model-based analysis that gets analysed. The developer who performs the analysis must provide the path to the model-based analysis. Furthermore, they must provide the layers that the model-based analysis uses. We transform the dependencies of the analysis classes into a graph notation. Each analysis class is represented as a node in the graph. If an analysis class references another analysis class, these dependencies es represented as an edge between these nodes. Each analysis class node contains information on which layer the class is located and which order number the layer has. The more generic a layer is the smaller its number. We distinguish between two violations of the layering. The first violation is when an analysis class (source) depends on another analysis class (target) on a more specific layer. To detect this violation, we compare the order number of the classes. If the order number of the source class is less than the order number of the destination class, the aforementioned layering rule is violated. The second violation is when an analysis class (source) depends on another analysis class (target) on a non-adjacent layer. To detect this violation, we compare the order number of the classes. If the order number of the source class is greater than the order number of the destination class, and if the difference between these numbers is greater than one, the aforementioned layering rule is violated.

```
def brokenModularityIdentification(languagePath, analysisPath):
   var languageTypes = getAllLanguageTypes(languagePath)
   var analysisTypes = getAllAnalysisTypes(analysisPath)
   var references = collectAllReferences(languageTypes)
   references.foreach(reference ->
      if not referencesAnalysisType(reference, analysisTypes):
         references.remove(reference)
   )
```

Listing 4.5: Identification of the Broken Modularity Smell

4.3.4 Identification of Modularity Smells

In this section, we present how we identify the bad smells that are related to the modularity category presented in Section 4.2.4.

4.3.4.1 Broken Modularity

To automatically detect the *Broken Modularity* smell, the developer who performs the analysis must provide the path to the DSML and the path to the model-based analysis. We collect all references of the language types in the DSML, and we also collect the analysis types of the model-based analysis. For each reference of the language, we filter out each reference that does not depend on an analysis type. The remaining references violate the *Broken Modularity* smell. Listing 4.5 shows the sequencing when identifying the *Broken Modularity* smell.

4.3.4.2 Degraded Modularity

To automatically detect the *Degraded Modularity* smell, the developer who performs the analysis must provide the path to the DSML and the path to the model-based analysis. We collect all language types and analysis types of the path provided by the developer. Then, we collect all references of analysis types to language types. Of the collected references, we filter the references on language types that originate from different analysis types. Listing 4.6 shows the sequencing when identifying the *Degraded Modularity* smell.

4.3.4.3 Missing Modularity

We can automatically detect the *Missing Modularity* smell when the layer information is part of the DSML and the model-based analysis. Currently, we support Eclipse Plugin projects. The developer who performs the analysis must provide the path to the DSML and the model-based analysis. We assume that the layer information is located in the metafile of the plugin. If the metafile does not contain information regarding the layer, we determine that the projects are missing a layered structure.

```
def degradedModularityIdentification(languagePath, analysisPath):
   var languageTypes = getAllLanguageTypes(languagePath)
   var analysisTypes = getAllAnalysisTypes(analysisPath)
   var references = collectAndGroupAllReferencesOnLanguageTypes(
      languageTypes,
      analysisTypes)
   references.foreach(referenceGroup ->
      if not containsDifferentAnalysisTypes(referenceGroup):
         references.remove(reference)
   )
```

Listing 4.6: Identification of the Degraded Modularity Smell

```
def degradedModularityIdentification(languagePath, analysisPath):
   var languageTypes = getAllLanguageTypes(languagePath)
   var analysisTypes = getAllAnalysisTypes(analysisPath)
   var references = collectAndGroupAllReferencesOnLanguageTypes(
      languageTypes,
      analysisTypes)
   references.foreach(referenceGroup ->
      if not (containsDifferentLanguageTypes(referenceGroup)
         and originateFromOneAnalysisType(referenceGroup)):
         references.remove(reference)
   )
```

Listing 4.7: Identification of the Rebellious Modularity Smell

4.3.4.4 Rebellious Modularity

To automatically detect the *Rebellious Modularity* smell, the developer who performs the analysis must provide the path to the DSML and the path to the model-based analysis. We collect all language types and analysis types of the DSML and model-based analysis. Then, we collect all references of analysis types to language types. Of the collected references, we filter the references on language types that originate from the same analysis type. Listing 4.6 shows the sequencing when identifying the *Degraded Modularity* smell.

4.3.4.5 Weakened Modularity

IDEs, like Eclipse and IntelliJ, are already capable of identifying cycles between classes, components, and projects. The cycles between a DSML and its corresponding model-based analysis can occur when the code for the DSML is generated and altered afterwards. As our approach is based on EMF-based metamodels, the generated code is embedded in regular Eclipse plugin projects. Eclipse provides a cycle detection for its plugins that is based on the metafiles of the plugin projects. Therefore, it is unnecessary to implement our own cycle detection.

5. Reuse of Model-based Analysis Components

After presenting decomposition and composition approaches for model-based analyses, and after presenting bad smells dedicated to model-based analyses and how to identify, categorise, and refactor these bad smells, in this final contribution chapter, we present an approach to specify, compare, and identify model-based analysis components. Our specification and identification approach decreases the effort required for reusing model-based analysis components. After identifying and refactoring bad smells in model-based analysis and modularising existing model-based analyses according to the process presented in Chapter 3 and Chapter 4, the analysis architect has a repository filled with analysis features and their corresponding analysis components. As motivated in Chapter 3, modularisation is necessary to reuse analyses in different analyses. However, reusing an analysis component is more than just using an analysis component more than once. Specifying a component regarding its desired structure and behaviour and identifying a possible analysis component that complies with a desired specification is also part of reusing an analysis component.

As a model-based analysis' complexity grows, it becomes more difficult to understand and, as a result, to maintain, extend, or reuse. Because of the increased complexity of the mode-based analysis, already created model-based analysis components must be reused in succeeding model-based analysis projects. Reusing model-based analysis components allows for saving time and resources. On the other hand, the specialisation of model-based analyses for a particular domain or even a specific system limits reusability for other domains or systems. On a syntactic level, the structure of a component (i. e., classes, interfaces) might be identified as a possible match for a reuse candidate. However, because a model-based analysis depends on a domain or system, it is difficult to determine whether the discovered component is a semantic match (i. e., exhibits the required behaviour). If the number of components to be analysed is vast or the components are complex, it may be prohibitively expensive.

In this chapter, we present an approach for specifying model-based analyses structure and behaviour. We use the specification of model-based analyses to find analysis components with a similar structure and behaviour. To specify model-based analysis components, we use a modelling technique based on metamodels and on a DSL. We also present an approach for identifying similar model-based analysis components by comparing model-based analysis components in structure and behaviour. The process of comparing analysis components is separated into two stages: We begin by comparing model-based analysis components regarding their structure. We then transform the requirements into graph notation and perform a graph-isomorphism analysis to detect similar structures. Second, we compare model-based analysis components based on their behaviour by translating the specification to the SMT notation and then detecting similar behaviour utilising an SMT-Solver. We decided to use an SMT-Solver because they are capable of processing a wide variety of theories, such as arithmetic, bit-vectors, arrays and others. SMT-Solvers are highly efficient and can quickly solve complex logical formulas. They use a combination of decision procedures, heuristics, and optimisations to explore the search space and find a solution efficiently.

The contribution in this chapter is based on our previous publications regarding the specification and reuse of model-based analysis components [Koc+22] and [KR22]. Our contributions in this chapter are structured as follows: After presenting the research questions in Section 5.1 and additional terms and definitions necessary to understand the content of this chapter in Section 2.5, we introduce the specification approach for specifying the structure and behaviour of model-based analyses in Section 5.2. Our approach to compare specified analysis components based on their structure is presented in Section 5.3. Our approach to compare specified analysis components based on their behaviour is presented in Section 5.4.

5.1 Hypothesis and Research Questions

In this section, we present the hypothesis and research questions for the third contribution of this thesis. Finding the suitable model-based analysis feature and its corresponding model-based analysis component is a non-trivial task. The analysis architect has to analyse each available model-based analysis component using its documentation or source code. Even if the analysis component is documented and the documentation is up-to-date, it is a costly and time-consuming task if done manually. We derive the following hypothesis for improving the reuse of model-based analysis components:

Hypothesis 3

The reuse of a model-based analysis will improve when we reduce the barrier to finding reusable analysis components.

To determine whether the analysis component fits the required needs, the analysis architect has to understand the analysis component they want to reuse in the model-based analysis. Instead of analysing the source code or the documentation, an analysis component specification could be used to compare analysis components. We distinguish between the specification of model-based analysis components and compare model-based analysis components.

5.1.1 Model-based Analysis Specification

Before comparing an analysis component, we need a common specification for model-based analyses which serves as the foundation to compare and identify model-based analysis components. Therefore, we ask the first research question:

Research Question 5.1

What methods or techniques can be employed to specify model-based analysis components that enable comparison between different components?

The specification needs to be an abstraction of the structure of the analysis component. Thus, we derive a sub-research question **RQ 5.1.1**: *How to specify the structure of a model-based analysis?* However, more than the structure of an analysis component may be needed to determine if a component fits the desired needs of the analysis architect. Even if the component fits structurally (i. e., programming language and interfaces), the analysis architect still must read the documentation or analyse the source code to determine whether the analysis component has the desired behaviour. Therefore, besides the structure, the specification must also be able to specify the behaviour of an analysis component. Thus, we derive another sub-research question **RQ 5.1.2**: *How to specify the behaviour of a model-based analysis?*

5.1.2 Model-based Analysis Component Identification

After specifying analysis components, the analysis architect must be able to identify existing components based on their specification rather than their documentation or, in the worst-case based on their source code. Therefore, we derive the second research question for this contribution:

Research Question 5.2

What methods can compare and accurately identify similar model-based analysis components?

The identification of analysis components utilises the two aspects of the specification. First, an analysis component should be identified based on its structure. Therefore, we derive the sub-research question **RQ 5.2.1**: *How to identify model-based analysis components with a similar structure?* There needs to be more than the structure to determine whether the behaviour also fits the desired specification. Thus, we derive the second sub-research question **RQ 5.2.2**: *How to identify model-based analysis components with a similar behaviour?*

5.2 Structure and Behaviour Specification of Model-based Analyses

In this section, we present our specification approach to address **RQ 5.1.1**: modelling the structure of a model-based analysis and **RQ 5.1.2**: modelling the behaviour of a model-based analysis. We focus on a subset of model-based analyses, the discrete event simulations. The structure and behaviour modelling will be done via a DSL that we have developed. To process the models specified via the DSL, we define a *metamodel* as the underlying abstract syntax of the language. We keep the language on a high level of abstraction. Hence, a comparison between simulation specifications has to handle as little unnecessary complexity as possible while still being precise enough to detect behavioural similarities. As a result, the language does not allow the specification of simulation output parameters or the creation and flow of entities between events. We keep the language on a high level of abstraction; as a result, a comparison between simulation specifications has to handle as unnecessary complexity as possible. The most crucial generalisation we make is to always refer to entities and attributes on the type-level, i. e., referring to static objects instead of specific instances; there is no tracking of the flow of entities between events or their creation and destruction. Furthermore, we exclude the specification of the simulation output because the definition of the simulation result does not impact simulation behaviour. Our DSL and our metamodel are separated regarding the entities concerned with the structure of a simulation from the entities concerned with a simulation's behaviour.

First, we define a discrete-event simulation in Section 5.2.1. Our specification DSML, which consists of a language feature to model the structure of a Discrete-event Simulation (DES), presented in Section 5.2.2 and a language feature to model the behaviour of a DES, presented in Section 5.2.3. Then, we introduce our grammar used in our DSL to model the structure and behaviour of DES, presented in Section 5.2.4. Finally, we introduce our SMT representation of the behaviour, presented in Section 5.2.5.

5.2.1 Discrete-event Simulation Definition

A simulation is a representation of a system and its behaviour over time. The purpose of a simulation is to analyse a system to get information that would otherwise be impossible to attain. Simulations have been proven helpful for characterising behaviour patterns in a target system [Dur21]. Simulations are utilised when experimenting with an existing system is too time-consuming, expensive, dangerous, or impossible since the system does not exist. A system is a collection of entities that interact according to mathematical or logical relationships in the context of simulations [DLK94]. A simulation has multiple analysis components. In the context of a simulation, an analysis feature is part of a simulation that reflects a simulation functionality, such as task scheduling or time progression. One or more analysis components are in charge of implementing a simulation feature [Hei+21b]. A simulation *model* represents and reflects the structure of a system. The simulation model generates the system's behaviour [Top+16]. The simulator, or the running part of a simulation, provides the behaviour used to understand the simulation model and handle events. The simulation *world* depicts the current condition of the system entities. Events can alter the state of one or more entities.

We categorise simulations regarding their understanding of time. In *continuous* simulations, model attributes such as state variables change continuously with respect to time. Examples of continuous simulations are traffic simulations, where the exact positions of vehicles are observed, or weather simulations, concerned with temperatures and wind speeds. In *discrete* simulations, model attributes only change at discrete, separated points in time. In this thesis, we will focus on DESs. DES is a type of discrete simulation where states only change at instantaneous points in time. These points in time are called *events*. Because state changes can only occur at events, no processing is required between two events. Simulation time can be advanced from one event to the next without consuming unnecessary computation time. In DES, the state of the simulation world is represented by a set of entities, each with a set of attributes. A common example for DES is the simulation of persons waiting in a queue. The queue represents an entity that contains a list of persons, and the list of persons represents one attribute of the entity. An example of an event is when a person is added to the queue or when a person is removed from the queue. We define a simulation S as a set of entities E, events V, and a starting event e_0:

$$S = \{E, V, e_0\}$$

We define the state of an Entity as Z, the initial state of an entity is Z_0, and the state of an entity at a given time is defined as $Z_t = Z_0 \oplus \Delta Z_m$. The state of an entity is a set of attributes a:

$$Z_t = \{a_1 \dots a_n\}$$

Only events can change the state of an entity. We define an event e_t as follows:

$$e_t(E) = \{E', V'\}$$

An event can change multiple entities, which are defined as a set of entities E. The result of an event is a set of entities E' that are changed by the event and a set of events V' that are scheduled by the event $e_t(E)$.

5.2.2 Model-based Structure Specification of Discrete Event Simulations

We define the structure of a simulation as the set of basic building blocks described in Section 2.5: events, entities and attributes. In this structural view of a simulation, an event is merely an identifiable object without any behavioural aspects attached to it. Figure 5.1 depicts the metamodel to model the structure of a discrete event simulation. A *simulation* contains a set of *Entities* and *Events*, and each entity contains a set of typed *Attributes*. Additionally, we model a *reads* relationship between events and attributes to describe

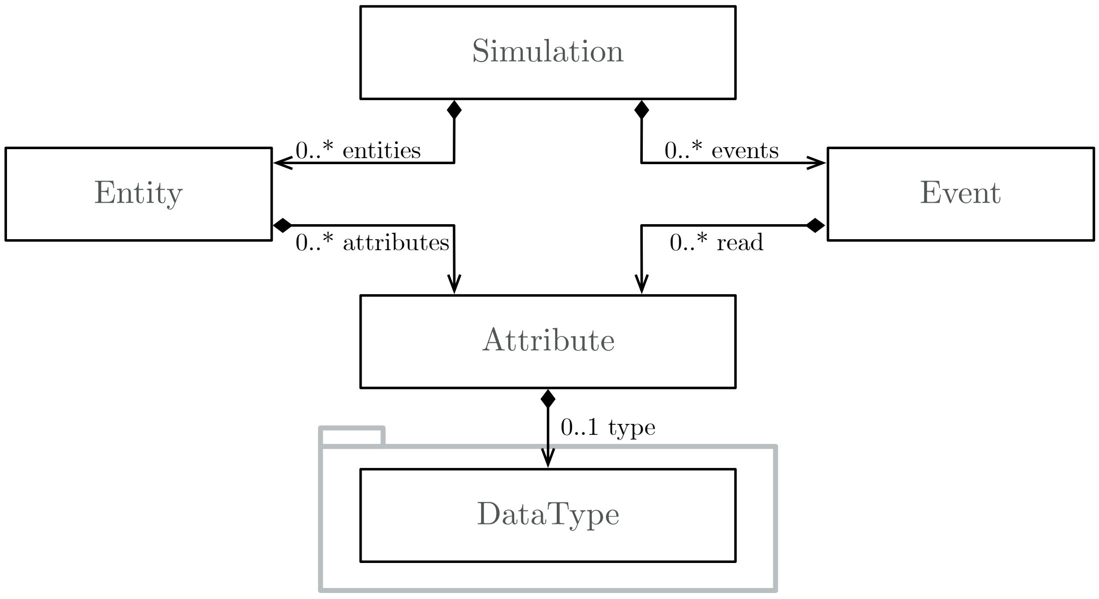


Figure 5.1.: Metamodel for Specifying the Structure of DES Components

which attributes affect the behaviour of an event. This relationship is part of the structural metamodel because a read-operation on an attribute does not affect the simulation world.

5.2.3 Model-based Behaviour Specification of Discrete Event Simulations

While there are different definitions of the term behaviour, we define the behaviour of a simulation as the effects of events on the state of the simulation world, i. e., the changes to attributes triggered through events. Besides the simulation structure, two additional concepts are necessary to specify the behaviour of a simulation. A simulation changes the simulation world during its runtime. In order to describe those changes, the language must allow a specification of changed attributes as part of the simulation specification. Attribute changes can be linked to events during which they occur since in DES, such changes can only happen at events. This is always the case in DES because an *event* is defined as any point in time that marks a change in the simulation world (cf. Section 2.5). The state of the simulation world is indirectly affected by the order and time that events are scheduled. Events may cause other events to be scheduled with a certain delay. Figure 5.2 shows the metamodel to describe these behavioural aspects.

The *Schedules* and *WritesAttribute* classes represent the aforementioned behavioural concepts. In addition to the references to events and attributes, these classes contain expressions to define the schedules- and writes-relationships further. Expressions can be constant values, attributes (as long as there is a reads-relationship for the corresponding event and attribute) or a combination of other expressions with the common logical and arithmetic functions. Expressions are used to describe the delay of a schedules-relationship and the new attribute value of a writes-relationship. Additionally, both elements can define a

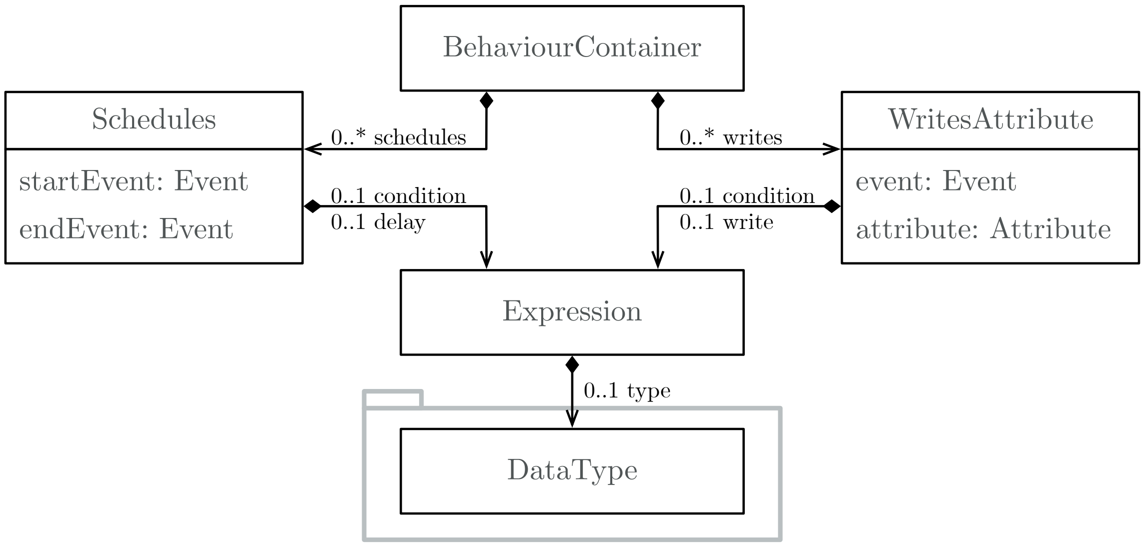


Figure 5.2.: Metamodel for Specifying the Behaviour of DES Components

condition expression that restricts when an event should be scheduled or an attribute should be overwritten.

It is generally allowed to model multiple schedules- and writes-relationships between two events or between an event and an attribute. For schedules-relationships, this results in an event A scheduling multiple instances of event B, for example, with different delays. For writes-relationships, different values can be written to the same attribute under different conditions.

5.2.4 Grammar-based Specification of Discrete Event Simulations

In this section, we present the concept of modelling a DES by utilising a grammar-based modelling approach. Therefore, we present the technical realisation to motivate and describe the concepts of our modelling approach. The specification grammar, the linchpin of our specification approach, utilises the metamodels for structure and behaviour. We use the grammar language of Xtext, a variation of the EBNF, to specify the grammar of our DSL. A simulation S is a set of simulation components S_C. A simulation component S_C also consists of a set of entities E, events V, and a starting event e_0: $S_C = \{E, V, e_0\}$. Thus, we can define a simulation S as a set of simulation components S_C: $S = \{S_C\}$.

The user can define each S and S_C individually. The non-terminal symbols of the grammar that serve as an entry point for the parser determine whether a simulation or a simulation component is specified. The non-terminal symbols are presented in Listing 5.1. The main non-terminal, or grammar specification, as it is called in Xtext, is the *ModelElement*; it produces either a *SimulationComponent* or a *simulation*. The *simulation*, indicated by the *simulation* keyword, defines a simulation. It consists of an arbitrary amount of simulation components ($S = \{S_C\}$). To enable a better understanding of a specified simulation, we added the possibility of adding a description.

```
ModelElement: SimulationComponent | Simulation;

SimulationComponent: 'component' name=QualifiedName
   (uses+=UseComponent)*
   (events+=Event | entities+=Entity | enums+=EnumDeclaration)*;

Simulation: 'simulation' name=ID '{'
   ('description' '=' description=STRING)?
      'components' components+=[SimulationComponent|QualifiedName]
         (',' components+=[SimulationComponent|QualifiedName])*
   '}';

UseComponent: 'use' component=[SimulationComponent];
```

Listing 5.1: Language Declaration – Main Parser Rules

The simulation components S_C, indicated by the *component* keyword, can depend on other simulation components, as they can schedule events and affect the attributes of other simulation components. The *UseComponent* grammar specification and the *use* keyword make the events and attributes of other simulation components available. A qualified name references each simulation component; thus, it is possible to separate the components' definition from the simulation's definition. A simulation component consists of the structure elements shown in Figure 5.1, namely *Events*, *Entities*, and *EnumDeclarations*.

Listing 5.2 depicts the grammar specification for the structural elements of a simulation component. First, we have the definition of events ($e_t(E) = \{E', V'\}$). An event has a name that serves as an identifier and can read attributes of other components; the attributes of other components are made available via the *use* keyword. Besides reading attributes, an event can also schedule other events or change attributes. Scheduling other events and changing attributes are part of the behaviour specification presented in the next paragraph. The structure specification syntax also supports the definition of entities. An entity, indicated by the *entity* keyword, is identified by a name and consists of attributes.

Listing 5.3 depicts the grammar specification for the behaviour elements of a simulation. The grammar specification *Schedules*, indicated by the *schedules* keyword, allows specifying which events are scheduled. The delay can be added if an event is scheduled later. The delay is indicated by the *with* and *delay* keywords. Some events are only scheduled if a condition is met. To both, the delay and the condition are expressions assigned. The delay can get integers or floats assigned, and the condition gets a boolean expression assigned. The grammar specifications *WriteToValue* and *WriteToArray* represent the assignment of values to attributes.

Listing 5.4 depicts the grammar specification for the type declaration of the specification language. The grammar specification *Type* can either be a *PrimitiveType*, an *EnumType* or an *ArrayType*. Primitive types are integers, doubles, or booleans. Enums allow the definition of dedicated types, as indicated by the *datatype* keyword. *ArrayTypes* are used when more than one type is needed.

```
Event returns structure::Event:
   ('event' {GEvent} name=ID '{'
      ('reads' readAttributes+=[structure::Attribute|QualifiedName]
         (',' readAttributes+=[structure::Attribute|QualifiedName])*)?

      (schedules+=GSchedules | writeAttributes+=GWritesAttribute |
         definitions+=Definition)*
   '}');

Entity returns structure::Entity:
   'entity' name=ID ('{'
      (attributes+=Attribute)*
   '}')?;

Attribute returns structure::Attribute:
   name=ID ':' type=Type;
```

Listing 5.2: Structure Specification Syntax

```
Schedules:
   'schedules' endEvent=[structure::Event]
   (delaySpec=Delay? & (conditionSpec=Condition)?);

WritesAttribute:
   'writes' writeFunction=(WriteToValue | WriteToArray)
   (conditionSpec=Condition)?;

Delay: 'with' 'delay' '=' delay=Expression;
Condition: 'when' condition=Expression;

WriteToValue returns WriteFunction:
   {WriteToValue} attribute=[structure::Attribute|QualifiedName]
      '=' value=Expression;

WriteToArray returns WriteFunction:
   {WriteToArray} attribute=[structure::Attribute|QualifiedName]
   '[' index=Expression ']' '=' value=Expression;

Definition: 'def' name=ID '=' expression=Expression;
```

Listing 5.3: Behaviour Specification Syntax

Listing 5.5 shows an example simulation to demonstrate the textual syntax for the simulation structure. For the sake of brevity, the example omits the definition of events. We will present an example in the following paragraph. Consider a pedestrian light that switches between the colours red and green and keeps track of the number of waiting pedestrians. We can model this concept with a single entity and two events. The example shows the definition of datatypes, entities and events, and the reads-relationships. In line one, the type *Colour* with two states, *Red* and *Green* is modelled. The entity *TrafficLight* is modelled

```
Type returns datatypes::DataType:
   {datatypes::BaseDataType} primitiveType=PrimitiveType |
   {datatypes::EnumType} declaration=[datatypes::EnumDeclaration] |
   ArrayType;

ArrayType returns datatypes::ArrayDataType: 'ARRAY' '[' contentType=Type ']';

EnumDeclaration returns datatypes::EnumDeclaration:
   'datatype' name=ID '{'
      (literals+=ID) (',' literals+=ID)*
   '}';
enum PrimitiveType returns datatypes::PrimitiveType: INT | DOUBLE | BOOL ;
```

Listing 5.4: Type Declaration

```
 1 datatype Colour { Red, Green }
 2
 3 entity TrafficLight {
 4    waitingPedestrians: INT
 5    colourPedestriansLight: Colour
 6    colourTrafficLight: Colour
 7 }
 8
 9 event PedestrianGreen { ... }
10 event PedestrianRed { ... }
```

Listing 5.5: Example of the Specification Language

in lines three to seven. The traffic light consists of the number of waiting pedestrians, the colour of the pedestrian light and the traffic light.

Listing 5.6 extends the structural specification of the *PedestrianGreen* event by the behavioural aspects. The event switches the pedestrian light to green and resets the number of waiting pedestrians (if there are any). It also schedules the event to switch the light back to red with a delay of 15 seconds. Line one to nine defines the event when the pedestrian light turns green, *PedestrianGreen.* It reads the number of waiting pedestrians in line two. It also changes two attributes in the simulation world. In line four, it changes the pedestrian light to green, and in line 5, the number of waiting pedestrians is reduced to zero. In the simulation, we assume that all pedestrians waiting at the pedestrian light want to cross the street; therefore, after the light turns green, all pedestrians will leave their waiting position. The number of waiting pedestrians is only reduced to zero if it had people waiting at the light. The event also schedules another event with some delay, shown in line eight. The event *PedestrianRed* is not shown, as it is similar to *PedestrianGreen.*

```
event PedestrianGreen {
   reads TrafficLight.waitingPedestrians

   writes TrafficLight.colourPedestriansLight = Colour.Green
   writes TrafficLight.waitingPedestrians = 0
   when waitingPedestrians != 0

   schedules PedestrianRed with delay = 15.0
}

event PedestrianRed { ... }
```

Listing 5.6: Example of the Specification Language with Behaviour

5.2.5 Representing Behaviour with Automated Reasoning and Logical Formulas

In this section, we present how we compare the behaviour of simulation components by modelling the effect of their schedules- and writes-relationships on the simulation world. To represent these effects, we use automated reasoning and logical formulas. Therefore, we specify the instances of the behaviour metamodel as logical formulas and transform them into SMT statements. We distinguish between relationships that directly or indirectly affect the attributes of the simulation world. The attributes are indirectly affected by schedules-relationships; these relationships can schedule events, and these events can directly affect attributes of the simulation world. To determine that two schedules-relationships have the same behaviour, these relationships must schedule the same event with an identical delay. Listing 5.7 shows the definition of waiting passengers and a concrete delay of 15 seconds.

```
∃ waitingPassengers ∈ ℤ :
  waitingPassengers > 0 ∧
∃ delay ∈ ℝ :
    delay = 15
```

Listing 5.7: Delay Specification

In order to compare two schedules-relationships, it is sufficient to model the condition and delay the expression of a schedules-relationship as separate SMT formulas (cf. Listing 5.8). The following SMT statements are written in the SMT-LIB syntax. These statements depict variable declarations for all attributes accessed in those statements. Formulas that specify the value of a variable (i. e., delays and write-functions) also include a declaration for the variable. The SMT equivalent to the given logical formula statement is shown in Listing 5.7.

We consider two writes-relationships to have the same behaviour if they affect the attribute in the same way, i. e., if the attribute value after the specified write-action is the same for every given assignment of input values. The condition cannot be modelled independently from the write-function for writes-relationships because the new attribute value depends

```
// condition:
(declare-fun waitingPassengers () Int)
(assert (> waitingPassengers 0))

// delay:
(declare-fun delay () Double)
(assert (= delay 15))
```

Listing 5.8: Delay Specification Modelled with SMT [Koc+22]

on both the condition and the write-function. An attribute will obtain the value specified by the write-function if the condition evaluates to true; otherwise, the attribute will keep its old value (cf. Section 5.2.3).

```
∀ old, value, input ∈ ℕ :

((input > 0) ⟶ (value = old + 1)) ∧
(¬(input > 0) ⟶ (value = old))
```

Listing 5.9: Write Specification

Listing 5.10 shows this correlation in SMT-LIB syntax with two implications. An attribute of type integer will be increased by one if the condition `input > 0` is met.

The SMT equivalent to the given logical formula in Listing 5.9 is:

```
(declare-fun old () Int) // old attribute value
(declare-fun value () Int) // new attribute value
(declare-fun input () Int) // additional input

(assert (=> (input > 0) (= value (+ old 1))))
(assert (=> (not (input > 0)) (= value old)))
```

Listing 5.10: Write Specification Modelled with SMT

This formula uses quantifiers to assert that the relationships hold for all integer values of old, value, and input. The first assertion states that if the input is greater than 0, the value should be equal to *old + 1*. The second assertion states that the value should be equal to *old* if the input is not greater than 0.

As described in Section 5.2.3, multiple writes-relationships to the same attribute from the same event are allowed. Because the final value of an attribute depends on all of those writes-relationships, a combination of all writes-relationships can represent the effect of an event on the attribute. The representation of writes-relationships can be extended to meet this requirement by creating an implication for each relationship. Then the attribute keeps its old value if none of the conditions is met.

For n writes-relationships from event A to attribute C with condition-expressions $C_{1..n}$ and write-functions $F_{1..n}$. Listing 5.11 shows the general write specification to describe the effect A has on C.

```
∃ old ∈ ℤ, value ∈ ℤ :
   (C1 ⇒ value = F1) ∧ ... ∧ (Cn ⇒ value = Fn) ∧
   (¬(C1 ∨ ... ∨ Cn) ⇒ value = old)
```

Listing 5.11: General Write Specification

Listing 5.12 shows the combined SMT formula based on Listing 5.11.

```
(declare-fun old () Int)
(declare-fun value () Int)
(declare-fun ...) // additional inputs

(assert (=> (C_1) (= value F_1)))
...
(assert (=> (C_n) (= value F_n)))
(assert (=> (not (or C_1 .. C_n)) (= value old)))
```

Listing 5.12: General Write Specification [Koc+22]

5.3 Utilising Model-based and Grammar-based Specification of Discrete Event Simulations for Structure Comparison

In this section, we present our approach to compare the structure of specified analysis components modelled with our DSL. We developed this approach to address research question **RQ 5.2.1**, identifying simulations based on their structure. We use a graph-based representation of these specifications with annotated nodes and edges. Entities, events and attributes are represented as nodes, while schedules- and writes-relationships, and parent-child relationships between entities and attributes are represented as edges. Figure 5.3 shows an example of such a graph representation. The picture shows the aforementioned traffic light simulation, modelled as a graph. The entity *Traffic Light* has two attributes, the *colour* the traffic light can take and the number of *waiting pedestrians* that wait at the traffic light. The graph also contains two events that read the number of waiting pedestrians and write to the colour attribute i. e., change the colour of the traffic light. The distinction between structure and behaviour in our metamodel differs slightly from the distinction we make for comparing structure and behaviour. The behaviour metamodel contains the entire specification of schedules- and writes-relationships; the presence of these relationships can be integrated into the graph representation of the elements from the structure metamodel. However, this information is irrelevant to the structure comparison, and thus, we omit it in Figure 5.3. The entire simulation specification cannot be compared using a graph-based approach because of the use of expressions in schedules- and writes-relationships, representing a paradigm orthogonal to the graph notation.

We consider two simulation specifications structurally similar if their graph representations are isomorphic, i. e., if there is a bijection between the structural elements (i. e., entities,

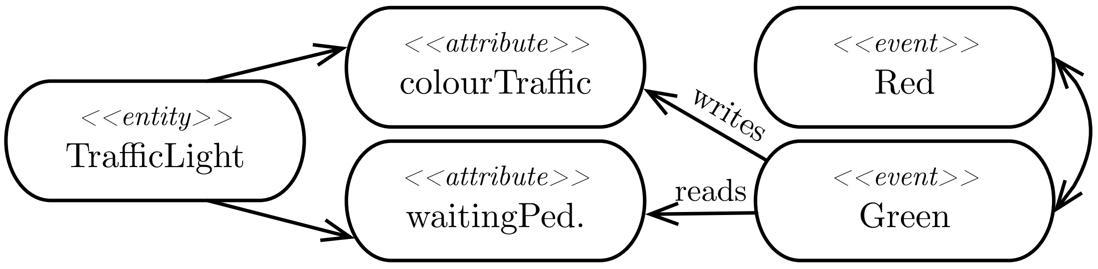


Figure 5.3.: Graph-representation of Structural Elements

attributes, and events) of both simulations. Regarding entities and attributes, graph isomorphism ensures that the simulation worlds of both simulations can store the same information. A bijection between the events of both simulations ensures that each event in simulation A has a uniquely associated event in simulation B that schedules the same events. Furthermore, the reads- and writes-relationships ensure that mapped events access the same properties of the simulation world. We will use this bijection as a starting point for a behaviour comparison that takes the expressions into consideration that specify the behaviour of schedules- and writes-relationships. There may be multiple isomorphisms between the graph representations of the two simulations. This is possible if two attributes of an entity only participate in reads-relationships from the same event. In this case, both attributes are structurally indistinguishable, and a behaviour comparison needs to be employed for each isomorphism.

5.4 Utilising Model-based and Grammar-based Specification of Discrete Event Simulations for Behaviour Comparison

In this section, we present our approach comparing specified analysis features addressing research question **RQ 5.2.1**, identifying simulations based on their behaviour. To explain our approach, we use the traffic light example shown in Figure 5.3. We will demonstrate how the example is modelled with our DSL and how the behaviour is represented as SMT formula. The usage of expressions in the behavioural metamodel renders the graph-isomorphism approach unusable for behaviour comparison, even though a description of the structure of simulations with the structural metamodel contains sufficient information to utilise a graph-based structural comparison. The expressions specifying the behaviour in the simulation specification are first-order logic statements. We use these statements as part of SMT instances (as introduced in Section 2.5). We use the first-order logic statements to derive SMT statements to build SMT instances whose satisfiability /validity helps to identify the behavioural similarity of two events [Koc+22].

Our approach compares simulation behaviour on a per-event basis, i. e., it will verify if an event in simulator A and simulator B share the same behaviour. With the definition of simulation behaviour explained in Section 5.2.3, this is the case if the events have the

```
// simulator S1
event A {
  reads Z.input
  schedules E with delay = 2 * (5 + input)
}
// simulator S2
event B {
  reads Z.input
  schedules F with delay = 10 + 2 * input
}
```

Listing 5.13: Schedules-Relationships with Identical Behaviour

same effect on the simulation world, i. e., they write the same values to the same attributes and schedule the same events.

First, we will introduce a representation of the behavioural concepts *event schedules event* and *event writes attribute* in SMT-LIB syntax. We will demonstrate the behaviour comparison on a simple example, and from there, we derive a general formula. When comparing behaviour between specifications of simulations S_1 and S_2 we assume that a structural isomorphism has already been found, i. e., for every attribute and event of S_2 there is an attribute or event respectively in S_2 with equal structural characteristics.

5.4.1 Comparing Schedules Relationships

For every pair of events E_a and E_b, we assume one schedules-relationship from E_a to E_b. Finding a bijection between the schedules-relationship from E_a to E_b in both simulator specifications will allow the developer to expand the following concepts to numerous schedules-relationships. This is feasible because each schedules-relationship's impact on the simulation world is self-contained and not dependent on the effects of any other schedules-relationships; however, this is not the case with writes-relationships. With this assumption and mapping of events, we can compare the (unique) schedules-relationship from event A to event E in simulator S_1 with the schedules-relationship from event B to event F in simulator S_2, where A and B as well as E and F need to be a structural match [Koc+22].

When we compare the scheduled events E and F the behaviour of event A and B is identical, if the delay- and condition-expressions are equivalent.

Consider the example depicted in Listing 5.13: The events A and B in simulator S_1 and S_2 respective schedule an event (that has been matched between the simulators S_1 and S_2) with slightly different delay expressions that always evaluate to the same value.

According to the representation of expressions in SMT-LIB syntax that we presented in Section 2.5, we can combine both delay specifications in a single SMT formula and declare them `delayA` and `delayB` to verify their equality. The SMT-LIB example is depicted in Listing 5.14. They are equivalent if they always evaluate to the same value for all possible

```
(declare-fun input () Double)
(declare-fun delayA () Double)
(declare-fun delayB () Double)

(assert (= delayA (* 2 (+ 5 input))))
(assert (= delayB (+ 10 (* 2 input))))
(assert (not (= delayA delayB)))
```

Listing 5.14: Example for Schedule Comparison

```
(declare-fun ...) // all read-attributes

(assert (not (= C_A C_B)))
(assert (not (= D_A D_B)))
```

Listing 5.15: General Schedule Comparison [Koc+22]

assignments of input variables, i. e., if the expression `(assert (= delayA delayB))` is *valid*, or equivalently if the negation of that expression is not *satisfiable*.

To compare conditions of the schedules-relationship, we define: Let C_A and C_B be the condition-expressions of the schedules-relationships from event A and B respectively and D_A and D_B the delay-expressions [Koc+22]. As a result, if the SMT statement depicted in Listing 5.15 is unsatisfiable, the behaviour of the schedules-relationships is identical.

A statement is satisfiable when the SMT solver finds an assignment of input variables where the condition- or delay-expressions show not identical values. This allows our approach to determine whether two events have the same behaviour and generate a mapping of attribute values (a subset of attribute states) to show how they match.

5.4.2 Comparing Writes Relationships

When comparing writes-relationships, we assumed that for schedules-relationships exist at most one schedules-relationship between one event and another. This assumption is plausible since schedules-relationships effects are independent. Additionally, a method for identifying schedules-relationships that are similar to those that satisfy the assumption can be expanded to support any number of schedules-relationships between two events. We cannot make a similar assumption for write-relationships. We assume that it exists one write-relationship in event A that writes to attribute C, at most. The combination of all write-relationships from A to C is the result of the effect of A on C. As a result, we cannot compare write-relationships separately. To illustrate our assumption, first, we present a single write-relationship that affects one attribute. Second, we derive a general statement that represents the concept. Listing 5.16 shows the statement, where two events affect an identical attribute: *waitingPassengers*. The presented conditions are not the same; however, the two conditions have the same effect on the attributes.

```
// simulator S1
event A {
  reads Z.waitingPassengers
  writes Z.waitingPassengers = 0
    when waitingPassengers != 0
}

// simulator S2
event B {
  reads Z.waitingPassengers
  writes Z.waitingPassengers = 0
}
```

Listing 5.16: Writes-Relationships with Identical Behaviour [Koc+22]

```
(declare-fun old () Int)
(declare-fun newA () Int)
(declare-fun newB () Int)

// writes-relationship of S1:
(assert (=> (not (= old 0)) (= newA 0)))
(assert (=> (= old 0) (= newA old)))

// S2, simplified, because the condition is always true:
(assert (= newB 0))
(assert (not (= delayA delayB)))
```

Listing 5.17: Example for the Write Comparison

The example in Listing 5.16 illustrates how the expression describing the new value of the attribute and the condition cannot be compared separately, as it is possible with schedules-relationships. We can use the SMT representation of writes-relationships introduced in Section 5.2.5 to combine both writes-relationships in a joint SMT formula using two variables to represent the two write-action outputs, as shown in Listing 5.17. A single variable is used for the old attribute value in the combined formula to assert that both write-functions access the same state of the simulation world. Similar to schedules-relationships, the joint formula will be satisfiable if there is a state of the simulation world for which both writes-relationships write different values to the same attribute, proving different behaviour.

Generalisation for the write comparison: The effect of an event A on an attribute C is the result of the combination of all write-relationships from A to C, each of which can contain a condition. Using the representation of multiple writes-relationships from A to C explained earlier, and this comparison can be extended to a general formula. Let $C_{A,1..n}$ and $C_{B,1..m}$ be those conditions of event A and B respectively and $F_{A,1..n}$ and $F_{B,1..m}$ the corresponding write-functions. Then the effect of A and B on the attribute is identical if the SMT formula shown in Listing 5.18 is not satisfiable:

```
(declare-fun old () Sort)
(declare-fun newA () Sort)
(declare-fun newB () Sort)
(declare-fun ...) // all read-attributes

// write-functions of event A
(assert (=> ($C_{A,1}$) (= newA $F_{A,1}$)))
...
(assert (=> ($C_{A,n}$) (= newA $F_{A,n}$)))
(assert (=> (not (or $C_{A,1}$ .. $C_{A,n}$)) (= newA old)))

// write-functions of event B
(assert (=> ($C_{B,1}$) (= newB $F_{B,1}$)))
...
(assert (=> ($C_{B,m}$) (= newB $F_{B,m}$)))
(assert (=> (not (or $C_{B,1}$ .. $C_{B,m}$)) (= newB old)))
(assert (not (= newA newB)))
```

Listing 5.18: General Example for the Write Comparison

5.5 Technical Contribution

In this section, we present the toolchain that allows analysis architects to specify the structure and behaviour of simulation components. Our toolchain follows our metamodel-based approach to specifying the structure and behaviour of simulation components presented in Section 5.2. We provide the analysis architect with a model-based textual editor for the specification of simulation components. The textual editor is based on our DSL to specify the structure and behaviour of simulation components presented in Section 5.2.4. Furthermore, we provide a tool that empowers the analysis architect to identify similar simulation components. In our toolchain, we implemented the structural and behavioural comparison according to Section 5.3 and Section 5.4.

5.5.1 Toolchain for Simulation Component Specification and Comparison

Our approach is separated into specifying and identifying simulation components; as a result, we also separated our toolchain accordingly. The first part of our toolchain is for specifying the structure and behaviour of simulation components. The second part of our toolchain is for comparing simulation components to find simulation components that are similar regarding structure and behaviour. Figure 5.4 displays the tools we created to realise simulation component specification and identification and the third-party tools we used in our toolchain. Our tools are depicted with black icons: the *Simulation Specification Editor*, the *Analysis CLI*, and the *Analysis Results*. The external tools we use in our toolchain are *Eclipse*, *EMF*, *Xtext* for the specification; *Neo4J* and *Docker* to store the specification; the *Z3 Solver* and *Neo4J* for the comparison. This section explains how we use and implement the toolchain in detail.

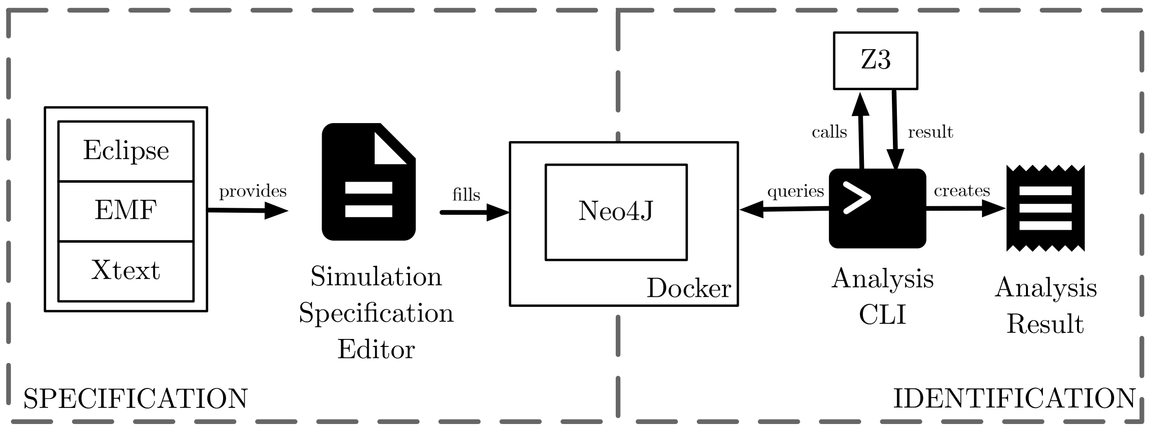


Figure 5.4.: Specification and Analysis Toolchain [KR22]

5.5.1.1 Specification of Simulation Components

The specification approach utilises the metamodels for specifying the structure (cf. Figure 5.1) and behaviour (cf. Figure 5.2) of simulation components. We created the metamodels in EMF. EMF provides graphical and textual editors to create such metamodels. Furthermore, it provides generators for creating code stubs of the metamodel classes, editing classes, and rudimentary tree editors. As shown in Figure 5.4, we use the EMF technology stack to create the simulation specification editors. Figure 5.5 depicts the tree editor with an example model. The tree editors are suited for small models and rapid prototyping; however, the tree editors are hard to navigate and understand. Therefore, we use Xtext to create a grammar for a textual editor. The grammar is presented in Section 5.2.4. Figure 5.6 shown the textual editor. The textual editor allows the analysis architect to specify structure and behaviour.

The analysis architect can work exclusively with the text editor; the model instances are created automatically. In the editor, each simulation component is stored in a `*.simspec` file, and the generated model files are stored in the `*.structure` files.

Each node in the editor has a distinct ID and name property. In addition to the ID and name, the developer can add a description to the root node, representing the simulation component. Entities and events are contained in the root node. Attributes on all entities are base datatypes like integers, booleans, arrays, or enums. Each event can relate to several attributes to express a reads relationship. The behaviour is represented independently to separate the structure from the behaviour. The structural and behaviour metamodels are built using the reference architecture for domain-specific modelling languages [HSR19]. This allows us to individually maintain and extend the metamodels while using an editor that accesses both metamodels. Writes attributes and schedules relationships comprise the behaviour. According to our metamodel is, each writing attribute associated with a single event. When the event is triggered, the writes attribute has a condition that, if true, changes the referenced attribute. The writes attribute also models how an attribute is changed.

Events can also schedule other events. Developers can add the schedules node to the tree editor to model event scheduling. A schedules node refers to the scheduling event and the event to be scheduled. The node also contains the condition and a reference to the attributes assessed to decide whether the simulation component plans an event. For the specification to be used for comparison, we convert it into a graph. As seen in Figure 5.4, the specification is saved in graph form in the graph database Neo4J[1]. Our tool has an interface for saving the specification to a database. For an easy setup of the Neo4J database, we propose running the database in a Docker container.

Although the Neo4J database is utilised to store the transformed specifications and execute the structural comparison, users can view each stored graph via the Neo4J UI. We recommend using the UI only for debugging purposes because the graphs are created from scratch for each analysis run. The graphs are created each time new because we want to avoid inconsistencies between the specification and the data stored in the Neo4J database. The graphs in Figure 5.7 depict six simulation components of varying complexity. The blue nodes represent a simulation component. The yellow nodes represent simulation component instances. The red nodes reflect simulation component events. The grey nodes are the simulation component's datatypes. Reads and writes are represented by arrows connecting events and datatypes. Arrows between events represent schedule-relationships as well. Besides the structural information that is depicted in Figure 5.7, the behavioural information is encoded as SMT statements that are annotated at the nodes and edges where necessary. Although the graphs in the Neo4J user interface can be modified, the specification in the tree editor cannot be automatically updated depending on the new graph. As a result, we advocate only using the text editor to amend the specifications.

5.5.1.2 Identification of Simulation Components

The specification of simulation components alone serves as a documentation tool. Using the specification for the identification of simulation components regarding their structure and behaviour, we present the second part of our toolchain: The second part of our toolchain accesses the graphs that are stored in the Neo4J database. We use these graphs to compare simulation components based on their specification. When we compare two simulation components, we use two approaches to first compare the structure and second, to compare the behaviour. As the first step, the tool performs a subgraph isomorphism analysis [Ull76], searching whether a graph can be part of another graph. To perform the graph-isomorphism analysis, we utilise a Neo4J plugin developed by Cordio [Cor22]. After the sub-graph isomorphism analysis confirms that the two graphs have structural similarities, the similarity analysis proceeds with the behaviour analysis. The annotations that store the behaviour information and the reads- and writes-relations of the metamodel are transformed into SMT statements based on the SMT-LIB standard[2]. We transmit the statements to an SMT-Solver to determine whether the behaviour is identical. We use for

[1] https://neo4j.com/
[2] https://smtlib.cs.uiowa.edu/

the behaviour analysis the *Z3 Theorem Prover* by Microsoft [Z3P19]. The theorem prover solves the SMT problems we extracted from the simulation component specifications.

5.5.1.3 Configuration

To compare simulation components based on their specification and to avoid invoking the subgraph isomorphism and behaviour analysis manually, we developed a CLI. Before starting the analysis, the tool must know where the Z3 Theorem Prover is located. The `PATH` variable must be extended to provide the location of our toolchain. We provide a command in the CLI to add the path to the Z3 installation to the `PATH` variable of the operating system; thus, the user must install the Z3 Theorem Prover manually. Our CLI provides the following command to provide the location of the Z3 installation:

```
sim-compare z3 <PATH TO libz3.dylib>
sim-compare z3java <PATH TO libz3java.dylib
```

Listing 5.19: Z3 Theorem Prover Setup

It depends on the used operating system whether the user must manually change `PATH` variable or use the CLI. Currently, the `z3` and `z3java` commands are tested for MacOS. Please consult the official Z3 website[3] or the GitHub page[4] for more information.

Neo4J can run locally or remotely; however, we assume the user has a standard Neo4J instance that runs locally. Suppose that is not the case; the user uses another IP address or username and password. In that case, the user must invoke the following commands to change the IP, username, and password according to their Neo4J installation:

```
sim-compare neoip <IP>
sim-compare neousr <USER>
sim-compare neopw <PASSWORD>
```

Listing 5.20: Neo4J Setup

5.5.1.4 Analysis Commands

The user can check which simulation components are saved in the Neo4J database; these are available for searching for similar simulation components. To display the available simulation components, the CLI can list all simulation components that are stored in the Neo4J database by name:

```
sim-compare list
```

Listing 5.21: List all Simulation Components

[3] https://www.microsoft.com/en-us/research/project/z3-3/
[4] https://github.com/Z3Prover/z3

The main feature of our toolchain is the search for similar simulation components in structure and behaviour. To compare two simulation components, the user can invoke the analysis with the following command:

```
sim-compare compare <SIM_A> <SIM_B>
```

Listing 5.22: Compare Simulation Components Command

This inconspicuous command consolidates the approaches to compare simulation components regarding their structure (cf. Section 5.3) and behaviour (cf. Section 5.4). Invoking the command for each simulation component can be tedious for the user when the database contains hundreds of simulation components. Thus, we recommend using the following command to search for one specific simulation component:

```
sim-compare list | xargs -L1 sim-compare <SIM_A>
```

Listing 5.23: Compare with all Available Simulation Components Command

The first part of the command lists all available simulation components that are stored in the Neo4J database. The list is piped into the sim-compare command, where `<SIM_A>` is the simulation component searched in the remaining available simulation components.

5.5.1.5 Analysis Results

After invoking the `sim-compare compare <SIM_A> <SIM_B>` command, the analysis result can have four outcomes. The first outcome is that the two compared simulation components do not match structurally. Listing 5.24 shows the result when the structure of the simulation component `SIM_A` is compared to the structure of the simulation component `SIM_B` and the subgraph isomorphism yields no result.

```
Compare SIM_A and SIM_B
No isomorphism between simulator graphs!
```

Listing 5.24: No Subgraph Found

Listing 5.25 shows an excerpt of the result when the subgraph isomorphism analysis was successful. Instead of the output `No isomorphism between simulator graphs!`, the analysis compares the subgraphs' mappings. The currently analysed mapping is indicated by the placeholder `n`, and the total number of mappings is indicated by the placeholder `m`.

```
Compare SIM_A and SIM_B
...
Testing mapping n out of m:
```

Listing 5.25: Successful Subgraph Analysis

After the subgraph isomorphism analysis, each mapping that yields a positive result for the subgraph isomorphism analysis (i. e., they are identical on a structural level) is analysed regarding the matching behaviour. As the subgraph isomorphism can yield more than one result, each result will be compared until the SMT-Solver finds a solution or the behaviour is not identical. Listing 5.26 shows the results for a mapping that is not identical (`SMT status:  NOT SATISFIABLE`). For each attribute that is compared, the CLI will print an info line like in line 4 of Listing 5.26.

```
1  Compare SIM_A and SIM_B
2  ...
3  Testing mapping n out of m:
4  Comparing 'XYZ␣writes␣demand' with 'ABC␣writes␣demand'
5  SMT status: NOT SATISFIABLE
```

Listing 5.26: Not Matching Behaviour

If the subgraph isomorphism analysis was successful and the behaviour is identical, the results show a mapping of the events and entities that yielded the result. Listing 5.27 shows the result of a successful subgraph isomorphism and behaviour analysis.

```
...
Testing mapping n out of m:
Comparing 'XYZ␣writes␣demand' with 'ABC␣writes␣demand'
Behaviour identical with mapping:
[Event] EventA = EventC
...
[Entity] EntityA = EntityZ
...
```

Listing 5.27: Matching Behaviour

5.6 Limitations

While our approaches for comparing the structure and behaviour of DES components have shown promising results, it is essential to acknowledge their limitations. In this section, we will discuss these limitations to provide a comprehensive understanding of our methods and their potential weaknesses.

In Section 5.6.1, we will discuss the limitations of the structure comparison approach. One limitation is that the structure comparison approach only considers isomorphic components with the same structure. However, components with different structures can have the same behaviour, which could result in false negatives. Additionally, the approach relies on the availability of a formal specification, which may only sometimes be the case. Furthermore, due to the computational cost of identifying isomorphism, the approach may only be suitable for a small and complex DES component.

In Section 5.6.2, we will discuss the limitations of the behaviour comparison approach. One limitation is that the approach relies on the availability of input-output traces, which may only sometimes be available or may be difficult to obtain. Additionally, the approach assumes that components with similar behaviour are functionally equivalent, which may not always be the case. Furthermore, the approach may not be able to detect subtle differences in behaviour that could be important in some applications.

By acknowledging these limitations, we hope to provide a clear understanding of the scope and applicability of our approaches. While these limitations may constrain the effectiveness of our methods in specific scenarios, they provide a solid foundation for future research in DES comparison and verification.

5.6.1 Limitations of the Structure Comparison

The first step in comparing DES components is to perform a structure comparison. This step is not intended to identify identical components but rather to filter the DES components with the same structure, i.e., isomorphic components. However, these isomorphic components can have different semantics and different behaviour.

For example, two DES components may have the same structure but different semantics. As a result, even though they may look identical, they have different meanings and functions. Similarly, two DES components with the same structure and semantics may exhibit different behaviour when subjected to the same input.

Using graph isomorphism is one way to preselect DES components based on their structure. Graph analysis can be used to check for isomorphism between DES components efficiently. This preselection step is important because comparing the behaviour of every available DES specification with the desired specification can require a lot of computation time due to the complexity of the task. By preselecting DES components based on their structure, the SMT solver can focus only on those likely to be behaviourally equivalent to the desired specification, which can significantly reduce the computation time of the behaviour comparison process.

In conclusion, performing a structure comparison is an essential first step when comparing DES components. Although isomorphic components may have the same structure, they can have different semantics and behaviour. Preselecting DES components based on their structure using the structural comparison can help avoid the need to compare the behaviour of every available DES specification with the desired specification, which can save time and computational resources.

5.6.2 Limitations of the Behaviour Comparison

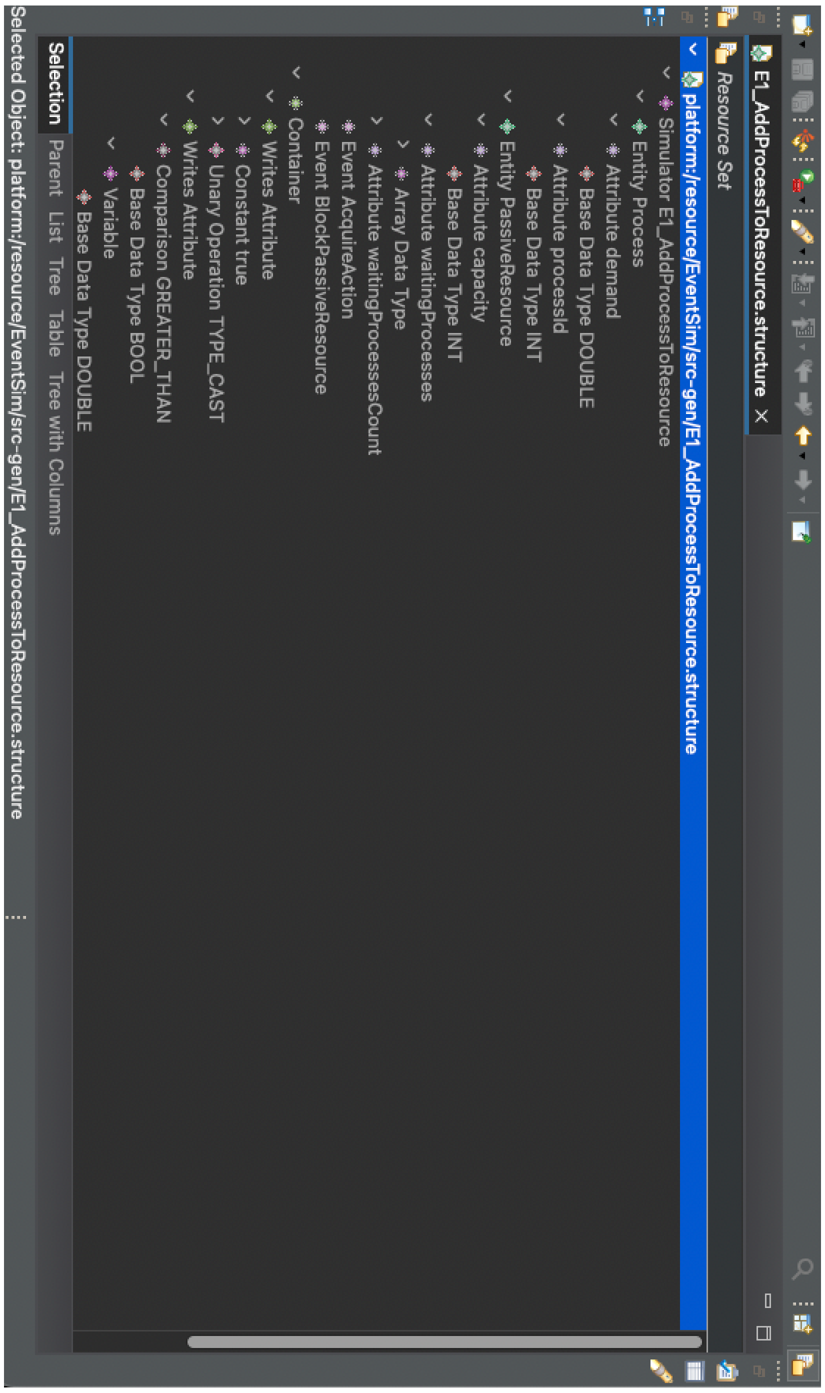
The time required for behaviour comparison of a DES component increases with the number of events and entities involved in the system. However, the preselection approach through structural comparison cannot guarantee that it only discards the DES components with the same behaviour. Nevertheless, preselection is still necessary to reduce the number of DES components that require behaviour comparison, thus reducing the overall computation time of the behaviour comparison.

It is essential to note that the behaviour specification focuses only on events and the variables that change due to those events. It does not consider computations of the DES components that are not affected by events. The specification approximates the behaviour of a DES component, which results in the behaviour comparison being a heuristic to reduce the search space for the developer. While this heuristic helps reduce the computation time, it is still the developer's responsibility to analyse the matching components to ensure their suitability for the desired purpose.

Furthermore, the approach's limitations in comparing the structure and behaviour of DES components need to be considered. One such limitation is the inability to identify complex behaviours due to the simplicity of the behaviour specification. Additionally, the structural comparison approach may need to be revised to identify the subtle differences between the components with different behaviours.

Another limitation could be the assumptions about the behaviour specification, which can result in the behaviour comparison being an approximation. As a result, the developer needs to be cautious when interpreting the results of the behaviour comparison.

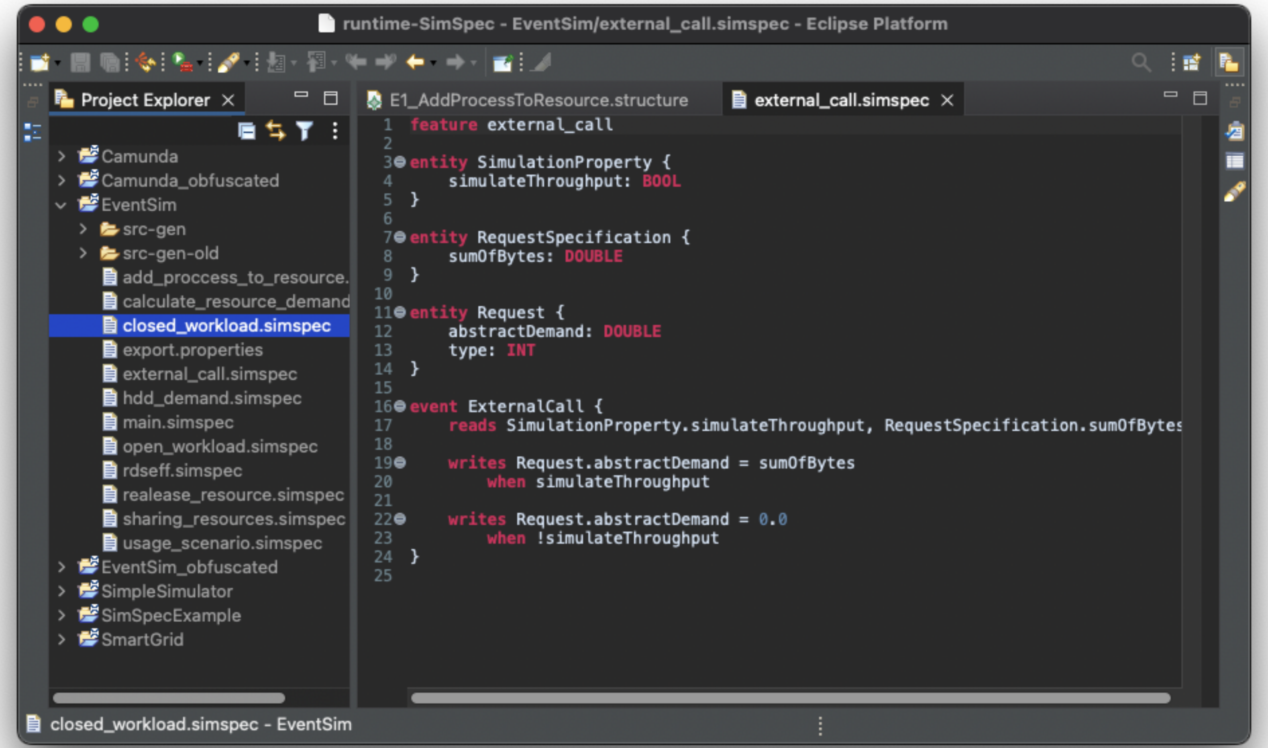
In conclusion, the preselection approach through structural comparison helps reduce the number of DES components that require behaviour comparison, thus reducing the computation time. However, the limitations of this approach and the approximations made in the behaviour specification need to be considered while interpreting the results of the behaviour comparison. Ultimately, the developer is responsible for analysing the matching components to ensure their suitability for the desired purpose.



```
platform:/resource/EventSim/src-gen/E1_AddProcessToResource.structure
  Simulator E1_AddProcessToResource
    Entity Process
      Attribute demand
        Base Data Type DOUBLE
      Attribute processId
        Base Data Type INT
    Entity PassiveResource
      Attribute capacity
        Base Data Type INT
      Attribute waitingProcesses
        Array Data Type
      Attribute waitingProcessesCount
    Event AcquireAction
    Event BlockPassiveResource
  Container
    Writes Attribute
      Constant true
      Unary Operation TYPE_CAST
    Writes Attribute
      Comparison GREATER_THAN
        Base Data Type BOOL
      Variable
        Base Data Type DOUBLE
```

Selected Object: platform:/resource/EventSim/src-gen/E1_AddProcessToResource.structure

Figure 5.5.: Simulation Specification Editor – Tree-Editor [KR22]



```
feature external_call

entity SimulationProperty {
    simulateThroughput: BOOL
}

entity RequestSpecification {
    sumOfBytes: DOUBLE
}

entity Request {
    abstractDemand: DOUBLE
    type: INT
}

event ExternalCall {
    reads SimulationProperty.simulateThroughput, RequestSpecification.sumOfBytes

    writes Request.abstractDemand = sumOfBytes
        when simulateThroughput

    writes Request.abstractDemand = 0.0
        when !simulateThroughput
}
```

Figure 5.6.: Simulation Specification Editor – Text-Editor [KR22]

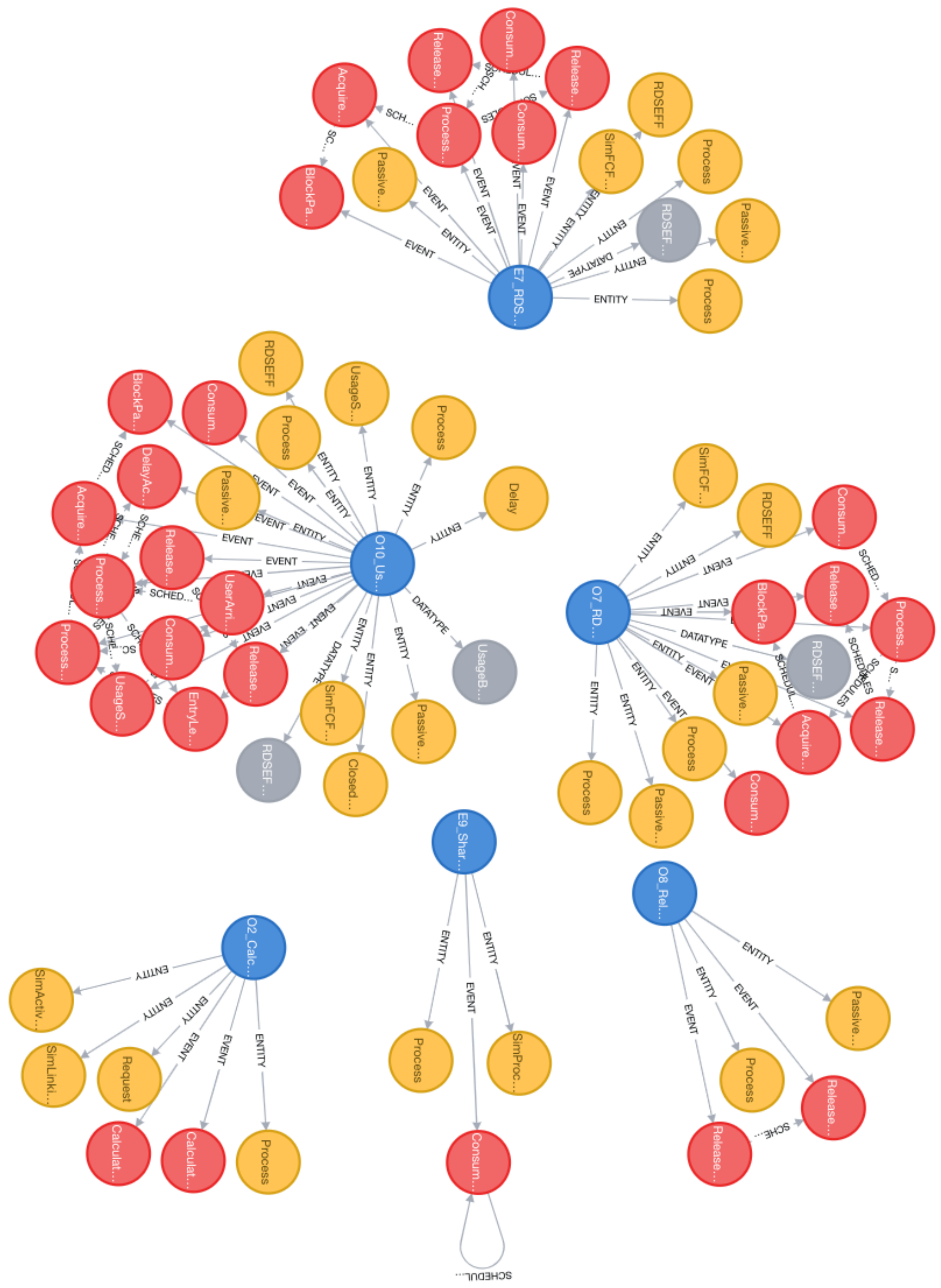


Figure 5.7.: Simulation Specification Graph Visualisation

Part III.

Validation

6. Case Studies

In this chapter, we introduce the case studies we use throughout evaluating our contributions. The research questions and metrics to validate our contributions are designed to work in the context of particular systems and particular cases. A suitable method for gathering the metrics and answers to our research questions is applying our approaches to such systems. According to Wohlin [Woh21] must, an empirical case study is a contemporary, real-world phenomenon. They must be actively developed to make case studies valuable for evaluation. Therefore, we decided to use four model-based analyses originating from different domains: *SimuLizar* [Reu+16], SimuLizar is a software architecture performance analysis tool based on the PCM; *Camunda BPM* [Gei+18], Camunda is a BPMN2-based workflow and simulation engine; the *Karlsruhe Architecture Maintainability Prediction for Automated Production Systems (KAMP4aPS)* [Hei+18], KAMP4aPS is an analysis for predicting the maintainability of automated production systems utilising change impact analysis; Moreover, the *Smart Grid Topology* (SmartGrid) [Ras+15], SmartGrid is a resilience analysis to reason about energy network grid topologies.

We use case studies to evaluate our three contributions. To evaluate our reference architecture for model-based analyses, we modularised the case studies so that they comply with the reference architecture. To evaluate the bad smells that arise from the co-dependency of DSMLs and model-based analyses, we searched the case studies for occurrences of bad smells. Further, we fixed the bad smells in the case studies to determine the impact on evolvability, understandability, and reusability. For evaluating our specification and reuse approach, we specified analysis components of the case studies and used the specification to apply our search approach.

In Section 6.1, we explain the criteria for selecting the model-based analyses. We present the case study SimuLizar in Section 6.2. The case study Camunda is presented in Section 6.3. KAMP4aPS is presented in Section 6.4, and the case study SmartGrid is presented in Section 6.5.

6.1 Selection Criteria

The selection criteria for the case studies are separated into four requirements. Our first requirement is independent of the contribution and the approach we want to evaluate. In order to be able to evaluate our contributions through representative case studies, we need to be able to change the source code of the model-based analyses. Therefore, the first and most important selection criteria the case studies must meet is that the model-based analyses must be publicly available as open-source software. Otherwise, we cannot test our hypothesis on real-world case studies.

Our second requirement is independent of the contribution and the approach we want to evaluate. The main research goal of this thesis is to improve the evolvability and reusability of model-based analyses. Ergo, the second selection criteria the case studies must meet is that the case studies use models as input, and the model must be based on a DSML.

Our third requirement concerns the first contribution, a reference architecture for model-based analyses, and the second contribution, bad smells in model-based analyses. We must be able to derive historical evolution scenarios from the case studies to determine the effect of our contributions on the evolvability of the case studies we investigate. In order to derive representative changes that happened during the development of the case studies, the developers of the model-based analysis must use some source code versioning.

The fourth requirement concerns the first contribution, a reference architecture for model-based analyses, and the second contribution, bad smells in model-based analyses. Our first and second contribution assumes that the DSML is modularised according to the reference architecture for DSMLs by Heinrich et al. [HSR19]. Hence, we focus on the four already modularised DSMLs: the PCM, the BPMN2, the KAMP4aPS metamodel, and the SmartGrid metamodel. Due to the pre-selection of metamodels, we could not influence the number of available analyses and the size of the case studies, which range from about 10.000 lines of code to more than 500,000 lines of code. Furthermore, we assume the corresponding analyses have decomposition potential due to the metamodels' decomposition potential. We investigated different kinds of model-based analyses in terms of new vs old, small vs large size, many vs few layers, and different domains.

6.2 The Palladio Simulator – Software Architecture Quality Prediction

The Palladio Simulator is an established software architecture quality analysis tool based on the PCM. It consists of three analyses (SimuLizar, SimuCom, and EventSim), each of which employs a distinct analysis approach and can make performance predictions based on the PCM.

The model-based analysis SimuLizar uses instances of the PCM to assume the modelled system's performance and reliability. Furthermore, SimuLizar can change the model and simulate reconfigurations of the system during runtime. SimuLizar is the most sophisticated and the best maintained of the three analyses; thus, we have selected it as a case study. SimuLizar represents a historically grown and versatile model-based analysis that can analyse multiple aspects of software quality.

The Palladio Simulator and the PCM are publicly available Open Source software. Heinrich et al. [HSR19] modularised the PCM so that it conforms to their reference architecture for DSMLs. Figure 6.1 shows the language components of the modular PCM. The modular

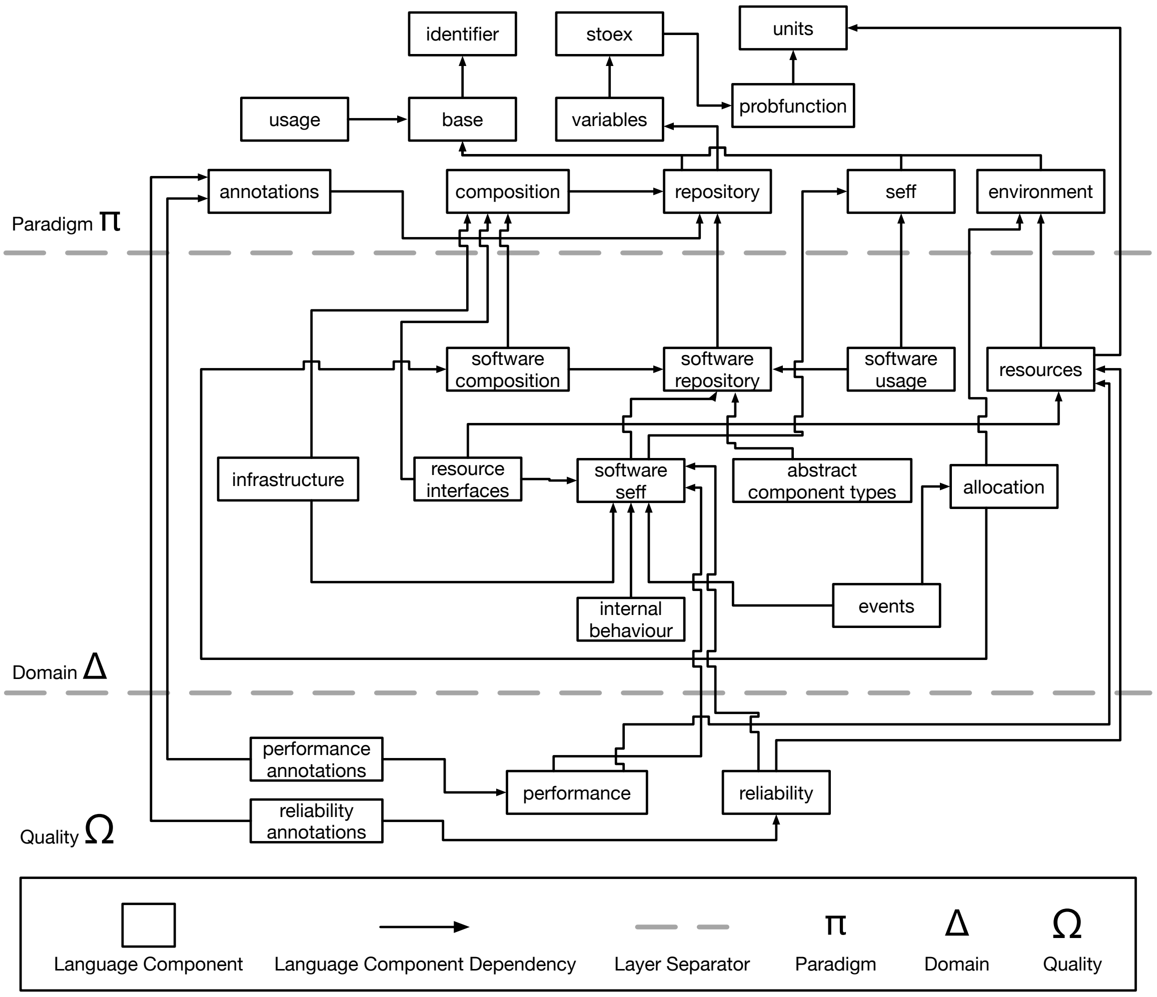


Figure 6.1.: Dependency Structure of the modular PCM [SHR18]

PCM consists of three layers; on the paradigm layer, they locate the essential language components needed to model software systems on an architectural level.

Paradigm Layer: The paradigm layer contains twelve language components. The *units*, *identifier*, *variables*, and *base* provide the basic building blocks for modelling software systems on an architectural level. The *seff* component allows the user to model the behaviour of entities of the DSML. For modelling the behaviour, statistical expressions are required; thus, the modular PCM provides the *stoex* language component. It depends on the *probfunction*, which allows the modelling of statistical distributions. To store and compose entities of the DSML, the modular PCM provides the *repository* and the *composition* component. The *composition* language component serves as a generic framework for all structures within the PCM. It incorporates the concept of *composition*, introducing a new superclass *Containable*, which serves as the base for all classes that can be included within a *ComposedStructure*. *AssemblyContexts* and *Connectors* are defined as *Containable* elements in this language component. The *repository* language component encompasses the fundamental abstractions of the repository view type. The *repository* component comprises Components, Interfaces, and their associated relationships (Roles).

Domain Layer: The domain layer contains the language components that are required for the domain of architecture and behaviour modelling of software systems. Therefore, the basic building blocks from the paradigm layer are extended to allow the user to model the architecture of software systems. The *software composition* extends the *composition* component; further specialisations are the *software repository* to store modelled software components, and the *software seff* and the *software usage* to model the behaviour of software components. The *software repository* language component extends its counterpart on the paradigm layer by incorporating domain-specific elements, including exceptions and interfaces that provide operations. It also defines an atomic component with an abstract class that serves as a generic extension point to specify the effects of services. While using this extension point, the language component for describing behaviour *software seff* is not limited to behaviour-specific specifications and can be utilised for other service effect specifications. As a result, the *software repository* language component is devoid of behaviour-related content. The *software repository* language component can be utilised to define software components, their interfaces, and operations. However, it is commonly used with the *composition* and *seff* language components. The *environment* component of the paradigm layer gets extended to allow the modelling of *resources* like Hard Disk Drives (HDDs) and Central Processing Units (CPUs), which can be *allocated* by different software components. The *resources* language component extends the functionality of the *environment* language component by incorporating hardware resource specifications into its containers and links. These specifications serve dual purposes; they can either be utilised solely for documentation or to simulate performance. These resources process the resource demands that can be extended into Service Effect Specifications (SEFFs). In addition to the dependency on the *environment* language component, the *resources* language component also relies on *units*. The *software composition* language component extends the concepts from the *composition* language component on the paradigm layer by domain-specific abstractions. It provides concrete implementations of abstract composition concepts through several classes, including System, CompositeComponent, SubSystem, and various Connectors. This language component is designed to be utilised with the *software repository* language component to describe the internal structure of ComposedStructures such as Systems and CompositeComponents. The *infrastructure* language component is an extension of the *seff*, *repository*, and *composition* view types. It introduces new abstractions, such as component types, interfaces, roles, connectors, and calls, referred to as *infrastructure*, to model middleware.

Quality Layer: On the quality layer, they annotated the two quality attributes *performance* and *reliability*. The quality layer introduces the analysis of two quality properties. First, it provides the *performance* and the *performance annotations* components that allows the user to analyse the performance of a software system. Second, it provides the *reliability* and the *reliability annotations* components that allows the user to analyse the reliability of a software system.

6.3 Camunda – Business Process Workflow and Simulation Engine

We selected the model-based analysis Camunda as our second case study. The analysis Camunda is a workflow and simulation engine that uses models of business processes. These business process models are based on the DSML BPMN2, developed by the Object Management Group (OMG). The BPMN2 standard serves as an ISO standard for modelling business processes. It provides symbols for domain experts to model and document business processes and workflows. The standard contains a formal description of the execution semantics for each model element. In the model instances, it is possible to compose and correlate events, supporting the description of human interaction in processes. The BPMN2 model is a type of flowchart which follows the tradition of programme sequence visualisation. The BPMN2 is related to Event-driven Process Chains (EPC), also used for modelling business processes.

The model-based analysis Camunda is also a software project developed as Open Source software; thus, it fulfils our first requirement for case studies. The developers of Camunda also use a source code versioning system, which fulfils our third requirement. It covers the additional domain of business process analysis, besides the standard BPMN2, it also supports the Case Management Model and Notation (CMMN 1.1) and the Decision Model Notation (DMN 1.1). Camunda is a fork of the free workflow management system Activiti, developed in 2010. In 2013 Camunda BPM was forked from Activiti as an open-source project by the company Camunda in Berlin. Our refactorings focus on the Camunda BPM Platform, consolidating the metamodel's dependencies. Due to the size of the Camunda BPM Platform (over 500,000 lines of code), we were unable to refactor it in a reasonable time frame; therefore, we focused our refactorings on our scenarios' affected components and files.

Figure 6.2 depicts an excerpt of the modular BPMN2 DSML [SHR18]. The DSML is separated into three layers, the paradigm layer, the domain layer, and a layer that does not correspond to the reference architecture for DSMLs by Heinrich et al. [HSR19].

Paradigm Layer: The paradigm layer contains the fundamental features for flow diagrams like *activities*, *resources*, or *flows*. Furthermore, it contains the *core* component, which implements fundamental concepts in BPMN2 modelling, including Definitions (root container for all models), RootElement (superclass for all primary concepts), Documentation, and BaseElement (provides ID and documentation reference). It also contains the language component *expressions*, which implements informal and formal expressions. Many BPMN2 concepts, such as Gateways, Subprocesses, Loops, Correlations, and Resources, use expressions to express conditions. The BPMN2 specification provides a *services* package that serves as a language component for modelling services. This package defines interfaces consisting of operations and service endpoints that can be extended externally. The *services* language component relies on the *messaging* component. Furthermore, this language component also has a transitive dependency on the *core* module. The DSML contains the *resources* language component on the paradigm layer for allocating resources

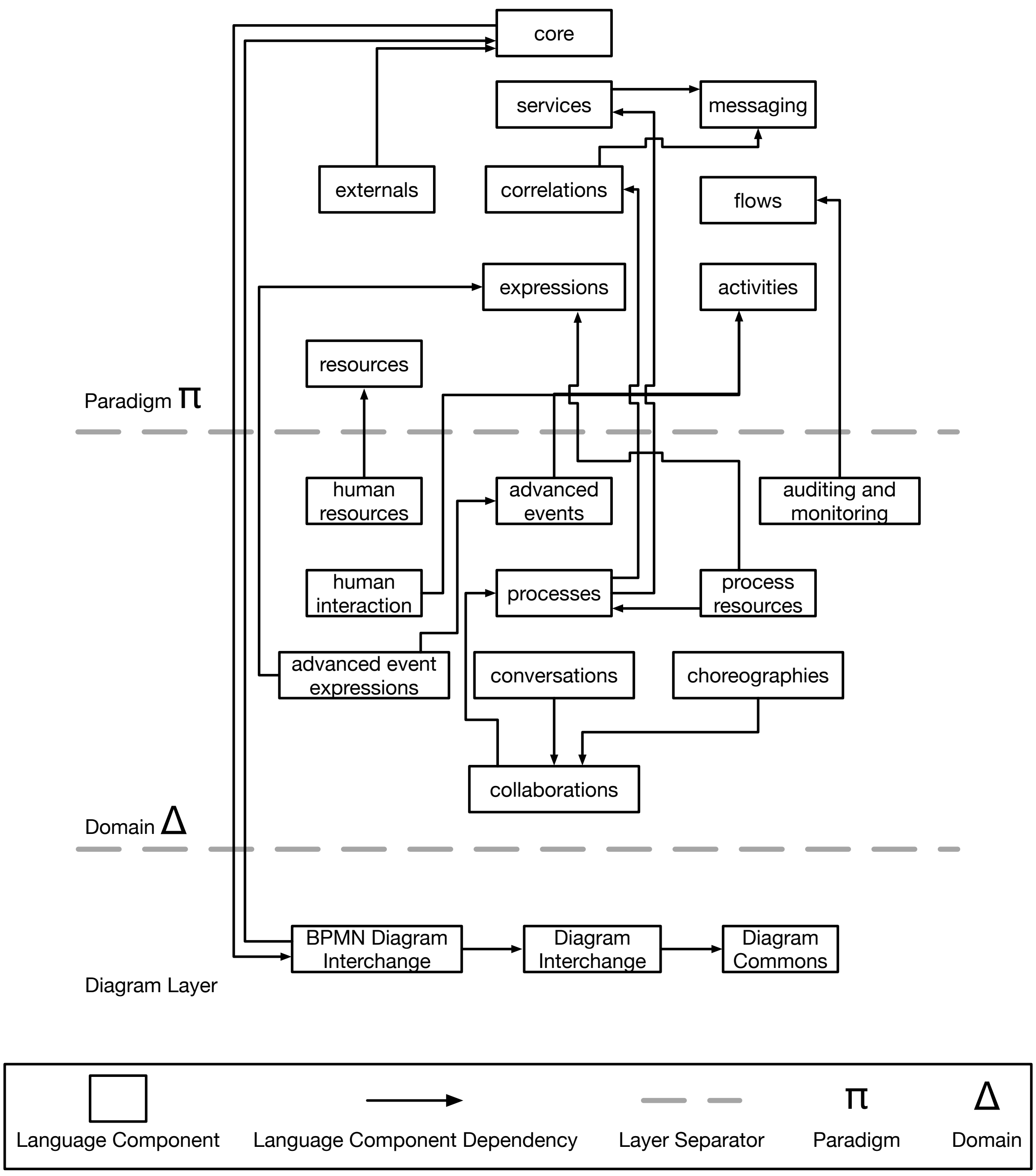


Figure 6.2.: Dependency Structure of the Modular BPMN2 DSML [SHR18]

to activities. According to the BPMN2 specification, *correlation* is utilised to link a specific *message* to an ongoing *conversation* between two process instances.

Domain Layer: The language components required to model business processes are on the domain layer. It allows modelling *human interaction* of activities and *human resources*. The language component *human resources* provides specific concepts related to human resources in BPMN2 modelling. Its sole dependency is on the *resources* language component. The concept of *processes* is introduced on the domain layer. It is part of the domain layer due to domain-specific properties. The *processes* depends on *artefacts* and *services*. The *advanced events* contains BPMN2 specific events that are too specific to be part of the paradigm layer. *Collaborations* are employed to express the interaction between *processes*. As such, the *collaboration* language component references the *process* language component. *Choreographies* are utilised to specify the sequential interaction between *processes*. The *choreographies* language component depends on the *collaborations* language component, as *choreography* is a specialisation of *collaboration*. *Conversations* provide an overview of participant interaction. They may reference *collaborations* between participants.

Diagram Layer: The diagram layer is an exception to the reference architecture for DSMLs; it contains exchange-specific information that should not be part of the DSML. However, because the DSML follows the BPMN2 standard, the features on the diagram layer remain in the DSML.

6.4 KAMP and KAMP4aPS – Change Propagation Analysis

We selected the model-based analysis KAMP4aPS as our third case study. The DSML used by the KAMP4aPS model-based analysis is a single-purpose DSML; its metamodel is only used by a single model-based analysis. Although the methodology [HBK18] of the KAMP-Framework is designed to support the domains of software systems [Ros+15], business processes [Ros+17], production systems [Hei+18], and the software of automated production systems that runs on Programmable Logic Controllers (PLCs) [Bus+18], each domain requires a dedicated analysis and metamodel. The analysis must contain the change propagation rules for the change scenarios that work on instances of their domain metamodel. For example, in the domain of automated Production System (aPS), the KAMP4aPS analysis contains the metamodel that allows the analysis user to model aPS.

The KAMP4aPS DSML was used as a case study by Heinrich et al. [HSR19]; as we need a modularised DSML for our evaluation, we will focus on the KAMP4aPS. In addition to the Palladio Simulator and Camunda, the DSML KAMP4aPS does cover an additional domain. Thus, we can further extend the diversity of our case studies. The KAMP4aPS metamodel and analysis has been under development since 2016; it contains six components, one of which consolidates the dependencies on the metamodel.

Figure 6.3 depicts an excerpt of the modular KAMP4aPS DSML. The DSML is separated into three layers, the paradigm layer, the domain layer, and the quality layer. The domain is further separated into three layers; due to the separation, the DSML architect can model the dependencies from a concrete aPS (Pick and Place Unit (PPU)) to the more generic aPS and the most generic Automated System (AS) domain.

Paradigm Layer: The paradigm layer contains the fundamental features for entity handling and modifications. It contains the *basic* language component, allowing model entities with a unique identifier and a name. The *modification marks* language component provides

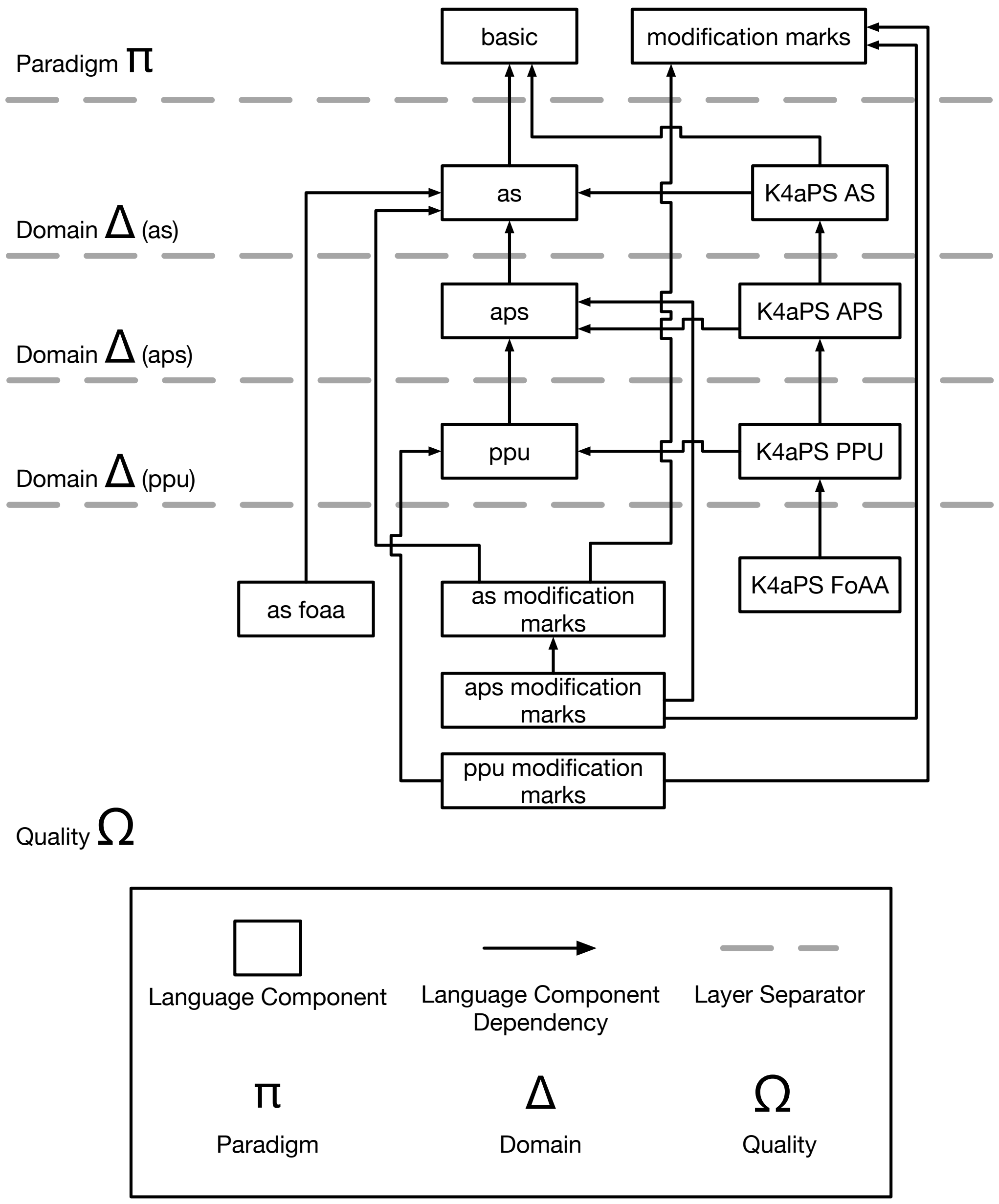


Figure 6.3.: Dependency Structure of the Modular KAMP4aPS DSML [SHR18]

the starting point for the change propagation analysis, where an arbitrary entity can be selected as the initial change.

Domain Layer: The domain layer contains the features required to model production systems. Strittmatter et al. [Str20] decided to separate the domain layer. They created three layers: one for the modelling of a wide range of automated systems (*as*); another layer for modelling specialised *as* automated production systems (*aps*). Moreover, the last domain layer allows modelling a pick-and-place-unit (*ppu*), a subset of automated production systems. The domain layer also contains the language components for modelling non-structural elements. In the context of the KAMP framework, these non-structural elements are called Field of Activity Annotations (FoAA). These FoAAs allow annotating additional elements like tests or documentation to the aPS elements. This allows refining the change impact analysis to consider artefacts that are not part of the aPS but are also affected by a change.

Quality Layer: On the quality layer are the features contained that allow the user to model modification information and further affected elements that are not part of the domain. The layer encompasses the *as foaa* language components, which implement modular design and solely incorporate the most generic concepts from the AS language component. Such elements are, for example, documentation or tests. All language components within the layer are situated here as they establish abstract representations required for evaluating the sustainability of a given automated system. Also, the quality layer introduces specification for the *modification marks* for the three domains (*as, aps,* and *ppu*).

6.5 SmartGrid – Energy Network Simulation

Like the model-based analysis KAMP4aPS, the model-based analysis SmartGrid also works with a single-purpose DSML. The SmartGrid energy network simulation performs an impact and resilience analysis. The metamodel is used to model topologies of smart grid energy networks. It also adds the domain of energy network analysis to our case studies; it is the second-youngest analysis; the development started in 2014.

Figure 6.4 depicts an excerpt of the modular SmartGrid DSML. The DSML is separated into three layers, the paradigm layer, the domain layer, and the quality layer.

Paradigm Layer: The paradigm layer contains the fundamental features for entity handling and graph notation. The *base* language component constitutes an abstract superclass and acts as a foundational element of all other language components. It inherits the attributes of both *name* and *ID* to its dependent components. Owing to its wide usage, dependencies on the *base* language component are not explicitly stated. Additionally, it is independent of any external components and has no incoming dependencies. Notably, the *base* language component is not a language construct and was factored out to serve the needs of multiple language components that required its functionality. The *graph* language component is an abstract representation that defines a basic network graph structure. The nodes in this graph are interlinked by logical and physical connections and can be attached to a power supply. This structure serves as the foundation for more complex network configurations, providing a clear and concise representation of the relationships between nodes in the network.

Domain Layer: The domain layer contains the features required to model topologies of different types. The *devices* language component provides a suite of device types tailored explicitly for use in smart grid systems. These device types are integrated into the network graph structure through subtyping. As a result, this language component is dependent on the *graph* language component. The *devices* language component provides a well-defined and specialised set of device types for use in smart grid networks. This language

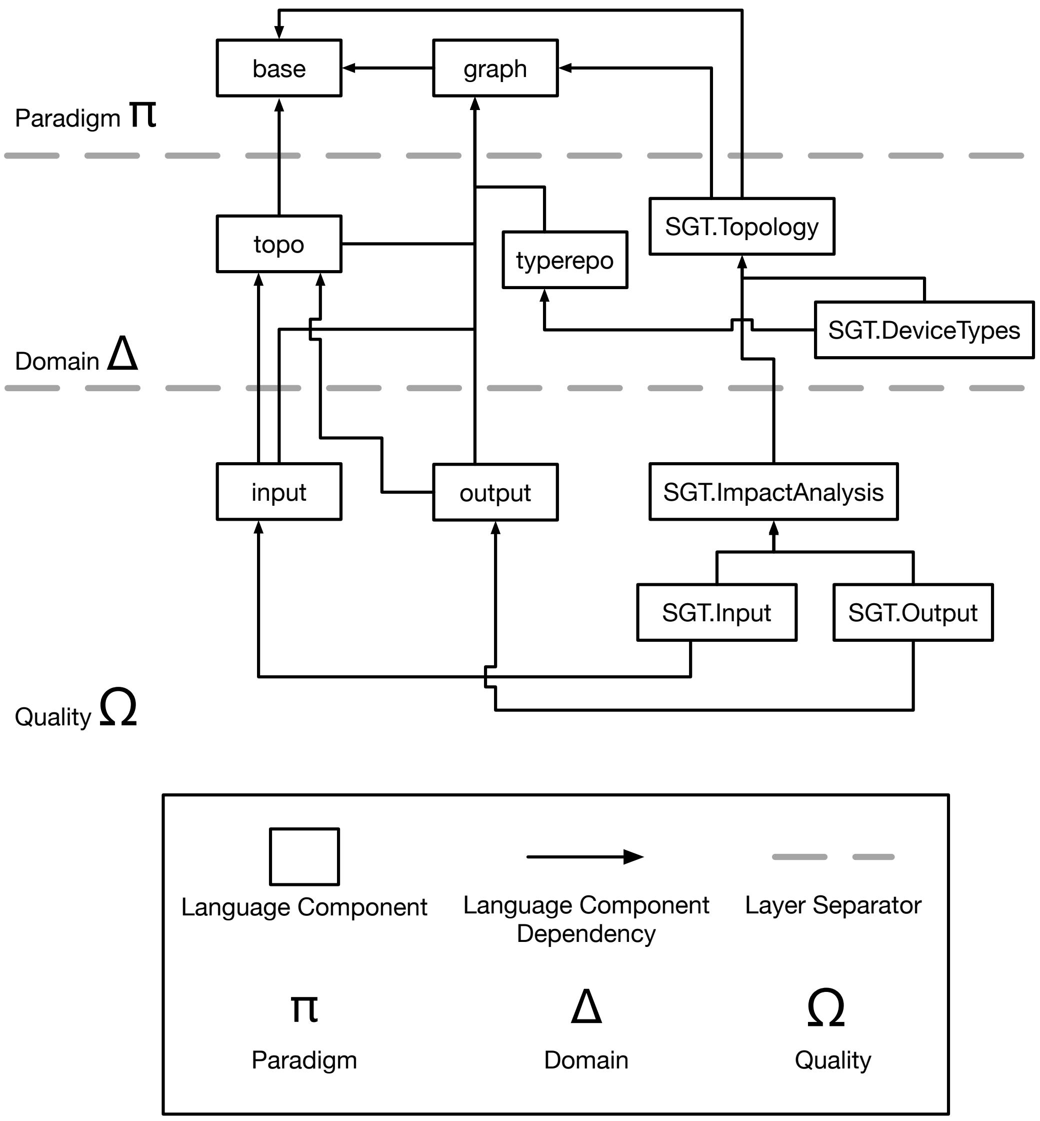


Figure 6.4.: Dependency Structure of the Modular SmartGrid DSML [SHR18]

component enables the construction of complex network configurations, where the various components of the grid are represented as interconnected devices. The *typerepo* language component extends the *graph* and *topo* language components. The types of the *typerepo* are stored in a separate repository, independent of any specific smart grid topology. The extended classes reside within the *topo* and *graph* language components. The *typerepo* language component provides a centralised repository for managing different types of smart grid components, enabling a clear separation of concerns and simplifying the maintenance and evolution of the smart grid system.

Analysis Layer: The features contained on the quality layer allow the user to model the *input* and *outputs* required for the analysis.

7. Reference Architecture Evaluation

In this chapter, we present the evaluation of our reference architecture for model-based analyses. In Section 7.1, we discuss whether our reference architecture fulfils the requirements, and in Section 7.2, we explain the goals and metrics for the evaluation. In Section 7.3, we present the design of our evaluation, and in Section 7.4, we present the results of the evaluation. In Section 7.5, we discuss the threats to validity. Finally, in Section 7.6, we summarise our findings and discuss the evaluation. In our evaluation, we use analysis components and not the analysis features for our discussion because we evaluate the source code of the case studies.

7.1 Discussion of the Requirements

In Chapter 6, we introduced the case studies we use throughout this thesis. Before we introduce our goals and metrics that show whether the requirements are met, we discuss the requirements that we can show are fulfilled by refactoring these four case studies.

The first requirement we discuss is **R2** (Non-intrusive Extension). During the refactoring process on the case study model-based analyses, we had to analyse the models that the system uses, identify the system's features, and then refactor the case studies to make them more modular and extensible. By analysing the models, we gained a deeper understanding of the case studies underlying structures, which have informed our decision-making during the refactoring process. Identifying the system's analysis features has helped us determine which parts of the system were only part of the model-based analysis and which parts represent the features of the corresponding DSML. We also identified when we had to introduce an extension inherited from more generic classes and analysis components. If the extension required to extend multiple classes, we had to introduce new interfaces, as Java does not allow multiple inheritance. As an interface alternative, we use aggregation or composition to introduce extensions. All these types of extensions have in common that they do not affect the more generic classes and analysis components, as none of the extended classes becomes altered. Thus, we can state that our reference architecture for model-based analyses meets the requirement **R2**, as extensions follow the structure of the DSML transitively.

The following requirement we discuss is **R3** (Consistent Dependencies). This thesis hypothesises that transferring the reference architecture for DSMLs to model-based analysis improves the evolvability, understandability, and reusability of the resulting model-based analyses. This hypothesis is based on the assumption that the concepts of the reference architecture for DSMLs provide a consistent and structured approach to modelling that can be transferred from one domain (DSMLs) to another (model-based analyses). To test this hypothesis, we refactored the case studies to follow the structure of their corresponding DSMLs. This involved making the structure of the DSML and its corresponding model-based analysis consistent, as outlined in Section 3.2. Through the process of refactoring the case studies, we were able to achieve a consistent structure of DSML and corresponding model-based analysis. This consistency demonstrates that the reference architecture for model-based analyses meets requirement **R3**, providing a structured approach to modelling that can be applied across different domains, leading to increased evolvability, understandability, and reusability of resulting models.

The third requirement we discuss is **R4** (Need-specific Reuse). Our reference architecture for model-based analyses uses feature models and the feature notation to modularise a model-based analysis. This way, the analysis architect can specify features of a model-based analysis. Each configuration (i. e., sub-graph) in our reference architecture can be reused in another, model-based analysis. An analysis architect can also use a configuration as an individual model-based analysis. A configuration of a model-based analysis is a valid subset of the whole feature model of the model-based analysis. This is only possible because the requirement **R2** (Non-intrusive Extensions) is met by our reference architecture; the only constraint is that the configuration must not depend on other configurations. Therefore, we can state that our reference architecture for model-based analyses meets the requirement **R4** (Need-specific Reuse).

The following requirement we discuss is **R5** (Need-specific Use). The analysis architect can also choose to extend a configuration to create a new or extend an already existing model-based analysis. Another possibility is that the analysis architect uses two configurations separately, for example, performance and reliability. If the analysis architect needs a performability analysis, they can combine these two configurations to create a new model-based analysis. Ideally, they can combine those two configurations; in a real-world example, they must add new features so that the two configurations can work together. Therefore, we can state that our reference architecture for model-based analyses meets the requirement **R5** (Need-specific Use).

The last requirement we want to discuss is **R1** (Improved Evolvability). Regarding the requirement **R1** (Improved Evolvability), we cannot determine whether it is satisfied by merely discussing the application of the reference architecture for model-based analyses to our four case studies, especially the evolvability of such complex systems, as our four case studies require additional research. Therefore, we present and run a GQM-based [CR94] evaluation to determine whether our reference architecture for model-based analyses improves the evolvability and understandability of our four case studies.

7.2 Research Goals and Metrics

We separated the goals and metrics section into three parts. The first part introduces our two research goals regarding the requirement **R1** (Improved Evolvability). The second part introduces our metrics to evaluate whether we reached our goals. Finally, the third part describes the structure of the scenario-based evaluation.

The first goal (**G1**) we derived from **R1** (Improved Evolvability) is:

Research Goal 7.1

We want to analyse whether our reference architecture for model-based analyses improves the evolvability of model-based analyses.

The second goal (**G2**) we derived from **R1** (Improved Evolvability) is:

Research Goal 7.2

We want to analyse whether our reference architecture for model-based analyses improves the understandability of model-based analyses.

For both research goals **G1** and **G2**, we use the four case studies introduced in Chapter 6. We compare the original, monolithic model-based analysis with its modular model-based analysis counterpart for each case study. We modularised each modular model-based analysis according to our reference architecture. Due to the size of the case studies, we focused on the metrics of the refactored scenarios. Modularising the whole model-based analyses is not feasible, as it does not alter the evaluation results (cf. Section 7.3). Also, we use the same metrics for both research goals **G1** and **G2** the same metrics to determine evolvability and understandability for both research goals.

We apply the properties of the software evolvability model by Breivold et al. [BCE08] to determine the evolvability of our approach. This model comprises the sub-characteristics of analysability, integrity, changeability, extensibility, portability, and testability. Regarding the ISO/IEC 25010 software quality model [ISO10], the characteristic of maintainability and portability map to the sub-characteristics of the software evolvability model. The sub-characteristics analysability, changeability, stability, and testability are part of the maintainability characteristic of ISO/IEC 25010, and the sub-characteristics of adaptability, installability, co-existence, and replaceability are part of the portability characteristic of ISO/IEC 25010. According to Briand et al. [BWL01], and Cruz-Lemus et al. [Cru+10], cognitive complexity affects the analysability and modifiability of software. To measure the cognitive complexity of a system, we refer to the amount of structural information within a system. We choose the same metrics as Heinrich et al. [HSR19] to measure the cognitive complexity of a system. They use the hypergraph metrics of Allen et al. [AGG07], which uses information size, complexity, and coupling to measure the information entropy of a software system. The formal definitions by Briand et al. [BMB96] are the foundation for the metrics by Allen et al. [AGG07].

The hypergraph metrics to evaluate our case studies are presented in Section 2.2.2.

7.3 Evaluation Design

In this section, we explain the reasoning behind the evaluation design. We explain which types of evolution scenarios we consider and how we selected the concrete evolution scenarios for each case study.

7.3.1 Evolution Scenarios

In general, we distinguish between modification changes and extension changes. However, these two changes are identical regarding the developer's effort. The analysis developer must understand the code base before modifying or extending the model-based analysis. Therefore, our evaluation does not distinguish between modification and extension changes to determine evolvability and understandability. An evaluation scenario represents changes in a model-based analysis. Ideally, each evolution scenario represents a change in one of the case studies. However, deriving real changes is sometimes possible (i. e., lost commit history, restricted access). Therefore, we distinguish three types of evolution scenarios. We present the types from the best (most representative) to the worst (least representative).

Historical Evolution Scenarios: The first type is the *historical evolution scenario*. A historical evolution scenario is derived from real changes in one case study. In the development history of a model-based analysis, analysis developers must adapt or extend a model-based analysis based on changes to the corresponding DSML. For example, when new features are added to the DSML, they can change existing components, resulting in a historical change. We searched the commit history of our four case studies to find such historical changes. The only constraint is that the change affects classes that depend on the DSML. It is sufficient if only one class has such a dependency; otherwise, our selection criteria are too strict about yielding results.

Potential Evolution Scenarios: The second type is the *potential evolution scenario*. Potential evolution scenarios were the first fallback when the search for historical evolution scenarios yielded insufficient results. If, for example, the commit history is lost, incomplete, or not accessible, the potential evolution scenarios can serve as an alternative to generating evolution scenarios. To generate a potential evolution scenario, the evaluation conductor must analyse the source code of the model-based analysis. They must identify classes that could be affected by a change. An example is to search for classes containing analysis algorithms with dependencies on language features of the DSML.

Random Evolution Scenarios: The third and last type is the *random evolution scenario*. The random evolution scenario is the last resort to generating evolution scenarios. If the conductor of the evaluation cannot access historical data and cannot identify potential evolution scenarios, they have to create random evolution scenarios. In such a scenario, the conductor randomly picks classes of the model-based analysis and groups them into one evolution scenario.

We created 40 evolution scenarios from historical data for our four case studies. Nonetheless, we decided to contain the potential and random evolution scenario as an option for further case studies. Our evaluation scenarios and raw results are available in our supplementary material [KHR22b] and our technical report [KHR22a].

7.3.2 Conduction of the Evaluation

We use the four case studies presented in Chapter 6 for the evaluation. For each case study, we selected ten evolution scenarios. The repositories of the case studies are all publicly available; thus we were able to extract ten historical evolution scenarios per case study. After we selected the historical evolution scenarios, we extracted the affected classes per scenario.

We refactored each scenario according to our reference architecture. For the refactoring, we used the refactoring techniques presented in Section 3.3.3. We also used our tool Refactor Lizar to fasten the refactoring process (cf. Section 3.6).

We compare the original evolution scenario to the refactored evolution scenario to assess whether the evolvability and understandability have improved. The refactoring can result in a different number of classes. We only consider the classes that would be affected by the change of the evolution scenario. For example, a change can affect only one line in a class method with over 500 lines of code. After the refactoring, we might have moved the method to another class; thus, the change would no longer affect the original class. As a result, we omit classes that are no longer affected by a change.

For each evolution scenario before and after the refactoring, we calculate the complexity, coupling, and cohesion according to Section 2.2.2. To calculate the metrics, we also use our tool Refactor Lizar (cf. Section 3.6). We present the results in the following section.

7.3.3 SimuLizar Refactoring

We started the modularisation with the release of version 4.3 of the Palladio-Simulator and used the modularised PCM presented in [HSR19; SHR18]. Before modularising SimuLizar, we had to exchange the PCM with its modularised counterpart, the modular PCM. In order to exchange the PCM, we had to change each dependency of SimuLizar from the PCM on the modular PCM. Changing the dependencies is necessary, as the modular PCM is not used in the Palladio-Simulator. After changing the dependencies, we analysed SimuLizar regarding the problems like the accumulation of dependencies, the scattering of dependencies, layer violations or cycles. We used our tool Refactor Lizar to find an accumulation of dependencies to identify which classes we have to separate the components into the three desired layers. The scattering of dependencies indicates which classes and components could be merged, as the refactoring of the accumulation of dependencies results in many small classes. The accumulation of dependencies analysis resulted in 18 occurrences, and the scattering of dependencies analysis resulted in 33 occurrences. The cycles and layer violations occurred during the refactoring; thus, we have no initial number of occurrences or fixes. First, we focused on accumulating dependencies of components that are supposed to be on different layers. Therefore, we applied horizontal-split refactoring to separate the analysis component in the layers π, Δ, and Ω, which resulted in three components. Then, we applied vertical-split refactorings to the three layers to separate the accumulation of dependencies still present in these layers. The final step was to merge the components where the language features were scattered over different classes and components. We could not fix all occurrences of the scattering of dependencies; for certain analysis operations, multiple language features are required. The model observing part of SimuLizar requires the *modelobserver* language feature and the *software usage* language feature. This resulted in nine components on π, 22 on Δ, and one on Ω. The component count increased from one component to 32 components. We reduced the number of accumulating dependencies from 18 to zero and the number of scattering dependencies from 33 to ten. In the following sections 7.3.4.1 and 7.3.4.2, we present detailed information about the modular structure of SimuLizar after refactoring. The details regarding the refactoring, especially the classes before and after the refactoring, can be found in our supplementary material [KHR22b].

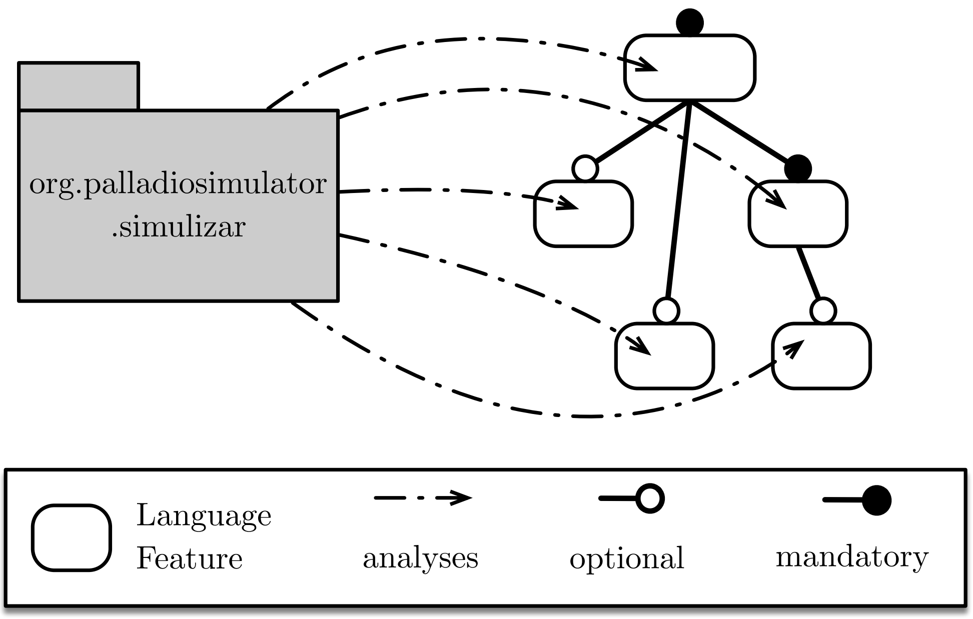


Figure 7.1.: SimuLizar Dependencies on the mPCM, simplified [KHR22a]

7.3.4 Modular SimuLizar– mSimuLizar

Figure 7.2 depicts the structure of SimuLizar after the modularisation. In Figure 7.2, we exclude the analysis components without representation in the language, e. g. events, the interpreter component, or the reconfiguration component, as most analysis components depend on them. Including these additional components renders the already complex figure unintelligible.

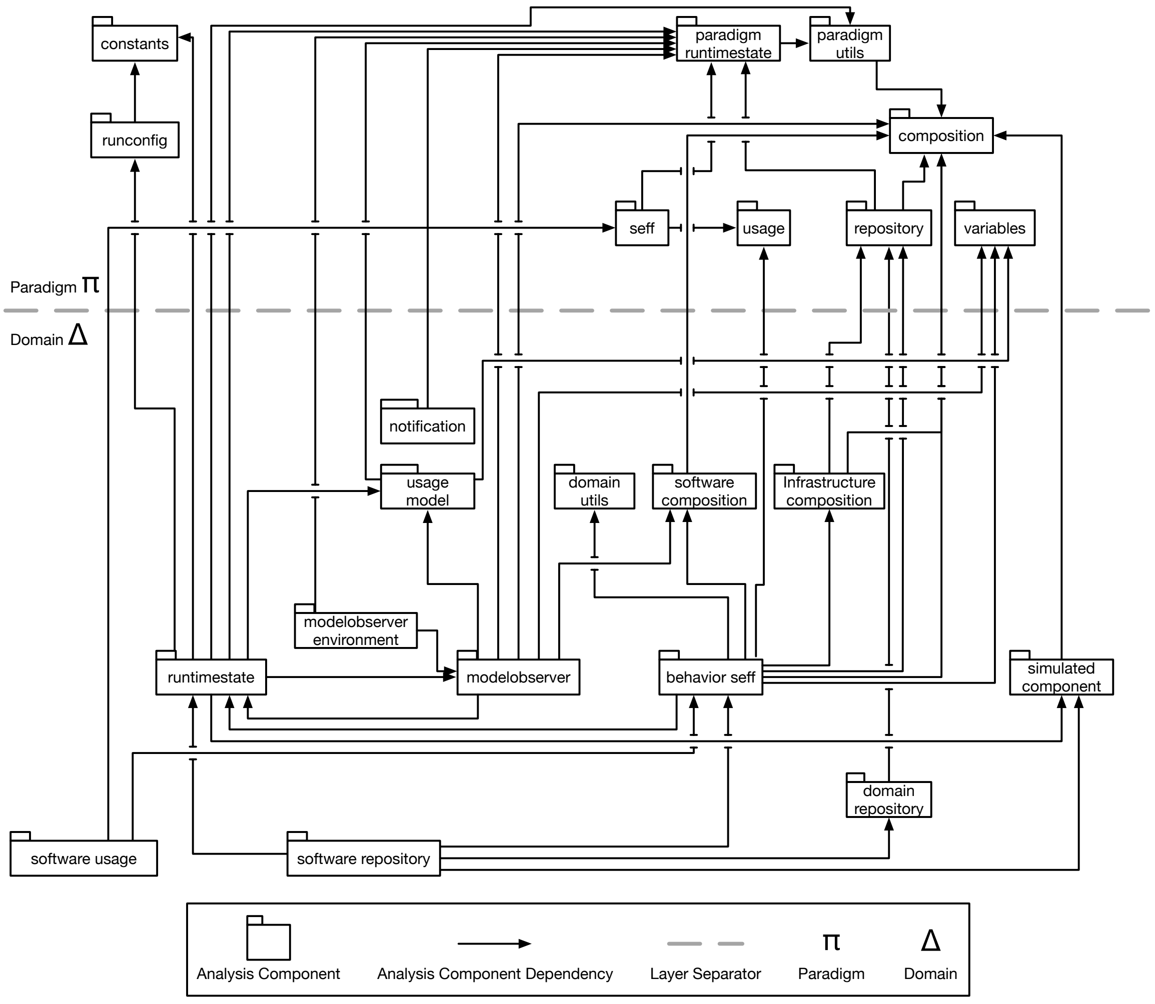


Figure 7.2.: Refactored SimuLizar, simplified

7.3.4.1 Paradigm Layer

Composition: The *composition component* handles the assembly of resources of the PCM. On the paradigm layer, the functionality of the composition component is prepared to handle any resources. The assembly of component types includes the preparation of resources. Preparing a resource means setting the context and the context hierarchy of the resource. The composition component provides functionality for adding or deleting a resource and the connectors required to compose resources.

Constants: The *constants* component provides the constants required by the analysis of all PCM instances.

Repository: The *repository* component on the paradigm layer manages the roles defined in the PCM. The PCM defines required and provided roles for components. In this component, the roles, e. g. provided and required roles are managed. It provides interfaces to receive these roles, and also it provides interfaces to receive the signatures defined in the PCM. The central portion of the repository component is the *repository switch*. The switch contains the interpretation of the roles. It also contains the analysis code concerning the required and provided roles. The signatures are implicitly used throughout the analysis code.

Runtimestate: The *runtimestate component* provides abstract classes and interfaces for managing the state of the analysis. It holds the PCM instance, the event notification helper, and a registry of the analysed components. The *component registry* is an interface for validating whether a component is available for analysis. It also provides *add* and *fetch* operations for the PCM components. The *event notification helper* is an interface for firing events and removing listeners.

Seff: The SEFF in the PCM represents the basic actions of a component. The *seff component* provides the interpretation and the analysis code for the elements of the seff language feature of the PCM. The seff component contains the interpreter for the seff types. For each seff type, the seff component contains the analysis code required for the elements.

Usage: The *usage component* provides the handling of probabilities defined in the usage language feature of the PCM. Probabilities are required when the analysis encounters a branch. The usage component determines in which direction the analysis must proceed. Besides branches, the usage component also provides the scheduling of delays. Another part of the usage component is the handling of loops. Based on the size of a loop, the usage component determines the time required to finish the loop. Furthermore, the usage component provides an interface to manage user actions.

Variables: The *variables component* provides the evaluation of the model instance. It creates an evaluator instance containing the variable characterisation of the PCM and the model evaluator. The evaluation provides a condition checker, which checks whether a boolean expression in a condition holds. The variable component also provides the generation of random variables.

7.3.4.2 Domain Layer

Behaviour Seff: The *behaviour seff* component provides the analysis code for the PCM model elements *external call action*, *acquire action*, *collection iterator action*, *set variable action*, and *release action*. The analysis code requires information about the *infrastructure*; thus, the dependencies remain on the infrastructure language feature in this component. The behaviour seff component also provides analysis code determining probabilistic transitions when encountering branches.

Domain Repository: The *domain repository* component provides an interface for implementing the analysis code for the PCM model elements *provided role* and *signature*.

Infrastructure Composition: The *infrastructure composition* component provides the analysis code for the PCM model elements *assembly infrastructure connector* and *required infrastructure delegation connector*. The component utilises the composition and repository component of the π layer.

Modelobserver: The *modelobserver* component provides the analysis code for the PCM model elements: *communication link resource specification*, *linking resource*, *processing resource specification*, *resource container*, *workload*, *closed workload*, *open workload*, and *usage scenario*. In addition to the modelobserver language feature, the component requires the *software usage* language feature; thus, it holds dependencies on PCM types of these two language features.

Modelobserver Environment: The *modelobserver environment* component provides the analysis code for the PCM model element *resource environment*. This component handles the modelobserver component and provides observers for the said model and the resource environment.

Notification: The *notification* component provides the analysis code for the PCM model elements: *operation provided role*, *operation signature*, *external call action*, *entry level system call*, and *usage scenario*. This component has dependencies on four language features to perform the analysis.

Runtimestate: The *runtimestate* component provides the analysis code for the PCM model elements *resource environment*, and *assembly context*. The runtimestate component has only two dependencies on two language features, but it consolidates the state of the analysed system. It utilises direct knowledge (i. e., usage model component), or it utilises the modelobserver component to manage the runtime state of the analysis.

Simulated Component: The *simulated component* provides the analysis code for the PCM model element *passive resource*. It represents two types of components mSimuLizar can analyse. The first component is a basic component that can be monitored, and it can acquire and release resources. The second component is a composite component, consisting of a set of basic components.

Software Composition: The *software composition* component provides the analysis code for the PCM model elements: *assembly connector*, *required delegation connector*, and *composite component*.

Software Repository: The *software repository* component provides the analysis code for the PCM model elements *basic component* and *service effect specification*.

Software Usage: The *software usage* component provides the analysis code for the PCM model elements: *entry level system call*, *usage scenario*, and *usage switch*.

Usage Model: The *simulated component* provides the analysis code for the PCM model elements: *usage model*, *usage scenario*, *workload*, *closed workload*, *open workload*, *software usage package*.

Before the refactoring, the state of SimuLizar was that all dependencies on the metamodel PCM were consolidated in one analysis component. First, we applied horizontal-split refactoring to separate the analysis component in the layers π, Δ, and Ω, which resulted in three components. Then, we applied vertical-split refactorings to the three layers to separate the language blobs still present on these layers. This resulted in 9 components on π, 22 components on Δ, and 1 component on Ω. The analysis grew from 21 components to 52 components.

7.3.5 SimuLizar Historical Evolution Scenarios

The third requirement for the case studies is that the developers of the model-based analysis use some source code versioning. In the case of SimuLizar, the developers first used *Apache Subversion (SVN)* and then they migrated the source code to *git*. The commit history was migrated from SVN to git; thus, we had access to the whole commit history. The source code is available on GitHub[1]. In Section 7.3.5, we provide an overview of the historical evolution scenarios extracted from the commit history of SimuLizar. Tor transparency reasons, we also provide the commit hash and the number of affected files if someone wants to recreate the change scenarios themselves.

7.3.6 Camunda Refactoring

Before we modularised Camunda, we had to exchange the BPMN2 DSML with its modularised counterpart the modular BPMN2 (mBPMN2) [HSR19; SHR18]. In order to exchange the BPMN2 DSML, we had to change each dependency of Camunda from the BPMN2 DSML on mBPMN2 DSML. Changing the dependencies is necessary, as the mBPMN2 is not used in Camunda. The turquoise nodes in Figure 7.3 are the modules that had to be modified. The dependencies of the Camunda BPM Platform regarding the mBPMN2 metamodel are similar to the structure shown in Figure 7.1. In the *org.camunda.bpm.model* module is the dependencies on the mBPMN2 metamodel consolidated. The details regarding the refactoring, especially the classes before and after the refactoring, can be found in our supplementary material [KHR22b].

7.3.7 Modular Camunda – mCamunda

Figure 7.4 depicts the structure of Camunda after the modularisation of the scenarios. We did not refactor the whole analysis. Therefore, we present only the components of Camunda that are affected by our refactoring and relevant for our calculation of the metrics complexity, coupling, and cohesion.

[1] https://github.com/PalladioSimulator/Palladio-Analyzer-SimuLizar

Scenario No.	Name	Commit ID	No. Affected Files
Scenario 01	RepositoryComponentSwitch Extension	75421342	4
Scenario 02	Deleted ModelAccess Class	534d5521	28
Scenario 03	Fix Project Structure	02511a37	5
Scenario 04	Explicitly Switch Based on Superclass	d9735115	3
Scenario 05	Add Monitor Repository to Feature Dependencies	(SVN) r34181	4
Scenario 06	Fixed Metadata for the HDD Patch	(SVN) r33820	2
Scenario 07	Include New Aggregation Plugin	(SVN) r32804	7
Scenario 08	Only Record Runtime Measurements	(SVN) r32416	25
Scenario 09	Generalized Response Times Aggregator	(SVN) r32166	4
Scenario 10	Add Missing Reconfiguration Rule	(SVN) r31800	6

Table 7.1.: Overview of the Historical Evolution Scenarios of the SimuLizar Case Study for the Evolvability and Reusability Evaluation of the Reference Architecture for Model-based Analyses

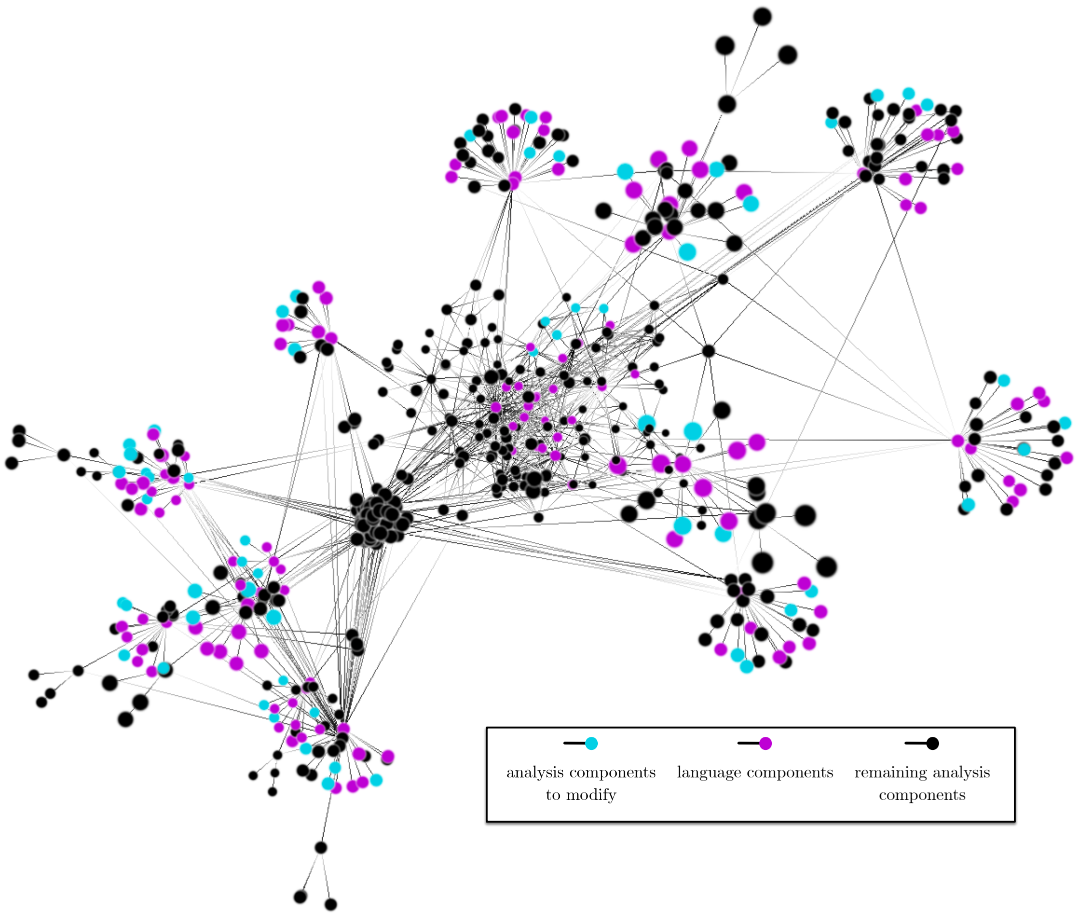


Figure 7.3.: Camunda BPM Platform Dependency Structure

7.3.7.1 Paradigm Layer

Core: The *ecore* component of Camunda contains the *BaseElement* class. It also provides a builder to create elements in the analysis context. We placed the *BPMN2* class also in this component, as it defines and, thus, contains all identifiers of the BPMN2 entities. Furthermore, variables are also defined in this component.

Flows: The *flow* component provides the notion of flows and classes to build flows. We placed the *AbstractFlowNodeBuilder* in this component. In addition to regular flows, the component also handles the sequencing of flows.

Messaging: The *messaging* component provides the foundation for the messaging in the analysis. It allows the user to build sender and receiver in the context of the Business Process Modeling Notation (BPMN) analysis. We had to refactor the *AbstractSendTaskBuilder* and the *AbstractReceiveTaskBuilder*.

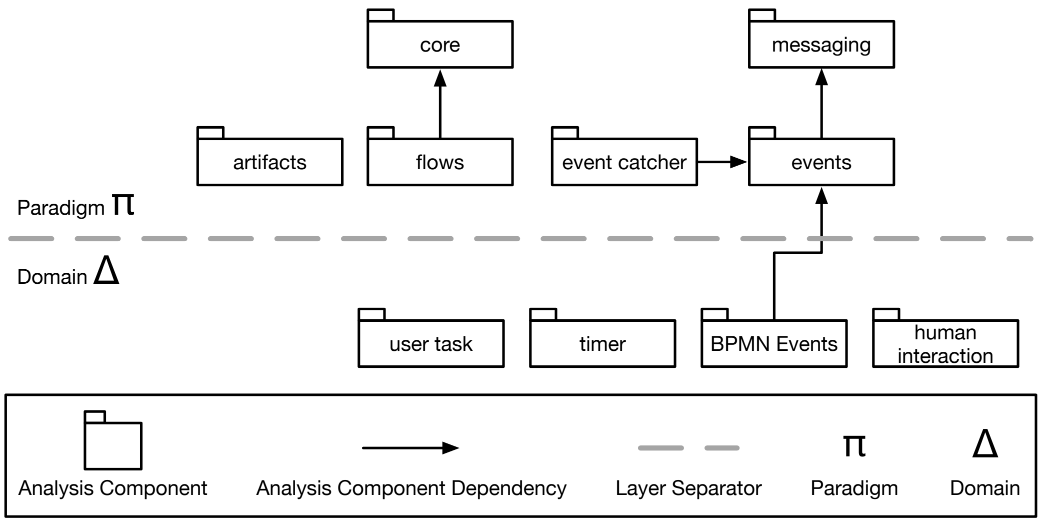


Figure 7.4.: Refactored Camunda, simplified [KHR22a]

Events: In the *events* component were the most changes located. In total, we had to refactor 98 classes across all ten scenarios that are correlated to this component. A class could be part of multiple scenarios; therefore the number of 98 classes represents not the number of different classes. The events component defines events in the context of the BPMN analysis. The *StartEvent*, *EndEvent*, and *ThrowEvent* are related to this component.

Event Catcher: The *event catcher* component is a utility component that manages the aggregation of events in the analysis.

Artefacts: The *artefacts* component is the abstract representation of model elements in the analysis. In the context of our refactoring, we had no classes that belonged to that component; however, the *AbstractFlowNodeBuilder* class located at the flows component hat dependencies to the language component *artefacts* that we were unable to refactor.

7.3.7.2 Domain Layer

User Task: The *user task* component is only part of the model-based analysis and is not part of the DSML. This component defines the analysis task and correlates to the tool user.

Time: The *time* component represents the notion of time in the analysis context. This component is only part of the model-based analysis and not part of the DSML.

Advanced Events: The *advanced events* component represents specialised events that are tailored to the domain of the business process analysis. Thus, it has dependencies of the language componentadvanced events.

Human Interaction: The *human interaction* component integrates the modelled human interaction into the analysis.

7.3.8 Camunda Historical Evolution Scenarios

The third requirement for the case studies is that the developers of the model-based analysis use some sort of source code versioning. In the case of Camunda, the source code is versioned with git. The source code is available on GitHub[2]. In Section 7.3.8, we provide an overview of the historical evolution scenarios we extracted from the commit history of Camunda. Tor transparency reasons, we also provide the commit hash and the number of affected files if someone wants to recreate the change scenarios by themselves.

7.3.9 KAMP4aPS Refactoring

Before we modularised the analysis KAMP4aPS, we had to exchange the KAMP4aPS DSML with its modularised counterpart, the modular KAMP4aPS DSML [HSR19; SHR18]. In order to exchange the KAMP4aPS DSML, we had to change each dependency of the KAMP4aPS analysis from the KAMP4aPS DSML on the modular KAMP4aPS DSML. Changing the dependencies is necessary, as the modular KAMP4aPS DSML is not used in the analysis KAMP4aPS. The dependencies of the analysis regarding the modular metamodel are like the structure shown in Figure 7.1. The dependencies on the modular KAMP4aPS metamodel are consolidated in the KAMP4aPS module.

The details regarding the refactoring, especially the classes before and after the refactoring, can be found in our supplementary material [KHR22b].

7.3.10 Modular KAMP4aPS – mKAMP4aPS

Figure 7.5 depicts the structure of the analysis KAMP4aPS after the modularisation of the scenarios. We did not refactor the whole analysis. Therefore, we present only the components of Camunda that are affected by our refactoring and relevant for our calculation of the metrics complexity, coupling, and cohesion.

7.3.10.1 Paradigm Layer

Activity: The *activity* component represents the user's activity to change the desired domain or system. KAMP4aPS analyses the domain of aPS; thus, the extension will handle the activities that are required to perform changes in an aPS. None of the classes located on the Δ or Ω layer were included in the scenarios; thus, the activity component has no incoming dependencies.

Workplan: The *workplan* component provides the functionality to derive and create a work plan. The components on the domain layer are specialisations of types that are analysed to create a work plan.

[2] https://github.com/camunda/camunda-bpm-platform

Scenario No.	Name	Commit Hash	No. Affected Files
Scenario 01	Add Timeout Task Listener	d53583a1	8
Scenario 02	Introduce Error Message	1db5469e	2
Scenario 03	Add Variable Specification	14ad97ae	7
Scenario 04	Remove Incremental Intervals Property	a337b8f6	10
Scenario 05	Set Marker in Exclusive Gateway	7cf3cdff	2
Scenario 06	Removed Error Message Attribute	4a5d7bc7	11
Scenario 07	Added Error Definition Variables	31e9a132	18
Scenario 08	Add Convenience Methods	1d2a508c	7
Scenario 09	Message with the Fluent Builder	677b3c6b	7
Scenario 10	Add Support for Connector Extension	c30dbc8e	6

Table 7.2.: Overview of the Historical Evolution Scenarios of the Camunda Case Study for the Evolvability and Reusability Evaluation of the Reference Architecture for Model-based Analyses

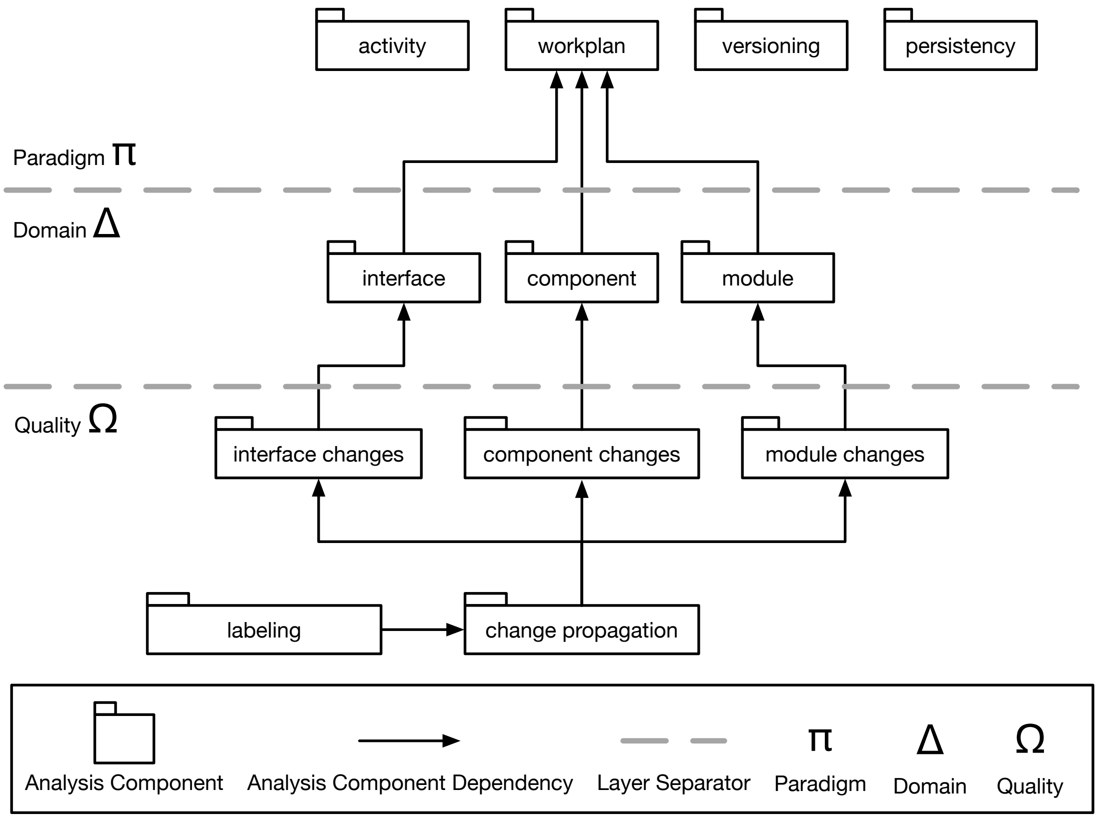


Figure 7.5.: Refactored KAMP4aPS, simplified [KHR22a]

Versioning: The KAMP4aPS analysis provides the functionality to analyse different versions of a system to derive changes. Although we had to refactor a class part of this component, it had no significant role in the evolution scenarios. Thus, the versioning component has no incoming dependencies.

Persistency: The *persistency* component allows the tool user to generate task lists containing the derived tasks necessary to perform changes in the analysed system. Although we had to refactor a class part of this component, it had no significant role in the evolution scenarios. Thus, the persistency component has no incoming dependencies.

7.3.10.2 Domain Layer

Interface: The *interface* component contains aPS interfaces. For example, it does contain the *BusInterface* with specialised classes for a different bus system that can be used in an aPS. Besides the bus types, it also contains mechanical interfaces like screws and bolts.

Component: The *component* component contains the basic building blocks of an aPS. Such building blocks are, for example, sensors and actors that are used in a production plant.

Module: The *module* component is a container that can contain other aPS components or modules. The differentiation is required by the tool user, as they have to distinguish aPS components from modules that can also be complex entities like a conveyor belt or a robotic arm.

7.3.10.3 Quality Layer

Interface Changes: The *interface change* component contains the change propagation rules for all interfaces in an aPS. These rules determine if an interface has to be changed, which can also affect other entities connected to the interface.

Component Changes: The *component change* component contains the change propagation rules for all aPS components. These rules determine if a component has to be changed, and other entities connected to the component can also be affected by the change.

Module Changes: The *module change* component contains the change propagation rules for all modules in an aPS. These rules determine if a module has to be changed, and the change can also affect other entities connected to the module.

Labeling: The *labelling* component annotates roles and documents to tasks. It also can annotate the estimated time required to perform a task. The estimated time is required when the tool user analyses multiple change scenarios. If the costs are annotated, they can select the most cost-effective scenario.

Change Propagation: The *change propagation* component coordinates the initial changes. The initial changes determine where the starting point of the analysis is located. It also coordinates the change impact analyses between the modules, components and interfaces.

7.3.11 KAMP4APS Historical Evolution Scenarios

The third requirement for the case studies is that the developers of the model-based analysis use some source code versioning. In the case of KAMP4aPS, the source code is versioned with git. The source code is available on GitHub[3]. In Section 7.3.11, we provide an overview of the historical evolution scenarios extracted from the commit history of KAMP4aPS. Tor transparency reasons, we also provide the commit hash and the number of affected files if someone wants to recreate the change scenarios themselves.

7.3.12 SmartGrid Refactoring

Before we modularised the analysis SmartGrid, we had to exchange the SmartGrid DSML with its modularised counterpart, the modular SmartGrid DSML [HSR19; SHR18]. In order to exchange the SmartGrid DSML, we had to change each dependency of the SmartGrid

[3] https://github.com/KAMP-Research

Scenario No.	Name	Commit Hash	No. Affected Files
Scenario 01	Implement KAMP4aPS Evaluation Scenario	3126580b	5
Scenario 02	Update KAMP4aPS Scenario	2d37dc02	5
Scenario 03	Add Class for Micro Switch Change	c17f986e	7
Scenario 04	Add Metaclass for Change	1f78d0c0	14
Scenario 05	Update Last Scenario	3f5acd29	2
Scenario 06	Apply Renamed Classes	8491dd9b	4
Scenario 07	HMI Implemented	d54511fe	5
Scenario 08	Change-impact Analysis Modified	5dae880b	6
Scenario 09	Adapt Annotation Look-Up	a5dcc00c	5
Scenario 10	Adapt Model Look-Up	29919f0	3

Table 7.3.: Overview of the Historical Evolution Scenarios of the KAMP4aPS Case Study for the Evolvability and Reusability Evaluation of the Reference Architecture for Model-based Analyses

analysis from the SmartGrid DSML on the modular SmartGrid DSML. Changing the dependencies is necessary, as the modular SmartGrid DSML is not used in the analysis SmartGrid. The dependencies of the SmartGrid regarding the modular SmartGrid metamodel are like the structure shown in Figure 7.1. The dependencies on the modular SmartGrid metamodel are consolidated in the *smartgrid.attackersimulation* and the *smartgrid.impactanalysis* module. Although technically, these two modules represent two different analyses; we consider them one. Each represents an analysis feature of the SmartGrid analysis. The

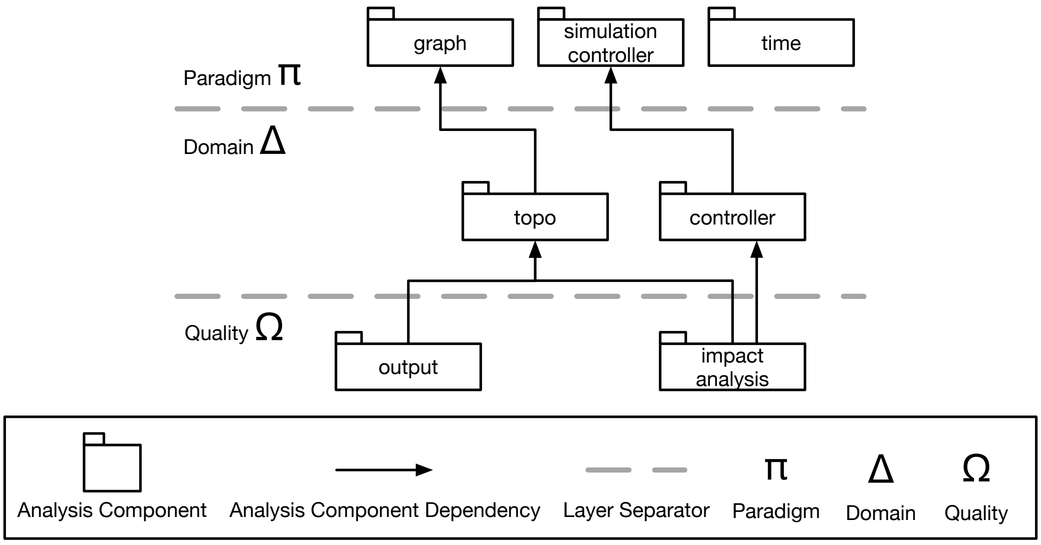


Figure 7.6.: Refactored SmartGrid, simplified [KHR22a]

details regarding the refactoring, especially the classes before and after the refactoring, can be found in our supplementary material [KHR22b].

7.3.13 Modular SmartGrid – mSmartGrid

Figure 7.6 depicts the structure of the analysis SmartGrid after the modularisation of the scenarios. We did not refactor the whole analysis. Therefore, we present only the components of Camunda that are affected by our refactoring and relevant for our calculation of the metrics complexity, coupling, and cohesion.

7.3.13.1 Paradigm Layer

Graph: The *graph* component represents a network graph structure. Either logical or physical edges can connect the nodes in this graph.

Simulation Controller: The *simulation controller* component provides the building blocks for running the simulation. This component is only part of the model-based analysis and has no representation in the DSML.

Time: The *time* component provides a notion of time for the analysis. In the attacker propagation analysis, the time component is not used. This component is only part of the model-based analysis and has no representation in the DSML.

7.3.13.2 Domain Layer

Topo: The devices in a smart-grid topology are contained in the *topo* component. This topology uses the graph structure provided in the π domain.

Controller: The *controller* component extends the simulation controller component of the π layer. It can handle the simulation and analysis of the smart-grid topology. This component is only part of the model-based analysis and has no representation in the DSML.

7.3.13.3 Quality Layer

Impact Analysis: The *impact analysis* component utilises change propagation rules, similar to the rules in KAMP4aPS, to determine the vulnerability of a smart-grid topology. It defines different types of attackers that can traverse through the system.

Output: The *output* component provides the information the analysis produces. Regarding the definition of our reference architecture for model-based analysis, the output should not be located at the Ω layer. However, we apply the reference architecture regarding the structure of the DSML and ignore possible bugs and errors. Therefore, we did not fix the layering issue.

7.3.14 SmartGrid Historical Evolution Scenarios

The third requirement for the case studies is that the developers of the model-based analysis use some sort of source code versioning. In the case of SmartGrid, the source code is versioned with git. The source code is available on GitHub[4]. In Section 7.3.14, we provide an overview of the historical evolution scenarios extracted from the commit history of SmartGrid. Tor transparency reasons, we also provide the commit hash and the number of affected files if someone wants to recreate the change scenarios themselves.

7.4 Evaluation Results

In this section, we present the evolvability and understandability evaluation results. In the presented results, we compare our scenarios' cohesion, coupling, and complexity before and after the refactoring.

[4] https://github.com/kit-sdq/Smart-Grid-ICT-Resilience-Framework

Scenario No.	Name	Commit Hash	No. Affected Files
Scenario 01	Pass Data to Power Load	dfe19981	2
Scenario 02	Report Generation	c8280939	2
Scenario 03	Support String IDs	72ecaa73	2
Scenario 04	Add Parametrised Initialisation	2d7a9c46	8
Scenario 05	Search for Viral Hacker	1648636e	4
Scenario 06	Finalizing RCP Commands	aae4a894	10
Scenario 07	Remove Launch Configuration	63ae1f49	4
Scenario 08	Randomly Hacking of Nodes	3d81da9e	1
Scenario 09	Attacker Simulation Disabling Root	5ee72f70	2
Scenario 10	Attacker Simulation Usable Attributes	4c257bea	2

Table 7.4.: Overview of the Historical Evolution Scenarios of the SmartGrid Case Study for the Evolvability and Reusability Evaluation of the Reference Architecture for Model-based Analyses

SIMULIZAR	**Complexity**		**Coupling**		**Cohesion**	
	before	*after*	*before*	*after*	*before*	*after*
Scenario 01	580.92	**360.81**	**7.99**	7.99	**0.0309**	**0.0114**
Scenario 02	1210.85	**992.80**	123.08	**26.37**	0.0011	0.0011
Scenario 03	0	0	0	0	0.0148	0.0148
Scenario 04	202.96	**106.86**	11.99	**7.99**	**0.0721**	0.0637
Scenario 05	578.92	**234.91**	7.99	**0**	**0.0309**	0.0288
Scenario 06	415.04	**127.03**	0	0	0.0744	**0.0803**
Scenario 07	674.65	**666.09**	9	**7.07**	0.0064	0.0064
Scenario 08	1042.06	**876.01**	120.85	**41.90**	**0.0019**	0.0016
Scenario 09	373.43	**334.59**	15.14	**4**	**0.0199**	0.01629
Scenario 10	242.44	**122.75**	0	0	0.0080	**0.0090**

Table 7.5.: Evolvability Metric Results for the Case Study SimuLizar

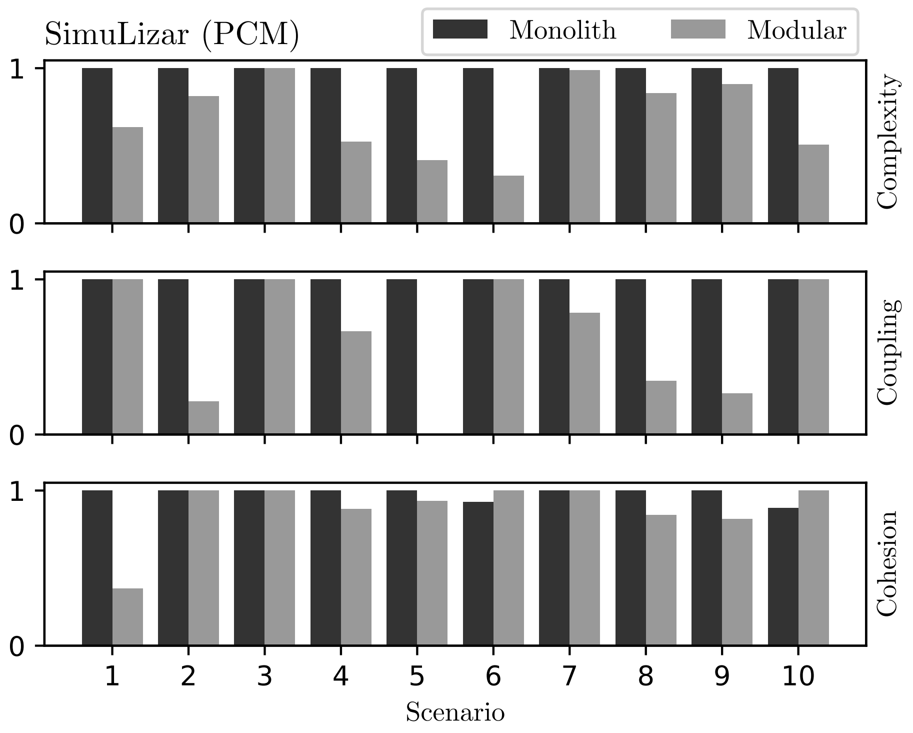


Figure 7.7.: Normalised Evolvability Metric Results for the Case Study SimuLizar

We present the raw results and the visualised results for each case study. To visualise our evaluation results, we created four diagrams. Each diagram represents the results of one of the four case studies. Figure 7.7 shows the results for the case study SimuLizar, Section 7.4 shows the results for the case study Camunda, Section 7.4 shows the results for the case study KAMP4aPS, and Section 7.4 shows the results for the case study SmartGrid. Each diagram is separated into three rows to represent the metrics we used in the evaluation. The rows are labelled on the right side. The results for the complexity metric are shown in

CAMUNDA	Complexity		Coupling		Cohesion	
	before	*after*	*before*	*after*	*before*	*after*
Scenario 01	131.54	**53.67**	0	0	1.55e-2	**5.29e-2**
Scenario 02	119.42	**10.56**	0	0	2.93e-2	**3.35e-2**
Scenario 03	85.04	**74.43**	**20**	31.90	**4.18-5**	1.46e-5
Scenario 04	208.67	**186.08**	19.02	**8**	8.74e-5	**9.72e-5**
Scenario 05	152.58	**145.24**	0	0	0.0054	**0.0073**
Scenario 06	119.43	**39.50**	**0**	8	**5.64e-5**	1.21e-5
Scenario 07	115.68	**33.77**	**8**	**18.25**	5.61e-5	**5.94e-5**
Scenario 08	117.41	**115.92**	0	0	6.25e-4	**6.91e-4**
Scenario 09	160.40	**37.22**	0	0	9.94e-4	**1.4e-2**
Scenario 10	206.14	206.14	47.12	47.12	8.80e-5	**8.93e-5**

Table 7.6.: Evolvability Metric Results for the Case Study Camunda

Figure 7.8.: Normalised Evolvability Metric Results for the Case Study Camunda

the first row. In the second row, the results of the coupling metric are shown. The results for the cohesion metric are shown in the third row.

We normalised the results to the range of zero to one. The results of the scenarios vary; for example, a low value like 0.0000121 and a high value like 206.14. The value range of the raw results is unbound. Due to the high variance of the results, we could not print the results in a meaningful way, and some results would not be visible. The normalisation is unproblematic, as we only compare the results of one scenario to each other. Ergo, the results per metric and scenario are shown on an ordinal scale. We set the higher value to 1.0 and calculated the smaller value concerning the higher value. This was necessary, as the difference was too big for some scenarios, so the difference could not be shown.

The value scale is shown on the y-axis, and the scenario number is shown on the x-axis. The results of the case study before the refactoring are shown in black, and the results of the refactored case study after the refactoring are shown in grey.

The results do not show a bar when the affected classes do not depend on one another. We can determine the ratio of our current cohesion concerning the cohesion of the maximal cohesive graph based on the results of the cohesion calculation. We were required to normalise the results because, without doing so, some graphs would have been illegible. A high level of cohesion is a positive indicator since it indicates that the classes have fewer outbound dependencies, which means that changes made to one class are less likely to affect the other classes.

7.5 Threats to Validity

In this section, we discuss the four types of validity by Runeson et al. [Run+12]. The four validity types are explained in Section 2.6.1.

KAMP4APS	**Complexity**		**Coupling**		**Cohesion**	
	before	*after*	*before*	*after*	*before*	*after*
Scenario 01	146.30	**46.28**	27.08	**0**	**9.85e-3**	7.75e-3
Scenario 02	387.13	**121.55**	**34.79**	86.73	**1.09e-2**	3.13-3
Scenario 03	303.31	**119.64**	**8**	**0**	**1.23e-2**	1.06e-2
Scenario 04	251.17	**95.46**	**64.10**	65.29	2.65e-3	**5.39e-3**
Scenario 05	218.45	**121.02**	8	**0**	4.46e-2	**5.46e-2**
Scenario 06	43.04	**43.04**	**0**	41.24	**2.16e-2**	7.69e-4
Scenario 07	563.83	**177.92**	108.20	**0**	9.31e-3	**1.43e-2**
Scenario 08	538.55	**265.57**	57.88	**16**	7.42e-3	**7.98e-3**
Scenario 09	686.31	**257.64**	63.48	**0**	**7.25e-3**	**6.33e-3**
Scenario 10	444.12	**117.31**	24.77	**0**	**1.17e-2**	**4.28e-3**

Table 7.7.: Evolvability Metric Results for the Case Study KAMP4aPS

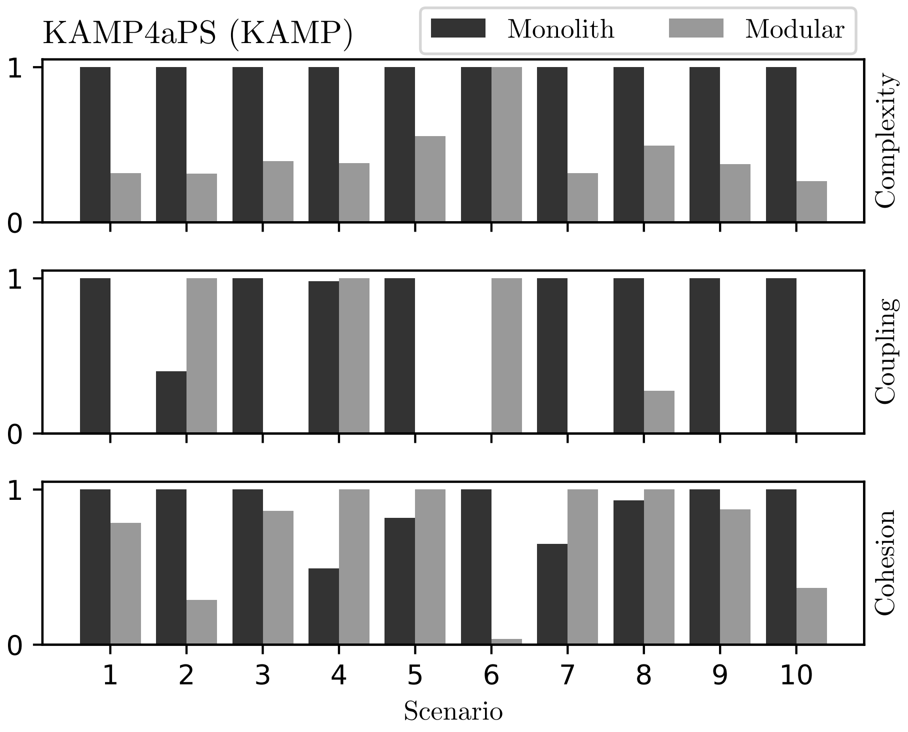


Figure 7.9.: Normalised Evolvability Metric Results for the Case Study KAMP4aPS

7.5.1 Internal Validity

In our evaluation, we analysed four case studies regarding their evolvability and reusability. We extracted scenarios from historical change scenarios, ten per case study. We modularised the affected classes in each scenario according to our reference architecture for model-based analyses. Then we compared the metrics complexity, coupling, and cohesion for each scenario in its state before and after the refactoring. To address the internal validity, we have to ensure that the structure of the reference architecture we have applied

SMART.	Complexity		Coupling		Cohesion	
	before	*after*	*before*	*after*	*before*	*after*
Scenario 01	187.457	**32.729**	0	**0**	**0.027**	0.025
Scenario 02	8.264	**2.75**	**0**	0	**0.133**	**0.172**
Scenario 03	**797.130**	898.998	**0**	589.431	**0.090**	0.015
Scenario 04	1007.02	**156.265**	**8.264**	8.265	**0.018**	**0.007**
Scenario 05	372.145	**103.701**	5.510	5.510	0.007	**0.008**
Scenario 06	601.112	**278.481**	103.364	**70.421**	0.003	0.003
Scenario 07	315.851	**45.995**	7.489	**2.75**	**0.115**	**0.009**
Scenario 08	**153.047**	154.547	**0**	**29.510**	0.112	**0.057**
Scenario 09	256.969	**88.281**	0	**0**	0.041	**0.567**
Scenario 10	16.223	16.223	**0**	2.755	**0.113**	0.062

Table 7.8.: Evolvability Metric Results for the Case Study SmartGrid

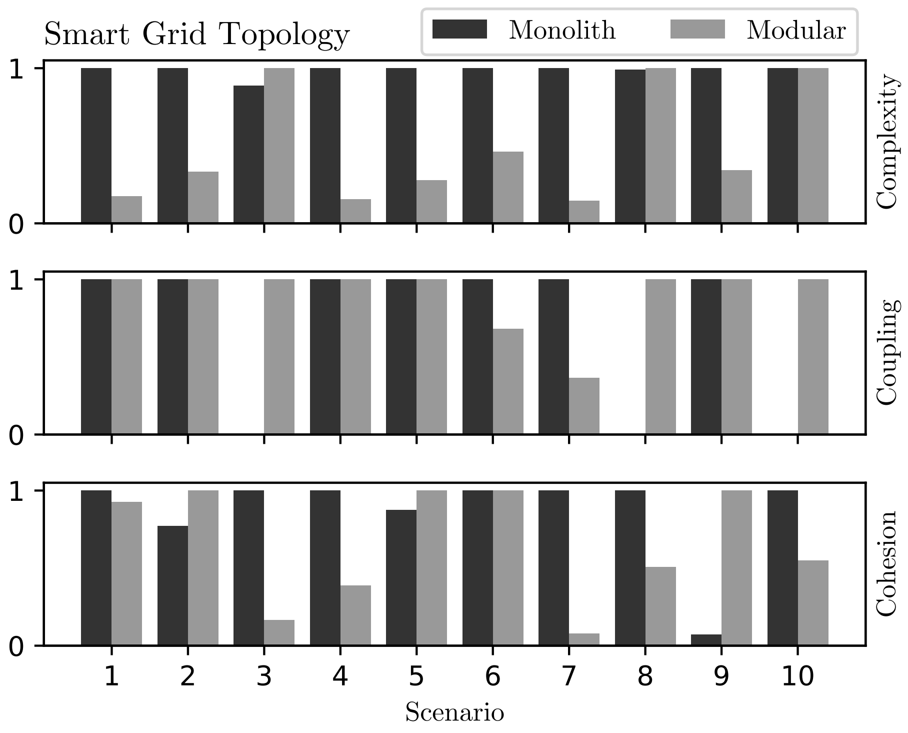


Figure 7.10.: Normalised Evolvability Metric Results for the Case Study SmartGrid

to the scenarios created the metrics changes. One possible cause for the change in the results is when we also fixed bugs and bad smells (cf. Chapter 4) unrelated to the reference architecture. Therefore, we only applied refactorings that helped us to apply the reference architecture to the case studies.

Two different developers refactored the four case studies. The case studies SimuLizar, Camunda, and KAMP4aPS were refactored by the same developer, and the case study SmartGrid was refactored by another developer. Although the structure of the reference architecture is the same for each case study, how a developer selects and sequences the refactorings to reach the desired structure can differ. Thus, the result also might differ, depending on the developer who refactors the case study. Using more than one developer ensures that the developer's experience does not influence the results.

7.5.2 External Validity

Selecting a sample case study that is not as generic as possible to represent a concrete system can allow gaining deeper insight and better realism of the system under study [Run+12]. The selected case studies represent no generic case study covering every model-based analysis possible. Ergo, the results we generated by analysing the four case studies do not represent every other arbitrary model-based analysis. Each case study has its individual properties. The case studies, however, emerge from different domains, and thus, they allow us to gain insights and model-based analyses with similar properties. In addition, our reference architecture might need to be more helpful or practical for generic model-based analysis depending on the circumstances. We decided to go with heterogeneous model-based studies so that we could address this threat. Chapter 6 provides details regarding our selection process and the case studies in general.

7.5.3 Construct Validity

The construct validity would be compromised if we selected case studies that would benefit the most when transferring them to our reference architecture, for example, a highly monolithic model-based analysis. We selected the same case studies as Heinrich et al. [HSR19] to avoid this case. They selected case studies with different degrees of modularity. Although they focussed on the DSML of the case studies, we reviewed the corresponding model-based analyses regarding their modularity. The model-based analyses also have different levels of modularity. This is the case because their structure is similar to their corresponding DSMLs. As a result, we could reuse these case studies, as their model-based analyses also show different levels of modularity. Due to our findings, we came to the knowledge that the more modular an analysis was and the model-based analysis was aligned with its corresponding DSML; the less improvement regarding the complexity metric we could see. For example, the case study Camunda was already well modularised. This case study shows where the analysis was already well modularised and, in some cases, well adapted to the structure of the DSML. Although Camunda was well modularised, adapting it to our reference architecture has shown that the results we got present improvements regarding complexity.

To address this threat further, we selected case studies rooted in different domains (i. e., information systems, business processes, production automation, and smart grids). Due to the different domains, we can ensure that the benefits of our reference architecture are not restricted to a single domain. We can conclude that model-based analyses of the domains we selected to benefit from our reference architecture for model-based analyses.

We focused only on the scenarios and the affected source code files instead of the whole model-based analyses. Furthermore, we do not test the functionality after the refactoring, which can be another threat to construct validity. However, the refactorings we perform only change the structure of the model-based analysis but not its behaviour. Our reference architecture does not affect the analysis algorithms; therefore, we consider this a minor threat.

Our selection process for the evolution scenario could also threaten the construct validity. How we selected the case studies is explained in Chapter 6, and how we selected the evolution scenarios is explained in Section 7.3. To address this threat, we defined three evolution scenarios: historical, potential, and randomised synthetic. The historical evolution scenarios present a minor threat because these evolution scenarios are derived from actual change scenarios of the model-based analyses. These changes in the historical evolution scenarios were applied in the past. The potential evolution scenarios threaten the construct validity as they represent changes that have the potential that the analysis developers will apply them. A guarantee, however, does not exist. The randomised synthetic evolution scenarios present a significant threat to the construct validity as they are changes that were selected randomly. Such changes have the potential that they will always be kept the same. Due to the very well-documented and accessible commit history of all our case studies, we could derive only historical evolution scenarios. Thus, we could avoid our fall-back scenarios of potential and randomised synthetic scenarios.

7.5.4 Conclusion Validity

To omit that only one researcher must interpret the results of our evaluation, we use objective metrics based on information theory. These metrics provide reasonable proof and limit the need for interpretation, eliminating the possibility of a researcher providing their subjective interpretation of the data. The purpose of the evaluation is to discredit the effects that could be attributed to the interpretation of a single researcher. If the evaluation is repeated for more case studies in the future, this will contribute statistically to the confirmation of the results.

7.6 Discussion

In this section, we discuss the results of the evolvability/understandability evaluation of our reference architecture for model-based analyses. We investigate the factors that led to the differences in findings between the modular and the original version of the evolution scenarios, as well as the implications of those distinctions.

7.6.1 Complexity

The lower the value for the complexity metric, the better the result. Regarding the 40 evolution scenarios, 34 of them showed a decrease in complexity after we refactored them. In four scenarios, complexity has neither improved nor worsened. The complexity of the remaining two scenarios increased after the refactorings. The complexity of nine scenarios dropped for each of the case studies SimuLizar, Camunda, and KAMP4aPS. One scenario for each of these three case studies stayed the same. In the SmartGrid case study, the level of complexity for seven different scenarios was reduced, while the level of complexity for two other scenarios grew, and one scenario maintained its previous level of complexity.

Scenario	1	2	3	4	5
SimuLizar	+38%	+18%	+-0%	+47%	+59%
Camunda	+59%	+91%	+13%	+11%	+5%
KAMP4aPS	+68%	+69%	+61%	+64%	+45%
SmartGrid	+83%	+67%	-11%	+84%	+72%

Scenario	6	7	8	9	10
SimuLizar	+69%	+1%	+16%	+10%	+49%
Camunda	+67%	+71%	+1%	+77%	+-0%
KAMP4aPS	+-0%	+68%	+51%	+62%	+74%
SmartGrid	+54%	+85%	-1%	+66%	+-0%

Table 7.9.: Overview – Changes of the Complexity Metric After the Refactoring

In the 34 scenarios where the complexity was improved, we can relate the improvement to the restrictive nature of our reference architecture for model-based analysis. Especially the analysis architect and the analysis component developer are limited regarding the design of the dependencies. The limitation results from the separation into layers and the strict layering. In addition, does our reference architecture permit dependency cycles and the model-based analysis must be aligned with the DSML. Also, is the design space restricted regarding slicing the remaining analysis features and analysis components. Each model-based analysis specific feature has a defined place on one of the layers. The only freedom the analysis architect has is how to separate features of the model-based analysis. This structure resulted in analysis components that are small and specialised. Due to the specialisation, the analysis components have ideally no overlap with other analysis features. The four evolution scenarios that show neither improved nor worsened complexity were already aligned with the DSML; thus, they did not need much change. The two outlier scenarios, in which the complexity was increased, were the result of many changes with different intentions. As a consequence, this led to a large commit that affected a large number of files; consequently, the number of files affected in the modularised version was significantly higher. The overall complexity of the case studies remained the same. However, we discovered that the complexity of the model-based analysis is lowered in the sections of the analysis that are relevant for the analysis developer who is working on an evolution scenario.

7.6.2 Coupling

The lower the value for the coupling metric, the better the result. The results for the coupling metric of all four evolution scenarios are mixed. In 16 out of 40 evolution scenarios, we were able to reduce the coupling. For 14 evolution scenarios, the coupling did not change, despite our refactorings. The coupling of ten evolution scenarios increased due to our refactorings.

The only case study where the coupling did not increase is SimuLizar. In six of ten evolution scenarios, we could decrease the coupling. The remaining four scenarios stayed the same.

For the case study Camunda, in one out of ten evolution scenarios, the coupling did decrease. In six of ten evolution scenarios, the coupling stayed the same. In the remaining three scenarios, the coupling increased.

The case study with the most improvements in the coupling metric is the model-based analysis KAMP4aPS. In seven of ten evolution scenarios, the coupling decreased. The remaining three evolution scenarios decreased regarding the coupling metric.

For the case study SmartGrid, in two out of ten evolution scenarios did, the coupling decrease. In four of ten evolution scenarios, the coupling stayed the same. In the remaining four scenarios, the coupling increased. The SmartGrid case study yields the worst results regarding the coupling metric.

7.6.3 Cohesion

The higher the value for the cohesion metric, the better the result. The cohesion metric produces mixed results for the four scenarios as well; however, the results are more drastic. In 17 of the 40 scenarios, cohesion was increased; in four cases, it stayed unchanged, and in 19 cases, it was decreased. For the case study SimuLizar, cohesion increased in two scenarios, deteriorated in five, and remained constant in three. For the case study Camunda, cohesion improved in eight scenarios while remaining unchanged in two. For the case study KAMP4aPS, cohesion improved in four scenarios while it deteriorated in six others. For the case study SmartGrid, cohesion increased in three scenarios, deteriorated in six, and remained constant in one. The key element that resulted in the worst results is the separation of components and classes, as they convert sections of component cohesion into coupling. Methods in large classes relied on private methods in the same class; splitting such a class resulted in extra dependencies and, as a result, increased coupling. The coupling was not impacted; however, the cohesion was reduced by lowering class internal calls. We skipped refactoring methods to add more private methods, address object-oriented design smells, and improve cohesiveness.

8. Evaluation of Bad Smells in Model-based Analyses

In this section, we present the evaluation of the bad smells we defined in Chapter 4. In Section 8.1, we explain the goals and metrics of our evaluation. In Section 8.2, we present the evaluation design; in Section 8.3, we present the results of the evaluation. In Section 8.4, we discuss the threats to validity. Finally, in Section 8.5, we summarise our findings and discuss the evaluation.

8.1 Research Goals and Metrics

We split the goals and metrics section for the bad smells evaluation into three parts. The first part introduces our research goals regarding the effect of bad smells on the evolvability of model-based analyses and the relevance of bad smells in general. The second part introduces our metrics to evaluate whether we reached our goals. Finally, the third part explains the structure of our scenarios and their use in the evaluation.

We derive the first research goal (*G1*) from our hypothesis (cf. Hypothesis 2) and the first research question (cf. Research Question 4.1), which bad smells arise from the co-dependency of DSMLs and model-based analyses:

Research Goal 8.1

We want to determine whether the bad smells we defined exist in model-based analyses.

The second research goal (*G2*) we derived from our hypothesis (cf. Hypothesis 2) and the second (cf. Research Question 4.2) and third (cf. Research Question 4.3) research questions. The second research question asks how to refactor bad smells for model-based analyses, and the third research question asks how bad smells affect the evolvability and reusability of model-based analyses and their corresponding DSMLs.

Research Goal 8.2

We want to determine whether the bad smells we identified negatively affect the evolvability and reusability of model-based analyses.

In order to determine whether we reached the goals *G1* and *G2* we utilise the four case studies we introduced in Chapter 6. For each case study, we determine the number of bad smells in their source code, and we refactor each bad smell to determine the effect on the evolvability of the affected files. Each bad smell is searched by utilising the identification description we provide in Chapter 4, and to fix the bad smells, we utilise the refactoring description, also provided in Chapter 4. Due to the size and complexity of the case studies, we focus on five occurrences per bad smell per case study.

Breivold et al. [BCE08] presented the software evolvability model. This model comprises the sub-characteristics of analysability, integrity, changeability, extensibility, portability, and testability. Regarding the ISO/IEC 25010 software quality model [ISO10], the characteristic of maintainability and portability map to the sub-characteristics of the software evolvability model. The sub-characteristics analysability, changeability, stability, and testability are part of the maintainability characteristic of ISO/IEC 25010, and the sub-characteristics of adaptability, installability, co-existence, and replaceability are part of the portability characteristic of ISO/IEC 25010. According to Briand et al. [BWL01], and Cruz-Lemus et al. [Cru+10], cognitive complexity affects the analysability and modifiability of software. To measure the cognitive complexity of a system, we refer to the amount of structural information within a system. We choose the same metrics as Heinrich et al. [HSR19] to measure the cognitive complexity of a system. They use the hypergraph metrics of Allen et al. [AGG07], which uses information size, complexity, and coupling to measure the information entropy of a software system. The formal definitions by Briand et al. [BMB96] are the foundation for the metrics by Allen et al. [AGG07].

The hypergraph metrics to evaluate our case studies are presented in Section 2.2.2.

8.2 Evaluation Design

In this section, we present our evaluation design for bad smells in model-based analyses. In Section 8.2.1, we introduce the evaluation scenarios and their selection process. How we conducted the evaluation is presented in Section 8.2.2.

8.2.1 Evolution Scenarios

To answer research questions 3.1 to 3.3 and to check whether we reached our goals *G1* and *G2*, we have to refactor the bad smells of the case studies and determine whether our refactorings can improve the evolvability and reusability of the case studies. These varied case studies allow us to empirically evaluate whether our approach improves model-based analysis evolvability and reusability. Wohlin [Woh21] states that an empirical case study must be a contemporary, real-world phenomenon. All case studies are actively developed, making them relevant. Instead of comparing the source code of a whole case study before and after the refactoring, we derive change scenarios of the case studies to determine the effect of our refactorings regarding evolvability and reusability. We decided to use change scenarios because when implementing a change, it is unnecessary for the analysis developer to modify the whole class; instead, they must change parts of a class of the model-based analysis. It is also unnecessary for the analysis developer to understand classes of the model-based analysis unaffected by the change. Thus, change scenarios better represent how changes occur in a software system. If we did not find occurrences of a bad smell, we would discuss the implications of their absence and possible adverse effects on evolvability and reusability.

In the context of this evaluation, we distinguish two types of change scenarios. The first type is derived from actual changes made in the case study; thus, we call them *historical evolution scenarios*. During the development of a model-based analysis, the analysis developer makes changes that affect code containing bad smells. To derive a historical change scenario, we searched the commit history of a case study for commits that contain a bad smell. A historical evolution scenario contains a set of classes that are part of the commit. It is irrelevant at which point of the commit history we extract the set as long as the classes before and after the refactoring originate from the same commit. If the commit history does not contain commits that affect files with bad smells, we derive *potential evolution scenarios*. A potential evolution scenario represents an artificial change scenario that could occur in the future. For example, extending an interface or changing an analysis technique can lead to changing a set of classes.

For our four case studies, we were able to derive 13 evolution scenarios for the bad smell *Duplicate Abstraction*, 20 evolution scenarios for the bad smell *Missing Abstraction*, ten evolution scenarios for the bad smell *Degraded Modularity*, and 16 evolution scenarios for the bad smell *Rebellious Modularity*. We were able to derive 59 evolution scenarios in total.

8.2.2 Conduction of the Evaluation

In Chapter 4, we presented twelve bad smells that we derived from the co-dependency of bad smells in open-source software and DSMLs. To check whether we achieved the goals *G1* and *G2*, we use the four case studies presented in Chapter 6. For the first research goal *G1*, we analyse how often each bad smell occurs per case study (*Frequency of Occurrence*) and for the second research goal *G2*, we determine how these bad smells affect the evolvability and reusability of the scenarios (*Evolvability and Reusability*).

To identify the frequency of occurrences of our twelve bad smells, evaluating goal *G1*, we determine how often each bad smell occurs in the four case studies. In order to calculate the number of bad smells, we use our tool RefactorLizar, where we implemented the identification algorithms for each bad smell presented in Section 4.2.

For the evaluation of goal *G2*, to assess whether the evolvability and reusability have improved after refactoring a bad smell, we compare the original evolution scenario to the refactored evolution scenario. The refactoring can result in a different number of classes. We only consider the classes that would be affected by the change of the evolution scenario. For example, a change can affect only one line in a class method with over 500 lines of code. After the refactoring, we might have moved the method to another class; thus, the change would no longer affect the original class. As a result, we omit classes that are no longer affected by a change.

For each evolution scenario before and after the refactoring, we calculate the cohesion, coupling, and complexity according to the metrics presented in Section 2.2.2. We also use our tool RefactorLizar to calculate the metrics. We present the results in the following section.

We cannot evaluate the second goal *G2* for every bad smell, either because fixing the bad smell would mainly affect the DSML and not the model-based analysis, or the bad smell was not present in any of our case studies. The bad smells we were unable to evaluate are *Unused Abstraction*, *Deficient Encapsulation*, all *Hierarchy* smells, *Broken Modularity*, *Missing Modularity*, and *Weakened Modularity*.

The *Unused Abstraction* smell contains the language types not used in the model-based analysis. The unused language types must be removed from the DSML to fix this bad smell; however, fixing this bad smell does not affect the metrics complexity, coupling and cohesion of the analysis code because the model-based analysis will not change. However, removing language types of the DSML does improve the evolvability of the DSML, as it reduces the total number of elements in the DSML.

The *Deficient Encapsulation* smell indicates that some language types of the DSML are always used together and, thus, that the language types could be merged. To fix this bad smell, the language architect must merge the language types, which reduces the complexity of the models, as there are fewer types to consider. We did not fix this bad smell, as we need to know whether these types are always used together in every model-based analysis that uses the DSML. However, merging language types reduces the coupling of the model-based analysis and the DSML, as the number of dependencies is reduced. Ergo, we can deduce that merging language types affected by the *Deficient Encapsulation* improves the metrics coupling and complexity of a model-based analysis.

The *Hierarchy* smells, and the *Broken Modularity* smell depend on a hierarchical structure of the DSML and the model-based analysis. We present a reference architecture for model-based analyses in Chapter 3; this reference architecture introduces such a hierarchical structure. We evaluate the benefits of the reference architecture for model-based analyses in Chapter 7; thus, we omit the evaluation in this section.

Fixing the *Broken Modularity* smell would require modifying the DSML when the DSML has dependencies on the model-based analysis. In the context of our four case studies, we did not find occurrences of the *Broken Modularity* smell; thus, we could not evaluate the impact on the metrics' complexity, coupling, and cohesion. Nevertheless, we can deduce that when the DSML depends on a model-based analysis, the complexity of the DSML increases due to unnecessary dependencies.

Cyclic dependencies make code challenging to comprehend and difficult to maintain over time. This, in turn, opens the door to software systems that are prone to errors and are challenging to test. If an architecture contains circular dependencies, any modification to a single module would probably produce a ripple effect of errors for the rest of the components in the cycle. We focus on cycles that form between the DSML and its corresponding model-based analysis. Dependency cycles negatively affect the maintainability of software systems, and they can increase the technical debt in such a system [Lil19]. As we could not find occurrences of the *Broken Modularity* smell, the *Weakened Modularity* smell can also not occur.

8.2.3 Refactoring Scenarios

In this section, we present the refactoring scenarios we selected to evaluate the effect of refactoring the bad smells on the evolvability and reusability of model-based analyses.

8.2.3.1 Refactoring Scenarios – Duplicated Abstraction

The *Duplicated Abstraction* smell occurs when two analysis classes depend on the same set of language classes. We ignored when two analysis classes depend on exactly one or two language classes. To refactor the *Duplicated Abstraction* smell, we introduce an indirection layer that encapsulates the access of the language components.

SimuLizar: For refactoring the *Duplicated Abstraction* smell, we selected three occurrences of the bad smell in the SimuLizar case study. In the first scenario *DA_S_S1*, the class *EventNotificationHelper* and the class *LogDebugListener* depend on four types of the DSML PCM. These language types are *OperationSignature*, *ExternalCallAction, UsageScenario*, and *EntryLevelSystemCall*. We identified one commit with the id *3aad7b* that affects both the *EventNotificationHelper* and the *LogDebugListener* class.

In the second scenario *DA_S_S2*, the class *ComposedStructureInnerSwitch* and the class *RepositoryComponentSwitch* depend on four types of the DSML. These types are *AssemblyContext*, *ComposedStructure*, *Signature*, and *Connector*. We found one commit with the id *fdc988* that affects both the *ComposedStructureInnerSwitch* and the *RepositoryComponentSwitch* class.

In the third and last scenario *DA_S_S3*, the class *AbstractInterpreterListener* and the class *AbstractProbeFrameworkListener* also depend on four types of the DSML. These types of the DSML are *OperationSignature*, *ExternalCallAction*, *UsageScenario*, and *EntryLevelSystemCall*. We identified one commit with the id *3aad7b* that affects both classes.

KAMP4aPS: For refactoring the *Duplicated Abstraction* smell, we selected five occurrences of the bad smell in the KAMP4aPS case study. In the first scenario *DA_K_S1*, the class *SignalInterfacePropagation* and the class *InterfaceChanges* depend on five types of the DSML. These types are *Interface*, *ModifyInterface*, *ModifyComponent*, *ChangePropagationDueToHardwareChange*, and *Component*. We were unable to find a commit that affects both classes; therefore, we created a potential evolution scenario, which contains the classes *InterfaceChanges*, *ScrewingChanges*, and *SignalInterfacePropagation*. The screwing in the context of the KAMP4aPS metamodel is an interface that connects two hardware components of an aPS. These changes affect the generic class *InterfaceChange* and can also affect the screwing, as screwing is a specialisation of the Interface class. Such a change can also affect the propagation of the element type, in this case, the propagation between interfaces.

In the second scenario *DA_K_S2*, the class *RampChange* and the class *SignalInterfacePropagation* depend on four types of the DSML. These types are *ModifyInterface*, *ModifyComponent*, *ChangePropagationDueToHardwareChange*, and *Component*. We were unable to find a commit that affects both classes; therefore, we created a potential evolution scenario, which contains the classes *ComponentChanges*, *RampChange*, and *SignalInterfacePropagation*. The ramp in an aPS is a hardware component used as a slider to move elements in the system. Changes that affect the generic component can also affect changes in any specialised component, such as a ramp. The components in an aPS are connected via hardware interfaces; changes to the components can also affect the change propagation between the components via the interfaces; hence, the *SignalInterfacePropagation* can also be affected by a change.

In the third scenario *DA_K_S3*, the class *SignalInterfacePropagation* and the class *SwitchChanges* depend on four types of the DSML. These types are *Interface*, *ModifyInterface*, *ChangePropagationDueToHardwareChange*, and *Component*. We were unable to find a commit that affects both classes; therefore, we created a potential evolution scenario, which contains the classes *Change*, *SignalInterfacePropagation*, and *SwitchChange*. The generic class *Change* can affect a specific change like the *SwitchChange* and how changes propagate through the system. Hence the *SignalInterfacePropagation* can also be affected.

In the fourth scenario *DA_K_S4*, the class *ScrewingChanges* and the class *SignalInterfacePropagation* depend on four types of the DSML. These types are *Interface*, *ModifyInterface*, *ModifyComponent*, and *ChangePropagationDueToHardwareChange*. We were unable to find a commit that affects both classes; therefore, we created a potential evolution scenario, which contains the classes *InterfaceChanges*, *ScrewingChanges*, *SensorChanges*, and *SignalInterfacePropagation*. The screwing of a component in an aPS is an interface that connects components in the system, and a sensor can be such a component that needs to be screwed on another component in the aPS.

In the fifth scenario *DA_K_S5*, the class *SwitchChanges* and the class *SignalInterfacePropagation* depend on four types of the DSML. These types are *ModifyInterface*, *ChangePropagationDueToHardwareChange*, *Component*, and *Interface*. We were unable to find a commit that affects both classes; therefore, we created a potential evolution scenario, which contains the classes *Change*, *InterfaceChanges*, *ScrewingChanges*, and *SignalInterfacePropagation*. If the generic class Change is modified, this can affect the specific classes *InterfaceChanges* and *ScrewingChanges*, and it also can affect the propagation rules; therefore, it can affect the class *SignalInterfacePropagation*.

SmartGrid: For refactoring the *Duplicated Abstraction* smell, we selected five occurrences of the bad smell in the SmartGrid case study. In the first scenario *DA_SG_S1*, the class *DFSStrategy* and the class *GraphAnalyzer* depend on four types of the DSML. These four types are *On*, *EntityState*, *NamedEntity*, and *Cluster*. We were unable to find commits that affect both the class *DFSStrategy* and the class *GraphAnalyzer*; therefore, we created a potential evolution scenario, which contains the classes *DFSStrategy* and *GraphAnalyzer*. Both classes could change together, as the strategy for a depth-first search (DFS) determines how to search a graph.

In the second scenario *DA_SG_S2*, the class *LoadOutputModelConformityHelper* and the class *GraphAnalyzer* depend on six types of the DSML. These six types are *SmartGrid-Topology*, *EntityState*, *NetworkEntity*, *Identifier*, and *ScenarioResult*. We were unable to find commits that affect both the *LoadOutputModelConformityHelper* class and the *Graph-Analyzer* class; therefore, we created a potential evolution scenario which contains the classes *GraphAnalyzer* and *LoadOutputModelConformityHelper*. These classes could change together when the loaded model is changed, affecting how the graphs are analysed.

In the third scenario *DA_SG_S3*, the class *ViralHacker* and the class *GraphAnalyzer* depend on five types of the DSML. These five types are *SmartGridTopology*, *On*, *EntityState*, *Identifier*, *Cluster*, and *ScenarioResult*. We were unable to find commits that affect both the *ViralHacker* and the *GraphAnalyzer* class; therefore, we created a potential evolution scenario which contains the classes *GraphAnalyzer*, *ScenarioModelHelper*, and *ViralHacker*. These three classes may change together; the models the scenario model helper handles can affect how a viral hacker model is handled.

In the fourth scenario *DA_SG_S4*, the class *GraphAnalyzer* and the class *AttackStrategies* depend on seven types of the DSML. These seven types are *On*, *EntityState*, *Communicatin-gEntity*, *LogicalCommunication*, *NetworkEntity*, *Cluster*, and *PhysicalConnection*. We were unable to find commits that affect both the *GraphAnalyzer* class and the *AttackStrategies* class; therefore, we created a potential evolution scenario which contains the classes *AttackStrategies*, *BFSStrategy*, *DFSStrategy*, and *GraphAnalyzer*. These three classes may change together when the strategies change.

In the fifth scenario *DA_SG_S5*, the class *LoadOutputModelConformityHelper* and the class *ScenarioModelHelper* depend on four types of the DSML. These four types are *SmartGrid-Topology*, *EntityState*, *NamedEntity*, and *ScenarioResult*. We were unable to find commits that affect both the *LoadOutputModelConformityHelper* class and the *ScenarioModelHelper* class; therefore, we created a potential evolution scenario which contains the classes *LoadOutputModelConformityHelper*, *ScenarioModelHelper*, and *ViralHacker*. The helper classes may change together, affecting the viral hacker model.

8.2.3.2 Refactoring Scenarios – Missing Abstraction

The *Missing Abstraction* smell occurs when the model-based analysis uses primitive types instead of dedicated types. To refactor the *Missing Abstraction* smell, we introduce dedicated types and replace the occurrences of the primitive types with the new dedicated types.

SimuLizar: For the refactoring of the *Missing Abstraction* smell, we selected five occurrences in the SimuLizar case study. In the first scenario *MA_S_S1*, the class *QVToReconfigu-rationTest* uses primitive types for measurements and paths. Instead of strings for the path, we use the Path class by the java.io package, and for the measurements, we introduce a dedicated type. The commit with the id *be9224* affects the *QVToReconfigurationTest* class; thus, we selected this commit for the first scenario.

For the second scenario *MA_S_S2*, we found primitive types in the class *ResourceUtil* that are used to transport file and path information. We introduce a new class for file handling, and for path handling, we utilise the java.io package. The commit with the id *8277cd* affects the *ResourceUtil* class; thus, we selected this commit for the second scenario.

In the third *MA_S_S3*, and fourth *MA_S_S4* scenario, the *SimulatedBasicComponentInstance* uses multiple primitive types to represent the notion of time. We replaced these primitive types with a time class representing the notion of time. Affected by this change is the *SimulatedBasicComponentInstance*; therefore, we searched for a commit that at least affects the *SimulatedBasicComponentInstance* class. The commit with the id *534d55* is suitable for this scenario; therefore, we selected it for the third scenario. The commit with the id *a55386* fits as a scenario; therefore, we selected it for the fourth scenario.

Camunda: For the refactoring of the *Missing Abstraction* smell, we selected five occurrences in the Camunda case study. In the first scenario *MA_C_S1*, the identifier of a job in the analysis is submitted as a string instead of a dedicated type. Thus, we introduced a new identifier class and replaced the primitive ids with the dedicated type. One class affected by that change is the *ManagementServiceImpl* class. We chose the commit with the id *5e6d48* for the historical evolution scenario because it affects the *ManagementServiceImpl* class.

For the second scenario *MA_C_S2*, in the *ActivityImpl* class, the exclusiveness of an asynchronous activity is set with a primitive boolean type. However, in addition to one boolean parameter that sets the exclusiveness of the activity, further parameters of the same primitive type are always used together. Instead of introducing a new type, we restructured the *ActivityImpl* class to reduce the parameters to one boolean per method. We chose the commit with the id *01b4b3* for the historical evolution scenario because it affects the *ActivityImpl* class.

In the third scenario *MA_C_S3*, the identifier of a worker in the analysis is submitted as a string instead of a dedicated type. Thus, we introduced a new identifier class and replaced the primitive identifiers with the dedicated type. One class affected by that change is the *ExternalTaskService* class. We chose the commit with the id *3b3e99* for the historical evolution scenario because it affects the *ManagementServiceImpl* class.

In the fourth scenario *MA_C_S4*, the results of an analysis task are submitted as an integer instead of a dedicated type. We introduce a new dedicated class that can store integer-based results to replace the primitive result types. Two classes affected by that change are the *OptimizeHistoricVariableUpdateQueryCmd* and the *OptimizeService* class. We chose the commit with the id *8ebe48* for the historical evolution scenario because it affects the two aforementioned classes.

For the fifth scenario *MA_C_S5*, we changed the handling of databases and database statements. The database type and the database statements are submitted as primitive strings. We introduced new classes that represent the database and the database statements. Affected by this change is the class *DbSqlSessionFactory*. We chose the commit with the id *18b7ed* for the historical evolution scenario because it affects the *DbSqlSessionFactory* class.

KAMP4aPS: For the refactoring of the *Missing Abstraction* smell, we selected five occurrences in the KAMP4aPS case study. In the first scenario *MA_K_S1*, we found primitive types in the class *APSArchitectureVersionPersistency* that are used to transport file and path information. For the file and the path handling, we utilise the java.io package instead of the primitive type string. We chose the commit with the id *5dae88* for the historical evolution scenario because it affects the aforementioned class.

For the second scenario *MA_K_S2*, we found primitive types in the class that are used to represent the version of an architecture version. We introduce a new dedicated type that represents the version of an architecture, for example, the version number and version name. As mentioned earlier, we chose the commit with the id *5dae88* for the historical evolution scenario because it affects the class.

In the third *MA_K_S3*, fourth *MAK_S4*, and fifth scenario *MAK_S5*, the identifier of a label in the analysis is submitted as a string instead of a dedicated type. Thus, we introduced a new identifier class and replaced the primitive ids with the dedicated type. One class that is affected by that change is the *LabelCustomizing* class. The commit with the id *47d3cc* fits as a scenario; thus, we selected it for the third historical evolution scenario. We chose the commit with the id *8e7cb9* for the fourth historical evolution scenario. We chose the commit with the id *69ab43* for the fifth historical evolution scenario because they affect the *LabelCustomizing* class.

SmartGrid: For the refactoring of the *Missing Abstraction* smell, we selected five occurrences in the SmartGrid case study. In the first *MA_SG_S1*, second MA_SG_S2 and fourth scenario MA_SG_S4, in the class *LocalHacker* the primitive type string is used to represent the hacking style and hacking speed of a hacker in the system. We introduce a dedicated type that contains the hacking style and speed to replace the occurrences of the primitive types. For the first historical evolution scenario MA_SG_S1, we chose the commit with the id *4b7c2b* because it affects the *LocalHacker* class. We chose the commit with the id *2d7a9c* for the second historical evolution scenario MA_SG_S2 because it affects the class as mentioned above. For the fourth historical evolution scenario MA_SG_S4, we chose the commit with the id *1f8a57* f because it affects the *LocalHacker* class.

In the third MA_SG_S3 and fifth scenario MA_SG_S5, we found primitive types in the class *RmiServer* that are used to transport file and path information. For the file and path handling, we utilise the java.io package. We chose the commit with the id *0fd5fe* for the third historical evolution scenario because it affects the aforementioned *RmiServer* class. We chose the commit with the id *0a4229* for the fifth historical evolution scenario because it affects the aforementioned *RmiServer* class.

8.2.3.3 Refactoring Scenarios – Degraded Modularity

The *Degraded Modularity* smell occurs when components of the model-based analysis depend on the same language component of the DSML. To refactor the *Degraded Modularity* smell, we merge classes of the analysis components and move them to a common analysis component. When the classes depend on different language components, we split the classes before we are able to merge them.

SimuLizar: For the refactoring of the *Degraded Modularity* smell, we selected five occurrences in the SimuLizar case study. In the first scenario *DM_S_S1*, the analysis classes *ControllerMappingImpl*, *EventNotificationHelper*, and *ControllerMapping* depend on the language type *OperationProvidedRole*. To fix this bad smell, we moved the dependencies on the language type into a dedicated class. We could not find a commit that affects all three classes; therefore, we created the first potential evolution scenario containing the classes mentioned above.

For the second scenario *DM_S_S2*, we selected the Degraded Modularity smell that affects the analysis classes *QVToReconfigurationTest*, *ComposedStructureInnerSwitch*, and *RepositoryComponentSwitch* that depend on the language type Connector. To fix this bad smell, we moved the dependencies of the classes *ComposedStructureInnerSwitch* and *RepositoryComponentSwitch* that depend on the language type Connector into a dedicated class. The test class is not part of the model-based analysis; thus, we did not refactor it. We found a commit that affects the class *RepositoryComponentSwitch*; as a result, we created the second evolution scenario as a historical evolution scenario based on the commit with the id *e0facd*.

Based on the Degraded Modularity smell of the second scenario, we derived further historical evolution scenarios for the evaluation. For the third scenario *DM_S_S3*, we were able to select a commit that affects the classes *ComposedStructureInnerSwitch*, *RepositoryComponentSwitch* and 20 more. As a result, we created the third evolution scenario as a historical evolution scenario based on the commit with the id *fdc988*. For the fourth scenario *DM_S_S4*, we were able to select a commit that affects the classes *ComposedStructureInnerSwitch*, *RepositoryComponentSwitch* and twelve more. As a result, we created the third evolution scenario as a historical evolution scenario based on the commit with the id *7e1b98*. For the fifth scenario *DM_S_S5*, we were able to select a commit that affects the classes *ComposedStructureInnerSwitch*, *RepositoryComponentSwitch* and six more. As a result, we created the third evolution scenario as a historical evolution scenario based on the commit with the id *657cbf*.

SmartGrid: For the refactoring of the *Degraded Modularity* smell, we selected five occurrences of it in the SmartGrid case study. In the first scenario *DM_SG_S1*, the analysis classes *ScenarioModelHelper*, *GraphAnalyzer*, and *LoadInputModelConformityHelper* depend on the language type *ScenarioState*. To fix this bad smell, we moved the dependencies on the language type *ScenarioState* into a dedicated class. We were unable to find a commit that affects all three classes; therefore, we created the first potential evolution scenario that contains the aforementioned classes.

For the second scenario *DM_SG_S2*, we selected the Degraded Modularity smell that affects the analysis classes *ScenarioModelHelper*, *GraphAnalyzer*, *LoadInputModelConformityHelper*, *ViralHacker*, and *LoadOutputModelConformityHelper* that depend on the language type *SmartGridTopology*. To fix this bad smell, we moved the dependencies that depend on the language type into a dedicated class. We were unable to find a commit that affects the aforementioned classes; as a result, we created the second evolution scenario as a potential evolution scenario.

In the third *DM_SG_S3*, fourth *DM_SG_S4*, and fifth *DM_SG_S5* scenario, we selected commits of the case study SmartGrid that affect a subset of the classes that are part of the *Degraded Modularity* smell. For the third scenario, we selected a commit that affects the classes *GraphAnalyzer*, *ViralHacker* and five other classes. As a result, we created the third evolution scenario as a historical evolution scenario based on the commit with the id *430554*. For the fourth scenario, we were able to select a commit that affects the classes *GraphAnalyzer*, *LoadOutputModelConformityHelper*, *ViralHacker* and three other classes. As a result, we created the fourth evolution scenario as a historical evolution scenario based on the commit with the id *959f06*. For the fifth scenario, we selected a commit that affects the classes *GraphAnalyzer* and *LoadInputModelConformityHelper*. As a result, we created the fifth evolution scenario as a historical evolution scenario based on the commit with the id *64d4fc*.

8.2.3.4 Refactoring Scenarios – Rebellious Modularity

The *Rebellious Modularity* smell occurs as an analysis component of the model-based analysis depending on multiple language components of the DSML. To refactor the *Rebellious Modularity* smell, we split the analysis classes to separate the dependencies on the language types into dedicated analysis classes or analysis components. We created historical evaluation scenarios based on the same commits, as different files are affected by the bad smells in the scenarios. For example, the scenarios *RM_K_S4* and *RM_K_S5* have the same commit, but the commit contains multiple *Rebellious Modularity* smells. We only refactored one bad smell and left the remaining unchanged not to spoil our evaluation results.

SimuLizar: For the refactoring of the *Rebellious Modularity* smell, we selected five occurrences in the SimuLizar case study. In the first scenario *RM_S_S1*, the class *LogDebugListener* depends on the language components mpcm.domain.repository, mpcm.domain.seff, and mpcm.domain.usage. After the refactoring, we identified the commit with the *ba19f7* that includes the *LogDebugListener* and twelve other classes. Based on the commit, we created the first historical evolution scenario.

For the second scenario *RM_S_S2*, we found that the analysis class *MonitorRepositoryUtil* depends on the language components mpcm.domain.seff, mpcm.domain.usage, mpcm.paradigm.seff, and mpcm.paradigm.repository. After the refactoring, we selected the commit with the id *055e9a* that includes the *MonitorRepositoryUtil*. Regarding the commit, we created the second historical evolution scenario to evaluate the *Rebellious Modularity* smell in the SimuLizar case study.

In the third scenario *RM_S_S3*, the class *PeriodicallyTriggeredUsageEvolver* depends on the language components mpcm.domain.usage and mpcm.paradigm.variables. After the refactoring, we selected the commit with the id *2d215c* that includes the *PeriodicallyTriggeredUsageEvolver* and 24 other classes. Based on the commit, we created the third historical evolution scenario.

For the fourth scenario RM_S_S4, we found that the analysis class *AbstractInterpreterListener* depends on the language components mpcm.domain.repository, mpcm.domain.seff, mpcm.domain.usage, and mpcm.paradigm.repository. After the refactoring, we selected the commit with the id *f066ea* that includes the *AbstractInterpreterListener* and five other classes. Based on the commit, we created the fourth historical evolution scenario.

In the fifth scenario *RM_S_S5*, the class *RepositoryComponentSwitch* depends on the language components mpcm.domain.repository, mpcm.domain.composition, mpcm.paradigm.composition, mpcm.paradigm.base, mpcm.paradigm.repository, and mpcm.quality.performance. After the refactoring, we selected the commit with the id *45e128* that includes the *RepositoryComponentSwitch* and five other classes. Based on the commit, we created the fifth historical evolution scenario.

Camunda: For the refactoring of the *Rebellious Modularity* smell, we selected one occurrence in the Camunda case study. In the only scenario *RM_C_S1*, the class *BpmnModelExecutionContext* depends on the language components flows-paradigm and core-classes. After the refactoring, we identified the commit with the id *b85d83* that includes the *BpmnModelExecutionContext* and 13 other classes. Based on the commit, we created the historical evolution scenario for refactoring Camunda in the context of the *Rebellious Modularity* smell.

KAMP4aPS: For the refactoring of the *Rebellious Modularity* smell, we selected five occurrences in the KAMP4aPS case study. In the first scenario *RM_K_S1*, the class *SensorChanges* depends on the language components mkamp.as.mm, mkamp.aps.mm, edu.kit.ipd.sdq.kamp.model.modificationmarks, and mkamp.aps. After the refactoring, we identified the commit with the *1f78d0* that includes the *SensorChanges* class and nine other classes. Based on the commit, we created the first historical evolution scenario.

For the second scenario *RM_K_S2*, we found that the analysis class *SignalInterfacePropagation* depends on the language components mkamp.as.mm and mkamp.as. After the refactoring, we selected the commit with the id *1f78d0* that includes the *SignalInterfacePropagation* and nine other classes. Regarding the commit, we created the second historical evolution scenario to evaluate the *Rebellious Modularity* smell in the KAMP4aPS case study.

In the third scenario *RM_K_S3*, the class *SwitchChanges* depends on the language components mkamp.as.mm, mkamp.aps.ppu, mkamp.as, edu.kit.ipd.sdq.kamp.model.modificationmarks, and mkamp.aps. After the refactoring, we selected the commit with the id

47d3cc that includes the *SwitchChanges* and 24 other classes. Based on the commit, we created the third historical evolution scenario.

For the fourth scenario *RM_K_S4*, we found that the analysis class *RampChange* depends on the language components edu.kit.ipd.sdq.kamp4aps.basic, mkamp.as.mm, mkamp.as, and mkamp.aps. After the refactoring, we selected the commit with the id *2d37dc* that includes the *RampChange* and three other classes. Based on the commit, we created the fourth historical evolution scenario.

In the fifth scenario *RM_K_S5*, the class *ModuleChanges* depends on the language components mkamp.as.mm, edu.kit.ipd.sdq.kamp4aps.basic, mkamp.as, and edu.kit.ipd.sdq.kamp.model.modificationmarks. After the refactoring, we selected the commit with the id *2d37dc* that includes the *ModuleChanges* and three other classes. Based on the commit, we created the fifth historical evolution scenario.

SmartGrid: For the refactoring of the *Rebellious Modularity* smell, we selected five occurrences in the SmartGrid case study. In the first scenario of the SmartGrid case study, *RM_SG_S1*, the class *GraphAnalyzer* depends on the language components msmartgrid.paradigm.graph, msmartgrid.domain.topo, msmartgrid.paradigm.base, msmartgrid.analysis.output, and msmartgrid.analysis.input. After the refactoring, we identified the commit with the *e4ce4f* that includes the *GraphAnalyzer* class and two other classes. Based on the commit, we created the first historical evolution scenario.

For the second scenario *RM_SG_S2*, we found that the analysis class *AttackStrategies* depends on the language components msmartgrid.paradigm.graph and msmartgrid.analysis.output. After the refactoring, we selected the commit with the id *a2dc4f* that includes the *AttackStrategies* class and ten other classes. Regarding the commit, we created the second historical evolution scenario to evaluate the Rebellious Modularity smell in the SmartGrid case study.

In the third scenario *RM_SG_S3*, the class *LocalHacker* depends on the language components msmartgrid.domain.topo, and msmartgrid.analysis.output. After the refactoring, we selected the commit with the id *d06686* that includes the LocalHacker and 26 other classes. Based on the commit, we created the third historical evolution scenario.

In the fourth scenario *RM_SG_S4*, the class GraphAnalyzer depends on the language components msmartgrid.paradigm.graph, msmartgrid.domain.topo, msmartgrid.paradigm.base, msmartgrid.analysis.output, and msmartgrid.analysis.input. After the refactoring, we identified the commit with the *d06686* that includes the *GraphAnalyzer* class and 26 other classes. Based on the commit, we created the fourth historical evolution scenario.

In the fifth scenario *RM_SG_S5*, the class *ViralHacker* depends on the language components msmartgrid.domain.topo, msmartgrid.paradigm.base, and msmartgrid.analysis.output. After the refactoring, we selected the commit with the id *a2dc4f* that includes the *ViralHacker* and three other classes. Based on the commit, we created the fifth historical evolution scenario.

	SimuLizar	Camunda	KAMP4aPS	SmartGrid
Duplicated Abstraction	15	0	76	18
Missing Abstraction	41	1153	5	12
Unused Abstraction	891	270	169	75
Deficient Encapsulation	6	1	9	4
Folded Hierarchy	0	0	0	0
Missing Hierarchy	0	0	0	0
Unexploited Hierarchy	0	0	0	0
Broken Modularity	0	0	0	0
Degraded Modularity	9	0	0	15
Missing Modularity	441	2201	28	60
Rebellious Modularity	13	4	21	11
Weakened Modularity	0	0	0	0

Table 8.1.: Number of Occurrences of the Bad Smells in the Four Case Studies.

8.3 Evaluation Results

In this section, we present the results for the *Frequency of Occurrences* of the bad smells in the four case studies to determine whether we reached our goal **G1**, and we present the results for the *Evolvability and Reusability*. We also present the results for the *Frequency of Occurrences* to determine whether we reached our goal **G2**; we show the number of bad smells for each case study. In the results for the *Evolvability and Reusability*, we compare our scenarios' metrics cohesion, coupling, and complexity before and after the refactoring.

8.3.1 Frequency of Occurrence Results

The *Duplicated Abstraction* smell predominately occurs in the KAMP4aPS case study and only in the Ecore-based model-based analyses. In KAMP4aPS, the smell does occur 76 times, in SmartGrid 18 times, and in SimuLizar 15 times. In the case study Camunda, we could not find occurrences of the *Duplicated Abstraction* smell. The *Missing Abstraction* smells occurs predominately in the Camunda case study, 19 times more than in the remaining three case studies combined. In Camunda, the smell occurs 1153 times, in SimuLizar 41 times, in SmartGrid 12 times and in KAMP4aPS five times. The *Unused Abstraction* smell occurs predominately in the SimuLizar case study, 1,7 times more than in the remaining three case studies combined. In the case study SimuLizar, the smell occurs 891 times, in Camunda 290 times, in KAMP4aPS 169, and in Smart Grid 75 times. The number *Deficient Encapsulation* smells in the case studies is low. In the case study SimuLizar, the smell occurs six times, in Camunda one time, in KAMP4aPS nine times and in SmartGrid four times. The *Folded Hierarchy* smell occurs predominately in the SmartGrid and the SimuLizar case studies.

The smell occurs 80 times in SmartGrid and 72 times in SimuLizar. In the KAMP4aPS case study, the smell occurs 26 times, and the Camunda case study has the lowest number with four occurrences. We could not find occurrences of the *Missing Hierarchy* smell in any of the four case studies. Why we could not find the smell and why the smell is still relevant will be discussed in Section 8.5. The *Unexploited Hierarchy* smell only occurs in the SmartGrid case study with 19 findings. The *Degraded Modularity* smell occurs in the SimuLizar and the SmartGrid case study with 9 and 15 occurrences, respectively. The bad smell with the highest number of findings in total and the highest number of findings in one case study is the *MissingModularity* smell. In the case study SimuLizar the smell occurs 441 times, in Camunda 2201 times, in KAMP4aPS 28 times and in SmartGrid 60 times. The *Rebellious Modularity* smell predominately occurs in the KAMP4aPS case study. In KAMP4aPS, the smell occurs 21 times, in SimuLizar 13 times, in SmartGrid eleven times and in Camunda four times. The last bad smell *Weakened Modularity* occurs in the case studies SimuLizar and Camunda with 20 and seven occurrences, respectively.

8.3.2 Evolvability, Understandability, and Reusability Results

Before we can start presenting the refactoring results, we discuss why we could not refactor all bad smells and why they are still relevant when we discuss recurring patterns that negatively affect the evolvability and reusability of model-based analyses. The bad smells we were unable to refactor manually or automatically are *Duplicated Abstraction*, *Unused Abstraction*, *Folded Hierarchy*, *Missing Hierarchy*, *Unexploited Hierarchy*, *Broken Modularity*, and *Missing Modularity*.

The bad smells related to the hierarchical structure of the DSML and its corresponding model-based analysis are addressed when the reference architecture for model-based analysis we present Chapter 3 is applied to the model-based analysis. How the hierarchical structure and the compliance to the reference architecture affects the evolvability and reusability is presented in Chapter 7.

We could not evaluate the *Broken Modularity* smell, as the case studies consist only of one model-based analysis that uses one of the DSMLs. We could not investigate the effect of refactoring multiple model-based analysis that use the same DSML. Nevertheless, the *Broken Modularity* smell negatively affects the evolvability and reusability of model-based analysis. If the DSML depends on one model-based analysis, changes to the model-based analysis can trigger changes in the DSML, which can affect the other depending model-based analysis. For example, suppose the DSML references the analysis engine of a specific model-based analysis to use its notion of time (i. e., milliseconds). In that case, all other model-based analyses must use the same notion of time. If the notion of time changes, the DSML and all correlated model-based analyses also have to change.

	Complexity		Coupling		Cohesion	
	before	*after*	*before*	*after*	*before*	*after*
DA_S_S1	1804	**1785**	**253**	254	**1.68E-4**	1.54E-4
DA_S_S2	1475	**1430**	355	355	**9.70E-4**	9.25E-4
DA_S_S3	1983	1983	265	265	1.59E-4	1.59E-4

Table 8.2.: SimuLizar– Duplicated Abstraction Refactoring

	Complexity		Coupling		Cohesion	
	before	*after*	*before*	*after*	*before*	*after*
DA_K_S1	298	**216**	183	**115**	0.016651	**0.016654**
DA_K_S2	129	**61**	69	**19**	0.021	**0.030**
DA_K_S3	30	**27**	13	**10**	0.038	0.038
DA_K_S4	295	**206**	183	**113**	**0.01311**	0.01305
DA_K_S5	320	**207**	197	**109**	0.01402	**0.01406**

Table 8.3.: KAMP4aPS – Duplicated Abstraction Refactoring

8.3.2.1 Duplicated Abstraction

In this section, we present the results for the *Duplicated Abstraction* refactorings. The tables show for each scenario the *complexity*, *coupling*, and *cohesion* before and after the refactoring.

The results for the case study SimuLizar are shown in Table 8.2. We refactored three change scenarios from the case study SimuLizar. The complexity and the cohesion improved in the first scenario *DA_S_S1*, but the coupling deteriorated. The complexity and cohesiveness improved in the second scenario *DA_S_S2*, but the coupling remained unchanged. The refactorings of the third scenario *DA_S_S3* had no influence on the metrics' complexity, cohesion, or coupling.

The case study KAMP4aPS results are shown in Table 8.3. We refactored five change scenarios from the case study KAMP4aPS. After the refactoring, the complexity in the first *DA_K_S1* and second scenario *DA_K_S2* improved, as did the coupling and cohesion. Only the complexity and coupling improved in the third scenario *DA_K_S3*, while the cohesion did not change. The complexity and coupling improved in the fourth scenario *DA_K_S4*, but the cohesion deteriorated. The three metrics complexity, coupling, and cohesion improved for the final scenario *DA_K_S5*.

Table 8.4 displays the results for the case study SmartGrid. We also refactored five scenarios from the SmartGrid case study. The complexity in the first scenario *DA_SG_S1* improved, as did the coupling, but the cohesion deteriorated. The complexity and cohesion improved in the second scenario *DA_SG_S2*, but the coupling deteriorated. Only the complexity improved in the third scenario *DA_SG_S3*, while the coupling and cohesion deteriorated. The complexity improved in the fourth scenario *DA_SG_S4*, but the coupling and cohesion worsened. The three metrics complexity, coupling, and cohesion improved for the final scenario *DA_SG_S5*.

	Complexity		Coupling		Cohesion	
	before	*after*	*before*	*after*	*before*	*after*
DA_SG_S1	991	**909**	110	**98**	**0.060**	0.056
DA_SG_S2	1001	**920**	**110**	111	0.053	**0.056**
DA_SG_S3	1200	**1130**	**160**	161	**0.021**	0.020
DA_SG_S4	997	**971**	**114**	145	**0.037**	0.032
DA_SG_S5	251	**221**	55	**45**	0.0126	**0.0135**

Table 8.4.: SmartGrid – Duplicated Abstraction Refactoring

	Complexity		Coupling		Cohesion	
	before	*after*	*before*	*after*	*before*	*after*
MA_S_S1	**1023**	1077	**109**	155	**0.00264**	0.00251
MA_S_S2	348	**341**	**76**	81	**8.88E-3**	6.97E-3
MA_S_S3	**2421**	2429	510	510	2.5E-4	**2.6E-4**
MA_S_S4	**70**	83	11	11	0.105	**0.107**

Table 8.5.: SimuLizar – Missing Abstraction Refactoring

8.3.2.2 Missing Abstraction

In this section, we present the results for the *Missing Abstraction* refactorings. The tables show for each scenario the *complexity*, *coupling*, and *cohesion* before and after the refactoring.

The results for the case study SimuLizar are shown in Table 8.5. We refactored four cases from the case study SimuLizar. The metrics complexity, coupling, and cohesion deteriorated in the first scenario *MA_S_S1*. The complexity improved in the second scenario *MA_S_S2*, but the coupling and cohesion worsened. Only the cohesion improved in the third scenario, *MA_S_S3*, while the coupling remained unchanged and the complexity deteriorated. The complexity of the fourth scenario *MA_S_S4* decreased; the coupling did not change, but the cohesion improved.

The results for the case study Camunda are shown Table 8.6. We refactored five cases from the case study Camunda. The metrics complexity and cohesion improved in the first *MA_C_S1* and second *MA_C_2* scenarios, while the coupling worsened. The metrics did

	Complexity		Coupling		Cohesion	
	before	*after*	*before*	*after*	*before*	*after*
MA_C_1	286	**267**	**82**	87	**2.7E-4**	2.6E-4
MA_C_2	166	**112**	**40.5**	40.9	**2.3E-4**	1.9E-4
MA_C_3	30	30	12	12	1.34E-3	1.34E-3
MA_C_4	**101**	103	**11**	14	**0.023**	0.021
MA_C_5	**647**	648	**173**	174	**2.71E-3**	2.66E-3

Table 8.6.: Camunda – Missing Abstraction Refactoring

	Complexity		Coupling		Cohesion	
	before	*after*	*before*	*after*	*before*	*after*
MA_K_S1	**991**	1023	**445**	477	**4.2E-3**	3.9E-3
MA_K_S2	991	**769**	445	**366**	4.2E-3	**4.9E-3**
MA_K_S3	**5499**	5553	2259	**2222**	**1.81E-4**	1.80E-4
MA_K_S4	68	**11**	**8**	21	**1.09E-5**	1.08E-5
MA_K_S5	**179**	181	65	**51**	**1.3E-3**	1.2E-3

Table 8.7.: KAMP4aPS – Missing Abstraction Refactoring

	Complexity		Coupling		Cohesion	
	before	*after*	*before*	*after*	*before*	*after*
MA_SG_S1	352	**351**	41	**37**	**0.030**	0.026
MA_SG_S2	1321	**1316**	161	**157**	**0.020**	0.018
MA_SG_S3	**312**	314	**76**	77	**0.03**	0.02
MA_SG_S4	198.7	**198.2**	22	**17**	**0.06**	0.05
MA_SG_S5	**312**	314	**76**	77	**0.03**	0.02

Table 8.8.: SmartGrid – Missing Abstraction Refactoring

not change in the third scenario *MA_C_3*. The complexity, coupling, and cohesion all deteriorated for the fourth *MA_C_4* and fifth *MA_C_5* scenarios.

The case study KAMP4aPS results are shown in Table 8.7. We refactored five scenarios from the case study KAMP4aPS. The metrics complexity, coupling, and cohesion deteriorated in the first scenario, *MA_K_S1*. The metrics complexity, coupling, and cohesion improved for the second scenario, *MA_K_S2*. The metric coupling improved while the metrics complexity and cohesion deteriorated in the third scenario, *MA_K_S3*. The metrics complexity and cohesion improved, while the metric coupling deteriorated in the fourth scenario, *MA_K_-S4*. The metric coupling improved, but the metrics complexity and cohesion deteriorated in the fifth scenario, *MA_K_S5*.

The results for the case study SmartGrid are shown in Table 8.8. We refactored five scenarios from the SmartGrid case study. The metrics complexity and coupling improved in the first *MA_SG_S1* and second *MA_SG_S2* scenarios, whereas metric cohesion deteriorated. The metrics complexity, coupling, and cohesion deteriorated in the third scenario, *MA_SG_S3*. The metrics complexity and coupling improved, but the metric cohesion deteriorated in the fourth scenario, *MA_SG_S4*. The metrics complexity, coupling, and cohesion deteriorated in the fifth scenario, *MA_SG_S5*.

8.3.2.3 Degraded Modularity

The results for the case study SimuLizar are shown in Table 8.9. We refactored five cases from the SimuLizar case study. The metrics complexity and coupling improved while metric cohesion deteriorated in the first scenario, *DM_S_S1*. The metrics complexity, coupling, and cohesion improved in the second scenario, *DM_S_S2*. For the third scenario, *DM_S_S3*,

	Complexity		**Coupling**		**Cohesion**	
	before	*after*	*before*	*after*	*before*	*after*
DM_S_S1	403	**379**	63	**52**	**0.019**	0.018
DM_S_S2	105	**97**	11	**10**	0.12	**0.16**
DM_S_S3	1475	**1431**	355	**347**	**9.7E-4**	9.6E-4
DM_S_S4	1254	**1235**	**252**	253	2.78E-3	**2.81E-3**
DM_S_S5	892	**872**	55	**54**	5.8E-3	**5.9E-3**

Table 8.9.: SimuLizar – Degraded Modularity Refactoring

	Complexity		**Coupling**		**Cohesion**	
	before	*after*	*before*	*after*	*before*	*after*
DM_SG_S1	1050.39	**1050.30**	143	143	0.022	**0.023**
DM_SG_S2	**528**	554	**75**	86	0.01	0.01
DM_SG_S3	42	**35**	14	**13**	0.027	**0.033**
DM_SG_S4	374	**373**	**61**	65	4.8E-3	**5.0E-3**
DM_SG_S5	4600	4600	1131	**1129**	1.0E-4	1.0E-4

Table 8.10.: SmartGrid – Degraded Modularity Refactoring

the metrics complexity and coupling improved while the metric cohesion deteriorated. The metrics complexity and cohesion improved, and the metric coupling decreased in the fourth scenario, *DM_S_S4*. The complexity, coupling, and cohesion metrics improved in the fifth scenario, *DM_S_S5*.

The results for the case study SmartGrid are displayed in Table 8.10. Five scenarios were refactored from the SmartGrid case study. The metric complexity improved, the metric coupling did not change, and the metric cohesion improved in the first scenario, *DM_SG_S1*. The metrics complexity and cohesion declined in the second scenario, *DM_SG_S2*, while coupling remained unchanged. The metrics complexity, coupling, and cohesion improved for the third scenario,*DM_SG_S3*. The metrics complexity and cohesion improved while the metric coupling deteriorated in the fourth scenario, *DM_SG_S4*. The metrics complexity and cohesion did not change in the fifth scenario, *DM_SG_S5*, and coupling improved.

	Complexity		**Coupling**		**Cohesion**	
	before	*after*	*before*	*after*	*before*	*after*
RM_S_S1	303	**296**	**62**	66	0.0016	**0.0021**
RM_S_S2	93	93	60	60	0.01	0.01
RM_S_S3	805	**790**	23	**22**	8.26E-4	**8.34E-4**
RM_S_S4	306	306	47	47	0.006	**0.008**
RM_S_S5	675	**652**	**43**	46	0.009	**0.010**

Table 8.11.: SimuLizar – Rebellious Modularity Refactoring

	Complexity		**Coupling**		**Cohesion**	
	before	*after*	*before*	*after*	*before*	*after*
RM_C_S1	3532	**3526**	368	**357**	4.93E-4	**4.94E-4**

Table 8.12.: Camunda – Rebellious Modularity Refactoring

	Complexity		**Coupling**		**Cohesion**	
	before	*after*	*before*	*after*	*before*	*after*
RM_K_S1	844	**841**	520	**517**	0.0021	**0.0022**
RM_K_S2	844	844	520	520	0.002	0.002
RM_K_S3	1007	**997**	545	**543**	5.0E-3	**5.1E-3**
RM_K_S4	1007	**989**	545	**534**	5.02E-3	**5.03E-3**
RM_K_S5	5499	**5470**	2259	**2231**	2.0E-4	**2.0E-4**

Table 8.13.: KAMP4aPS – Rebellious Modularity Refactoring

8.3.2.4 Rebellious Modularity

The results for the case study SimuLizar are shown in Table 8.11. We refactored five cases from the case study SimuLizar. The metric complexity and cohesion improved in the first scenario *RM_S_S1*, but the metric coupling deteriorated. The metrics complexity and cohesion deteriorated in the second scenario, but coupling stayed unchanged in the second scenario *RM_S_S2*. The metrics complexity, coupling, and cohesion improved for the third scenario,*RM_S_S3*. For the fourth scenario, *RM_S_S4*, the metric cohesion improved while the metrics complexity and coupling remained the same. The metrics complexity and cohesion improved, while the metric coupling deteriorated in the fifth scenario *RM_S_S5*.

The findings from the Camunda case study are presented in Table 8.12. We refactored one scenario from the Camunda case study. The complexity, coupling, and cohesion metrics improved for scenario *RM_C_S1*.

The results for the case study acKAMP4APS are shown in Table 8.11. We refactored five scenarios from the case study KAMP4aPS. Complexity, coupling, and cohesion metrics all improved in the initial scenario *RM_K_S1*. The complexity, coupling, and cohesion metrics were unchanged in scenario *RM_K_S2*. Complexity, coupling, and cohesion metrics all improved for scenario *RM_K_S3*. The metrics complexity, coupling, and cohesion increased for scenario *RM_K_S4*. Improvements were made for the metrics complexity, coupling, and cohesion in scenario *RM_K_S5*.

The results for the case study SmartGrid are shown in Table 8.14. We refactored five scenarios from the SmartGrid case study. The metrics complexity and cohesion improved in the first scenario *RM_SG_S1*, while the metric coupling declined. The metrics complexity and coupling deteriorated in the second scenario *RM_SG_S2*, while metric cohesion remained unchanged. The metrics complexity and coupling improved for the third scenario, *RM_-SG_S3*, while metric cohesion remained unchanged. The metrics complexity and coupling improved, whereas the metric cohesion degraded in the fourth scenario, *RM_SG_S4*. The

	Complexity		Coupling		Cohesion	
	before	*after*	*before*	*after*	*before*	*after*
RM_SG_S1	1181	**1147**	**207**	211	0.0220	**0.0223**
RM_SG_S2	**540**	549	**142**	148	0.003	0.003
RM_SG_S3	6289	**6041**	2239	**2012**	0.002	0.002
RM_SG_S4	6696	**6289**	2559	**2239**	**1.548E-3**	1.546E-3
RM_SG_S5	**540**	573	**142**	163	0.003	0.003

Table 8.14.: SmartGrid – Rebellious Modularity Refactoring

metrics complexity and coupling declined in the fifth scenario *RM_SG_S5*, while the metric coupling remained unchanged.

8.4 Threats to Validity

In this section, we discuss the four types of validity by Runeson et al. [Run+12]. The four validity types are explained in Section 2.6.1.

8.4.1 Internal Validity

In our evaluation, we analysed four case studies regarding 12 bad smells dedicated to model-based analyses. We investigated the relevance of the bad smells in four case studies and how they affect evolvability and reusability. We identified the occurrences of each bad smell in each case study. Based on these occurrences, we derived change scenarios based on historical or potential changes. Then, we compared the metrics complexity, coupling, and cohesion for each scenario before and after the refactoring. To address the internal validity, we have to ensure that the refactoring of the bad smell we have applied to the scenarios created the changes in the metrics. One possible cause for the change in the results is when we also fixed bugs and bad smells (cf. Chapter 4) unrelated to the bad smell under study. Therefore, we only applied refactorings that helped us to fix the bad smell under study to the case studies.

8.4.2 External Validity

Choosing a sample case study that is not as general as possible to depict a concrete system will provide greater insight and a better understanding of the system under investigation [Run+12]. The case studies we chose do not represent a generic case study that covers all possible model-based analyses. As a result, the findings from the four case studies do not represent every other arbitrary model-based research. On the other hand, the case studies derive from distinct domains, allowing us to get insights into model-based analyses with similar qualities. To counter this issue, we chose to conduct heterogeneous model-based analyses. Chapter 6 goes through our selection procedure and the case studies in general.

8.4.3 Construct Validity

The construct validity will be compromised if we choose case studies that would profit the most from fixing the bad smells, such as those with many bad smells. We chose the same case studies as Heinrich et al. [HSR19] to avoid this issue. Heinrich et al. focussed on the DSML and its evolvability and reusability; we, on the other hand, reviewed the corresponding model-based analyses regarding the occurrences of the bad smells. Each case study that we analysed has a different number of bad smells. Based on our findings, we concluded that removing bad smells can improve the complexity of a model-based analysis.

In order to better address this threat, we chose case studies that originated from various diverse domains (i. e., information systems, business processes, production automation, and smart grids). Because of the several domains, we ensure that the benefits of removing bad smells are not limited to a single domain. We can conclude that improving the internal quality by removing bad smells in model-based analyses would be beneficial to the model-based analyses of the domains we choose.

Our decision-making process for the evolution scenario could likewise jeopardise the construct validity. Chapter 6 explains how we chose the case studies, and Section 7.3 explains how we choose the evolution scenarios. To handle this issue, we established two evolution scenarios: historical and potential. The historical evolution scenarios pose the least risk because they are generated from actual change scenarios of the model-based analyses. These historical evolution scenario changes were used in the past. The potential evolution scenarios risk the construct's validity because they indicate changes the analysis developers may apply, but we cannot guarantee these changes will occur. We were able to extract historical and potential evolution scenarios from the very well-documented and available commit history of all of our case studies.

8.4.4 Conclusion Validity

To omit that only one researcher must interpret the results of our evaluation, we use objective metrics based on information theory. These metrics provide reasonable proof and limit the need for interpretation, eliminating the possibility of a researcher providing their subjective interpretation of the data. The purpose of the evaluation is to discredit the effects that could be attributed to the interpretation of a single researcher. If the evaluation is repeated for more case studies in the future, this will contribute statistically to the confirmation of the results.

8.5 Discussion

In this section, we discuss if the findings of the existence and relevance evaluation of our bad smells for model-based analyses indicate that we have accomplished our objectives.

8.5.1 Existence

The results for the existence evaluation show that we found seven out of 12 bad smells in the four case studies. The bad smells we did not find are smells from the hierarchical category and the *Weakened Modularity* smell. We could not find these bad smells because they exist only in model-based analyses that follow our reference architecture for model-based analysis. Because we did refactor the case studies according to our reference architecture for model-based analyses, it is impossible that the model-based analysis contains these bad smells, else they would violate the constraints of our reference architecture. However, during the refactoring to apply our reference architecture, we used our tool RefactorLizar to identify occurrences of the bad smells to identify points in the model-based analysis that needed more refactoring. Therefore, we conclude that although the initial results did not contain occurrences of the hierarchical and the *Weakened Modularity* smell, the definition and identification can guide the analysis developer to implement our reference architecture for model-based analysis.

Regarding the *Broken Modularity* smell, we were unable to find occurrences of it because the smell requires multiple model-based analysis that correspond to one DSML However, we only investigated one model-based analysis per DSML. Thus, the *Broken Modularity* is the only bad smell we cannot determine whether it exists in model-based analyses.

Surprisingly, Camunda, as an industrial product, has more bad smells than, for example, SimuLizar, which was developed exclusively in a university context. An industrial product intending to thrive and be maintainable and evolvable should have enough resources available to improve and polish the product over time. We put the higher number of bad smells down to the difference in the size of the case studies; the more extensive the case study, the more bad smells can occur.

We demonstrated that of the remaining seven bad smells, all can be found in our case studies. The most common bad smells we found are the *Missing Modularity* smell with a total number of 2730 occurrences, the *Unused Abstraction* with 1405 occurrences, and the *Missing Abstraction* smell with 1211 occurrences in the four case studies. The high number of occurrences of the *Missing Modularity* smell is an exception because the case studies we investigated have no layering structure; thus, each component corresponds to one bad smell. In model-based analyses with a layered structure, the *Missing Modularity* smell can help find classes and components that are not part of a layer in the system. The smell *Unused Abstraction* occurs 1405 times, and the *Missing Abstraction* smell occurs 1211 times in all four case studies combined. The reason that the smell of *Unused Abstraction* is a typical bad smell can be explained by the fact that not all types from the language are needed in the analyses. The analysis SimuLizar, for example, is specialised in performance simulation and therefore does not need the types in the PCM that cover e. g. failure probabilities of components. The number of *Missing Abstraction* smells can be explained by the evolutionary development process and the size of the case studies. At the beginning of the development, it is often unclear what kind of specialised types are needed; thus, the analysis developer sticks with basic types. The bigger the analysis, the more effort the developer requires to replace primitive types with dedicated types, which are often discarded because new features are prioritised. This explains why the smell *Missing Abstraction* occurs more frequently in the largest case study Camunda than in the remaining smaller ones. The remaining bad smells *Duplicated Abstraction*, *Deficient Encapsulation*, *Degraded Modularity*, and *Rebellious Modularity* occur under 200 times per bad smell. Nevertheless, except for the *Broken Modularity*, each bad smell occurs in the case studies we investigated. As a result, we conclude that the bad smells we defined exist in model-based analyses. The following section discusses the results regarding the relevance of the bad smells.

8.5.2 Relevance

The results for the relevance evaluation show that the *Duplicated Abstraction* smell negatively affects the complexity of a model-based analysis and its evolvability; a system that is hard to comprehend, due to its complexity, is also hard to evolve and maintain, as the effects of changes are unpredictable. Although we did not find the *Duplicated Abstraction* smell in each case study and the number of findings ranged from 15 to 76 occurrences, we determine the *Duplicated Abstraction* smell as a relevant bad smell that, when fixed, positively impacts the evolvability and reusability of model-based analyses. Therefore, it is worth fixing the *Duplicated Abstraction* smell.

The results for the *Missing Abstraction* smell need to be clarified. After the refactoring, the complexity is reduced in ten scenarios and worsened in eight. Introducing dedicated types can add more complexity to a model-based analysis, especially when it is a one-to-one substitution. If, for example, in a method signature, one primitive type is replaced by one dedicated type, the developer does not gain any advantage. The scenarios where the complexity worsened were such one-to-one substitutions. However, if fixing the *Missing Abstraction* smell means that multiple method parameters can be replaced, the complexity can be reduced. The scenarios where the complexity improved were such substitutions. Therefore, we determine that the analysis developer must decide whether introducing dedicated types to the model-based analysis has any positive effects on the existing code base. Alternatively, the analysis developer can accept a deterioration of the complexity in the short-term, for the possibility to avoid the *Missing Abstraction* smell in the future. Thus, we determine the *Missing Abstraction* smell as relevant in developing model-based analyses.

The results for the *Degraded Modularity* show that it negatively affects the complexity of model-based analyses. Of the ten scenarios we refactored, the complexity improved for eight of the ten scenarios when we fixed the *Degraded Modularity* smell. The *Degraded Modularity* smell only occurred in 50% of our case studies with a total number of 24.

Although the smell occurred not often, in 80% of the scenarios, we improved the complexity of the evolution scenarios. Fixing the *Degraded Modularity* smell requires that the analysis developer moves dependencies from analysis classes on language types. As we provide refactorings (cf. Section 3.3.3) to fix this bad smell, we determine the effort of fixing it as mitigable. Due to the positive effect on the internal quality when fixing the bad smell, and the rare occurrences of the bad smell as well as the mitigable effort that is required to fix the bad smell, we determine the *Degraded Modularity* smell as a relevant bad smell that is worth fixing.

The results for the *Rebellious Modularity* show that it negatively affects the complexity of model-based analyses. Of the eleven scenarios we refactored, the complexity improved for eight scenarios when we fixed the *Degraded Modularity* smell, and the remaining three did not change. The *Rebellious Modularity* smell occurs in every case study we analysed with a total number of 49 occurrences. Fixing the bad smell requires access to the DSML, with the possibility of making changes or splitting the affected analysis class. The best refactoring is different for each case, but for the refactoring of the model-based analysis, we provide refactorings on class and component level (cf. Section 3.3.3). We determine the effort of refactoring the model-based analysis as mitigable, especially if we consider the improved evolvability when fixing the bad smell. Due to the positive effect on the internal quality when fixing the bad smell, and the mitigable effort that is required to fix the bad smell, we determine the *Degraded Modularity* smell as a relevant Therefore, we determine the *Rebellious Modularity* smell as a relevant bad smell worth fixing.

Relevance of the remaining bad smells: In this section, we discuss why the bad smells we did not evaluate with the complexity, coupling, and cohesion metrics are nonetheless relevant for the evolvability and reusability of model-based analyses and their corresponding DSML. The *Unused Abstraction* smell does not directly affect the evolvability and reusability of a model-based analysis; however, if types of a DSML are not used, the language is unnecessarily complex. The unused type can lead the tool user to make false assumptions regarding the effect of types on the analysis result. If, for example, the analysis does not use the defined bandwidth that can be modelled with the DSML, the tool user could connect the result to the irrelevant bandwidth. Furthermore, the model can be unnecessarily complex, as unused types can still be modelled.

The *Deficient Encapsulation* smell also does affect primarily the DSML and not the model-based analysis. Across all model-based analysis that use a DSML, if types are always used together, merging them reduces the number of types and the dependencies between types and components of a language. As a result, the DSML becomes cleaner and easier to understand for the tool user.

The *Folded Hierarchy* smell affects the dependencies of specialised analysis classes on more generic language classes. According to Heinrich et al. [HSR19], these dependencies will change more specialised classes if generics change. It results in unnecessary changes and more effort for the analysis developer when the corresponding DSML changes. In Chapter 3 and Chapter 7, we present a layered reference architecture for model-based analyses. The remaining bad smells related to the hierarchical structure and the modularity of the DSML and the model-based analysis are evaluated in Chapter 7.

9. Evaluation of the Model-based Analysis Specification and Reuse of Model-based Analysis Components

In this section, we present the evaluation of our approach to specify and compare simulation components. The evaluation goals and metrics are presented in Section 9.1. The evaluation results for the applicability of the simulation specification are presented in Section 9.2.1, and the results for the accuracy of the approach to compare simulation components, based on their specification, are presented in Section 9.2.2. The threats to validity are discussed in Section 9.3, and the discussion of the results is presented in Section 9.4.

9.1 Research Goals and Metrics

We split the goals and metrics section to evaluate our specification and compare approaches into three parts. The first part introduces our research goals regarding the applicability and accuracy of our specification metamodel to DESs. The second part introduces our metrics to evaluate whether we reached our goals. Finally, the third part explains the structure of our scenarios and their use in the evaluation.

We derive the first research goal (RG 9.1) from our hypothesis (cf. Hypothesis 3) and the first research question Research Question 5.1, how to specify model-based analysis components to enable analysis component comparison:

Research Goal 9.1

We want to determine how applicable our DSML for the specification of DES is, which covers structural and behavioural information when it is used to specify components of real-world DES.

The second and third research goals (RG 9.2 and RG 9.3) we derived from our hypothesis (cf. Hypothesis 3) and the second research question (cf. Research Question 5.2), how to compare and correctly identify similar model-based analysis components:

Research Goal 9.2

We want to determine how accurately our approach can identify similar simulation components based on their structure specification.

Research Goal 9.3

We want to determine how accurately our approach can identify similar simulation components based on their behaviour specification.

We split the Research Question 5.2 into two goals because our approach is separated into two steps. The first step is to compare the specifications of simulation components regarding their structure. The second step is to compare the specifications of simulation components regarding their behaviour. We use two different DES that serve as case studies to evaluate our approach. The evaluation of our specification DSML and our comparing approach follow the Goal Question Metric (GQM) approach [BCR94].

In order to determine whether we reached the goals RG 9.1, RG 9.2, and RG 9.3, we utilise the case studies *Palladio Simulator* and *Camunda* that we introduced in Chapter 6. We derive ten scenarios for each case study, each containing one simulation component. We use our specification language to model each scenario to determine whether we can model the simulation component with our specification language, i. e., how applicable our specification language is to real-world simulation components. After we modelled the scenarios, we used our approach to compare the specifications of the simulation components to each other. To compare specifications, we use our approach to compare the specifications of simulation components regarding their structure (cf. Section 5.3) and to compare the specifications of simulation components regarding their behaviour (cf. Section 5.4).

9.1.1 Applicability Metric

In this section, we introduce the applicability metric we used to check whether we reached the Research Goal 9.1. We select components of a model-based simulation to identify whether the metamodel to specify components of a model-based simulation is suitable for modelling simulation components. Then we model these components with our specification language. How we selected the components of the model-based simulations is described in Section 9.1.3. For the evaluation, we use two of the case studies presented in Chapter 6, Camunda and the Palladio Simulator. For each simulation component, we identify the entities and events it contains. Regarding the events, we have to identify how an event affects the simulation world, i. e., which attributes are changed and how the change affects the scheduled delay of the event. Furthermore, we also have to identify which attributes are read by an event and which other events are scheduled by each event. As a result, events can schedule events of other simulation components; therefore, we also have to identify these components and their events. After identifying the simulation components' entities and events, we modelled each component. We checked whether we could model each entity and event with our specification DSML. We calculate for each simulation component how many elements are covered by our DSML as follows:

$$M_n = E_{model} + V_{model}$$

$$S_n = E_{simulation} + V_{simulation}$$

$$C_n = \frac{M_n}{S_n}$$

We calculate the coverage C for the simulation component n by dividing the sum of the modelled entities of the component E_{model} and the events of the component V_{model} by the sum of entities in the component $E_{simulation}$ and the events in the component V_{model}. After calculating the coverage of each simulation component, we determine the applicability A of our specification language for simulation components by calculating the average coverage C over the number of simulation components m.

$$A = \frac{\sum_1^m C_m}{m}$$

9.1.2 Accuracy Metric

To determine whether we reached the goals Research Goal 9.2 and Research Goal 9.3, we also use a scenario-based evaluation to determine how accurate our approach can identify similar simulation components based on their structure and behaviour specification. How we selected the components of the model-based simulations for the scenarios is described in Section 9.1.3. First, we compare the structure specification of the simulation components of the scenarios by using the graph-isomorphism approach presented in Section 5.3. Then, if we could identify a structural match, we compare the behaviour using our SMT-based approach presented in Section 5.4. We determine the accuracy of our approach to finding simulation components by comparing their structure and behaviour specification. To determine the accuracy, we calculate the metric F_1 score. The F_1 score is the harmonic mean of precision and recall. Precision and recall aggregate the number of true positives, false positives, and false negatives. The number of true positives (t_p), false positives (f_p), and false negatives (f_n) is calculated by comparing the list of identified components with the list of components that should be identified. Identifying t_p, t_n, and f_n is scenario specific. Therefore, we explain how we identify them when we introduce the scenarios. Given the amount true positives, false positives, and false negatives, precision and recall are calculated as [Pow20]:

$$precision = \frac{t_p}{t_p + f_p}$$

and

$$recall = \frac{t_p}{t_p + f_n}$$

The F_1 score is calculated as the harmonic mean of precision and recall:

$$f_1 = 2\frac{precision \times recall}{precision + recall}$$

9.1.3 Scenarios

We developed five scenarios to answer the Research Question 5.2 and to show whether our approach can find the right simulation component when it is compared to other simulation components or to find a simulation component in a set of simulation components that represent a whole DES. In addition to the specification of the simulation components C_{C1} to C_{C10} and C_{E1} to C_{E10}, we also added obfuscated simulation components O_1 to O_{20}. We obfuscated the names of the entities, attributes, and events in the specification of the simulation components to have a difference in the naming of the elements. For example, the event *identify_task* becomes a name with no relation to other events or the domain of the simulation component. The scenarios can be divided into two categories: The first category (scenarios S_1 and S_2) evaluates whether the suitable simulation component can be identified in a set of simulation components by individually comparing each. The second category (scenarios S_3 to S_5) evaluates whether a simulation component can be identified in a set of simulations representing a DES. We distinguish the following scenarios:

The first scenario, S_1, shows that our approach can identify the suitable simulation component in a set of different components. Therefore, the simulation components of the two case studies are compared to find the correct match. For example, we compare C_{C1} to the other simulation components of Camunda C_{C2} to C_{C10} and itself. This example is correctly identified when C_{C1} is identified as a matching simulation component. It counts as wrongly identified when the result is any of the other simulation components C_{C2} to C_{C10}. If the correct simulation component is identified, we count it as t_p; if a wrong simulation component is identified, we count it as f_p. If the simulation component cannot be identified, we count it as f_n.

The second scenario, S_2, shows that our approach can identify the suitable simulation component in a set of obfuscated simulation components. Therefore, each simulation component of the two scenarios is compared to each obfuscated component O_1 to O_{20} to find the correct match. For example, we compare C_{C1} to the other obfuscated simulation components O_1 to O_{20}. This example is correctly identified when O_1 is identified as a matching simulation component. It counts as wrongly identified when the result is any of the other simulation components O_2 to O_{20}. If the correct simulation component is identified, we count it as t_p; if a wrong simulation component is identified, we count it as f_p. If the simulation component cannot be identified, we count it as f_n.

The third scenario, S_3, shows that our approach can identify a simulation component in a set of simulation components that represent a DES. Therefore, we modelled a simulation specification C_{C0} that contains all the simulation components C_{C1} to C_{C10} and the simulation specification C_{E0} that contains all the simulation components C_{E1} to C_{E10}. The simulation specification C_{C0} is searched for each component C_{C1} to C_{C10}, and the simulation specification C_{E0} is searched for each component C_{E1} to C_{E10}. For example, we search C_{C0} for the simulation component C_{C1}. This example counts as correctly identified when our approach finds a mapping of the simulation component C_{C1} that is contained in C_{C0} on C_{C1}. It counts as wrongly identified when the result is any other mapping that does not contain the complete elements of C_{C1}. If the suitable simulation component is identified in

C_{C0} or C_{E0}, we count it as t_p; if a wrong simulation component is identified, we count it as f_p. If the simulation component cannot be identified, we count it as f_n.

The fourth scenario, S_4, demonstrates that our approach can identify a simulation component in a set of simulation components that represent a DES even if it was obfuscated. Therefore, we created an obfuscated simulation specification O_0 that contains all simulation components O_1 to O_{20}. The simulation O_0 is searched for each simulation component C_{C1} to C_{C10} and C_{E1} to C_{E10}. For example, we search O_0 for the simulation component C_{C1}. This example counts as correctly identified when our approach finds a mapping of the simulation component O_1 that is contained in O_0 on C_{C1}. It counts as wrongly identified when the result is any other mapping that does not contain the complete elements of C_{C1}. If the proper component is identified in O_0, we count it as t_p; if a wrong component is identified, we count it as f_p, and if the component cannot be identified, we count it as f_n.

The fifth scenario, S_5, serves as an inverse example. It demonstrates that no component can be identified when another set of simulation components that represent a DES, that does not contain any of the components C_{C1} to C_{C10} or C_{E1} to C_{E10} is searched for. For this example, we search the specifications of Camunda for specifications of the Palladio Simulator, and we search the Palladio Simulator specification of Camunda. If a component is not identified, we count it as t_p; if a component is identified, we count it as f_p; if the search yields no result, we count it as f_n. We count it a f_n when the search takes more than one minute and the search has to be terminated.

9.1.4 Simulation Components of the Palladio Simulator used for the Evaluation

The first case study is the simulator *EventSim*; *EventSim* is part of a simulator for the quality analysis of software architectures, the Palladio Simulator. The Palladio-Simulator is part of the Palladio approach [Reu+16]. This tool-supported approach allows for the modelling and analysis of software architectures for different quality properties such as performance, reliability [Bro+12], and maintainability [Ros+17]. EventSim represents a historically grown and versatile model-based analysis that can analyse more than one aspect of software quality. As input, EventSim requires an instance of the PCM. The PCM is also part of the Palladio approach; it allows for the specification, documenting, and analysis of software architectures. The components we selected in our first case study should be reusable in the same domain; hence, each component is related to performance simulation or software architecture simulation. The following components we modelled consist of 48 entities and 60 events:

C_{E1}– *Add Process to Resource:* This component takes a resource, for example, a CPU or HDD, with a fixed set of computational capacity, for example, ten CPU workload units per second. Moreover, it determines whether the demand matches the available capacity of the resource, for example. If the check is successful, the resource is blocked for the time the demand requires, and the following process is added to the list of processes waiting to be processed by the resource.

C_{E2} – *Calculate Resource Demand:* This component calculates the demand for a resource, such as a CPU or HDD, by considering the latency of a linking resource, the demand for the resource, and the throughput.

C_{E3} – *Closed Workload:* This component represents a closed workload with a fixed population count that schedules processes if the queue is not empty. A closed workload contains a fixed number of concurrent workers isolated from one another and completes a defined sequence of tasks (the workload) in a loop. If a process is finished, the population count is increased.

C_{E4} – *Open Workload:* In contrast to the closed workload, where the population is fixed, the open workload determines an interarrival time and schedules processes according to this time. The pace at which new workers are spawned in an open workload is set. Workers are segregated from one another and complete a predefined set of tasks before being removed. Started workers are autonomous, and new workers are launched independently of the status of presently active workers.

C_{E5} – *External Call:* This component calculates the demand when an external resource is called. It contains the number of bytes transmitted to the external resource and the throughput. The throughput can be ignored; then, a fixed demand is returned. ExternalCallAction represents the execution of a service defined in a required interface. As a result, it refers to a Role from which the giving component can be determined and a Signature that specifies the called service. ExternalCallActions represent synchronous calls to necessary services, in which the caller waits for the called service to complete execution before resuming execution.

C_{E6} – *HDD Demand:* This component calculates the demand for a hard disk drive. It contains the read-and-write processing rate and considers the abstract demand of a request when calculating the actual demand.

C_{E7} – *RDSEFF:* This is a rather complex component; it represents the Resource Demand (RD) of a SEFF. A SEFF is the abstract representation of the control flow of a software component. It simulates the effect of calling a specific service of a basic component. To identify the specified service, it refers to a Signature from an Interface for which the component has a ProvidedRole. It depends on C_{E1} and C_{E8}, and C_{E7} invokes these two scenarios.

C_{E8} – *Release Resource:* This component releases a resource if the demand for the resource is fulfilled. Releasing a resource means that a task no longer occupies it, and the resource is free for the next task.

C_{E9} – *Sharing Resources:* Represents a processor sharing resource. It determines whether the capacity of a sharing resource is not exceeded and calculates the processing speed according to the capacity.

C_{E10} – *Usage Scenario:* This component schedules delays and events provided by other components.

9.1.5 Simulation Components of Camunda used for the Evaluation

C_{C1} – *Handle External Task:* C_{C1} receives a task and checks whether the task exists and whether the task is unlocked, and then the task gets scheduled to be executed. C_{C1} consists of a *Worker* entity with an identifier and an *External Task* entity that can be locked. The initial event *execute* of C_{C1} checks whether the *External Task* exists and is not locked. When the conditions for *execute* are met, the C_{C1}'s second event is called: *schedule task.*

C_{C2} – *Lock External Task:* C_{C2} receives a task and checks whether the task exists, then the task gets locked when the expiration date for the lock is set after the current simulation time. C_{C2} consists of a *Task* entity that can be locked and has an expiration date until the task must be resolved. Besides the *Task* entity, it also has a *Date* entity to represent the expiration date of *Task*. The only event of C_{C2} is the *execute* event. It checks whether the lock task is free and whether the expiration date has not passed. If all conditions are met, the task is locked, and an expiration date is set.

C_{C3} – *Resolve Task:* C_{C3} also receives a task and checks whether the task is already resolved and if the task was scheduled by another task (parent task). If the conditions are met, an update event is triggered. C_{C3} consists of a *Task* entity with an assignee and a parent task. If the task is updated, it triggers an event. The initial event *execute* of C_{C3} schedules the *resolve* event and the *trigger update event.* The event *resolve* checks whether the task is finished and has an assignee to inform about the update of the task.

C_{C4} – *Save Task:* C_{C4} takes the revision of a task as input, and if it is the first time the task gets scheduled, the task gets initialised. Then the metrics of the task get saved. C_{C4} consists of a *Command Context* entity, a *Task* entity, a *Metrics* entity, and a *Process Engine Configuration* entity. The initial event *execute* of C_{C4} schedules the *init* event. The *init* event schedules the *metrics calculate* event as well as the *authorisation* event. If the task is authorised, the *trigger update event* is scheduled.

C_{C5} – *Set Task Priority:* C_{C5} receives a task, sets its priority, and schedules an update event so that the simulation can take the new priority of the task into account. C_{C5} consists of a *Task* entity which can be updated. The initial event *execute* of C_{C5} sets the task's priority and schedules a *trigger update event.*

C_{C6} – *Unlock User:* This component unlocks a user if the task is authorised and has admin privileges. C_{C6} consists of a *Context* entity which checks whether a user is an admin and whether a user is authorised. The *User* entity represents the user. The initial event *execute* checks whether a user is an admin and authorised, using the *Context* entity. If the conditions are met, the user gets unlocked via the *unlock user* event.

C_{C7} – *Job Retry:* This component manages the retry of a job. It takes care that a job's retries are correctly handled. C_{C7} consists of the *Job* entity, which has an identifier and a retry count. If a retry is performed, the C_{C7} components schedule the *decrement retry* event to manage the retry count of the *Job* entity.

C_{C8} – *Fetch Events of a Task:* This component is small compared to the remaining components; it gets all the events of a task. C_{C8} consists of a *Task* entity with an identifier and contains a set of events. The *execute* event fetches all events of a given task.

C_{C9} – *Handle Task Escalation:* This component handles the escalated task by verifying that the task still exists and then scheduling an escalation event. C_{C9} consists of a *Task* entity which has an identifier. It also is marked as an active task. The component also consists of the *Escalation* and a *Activity Execution* entity, which determines whether the task is running and which escalation strategy has to be used. The initial event *execute* checks the task by its identifier and the escalation to schedule a *escalation* event depending on the desired escalation strategy. The *escalation* event checks the active task and sets the activity execution.

C_{C10} – *Execute Jobs:* This component checks whether the job and execution context is valid. Then that job is set as the current job, and an event is scheduled that executes the selected job. C_{C10} consists of a *Job* entity which consists of an identifier and a *Job Execution Context*. The initial event *execute* checks the job and its context, and after that, it schedules the *check update job* and then sets the active job for execution.

9.2 Evaluation Results

In this section, we present the results for the applicability of our specification approach where we determine whether we reached our goal Research Goal 9.1: how applicable is our DSML for the specification of DES. We also present the results for the accuracy of our approach to compare specified analysis components based on their structure and behaviour. We determine whether we reached our goals Research Goal 9.2 and Research Goal 9.3: how accurate our approach can identify similar simulation components based on their structure and behaviour specification.

9.2.1 Results for the Applicability Evaluation

	C_{E1}	C_{E2}	C_{E3}	C_{E4}	C_{E5}	C_{E6}	C_{E7}	C_{E8}	C_{E9}	C_{E10}	**Total**
E_{model}	2	4	10	8	3	3	4	1	2	7	**44**
V_{model}	2	2	14	13	1	1	7	2	1	12	**55**

Table 9.1.: Number of Entities and Events per Simulation Component – Palladio Simulator - EventSim

The ten components (C_{E1} to C_{E10}) of the case study Palladio Simulator – EventSim contain 44 entities. The component with the most entities is C_{E3} – *Closed Workload*, with ten entities, and C_{E8} – *Release Resource*, contains one entity; this is the component with the least number of entities. We modelled all 44 entities with our DSML. Besides the entities, the components also contain 55 events in total. The component with the most entities is C_{E3} – *Closed Workload* with a total of fourteen events, and C_{E5} – *External Call*, C_{E6} – *HDD Demand*, and C_{E9} – *Sharing Resources* are the components that contain a single event; thus, they are the components with the least number of events. We modelled each of the 55 events of the case study Palladio Simulator – EventSim.

	C_{C1}	C_{C2}	C_{C3}	C_{C4}	C_{C5}	C_{C6}	C_{C7}	C_{C8}	C_{C9}	C_{C10}	**Total**
E_{model}	2	2	1	4	1	2	1	1	3	2	**19**
V_{model}	2	1	3	6	2	2	1	1	2	6	**26**

Table 9.2.: Number of Entities and Events per Simulation Component – Camunda

The ten components (C_{C1} to C_{C10}) of the case study Camunda contain 19 entities. We modelled all 19 entities with our DSML. Besides the entities, the components also contain 26 events in total. Almost every component contains an event called *execute* as its initial event. The behaviour of this event is different for each component; thus, we had to model each individually. We were able to model each of the 26 events. We designed our DSML to model the structure and behaviour of simulation components. The results show that precisely this is the case. However, without further investigation regarding the identification of components, the ability to model simulations has no benefit or additional value. Therefore, we need the results of the accuracy evaluation to justify the existence of the DSML and our approach to compare and find simulation components based on their specification.

9.2.2 Results for the Accuracy Evaluation

The results for precision, recall and F_1 are depicted in Table 9.3, Table 9.4, and Table 9.5. In Table 9.3, we present the accuracy results for the case study EventSim. In Table 9.4, we present the accuracy results for the case study Camunda, and the total results for both case studies are shown in Table 9.5. Regardless of identical components (S_1) or obfuscated components (S_2), the results for scenarios S_1 and S_2 show that our approach can identify individual components for both case studies. Also, for these scenarios, was no component missing or misinterpreted. These results lead to a score of precision, recall, and F_1 of 1.0. Analysing a whole simulation with all components (S_3) and all obfuscated components (S_4) yielded different case study results. For the EventSim case study in scenario S_3, in three cases, we had to stop the analysis manually, which resulted in seven t_p and three f_n, a precision of 0.70, a recall of 1.00 and F_1 of 0.82. For the EventSim case study in scenario S_4, in one case, we had to stop the analysis manually, which resulted in nine t_p and one f_n, a precision of 0.90, a recall of 1.00 and F_1 of 0.95. For the Camunda case study, in scenarios S_3 and S_4, our approach could identify all ten components in the whole simulation with all components (S_3) and the whole simulation with all obfuscated components (S_4). These results lead to a score of precision, recall, and F_1 of 1.0. The results for the fifth and final scenario S_5 show that for EventSim, all components were not present in the other simulation. These results lead to a score of precision, recall, and F_1 of 1.0. The results for Camunda show that eight of the ten components could not be found in a different simulation. Two components, C_{C7} and C_{C8} had a match in EventSim. These results lead to a precision of 1.00, a recall of 0.80 and F_1 of 0.88. In total, 94 components were identified by our approach; four were missing, and two were falsely identified. The case study EventSim yielded 0.92 for precession, 1.00 for recall, and 0.96 for the F_1 score. The case study Camunda yielded 1.00 for precession, 0.96 for recall, and 0.98 for the F_1 score. The overall results for our evaluation are 0.96 for precision, 0.98 for recall and 0.97 for F_1.

	EventSim S_1	EventSim S_2	EventSim S_3	EventSim S_4	EventSim S_5	**Total**
t_p	10	10	7	9	10	**46**
f_n	0	0	3	1	0	**4**
f_p	0	0	0	0	0	**0**
Prec.	1.00	1.00	0.70	0.90	1.00	**0.92**
Rec.	1.00	1.00	1.00	1.00	1.00	**1.00**
F_1	1.00	1.00	0.82	0.95	1.00	**0.96**

Table 9.3.: Results for the Accuracy Evaluation of the Case Study Palladio Simulator – EventSim

	Camunda S_1	Camunda S_2	Camunda S_3	Camunda S_4	Camunda S_5	**Total**
t_p	10	10	10	10	8	**48**
f_n	0	0	0	0	0	**0**
f_p	0	0	0	0	2	**2**
Prec.	1.00	1.00	1.00	1.00	1.00	**1.00**
Rec.	1.00	1.00	1.00	1.00	0.80	**0.96**
F_1	1.00	1.00	1.00	1.00	0.88	**0.98**

Table 9.4.: Results for the Accuracy Evaluation of the Case Study Camunda

The results for comparing simulation components are promising. We can identify components that match the structure and behaviour of individual components. We had to terminate four searches in the EventSim case study manually. Due to the size and complexity of the case studies, our approach is better suited to search and identify individual components. The two outlier results in scenario S_5 of the Camunda case study are where our approach identified a matching graph in another simulation specification, although no results were expected. We assume that the components C_{C7} and C_{C8} are too small and, therefore, can easily find a matching structure. Regarding the behaviour, C_{C7} has a simple

	EventSim	Camunda	**Total**
t_p	46	48	**94**
f_n	4	0	**4**
f_p	0	2	**2**
Prec.	0.92	1.00	**0.96**
Rec.	1.00	0.96	**0.98**
F_1	0.96	0.98	**0.97**

Table 9.5.: Results for the Accuracy Evaluation of the Case Studies Palladio Simulator and Camunda Compared

decrement operation, a common occurrence in software. C_{C8} is a simple fetch operation that is common in software. Thus, it is no surprise that such small operations can be found in not only one simulation.

9.3 Threats to Validity

Following the classification of Runeson et al. [Run+12], we distinguish four classes of threats to validity. The four validity types are explained in Section 2.6.1.

9.3.1 Internal Validity

In our evaluation, we modelled twenty simulation components of two case studies with our DSML to specify simulation components based on their structure and behaviour. Based on the specified simulation components, we developed an approach to compare their structure and behaviour specifications. One of the case studies is a commercially used process simulation, and the other is a publicly available software performance simulation. Both are used outside a purely scientific context; thus, the case studies represent real-world simulation examples. The evaluation results depend on the quality of the simulation components that have been modelled.

9.3.2 External Validity

We evaluated the modelling and identification of simulation components. As a result, the findings may not be generalisable to other fields. However, the discipline of process and performance simulation comprises general aspects, such as entities, scheduling events, and changing or reading the states of entities. As a result, we demonstrated how our approach might be used for various event-based simulations. The goal is to demonstrate how our approach may help improve the reuse of simulation components.

9.3.3 Construct Validity

The case studies in our evaluation only represent some of the many types of event-based simulations. On the other hand, we did not select an event-based simulation that is just a small running example. We have chosen to extract the simulation components from two open-source simulations used in scientific and commercial environments. The application of our approach to an event-based simulation demonstrates that using our approach for modelling and finding simulation components can identify already existing components. Thus, it is possible to avoid implementing existing simulation components and improve the reuse of simulation components.

9.3.4 Conclusion Validity

To omit that only one researcher must interpret the results of our evaluation, we use objective metrics based on information theory. These metrics provide reasonable proof and limit the need for interpretation, eliminating the possibility of a researcher providing their subjective interpretation of the data. The purpose of the evaluation is to discredit the effects that could be attributed to the interpretation of a single researcher. If the evaluation is repeated for more case studies in the future, this will contribute statistically to the confirmation of the results.

9.4 Discussion

In this section, we discuss if the findings of the applicability and accuracy evaluation of our approach for specifying and finding simulation components indicate that we have accomplished our objectives.

9.4.1 Applicability

The applicability evaluation results show that we could model all of the 20 simulation components of our case studies. We selected existing DES that are established in the community of process analysis and process simulation (Camunda) and the community of performance analysis (Palladio Simulator). We selected ten simulation components for each case study that we specified with our DSL. We chose different sizes of simulation components to have a diverse set of simulation components. The smallest simulation component contains one entity and one event; the biggest contains ten entities and fourteen events. Due to the diversity of the simulation components and our chosen established simulations, we could demonstrate that the first goal Research Goal 9.1 is reached. The results show that our specification language can specify the structure, behaviour, or a real-world simulation.

9.4.2 Accuracy

The results for the accuracy evaluation show promising results regarding identifying simulation components. Regarding the Research Goal 9.2, identifying simulation components based on their structure, the results show that our approach can identify similar structures. However, the structure alone is insufficient to identify a simulation component. The results also show that our approach can reach the Research Goal 9.3, identifying simulation components based on their behaviour. In a set of individual components, we can identify components that match regarding structure and behaviour. For the case study Palladio Simulator, we had in S_3 three simulation components and S_4, we had one simulation component, where we had to stop the analysis. We deduce that our approach is better suited to compare the specification of simulation components to each other instead of searching a whole DES that consists of many specified simulation components. For the case study Camunda, we had two outlier results in scenario S_5. Our approach identified a matching graph in the overall simulation specification, although no results were expected. We assume that the size of the components C_{C7} and C_{C8} are too small, making it easy to find a matching structure. Regarding the behaviour, C_{C7} has a simple decrement operation, a common occurrence in software. C_{C8} is a simple fetch operation, a common occurrence in software. Thus, it is of no surprise that such small operations can be found in not only one simulation.

Part IV.

Epilogue

10. Related Work

In this thesis, we focus on improving the internal quality of software systems that use models of other systems to gain insight without the need to have the system at hand. In this thesis, we focus on models that are based on DSMLs, and we also focus on software systems that take these models as input to reason about them.

In our research, we concentrate on improving the internal quality properties *evolvability*, *understandability*, and *reusability* of model-based analyses. In contrast to the work of Heinrich et al. [HSR19] and Strittmatter [Str20], where they focus on the internal quality of DSMLs or the wide range of improving the internal quality of object-oriented software, we focus on the co-dependency of DSMLs and model-based analyses. We are particularly interested in aligning their architectural structure, i. e. having the same features, components, and dependency structure, and how it affects the aforementioned internal quality properties.

Aligning the structure of both the DSML and the model-based analysis requires that the analysis developer modularises the source code of the model-based analysis. Therefore, we are interested in related research that decomposes analysis code into components. Before the structure of the model-based analysis can be aligned with its corresponding DSML, we are looking into related research that allows the developer to modularise model-based analyses. Besides the decomposition of model-based analyses, we are especially interested in related work that creates model-based analyses from analysis components. In particular, we looked into research that integrates components of model-based analyses with a different notion of time, different DSMLs or components that come from different domains. Furthermore, we are also looking into research that investigates the orchestration of model-based analyses and model-based analyses components.

In addition to the decomposition and composition of model-based analyses, we looked into related research that integrates DSMLs and model-based analyses. We focussed on language workbenches that could integrate a model-based analysis into their development capabilities. In addition to language workbenches, we looked into research that provides tools for developing DSMLs and how they handle tools, especially model-based analyses, that work with the developed DSMLs.

We also researched reoccurring patterns in the structure of DSMLs and model-based analyses that negatively affect the *evolvability*, *understandability*, and *reusability* of model-based analyses that result from the co-dependency of both. Therefore, we looked into related research that copes with detecting bad smells, especially in DSMLs and model-based analysis and how they incorporate their co-dependency. Furthermore, we looked into related research that deals with fixing bad smells in both DSMLs and model-based analyses.

When a DSML and its corresponding model-based analysis follow a modular structure, we wanted to provide an approach to find model-based analysis components that already exist and fit into the requirements of another model-based analysis. The goal is to enhance the reusability of already existing model-based analysis components. In our research, we focus on model-based simulation components, a sub-set of model-based analysis components. Nevertheless, we looked into related research that allows the analysis developer to compare analysis components on the code level. We focus on approaches that support at least object-oriented source code, like code written in Java, C++, or C#. Furthermore, we looked into related research that provides approaches to specify and reuse simulations or parts of simulations.

This chapter is structured as follows: In Section 10.1, we introduce related research related to the decomposition and composition of analyses. This section looks into approaches that cover all kinds of analyses, not only model-based analyses. First, in Section 10.1.1, we address related research that focuses on integrating analyses. Second, in Section 10.1.2, we address related research that focuses on the orchestration of analyses. In Section 10.2, we introduce related research concerned with the integration of DSMLs and model-based analyses. First, in Section 10.2.1, we address related research on language workbenches. Second, in Section 10.2.2, we address related research on language engineering tools. In Section 10.3, we look into related research that covers bad smells in DSMLs and object-oriented software. First, in Section 10.3.1, we address related research that focuses on detecting bad smells. Second, in Section 10.3.2, we address related research that focuses on refactoring bad smells. In Section 10.4, we look into related research that covers the reuse of model-based simulation components. First, in Section 10.4.1, we address related research that compares source code. Second, in Section 10.4.2, we address related research that focuses on the specification and reuse of simulations. Finally, in Section 10.5, we summarise the related work and set the contributions of this thesis in the overall context.

10.1 Decomposition and Composition of Model-based Analyses

Research in the domain of decomposing and composing analyses has yielded relevant contributions to reusing and composing analysis fragments to create analyses. This section presents related approaches to decomposing and composing fragments of analyses.

10.1.1 Analysis Integration

The coupling of analysis components is essential to integrating analyses because the coupling describes how the desired analyses can be combined. There are various coupling approaches for various analyses, some of which are mentioned in the next paragraph.

FMI [Blo+12], DIS [IEE95], and its successor High-Level Architecture (HLA) [IEE10] enable the coupling of simulations on one system but also across distributed systems. DIS and HLA were developed for co-simulation. Co-simulation refers to combining numerous simulations that were not designed to function together in the first place. An overview of challenges and state-of-the-art of co-simulation is presented in [Gom+18]. There are extensions for FMI [Tav+16; Bog+15; FG19] and HLA [Awa+15]. The coupling of FMI and HLA is still the subject of current research [FG19]. Existing approaches for simulation coupling are limited to event exchange to allow interoperability between simulations. However, in contrast to our reference architecture, they only provide the technical structure to couple simulations, but they do not provide a structure or a process of how the components can be (de)composed. However, they focus on aligning the concept of time but still need decomposition and composition concepts.

Another challenge when integrating analyses is the behavioural aspect, manifested by which analysis component influences what [Lam78; MM79]. By synchronising each participating simulation, the decomposition of simulations increases communication overhead. Approaches such as computation allocation [Muz+10; VV14], bridging the hierarchical encapsulation [VV15], or dead-reckoning models [Lee+00] make it possible to reduce the communication effort. Also, the decomposition and composition are left to the user with these approaches. Ptolemy II [Pto14], a framework for actor modelling, an orchestrator block is introduced to manage a set of connected actors. Although an actor can be seen as a simulation unit composed of simulation features, Ptolemy does not provide decomposition support for existing simulations. There also are various approaches to model variability (e. g., [KCO15], [Mén+16b]) utilising feature diagrams to apply product line techniques. However, in contrast to our work, these approaches do not provide a process for identifying and structuring analysis features.

An approach where analysis tools can be merged based on evidence is proposed by Dwyer et al. [DE10]. The authors argue for a standard representation and storage of analytical outcomes and meaningful composition of these results but disregard the analysis structure and the dependent metamodel or language in contrast to our work. ToolBus [BK96] and the Electronic Tool Integration (ETI) [BMW97] platform are two approaches to integrating analyses. Both approaches share similar assumptions and goals: integrating existing tools into foreign processes is a difficult task that requires effective data exchange and communication channels with these technologies. The problem is that, in contrast to our reference architecture, these approaches assume nothing about the structuring of the analysis and do not incorporate any dependent language or metamodel.

At a particular abstraction level, modelling analysis can also be regarded as a form of model transformation. However, the typical model transformation approaches [Tae+05] do not help to cope with the complexity of analytical algorithms. Usually, analysis techniques include complex intermediate states and data structures, but they are used in a straightforward sequential manner. We do not regard the research field of model transformations as more relevant for our endeavour and therefore omit a detailed discussion.

10.1.2 Analysis Orchestration

Analyses like discrete event-based simulations, for example, can be specified by the composition approaches Discrete Event System Specification (DEVS) or Composable Discrete-Event Scalable Simulation (CoDES). Implementing a simulation strictly according to the CoDES [TS08] specification makes a semantic composition of component-based simulations possible. DEVS represents a formalism with which simulations can be modelled and analysed [Zei76]. Also, DEVS offers the possibility to specify parallel running simulations [Cho96]. In contrast to our reference architecture, DEVS and CoDES do not specify how to slice features or incorporate them into a simulation. All these composition approaches also lack a decomposition concept for already existing analyses.

Multi-Paradigm Modelling (MPM) is an approach that proposes to address the challenges associated with the composition of analyses by viewing languages and workflows as distinct paradigms [Amr+19; Amr+21]. MPM is designed to establish the foundations for formalising paradigms, accomplished by representing them as the combination of languages and workflows. In this approach, languages are defined as sets of domain-specific concepts, syntax rules, and semantics, while workflows are sequences of actions executed on a set of inputs to produce a set of outputs. By defining languages and workflows as paradigms, MPM aims to facilitate the development of a formal framework for characterising and analysing paradigms. However, while MPM has been proposed as a promising approach for addressing the challenges associated with the composition of analyses, the outcomes of the approach in the direction of formalisation have yet to be demonstrated. More research is needed to determine the effectiveness of MPM in achieving its objectives and to assess its potential for improving the efficiency and effectiveness of analysis composition. Furthermore, the development of MPM requires the integration of multiple disciplines, including formal methods, domain-specific languages, and workflow management, which presents additional challenges that must be addressed. Despite these challenges, MPM represents a promising approach for advancing the field of analysis composition and is an area of active research.

In conclusion, the current methods used for analysing coupling and orchestration in model-based analyses need to be revised to provide a comprehensive semantic decomposition and composition of analyses. While these approaches can manage the interoperability between model-based analyses, they need to provide a deeper understanding of the meaning and relationships between the different components of the analyses. Coupling and orchestration are essential concepts in model-based analyses, where multiple analyses are combined to provide a more comprehensive understanding of a system or process. However, the current approaches often focus solely on the technical aspects of interoperability, such as data exchange and synchronisation, while neglecting the more nuanced semantic relationships between the different components of the analyses. More scientific and advanced methods are needed to provide a deeper understanding of the meaning and relationships between the different components of the analyses. These methods should focus on semantic decomposition and composition of the model-based analyses, allowing for a more nuanced understanding of how the different components of the analyses relate to one another and how they contribute to the overall understanding of the system or process being studied. Overall, the current approaches to analysing coupling and orchestration in simulations need to be expanded to provide a more comprehensive understanding of the meaning and relationships between the different components of the analyses. This can be achieved by developing more advanced methods focusing on semantic decomposition and composition of the analyses.

10.2 Integration of Domain-specific Modelling Languages and Model-based Analyses

The field of software language engineering has made significant contributions to creating modelling languages by reusing and combining language fragments. This section describes various approaches that have been developed to combine modelling languages with model-based analyses.

10.2.1 Language Workbenches

In this section, we present different tools and workbenches that focus on modelling, building and composition of modelling languages. Neverlang is an open-source language workbench that allows developers to design and implement programming languages and DSLs [CV13]. It provides a set of tools and frameworks for creating and modifying language syntax and semantics, as well as for generating parsers and compilers. The goal of Neverlang is to simplify the process of language design and implementation, making it more accessible to a wider range of developers and users. It allows for the reuse and composition of language components. Neverlang is similar to action-semantics modules [DM03] and role-based composition of language syntaxes with interpreters [Wen12], but it does not support the orchestration of analysis features.

AToMPM, is a framework for building modelling languages and tools, with support for graphical editors and code generation [Syr+13]. GEMOC Studio [CBW17] is an open-source development environment that allows users to create specific DSMLs and their corresponding workbenches. The software provides tools and frameworks, such as graphical editors, simulation engines, model transformation engines, and code generation facilities, which facilitate the creation of DSMLs and workbenches. With GEMOC Studio, users can develop custom DSMLs and workbenches for diverse domains, including software engineering, system engineering, and scientific modelling. The software is based on the EMF, which offers a modular architecture and APIs for modelling tools and applications. Additionally, GEMOC Studio can integrate with other Eclipse-based plugins, like EMF, Xtext, and Sirius. However, the compositionality of GEMOC is purely syntactic, based on the composition mechanisms of the Java programming language. This means that GEMOC Studio does not support the semantic composition of modelling languages and analyses, unlike our reference architecture. LISA [Mer13] and Xtext [Bet16] also support methods for reusing specialised syntax.

Further tools are, for example, MetaEdit+, which is a tool for creating modelling languages and tools, which includes support for creating graphical editors and code generation [TR03], and Papyrus, an open-source modelling tool that provides support for designing and customising modelling languages and tools, with an emphasis on UML modelling [Lan+09]. Meta Programming System (MPS) [Völ11] is an open-source, language workbench and IDE designed for building DSMLs and Language Oriented Programming (LOP) systems. MPS is a software development tool that allows users to design and develop languages. With MPS, users can create their own DSLs and build customised IDEs that are tailored to their specific needs. MPS provides a number of features to support DSL development, including a projectional editor, language composition mechanisms, type-system support, code generation, and refactorings. MPS is written in Java and is built on top of the IntelliJ IDEA platform, which provides a rich set of tools for software development. It is also highly extensible and customisable, allowing users to add their own plugins and tools to the IDE. Another tool for the generation of code, which is based on the programming language python is TextX [Dej+17].

However, none of these workbenches support the semantic composition of model-based analyses, which is a key feature of our reference architecture.

10.2.2 Language Engineering Tools

The field of language engineering has contributed various tools that cater to specific aspects of language artefacts, including syntax definition and transformation specification. Such tools aim to aid in the composition of language artefacts. Some examples of such tools are ATL [Jou+06], Epsilon Transformation Language [KPP08], and Xtend [Bet16] code generation language. However, these tools do not provide support for the semantic composition of analyses. EMF Splitter [Gar+14] is a tool that decomposes monolithic metamodels based on their structure but neglects semantics. In contrast, GTSMorpher [GTS23] is a tool based on GEMOC studio that facilitates the safe composition of behavioural analyses through structured operational semantics. Puzzle [Mén+16a] is an application that detects specification clones and extracts reusable language modules to facilitate the refactoring of modelling languages. This tool helps to improve the maintainability of language artefacts by reducing the duplication of code, which can lead to consistency issues and increase the complexity of the system. EMF Refactor [Fou23] is a tool that identifies and refactors design smells based on model metrics in modelling languages. This tool can help improve the quality of language artefacts by removing any design issues that may lead to incorrect or suboptimal performance. In summary, the field of language engineering has produced a variety of tools that cater to different aspects of language artefacts, including syntax definition, transformation specification, the semantic composition of analyses, and design refactoring. These tools aid in the development of high-quality language artefacts by improving their maintainability, performance, and consistency.

The aforementioned approaches rely on abstract syntaxes, either with restricted variability of the abstract syntax or coupled with interpreters, but do not account for the semantic structure of analyses. To address this limitation, it is imperative for existing techniques to incorporate the semantics of analyses when decomposing and composing modelling languages and model-based analyses. Additionally, when modular model-based analyses are utilised, their composition is strictly syntactic. Regrettably, current approaches disregard the semantics of a given domain or quality attribute.

10.3 Bad Smells and Anti-Pattern in Model-based Analyses

Research in bad smell definition and detection has yielded relevant contributions to improve internal quality attributes of DSMLs and software systems. In this section, we present related approaches regarding detecting and refactoring bad smells and anti-patterns in the domain of DSMLs and software systems.

10.3.1 Bad Smell Detection

Catalogues of bad smells, for example, for code smells [Fow18], Dependency Injection (DI) [Lai+22], or anti-patterns [Bro+98; Lar12] contain descriptions of the bad smells and how they can be detected. However, in contrast to our approach, bad smells that arise due to the co-dependency of DSMLs and model-based analysis are not researched. Furthermore, our approach can automatically detect the thirteen bad smells we found. Studies of code smells [Lac+20; SSS14] focus on code in general; the specific context of DSMLs and corresponding model-based analyses is not researched.

Strittmatter [Str20] proposes a set of bad smells and anti-patterns for DSMLs. They derive bad smells from object-oriented design, and in contrast to a mere catalogue, they provide automated detection of these bad smells. However, they focus solely on DSMLs; the bad smells that arise from the co-dependency of DSMLs and corresponding model-based analyses are not researched.

Llano et al. [LP09] analyse software systems based on their architecture. They use UML-based anti-pattern specifications and propose transformations to correct these anti-patterns. However, their approach focuses on anti-patterns in object-oriented design; ergo, it analyses software on an architectural level. In contrast to our work, it neither does analyse DSMLs regarding bad smells and anti-patterns nor does their approach analyse bad smells that arise from the co-dependency of DSMLs and model-based analyses.

Besides metric-based bad smell detection approaches exist machine learning approaches Kovačević et al. [Kov+22] propose an approach with pre-trained neural source code embeddings for code smell detection. They used pre-trained Code Understanding BERT (CuBERT) embeddings that outperformed the detection of metric-based bad smell. Bidirectional Encoder Representations from Transformers (BERT) and CuBERT are used in the Natural Language Processing (NLP) community for the pre-training of transformer-based NLP models [Dev+18]. BERT can also be used to analyse architectural design decisions [Kei+20] and for the classification of requirements [Hey+20]. However, their approach is limited to detecting the God Class and Long Method smells [Kov+22] or is unable to detect bad smells in DSMLs and model-based analyses [Kei+20; Hey+20].

Tools like SonarQube [Son23] can detect duplicated code, and its capabilities are well researched [Paa16]. The performance of SonarQube as the de facto industry standard is used to measure the capability of similar approaches [FS15]. However, SonarQube cannot handle DSMLs, let alone the co-dependency of DSMLs and model-based analyses.

10.3.2 Bad Smell Refactoring

The effect of refactorings on the internal quality of the software is well-researched. How refactorings affect the security of a system is analysed by Almogahed et al. [AOZ22]. The effects on testability [EA09; EA11; EA12], adaptability [EA11; EA12; MJ19], completeness [EA11; EA12; MJ19], and reliability [AAE13], flexibility [AAE13] are well documented. Furthermore, are the aspects of maintainability and the effect of refactorings [MC16; MJ19; EA12] like understandability, abstraction, modifiability and extensibility also well researched [MC16]. These references show only a small part of the research investigating the effects of refactorings on internal software quality. However, these categories are not sufficiently comprehensive, as there is no broad coverage of refactoring techniques and internal quality attributes. They are limited to a set of refactoring techniques that do not cover the refactorings of DSMLs and model-based analyses.

Strittmatter [Str20] proposes, in addition to the set of bad smells and anti-patterns for DSMLs, refactoring operations to fix aforementioned bad smells in DSMLs. However, as for the detection of their bad smells, they focus solely on DSMLs; the bad smells that arise from the co-dependency of DSMLs and corresponding model-based analyses are not researched.

The Move Method Refactoring Using Coupling, Cohesion, and Contextual Similarity (MMRUC3) approach [Rah+18], developed by Rahmann et al., focuses on the feature envy bad smell. It proposes a solution by analysing the source code and providing the refactoring *move method.* The authors showed that their approach improves the metrics coupling and cohesion. Their MMRUC3 approach uses contextual information based on information retrieval techniques, along with dependency information, to derive the recommendations. To detect refactorings, MMRUC3 utilises the tools Ref-Finder [Bav+15] and JDeodorant [Fok+11]. Although their approach helps increase software modularisation by incorporating static and non-static entities in the recommendation process, their approach is limited to fixing one bad smell. MMRUC3 recommends points in the could that could refactor one bad smell; our approach, on the other hand, can identify up to twelve bad smells. Furthermore, is the MMRUC3 approach limited to smells on the code level, DSMLs and the co-dependency to model-based analysis is not considered.

Carvalho et al. [Car+17b] developed the Refactoring Recommender System (RESYS) approach to link the ontologies Ontology for Software Refactoring (OSORE) and Ontology for Code smell Analysis (ONTOCEAN) [Car+17a] for automatically chose refactorings and semantically link each refactoring to the causing bad smell. The authors based their OSORE ontology on their previous ontology ONTOCEAN. OSORE is a catalogue of refactorings that can use semantic information to support the recommendation of refactorings. It also contains a collection of templates to show how each refactoring can be applied. Our approach and RESYS together with OSORE have in common that both can point to the place in the code where the bad smell is located. However, the approach by Carvalho et al. [Car+17b] is limited to smells on the code level, DSMLs and the co-dependency to model-based analysis is not considered.

Tsantalis et al. [Tsa+13] propose an approach to analyse the refactoring activity in a software development project. The authors extract changes between revisions of the code. Based on the revisions, their tool *Refactoring Miner* and *Ref-Detector* create a simplified UML model. With that model, they derive whether a refactoring was applied in that revision. Their approach is also able to determine which refactoring operation was applied. Cedrim et al. [Ced+17; Ced+16] used the tools *Refactoring Miner* and *Ref-Detector* to determine which refactoring operations are often used, whether a refactoring reduced the total number of bad smells or even if the refactorings introduced new bad smell. Although the overall analysis of the impact of refactorings is fascinating, unfortunately, the authors do not consider bad smells in DSMLs or model-based analyses.

The tool JDeodorant [Fok+11] is a tool that supports the bad smells Feature Envy [TCC08; FTC07], Type/State Checking [TC10; TCC08], Long Method [TC11], God Class [Fok+12; Fok+09] and Duplicated Code [TMK15; TMR17]. This tool can detect bad smells in Java source code and recommend refactorings to resolve the bad smells. Sehgal et al. [SMB17], for example, use JDeodorant to determine the positive effect of refactorings on the internal code quality. However, the JDeodorant is limited to code smells of Java source code; thus, it cannot detect bad smells of DSMLs and the co-dependency to model-based analysis.

Higo et al. [Hig+04] propose the tool CCShaper as a solution for identifying and refactoring the bad smell *Duplicated code*. Their tool utilises the refactoring operations *Extract Method* and *Pull Up Method* to fix the *Duplicated code* smell. However, the CCShaper is limited to code smells of object-oriented source code; thus, it cannot find bad smells that arise from the co-dependency DSMLs and model-based analyses.

Liu et al. [Liu+16] propose an approach that analyses the effect of refactoring methods on other methods. Their change impact analysis is limited to the refactoring of moving a method. They utilise the approach *Extract Method Detector* by Wenmei et al. [LL16] to identify methods that could be extracted. In contrast to our approach, the change impact analysis focuses only on source code; thus, it cannot handle bad smells that arise from the co-dependency of DSMLs and model-based analyses.

Fontana et al. [FZZ15] propose their approach Duplicated Code Refactoring Advisor (DCRA) that can detect the bad smell *Duplicated code* in Java source code. Their approach also suggests refactorings that remove the *Duplicated Code* smell. The claim is that their approach can suggest the best refactoring that solves the *Duplicated Code* smell. Therefore, they classify code clones intending to reduce manual interaction when refactoring the bad smell. In contrast to our approach, the DCRA approach is limited to Java source code and only one bad smell; as a result, it is unable to handle bad smells that arise from the co-dependency of DSMLs and model-based analyses.

10.4 Reuse of Simulation Components

In this section, first, we present related research that is concerned with finding source code identical in structure and behaviour. Then, we present related research that deals with the specification and the reuse of simulations.

10.4.1 Source Code Comparison

Prechtel et al. [PMP02] propose JPlag, a tool to find similarities in Java, C#, C, and C++ source code files to detect software plagiarism. The tool JPlag takes source code as input and compares files pair-wise. For each pair, it computes a similarity score and a set of similarity regions. As an output, it provides a detailed, thorough hyperlink navigable report.

The tool SIM, developed by Gitchel et al. [GT99], can compare programs written in the programming languages C, Java, Pascal, and Lisp. Similar to JPlag, it uses a tokeniser approach to compare the source code. Furthermore, SIM compares the correctness, style and uniqueness of a program. Each programme is first parsed with a lexical analyser, producing a sequence of tokens. The tokens for keywords, special characters and comments are predefined, while the tokens for identifiers are dynamically assigned and stored in a common symbol table.

Schleimer et al. [SWA03] developed the tool Measure Of Software Similarity (MOOS). According to Ahadi et al. [AM19], MOOS supports the programming languages C, C++, Java, C#, Python, Visual Basic, JavaScript, FORTRAN, ML, Haskell, Lisp, Scheme, Pascal, Modula2, Ada, Perl, TCL, Matlab, VHDL, Verilog, Spice, MIPS assembly, a8086 assembly, a8086 assembly, MIPS assembly, and HCL2. MOOS uses the k-gram approach that divides a document into adjacent substrings. The distance k can be determined by the user of MOOS.

Although similar source code files exhibit similar behaviour, the focus of JPlag, SIM and MOOS is finding similarities in the source code structure and not in the program's behaviour.

Bonchi et al. [Bon+18] propose a simulation-based matching of cloud components. Their approach considers the behaviour when matching applications in a cloud environment. The approach by Bonchi et al. [Bon+18] extends the work of Brogi et al. [BS16] to identify whether operations of a component are equivalent. Their approach defines management protocols to determine the equivalence of component operations. Furthermore, the approach by Bonchi et al. can substitute an operation with a sequence of other provided operations to create the desired behaviour. Further approaches, for example, the matchmaking for OWL-S services by Klusch et al. [KFS09] or a heuristic black-box matching approach by Eshuis et al. [EG07], consider the input and output data when matching software components. In their paper, Reussner et al. [RBF04] outline an approach for adjusting components using parametric contracts, which allow for the modification of interfaces based on contextual properties in a potentially more expressive manner than our approach. They utilise finite state machines to model interaction protocols. In contrast to the approaches for matching components and interfaces, our approach does focus on the events and the entities in a DES; none of the presented related approaches considers the effects of events on the overall simulation state. Furthermore, due to the focus on DES, the analysis developer can focus on modelling the interaction of events and entities; thus, it is a lightweight approach as the internal behaviour of a simulation component is not considered.

10.4.2 Simulation Specification and Reuse

In this section, we list related approaches and research concerned with reuse in simulation and the description and comparison of discrete event simulations. The FOCUS approach gives mathematical semantics for the structure and behaviour of software systems [RR11], and it also supports the representation of quality properties and domain-specific properties [Mao+17]. However, these approaches are too broad and ambiguous for non-domain experts to model DES. Various approaches to specifying DES are in use today. Graphical approaches such as UML-based Activity Diagrams, Flow Diagrams or Activity Cycle Diagrams can be used to describe the structure of a simulation and specify the flow of events [BM03]. These diagrams are well suited to convey the behaviour of a simulation to other people but need a formal metamodel for behaviour specification. Usually, the edges are labelled in natural language, complicating the automated comparison of behaviour. Heinrich et al. [HSR19] propose a reference architecture for DSMLs used for quality analysis. Their reference architecture focuses on improving the evolvability and reusability of quality models. However, their reference architecture focuses only on the input models of the quality analysis/simulation.

Approaches like first-order predicate logic [Tom13] investigate logical implications for various forms of logic. Milner [Mil89] investigates relation refinement and various forms of (bi)-simulation dependencies. Clarke et al. [CES86] investigate the satisfaction of temporal logic formulas by automata, and Richters et al. [RG00] check the consistency of object structures regarding data structures (e. g., class structure). The DEVS formalism [ZPK00] is a formal approach to describing and analysing discrete event systems. In the DEVS formalism, a discrete event system model consists of a set of input and output events, a set of states and functions that define the lifetime of states and how the state should be updated in response to input events or the elapsed time of a state. The Coupled DEVS formalism allows the modularisation of system specifications by defining sub-components and their connections. The simplification of the simulation world to a set of states and the behaviour to a sequence of input and output events are a drawback for the specification of a complex system where a set of entities with attributes might be more desirable to describe the simulation world.

There are approaches to combining a formal simulation specification with an intuitive description language, such as the Condition Specification Language [ON85] or the OMNeT++ framework, primarily focused on network simulation. Because of the imperative nature of the behaviour specifications in these approaches (often in C- or Java-like code), it is difficult to extract a description that can be compared between simulations. Our approach allows the straightforward transformation of declarative expressions to SMT-instances and their comparison with an SMT-solver.

The HLA, developed by the Modelling and Simulation Coordination Office of the US Department of Defence, is standard defining an architecture for distributed simulation with a focus on enabling interoperability and reuse [IEE10]. The Object Model Template Specification specifies how HLA federates exchange information through object models, e. g. the Federate Object Model (FOM) that defines data objects and interactions a federation provides. Moeller et al. [ML07] investigate how new developments in the HLA standard can ease the reuse of federates with modular FOMs. The FMI [Blo+12] is a standard defining an interface for exchanging information and coupling between heterogeneous software systems used for Model Exchange and Co-Simulation. Falcone et al. [FG19] combine the HLA and FMI standards to facilitate the reuse of simulation models in complex engineered systems with minimal changes to the reused system. However, these approaches need to cover the identification of components/simulations that match a given specification.

10.5 Discussion

In this chapter, we discussed the related work regarding the three contributions of this thesis. Regarding our first contribution, we discussed the importance of coupling when integrating analyses, and we presented various coupling approaches for simulations, such as FMI, DIS, and HLA. We also highlighted the challenges of co-simulation and the limitations of existing approaches. We have shown that the presented approach either lacks concepts for the decomposition or the composition of model-based analyses. We discussed the behavioural aspect of integrating analyses, such as the communication overhead and decomposition and composition concepts, along with various approaches to address them and that these approaches lack processes for identifying or structuring analysis features. We also briefly touched upon other approaches to integrating analyses, such as ToolBus and the ETI platform, but highlighted their limitations in incorporating dependent language or metamodel. The predicament lies in the fact that, in opposition to our reference architecture, these approaches make no presumptions about the organisation of the analysis and neglect to integrate any interdependent language or metamodel.

We have also discussed that the current approaches used for coupling and orchestration of analyses require revision to provide a more comprehensive semantic decomposition and composition of the analyses. While these approaches can manage the interoperability between the analysis component, they need to provide a deeper understanding of the meaning and relationships between the different components of the analyses. In contrast to our contribution, current methods such as DEVS and CoDES lack a decomposition concept and do not specify how to slice features or incorporate them into a simulation. Advanced methods are required to provide a more nuanced understanding of the meaning and relationships between the different components of the analyses. Our approach focuses on semantic decomposition and composition of the analysis components, allowing for a more comprehensive understanding of model-based analyses and their corresponding DSML. Overall, our approach provides a more comprehensive understanding of the meaning and relationships between the different components of the analyses, which is necessary to advance the field of analysis composition.

Regarding the integration of DSMLs and model-based analyses, tools like the GEMOC Studio that enable the creation of modelling languages using metamodels with built-in interpretation and analyses focus only on syntactic compositionality, whereas our reference architecture supports semantic compositionality of modelling languages and analyses. Other tools also enable language component reuse and composition, but they do not support orchestrating analysis features. Our reference architecture, on the other hand, allows the orchestration of analysis features and analysis components with regard to its corresponding DSML.

The field of language engineering has produced various tools that aid in the development of language artefacts by addressing different aspects, such as syntax definition, transformation specification, design refactoring, and semantic composition of analyses. However, most of these tools do not incorporate the semantics of analyses when decomposing and composing model-based analyses. Current approaches only support the syntactic composition of model-based analyses, but semantic compositionality is essential for more effective and efficient analysis. Therefore, our approach considers the semantics of the domain or quality attributes when composing model-based analyses.

Regarding our second contribution, bad smells and anti-patterns in model-based analyses, we discussed various catalogues for bad smells, such as code smells, design patterns, and anti-patterns, which provide descriptions of the bad smells and how they can be detected. However, in contrast to our contribution, these catalogues do not investigate bad smells arising from the co-dependency of DSMLs and model-based analyses. On the other hand, we have shown approaches for the automated detection of bad smells and anti-patterns for DSMLs, but their approach is limited to DSMLs only. Similarly, other approaches provide anti-patterns for software systems based on their architecture and provide transformations to correct these anti-patterns, but these approaches focus on anti-patterns in object-oriented design and do not analyse bad smells that arise from the co-dependency of DSMLs and model-based analyses. Additionally, machine learning approaches, such as pre-trained neural source code embeddings, have been proposed for bad smell detection, but they are limited in their ability to detect bad smells in DSMLs and model-based analyses. Finally, industry-standard tools like SonarQube can detect duplicated code, but they cannot handle DSMLs or the co-dependency of DSMLs and model-based analyses.

The impact of refactorings on the internal quality of software has been extensively researched, with studies analysing their effects on security, testability, adaptability, completeness, reliability, flexibility, and maintainability. However, these studies do not provide comprehensive coverage of all refactoring techniques and internal quality attributes, with some studies limited to a set of refactoring techniques that do not cover refactoring of DSMLs and model-based analyses. Some studies have proposed solutions to specific bad smells in DSMLs and on the code level, but none has addressed their co-dependency. Other approaches focus on the feature of certain smells or use semantic information to recommend refactorings. In contrast to our contribution, they are all limited to smells on the code level and do not consider the co-dependency between DSMLs and model-based analyses.

Regarding our third contribution, the reuse of simulation components, we discussed tools for source code comparison. For example, JPlag is a tool that uses a pairwise comparison of Java, C#, C, and C++ source code files to detect software plagiarism. It computes a similarity score and similarity regions for each pair of files and produces a detailed report. Similarly, the tool SIM, uses a tokeniser approach to compare C, Java, Pascal, and Lisp programs and evaluates their correctness, style, and uniqueness. In contrast to our contribution, focus the presented tools on detecting similarities in source code structure rather than program behaviour.

We also discussed various approaches and research related to simulation reuse and the description of discrete event simulations. Some of the approaches discussed include FOCUS, UML-based activity diagrams, first-order predicate logic, DEVS formalism, and HLA and FMI standards. While these approaches provide a formal and structured way of describing simulations, they may be too broad or ambiguous for non-domain experts to use. In contrast, our approach can identify simulation components that match a given specification.

11. Conclusion and Future Work

In this chapter, we bring this thesis to an end. For each contribution, we summarise our achievements and discuss whether we reached our research goals. We identify the limitations of our contributions, and we address possible future work. We discuss our first contribution in Section 11.1, the reference architecture for model-based analyses. We discuss our second contribution, the bad smells in model-based analyses, in Section 11.2. In Section 11.3, we discuss our third contribution, the specification and reuse of model-based simulation components.

11.1 Decomposition and Composition of Model-based Analyses

In this section, we conclude our first contribution, the reference architecture for model-based analysis. First, in Section 11.1.1, we summarise the contribution and set our findings into the context of our research questions. In Section 11.1.2, we address the approach's limitations; in Section 11.1.3, we present possible future work.

11.1.1 Summary

As our first contribution, we presented a novel reference architecture for model-based analyses. We used the concept of a reference architecture for DSMLs to create a reference architecture for model-based analyses that considers the architecture of its corresponding DSML. The reference architecture for model-based analyses provides an approach for specifying model-based analyses and an approach to modularise a model-based analysis according to the structure of its associated DSML. By using the structure of the DSML as a guideline for the model-based analysis, the analysis developer has a reference for designing and implementing features of a model-based analysis. Due to the alignment of both the DSML and the model-based analysis, our reference architecture for model-based analyses gives a solution for Problem Statement 1, the deterioration of the evolvability, understandability, and reusability of model-based analyses due to evolutionary changes.

First, we improved the understandability of model-based analyses by introducing a reference architecture for model-based analyses which follows the structure of the reference architecture for DSMLs. Therefore, extended the concept of language features and language components by Heinrich et al. [HSR19] to introduce analysis features and analysis components. A language feature is the expression of a concept, a system property, and a language component is the implementation of a system property. We transferred the concept of features and components to model-based analyses. In the context of a model-based analysis is an analysis feature, the abstraction of the analysis of a system property and the analysis component implements the analysis of the system property. An analysis feature analyses the system property represented by a language feature. The separation into features allows the analysis developer to distinguish the concerns of a model-based analysis. Due to the dependency between the DSML and the model-based analysis, the reference architecture for model-based analyses considers the structure of the model-based analysis and its corresponding DSML. In our approach, each language feature has an associated analysis component; thus, we can ensure that each language feature can be analysed.

Second, we improved the evolvability and reusability of model-based analyses, by arranging the analysis features and components of an model-based analysis into layers. Analysis components that implement an analysis feature are located on the same layer as the analysis feature. The dependencies of the analysis components are restricted. The reference architecture allows only dependencies on analysis components on the same or a more generic layer. Thus, components on the same layer are interchangeable, and changes to components on a more specific layer do not affect components on a more generic layer. In general, the number of layers is determined by the layers of the DSML; however, in our contribution, we created an example instantiation of our reference architecture for model-based quality analyses. Our reference architecture for model-based quality analyses consists of five layers, which supports four layers of the reference architecture for DSMLs (basic features π, domain-specific features Δ, quality-related features Ω, and analysis configuration Σ). In addition to these four layers, we added the experiment layer (Φ) to the structure of the reference architecture. The Φ layer is only part of the model-based analysis; it does not affect the architecture of the DSML. The layered architecture serves as a template structure, reducing internal quality erosion. The template structure prevents the uncontrolled growth of dependencies and the erosion of the project structure (cf. Section 3.2). Because of the layered architecture and the template structure, the model-based analyses that use our reference architecture are better understood. A structure that is easier to understand is also better evolvable, as the architecture of the model-based analysis follows strict dependency and extension rules. The strict dependency rules improve the reusability of analysis features and analysis components.

As a third measure to make model-based analyses better evolvable, understandable, and reusable, we present refactoring operations for analysis developers to transform an existing, arbitrary model-based analysis to our reference architecture. The DSML must already conform to the reference architecture for DSMLs that separates it into layers and ensures that dependencies are directed in a specific way. The refactorings are divided into those that operate on the analysis class level (splitting or merging classes, fixing dependency cycles) and those that operate on the analysis component level (splitting or merging components, extracting features). The refactorings are based on previous work in DSML refactorings and object-oriented programming refactorings. They transform a monolithic model-based analysis into a modular structure that follows our reference architecture. Suppose any problems presented in Section 3.2 (i. e., project structure erosion, uncontrolled growth of dependencies, and feature drift) occur in a model-based analysis. In that case, the refactoring operations are designed to fix and prevent evolvability, understandability, and reusability deterioration.

Besides the layered structure of the reference architecture, we provide processes to apply our reference architecture for model-based analyses. We differentiate three scenarios during the lifetime of a model-based analysis where an analysis developer can apply the reference architecture: (i) refactoring an existing model-based analysis, (ii) developing a model-based analysis from scratch, and (iii) extending an existing model-based analysis. Although the processes restrict the analysis developers' freedom to design, implement, and extend model-based analyses, they provide a structure for the analysis developers can follow, which unifies the design, development, and extension process [HSR19]. Because of the processes, analysis developers can easily apply our reference architecture, which in return makes the affected model-based analyses better evolvable, understandable, and reusable.

For the evaluation of our approach, we refactored four case studies from different domains to our reference architecture for model-based analyses. Due to the size of the case studies, we focussed on historical evolution scenarios. We derived these scenarios from the commit history of the case studies. We refactored the historical evolution scenarios according to our guidelines using the refactoring operations provided in Section 3.3.3. The case studies are (i) the Palladio Simulator, a performance and reliability analysis for component-based software systems; (ii) Camunda, a business process analysis and workflow engine; (iii) KAMP4aPS, a maintainability analysis for automated production systems; and (iv) SmartGrid, an impact and resilience analysis of energy networks.

We used the four case studies to evaluate whether the application of our reference architecture improved the evolvability, understandability, and reusability of the model-based analyses, and thus, our first contribution solves Problem Statement 1. We chose the entropy-based metrics *complexity, coupling*, and *cohesion*. We calculated the metrics on hypergraphs that we transformed from the source code of the case studies. The entropy-based metrics are better suited than simple counting metrics. Per case study, we extracted ten historical evolution scenarios. After refactoring the forty evolution scenarios, we compared the complexity, cohesion, and coupling of the modular model-based analyses with the original, monolithic, model-based analyses' complexity, cohesion, and coupling. The results for cohesion and coupling show that they are interchangeable. Improving one of them results in the deterioration of the other. However, we could demonstrate that for the evolution scenarios, the complexity was reduced. Therefore, the results show that our first hypothesis (Hypothesis 1) is true: the evolvability, understandability and reusability of model-based analyses improve when we transfer the concepts of the reference architecture for DSMLs to model-based analyses. We could transfer the concepts of the reference architecture for DSMLs to model-based analyses. With the evaluation, we have shown that our reference architecture reduced the complexity of the refactored case studies; thus, we can answer Research Question 3.1 and Research Question 3.2 that our reference architecture is able to improve the evolvability and understandability of model-based analysis. We could also answer Research Question 3.3, that due to the reduced complexity, our reference architecture improves the reusability of model-based analysis.

11.1.2 Limitations

In this thesis, we investigated model-based analyses that work with EMOF-based DSMLs. Therefore, we can only claim that our reference architecture for model-based analyses improves the evolvability, understandability, and reusability of model-based analyses that work with EMOF-based DSMLs. Developers, for example, could use ontologies or grammars to create DSMLs; however, we cannot claim that our reference architecture also works for ontology-based or grammar-based DSMLs.

Another limitation is that the DSML must follow the reference architecture for DSMLs. If the DSML cannot be changed, we cannot determine whether our reference architecture has the same impact on the evolvability, understandability, or reusability. Furthermore, we only evaluated model-based analyses that analyse quality attributes of their corresponding DSML. As a result, we cannot claim that our reference architecture for model-based analysis has the same positive impact on other kinds of model-based analyses.

Furthermore, in our evaluation, we investigated three types of model-based quality analyses. We have shown that our reference architecture improves the evolvability, understandability, and reusability of discrete event simulations, process analyses, and change propagation analyses. Our reference architecture for model-based analysis is so designed that it applies to other model-based analyses, as long as they work with a DSML. However, as we only investigated the types of analyses mentioned above, we cannot claim that our reference architecture for model-based analyses generally improves the evolvability, understandability, and reusability of all kinds of model-based analyses.

11.1.3 Future Work

The evaluation of our reference architecture for model-based analyses could be extended to include case studies that analyse different attributes of their corresponding DSML instead of only quality attributes to address the limitations. Furthermore, as long as the software works with a DSML, we could investigate whether our reference architecture is applicable for model-based software in general. Therefore, we must modularise further DSMLs and software systems, preferably of different domains.

In this thesis, we focused on DSMLs that are used for quality analyses; however, DSMLs are also used for other purposes, for example, to model software and generate code. Aligning the structure of the generated code or the transformations that generate the code with the reference architecture could also improve the understandability and reusability of the generated code and the evolvability, understandability, and reusability of the transformations.

Furthermore, the tooling could be extended to create the foundation for an integrated toolset to develop model-based analyses and model-based software. The integrated tooling could support automated analysis and refactoring of model-based software. In addition to the automation of our tooling, we plan to add features for visualising the feature and component structure to allow drag-and-drop actions to simplify the refactoring process.

11.2 Bad Smells in Model-based Analyses

In this section, we conclude our second contribution, the model-based analysis bad smells. First, in Section 11.2.1, we summarise the contribution and set our findings into the context of our research questions. In Section 11.2.2, we address the approach's limitations, and in Section 11.2.3, we present possible future work.

11.2.1 Summary

As our second contribution, we presented 12 novel bad smells in model-based analyses that arise because of the co-dependency of a DSML and their associated model-based analyses. This contribution is the solution for Problem Statement 2, avoiding the deterioration of the evolvability, understandability, and reusability of model-based analyses because of the co-dependency of model-based analyses and their corresponding DSML. In addition to describing the bad smells, we provided strategies to identify each bad smell and proposed a refactoring strategy per bad smell. We discussed the potential for bad smells in model-based analysis and the corresponding DSMLs, and we found that the co-dependency of these areas regarding bad smells has not been explored yet. We identified bad smells specific to model-based analyses. Therefore, we analysed existing model-based analyses and derived 12 bad smells from bad smells in object orientation and bad smells in DSMLs. We separated the 12 bad smells into four categories, introduced by Ganesh et al. [GSS13]: *abstraction*, *encapsulation*, *modularity*, and *hierarchy*. The categories help the analysis developer to understand the cause and effect of the bad smells on a broader scale. We found three bad smells for the categories *abstraction* and *hierarchy*, respectively. We found five bad smells for the category *modularity*, and for the category *encapsulation*, we found one bad smell.

First, we presented the 12 bad smells we found. We analysed the bad smells regarding their adverse effects on the evolvability, understandability, and reusability of each bad smell and the causes that lead to the occurrences of the bad smells. For each bad smell, we presented a process for the analysis developer to identify and refactor them. These processes help the analysis developer to better understand the root of the bad smell, and they provide a solution to make the model-based analysis better evolvable, understandable, and reusable. We started with the bad smells in the *abstraction* category. An *abstraction* refers to identifying and representing an object's fundamental characteristics that distinguish it from other types of objects. This process results in establishing clearly defined conceptual boundaries, as the observer perceives.

Following the *abstraction* category, we presented the bad smells of the *encapsulation* and of the *hierarchy* category. *Encapsulation* is a software design principle that involves modularising the elements of an abstraction that determine its behaviour and structure. The primary objective of *encapsulation* is to maintain the separation between the interface and implementation of abstraction to promote *encapsulation* and maintain the integrity of the abstraction. A hierarchical organisation, or order, of abstractions, is referred to as a *hierarchy*. In model-based analysis, bad smells of the hierarchical type can be identified when the analysis is developed following our reference architecture for model-based analyses.

The *modularisation* category is the last one we discussed. Modularity refers to the ability of a system to be divided into a collection of self-contained and loosely coupled modules. In the context of model-based analysis, the loose coupling of modules allows for modifications to be made to individual modules without affecting the functionality of other modules. Additionally, a well-defined dependency structure, such as that provided by a reference architecture for model-based analysis, can enhance the evolvability and reusability of model-based analyses.

For the evaluation of our second contribution, we analysed four case studies from different domains and whether they contained bad smells. Due to the size of the case studies and the number of bad smells, we did not fix every bad smell that occurred. We focussed on historical evolution scenarios derived from the case studies commit history. The case studies are (i) the Palladio Simulator, a performance and reliability analysis for component-based software systems; (ii) Camunda, a business process analysis and workflow engine; (iii) KAMP4aPS, a maintainability analysis for automated production systems; and (iv) SmartGrid, an impact and resilience analysis of energy networks.

We used the four case studies to evaluate the relevance of the bad smells by determining the number of occurrences of the bad smells. To count the occurrences of bad smells, we implemented an automated identification analysis, allowing us to analyse all case studies' complete source code. Furthermore, we evaluated the effects of the bad smells on evolvability, understandability, and reusability. We chose entropy-based metrics complexity, coupling, and cohesion that we extracted by transforming the source code into hypergraphs. The entropy-based metrics are better suited than simple counting metrics. Per case study and bad smell, we refactored up to ten occurrences. After refactoring the bad smells, we compared the complexity, cohesion, and coupling of the scenario with the original scenario's complexity, cohesion, and coupling. We could show that for the evolution scenarios, the complexity could be reduced for the *Duplicated Abstraction*, the *Degraded Modularity*, and the *Rebellious Modularity* smell. Fixing the *Missing Abstraction*, bad smells showed mixed results because refactoring primitive types can result in more types and, thus, more dependencies. The bad smells of the hierarchical type emerge due to the wrong application of our reference architecture. The evaluation of our reference architecture has shown that it positively affects the evolvability, understandability, and reusability of model-based analyses. Therefore, the results show that our first hypothesis (Hypothesis 2) is true: the evolvability, understandability and reusability of model-based analyses improve when we fix bad smells that originate from the co-dependency of model-based analyses and their corresponding DSML.

We answered Research Question 4.1 by deriving bad smells from object-oriented software development and DSML development and searching our four case studies for the occurrences of the bad smells of model-based analyses. We answered Research Question 4.2 by developing refactoring strategies and applying these refactorings to the occurrences in our case studies. To answer the last research question Research Question 4.3, we refactored the bad smells we found in our four case studies. With the evaluation, we have shown that the bad smells of model-based analyses impede the evolvability, understandability and reusability of model-based analyses.

11.2.2 Limitations

We derived our bad smells by investigating model-based analyses that analyse quality attributes of systems modelled with their corresponding DSML. Thus, we cannot determine whether the bad smells also apply to model-based analyses that analyse different system attributes, even if the bad smells are generally valid for model-based analyses. Furthermore, we looked only at DSMLs based on the EMOF standard. Therefore, we can only claim that our bad smells of model-based analyses affect the evolvability, understandability, and reusability of model-based analyses that work with EMOF-based DSMLs. We must determine if the number of bad smells we identified is complete. Another limitation is that the DSML must be changed to fix bad smells. When the DSML cannot be changed, for example, because it follows a standard, we cannot fix the bad smells that require a change of the DSML.

It can be difficult for the analysis developer to identify the source of the smell. In some cases, it can be challenging to pinpoint the exact source of a bad smell in a software system, which can make it challenging to address the problem. It can also be difficult for the analysis developer to understand the source code and the metamodel. Even if the source of a bad smell is identified, it may be challenging to understand the code, especially if it is a legacy codebase or written by someone else. Due to the co-dependency of the model-based analyses and its DSML, the impact on other system components can be difficult to determine before fixing the bad smell. Fixing bad smells in one part of the system may have unintended consequences on other parts of the system, which can be challenging to anticipate and address. The process of fixing bad smells can be time-consuming. Identifying and fixing bad smells in a model-based analysis can be tedious, especially if the model-based analysis is large and complex.

11.2.3 Future Work

To address the limitations, further model-based analyses, especially analyses of different domains, could be analysed to derive more bad smells. Especially software that works with a DSML could be investigated first to determine whether the bad smells also apply to model-based software in general and second, to determine whether the bad smells can improve the evolvability, understandability, and reusability of model-based software in general. Furthermore, we plan to extend our tooling to create the foundation for an integrated toolset to develop model-based analyses and model-based software. The integrated tooling could support automated analysis and refactoring of bad smells of model-based software. In addition to the automation of our tooling, we plan to add features for visualising the feature and component structure to allow drag-and-drop actions to simplify the refactoring process.

11.3 Structure and Behaviour Specification and Reuse of Model-based Analysis Components

In this section, we conclude our third contribution, the specification and reuse of the structure and behaviour of model-based analysis components. First, in Section 11.3.1, we summarise the contribution and set our findings into the context of our research questions. In Section 11.3.2, we address the approach's limitations; in Section 11.3.3, we present possible future work.

11.3.1 Summary

As our third contribution, we presented a novel domain-specific modelling language for specifying simulation components' structure and behaviour. Our approach is dedicated to decreasing the effort required to reuse model-based analysis components. This contribution serves as the solution to Problem Statement 3, the increasing complexity and the reduced reusability of model-based analyses due to historical changes. Developers can use the specification to compare and identify simulation components that match the desired specification. The structure comparison transforms the specification into a graph notation. We use a graph-isomorphism approach to identify similar structures of specified simulations based on the graph notation. The behaviour comparison transforms the specification into an SMT notation, which we use to identify similar behaviour of specified simulations. Finding similar simulation components enables developers to reuse existing simulation components and reduce the effort required to develop new simulation components.

Our contribution involves identifying and refactoring bad smells in model-based analysis, and we presented a process for modularising existing analyses. The goal is to make a repository filled with analysis components or publicly available analysis components searchable. We did emphasise the importance of modularisation for enabling the reuse of analysis components. We also note that specifying and identifying a component with a desired structure and behaviour is also a part of reusing an analysis component. When the complexity of a model-based analysis increases, it becomes harder to understand and maintain, extend, or reuse. To address this problem, we suggest reusing model-based analysis components in future projects to save time and resources. However, a model-based analysis for a specific domain or system can limit its reusability for other domains or systems. We discussed the difficulty of determining whether a discovered component is a semantic match (i. e. exhibits the required behaviour) for a reuse candidate, as it may be challenging to determine whether the component is a semantic match, mainly if the number of components to be analysed large or the components are complex. We presented an approach for specifying the structure and behaviour of model-based analyses using a modelling technique based on metamodels and a Domain-Specific Language (DSL). The approach also includes a method for identifying similar model-based analysis components by comparing them in structure and behaviour. The process of comparing components is divided into two stages, first by comparing the structure of the components using graph notation and graph-isomorphism analysis and second by comparing their behaviour using Satisfiable Modulo Theories (SMT) notation and an SMT-solver.

For the evaluation of our third contribution, we used two case studies to evaluate the applicability of the DSML. Then we modelled these components with our specification language. Therefore, we calculated the coverage by counting the entities and events we could model for each simulation component. Furthermore, we evaluated the accuracy of our approach to identifying similar simulation components based on their structure and behaviour specifications. We developed five evaluation scenarios that cover the search for a simulation component in a set of components and the search for the functionality of a component in a larger component. Furthermore, we obfuscated the components we searched for in three scenarios, showing that our approach does not simply compare the names of the entities and events contained in the specifications. We determined the accuracy of our approach by calculating the F1 score, a metric that combines precision and recall. The F1 score is calculated by comparing the identified components with the expected components and counting the number of true positives, false positives, and false negatives. The identification of these values is specific to the scenario being considered.

The evaluation results show that our third hypothesis (Hypothesis 3) is true: the reusability of model-based analyses improves when we reduce the barrier of finding reusable analysis components. We answered Research Question 5.1 by developing a DSL to specify model-based simulation components' structure and behaviour. We evaluated this research question by specifying the components of two simulations. By modelling these existing simulations, we evaluated the applicability of our approach. Besides modelling the structure and the behaviour of simulation components, we also evaluated the accuracy of identifying similar simulation components. We answered Research Question 5.2 by comparing the specifications of the simulation components and identifying the right match. We utilised our approach to compare the specified components with those of the case studies to assess the accuracy of our approach. The findings show that our approach can identify simulation components with similar structures and behaviour.

11.3.2 Limitations

In this work, however, we have only tested the applicability of our approach by modelling and analysing two case studies. We plan to model more simulations to investigate our approach's applicability further. We derived our specification DSL by investigating model-based analyses that analyse quality attributes of systems modelled with their corresponding DSML. We looked only at model-based DES and no other analyses. Therefore, we can only claim that our approach improves the reusability of model-based DES. Another limitation is that to specify simulation components, we need manual labour for the creation. Thus, we cannot exclude manual errors and specification styles, which can negatively affect the result when comparing the specifications. Furthermore, we evaluated only model-based DES that analysed quality attributes of their corresponding DSML. As a result, we cannot claim that our approach works for DES or even model-based analyses in general.

11.3.3 Future Work

To address the limitations, our approach to specifying and comparing model-based simulation DES could be extended to include model-based DES components that analyse different attributes of their corresponding DSML. The specification approach could be extended to support model-based analyses in general. In this case, the challenge is finding a formalism that generalises events. In this contribution, we focused on model-based simulations used for quality analyses; however, simulations are also used for other purposes. Making all kinds of analysis components searchable could improve the reusability of analysis components in general. Furthermore, we plan to extend our tooling to create the foundation for an integrated tool-set to develop model-based analyses and model-based software. In the future, we intend to provide a transformation to extract the simulation specification automatically or semi-automatically. We also intend to apply our approach to other domains to investigate different application areas.

Bibliography

[AAE13] M. Alshayeb, H. Al-Jamimi, and M. O. Elish. “Empirical taxonomy of refactoring methods for aspect-oriented programming”. In: *Journal of Software: Evolution and Process* 25.1 (2013), pp. 1–25.

[AGG07] E. B. Allen, S. Gottipati, and R. Govindarajan. “Measuring size, complexity, and coupling of hypergraph abstractions of software: An information-theory approach”. In: *Software Quality Journal* 15.2 (2007), pp. 179–212.

[AK09] S. Apel and C. Kästner. “An overview of feature-oriented software development.” In: *J. Object Technol.* 8.5 (2009), pp. 49–84.

[All02] E. B. Allen. “Measuring graph abstractions of software: an information-theory approach”. In: *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on.* 2002, pp. 182–193.

[AM19] A. Ahadi and L. Mathieson. “A Comparison of Three Popular Source code Similarity Tools for Detecting Student Plagiarism”. In: *ACM International Conference Proceeding Series.* Association for Computing Machinery, 2019, pp. 112–117.

[Amr+19] M. Amrani, D. Blouin, R. Heinrich, A. Rensink, H. Vangheluwe, and A. Wortmann. “Towards a Formal Specification of Multi-Paradigm Modelling”. In: *22nd International Conference on Model Driven Engineering Languages and Systems Companion.* IEEE. 2019, pp. 419–424.

[Amr+21] M. Amrani, D. Blouin, R. Heinrich, A. Rensink, H. Vangheluwe, and A. Wortmann. “Multi-paradigm modelling for cyber–physical systems: a descriptive framework”. In: *Software and Systems Modeling* 20 (2021), pp. 611–639.

[AOZ22] A. Almogahed, M. Omar, and N. H. Zakaria. “Refactoring codes to improve software security requirements”. In: *Procedia Computer Science* 204 (2022), pp. 108–115.

[Ape+08] S. Apel, C. Lengauer, B. Möller, and C. Kästner. “An algebra for features and feature composition”. In: *Lecture Notes in Computer Science* 5140 LNCS (2008), pp. 36–50.

[Awa+15] M. U. Awais, W. Mueller, A. Elsheikh, P. Palensky, and E. Widl. “Using the HLA for distributed continuous simulations”. In: *Proceedings - 8th EUROSIM Congress on Modelling and Simulation.* IEEE, 2015, pp. 544–549.

[Bab16] L. Babai. “Graph isomorphism in quasipolynomial time”. In: *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing.* 2016, pp. 684–697.

[Bav+15] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba. “An experimental investigation on the innate relationship between quality and refactoring”. In: *Journal of Systems and Software* 107 (2015), pp. 1–14.

[BCE08] H. P. Breivold, I. Crnkovic, and P. J. Eriksson. “Analyzing Software Evolvability”. In: *32nd Annual IEEE International Computer Software and Applications Conference*. 2008, pp. 327–330.

[BCR94] V. R. Basili, G. Caldiera, and H. D. Rombach. “The goal question metric approach”. In: *Encyclopedia of Software Engineering* 2 (1994), pp. 528–532.

[Bet16] L. Bettini. *Implementing domain-specific languages with Xtext and Xtend.* Packt Publishing, 2016.

[BFT17] C. Barrett, P. Fontaine, and C. Tinelli. *The SMT-LIB Standard: Version 2.6.* Tech. rep. Department of Computer Science, The University of Iowa, 2017.

[BK96] J. A. Bergstra and P. Klint. “The ToolBus coordination architecture”. In: *International Conference on Coordination Languages and Models.* Springer. 1996, pp. 75–88.

[BKL83] L. Babai, W. M. Kantor, and E. M. Luks. “Computational complexity and the classification of finite simple groups”. In: *24th Annual Symposium on Foundations of Computer Science.* 1983, pp. 162–171.

[BKR09] S. Becker, H. Koziolek, and R. Reussner. “The Palladio component model for model-driven performance prediction”. In: *Journal of Systems and Software* 82 (2009), pp. 3–22.

[Blo+12] T. Blockwitz, M. Otter, J. Akesson, M. Arnold, C. Clauß, H. Elmqvist, M. Friedrich, A. Junghanns, J. Mauss, D. Neumerkel, H. Olsson, and A. Viel. “Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models”. In: *9th International Modelica Conference.* 2012.

[BM03] S. Balsamo and M. Marzolla. “Simulation modeling of UML software architectures”. In: *17th European Simulation Mulitconference.* Vol. 3. Society for Modelling and Simulation International. SCS European Publishing House, 2003, pp. 562–567.

[BMB96] L. C. Briand, S. Morasca, and V. R. Basili. “Property-based software engineering measurement”. In: *IEEE Transactions on Software Engineering* 22.1 (1996), pp. 68–86.

[BMW97] V. Braun, T. Margaria, and C. Weise. “Integrating tools in the ETI platform”. In: *International Journal on Software Tools for Technology Transfer (STTT)* 1 (1997), pp. 31–48.

[Bog+15] S. Bogomolov, M. Greitschus, P. G. Jensen, K. G. Larsen, M. Mikucionis, T. Strump, and S. Tripakis. “Co-Simulation of Hybrid Systems with SpaceEx and Uppaal”. In: *Proceedings of the 11th International Modelica Conference.* Vol. 118. Linköping University Electronic Press, 2015, pp. 159–169.

[Bon+18] F. Bonchi, A. Brogi, A. Canciani, and J. Soldani. "Simulation-based matching of cloud applications". In: *Science of Computer Programming* 162 (2018), pp. 110–131.

[Bro+12] F. Brosch, H. Koziolek, B. Buhnova, and R. Reussner. "Architecture-Based Reliability Prediction with the Palladio Component Model". In: *IEEE Transactions on Software Engineering* 38.6 (2012), pp. 1319–1339.

[Bro+98] W. H. Brown, R. C. Malveau, H. W. S. McCormick, and T. J. Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis.* John Wiley & Sons, Inc., 1998.

[BS01] M. Broy and K. Stølen. *Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement.* Springer, 2001.

[BS16] A. Brogi and J. Soldani. "Finding available services in TOSCA-compliant clouds". In: *Science of Computer Programming* 115 (2016), pp. 177–198.

[Bus+18] K. Busch, J. Rätz, S. Koch, R. Heinrich, R. Reussner, S. Cha, and B. Vogel-Heuser. "A Metamodel-Based Approach to Calculate Maintainability Task Lists of PLC Programs for Factory Automation". In: *44th Annual Conference of the IEEE Industrial Electronics Society (IECON).* IEEE, 2018.

[But+19] A. Butting, R. Eikermann, O. Kautz, B. Rumpe, and A. Wortmann. "Systematic Composition of Independent Language Features". In: *Journal of Systems and Software* 152 (2019), pp. 50–69.

[BWL01] L. C. Briand, J. Wüst, and H. Lounis. "Replicated Case Studies for Investigating Quality Factors in Object-Oriented Designs". In: *Empirical Software Engineering* 6.1 (2001), pp. 11–58.

[Cai+21] D. Caivano, P. Cassieri, S. Romano, and G. Scanniello. "An Exploratory Study on Dead Methods in Open-Source Java Desktop Applications". In: *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement.* ESEM. Association for Computing Machinery, 2021.

[Car+17a] D. S. L. P. Carvalho, R. Novais, D. N. L. Salvador, and D. M. M. G. Neto. "An ontology-based approach to analyzing the occurrence of code smells in software". In: *ICEIS 2017 - Proceedings of the 19th International Conference on Enterprise Information Systems.* Vol. 2. 2017, pp. 155–165.

[Car+17b] L. P. d. S. Carvalho, R. L. Novais, L. d. N. Salvador, and M. G. d. M. Neto. "An approach for semantically-enriched recommendation of refactorings based on the incidence of code smells". In: *International Conference on Enterprise Information Systems.* Springer. 2017, pp. 313–335.

[CBW17] B. Combemale, O. Barais, and A. Wortmann. "Language Engineering with the GEMOC Studio". In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW).* 2017, pp. 189–191.

[CE00] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* Addison Wesley, 2000, p. 864.

[Ced+16] D. Cedrim, L. Sousa, A. Garcia, and R. Gheyi. “Does refactoring improve software structural quality? a longitudinal study of 25 projects”. In: *Proceedings of the 30th Brazilian Symposium on Software Engineering.* 2016, pp. 73–82.

[Ced+17] D. Cedrim, A. Garcia, M. Mongiovi, R. Gheyi, L. Sousa, R. de Mello, B. Fonseca, M. Ribeiro, and A. Chávez. “Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects”. In: *Proceedings of the 2017 11th Joint Meeting on foundations of Software Engineering.* 2017, pp. 465–475.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. “Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications”. In: *ACM Trans. Program. Lang. Syst.* 8.2 (1986), pp. 244–263.

[CG20] J. P. Castellanos Ardila and B. Gallina. “Separation of Concerns in Process Compliance Checking: Divide-and-Conquer”. In: *Systems, Software and Services Process Improvement.* Springer International Publishing, 2020, pp. 135–147.

[Cho96] A. C. Chow. “Parallel DEVS: A parallel, hierarchical, modular modeling formalism and its distributed simulator”. In: *Transactions of the Society for Computer Simulation* 13.2 (1996), pp. 55–67.

[CKM22] A. Cortinovis, D. Kressner, and S. Massei. “Divide-and-Conquer Methods for Functions of Matrices with Banded or Hierarchical Low-Rank Structure”. In: *SIAM Journal on Matrix Analysis and Applications* 43.1 (2022), pp. 151–177.

[CLZ04] H. C. Cunningham, Y. Liu, and C. Zhang. “Using the divide and conquer strategy to teach Java framework design”. In: *Proceedings of the 3rd international symposium on Principles and practice of programming in Java.* 2004, pp. 40–45.

[Com+18] B. Combemale, J. Kienzle, G. Mussbacher, O. Barais, E. Bousse, W. Cazzola, P. Collet, T. Degueule, R. Heinrich, J.-M. Jézéquel, M. Leduc, T. Mayerhofer, S. Mosser, M. Schöttle, M. Strittmatter, and A. Wortmann. “Concern-Oriented Language Development (COLD): Fostering Reuse in Language Engineering”. In: *Computer Languages, Systems & Structures* (2018).

[Cor22] S. Cordio. *csb/neo4j-plugins/subgraph-isomorphism at master · msstate-dasi/csb.* 2022.

[CR94] V. R. B. G. Caldiera and H. D. Rombach. “The goal question metric approach”. In: *Encyclopedia of software engineering* (1994), pp. 528–532.

[Cru+10] J. A. Cruz-Lemus, A. Maes, M. Genero, G. Poels, and M. Piattini. “The Impact of Structural Complexity on the Understandability of UML Statechart Diagrams”. In: *Information Sciences* 180.11 (2010), pp. 2209–2220.

[CV13] W. Cazzola and E. Vacchi. “Neverlang 2–Componentised Language Development for the JVM”. In: *International Conference on Software Composition.* Springer. 2013, pp. 17–32.

[Cza+12] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, and A. Wąsowski. “Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches”. In: *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*. VaMoS ’12. Association for Computing Machinery, 2012, pp. 173–182.

[DB11] L. De Moura and N. Bjørner. “Satisfiability modulo Theories: Introduction and Applications”. In: *Commun. ACM* 54.9 (2011), pp. 69–77.

[DE10] M. B. Dwyer and S. Elbaum. “Unifying verification and validation techniques”. In: *FSE/SDP workshop on Future of software engineering research*. ACM, 2010.

[Dej+17] I. Dejanović, R. Vaderna, G. Milosavljević, and Ž. Vuković. “Textx: a python tool for domain-specific languages implementation”. In: *Knowledge-based systems* 115 (2017), pp. 1–4.

[Dev+18] J. Devlin, M. Chang, K. Lee, and K. Toutanova. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *Computing Research Repository (CoRR)* abs/1810.04805 (2018).

[DLK94] R. B. Deal, A. M. Law, and W. D. Kelton. “Simulation Modeling and Analysis”. In: *Technometrics* 36.4 (1994), pp. 2–8.

[DM03] K.-G. Doh and P. D. Mosses. “Composing programming languages by combining action-semantics modules”. In: *Science of Computer Programming* 47.1 (2003), pp. 3–36.

[DMV20] S. Dustdar, O. Mutlu, and N. Vijaykumar. “Rethinking Divide and Conquer—Towards Holistic Interfaces of the Computing Stack”. In: *IEEE Internet Computing* 24.6 (2020), pp. 45–57.

[Dur21] J. M. Durán. “A Formal Framework for Computer Simulations: Surveying the Historical Record and Finding Their Philosophical Roots”. In: *Philosophy & Technology* 34.1 (2021), pp. 105–127.

[EA09] K. O. Elish and M. Alshayeb. “Investigating the Effect of Refactoring on Software Testing Effort”. In: *2009 16th Asia-Pacific Software Engineering Conference*. 2009, pp. 29–34.

[EA11] K. O. Elish and M. Alshayeb. “A classification of refactoring methods based on software quality attributes”. In: *Arabian Journal for Science and Engineering* 36.7 (2011), pp. 1253–1267.

[EA12] K. O. Elish and M. Alshayeb. “Using Software Quality Attributes to Classify Refactoring to Patterns.” In: *Journal of Software* 7.2 (2012), pp. 408–419.

[EG07] R. Eshuis and P. Grefen. “Structural matching of bpel processes”. In: *Fifth European Conference on Web Services (ECOWS’07)*. IEEE. 2007, pp. 171–180.

[ES21] S. Efftinge and M. Spoenemann. *The grammar language*. 2021.

[FG19] A. Falcone and A. Garro. “Distributed Co-Simulation of Complex Engineered Systems by Combining the High Level Architecture and Functional Mock-up Interface”. In: *Simulation Modelling Practice and Theory* 97 (2019), p. 101967.

[Flo67] R. W. Floyd. “Nondeterministic Algorithms”. In: *J. ACM* 14.4 (1967), pp. 636–644.

[FMS14] S. Friedenthal, A. Moore, and R. Steiner. *A practical guide to SysML: the systems modeling language*. Morgan Kaufmann, 2014.

[Fok+09] M. Fokaefs, N. Tsantalis, A. Chatzigeorgiou, and J. Sander. “Decomposing object-oriented class modules using an agglomerative clustering technique”. In: *IEEE International Conference on Software Maintenance, ICSM*. 2009, pp. 93–101.

[Fok+11] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou. “JDeodorant: Identification and application of extract class refactorings”. In: *Proceedings - International Conference on Software Engineering*. 2011, pp. 1037–1039.

[Fok+12] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou. “Identification and application of Extract Class refactorings in object-oriented systems”. In: *Journal of Systems and Software* 85.10 (2012), pp. 2241–2260.

[Fou23] E. Foundation. *EMF Refactor*. `https://www.eclipse.org/emf-refactor`. accessed 2023.

[Fow01] M. Fowler. “Reducing coupling”. In: *IEEE Software* 18.4 (2001), pp. 102–104.

[Fow18] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[Fow96] M. Fowler. *Analysis Patterns: Reusable Object Models*. Object Technology Series. Addison-Wesley, 1996.

[Fow99] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[FS15] S. Fu and B. Shen. “Code bad smell detection through evolutionary data mining”. In: *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE. 2015, pp. 1–9.

[FTC07] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou. “JDeodorant: Identification and removal of feature envy bad smells”. In: *IEEE International Conference on Software Maintenance, ICSM*. 2007, pp. 519–520.

[FZZ15] F. A. Fontana, M. Zanoni, and F. Zanoni. “A duplicated code refactoring advisor”. In: vol. 212. Springer Verlag, 2015, pp. 3–14.

[Gam+95] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[Gar+14] A. Garmendia, E. Guerra, D. Kolovos, and J. Lara. “EMF splitter: A structured approach to EMF modularity”. In: *3rd Workshop on Extreme Modeling* (2014), pp. 22–31.

[Gei+18] M. Geiger, S. Harrer, J. Lenhard, and G. Wirtz. “BPMN 2.0: The state of support and implementation”. In: *Future Generation Computer Systems* 80 (2018), pp. 250–262.

[Gom+18] C. Gomes et al. “Co-Simulation: A Survey”. In: *ACM Computing Surveys* 51.3 (2018), pp. 1–33.

[GSS13] S. Ganesh, T. Sharma, and G. Suryanarayana. “Towards a Principle-based Classification of Structural Design Smells.” In: *J. Object Technol.* 12.2 (2013), pp. 1–1.

[GT99] D. Gitchell and N. Tran. “Sim: A utility for detecting similarity in computer programs”. In: *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education)* 31.1 (1999), pp. 266–270.

[GTS23] GTS-Morpher. *Timed PLS (Gemoc)*. `https://github.com/gts-morpher/timed_pls_gemoc`. accessed 2023.

[Hah17] R. Hahn. “Bad Smells and Anti-Patterns in Metamodeling”. Master’s Thesis. Karlsruhe Institute of Technology, 2017.

[HBK18] R. Heinrich, K. Busch, and S. Koch. “A Methodology for Domain-spanning Change Impact Analysis”. In: *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2018, pp. 326–330.

[Hei+18] R. Heinrich, S. Koch, S. Cha, K. Busch, R. Reussner, and B. Vogel-Heuser. “Architecture-based change impact analysis in cross-disciplinary automated production systems”. In: *Journal of Systems and Software* 146 (2018), pp. 167–185.

[Hei+21a] R. Heinrich, E. Bousse, S. Koch, A. Rensink, E. Riccobene, D. Ratiu, and M. Sirjani. “Integration and Orchestration of Analysis Tools”. In: *Composing Model-Based Analysis Tools*. Springer International Publishing, 2021, pp. 71–95.

[Hei+21b] R. Heinrich, J. Henss, S. Koch, and R. Reussner. “Challenges in the Evolution of Palladio—Refactoring Design Smells in a Historically-Grown Approach to Software Architecture Analysis”. In: *Composing Model-Based Analysis Tools*. Springer International Publishing, 2021, pp. 235–257.

[Hey+20] T. Hey, J. Keim, A. Koziolek, and W. F. Tichy. “NoRBERT: Transfer Learning for Requirements Classification”. In: *2020 IEEE 28th International Requirements Engineering Conference (RE)*. 2020, pp. 169–179.

[Hig+04] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. “Refactoring support based on code clone analysis”. In: *International Conference on Product Focused Software Process Improvement*. Springer. 2004, pp. 220–233.

[HKR21] K. Hölldobler, O. Kautz, and B. Rumpe. *MontiCore Language Workbench and Library Handbook: Edition 2021*. Aachener Informatik-Berichte, Software Engineering, Band 48. Shaker Verlag, 2021.

[HR04] D. Harel and B. Rumpe. “Meaningful Modeling: What’s the Semantics of ”Semantics”?” In: *IEEE Computer* 37.10 (2004), pp. 64–72.

[HRW18] K. Hölldobler, B. Rumpe, and A. Wortmann. “Software language engineering in the large: towards composing and deriving languages”. In: *Computer Languages, Systems & Structures* 54 (2018), pp. 386–405.

[HSR19] R. Heinrich, M. Strittmatter, and R. H. Reussner. “A Layered Reference Architecture for Metamodels to Tailor Quality Modeling and Analysis”. In: *IEEE Transactions on Software Engineering* (2019).

[IEE10] IEEE. *1516-2010 - IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA).* Tech. rep. 2010, pp. 1–38.

[IEE95] IEEE 1278.2-1995. *Standard for Distributed Interactive Simulation - Communication Services and Profiles.* Tech. rep. IEEE, 1995.

[ISO10] ISO/IEC. *ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models.* Tech. rep. ISO/IEC, 2010.

[Jou+06] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez. “ATL: a QVT-like transformation language”. In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications.* ACM. 2006, pp. 719–720.

[Jun16] R. Jung. “Generator-Composition for Aspect-Oriented Domain-Specific Languages”. Doctoral thesis. Kiel University, 2016.

[KCO15] T. Kühn, W. Cazzola, and D. M. Olivares. “Choosy and picky: configuration of language product lines”. In: *19th International Conference on Software Product Line.* ACM. 2015, pp. 71–80.

[Kei+20] J. Keim, A. Kaplan, A. Koziolek, and M. Mirakhorli. “Does BERT Understand Code? – An Exploratory Study on the Detection of Architectural Tactics in Code”. In: *Software Architecture.* Springer International Publishing, 2020, pp. 220–228.

[KFS09] M. Klusch, B. Fries, and K. Sycara. “OWLS-MX: A hybrid Semantic Web service matchmaker for OWL-S services”. In: *Journal of Web Semantics* 7.2 (2009), pp. 121–133.

[KHR22a] S. Koch, R. Heinrich, and R. Reussner. *Supplementary Material for the Evaluation of the Publication – A Layered Reference Architecture for Model-based Quality Analysis.* Tech. rep. Karlsruher Institut für Technologie (KIT), 2022. 74 pp.

[KHR22b] S. Koch, R. Heinrich, and R. Reussner. *Supplementary Material to "A Layered Reference Architecture for Model-based Quality Analysis".* 2022.

[Koc+22] S. Koch, E. Hamann, R. Heinrich, and R. Reussner. “Feature-based Investigation of Simulation Structure and Behaviour”. In: *European Conference on Software Architecture.* Springer. 2022, p. 8.

[Kov+22] A. Kovačević, J. Slivka, D. Vidaković, K.-G. Grujić, N. Luburić, S. Prokić, and G. Sladić. “Automatic detection of Long Method and God Class code smells through neural source code embeddings”. In: *Expert Systems with Applications* 204 (2022), p. 117607.

[Koz08] H. Koziolek. “Goal, Question, Metric”. In: *Dependability Metrics: Advanced Lectures.* Springer Berlin Heidelberg, 2008, pp. 39–42.

[KPP08] D. S. Kolovos, R. F. Paige, and F. A. Polack. “The epsilon transformation language”. In: *International Conference on Theory and Practice of Model Transformations*. Springer. 2008, pp. 46–60.

[KR19] S. Koch and F. Reiche. “Towards a Correspondence Model for the Reuse of Software in Multiple Domains”. In: *10. Workshop „Design For Future – Langlebige Softwaresysteme“*. 2019 (May 6–8, 2019). Softwaretechnik-Trends. Softwaretechnik-Trends, 2019, pp. 41–42.

[KR22] S. Koch and F. Reiche. “A Toolchain for Simulation Component Specification and Identification”. In: *European Conference on Software Architecture*. accepted, to appear. Springer. 2022, p. 16.

[KT06] J. Kleinberg and E. Tardos. *Algorithm design*. Pearson Education India, 2006.

[KWa] S. Koch and M. Wittlinger. *Maven Central – Refactorlizar*. URL: `https://search.maven.org/search?q=g:org.mosim.refactorlizar` (visited on 07/12/2022).

[KWb] S. Koch and M. Wittlinger. *MoSimEngine/RefactorLizar*. URL: `https://github.com/MoSimEngine/RefactorLizar` (visited on 07/12/2022).

[KWc] S. Koch and M. Wittlinger. *MoSimEngine/RefactorLizarCLI*. URL: `https://github.com/MoSimEngine/RefactorLizarCLI` (visited on 07/12/2022).

[Lac+20] G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Guéhéneuc. “Code smells and refactoring: A tertiary systematic review of challenges and observations”. In: *Journal of Systems and Software* 167 (2020), p. 110610.

[Lai+22] R. Laigner, D. Mendonça, A. Garcia, and M. Kalinowski. “Cataloging dependency injection anti-patterns in software systems”. In: *Journal of Systems and Software* 184 (2022), p. 111125.

[Lam78] L. Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. In: *Communications of the ACM* 21.7 (1978), pp. 558–565.

[Lan+09] A. Lanusse, Y. Tanguy, H. Espinoza, C. Mraidha, S. Gerard, P. Tessier, R. Schnekenburger, H. Dubois, and F. Terrier. “Papyrus UML: an open source toolset for MDA”. In: *Proc. of the Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009)*. Citeseer. 2009, pp. 1–4.

[Lar12] C. Larman. *Applying UML and patterns: an introduction to object oriented analysis and design and interative development*. Pearson Education India, 2012.

[Law15] A. M. Law. *Simulation Modeling & Analysis*. 5th ed. McGraw-Hill, 2015.

[Lee+00] B. S. Lee, W. Cai, S. J. Turner, and L. Chen. “Adaptive dead reckoning algorithms for Distributed Interactive Simulation”. In: *International Journal of Simulation: Systems, Science and Technology* 1.1-2 (2000), pp. 21–34.

[Leh80] M. Lehman. “Programs, life cycles, and laws of software evolution”. In: *Proceedings of the IEEE* 68.9 (1980), pp. 1060–1076.

[Lil19] C. Lilienthal. *Sustainable Software Architecture: Analyze and Reduce Technical Debt*. dpunkt. verlag, 2019.

[Liu+16] H. Liu, Y. Wu, W. Liu, Q. Liu, and C. Li. "Domino effect: Move more methods once a method is moved". In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. IEEE. 2016, pp. 1–12.

[LL16] W. Liu and H. Liu. "Major motivations for extract method refactorings: analysis based on interviews and change histories". In: *Frontiers of Computer Science* 10.4 (2016), pp. 644–656.

[LP09] M. T. Llano and R. Pooley. "UML specification and correction of object-oriented anti-patterns". In: *2009 Fourth International Conference on Software Engineering Advances*. IEEE. 2009, pp. 39–44.

[Mao+17] S. Maoz, F. Mehlan, J. O. Ringert, B. Rumpe, and M. von Wenckstern. "OCL Framework to Verify Extra-Functional Properties in Component and Connector Models". In: *Workshop on Model-Driven Engineering for Component-Based Software Systems*. Vol. 2019. CEUR workshop proceedings. 3rd International Workshop on Executable Modeling, Austin (USA). RWTH Aachen, 18, 2017.

[Mar+03] R. Martin, J. Rabaey, A. Chandrakasan, J. Newkirk, B. Nikolić, and R. Koss. *Agile Software Development: Principles, Patterns, and Practices*. Alan Apt series. Pearson Education, 2003.

[Mar03] R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2003.

[MB09] L. de Moura and N. Bjørner. "Satisfiability Modulo Theories: An Appetizer". In: *Formal Methods: Foundations and Applications*. Springer Berlin Heidelberg, 2009, p. 24.

[MC16] R. Malhotra and A. Chug. "An empirical study to assess the effects of refactoring on software maintainability". In: *2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. IEEE. 2016, pp. 110–117.

[McK+81] B. D. McKay et al. *Practical graph isomorphism*. 1981.

[Mén+16a] D. Méndez-Acuña, J. A. Galindo, B. Combemale, A. Blouin, and B. Baudry. "Puzzle: A Tool for Analyzing and Extracting Specification Clones in DSLs". In: *Software Reuse: Bridging with Social-Awareness*. Springer, 2016, pp. 393–396.

[Mén+16b] D. Méndez-Acuña, J. A. Galindo, T. Degueule, B. Combemale, and B. Baudry. "Leveraging software product lines engineering in the development of external dsls: A systematic literature review". In: *Computer Languages, Systems & Structures* 46 (2016), pp. 206–235.

[Mer13] M. Mernik. "An Object-oriented Approach to Language Compositions for Software Language Engineering". In: *Journal of Systems and Software* 86 (2013), pp. 2451–2464.

[Mil89] R. Milner. *Communication and concurrency.* PHI Series in computer science. Prentice Hall, 1989.

[MJ19] R. Malhotra and J. Jain. "Analysis of refactoring effect on software quality of object-oriented systems". In: *International Conference on Innovative Computing and Communications.* Springer. 2019, pp. 197–212.

[ML07] B. Möller and B. Löfstrand. "Use cases for the HLA Evolved modular FOMs". In: 2007.

[MM79] K. Mani Chandy and J. Misra. "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs". In: *IEEE Transactions on Software Engineering* SE-5.5 (1979), pp. 440–452.

[MP14] B. D. McKay and A. Piperno. "Practical graph isomorphism, II". In: *Journal of Symbolic Computation* 60 (2014), pp. 94–112.

[Muz+10] A. Muzy, L. Touraille, H. Vangheluwe, O. Michel, M. K. Traoré, and D. R. Hill. "Activity regions for the specification of discrete event systems". In: *Spring Simulation Multiconference 2010, SpringSim'10.* ACM Press, 2010, p. 1.

[NKB00] R. V. van Nieuwpoort, T. Kielmann, and H. E. Bal. "Satin: Efficient Parallel Divide-and-Conquer in Java". In: *Euro-Par 2000 Parallel Processing.* Springer Berlin Heidelberg, 2000, pp. 690–699.

[NLD11] C. J. Neill, P. A. Laplante, and J. F. DeFranco. *Antipatterns: managing software organizations and people.* CRC Press, 2011.

[ON85] C. Overstreet and R. Nance. "A Specification Language to Assist in Analysis of Discrete Event Simulation Models." In: *Commun. ACM* 28 (1985), pp. 190–201.

[Paa16] T. Paananen. "Analyzing Java EE application security with SonarQube". Master's Thesis. JAMK University of Applied Sciences, 2016.

[Par79] D. L. Parnas. "Designing software for ease of extension and contraction". In: *IEEE transactions on software engineering* 2 (1979), pp. 128–138.

[Paw+15] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier. "Spoon: A Library for Implementing Analyses and Transformations of Java Source Code". In: *Software: Practice and Experience* 46 (2015), pp. 1155–1179.

[PMP02] L. Prechelt, G. Malpohl, and M. Philippsen. "Finding plagiarisms among a set of programs with JPlag". In: *Journal of Universal Computer Science* 8.11 (2002), pp. 1016–1038.

[Pow20] D. M. W. Powers. "Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation". In: *Computing Research Repository (CoRR)* abs/2010.16061 (2020). arXiv: `2010.16061`.

[Pto14] C. Ptolemaeus. *System Design, Modeling, and Simulation. Using Ptolemy II.* Ptolemy.org, 2014, p. 690.

[Rah+18] M. M. Rahman, R. R. Riyadh, S. M. Khaled, A. Satter, and M. R. Rahman. "MMRUC3: A recommendation approach of move method refactoring using coupling, cohesion, and contextual similarity to enhance software design". In: *Software: Practice and Experience* 48.9 (2018), pp. 1560–1587.

[Ras+15] W. Raskob, V. Bertsch, M. Ruppert, M. Strittmatter, L. Happe, B. Broadnax, S. Wandler, and E. Deines. "Security of electricity supply in 2030". In: *Critical infrastructure protection and resilience Europe (CIPRE) conference & expo, The Hague, The Netherlands.* 2015.

[RBF04] R. H. Reussner, S. Becker, and V. Firus. "Component composition with parametric contracts". In: *Tagungsband der Net. ObjectDays* 2004 (2004), pp. 155–169.

[Reu+16] R. H. Reussner, S. Becker, J. Happe, R. Heinrich, A. Koziolek, H. Koziolek, M. Kramer, and K. Krogmann. *Modeling and Simulating Software Architectures – The Palladio Approach.* MIT Press, 2016. 408 pp.

[RF20] M. Richards and N. Ford. *Fundamentals of Software Architecture: An Engineering Approach.* O'Reilly Media, 2020.

[RG00] M. Richters and M. Gogolla. "Validating UML Models and OCL Constraints". In: *«UML» 2000 - The Unified Modeling Language, Advancing the Standard, Third International Conference.* Vol. 1939. LNCS. Springer, 2000, pp. 265–277.

[Ric15] M. Richards. *Software architecture patterns.* Vol. 4. O'Reilly Media, Incorporated 1005 Gravenstein Highway North, Sebastopol, CA, 2015.

[Ros+15] K. Rostami, J. Stammel, R. Heinrich, and R. Reussner. "Architecture-based Assessment and Planning of Change Requests". In: *11th International Conference on Quality of Software Architectures.* ACM, 2015, pp. 21–30.

[Ros+17] K. Rostami, R. Heinrich, A. Busch, and R. Reussner. "Architecture-based Change Impact Analysis in Information Systems and Business Processes". In: *2017 IEEE International Conference on Software Architecture (ICSA2017).* IEEE, 2017, pp. 179–188.

[RR11] J. O. Ringert and B. Rumpe. "A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing". In: *International Journal of Software and Informatics* 5 (2011), pp. 29–53.

[Rum17] B. Rumpe. *Agile Modeling with UML: Code Generation, Testing, Refactoring.* Springer, 2017.

[Run+12] P. Runeson, M. Höst, A. Rainer, and B. Regnell. *Case Study Research in Software Engineering: Guidelines and Examples.* 2012.

[Sch77] D. Schütt. "On a Hypergraph Oriented Measure For Applied Computer Science". In: *COMPCON Fall '77.* 1977, pp. 295–296.

[Sei+22] S. Seifermann, R. Heinrich, D. Werle, and R. Reussner. "Detecting Violations of Access Control and Information Flow Policies in Data Flow Diagrams". In: *The journal of systems and software* 184 (2022). 46.23.03; LK 01, Art.–Nr. 111138.

[SHR18] M. Strittmatter, R. Heinrich, and R. Reussner. *Supplementary Material for the Evaluation of the Layered Reference Architecture for Metamodels to Tailor Quality Modeling and Analysis*. Tech. rep. 11. Karlsruhe Institute of Technology, 2018. 42 pp.

[SMB17] R. Sehgal, D. Mehrotra, and M. Bala. “Analysis of code smell to quantify the refactoring”. In: *International Journal of System Assurance Engineering and Management* 8.2 (2017), pp. 1750–1761.

[Som18] I. Sommerville. *Software Engineering*. 10th. Addison-Wesley, 2018.

[Son23] SonarSource. *SonarQube*. `https://www.sonarqube.org/`. accessed 2023.

[SSS14] G. Suryanarayana, G. Samarthyam, and T. Sharma. *Refactoring for software design smells: managing technical debt*. Morgan Kaufmann, 2014.

[Sta73] H. Stachowiak. *Allgemeine Modelltheorie*. 1973.

[Ste+09] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. 2nd ed. Eclipse Series. Addison-Wesley, 2009.

[Str+16] M. Strittmatter, G. Hinkel, M. Langhammer, R. Jung, and R. Heinrich. “Challenges in the Evolution of Metamodels: Smells and Anti-Patterns of a Historically-Grown Metamodel”. In: *10th International Workshop on Models and Evolution (ME)*. CEUR Vol-1706, 2016.

[Str20] M. Strittmatter. “A Reference Structure for Modular Metamodels of Quality-Describing Domain-Specific Modeling Languages”. PhD thesis. Karlsruhe Institute of Technology (KIT), 2020.

[SVC06] T. Stahl, M. Völter, and K. Czarnecki. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, Inc., 2006.

[SWA03] S. Schleimer, D. S. Wilkerson, and A. Aiken. “Winnowing: Local Algorithms for Document Fingerprinting”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2003, pp. 76–85.

[Syr+13] E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, S. Van Mierlo, and H. Ergin. “AToMPM: A web-based modeling environment”. In: *Joint proceedings of MODELS’13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013): September 29-October 4, 2013, Miami, USA*. 2013, pp. 21–25.

[Tae+05] G. Taentzer, K. Ehrig, E. Guerra, J. d. Lara, L. Lengyel, T. Levendovszky, U. Prange, D. Varro, and S. Varro-Gyapay. “Model transformation by graph transformation: A comparative study”. In: *Model Transformation in Practice* (2005).

[Tal+21a] C. Talcott, S. Ananieva, K. Bae, B. Combemale, R. Heinrich, M. Hills, N. Khakpour, R. Reussner, B. Rumpe, P. Scandurra, and H. Vangheluwe. “Composition of Languages, Models, and Analyses”. In: *Composing Model-Based Analysis Tools*. Springer International Publishing, 2021, pp. 45–70.

[Tal+21b] C. Talcott, S. Ananieva, K. Bae, B. Combemale, R. Heinrich, M. Hills, N. Khakpour, R. Reussner, B. Rumpe, P. Scandurra, H. Vangheluwe, F. Durán, and S. Zschaler. “Foundations”. In: *Composing Model-Based Analysis Tools.* Springer International Publishing, 2021, pp. 9–37.

[Tav+16] J. P. Tavella, M. Caujolle, S. Vialle, C. Dad, C. Tan, G. Plessis, M. Schumann, A. Cuccuru, and S. Revol. “Toward an accurate and fast hybrid multi-simulation with the FMI-CS standard”. In: *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA.* Vol. 2016-Novem. Institute of Electrical and Electronics Engineers Inc., 2016.

[TC10] N. Tsantalis and A. Chatzigeorgiou. “Identification of refactoring opportunities introducing polymorphism”. In: *Journal of Systems and Software* 83.3 (2010), pp. 391–404.

[TC11] N. Tsantalis and A. Chatzigeorgiou. “Identification of extract method refactoring opportunities for the decomposition of methods”. In: *Journal of Systems and Software* 84.10 (2011), pp. 1757–1782.

[TCC08] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou. “JDeodorant: Identification and removal of type-checking bad smells”. In: *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR.* 2008, pp. 329–331.

[TMK15] N. Tsantalis, D. Mazinanian, and G. P. Krishnan. “Assessing the Refactorability of Software Clones”. In: *IEEE Transactions on Software Engineering* 41.11 (2015), pp. 1055–1090.

[TMR17] N. Tsantalis, D. Mazinanian, and S. Rostami. “Clone Refactoring with Lambda Expressions”. In: *IEEE/ACM 39th International Conference on Software Engineering.* 2017, pp. 60–70.

[Tom13] P. Tomassi. “An Introduction to First Order Predicate Logic”. In: *Logic.* Routledge, 2013, pp. 189–264.

[Top+16] O. Topçu, L. Yilmaz, H. Oguztüzün, and U. Durak. “Distributed simulation”. In: *A Model Driven Engineering Approach. Springer* (2016), p. 9.

[TR03] J.-P. Tolvanen and M. Rossi. “Metaedit+ defining and using domain-specific modeling languages and code generators”. In: *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.* 2003, pp. 92–93.

[TS08] Y. M. Teo and C. Szabo. “CoDES: An integrated approach to composable modeling and simulation”. In: *41st Annual Simulation Symposium.* IEEE, 2008, pp. 103–110.

[Tsa+13] N. Tsantalis, V. Guana, E. Stroulia, and A. Hindle. “A Multidimensional Empirical Study on Refactoring Activity”. In: *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research.* CASCON ’13. IBM Corp., 2013, pp. 132–146.

[Ull76] J. R. Ullmann. *An Algorithm for Subgraph Isomorphism.* Tech. rep. 1. 1976, pp. 31–42.

[Völ11] M. Völter. “Language and IDE Modularization, Extension and Composition with MPS”. In: *Pre-proceedings of Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE)* LNCS Vol. 7680 (2011), pp. 383–430.

[VV14] Y. Van Tendeloo and H. Vangheluwe. “Activity in PythonPDEVS”. In: *ITM Web of Conferences* 3 (2014), p. 01002.

[VV15] Y. Van Tendeloo and H. Vangheluwe. “PythonPDEVS: A distributed parallel DEVS simulator”. In: *Simulation Series.* Vol. 47. 8. 2015, pp. 91–98.

[Wen12] C. Wende. “Language Family Engineering - with Features and Role-Based Composition”. PhD thesis. Technische Universität Dresden, 2012.

[WHR22] M. Walter, R. Heinrich, and R. Reussner. “Architectural Attack Propagation Analysis for Identifying Confidentiality Issues”. In: *2022 IEEE 19th International Conference on Software Architecture (ICSA).* 2022, pp. 1–12.

[Woh21] C. Wohlin. “Case Study Research in Software Engineering—It is a Case, and it is a Study, but is it a Case Study?” In: *Information and Software Technology* 133 (2021), p. 106514.

[Z3P19] Z3Prover. *z3: The Z3 Theorem Prover*. 2019.

[Zei76] B. P. Zeigler. *Theory of modelling and simulation.* 1976.

[ZMK18] B. P. Zeigler, A. Muzy, and E. Kofman. *Theory of Modeling and Simulation: Discrete Event and Iterative System Computational Foundations.* 3rd. Academic Press, Inc., 2018.

[ZPK00] B. P. Zeigler, H. Prähofer, and T. G. Kim. *Theory of modeling and simulation : integrating discrete event and continuous complex dynamic systems.* 2. ed. Academic Press, 2000.