Multiperspective Change Impact Analysis to Support Software Maintenance and Reengineering

Dissertation with the aim of achieving the doctoral degree of

Doktor der Naturwissenschaften (Dr. rer. nat.)

at the Faculty of Mathematics, Informatics and Natural Sciences
Department of Software Engineering and Construction Methods
at the University of Hamburg

submitted by

Dipl.-Inf. Steffen Lehnert

Hamburg, 2015

Day of oral defense: 07.07.2015

The following evaluators recommend the admission of the dissertation:

1st Examiner: Prof. Dr.-Ing. habil. Matthias Riebisch, University of Hamburg
2nd Examiner: Prof. Dr. rer. nat. Horst Lichter, RWTH Aachen University
3rd Examiner: Prof. Dr. rer. nat. Andreas Winter, University of Oldenburg

Abstract

The lifecycle of software is characterized by frequent and inevitable changes to address varying requirements and constraints, remaining defects, and various quality goals, such as portability or efficiency. Adapting long-living systems, however, poses the risk of introducing unintended side effects, such as new program errors, that are typically caused by dependencies between software artifacts of which developers are not aware of while changing the software. Consequently, research has focused on developing approaches for software change impact analysis to aid developers with assessing the consequences of changes prior to their implementation. Yet, current change impact analysis approaches are still code-based and only able to assess the consequences of changing source code artifacts. Software development on the other hand is accompanied by various different types of software artifacts, such as software architectures, test cases, source code files, etc. which in turn demand for comprehensive change impact analysis.

This thesis presents a novel approach for change impact analysis that is able to address heterogeneous software artifacts stemming from different development stages, such as architectural models and source code. It allows to forecast the impacts of changes prior to their implementation and is able to address a multitude of different change operations. The approach is based on a novel concept for computing the propagation of changes, for which the interplay of the types of changes, types of software artifacts, and the types of dependencies between them is analyzed by a set of predefined impact propagation rules. To accomplish this, the heterogeneous software artifacts are first mapped on a common meta-model to allow for a multiperspective analysis across the different views on software. Secondly, their dependencies are extracted and explicitly recorded as traceability links, while the type of change to be implemented is specified and modeled using a taxonomy and meta-model for change operations supplied by this thesis. A set of impact propagation rules is then recursively executed to compute the overall impact of the change, where each rule addresses a specific change operation and determines the resulting impact. Meanwhile, the impact computed by each rule is fed back into the analysis process where it may trigger the execution of further rules. Once the overall impact of a change has been determined, the impacted software artifacts are presented to the developer who is then able to understand the overall implications of the change and to implement it more efficiently.

The presented approach is implemented by the prototype tool EMFTrace and currently enables change impact analysis of UML models, Java source code, and JUnit test cases to support developers with their everyday modifications of software systems. The approach was deployed during a comprehensive case study to evaluate its efficiency and correctness when determining the impacts of changes. The approach helped to maintain the consistency of the architecture and the source code of the test system by reliably forecasting the impact propagation between both. The study furthermore confirmed that the approach achieves both high precision (90%) and recall (93%) when determining impacted software artifacts. Consequently, it computed only few false-positives and did not overestimate the impact propagation, which in turn enables developers to understand the effects of changes more easily and with less effort.

Kurzfassung

Der Lebenszyklus von Software wird von fortlaufenden Änderungen bestimmt, durch welche die Systeme an sich an stetig ändernde Anforderungen und Rahmenbedingungen angepasst, bestehende Defekte behoben sowie Qualitätsziele wie Portabilität und Effizienz adressiert werden. Änderungen bestehender, langlebiger Systeme bergen jedoch immer das Risiko ungewollter Seiteneffekte in sich, die durch Abhängigkeiten zwischen den einzelnen Bestandteilen der Software verursacht und häufig von Entwicklern übersehen werden. So können unbewusst durch Änderungen etwa zusätzliche Programmfehler oder Sicherheitslücken entstehen. Infolgedessen wurden Methoden entwickelt, mit deren Hilfe die Auswirkungen von Änderungen noch vor deren Umsetzung abgeschätzt werden können. Jedoch sind diese Ansätze hauptsächlich auf Quellcode beschränkt und vernachlässigen andere Arten von ebenso relevanten Beschreibungsmitteln, wie etwa Architekturmodelle oder Testfälle.

Diese Arbeit präsentiert einen neuen Ansatz zur Abschätzung der Auswirkungen von Änderungen, der in der Lage ist, unterschiedliche Softwareartefakte verschiedener Entwicklungsphasen, wie etwa Architekturmodelle und Quellcode, zu analysieren. Er ermöglicht es, die Konsequenzen von einer Vielzahl unterschiedlicher Änderungen noch vor deren Umsetzung zu erfassen. Der Ansatz basiert auf einem neuen Konzept zur Vorherbestimmung der Ausbreitung von Änderungen, wofür das Zusammenspiel von Änderungsoperationen, Softwareartefakten und deren Abhängigkeiten von Impact-Regeln analysiert wird. Dafür werden die heterogenen Softwareartefakte zunächst auf ein vereinheitlichendes Metamodell abgebildet, um einen sichtenübergreifenden Zugriff auf diese zu ermöglichen. Anschließend werden deren Abhängigkeiten extrahiert und explizit als Traceability-Links gespeichert, während die Änderungen anhand ihres Typs klassifiziert und ebenfalls explizit modelliert werden, wofür jeweils eine entsprechende Taxonomie sowie ein Metamodell bereitgestellt werden. Durch die anschließend rekursive Ausführung der Impact-Regeln werden die Auswirkungen der Änderung bestimmt, wobei jede Regel genau eine Änderung adressiert. Die von einer Regel berechneten Auswirkungen werden zurück in den Analyseprozess eingespeist, wo sie ggf. die Ausführung weiterer Regeln bedingen. Sobald die rekursive Regelverarbeitung abgeschlossen ist, werden dem Entwickler alle die von der geplanten Änderung betroffenen Softwareartefakte aufgezeigt.

Der entwickelte Ansatz wird prototypisch durch das Werkzeug EMFTrace implementiert und ermöglicht die Analyse von UML-Modellen, Java-Quellcode und JUnit-Testfällen, um Entwickler bei der Modifikation bestehender Systeme zu unterstützen. Der Ansatz wurde in einer umfangreichen Fallstudie hinsichtlich seiner Effektivität und Korrektheit bei der Vorherbestimmung der Auswirkungen von Änderungen evaluiert. Mit Hilfe des entwickelten Ansatzes konnte die Konsistenz zwischen der Architektur und dem Quellcode des Testsystems während der Umsetzung verschiedener Modifikationen aufrechterhalten werden. Gleichzeitig konnte gezeigt werden, dass der Ansatz in der Lage ist, die Auswirkungen von Änderungen sowohl möglichst korrekt (90%), als auch möglichst vollständig (93%) zu bestimmen, wodurch Entwickler effektiv bei der Implementierung von Änderungen unterstützt werden.

Acknowledgment

First and foremost I would like to thank my supervisor Prof. Matthias Riebisch for his guidance, helpful advices, and support during my PhD studies. I would like to thank him for always asking the right questions and for providing me with the opportunity to be a part of his research group. Furthermore, I would like to thank Prof. Andreas Winter and Prof. Horst Lichter for taking their time to review my thesis and for their many useful hints and advices. Likewise, I am grateful for all the support I received from Prof. Ilka Philippow and from Prof. Vesselin Detschew.

Moreover, I had the pleasure to work together with many great colleagues who accompanied me during the pursuit of my PhD and who also deserve to be mentioned here. Stephan Bode for his assistance during the early stages of a PhD and for the many useful advices what to do and what not to do. Annie for being the best pal in the office, for our endless discussions, and all the nice food you've brought along. Danny and Sebastian for all the fun we had with our "Ajax voodoo" during seemingly endless debug-sessions. Yibo and Sebastian for their support with EMFTrace. And last but not least Elke...I will miss our talks during lunch.

Finally, I would like to express my deepest gratitude to my family and friends who supported me during the last couple of years.

Contents

1. Introduction

This chapter provides an overview of the context of this thesis, elaborates on its research goals, and explains what this thesis contributes to research on software change impact analysis and software evolution. Moreover, the structure of the thesis at hand is outlined to guide the reader.

1.1. Motivation

The majority of today's software is in a state of constant evolution and development, which is triggered by frequent requests for changes [Leh80]. This constant need for changes is inherent to all stages of the software lifecycle and has various reasons, such as:

- Changing customer needs demanding for the addition of new features.
- Fixing remaining bugs and security issues.
- The advent of new hardware devices demanding for software support, e.g. adding support for touch-based user interaction via finger gestures.
- Changing legal circumstances and regulations that must be adhered by software systems.
- Performance improvements demanding for optimizations of the software.
- Integrating new middleware or other COTS[1] components.

Consequently, frequent changes are vital for software systems that are deployed in an ever-changing context. Due to the importance of changes even after the initial development of software, software maintenance[2] has become increasingly relevant for developers, researchers, and entire companies. Many studies even indicate that companies nowadays spent most of their costs for software, between 50% and up to 70%, on the maintenance of their software systems [Leh80, Ben90, CDPC11]. Likewise, software reengineering[3] as a prelude to many software engineering activities plays a similar important role for developers dealing with legacy software and also results in frequent and comprehensive changes of legacy systems [DDN08].

However, apart from adding new features, fixing bugs, etc. the changes themselves can also have adverse effects on software systems. Changes are often accompanied by unintended side effects that are byproducts of changing complex and long-living software systems [YCM78]. These side effects are potentially able to deteriorate the quality of software in manifold ways, which, over the course of time, typically results in the following phenomena and defects:

[1]"Commercial-Off-The-Shelf", see [Boh02a]

[2]"modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment", see [Ins98b]

[3]"Reengineering ... is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form", see [CC92]

- Changes increase the drift between the originally planned architecture and the actual implementation in source code [IK06].
- Changes introduce new bugs [HH04] or impose new security issues [YC04].
- Changes result in a loss of internal structure and architectural erosion [RSN09].
- Frequent changes decrease the maintainability[4] of a system [CKKL99].

These side effects typically result from changes that were implemented in an inconsistent or incorrect manner, which is mostly caused by overlooked or misinterpreted dependencies that exist between the involved software artifacts [Raj97]. These dependency relations carry the effects of changes to other software artifacts of which software developers are often not aware of due to the complexity of today's software [GS82,Han96,Boh02a]. Fittingly, Yau *et al.* coined the term "ripple effect" [YCM78] to describe this phenomena of "hidden" change propagation which is in the focus of research on software maintenance for almost four decades by now.

When software is changed, it is therefore necessary to assess all the consequences of those changes prior to their implementation, which is typically achieved by applying *change impact analysis* approaches. Early research conducted by Bohner and Arnold [AB93, Boh95, BA96, Boh96] investigated the foundations of software change impact analysis and provided the following definition of the term *impact analysis* that has been adapted by most researchers today:

> "Identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change [BA96]."

Later on, this definition was extended and adapted by the IEEE Standard for Software Maintenance as follows:

> "impact analysis: Identifies all system and software products that a change request affects and develops an estimate of the resources needed to accomplish the change. This includes determining the scope of the changes to plan and implement work, accurately estimating the resources needed to perform the work, and analyzing the requested change's cost and benefits [Int06]."

Years of ongoing research on impact analysis have put forth many different kinds of approaches which, however, share several disadvantages. The most striking and severe limitation of current research is the lack of impact analysis support for the interplay of heterogeneous software artifacts, as the majority of current research is still solely focused on source code [Leh11a]. Hence, these approaches fail to assess the consequences of changes that simultaneously affect different types of software artifacts at the same time, which constraints their applicability.

In contrast to what is addressed by current impact analysis approaches, software systems are comprised of heterogeneous types of software artifacts, such as source code files, architectural diagrams, requirements descriptions, documentation files, configuration files, etc. These different types of software artifacts are typically associated with certain views or perspectives of software, such as the architectural view or the code view, that represent a cut-out of the system in regard to a specific purpose and thus are interconnected with each other [OG02]. Moreover, the perspectives are associated with different stakeholders, such as programmers, architects, requirements engineers or testers who are responsible for maintaining the artifacts that typically constitute a certain perspective [SF01]. Therefore, the term *multiperspective software* was

[4]"the capability of the software product to be modified", see ISO 9126 [ISO01]

introduced by Sunetnanta and Finkelstein to describe this interplay of different stakeholders, artifacts, and perspectives [SF01].

Consequently, today's *multiperspective software* also requires *multiperspective change impact analysis* to cope with the frequent changes. Since changes spread across all perspectives and affect different types of software artifacts alike [FMP99], they have to be addressed adequately in each perspective to prevent potential inconsistencies that cause the various aforementioned problems and defects. Architectural changes, for example, must as well be reflected in the source code to prevent the drift between architecture and implementation. This support, however, is not yet sufficiently provided by current research [Leh11a, LFR13].

Additionally, most impact analysis approaches do not distinguish between different types of changes and the different effects caused by them [Leh11a, LFR13]. Consequently, they fail to precisely determine the effects of different types of change operations, which either results in too many false-positives being detected or large amounts of actual impacts being missed, or both at the same time. Thus, they are not sufficiently supporting developers changing software.

This thesis therefore investigates change impact analysis in the light of multiperspective software systems that are comprised of heterogeneous software artifacts. As its main contribution, it proposes an approach to assess the impacts of changes prior to their implementation while facilitating the analysis of different types of software artifacts belonging to different perspectives and views. The proposed methodology is able to adequately address different types of change operations and assists with maintaining the consistency of the heterogeneous artifacts and perspectives by enabling developers to understand and retrace the propagation of changes and their impacts across the entire software system.

1.2. Goals of the Thesis

The main goal of this thesis in regard to the previous problem statement is to study the propagation of changes between the different views on software and across heterogeneous software artifacts to assess their impacts on the whole software system. Therefore, a novel approach for analyzing the impacts of changes shall be developed that can be applied in the context of heterogeneous software artifacts, which in turn shall enable its applicability during different phases or stages of software development. The main goal of the approach to be developed is to identify all the software artifacts that are impacted by a change and to do so with high correctness, thus greatly reducing the amount of false-positives. Hence, the following two goals are defined.

Goal 1: Develop a change impact analysis approach that provides a high reliability when assessing the effects of changes prior to their implementation. The approach should identify all artifacts impacted by a change and determine as few false-positives as possible.

Goal 2: Develop a change impact analysis approach that enables the analysis of different types of software artifacts to keep them synchronized and consistent to each other. The approach to be developed shall at least support the following types of artifacts: architectural models (design view), source code (code view), and test cases (test view).

Furthermore, it is important to determine how a software artifact is impacted by a change as no appropriate counter measures can be taken and no precise effort estimation for the implementation of this change can be established otherwise. Hence, the following third goal is defined.

Goal 3: Develop a change impact analysis approach that allows developers to understand how and why software artifacts are impacted by a certain change. The approach should be applicable during forward engineering, reengineering, and software maintenance in general.

Finally, the existing literature on change impact analysis lacks a precise description of the term "change" and neglects discussions of the different impacts caused by different types of changes [Leh11a]. Analyzing changes for their impacts, however, first of all requires a thorough understanding of the actual changes. Therefore, the following fourth goal is defined.

Goal 4: Study the influence of different types of change operations on the propagation of changes and on change impact analysis to support a wide variety of change operations, including typical reengineering, refactoring, and maintenance activities.

1.3. Challenges

Developing an approach for multiperspective change impact analysis faces several challenges that are issued by the interplay of heterogeneous software artifacts and the usage of different perspectives, which are discussed in this section. Later on, the implications of these challenges for the proposed impact analysis approach and the means to address them will be discussed in the upcoming chapters of this thesis.

Degree of Formalization: The degree of formalization determines whether the structure and content of a software artifact adhere to any formal specification. Typical degrees of formalization as encountered in practice are, for example, unstructured free text, semi-structured text, and artifacts adhering a meta-model or any other formal specification, such as a programming language specification. The degree of formalization determines to which extent a software artifact can be analyzed by automated approaches that usually demand for a certain level of formalization. Thus, a low degree of formalization or the complete absence of any formalization might hamper any change impact analysis effort or even render it impossible.

Degree of Abstraction/Completeness: The degree of abstraction (or completeness) describes to which extent the information that could potentially be provided by a certain software artifact are actually present. It is important to note that software artifacts of the same type can have a different level of abstraction and completeness. For example, a UML component diagram providing a high-level overview of a software architecture possesses a higher level of abstraction than a fined-grained UML component diagram describing interfaces and sub-components of a certain architectural layer. Moreover, some artifacts may lack certain details because they were simply forgotten to add or were neglected due to time and budget constraints. Likewise, some information might be added in a later stage of development and are thus not yet available to developers. The level of abstraction and completeness influences to which extent the impact of a change can be determined for a given software artifact. In contrast to the degree of formalization, a strictly formalized but yet incomplete artifact still poses a challenge for any change impact analysis approach, as the required information for detecting potential impacts might not be present. Furthermore, due to an artifact's incompleteness, its relations to other software artifacts might not be detectable and thus analyzable for determining the impact propagation.

Inconsistencies: Finally, the consistency of the heterogeneous software artifacts and perspectives issues another challenge. Software artifacts of different perspectives typically posses a different level of abstraction and formalization; however, they still represent the same concepts, only with a different view. For example, there are UML diagrams to model the structure of software systems while other UML diagrams are designed for modeling the behavior of software systems. In practice, however, the ongoing evolution of most software systems often introduces inconsistencies to those artifacts, as not all of the artifacts are adequately changed and maintained over time [IK06, RSN09]. These inconsistencies in turn complicate the actual change impact analysis, as the dependencies between the heterogeneous software artifacts become "invisible" to developers and tools alike. Additionally, the usage of different vocabulary by different stakeholders (e.g. architects, programmers, testers) for expressing the same concerns and concepts might also lead to additional inconsistencies between the perspectives [TLCvV11b].

1.4. Contribution

As its major innovation this thesis presents a novel approach for multiperspective change impact analysis. The approach assists developers with maintaining the consistency of heterogeneous software artifacts when changes were applied on them. This change impact analysis approach is able to automatically determine the impacts of changes and can be customized for different types of software artifacts. The proposed approach currently supports impact assessments of UML models, Java source code, and JUnit test cases.

The presented approach is based on two fundamental concepts that are introduced in this thesis. The first of these is the observation that the impacts of changes can be determined by analyzing the interplay of change operations, software artifacts, and the dependency relations between them. Secondly, a recursive rule-based concept is presented to analyze and monitor this interplay of changes and dependencies in order to estimate the impacts of changes prior to their implementation through a set of impact propagation rules. To accomplish this, a step-wise approach for the definition of impact propagation rules is introduced.

Additionally, the approach provides two important benefits. First, the rule-based analysis of the interplay of change types and dependency types allows for determining *how* software artifacts are impacted by changes, which in turn enables more precise effort estimations and assists with actually implementing the changes. Secondly, the presented approach is not restricted to certain types of software artifacts. The incorporation of further artifacts into the impact analysis process only demands for the addition of new impact propagation rules, but does not require any changes in the underlying concepts and algorithms. Hence, it offers a greater flexibility.

The research presented in this thesis furthermore contributes towards a deeper understanding of change operations and dependency relations by providing taxonomies for the classification of both that are based upon comprehensive reviews and analyses of existing research. Likewise, a taxonomy for the classification and comparison of change impact analysis approaches is established that eases the comparison of the presented approach with existing works.

1.5. Thesis Outline

The remainder of the thesis at hand is organized as follows.

Chapter 2: Change Impact Analysis provides an overview of current state-of-the-art research on impact analysis. It presents and discusses the results of a systematic literature review and the taxonomy for impact analysis that was derived from it. Existing approaches are analyzed for their support of multiperspective impact analysis and yet unsolved problems are outlined.

Chapter 3: Thesis Foundations discusses related work on multiperspective modeling and consistency checking and its relation to impact analysis. Further works are being discussed dealing with the classification and modeling of dependencies between software artifacts, as well as with the classification and modeling of change operations as a prelude to impact analysis.

Chapter 4: Overview of the Approach provides a high-level overview of the approach presented in this thesis and outlines its four major steps: the integration of software artifacts, the detection and classification of their dependencies, the specification of the changes, and finally the impact analysis. Furthermore, the initial research goals are refined, the main research hypothesis is presented and discussed, and the assumptions of the approach are laid out.

Chapter 5: Comprehensive Artifact Integration examines approaches for the integration of heterogeneous software artifacts for multiperspective impact analysis. Based on that, the chapter presents an approach how heterogeneous types of software artifacts are joined through a unifying modeling framework and model repository, and illustrates the mapping of the heterogeneous artifacts upon the common meta-model provided by the modeling framework.

Chapter 6: Dependency Detection presents an approach how potential dependency relations between software artifacts can be detected and classified according to their type to utilize them for impact analysis tasks. The chapter also introduces a purpose-based taxonomy of dependency types to provide the required means for classifying the detected dependency relations.

Chapter 7: Change Comprehension introduces an approach how change activities can be modeled using the concepts of atomic and composite change operations, and demonstrates how different types of refactoring operations can be modeled with the help of these concepts in order to enable a later impact analysis. Therefore, this chapter also presents a taxonomy for change operations and illustrates how real changes can be classified according to its criteria.

Chapter 8: Rule-based Impact Analysis describes our novel approach for change impact analysis as the main innovation provided by this thesis. The chapter discusses the underlying impact propagation concept and explains how the change propagation is monitored. It elaborates on the concept of impact propagation rules and provides a scheme for defining such rules.

Chapter 9: The EMFTrace Prototype introduces the prototype tool EMFTrace that implements the concepts presented in this thesis, describes its architecture, illustrates its typical use cases, and outlines its current status and future development.

Chapter 10: Evaluation elaborates on the evaluation of the presented impact analysis approach with the help of a comprehensive case study. This chapter presents the design of the study, the research questions, the results, and discusses possible threats to validity.

Chapter 11: Conclusions and Future Work summarizes the content and contribution of this thesis, outlines future work, and performs a final critical review of this thesis' research.

Appendix A lists all the dependency detection rules that accompany this thesis.

Appendix B lists all the impact rules that accompany this thesis.

Appendix C provides all the material and data of the case study reported in Chapter 10.

2. Change Impact Analysis

This chapter explores the state of the art of current research on change impact analysis according to the goals of this thesis. We report on the findings of a comprehensive literature survey and analyze various techniques that have been proposed for change impact analysis in detail. In order to keep this thesis self-contained, this chapter also briefly recaps the systematic literature review and the taxonomy for change impact analysis approaches that was derived from it, although both were already reported in [Leh11b, Leh11a]. As a novel contribution, this chapter systematically outlines the strengths and weaknesses of the proposed techniques in regard to the interplay of heterogeneous software artifacts. Moreover, this chapter summarizes open research problems and prepares the ground for refining the research goals of this thesis.

2.1. A Comprehensive Literature Review

This section elaborates on the comprehensive literature review on change impact analysis that was conducted at the beginning of the research that eventually resulted in the thesis at hand. This section briefly describes the intentions and goals of the review, the structure of the review process, and the obtained results in relation to its initial research questions.

Overall, the review was conducted with the following initial intentions:

1. Provide an overview of the state-of-the-art of research on change impact analysis.
2. Identify open research questions and problems in regard to the goals of this thesis.
3. Refine the goals of the thesis at hand if necessary.

The review process adhered to the guidelines for conducting systematic literature reviews in software engineering as reported by Biolchi *et al.* [BMNT05]. The research questions to be answered were laid out in advance, while the data collection and analysis were planned by peer-reviewed protocols. The initial review was conducted between January and November 2011 and identified approximately 180 relevant studies that were examined in detail [Leh11a]. Further and later studies were subsequently examined afterwards based on the same criteria.

The results of the review were threefold and in line with its initial intentions. First, a taxonomy for change impact analysis approaches was derived and published that enables the comparison of the identified approaches [Leh11b]. Second, the studied approaches were classified according to the criteria of the taxonomy. The outcome of this classification was later on published as a technical report [Leh11a]. As one important outcome of this classification, the lack of impact analysis support for the interplay of heterogeneous software artifacts was revealed [Leh11a]. Hence, the motivation for the research presented in this thesis was strengthened. Finally, a set of open research questions was identified and the research goals of this thesis were refined [Leh11a].

2.1.1. Research Questions and Review Process

Our literature review was comprised of the five steps as outlined by Biolchi *et al.* [BMNT05].

Step 1: Defining the Research Questions. Each literature review starts from a set of research questions that were formulated as follows for the thesis at hand:
RQ1: What types of software artifacts are covered by change impact analysis approaches?
RQ2: What type of input and user interaction is required by these approaches?
RQ3: What types of change operations are supported by these approaches?
RQ4: What techniques or algorithms are used by these approaches?

Step 2: Study Selection. The literature review addressed studies that were published between 1991 and 2011 in the field of change impact analysis. We accepted studies that were published as PhD thesis, Master thesis, journal articles, conference papers, workshop papers, and technical reports in either English or German. However, studies describing manual impact analysis approaches were excluded, as they do not scale well with complex software systems [Boh02b].

Step 3: Search Process. The identification of relevant studies was accomplished using the two academic search engines *Google Scholar*[1] and *CiteSeerX*[2] and by applying a three-pronged search strategy consisting of the following steps.

1. Performing a direct search for relevant studies using combinations of primary and secondary search terms (see below).
2. Scanning the bibliography of identified relevant studies for further relevant studies.
3. Searching for studies that cite studies that were identified in the previous steps.

For each study identified during the execution of those steps the abstract and conclusion were scanned to further omit irrelevant studies. Finally, a full text scan was performed before any study was accepted. The lists of primary and secondary search terms used during the first phase of the search process were comprised of the following terms:

- Primary terms: impact analysis, change impact analysis, change propagation
- Secondary terms: software, evolution, maintenance, review, taxonomy, classification, comparison, study

Step 4: Data Extraction. The studies that were identified as relevant according to our research questions and goals were archived and analyzed in a structured way using the following scheme for extracting all relevant data.

- Addressed problems - What problems are addressed by the study?
- Research questions - What questions does the study try to answer?
- Contribution - What is the actual contribution of the study?
- Proposed solution - What is the solution for the stated problems?
- Open issues - What are the unsolved issues and future work acknowledged by the authors?

[1]http://scholar.google.de/
[2]http://citeseer.ist.psu.edu/

Step 5: Data Analysis and Results. Finally, the extracted data was analyzed and processed according to the intentions of the review and resulted in the following outcome.

1. We established a taxonomy for change impact analysis approaches in order to compare them using a fixed set of criteria, which is discussed in Section 2.2.
2. We identified six major techniques that are being used for change impact analysis, which are presented in Section 2.3.
3. We identified existing multiperspective impact analysis approaches and the types of software artifacts supported by them, which is discussed in Section 2.4.
4. We discovered open research problems that helped to refine our research goals, which is explained in Section 2.5.

2.2. A Taxonomy for Change Impact Analysis

The systematic literature review identified approximately 180 relevant studies that had to be examined according to the research goals of this thesis. Identifying similarities and comparing the studies, however, demands for precise and well-defined criteria supplied by a taxonomy for change impact analysis. Yet, existing taxonomies did not sufficiently fulfill these requirements and thus had to be revised [Leh11b], which is described in the following sections.

We defined a set of key criteria that must be obtained from existing impact analysis approaches in order to compare and evaluate them in a comprehensive manner and to answer our initial research questions. Consequently, they also serve as requirements for the revised change impact analysis taxonomy.

- **REQ1:** Provide information on the required input (types of artifacts, types of changes).
- **REQ2:** Provide information on the computed output (types of impacts).
- **REQ3:** Provide information on the utilized algorithm/technology.
- **REQ4:** Provide information on the supported modeling/programming languages.
- **REQ5:** Provide information on the availability of (semi)-automated tool support.
- **REQ6:** Provide information on the scalability of the approach.
- **REQ7:** Provide information on the quality of achieved results.
- **REQ8:** Provide information on the required manual interaction and effort.

Based on these requirements we first evaluate the existing change impact analysis taxonomies in the next section. We then outline why they are not sufficient and why a new taxonomy is required that is a part of the contribution of the research presented in this thesis.

2.2.1. Evaluation of Existing Taxonomies

An initial framework for comparing change impact analysis approaches was proposed by Arnold and Bohner [AB93]. Their framework allows for classifying impact analysis approaches according to the applied impact analysis technique, supported software artifacts, quality of obtained results, and the required amount of human interaction. In contrast to the requirements stated in the previous section, however, their framework does not provide any information on the supported change types, computed results, scalability of the approaches, supported programming and modeling languages, and on potential tool support. Thus, their framework lacks the required applicability in regard to our goals. On the other hand, it already provides a solid baseline for establishing an enhanced taxonomy of change impact analysis approaches.

A different classification was later introduced by Kilpinen [Kil08] who divides change impact analysis approaches into three groups: *Traceability Impact Analysis*, *Dependency Impact Analysis*, and *Experimental Impact Analysis*. This distinction according to the utilized technology, however, does only fulfill one of our eight requirements. Her classification does not provide any information on the software artifacts and change operations that are supported by an approach. Furthermore, important information regarding the scalability and performance of the approaches are not considered by her taxonomy. Thus, the classification of Kilpinen does not fulfill our requirements and is therefore not applicable in regard to our goals.

Table 2.1 summarizes the above stated findings of our review.

Our requirements	**Arnold and Bohner**	**Kilpinen**
REQ1 required input	partly	-
REQ2 computed output	-	-
REQ3 utilized algorithm	yes	partly
REQ4 supported software artifacts	yes	-
REQ5 tool support	partly	-
REQ6 scalability	-	-
REQ7 quality of results	yes	-
REQ8 manual effort	partly	-

Table 2.1.: Coverage of our requirements by existing taxonomies

2.2.2. Towards a More Fine-grained Taxonomy

As the previous section has shown, none of the existing taxonomies is truly suitable for comparing and classifying impact analysis approaches according to our needs. Hence, a new taxonomy is required that is supplied as a part of the contribution of this thesis and was originally published in [Leh11b]. According to our requirements defined above and the analysis of existing classification schemes, we introduce the following criteria that constitute our taxonomy.

- **Scope of Analysis:** defines what types of software artifacts are analyzed, e.g. code, architectural models or a combination of heterogeneous software artifacts (see **REQ1**).
- **Granularity of Artifacts:** defines what artifacts are required as input, what changes are supported, and what types of impacts are detected (see **REQ2**).
- **Utilized Technique:** defines what impact analysis algorithm is used (see **REQ3**).
- **Supported Languages:** defines what programming languages, modeling languages, etc. are supported (see **REQ4**).
- **Tool Support:** provides information whether a tool is available or not (see **REQ5**).

- **Scalability:** defines the time and space complexity of the approach (see **REQ6**).
- **Experimental Results:** reports on the size of the conducted case studies and the achieved results in regard to execution time, precision, and recall (see **REQ7**).
- **Style of Analysis:** defines how the change impact analysis is performed, i.e. exploratory, search-based or as a global analysis (see **REQ8**).

The criterion *Scope of Analysis* is further refined into the categories of "source code", "models", and "miscellaneous artifacts", as most approaches were either focused on source code, various kinds of models or different (but not specified) types of files. Approaches analyzing source code are further refined into approaches based on "static", "dynamic" or "online" analysis of source code artifacts [Leh11b]. Similarly, the scope of models was refined into "architectural models" and "requirements models" as both constitute the majority of the identified works.

Figure 2.1 summarizes our taxonomy and provides a comprehensive view upon it. Practical examples of applying our taxonomy for the classification of approaches can be found in [Leh11b, Leh11a] and in Section 8.4.5 where we classify our approach proposed in this thesis.

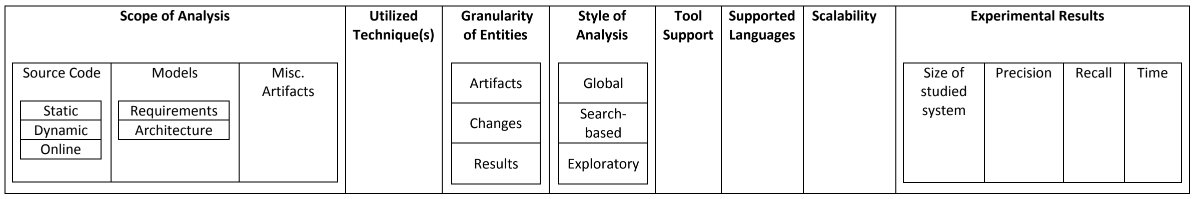


Figure 2.1.: Our taxonomy for change impact analysis approaches [Leh11b]

2.2.3. Investigating the Applicability of our Taxonomy

Further, we investigated the applicability of our taxonomy by classifying all the studies that were identified as relevant by our literature review according to its criteria. During this classification process we also measured the coverage of our proposed criteria in regard to the studied literature [Leh11a]. Our findings are discussed in the following and are finally summarized by Figure 2.2.

While evaluating the coverage of our criteria, we notice the trend that most researchers did not provide any information on the actual performance of their approaches, i.e. there are only few reports on the obtained precision (25%) and recall (21%) or the required computation time (13%). Furthermore, only few researchers (10%) reported on the time and space complexity of their approaches, which might be due to difficulties in obtaining these figures.

In contrast, information on the types and granularity of the involved software artifacts (94%) and computed impacts (91%) could be obtained from most studies. The only exception is the granularity of the addressed change operations, which only 60% of all studies mentioned explicitly. Likewise, information on existing tool support (75%) and the supported modeling or programming languages (73%) could also be extracted from most studies.

In conclusion, our taxonomy is applicable for classifying the change impact analysis approaches proposed by current research, as most of its criteria are covered by the majority of the approaches identified by our review. Moreover, our taxonomy provides more detailed information on the classified studies than existing taxonomies and therefore eases their comparison.

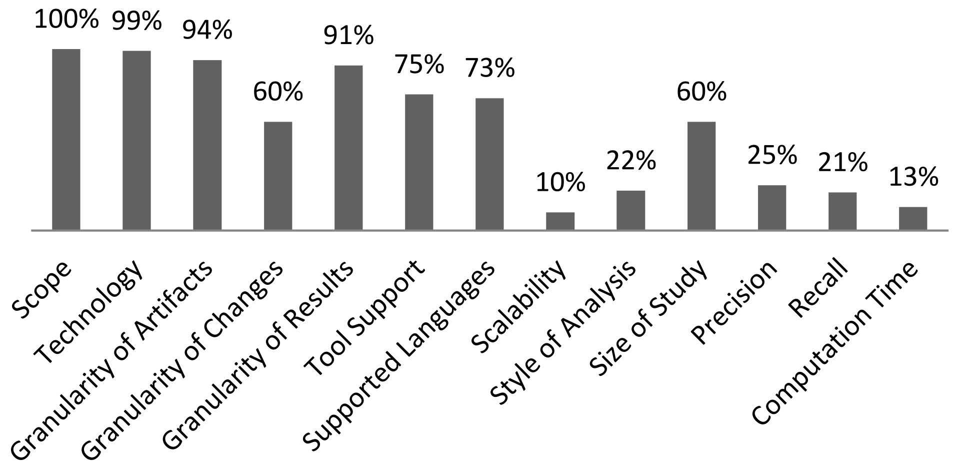


Figure 2.2.: Coverage of our criteria in the studied literature [Leh11a]

2.3. Change Impact Analysis Techniques

The following analyzes related work on change impact analysis identified by our review according to the utilized impact analysis approach. The upcoming sections will focus on the major techniques proposed in the literature, examine their basic ideas, and evaluate them according to their potential of supporting multiperspective impact analysis. Our three main requirements for analyzing the existing change impact analysis approaches are derived from our research goals as laid out in Section 1.2 and are therefore defined as follows.

- The capability of analyzing heterogeneous types of software artifacts.
- The support for developers trying to comprehend the impacts of their changes.
- The support for different types of change operations.

In the following subsections we are discussing the six major categories of change impact analysis techniques as identified by our review. Likewise, a later review conducted by Li *et al.* on source code based change impact analysis techniques came to a similar classification [LSLZ13]. However, our classification as initially presented in [Leh11b, Leh11a] is more comprehensive due to its wider scope (see also Section 2.2.2).

2.3.1. Dependency Analysis

Our review identified many change impact analysis approaches utilizing dependency relations between software artifacts for the actual change impact analysis. These approaches can be further divided into five distinct groups that are discussed in the following sections.

2.3.1.1. Distance-based Graph Analysis

Early approaches for impact analysis explicitly extracted the dependencies of a software system and utilized the obtained dependency graph for estimating the propagation of changes across the dependent software artifacts using reachability analysis. However, propagating changes across all dependencies within the graph often lead to an "explosion of impacts" [Boh02a] that left the entire software system affected. Hence, the impact computed for a given change was likely to be overestimated and thus not useful to support maintenance or reengineering activities.

Consequently, an extension of the approach was proposed to limit the propagation of changes using a fixed cut-off distance. The underlying assumption is that changes are only having a local impact, which is why further change propagation is cut off once a certain distance to the initially changed artifact is reached (i.e. a certain amount of dependency relations were "crossed") [Boh02a]. However, these approaches typically suffer from the choice of the propagation distance, which still remains an open research question [HBG$^+$11]. If the distance is too large, many false-positives will be detected as all software artifacts in range are consider as being impacted. In contrast, a too narrow distance may result in many missed impacts, since not all of the actually impacted software artifacts can be reached.

Advantages	**Disadvantages**
(+) Multiperspective analysis possible	(-) Hard to determine correct distance
(+) Easy to understand and implement	(-) No distinction between different changes

Table 2.2.: Summary of the distance-based graph analysis approach

2.3.1.2. Message Dependency Graph Analysis

A special kind of dependency-based impact analysis approaches was developed for distributed and event-based systems, for which the message communication between the possibly remote components of a system is analyzed. By monitoring the exchanged messages and recording the obtained communication paths, a dependency graph can be constructed that can be utilized for a later impact analysis. Examples can be found in the works of Yoo and Choi [YC04] and Popescu *et al.* [PGBM10, Pop10].

Advantages	**Disadvantages**
(+) Analysis of distributed systems	(-) Requires execution of software
(+) Analysis of event-based systems	(-) Requires monitoring of messages
	(-) Coarse-grained analysis

Table 2.3.: Summary of the message dependency graph analysis approach

2.3.1.3. Call Graph Analysis

Assessing the impacts of changes on existing source code can be accomplished by studying the call-relations of the methods and functions that constitute the code. Therefore, method and function calls have to be extracted from the source code and stored in a so called "call graph". Once the extraction is done, the transitive closure of the obtained graph is computed, which is later on utilized for impact analysis tasks. In this approach methods are considered as being impacted by a change if they call a changed method, or if a changed method is called by one of their called methods (recursively). Examples can be found in works of Ryder and Tip [RT01], Ren *et al.* [RST+03, RST+04, RRST05, Ren07], and Störzer *et al.* [SRRT06].

Advantages	Disadvantages
(+) Easy to extract call graph	(-) Only applicable on source code
	(-) Granularity limited to methods
	(-) No distinction between different changes

Table 2.4.: Summary of the call graph analysis approach

2.3.1.4. Dynamic Execution Trace Analysis

Approaches for call graph based change impact analysis typically suffer from low precision [LR03]. One way of improving their results is to only consider those methods that were actually called during the execution of a program. Hence, the idea of monitoring the execution of programs for change impact analysis purposes was developed. This, however, requires programs to be instrumented and executed to monitor their call behavior. The recorded execution traces can then be utilized for change impact analysis, where all the methods of a trace containing at least one changed method are considered as being impacted. Examples can be found in the works of Orso *et al.* [OAH03, OAL+04], Law and Rothermel [LR03], Breech *et al.* [BTP05], Apiwattanapong *et al.* [AOH05], Huang and Song [HS06, HS07, HS08], and Gupta *et al.* [GSC09, GSC10].

Advantages	Disadvantages
(+) More precise than call graphs	(-) Only applicable on source code
	(-) Granularity limited to methods
	(-) Requires instrumentation of code
	(-) Requires execution of code
	(-) No distinction between different changes

Table 2.5.: Summary of the dynamic execution trace analysis approach

2.3.1.5. Program Slicing

Program slicing is a technique that is often used for program comprehension and software maintenance, and can potentially be used for change impact analysis as well. Slicing "brushes away" all the code statements that do not affect a certain program variable, leaving behind the code statements that might be affected by changes to the variable. A slice is computed by analyzing data dependencies (e.g. value assignments) and control dependencies (e.g. if-conditions) of program statements that access or modify a variable. Hence, it can be applied in a forward manner ("What will happen with this variable?") or in a backward manner ("Where does the current value come from?") [Tip94]. Thus, when changing a certain program statement, slicing can be used to identify all the statements that are affected by the change. Examples can be found in the works of Gallagher and Lyle [GL91], Tonella [Ton03], Vidács *et al.* [VBF07], Korpi and Koskinen [KK07], Santelices and Harrold [SH10], Sun *et al.* [SLTZ11], and Yazdanshenas and

Moonen [YM12]. Slicing tools are also available for different programming languages and IDEs, such as *Indus* for Java [Ind] or *Wisconsin* for C/C++ [Wis14] for example.

Advantages	**Disadvantages**
(+) Good tool support	(-) Only applicable on source code
(+) Acceptance by programmers	

Table 2.6.: Summary of the program slicing approach

2.3.2. Mining of Software Repositories (MSR)

In contrast to the techniques discussed so far, MSR-based approaches do not extract dependencies from software artifacts but from the software repositories the artifacts evolved in. They investigate evolutionary dependencies between software artifacts that are only obtainable from software repositories keeping track of an artifacts' version history, such as CVS, SVN or Git. MSR-based approaches investigate the co-change or co-evolution patterns of software artifacts to determine if they were frequently changed together in the past. By reasoning about the frequency of potential co-changes, MSR-based approaches are able to propose possibly impacted artifacts if they were frequently changed together. Additionally, most MSR-based approaches apply a sliding window technique to improve the precision of their predictions, as otherwise early and meanwhile outdated co-change events might mislead current predictions [GDL04]. Examples can be found in the works of Gîrba *et al.* [GDL04], Hassan and Holt [HH04], Ying *et al.* [YMNCC04], Zimmermann *et al.* [ZWDZ05], Fluri *et al.* [FGP05, FG06], Kagdi *et al.* [KM07a, Kag07, Kag08, KGPC10], Nadi *et al.* [NHM10], Canfora *et al.* [CCCDP10a], and Hassaine *et al.* [HBG$^+$11].

Advantage	**Disadvantage**
(+) Multiperspective analysis possible	(-) Requires availability of history
	(-) Dependent on commit-behavior
	(-) Artifacts must evolve in same repository
	(-) Not applicable in early phases
	(-) Selection of sliding window critical
	(-) No distinction between different changes

Table 2.7.: Summary of the MSR-based approach

2.3.3. Information Retrieval (IR)

IR-based approaches scan the names, identifiers, and other free text components contained by software artifacts for similar textual patterns. If such textual similarities between two artifacts were found, it is assumed that changing one artifact also impacts the other one as well. Similar to the previously discussed MSR-based approaches, they are potentially able to analyze different types of software artifacts, as only their textual components are considered. Typical IR-based approaches consist of two phases: the preprocessing and the actual text analysis. During the preprocessing phase, techniques such as word stemming or stop word elimination are applied to reduce the potential search space [Bod11]. The actual text analysis is then accomplished using techniques such as n-gram-matching [CT94] or by applying latent semantic indexing [MM03] for instance. However, these approaches are not able to distinguish between different types of change operations, nor are they able to specify the resulting impacts. Examples can be found in the works of Antoniol *et al.* [ACCDL00], Vaucher *et al.* [VSV08], Poshyvanyk *et al.* [PMFG09], and Binkley and Lawrie [BL10].

Advantages	**Disadvantages**
(+) Multiperspective analysis possible	(-) No distinction between different changes (-) Only lexical similarities considered

Table 2.8.: Summary of the IR-based approach

2.3.4. Probabilistic Approaches

A whole set of probabilistic approaches have been proposed to model and analyze software systems under change using probabilistic models, such as Bayesian Belief Networks or Markov chains. Once a system has been modeled accordingly, probabilistic analysis can be performed for change impact analysis tasks, such as Bayesian inference or Granger Causality Tests. According to the computed likelihood, a software artifact is then considered as being impacted and reported to the developer. Examples can be found in the works of Lock and Kotonya [LK99], Tang *et al.* [TNJH07], Sharafat and Tahvildari [ST07, ST08], Zhou *et al.* [ZWG$^+$08], Abdi *et al.* [ALS09b, ALS09a], Ceccarelli *et al.* [CCCDP10b], and Canfora *et al.* [CCCDP10a].

Advantages	**Disadvantages**
(+) Multiperspective analysis possible	(-) Difficult to model the interplay of heterogeneous software artifacts (-) Difficult to understand the computed change propagation (-) No distinction between different changes

Table 2.9.: Summary of probabilistic approaches

2.3.5. Rule-based Approaches

A different impact analysis methodology is applied by rule-based approaches that utilize explicit rules to forecast the impacts of changes. The underlying assumption is that a set of rules can be established that are able to compute the impact of applying a specific change operation on a software system. The actual definition or creation of the required rules differs from approach to approach. Some researchers propose to utilize developer knowledge, domain knowledge or knowledge of design methodologies for constructing valid impact analysis rules. Dependent on the types of software artifacts to be analyzed, different query languages can be utilized to implement the rules, such as OCL [OMG12], XPath [W3C10] or EMF Query [EMFb]. Examples can be found in the works of Queille *et al.* [QVWM94], Barros *et al.* [BBE$^+$95], Briand *et al.* [BLBS02, BLO03, BLOS06], Feng and Maletic [FM06], Keller *et al.* [KSD09, KD11], ten Hove *et al.* [tHGK$^+$09], and Müller and Rumpe [MR14].

Advantages	Disadvantages
(+) Multiperspective analysis possible	(-) Creation of rules is time consuming
(+) Can address different changes	(-) Hard to address ambiguous impacts
(+) Can be enhanced with repair plans	(-) Rules require maintenance
(+) Allows addition/change of rules	(-) Rules require validation
	(-) Creation of rules not addressed by current research

Table 2.10.: Summary of the rule-based approach

2.3.6. Hybrid Approaches

There are hybrid approaches that combine various of the aforementioned techniques for conducting change impact analysis. As for example the work of Gethers *et al.* [GDKP12] who utilize a combination of information retrieval, mining of software repositories, and dynamic execution trace analysis. Although their approach is potentially able to detect the impacts of changes applied to heterogeneous software artifacts, it is not able to determine the exact types of impacts and furthermore suffers from other disadvantages of the approaches it combines.

2.4. Multiperspective Approaches

One of the key questions of our literature review was whether there exist approaches for multiperspective change impact analysis and if they meet our requirements. Hence, in this section we present the results of our review in regard to the types of software artifacts and perspectives covered by recent studies. In accordance to our impact analysis taxonomy, we consider three major types of software artifacts and their associated perspectives: source code, software architectures, and requirements. Moreover, some studies also analyze configuration and documentation files, which we refer to as “other artifacts” due to their inhomogeneous purposes and structure (see Section 2.2.2). Figures 2.3 and 2.4 below summarize the findings of our review.

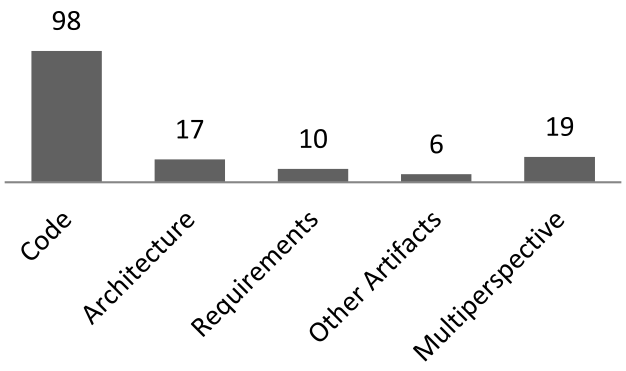


Figure 2.3.: Distribution of scopes among the studied approaches [Leh11a]

It is obvious that the vast majority of the change impact analysis approaches proposed in the literature is still solely focused on source code [Leh11a], while other types of software artifacts are more or less being neglected. We noticed an increasing interest in supplying change impact analysis support for the architectural level and architectural design decisions, though.

Secondly, we investigated how many perspectives and different types of software artifacts are covered by the few existing multiperspective approaches. Figure 2.4 summarizes the distribution of the different perspectives among the few available multiperspective approaches.

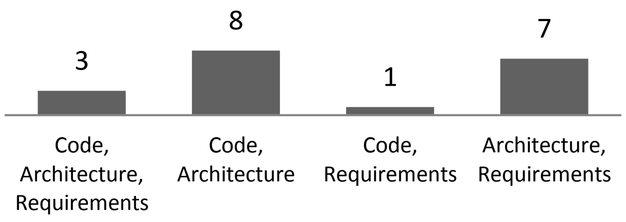


Figure 2.4.: Scopes supported by multiperspective approaches [Leh11a]

The vast majority of the existing multiperspective approaches (16 out of 19) is focused on exactly two perspectives, while only three approaches are truly able to address heterogeneous types of software artifacts (see Figure 2.4). Additionally, most of the approaches are strictly limited in their support of different types of change operations.

In the remainder of this section we discuss the few truly multiperspective approaches in detail. However, we do not discuss all works identified by our review because some of them only represent extended versions of other preceding studies (e.g. journal publications that are based on preceding conference papers of the same authors etc.).

Hammad *et al.* [HCM09] and Sharafat and Tahvildari [ST07] propose approaches for analyzing the change propagation between source code classes and UML design classes. Hammad *et al.* rely on dependency analysis for determining the effects of “add” and “delete” operations, while Sharafat and Tahvildari apply a probabilistic algorithm for the same purpose. However, both code and UML classes are instances of one and the same concept and therefore quite similar. Their approaches do not provide any means for analyzing other important UML models, such as component diagrams, nor do they allow for analyzing actual source code statements. Thus, their approaches are not truly “multiperspective” in regard to the goals of this thesis.

In a similar fashion, Kotonya and Hutchinson [KH05], as well as Khan and Lock [KL09], analyze the change propagation between architectural components and system requirements using approaches built upon dependency analysis. Although this is an import step for analyzing the overall change propagation, further connections to other architectural artifacts and implementation artifacts are required to provide a truly holistic estimation of the impact of a change. Currently, there is still a yawning abstraction gap between high-level architectural components and fine-grained design artifacts, let alone implementation classes and source code statements.

Kim *et al.* [KKK10] and Hassan *et al.* [HDB10] propose approaches for assessing the propagation of changes in between source code and architectural components. Hassan *et al.* utilize a rule-based approach, while Kim *et al.* apply a simple dependency-based fan-in fan-out approach. However, as already pointed out for the approaches discussed above, there is a gap between the analyzed abstract architectural components and the fine-grained design that complicates the change impact analysis and is not addressed by their works. Furthermore, in the approach of Kim *et al.* architectural components are extracted from source code using a revised version of the reflexion model approach of Murphy *et al.* [MNS01], which in turn restricts the addressable architectural components to those that can be recovered from the source code.

Finally, our systematic review discovered only two approaches that are truly applicable for the interplay of heterogeneous types of software artifacts. We discuss them in the following and point out why they are still not sufficient and why further research is necessary.

Briand *et al.* [BLBS02] propose an approach for change impact analysis and regression test selection that is capable of analyzing test cases, use cases, class diagrams, and sequences diagrams. By the means of change impact analysis the authors propose an approach for classifying test cases as either reusable, re-testable or obsolete. The proposed impact analysis approach is based on traceability relations between the UML models and the test cases and can be classified as dependency-based impact analysis. However, the approach does not cover important structural UML artifacts, such as packages or components that are often used in practice. Additional behavioral diagrams, such as state charts or activity diagrams, are also not considered, which limits the applicability of the approach and reduces its capability for multiperspective change impact analysis in regard to the goals of this thesis. Moreover, source code artifacts are also entirely neglected from the impact analysis, thus further limiting the scope of the approach.

The approach proposed by Ibrahim *et al.* [IIMD05a, IIMD05b, IIMD06] allows for analyzing requirements, test cases, classes, and methods using dependency analysis. The required dependencies are obtained by a three-pronged approach. Static code analysis is used for extracting a call graph from the source code, while the test cases are executed to gather further dependency relations between methods and requirements. Thirdly, additional dependencies are supplied by applying a manual traceability mining technique. The recorded dependencies are stored as traceability links that are used for analyzing the change propagation between the software artifacts. However, the approach does not provide support for distinct types of change operations, nor does it support all source code artifacts or architectural diagrams. Furthermore, the approach requires the instrumentation and execution of test cases for linking methods with requirements, as well as additional manual traceability detection. This instrumentation and manual analysis vastly adds to the cost of applying this approach on a system of realistic size and complexity (see discussions in Section 2.3.1.4) which contradicts with our goals.

2.5. Open Research Issues

Our review identified four general problems of existing works on impact analysis that correspond to the goals of this thesis and therefore justify the conducted research.

P1 - Multiperspective Impact Analysis. There is a lack of change impact analysis support for heterogeneous software artifacts and for multiperspective software. As revealed by our review of current impact analysis approaches, almost two thirds of the them are still solely focused on source code [Leh11a]. Only a minority of the proposed impact analysis approaches is capable of analyzing more than one type of software artifact (e.g. source code and UML models). However, in Section 2.4 we have shown that the few available approaches are accompanied by various problems and limitations and thus do not solve the goals of this thesis.

P2 - Change Operations. Our review revealed that there exists no consistent approach for the modeling of change operations among the studied impact analysis approaches. To the contrary, almost every study introduced its own classification of changes. Change impact analysis on the other hand first of all demands for a precise classification and modeling of change operations. Therefore, a comprehensive investigation of the different types of change operations is required.

P3 - Dependency Relations. A similar problem has been identified for the dependency relations that cause changes to propagate to other software artifacts. The studied dependencies are often not explicitly specified nor properly classified. Furthermore, the proposed classifications either contradict with each other or are still incomplete. Consequently, a thorough investigation of dependency relations is required to allow for reliable change impact analysis support.

P4 - Expressiveness of the Impact Analysis. The outcome of current research on impact analysis is very limited in regard to **Goal 3** of this thesis. Only very few approaches are actually able to inform developers about why and how a certain software artifact is impacted by a change and what they should do about it. Independent of the later usage of the results of the change impact analysis (change planning, change implementation, etc.), it is important to know why and how a certain software artifact is impacted, as otherwise all impacts have to be treated in the same way.

2.6. Summary

In this chapter we discussed the process and the results of a systematic literature review conducted in the field of change impact analysis according to the research goals of this thesis. First, the review provided an overview of existing change impact analysis approaches and the different techniques applied by them. Second, it discovered yet unsolved problems that correlate with the research goals of the thesis. Third, a taxonomy for impact analysis approaches was developed and evaluated that assists with comparing impact analysis approaches and allows for classifying the solution proposed by this thesis. We identified five main categories of impact analysis techniques, out of which four are potentially able to analyze the change propagation between heterogeneous types of software artifacts. Subsequently, we discussed their potential and main limitations in regard to the goals of this thesis. We further investigated to which extent current impact analysis approaches support multiperspective impact analysis. However, in the recent literature only very few approaches actually support multiperspective impact analysis. They do either not cover the required software artifacts and development phases or their applicability is constrained by the required amount of manual effort, e.g. for instrumenting and executing the software under change. Finally, our literature review identified the four main limitations of current impact analysis approaches that we address with the research presented in this thesis, namely: the lack of multiperspective impact analysis support, the adequate treatment of different types of changes and dependencies, and providing support for developers to enable them to understand the outcome of the impact analysis.

3. Thesis Foundations

In this chapter we explore related work that constitutes the foundation of research on change impact analysis. To begin with, we analyze the concept of software views and its relation to impact analysis in Section 3.1. We then analyze current research on multiperspective modeling and consistency checking in Section 3.2 which shares common aims and challenges with multiperspective change impact analysis. In Section 3.3 we analyze how dependencies between the different views and software artifacts can be elicited, recorded, and classified in order to prepare the ground for understanding how the effects of changes propagate across them. Section 3.4 explores possible means for modeling and classifying change operations to provide a solid understanding of changes according to the needs of impact analysis. Finally, Section 3.5 summarizes our findings to later refine the research questions of this thesis.

3.1. Views on Software

A *view* or *perspective* describes a certain cutout of a software system that is tailored for a specific purpose, such as presenting a general overview of the structure of a system or describing the services provided by a system [Kru95, FMP99]. Thus, each view conveys an excerpt of the concepts that constitute a software [SF01]. Many of these views have been proposed and explored in existing works and this section therefore presents an overview of typical views and their purposes to highlight their relevance for change impact analysis.

Initially, Kruchten [Kru95] defined five views on software architectures, namely the *logical view*, the *process view*, the *physical view*, the *development view*, and the *use case view*. For our work we can apply these views not just for the architecture of a software system, but also for its other aspects, such as the final implementation in source code. In fact, the heterogeneous artifacts to be addressed by impact analysis approaches represent excerpts of those views on different levels of abstraction. Due to our goals, however, this thesis will mostly focus on the *logical view* describing the decomposition of the design that contains most of the typical software artifacts and the *process view* describing the interactions of those artifacts.

Cook *et al.* [CJH01] present an approach to study software evolution based on the *dynamic* and the *static view* of software. The dynamic view describes the trends of a system's evolution, whereas the static view describes the characteristics of the involved software artifacts. Additionally, each view is further refined by three different conceptual levels, namely the *requirements level*, the *architectural level*, and the *fine design and source code level*. For our work it is important to distinguish between the different conceptual levels when performing impact analysis, even though this thesis is only focused on the architectural and fine design level of Cook *et al.*

The work of Dueñas and Capilla [DnC05] goes beyond previous approaches by explicitly capturing design decisions in the *architectural decision view*. Explicitly recorded decisions present a valuable asset for change impact analysis [TLCvV11b]; however, a meta-model for capturing design decisions is still subject to ongoing research [GLR14]. Moreover, for the majority of software systems these information no longer exist as separate items, since they were not explicitly recorded in the first place. Thus, we exclude this view from our current work.

A further extension of the set of views is proposed by El Ghazi and Assar [EGA08] who introduce the *actors perspective*, *product perspective*, *process perspective*, *evolution perspective*, *configuration perspective*, *rationale perspective*, and the *traceability perspective*. However, as our approach is focused on software artifacts and not on software development processes, the proposed views are less helpful for change impact analysis of software artifacts. Nevertheless, their research provides a valuable starting point if one wants to extend impact analysis beyond software artifacts to address the actual software development processes.

3.2. Managing Heterogeneous Software Artifacts

Software development consists of different phases that require different types of software artifacts to express the concerns of each phase [SF01]. Thus, the different artifacts are dependent on each other and must evolve together [EPRV08]. However, in most software systems that evolve over time there is a drift between the different types of software artifacts [IK06]. To counter this drift, research on multiperspective modeling and multiperspective consistency management has been conducted with the aim of keeping the artifacts synchronized. The following analyzes how related work deals with the interplay and evolution of heterogeneous software artifacts. Additionally, we discuss how these approaches bridge the gap between the different types of software artifacts and how they deal with the emerging inter-model dependencies.

3.2.1. Multiperspective Modeling

The work of Yie *et al.* [YCDW09] addresses the problem that different models and views are used for modeling the different concerns of an application, which in turn have to be composed to obtain a complete representation of the system. They propose to transform the heterogeneous models into a unified representation that is based on a common low-level modeling language. In a second step the final system model is obtained by composing the unified low-level models, for which correspondences between them are derived. However, composing the final model from the set of intermediate low-level models further demands for the correspondences between the models and their low-level representations to be explicitly established too. This explicit linkage is achieved by recording the correspondences as traceability links between the models.

In a similar fashion, Eramo *et al.* [EPRV08] present a framework for multi-view modeling utilizing multiple independent views and viewpoints that are connected by correspondence relations. However, as the views and viewpoints evolve, synchronization is required, which is why the *correspondence* relations have to be explicitly modeled. The authors propose to use the concept of *Answer Set Programming* that is embedded into a transformation engine to ensure that the consistency of the involved views is maintained using declarative logic and proof procedures.

Demuth *et al.* [DLHE11] present an approach and an accompanying tool for cross-layer modeling to facilitate the co-evolution of different meta-models and their model instances. Their approach allows for linking between models of different layers using UML dependency connections, and provides rules for checking their consistency. The core concept of their approach is that instances of a certain element can be created on different meta levels with a different degree of abstraction and level of detail. However, they also support the usage of different models for expressing the different concerns of a system and thus allow for multiperspective modeling.

In conclusion, an idea common to all of the above mentioned approaches is the usage of explicitly modeled correspondence relations between the heterogeneous software artifacts to express dependencies and overlappings between them. Furthermore, another interesting idea that was brought up is to map the heterogeneous software artifacts on a common low-level model whose instances are then used for further analysis.

3.2.2. Multiperspective Consistency Management

Fradet *et al.* [FMP99] introduce a framework for representing and analyzing software architectures that are build upon multiple views, whereas each view is comprised of different types of models adhering a graph-based structure. The authors further introduce a constraint checking algorithm to deal with potential inconsistencies that can be utilized during the phase of architectural design. To accomplish this, the proposed algorithm analyzes the dependency relations connection the software artifacts, which is why it is closely related to the dependency-based impact analysis approaches as discussed in the previous chapter.

Sunetnanta and Finkelstein [SF01] explore why software development requires different perspectives, models, and diagrams from an end-users point of view. The authors investigate the different types of end-users, such as requirements engineers or architects, and explain why they demand for different modeling perspectives. They conclude that the usage of different perspectives, however, complicates the process of maintaining the consistency among them. To cope with the different views, they propose a multiperspective viewpoint framework that is based on conceptual graphs (CG) for linking the different types of models. Each CG consists of the concepts to be modeled (nodes) and potential relations between them (edges), whereas relations are also required for connecting models of different perspectives. Finally, a set of simple consistency checking rules is applied to assist with maintaining the consistency of the perspectives.

Olsson and Grundy [OG02] propose an approach for multiperspective traceability and inconsistency management that covers requirements descriptions, UML use cases, and black-box test plans. Their goal is to adapt the entire system to evolutionary changes of single artifacts in order to maintain the overall consistency of the system. Therefore, changes are propagated across dependency relations between the software artifacts, for which they are either implicitly linked by their underlying meta-model or explicitly by users of their prototype tool that implements their approach. The degree of automation of the change propagation, however, depends on the applied change type, as only “simple” changes, such as rename operations for example, are handled by their approach. More complex change scenarios are directly presented to the user.

Muskens *et al.* [MBC05] investigate inconsistencies between the various views of software that emerge from the different phases of software development. Their approach allows for consistency checking among and in-between different phases, for example spanning high-level design artifacts and the implementation-related, fine-grained design of classes. They provide a set of consistency checking rules for UML models and C++ code that are based on relational partitioning algebra. These rules are designed to analyze the dependency relations that exist between the software artifacts, although it is not mentioned where the relations stem from. The key part of their approach is to determine which model's evolutionary changes are prevailing and thus serve as a reference for the evolution of all other artifacts. Overall, their approach is similar to the rule-based impact analysis approaches as discussed in Section 2.3.5.

The work of Kolovos *et al.* [KPP08] addresses the problem of inconsistency management that arises from the usage of multiple DSLs and modeling techniques in software development. The authors recognized the need for a distinct modeling and classification of inter- and intra-model dependencies and therefore analyzed the types of dependency relations between the models. They propose a novel classification of dependencies, which, however, seems not to be based on a thorough review process. Moreover, they do not state where the addressed dependencies stem from. As we will show in Section 3.3.1, there are different sources of dependencies, which is why their classification is incomplete, which in turn might affect their ability to thoroughly analyze the interplay of the different models.

In summary, our analysis further strengthens the need for a thorough investigation of multiperspective dependency relations, especially in regard to how they can be modeled and classified. Likewise, they emphasize the need for addressing the heterogeneous artifacts in a homogeneous manner, such as a common low-level meta-model for example.

3.3. Dependency Relations

In Section 2.3.1 we have reviewed different types of change impact analysis algorithms which, however, all rely on the analysis of dependency relations for estimating the propagation of changes. Consequently, dependency relations play an important role for change impact analysis [GS82, Raj97, Boh02a]. In order to utilize dependency relations for impact analysis, it is therefore necessary to thoroughly understand the dependencies, their origins, and their types in the first place. Hence, this section explores the origins and types of dependency relations between software artifacts as a prerequisite for change impact analysis. Moreover, we discuss the concept of traceability links as one possible means for expressing dependency relations between software artifacts. We investigate how traceability links can be recovered from software artifacts, and how these concepts can be utilized for multiperspective dependency detection.

3.3.1. Origin of Dependencies

In this section we analyze the potential sources of dependencies. In order to conduct multiperspective change impact analysis, one has to be aware of these sources to take the dependencies that are emerging from them into account during the actual impact analysis process.

1) Dependencies defined by the meta-models of software artifacts. A meta-model, such as the UML meta-model or the C++ language specification, defines the elements and possible relations a software artifact can be comprised of. Hence, various dependencies are directly encoded in them. For example, there exist dependencies between UML use cases and UML use case actors or between Java interfaces and the operations provided by them. These relations are also referred to as "structural relations" by De Lucia *et al.* [DLFO08].

2) Dependencies defined by software development paradigms. Dependencies also stem from the concepts enforced by certain development paradigms. Our work focuses on the object oriented paradigm, which introduces additional dependencies that do not occur in procedural software systems [Boo94]. Typical examples are *inheritance* relations between classes or *implementation* relations between classes and interfaces. These relations are also classified as "structural relations" by De Lucia *et al.* [DLFO08].

3) Dependencies introduced by development methodologies. Additional dependencies are also introduced by development methodologies that are applied by system analysts, architects, and developers. Within the scope of this thesis we are extracting such dependency relations by analyzing the development methodologies of *Object Oriented Analysis* [CY91, Boo94] and *Object Oriented Design* [Boo94]. When following these guidelines, various dependencies are introduced between the architecture and implementation of a system. Such relations are referred to as "knowledge-based relations" by De Lucia *et al.* [DLFO08].

4) Dependencies stemming from the usage of different views. Finally, further dependencies are introduced by the usage of different views and perspectives for expressing the different aspects of a software system. There are views that are not bound to a specific development methodology or paradigm, such as the behavioral or structural view (see discussions in Section 3.1), that are interrelated by dependencies. There are, for example, overlappings between components of the structural view and their representation in the behavioral view.

It is important to note that all these sources are not disjoint, as for example all object oriented relations are already contained in the UML meta-model or the Java language specification. Thus, there are overlappings which, however, do not reduce the benefits of addressing a wide scope of potential sources of dependencies.

Moreover, we like to point out that there are additional sources of dependencies that are not yet covered by this thesis, such as software development processes for instance. We are further convinced that determining a finite list of all the sources of dependencies is not feasible; however, in this thesis we address potential sources that, even though to a varying extent, are inherent to the majority of software development projects.

3.3.2. Types of Dependency Relations

Apart from explicitly addressing the origins of different dependencies, it is also crucial to understand the types and purposes of those relations. Research on software dependencies has brought up a wide variety of different types of dependency relations, which potentially have to be addressed by change impact analysis approaches. However, since the amount of proposed dependency types is vast (see Table 3.1), it is necessary to group and classify them to enable their consistent usage for subsequent software engineering activities. This need has first been recognized in the field of requirements traceability where several classifications have been proposed so far. We conducted a literature survey for such classifications and in the following report on its findings.

An initial classification of requirements traceability was introduced by Pohl [Poh96a, Poh96b]. He divided dependencies of requirements into five clusters that are organized as follows.

- **Content dependencies:** relations that signify comparisons, contradictions, and conflicts between requirements.

- **Condition dependencies:** relations between requirements and restrictions associated with them.
- **Evolution dependencies:** relations indicating the replacement of (previous) requirements.
- **Documentation dependencies:** relations associating different types of software documents to a requirement.
- **Abstraction dependencies:** relations representing abstractions like generalizations and refinements between requirements.

A similar classification was later proposed by Ramesh and Jarke to serve as a reference model for requirements traceability [RJ01]. The proposed classification is comprised of four clusters that altogether contain twenty-five different types of dependency relations (see Table 3.1).

- **Satisfaction dependencies:** relations between design/implementation artifacts and requirements artifacts realized by them.
- **Evolution dependencies:** relations documenting the input-output relationships of actions leading from existing artifacts to modified artifacts.
- **Rationale dependencies:** relations representing the rationale behind artifacts or documenting the reason for their existence.
- **Dependency relations:** relations expressing general dependencies among artifacts.

While concerned with identifying directions for future research on software traceability, Spanoudakis and Zisman [SZ05] also reviewed the types of traceability relations as used in the existing literature and organized them in a new classification scheme. Their review covered more perspectives than previous research since they also investigated relations between requirements and source code, and requirements and design models. Spanoudakis and Zisman distinguish between eight different types of traceability relations which they define as follows.

- **Dependency:** exists between two artifacts when the existence of one depends on the other or when changes applied on one artifact must be reflected by the other artifact as well.
- **Generalization/Refinement:** relations used to identify how complex elements of a system can be broken down into components, how elements of a system can be combined to form other elements, and how an element can be refined by another element.
- **Evolution:** relations signifying the evolution of elements of software artifacts.
- **Satisfiability:** relations expressing how one entity meets the expectations, needs, and desires of another entity or how it complies with a condition represented by another entity.
- **Overlap:** relations expressing common features between two entities.
- **Conflict:** relations indicating conflicts between two entities.
- **Rationalization:** relations used to represent and maintain the rationale behind the creation and evolution of entities at different levels of granularity.
- **Contribution:** relations used to represent associations between requirement artifacts and stakeholders that contributed to the generation of the requirements.

Khan *et al.* [KGGR08] analyzed the interplay of requirements dependencies and architectural evolution, from which they derived a taxonomy for the dependencies connecting requirements and architectural components. They propose the following seven dependency clusters.

- **Goal dependencies:** relate system quality attributes (problem space) with their realization (solution space).
- **Service dependencies:** relate requirements with their realizing operations and functions of the system.
- **Conditional dependencies:** relate conditions, constraints, and decisions taken at the requirements level with their realization at the architectural level.
- **Temporal dependencies:** relate requirements specifying the time frame of an event to occur, processes to complete, or condition to hold true with their realization at the architectural level.
- **Task dependencies:** trace the connections between artifacts that require input and feedback from other tasks, processes or users for their completion.
- **Infrastructure dependencies:** relate resources, infrastructure, technical standards/details, design constraints, and compatibility issues to the architectural conception.
- **Usability dependencies:** relate user interaction with the responsible system components.

El Ghazi and Assar [EGA08] proposed a framework for the management of dependencies connecting the different views of software systems. As one constituent of their framework they proposed a taxonomy of traceability relations that is comprised of the following clusters.

- **Satisfaction relations:** express the degree of satisfaction between requirements and system components.
- **Dependency relations:** express generic dependencies between software artifacts.
- **Evolution relations:** express the evolution of software artifacts over time.
- **Rationale relations:** represent the context in which software artifacts are produced.
- **Containment relations:** express structural dependencies between software artifacts.
- **Contribution relations:** link agents and actors that contribute towards software artifacts.

Moreover, our literature survey identified several works that either provide or utilize distinct types of dependency relations without organizing them by a taxonomy or by any other formal approach. The following Table 3.1 summarizes the dependency types that are referred to by those works and the types proposed by the above discussed classifications.

In conclusion, there is still demand for a consistent classification of dependency relations for change impact analysis tasks. The classifications that were proposed so far are either inconsistent to each other, incomplete or contain too many overlappings. Consequently, a comprehensive investigation of the purposes of dependency relations is required to formulate a strict taxonomy of potential relation types for impact analysis tasks. This statement is also supported by the research of Zhang *et al.* [ZLZ$^+$14] who evaluated two existing dependency classifications in an industrial setting and came to similar conclusions in regard to the ambiguous, overlapping, and incomplete character of these classifications.

Approach	Proposed Dependency Types
Tang *et al.* [TLCvV11a]	affected_by, implemented_by, satisfied_by, realized_by, is_proposed_by, comprised_of, is_a, identifies, supercedes, depends_on, results_in, related_to, addressed_by
Constantopoulos *et al.* [CJMV95]	generalization, specialization, generality, specificity, correspondence, similarity, derivedFrom, importsFrom
Walderhaug *et al.* [WJSA06]	refine, transform, evolves_to, generation, justifies, modifies, uses, implements, owns, executes, validates
Sherba and Anderson [SA03]	implemented_by, tested_by, validated_by, allocated_to, elaborated_by, discussed_by, elaborate, depend_on, part_of, refines, replaces, based_on, formalizes, allocated_to
Mäder *et al.* [MPR07]	refinement, realize, verify, define
Espinoza *et al.* [EG11]	is_tested_by, satisfies_test
Aizenbud-Reshef *et al.* [ARPR+05]	rationaleOf, validatedBy, responsibleOf, calls
Jirapanthong and Zisman [JZ09]	satisfiability, depends_on, overlaps, evolution, implements, refinement, containment, similar, different, variability
Espinoza and Garbajosa [EAG06]	satisfies, dependency, rationale, validation, verification, evolution
Ramesh and Jarke [RJ01]	satisfies, dependency, evolution, rationale, allocates, perform, part_of, derive, modify, used_by, address, derived_from, verified_by, developed_for, generate, is_a, elaborated, influence, affect, define, create, comply, supports, manages, resolve
Spanoudakis *et al.* [SZPMK04]	overlap, requires_execution_of, requires_feature_in, can_partially_realise
Olsson and Grundy [OG02]	exact, specialization, generalization, similar, splits, merges, exact group
Paige *et al.* [POK+08, PDK+11]	consistent_with, dependency, has_a, is_a, part_of, import, export, includes, usage, refinement, calls, notifies, generates, builds, synchronized_with, verifies, certifies, satisfies, allocated_to, performs, explains, supports
Letelier [Let02]	validatedBy, traceTo, verifiedBy, assignedTo, rationaleOf, partOf, modifies, responsibleOf

Table 3.1.: Dependency types proposed in related work

3.3.3. Traceability Links

The initial concept of traceability was developed to enable stakeholders to trace the lifecycle of software artifacts, such as requirements for example. A widely accepted definition of the term traceability was given by Gotel and Finkelstein:

> ”The ability to describe and follow the life of an artifact, in both a forwards and backwards direction [GF94].”

A traceability link connects two software artifacts and typically expresses a dependency relation between them. Each traceability link consists of three mandatory attributes: a reference to the *source of the link*, a reference to the *target of the link*, and the *type of the relation*. A link can be further enhanced with additional attributes, such as timestamps, authorship information, etc. An overview of possible types of relations was already provided in the previous section.

As traceability links allow for interconnecting different types of software artifacts [ARNRSG06], they are a suitable data structure for expressing dependencies between heterogeneous software artifacts [III08]. Moreover, traceability links can be utilized for ripple effect analysis [WvP10], and hence for impact analysis [BGW13].

3.3.4. Traceability Detection Techniques

In this section we analyze five techniques that were proposed for traceability detection and that could potentially be reused for multiperspective dependency detection. We review them in regard to their support for heterogeneous software artifacts and their ability to determine the types of the dependency relations. We exclude manual traceability detection approaches from our review as they stand in contrast to our overall goal of providing automated tool support. Further reviews and comparisons of existing traceability detection techniques can also be found in the works of Rochimah *et al.* [RWKA07], Imtiaz *et al.* [III08], De Lucia *et al.* [DLFO08], and in our previous works [Leh10, BLR11, RBFL11].

3.3.4.1. Information Retrieval (IR)

Various approaches for IR-based traceability detection have been proposed in the literature, such as the vector space model [ACC$^+$02] or latent semantic indexing [MM03, LFOT07]. Similar to the IR-based change impact analysis approaches as discussed in Section 2.3.3, they analyze the names and identifiers of software artifacts. Hence, if the names of two artifacts are "similar" a traceability relation between both is created. Many approaches utilize further preprocessing techniques, such as stop word elimination or word stemming, to increase the precision of the link detection. A comprehensive review conducted by Oliveto *et al.* [OGPDL10] revealed that there is little difference between the results achieved by various IR techniques. In fact they turned out to be almost equivalent to each other. In an earlier experiment Abadi *et al.* [ANS08] also compared several IR-based techniques and came to the conclusion that a combination of different techniques might yield better results. However, IR-based approaches lack the required precision and recall when compared to other approaches [Bod11] and are therefore not suitable for dependency detection in multiperspective environments, as too many false-positives are detected. Furthermore, IR-based approaches are also not able to determine the types of the detected relations, which in turn restricts the reusability of the detected dependencies for a later impact analysis.

Advantages	**Disadvantages**
(+) Multiperspective dependency detection	(-) Not able to determine dependency types
(+) Completely automatable	(-) Typically low precision

Table 3.2.: Summary of IR approaches for multiperspective dependency detection

3.3.4.2. Mining of Software Repositories (MSR)

Similar to the previously discussed approaches for history-based impact analysis (see Section 2.3.2), MSR-based approaches can also be applied for traceability recovery [KM07b, KMS07, Kag08]. Therefore, the same assumption is applied as for change impact analysis: if two artifacts were frequently changed together, a traceability relation is likely to exists between them. Consequently, applying MSR for traceability detection suffers from the same limitations. First of all, only evolutionary couplings can be detected, whereas other types of dependencies are entirely neglected. Secondly, different types of software artifacts usually evolve in different repositories and thus do not share a common "history" that can be mined. Moreover, MSR cannot be applied when the version history is incomplete or missing (e.g. legacy systems) or in the early stages of software development when the software is an unstable state.

Advantages	Disadvantages
(+) Multiperspective dependency detection	(-) Not able to determine dependency types
(+) Completely automatable	(-) Dependent on commit-behavior
	(-) Artifacts must evolve in same repository
	(-) Not applicable in early phases

Table 3.3.: Summary of MSR approaches for multiperspective dependency detection

3.3.4.3. Dependency Detection Rules

Researchers and developers can define and execute rules to detect dependencies between software artifacts and to record them as traceability links. To accomplish this, the rules may query the structure, attributes or relations of software artifacts in order to identify dependencies. Typical examples can be found in the fields of requirements traceability [FZS03, SZPMK04, JZ09] and in works on model driven engineering [DPFK08, Kol09, BLR11, RPB11]. These rule-based approaches for traceability detection share the same advantages and disadvantages as the rule-based approaches for change impact analysis reviewed in Section 2.3.5. Namely, they require manual upfront effort for creating and maintaining the rule database, which cannot be automated. On the other hand, they provide more reliable results [Bod11] and allow for a better understanding of their results when compared to other approaches. Furthermore, they are also able to determine the types of the detected dependency relations. Likewise, the concept of detection rules is easier to adapt to other software artifacts as new rules can be added for them, whereas other approaches might require severe changes in their underlying algorithms. However, the identification and creation of dependency detection rules is currently not covered in related works.

Advantages	Disadvantages
(+) Multiperspective dependency detection	(-) Rule-creation causes effort
(+) Can determine types of dependencies	(-) Rule-maintenance causes effort

Table 3.4.: Summary of rule-based approaches for multiperspective dependency detection

3.3.4.4. Semantic Wikis and Ontologies

A fourth group of traceability detection techniques utilizes the concept of semantic modeling for eliciting dependency relations between software artifacts. These approaches are motivated by the fact that software development involves various stakeholders, each using his own special vocabulary that introduces synonyms and homonyms of the same concepts [SF01], and the inevitable drift between software artifacts due to the ongoing evolution [TLvV11]. These approaches propose to index software documents with an ontology that in turn allows for retrieving knowledge from the software artifacts from which traceability links can be inferred. Such an approach is for example discussed by Bode and Wagner [Wag10, Bod11] to bridge the gap between URN models [ITU08], UML models, and factor tables and issue cards [HNS05] using an OWL ontology [W3C09]. Likewise, Tang *et al.* proposed a similar concept [TLCvV11b, TLCvV11a, TLvV11] that allows for a (semi)-automated indexing of documents for dependency retrieval once the concepts of the ontology have been defined. The major difficulty and drawback of such approaches is the initial creation and definition of the concepts that constitute the software and to model them in an ontology. On the other hand they allow for traceability detection in the context of heterogeneous and possibly inconsistent software artifacts.

Advantages	Disadvantages
(+) Multiperspective dependency detection	(-) Creation of ontology causes effort
(+) Can determine types of dependencies	(-) Maintenance of ontology causes effort
(+) Can resolve inconsistencies	

Table 3.5.: Summary of Semantic Wikis for multiperspective dependency detection

3.3.4.5. Machine Learning (ML)

Machine learning approaches apply algorithms that are able to automatically "learn" traceability links from software artifacts based on a given training set of dependency relations. These training sets are either supplied by developers [SdGZ03] or they are comprised of a combination of manually established links and links identified by program analysis and runtime-monitoring [GMP07], dependent on the level of granularity. If the training sets contain traceability links connecting heterogeneous software artifacts, the approaches are potentially able to elicit such links as well. Moreover, these approaches are potentially able to distinguish between different types of relations, if the types are reflected by their training set. However, case studies indicate that the precision of the obtained results highly varies [GMP07].

Advantages	Disadvantages
(+) Multiperspective dependency detection	(-) Requires training set
(+) Can determine types of dependencies	(-) Unstable results

Table 3.6.: Summary of ML approaches for multiperspective dependency detection

3.4. Change Operations

Understanding changes is an essential part of change impact analysis, since understanding the types of changes is important for determining their effects. In this section we therefore investigate how change operations can be modeled and classified to allow for automated change impact analysis. The following discussions are based on a literature survey on the modeling and classification of changes in the fields of impact analysis and regression testing [LFR12].

3.4.1. Modeling of Change Operations

A change operation, in short "a change", transforms a software system or one of its constituents from version n to version $n + 1$. In order to describe and model changes, the following information are required: the element that should be changed, a description of the change activity, and a description of the element after the change.

For describing such change activities, Fluri and Gall [FG06] introduced the concept of basic (or atomic) changes. As the name suggests, atomic changes describe change operations that cannot be further refined or broken down to other changes. Fluri and Gall list *add*, *delete*, *modification*, and *move* as their set of basic change operations.

This concept was extended by Mäder *et al.* [MRP06b, MÏ0] who introduced the notion of composite operations that are comprised of sequences of other atomic operations. Mäder *et al.* propose the same atomic operations as Fluri and Gall except for the *move*-operation which they consider a composite operation.

The work of Robbes [Rob08] also distinguishes between atomic and composite changes, where the latter may also consist of sequences of atomic changes. Moreover, composite operations are further subdivided into developer-level actions (behavior-modifying changes) and refactorings (behavior-preserving changes [Fow99]). As his basic units of change Robbes proposes *Creation*, *Deletion*, *Destruction*, *Addition*, *Removal*, *Insertion*, and *Change Property* operations.

In conclusion, the concept of atomic and composite operations is most suitable for providing a solid base for the modeling of change operations for impact analysis tasks. The concept allows for combining existing atomic operations into composite operations, which can be further nested if necessary. However, the current concepts must be extended to allow for composite operations also being modeled as sequences of other composite operations. Secondly, the set of atomic operations should be kept as small and unambiguous as possible, which is why we have to revise the sets of atomic operations presented by the discussed works.

3.4.2. Classification of Change Operations

If the impact of a change shall be analyzed, it is first of all necessary to understand the actual change. Understanding the changes can be assisted by means for classifying changes according their type, scope, purpose, etc. As a result of our literature review [LFR12], we identified different approaches for the classification of change operations which we are discussing in the following. The reviewed studies encompass approaches proposed for change impact analysis [KGHW94, LO96, RT01, FM06, GSC10, SLT$^+$10], regression testing [RST$^+$03, RST$^+$04, SRRT06], software design [Fow99, BC00], traceability maintenance [MRP06b, MÏ0], change coupling analysis [XS04a,XS04b,XS05,FG06], and requirements management [MG09,MG11]. Overall, we identified two groups of approaches for the classification of changes:

1. Classification based on how the changes can be modeled.
2. Classification based on the purpose of the changes.

The classifications proposed by Fluri and Gall [FG06] and Mäder *et al.* [MRP06b, MÏ0] are based on the approach that is used to actually model the changes, e.g. the concept of atomic and composite changes as discussed in the previous section. Thus, both classifications only reflect the structure of the changes while they are neglecting, for example, their scopes or purposes.

In contrast, Gupta *et al.* [GSC10] propose to classify changes according to their purpose. Therefore, the authors distinguish between changes that either affect the functionality, behavior, structure or the logic of a software system. This approach, however, is ambiguous in regard to the purposes of changes. *Deleting* an entire component from a software system, for example, alters its structure as well as its functionality and behavior, since all the functionality provided by the component is removed as well. Hence, we do not consider this type of classification as applicable for research and practice due to its ambiguousness.

The vast majority of the studies that were analyzed during our review, however, did at best list the change operations that are supported by their approaches. Yet, they did not introduce any systematic means for classifying and modeling the changes, which is why the proposed changes are overlapping or inconsistent for the most part. Hence, a systematic classification of changes is required to allow for comparing change impact analysis approaches according to the change operations supported by them. Yet, such a classification is still missing in related work but required for performing change impact analysis.

3.5. Open Research Issues

Based on the analyses presented in the previous sections, we identified a set of open problems that are directly related to the scope and the goals of this thesis and thus require further research.

P5 - Integrating Heterogeneous Software Artifacts. Approaches for multiperspective modeling and consistency checking advocate the need for integrating and unifying the heterogeneous software artifacts prior to any subsequent analyses. However, different concepts were proposed for this purpose, where the concept of utilizing a common meta-model for bridging between the different types of software artifacts seems to be most promising.

P6 - Recording Dependency Relations. Finding a suitable data structure and concept for explicitly recording the various dependencies of the heterogeneous software artifacts is an important precondition for conducting change impact analysis. Additionally, multiperspective impact analysis requires a unified treatment of the different types of dependency relations as well.

P7 - Multiperspective Dependency Detection. Moreover, it is necessary to establish a suitable technique for multiperspective dependency detection. Various approaches for traceability detection were discussed that can potentially be reused to accomplish this. However, a suitable approach must be found that allows for multiperspective dependency detection according to the goals of this thesis

P8 - Dependency Classification. Our review of related work on software dependencies revealed the need for a thorough classification of dependencies. Currently, many different classifications have been proposed in the literature that are either inconsistent or overlap with each other. Furthermore, not all of the required dependency types are covered by current classifications. This issues therefore corresponds to problem **P3** as already discussed in Section 2.5.

P9 - Change Classification. Finally, our analysis of related work regarding the modeling and classification of change operations revealed the need for a consistent classification of change types and a practical way of modeling change operations. In contrast, the classifications that are proposed in the current literature are either inconsistent to each other or contain too many overlappings and redundancies. Thus, an appropriate scheme for the modeling of change operations is required to provide adequate support for change impact analysis tasks. This issue therefore corresponds to problem **P2** as already discussed in Section 2.5.

3.6. Summary

In this chapter we reviewed related work that builds the foundations for research on change impact analysis. First, we explored the concept of views to identify the views on software that are most relevant for this work. Secondly, we studied approaches dealing with the modeling and consistency management of multiperspective software systems due to their tight coupling with multiperspective change impact analysis. Consequently, typical dependencies of heterogeneous software artifacts were studied in regard to their purposes and in regard to how they are classified by existing works since they carry the effects of changes. We then outlined the similarities between the concepts of traceability links and dependency relations and analyzed various techniques that were proposed for traceability detection with the goal of reusing the latter to support change impact analysis. We also analyzed current works on software maintenance in regard to how they deal with different types of change operations. Finally, we summarized open research issues that are also related to the scope and the goals of this thesis.

4. Overview of the Approach

In this chapter we provide the big picture of the approach that is presented in this thesis. To begin with, we introduce our main research hypothesis that was formulated based on our analyses of related work according to the goals of the thesis at hand and own experiences from various software development projects. Secondly, the research goals of this thesis are refined according to conclusions drawn from related work studied in the previous two chapters. Finally, an overview of the proposed approach is given that elaborates on its main steps and assumptions.

4.1. Research Hypothesis

The central concern of this thesis is to identify *if* and *how* a change that is applied on a certain software artifact affects the software artifacts that are dependent on it, and how these questions can be answered when different types of software artifacts belonging to different perspectives and development phases are involved. As acknowledged by related work and confirmed by own observations from several software development projects, dependency relations between software artifacts carry the effects of changes [Han96, Boh02a]. However, we also observed that not every type of dependency relation carries the effects of every type of change from one software artifact to others. Therefore, the following initial hypothesis was developed and acted as the central driver for the research presented in this thesis.

> ***Initial Hypothesis:*** The impact of a change can be determined by inspecting the dependency relations that connect the software artifact to be changed with other software artifacts under consideration of the type of the change. The interplay of change type and dependency type determines further change propagation.

According to our hypothesis, the impacts of a change on a software system can be determined by analyzing the interplay of the following information:

- The type of the software artifact to be changed.
- The type of the dependent software artifact (i.e. the possibly impacted one).
- The type of the dependency relation between both.
- The type of the change to be applied.

Furthermore, we believe that the above stated interplay can be analyzed by a rule-based concept, which, when applied prior to the implementation of a change, can determine its impact on the software system. Consequently, our research hypothesis has been further refined as follows.

> ***Revised Hypothesis:*** The impact of a change can be determined by inspecting the dependency relations that connect the software artifact to be changed with other software artifacts under consideration of the type of the change. The interplay of

change type, dependency type, and artifact type determines further change propagation. This interplay can be analyzed by impact propagation rules that can be utilized for determining the impact of a change prior to its implementation.

The concept that emerges from our refined hypothesis is realized by the approach presented in Chapters 5 to 8, and the prototype tool presented in Chapter 9. Finally, our hypothesis is evaluated by a comprehensive case study discussed in Chapter 10.

4.2. Refined Goals

In Section 2.5 and Section 3.5 we summarized the problems and limitations of current approaches for impact analysis, heterogeneous modeling, modeling of change operations, and dependency detection and classification in regard to our goals. To address these issues, we refine and revise our research goals as introduced in Section 1.2 under consideration of our research hypothesis as presented in the previous section and the findings of our literature reviews.

Based on our research hypothesis and our analysis of current impact analysis approaches (see Section 2.3), we propose to utilize a rule-based concept for analyzing the interplay of changes and dependencies. A rule-based concept can be applied on heterogeneous artifacts, is extensible to address new types of software artifacts, and allows to further process the detected impacts. However, our rule-based concept will be substantially different when compared to existing approaches as it will be based on a different propagation strategy (i.e. the analysis of the interplay of changes, dependencies, and software artifacts). We can therefore refine **Goal 1** as follows.

Goal 1: Develop a rule-based change impact analysis approach that provides a high reliability when assessing the effects of changes prior to their implementation. The approach should identify all artifacts impacted by a change and determine as few false-positives as possible. Various experiments shall be conducted to evaluate its performance and efficiency.

To address the main limitation of existing rule-based approaches, namely the creation and validation of impact rules, we have to define another goal which this thesis has to meet.

Goal 5: Provide a methodology for the creation of additional impact rules to extend the approach and to provide support for change impact analysis of new types of software artifacts not yet covered by thesis.

According to our analysis of existing impact analysis approaches and the fact that support for multiperspective impact analysis is still missing (see Section 2.5), **Goal 2** remains unchanged. Moreover, the few available multiperspective impact analysis approaches are limited in regard to their support of heterogeneous artifacts and are only able to assess the change propagation between two types of software artifacts (see Section 2.4).

Goal 2: Develop a rule-based change impact analysis approach that enables the analysis of different types of software artifacts to keep them synchronized and consistent to each other. The approach to be developed shall at least support the following types of artifacts: architectural models (design view), source code (code view), and test cases (test view).

As revealed by our literature review, only very few impact analysis approaches are able to determine how a software artifact is impacted by changes (see Section 2.5). Consequently, developers have to further inspect the artifact to determine how it is affected, which causes an additional manual overhead. Thus, **Goal 3** remains unchanged as it addresses this problem.

Goal 3: Develop a rule-based change impact analysis approach that allows developers to understand how and why software artifacts are impacted by a certain change. The approach should be applicable during forward engineering, reengineering, and software maintenance in general.

After analyzing how changes are modeled and how the different types of changes are classified by related work, we have to refine **Goal 4** to support our impact analysis approach with a thorough concept for the modeling and classification of change operations.

Goal 4: Develop a meta-model for change operations that allows for the modeling of arbitrary changes to study their impacts on software artifacts. Furthermore, it is necessary to establish a taxonomy of change operations to classify changes according to their type of operation.

Additionally, to enable multiperspective change impact analysis it is necessary to bridge the gap between the heterogeneous types of software artifacts. In order to be able to apply the same impact analysis concept on all types of software artifacts, means must be sought to address them in a homogeneous manner. To accomplish this, the concept of utilizing a common representation for all types of software artifacts as discussed in Section 3.2 seems most promising.

Goal 6: Find or develop a suitable common meta-model for all the software artifacts addressed by this thesis and the required means for mapping them on this meta-model if necessary.

Based on our analysis of existing multiperspective approaches, we have to derive two additional goals to adequately address the dependency relations connecting the heterogeneous software artifacts that are responsible for propagating the effects of changes between them.

Goal 7: Find or develop a suitable meta-model to explicitly model different types of dependency relations. This model should also be accompanied by suitable means for determining the types of the dependency relations for the later change impact analysis.

Goal 8: Extend or develop a suitable multiperspective dependency detection technique as a precondition for the later change impact analysis.

4.3. Proposed Approach

We now present a general overview of the change impact analysis approach that is proposed in this thesis to put the upcoming discussions into a broader context.

The approach is divided into the two phases of *preprocessing* and *change impact analysis*, whereas each phase is further comprised of two distinct steps. The preprocessing phase is responsible for providing unified access to the various software artifacts that constitute a software system (step 1), such as source code files or UML diagrams, and to explicitly record their dependencies for the later change impact analysis (step 2). While recording the dependencies, our approach is able to address inter- and intra-artifact dependencies alike and to explicitly determine their types. The 2nd phase starts with the classification of the change operation that is about to be applied on the software system (step 3), which then triggers the actual change impact analysis to assess the consequences of this change (step 4). The classification of changes must be accomplished by the developer who is applying our approach, whereas the impact analysis is accomplished in an automated manner. This overview is also illustrated by Figure 4.1.

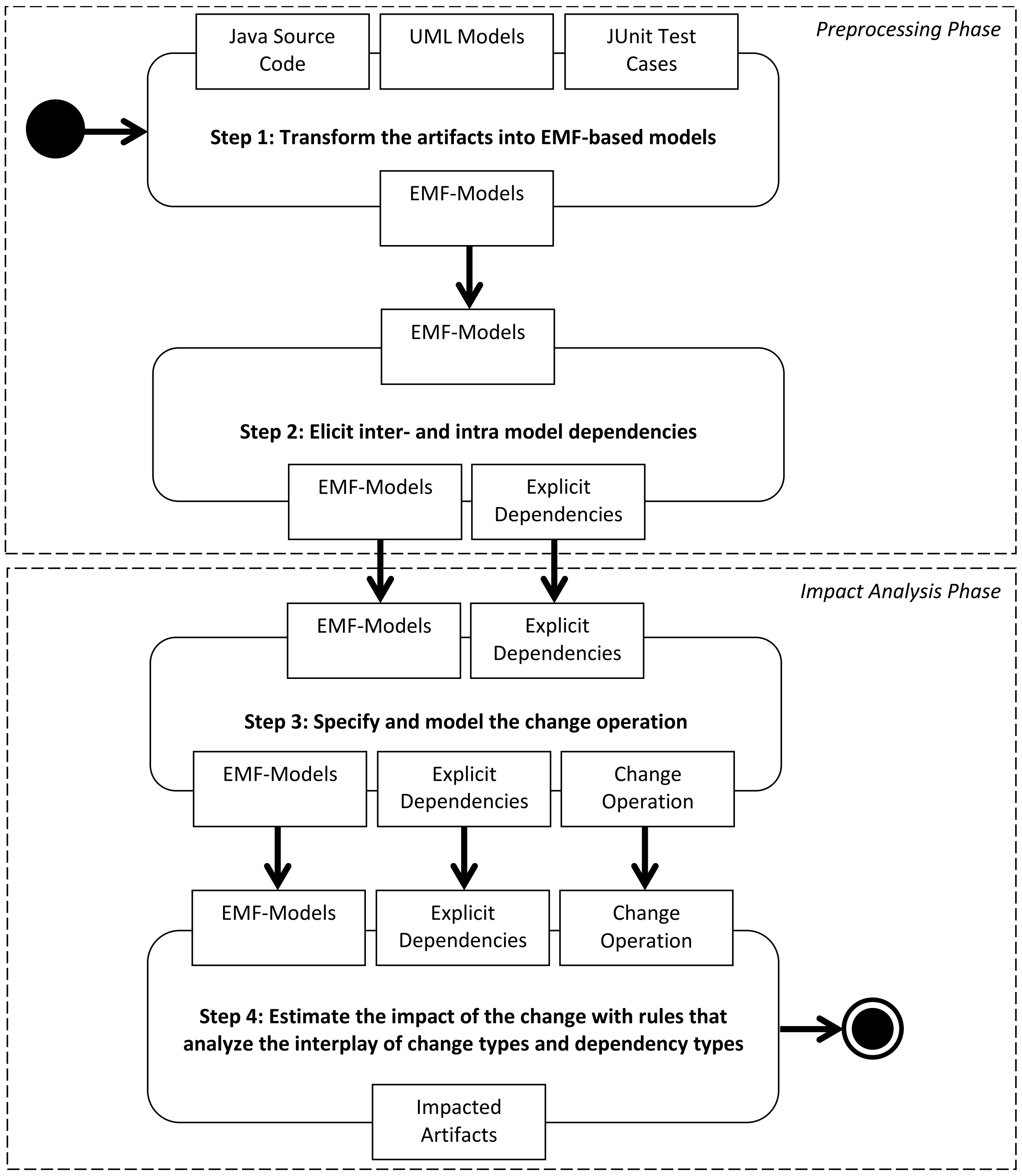


Figure 4.1.: Overview of the proposed approach

Step 1: Artifact Integration. We bridge the gap between the different types of software artifacts by mapping them on a unifying meta-model supplied by the Eclipse Modeling Framework (EMF) [EMFa] to allow for a unified access to the software artifacts. The unified software artifacts are then imported into a model-repository which provides the basis for the introduced multiperspective impact analysis approach. Chapter 5 elaborates on this step.

Step 2: Dependency Detection and Recording. Potential dependency relations connecting the heterogeneous software artifacts are elicited by applying an automated traceability mining approach on the artifacts stored in the model-repository. To accomplish this, a rule-based detection approach is applied that utilizes a set of pre-defined rules for dependency detection. These detection rules are able to determine the types of the dependency relations and to record the identified relations as traceability links. The classification of the dependencies is accomplished by a taxonomy of dependency types introduced in this thesis. Chapter 6 elaborates on this step.

Step 3: Change Type Classification. As a first step of the actual change impact analysis, the developer who is changing the system has to specify the artifact(s) to be changed. Secondly, the changes to be applied on them must be classified according to their type, for which the developer is supported by a catalog of change operations that allows him choosing the right type. This catalog of change operations is based on a novel taxonomy and modeling approach for change operations that are both introduced in this thesis. Chapter 7 elaborates on this step.

Step 4: Change Impact Analysis. According to our research hypothesis, the impact of a change can be determined by analyzing the dependency relations of software artifacts according to the type of the change operation that is applied upon them. We accomplish this by applying a rule-based concept to analyze this interplay and to determine further change propagation. Once a change is classified and the impact analysis is triggered, a set of impact propagation rules is applied which require as input the type of the change and the changed artifact. Our approach then recursively explores the artifacts that are related to the changed one by analyzing the dependency relations between them. Based on the types of the dependency relations and changes, potential impacts are determined by the impact propagation rules. If an impact is detected, it is feed back into the recursive analysis process where it may trigger the execution of further rules. Thus, the impact propagation rules are recursively applied until no further change propagation is triggered by them. After the recursive propagation has come to an end, all impacted elements will be reported to the developer. Chapter 8 elaborates on this step.

4.3.1. Handling of Problem Space and Solution Space

In this section we discuss whether the proposed approach is situated on either the problem space, the solution space or whether it covers both. The problem space generally denotes a variety of factors surrounding a software system, in particular its requirements, constraints, and human factors [CE00]. In contrast, the solution space is comprised of entities that contribute towards solving the requirements of the software system and problem space, which can be, for example, architectural decisions, COTS components, design patterns, and algorithms [CE00].

The research presented in this thesis is solely focused on the solution space, while the problem space is entirely neglected for two reasons. First and foremost, the interconnection between both worlds is still subject to ongoing research. Although many approaches are focused on bridging this gap, e.g. [HNS05] and [Bod11], there is still no comprehensive approach available to accomplish this linking, which in turn would shift the focus of this thesis away from understanding change impact analysis towards problem-solution-mapping. Secondly, this thesis is aimed upon research on impact analysis with the goal of supporting software maintenance and reengineering. These two tasks, however, are mainly focused on existing software artifacts stemming from the solution space, such as source code, software architectures, and test cases. Thus, we more or less concentrate on them only.

4.3.2. Covered Views

In this thesis we propose an impact analysis approach that can be applied on artifacts stemming from the structural view, the behavioral view, the code view, and the test view. These views were chosen because they are inherent to any software development project and are independent of the applied development methodology and technology (see discussions in Section 3.1).

The structural view describes the components and relations a system is comprised of, and is typically modeled by UML component, class, deployment, and package diagrams. The behavioral view describes the services and operations provided by a system, internal communication paths, data exchange, and to a certain extent, user interaction. Hence, UML sequence, activity, and use case diagrams are considered as relevant for this view. The code view represents the actual implementation of the system and consists of Java source code entities such as classes, interfaces, methods, and data types. Finally, the test view consists of similar artifacts as the code view, whereas its purpose is to test the functionality implemented in the code view.

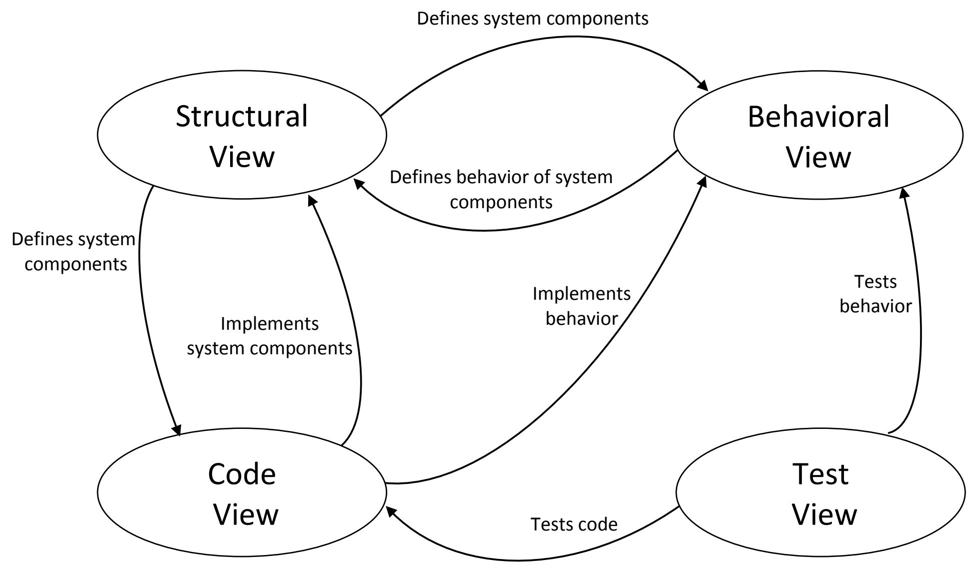

Defines system components
Structural View
Behavioral View
Defines behavior of system components
Defines system components
Implements system components
Implements behavior
Tests behavior
Code View
Test View
Tests code

Figure 4.2.: Views considered by our approach (ellipses) and their relations (arrows)

Figure 4.2 summarizes the four views considered in this thesis along with their general dependencies (directed arrows). Arrows pointing from one view to another view indicate the role that artifacts of the one view play for artifacts of the related view. For example, the relation "Defines System Components" connecting the *Structural View* and the *Behavioral View* denotes that the *Behavioral View* can only describe the behavior of those artifacts (e.g. system components) that are defined in the *Structural View* and vice versa.

4.3.3. Assumptions

The multiperspective change impact analysis approach presented in this thesis is build upon the following set of assumptions. Later on, we discuss the influence of these assumptions on the feasibility and applicability of the proposed approach.

Assumption 1 - Software is developed using Object Oriented concepts. The majority of the concepts presented in this thesis assumes that object oriented software systems shall be analyzed and object oriented concepts are applied. This, however, can be taken as granted considering today's software and the software systems constructed during the last two decades.

Assumption 2 - The architecture of software systems is realized by UML models. Although the general concepts presented in this thesis are not tailored to certain types of software artifacts, all the impact propagation rules developed for this thesis, as well as all the examples utilized throughout this thesis, assume an UML-based representation of software architectures (e.g. by UML class and component diagrams).

Assumption 3 - The architecture and the source code of software systems are available to developers. The presented approach assumes that both the architecture and the source code of a software system are available to developers in their native form and that both are freely accessible, e.g. that the source code is not only provided as byte code or compiled binary files.

Assumption 4 - The software artifacts to be analyzed are consistent to each other. In order for the change impact analysis to predict correct results, the artifacts to be analyzed must be in a consistent state. For example, the modeled architecture and the implemented architecture must both be consistent to each other in terms of applied concepts and architectural entities. They may, however, possess a different level of abstraction. Hence, the change impact analysis requires a solid baseline to embrace future changes, which is in line with, for example, all regression testing approaches that also require a fixed baseline test suite [FLR14].

Assumption 5 - Unified access to all artifacts is required. The application of impact propagation rules on the software artifacts and their dependencies demands for a homogenous access to all the software artifacts. All the heterogeneous artifacts must be analyzable using the same approach, as otherwise a generalizable rule-based concept cannot be developed and applied.

Assumption 6 - Dependencies must be made explicit. Dependency relations, either between or contained within the heterogeneous software artifacts, must be made explicit and accessible for the change impact analysis approach in a homogenous way. Furthermore, it is necessary that the type of each dependency relation is explicitly known to the change impact analysis approach.

4.4. Summary

In this chapter we presented the research hypothesis of this thesis that is based on our initial research questions and the results of our reviews of related work in Chapters 2 and 3. Subsequently, the goals of the thesis were refined according to the results of our analysis of related work and the established research hypothesis. Moreover, an overview of the proposed approach was given that summarized its core concepts and steps to put the discussions in the upcoming chapters into a broader context. Finally, the assumptions of the approach were laid out and the views and perspectives that are considered by our approach were presented.

5. Comprehensive Artifact Integration

Multiperspective change impact analysis requires a unified treatment of the different types of software artifacts if one and the same impact analysis approach shall be applied on them (see **Goal 2**). To accomplish this, in this chapter we introduce an approach for integrating various different types of software artifacts. We therefore explain how the heterogeneous artifacts can be mapped on a unifying representation to accomplish **Goal 6**. The upcoming discussions are based on the findings of our state-of-the-art review regarding multiperspective approaches in Section 3.2, our works on multiperspective dependency detection [Leh10, BLR11, RBFL11, FLR14], and our works on multiperspective impact analysis [LFR12, LR12, LFR13].

This chapter is organized as follows. In Section 5.1 we compare two approaches how heterogeneous software artifacts can be integrated and prepared for multiperspective change impact analysis. Based on our findings we then introduce our integration approach in Section 5.2. We perform a final critical review of our approach and outline further extensions in Section 5.3.

5.1. Comparing Integration Techniques

In order to be able to apply the same impact analysis approach independent of the artifacts to be analyzed, appropriate means are required to bridge the gap between the different types of artifacts. The analysis of related work in Section 3.2 brought up the idea of mapping the heterogeneous artifacts on a common model to enable a unified treatment of them. In the following, we review and compare two possible options that are currently proposed in related work to accomplish this mapping. Afterwards, we introduce our mapping approach which is based on the outcome of this comparison under consideration of our goals and our previous works.

5.1.1. Mega-Models and Model Weaving

An approach that has attracted quite some attention in the model-driven engineering (MDE) community is the concept of mega-models. Current software development practice has to deal with software artifacts that are instances of a variety of meta-models, such as the UML meta-model or certain requirements definition languages, such as URN [ITU08] or ReqIF [OMG11] for instance. Instead of dealing with this variety of different meta-models, the concept of mega-models proposes to merge them into one unified mega-model using the concepts of model weaving and model transformations [BJV04, FN05, RKRS05, SNG10]. After joining the meta-models, their dependencies can be modeled explicitly and integrated into the mega-model as well [SNG10]. Figure 5.1 illustrates an exemplary mega-model.

The main advantage of this approach results from the usage of only one model that encompasses all the required artifacts along with their dependencies. This direct integration of dependencies and their treatment as first-class citizens next to the actual models embraces the utilization of the dependencies for tasks such as impact analysis or model comprehension.

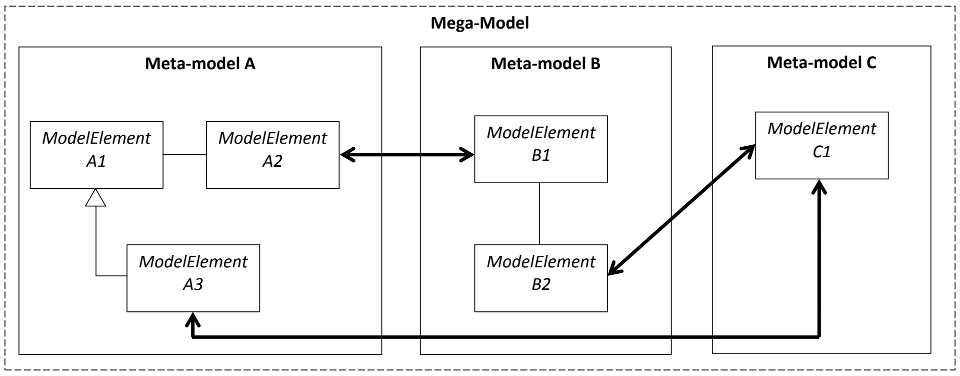


Figure 5.1.: The mega-model approach. Thick arrows indicate cross-model dependencies

On the other hand, a mega-model requires sophisticated tool support to be of use in practice. However, various different tools are being used to create and maintain the different types of artifacts, whereas adapting all the involved tools is either impossible or simply not feasible in practice. Furthermore, such a combined mega-model is also prone to changes, as it has to be adapted to changes of the meta-models that were weaved into it. Since modeling languages, such as UML or BPMN, and programming languages, such as Java or C++, constantly evolve, their integration into a mega-model could result in increased effort when maintaining the model.

The strengths and weaknesses of such approaches are summarized by Table 5.1.

Strengths	Weaknesses
(+) Everything in one model	(-) Lack of tool support
(+) Dependencies explicitly integrated	(-) Maintenance of model
	(-) Complexity of model

Table 5.1.: Summary of the mega-model approach

5.1.2. Combining Modeling Frameworks and Model Repositories

Instead of merging all artifacts into one comprehensive model, the software artifacts can also be mapped on a common low-level model which allows to represent all artifacts in a homogenous way [YCDW09] (see also the discussions in sections 3.2.1 and 3.2.2). Once mapped on such a common representation, the different software artifacts can be accessed and analyzed by the same impact analysis approach. The required common low-level model can be supplied by modeling frameworks that provide a generic graph-based meta-model, such as the Eclipse Modeling Framework (EMF) [EMFa] for example. Since most software artifacts adhere to a hierarchical, graph-based structure [DST11,LFR12], such as UML diagrams or the AST of any programming language, they can easily be mapped on a graph-based meta-model.

The utilization of such an approach requires means to map the heterogeneous artifacts on the common model, such as model transformation languages like XSLT [W3C07] or ATL [ATL] for instance. Hence, this mapping causes an initial manual effort as convenient transformation rules

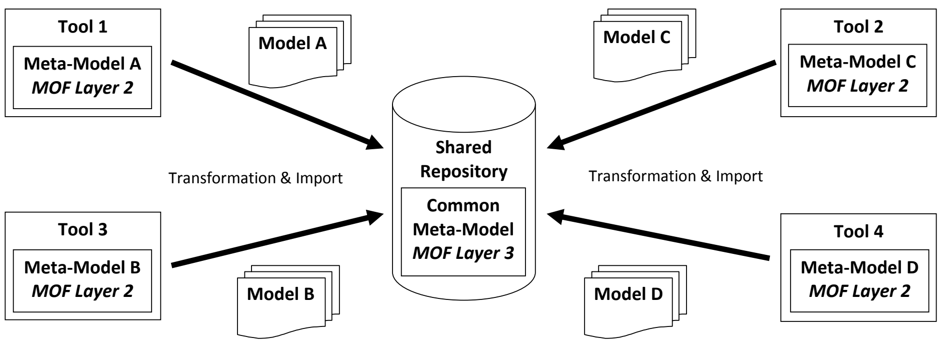


Figure 5.2.: The joint model repository approach. Arrows indicate the transformation and import of software artifacts into the repository

must be supplied. Additionally, a central model repository is required to store the transformed artifacts as they can no longer be maintained in their original tools. However, they can now be accessed and queried through the same means, such as model query languages like EMF Query [EMFb] for example. Furthermore, all the change impact analysis infrastructure can be integrated into the repository as well. Figure 5.2 visualizes this concept.

The major disadvantage of this approach is that it requires effort for synchronizing the models inside the repository with the original artifacts stored in other tools. However, most tools offer APIs or services for recording and exchanging change events which can be forwarded to the model repository to update the transformed artifacts. On the other hand, this approach allows to reuse existing IDEs, CASE tools, etc. as they are still used for editing the artifacts.

The strengths and weaknesses of such approaches are summarized by Table 5.2.

Strengths	**Weaknesses**
(+) Simple integration of new artifacts	(-) Requires additional repository
(+) Impact logic in one place	(-) Requires re-synchronization with tools
(+) Changes in tools only affect transformations	(-) Requires transformation of artifacts
(+) Meta-model changes only affect transformations	(-) Transformations require manual effort

Table 5.2.: Summary of combining modeling frameworks and model repositories

5.2. Integration Approach

Based on the findings of the previous section, we extend our existing works on the integration of different modeling languages through a unifying modeling framework. The approach was originally developed for interconnecting UML, URN, and OWL models using the Eclipse Modeling Framework (EMF) [EMFa] and a centralized repository for storing the transformed artifacts [Leh10, BLR11, RBFL11]. During the course of this thesis' research, the approach was extended to encompass Java source code and JUnit test cases [LFR12, LFR13], as well as BPMN models [FLR14]. The overall approach works as follows. In a first step, the software artifacts maintained by different tools are transformed into an EMF-based representation. This mapping is supplied by model transformations that are realized by different means, dependent on the artifacts to be transformed. Secondly, all the transformed software artifacts are then imported into a central model repository that utilizes the common EMF meta-model for managing the various types of software artifacts. This mapping and transformation of software artifacts into EMF-based models is explained in Section 5.2.1, while Section 5.2.2 explains the concept and usage of a central model repository. Moreover, Figure 5.3 summarizes our entire concept.

We prefer this approach over the concept of mega-models and model weaving as the lack of tool support is an important limitation which cannot be neglected due to its influence on the applicability of the approach. Furthermore, we circumvent the problem of updating the inter-weaved meta-models. In the case of meta-model changes, our approach only requires the adaptation of its transformation rules. This, however, also applies to the mega-model approach where the transformations must be adjusted likewise, hence causing greater effort. However, we like to point out that the impact analysis concept presented in this thesis is not hard-wired to one of the concepts discussed in the previous sections. It can be build on any of the presented integration techniques, whereas the decision for a certain integration technique might be influenced by the amount and types of artifacts to be interconnected, the available budget and developers, etc.

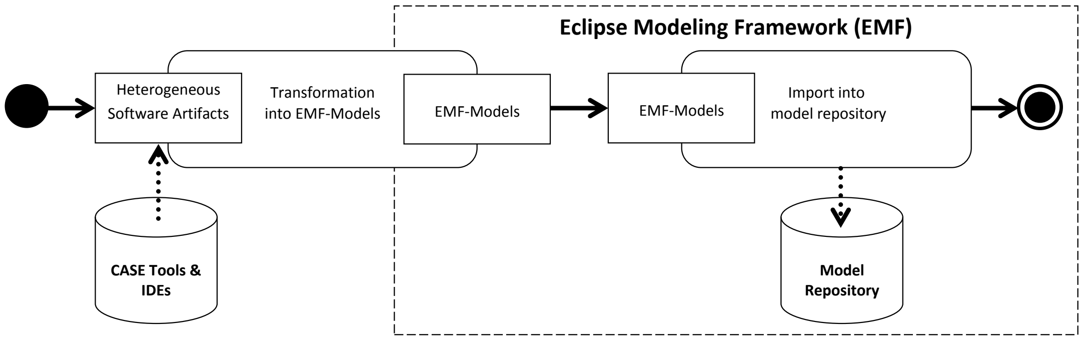

Eclipse Modeling Framework (EMF)
Heterogeneous Software Artifacts
Transformation into EMF-Models
EMF-Models
EMF-Models
Import into model repository
CASE Tools & IDEs
Model Repository

Figure 5.3.: Overview of our artifact integration approach

5.2.1. Transformation into EMF-models

Mapping the heterogeneous software artifacts on a common meta-model demands for their transformation into a graph-based representation. In [LFR12] we already outlined that the different types of software artifacts can be mapped on a graph-based representation $G = (V, E)$, where software artifacts, such as the UML components or Java classes, are inserted as nodes (V) and dependencies between them, such as inheritance-relations, are inserted as edges (E). In our approach, this graph-based meta-model is supplied by the *Eclipse Modeling Framework* (EMF) [EMFa] that provides the *Ecore* meta-model. The *Ecore* meta-model is the core component of the framework and offers the *EClass*-object to model arbitrary types of entities, while relations between those entities can be modeled using the *EReference*-class. The entire Ecore meta-model and all its constituents are displayed by Figure 5.4 which is taken from the project's API documentation[1]. Since the *Ecore* meta-model adheres to the MOF specification [OMG14],

[1]http://download.eclipse.org/modeling/emf/emf/javadoc/2.8.0/org/eclipse/emf/ecore/package-summary.html

all *Ecore*-models are by definition hierarchical graphs. Consequently, the mapping between *Ecore* model elements and the nodes and edges of a graph is realized as follows [LFR12]:

- **Nodes (V):** EPackage, ECLass, EDataType, EEnum, EAnnotation, EOperation, EParameter, EAttribute, EEnumLiteral.
- **Edges (E):** EReference, EInheritance, EAnnotationLink.

By using EMF it is therefore possible to recreate the meta-models of all the artifacts to be integrated based on *EClass*-objects and the other constituents of the Ecore-model. Hence, EMF allows to express all types of artifacts with only one underlying meta-model. By default, EMF-based meta-models for many modeling languages are already provided by the Eclipse Modeling Tools project [EMP], such as EMF-based UML and BPMN meta-models for example. Additionally, an EMF-based meta-model of the Java language specification was made available by the MoDisco project [Mod], which can be reused for our purposes. In regard to **Goal 2** this allows us to cover architectural models (UML), source code (Java), and test cases (JUnit).

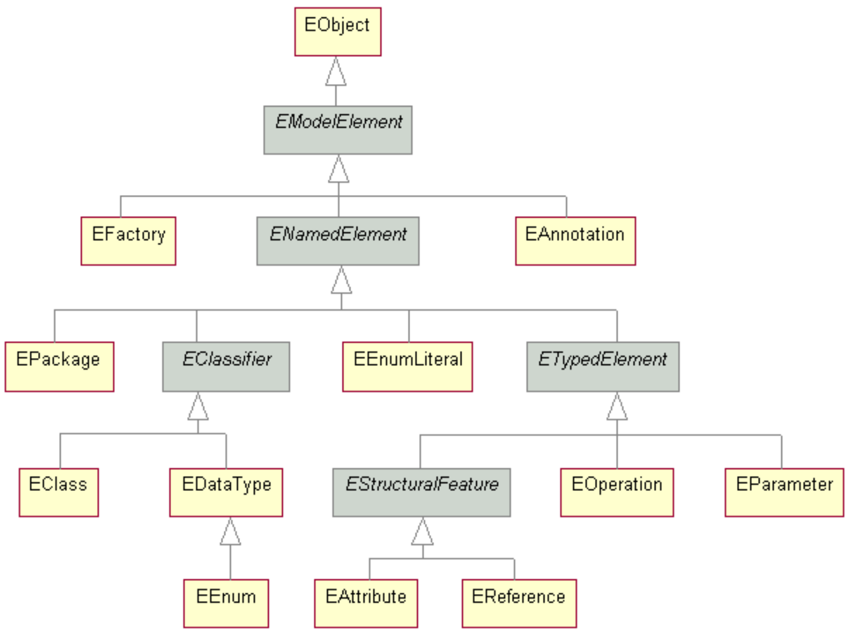


Figure 5.4.: The constituents of the Ecore meta-model

The conversion of the original artifacts into EMF-based models is accomplished by graph transformations using a specific set of transformation rules for each of the addressed modeling languages. These transformation rules are realized by XSLT-templates that were created in one of our previous works [Leh10]. They were published under the Eclipse Public License [EPL] and are available for download on our project website [EMF14]. Additional XSLT-templates for mapping URN and OWL models to EMF-models are also available; yet, these modeling languages are out of the scope of this thesis' research.

Further mapping support for transforming Java source code into EMF-models is supplied by the MoDisco [Mod] tool environment, which is able to transform entire Java-projects into EMF-models. For the details of this mapping process, we refer to the documentation of the MoDisco tool suite [Mod].

5.2.2. Integration into an EMF-based Model-Repository

The EMF-models resulting from the model transformations require a centralized storage from where they can be accessed by the later impact analysis as they can no longer be maintained by their original tools. Consequently, our approach demands for an EMF-based model repository that complements the integration of the heterogeneous software artifacts. Once the artifacts are converted into EMF-models they are imported into the repository, which then stores an EMF-based copy of them. Finally, with all the different artifacts in place, it is possible to study the propagation of changes and their impacts on the heterogeneous software artifacts.

The selection of a suitable repository was comprised of two steps: searching for available tools and comparing them based on well-defined criteria derived from our research goals. The criteria that were used for evaluating the repositories discovered by our search were as follows [Leh10]:

- **REQ1: Maturity** - how mature is the tool to apply it in an academic environment?
- **REQ2: Provided Features** - which features are already provided by the tool?
- **REQ3: Documentation** - how well is the tool documented? Are tutorials and demos supplied to guide further extensions?
- **REQ4: Extensibility** - is it possible to extend the tool with custom features and how much effort is required to accomplish these extensions?

The initial search returned ten different repositories out of which three were examined in detail: EMFStore [EMFc], CDO [CDO], and the Eclipse Model Repository [Ecl10]. As a result of this analysis, the EMFStore repository was chosen due to its well-documented features, continuous developer support, good extensibility, and ongoing development [Leh10]. Further details regarding the repository are presented along with our prototype case tool in Chapter 9.

5.3. Critical Discussion and Limitations

Our approach of mapping the heterogeneous software artifacts on a common meta-model and importing them into a model repository gives rise to several issues which we are going to discuss in the following.

The chosen approach of transforming the heterogeneous software artifacts into EMF-models introduces a certain overhead to the change impact analysis processes, and results in the need for a unifying model repository for storing the transformed artifacts. However, if no unified representation of the artifacts is employed, the change impact mechanisms to be developed must be specifically tailored for each type of software artifact. In contrast, a unified representation demands for only one change impact analysis approach and thus requires less overall effort.

The most obvious limitation of our approach is that the repository stores duplicates of the artifacts that are not updated when developers modify the original artifacts in their original tools. As a consequence of this, the model repository approach requires additional and potentially manual effort for synchronizing its data with the tools that are used for creating and maintaining the original software artifacts. A possible solution could be to connect those tools with the repository using special adapters or plug-ins which could then forward change events and updates between the repository and the tools as illustrated by Figure 5.5. With the help of this extension, the impact logic could still remain as a centralized component in the repository, while it would involve less (manual) effort for synchronizing the artifacts after changes were applied on them.

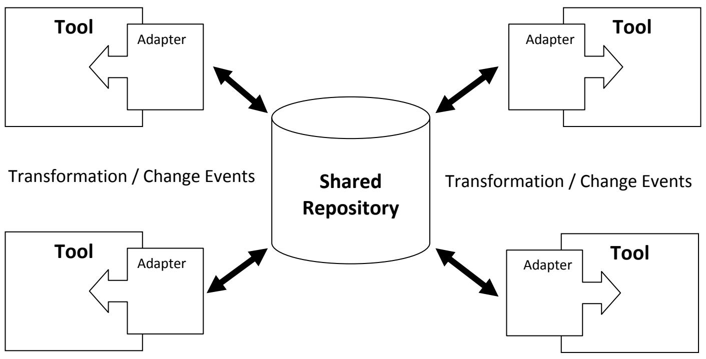


Figure 5.5.: Enhancing the repository approach with adapters forwarding change events

Another limitation of our approach is the effort that is required for updating the model transformations in the case of meta-model changes, e.g. when a language specification is updated to a new revision. A similar but yet more frequent problem arises when CASE tool vendors update the internal meta-models of their CASE tools, which in turn may alters their output. For example, almost every UML modeling tool has its own specifics that are not necessarily reflected by the OMG standard for UML and hence must be considered by the model transformations. Unfortunately, this problem cannot be addressed in an academic way and will always remain a challenge for any formal approach relying on the concepts of model transformation and model integration.

5.4. Summary

In this chapter we addressed the problem of integrating heterogeneous types of software artifacts for multiperspective change impact analysis. We proposed an approach to map the software artifacts under consideration on a common meta-model provided by the Eclipse Modeling Framework to accomplish **Goal 6** and to contribute towards **Goal 2**. We explained how the software artifacts are mapped to EMF-models and how they are integrated into a unifying EMF-based model repository that later on serves as the starting point for the impact analysis approach proposed in this thesis. Moreover, we also discussed the shortcomings of our approach resulting from the usage of duplicated data stored in the external repository, and explained a potential solution to mitigate its impact on the applicability of the overall approach.

6. Dependency Detection

After integrating the different types of software artifacts in a model repository using a common meta-model, their dependencies must be elicited and explicitly recorded to enable a later change impact analysis. However, current research on dependency detection does not provide such a concept that sufficiently meets these demands. The following sections therefore introduce a comprehensive approach for multiperspective dependency detection. We explain how inter- and intra-artifact dependencies can be recorded to accomplish **Goal 8**, and how the challenges outlined in Section 1.3 are addressed. Furthermore, a taxonomy of dependency types is introduced that allows for the correct classification of dependencies and thus addresses **Goal 7** and further prepares the ground for the proposed impact analysis approach. First, however, we introduce our concept and data structure for storing dependency relations as traceability links.

The presented approach is build on our works on rule-based dependency detection for different modeling languages [Leh10, BLR11, RBFL11, FLR14]. New aspects contributed by this thesis are *a)* the thorough analysis of the different sources of dependencies, *b)* the analysis of the properties of software artifacts that can be utilized for dependency retrieval, *c)* our novel approach for the classification of dependencies, *d)* a systematic approach for the definition of dependency detection rules which, up until now, is only accomplished in an ad-hoc manner by existing work, and *e)* an evaluation of our approach by three additional case studies.

This chapter is organized as follows. To begin with, we provide definitions for the terms "dependency" and "dependency relation". Subsequently, Section 6.2 introduces our traceability meta-model that allows for modeling dependency relations as traceability links. Section 6.3 explains our taxonomy of dependency types and provides assistance for classifying dependency relations between software artifacts. In Section 6.4 we introduce our approach for dependency detection that is build upon our traceability meta-model and our dependency type taxonomy. We evaluate and compare our approach against other dependency detection approaches in Section 6.5 and finally discuss its critical points in Section 6.6.

6.1. Defintion of Dependencies

Unfortunately, software engineering literature does not provide a standard definition of the term "dependency". Therefore, we provide the following definition for the context of our work:

> ***Dependency:*** An element *A* is dependent on an element *B* if a change in *A* affects *B* or vice versa.

We are further defining the term "dependency relation" as follows:

> ***Dependency Relation:*** A dependency relation expresses a dependency and is defined as a 3-tuple consisting of a *source* element, a *target* element, and a *type* that

characterizes the dependency.

The *type* of a relation further provides semantics to the relation according to its purpose, which we are discussing in the next sections.

6.2. Modeling of Dependencies

Our first requirement for the utilization of dependencies for change impact analysis is the need for expressing arbitrary dependencies. As outlined in Section 3.3.3, traceability links are a suitable means for recording the dependencies between different software artifacts [III08], [ARNRSG06]. Hence, we propose an approach to model dependencies as traceability links to accomplish **Goal 7**. Although the general structure of traceability links has already been discussed in Section 3.3.3, we refine it by providing an EMF-based meta-model for traceability links. An initial version of this meta-model was originally developed in one of our previous works [Leh10]; however, it has been revised and simplified due to ongoing research and experiences gained from several case studies. Figure 6.1 illustrates our traceability meta-model.

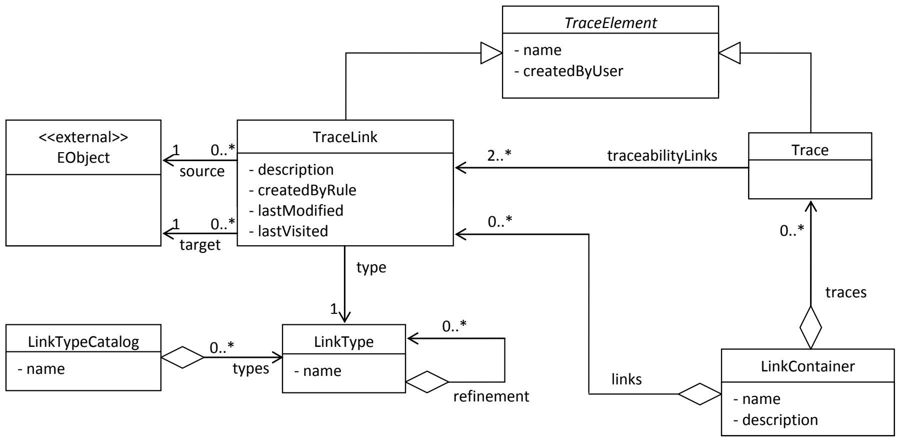


Figure 6.1.: Traceability meta-model. Improved and simplified version of [Leh10]

For the concepts presented in this thesis, the classes *TraceLink* and *LinkType* are of importance. The class *TraceLink* represents an actual traceability link, while the class *LinkType* represents the type of the dependency relation that is modeled. The actual source and target of a dependency are encoded as references to *EObjects*, as any software artifact is mapped upon this base class of the EMF meta-model (see Section 5.2.1). Due to usability reasons, *LinkTypes* can be nested and are stored in a *LinkTypeCatalog*. The resulting type hierarchy corresponds to the dependency type taxonomy which will be introduced in Section 6.3. The class *Trace* is part of our previous work on traceability detection and utilization [Leh10, BLR11]. A trace encompasses a set of transitively related traceability links, i.e. links that share common source or target elements. The class *LinkContainer* serves as a container element for traceability links and traces, and assists with organizing the workspace of our prototype tool (see Chapter 9).

In contrast to the traceability models proposed by Drivalos *et al.* [DKPF09] and Jaber *et al.* [JSL13], our model is much simpler in terms of neglecting any process related information. In regard to our goals we need to be able to link arbitrary types of software artifacts and to provide the information required for the later impact analysis (i.e. the dependencies and their types). Therefore, our model suffices. Moreover, when compared to the traceability model of Schwarz *et al.* [SEW09], our meta-model involves less effort when integrating new types of software artifacts because their model requires the traceable elements to be explicitly integrated into the meta-model. In contrast, in our approach every type of artifact can be linked without further changes to the traceability meta-model since all artifacts are mapped upon EMF-models.

6.3. Classification of Dependencies

Our next step towards the utilization of dependency relations for impact analysis is to classify the relations according to their type in order to facilitate the comprehension of their purposes. Therefore, this section introduces our classification of dependency types to accomplish **Goal 7**. The presented classification is based on the purposes of dependency relations that are discussed in the next section and the existing dependency classifications as discussed in Section 3.3.2.

6.3.1. Purposes of Dependencies

Each dependency relation is defined by its purpose or rationale that characterizes the relation and determines its effects on the propagation of changes [LFR13,FLR14]. We identified a whole set of different types of dependency relations where each is reflecting a different purpose. From these dependencies we distill a set of abstract dependency types that express the basic purposes of dependencies. For example, there are *structural* relations between the sub-components of a system and the system as a whole, such as *aggregations* or *compositions*. We identified a set of nine *general purposes* of dependency relations which were obtained from existing impact analysis approaches, existing dependency classifications (see Section 3.3.2), and our own observations and experiences gained through industrial projects and research case studies.

- **Purpose 1:** Describe the *structure* of software artifacts. The static structure or the dynamic allocation of software artifacts is one important aspect for impact analysis, e.g. the composition of components through sub-components.
- **Purpose 2:** Describe the *behavior* of software artifacts. The data flow within a system or the services provided by a component are useful sources for impact analysis, e.g. recording the execution traces of methods or functions.
- **Purpose 3:** Describe the *evolution* of software artifacts. Co-evolution patterns identified by repository mining approaches can indicate important dependencies between software artifacts.
- **Purpose 4:** Describe *conditions* and *constraints* which exist between software artifacts, e.g. the compliance of a requirement as precondition for another requirement [CSL^{+}01].
- **Purpose 5:** Describe *similarities* between software artifacts which are caused by either textual or conceptual overlappings between software artifacts. For example, overlappings between documentation and source code [ACC^{+}02].
- **Purpose 6:** Describe *abstraction* levels between software artifacts. Different views and artifacts often describe a software system on different levels of abstraction. For example,

there are high-level system goals and derived fine-grained requirements [SZ05].

- **Purpose 7:** Describe how software artifacts *realize* other software artifacts, e.g. how requirements are implemented through software architectures [SA03].
- **Purpose 8:** Describe how one software artifact *defines* another software artifact, e.g. how requirements are defined by design processes [RJ01].
- **Purpose 9:** Describe *cause-effect* relations between software artifacts, e.g. how a change in one software artifact can affect others [RJ01].

It might, however, be possible that there are additional purposes of dependency relations that our classification is not yet able to grasp and that other authors may propose an entirely different classification. However, our classification covers the basic purposes of dependencies that encompass the majority of the dependency relations under study and is thus sufficient.

6.3.2. A Taxonomy of Dependencies

This section introduces our classification scheme for dependency types that is based on the hypothesis that dependency types can be distinguished by their purpose. Thus, grouping dependency types with a similar purpose helps in formulating precise clusters of dependency types. To enhance the comprehension, a hierarchical arrangement of the dependency clusters is employed, where each level refines its predecessor by further specializing the purposes of the dependency relations it contains. In doing so, we provide a scheme for classifying dependency relations in a step-wise manner. Our taxonomy is comprised of four abstraction levels that provide a refinement of the dependencies according to their purpose. Figure 6.2 illustrates our taxonomy and its four levels of granularity that are discussed in detail in the remainder of this section.

- ***Level-0:*** The general dependency as located on Level-0 of our hierarchy represents the abstract concept of software dependencies. Such dependency relations do not carry a further meaning except that the two related software artifacts are dependent on each other. Level-0 dependencies are therefore detectable by any approach for dependency analysis, such as program slicers [Tip94] or call graph extractors.

  Some impact analysis approaches even do not go beyond analyzing Level-0 dependencies, such as the approaches of Bohner [Boh02a] or Hassaine *et al.* [HBG$^+$11] for instance, as the types and purposes of dependencies are of no relevance to them.

  When manually analyzing a software system for dependencies, developers may come to the conclusion that two software artifacts are dependent on each other. However, they might not yet be able to correctly say *why* or *how* as the exact type of the relation is not yet clear to them. Hence, they can label it as an abstract Level-0 *dependency* and later refine it once their knowledge of the system and its constituents has increased.

- ***Level-1:*** By introducing the notion of purpose to the abstract dependencies, Level-1 dependency clusters adhere to the nine different purposes of dependencies as identified in Section 6.3.1. In contrast to Level-0 dependencies, Level-1 allows for a more fine-grained selection of dependencies for certain types of impact analysis approaches. When performing history-based impact analysis for example, only *Evolutionary* dependencies are of interest (see Section 2.3.2).

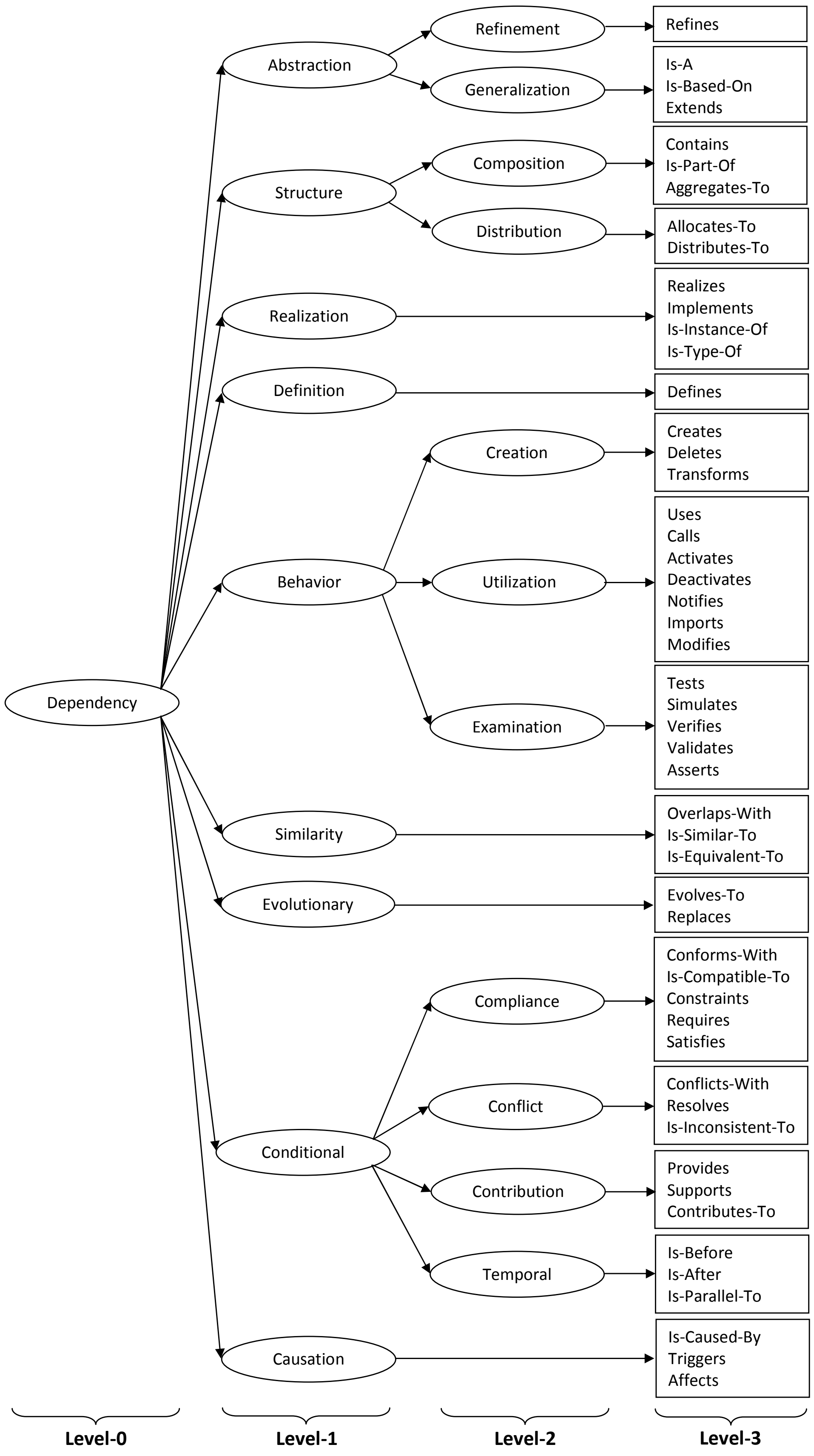


Figure 6.2.: Our taxonomy of dependency types

Furthermore, some impact analysis approaches are only capable of utilizing certain Level-1 dependencies and their corresponding concrete types as situated on Level-3. For example, information retrieval approaches are only able to process *Similarity* relations between software artifacts that are based on similar names or identifiers of software artifacts (see Section 2.3.3). Moreover, some Level-1 clusters, such as *Similarity* dependencies for instance, cannot be further refined and directly link to concrete Level-3 dependency types.

In regard to a potential, manual dependency analysis the dependency clusters of Level-1 allow for abstract dependencies to be refined according to their general purpose. This allows developers to actually comprehend the rationale of the dependencies.

- ***Level-2:*** The dependency clusters provided by the 3rd level of our taxonomy allow for a more precise classification of dependencies and further refine the notion of abstract purposes as introduced by Level-1. The Level-1 dependency clusters are refined as follows.

  The *Abstraction* cluster is sub-divided into the *Refinement* cluster expressing how software artifacts refine others to a more concrete form and into the *Generalization* cluster to express how software artifact extend the capabilities of others. *Structural* relations are refined to *Composition* relations to model dependencies between the constituents of a system and into *Distribution* relations to model allocations between different types of artifacts. The *Conditional* cluster is sub-divided into *Temporal* relations that express the order in which artifacts must be executed, *Contribution* relations that reflect how software artifacts contribute towards others (e.g. in a GRL goal graph), *Conflict* relations that model conflicts (e.g. conflicting requirements), and *Compliance* relations that model constraints and conditions between software artifacts. The *Behavior* dependencies are further refined into dependencies that express how artifacts *utilize* other software artifacts, how artifacts *create or destroy* other artifacts, and how artifacts are used to *examine* others.

  Refining a dependency to a Level-2 dependency already allows for a detailed assessment of how this relation propagates the effects of changes to dependent artifacts. For example, refining the *Abstraction* cluster into the *Refinement* and *Generalization* clusters allows to separate changes applied on the “fine-grained artifact” from changes applied on the “coarse-grained artifact”, as changes applied on the latter propagate to all of the more fine-grained artifacts but not necessarily vice versa. Our Level-2 clusters therefore provide developers with further means for a more precise classification of Level-1 dependencies.

- ***Level-3:*** The fourth and final level of our taxonomy consists of lists of concrete dependency types that constitute Level-1/2 dependency clusters. To better distinguish between concrete dependency types and dependency type clusters, all concrete dependency types are named as verbs since they indicate a former or still ongoing developer activity between the two dependent software artifacts.

  Our clusters of concrete dependency types as shown in Figure 6.2 are based on the outcome of our comprehensive literature review (see Section 3.3.2 and Table 3.1) and our own findings from various case studies and industrial projects.

  However, prior to populating Level-3 dependency clusters with concrete dependencies, we had to consolidate the available dependency types proposed by related work in order to merge duplicates, resolve inconsistencies, and to add yet missing types. The process of revising the dependency types consisted of the following three steps:

1. *Merge duplicated and similar dependency types.* Many approaches proposed the usage of similar or even the same types of dependencies, however with different names, which therefore had to be merged. For example, there are three different versions of the *Refinement* dependency that can be found in related work: "refines" [SA03], "refinement" [MPR07, JZ09, PDK$^+$11], and "refine" [WJSA06].
2. *Merge dependency types and their inverse types.* There is no conceptual difference whether "A *tests* B" or "B *is tested by* A". Hence, such dependency types were merged as well, which helped to further reduce the number of dependency types.
3. *Assign the remaining types to our dependency clusters.* The collected dependency types were sorted into one of the nine purpose clusters defined in Section 6.3.1.

Finally, we obtained a list of 51 concrete dependency types as situated on Level-3 of our taxonomy (see Figure 6.2). In doing so, we were able to reduce the amount of dependency types by more than 70% when compared to the types listed in Table 3.1, which makes our taxonomy much more suitable for developers and researchers applying it in practice.

Which of our concrete dependency types are utilized by a certain impact analysis approach depends on the approach itself. A call graph based approach would analyze "Calls" relations (*Utilization* cluster), whereas an approach for analyzing the impacts of changes on source code packages would, for example, inspect "Imports" relations (also *Utilization* cluster). Other impact analysis approaches, such as program slicing for example, analyze a multitude of different dependency types, including *Behavioral*, *Structural*, and *Conditional* relations.

6.4. Rule-based Dependency Detection

After providing suitable means for modeling and classifying dependency relations, we introduce an approach to elicit the actual dependencies in order to utilize them for the proposed change impact analysis approach and to accomplish **Goal 8**. The discussion of our approach is structured as follows. To begin with, in Section 6.4.1 we analyze what properties of software artifacts can be utilized for dependency detection. Second, in Section 6.4.2 we introduce our dependency detection approach. Third, in Section 6.4.3 we discuss how dependencies of different origin can be detected. Finally, in Section 6.4.4 we discuss our dependency detection rules, their structure, and define a methodology how dependency detection rules can be defined and derived for different types of software artifacts to expand the scope of the approach.

6.4.1. Properties for Dependency-Retrieval

Before we introduce our approach for dependency detection we have to analyze what properties of software artifacts can actually be analyzed to derive potential dependencies.

An initial survey with the same intentions was carried out by Antoniol *et al.* [ACPT01] which, however, was only focused on source code artifacts and did not address all of the available properties of software artifacts. Antoniol *et al.* consider the names of classes, methods, and attributes, as well as inheritance and collaboration relations as relevant for detecting dependencies. Furthermore, they combine and prefix the names of methods and attributes with the name of their class. However, this covers the available properties only in a partial manner.

For our approach, we distinguish between three different categories of properties of software artifacts that can be utilized when searching for dependencies:

1. Names and identifiers of software artifacts.
2. Structural aspects of software artifacts.
3. Existing relations between or within software artifacts.

The first category of properties is similar to the information considered by Antoniol *et al.*, i.e. we also examine text-based attributes, such as the names of classes, the names of data types etc., as they are an important indicator of dependencies [CT99, DP06]. The second category encompasses all types of structural information that can be used for dependency detection, such as hierarchical containment relations (e.g. "a class is a part of a package"). The last category is comprised of explicit relations that already exist in or in-between software artifacts, such as explicit inheritance relations stemming from the usage of object oriented concepts.

Consequently, our dependency detection approach will investigate all three types of properties when searching for dependency relations stemming from one of the four sources of dependency relations as outlined in Section 3.3.1.

6.4.2. Detection Approach

We now present a holistic rule-based approach for detecting and explicitly recording the dependencies of heterogeneous software artifacts. Therefore, our approach takes into account the different sources of dependencies (see Section 3.3.1) and the different properties of software artifacts which can be exploited for dependency detection (see previous section).

We utilize a set of dependency detection rules that are able to elicit and record dependency relations as traceability links using our traceability model as presented in Section 6.2. The rules are applied on the heterogeneous software artifacts that were converted into unified EMF-models and later on stored in the model repository (see previous chapter). Once applied on a set of EMF-models, the rules query and compare their attributes and existing relations to determine potential dependencies. They are also capable of traversing the graph-like structure of the software artifacts in order to assess existing structural relations. Consequently, they are able to address all of the potential properties of software artifacts as discussed in Section 6.4.1. Moreover, as the rules are pre-defined to address specific dependencies, they are able to capture the semantics of potential dependency relations (i.e. their type and direction).

Therefore, for each potential dependency relation, a detection rule must be established that is capable of capturing this dependency and to record it as a traceability link. The required detection rules are implemented by a query language which is able to analyze the artifacts that were converted into EMF-based models as described in the previous chapter. Each rule encodes a set of conditions that must be fulfilled by the software artifacts in order to detect and record a dependency relation between them. Furthermore, each rule contains information that are required for determining the types and directions of potential dependency relations. The latter only applies if a relation is directed, such as *inheritance*-relations between classes and their superclasses for instance. In contrast, typical examples for undirected dependency relations are *equivalence*-relations between UML design classes and their corresponding Java code classes.

In summary, our approach for rule-based dependency detection consists of the following four steps which are also illustrated by Figure 6.3:

1. *Manual identification of potential dependency relations* by studying the meta-models of software artifacts, development paradigms, development methodologies, views, etc.
2. *Understanding the potential dependencies* by determining why, when, and where they exist. Furthermore, their type has to be classified according to their purpose.
3. *Definition of dependency detection rules* based on the information obtained in the previous steps. The rules encode the conditions to detect the dependencies and conditions to determine their type and direction.
4. *Application of the established dependency detection rules* on a set of software artifacts to elicit their dependencies and to explicitly record them as traceability links.

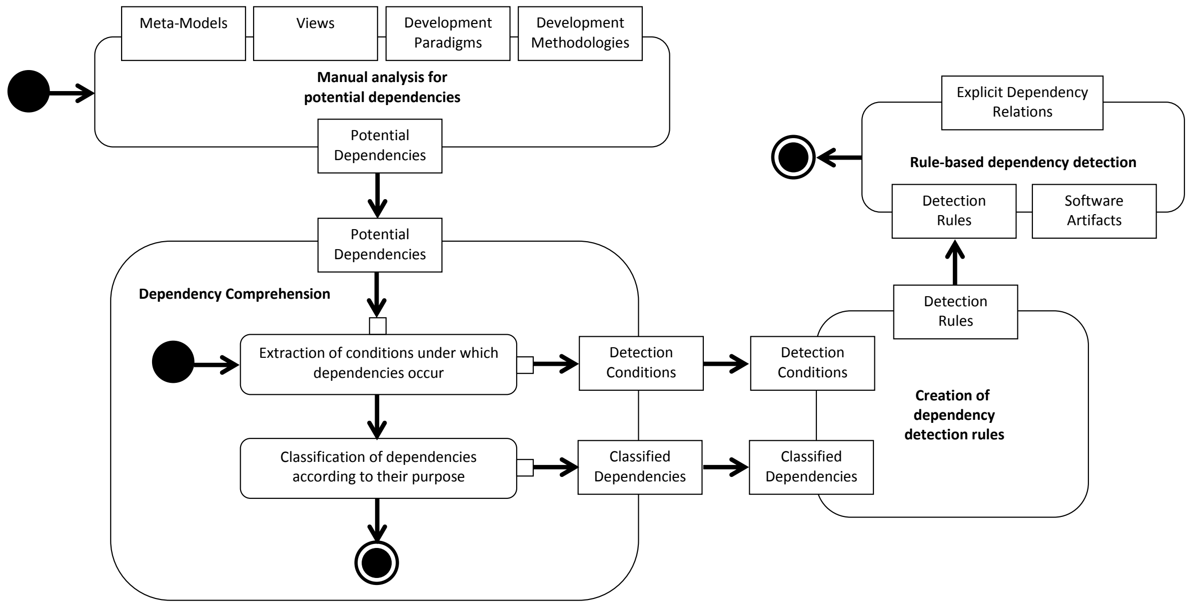

Meta-Models
Views
Development Paradigms
Development Methodologies
Manual analysis for potential dependencies
Potential Dependencies
Potential Dependencies
Dependency Comprehension
Extraction of conditions under which dependencies occur
Detection Conditions
Classification of dependencies according to their purpose
Classified Dependencies
Detection Conditions
Classified Dependencies
Detection Rules
Creation of dependency detection rules
Explicit Dependency Relations
Rule-based dependency detection
Detection Rules
Software Artifacts

Figure 6.3.: Overview of our dependency detection approach

In stark contrast to the ability of our approach to analyze all the properties of software artifacts and all the sources of dependencies, existing dependency detection approaches as discussed in Section 3.3.4 are much more restricted in this regard. IR-based approaches, for example, cannot analyze the structure of software artifacts nor existing relations between them as they are limited to the analysis of textual information. Likewise, most approaches are not able to take into account all the different origins of dependencies. For example, IR-based approaches are not able to detect dependencies encoded in the meta-models of software artifacts, such as simple *inheritance*-relations as defined in the Java or UML meta-models.

On the other hand, our approach requires a set of rules to be available before any automated dependency detection can be conducted. Hence, our rule-based approach requires an initial manual effort for creating the dependency detection rules. We have to manually analyze meta-models, programming language specifications, development methodologies etc. to identify potential dependencies and establish a set of detection rules which are then used for detecting these dependencies in real software development projects. However, the upcoming section presents a scheme to guide the manual dependency detection, which in turn reduces the required effort.

Our approach's most important advantage, however, is its ability to determine the types of the dependency relations due to the preceding manual analysis of the software artifacts and their underlying concepts. This in turn is important for any change impact analysis effort since the type of a dependency relation assists with determining further change propagation [LFR13]. In contrast, IR-based and MSR-based approaches are not able to distinguish between different types of dependencies, which limits their applicability for dependency-based impact analysis.

Additional benefits of our approach are the increased precision and recall when detecting dependencies. We conducted five case studies on rule-based dependency detection and achieved an average precision and recall of 85%, which is well above results achieved with IR-based or MSR-based approaches (see the upcoming discussions in Section 6.5). Moreover, while the initial effort for creating the dependency detection rules accounts only once, the higher ratios of false-positives as detected by IR-based and MSR-based approaches result in frequent effort for developers when examining the proposed dependencies.

Thus, with our approach we seek to combine knowledge of meta-models, development paradigms, and development methodologies with the ability for automation. We transform this knowledge into rules that afterwards can automatically be applied for multiperspective dependency detection. With this approach we reduce the required manual effort to an initial phase of rule-creation, whereas the resulting rules can be reused throughout different projects.

6.4.3. Identification of Potential Dependencies

In the following sections we elaborate on the initial manual detection of different types of dependencies as a precondition for defining the required dependency detection rules. We focus on the specific properties of each source of dependencies and illustrate how they influence the dependency detection. However, we cannot discuss each of the potential dependencies in detail, therefore we present typical examples instead. Additionally, we analyze how the challenges discussed in Section 1.3 influence multiperspective dependency detection and how our approach copes with them. Finally, we also analyze how existing dependency detection approaches cope with those challenges and how they cope with the different types and sources of dependencies in comparison to our approach.

6.4.3.1. Meta-model Dependencies

The set of meta-model dependencies this thesis is dealing with encompass the UML meta-model, the Java language specification, and the JUnit specification. In general, the UML and Java meta-models share many commonalities regarding classes, interfaces, methods, attributes, etc. which simplifies the detection of dependencies among them. In both specifications, classes and interfaces have the same dependencies towards attributes and methods, such as that classes and interfaces *define* methods, while methods return *instances of* classes for example.

Consequently, such dependencies can be identified by studying the respective meta-models, which requires an initial manual effort. However, once identified, those dependencies can be made explicit for every software project using the same set of rules. Meta-model dependencies encompass all types of relations as identified in Section 6.3.1, except for *similarity* and *cause-effect* dependencies. An excerpt of those dependencies is presented by Table 6.1.

Source	Type	Target
Component	Requires	Interface
Component	Provides	Port
Class	Defines	Method
Method	Defines	Parameter
Method	Calls	Method
Attribute	Is-Instance-Of	Class
Parameter	Is-Type-Of	Data Type

Table 6.1.: An excerpt of meta-model dependencies

For example, one of the most common meta-model dependencies is the *call*-relation between two methods, which is utilized by call graph based change impact analysis approaches (see Section 2.3.1.3). In the following, we present and discuss the three steps that are required to elicit such dependencies and demonstrate the required conditions as pseudo code in Listing 6.1 (for an explanation of the utilized query statements see Table 6.5 in Section 6.4.4.2). First, one has to determine the method being called by a (Java) code statement. Secondly, the method containing this code statement must be determined by traversing the structure of the program AST. Finally, the resulting *call*-relation between the method containing the calling statement (caller) and the called method (callee) can be established.

```
IF
  ModelRelatedTo(codestatement, 'Calls', method1) AND
  ModelParentOf(method2, codestatement)
THEN
  CreateRelation(method2, 'Calls', method1)
```

Listing 6.1: Conditions for detecting call relations

In the following paragraphs we discuss the influence of the challenges outlined in Section 1.3 on the detection of meta-model dependencies.

Degree of Formalization: The degree of formalization does not influence the detection of meta-model dependencies because each meta-model defines the degree of formalization of its instances beforehand. If an artifact does not adhere to any meta-model or language specification, such as free text for instance, no meta-model dependencies can be detected as there are simply none present. Moreover, if an artifact violates its underlying meta-model, further errors are likely to occur, such as compilation errors (code) or validation errors (models).

Level of Abstraction/Completeness: An artifact's abstraction level determines which of the potential meta-model dependencies are present in the artifact and thus potentially detectable. For example, consider a detailed component diagram that contains various meta-model dependencies between the depicted components, interfaces, and ports, such as "component A *provides* interface B". In contrast, another component diagram displaying the same context on a higher level of abstraction (e.g. system components only) might neglect all the fine-grained details, such as internal components and interfaces, and hence contains less meta-model dependencies.

Inconsistencies: Inconsistencies affecting the detection of meta-model dependencies are usually related to inconsistencies between two or more software artifacts, which results in a loss of meta-model dependencies between them. For example, if no class, use case actor or component matches with a lifeline of a process or sequence diagram, no *Is-Instance-Of*-relations are present and thus detectable. On the other hand, these cross-artifact inconsistencies do not interfere with the dependencies that are contained within a certain software artifact. For example, even if a UML class and its corresponding Java class are inconsistent to each other, the meta-model dependencies within each class are still present and hence detectable by the corresponding rules.

Finally, we analyze how IR and MSR-based approaches deal with meta-model dependencies under consideration of those three challenges and how they compare to our rule-based approach. IR-based approaches typically cannot detect meta-model dependencies as there is often no textual similarity between the artifacts, e.g. between methods and classes or between class attributes and data types. Likewise, MSR-based approaches are hardly able to detect meta-model dependencies as they are typically contained within a single file. For example, dependencies between a class and its methods are contained within the same Java-file, hence there are no co-changes to be detected if the co-change analysis is conducted on the granularity level of files, which, however, is most typical. A more fine-grained analysis could only be accomplished by analyzing the co-change behavior of entities such as methods and attributes. This, however, is a very costly operation as the entire version history of a repository has to be searched at this fine-grained level. To the best of our knowledge, no such approach has yet been considered in current research (see Section 2.3.2 and Section 3.3.4.2). Furthermore, both MSR-based and IR-based approaches are not able to distinguish between the different types of dependencies. However, both approaches are not affected by varying degrees of formalization or abstraction. Yet, inconsistencies between software artifacts have the same severe impact as on our approach.

6.4.3.2. Object Oriented Dependencies

Since this thesis deals with software that is developed using object oriented concepts and technologies, we have to discuss the dependencies that are introduced by applying these concepts. The most important object oriented dependency relation is the inheritance relation between classes as one of the key features of object orientation [Mey96]. Other relevant dependencies encompass the “implementation” of interfaces by classes, as well as the “creation” and “deletion” of objects through constructors and destructors defined by classes.

Source	**Type**	**Target**
Class	Extends	Class
Interface	Extends	Interface
Class	Implements	Interface
Method	Implements	Method
Method	Creates	Class
Method	Deletes	Class

Table 6.2.: An excerpt of object oriented dependencies

Dependencies introduced by the object oriented paradigm require an initial manual analysis of object oriented concepts in order to establish the required dependency detection rules. An excerpt of those dependencies is presented by Table 6.2. As a running example we discuss how implementation dependencies between interface-methods and class-methods can be detected because if a class implements an interface, the class’ methods are supposed to implement the methods defined by the interface (see Figure 6.4).

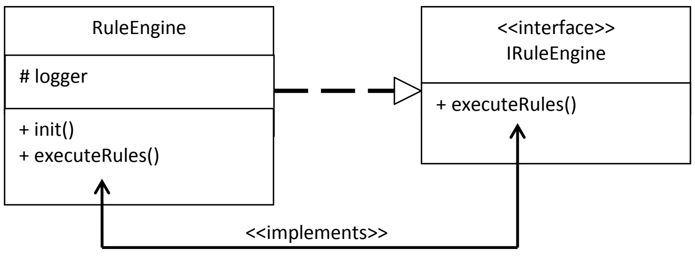


Figure 6.4.: *implements*-dependency between two methods

Hence, there is a dependency between both "types" of methods that requires four steps to be detected. These steps are illustrated by the pseudo code depicted by Listing 6.2 (for an explanation of the utilized query statements see Table 6.5 in Section 6.4.4.2). First, a set of *defines*-relations between a class and its methods must be established. Likewise, the same has to be established between the interface and its methods. Thirdly, the *implements*-relation between the class and the interface must be detected. Finally, the corresponding methods of those interfaces and classes can be linked based on the equivalence of their names or method signatures.

```
IF
  ModelRelatedTo(class, 'Defines', method1) AND
  ModelRelatedTo(interface, 'Defines', method2) AND
  ModelRelatedTo(class, 'Implements', interface) AND
  ValueEquals(method1::name, method2::name)
THEN
  CreateRelation(method1, 'Implements', method2)
```

Listing 6.2: Conditions for detecting implementation relations between methods

Furthermore, we have to discuss the influence of the challenges outlined in Section 1.3 on the detection of object oriented dependencies.

Degree of Formalization: The discussion of the consequences of insufficient formalization can be omitted for object oriented dependencies, as the concept of object orientation itself defines a certain level of formalization. Moreover, all types of software artifacts considered by this thesis (UML, Java, JUnit) adhere to a strict meta-model, hence the lack of formalization has no influence as already laid out for meta-model dependencies in the previous section.

Level of Abstraction/Completeness: Likewise, the influence of different abstraction levels and incomplete artifacts on the detection of object oriented dependencies is quite similar. The level of completeness defines which of the potential dependencies are present in an artifact and hence can be detected. Therefore, incomplete artifacts result in missing dependencies.

Inconsistencies: Inconsistencies between artifacts of different views are not related to object oriented dependency relations, as object oriented dependencies are limited to artifacts of a single view. For example, inheritance relations between classes of the data model are always restricted to the data view, while inheritance relations between system components are situated on the structural view and do not interfere with them. However, inconsistencies within artifacts may occur, such as that an attribute's type refers to a non-existing class and thus cannot be resolved.

In this final paragraph we analyze how IR and MSR-based approaches deal with object oriented dependencies under consideration of those three challenges and how they compare to our rule-based approach. To begin with, we have to point out again that both IR and MSR-based approaches are not able to determine the types of potential object oriented dependencies.

The effectiveness of IR-based approaches directly depends on the similarity of the names, identifiers, etc. provided by the artifacts to be analyzed. This assumption, however, does not always hold. This is especially true in regard to *implementation* and *inheritance* dependencies, where the names of the involved software artifacts do often not match. The following code example represents a cutout of our prototype's source code (see Chapter 9) which illustrates this problem.

```
public class RuleApplicationOperation implements IRunnableWithProgress {...};

public interface IRuleEngine extends IProcessingComponent {...};
```

Listing 6.3: Java code illustrating object oriented dependencies

Since there are no textual similarities in the code-snippet presented in Listing 6.3, IR-methods fail to detect the *inheritance*-dependency as well as the *implementation*-dependency. In contrast, the required dependency detection rules can be easily written (see Appendix A).

MSR-based approaches are better suited for detecting object oriented dependencies because most inheritance and implementation relations exist between entities that are usually contained by different Java source code files. For example, the dependencies given in the code-snippet above (Listing 6.3) could be detected if the Java files were frequently changed together. On the other hand, conducting a co-evolution analysis in this scenario is actually impossible since the *IRunnableWithProgress* interface is a part of the Java API, while the *RuleApplicationOperation* class is a custom class defined in our prototype. Hence, they cannot evolve together in the same physical repository. Likewise, it is hard to directly couple the evolution of source code with the evolution of UML diagrams as both usually evolve in separate repositories as well. Moreover, the detection of object oriented dependencies through MSR-based approaches is limited to the granularity of class-files when applied for Java source code.

6.4.3.3. Design Methodology Dependencies

In this section we discuss dependencies introduced by design methodologies, how they can be detected, and how the challenges outlined in Section 1.3 influence their detection. In general, design methodologies define steps to be followed when creating a software system and means to accomplish these steps. In this thesis we focus on *Object Oriented Analysis* (OOA) [CY91, Boo94] and *Object Oriented Design* (OOD) [Boo94] as two examples of such methodologies. Both approaches define a sequence of steps leading from the initial system requirements towards the final implementation in source code. At the same time they also define the types of artifacts to be created during each step, such as classes or components, and the services provided by them. Through this step-wise refinement, these methodologies also introduce dependencies to the software artifacts that describe different aspects of the resulting design and implementation.

The following presents a brief overview of the steps of Object Oriented Analysis (OOA) as proposed by Coad and Yourdon [CY91] and shows *how* and *which* dependencies are introduced by applying them.

The initial step "Finding classes and objects" aims to define the classes that constitute a software system. The second step "Identifying structures" defines the communication paths between those classes by introducing dependency relations between them, such as inheritance relations or aggregations. Hence, this step introduces a set of object oriented dependencies and meta-model dependencies that can be detected as discussed in Section 6.4.3.2. The third step "Identifying subjects" defines the system components to which classes are assigned, thereby introducing structural meta-model dependencies when utilizing UML components and UML/Java classes that can be captured as discussed in Section 6.4.3.1. Once the structure of the system is laid out, the next step "Defining attributes" assigns attributes to the classes, which again introduces meta-model dependencies, this time however between classes and attributes. Finally, the last step "Defining services" assigns the required services (methods) to the classes, which also introduces meta-model dependencies between the classes and methods.

Overall, by analyzing the steps proposed by a methodology it is possible to derive rules to capture the dependencies that are introduced by the methodology. This manual analysis is therefore required for each development methodology that is applied in a certain software development project. An excerpt of the dependencies introduced by OOA/OOD is presented by Table 6.3.

Source	**Type**	**Target**
(Java)-Class	Is-Equivalent-To	(UML)-Class
(UML)-Class	Refines	(UML)-Component
(Java)-Package	Is-Equivalent-To	(UML)-Package
(UML)-Activity	Refines	(UML)-Use Case
(UML)-ActivityNode	Is-Equivalent-To	(UML)-Message
(UML)-Activity	Realizes	(UML)-Use Case

Table 6.3.: An excerpt of design methodology dependencies

A typical example of such a design methodology dependency is the refinement of a use case by UML activity or sequence diagrams during the "Defining services"-step of OOA. Figure 6.5 presents an example in which a use case diagram is refined by a sequence diagram, where grey arrows indicate meta-model dependencies and the red arrow indicates the introduced design methodology dependency. The detection of this dependency involves the following four steps that are also illustrated as pseudo code by Listing 6.4 (for an explanation of the utilized query statements see Table 6.5 in Section 6.4.4.2). First, the general *equivalence* of the sequence diagram and the use case diagram has to be detected. Secondly, a *contains*-dependency must be established between the use case diagram and the actual use case. Likewise, *contains*-dependencies must be established between the sequence diagram and the messages exchanged between its swimlanes. Finally, the *equivalence*-relation between the message and the use case can be established based on similar names and the equivalence of their containing objects.

```
IF
  ModelRelatedTo(use case diagram, 'Is-Equivalent-To', sequence diagram) AND
  ModelRelatedTo(use case diagram, 'Contains', use case) AND
  ModelRelatedTo(sequence diagram, 'Contains', message) AND
  ValueEquals(use case::name, message::name)
THEN
  CreateRelation(use case, 'Is-Equivalent-To', message)
```

Listing 6.4: Conditions for detecting dependencies between use cases and sequence diagrams

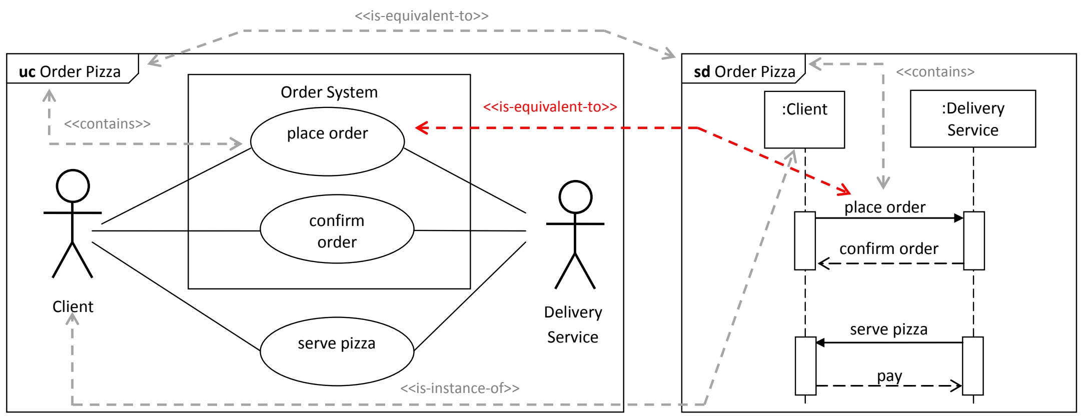


Figure 6.5.: Design dependencies between a use case and a sequence diagram (red arrow)

Finally, we discuss how our approach is influenced by the challenges outlined in Section 1.3 and how IR and MSR-based approaches are affected by them during the detection of design methodology dependencies. Prior to this discussion we like to highlight the fact that, since design methodologies define the steps to be followed and the means to accomplish these steps, they also define the level of formalization (i.e. which types of artifacts are created) and the level of abstraction (i.e. which information are encoded in the artifacts) for each of their steps.

Degree of Formalization: Varying levels of formalization or the absence of any formalization while applying a design methodology eventually lead to one of the following two scenarios. If the artifacts that are created within a certain step of a methodology do not adhere to any formalization, they cannot be linked to other artifacts of previous or upcoming steps by our rules. In contrast, IR and MSR approaches are potentially able to do so, if either textual similarities or evolutionary couplings are present. Secondly, if the artifacts that were created during the transition from one step to another step adhere a certain but different degree of formalization, rules can be written to record these dependencies, for example for the transition from use cases towards the initial system components. Likewise, IR and MSR-based approaches are potentially able to record such dependencies as well. They are, however, not able to determine any auxiliary information of the dependencies, such as their types for example.

Level of Abstraction/Completeness: As discussed in the previous sections for meta-model dependencies and object oriented dependencies, the level of abstraction determines which of the potential dependencies are present and thus potentially detectable by rules. Hence, the detection rules can be adjusted according to the level of abstraction defined by a design methodology, for example for the transition from high-level component diagrams describing the overall system architecture to fine-grained component diagrams describing sub-components that represent a refined excerpt of the former. In contrast, IR and MSR-based approaches are independent of the level of abstraction, as long as their underlying assumptions are met. Therefore, they are also able to record such dependencies, however, without being able to gather any additional information like the types of the dependencies for instance.

Inconsistencies: The influence of cross-artifact inconsistencies on the detection of design-methodology dependencies is twofold. The first case is characterized by applying a design or development methodology without executing all of its steps. If certain steps were not executed, the transitions between certain views might be missing. Consequently, the cross-view-dependencies have never been introduced to the system and hence cannot be detected, neither by rules nor by IR and MSR-based approaches. The second case is characterized by the incorrect application of a design or development methodology, which leads to inconsistencies between the views, which in turn results in incorrect or missing dependencies. Thus, our rules cannot detect the potential dependencies. The same applies to IR-based approaches, as the names of the involved software artifacts are unlikely to match due to the inconsistencies. MSR-based approaches on the other hand are able to detect potential evolutionary couplings despite inconsistencies, which, however, assumes that the artifacts of the inconsistent views evolved together.

In conclusion, it can be stated that dependency detection rules can be established in a way that they are able to follow the steps of a design methodology, while MSR and IR-based approaches cannot be adapted to do so directly. However, IR and MSR-based approaches can help to establish dependencies if design or development methodologies were not applied correctly but for the price of losing all auxiliary information, such as dependency types.

6.4.3.4. Multiperspective Dependencies

Finally, we address the detection of multiperspective dependencies and the challenges associated with them. As stated in Section 4.3.2, in this thesis we consider the structural view, the behavioral view, the code view, and the test view of software that all describe a specific cutout of a system and are therefore discussed in the following.

We consider multiperspective dependencies that exist between Java source code entities, JUnit test cases, and different types of UML models that emerge from the interplay of the above mentioned views. For example, there are dependencies between Java classes (code view) and JUnit classes (test view), as well as between Java methods (code view) and JUnit methods (test view), such as that there should exist a unit test for each class and at least one test-method for each method. Consequently, these dependencies connect elements of the code view with elements of the test view. An excerpt of those dependencies is presented by Table 6.4.

Source	**Type**	**Target**
(UML)-Lifeline	Is-Instance-Of	(UML/Java)-Class
(Java)-CodeStatement	Is-Equivalent-To	(UML)-ActivityNode
(Java)-CodeStatement	Is-Equivalent-To	(UML)-Message
(JUnit)-Class	Tests	(Java)-Class
(JUnit)-Method	Tests	(Java)-Method

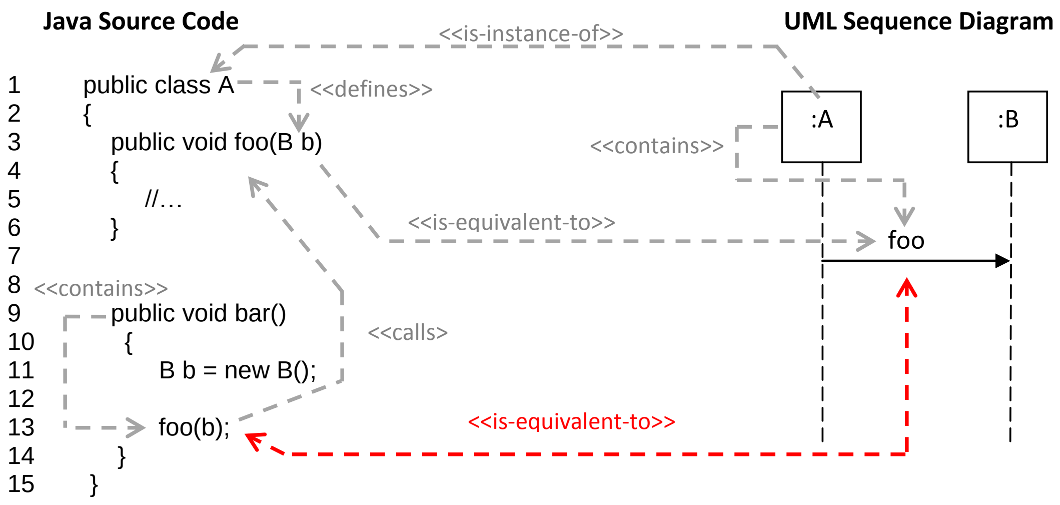
Table 6.4.: An excerpt of multiperspective dependencies

Detecting dependencies between source code statements and behavioral UML diagrams is especially important as they allow to bridge the gap between high-level design and the actual implementation in source code. Behavioral UML models, such as sequence diagrams or activity diagrams, model the behavior of a system on a more abstract level than source code. At the same time, they are also related to structural UML diagrams, such as component diagrams for instance, that describe the participants of the dynamic processes. Consequently, they allow for establishing connections between artifacts of the structural view and artifacts of the more fine-grained code view.

A typical multiperspective dependency exists between Java method-calls (code-view) and UML sequence diagrams (behavioral-view), where the method-calls correspond to messages that are exchanged between the lifelines of a sequence diagram. This example is illustrated by Figure 6.6, where the red arrow indicates the dependency connecting the artifacts of both views. The detection of such dependency relations requires the execution of the following five steps. First, a dependency between the UML lifeline and the Java class has to be detected, stating that the lifeline is an *instance of* the class. Second, the Java class has to be linked with all the methods it *defines*. Likewise, the lifeline must be linked with all the messages *contained* by it. Once this is done, the *equivalence* between the Java method and the UML message has to be detected. Subsequently, a *call*-relation between the Java source code statement and the Java method has to be established. Finally, the *equivalence*-relation between the Java source code statement and the UML message can be established. These steps are also illustrated in pseudo code by Listing 6.5 (for an explanation of the utilized query statements see Table 6.5 in Section 6.4.4.2).

```
IF
  ModelRelatedTo(Lifeline, 'Is-Instance-Of', ClassDeclaration) AND
  ModelRelatedTo(ClassDeclaration, 'Defines', Method) AND
  ModelRelatedTo(Lifeline, 'Contains', Message) AND
  ModelRelatedTo(Method, 'Is-Equivalent-To', Message) AND
  ModelRelatedTo(JavaStatement, 'Calls', Method)
THEN
  CreateRelation(JavaStatement, 'Is-Equivalent-To', Message)
```

Listing 6.5: Conditions for detecting equivalences between code statements and sequence charts



```
public class A
{
   public void foo(B b)
   {
      //...
   }

   public void bar()
    {
       B b = new B();
       foo(b);
    }
 }
```

Figure 6.6.: Dependencies between the code view and the behavioral view

To conclude this section, we discuss the influence of the challenges outlined in Section 1.3 on the detection of multiperspective dependencies. At the same time, we also analyze how IR and MSR-based approaches deal with the detection of multiperspective dependencies under consideration of the challenges, and how they compare to our approach.

Degree of Formalization: We have to discuss the influence of different levels of formalization and the absence of any formalization respectively. First, if the artifacts of one view are not formalized at all (e.g. free-text requirements), then the only remaining option is to try to map the names of the artifacts. Our approach is able to address this situation by applying a rule that only compares the names of arbitrary artifacts. However, IR-based approaches are more suitable in this context. Second, if the level of formalization of one view is different than the formalization of a second view, the rules can be adapted accordingly as well. For example, our rules are able to analyze the names and structural aspects encoded in semi-structured artifacts, which potentially provides more information than plain text analysis as implemented by IR-based approaches. Likewise, MSR-based approaches are applicable on any type of files, even if the content of the files it not formalized. However, as discussed before, the detection of dependencies is then limited to files only, while no additional information can be provided.

Level of Abstraction/Completeness: A varying level of abstraction among the views limits the detection of dependencies to those of the most "abstract" (or less detailed) view. Dependencies of more fine-grained levels, however, cannot be detected as the more fine-grained dependencies are not present in the "abstract view". The same limitation applies for IR-based approaches as the level of abstraction determines which names and identifiers are provided by the software artifacts. In contrast, MSR-based approaches are applicable independent of any abstraction level, as long as sufficient historical information are available.

Inconsistencies: In general, inconsistencies between the views prevent any dependency from being detected by our rules because the artifacts of the views are inconsistent to each other. Typical examples are test cases (test view) that refer to classes that do exist in the code view or UML lifelines (behavioral view) that refer to non-existing components of the structural view. The same limitation applies to IR-based approaches as the names and identifiers of the software artifacts are unlikely to match if the artifacts themselves are inconsistent to each other. MSR-based approaches on the other hand are able to detect potential evolutionary couplings between files despite inconsistencies. However, the chances are likely that the detected dependencies are incorrect due to the inconsistencies and thus not useful for a later impact analysis.

6.4.4. Detection Rules

In this section we focus on the rules that implement our approach for dependency detection and provide the required means to elicit the potential dependencies as discussed in the previous sections. Therefore, we explain the structure of our rules in Section 6.4.4.1 and discuss an exemplary rule to further illustrate our concepts. We present the query operators which can be used by our rules in Section 6.4.4.2 and introduce a scheme for the definition of additional rules in Section 6.4.4.3. Moreover, Appendix A lists all dependency detection rules that accompany this thesis. They can also be obtained from the website of our prototype tool [EMF14].

6.4.4.1. Structure of the Rules

The structure of our rules is very similar to typical SQL-queries and consists of three parts:

1) ***Element-Definition:*** Within the *Element-Definition*, all types of artifacts that are addressed by a rule are specified. Furthermore, an *alias*-name is assigned to each type to allow for writing more readable query-statements. This part therefore corresponds to the "SELECT"-statement of SQL, while the *alias* corresponds to its "SELECT AS"-statement.

2) ***Query-Definition:*** The *Query-Definition* contains the actual query-conditions to identify potential dependencies between the artifacts declared in the *Element-Definition* part. The query corresponds to the “WHERE”-clause of a SQL-query and may contain nested sub-queries.

3) ***Result-Definition:*** Last, the *Result-Definition* is responsible for processing the results of the query, i.e. it determines which type of traceability link shall be created and which artifacts shall be linked. Our rule concept supports the creation of multiple traceability links using one rule, which allows us to even link sub-parts of artifacts using only one rule.

The structure of our rules is also reflected by their meta-model that is presented by Figure 6.7. The *Query-Definition* is modeled by the *LogicCondition*-class that is used to model logical operations, such as AND. The *BaseCondition*-class is used for modeling query-operations that assess attributes and relations of models, which is explained in the next section. The *ActionDefinition*-class is used for determining how the output of a rule shall be processed, i.e. which models should be linked by which type of dependency relation. To ease the usage of our dependency detection rules, they can be grouped in rule catalogs implemented by the *RuleCatalog*-class.

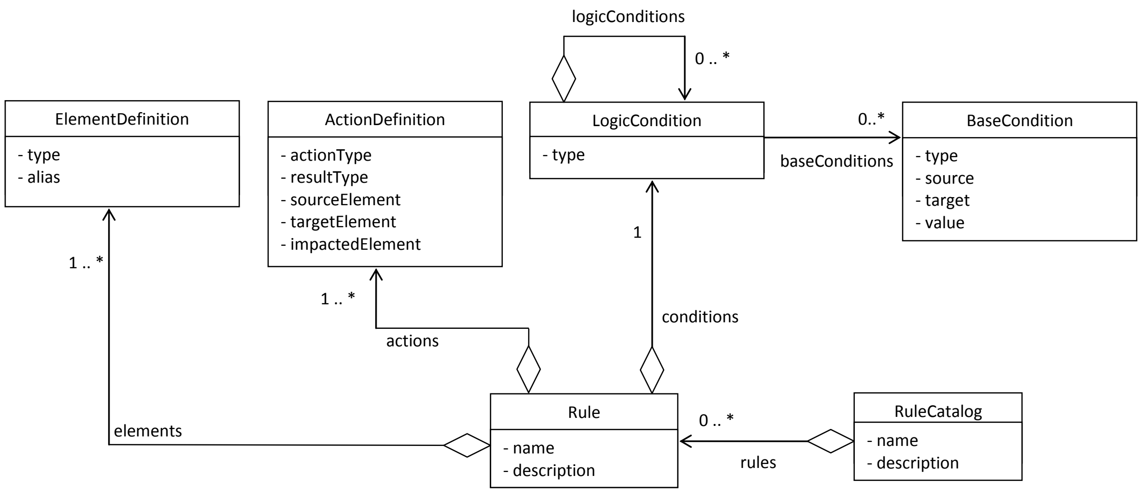


Figure 6.7.: Meta-model for dependency detection rules

Our rules are based on EMF as well and can be imported into the model repository along with the actual software artifacts. The syntax and concept behind our rules was originally published in [Leh10, BLR11, RBFL11] and has since then been extended according to our goals. In contrast to [Leh10, BLR11, RBFL11], five new query operators were added (ValueStartsWith, ValueEndsWith, ModelRelatedTo, ModelUndirectedRelatedTo, ModelIndirectlyRelatedTo), which are explained in the next section.

The reasons why a custom rule-concept was chosen instead of existing, similar concepts, such as OCL [OMG12] for example, are threefold. First, the creation of traceability-links as a result of executing the rules demands for a custom implementation of a rule-processing infrastructure anyway, thus the overhead of our approach is negligible. Second, the EMF-based modeling of our rules allows for their direct integration into EMFStore without requiring further transformations, etc. Moreover, the rules benefit from the validation features supplied by EMF and EMFStore as well as from the versioning support provided by EMFStore. Third, our rules are easier to read and therefore easier to comprehend than typical OCL expression, which in turn results in an improved maintainability and helps creating additional rules must faster.

Finally, we discuss an exemplary rule to illustrate our concepts. Therefore, Listing 6.6 presents a typical dependency detection rule which illustrates the above discussed structure of our rules. It refers back to the example that was already discussed in Section 6.4.3.2 and is illustrated by Figure 6.4 (to briefly recap the example: it describes an "implementation"-dependency between the methods of a class and the methods of an interface when the class implements the interface). Although the basic conditions to uncover such relations where already provided by Listing 6.2 in Section 6.4.3.2, Listing 6.6 as presented in this section displays the complete EMF-based rule in its native XML-form. Moreover, the rule can also be found in Appendix A.

```
<Rule id="TR_Mth_009">
  <Elements alias="e1" type="Method"/>
  <Elements alias="e2" type="Method"/>
  <Elements alias="e3" type="Class"/>
  <Elements alias="e4" type="Interface"/>
  <Conditions type="And">
    <BaseCondition type="ModelRelatedTo" source="e3" target="e4" value="Implements"/>
    <BaseCondition type="ModelDirectParentOf" source="e3" target="e1"/>
    <BaseCondition type="ModelDirectParentOf" source="e4" target="e2"/>
    <BaseCondition type="ValueEquals" source="e1::name" target="e2::name"/>
  </Conditions>
  <Actions actionType="CreateLink" source="e1" target="e2" resultType="Implements"/>
</Rule>
```

Listing 6.6: A rule linking corresponding methods of interfaces and classes.

The *Element-Definition* part of this rule consists of four declarations since it has to access four software artifacts: one *Class*, one *Interface*, and two *Methods*. The actual *Query-Definition* consists of four distinct operations that are nested in an encapsulating *AND*-condition. The first *BaseCondition* checks whether the *Class* "e3" is related to the *Interface* "e4" through an "Implements"-relation. The second and third *BaseCondition* check whether the methods belong to either *Class* "e3" or *Interface* "e4" (the *implemented* interface operation) respectively. The fourth *BaseCondition* checks whether the names of both methods are equal. The *name*-property of the methods is accessed using the scope-operator "::". Finally, the *Action-Definition* part creates an "Implements" traceability link between both methods if all conditions were met.

6.4.4.2. Query Operators

Our rules offer three distinct types of operations which can be used within the *Query-Definition* part. First, there are logical operators for nesting sub-queries that are realized by instances of the *LogicCondition*-class. Secondly, there are operators to access and compare properties of software artifacts that are realized by instances of the *BaseCondition*-class. Thirdly, there are operators to analyze relations between models that are also realized by instances of the *BaseCondition*-class. Table 6.5 below summarizes and briefly explains our query-operators.

Operator	Description
AND	implements the logical *and*-operator
OR	implements the logical *or*-operator
NOT	implements the logical *not*-operator
XOR	implements the logical *exclusive-or*-operator
ValueEquals	compares two values for equality
ValueLesserThan	checks if the value of a property is lesser than another value

ValueGreaterThan	checks if the value of a property is greater than another value
ValueContains	checks if the value of a property contains a certain string
ValueStartsWith	checks if the value of a property starts with a certain string
ValueEndsWith	checks if the value of a property ends with a certain string
ValueSimilarTo	checks if one value is similar to another value. This operation is implemented by the n-gram algorithm [CT94]
ValueNotNull	checks if the value of a property exists and is not null
ModelEquals	checks if two models are equal
ModelRelatedTo	checks if there is a dependency between two models of a certain type under consideration of the direction of the relation
ModelUndirectedRelatedTo	checks if there is a dependency between two models of a certain type while ignoring the direction of the relation
ModelIndirectlyRelatedTo	checks if two models are transitively related through a chain of dependency relations of a certain type
ModelParentOf	checks if one model transitively contains another model
ModelDirectParentOf	checks if one model directly contains another model

Table 6.5.: Overview of all query-operators supported by our rule-concept

6.4.4.3. Definition of Dependency Detection Rules

A rule-based concept like ours demands for an ongoing refinement and addition of rules to cope with new types of software artifacts, development methodologies, and views (see Section 3.3.4.3). Therefore, in this section we present a scheme for defining dependency detection rules to complement our approach. This section stands in a stark contrast to related rule-based works that do not provide any details regarding how the required rules should actually be created.

The general idea of our approach is that, once a source of a possible dependency relation is known, a rule or a set of rules can be establish for recording this dependency. However, it requires more than to simply identify dependencies to formulate strict rules for detecting them in an automated manner. First and foremost, the dependent software artifacts have to be explicitly identified, meaning that their type or class has to be determined. Secondly, the purpose of the relation has to be understood in order to specify the actual type of the relation. For this purpose, the taxonomy and clusters of dependency relations introduced in Section 6.3 come into play and are used as a reference. Thirdly, the conditions under which this dependency occurs must be identified as otherwise no rules can be established for the automated detection of this very dependency. In a final step, the rule should be tested and evaluated by applying it on real software development projects to verify its correctness.

The following presents the step-wise scheme we developed for guiding researchers and developers when creating dependency detection rules to extend the concepts presented in this thesis.

- **Step 1 - Identify the dependency relation.** This step can be accomplished in different ways. Manual analysis of software artifacts and their relations can identify dependencies (see Section 6.4.3), as well as couplings inferred by MSR-based or IR-based approaches can point towards dependencies.
- **Step 2 - Identify the related artifacts.** The artifacts that are involved in the dependency relation have to be explicitly determined. Likewise, their class or type has to be deter-

mined. This step is important for accomplishing the *Element-Definition* part of a rule.

- **Step 3 - Determine the purpose of the relation.** In order to define the actions to be taken by a rule, the type of the traceability link to be create must be defined. The taxonomy of dependency types presented in this thesis can serve as reference for this step. This step assists with creating the *Action-Definition* part of a rule.
- **Step 4 - Define the conditions under which the relation exists.** Once a relation, its type, and the involved artifacts are determined, the conditions under which the relation occurs must be identified. This can be accomplished by manually studying the relation.
- **Step 5 - Transform the conditions into query-statements.** The conditions identified in the previous step must be transformed into proper query-statements using the operators provided by our rules (see Section 6.4.4.2). As a result of this step, the *Query-Definition* part of a rule can be created.
- **Step 6 - Validation.** We propose to first test the rule using artificial test projects containing mock-objects to verify the general correctness and functionality of a rule. Secondly, each rule should be further evaluated during a comprehensive case study.

6.5. Evaluation

After introducing our approach for dependency detection, we have to report on the evaluation of our concepts to outline their applicability. We evaluated our dependency detection approach with the help of five case studies in which different software systems were analyzed for dependencies. This section reports on the research questions, the setup, and the achieved results of our studies. We also compare the results of our approach with those of existing dependency detection approaches. We can show that our approach achieves both better precision and recall and is thus better suited for detecting dependencies between heterogeneous software artifacts.

6.5.1. Setup and Research Questions

In this section we first describe our case study subjects, before we outline our research goals and the metrics and measures that were applied during our studies.

The following briefly presents our case study subjects and provides an overview of the types of software artifacts they are comprised of. Our first test subject is the robot control framework *RSIFramework* developed at the Ilmenau University of Technology, which provides several UML models, an OWL ontology, and a URN goal model [Leh10]. Secondly, we analyzed our own prototype tool *EMFTrace* (see Chapter 9) providing UML models, Java source code, and JUnit test cases [Leh10]. Further case study subjects are supplied by *EMFfit* [Wag10] and *QUARC* [Mot12]. Both tools are stand-alone extensions of EMFTrace and allow for analyzing dependencies between UML models and Java source code [Gup13]. *EMFfit* assists with the transition from requirements to software architectures according to the methodology of Hofmeister *et al.* [HNS05], while *QUARC* implements the *Goal Solution Scheme* (GSS) proposed by Bode [Bod11] and assists with architectural decision making. The fifth case we applied our detection approach on is the CoCoME project [CoC] that implements an enterprise application and consists of Java source code artifacts and different types of UML models.

Unfortunately, we could not use the de facto standard benchmarks for traceability detection approaches as established by the *Center of Excellence for Software Traceability* (COEST) [COE] since they only contain requirements descriptions[1]. Furthermore, most of the offered benchmarks are rather small in size and only provide a few hundred software artifacts, which results in a comparable low figure of traceability links. Hence, we had to refer to other case study subjects providing heterogeneous software artifacts, such as source code, test cases, and different types of UML diagrams. Table 6.6 summarizes our five case study subjects, their size, and the software artifacts they are comprised of to provide an overview of our case study subjects. The *source lines of code* (SLOC) were obtained using *Code Analyzer* [Cod13].

All five software systems were manually analyzed for dependencies to provide an *oracle* or *golden standard* [BGA06] to compare our approach against. Apart from the CoCoME project, this manual analysis was conducted by the developers and researchers who were involved in the development of the systems to ensure a correct detection of the dependency relations. Furthermore, each link was verified by at least two persons. For the CoCoME application this information was obtained by the author of this thesis only. The manual analysis was also supported by the execution of unit tests and the dependency detection features of the Eclipse IDE.

Case Study Subject	**Java Files**	**Java SLOC**	**JUnit Tests**	**#UML Diagrams**
EMFTrace	250	25900	124	4 Component 6 Package 26 Class 4 Use Case 1 Activity
RSI-Framework	na	na	na	2 Sequence 3 Component 1 Composite Structure + 1 OWL Ontology + 1 GRL Goal-model
EMFfit	197	12600	172	1 Component 5 Object 3 Activity
QUARC	210	21500	275	6 Component 8 Activity
CoCoME	171	7300	11	1 Use Case 2 Class 10 Sequence 6 Component 1 Deployment

Table 6.6.: Overview of all cases study subjects

The quality of our approach's results was assessed with the help of the precision and recall metrics. Therefore, the results of the manual analysis ($Results_{MA}$) and the results of the automated analysis ($Results_{AA}$) were compared to obtain the relevant figures for precision (*P*), recall (*R*), and their combined $F_1 - score$, for which the following formulas were applied:

[1] http://coest.org/index.php/resources/dat-sets

$$P = \frac{|Results_{AA} \cap Results_{MA}|}{|Results_{AA}|} \qquad R = \frac{|Results_{AA} \cap Results_{MA}|}{|Results_{MA}|} \qquad F_1 - score = \frac{2 \cdot P \cdot R}{P+R}$$

6.5.2. Results and Discussion

In the following, we discuss the results achieved by our approach in regard to its precision, recall, and $F_1 - score$. The achieved results are also summarized by Table 6.7.

To begin with, it is important to note that all the results obtained during our case study reflect the ability of our approach to detect multiperspective "high-level" dependencies between UML models, JUnit test cases, and Java source code that are not provided by current tools. Rules for the detection of "low-level" dependencies of Java source code (e.g. "code statement A *Initializes* the value of Attribute B", "Method C *Calls* Method D", etc.) were not executed due to the vast amount of resulting dependency relations, which is due to three reasons. First of all, the large amount of mostly code-based "low-level" dependencies ($\geq$ 100K for each case study subject) would have overshadowed the comparably small amount of true multiperspective "high-level" dependencies, which ranged between 900 and 20000, depending on the case study subject. Consequently, the obtained precision and recall would not have reflected the approach's ability of detecting multiperspective dependencies. Second, the large amount of dependencies would have rendered any means of a manual validation of the links impossible, hence preventing any elicitation of precision and recall within reasonable time and with reasonable effort. Third, the neglected and mostly code-based dependencies can be easily obtained by dependency browsers, call graph extractors, etc. that are already integrated in modern IDEs.

Case Study Subject	**Precision**	**Recall**	**F_1-score**
EMFTrace	0.7125	0.8370	0.7697
RSI-Framework	0.9685	0.8575	0.9096
QUARC	0.9723	0.9789	0.9755
EMFfit	0.8876	0.8884	0.8879
CoCoMe	0.8536	0.8077	0.8300

Table 6.7.: Results achieved with our approach

Based on the results obtained for each of the five case study subjects (see Table 6.7 above), our rule-based detection approach achieved a mean precision of 0.8789, a mean recall of 0.8739, and a mean $F_1 - score$ of 0.8745 respectively.

Finally, we compare and discuss our results with those obtained by other IR, MSR, ML, and rule-based (RB) approaches, which is also summarized by Table 6.8 (for a general discussion of these techniques see Section 3.3.4).

As indicated by Table 6.8, the F_1-scores of rule-based approaches, including ours, are much better than those of IR or MSR-based approaches; in average by a factor of two. Thus, a rule-based dependency detection approach as presented in this thesis is able to provide more stable and reliable results when compared to other IR, ML, and MSR-based approaches. Likewise, our approach was evaluated with the help of five different case study subjects and we obtained similar figures for precision and recall for each of them.

When compared to the work of Jirapanthong and Zisman [JZ09], our case studies did not just cover UML models but also Java source code, JUnit test cases, as well as OWL and GRL

Approach	**Method**	**Avg. Precision**	**Avg. Recall**	F_1**-score**
Antoniol *et al.* [ACC+02]	IR	0.500	0.480	0.489
Marcus and Maletic [MM03]	IR	0.274	0.822	0.411
De Lucia *et al.* [LFOT07]	IR	0.460	0.700	0.555
Grechanik *et al.* [GMP07]	ML	0.711	0.676	0.693
Kagdi [Kag08]	MSR	0.815	0.165	0.274
Jirapanthong and Zisman [JZ09]	RB	0.853	0.833	0.843
Our Approach	RB	0.879	0.874	0.875

Table 6.8.: Comparison of traceability detection approaches

models stemming from five different projects. This means that our approach was evaluated with a greater variety of software artifacts but still produced slightly better results. In contrast to their work, we also provide a scheme for defining dependency detection rules and we take into account dependencies stemming from four different sources. Hence, we consider our approach as a suitable asset to the impact analysis approach proposed in this thesis.

6.6. Critical Discussion and Limitations

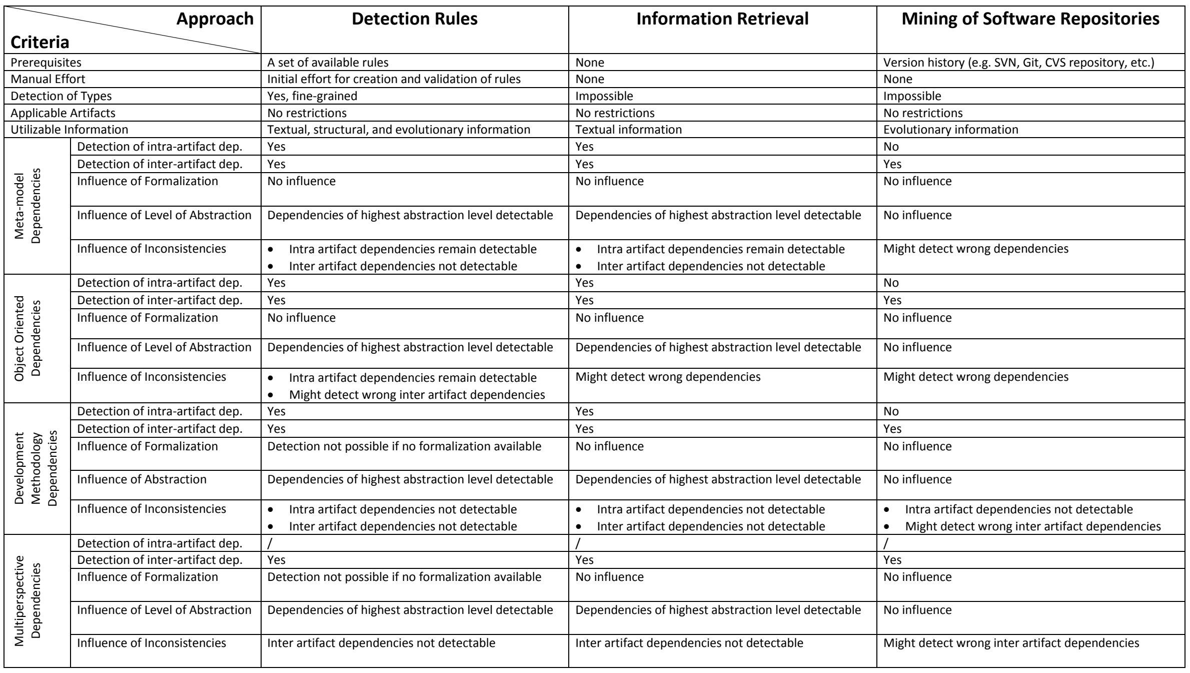
In this final section we discuss potential limitations of the proposed dependency detection approach, for which we analyze the implications of missing and incorrect dependency detection rules. Additionally, we thoroughly compare our approach to MSR and IR-based approaches and analyze its space and time complexity.

6.6.1. Comparison to Existing Approaches

To allow for a final verdict on our proposed dependency detection approach, we summarize how it compares to existing MSR and IR-based techniques in regard to the detection of meta-model dependencies, object oriented dependencies, design methodology dependencies, and multiperspective dependencies. Therefore, we summarize and conclude on the findings of Section 6.4.3. Furthermore, we analyze and compare all three approaches for their ability of detecting inter- and intra-artifact dependencies, and we outline the preconditions for applying the approaches. The results of the upcoming discussion are summarized in a tabular manner by Figure 6.8.

The most notable difference between the approaches are the sources of dependencies that can be analyzed by them. While IR-based approaches are restricted to the analysis of textual information and MSR-based approaches are restricted to the investigation of evolutionary couplings, our rule-based approach currently analyzes structural and textual information of software artifacts, as well as existing dependencies between them, therefore covering a wider range of potential dependency relations. Moreover, future work could expand our approach to take evolutionary couplings into consideration as well. When compared to MSR-based approaches, another benefit of our approach is its capability of detecting inter- and intra-artifact dependencies, whereas MSR techniques cannot detect dependencies within single files with reasonable effort.

The main limitation of our approach is its dependence on a certain level of formalization,



Criteria		Approach: Detection Rules	Information Retrieval	Mining of Software Repositories
Prerequisites		A set of available rules	None	Version history (e.g. SVN, Git, CVS repository, etc.)
Manual Effort		Initial effort for creation and validation of rules	None	None
Detection of Types		Yes, fine-grained	Impossible	Impossible
Applicable Artifacts		No restrictions	No restrictions	No restrictions
Utilizable Information		Textual, structural, and evolutionary information	Textual information	Evolutionary information
Meta-model Dependencies	Detection of intra-artifact dep.	Yes	Yes	No
	Detection of inter-artifact dep.	Yes	Yes	Yes
	Influence of Formalization	No influence	No influence	No influence
	Influence of Level of Abstraction	Dependencies of highest abstraction level detectable	Dependencies of highest abstraction level detectable	No influence
	Influence of Inconsistencies	• Intra artifact dependencies remain detectable • Inter artifact dependencies not detectable	• Intra artifact dependencies remain detectable • Inter artifact dependencies not detectable	Might detect wrong dependencies
Object Oriented Dependencies	Detection of intra-artifact dep.	Yes	Yes	No
	Detection of inter-artifact dep.	Yes	Yes	Yes
	Influence of Formalization	No influence	No influence	No influence
	Influence of Level of Abstraction	Dependencies of highest abstraction level detectable	Dependencies of highest abstraction level detectable	No influence
	Influence of Inconsistencies	• Intra artifact dependencies remain detectable • Might detect wrong inter artifact dependencies	Might detect wrong dependencies	Might detect wrong dependencies
Development Methodology Dependencies	Detection of intra-artifact dep.	Yes	Yes	No
	Detection of inter-artifact dep.	Yes	Yes	Yes
	Influence of Formalization	Detection not possible if no formalization available	No influence	No influence
	Influence of Abstraction	Dependencies of highest abstraction level detectable	Dependencies of highest abstraction level detectable	No influence
	Influence of Inconsistencies	• Intra artifact dependencies not detectable • Inter artifact dependencies not detectable	• Intra artifact dependencies not detectable • Inter artifact dependencies not detectable	• Intra artifact dependencies not detectable • Might detect wrong inter artifact dependencies
Multiperspective Dependencies	Detection of intra-artifact dep.	/	/	/
	Detection of inter-artifact dep.	Yes	Yes	Yes
	Influence of Formalization	Detection not possible if no formalization available	No influence	No influence
	Influence of Level of Abstraction	Dependencies of highest abstraction level detectable	Dependencies of highest abstraction level detectable	No influence
	Influence of Inconsistencies	Inter artifact dependencies not detectable	Inter artifact dependencies not detectable	Might detect wrong inter artifact dependencies

Figure 6.8.: Comparison of dependency detection approaches regarding the detection of meta-model, object oriented, design methodology, and multiperspective dependencies

whereas IR and MSR-based approaches are independent of any formalization of the software artifacts under scrutiny. However, most software artifacts adhere to a strict specification which mitigates the impact of this limitation to a certain extent. Likewise, our rule-based approach is more prone to be affected by software artifacts having a different level of abstraction. In contrast, potential inconsistencies have the same negative impact on all detection approaches as they result in the detection of false-positives.

Arguably the most important advantage of our approach on the other hand, is its capability of detecting the types of the dependency relations, whereas IR and MSR-based approaches are not able to do so. It is, however, most crucial to be able to distinguish between the different types of dependencies when performing impact analysis [LFR13], regression testing [FLR14], etc.

The last major difference between our approach and others is the manual effort that is required for creating the initial set of dependency detection rules. However, when putting the effort for creating the rules in relation to the effort for dealing with the higher quotas of false-positives as detected by IR and MSR-based approaches [Bod11], the overhead decreases significantly. Additionally, while the tremendous effort for checking for false-positives (up to 80%, see Table 6.8) accounts every time IR or MSR-based approaches are applied, the effort for creating the rules accounts only once. Thus, our approach requires overall less manual effort.

6.6.2. Correctness of the Dependency Detection Rules

The correctness of the detected dependency relations is of great importance for the impact analysis approach presented in this thesis. Incorrect (or false-positive) dependencies may derail the impact analysis and result in impact sets that are useless to developers. In the worst case, these incorrect dependencies might even mislead developers by overestimating the overall impacts of changes. Thus, the correctness of the impact estimations depends on the correctness of the recorded dependency relations, which in turn directly depends on the correctness of the applied dependency detection rules. We identified three potential threats concerning the proposed dependency detection rules which we are discussing in the following:

- A rule detects a dependency where there is none.
- A rule fails to detect an existing dependency.
- A rule determines the wrong type of a dependency.

The first case is either triggered by wrong conditions encoded within a rule, which is due to misunderstandings of the creator of the rule, or by an over-specialization of the rule for a given software system (i.e. tailoring the rule according to the attributes of this specific software). In either case, the problem can be solved by manually adjusting the (defect) query condition(s).

The second problem arises when the reasons why a dependency exists between two software artifacts were not understood correctly and hence were not correctly transformed into suitable query conditions for detection rules. Consequently, a thorough analysis of the conditions leading to the dependency is required in order to correct the rule. Depending on the source of the dependency, it might help to inspect this source under consideration of the dependency to identify suitable query-conditions, e.g. studying an artifacts' meta-model if the dependency arises from the meta-model. Additionally, our scheme for creating dependency detection rules as established in Section 6.4.4.3 can also assist developers when fixing broken rules.

The third problem points towards an incorrect specification of the *ActionDefinition*-part of a rule by the developer who created the rule. This problem can be easily addressed by adjusting the type of the dependency relation to be created in the *ActionDefinition*-part of the rule. This adjustment is supported by our taxonomy of dependency types as introduced in Section 6.3.1 that assists with determining the type of a relation based on its purpose in a step-wise manner.

Despite these potentials threats, our detection rules achieved reliable results during their application on five different case study subjects (see Section 6.5.2). Therefore, our dependency detection concept provides a solid basis for the impact analysis approach presented in this thesis.

6.6.3. Completeness of the Dependency Detection Rules

Apart from detecting correct dependency relations only, it is also important to elicit all the dependencies that exist between the software artifacts to enable a later impact analysis. In contrast to incorrect detection rules, the implications of missing detection rules are twofold. First of all, missing (explicit) dependencies hamper a developer's ability to understand the software system since relations to dependent parts and concepts are hidden from him [RBFL11]. Secondly, missing dependency relations decrease the effectiveness of change impact analysis approaches.

Unfortunately, we cannot discuss the completeness of our set of detection rules in a theoretical manner. However, to mitigate the implications of missing dependencies, we demonstrated the detection of dependencies stemming from four different sources, namely meta-models, object oriented concepts, design methodologies, and the usage of different perspectives. By reasoning about the dependencies stemming from these sources, we provide better and more comprehensive coverage of dependency relations than existing works. Moreover, we established an easy-to-follow guideline for the creation of additional dependency detection rules, if a missing rule should be identified (see Section 6.4.4.3).

As already pointed out in the previous section, the set of detection rules that accompanies this thesis has been tested, refined, and complemented during its application on five different software systems. During each of these studies our rules achieved a recall of at least 80% while maintaining a similar high figure for precision (see Section 6.5.2). Thus, our detection rules and their underlying concepts provide a solid basis for multiperspective dependency detection.

6.6.4. Complexity of the Approach

A critical discussion of our dependency detection approach also demands for a discussion of its efficiency in terms of space and time requirements. Therefore, we discuss these figures in a theoretical manner using the Bachmann–Landau notation ("big O notation").

The theoretical worst case space complexity $s(n)$ of our approach can be computed as follows for n artifacts whose dependencies shall be detected. Each dependency detection rule consists of a set of conditions that query model elements, whereas each condition can only query two artifacts at once (i.e. a source and a target, see 6.4.4.1). Therefore, for each condition there is a maximum of $n \times n$ tuples being computed and checked. As all the conditions of a rule are executed in a sequential manner, the amount of conditions to be executed has no influence on the required space. Hence, each rule has a maximum space requirement of $O(n^2)$. Now as the detection rules are executed in a sequential manner as well, the worst case space complexity of a single rule equals the worst case space complexity of our entire approach, thus:

$$s(n) = O(n^2) \tag{6.1}$$

The theoretical worst case time complexity $t(n)$ of our approach can be computed as follows for n artifacts whose dependencies shall be detected. As stated above, for each condition encoded within a rule a maximum of $n \times n$ tuples are being checked, which results in n^2 operations per condition. Consequently, a rule containing *k* conditions requires $k \cdot n^2$ operations, where k is independent of n. Now let r denote the total number of rules to be executed, where r is also independent of n. Hence, a total amount of $r \cdot k \cdot n^2$ operations is required. Therefore, the total worst case time complexity of executing these rules can be estimated as follows:

$$t(n) = O(n^2) \tag{6.2}$$

Although these figures seem high at first, a typical rule contains between 2 to 5 conditions, which reduces its execution time to a couple of milliseconds for the cases discussed in Section 6.5.1. Moreover, due to the sequential processing of the conditions, no condition will require the n^2 operations in practice, as model elements are constantly being filtered by previous conditions, thus vastly reducing the amount of tuples to be checked. Furthermore, our current rule-processing infrastructure (see also Section 9.3) can be replaced by optimized industrial query-processing engines that would therefore significantly speed up the processing of rules.

6.6.5. Addressing (Textual) Inconsistencies

Despite our previous discussions of the influence of inconsistencies on the detection of (multiperspective) dependencies, we outline possible future work to deal with textual inconsistencies introduced by the usage of synonyms[2] and homonyms[3] by different stakeholders during software development [TLCvV11b]. This challenge can be tackled by blending concepts like ontologies or semantic wikis into the rule-based approach proposed in this thesis. The different vocabulary introduced by different stakeholders can be resolved by utilizing an ontology [Bod11] or a word similarity database [TLL14] for resolving the possibly inconsistent terms. Consequently, the application of such a concept requires the establishment of an ontology prior to the dependency analysis and the integration of the resulting ontology into the dependency detection process. The proposed rule-concept could be extended with a new type of query operation to compare text-based values with the help of the resolving ontology, such as for example a “ValueMatchesSynonym”-operation. Similar concepts were already proposed by Bode [Bod11] to bridge between architectural issues and requirements, which therefore could be reused. However, not all inconsistencies can be resolved through such an approach, which furthermore requires manual effort for creating and maintaining the ontology (or what else is used to establish the mapping). For these reasons, such an extension is currently omitted from this thesis.

6.7. Summary

In this chapter we discussed how dependencies between different types of software artifacts can be uncovered, recorded as traceability links, and classified according to their type to allow for a later impact analysis. Therefore, we presented a meta-model providing a traceability link model to explicitly store the identified dependencies. We introduced a taxonomy for dependency types that is based on the idea of distinguishing between different types of dependencies by their purpose to address the problem of dependency classification. The proposed taxonomy consists of clusters of similar dependency types which are arranged in a hierarchical manner and enable a step-wise refinement of the dependency relations according to their purposes. As the main contribution of this chapter, we introduced a rule-based approach for the automated detection of dependency relations between heterogeneous software artifacts. We therefore discussed where the dependencies stem from and how they can be detected. We illustrated the structure of our dependency detection rules and presented a scheme for the definition of additional detection rules. Finally, we conducted an evaluation of our detection approach with the help of five case studies and discussed the critical points and limitations of our concept.

In conclusion, the research presented in this chapter contributes towards **Goal 7** and **Goal 8** by introducing a meta-model for dependencies, supplying means for classifying dependencies according to their relation type, and by providing an approach for detecting dependency relations between heterogeneous types of software artifacts.

[2]“a word having the same or nearly the same meaning as another [...]”, see: http://dictionary.reference.com/browse/synonym

[3]“a word pronounced the same as another but differing in meaning [...]”, see: http://dictionary.reference.com/browse/homonym

7. Change Comprehension

When performing impact analysis, developers first of all require a solid understanding of the changes to be applied on the software. This, however, is not sufficiently dealt with by current research on impact analysis [Leh11a]. Therefore, based on our refined goals in Section 4.2 and **Goal 4** in particular, we present an approach to model arbitrary change operations and to classify them according to their type to alleviate this shortcoming of current research. We present a meta-model for change operations that extends the concept of atomic and composite changes and is implemented using the Eclipse Modeling Framework (EMF) [EMFa]. Furthermore, we introduce a taxonomy that is build upon our modeling approach and allows for the correct classification of changes. Parts of the research presented in this chapter were already discussed in [LFR12,LFR13,FLR14]. Novel contributions of this thesis are *a)* our meta-model for change operations, *b)* strict definitions of the terms atomic and composite changes, and *c)* detailed discussions of how more complex refactoring scenarios can be modeled using our approach.

This chapter is organized as follows. First, in Section 7.1 we present our concept and meta-model for the modeling of arbitrary change operations. Section 7.1.3 then elaborates on how typical refactoring activities can be modeled using our approach for the modeling of change operations. Finally, Section 7.2 presents a taxonomy for the classification of change operations as one precondition for our proposed impact analysis approach.

7.1. Modeling of Change Operations

In Section 3.4.1 we investigated how approaches for regression testing, impact analysis, etc. model different types of change operations and refactoring activities. Our analysis revealed that the concept of atomic and composite change operations provides a solid base for the modeling of change operations. However, we also noticed that current approaches suffer from three limitations, namely an ambiguous set of atomic operations, the inability of comprising composite operations of other composite operations (see discussions in Section 3.4.1), and the absence of strict definitions of composite operations. Therefore, we have to refine these approaches by introducing a modeling approach for change operations that can be applied for change impact analysis and that allows to model arbitrary types of changes and entire refactoring scenarios.

7.1.1. Atomic and Composite Operations

When modeling change operations, it requires a fixed set of reference-operations to compose the actual changes [LFR12]. Such a set of reference-operations can be supplied by the concept of atomic and composite changes. An atomic operation, as the name suggests, cannot be further broken down into other operations, whereas composite operations are comprised of other types of change operations. However, it requires strict definitions for how both atomic and composite operations can be modeled. Therefore, in the following we extend existing works on the modeling of changes and introduce strict definitions for both types of change operations.

- **Atomic Change:** a change activity that is comprised of exactly one non-interruptible operation. Each atomic operation can only involve a maximum of two software artifacts.
- **Composite Change:** a change activity that is comprised of at least two atomic or composite change operations. The sequence of operations is potentially interruptible. A composite operation may involve at least two or more software artifacts.

Our definition of atomic operations stands in contrast to the definition of Engels *et al.* [EHKG02] because the atomic *add*-operation must address two software artifacts, namely the artifact to be added and the artifact it should be added to. Likewise, our definition of composite changes is slightly different than the one of Mäder [MÖ9] because in our approach composite operations can be comprised of other composite operations as well.

However, as proposed by Mäder *et al.* [MRP06b, MÖ9, MÏ0], we utilize the following set of atomic operations as our base for modeling further change operations:

$$OP_{\text{atomic}} := \{add_{\text{node}}, delete_{\text{node}}, add_{\text{edge}}, delete_{\text{edge}}, update\_property\}$$

The *add*- and *delete*-operations either add or delete edges or nodes to a graph (the software system), whereas the *update_property*-operation on the other hand modifies a property of a node or edge, such as its name or visibility for example.

The above presented definitions apply to the logical structure of a graph and hence to any type of software artifact under consideration by this thesis since any piece of source code can be mapped to an AST and all modeling languages are built upon graph-based meta-models [DST11,LFR12]. However, depending on the context and the artifacts to be changed, the add_{node} and add_{edge} operations may be represented by the same real change activity, which is then simply referred to as add-operation. The same applies to both cases of the *delete*-operation. Hence, in the following we will refer to them as simply *add* and *delete*.

Furthermore, the presented atomic operations can be combined into composite change operations. In contrast to previous works analyzed in Section 3.4.1, we allow composite operations to be modeled by other composite operations as well. Therefore, in our approach composite operations may consist of sequences of either atomic or other composite change operations. To provide additional support for impact analysis, we define a set of the most common composite changes as encountered in ever-day software engineering tasks. This set is composed of composite changes as proposed in [FG06, MÖ9] and complemented with the yet missing *swap*-operation [LFR12]. Hence, we obtain the following set of basic composite operations:

$$OP_{\text{composite}} := \{move, replace, split, merge, swap\}$$

Moreover, existing works lack a precise and formal definition of those operations. Therefore, in the upcoming sections we also discuss how each composite operation can be modeled using our concept. The operation $P(x)$ thereby refers to the parent-node of *x*, i.e. its containing element. For example, when P is applied on a method it returns the class containing the method.

7.1.1.1. Move-operation

The move-operation allows to move nodes or entire sub-graphs within a graph and to attach them to other nodes. We support two versions of the move-operation. First, developers might want to move an entire sub-graph x to another node y, which is displayed by Figure 7.1. A typical example for such an operation is when a class is moved to another package, as all the methods and attributes contained within the class are also moved.

$$move(x, y) := delete_{\text{edge}}(x, P(x)), add_{\text{edge}}(x, y).$$

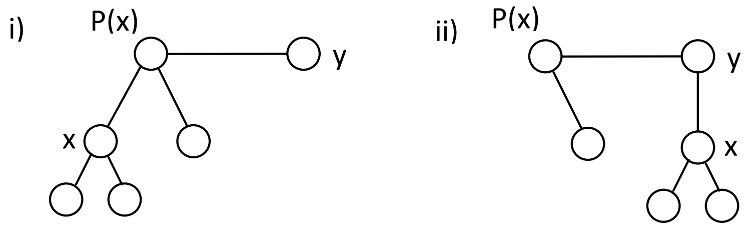


Figure 7.1.: The structure of the graph before (i) and after (ii) applying the *Move*-operation

We also support to only move the node x to another node y, while leaving all the potential child nodes x_i of x in place. This operation is visualized by Figure 7.2. For example, this operation might be applied when one is moving an attribute up to the base class of a class hierarchy.

$$move'(x, y) := \bigwedge_{i=0}^{n} move(x_i, P(x)), move(x, y).$$

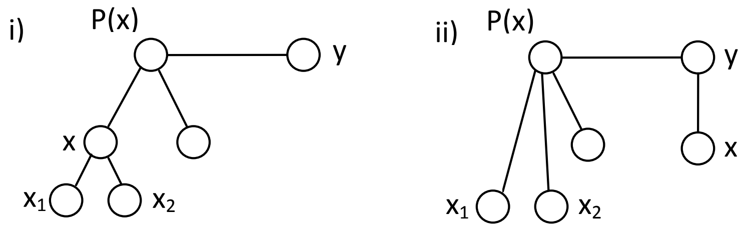


Figure 7.2.: The structure of the graph before (i) and after (ii) applying the *Move*-operation

7.1.1.2. Replace-operation

The replace-operation allows to replace parts of a graph by another sub-graph. For this operation we also support two cases. First, developers might want to replace the entire sub-graph x by another sub-graph y, which is displayed by Figure 7.3. A practical example would be when one is overwriting a method that was inherited from the superclass.

$$replace(x, y) := delete_{\text{node}}(x), move(y, P(x)).$$

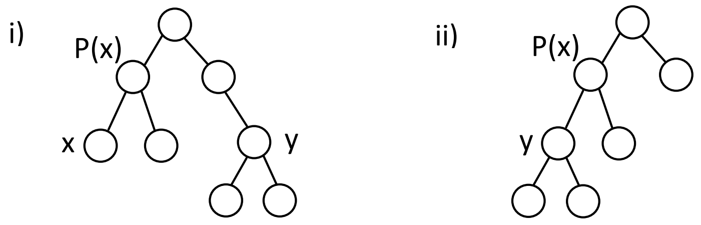


Figure 7.3.: The structure of the graph before (i) and after (ii) applying the *Replace*-operation

Secondly, we also support replacing the node x by node y, while leaving all the child nodes x_i of x in place, which is displayed by Figure 7.4.

$$replace'(x,y) := \bigwedge_{i=0}^{n} move(x_i, y), replace(x,y).$$

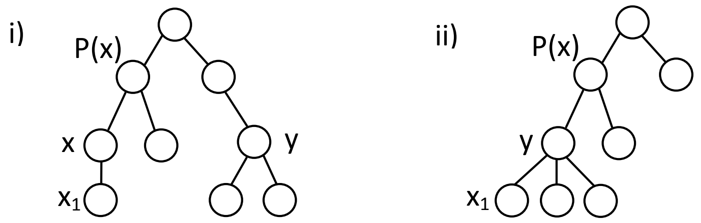


Figure 7.4.: The structure of the graph before (i) and after (ii) applying the *Replace*-operation

7.1.1.3. Split-operation

The split-operation allows to divide a graph into a set of graphs, which is shown in Figure 7.5. It creates a set of *n* nodes ($n \in \mathbb{N}, n \geq 2$) which are of the same type as x and moves all child elements y of x to the respective new node x'_i. Each tuple (s_a, d_b) denotes that the s_a-*th* sub-graph of the node x should be moved to the d_b-*th* sub-graph of the resulting set x'.

$$split(x, n, (s_0, d_0) \dots (s_m, d_m)) := \bigwedge_{i=0}^{n} add_{\text{node}}(x'_i, P(x)), \bigwedge_{j=0}^{m} move(y_{s_j}, x'_{d_j}).$$

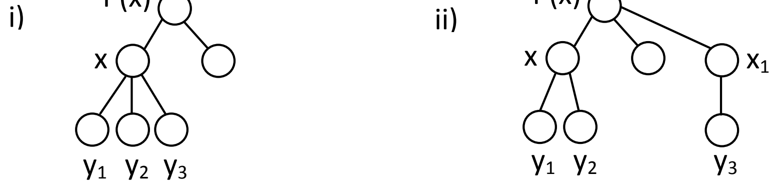


Figure 7.5.: The structure of the graph before (i) and after (ii) applying the *Split*-operation

An example for such an operation is the extraction of a class from another class. A new class is created and all the methods and attributes that should be extracted are moved to the new class.

7.1.1.4. Merge-operation

The merge-operation allows to merge several graphs into one graph, which is displayed by Figure 7.6. It is the inverse operation to *split* and bundles n entities $(n \in \mathbb{N}, n \geq 2)$ of the same type into one, where y_{i_j} is the j-*th* sub-graph of x_i. For example, a merge-operation is applied when a set of rather similar classes are merged into a new base class.

$$merge(x_0 \ldots x_n) := \bigwedge_{i=1}^{n} (\bigwedge_{j=0}^{m} move(y_{i_j}, x_0)), \bigwedge_{i=1}^{n} (delete_{\text{node}}(x_i)).$$

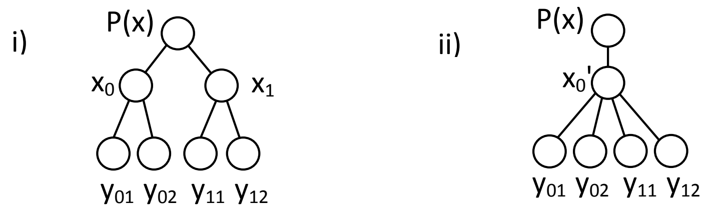


Figure 7.6.: The structure of the graph before (i) and after (ii) applying the *Merge*-operation

7.1.1.5. Swap-operation

Finally, the swap-operations allows for exchanging two nodes or even entire graphs by one another. The case of swapping entire sub-graphs is illustrated by Figure 7.7 below. This operation can be applied when one is exchanging methods between classes due to refactorings.

$$swap(x, y) := move(x, P(y)), move(y, P(x)).$$

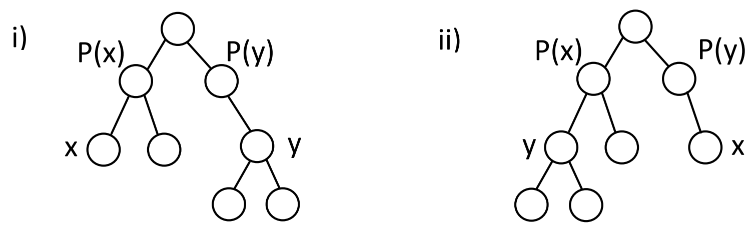


Figure 7.7.: The structure of the graph before (i) and after (ii) applying the *Swap*-operation

Secondly, we also support exchanging nodes only, which attaches the potential child nodes x_i of x to y, and vice versa. This case is displayed by Figure 7.8.

$$swap'(x, y) := \bigwedge_{i=0}^{n} move(x_i, y), \bigwedge_{j=0}^{m} move(y_j, x), swap(x, y).$$

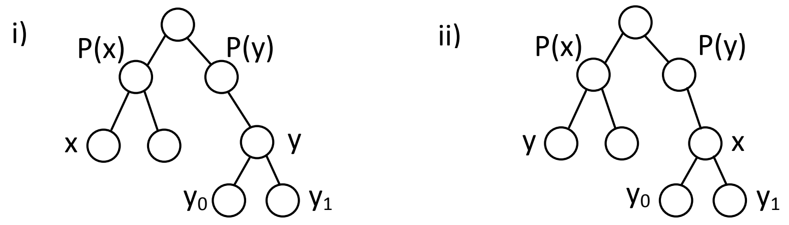


Figure 7.8.: The structure of the graph before (i) and after (ii) applying the *Swap*-operation

7.1.2. A Meta-model for Change Operations

To address **Goal 4** and to implement the above presented concept of atomic and composite operations, we need to establish a meta-model for change operations. This meta-model in turn should allow for the modeling of real change operations for impact analysis tasks. Hence, we provide an EMF-based meta-model for changes that is integrated into the EMF-based repository, along with all software artifacts and their dependencies (see chapters 5 and 6). Our meta-model for change operations is illustrated by Figure 7.9 and is discussed in the following.

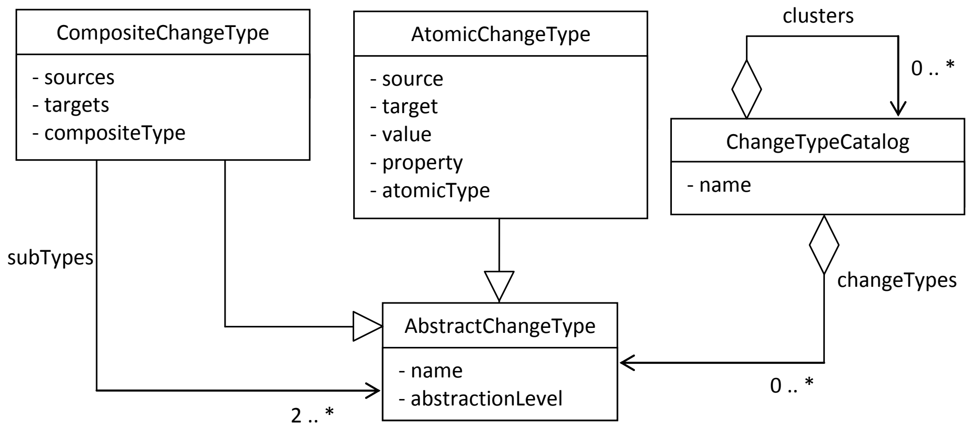


Figure 7.9.: Our EMF-based meta-model for change operations

First, we have to provide means to model the atomic and composite operations, for which we introduce the classes *AtomicChangeType* and *CompositeChangeType*. While atomic operations can address a maximum of two software artifacts, composite operations can address more than two at once, for example when merging multiple elements into one. Consequently, each atomic operation may have only one source artifact and only one target artifact, whereas a composite operation may have multiple. The *AtomicChangeType*-class further possesses optional *value* and *property* fields to model the *update_property*-operation. The composition of composite changes is expressed through the *subTypes*-reference of the *CompositeChangeType*-class.

In order to enable the grouping of similar change types and the grouping of change types per type of software artifact, we furthermore introduce the container class *ChangeTypeCatalog*. This container class allows to create arbitrary sub-clusters of change operations, which is required to provide researchers and developers with a catalog of change types to ease the application of our concept in practice (e.g. for grouping all the changes that are applicable on Java classes). A cutout of an exemplary change catalog is illustrated in Figure 9.4 in Section 9.2.3.

7.1.3. Modeling of Refactoring Activities

In this section we discuss how typical refactoring activities can be modeled using the approach of atomic and composite change operations as presented in the previous sections. Furthermore, we illustrate how composite operations can be instantiated by sets of real atomic and composite operations. We demonstrate this using the following three refactoring scenarios:

- Scenario 1: Renaming a method.
- Scenario 2: Extracting a sub-class from an existing class.
- Scenario 3: Moving an attribute up to the common base class.

The following discussions are based on the code snippet illustrated by Listing 7.1 which presents a simple code scenario consisting of the two Java classes *Vehicle* and *Car*.

```
package vehiclepackage;

public class Vehicle
{
        public enum TireType
        {
                UNKNOWN,
                WINTER,
                SUMMER
        };

        private int numTires;
        private float tireDiameter;
        private float tireWeight;
        private TireType tireType;

        public void steer();
        public void move();
}

public class Car extends Vehicle
{
        private int numPassengers;

        public void startEngine();
        public void stopEngine();
}
```

Listing 7.1: Example Java source code scenario

7.1.3.1. Scenario 1: Renaming a method

If the method *steer()* of the *Vehicle*-class shall be renamed to *steerVehicle()*, the resulting change operation can be modeled using the atomic *update_property*-operation as follows:

1. update_property('Vehicle::steer::name', 'steerVehicle')

The *steer*-method as well as its *name*-property are accessed using the scope-operator ("::").

7.1.3.2. Scenario 2: Extracting a sub-class from an existing class

The class *Vehicle* contains many attributes that could be outsourced to a separate *Tire*-class which would then be responsible for managing all tire-related data. Hence, a sub-class is extracted from *Vehicle*, which is modeled using a *Split*-operation that is instantiated as follows:

1. add('Tire', 'vehiclepackage')
2. move('TireType', 'Tire')
3. move('tireDiameter', 'Tire')
4. move('tireWeight', 'Tire')
5. move('tireType', 'Tire')

To complete the refactoring, the following operation is afterwards applied on the *Vehicle*-class:

1. add('Tire[] tires', 'Vehicle')

7.1.3.3. Scenario 3: Moving an attribute up to the base class

The integer property *numPassengers* of the class *Car* could be moved to the base class *Vehicle* as it will be required for any type of vehicle. This *move*-operation is instantiated as follows:

1. delete('numPassengers', 'Car')
2. add('numPassengers', 'Vehicle')

7.2. Classification of Change Operations

Next to the explicit support for modeling change operations, our impact analysis approach also requires detailed information on the purposes and structure of the changes to be applied. For the concepts presented in this thesis, it is therefore necessary to classify change operations according to their type of operation, their scope, and their structure. This classification can only be supplied by a taxonomy for change operations which is discussed in the following.

A potentially suitable taxonomy was already established by Mens *et al.* [MBZR03] and was later refined by Buckley *et al.* [BMZ$^+$05]. The proposed taxonomy is comprised of four dimensions of changes, namely the *System properties (what)*, the *Object of change (where)*, the *Temporal properties (when)*, and the *Change support (how)*. Based on this taxonomy information on the types of changes can be derived from the *Change support* dimension, whereas the scopes of the changes can be obtained from the *Object of change* dimension. In contrast to our work though, the authors distinguish between structural changes and semantic changes when referring to the "types of changes". This distinction, however, suffers from the same problems as the approach of Gupta *et al.* [GSC10] that was discussed in Section 3.4.2 and that distinguishes between functional, behavioral, structural, and logical changes. Consequently, the resulting classification is ambiguous and thus less useful for change impact analysis tasks.

Hence, it requires a more sophisticated taxonomy for classifying change operations based on their types, scope, and structure to accomplish **Goal 4** and to provide the required support for the impact analysis approach presented in this thesis. Therefore, we propose a different taxonomy of change operations that is comprised of three dimensions [LFR12]: the *scope* of a change, the *abstraction level* of a change, and the *structure* of a change. In the remainder of this section we discuss each dimension and introduce the criteria each dimension is comprised of.

The structural dimension of the proposed classification is supplied by the distinction between atomic and composite changes and by providing concrete sets of change types for each category (see Section 7.1.2). We express this information with the help of the two criteria *Composition Type* and *Type of Operation*. The *Composition Type* denotes whether a change is of atomic character (atomic change) or it is comprised of other change operations (composite change). The *Type of Operation* reflects the actual type of a change operation, such as *add* or *merge*. While the *Composition Type* provides the information if a change operation is comprised of other sub-operations, the *Type of Operation* provides the explicit information *how*. Although this seems like an overlapping or redundancy at first, we consider both criteria as important because the actual type of an operation is not always unambiguous, thus having both criteria provides easy clarification for developers applying the concept.

To address the scopes of the changes to be modeled, we introduce the criterion *Scope of Change*. With the help of this criterion, one is able to provide information on the context of a change, such as if the change applies to architectural models, source code statements, test specifications or other types of software artifacts. This criterion is especially important when developers are evaluating different impact analysis approaches for their support of multiperspective analysis, as the scopes of the supported operations may turn out to be a decisive criterion in this regard.

Finally, we introduce the criterion *Abstraction Level* to our taxonomy to allow for distinguishing between generic and concrete change operations. By the term *generic* change operations we refer to the change operations as discussed in Section 7.1 of this chapter, as we did not discuss explicit examples like “move class A to package B”. Instead, we were only talking about generic “move” operations. In contrast, a *concrete* change operation is tailored for a specific software artifact, such as the refactoring operations we discussed in Section 7.1.3 for example.

The following listing summarizes our criteria for classifying change operations that comprise our taxonomy [LFR12]:

- **Abstraction Level:** Generic, Concrete.
- **Composition Type:** Atomic, Composite.
- **Type of Operation:** Add, Delete, Update, Move, Merge, Split, Replace, Swap.
- **Scope of Change:** Requirements, Architecture, Code, Test, Configuration, etc.

7.2.1. Classification of Refactoring Activities

The following section illustrates how the presented taxonomy can be utilized for classifying refactoring activities proposed in related work in order to assist with change impact analysis. The purpose of this classification is to outline the usefulness of our taxonomy when comparing different types of change operations and to outline its feasibility when determining the types of the changes for a later impact analysis. Therefore, we classify exemplary refactoring operations that were proposed in related work and the refactoring activities discussed in Section 7.1.3. The results of this classification process are also summarized by Table 7.1.

Change	**Source**	**Abstraction Level**	**Composition Type**	**Type of Operation**	**Scope of Change**
move method	[Fow99]	generic	composite	move	code
move method to superclass	[VGSMD03]	generic	composite	move	code
replace code statement	[VGSMD03]	generic	composite	replace	code
split state machine	[SPLTJ01]	generic	composite	split	architecture
replace transition	[SPLTJ01]	generic	composite	replace	architecture
merge transitions	[SPLTJ01]	generic	composite	merge	architecture
merge states	[BSF03]	generic	composite	merge	architecture
replace element	[vdWvdH02]	generic	composite	replace	architecture
rename method	Sec. 7.1.3.1	concrete	atomic	update	code
split class	Sec. 7.1.3.2	concrete	composite	split	code
move attribute to superclass	Sec. 7.1.3.3	concrete	composite	move	code

Table 7.1.: Classification of the discussed changes according to our taxonomy

The studied works can be characterized as follows. Fowler [Fow99] proposes the *movement* of methods in between classes as a refactoring step to improve the structure and understandability of source code. In a similar fashion Van Gorp *et al.* [VGSMD03] utilize *move*-operations to pull methods up to a superclass to limit the redundancy of code clones, which is similar to our third scenario discussed in Section 7.1.3. Moreover, the authors propose the extraction of methods from blocks of code statements to *replace* them by method calls.

Sunyé *et al.* [SPLTJ01] are concerned with refactorings of UML models to increase their extensibility and therefore propose, for instance, to extract sub-states from UML state machines by *splitting* them into a set of states and *replacing* or *merging* UML state machine transitions. Boger *et al.* [BSF03] propose a refactoring browser for UML models that also supports refactorings of state machines, including the *merging* of states.

Finally, the work of Westhuizen and Hoek [vdWvdH02] is concerned with understanding architectural evolution and is especially focused on the identification and comprehension of *replace*-operations applied on architectural elements during the ongoing development process.

The results of this classification as illustrated by Table 7.1 show that one is able to precisely classify change operations for a later impact analysis with the help of our taxonomy. Our taxonomy allows to extract and explicitly record the required information in a structured manner and therefore supports the comprehension of changes.

7.3. Critical Discussion and Limitations

Finally, we discuss the two critical points of our approach for the modeling of change operations, namely its completeness and its applicability for real software development projects.

By utilizing our concept of atomic and composite change operations developers are enabled to model any type of change and refactoring operation that might occur in practice. Our approach allows for combining existing change operations into new types of operations, like for example an *add* and a subsequent *delete*-operation are combined into a *move*-operation. To illustrate this property of our approach, we applied our modeling concept during our case study for the modeling of a series of refactoring scenarios that were applied on an existing software system (see Section 10.2.3). Our case study required the modeling of various different kinds of change operations, including typical refactorings like the splitting and merging of classes, but also comprehensive changes required for the adaptation of existing Java source code to changed APIs and meta-models. Therefore, our case study covered a wide range of typical changes applied by developers that could easily be modeled using our approach.

Moreover, our case study has shown that even complex and extensive change scenarios can be modeled with reasonable effort and that the outcome is still easily comprehensible by humans. In fact, when applying changes on an existing system, developers have to think about how to implement them in the first place anyway. Hence, the explicit modeling of the changes is just a minor overhead as it requires developers to actually "write down" the changes before implementing them. Therefore, we also deem our modeling approach as applicable for both research and practice alike in this regard.

7.4. Summary

In this chapter we presented a concept how change operations can be modeled as sequences of atomic and composite operations to accomplish **Goal 4**. Our atomic operations represent the basic units of change, whereas composite operations are comprised of sequences of atomic and other composite changes. Moreover, we introduced a set of basic atomic change operations and a set of basic composite operations to enable the precise modeling of change operations as required for typical software development tasks. Our atomic operations encompass the addition, deletion, and modification of elements, whereas the set of composite operations is comprised of move, merge, split, replace, and swap operations.

Secondly, we introduced a meta-model for change operations that implements our concept of atomic and composite operations. This meta-model is realized as an EMF-model and goes in line with the EMF-based modeling of the software artifacts under consideration and the EMF-based modeling of their dependency relations as introduced in the previous two chapters.

Finally, we proposed a taxonomy for the classification of change operations consisting of four distinct criteria. The taxonomy enables researchers to classify the changes under study according our pre-defined criteria, which eases the comparison with other works. For practitioners our taxonomy provides valuable information on how a change operation is structured and in which context it may be applied, which is especially useful when developers are confronted with the task of classifying changes for a later impact analysis.

8. Rule-based Impact Analysis

The following chapter introduces our novel rule-based change impact analysis approach that is based on our research hypothesis presented in Section 4.1 and that addresses **Goal 1** and **Goal 2** of this thesis. To begin with, we discuss the general concept of our approach and how the propagation of changes is monitored by our approach. We illustrate how our novel impact propagation approach allows developers to understand and retrace the propagation of changes in order to meet **Goal 3**. Furthermore, we elaborate on our impact propagation rules, their structure, and introduce a scheme how impact propagation rules can be defined to fulfill **Goal 5**. At the same time, we also discuss the influence of the challenges outlined in Section 1.3 on the determination of impacts and the creation of impact propagation rules. Parts of the research presented in this chapter were already discussed in [LFR12, LR12, LFR13, FLR14]. Novel contributions of this chapter are *a)* a thorough discussion of how the impacts of changes are determined and *b)* the strict definition of a scheme for creating impact propagation rules.

The remainder of this chapter is organized as follows. Section 8.1 describes our approach for computing the impacts of changes using the concept of impact propagation rules and explains how developers are enabled to understand the resulting impact propagation. We discuss how the effects of different types of changes can be determined in Section 8.2. Subsequently, Section 8.3 introduces the concept and structure of our impact propagation rules and presents a scheme for developing impact propagation rules. Finally, in Section 8.4 the approach is critically discussed and it is classified according to the taxonomy of impact analysis approaches as introduced in Section 2.2 to enable an easy comparison with related work and to further outline its benefits.

8.1. Impact Propagation Concept

Our research hypothesis for studying the propagation of changes and for estimating their impacts on heterogeneous software artifacts is based on the analysis of the interplay of change operations, software artifacts, and dependency relations (see Section 4.1). Based on our hypothesis we propose a concept to analyze this interplay with the help of impact propagation rules that can be executed even before changes are applied on a software system and that are able to determine the impacts resulting from those changes. Figure 8.1 illustrates the situation developers are confronted with when changing software and outlines where the information required by our impact propagation rules can be obtained from.

In our approach each impact propagation rule is designed to react on a specific combination of change operations, software artifacts, and dependency relations and analyzes this interplay in accordance to our research hypothesis. The two main functions of our impact propagation rules are therefore to *a)* determine how the effects of changes propagate across the interplay of software artifacts and *b)* to determine how the artifacts are affected by those changes. Each rule consists of a set of conditions that encode the required dependency type, the change type

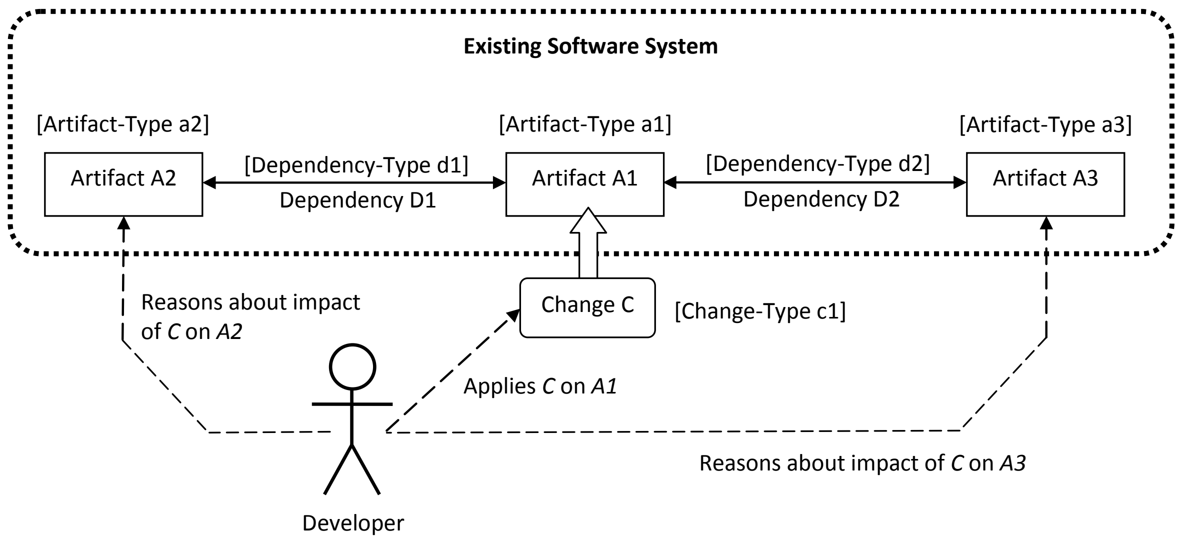


Figure 8.1.: All information that are potentially available when determining the impacts of changes

to be applied, the type of the artifact to be changed, and the type of the dependent and thus potentially impacted artifact. Additionally, each rule contains information to determine how the dependent artifact is impacted by the change. If a rule is triggered and all of its conditions are met, it declares the dependent artifact as impacted by the initial change and determines how this artifact is impacted according to the information encoded within the rule. To accomplish the latter, an impact report is created that afterwards can be inspected by the software developers conducting the impact analysis. Above all, this impact report provides information on the type of the change that is required in the related software artifact as a reaction to the initial change (i.e. the "impact").

If the execution of a rule resulted in an impact being reported, the rule-based impact analysis starts all over at the impacted artifact using the previously computed change type (i.e. the impact) as the new change trigger. This propagation of impacts continues as long as further rules are being triggered to react on new impact reports created during the impact analysis process. Hence, our approach operates in a recursive manner when predicting the propagation of changes. This concept enables our approach to better estimate the propagation of changes across the different views on software as only the aforementioned interplay of the different types determines where and how the impact of a change is being propagated to. Thus, one of the core benefits of our approach is that, as long as proper impact propagation rules are supplied to react on potential changes of software artifacts, any type of software artifact can be analyzed by it. Figure 8.2 illustrates the above discussed recursive concept and highlights where manual interaction is required and which parts of the approach are fully automated.

More importantly though, our rule-based propagation concept provides another improvement which is one of its main differences when compared to existing impact analysis approaches. In our approach, the type of impact resulting from the execution of a rule must not necessarily equal the type of change that triggered the initial execution of the rule. For example, merging two classes requires developers to adjust the data types of all the attributes and variables that were instances of the former classes. In this case, the initial change is a composite *merge*-operation, whereas the resulting impacts are atomic *property_updates* (see Section 7.1).

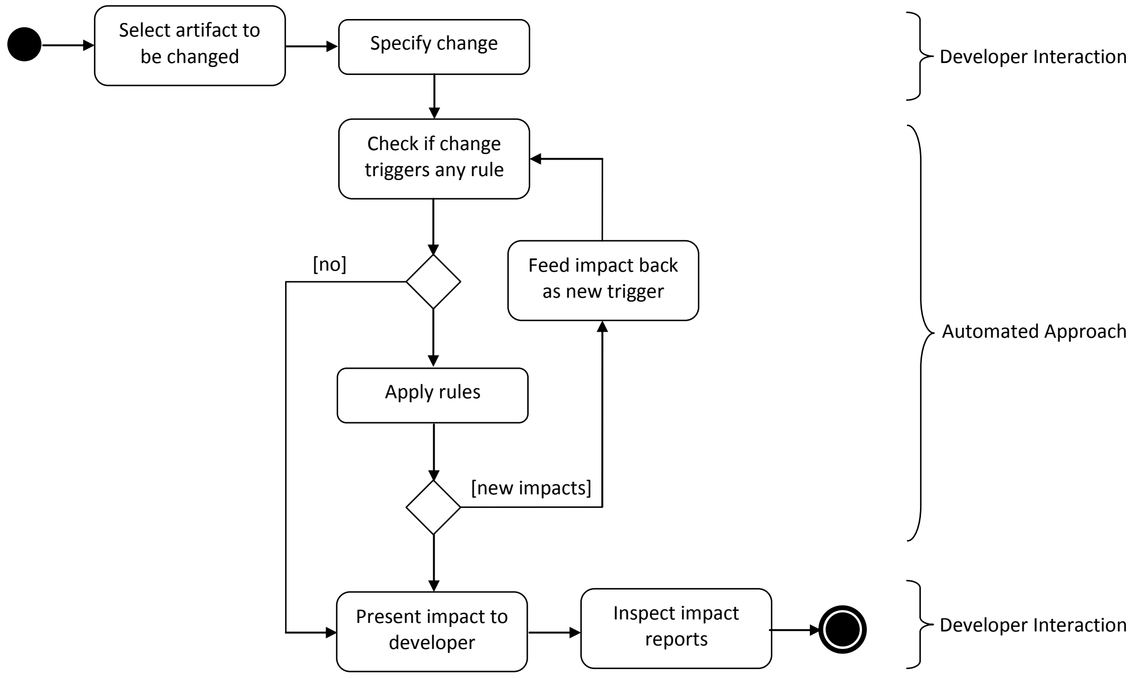


Figure 8.2.: Overview of our impact analysis approach

Modifying the data types of those attributes and variables in turn requires developers to adjust all the source code statements referring to them and so on. Consequently, this causes a multitude of subsequent changes that are required for adapting the software system to the initial change of merging the two classes. Yet, existing approaches try to estimate the propagation of a single type of change operation across the entire software system. However, as software is comprised of different types of artifacts, different views and perspectives, the initial change will result in different types of impacts at different parts of the software system, which they are therefore not able to grasp. In contrast, in our approach a single change may result in different types of impacts being propagated to different parts of the software system, which is only determined by the interplay of change types, artifact types, and dependency relations. Therefore, our approach is able to determine the propagation of changes more reliably than existing ones and it is independent of the actual artifacts to be analyzed.

Finally, we discuss the preconditions and requirements of our approach. First, all types of software artifacts must be accessible to our impact propagation rules, which requires a common meta-model and a model repository as discussed in Chapter 5 to unify the heterogeneous artifacts. Second, our approach requires that the dependencies between the software artifacts have been recorded and that the types of the dependencies have been determined according to the approach presented in Chapter 6. It furthermore requires that the changes to be applied on the software artifacts have been modeled and classified according to our concept of atomic and composite change operations and our taxonomy of change types as presented in Chapter 7. Likewise, the impacts resulting from the changes must also be classified according to their type.

Similar to our rule-based dependency detection approach presented in Chapter 6, our concept of impact propagation rules requires the rules to be set up in advance by developers or researchers. Thus, it requires an initial manual effort for analyzing the interplay of changes, dependencies, and software artifacts to formulate the rules. Therefore, to support the creation of such rules we discuss in detail how the impacts of changes can be determined in Section 8.2 and present a scheme to guide developers when creating or maintaining impact rules (see Section 8.3.2).

8.1.1. Modeling the Change Impact

In contrast to existing works we also define what we understand of the term "change impact" and how the impact of a change can be modeled in a structured manner. In accordance to the initial definition of the activity of change impact analysis as established by Bohner [BA96], the following excerpt of his definition can be reused for modeling the impacts of changes:

> "Identifying [...] what needs to be modified to accomplish a change [BA96]."

Now for defining the impact of a change, we have to extend this part of the definition as follows:

> "Identifying [...] what needs to be modified *and how it needs to be modified* to accomplish a change."

Thus, the structure of potential impacts is not different than the structure of the initial changes and can be modeled accordingly. Consequently, our concept of atomic and composite change operations as introduced in Chapter 7 can be applied for this purpose as well, which means that the impacts of changes are modeled using sequences of *add*, *delete*, *update_property*, *move*, *merge*, *swap*, and *replace* operations [LFR12, LFR13]. For example, if two classes are about to be merged (type of change: *merge*) all attributes and variables that are instances of one of these classes should change their type in a similar manner (type of impact: *update_property*).

A different approach for the classification of impacts is discussed by Wilkerson who distinguishes between direct and indirect impacts of changes [Wil12]. However, his taxonomy only encompasses a small subset of changes that are applicable on source code and completely neglects any changes and impacts on other software artifacts. Furthermore, in his taxonomy the impacts of changes are semantically and syntactically distinguished from the actual changes, which in turn limits its applicability. In contrast, with our approach of modeling impacts as change operations we present a concept that allows impacts already determined by our approach to act as triggers for additional impacts. This enables our approach to better estimate how the effects of changes spread across a software system and to more precisely capture the iterative change and maintenance process of software developers.

8.1.2. Understanding the Change Impact

Apart from determining the impacts of changes and their propagation across the software system, it is also important to enable developers to trace and understand this propagation and the different effects of changes, as they are supposed to actually change the software based on the computed impact assessments. However, as our investigation in Section 2.5 has turned out, recent approaches for impact analysis fail to convey to developers *why* and *how* software artifacts are impacted by changes, which stands in contrast to **Goal 3** of this thesis.

Our rule-based approach that creates impact reports as explained in Section 8.1 accomplishes **Goal 3** by providing the necessary information for actually implementing the changes. Our concept of impact reports provides developers with the following information:

- Which artifact is impacted.
- Which type of change must be applied on it.
- Which artifact caused the impact.
- Which change operation caused the impact.
- Which dependency relation exists between the two artifacts.
- Which rule created the impact report.

To implement this approach, each impact report generated during the impact analysis process is defined as a 5-tuple providing the following information:

- Source of the Change: defines the initially changed element that triggered the creation of the impact report.
- Impact Target: defines the impacted element.
- Initial Change: stores a reference to the change that triggered the rule.
- Required Change: defines the change operation which must be applied on the impacted element as a reaction to the initial change.
- Dependency: stores a reference to the dependency relation that carried the impact.

The creation of such impact reports is defined within our impact propagation rules and depends on the artifacts and changes analyzed by them.

In stark contrast to our concept of impact reports, existing impact analysis approaches as discussed in Section 2.3 are only able to determine which artifacts are impacted by a change without explicitly stating *why* and *how*. Only probabilistic approaches are moreover able to determine the likelihood that an artifact is impacted by a change (see Section 2.3.4). However, when compared to the information provided by our concept, they fail to convey the information that are necessary for implementing the changes as no information are given how the impacted artifacts should be changed. In contrast, in our approach developers are supplied with a list of impact reports that reflect the propagation and overall impact of the initial change on the software system. Moreover, the computed impact reports contain step-by-step instructions how the software should be adapted in regard to the initial change operation.

Additionally, our concept of detailed impact reports even enables a (semi)-automated implementation of changes as we are going to discuss in Section 8.4.6.

8.1.3. Monitoring the Impact Propagation

One of the key challenges of our approach is to monitor the recursive propagation of changes to prevent impacts from being mutually propagated between the same software artifacts in an indefinite loop due to cyclic dependencies. As most software artifacts are typically dependent on more than one other software artifact, the dependency relations between them may form a cyclic graph. Thus, a change can be propagated anywhere within this graph and into any direction, and it might even be propagated back to the initial trigger of the impact analysis. Consequently, one has to keep track of the change propagation to break chains of mutual dependencies and to ensure that each potential impact path is only taken once in reaction to the initial change.

The naive solution of limiting the number of dependency relations that can be searched starting from a given software artifact using a fixed cut-off distance (see Section 2.3.1.1) produces poor results when applied on real software systems [LFR13]. Moreover, this approach does not truly solve the problem of cyclic dependencies as iterations are allowed as long as the propagation limit is not reached. Hence, to be best of our knowledge, no further research was conducted on such approaches since the early 2000s.

This problem can only be tackled if one is able to keep track of those dependency relations that have already been "visited" by propagation rules and those "impact paths" that have already been explored and thus are no longer relevant for the impact propagation. Similar problems are being studied in the fields of artificial intelligence and pathfinding in particular. A typical pathfinding problem is to find a way from a given source node of a graph to a given destination node while avoiding circles within the path itself and performing potential backtracking due to dead-ends and obstacles. Therefore, in our approach we are blending pathfinding concepts into the impact analysis process to record the "paths" already taken by the impact propagation in order to identify and prevent potential infinite loops caused by cyclic dependencies.

A suitable and widely-used pathfinding algorithm that allows to keep track of already visited paths is the *A**-pathfinding algorithm [HNR68] that is nowadays often used for pathfinding of virtual agents in environments such as 3D simulations and computer games [Mat02, Hig02]. Just like our impact propagation concept, *A** operates in a recursive manner and thus allows for nodes to be visited more than once. To keep track of the nodes that were already visited and processed by the algorithm, *A** maintains two lists: the *ClosedList* and the *OpenList*. The *ClosedList* contains all nodes that have already been completely processed, whereas the *OpenList* contains all nodes which are still to be inspected by *A**. Each time a node of the graph is investigated, the *ClosedList* is checked for whether the same node was already completely processed before. If not, the node is added to the *OpenList* for further inspection.

For our impact analysis approach we adjust and reuse this concept as follows. The *ClosedList* stores all impact reports whose effects on other software artifacts have been fully explored, meaning that all directly related elements were visited by impact propagation rules and no directly dependent artifacts are left for inspection. In contrast, the *OpenList* stores all impact reports where a further propagation of impacts might be possible and at least one dependent element is still left for inspection by our impact propagation rules. Whenever a new impact report is created, the *ClosedList* is searched for entries containing the same elements. If such an entry is found, the impact report is dropped from the *OpenList* and further propagation on this path is stopped since is has already been computed before. The recursive propagation then continues as long as the *OpenList* is not empty. Therefore, at the beginning of the impact analysis, the initial change is transformed into an impact report and is inserted as the first element into the *OpenList*, which then triggers the subsequent execution of our impact propagation rules.

This concept is illustrated by the example presented in Figure 8.3 that depicts the propagation of a change and the changing contents of the *OpenList* and the *ClosedList*. In this example, the node *A* is the initially changed artifact (step i) and its dependent artifacts are to be analyzed whether they are impacted by this change. Thus, in a second step the artifacts *B* and *D* and their dependencies towards *A* are examined by our rules (step ii), which is indicated by the grey ellipses that represent the current search space. Now we assume that *D* is impacted by the change and therefore added to the *OpenList* (step iii), while the resulting impact is fed back as a new change trigger into the propagation process. Consequently, the impact propagation recursively

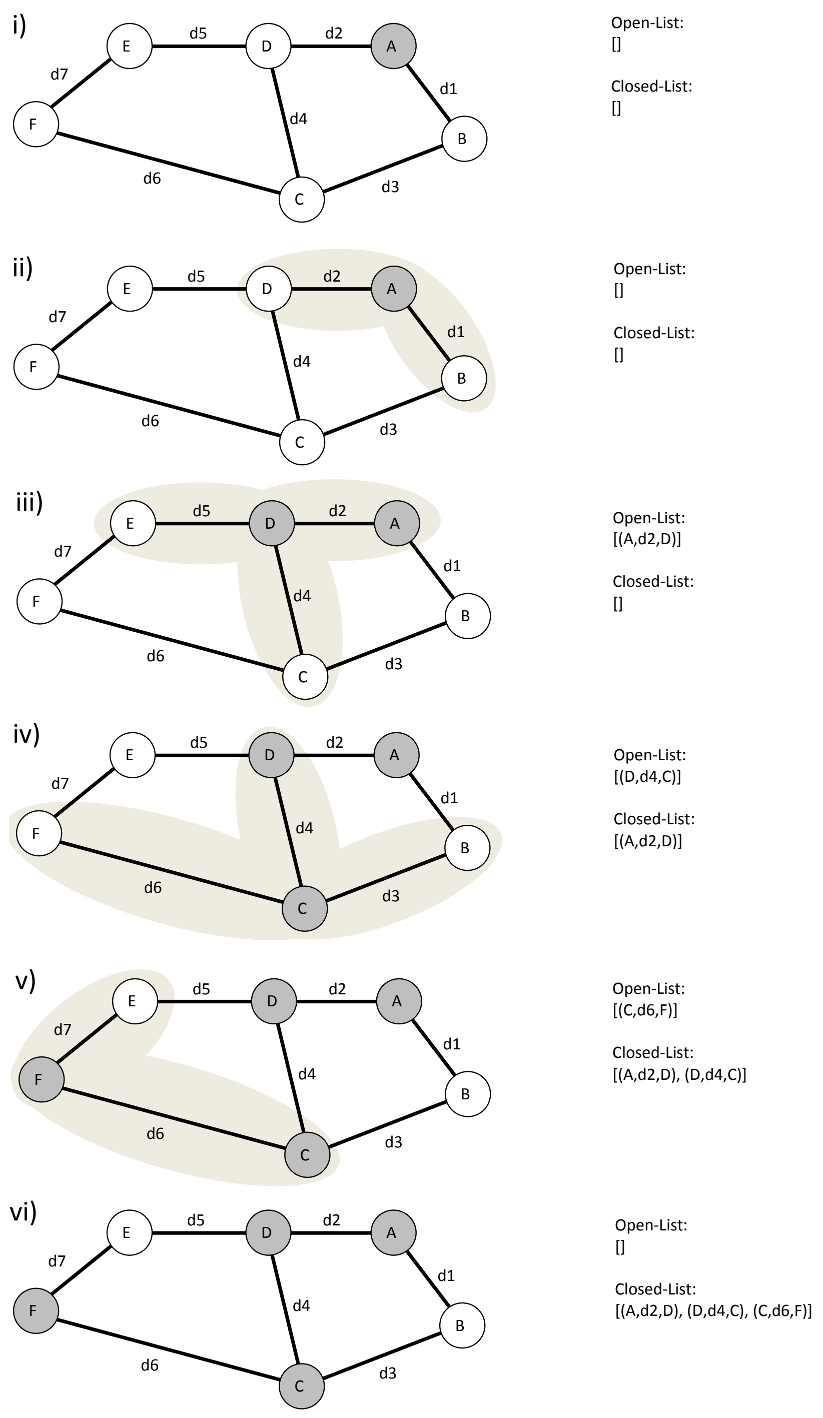


Figure 8.3.: The propagation of changes and the changing contents of the closed and open list. Grey ellipses indicate the current search space. Grey nodes indicate impacted artifacts

restarts at this node by inspecting the nodes *E* and *C*, where *C* is declared as impacted and the tuple (*A*, *d2*, *D*) is now added to the *ClosedList* since all directly dependent software artifacts are explored (step iv). Finally, the propagation comes to an end once the *OpenList* is empty (step vi), whereas the *ClosedList* contains the entire impact of the initial change applied on node *A*.

Finally, we present our modified version of the A*-algorithm that implements the above discussed concept and that is demonstrated as pseudo code by Listing 1. The modified algorithm was initially published in [LFR13] and works as follows. In line 1 the initial change and the artifact it is applied upon are inserted into the *OpenList* as the first impact report, while the *dependency*, *impacted artifact*, and *type of impact* are left blank. Line 2 then checks if there are impact reports left to be processed, which is true for the initial change as a new report has just been added. In line 3 the actual impact propagation rules are applied on all directly dependent artifacts, which potentially returns a list of further impact reports. Then, for all the impact reports created by our rules lines 4 to 8 check whether they should be added to the *OpenList* or if they have already been explored. Line 10 adds the initial impact report to the *ClosedList* as all directly dependent neighbors of the changed software artifact have been explored. Finally, in line 11 the initial impact report is dropped from the *OpenList* and the recursive propagation continues using the next impact report from the *OpenList* or terminates if there is none.

Algorithm 1 Monitoring the recursive change propagation approach

```
 1: openList.add(new ImpactReport(changedModel, change, null, null, null));
 2: while $(!openList.isEmpty())$ do
 3:     $List < ImpactReport > tmp = executeRules(openList.get(0));$
 4:     for $(i = 0; i < tmp.size())$ do
 5:         if $(!containsTuple(openList, tmp.get(i)))$ then
 6:             openList.add( tmp.get(i) );
 7:         end if
 8:     end for
 9:     if $(!containsTuple(closedList, openList.get(0)))$ then
10:         closedList.add( openList.get(0) );
11:         openList.remove(0);
12:     end if
13: end while
```

8.1.4. Impact Analysis Process

This section illustrates the change impact analysis process the approach presented in the previous sections is embedded into, to further outline where manual interaction is required and how our prototype tool to be presented in the next chapter is incorporated into the impact analysis.

The actual impact analysis process is triggered by developers who have to select the software artifacts they are going to change due to a given change request, bug report, etc. Secondly, they have to specify the type of the change they want to apply on this software artifact using our approach of atomic and composite change operations as explained in Chapter 7. Finally, they have to trigger the execution of our impact propagation rules that compute the impacts of this change. Once all the impacted artifacts were determined, a list of impact reports is presented to them, based on which they can then modify the software artifacts under consideration. If further impact analysis support is required during those modifications, the approach has to be restarted at the first step.

The process outlined above is also illustrated as a BPMN [OMG13] process diagram by Figure 8.4, which features the process participants *Developer* and our prototype tool *EMFTrace* (see Chapter 9). We discuss the potential shortcomings of this process in detail in Section 8.4.2.3, such as the influence of impacts on existing dependency relations and the necessity for re-executing the dependency analysis due to changes of dependencies.

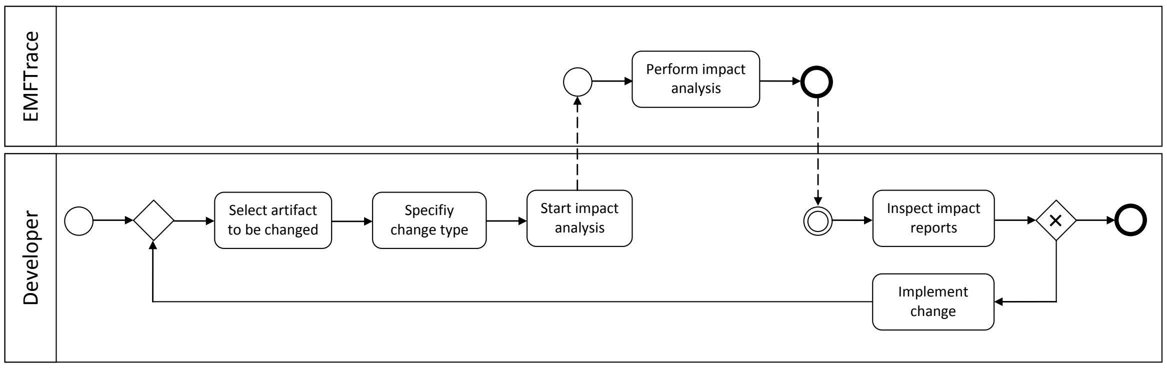


Figure 8.4.: The impact analysis process illustrated as a BPMN process diagram

8.1.5. Influence of the Challenges

In this section we discuss the influence of the challenges outlined in Section 1.3 on the proposed rule-based change impact analysis approach. We therefore analyze how each challenge potentially hampers the change impact analysis, discuss possible solutions to tackle those challenges, and outline the implications if the challenges cannot be addressed in an adequate manner.

8.1.5.1. Varying Formalization

A varying degree of formalization among the different types of software artifacts restricts the creation of impact propagation rules to address potential changes. Even though all types of software artifacts can be transformed into EMF-based models (see Section 5.2.1), not all of them can be accessed using the same structured approach afterwards. For example, defining an EMF-based model for free text requirements is a trivial task, yet the information still remain “hidden” within the text that offers no structural features and relations. Thus, a lack of formalization goes hand in hand with a lack of proper impact propagation rules. In this case information retrieval based approaches could be used as a potential fallback solution. However, they do not provide any type information and are less useful due to their comparably high ratios of false positives as discussed in Section 2.3.3. Consequently, not all types of software artifacts can be addressed by a rule-based approach in accordance to the requirements of this thesis. Nevertheless, our rule-based concept allows for analyzing free text using n-gram-matching and other text-based operations (see Sections 6.4.4.2 and 8.3.1), which mitigates this limitation to a certain extent. Additionally, most software artifacts of the solution space that are addressed by this thesis can be analyzed by impact propagation rules as they adhere to a more thorough formalization than most artifacts of the problem space (see Section 4.3.1).

8.1.5.2. Incomplete Artifacts and Missing Information

The absence of information, such as missing properties and relations of software artifacts for example, may restrict the applicability of our concept of impact propagation rules. Missing information might result in impact propagation rules not being triggered when they are supposed to, as the conditions encoded within the rules cannot be fulfilled. Likewise, the complete absence of certain software artifacts can cause the same issue. For example, when a design methodology was applied but not followed correctly and thus not all of the required artifacts and dependencies were introduced, all impact rules that react on these combinations of artifacts and dependencies will never be triggered. Consequently, the overall estimation of the impact propagation will be incomplete. On the other hand, missing information do not affect the quality of already determined impacts, i.e. they do not result in false-positives.

To the best of our knowledge there is no approach to cope with missing information in an automated manner. Even developers conducting manual impact analysis face the same problem as they cannot estimate the impacts of a change if the required information are missing. Instead, one could use other techniques as a fallback mechanism, such as information retrieval based approaches for example. Yet, there is no empirical evidence that these approaches are able to cope with such situations, as the reported experiments were conducted with complete case study subjects only, e.g. [PMFG09, GP10]. Other studies applying Bayesian belief networks for impact analysis also reported that they were only partially able to cope with missing information [ZWG+08]. Hence, the problem of missing information cannot be addressed by impact rules in an adequate manner, nor by other approaches proposed up until now.

8.1.5.3. Inconsistencies between Artifacts and Views

The impacts of inconsistencies on our concept are twofold. First of all, potential inconsistencies prevent our impact rules from being triggered due to the absence of (correct) dependency relations between the inconsistent software artifacts. Likewise, inconsistencies between actually dependent software artifacts prevent the conditions encoded by our impact rules from being fulfilled, such as for example when the names of the software artifacts are inconsistent (e.g. the name of a UML class and its corresponding Java class do not match at all).

Secondly, inconsistencies might trigger the wrong impact rules, which in turn results in an incorrect assessment of the impacts of a certain change, thus misleading the overall impact propagation and eventually resulting in false-positives and missed impacts. In a similar fashion inconsistent software artifacts may cause the impact rules to create the wrong types of impact reports, which also leads to the above stated problems.

Yet, there is no comprehensive approach to cope with inconsistencies during change impact analysis. Inconsistencies are too manifold to be successfully tackled in an automated manner. There are, for example, "simple" inconsistencies due to the application of an inconsistent naming scheme and the more severe conceptual inconsistencies (e.g. the modeled software architecture is not reflected in the source code). While the influence of inconsistent names might be mitigated by the usage of IR-based techniques (see discussions in the previous section), it is hard to correct conceptual inconsistencies in an automated fashion. Consequently, we advocate the usage of (semi)-automated impact analysis approaches prior to any changes to prevent such inconsistencies.

8.2. Determining the Effects of Changes

After introducing the general concepts of our approach in the previous sections we now elaborate on how the actual impacts of changes are determined based on the interplay of changes, dependencies, and software artifacts. Consequently, the next important step of our approach is to analyze and determine how different types of changes and dependencies influence the propagation of impacts in order to formulate strict impact propagation rules.

In the following four sub-sections we discuss the different steps that can be applied for determining the effects of a certain change, where each step is focused on analyzing a certain aspect of the dependency relations, change types, and software artifacts. The first step *Analyzing the Directions of Dependency Relations* is aimed upon determining whether a certain relation propagates a change or not. However, it does not always answer this question immediately. Thus, our second step *Analyzing the Origins of Dependency Relations* assists with determining the reason why a dependency has been introduced in the first place and therefore helps to determine its role for the impact propagation. The third step *Analyzing the Interplay of Changes and Dependencies* finally determines if and how a change propagates across a dependency relation if the previous steps were not able to determine these information. Finally, the last step *Analyzing the Interplay of Artifact Type and Change Type* tailors the impact according to the concrete software artifact that is impacted, which is then fed back into the recursive propagation and later on handed to the developer in the form of a comprehensive impact report.

8.2.1. Analyzing the Directions of Dependency Relations

In a first step we examine the directions of dependency relations as they help to determine the propagation of impacts to a certain extent. Therefore, one has to distinguish between two cases when studying dependency relations: the directed and the undirected dependencies.

Undirected dependency relations either express equivalences and similarities between two software artifacts, or they represent uncertain dependencies whose purpose could not be precisely determined, which is why they lack a concrete dependency type.

If a dependency relation is undirected due to equivalences, the impact will spread in either of the two directions, such as for example when a dependency was detected between a UML class and its corresponding Java class. Consequently, the dependent software artifact demands for the application of the same type of change operation which only has to be tailored according to the specific type of the software artifact (see Section 8.2.4).

Imprecise relations whose type could not be determined are excluded from the impact analysis process, as our concepts demand that the types of the dependency relations are explicitly known. Incorporating such unclassified and potentially incorrect dependencies would most likely introduce many false-positives to the estimated impact sets as no specific types of impacts can be determined for them. This phenomena can be observed in approaches that propagate changes across any dependency relation regardless of their type or direction, e.g. the work of Bohner [Boh02a] as discussed in Section 2.3.1.1.

The effects of directed dependency relations entirely depend on the combination of change type and dependency type and thus require further analysis. Hence, this discussion will be interweaved with the discussion of their interplay with changes in the next sections.

8.2.2. Analyzing the Origins of Dependency Relations

In a second step we analyze the origins of dependency relations, as the origin of a dependency provides information on the background of the dependency and why this dependency exists between certain software artifacts. Hence, these information also assist with determining further change propagation and thus need to be discussed.

Meta-model Dependencies and Object Oriented Dependencies: the influence of dependencies stemming from these origins is very similar. For the artifacts considered in this thesis (UML models and Java source code) it is precisely the same. Since a meta-model or language specification defines all the potential states of its instances, it also assists with determining the effects of changes that were applied on artifacts of the same meta-model or language specification. For example, the Java language specification demands to either delete or modify attributes and variables after the class they instantiated was deleted. Consequently, these information can be converted into impact propagation rules that are triggered by deleting classes.

Design Methodology dependencies: design methodologies define steps leading from one type of software artifact (e.g. requirements) to another type of software artifact (e.g. architectural design) to realize a certain aspect of a software system. Hence, these steps also apply when developers change software. We illustrate this using the exemplary approach of Object Oriented Analysis (OOA) [CY91] as discussed in Section 6.4.3.3. When applying OOA, the initial step "Finding classes and objects" aims to define the classes that comprise the software, while the later step "Identifying subjects" defines the system components to which these classes are assigned. Now if one of these components is changed, the related classes require a similar treatment as they originally resulted from the meanwhile modified component. Thus, by identifying which step of a design methodology connects the dependent artifacts, one can potentially derive an initial set of impact conditions and even determine the type of the resulting impact.

Multiperspective dependencies: the view or perspective a software artifact is associated with helps to determine its purpose by providing its general context, e.g. whether it is expressing an excerpt of the structure or behavior of a software system. These information in turn assist with determining the effects of changes on software artifacts of a different perspective. For example, a relation between an artifact of the structural view and an artifact of the behavioral view either indicates that the behavioral artifact describes the functional features of the structural artifact (e.g. a state chart diagram defining the legal states of a component's port) or that the behavioral artifact is an instance of a structural entity (e.g. a lifeline or actor that instantiates a class or component). Therefore, if the reasons why and how both artifacts are dependent on each other are clear, the effects of changes can be determined more easily. However, if both artifacts stem from the same view this information is of no additional use because both are having the same general purpose.

8.2.3. Analyzing the Interplay of Changes and Dependencies

The most important step when determining the effects of changes is to study the interplay of change operations and dependency relations according to our research hypothesis. This can be accomplished by studying the purpose of the involved dependency relations in regard to the type of change to be applied. Therefore, we utilize our purpose-based taxonomy of dependency types as introduced in Section 6.3 and our approach of modeling change operations based on the concept of atomic and composite operations as introduced in Section 7.1. In the remainder of this section we discuss the effects of changes according to the nine general purposes of dependencies that build the foundation of our dependency type taxonomy (see Section 6.3.1).

Abstraction Dependencies: the effects of abstraction relations are determined by their direction, for which we distinguish between two cases. First, changes applied on the more "abstract" artifact have to be reflected by the "fine-grained" artifact as well, since this artifact "inherits" parts of its content and concepts from it, regardless of the type of operation. For example, every change applied on the base class of an inheritance hierarchy has to be reflected by all of its sub-classes (e.g. when an attribute is deleted from the base class). Secondly, changing the "fine-grained" artifact does not necessarily affect the more "abstract" artifact and depends on the type of change. For example, adding methods to a sub-class does not affect the base class, whereas changing the name of a component refining another component also affects the refined component which should be renamed as well. Likewise, the splitting and merging of "fine-grained" artifacts affects the "abstract" artifact in a similar manner. Table 8.1 summarizes the impact propagation between the "abstract" artifact *A* and the "fine-grained" artifact *B*. The exceptions of inheritance relations are marked by an * to indicate their alternative impacts.

Change applied on B	**Impact on abstract artifact A**	**Change applied on A**	**Impact on fine-grained artifact B**
add to B	no impact	add to A	add to B
delete B	no impact	delete A	delete B
update B	update A*	update A	update B
merge B	merge A*	merge A	merge B
split B	split A*	split A	split B
move B	move A*	move A	move B
replace B	replace A*	replace A	replace B
swap B	swap A*	swap A	swap B

Table 8.1.: Impact propagation based on *abstraction*-dependencies. Inheritance relations are not impacted by this type of change (*)

Structural Dependencies: the direction of structural dependencies determines their influence on the propagation of changes. If the artifact that contains or aggregates other artifacts (i.e. their "container") is deleted, all its "content" must be deleted as well. If containers are merged into one, all their content must be moved to the merged container. In contrast, if a container is split, all its content must be moved to the container that resulted from the split-operation. Changes affecting the container (i.e. the opposite direction) are the deletion of contained elements or the movement of contained elements to other containers. However, the splitting or merging of contained content does not affect the container itself, as all the functionality provided by the modified content still remains within the container. Table 8.2 summarizes the impact propagation between the containing artifact *A* and the contained artifact *B*.

Realization Dependencies: there are two different cases of realization-dependencies between which we have to distinguish. The first case encompasses artifacts that are instances of others, such as that a variable or attribute is an instance of a class. The second case encompasses artifacts that implement the concept represented by a different artifact, for example Java classes implementing UML use case actors or classes implementing interfaces. The first case typically demands for an impact that is different than the original change, as for example the renaming,

Change applied on B	Impact on container artifact A	Change applied on A	Impact on contained artifact B
add to B	no impact	add to A	no impact
delete B	update A	delete A	delete B
update B	no impact	update A	no impact
merge B	no impact	merge A	move B
split B	no impact	split A	move B
move B	update A	move A	no impact
replace B	update A	replace A	delete B
swap B	update A	swap A	no impact

Table 8.2.: Impact propagation based on *structural*-dependencies

splitting or merging of classes requires to modify the data types of all attributes, variables, and method parameters that are instances of these classes. The second case demands for applying the same type of operation on the related artifact. For example, changing a method defined by an interface demands for changing all (class)-methods that implement this method. Therefore, Table 8.3 only summarizes the impact propagation for the first case, i.e. between the artifact *A* and one of its instances *B*.

Change applied on B	Impact on realized artifact A	Change applied on A	Impact on realizing artifact B
add to B	no impact	add to A	no impact
delete B	no impact	delete A	delete B
update B	no impact	update A	update B
merge B	merge A	merge A	update B
split B	split A	split A	update B
move B	no impact	move A	no impact
replace B	no impact	replace A	replace B
swap B	no impact	swap A	swap B

Table 8.3.: Impact propagation based on *realization*-dependencies

Definition Dependencies: when determining the effects of definition-dependencies, one has to distinguish between the effects of delete, merge, split, and replace operations. Property updates, additions, and swap operations on the other hand can be neglected, as they do not affect neither of the two artifacts when applied on the other. The delete operation is "direction sensitive" as the "defined" elements have to be deleted when the element defining them is deleted, for example when an interface is deleted that defined several operations. The same applies to the merge operation, where the "defined" artifacts have to be moved if the artifacts that defined them were merged. When splitting the defining artifact, the artifacts that are defined by them either have to be moved to another artifact or they have to be split as well, which depends on the exact type of the software artifact. When replacing the defining artifact, the artifacts that are defined by them either have to be deleted or moved to the artifact that replaced the initial defining artifact, which again depends on the artifacts the change is applied upon. Table 8.4 outlines the impact propagation between artifact *A* and artifact *B* that is defined by it.

Change applied on B	**Impact on defining artifact A**	**Change applied on A**	**Impact on defined artifact B**
add to B	no impact	add to A	no impact
delete B	no impact	delete A	delete B
update B	no impact	update A	no impact
merge B	no impact	merge A	move B
split B	no impact	split A	move or delete B
move B	no impact	move A	move B
replace B	update A	replace A	move or delete B
swap B	no impact	swap A	no impact

Table 8.4.: Impact propagation based on *definition*-dependencies

Behavioral Dependencies: the effects of behavioral relations depend on the exact sub-cluster they stem from. Yet, the influence of *Examination* and *Utilization* dependencies is the same and is furthermore direction sensitive. This means that each change applied on an artifact that is either used or examined by another artifact has to be reflected by the other artifact as well. For example, adding an additional parameter to a method requires all code statements calling this method to add this parameter as well. In contrast, changes applied on the artifacts that examine or utilize other artifacts do not affect the other artifacts. Table 8.5 summarizes the impact propagation between artifact *A* and the used or examined artifact *B*.

Change applied on B	**Impact on examining/using artifact A**	**Change applied on A**	**Impact on used/examined artifact B**
add to B	no impact	add to A	no impact
delete B	update A	delete A	no impact
update B	update A	update A	no impact
merge B	update A	merge A	no impact
split B	update A	split A	no impact
move B	update A	move A	no impact
replace B	update A	replace A	no impact
swap B	update A	swap A	no impact

Table 8.5.: Impact propagation based on *examination*- and *utilization*-dependencies

Dependencies of the *Creation* cluster have a much more diverse effect on the propagation of changes. Therefore, let software artifact *A* create, delete or transform software artifact *B*. Whenever artifact *B* is changed in any way, artifact *A* has to be equally adapted in order to ensure that *B* is correctly created, deleted or transformed by *A*. For example, the constructor of a class must be updated if a new attribute is added to the class. The same applies when *B* is split, merged, replaced or swapped. On the other hand, if artifact *A* is deleted or replaced, artifact *B* has to be removed as well. Likewise, if artifact *A* is split, artifact *B* must be split as well. However, updating, merging or moving artifact *A* does not affect artifact *B*. Table 8.6 summarizes the impact propagation between artifact *A* and artifact *B* that is either created, deleted or transformed by it.

Similarity Dependencies: dependency relations stating that two software artifacts are similar or equivalent to each other require that every change operation is propagated between them in a similar fashion. Therefore, no further discussions of certain types of changes are necessary.

Change applied on B	Impact on creating/deleting artifact A	Change applied on A	Impact on created/deleted artifact B
add to B	update A	add to A	no impact
delete B	no impact	delete A	delete B
update B	update A	update A	no impact
merge B	update A	merge A	no impact
split B	update A	split A	delete or split B
move B	update A	move A	no impact
replace B	update A	replace A	delete B
swap B	update A	swap A	delete or update B

Table 8.6.: Impact propagation based on *creation*-dependencies

For example, if there exists an “Is-Equivalent-To” relation between a UML class and a Java class each change applied on the UML class has to be reflected by the Java class and vice versa. Otherwise inconsistencies between the architecture and implementation would arise.

Evolutionary Dependencies: evolutionary relations are currently not addressed by our type-based impact analysis approach. This is due to the reason that the existence of a common version history cannot be taken as granted for the heterogeneous software artifacts, since UML models, requirements, source code, etc. typically not evolve in the same version control system. Moreover, their interplay with different types of changes cannot be determined since nothing can be stated about the true purposes of evolutionary couplings.

Conditional Dependencies: the effects of conditional relations depend on their exact purpose. For example, deleting an element which is part of a *conflict*-relation might resolve the conflict, whereas deleting an element of a *contribution*-relation has the opposite effect on the system. However, these types of relations are currently excluded from this thesis' research as they are mostly focused on artifacts of the problem space, whereas this thesis is solely focused on the solution space (see discussions in Section 4.3.1).

Causation Dependencies: such dependencies are not considered by the approach presented in this thesis, as they describe dependencies resulting from impacts on software artifacts. In our approach, however, such relations are never created, as our rule-based concept directly determines the effects of changes and is able to react on new types of impacts created during the impact analysis due to its recursive rule-based propagation concept.

8.2.4. Analyzing the Interplay of Artifact Type and Change Type

Finally, after analyzing the interplay of change types and dependency types to determine if and how a change propagates to dependent artifacts, one still needs to determine the true types of the resulting impacts, i.e. the resulting change operations that are required to maintain the consistency of the dependent artifacts. Based on the interplay of change type and dependency type one can determine how a dependent artifact is impacted, which, however, must be tailored according to the actual type of the dependent artifact. For example, if artifact *A* “refines” artifact *B* and *B* is about to be “renamed”, then *A* has to be “renamed” as well. Yet, dependent on the type of *A*, different types of rename-operations are required in order to precisely determine the impact of the change. For example, if a UML component refines a UML use case system the resulting change operation would be "Rename component". In contrast, if a UML class refines the same UML use case system the resulting change operation would be "Rename class".

Although this step might seem obvious at first, tailoring the resulting impacts is important for three reasons. First of all because the resulting change operations might act as triggers for further change propagation, hence precise type information are required. Secondly, the final impact set is presented to a developer who should be able to unambiguously comprehend and implement the resulting changes. Finally, it is also possible to realize a (semi)-automated implementation of the resulting impacts (see discussions in Section 8.4.6), which therefore requires a strict classification of the necessary change operations.

8.3. Impact Propagation Rules

In this section we introduce the aforementioned impact propagation rules that implement our concept of type-based impact analysis. First, we explain the general structure of our rules in Section 8.3.1 and demonstrate their relation to our dependency detection rules as presented in Section 6.4.4. Secondly, in Section 8.3.2 we discuss a scheme for developing impact propagation rules which is based on the discussions of the previous sections to accomplish **Goal 5**. Finally, we present several exemplary rules to further illustrate our concepts of impact propagation rules and recursive impact propagation in Section 8.3.3. Moreover, Appendix B lists all the impact propagation rules that accompany this thesis. They can also be obtained from the website of our prototype tool [EMF14].

8.3.1. Structure of the Rules

The impact propagation rules presented in this thesis are derived from our dependency detection rules as introduced in Section 6.4.4.1, since they already encode the required dependency relations and the software artifacts involved in the dependency. Therefore, the impact propagation rules inherit their structure from the dependency detection rules, i.e. they are comprised of the same three parts: *ElementDefinition*, *QueryDefinition*, and *ResultDefinition*. Furthermore, they utilize the same query operators as defined in Section 6.4.4.2. Yet, they have to be extended with means for analyzing change operations and dependencies according to our research hypothesis.

Our impact rules analyze the interplay of changes, dependencies, and software artifacts as follows. First, with the help of the ***ModelRelatedTo*** and ***ModelUndirectedRelatedTo*** operations our rules are able to analyze the dependency relations connecting the dependent software artifacts. Secondly, the changes to be applied are analyzed by an ***ValueEquals*** condition that is added to each rule to compare the name of a given change operation with the type of change that should be addressed by a rule. Finally, the types of the related software artifacts do not need to be specifically addressed by additional query conditions as they are already specified in the *ElementDefinition*-part of the rules.

The major difference between our dependency detection rules and our impact propagation rules resides in the *ResultDefinition*-part of the impact propagation rules, as they trigger the creation of impact reports in response to changes, whereas our dependency detection rules create traceability links. However, apart from that there are no conceptual differences between the two types of rules. Therefore, the impact propagation rules are also able to query and assess the structure and properties of software artifacts, as well as the dependency relations between them.

The reasons why a custom rule-concept was chosen to implement our rule-based impact analysis approach (and not OCL as for example by Briand *et al.* [BLOS06]) are the same as already discussed for our dependency detection rules in Section 6.4.4.1.

8.3.2. Definition of Impact Rules

In order to accomplish **Goal 5** and to support the practical applicability of our impact analysis approach we have to provide a scheme for defining the required impact propagation rules. The scheme presented in this thesis can be utilized for creating further impact propagation rules to address additional software artifacts that are not yet considered by this thesis, such as configuration files or ontologies for instance.

In contrast to our scheme for defining dependency detection rules as introduced in Section 6.4.4.3, the proposed scheme for defining impact propagation rules is conceptually different. As stated in the previous section, our impact propagation rules are based on existing dependency detection rules for two reasons: they already contain the required dependencies and they already contain the required types of software artifacts. What is still missing are means to address potential change operations, which is why the rules have to be extended in this regard.

This extension is accomplished by computing the cross product of a dependency detection rule and all types of meaningful change operations that can be applied on the source artifact of the encoded dependency relation. In the context of our research by the term “meaningful operations” we refer to change operations that actually occur in practice. In contrast, an example for a non-meaningful operation is the application of a split-operation on an attribute of a class. Thus, the following formula describes how the set of potential impact propagation rules (PIR) can be obtained from a set of existing dependency detection rules (DR) and the change operations (AC) that are applicable on the source artifacts of the dependency relations:

$$PIR := \{DR \times AC\} \tag{8.1}$$

Once a set of potential impact propagation rules has been established, the effects of each change operation in regard to the type of the dependency relation and the types of the involved software artifacts must be determined. This step is supported by the outcome of our discussion of the effects of different types of changes in Section 8.2. Finally, the following seven steps should be taken when defining impact propagation rules:

- **Step 1 - Identify the relevant changes that are applicable on the source artifact.** To reduce the set of potential impact propagation rules it is first of all necessary to exclude changes that are not applicable on the source artifact of a dependency detection rule.
- **Step 2 - Compute the cross product of the dependency detection rule and all possible change operations that are applicable on the source artifact.** By constructing the cross product of both sets one obtains the final set of potential impact propagation rules.
- **Step 3 - Determine the effects of each change on the dependent artifact.** For each of the potential changes to be applied on the source artifact of a dependency relation one has

to determine whether they impact the related artifact and if so, how they impact it. The impact guidelines as discussed in Section 8.2 assist with this task.

- **Step 4 - Specify the type of the change to be analyzed.** An additional *ElementDefinition* has to be added to the rule in order to address the change operation to be analyzed by it.
- **Step 5 - Adjust the query conditions encoded by the rule.** The query-conditions encoded in the rule must be revised to query the change operation and the dependency relation connecting the software artifacts addressed by the rule.
- **Step 6 - Define the impact report to be created by the rule.** The *ResultDefinition*-part of the rule must be changed in order to create *Impact Reports* instead of traceability links.
- **Step 7 - Validation.** We suggest that each rule should undergo a thorough test and evaluation process prior to its application in real software development projects. This can be accomplished with the help of real world case studies and artificial mock projects.

8.3.3. Example Scenario

In the remainder of this section we present a set of exemplary impact propagation rules to further illustrate our approach. We discuss from which dependency detection rules they were derived from and how they interact to determine the impacts of changes.

In Listing 6.6 in Section 6.4.4.1 we already presented an exemplary dependency detection rule (*TR_Mth_009*) to elicit *implementation*-dependencies between methods of classes and methods of interfaces (see also Figure 6.4 in Section 6.4.3.2). Due to this dependency, the changes that are applied on one of the methods should be reflected by the other method as well. Therefore, the cross-product of the dependency detection rule and potential changes is computed. One of the resulting impact propagation rules is *IR_Mth_017* that is illustrated by Listing 8.1 below.

```
<Rule Description="Rename␣the␣implementation␣of␣the␣method" RuleID="IR_Mth_017">
  <Elements type="MethodDeclaration|Operation" alias="e1"/>
  <Elements type="MethodDeclaration|Operation" alias="e2"/>
  <Elements type="AtomicChangeType" alias="e3"/>
  <Conditions>
    <BaseCondition type="ValueEquals" source="e3::name" target="" value="Rename␣method"/>
    <BaseCondition type="ModelRelatedTo" source="e2" target="e1" value="Implements"/>
  </Conditions>
  <Action actionType="ReportImpact" resultType="Rename␣method" source="e1" impact="e2"/>
</Rule>
```

Listing 8.1: Propagating rename-changes between methods of interfaces and classes

This impact propagation rule reuses the structure of the dependency detection rule *TR_Mth_009* but replaces its *ActionDefinition* to create impact reports. Likewise, a new *ElementDefinition* was added to address the change operation.

Yet, a *rename*-operation does not just affect the implemented method, but also potential unit test cases and the equivalent UML operations in class diagrams (see Figure 8.5). Therefore, additional impact propagation rules have to be derived from our dependency detection rules to address those changes. For example, impact rule *IR_Mth_016* is deployed to propagate *rename*-operations between Java and UML methods (see Listing 8.2). Likewise, impact rule *IR_Mth_018* as presented in Listing 8.3 is able to determine potential impacts on unit tests.

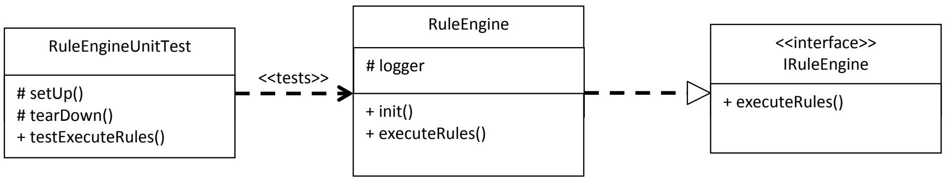


Figure 8.5.: Renaming the interface-method *executeRules* would impact the class-method *executeRules*, which in turn would impact the JUnit test-method *testExecuteRules*

A typical scenario that involves all three rules could be triggered by a developer who is renaming the method of an interface, which in turn would trigger rule *IR_Mth_017* that declares the corresponding class-method as being impacted by this change. This in turn would trigger rules *IR_Mth_016* and *IR_Mth_018*, which, based on the impacted class-method, would determine the potential impacts of the initial change on unit tests and on related UML-operations respectively.

```
<Rule Description="Rename the corresponding UML/Java method" RuleID="IR_Mth_016">
  <Elements type="MethodDeclaration|Operation" alias="e1"/>
  <Elements type="MethodDeclaration|Operation" alias="e2"/>
  <Elements type="AtomicChangeType" alias="e3"/>
  <Conditions>
    <BaseCondition type="ValueEquals" source="e3::name" target="" value="Rename method"/>
    <BaseCondition type="ModelUndirectedRelatedTo" source="e2" target="e1" value="Is-
        Equivalent-To"/>
  </Conditions>
  <Action actionType="ReportImpact" resultType="Rename method" source="e1" impact="e2"/>
</Rule>
```

Listing 8.2: Propagating rename-changes between Java and UML methods

```
<Rule Description="Rename the test method of the method" RuleID="IR_Mth_018">
  <Elements Type="MethodDeclaration|Operation" Alias="e1"/>
  <Elements Type="MethodDeclaration|Operation" Alias="e2"/>
  <Elements Type="AtomicChangeType" Alias="e3"/>
  <Conditions>
    <BaseCondition type="ValueEquals" source="e3::name" target="" value="Rename method"/>
    <BaseCondition type="ModelRelatedTo" source="e2" target="e1" value="Tests"/>
  </Conditions>
  <Action actionType="ReportImpact" resultType="Rename method" source="e1" impact="e2"/>
</Rule>
```

Listing 8.3: Propagating rename-changes between methods and test-methods

8.4. Critical Discussion and Limitations

In this section we address the potential limitations and threats to the validity of our proposed impact analysis approach. We are therefore discussing the correctness and the completeness of our set of impact propagation rules and the influence of dependency relations on the impact analysis approach. For the latter we especially focus on the effects of missing and incorrect dependency relations. Additionally, we analyze the complexity of our approach and classify it according to our taxonomy for change impact analysis approaches. Furthermore, we analyze the costs of applying our approach and discuss possible extensions of the approach.

8.4.1. Impact Rules

The following three sections address the potential implications of ambiguous, incomplete, incorrect, and missing impact propagation rules for the proposed impact analysis approach.

8.4.1.1. Ambiguous Impacts

The change impact analysis approach presented in this thesis assumes that there is only one type of impact resulting from a specific combination of change operations, software artifacts, and dependency relations. In practice, however, developers might favor alternate solutions that are not reflected by our current rules. This assumption constraints our approach and might limit its applicability as it potentially results in false-positives and missed impacts.

There are five different approaches to cope with ambiguous impacts if there is more than one impact propagation rule to react on a certain change.

1. One could apply all the n different impact rules to assess the impact of a certain type of change, which in turn results in n potentially different impact sets which have to be inspected by the developers, i.e. they have to decide for one of the alternatives. However, the benefit of its easy implementation is counteracted by the resulting manual effort.
2. The ambiguous rules could be entirely excluded from the impact analysis. Hence, the precision of our approach would increase, however at the cost of sacrificing its recall and thus resulting in missed impacts. This, however, contradicts with our goal of identifying all the artifacts that are impacted by a change and is thus less feasible in practice.
3. The developers could directly participate in the impact analysis process by automatically presenting ambiguous cases to them and letting them decide which rule is to be applied. Thus, whenever there is more than one rule to react on a given change, they have to chose between a set of rules according to the current situation. Consequently, this approach results in additional manual effort but at the same time improves the efficiency of the impact analysis in ambiguous situations.
4. If the developers are allowed to chose which rules are to be applied (see third option above), their choices could be recorded to afterwards compute the likelihood that a certain rule is always chosen in a given situation. Thus, after recording and processing a sufficient amount of developer decisions, probabilistic reasoning could be applied to automatically select a specific rule when there are several potentially suitable rules available.
5. As a further derivation of the 4th option, one could also apply the probabilistic concept to only suggest or recommend certain rules to the developers while leaving the final decision to them. In doing so, the developer is still in control but is able to benefit from his previous actions and thus requires less time to take a decision.

Although we did not implement or evaluate any of the above listed solutions, the third option seems most promising as it allows for a tighter integration of change impact analysis into the workflows of software developers. Given that a suitable probabilistic approach can be found, the fifth option might be applied as well, if it results in a significant reduction of manual effort.

8.4.1.2. Correctness of the Impact Rules

The second factor that might hamper the applicability of our approach is the correctness of the impact propagation rules that implement the analysis of the interplay of changes and dependencies. We identified three potential threats to the correctness of our impact rules that are almost identical to the threats to our dependency detection rules as discussed in Section 6.6.2:

- A rule triggers when it is not supposed to.
- A rule fails to trigger when required.
- A rule determines the wrong type of impact.

The first two threats both point towards the same underlying problem, namely that the assumed interplay of change types, dependency types, and artifact types encoded within a rule is incorrect. Thus, the rule has to be revised. Our concept of impact reports assists developers with tracing the propagation of impacts and therefore allows them to determine the incorrect impact reports based on the software artifacts, dependency relations, and changes that are encoded in each impact report. If an incorrect impact report is identified, the rule that created this report is revealed too since each report is linked with the responsible rule. Once the incorrect rule is identified, it can be revised according to the results of our discussion of the effects of changes in Section 8.2 and our scheme for developing impact propagation rules (see Section 8.3.2).

The third case requires a different treatment as the impact that is determined by a rule is not correctly encoded, i.e. either the wrong type of impact or the wrong impacted element is specified in the rule. Consequently, the *ActionDefinition*-part of the rule should be revised.

Moreover, since developers are able to retrace the propagation of changes and their impacts by inspecting the generated impact reports, they are able to spot false-positives more easily when compared to other approaches that do not provide these information, since no impact analysis approach will ever achieve a precision of 100%.

8.4.1.3. Completeness of the Impact Rules

The third category of threats to the validity of our approach is characterized by the absence of rules implementing the interplay of certain types of changes, artifacts, and dependencies. Hence, the obvious consequence of a missing rules is that the impact of a certain change cannot be entirely forecast, which in turn will render the overall impact estimation incomplete as well.

In order to ensure that the set of impact rules accompanying this thesis is as complete as possible and to facilitate the addition of new impact rules, we established a guideline for the creation of impact rules in Section 8.3.2. If a change that is not covered by any rule is observed during the application of our approach, the necessary rule(s) can be created according to our guidelines, added to the set of already available rules, and can then be applied on any software system thereafter. Moreover, with our approach of computing the cross-product of an existing dependency detection rule and the change operations that can be applied on the artifacts addressed by the rule, we provide complete coverage of all the dependency relations that exist between those software artifacts, which cannot be guaranteed by other impact analysis approaches.

On the other hand, the completeness of our rule catalog cannot be measured or determined in a theoretical manner, though. However, as the upcoming evaluation in Chapter 10 will show, the current rule catalog already provides solid coverage of many types of changes and software artifacts. Furthermore, an earlier version of this rule catalog was applied in an initial case study reported in [LFR13] and has been enhanced with yet missing rules due to ongoing research.

Additionally, even though our catalog of impact rules might not be complete and thus never achieve a recall of 100%, it at least provides developers with a good estimate of the impacts of a change and allows them to inspect those software artifacts and dependencies more closely that are considered as being impacted to identify potentially missing impacts.

8.4.2. Influence of Dependency Relations

In the following we are discussing the implications of missing and incorrect dependency relations on the proposed impact analysis approach, since the detection of dependency relations is no trivial task and faces various obstacles (see discussions in Section 6.6). Secondly, we illustrate how the impacts of changes influence dependency relations and how these impacts on dependencies can be addressed.

8.4.2.1. Missing Dependencies

We have to distinguish between two types of worst-case scenarios that revolve around missing dependencies. First, if no dependency relations at all were made explicit our impact analysis approach cannot be applied. Secondly, if the actually present dependency relations have only been partially recorded, then only a subset of our impact rules can be applied, which in turn results in an incomplete estimation of the overall impact of a change.

Both issues can only be addressed by providing reliable means for multiperspective dependency detection. Therefore, we developed a comprehensive approach for detecting dependencies that is able to analyze the different sources and types of dependency relations, as well as the different properties of software artifacts (see Section 6.4). In contrast to other dependency detection approaches, our approach takes into account a combination of structural and textual information of software artifacts, as well as already existing relations between them, and is thus able to detect more and more fine-grained dependencies (see Section 6.6.1). In a similar fashion in our approach we consider four distinct sources of potential dependency relations which, to the best of our knowledge, is not done by other works. Consequently, our approach covers a wider variety of dependency relations, which is also confirmed by our case studies on rule-based dependency detection (see Section 6.5).

If still no dependency relations can be recorded other impact analysis techniques might be applied instead. However, the majority of the proposed approaches also depends on explicit dependency information and therefore suffers from the same limitations (see Section 2.3).

8.4.2.2. Incorrect Dependencies

The quality and correctness of the impact assessments computed by our approach also depends on the quality of the underlying dependency relations that are used to determine the propagation of changes. In contrast to the absence of dependencies, the presence of incorrect dependencies may mislead the recursive impact propagation as the wrong impact rules could be triggered.

Consequently, the estimated overall impact will be incorrect as well. Likewise, incorrect dependencies may cause the wrong type of impact to be detected by our impact propagation rules.

Thus, the precision of our approach also depends on the precision of our dependency detection rules that are being used for eliciting the dependencies. However, we already discussed the threats to the correctness of our dependency detection rules in Section 6.6.2 and outlined potential solutions to mitigate the effects of incorrect dependency detection rules.

Finally, when we compare our rule-based approach to other impact analysis techniques as reviewed in Section 2.3, the precision of most approaches is also directly tied to the quality of the analyzed dependency relations, which is why they share the same potential limitation.

8.4.2.3. Impacts on Dependencies

The impacts of changes do not only influence software artifacts, such as UML diagrams or Java source code, but also the dependency relations between them. However, the approach presented in this thesis does not yet address these impacts on dependency relations. Our approach currently assumes that the dependency relations are to be re-detected after changes were applied on the system in order to capture the modified dependencies and to remove possibly outdated dependencies. Thus, the frequent re-detection of dependencies might introduce a certain overhead to the entire change impact analysis process. This overhead is partially reduced by the automated approach for multiperspective dependency detection as introduced in Chapter 6, which can be executed immediately after changes were applied on the software artifacts.

However, maintaining and updating dependency relations during the evolution of software is not in the scope of this thesis and is being dealt with in a different field of research, namely *traceability maintenance*. A detailed discussion of this specific maintenance problem can be found in the works of Mäder *et al.* [MRP06a, MGP08, Mö9] and in the work of Murta *et al.* [MvdHW06]. Nevertheless, we will refer back to this problem when we discuss how future works can extend the concepts presented in this thesis in Section 11.3.

8.4.3. Cost Trade-Off of the Approach

Our approach also demands for a discussion of its overall costs issued by the effort required for transforming the software artifacts into EMF-based models, importing them into the model repository, eliciting their dependencies, and applying the impact propagation rules. The underlying question is whether these costs are higher than the costs inflicted by incomplete or incorrect implementations of changes. One could argue that, if the costs of our approach are higher, no impact analysis or pure manual analysis should be conducted instead.

It is, however, obvious that this discussion cannot be based on real (monetary) figures taken from existing projects as they would hardly be comparable in practice. Moreover, such figures would also be hard to obtain since no one would implement the same change twice using two different approaches. Instead, we discuss the conceptual effort issued by our approach and put it into relation with other benefits that can be drawn from the data generated by our approach, such as the explicit dependency relations for example.

Most of our approach's effort is spent on the detection of dependencies and the later application of our impact propagation rules. However, the effort for the detection of dependencies is not in vain as the recorded dependencies can be reused for multiple purposes. They can be utilized for regression testing [FLR14], program comprehension [BFV00], program visualization [LM12], and the documentation of design decisions [GLR14]. They also support the traceability that is required to comply to software development standards, such as IEEE830-1998 [Ins98a] for instance. Moreover, once an initial set of dependencies is discovered, a suitable approach for maintaining these dependencies can be applied, making the frequent re-detection of dependencies obsolete and thus further reducing the overall costs of our approach.

The costs of the actual change impact analysis stand in contrast to costs issued by potential bugs and system failures, which of course cannot be forecast. However, there are various examples of accidents that were caused by software failures after changes were applied on the systems. These failures eventually resulted in enormous costs and in the loss of human life that could have been prevented by proper impact analysis. A rather tragic example is a series of incidents caused by software updates applied on a cancer radiation therapy machine resulting in radiation overdoses killing and severely injuring several people [LT93]. Additionally, our approach allows developers to simulate alternative solutions for a given problem if a change can be implemented in different ways. Therefore, it provides additional benefits over more time consuming manual analyses, which do not allow for simulating such alternatives with reasonable effort.

The only case when the costs of the impact analysis and all of its necessary preprocessing steps are higher than potential gains are very simple change scenarios that can easily be resolved by developers. In such cases the overhead of applying our approach will be much higher than the effort for implementing the changes right away without conducting any impact analysis. Likewise, when only source code is available the inspection of program dependencies as provided by modern IDEs (e.g. Eclipse) might be sufficient to support the implementation of changes.

8.4.4. Complexity of the Approach

Next to the efficiency of our approach in terms of precision and recall (see our evaluation in Chapter 10) we also report on its theoretical space and time complexity. Therefore, we first illustrate the worst case scenario in regard to our recursive rule-based impact analysis in which every artifact of a software system is potentially dependent on any other artifact of the same system. Thus, the whole set of dependencies forms one comprehensive graph that is equivalent to the transitive closure of all dependencies. Consequently, there are $n(n-1)/2$ edges (i.e. dependency relations) in a graph containing n nodes (i.e. software artifacts).

The space complexity of our approach can be estimated by calculating the space complexity of executing a single impact propagation rule, as in our concept only one rule is executed at a time. Furthermore, the memory consumption of our modified A*-algorithm that is applied to monitor the recursive rule application adds to the overall space complexity of the approach. Since our impact propagation rules are directly derived from our dependency detection rules, they share the same space complexity, namely $O(n^2)$ (see Section 6.6.4). The maximum space required by the modified A*-algorithm can be determined as follows. Since there are $n(n-1)/2$ dependency relations that can be inspected by our rules and each relation can carry an impact only once, the maximum size of the *OpenList* and *ClosedList* stored by our modified algorithm also equals $n(n-1)/2$. Therefore, the total worst case space complexity of our approach computes to

$n^2 + n(n-1)/2 + n(n-1)/2$, which finally results in the following:

$$s(n) = O(n^2) \tag{8.2}$$

Due to our recursive propagation concept, the time complexity of our approach is computed in a different way. Therefore, we have to compute the time complexity of executing a single rule and determine how often the rules are being executed. Additionally, the overhead of the modified A*-algorithm must be taken into account as well. The first question is easily answered as our impact propagation rules are derived from our dependency detection rules, thus both possess the same worst case time complexity of $O(n^2)$. The second question demands for a more thorough discussion of what is happening in the worst case when there are $n(n-1)/2$ dependencies. When our approach is applied on this set of relations, the recursive propagation inspects each dependency relation and executes all impact rules on each dependency relation. However, due to our concept, each dependency relation can propagate an impact only once, thus each impact rule is maximally executed $n(n-1)/2$ times. Thirdly, the computational overhead issued by our modified A*-algorithm accounts every time a rule creates an impact report as the *ClosedList* has to be searched for the same element, which requires $n(n-1)/2$ operations. Therefore, the time complexity is computed as $n(n-1)/2 \cdot (n^2 + n(n-1)/2)$, which leads to the following overall worst case time complexity of our rule-based impact analysis approach:

$$t(n) = O(n^4) \tag{8.3}$$

8.4.5. Classification of the Approach

In this section we classify the presented approach according to our taxonomy of impact analysis approaches as introduced in Section 2.2. By classifying our own approach, we facilitate its comparability with other approaches proposed in the literature, which in turn assists with outlining its contributions and may help developers or researchers deciding for an approach.

- Scope of Analysis: code, architecture, test cases
- Granularity of Artifacts: adaptable
- Granularity of Changes: add, delete, update, move, merge, split, swap, replace
- Granularity of Results: adaptable
- Utilized Technique: rule-based analysis
- Tool Support: Eclipse-based EMFTrace
- Supported Languages: Java, UML, JUnit
- Style of Analysis: search-based
- Scalability - Time complexity: $O(n^4)$
- Scalability - Space complexity: $O(n^2)$
- Experimental Results - Size: 25 kLOC, 124 unit tests, 40 UML diagrams, 210 changes
- Experimental Results - Precision: 90.96%
- Experimental Results - Recall: 93.96%

- Experimental Results - Computation Time: 5 minutes

8.4.6. (Semi)-Automated Change Implementation

Our concept of impact propagation rules and the impact reports created by them offer an additional advantage, as they allow for a (semi)-automated implementation of changes. Instead of presenting the impact reports to developers only, one could also directly implement the changes encoded within them in an automated fashion, given that an adequate processing concept is available. Our impact reports contain the necessary information for implementing most changes, such as the artifact to be changed and the type of the change to be performed. Thus, a wide variety of change operations could already be automated with the given concept, such as *add*, *delete* or *rename* operations. Even complex operations, such as merging or splitting interfaces and classes, could be realized to a certain extent. For example, the refactoring operation of “merging two classes” could be fully automated except for providing a meaningful name to the new class. Hence, certain types of change operations would still require manual interference by developers. All other sub-operations, such as moving all the methods and attributes to the merged class or deleting the 2nd class after the merge, can be fully automated by utilizing our change type taxonomy and the information provided by the impact reports.

However, this concept is out of the scope of the thesis at hand which is focused on providing a reliable mechanism for studying the propagation of changes. Nevertheless, this thesis can provide the basis for a (semi)-automated approach for the implementation of changes.

8.5. Summary

In this chapter we presented our novel approach for conducting multiperspective change impact analysis in order to accomplish **Goal 1**, **Goal 2**, and **Goal 3** of this thesis. We illustrated how a set of impact propagation rules can be utilized for estimating the propagation of changes and their impacts on the interplay of different types of software artifacts and how this propagation can be computed in a recursive manner. We discussed how this recursive propagation strategy is monitored and how the impact reports created by our concept enable developers to understand the resulting impact propagation. Moreover, the impact propagation rules were presented and discussed in regard to their structure and origin and a step-wise guide was developed for creating additional impact propagation rules to accomplish **Goal 5**. The proposed approach was classified according to our taxonomy for change impact analysis approaches as presented in Section 2.2 to facilitate the comparison of our approach with other change impact analysis approaches. Finally, the critical points of the presented approach were discussed, its time and space complexity were analyzed, and potential directions for further research were outlined.

9. The EMFTrace Prototype

This chapter presents the prototype tool EMFTrace that implements the approach proposed in this thesis. This chapter is organized as follows. Section 9.1 presents an overview of the tool and explains its purpose, while Section 9.2 discusses the main use cases EMFTrace has been developed for. Subsequently, Section 9.3 explains the overall architecture and the main components of our prototype and discusses the extensibility of our architecture. Finally, Section 9.4 reports on the current status of EMFTrace and outlines ongoing and future development.

9.1. Overview

EMFTrace was initially developed for performing automated traceability mining among UML, OWL, and URN models using a rule-based approach [Leh10, BLR11, RBFL11]. EMFTrace is build upon the Eclipse Modeling Framework (EMF) [EMFa] that provides a common meta-model on which all the software artifacts can be mapped upon (see Section 5.2.1). It extends the EMFStore model repository [EMFc] that is used for managing the heterogeneous software artifacts in a centralized manner. Furthermore, the Eclipse Client Platform (ECP) [ECP] supplies the basic components of the user interface of EMFTrace. The entire tool is programmed in Java and is comprised of several Eclipse plug-ins.

EMFTrace was continuously developed and extended to implement the novel concepts presented in this thesis. It supports the rule-based dependency detection (see Chapter 6), rule-based impact analysis (see Chapter 8), and the distinct modeling of dependency relations (see Chapter 6) and change operations (see Chapter 7).

Additionally, EMFTrace offers the following features:

- Import of Java source code and JUnit test cases from the Eclipse IDE.
- Import of various design models from different CASE tools, e.g. UML models.
- Creation and validation of dependency detection rules.
- Creation and validation of impact analysis rules.
- Basic maintenance support for traceability links.
- Visualization of dependencies and transitive dependency chains.

EMFTrace is an open source project published under the Eclipse Public License (EPL) [EPL] and is available for download on Sourceforge [EMF14]. EMFTrace currently supports and interacts with the following case tools and IDEs as shown in Figure 9.1.

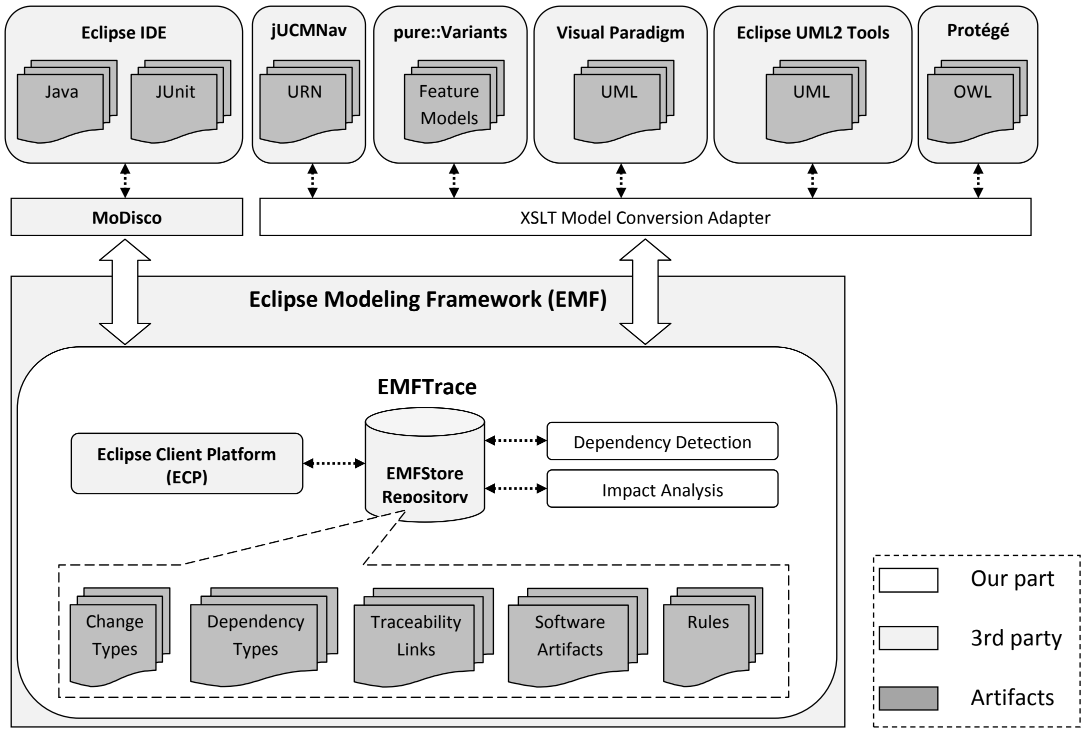


Figure 9.1.: EMFTrace and its relation to other 3rd party frameworks and tools

9.2. Typical Use Cases

In the following sections we describe the three main use cases of EMFTrace and illustrate how EMFTrace supports software developers when they are analyzing and maintaining software systems. It is assumed that the underlying EMFStore repository is already populated with the elements to be analyzed, as this section is solely focused on the application of EMFTrace for typical software maintenance and reengineering tasks. For detailed descriptions of how EMF-Trace supports the population of EMFStore with various types of software artifacts, we refer to the online documentation of EMFTrace[1] and to the online documentation of EMFStore[2].

9.2.1. Automated Dependency Detection

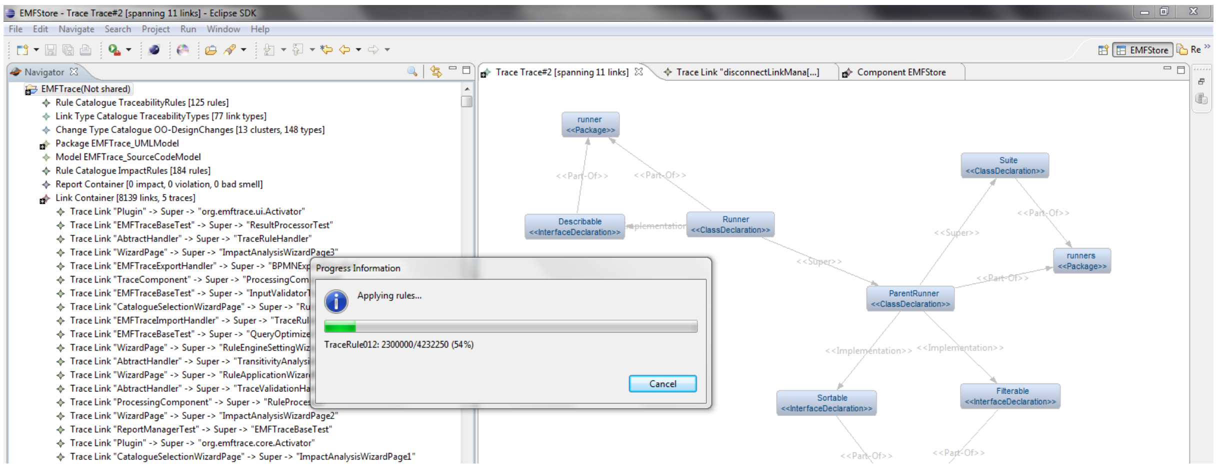
Developers need to be aware of potential dependency relations between software artifacts when they are analyzing and changing software systems. We therefore provide them with the required means for eliciting those dependencies using our prototype tool that implements the comprehensive rule-based concept for multiperspective dependency detection as introduced in Chapter 6. Consequently, EMFTrace provides means to execute dependency detection rules and to record the resulting dependency relations as traceability links. The following steps should be taken by developers in order to execute the automated dependency analysis:

1. Import dependency detection rules into EMFTrace.

[1]https://sourceforge.net/p/emftrace/wiki/Home/
[2]http://eclipse.org/emfstore/documentation.html

2. Select the rule(s) to be applied.
3. Select the software artifact(s) to be searched.
4. Start the execution of rule(s).

The developers are guided by Eclipse UI wizards during each of the above listed tasks to increase their productivity and to limit human error. Figure 9.2 presents an actual screenshot of EMFTrace while executing rules and extracting dependencies. Once the extraction is done, all the obtained links can be visualized, which is further explained in the next section.


EMFStore - Trace Trace#2 [spanning 11 links] - Eclipse SDK
File Edit Navigate Search Project Run Window Help
Navigator
Trace Trace#2 [spanning 11 links]
Trace Link "disconnectLinkMana[...]
Component EMFStore
EMFStore
Re
runner <<Package>>
<<Part-Of>>
<<Part-Of>>
Describable <<InterfaceDeclaration>>
Implementation
Runner <<ClassDeclaration>>
Suite <<ClassDeclaration>>
<<Part-Of>>
<<Super>>
runners <<Package>>
<<Super>>
<<Part-Of>>
ParentRunner <<ClassDeclaration>>
<<Implementation>>
<<Implementation>>
Sortable <<InterfaceDeclaration>>
Filterable <<InterfaceDeclaration>>
<<Part-Of>>
<<Part-Of>>
Progress Information
Cancel

```
EMFTrace(Not shared)
  Rule Catalogue TraceabilityRules [125 rules]
  Link Type Catalogue TraceabilityTypes [77 link types]
  Change Type Catalogue OO-DesignChanges [13 clusters, 148 types]
  Package EMFTrace_UMLModel
  Model EMFTrace_SourceCodeModel
  Rule Catalogue ImpactRules [184 rules]
  Report Container [0 impact, 0 violation, 0 bad smell]
  Link Container [8139 links, 5 traces]
    Trace Link "Plugin" -> Super -> "org.emftrace.ui.Activator"
    Trace Link "EMFTraceBaseTest" -> Super -> "ResultProcessorTest"
    Trace Link "AbtractHandler" -> Super -> "TraceRuleHandler"
    Trace Link "WizardPage" -> Super -> "ImpactAnalysisWizardPage3"
    Trace Link "EMFTraceExportHandler" -> Super -> "BPMNExp
    Trace Link "TraceComponent" -> Super -> "ProcessingCom
    Trace Link "EMFTraceBaseTest" -> Super -> "InputValidatorT
    Trace Link "CatalogueSelectionWizardPage" -> Super -> "Ru
    Trace Link "EMFTraceImportHandler" -> Super -> "TraceRul
    Trace Link "EMFTraceBaseTest" -> Super -> "QueryOptimize
    Trace Link "WizardPage" -> Super -> "RuleEngineSettingWiz
    Trace Link "AbtractHandler" -> Super -> "TransitivityAnalysi
    Trace Link "WizardPage" -> Super -> "RuleApplicationWizar
    Trace Link "AbtractHandler" -> Super -> "TraceValidationHa
    Trace Link "ProcessingComponent" -> Super -> "RuleProces
    Trace Link "WizardPage" -> Super -> "ImpactAnalysisWizardPage2"
    Trace Link "ReportManagerTest" -> Super -> "EMFTraceBaseTest"
    Trace Link "Plugin" -> Super -> "org.emftrace.core.Activator"
    Trace Link "CatalogueSelectionWizardPage" -> Super -> "ImpactAnalysisWizardPage1"
```

```
Applying rules...
TraceRule012: 2300000/4232250 (54%)
```

Figure 9.2.: EMFTrace while searching for dependency relations (screenshot)

9.2.2. Program Comprehension

Once the dependency relations are explicitly recorded, developers must be enabled to browse, query, and comprehend them. Understanding the relations can best be supported by providing an appropriate visualization concept for them. Related works on software visualization already proposed a variety techniques for visualizing dependency relations (e.g. [Kos03], [GC08], and [LM12]), out of which EMFTrace implements two visualization concepts. EMFTrace provides a fish-eye-view [Fur86] of the direct dependencies of a certain software artifact and a second view for visualizing transitive chains of dependency relations. Developers have to execute the following steps in order to browse and explore the dependencies:

1. Execute the dependency analysis (see previous section).
2. Select the model of interest and bring up its context menu.
3. Select “Show Model Dependencies” from the menu.

EMFTrace allows developers to browse through all the dependencies in a step-wise manner (fish-eye-view) or to render all the dependencies into one comprehensive graph. For the most time, developers ought to prefer the fish-eye-view as holistic dependency graphs tend to be too complex for comprehension. Figure 9.3 provides a screenshot illustrating how both visualization techniques are implemented by EMFTrace.

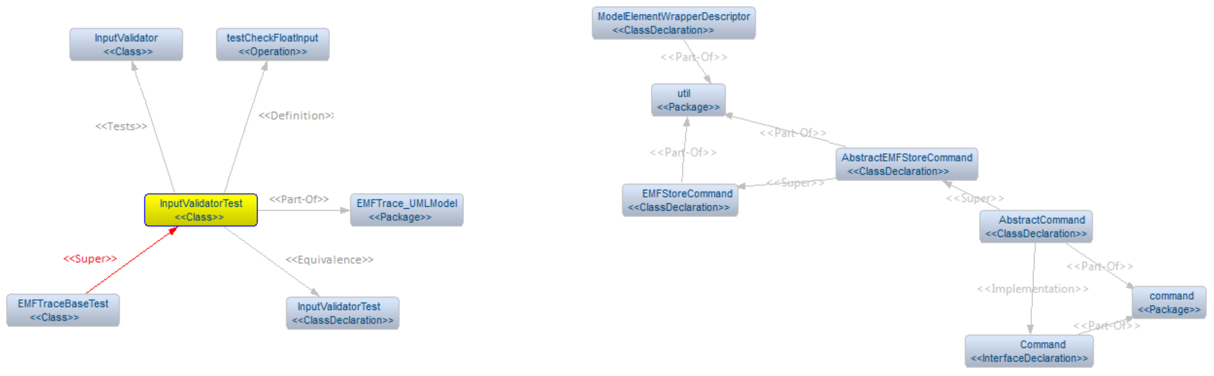

InputValidator <<Class>>
testCheckFloatInput <<Operation>>
<<Tests>>
<<Definition>>
InputValidatorTest <<Class>>
<<Part-Of>>
EMFTrace_UMLModel <<Package>>
<<Super>>
<<Equivalence>>
EMFTraceBaseTest <<Class>>
InputValidatorTest <<ClassDeclaration>>
ModelElementWrapperDescriptor <<ClassDeclaration>>
<<Part-Of>>
util <<Package>>
<<Part-Of>>
<<Part-Of>>
AbstractEMFStoreCommand <<ClassDeclaration>>
EMFStoreCommand <<ClassDeclaration>>
<<Super>>
<<Super>>
AbstractCommand <<ClassDeclaration>>
<<Part-Of>>
<<Implementation>>
command <<Package>>
<<Part-Of>>
Command <<InterfaceDeclaration>>

Figure 9.3.: Visualization of direct dependencies using the fish-eye-view (left) and transitive dependency chains (right) in EMFTrace (screenshot)

9.2.3. Change Impact Analysis

After analyzing the software artifacts for dependencies, developers might want to change a certain artifact and assess the impacts of this change. Therefore, EMFTrace also implements the impact analysis approach presented in Chapter 8 and guides the developers through the impact analysis process with the help of various Eclipse-based wizards and dialogs. Developers have to follow the steps listed below in order to perform the change impact analysis:

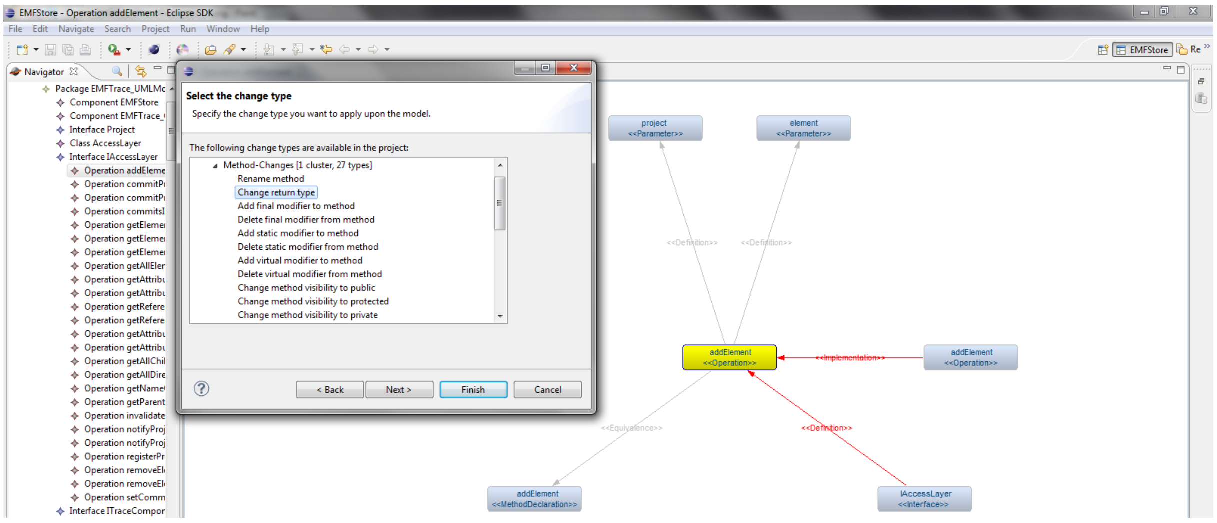
1. Import a set of impact analysis rules into EMFTrace.
2. Select the model to be changed.
3. Select the type of change to be applied on the model.
4. Start the rule execution.
5. Inspect the generated impact report(s).

EMFTrace then applies the impact analysis rules according to the approach introduced in Chapter 8. After the impact analysis rules were executed, the developers will be informed about the amount of impacted artifacts and they will be supplied with comprehensive reports that state why and how certain software artifacts are impacted by the initial change. Before the impact analysis is executed, however, the developers can chose whether they want to receive one comprehensive impact report or rather a set of reports (i.e. one for each impacted software artifact). Figure 9.4 provides a screenshot of conducting impact analysis using EMFTrace.

9.3. Architecture

This section provides a brief overview of the architecture of EMFTrace and elaborates on how it can be extended in future work.

As previously stated, EMFTrace is based on Eclipse technology, most notably the Eclipse plug-in concept. It extends the EMFStore repository by providing additional features that are hooked into the existing EMFStore client. This provides potential users with a unified view and reduces the amount of tools they have to deal with. Overall, EMFTrace consists of four plug-ins:



Select the change type

Specify the change type you want to apply upon the model.

The following change types are available in the project:

- Method-Changes [1 cluster, 27 types]
  - Rename method
  - Change return type
  - Add final modifier to method
  - Delete final modifier from method
  - Add static modifier to method
  - Delete static modifier from method
  - Add virtual modifier to method
  - Delete virtual modifier from method
  - Change method visibility to public
  - Change method visibility to protected
  - Change method visibility to private

Figure 9.4.: Performing impact analysis when changing the return type of the method "addElement" that is centered on the screen (screenshot)

- *org.emftrace.core*: contains the core functionality provided by EMFTrace.
- *org.emftrace.ui*: contains extensions to the user interface of EMFStore.
- *org.emftrace.metamodel*: contains the meta-models supplied by EMFTrace.
- *org.emftrace.metamodel.edit*: contains the editor code to manipulate these models.

The functional core of EMFTrace is comprised of five main components that are displayed by Figure 9.5 and are explained in the remainder of this section.

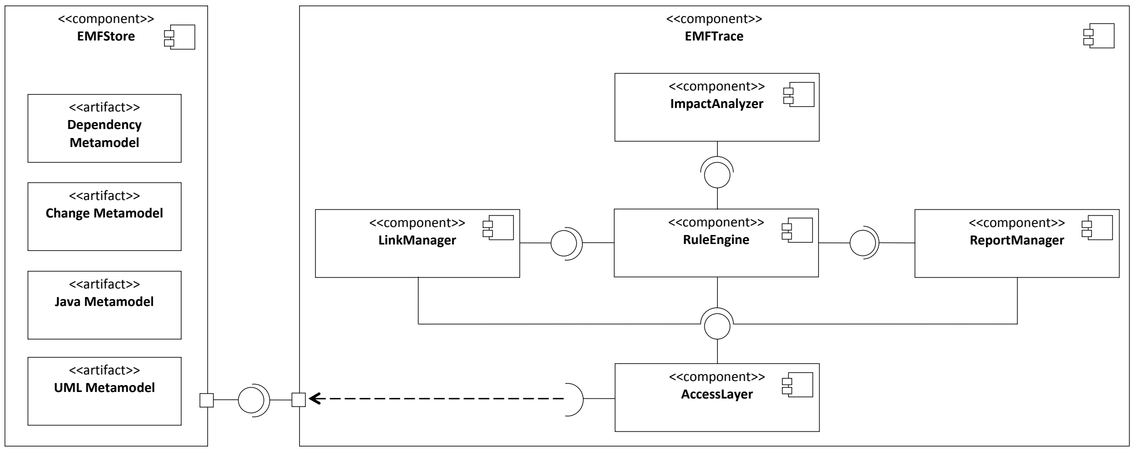

<<component>> EMFStore
<<artifact>> Dependency Metamodel
<<artifact>> Change Metamodel
<<artifact>> Java Metamodel
<<artifact>> UML Metamodel
<<component>> EMFTrace
<<component>> ImpactAnalyzer
<<component>> LinkManager
<<component>> RuleEngine
<<component>> ReportManager
<<component>> AccessLayer

Figure 9.5.: The core components of EMFTrace

The **AccessLayer** is *the* core component of EMFTrace. It acts as a wrapper for EMFStore by providing access to all the models deposited in the repository. Moreover, it provides a set of operations to simplify the querying for relations, attributes, and the structure of those models.

The **RuleEngine** is the 2nd core component next to the AccessLayer. It is responsible for executing the dependency detection rules (see Section 6.4) and the impact analysis rules (see

Section 8.3). It offers features for executing the query conditions of our rules and for joining the results of different query conditions afterwards. Furthermore, it also provides means for validating the rules prior to their application (i.e. performing syntax checking).

The **LinkManager** is responsible for creating and maintaining traceability links that express dependencies. It is able to search for existing links to prevent the creation of duplicated entries and it can repair or remove broken or outdated links. Additionally, it allows to search for transitive relations between a set of links that are then grouped by a trace (see Section 6.2).

The **ReportManager** is responsible for creating different types of reports, such as impact analysis reports and consistency checking reports. Its functionality is therefore similar to the LinkManager, i.e. it allows for creating and validating reports.

The **ImpactAnalyzer** implements the proposed recursive rule-based impact propagation approach as introduced in Section 8.1 and the distance-based impact analysis approach as discussed in Section 2.3.1.1 due to the 2nd goal of our case study (see Section 10.1.2).

The internal architecture of EMFTrace is realized as illustrated by Figure 9.6. All core components extend the *TraceComponent* that is responsible for granting access to the *AccessLayer*-component and furthermore provides a comprehensive logging mechanism. Any new feature that shall be added to EMFTrace can thus extend the *TraceComponent* to obtain full and comfortable access to the models stored in the repository along with the optional logging features.

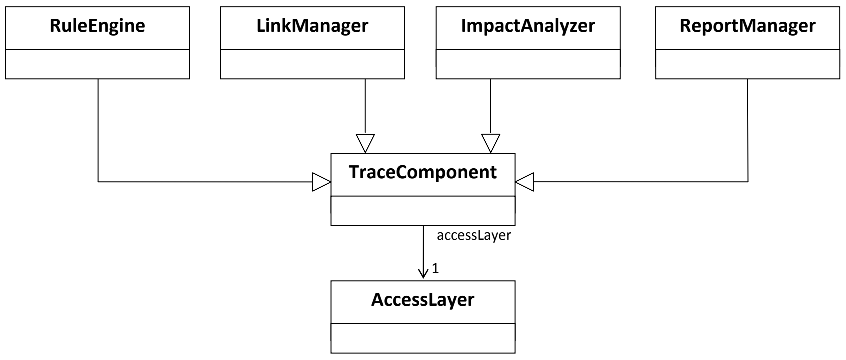


Figure 9.6.: The internal structure of the core components of EMFTrace

9.4. Status and Summary

In this chapter we discussed our prototype tool EMFTrace that implements the two main concepts and innovations introduced in this thesis: multiperspective dependency detection and multiperspective impact analysis. EMFTrace extends the EMFStore model repository and is comprised of five main components that provide means for applying rules for dependency detection and impact analysis, and support the evolution of traceability links. The architecture of EMFTrace is extensible and allows for the addition of new components to support new use cases. EMFTrace is still subject to ongoing development and improvement. Further development is currently focused on two aspects: replacing the current rule-processing infrastructure by query-frameworks, such as EMF Query [EMFb], and improving the dependency visualization.

10. Evaluation

The change impact analysis approach proposed in thesis and all the impact rules that were developed during the course of this thesis' research were evaluated by a comprehensive case study which is discussed in this chapter. During the course of this case study various reengineering and maintenance measures were applied to an existing software system, while our approach was deployed to forecast the impacts of those changes. Afterwards, the output of our approach was compared against the true impacts of the changes as determined by manual analysis and the output computed by other tools and approaches. Overall, our evaluation follows the *Goal Question Metric* approach [BCR94] and the guidelines for conducting and reporting case study research in software engineering [RH09].

This chapter is organized as follows. First, we present our research goals and research questions in Section 10.1 and discuss the design of our evaluation in Section 10.2. We then present the results of our study in Section 10.3 and interpret and explain them in Section 10.4. Finally, we discuss and address potential threats to the validity of our case study in Section 10.5.

10.1. Goals and Research Questions

In this section we present the two research goals that act as drivers for our evaluation. We further present the measures, metrics, and hypothesis that are related to these goals in order to conduct our evaluation.

10.1.1. RQ1: Support for Heterogeneous Software Artifacts

Is the proposed change impact analysis approach able to predict the impacts of changes that are applied on multiperspective software systems?

First and foremost, we need to evaluate if our impact analysis approach is able to correctly determine the impacts of changes that are applied on heterogeneous software artifacts. We therefore apply changes on different types of software artifacts and study the resulting impact propagation, where our approach should achieve reliable results for each kind of software artifact. This is achieved by comparing the results computed by our approach against the results determined by an "oracle". This research question directly corresponds to **Goal 1** and **Goal 2** of this thesis.

10.1.1.1. Measures

There are two important measures when determining the impacts of changes, the *Actual Impact Set* (AIS) and the *Estimated Impact Set* (EIS). The AIS encompasses the set of impacted software artifacts as determined by the oracle [AB93]. This set constitutes the true impact of a change and is obtained for each change operation (AIS_i). In contrast, the EIS denotes the set of impacted software artifacts as determined by the impact analysis algorithm [AB93]. Consequently, this set might contain false-positives or lack impacted artifacts, depending on the quality of the impact analysis. This set is also obtained for each change operation (EIS_i).

10.1.1.2. Metrics

The empirical evaluation of our impact analysis approach can be accomplished by utilizing the metrics of *precision* and *recall* that are based on the previously defined AIS and EIS. The *precision*-metric (P) determines the correctness of the computed impact sets. Hence, we have to determine the precision for each change operation P_i and the global average precision $P_{\varnothing}$ as follows.

$$P_i = \frac{|EIS_i \cap AIS_i|}{|EIS_i|} \qquad P_{\varnothing} = \frac{\sum_{i=1}^{n} P_i}{n}$$

Similarly, the *recall*-metric (R) determines whether all impacted software artifacts were identified by an impact analysis algorithm. Likewise, we have to determine the recall for each change operation R_i, as well as the global average recall $R_{\varnothing}$ as follows.

$$R_i = \frac{|EIS_i \cap AIS_i|}{|AIS_i|} \qquad R_{\varnothing} = \frac{\sum_{i=1}^{n} R_i}{n}$$

In order to answer RQ1, we have to determine the precision and recall of our approach per type of software artifact (e.g. component, method, use case, etc.) and set it in relation to the global average precision and recall of our approach. Hence, we also have to determine the average precision per type of software artifact P_t, the average recall per type of software artifact R_t, and the standard deviation of precision $sd(P)$ and recall $sd(R)$ respectively.

$$sd(P) = \sqrt{\frac{\sum_{i=1}^{n} (P_i - P_{\varnothing})^2}{n-1}} \qquad sd(R) = \sqrt{\frac{\sum_{i=1}^{n} (R_i - R_{\varnothing})^2}{n-1}}$$

Finally, we can determine the deviation of precision per type of software artifact $d(P_t)$ and the deviation of recall per type of software artifact $d(R_t)$ as follows.

$$d(P_t) = |P_{\varnothing} - P_t| \qquad d(R_t) = |R_{\varnothing} - R_t|$$

10.1.1.3. Hypotheses

We apply statistical hypothesis testing to answer our research question. Thus, we have to establish a null hypothesis H_0 that must be rejected through significance testing. We reject the null hypothesis H_0 and instead prefer the alternative hypothesis H_a if the data collected during our evaluation supports the null hypothesis H_0 with a probability of $\alpha = 0.05$ or less. Our null hypothesis H_0 regarding RQ1 is that the deviation of recall and precision for at least one type of software artifact is greater than the total standard deviation of recall and precision, which means that our approach is dependent on the software artifacts to be analyzed and is thus hardly generalizable. The alternative hypothesis H_a states that the deviation of recall and precision for each type of artifact is lower or equal compared to the standard deviation of precision and recall (see Table 10.2). In this case our approach would achieve similar and stable results for each type of software artifact. The hypothesis is tested by *Student's one-sided one-sample t-test.*

Variable	**Null hypothesis** H_0	**Alternative hypothesis** H_a
deviation of precision	$\exists d(P_t) : \lvert d(P_t)\rvert > \lvert sd(P)\rvert$	$\forall d(P_t) : \lvert d(P_t)\rvert \leq \lvert sd(P)\rvert$
deviation of recall	$\exists d(R_t) : \lvert d(R_t)\rvert > \lvert sd(R)\rvert$	$\forall d(R_t) : \lvert d(R_t)\rvert \leq \lvert sd(R)\rvert$

Table 10.2.: Tested hypotheses to answer RQ1

10.1.2. RQ2: Performance Improvements

Does the proposed rule-based impact analysis approach result in better precision and recall than existing dependency-based impact analysis approaches?

Apart from comparing the results of our approach against the oracle, we also need to find out how our approach compares to other dependency-based impact analysis approaches to demonstrate its benefits in terms of better precision and recall. Therefore, we compare our approach against the concept of distance-based impact analysis as proposed by Bohner [Boh02a] (see also Section 2.3.1.1). The reasons why this approach was chosen are manifold. First, we cannot apply MSR-based techniques due to the lack of a common version history (see Section 2.3.2). Second, other dependency-based techniques, such as call graph analysis or program slicing, are only applicable on source code (see Section 2.3.1). Third, like our approach Bohner explicitly takes into account design and code and is thus comparable in regard to its goals.

The point is that a more sophisticated approach like ours should yield better results, as otherwise the more simple solution could be utilized instead. For this purpose we apply the distance-based change impact analysis approach with a propagation distance of 2, which is based on empirical results of Hassaine *et al.* [HBG$^+$11] who studied the propagation distance of various change operations. This research question also corresponds to **Goal 1** and **Goal 2** of this thesis.

10.1.2.1. Measures

For answering this research question we apply the same measures as for RQ1. In addition, however, we also have to consider the impact sets computed by the distance-based approach (EIS_{db_i}). Thus, for a better distinction, we refer to the impact sets computed by our rule-based approach as EIS_{rb_i} instead of just EIS_i. The actual impact sets (AIS_i) remain the same.

10.1.2.2. Metrics

In order to answer RQ2 we have to compute the global average precision and recall for both techniques, which can be accomplished by using the same formulas as for RQ1. The precision

(P_{rb}) and recall (R_{rb}) of our rule-based approach can be taken from RQ1 ($P_{rb} = P\varnothing, R_{rb} = R\varnothing$), while the precision and recall of the distance-based approach are determined as follows.

$$P_{db} = \frac{\sum_{i=1}^{n} \frac{|EIS_{db_i} \cap AIS_i|}{|EIS_{db_i}|}}{n} \qquad R_{db} = \frac{\sum_{i=1}^{n} \frac{|EIS_{db_i} \cap AIS_i|}{|AIS_i|}}{n}$$

10.1.2.3. Hypotheses

The null hypothesis H_0 regarding RQ2 is that the precision and recall of our rule-based approach is lower or equal compared to the precision and recall of the distance-based approach. Consequently, the alternative hypothesis H_a states that the precision and recall of our rule-based approach is greater than the recall and precision of the distance-based approach. The hypothesis is tested by *Welch's t-test* for two independent samples with different variance (results of rule-based approach vs. results of distance-based approach). The α-value remains at 0.05.

Variable	**Null hypothesis** H_0	**Alternative hypothesis** H_a
Precision	$P_{rb} \leq P_{db}$	$P_{rb} > P_{db}$
Recall	$R_{rb} \leq R_{db}$	$R_{rb} > R_{db}$

Table 10.3.: Tested hypotheses to answer RQ2

10.2. Study Design

In this section we outline how our evaluation was conducted. We describe the refactorings and changes that were applied on our case study subject, and how the oracle or golden standard for comparing the results was built. First, however, we introduce our case study subject.

10.2.1. Case Study Subject

In order to thoroughly evaluate our approach in a real world context we had to identify a suitable software system that could be utilized as a test system. This section outlines the search process for possible case study candidates and explains how and why we selected the final candidate. Our requirements for such a case study candidate were defined as follows:

- The project should be accompanied by heterogeneous software artifacts.
- The project should have evolved for several years (i.e. no academic toy projects).
- The project should be freely accessible, i.e. no closed-source software.
- The project should be in need of changes, e.g. the addition of new features.
- The project should provide support for implementing the changes in order to determine the oracle for the change impact analysis.

Thus, we carried out a search for open source projects that could be used as potential candidates for our case study. We analyzed open source hosting platforms like SourceForge[1] for suitable projects, as well as several websites of academic research projects, such as CoCoME (see Section 6.5.1). Finally, we identified 20 potentially suitable open source projects that were analyzed in detail based on our requirements. However, it turned out that the majority of the studied projects is only accompanied by source code artifacts, while no UML diagrams, test cases or other models are being provided. Secondly, we were not able to identify relevant change scenarios for all of the studied projects for various reasons, such as insufficient documentation or missing contact to the original contributors of the project. Thus, only artificial changes could have been applied. Thirdly, most academic projects did not qualify as candidates either, because they were too small in size. Moreover, for projects only providing source code it would have been possible to reverse engineer architectural diagrams; however, the resulting 1-1 mapping between architectural components and source code components would not have provided a realistic context for our study. Consequently, we selected our own research prototype EMFTrace which is developed in a joint effort of the Ilmenau University of Technology and the University of Hamburg as case study subject for the following reasons:

- The project provides source code, architectural diagrams, and a rich set of unit tests.
- The project was in need of various refactoring and maintenance measures.
- The project evolved for more than 4 years and was frequently changed by various developers (employees and students of both universities).
- The whole project is freely available under the Eclipse Public License [EPL].
- The author of this thesis possesses decent knowledge of the internals of EMFTrace since he has been involved in its development for more than 4 years and is thus able to plan, implement, and validate the necessary changes.

A summary of EMFTrace and its internal structure was already presented in Chapter 9 of this thesis. Furthermore, Section 6.5.1 and Table 6.6 in particular provide an overview of the available types of software artifacts and the complexity of EMFTrace. Links to the source code of EMFTrace before and after applying the changes can be found in Appendix C.1.

10.2.2. Evaluation Process

According to the underlying hypothesis of our impact analysis approach (see Section 4.1), we have to investigate how the interplay of change types, dependency types, and artifact types influences the propagation of changes. To accomplish this, we applied a series of refactoring activities and other change operations on our case study subject to address new requirements, fix remaining defects, etc. with the overall goal of evaluating our approach in a real world context. Before we elaborate on the change scenarios in the next section, this section briefly summarizes our evaluation process which consisted of the following seven steps:

1. Identifying the necessary changes to be applied on EMFTrace.
2. Designing the change operations and refactoring scenarios.
3. Constructing the oracle for each change, i.e. defining the AIS.

[1]http://www.sourceforge.net

4. Applying and testing the changes on EMFTrace to verify the AIS.
5. Executing our impact analysis approach to obtain the EIS.
6. Comparing the results of our approach against the oracle.
7. Assessing the gathered data according to our research goals.

10.2.3. Change Scenarios

The following sub-sections describe the intention and the structure of our changes. We are also listing the software artifacts and views that are addressed by them.

10.2.3.1. Scenario 1: Refactoring of the Impact Analyzer Components

Since being a research prototype, EMFTrace is subject to frequent and spontaneous changes in order to implement and test new hypothesis and research approaches. As a result of extending EMFTrace while conducting a preliminary case study on change impact analysis, several impact analysis algorithms were integrated into EMFTrace. Later on it turned out that this addition of multiple algorithms introduced cloned data and functionality to the system. Hence, a refactoring

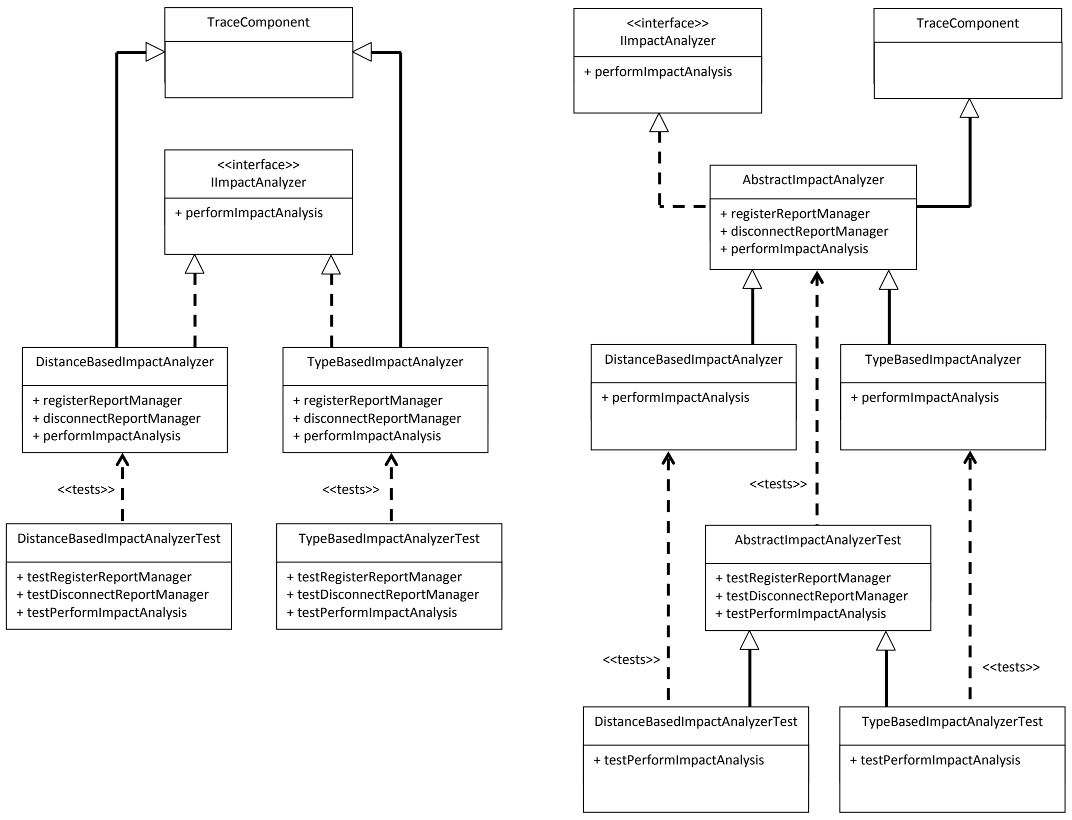

TraceComponent
<<interface>>
IImpactAnalyzer
+ performImpactAnalysis
DistanceBasedImpactAnalyzer
+ registerReportManager
+ disconnectReportManager
+ performImpactAnalysis
TypeBasedImpactAnalyzer
+ registerReportManager
+ disconnectReportManager
+ performImpactAnalysis
<<tests>>
<<tests>>
DistanceBasedImpactAnalyzerTest
+ testRegisterReportManager
+ testDisconnectReportManager
+ testPerformImpactAnalysis
TypeBasedImpactAnalyzerTest
+ testRegisterReportManager
+ testDisconnectReportManager
+ testPerformImpactAnalysis
<<interface>>
IImpactAnalyzer
+ performImpactAnalysis
TraceComponent
AbstractImpactAnalyzer
+ registerReportManager
+ disconnectReportManager
+ performImpactAnalysis
DistanceBasedImpactAnalyzer
+ performImpactAnalysis
TypeBasedImpactAnalyzer
+ performImpactAnalysis
<<tests>>
AbstractImpactAnalyzerTest
+ testRegisterReportManager
+ testDisconnectReportManager
+ testPerformImpactAnalysis
<<tests>>
<<tests>>
DistanceBasedImpactAnalyzerTest
+ testPerformImpactAnalysis
TypeBasedImpactAnalyzerTest
+ testPerformImpactAnalysis

Figure 10.1.: The class hierarchy of impact analyzers before (left) and after (right) the refactoring

of the impact analysis infrastructure was required in order to comply to general software quality standards and to improve the maintainability[2] and evolvability[3] of EMFTrace.

The general architecture of EMFTrace's impact analysis infrastructure before and after the refactoring is illustrated by Figure 10.1. The applied refactoring represents the classical case of extracting a common base class from a set of similar classes [Fow99]. We had to identify suitable attributes and methods to be moved to the common base class, redesign the class hierarchy accordingly, and update the remaining system thereafter.

This scenario is focused on the structural view and the behavioral view (see Figure 4.2) of EMFTrace and takes place on its architectural level. The concrete change operations of this scenario are modeled according to our approach presented in Chapter 7 and are listed in Appendix C.2.

10.2.3.2. Scenario 2: Extraction of a Cache Component

The *AccessLayer*-component of EMFTrace is responsible for encapsulating and providing access to the models stored in the EMFStore repository (see Section 9.3). During the initial development of EMFTrace a cache was added to the AccessLayer in order to speed up the processing of rules and other operations. However, due to the concept of "separation of concerns" [Dij82], it was decided to extract this cache and instead establish it as a distinct component of EMFTrace. Hence, all the data structures representing the model element cache and all the associated functions had to be moved to this new component, along with all the JUnit test cases. Likewise, every access to the cache in the remaining code of the AccessLayer had to be revised accordingly.

This scenario is focused on the structural view and the behavioral view (see Figure 4.2) of EMFTrace and takes place on its architectural level. The concrete change operations of this scenario are modeled according to our approach presented in Chapter 7 and are listed in Appendix C.3.

10.2.3.3. Scenario 3: Replacement of the Logging Features

During the initial development of EMFTrace in 2010 a simple logging mechanism was implemented to dump warnings and progress reports on the Eclipse console [Leh10]. In order to extend the logging capabilities of EMFTrace (e.g. directing the log output to a file), a replacement was sought. For this purpose, the JavaLogging API[4] was identified to be the most suitable candidate as it provided all the necessary functionality. Thus, the internal logging features of EMFTrace had to be replaced with calls to the JavaLogging API and its components.

This scenario is focused on the code view (see Figure 4.2) of EMFTrace and takes place on its implementation level. The concrete change operations of this scenario are modeled according to our approach presented in Chapter 7 and are listed in Appendix C.4.

[2]"the capability of the software product to be modified", see ISO 9126 [Int01]

[3]"ability of a software system throughout its lifespan to accommodate to changes and enhancements in requirements and technologies, that influence the system's architectural structure, with the least possible cost while maintaining the architectural integrity", see [Bod11]

[4]http://docs.oracle.com/javase/7/docs/technotes/guides/logging/overview.html

10.2.3.4. Scenario 4: Migration to EMFStore/ECP 1.2.x

As previously stated in Chapter 9, EMFTrace is build upon the EMFStore model repository and the Eclipse Client Platform (ECP) framework. Since EMFStore and ECP are subjects to frequent changes themselves, EMFTrace must adapt to these changes as well. Both EMFStore and ECP underwent a series of major revisions while evolving from version 0.9.3 to version 1.2.x and leaving the Eclipse Incubation phase to become fully-fledged Eclipse projects. These major revisions heavily affected the APIs of EMFStore and ECP as many interfaces and operations were either refactored, deleted, renamed or replaced. Furthermore, the data models shipped and supported by EMFStore/ECP were revised as well. Additionally, most of the previously provided ECP Eclipse Extension Points[5] were no longer available and were replaced by new extension points. As a consequence of these changes, EMFTrace did not compile with the new API out-of-the-box nor was it executable anymore. Thus, major migration work was needed to address *a)* the new API *b)* the new data models and *c)* the new Eclipse Extension Points.

The migration of EMFTrace's core components responsible for dependency detection, impact analysis, etc. required more than 60 change operations, while migrating the user interface required an even larger amount of work due to its tight coupling with former ECP extension points. The following describes some of the necessary changes of EMFTrace.

A change in EMFStore's data model rendered several operations and use case of EMFTrace obsolete. The previously available model element id's were moved to the internals of EMFStore and were no longer accessible from the outside. Hence, all operations that were based on those id's had to be revised or deleted, as the access to the internal API of EMFStore was either denied or strongly discouraged. Secondly, major changes of the API of EMFTrace were required due to the replacement of EMFStore *Projects* by *ECPProjects* which, unfortunately, are incompatible to each other. Consequently, the interface of EMFTrace's lowest architectural layer had to be revised (the *AccessLayer*, see Section 9.3). Together with the replacement of EMFStore *Projects* by *ECPProjects*, the addition and deletion of models to a project had to be adjusted accordingly as well (e.g. when creating traceability links during the rule-based dependency detection).

This scenario is focused on the code view (see Figure 4.2) of EMFTrace and takes place on its implementation level. The concrete change operations of this scenario are modeled according to our approach presented in Chapter 7 and are listed in Appendix C.5.

10.2.3.5. Scenario 5: Miscellaneous Changes

Although the above presented scenarios cover a wide range of typical change activities and software artifacts, further changes were necessary to thoroughly test our approach with heterogeneous types of software artifacts. Consequently, we applied a series of additional changes on EMFTrace to test combinations of change operations, dependencies, and artifacts that were not covered by the changes of the above presented scenarios, such as for example use case diagrams. To accomplish this, we tested any relevant combination of change type and artifact type that was not yet covered by the other scenarios. Hence, the remaining set of change operations $C(t)$ for each type of artifact t was determined as follows: $C(t) = t \times \{RelevantChangeTypes\}$.

The concrete change operations of this scenario are modeled according to our approach pre-

[5]http://wiki.eclipse.org/FAQ_What_are_extensions_and_extension_points%3F

sented in Chapter 7 and are listed in Appendix C.6.

10.2.4. Construction of the Oracle

The evaluation of the proposed impact analysis approach requires an oracle against which the obtained impact sets can be compared to. This oracle was constructed by manually analyzing the case study subject for the impacts of the aforementioned changes with the help of additional tool support, which was accomplished by a four-pronged approach.

First and foremost, the potential impacts of our changes were determined by manually analyzing the software artifacts prior to the implementation of the changes, which was accomplished by the author of this thesis. Secondly, we validated the identified impacts with the help of tools built into the Eclipse IDE, such as the Java refactoring browser, the Java compiler, and the execution of unit tests. Thirdly, we compared the modified version of EMFTrace (version $n + 1$) against the original project (version n) with the help of *diff*-tools to additionally validate the estimated impact sets based on the computed diffs. Finally, the modified version of EMFTrace was tested by three other software developers to ensure that the changes were implemented correctly, thus guaranteeing a high quality of the estimated oracle.

By applying the above mentioned strategy we were able to obtain the complete actual impact sets (AIS) for each of our change operations as discussed in the previous sections. We could furthermore assure that no impact was missed while constructing the oracle.

10.3. Results

This section presents the results of our case study in accordance to the initial goals of our evaluation. Therefore, we report on the precision and recall achieved by our approach in comparison to a) the oracle and b) the distance-based impact propagation approach. A critical discussion and analysis of these results then follows in Section 10.4.

Figures 10.2, 10.3, 10.4, 10.5, and 10.6 illustrate the resulting precision and recall for each change operation applied on our case study subject for both rule-based and distance-based impact propagation, where each figure summarizes the results of a specific scenario.

Subsequently, Tables 10.4 and 10.5 summarize the results of both approaches per scenario. They present the minimum and average precision and recall, their combined F_1-score, as well as the average sizes of the actual impact sets ($|AIS|$) and both estimated impact sets ($|EIS|$).

Table 10.6 illustrates the overall results of both approaches in terms of precision, recall, and F_1-score. Furthermore, it lists the standard deviation of precision and recall of both approaches. Moreover, Table 10.7 lists the t-values and the t-quantile of both approaches for a total of 210 conducted change operations.

Finally, Table 10.8 lists the average precision and recall per type of software artifact and the deviation of precision and recall per type of software artifact.

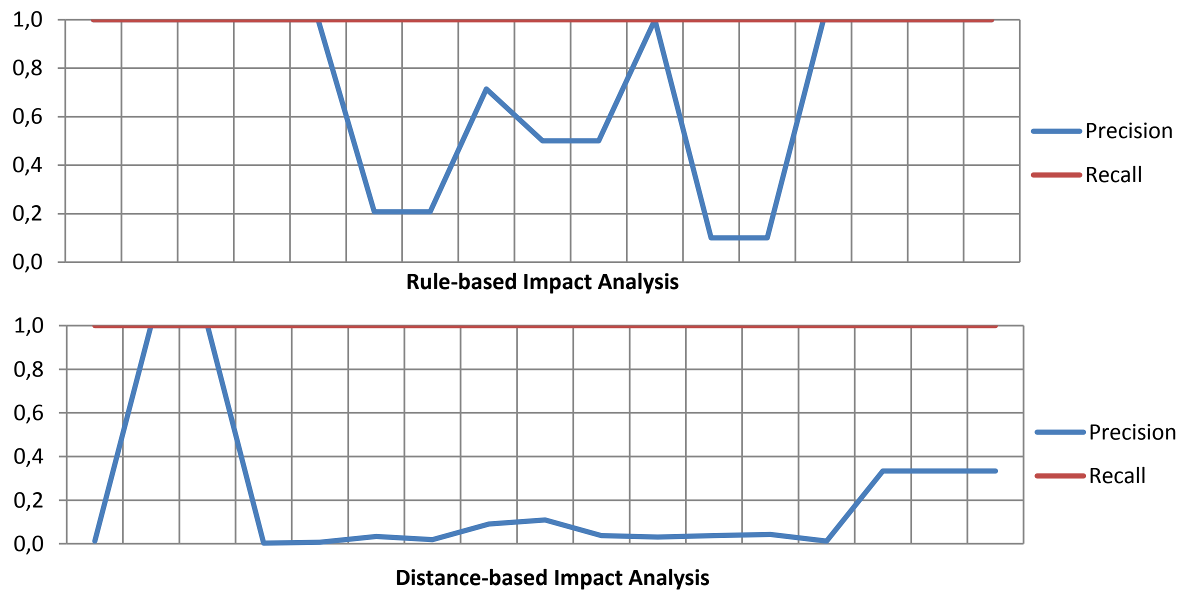


Figure 10.2.: Results of Scenario 1 (17 changes, see Appendix C.2)

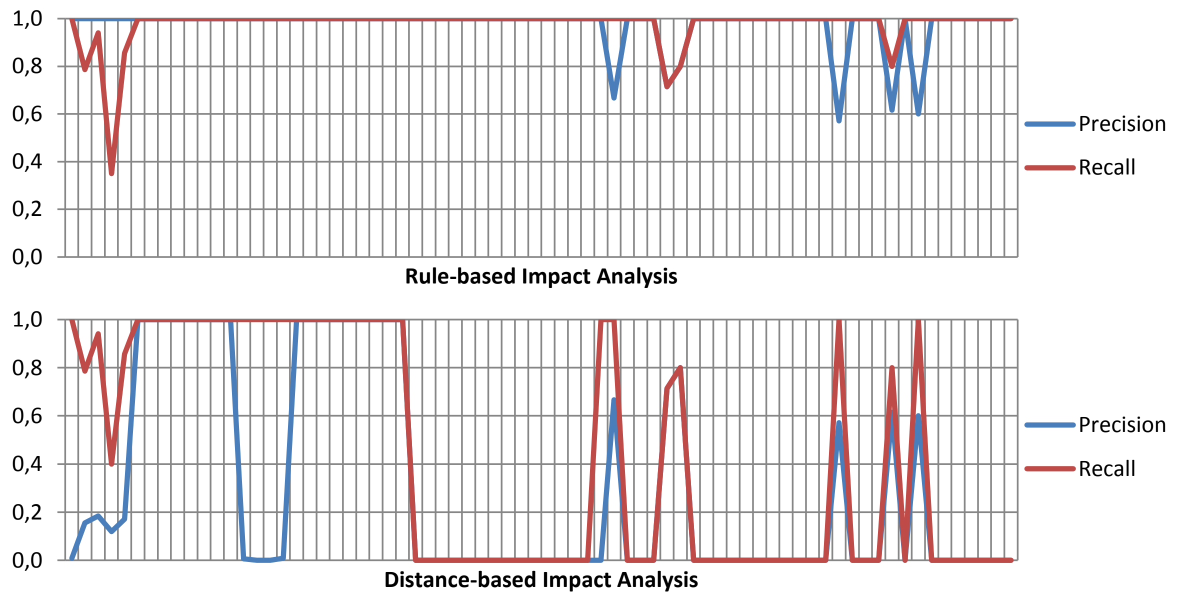


Figure 10.3.: Results of Scenario 2 (72 changes, see Appendix C.3)

Scenario	**Changes**	$\lvert AIS\varnothing \rvert$	$\lvert EIS\varnothing \rvert$	P_{min}	P∅	R_{min}	R∅	$F_1\varnothing$
S1	17	2	6	0.1000	0.7254	1.0000	1.0000	0.8408
S2	72	2	2	0.5714	0.9718	0.3500	0.9757	0.9737
S3	7	1	1	1.0000	1.0000	1.0000	1.0000	1.0000
S4	69	3	4	0.3333	0.8429	0.3333	0.9141	0.8737
S5	45	13	12	0.5560	0.9774	0.2940	0.9112	0.9431

Table 10.4.: Average results of our rule-based propagation approach per scenario

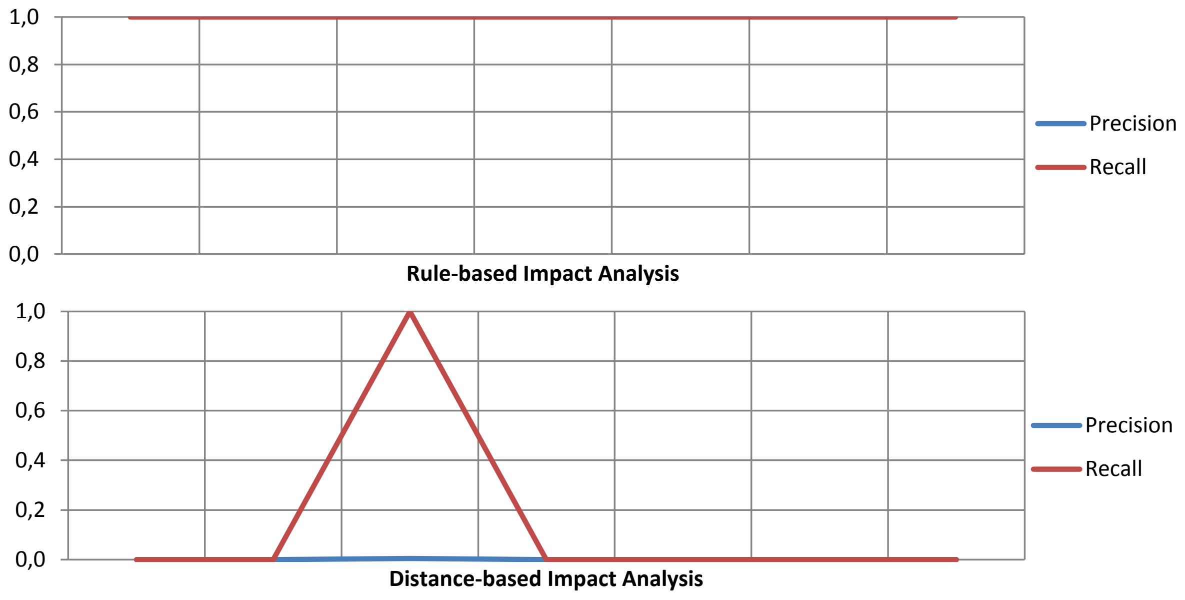


Figure 10.4.: Results of Scenario 3 (7 changes, see Appendix C.4)

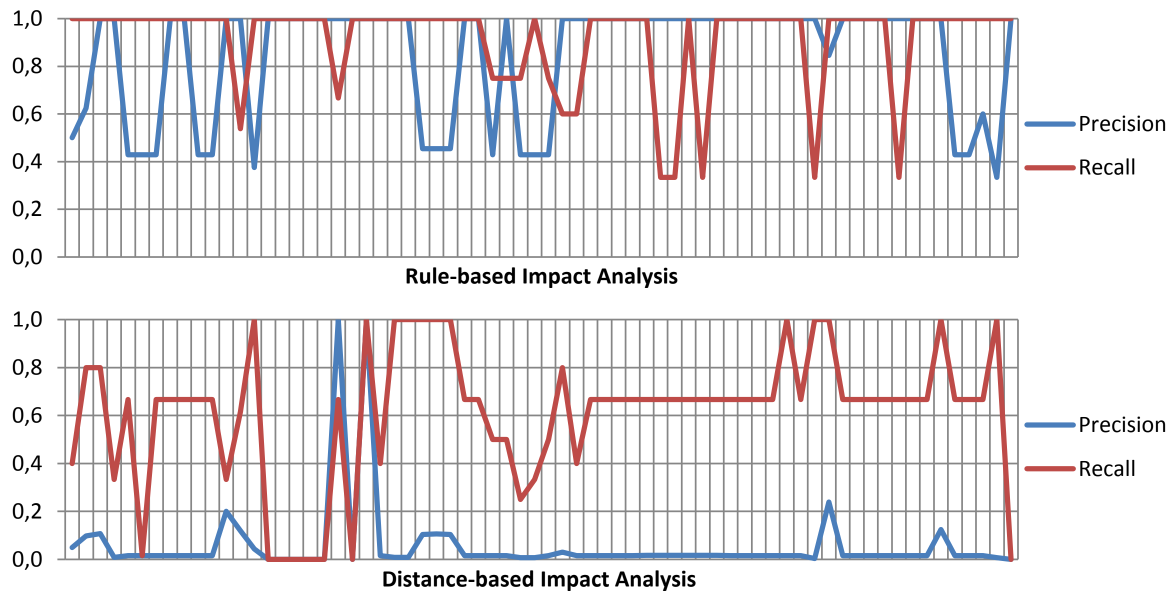


Figure 10.5.: Results of Scenario 4 (69 changes, see Appendix C.5)

Scenario	**Changes**	$\|AIS\varnothing\|$	$\|EIS\varnothing\|$	P_{min}	P∅	R_{min}	R∅	$F_1\varnothing$
S1	17	2	72	0.0037	0.2021	1.0000	1.0000	0.3363
S2	72	2	49	0.0000	0.3003	0.0000	0.4347	0.3552
S3	7	1	118	0.0000	0.0007	0.0000	0.1429	0.0014
S4	69	3	114	0.0000	0.0698	0.0000	0.6181	0.1255
S5	45	13	58	0.0045	0.1361	0.1111	0.6251	0.2235

Table 10.5.: Average results of the distance-based propagation approach per scenario

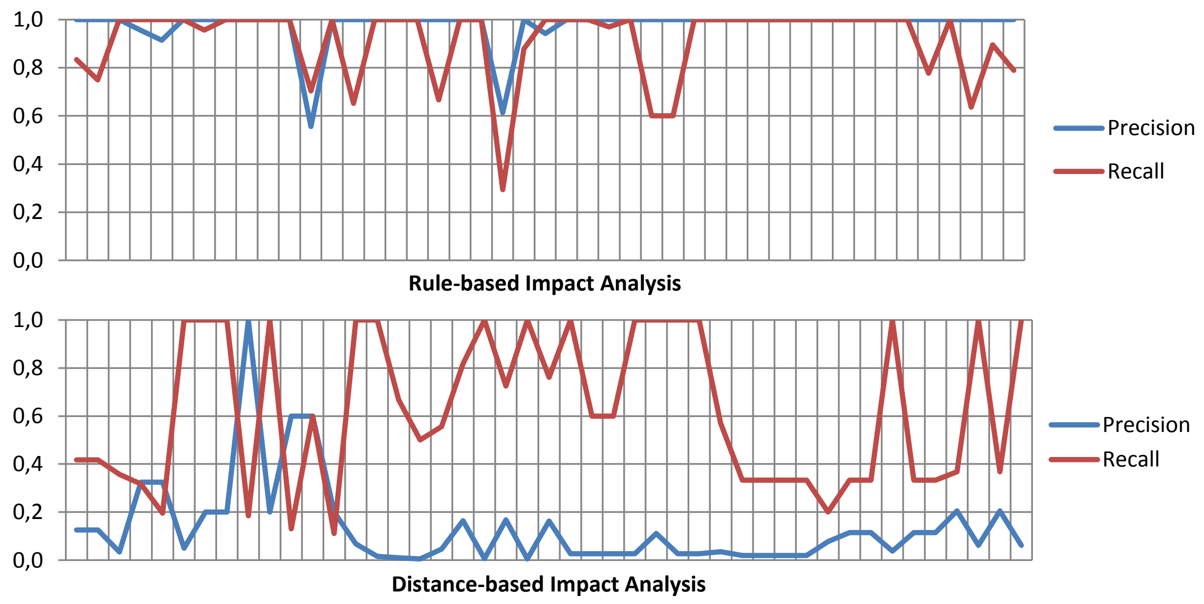


Figure 10.6.: Results of Scenario 5 (45 changes, see Appendix C.6)

Propagation Approach	P	R	F_1	sd_P	sd_R
Rule-based	0.9096	0.9396	0.9244	0.2033	0.1478
Distance-based	0.1714	0.5686	0.2634	0.3189	0.4020

Table 10.6.: Comparison of rule-based and distance-based propagation regarding precision, recall, F_1-score, and standard deviation of precision and recall

n	m	t_P	t_R	v_P	v_R	$t(1-\alpha, v)$
210	210	28.28	12.54	354	265	1.66

Table 10.7.: *t*-values for precision and recall, degrees of freedom, and *t*-quantile for $\alpha = 0.05$ for the comparison of rule-based and distance-based propagation. Each approach was tested with $n = m = 210$ changes. *Welch's t-test* for two independent samples is applied

Artifact Type	n	P_t	$d(P_t)$	t_P	R_t	$d(R_t)$	t_R	$t(1-\alpha, n-1)$
Component	6	0.9784	0.0688	-1.62	0.9306	0.0091	-2.34	2.01
Package	11	0.9596	0.0501	-2.49	0.9691	0.0295	-2.83	1.81
Class	18	0.8874	0.0222	-3.77	0.9162	0.0235	-3.75	1.73
Method	36	0.8664	0.0432	-4.72	0.9769	0.0373	-4.89	1.68
Attribute	19	0.8826	0.0270	-3.77	0.8728	0.0688	-2.92	1.73
Parameter	52	0.8522	0.0547	-5.17	0.8949	0.0448	-5.62	1.67
Use Case	4	1.0000	0.0940	-1.11	0.8303	0.1094	-0.92	2.35
Code Statement	64	0.9495	0.0399	-6.42	0.9685	0.0288	-6.86	1.67

Table 10.8.: Number of changes per type of software artifact (n), precision and recall per type of software artifact, deviation of precision and recall per type, *t*-values for precision and recall per type, and *t*-quantile for $\alpha = 0.05$. *Student's One-sample t-test* is applied

10.4. Discussion

The following discusses our results in regard to our two research questions as stated in Section 10.1.1 and Section 10.1.2 respectively.

10.4.1. RQ1: Support for Heterogeneous Software Artifacts

In order to answer our first research question we have to analyze the data listed in Table 10.8 that accumulates the results of our case study per type of software artifact. The precision achieved by our approach varies between 85% (method parameter) and 100% (use case), while the recall varies between 83% (use case) and 97% (method). Furthermore, the deviation of precision per type of artifact varies between 3% and 9%, which can be regarded as being overall stable. Likewise, the deviation of recall per type of artifact varies between 1% and 11%, which is slightly worse than similar figures for precision but still in an acceptable range.

The conducted change operations were distributed across different views (e.g. the structural view, behavioral view, test view, etc.) and covered the levels of software architecture (Scenario 1 & 2) and source code (Scenario 3 & 4). Moreover, the applied change operations represent typical real-world operations that occur in many software development projects. Overall, our approach was able to compute the impacts of all those changes with similar good results in regard to precision and recall (see Table 10.4). Therefore, our approach is applicable for change impact analysis of multiperspective software comprised of heterogeneous software artifacts.

To reject the null hypothesis regarding RQ1 with a probability of at least 95%, the following equations have to hold true for all types of software artifacts in order to prefer the alternative hypothesis (Student's one-sample one-sided t-test, see Table 10.2 in Section 10.1.1):

$$t_P < -t(1-\alpha, n-1) \qquad t_R < -t(1-\alpha, n-1)$$

For each type of software artifact addressed by our changes both equations hold, except for UML use cases and UML components (see Table 10.8). It turned out that the results that were achieved for those use cases and components are statistically not significant, as their t-values are greater than the corresponding t-quantile, thus violating the above stated equations. Consequently, both null hypothesis of RQ1 cannot be rejected and thus may or may not be true.

The reason for these results not being statistically significant is the comparably low number of change operations that were applied on use cases and components when compared to the amount of changes involving other types of software artifacts. For example, there are only 4 changes targeting use cases and 6 changes involving components, whereas there are 18 changes that were directly applied on classes and 52 changes involving method parameters.

However, for all other types of software artifacts our results are statistically significant (see t-values in Table 10.8). Moreover, there are no major deviations in between the results of our approach when applied on software artifacts of different type and granularity.

Another interesting aspect which, however, was not precisely measured during the course of our case study is the reduction of the manual effort that is required for the change impact analysis. It took several days to perform the manual impact analysis for each of the changes to be applied on either the source code or the architecture of our case study subject. In contrast, the application of our approach resulted in an effort of roughly five hours, which is caused by user interaction with our prototype tool (i.e. selecting the artifacts to be changed, select the changes to be applied, browsing the created impact reports, etc.). Yet, a more thorough analysis of potential effort savings is out of the scope of this thesis and may be conducted in future works.

10.4.2. RQ2: Performance Improvements

With the help of our previous research question we were able to underpin the applicability of our approach for multiperspective change impact analysis. Yet, we have to analyze how it compares to other approaches to allow for a final verdict on whether it fulfills our research goals or not.

By comparing the precision and recall of our rule-based approach against the distance-based impact analysis approach as illustrated by Figures 10.2 to 10.6, it becomes apparent that our rule-based approach achieves both better precision and recall and overall more stable results. The same conclusions can be drawn when analyzing the average precision and recall of both approaches for each of the five scenarios of our study (see Tables 10.4 and 10.5). The minimum average precision of our rule-based approach (72%, Scenario 1) is still more than two times better than the maximum average precision of the distance-based approach (30%, Scenario 2). Consequently, the overall average precision and recall of our approach are much better than the average precision (91% vs. 17%) and recall (94% vs. 57%) of the distance-based approach (see Table 10.6). Likewise, the standard deviation of precision and recall of our approach is lower than the one of the distance-based approach, which indicates more stable results (20% vs. 31% and 15% vs. 40% respectively).

The performance of the distance-based approach decreases significantly when the impacts of source code changes shall be analyzed (Scenario 3 and 4). In contrast, our approach performs equally well, independent of whether the changes are applied on the UML diagrams constituting the architecture of the system (Scenario 1 and 2) or on the source code of the system (Scenario 3 and 4). In general it can be stated that the more views are affected by a change, the more the performance of the distance-based approach declines. One could increase the recall of the distance-based approach by adjusting the applied propagation distance, however at the cost of further reducing the obtained precision. Hence, our rule-based concept is better suited for estimating the propagation of impacts in between different views and abstraction levels.

Another interesting aspect is the size of the computed impact sets (see Tables 10.4 and 10.5). While the average estimated impact set computed by our rule-based approach is of similar size as the actual impact set (i.e. the oracle), the estimated impact sets computed by the distance-based approach are between 4 and 118 times larger. When applied in a real-world context this means that developers have to face huge impact reports containing many false-positives. Thus, our approach performs much better than the distance-based approach in this regard as well.

To reject the null hypothesis regarding RQ2 with a probability of at least 95%, the following equations have to hold true for the precision and recall of our approach in comparison to the precision and recall of the distance-based approach in order to prefer the alternative hypothesis (Welch's t-test for two independent samples, see Table 10.3 in Section 10.1.2):

$$t_P > t(1-\alpha, v) \qquad t_R > t(1-\alpha, v)$$

As the *t*-values (28.28 and 12.54) and the corresponding *t*-quantile (1.66) in Table 10.7 indicate, both of the above stated equations hold true. Hence, we can reject both null hypothesis with a probability of at least 95% and instead prefer the alternative hypothesis, stating that our rule-based approach performs better than the distance-based approach. Welch's t-test had to be performed instead of Student's two-sample t-test because the variance of the distance-based results was much greater than the variance of the rule-based results, thus violating the requirements of Student's two-sample t-test that assumes equal variance among the samples.

In conclusion, we were able to show that our approach clearly outperforms distance-based impact analysis when applied on the architectural level and on the code level. Our approach computed less false-positives, missed fewer impacts, and provided overall more stable results.

10.5. Threats to Validity

Each scientific case study or experiment faces certain threats that might hamper the validity and generality of its results [RH09, KD11]. There are four general categories of such potential threats to validity [RH09] which we are discussing in the following. We also explain what was done in advance to lessen their effects on the validity of the presented evaluation.

10.5.1. Construct Validity

Do we measure what was intended according to our research goals? We could only assess the performance of our impact analysis approach by changing an existing software system and applying our approach to compute the impacts of those changes on the system. We determined the true impacts of the changes in advance in order to compare the output of the approach against them. The creation of this oracle or golden standard was accomplished by a four-pronged approach involving a combination of manual analysis and tool support. Based on the established oracle we measured the impacts of our changes according to the amount of impacted artifacts and applied the formulas for precision and recall as established in the literature. To answer RQ1 we determined the precision and recall for each type of software artifact that was provided by our case study subject. We then analyzed how the precision and recall per type of software artifact differed from the overall precision and recall to draw conclusions on the applicability of our approach for heterogeneous software artifacts. To answer RQ2 and to determine whether our approach is better suited than existing distance-based impact analysis algorithms, we applied both approaches to forecast the impacts of the exact same changes and then compared the obtained results. Hence, we assured that both approaches were applied on the exact same scenario under the exact same conditions.

10.5.2. Internal Validity

Are there unknown factors which might affect the causal dependencies? We have to analyze whether there exist yet unknown factors influencing the precision and recall of our approach in regard to the changes applied on our case study subject. Our approach depends on the classification of change operations and the classification of dependency relations between software artifacts according to their type. We supply a precise and easy-to-use approach for the modeling of changes that also builds the foundation of our change classification. As the studied changes were not of artificial character but instead applied on a real software system, it was assured that the operations were correctly modeled prior to the application of our approach. Likewise, the detection and classification of the dependency relations was accomplished prior to the application of our impact analysis approach. These results were furthermore evaluated and validated through preceding case studies, while excerpts of them were already published in peer-reviewed publications, e.g. [BLR11, LFR13, FLR14].

10.5.3. External Validity

To what extent it is possible to generalize the findings of our evaluation? Our change scenarios reflect typical real world problems which developers have to face every day. The applied changes are of varying complexity, cover a wide range of possible change operations, and were applied on a real system that is under ongoing development. Yet, it is still not entirely possible to draw final conclusions on our approach without conducting further studies involving different types of software systems. Likewise, for other systems where there are no such detailed UML diagrams other impact analysis approaches may suffice as well. However, the underlying trends should be similar as our approach addresses reoccurring dependency relations emerging from meta-models and the object oriented paradigm that are present in most of today's software.

10.5.4. Reliability

Are the results dependent on the researcher, methodologies, and the tools? A replication of our change scenarios should yield similar results if conducted by other researchers. Therefore, we provide other researchers with our prototype tool and detailed descriptions of our changes as listed in the appendix of this thesis. The conducted case study should be easy to replicate, since we defined the applied changes in a step-wise manner and carefully described our process of data collection and analysis. Some researchers, however, may propose a different oracle for certain changes due to alternate solutions preferred by them. Hence, they will obtain slightly different results. Yet, the overall precision and recall should remain the same. Moreover, the utilization of our own prototype tool as a case study candidate may raise certain doubts regarding the reliability of the study. However, our approach was not specifically tailored for applying it on our own tool as all the concepts and impact rules were developed in advance and are based on the research discussed in this thesis. Likewise, the changes that were applied on our prototype tool were not specifically related to our study either as they had to be implemented anyway due to the ongoing development of our prototype tool. Additionally, we already discussed the negative implications of utilizing 3rd party software as case study candidates in regard to the validity of the obtained oracle (see Section 10.2.1).

10.6. Summary

In this chapter we reported on the evaluation of our concepts with the help of a comprehensive case study. We applied a series of necessary changes and refactorings on a software system that evolved for more than four years and utilized our approach to forecast the impacts of those changes. Our concept of impact propagation rules achieved an average precision of 0.9096 and an average recall of 0.9396 during the application of 210 change operations on an existing software system that evolved for several years. The results of our study imply that our approach is a suitable option for conducting change impact analysis even for multiperspective software systems. We obtained similar results for different types of software artifacts, including elements of the software architecture, source code, and test cases. Moreover, our evaluation has shown that our approach performs much better than existing distance-based impact analysis approaches, whose results were less stable and the achieved precision and recall were overall much lower. Both approaches were tested with the exact same set of changes and software artifacts, which enables us to draw such a clear conclusion. Additionally, the conducted evaluation contributes towards **Goal 1** and **Goal 2** of this thesis as we were able to illustrate the applicability of the proposed change impact analysis approach. Our evaluation illustrates the feasibility of establishing a set of impact propagation rules that can be applied to assess the impacts of changes prior to their implementation.

11. Conclusions and Future Work

This chapter summarizes the contributions of this thesis, performs a final critical review of the presented work, and based on the current achievements discusses possible future research.

11.1. Contributions

This thesis presents a novel change impact analysis approach that can be applied in the context of software which is comprised of heterogeneous types of software artifacts. The presented approach supports developers performing maintenance and reengineering tasks that involve frequent changes of existing and potentially long-living software systems. It provides a forecast of the impacts of a change and supplies a potential solution for maintaining the consistency of the impacted software artifacts after applying the change. It furthermore allows developers to estimate the expected costs of a change, plan the implementation of the change, and decide between alternative solutions based on their impacts before the change is actually being implemented.

Rule-based Multiperspective Change Impact Analysis. The presented impact analysis approach is based on the observation that the effects of changes propagate across dependencies to related software artifacts. It was further observed that whether a dependency carries the effects of a change or not depends on the type of the change and the type of the dependency itself. The proposed approach analyzes this interplay of change types and dependency types using a set of impact propagation rules that are designed to react on certain combinations of changes and dependencies. The impact rules are triggered by changes and are able to determine how these changes affect related software artifacts, which is accomplished in a recursive manner. Each impact that is determined by a rule is feed back into the recursive impact analysis process where it may trigger the execution of further impact rules. In doing so, our approach is able to forecast the propagation of changes more reliably than existing techniques and across the different views on software. In the end, the set of impacted software artifacts is presented to the developer, along with the information how and why each artifact is impacted by the change.

Comprehensive Dependency Detection. Since our multiperspective impact analysis approach is based on the analysis of dependency relations connecting the heterogeneous software artifacts, these dependencies first of all have to be elicited and explicitly recorded. Yet, current research does not provide a comprehensive concept for multiperspective dependency detection. Hence, we extended a rule-based detection approach that was initially developed in one of our previous works for detecting dependencies of UML models. This enhanced approach utilizes a set of dependency detection rules to elicit and record potential dependency relations as traceability links. Our rules are furthermore able to determine the types of the dependencies, which is also accomplished during the detection process. Moreover, our approach analyzes dependencies stemming from four different sources and is capable of exploiting structural and textual properties of software artifacts, as well as existing relations between them.

Classification of Dependency Relations. Once all the dependencies of a software system have been elicited, it is necessary to determine their types to enable a later impact analysis. We propose to classify the dependencies according to their purpose to provide the rationale of the relations. To achieve this, we introduced a novel taxonomy for dependency types that is based on a step-wise refinement of the purposes of dependencies using several abstraction levels.

Modeling of Change Operations. Our approach supports the precise modeling of the change operations that act as triggers for the actual change impact analysis. We extend existing works on change modeling by introducing an enhanced concept that is based on atomic and composite operations. The proposed atomic changes constitute the basic units of change while the composite operations are comprised of other atomic or composite changes. We demonstrated the applicability of this concept by modeling all the change operations that were applied during the course of our case study according to it. Moreover, the precise modeling of change operations also allows us to model the exact types of the impacts that are determined by our impact rules.

Prototype Implementation and Evaluation. We provide additional tool support for the approach presented in this thesis. Our prototype tool EMFTrace implements and supports all four steps of our approach and currently allows for change impact analysis of UML models, Java source code, and JUnit test cases. With the help of our prototype we conducted an initial evaluation of our approach [LFR13] and the comprehensive evaluation reported in this thesis. We compared our rule-based change impact analysis approach against distance-based change impact analysis and were able to show that our approach achieves both better precision (90%) and recall (93%) when estimating the impacts of changes. Thus, developers have to face less false-positives and are enabled to better understand the implications of their changes.

11.2. Critical Review

While there is no doubt about the usefulness of change impact analysis support for software maintenance and reengineering, the following critically discusses the proposed approach in regard to its underlying assumptions, the correctness of the established impact propagation rules, and the soundness of the conducted evaluation.

Assumptions of the Approach. In Section 4.3.3 we described the assumptions that are inherent to our approach. In the following we are going to discuss them in regard to their implications on the applicability of our approach for real software development projects.

Our approach is designed for software that is developed using object oriented technologies and concepts, and in regard to the current state of the art in software engineering we believe that this assumption holds because the vast majority of today's software is build upon these concepts. Nevertheless, our approach can as well be tailored for software systems that are developed using other paradigms, such as procedural legacy systems for instance.

Our decision of utilizing UML for the modeling of software architectures may limit the applicability of our current approach when other ADLs are used for this task. However, UML and many of its offshoots, such as SysML for example, are the nowadays de facto industry standard for the modeling of complex systems. Consequently, our current approach is applicable for them. Moreover, our approach can be extended to encompass further modeling languages and programming languages as already discussed in sections 5.2.1, 6.4.4.3, and 8.3.2 of this thesis.

The major assumption of the research presented in this thesis is that the different software artifacts must be consistent to each other prior to any change impact analysis activity. In practice, however, the consistency of all the involved software artifacts (source code, architecture, etc.) might not be taken as granted for all software development projects. Yet, any change impact analysis approach requires a solid and consistent baseline for estimating the propagation of changes as otherwise inconsistencies may result in missed impacts and false-positives. Unfortunately, this cannot be tackled conceptually, except by emphasizing the need for change impact analysis right from the beginning of software development to obviate potential inconsistencies.

The applicability of our approach might be further constrained by its demand for explicitly recorded dependencies. This, however, also applies to the vast majority of the existing change impact analysis approaches as discussed in Section 2.3. To mitigate this limitation we supply a comprehensive approach for multiperspective dependency detection.

Reliability and Completeness of the Analyzed Dependency Relations. Since our approach is based on the analysis of explicitly recorded dependency relations, their quality and completeness also influences the quality and completeness of the impact sets computed by our approach. To ensure a high quality of the detected relations, we defined a scheme for creating dependency detection rules (Section 6.4.4.3), provide developers with a taxonomy of dependency types to classify the detected dependencies (Section 6.3), and discussed the detection of dependencies stemming from four different origins (Section 6.4.2). Additionally, we analyzed the potential threats to our dependency detection rules in Section 6.6 and illustrated how they can be resolved. Related work on dependency detection was thoroughly compared to our approach in Section 6.6.1 and the benefits of our detection approach were outlined. Moreover, the results of several case studies indicate that our dependency detection approach provides better recall and precision than existing works (see Section 6.5.2). However, even the results of five case studies do not allow for a final verdict on our approach as a theoretical discussion of the correctness and completeness of our dependency detection rules is not feasible.

Reliability and Completeness of our Impact Propagation Rules. The success of the proposed change impact analysis approach depends on four factors: the quality and completeness of the dependency relations as discussed above and the quality and completeness of our impact propagation rules. The correctness and completeness of our impact propagation rules determine to which extent only correct impacts are reported and whether all impacted artifacts can be identified. We already discussed the potential threats to the completeness and correctness of our impact rules and outlined means to limit their influence on the impact analysis (see Section 8.4.1). However, it still remains a manual task of creating, validating, and maintaining the impact propagation rules, even though these steps are supported by our scheme for developing impact rules (see Section 8.3.2). On the other hand, the results of an initial case study [LFR13] and the evaluation reported in this thesis already indicate a high quality of our impact rules.

Soundness of our Evaluation. The evaluation was conducted with a real software system under study and a set of real changes. In contrast to most existing works, the changes were actually implemented to realize a series of refactorings and the addition of new features. We compared our approach against manual impact assessments and against distance-based impact analysis to obtain convincing results. Our approach achieved similar good results for both precision and recall, which were furthermore much more stable than those of the distance-based approach. However, one may not generalize the findings of our evaluation without first conducting further case studies involving different types of systems and systems of varying complexity.

11.3. Future Work

Based on the presented change impact analysis approach as the main contribution of this thesis there are several potential directions for further research to extend the proposed concepts.

Semi-automated Implementation of Changes. A possible extension of our approach can be achieved by integrating the concept of (semi)-automated change implementation into our impact analysis approach as discussed in Section 8.4.6. Since the proposed impact propagation rules are able to determine the exact types of impacts, it is possible to extend the impact rules and the rule execution infrastructure with means for adjusting the impacted software artifacts on the fly. To accomplish this, the rules have to be extended with "repair plans" [KPP08] that are applied after the impact has been determined. Although not all types of impacts can be resolved automatically (e.g. complex refactorings), the vast majority of changes that occur in real software development projects could be addressed in an automated manner (e.g. renaming methods, changing data types, etc.). Similar support for simple refactoring activities of Java source code is already provided by the Eclipse IDE, which therefore could be complemented with our approach. The (semi)-automated implementation of changes would further reduce the costs of changes as a lot of manual work for actually realizing the changes could be saved.

Combining Traceability Maintenance and Impact Analysis. In a similar fashion the impact analysis approach could be extended to update and modify the analyzed dependency relations according to the results of the impact analysis (see Section 8.4.2.3). This extension could be based upon the work of Mäder [MÖ9] who already updates traceability relations according to changes applied by developers. At the moment, our approach requires the re-detection of dependencies after each change that was applied on the software system. This overhead could be reduced if the dependencies would automatically be updated during the change impact analysis process. Hence, a combination of Mäder's approach and ours would help to further reduce the costs of changes by decreasing the costs of the dependency detection that is required for conducting the change impact analysis and related tasks, such as regression testing.

Expanding the Impact Analysis. Furthermore, the proposed impact analysis approach could be extended to encompass additional software artifacts and views, such as requirements descriptions or (BPMN) process models for instance. Such an extension first of all requires the identification and detection of potential dependencies between the artifacts to be integrated and those artifacts that are already addressed by our current approach. Likewise, it requires the creation of additional impact propagation rules to address the new types of software artifacts.

Integration into Development Environments. Finally, the direct integration of our impact analysis approach into existing IDEs like Eclipse could help to increase its applicability by making additional tools like EMFTrace obsolete and streamlining the actual impact analysis process. This in turn demands for implementing a change listening mechanism that transforms IDE-internal change events into our representation of changes that is based on atomic and composite operations, which could then automatically trigger the change impact analysis. Similar ideas are already proposed in the work of Robbes and Lanza [RL08], which therefore could be reused and incorporated into our approach. Moreover, the results of an experiment conducted in an industrial setting by Goeritzer [Goe11] emphasize the need for a direct incorporation of change impact analysis features into the tools and processes applied by software developers.

Bibliography

[AB93] Robert S. Arnold and Shawn A. Bohner. Impact analysis - towards a framework for comparison. In *Proceedings of the IEEE Conference on Software Maintenance (CSM '93)*, pages 292–301, Montreal, Quebec, Canada, September 1993.

[ACC$^+$02] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *IEEE Trans. Softw. Eng.*, 28(10):970–983, Oct 2002.

[ACCDL00] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, and Andrea De Lucia. Identifying the starting impact set of a maintenance request: A case study. In *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering*, pages 227–230, Zurich, Switzerland, February 2000.

[ACPT01] Guiliano Antoniol, B. Caprile, A. Potrich, and Paolo Tonella. Design-code traceability recovery: selecting the basic linkage properties. *Science of Computer Programming*, 40:213–234, 2001.

[ALS09a] M.K. Abdi, H. Lounis, and H. Sahraoui. Predicting change impact in object-oriented applications with bayesian networks. In *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC '09)*, pages 234–239, Seattle, WA, USA, July 2009.

[ALS09b] M.K. Abdi, H. Lounis, and H. Sahraoui. A probabilistic approach for change impact prediction in object-oriented systems. In *Proceedings of the 2nd Workshop on Artificial Intelligence Techniques in Software Engineering (AISEW 2009)*, pages 189–200, Thessaloniki, Greece, April 2009.

[ANS08] Aharon Abadi, Mordechai Nisenson, and Yahalomit Simionovici. A traceability technique for specifications. In *Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC 2008)*, pages 103–112, Amsterdam, Netherlands, June 2008.

[AOH05] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pages 432–441, St. Louis, Missouri, USA, May 2005.

[ARNRSG06] Netta Aizenbud-Reshef, B. T. Nolan, Julia Rubin, and Yael Shaham-Gafni. Model traceability. *IBM Systems Jounal*, 45(3):515–526, July 2006.

[ARPR$^+$05] Netta Aizenbud-Reshef, Richard F. Paige, Julia Rubin, Yael Shaham-Gafni, and Dimitrios S. Kolovos. Operational semantics for traceability. In *Proceedings of the 1st ECMDA Workshop on Traceability*, pages 7–14, Nürnberg, Germany, 2005.

[ATL] Atlas Transformation Language (ATL). http://eclipse.org/atl/. (Accessed on November, 20th 2014).

[BA96] Shawn A. Bohner and Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Publications Tutorial Series, Los Alamitos, California, USA, 1996.

[BBE$^+$95] S. Barros, T. Bodhuin, A. Escudie, J.P. Queille, and J.F. Voidrot. Supporting impact analysis: a semi-automated technique and associated tool. In *Proceedings of the 11th International Conference on Software Maintenance (ICSM'95)*, pages 42–51, Opio (Nice), France, October 1995.

[BC00] Carliss Y. Baldwin and Kim B. Clark. *Design Rules: The Power of Modularity*. MIT Press, Cambridge, Massachusetts, USA, 2000.

[BCR94] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. The goal question metric approach. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 528–532. John Wiley & Sons, 1994.

[Ben90] Keith H. Bennett. An introduction to software maintenance. *Information and Software Technology*, 12(4):257–264, 1990.

[BFV00] Alessandro Bianchi, Anna Rita Fasolino, and Giuseppe Visaggio. An exploratory case study of the maintenance effectiveness of traceability models. In *Proceedings of 8th International Workshop on Program Comprehension (IWPC'00)*, pages 149 – 158. IEEE, 2000.

[BGA06] Salah Bouktif, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Extracting change-patterns from CVS repositories. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE '06)*, pages 221–230, Benevento, October 2006.

[BGW13] Markus Borg, Orlena Gotel, and Krzysztof Wnuk. Enabling traceability reuse for impact analyses - a feasibility study in a safety context. In *Proceedings of the 7th International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE 2013)*, pages 72–78, San Francisco, California, USA, May 2013.

[BJV04] Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. On the need for megamodels. In *Proceedings of the OOPSLA and GPCE Workshop*, Vancouver, Canada, 2004.

[BL10] David Binkley and Dawn Lawrie. Information retrieval applications in software maintenance and evolution. In P. Laplante, editor, *Encyclopedia of Software Engineering*, chapter 2. Taylor & Francis LLC, 2010.

[BLBS02] Lionel C. Briand, Yvan Labiche, K. Buist, and G. Soccar. Automating impact analysis and regression test selection based on UML designs. In *Proceedings of the 18th IEEE International Conference on Software Maintenance (ICSM'02)*, pages 252–261, Montreal, Quebec, Canada, October 2002.

[BLO03] Lionel C. Briand, Yvan Labiche, and L. O'Sullivan. Impact analysis and change management of UML models. In *Proceedings of the 19th International Conference on Software Maintenance*, pages 256–265, Amsterdam, Netherlands, September 2003.

[BLOS06] L. Briand, Yvan Labiche, L. O'Sullivan, and Michal Sówka. Automated impact analysis of UML models. *Journal of Systems and Software*, 79:339–352, 2006.

[BLR11] Stephan Bode, Steffen Lehnert, and Matthias Riebisch. Comprehensive model integration for dependency identification with EMFTrace. In *Joint Proceedings of the First International Workshop on Model-Driven Software Migration (MDSM 2011) and the Fifth International Workshop on Software Quality and Maintainability (SQM 2011)*, pages 17–20, Oldenburg, Germany, March 2011. CEUR.

[BMNT05] Jorge Biolchi, Paul Gomes Mian, Ana Candida Cruz Natali, and Guilherme Horta Travassos. Systematic review in software engineering. Technical Report RT - ES 679/05, Systems Engineering and Computer Science Department, University of Rio de Janeiro, May 2005.

[BMZ$^+$05] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. Towards a taxonomy of software change. *Journal of Software Maintenance and Evolution: Research and Practice*, 17:309–332, September 2005.

[Bod11] Stephan Bode. *Quality goal oriented architectural design and traceability for evolvable software systems*. PhD thesis, Ilmenau University of Technology, Ilmenau, Germany, April 2011.

[Boh95] Shawn A. Bohner. *A graph traceability approach for software change impact analysis*. PhD thesis, George Mason University, Fairfax, Virginia, USA, 1995.

[Boh96] Shawn A. Bohner. Impact analysis in the software change process: a year 2000 perspective. In *Proceedings of the 12th International Conference on Software Maintenance (ICSM'96)*, pages 42–51, Monterey, California, USA, November 1996.

[Boh02a] Shawn A. Bohner. Extending software change impact analysis into COTS components. In *Proceedings of the 27th Annual NASA Goddard/IEEE Software Engineering Workshop*, pages 175–182, Greenbelt, Maryland, USA, December 2002.

[Boh02b] Shawn A. Bohner. Software change impacts - an evolving perspective. In *Proceedings of the 18th International Conference on Software Maintenance*, pages 263–272, Montreal, Quebec, Canada, October 2002.

[Boo94] Grady Booch. *Object Oriented Analysis and Design With Applications*. Addison-Wesley Longman, Amsterdam, 2nd edition, October 1994.

[BSF03] Marko Boger, Thorsten Sturm, and Per Fragemann. Refactoring browser for UML. *Lecture Notes in Computer Science*, 2591:366–377, 2003.

[BTP05] Ben Breech, Mike Tegtmeyer, and Lori Pollock. A comparison of online and dynamic impact analysis algorithms. In *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, pages 143–152, Manchester, United Kingdom, March 2005.

[CC92] Elliot J. Chikofsky and James H. Cross. *Software Reengineering*, chapter Reverse engineering and design recovery: A taxonomy, pages 54–58. IEEE Computer Society Press, 1992.

[CCCDP10a] Gerardo Canfora, Michele Ceccarelli, Luigi Cerulo, and Massimiliano Di Penta. Using multivariate time series and association rules to detect logical change coupling: an empirical study. In *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM 2010)*, pages 1–10, Timisoara, Romania, September 2010.

[CCCDP10b] Michele Ceccarelli, Luigi Cerulo, Gerardo Canfora, and Massimiliano Di Penta. An eclectic approach for change impact analysis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 163–166, Cape Town, South Africa, May 2010.

[CDO] CDO Model Repository. http://eclipse.org/cdo/. (Accessed on November, 20th 2014).

[CDPC11] Gerardo Canfora, Massimiliano Di Penta, and Luigi Cerulo. Achievements and challenges in software reverse engineering. *Commun. ACM*, 54(4):142–151, April 2011.

[CE00] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addision Wesley, Boston, 2000.

[CJH01] Stephen Cook, He Ji, and Rachel Harrison. Dynamic and static views of software evolution. In *Proceedings of the 17th IEEE International Conference on Software Maintenance (ICSM'01)*, pages 592–601, Los Alamitos, California, USA, November 2001. IEEE Computer Society.

[CJMV95] Panos Constantopoulos, Matthias Jarke, John Mylopoulos, and Yannis Vassiliou. The software information base: a server for reuse. *The International Journal on Very Large Data Bases*, 4(1):1–43, 1995.

[CKKL99] M. Ajmal Chaumun, Hind Kabaili, Rudolf K. Keller, and Francois Lustman. A change impact model for changeability assessment in object-oriented software systems. In *Proceedings of the Third European Conference on Software Maintenance and Reengineering*, pages 130–149, Amsterdam, Netherlands, March 1999.

[CoC] Common Component Modelling Example (CoCoME). http://cocome.org/index.htm. (Accessed on November, 20th 2014).

[Cod13] Code Analyzer. http://sourceforge.net/projects/codeanalyze-gpl/, April 2013. (Accessed on November, 20th 2014).

[COE] Center of Excellence for Software Traceability (COEST). http://coest.org/. (Accessed on November, 20th 2014).

[CSL$^+$01] Pär Carlshamre, Kristian Sandahl, Mikael Lindvall, Björn Regnell, and Johan Natt och Dag. An industrial survey of requirements interdependencies in software product release planning. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*, pages 84–91, Toronto, Ontario, Canada, August 2001.

[CT94] William B. Cavnar and John M. Trenkle. N-gram-based text comparison. In *Proceedings of the 3rd Annual Symposium on document Analysis and Information Retrieval (SDAIR-94)*, pages 161–175, 1994.

[CT99] Bruno Caprile and Paolo Tonella. Nomen est omen: Analyzing the language of function identifiers. In *Proceedings of the 6th Working Conference on Software Reverse Engineering*, pages 112–122, Atlanta, Georgia, USA, October 1999.

[CY91] Peter Coad and Edward Yourdon. *Object-Oriented Analysis*. Prentice-Hall, Inc., 2nd edition, 1991.

[DDN08] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Square Bracket Associates, Kehrsatz, Switzerland, June 2008.

[Dij82] Edsger W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A personal Perspective*, Texts and Monographs in Computer Science, pages 60–66. Springer New York, 1982.

[DKPF09] Nikolaos Drivalos, Dimitrios S. Kolovos, Richard F. Paige, and Kiran J. Fernandes. Engineering a DSL for software traceability. In Dragan Gašević, Ralf Lämmel, and Eric Van Wyk, editors, *Software Language Engineering*, volume 5452 of *Lecture Notes in Computer Science*, pages 151–167. Springer Berlin Heidelberg, 2009.

[DLFO08] Andrea De Lucia, Fausto Fasano, and Rocco Oliveto. Traceability management for impact analysis. In *Proceedings of Frontiers of Software Maintenance (FoSM 2008)*, pages 21–30, Beijing, China, October 2008.

[DLHE11] Andreas Demuth, Roberto E. Lopez-Herrejon, and Alexander Egyed. Cross-layer modeler - a tool for flexible multilevel modeling with consistency checking. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 452–455, Szeged, Hungary, September 2011.

[DnC05] Juan C. Dueñas and Rafael Capilla. The decision view of software architecture. In Ron Morrison and Flavio Oquendo, editors, *Software Architecture*, volume 3527 of *Lecture Notes in Computer Science*, pages 222–230. Springer Berlin Heidelberg, June 2005.

[DP06] Florian Deissenboeck and Markus Pizka. Concise and consistent naming. *Software Quality Journal*, 14(3):261–282, 2006.

[DPFK08] Nicholas Drivalos, Richard F. Paige, Kiran J. Fernandes, and Dimitrios S. Kolovos. Towards rigorously defined model-to-model traceability. In *Proceedings of the 4th ECMDA Traceability Workshop (ECMDA-TW)*, pages 17–26, Berlin, Germany, June 2008.

[DST11] Robert Dabrowski, Krzysztof Stencel, and Grzegorz Timoszuk. Software is a directed multigraph. In Ivica Crnkovic, Volker Gruhn, and Matthias Book, editors, *Software Architecture*, volume 6903 of *Lecture Notes in Computer Science*, pages 360–369. Springer Berlin Heidelberg, 2011.

[EAG06] Angelina Espinoza, Pedro P. Alarcón, and Juan Garbajosa. Analyzing and systematizing current traceability schemas. In *Proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop SEW-30 (SEW'06)*, pages 21–32, Columbia, Maryland, USA, April 2006. IEEE.

[Ecl10] Eclipse Model Repository. http://modelrepository.sourceforge.net, April 2010. (Accessed on November, 20th 2014).

[ECP] Eclipse Client Platform (ECP). http://www.eclipse.org/ecp. (Accessed on November, 20th 2014).

[EG11] Angelina Espinoza and Juan Garbajosa. A study to support agile methods more effectively through traceability. *Innovations in Systems and Software Engineering*, 7(1):53–69, 2011.

[EGA08] Hamid El Ghazi and Said Assar. A multi view based traceability management method. In *Proceedings of the 2nd International Conference on Research Challenges in Information Science (RCIS '08)*, pages 393–400, Marrakech, Morocco, June 2008.

[EHKG02] Gregor Engels, Reiko Heckel, Jochen M. Küster, and Luuk Groenewegen. Consistency-preserving model evolution through transformations. In Jean-Marc Jézéquel, Heinrich Hussmann, and Stephen Cook, editors, *UML 2002 - The Unified Modeling Language*, volume 2460 of *Lecture Notes in Computer Science*, pages 212–226. Springer Berlin Heidelberg, 2002.

[EMFa] Eclipse Modeling Framework (EMF). http://www.eclipse.org/modeling/emf/. (Accessed on November, 20th 2014).

[EMFb] EMF Query. http://projects.eclipse.org/projects/modeling.emf.query. (Accessed on November, 20th 2014).

[EMFc] EMFStore. http://www.eclipse.org/emfstore/. (Accessed on November, 20th 2014).

[EMF14] EMFTrace. https://sourceforge.net/projects/emftrace/, June 2014. (Accessed on November, 20th 2014).

[EMP] Eclipse Modeling Project. http://www.eclipse.org/modeling/. (Accessed on November, 20th 2014).

[EPL] Eclipse Public License v1.0. http://www.eclipse.org/legal/epl-v10.html. (Accessed on November, 20th 2014).

[EPRV08] R. Eramo, A. Pierantonio, J. R. Romero, and A. Vallecillo. Change management in multi-viewpoint system using ASP. In *Proceedings of the 5th International Workshop on ODP for Enterprise Computing (EDOC 2008)*, pages 19–28, Munich, Germany, September 2008.

[FG06] Beat Fluri and Harald C. Gall. Classifying change types for qualifying change couplings. In *Proceeding of the 14th IEEE International Conference on Program Comprehension (ICPC 2006)*, pages 35–45, Athens, Greece, June 2006.

[FGP05] Beat Fluri, Harald C. Gall, and Martin Pinzger. Fine-grained analysis of change couplings. In *Proceeding of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation 2005*, pages 66–74, November 2005.

[FLR14] Qurat-Ul-Ann Farooq, Steffen Lehnert, and Matthias Riebisch. Analyzing model dependencies for rule-based regression test selection. In *Proceedings of Modellierung 2014*, pages 305–320, Vienna, Austria, March 2014.

[FM06] Tie Feng and Jonathan I. Maletic. Applying dynamic change impact analysis in component-based architecture design. In *Proceeding of the Seventh International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 2006)*, pages 43–48, Las Vegas, Nevada, USA, June 2006.

[FMP99] Pascal Fradet, Daniel Métayer, and Michaël Périn. Consistency checking for multiple view software architectures. In Oscar Nierstrasz and Michel Lemoine, editors, *Software Engineering - ESEC/FSE 1999*, volume 1687 of *Lecture Notes in Computer Science*, pages 410–428. Springer Berlin / Heidelberg, 1999.

[FN05] Jean-Marie Favre and Tam NGuyen. Towards a megamodel to model software evolution through transformations. *Electronic Note*, 127(3):59–74, 2005.

[Fow99] Martin Fowler. *Refactoring: Improving the design of existing code*. Addison Wesley, Longman, Inc., Amsterdam, 1999.

[Fur86] George W. Furnas. Generalized fisheye views. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 16–23, 1986.

[FZS03] Gilberto A. A. Cysneiros Filho, Andrea Zisman, and George Spanoudakis. Traceability approach for i* and UML models. In *Proceedings of the 2nd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS'03)*, Portland, Oregon, USA, May 2003.

[GC08] Yaser Ghanam and Sheelagh Carpendale. A survey paper on software architecture visualization. Technical report, University of Calgary, Canada, 2008.

[GDKP12] Malcom Gethers, Bogdan Dit, Huzefa Kagdi, and Denys Poshyvanyk. Integrated impact analysis for managing software changes. In *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)*, pages 430–440, Zurich, Switzerland, June 2012.

[GDL04] Tudor Gîrba, Stéphane Ducasse, and Michele Lanza. Yesterday's Weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proceeding of the 20th IEEE International Conference on Software Maintenance (ICSM '04)*, pages 40–49, Chicago, Illinois, USA, September 2004. IEEE Computer Society.

[GF94] Orlena C. Z. Gotel and Anthony C. W. Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of the First International Conference on Requirements Engineering*, pages 94–101, Colorado Springs, Colorado, USA, April 1994. IEEE Computer Society Press.

[GL91] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991.

[GLR14] Sebastian Gerdes, Steffen Lehnert, and Matthias Riebisch. Combining architectural design decisions and legacy system evolution. In *Proceedings of the 8th European Conference on Software Architectures (ECSA2014)*, pages 50–57, Vienna, Austria, August 2014.

[GMP07] Mark Grechanik, Kathryn S. McKinley, and Dewayne E. Perry. Recovering and using use-case-diagram-to-source-code traceability links. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC-FSE '07)*, pages 95–104, Cavtat, Croatia, September 3-7 2007.

[Goe11] Robert Goeritzer. Using impact analysis in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 1155–1157, Honoulu, Hawaii, USA, May 2011.

[GP10] Malcom Gethers and Denys Poshyvanyk. Using relational topic models to capture coupling among classes in object-oriented software systems. In *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM'10)*, pages 1–10, Timisoara, Romania, September 2010.

[GS82] Chris P. Gane and Trish Sarson. Structured system analysis. Technical report, McDonnell Douglas, 1982.

[GSC09] Chetna Gupta, Yogesh Singh, and Durg Singh Chauhan. An efficient dynamic impact analysis using definition and usage information. *International Journal of Digital Content Technology and its Applications*, 3(4):112–115, 2009.

[GSC10] Chetna Gupta, Yogesh Singh, and Durg Singh Chauhan. A dynamic approach to estimate change impact using type of change propagation. *Journal of Information Processing Systems*, 6(4):597–608, December 2010.

[Gup13] Nikhilumar Gupta. Rule-based Dependency Detection Between Source Code and UML Models. Master's thesis, Ilmenau University of Technology, Ilmenau, Germany, February 2013.

[Han96] Jun Han. Supporting impact analysis and change propagation in software engineering environments. Technical Report 96-09, Monash University, Peninsula School of Computing & Information Technology, McMahons Road, Frankston, Victoria 3199, Australia, October 1996.

[HBG$^+$11] Salima Hassaine, Ferdaous Boughanmi, Yann-Gaël Guéhéneuc, Sylvie Hamel, and Giuliano Antoniol. A seismology-inspired approach to study change propagation. In *Proceedings of the 27th International Conference on Software Maintenance (ICSM 2011)*, pages 53–62, Williamsburg, Virginia, USA, September 2011.

[HCM09] Maen Hammad, Michael L. Collard, and Jonathan I. Maletic. Automatically identifying changes that impact code-to-design traceability. In *Proceedings of the IEEE 17th International Conference on Program Comprehension (ICPC '09)*, pages 20–29, Vancouver, British Columbia, Canada, May 2009.

[HDB10] Mohamed Oussama Hassan, Laurent Deruelle, and Henri Basson. A knowledge-based system for change impact analysis on software architecture. In *Proceedings of the Fourth International Conference on Research Challenges in Information Science (RCIS)*, pages 545–556, Nice, France, May 2010.

[HH04] Ahmed E. Hassan and Richard C. Holt. Predicting change propagation in software systems. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM 2004)*, pages 284–293, Washington, DC, USA, September 2004. IEEE Computer Society.

[Hig02] Dan Higgins. *AI Game Programming Wisdom*, volume 1, chapter Generic A* Pathfinding, pages 114–121. Charles River Media, 2002.

[HNR68] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[HNS05] Christine Hofmeister, Robert Nord, and Dilip Soni. Global analysis: moving from software requirements specification to structural views of the software architecture. *IEE Proceedings - Software*, 152(4):187–197, August 2005.

[HS06] Lulu Huang and Yeong-Tae Song. Dynamic impact analysis using execution profile tracing. In *Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications (SERA'06)*, pages 237–244, Seattle, Washington, USA, August 2006.

[HS07] Lulu Huang and Yeong-Tae Song. Precise dynamic impact analysis with dependency analysis for object-oriented programs. In *Proceedings of the 5th ACIS International Conference on Software Engineering Research, Management & Applications (SERA 2007)*, pages 374–384, Busan, South Korea, August 2007.

[HS08] Lulu Huang and Yeong-Tae Song. A dynamic impact analysis approach for object-oriented programs. In *Proceedings of the Conference on Advanced Software Engineering and Its Applications (ASEA '08)*, pages 217–220, Hainan Island, December 2008.

[III08] Salma Imtiaz, Naveed Ikram, and Saima Imtiaz. Impact analysis from multiple perspectives: Evaluation of traceability techniques. In *Proceedings of the 3rd International Conference on Software Engineering Advances*, pages 457–464, Sliema, Malta, October 2008.

[IIMD05a] Suhaimi Ibrahim, Norbik Bashah Idris, Malcolm Munro, and Aziz Deraman. Integrating software traceability for change impact analysis. *The International Arab Journal of Information Technology*, 2(4):301–308, October 2005.

[IIMD05b] Suhaimi Ibrahim, Norbik Bashah Idris, Malcolm Munro, and Aziz Deraman. A requirements traceability to support change impact analysis. *Asean Journal of Information Technology*, 4(4):345–355, 2005.

[IIMD06] Suhaimi Ibrahim, Norbik Bashah Idris, Malcolm Munro, and Aziz Deraman. A software traceability validation for change impact analysis of object oriented software. In *Proceedings of the International Conference on Software Engineering Research and Practice & Conference on Programming Languages and Compilers, SERP 2006*, volume 1, pages 453–459, Las Vegas, Nevada, USA, June 2006.

[IK06] Igor Ivkovic and Kostas Kontogiannis. Towards automated establishment of model dependencies using formal concept analysis. *International Journal of Software Engineering and Knowledge Engineering*, 16(4):499–522, Aug 2006.

[Ind] Indus Java Program Slicer. http://indus.projects.cis.ksu.edu/index.shtml. (Accessed on August, 21th 2014).

[Ins98a] Institute of Electrical and Electronics Engineers. IEEE Recommended Practice for Software Requirements Specifications, 1998.

[Ins98b] Institute of Electrical and Electronics Engineers. IEEE Standard for Software Maintenance. IEEE Std 1219-1998, Oct 1998.

[Int01] International Standardization Organisation. ISO/IEC 9126-1 International Standard. Software Engineering – Product quality – Part 1: Quality models, June 2001.

[Int06] International Standardization Organisation. Software Engineering—Software Life Cycle Processes—Maintenance. ISO/IEC 14764:2006, IEEE Std 14764-2006, 2006.

[ISO01] ISO/IEC. *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001.

[ITU08] ITU-T. Recommendation ITU-T Z.151 User requirements notation (URN) – Language definition, November 2008.

[JSL13] Khaled Jaber, Bonita Sharif, and Chang Liu. A study on the effect of traceability links in software maintenance. *Access, IEEE*, 1:726–741, 2013.

[JZ09] Waraporn Jirapanthong and Andrea Zisman. Xtraque: traceability for product line systems. *Software and Systems Modeling*, 8(1):117–144, 2009.

[Kag07] Huzefa Kagdi. Improving change prediction with fine-grained source code mining. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 559–562, New York, NY, USA, 2007.

[Kag08] Huzefa Kagdi. *Mining Software Repositories to support software evolution*. PhD thesis, Kent State University, Kent, Ohio, USA, August 2008.

[KD11] Anne Keller and Serge Demeyer. *Change Impact Analysis for UML Model Maintenance*, chapter 2, pages 32–56. IGI Global, 2011.

[KGGR08] Safoora Shakil Khan, Phil Greenwood, Alessandro Garcia, and Awais Rashid. On the interplay of requirements dependencies and architecture evolution: An exploratory study. In *Proceedings of the 20th International Conference Advanced Information Systems Engineering (CAiSE '08)*, pages 243–257, Montpellier, France, June 2008.

[KGHW94] D. Kung, J. Gao, P. Hsia, and F. Wen. Change impact identification in object oriented software maintenance. In *International Conference on Software Maintenance*, pages 202–211, Victoria, British Columbia, Canada, September 1994.

[KGPC10] Huzefa Kagdi, Malcom Gethers, Denys Poshyvanyk, and Michael L. Collard. Blending conceptual and evolutionary couplings to support change impact analysis in source code. In *Proceedings of the 17th IEEE Working Conference on Reverse Engineering (WCRE'10)*, pages 119–128, Beverly, Massachusetts, USA, October 2010.

[KH05] Gerald Kotonya and John Hutchinson. Analysing the impact of change in

COTS-based systems. *Lecture Notes in Computer Science*, 3412:212–222, 2005.

[Kil08] Malia Sofia Kilpinen. *The Emergence of Change at the Systems Engineering and Software Design Interface - An Investigation of Impact Analysis*. PhD thesis, Cambridge University, Engineering Department, Cambridge, United Kingdom, August 2008.

[KK07] Jaakko Korpi and Jussi Koskinen. Supporting impact analysis by program dependence graph based forward slicing. In Khaled Elleithy, editor, *Advances and Innovations in Systems, Computing Sciences and Software Engineering*, pages 197–202. Springer Netherlands, 2007.

[KKK10] Tae-hyung Kim, Kimun Kim, and Woomok Kim. An interactive change impact analysis based on an architectural reflexion model approach. In *Proceedings of the IEEE 34th Annual Computer Software and Applications Conference (COMPSAC '10)*, pages 297–302, Seoul, South Korea, July 2010.

[KL09] Safoora Shakil Khan and Simon Lock. Concern tracing and change impact analysis: An exploratory study. In *Proceedings of the 2009 ICSE Workshop on Aspect-Oriented Requirements Engineering and Architecture Design*, pages 44–48, Vancouver, British Columbia, Canada, May 2009.

[KM07a] Huzefa Kagdi and Jonathan I. Maletic. Combining single-version and evolutionary dependencies for software-change prediction. In *Proceedings of 4th International Workshop on Mining Software Repositories (MSR'07)*, pages 107–110, Minneapolis, Minnesota, USA, May 2007.

[KM07b] Huzefa Kagdi and Jonathan I. Maletic. Software repositories: A source for traceability links. In *Proceedings of the 4th ACM International Workshop on Traceability in Emerging Forms of Software Engineering (GCT/TEFSE'07)*, pages 32–39, Lexington, KY, USA, March 2007.

[KMS07] Huzefa Kagdi, Jonathan I. Maletic, and Bonita Sharif. Mining software repositories for traceability links. In *Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC '07)*, pages 145–154, Banff, Alberta, British Columbia, Canada, June 2007.

[Kol09] Dimitrios S. Kolovos. Establishing correspondences between models with the epsilon comparison language. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, ECMDA-FA '09, pages 146–157, Berlin, Heidelberg, 2009. Springer-Verlag.

[Kos03] Rainer Koschke. Software visualization in software maintenance, reverse engineering, and reengineering: a research survey. *Journal of Software Maintenance and Evolution: Research and Practice*, 15:87–109, 2003.

[KPP08] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. Detecting and repairing inconsistencies across heterogeneous models. In *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation*, pages 356–364, Lillehammer, Norway, April 2008.

[Kru95] P. B. Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, 12(6):42–

50, 1995.

[KSD09] Anne Keller, Hans Schippers, and Serge Demeyer. Supporting inconsistency resolution through predictive change impact analysis. In *Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation*, Denver, Colorado, USA, October 2009.

[Leh80] M.M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, Sept. 1980.

[Leh10] Steffen Lehnert. Softwarearchitectural Design and Realization of a Repository for Comprehensive Model Traceability (in German: Softwarearchitektur-Entwurf und Realisierung eines Repositories für Modell-übergreifende Traceability). Diploma thesis, Ilmenau University of Technology, Ilmenau, Germany, November 2010.

[Leh11a] Steffen Lehnert. A review of software change impact analysis. Technical report, Ilmenau University of Technology, Department of Software Systems / Process Informatics, Ilmenau, Germany, December 2011.

[Leh11b] Steffen Lehnert. A taxonomy for software change impact analysis. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution (IWPSE-EVOL 2011)*, pages 41–50, Szeged, Hungary, September 2011. ACM.

[Let02] Patricio Letelier. A framework for requirements traceability in UML-based projects. In *Proceedings 1st International Workshop on Traceability in Emerging Forms of SE (TEFSE'02)*, pages 32–41, Edinburgh, United Kingdom, 2002. ACM.

[LFOT07] Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(4):13–62, September 2007.

[LFR12] Steffen Lehnert, Qurat-Ul-Ann Farooq, and Matthias Riebisch. A taxonomy of change types and its application in software evolution. In *Proceedings of the 19th Annual IEEE International Conference on the Engineering of Computer Based Systems*, pages 98–107, Novi Sad, Serbia, April 2012.

[LFR13] Steffen Lehnert, Qurat-Ul-Ann Farooq, and Matthias Riebisch. Rule-based impact analysis for heterogeneous software artifacts. In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR2013)*, pages 209–218, Genova, Italy, March 2013.

[LK99] Simon Lock and Gerald Kotonya. An integrated, probabilistic framework for requirement change impact analysis. *Australasian Journal of Information Systems*, 6(2):38–63, September 1999.

[LM12] Yang Li and Walid Maalej. Which traceability visualization is suitable in this context? a comparative study. *Lecture Notes in Computer Science*, 7195:194–210, 2012.

[LO96] Li Li and A. Jefferson Offutt. Algorithmic analysis of the impact of changes on object-oriented software. In *Proceedings of the International Conference on Software Maintenance*, pages 171–184, Monterey, California , USA, November 1996.

[LR03] James Law and Gregg Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the 25th International Conference on Software Engineering (2003)*, pages 308–318, Portland, Oregon, USA, May 2003.

[LR12] Steffen Lehnert and Matthias Riebisch. Tackling the challenges of evolution in multiperspective software design and implementation. *Softwaretechnik Trends*, 32(2):27–28, May 2012.

[LSLZ13] Bixin Li, Xiaobing Sun, Hareton Leung, and Sai Zhang. A survey of code-based change impact analysis techniques. *Software Testing, Verification and Reliability*, 23(8):613–646, December 2013.

[LT93] Nancy G. Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.

[MÖ9] Patrick Mäder. *Rule-Based Maintenance of Post-Requirements Traceability*. PhD thesis, TU Ilmenau, Ilmenau, Germany, 2009.

[MÏ0] Patrick Mäder. *Rule-Based Maintenance of Post-Requirements Traceability*. MV-Verlag, Münster, 2010.

[Mat02] James Matthews. *AI Game Programming Wisdom*, volume 1, chapter Basic A* Pathfinding Made Simple, pages 105–113. Charles River Media, 2002.

[MBC05] J. Muskens, R. J. Bril, and M. R. V. Chaudron. Generalizing consistency checking between software views. In *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA 2005)*, pages 169–180, Pittsburgh, Pennsylvania, USA, November 2005.

[MBZR03] Tom Mens, Jim Buckley, Matthias Zenger, and Awais Rashid. Towards a taxonomy of software evolution. In *Proceedings of the 2nd International Workshop on Unanticipated Software Evolution*, pages 1–18, Warsaw, Poland, April 2003.

[Mey96] Bertrand Meyer. The many faces of inheritance: A taxonomy of taxonomy. *IEEE Computer*, 29(5):105–108, May 1996.

[MG09] Sharon McGee and Des Greer. A software requirements change source taxonomy. In *Proceedings of the Fourth International Conference on Software Engineering Advances (ICSEA '09)*, pages 51–58, Porto, Portugal, September 2009.

[MG11] Sharon McGee and De Greer. Software requirements change taxonomy: Evaluation by case study. In *Proceedings of the 19th IEEE International Requirements Engineering Conference (RE 2011)*, pages 25–34, Trento, Italy, September 2011.

[MGP08] Patrick Mäder, Orlena Gotel, and Ilka Philippow. Rule-based maintenance of post-requirements traceability relations. In *Proceedings of the 2008 16th IEEE International Requirements Engineering Conference (RE '08)*, pages 23–32, Washington, DC, USA, 2008. IEEE Computer Society.

[MM03] Andrian Marcus and Jonathan I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *International Conference on Software Engineering (ICSE'03)*, pages 125–135, Los Alamitos, California, USA, May 2003. IEEE.

[MNS01] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4):364–380, April 2001.

[Mod] MoDisco. http://www.eclipse.org/MoDisco/. (Accessed on November, 20th 2014).

[Mot12] Daniel Motschmann. Multikriterielle Suche in einem Eclipse-basierten Repository für Software-Architekten. Diplomarbeit, Ilmenau University of Technology, Ilmenau, Germany, March 2012.

[MPR07] Patrick Mäder, Ilka Philippow, and Matthias Riebisch. A traceability link model for the unified process. In *Proceedings of the 8th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, pages 700–705, Qingdao, China, July 2007.

[MR14] Klaus Müller and Bernhard Rumpe. A model-based approach to impact analysis using model differencing. In *Proceedings of the 8th International Workshop on Software Quality and Maintainability (SQM2014)*, Antwerp, Belgium, February 2014.

[MRP06a] Patrick Mäder, Matthias Riebisch, and Ilka Philippow. Maintaining traceability links during evolutionary software development. *Softwaretechnik Trends*, 26(3):89–90, May 2006.

[MRP06b] Patrick Mäder, Matthias Riebisch, and Ilka Philippow. Traceability for managing evolutionary change. In *Proceedings of the 15th International Conference on Software Engineering and Data Engineering (SEDE-2006)*, pages 1–8, Los Angeles, California, USA, July 2006.

[MvdHW06] Leonardo G. P. Murta, André van der Hoek, and Cláudia M. L. Werner. Archtrace: Policy-based support for managing evolving architecture-to-implementation traceability links. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 135–144, Tokyo, Japan, September 2006.

[NHM10] Sarah Nadi, Ric Holt, and Serge Mankovskii. Does the past say it all? using history to predict change sets in a CMDB. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering*, pages 97–106, Madrid, Spain, March 2010.

[OAH03] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. Leveraging field data for impact analysis and regression testing. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE'03)*, pages 128–137, Helsinki, Finland, 2003.

[OAL$^+$04] Alessandro Orso, Taweesup Apiwattanapong, James Law, Gregg Rothermel, and Mary Jean Harrold. An empirical comparison of dynamic impact analysis algorithms. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, pages 491–500, Edinburgh, Scotland, 2004.

[OG02] Thomas Olsson and John Grundy. Supporting traceability and inconsistency management between software artifacts. In *Proceedings of the IASTED International Conference on Software Engineering and Applications*, pages 63–78, 2002.

[OGPDL10] Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. On the equivalence of information retrieval methods for automated traceability link recovery. In *Proceedings of the IEEE 18th International Conference on Program Comprehension (ICPC 2010)*, pages 68–71, Braga, Portugal, June 2010.

[OMG11] OMG. Requirements Interchange Format (ReqIF), April 2011.

[OMG12] OMG. Object Constraint Language (OCL). ISO/IEC 19507, April 2012.

[OMG13] OMG. Business Process Model and Notation (BPMN) Version 2.0.1. ISO/IEC 19510:2013, July 2013.

[OMG14] OMG. MetaObject Facility. ISO/IEC 19508, April 2014.

[PDK$^+$11] Richard F. Paige, Nikolaos Drivalos, Dimitrios S. Kolovos, Kiran J. Fernandes, Christopher Power, Goran K. Olsen, and Steffen Zschaler. Rigorous identification and encoding of trace-links in model-driven engineering. *Software and Systems Modeling*, 10(4):469–487, 2011.

[PGBM10] Daniel Popescu, Joshua Garcia, Kevin Bierhoff, and Nenad Medvidovic. Helios: Impact analysis for event-based systems. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 531–532, Cape Town, South Africa, May 2010.

[PMFG09] Denys Poshyvanyk, Andrian Marcus, Rudolf Ferenc, and Tibor Gyimóthy. Using information retrieval based coupling measures for impact analysis. *Empirical Software Engineering*, 14(1):5–32, 2009.

[Poh96a] Klaus Pohl. PRO-ART: Enabling requirements Pre-traceability. In *Proceedings of the Second International Conference on Requirements Engineering, ICRE*, pages 76–84. IEEE Computer Society, Apr 1996.

[Poh96b] Klaus Pohl. *Process-Centered Requirements Engineering*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[POK$^+$08] Richard F. Paige, Goran K. Olsen, Dimitrios S. Kolovos, Steffen Zschaler, and Christopher Power. Building model-driven engineering traceability classifications. In *Proceedings of the ECMDA Traceability Workshop*, pages 49–58, 2008.

[Pop10] Daniel Popescu. Impact analysis for event-based components and systems. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 401–404, Cape Town, South Africa, May 2010.

[QVWM94] Jean-Pierre Queille, Jean-Francois Voidrot, Norman Wilde, and Malcom Munro. The impact analysis task in software maintenance: A model and a case study. In *Proceedings of the International Conference on Software Main-*

tenance, pages 234–242, Victoria, British Columbia, Canada, September 1994.

[Raj97] Václav Rajlich. A model for change propagation based on graph rewriting. In *Proceedings of the 13th International Conference on Software Maintenance (ICSM '97)*, pages 84–91, Bari, Italy, October 1997.

[RBFL11] Matthias Riebisch, Stephan Bode, Qurat-Ul-Ann Farooq, and Steffens Lehnert. Towards comprehensive modelling by inter-model links using an integrating repository. In *Proceedings of the 8th IEEE Workshop on Model-Based Development for Computer-Based Systems - Covering Domain and Design Knowledge in Models*, pages 284–291, Las Vegas, Nevada, USA, April 2011.

[Ren07] Xiaoxia Ren. *Change Impact Analysis for Java programs and applications*. PhD thesis, New Brunswick Graduate School, Rutgers University, New Brunswick, New Jersey, USA, October 2007.

[RH09] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Journal of Empirical Software Engineering*, 14:131–164, 2009.

[RJ01] Balasubramaniam Ramesh and Matthias Jarke. Toward reference models for requirements traceability. *IEEE Trans. Softw. Eng.*, 27(1):58–93, 2001.

[RKRS05] T. Reiter, E. Kapsammer, W. Retschitzegger, and W. Schwinger. Model integration through mega operations. In *Proceedings of the Workshop on Model-driven Web Engineering*, pages 20–29, Sydney, Australia, July 2005.

[RL08] Romain Robbes and Michele Lanza. SpyWare: A change-aware development toolset. In *Proceedings of the 30th international conference on Software engineering (ICSE '08)*, pages 847–850, Leipzig, Germany, May 2008.

[Rob08] Romain Robbes. *Of Change and Software*. PhD thesis, Faculty of Informatics of the University of Lugano, Lugano, Switzerland, December 2008.

[RPB11] Matthias Riebisch, Alexander Pacholik, and Stephan Bode. Towards optimization of design decisions for embedded systems by exploiting dependency relationships. In *Proceedings of the Workshop Modellbasierte Entwicklung eingebetteter Systeme (MBEES2011)*, Dagstuhl, Germany, February 2011. Dagstuhl.

[RRST05] Xiaoxia Ren, Barbara G. Ryder, Maximilian Störzer, and Frank Tip. Chianti: A change impact analysis tool for Java programs. In *Proceedings of the 27th international conference on Software Engineering (ICSE '05)*, pages 664–665, New York, NY, USA, 2005. ACM.

[RSN09] Mehwish Riaz, Muhammad Sulayman, and Husnain Naqvi. Architectural decay during continuous software evolution and impact of 'design for change' on software architectures. In Dominik Slezak, Tai-hoon Kim, Akingbehin Kiumi, Tao Jiang, June Verner, and Silvia Abrahao, editors, *Advances in Software Engineering*, volume 59 of *Communications in Computer and Information Science*, pages 119–126. Springer, 2009.

[RST$^+$03] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, Ophelia Chesley, and Julian Dolby. Chianti: A prototype change impact analysis tool for Java. Technical Report DCS-TR-533, Rutgers University, Department of Computer Science, New Brunswick, New Jersey, USA, September 2003.

[RST$^+$04] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: A tool for change impact analysis of Java programs. In *Proceedings of the 19th annual ACM SIG-PLAN Conference on Object-oriented programming, systems, languages, and applications (OOPSLA '04)*, pages 432–448, Vancouver, British Columbia, Canada, October 2004.

[RT01] Barbara G. Ryder and Frank Tip. Change impact analysis for object-oriented programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE '01)*, pages 46–53, Snowbird, Utah, USA, June 2001.

[RWKA07] Siti Rochimah, Wan M.N. Wan Kadir, and Abdul H. Abdullah. An evaluation of traceability approaches to support software evolution. In *Proceedings of the 2nd International Conference on Advances in Software Engineering*, pages 19–27, Cap Esterel, France, August 2007.

[SA03] Susanne A. Sherba and Kenneth M. Anderson. A framework for managing traceability relationships between requirements and architectures. In *Proceedings of the International Conference on Software Engineering*, pages 150–156, Portland, Oregon, May 2003.

[SdGZ03] George Spanoudakis, A. d'Avila Garces, and Andrea Zisman. Revising rules to capture requirements traceability relations: A machine learning approach. In *Proceedings of the 15th International Conference in Software Engineering and Knowledge Engineering (SEKE 2003)*, pages 570–577. Knowledge Systems Institute, Skokie, 2003.

[SEW09] Hannes Schwarz, Jürgen Ebert, and Andreas Winter. Graph-based traceability: a comprehensive approach. *Software and Systems Modeling*, 9(4):473–492, 2009.

[SF01] Thanwadee Sunetnanta and Anthony Finkelstein. Automated consistency checking for multiperspective software specifications. In *Proceedings of the International Conference on Software Engineering Workshop on Advanced Separation of Concerns*, pages 1–12, Toronto, Ontario, Canada, May 2001.

[SH10] Raul Santelices and Mary Jean Harrold. Probabilistic slicing for predictive impact analysis. Technical report, Georgia Tech Center for Experimental Research in Computer Systems (CERCS), Atlanta, Georgia, USA, 2010.

[SLT$^+$10] Xiaobing Sun, Bixin Li, Chuanqi Tao, Wanzhi Wen, and Sai Zhang. Change impact analysis based on a taxonomy of change types. In *Proceedings of the IEEE 34th Annual Computer Software and Applications Conference*, pages 373–382, Seoul, South Korea, July 2010.

[SLTZ11] Xiaobing Sun, Bixin Li, Chuanqi Tao, and Sai Zhang. HSM-based change impact analysis of object-oriented Java programs. *Chinese Journal of Electronics*, 20(2):247–251, April 2011.

[SNG10] Andreas Seibel, Stefan Neumann, and Holger Giese. Dynamic hierarchical mega models: comprehensive traceability and its efficient maintenance. *Software and Systems Modeling*, 9(4):493–528, 2010.

[SPLTJ01] Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel. Refactoring UML models. *Lecture Notes in Computer Science*, 2185:134–148, 2001.

[SRRT06] Maximilian Störzer, Barbara G. Ryder, Xiaoxia Ren, and Frank Tip. Finding failure-inducing changes in Java programs using change classification. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 57–68, Portland, Oregon, USA, 2006.

[ST07] Ali R. Sharafat and Ladan Tahvildari. A probabilistic approach to predict changes in object-oriented software systems. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR '07)*, pages 27–38, Amsterdam, Netherlands, March 2007.

[ST08] Ali R. Sharafat and Ladan Tahvildari. Change prediction in object-oriented software systems: A probabilistic approach. *Journal of Software*, 3(5):26–39, May 2008.

[SZ05] George Spanoudakis and Andrea Zisman. Software traceability: A roadmap. In Chang S. K., editor, *Handbook of Software Engineering and Knowledge Engineering*, volume III, pages 395–428. World Scientific Publishing Co., River Edge, NJ, 2005.

[SZPMK04] George Spanoudakis, Andre Zisman, Elena Perez-Minana, and Paul Krause. Rule-based generation of requirements traceability relations. *Journal of Systems and Software*, 72(2):105–127, 2004.

[tHGK$^+$09] David ten Hove, Arda Goknil, Ivan Kurtev, Klaas van den Berg, and Koos de Goede. Change impact analysis for sysml requirements models based on semantics of trace relations. In *Proceedings of the ECMDA Traceability Workshop (ECMDA-TW)*, pages 17–28, Enschede, Netherlands, June 2009.

[Tip94] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1994.

[TLCvV11a] Antony Tang, Peng Liang, Viktor Clerc, and Hans van Vliet. *Relating Software Requiremens and Software Architecture*, chapter Supporting Co-evolving Architectural Requirements and Design through Traceability and Reasoning, pages 35–60. Springer, 2011.

[TLCvV11b] Antony Tang, Peng Liang, Viktor Clerc, and Hans van Vliet. *Supporting Co-evolving Architectural Requirements and Design through Traceability and Reasoning*, pages 35–60. Springer, 2011.

[TLL14] Yuan Tian, David Lo, and Julia Lawall. Automated construction of a software-specific word similarity database. In *Proceedings of the IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE 2014)*, pages 44–53, Antwerp, Belgium, February 2014.

[TLvV11] Antony Tang, Peng Liang, and Hans van Vliet. Software architecture documentation: The road ahead. In *Proceedings of the 2011 Ninth Working IEEE/IFIP Conference on Software Architecture*, pages 252–255, Boulder, Colorado, USA, June 2011.

[TNJH07] Antony Tang, Ann Nicholson, Yan Jin, and Jun Han. Using bayesian belief networks for change impact analysis in architecture design. *The Journal of Systems and Software*, 80:127–148, 2007.

[Ton03] Paolo Tonella. Using a concept lattice of decomposition slices for program understanding and impact analysis. *IEEE Transactions on Software Engineering*, 29(6):495–509, June 2003.

[VBF07] László Vidács, Árpád Beszédes, and Rudolf Ferenc. Macro impact analysis using macro slicing. In *Proceedings of the Second International Conference on Software and Data Technologies (ICSOFT '07)*, pages 230–235, July 2007.

[vdWvdH02] Christian van der Westhuizen and André van der Hoek. Understanding and propagating architectural changes. In Jan Bosch, Morven Gentleman, Christine Hofmeister, and Juha Kuusela, editors, *Software Architecture*, volume 97 of *IFIP - The International Federation for Information Processing*, pages 95–109. Springer US, 2002.

[VGSMD03] Pieter Van Gorp, Hans Stenten, Tom Mens, and Serge Demeyer. Towards automating source-consistent uml refactorings. *Lecture Notes in Computer Science*, 2863:144–158, 2003.

[VSV08] Stephane Vaucher, Houari Sahraoui, and Jean Vaucher. Discovering new change patterns in object-oriented systems. In *Proceedings of the 2008 15th Working Conference on Reverse Engineering (WCRE '08)*, pages 37–41, Washington, DC, USA, 2008.

[W3C07] W3C. XSL Transformations (XSLT) Version 2.0. W3C Recommendation, Januar 2007.

[W3C09] W3C OWL Working Group. OWL 2 Web Ontology Language Document Overview. W3C Recommendation, October 27 2009.

[W3C10] W3C. XML Path Language (XPath) 2.0 (Second Edition). W3C Recommendation, December 2010.

[Wag10] Philipp Wagner. Tool Support for the Analysis during Software Architectural Design (in German: Werkzeugunterstützung für die Analyse beim Softwarearchitekturentwurf). Bachelor thesis, Ilmenau University of Technology, Ilmenau, Germany, December 2010.

[Wil12] Jerod W. Wilkerson. A software change impact analysis taxonomy. In *Proceedings of the 28th IEEE International Conference on Software Maintenance*, pages 625–628, Riva del Garda, Trento, Italy, September 2012.

[Wis14] The Wisconsin Program-Slicing Tool. http://research.cs.wisc.edu/wpis/html/, May 2014. (Accessed on November, 20th 2014).

[WJSA06] Ståle Walderhaug, Ulrik Johansen, Erlend Stav, and Jan Aagedal. Towards a generic solution for traceability in MDD. In *Proceedings of the ECMDA Traceability Workshop*, pages 41–50, Sintef, Trondheim, Norway, 2006.

[WvP10] Stefan Winkler and Jens von Pilgrim. A survey of traceability in requirements engineering and model-driven development. *Software and Systems Modeling*, 9(4):529–565, 2010.

[XS04a] Zhenchang Xing and Eleni Stroulia. Data-mining in support of detecting class co-evolution. In *Proceedings of the 16th International Conference on Software Engineering & Knowledge Engineering (SEKE'04)*, pages 123–128, June 2004.

[XS04b] Zhenchang Xing and Eleni Stroulia. Understanding class evolution in object-oriented software. In *Proceedings of the 12th IEEE International Workshop on Program Comprehension (IWPC'04)*, pages 34–43, June 2004.

[XS05] Zhenchang Xing and Eleni Stroulia. UMLDiff: An algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (ASE '05)*, pages 54–65, Long Beach, California, USA, November 2005.

[YC04] Namho Yoo and Hyeong-Ah Choi. An XML-based approach for interface impact analysis in sustained system. In *Proceedings of the International Conference on Information and Knowledge Engineering (IKE'04)*, pages 161–167, Las Vegas, Nevada, USA, June 2004.

[YCDW09] Andres Yie, Rubby Casallas, Dirk Deridder, and Dennis Wagelaar. A practical approach to multi-modeling views composition. In *Proceedings of the 3rd International Workshop on Multi-Paradigm Modeling: Concepts and Tools, colocated with the ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems*, pages 1–11, Denver, Colorado, USA, October 2009.

[YCM78] S. S. Yau, J. S. Collofello, and T. M. McGregor. Ripple effect analysis of software maintenance. In *Proceedings Computer Software and Applications Conference (COMPSAC '78)*, pages 60–65. IEEE Computer Society Press: Piscataway NJ, 1978.

[YM12] Amir Reza Yazdanshenas and Leon Moonen. Fine-grained change impact analysis for component-based product families. In *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM 2012)*, pages 119–128, Riva del Garda, Trento, Italy, September 2012.

[YMNCC04] Annie T.T. Ying, Gail C. Murphy, Raymond Ng, and Mark C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, September 2004.

[ZLZ$^+$14] He Zhang, Juan Li, Liming Zhu, Ross Jeffery, Yan Liu, Qing Wang, and Mingshu Li. Investigating dependencies in software requirements for change propagation analysis. *Information and Software Technology*, 56(1):40–53, 2014.

[ZWDZ05] Thomas Zimmermann, Peter Weissgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, June 2005.

[ZWG$^+$08] Yu Zhou, Michael Wuersch, Emanuel Giger, Harald Gall, and Jian Lue. A bayesian network based approach for change coupling prediction. In *Proceedings of the 15th Working Conference on Reverse Engineering 2008*, pages 27–36, Antwerp, Belgium, October 2008.

List of Figures

List of Tables

A. Dependency Detection Rules

Rule(s)	**Description**
TR_Cls_001	**Link classes with their superclass (UML)**
Elements	Class e1, Class e2, Generalization e3
Conditions	AND(modelEquals(e3::general, e2), modelDirectParentOf(e1, e3))
Action	createLink(e1, 'Is-A', e2)
TR_Cls_002	**Link classes with their superclass (Java)**
Elements	ClassDeclaration e1, ClassDeclaration e2, TypeAccess e3
Conditions	AND(modelEquals(e1::superclass, e3), modelEquals(e3::type, e2)))
Action	createLink(e1, 'Is-A', e2)
TR_Cls_003	**Link classes with their implemented interface (UML)**
Elements	Class e1, Interface e2, InterfaceRealization e3
Conditions	AND(modelEquals(e3::client, e3), modelEquals(e3::supplier, e2)))
Action	createLink(e1, 'Implements', e2)
TR_Cls_004	**Link classes with their implemented interface (Java)**
Elements	ClassDeclaration e1, InterfaceDeclaration e2, TypeAccess e3
Conditions	AND(modelDirectParentOf(e1, e3), modelEquals(e3::type, e2)))
Action	createLink(e1, 'Implements', e2)
TR_Cls_005	**Link UML classes and corresponding source code classes**
Elements	Class e1, ClassDeclaration e2
Conditions	valueEquals(e1::name, e2::name)
Action	createLink(e1, 'Is-Equivalent-To', e2)
TR_Cls_006	**Link classes with UML components refined by them**
Elements	Class|ClassDeclaration e1, Component e2
Conditions	valueEquals(e1::name, e2::name)
Action	createLink(e1, 'Refines', e2)
TR_Cls_007|8	**Find UML classes that implement UML [use case systems|use cases]**
Elements	Class|ClassDeclaration e1, Model|Actor e2
Conditions	valueEquals(e1::name, e2::name)
Action	createLink(e1, 'Implements', e2)
TR_Cls_009	**Find UML classes that realize UML use cases**
Elements	Class e1, UseCase e2
Conditions	ValueSimilarTo(e1::name, e2::name)
Action	createLink(e1, 'Realizes', e2)
TR_Cls_010	**Find UML classes that realize UML use cases**
Elements	Class e1, Operation e2, UseCase e3
Conditions	AND(modelDirectParentOf(e1, e2), OR(valueEquals(e1::name, e2::name), AND(valueContains(e3::name, e1::name), valueContains(e3::name, e2::name))))
Action	createLink(e1, 'Realizes', e2)
TR_Cls_011|13	**Link classes with their methods**
Elements	Class|ClassDeclaration e1, Operation|MethodDeclaration e2
Conditions	modelDirectParentOf(e1, e2)
Action	createLink(e1, 'Defines', e2)
TR_Cls_012|14	**Link classes with their attributes**
Elements	Class|ClassDeclaration e1, Property|FieldDeclaration e2
Conditions	modelDirectParentOf(e1, e2)
Action	createLink(e1, 'Defines', e2)
TR_Cls_015|16	**Link classes with their corresponding test cases**
Elements	Class|ClassDeclaration e1, Class|ClassDeclaration
Conditions	AND(valueStartsWith(e1::name, e2::name), valueEndsWith(e1::name, 'Test'))
Action	createLink(e1, 'Tests', e2)
TR_Cls_017	**Find classes that are a part of a component**
Elements	Class e1, Component e2
Conditions	modelDirectParentOf(e2, e1)
Action	createLink(e1, 'Is-Part-Of', e2)
TR_Cls_018|19	**Link classes with packages containing them**
Elements	Class|ClassDeclaration e1, Package e2
Conditions	modelDirectParentOf(e2, e1)
Action	createLink(e1, 'Is-Part-Of', e2)

TR_Cls_020	**Find UML classes that are part of an UML collaboration**
Elements	Class e1, Collaboration e2, CollaborationUse e3
Conditions	AND(modelDirectParentOf(e2, e3), valueEquals(e1::name, e3::name))
Action	createLink(e1, ’Is-Part-Of’, e2)
TR_Int_001	**Link interfaces with their superclass (UML)**
Elements	Interface e1, Interface e2, Generalization e3
Conditions	AND(modelEquals(e3::general, e2), modelDirectParentOf(e1, e3))
Action	createLink(e1, ’Is-A’, e2)
TR_Int_002	**Link interfaces with their superclass (Java)**
Elements	InterfaceDeclaration e1, InterfaceDeclaration e2, TypeAccess e3
Conditions	AND(modelEquals(e1::superclass, e3), modelEquals(e3::type, e2)))
Action	createLink(e1, ’Is-A’, e2)
TR_Cls_003	**Link UML interfaces and corresponding source code interfaces**
Elements	Interface e1, InterfaceDeclaration e2
Conditions	valueEquals(e1::name, e2::name)
Action	createLink(e1, ’Is-Equivalent-To’, e2)
TR_Cls_004	**Find interfaces that correspond to interfaces defined by components**
Elements	Interface e1, Interface e2, Component e3
Conditions	AND(valueEquals(e1::name, e2::name), OR(modelRelatedTo(e3, ’Provides’, e1), modelRelatedTo(e3, ’Provides’, e1), modelDirectParentOf(e3, e1)))
Action	createLink(e2, ’Refines’, e1)
TR_Int_005|7	**Link interfaces with their methods**
Elements	Interface|InterfaceDeclaration e1, Operation|MethodDeclaration e2
Conditions	modelDirectParentOf(e1, e2)
Action	createLink(e1, ’Defines’, e2)
TR_Int_006|8	**Link interfaces with their attributes**
Elements	Interface|InterfaceDeclaration e1, Property|FieldDeclaration e2
Conditions	modelDirectParentOf(e1, e2)
Action	createLink(e1, ’Defines’, e2)
TR_Int_009|10	**Link interfaces with packages containing them**
Elements	Interface|InterfaceDeclaration e1, Package e2
Conditions	modelDirectParentOf(e2, e1)
Action	createLink(e1, ’Is-Part-Of’, e2)
TR_Cmp_001	**Link components refining other components**
Elements	Component e1, Component e2, Component|Interface e3
Conditions	AND(valueEquals(e1::name, e2::name), modelDirectParentOf(e1,e3), NOT(modelDirectParentOf(e2, e3)))
Action	createLink(e2, ’Refines’, e1)
TR_Cmp_002|3	**Find components implementing a use case [system|actor]**
Elements	Component e1, Model|Actor e2
Conditions	valueEquals(e1::name, e2::name)
Action	createLink(e1, ’Implements’, e2)
TR_Cmp_004|5|10	**Find components that are a part of a [component|package|deployment node]**
Elements	Component e1, Component|Package|Node e2
Conditions	modelDirectParentOf(e1::name, e2::name)
Action	createLink(e2, ’Is-Part-Of’, e1)
TR_Cmp_006	**Find UML interfaces required by UML components**
Elements	Interface e1, Component e2, Usage e3
Conditions	AND(modelEquals(e3::supplier, e1), modelEquals(e3::client, e2))
Action	createLink(e2, ’Requires’, e1)
TR_Cmp_007	**Find UML interfaces provided by UML components**
Elements	Component e1, Interface e2, InterfaceRealization e3
Conditions	AND(modelDirectParentOf(e1, e3), modelEquals(e3::supplier, e2))
Action	createLink(e1, ’Provides’, e2)
TR_Cmp_008	**Link components with their ports**
Elements	Component e1, Port e2
Conditions	modelDirectParentOf(e1, e2)
Action	createLink(e1, ’Defines’, e2)
TR_Cmp_009	**Link components with their artifacts**
Elements	Component e1, Port e2
Conditions	modelDirectParentOf(e1, e2)
Action	createLink(e1, ’Contains”’, e2)
TR_Prt_001	**Link ports with required interfaces**
Elements	Port e1, Interface e2, Usage e3
Conditions	AND(modelEquals(e3::supplier, e2), modelEquals(e3::client, e1))
Action	createLink(e1, ’Requires’, e2)
TR_Prt_002	**Link ports with interfaces provided by them**
Elements	Port e1, Interface e2, Usage e3
Conditions	AND(modelEquals(e3::supplier, e1), modelEquals(e3::client, e2))
Action	createLink(e1, ’Provides’, e2)
TR_Pck_001|2	**Link packages with their sub-packages**

Elements	Package e1, Package e2
Conditions	modelDirectParentOf(e2, e1)
Action	createLink(e1, 'Is-Part-Of', e2)
TR_Pck_003	**Link UML packages with corresponding Java packages**
Elements	Package e1, Package e2
Conditions	AND(valueNotNull(e1::umlId), valueEquals(e1::name, e2::name))
Action	createLink(e1, 'Is-Equivalent-To', e2)
TR_Mth_001	**Link methods with their return types (UML)**
Elements	Operation e1, Parameter e2, DataType e3
Conditions	OR(modelEquals(e1::type, e3), AND(modelEquals(e2::type, e3), modelDirectParentOf(e1, e2), NOT(valueNotNull(e2::name))))
Action	createLink(e1, 'Is-Type-Of', e3)
TR_Mth_002|3	**Link methods (UML) with their return types if they return an instance of a [class|interface]**
Elements	Operation e1, Parameter e2, Class|Interface e3
Conditions	AND(modelDirectParentOf(e1, e2), modelEquals(e2::type, e3), valueEquals(e2::direction, 'Return'))
Action	createLink(e1, 'Is-Type-Of', e3)
TR_Mth_004|5	**Link [class|interface]-attributes and their getter-methods (UML)**
Elements	Operation e1, Property e2, Class|Interface e3
Conditions	AND(modelDirectParentOf(e3, e1), modelDirectParentOf(e3, e2), valueStartsWith(e1::name, 'get'), valueEndsWith(e1::name, e2::name))
Action	createLink(e1, 'Uses', e3)
TR_Mth_006|7	**Link [class|interface]-attributes and their setter-methods (UML)**
Elements	Operation e1, Property e2, Class|Interface e3
Conditions	AND(modelDirectParentOf(e3, e1), modelDirectParentOf(e3, e2), valueStartsWith(e1::name, 'set'), valueEndsWith(e1::name, e2::name))
Action	createLink(e1, 'Modifies', e3)
TR_Mth_008|22	**Link test-methods with the methods they test**
Elements	Class|ClassDeclaration e1, Class|ClassDeclaration e2, Operation|MethodDeclaration e3, Operation|MethodDecl. e4
Conditions	AND(modelRelatedTo(e1, 'Tests', e2), modelDirectParentOf(e1, e3), modelDirectParentOf(e2, e4), valueContains(e3::name, e4::name), valueStartsWith(e3::name, 'test'))
Action	createLink(e3, 'Tests', e4)
TR_Mth_009|23	**Link interface methods with methods of classes implementing them**
Elements	Class|ClassDeclaration e1, Operation|MethodDeclaration e2, Interface|InterfaceDecl. e3, Operation|MethodDecl. e4
Conditions	AND(modelRelatedTo(e1, 'Implements', e3), modelDirectParentOf(e1, e2), modelDirectParentOf(e3, e4), valueEquals(e2::name, e4::name))
Action	createLink(e3, 'Implements', e4)
TR_Mth_010|24	**Link methods with their method parameters**
Elements	Parameter|SingleVariableDeclaration e1, Operation|MethodDeclaration e2
Conditions	modelDirectParentOf(e2, e1)
Action	createLink(e2, 'Defines', e1)
TR_Mth_011	**Link UML operations with corresponding source code methods**
Elements	Operation e1, MethodDeclaration e2, Class|Interface e3, ClassDeclaration|InterfaceDeclaration e4
Conditions	AND(valueEquals(e1::name, e2::name), valueEquals(e3::name, e4::name), modelDirectParentOf(e3, e1), modelDirectParentOf(e4, e2))
Action	createLink(e1, 'Is-Equivalent-To', e2)
TR_Mth_012|13|14	**Find methods that realize UML [sequences|use cases|activities]**
Elements	Operation|MethodDeclaration e1, UseCase|Interaction|CallBehaviorAction|Activity e2
Conditions	OR(valueEquals(e1::name, e2::name), valueSimilarTo(e1::name, e2::name))
Action	createLink(e1, 'Realizes', e2)
TR_Mth_015|16|17	**Link methods with their return types (Java)**
Elements	MethodDeclaration e1, TypeAccess e2, ClassDeclaration|InterfaceDeclaration|DataType e3
Conditions	AND(modelDirectParentOf(e1, e2), modelEquals(e2::type, e3))
Action	createLink(e1, 'Is-Type-Of', e3)
TR_Mth_018|19	**Link attributes and their getter-methods (Java)**
Elements	MethodDeclaration e1, FieldDeclaration e2, VariableDeclarationFragment e3, ClassDeclaration|InterfaceDeclaration e4
Conditions	AND(modelDirectParentOf(e2, e3), modelDirectParentOf(e4, e1), modelDirectParentOf(e4, e2), valueStartsWith(e1::name, 'get'), valueEndsWith(e1::name, e3::name))
Action	createLink(e1, 'Uses', e2)
TR_Mth_020|21	**Link attributes and their setter-methods (Java)**
Elements	MethodDeclaration e1, FieldDeclaration e2, VariableDeclarationFragment e3, ClassDeclaration|InterfaceDeclaration e4
Conditions	AND(modelDirectParentOf(e2, e3), modelDirectParentOf(e4, e1), modelDirectParentOf(e4, e2), valueStartsWith(e1::name, 'set'), valueEndsWith(e1::name, e3::name))
Action	createLink(e1, 'Modifies', e2)
TR_Mth_025-30	**Link code statements with the method they belong to**
Elements	MethodDeclaration e1, ExpressionStatement|ReturnStatement|IfStatement|ForStatement|WhileStatement|DoStmt. e2
Conditions	modelParentOf(e1, e2)
Action	createLink(e1, 'Contains', e2)
TR_Mth_031	**Link Java methods with other methods called by them**
Elements	MethodDeclaration e1, MethodDeclaration e2, ExpressionStatement e3
Conditions	AND(modelParentOf(e1, e3), modelRelatedTo(e3, 'Calls', e2))

Action	createLink(e1, 'Calls', e2)
TR_Att_001|2|3	**Link attributes with their datatype (UML)**
Elements	Property e1, Class|Interface|DataType e2
Conditions	OR(modelEquals(e1::type, e2), modelEquals(e1::datatype, e2))
Action	createLink(e1, 'Is-Instance-Of', e2)
TR_Att_004	**Link UML properties with corresponding source code attributes**
Elements	Property e1, FieldDeclaration e2, VariableDeclarationFragment e3, Class e4, ClassDeclaration e5
Conditions	AND(valueEquals(e1::name, e3::name), modelDirectParentOf(e2, e3), valueEquals(e4::name, e5::name), modelDirectParentOf(e4, e1), modelDirectParentOf(e5, e2))
Action	createLink(e1, 'Is-Equivalent-To', e2)
TR_Att_005|6|7	**Link attributes with their datatype (Java)**
Elements	FieldDeclaration e1, TypeAccess e2, ClassDeclaration|InterfaceDeclaration|ParameterizedType|PrimitiveType e3
Conditions	AND(modelEquals(e2::type, e3), modelDirectParentOf(e1, e2))
Action	createLink(e1, 'Is-Instance-Of', e2)
TR_Par_001|2|3	**Link method-parameters with their datatype (UML)**
Elements	Parameter e1, Class|Interface|DataType e2
Conditions	AND(modelEquals(e1::type, e2), NOT(valueEquals(e1::direction, 'return')))
Action	createLink(e1, 'Is-Instance-Of', e2)
TR_Par_004	**Link UML parameters with corresponding source code parameters**
Elements	Parameter e1, SingleVariableDeclaration e2, Operation e3, MethodDeclaration e4
Conditions	AND(valueEquals(e1::name, e2::name), modelDirectParentOf(e3, e1), modelDirectParentOf(e4, e2), modelUndirectedRelatedTo(e3, 'Is-Equivalent-To' , e4))
Action	createLink(e1, 'Is-Equivalent-To', e2)
TR_Par_005|6|7	**Link attributes with their datatype (Java)**
Elements	SingleVariableDeclaration e1, TypeAccess e2, ClassDeclaration|InterfaceDecl.|ParameterizedType|PrimitiveType e3
Conditions	AND(modelEquals(e2::type, e3), modelDirectParentOf(e1, e2))
Action	createLink(e1, 'Is-Instance-Of', e2)
TR_Dat_001	**Link UML data types with corresponding source code data types**
Elements	DataType e1, ParameterizedType|PrimitiveType e2
Conditions	valueEquals(e1::name, e2::name)
Action	createLink(e1, 'Is-Equivalent-To', e2)
TR_Usc_001	**Find similar state machines and use cases that share a similar name**
Elements	StateMachine e1, UseCase e2
Conditions	valueSimilarTo(e1::name, e2::name)
Action	createLink(e1, 'Overlaps-With', e2)
TR_Usc_002	**Find UML use cases that represent evolutionary steps of other UML use cases**
Elements	UseCase e1, UseCase e2, Model e3, Model e4
Conditions	AND(modelParentOf(e3, e1), modelParentOf(e4, e2), valueEquals(e3::name, e4::name), valueContains(e1::name, e2::name))
Action	createLink(e1, 'Evolves-To', e2)
TR_Usc_003	**Link actors and use cases**
Elements	UseCase e1, Actor e2, Association e3, Property e4, Property e5
Conditions	AND(modelParentOf(e3, e4), modelDirectParentOf(e3, e5), modelEquals(e4::type, e1), modelEquals(e5::type, e2))
Action	createLink(e2, 'Uses', e1)
TR_Seq_001	**Find UML sequence diagrams that overlap with UML activities**
Elements	Interaction e1, CallBehaviorAction|Activity e2
Conditions	OR(valueEquals(e1::name, e2::name), valueContains(e1::name, e2::name), valueSimilarTo(e1::name, e2::name))
Action	createLink(e1, 'Overlaps-With', e2)
TR_Seq_002	**Find UML sequence diagrams that realize UML use cases**
Elements	Interaction e1, UseCase e2
Conditions	OR(valueEquals(e1::name, e2::name), valueContains(e1::name, e2::name), valueSimilarTo(e1::name, e2::name))
Action	createLink(e1, 'Realizes', e2)
TR_Seq_003	**Find UML sequence diagrams that realize UML use cases**
Elements	Interaction e1, UseCase e2, Message e3
Conditions	AND(modelParentOf(e1, e3), OR(valueEquals(e3::name, e2::name), valueContains(e3::name, e2::name), valueSimilarTo(e3::name, e2::name)))
Action	createLink(e1, 'Realizes', e2)
TR_Seq_004	**Find UML sequence diagrams that overlap with UML state machines**
Elements	Interaction e1, StateMachine e2
Conditions	OR(valueEquals(e1::name, e2::name), valueContains(e1::name, e2::name), valueSimilarTo(e1::name, e2::name))
Action	createLink(e1, 'Overlaps-With', e2)
TR_Seq_005	**Find UML sequence diagrams that refine a message in another UML sequence diagram**
Elements	Interaction e1, Message e2
Conditions	AND(NOT(modelParentOf(e1, e2), OR(valueEquals(e1::name, e2::name), valueContains(e1::name, e2::name), valueSimilarTo(e1::name, e2::name)))
Action	createLink(e1, 'Refines', e2)
TR_Seq_006|7|8|9	**Find UML sequence diagrams that use [classes|interface|components|deployment nodes|actors|systems]**
Elements	Interaction e1, Lifeline e2, Class|ClassDeclaration|Interface|InterfaceDeclaration|Component|Node|Actor|Model e3
Conditions	AND(modelParentOf(e1, e2), OR(valueEquals(e2::name, e3::name), valueSimilarTo(e2::name, e3::name)))
Action	createLink(e1, 'Uses', e3)

TR_Seq_010	**Find UML lifelines that construct or init other UML lifelines**
Elements	Lifeline e1, Lifeline e2, Message e3
Conditions	AND(valueEquals(e3::messageSort, 'createMessage'), valueContains(e1::coveredBy, e3::sendEvent), valueContains(e2::coveredBy, e3::receiveEvent))
Action	createLink(e1, 'Activates', e2)
TR_Seq_011	**Find UML lifelines that destroy/close/de-init other UML lifelines**
Elements	Lifeline e1, Lifeline e2, Message e3
Conditions	AND(valueEquals(e3::messageSort, 'deleteMessage'), valueContains(e1::coveredBy, e3::sendEvent), valueContains(e2::coveredBy, e3::receiveEvent))
Action	createLink(e1, 'Activates', e2)
TR_Seq_012	**Find UML lifelines that call other UML lifelines**
Elements	Lifeline e1, Lifeline e2, Message e3
Conditions	AND(valueContains(e1::coveredBy, e3::sendEvent), valueContains(e2::coveredBy, e3::receiveEvent), NOT(valueEquals(e3::messageSort, 'deleteMessage')), NOT(valueEquals(e3::messageSort, 'createMessage')))
Action	createLink(e1, 'Uses', e2)
TR_Seq_013|14	**Find [classes|interfaces] that are part of an interaction**
Elements	Interaction e1, Property e2, Lifeline e3, Class|ClassDeclaration|Interface|InterfaceDeclaration e4
Conditions	AND(modelParentOf(e1, e2), OR(valueEquals(e3::name, e4::name), AND(modelParentOf(e1, e2), modelEquals(e2::type, e4), valueEquals(e2, e3::represents))))
Action	createLink(e3, 'Is-Instance-Of', e4)
TR_Seq_015	**Find components that are part of an interaction**
Elements	Interaction e1, Lifeline e2, Component e3
Conditions	AND(modelParentOf(e1, e2), OR(valueEquals(e2::name, e3::name), valueSimilarTo(e2::name, e3::name)))
Action	createLink(e2, 'Is-Instance-Of', e3)
TR_Seq_016|17	**Find lifelines that are instances of a [deployment node|system]**
Elements	Node|Model e1, Lifeline e2
Conditions	AND(valueNotNull(e1::umlId), valueEquals(e1::name, e2::name))
Action	createLink(e2, 'Is-Instance-Of', e1)
TR_Seq_018	**Find lifelines that are equivalent to an use case actor**
Elements	Lifeline e1, Actor e2
Conditions	AND(valueNotNull(e2::umlId), valueEquals(e1::name, e2::name))
Action	createLink(e1, 'Is-Equivalent-To', e2)
TR_Seq_019	**Find messages in sequence diagrams that correspond to activities in activity diagrams**
Elements	Message e1, CallBehaviorAction|Activity e2
Conditions	OR(valueSimilarTo(e1::name, e2::name), valueEquals(e1::name, e2::name))
Action	createLink(e1, 'Is-Equivalent-To', e2)
TR_Seq_020	**Find messages in sequence diagrams that correspond to operations in class diagrams**
Elements	Message e1, Operation|MethodDeclaration e2, MessageOccurrenceSpecification e3, Lifeline e4, Class|ClassDecl. e5
Conditions	AND(modelDirectParentOf(e5, e2), valueEquals(e4::name, e5::name), modelEquals(e3::covered, e4), modelEquals(e3::message, e1), OR(valueEndsWith(e1::name, e2::name), valueEquals(e1::name, e2::name))))
Action	createLink(e1, 'Is-Equivalent-To', e2)
TR_Seq_021	**Find messages in sequence diagrams that correspond to classes**
Elements	Message e1, Class|ClassDeclaration e2
Conditions	valueEquals(e1::name, e2::name)
Action	createLink(e1, 'Is-Instance-Of', e2)
TR_Seq_022	**Find messages in sequence diagrams that correspond to operations in class diagrams**
Elements	Message e1, Operation|MethodDeclaration e2
Conditions	OR(valueEndsWith(e1::name, e2::name), valueEquals(e1::name, e2::name))
Action	createLink(e1, 'Is-Equivalent-To', e2)
TR_Dep_001|2|3	**Find deployment nodes that are equivalent to an [class|actor|component]**
Elements	Node e1, Actor|Component|Class|ClassDeclaration e2
Conditions	AND(valueNotNull(e1::umlId), valueEquals(e1::name, e2::name))
Action	createLink(e1, 'Is-Equivalent-To', e2)
TR_Dep_004	**Link nodes of a deployment diagram with its contained artifacts**
Elements	Node e1, * e2
Conditions	modelDirectParentOf(e1, e2)
Action	createLink(e1, 'Is-Part-Of', e2)
TR_Stm_001	**Find UML state machines that overlap with UML activity diagrams**
Elements	StateMachine e1, Activity e2
Conditions	valueContains(e1::name, e2::name)
Action	createLink(e1, 'Overlaps-With', e2)
TR_Stm_002	**Find UML state machines that overlap with UML classes**
Elements	StateMachine e1, State e2, Class e3, Operation e4
Conditions	AND(modelParentOf(e1, e2), modelParentOf(e3, e4), valueContains(e2::name, e4::name))
Action	createLink(e1, 'Overlaps-With', e3)
TR_Stm_003	**Find UML state machines that realize UML use cases**
Elements	StateMachine e1, UseCase e2, Transition|State e3
Conditions	AND(modelParentOf(e1, e2), valueEquals(e2::name, e3::name))
Action	createLink(e1, 'Realizes', e2)
TR_Act_001	**Find UML activities that realize UML use cases**

Elements	CallBehaviorAction|Activity e1, UseCase e2
Conditions	OR(valueEquals(e1::name, e2::name), valueSimilarTo(e1::name, e2::name))
Action	createLink(e1, 'Realizes', e2)
TR_Act_002	**Find UML activities that refine other UML activities**
Elements	CallBehaviorAction|Activity e1, Activity e2
Conditions	OR(valueEquals(e1::name, e2::name), valueSimilarTo(e1::name, e2::name))
Action	createLink(e1, 'Refines', e2)
TR_Act_003|4	**Find activity [swimlanes|object nodes] that are instances of UML classes**
Elements	ActivityPartition|ObjectNode e1, Class e2
Conditions	OR(valueEquals(e1::name, e2::name), valueSimilarTo(e1::name, e2::name))
Action	createLink(e1, 'Is-Instance-Of', e2)
TR_Src_001	**Find code statements that modify values of attributes**
Elements	FieldDeclaration e1, VariableDeclarationFragment e2, ExpressionStatement e3, Assignment e4, SingleVariableAccess e5
Conditions	AND(modelDirectParentOf(e1, e2), modelDirectParentOf(e3, e4), modelDirectParentOf(e4, e5), modelEquals(e5::variable, e2))
Action	createLink(e3, 'Modifies', e1)
TR_Src_002	**Find code statements that evaluate attributes**
Elements	FieldDeclaration e1, VariableDeclarationFragment e2, IfStatement|ForStatement|WhileStatement|DoStatement e3, InfixExpression e4, SingleVariableAccess e5
Conditions	AND(modelDirectParentOf(e1, e2), modelDirectParentOf(e3, e4), modelDirectParentOf(e4, e5), modelEquals(e5::variable, e2))
Action	createLink(e3, 'Examination', e1)
TR_Src_003	**Find code statements that use values of attributes**
Elements	FieldDeclaration e1, VariableDeclarationFragment e2, ExpressionStatement e3, Assignment e4, InfixExpression e5, SingleVariableAccess e6
Conditions	AND(modelDirectParentOf(e1, e2), modelDirectParentOf(e3, e4), modelDirectParentOf(e4, e5), modelDirectParentOf(e5, e6), modelEquals(e6::variable, e2))
Action	createLink(e3, 'Uses', e1)
TR_Src_004	**Find code statements where attributes are passed as method parameters**
Elements	FieldDecl. e1, VariableDeclarationFragment e2, ExpressionStmt. e3, MethodInvocation e4, SingleVariableAccess e5
Conditions	AND(modelParentOf(e1, e2), modelParentOf(e3, e4), modelDirectParentOf(e4, e5), modelEquals(e5::variable, e2))
Action	createLink(e3, 'Uses', e1)
TR_Src_005	**Find code statements that call methods**
Elements	MethodDeclaration e1, Statement e2, InfixExpression e3, MethodInvocation e4
Conditions	AND(modelDirectParentOf(e2, e3), modelEquals(e3::method, e1))
Action	createLink(e1, 'Calls', e2)
TR_Src_006	**Find code statements that call methods**
Elements	MethodDeclaration e1, Statement e2, MethodInvocation e3
Conditions	AND(modelParentOf(e2, e3), modelDirectParentOf(e3, e4), modelEquals(e4::method, e1))
Action	createLink(e1, 'Calls', e2)
TR_Src_007	**Find code statements that are similar to decision nodes in activity diagrams**
Elements	VariableDeclarationStatement e1, DecisionNode e2, ControlFlow e3
Conditions	AND(valueSimilarTo(e1::name, e2::name), OR(modelEquals(e2, e3::source), modelEquals(e2, e3::target)))
Action	createLink(e1, 'Is-Equivalent-To', e2)
TR_Src_008	**Find code statements that define variables**
Elements	VariableDeclarationStatement e1, VariableDeclarationFragment e2
Conditions	modelDirectParentOf(e1, e2)
Action	createLink(e1, 'Defines', e2)
TR_Src_009	**Find code statements that modify variables**
Elements	ExpressionStatement e1, VariableDeclarationFragment e2, Assignment e3 SingleVariableAccess e4
Conditions	AND(modelParentOf(e1, e3), modelParentOf(e3, e4), modelEquals(e4::variable, e2))
Action	createLink(e1, 'Modifies', e2)
TR_Src_010	**Find code statements that access variables**
Elements	Statement e1, VariableDeclarationFragment e2, SingleVariableAccess e3
Conditions	AND(modelParentOf(e1, e3), modelEquals(e3::variable, e2))
Action	createLink(e1, 'Uses', e2)
TR_Src_011	**Find code statements that import packages**
Elements	Package e1, ClassDeclaration e2, ImportDeclaration e3
Conditions	AND(modelRelatedTo(e2, 'Is-Part-Of', e1), modelEquals(e3::importedElement, e2))
Action	createLink(e3, 'Imports', e1)
TR_Src_012	**Find code statements that use parameters declared by methods**
Elements	MethodDeclaration e1, Statement e2, SingleVariableDeclaration e3, MethodInvocation e4, SingleVariableAccess e5
Conditions	AND(modelParentOf(e1, e2), modelDirectParentOf(e1, e3), modelParentOf(e2, e4), modelParentOf(e4, e5), modelEquals(e5::variable, e3))
Action	createLink(e2, 'Uses', e3)
TR_Var_001|2|3	**Find variables that are instances of a [class|interface|datatype]**
Elements	VariableDeclarationStatement e1, VariableDeclarationFragment e2, TypeAccess e3, ClassDecl.|InterfaceDecl.|Type e4
Conditions	AND(modelDirectParentOf(e1, e2), modelDirectParentOf(e1, e3), modelEquals(e3::type, e4))
Action	createLink(e2, 'Is-Instance-Of', e4)

B. Impact Propagation Rules

Rule(s)	Description
IR_Usc_001|2|3|4|5	**Delete the [method|activity|activity node|sequence|state machine] realizing the use case**
Elements	Operation|Activity|ActivityNode|Interaction|StateMachine e1, UseCase e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete use case'), modelRelatedTo(e1, 'Realizes', e2))
Action	reportImpact(e2, 'Delete [method|activity|activity node|sequence|state machine]', e1)
IR_Usc_006|7|8|9|10	**Rename the [method|activity|activity node|sequence|state machine] realizing the use case**
Elements	Operation|Activity|ActivityNode|Interaction|StateMachine e1, UseCase e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename use case'), modelRelatedTo(e1, 'Realizes', e2))
Action	reportImpact(e2, 'Rename [method|activity|activity node|sequence|state machine]', e1)
IR_Usc_011	**Move the method realizing the use case to the class implementing the new system**
Elements	System e1, UseCase e2, Class e3, Operation e4, CompositeChangeType e5
Conditions	AND(valueEquals(e5::name, 'Move use case to other system'), modelRelatedTo(e41, 'Realizes', e2), modelRelatedTo(e3, 'Implements', e1))
Action	reportImpact(e2, e1, 'Move method to other class', e4, e3)
IR_Usc_012|13|14|15|16	**Split the [method|activity|activity node|sequence|state machine] realizing the use case**
Elements	Operation|Activity|ActivityNode|Interaction|StateMachine e1, UseCase e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split use case'), modelRelatedTo(e1, 'Realizes', e2))
Action	reportImpact(e2, 'Split [method|activity|activity node|sequence|state machine]', e1)
IR_Usc_017|18|19|20|21	**Merge the [method|activity|activity node|sequence|state machine] realizing the use case**
Elements	Operation|Activity|ActivityNode|Interaction|StateMachine e1, Operation|Activity|ActivityNode|Interaction|StateMachine e2, UseCase e3, UseCase e4, CompositeChangeType e5
Conditions	AND(valueEquals(e5::name, 'Merge use case'), modelRelatedTo(e1, 'Realizes', e3), modelRelatedTo(e2, 'Realizes', e4))
Action	reportImpact(e2, e4, 'Merge [methods|activities|activity nodes|sequences|state machines]', e1, e2)
IR_Usc_022|23	**Delete the [component|class] implementing the actor**
Elements	Component|Class e1, Actor e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete actor'), modelRelatedTo(e1, 'Implements', e2))
Action	reportImpact(e2, 'Delete [component|class]', e1)
IR_Usc_024|25	**Rename the [component|class] implementing the actor**
Elements	Component|Class e1, Actor e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename actor'), modelRelatedTo(e1, 'Implements', e2))
Action	reportImpact(e2, 'Rename [component|class]', e1)
IR_Usc_026|27	**Merge the [components|classes] implementing the actors**
Elements	Component|Class e1, Component|Class e2, Actor e3, Actor e4, CompositeChangeType e5
Conditions	AND(valueEquals(e5::name, 'Merge actors'), modelRelatedTo(e1, 'Implements', e3), modelRelatedTo(e2, 'Implements', e4))
Action	reportImpact(e3, e4, 'Merge [components|classes]', e1, e2)
IR_Usc_028|29	**Split the [component|class] implementing the actors**
Elements	Component|Class e1, Actor e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split actors'), modelRelatedTo(e1, 'Implements', e2))
Action	reportImpact(e2, 'Split [components|classes]', e1)
IR_Usc_030|31	**Delete the [component|class] implementing the system**
Elements	Component|Class e1, Model e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete system'), modelRelatedTo(e1, 'Implements', e2))
Action	reportImpact(e2, 'Delete [component|class]', e1)
IR_Usc_032|33	**Rename the [component|class] implementing the system**
Elements	Component|Class e1, Model e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename system'), modelRelatedTo(e1, 'Implements', e2))
Action	reportImpact(e2, 'Rename [component|class]', e1)
IR_Usc_034|35	**Split the [component|class] implementing the actors**
Elements	Component|Class e1, Model e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split actors'), modelRelatedTo(e1, 'Implements', e2))
Action	reportImpact(e2, 'Split [components|classes]', e1)
IR_Usc_036|37	**Merge the [components|classes] implementing the systems**
Elements	Component|Class e1, Component|Class e2, Model e3, Model e4, CompositeChangeType e5
Conditions	AND(valueEquals(e5::name, 'Merge systems'), modelRelatedTo(e1, 'Implements', e3), modelRelatedTo(e2, 'Implements', e4))
Action	reportImpact(e3, e4, 'Merge [components|classes]', e1, e2)

IR_Cmp_001|5	**Delete the refining [component|class] as the refined component was deleted**
Elements	Component|Class e1, Component e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete component'), modelRelatedTo(e1, 'Refines', e2))
Action	reportImpact(e2, 'Delete [component|class]', e1)
IR_Cmp_002	**Remove the sub-components of a deleted component**
Elements	Component e1, Component e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete component'), modelRelatedTo(e2, 'Is-Part-Of', e1))
Action	reportImpact(e1, 'Delete component', e2)
IR_Cmp_003	**Remove the ports of a deleted component**
Elements	Component e1, Port e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete component'), modelRelatedTo(e1, 'Defines', e2))
Action	reportImpact(e1, 'Delete port', e2)
IR_Cmp_004	**Remove the artifacts of a deleted component**
Elements	Component e1, Artifact e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete component'), modelRelatedTo(e1, 'Contains', e2))
Action	reportImpact(e1, 'Delete artifact', e2)
IR_Cmp_006|7	**Delete the use case [system|actor] implemented by the component**
Elements	Component e1, Model|Actor e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete component'), modelRelatedTo(e1, 'Implements', e2))
Action	reportImpact(e1, 'Delete [use case system|actor]', e2)
IR_Cmp_008	**Delete the interfaces provided by the component**
Elements	Component e1, Interface e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete component'), modelRelatedTo(e1, 'Provides', e2))
Action	reportImpact(e1, 'Delete interface', e2)
IR_Cmp_009	**Delete the lifelines that are instances of the component**
Elements	Component e1, Lifeline e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete component'), modelRelatedTo(e1, 'Is-Instance-Of', e2))
Action	reportImpact(e1, 'Delete lifeline', e2)
IR_Cmp_010|11	**Rename the refining [component|class] according to the renamed component**
Elements	Component|Class e1, Component e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename component'), modelRelatedTo(e1, 'Refines', e2))
Action	reportImpact(e2, 'Rename [component|class]', e1)
IR_Cmp_012|13	**Rename the implemented [system|actor] according to the renamed component**
Elements	Component e1, Model|Actor e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename component'), modelRelatedTo(e1, 'Implements', e2))
Action	reportImpact(e1, 'Rename [use case system|actor]', e2)
IR_Cmp_014	**Rename the lifelines that are instances of the component**
Elements	Component e1, Lifeline e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename component'), modelRelatedTo(e1, 'Is-Instance-Of', e2))
Action	reportImpact(e1, 'Rename lifeline', e2)
IR_Cmp_015|16	**Merge the refining [components|classes]**
Elements	Component e1, Component e2, Component|Class e3, CompositeChangeType e4, Component|Class e5
Conditions	AND(valueEquals(e4::name, 'Merge components'), modelRelatedTo(e3, 'Refines', e1), modelRelatedTo(e5, 'Refines', e2))
Action	reportImpact(e1, e2, 'Merge [component|class]', e3, e5)
IR_Cmp_017|18	**Merge the implemented [systems|actors] according to the merged components**
Elements	Component e1, Component e2, Model|Actor e3, Model|Actor e4, CompositeChangeType e5
Conditions	AND(valueEquals(e4::name, 'Merge components'), modelRelatedTo(e3, 'Implements', e1), modelRelatedTo(e4, 'Implements', e2))
Action	reportImpact(e1, e2, 'Merge [use case systems|actors]', e3, e4)
IR_Cmp_019	**Move the artifacts to the merged component**
Elements	Component e1, Component e2, Artifact e3, CompositeChangeType e4
Conditions	AND(valueEquals(e4::name, 'Merge components'), OR(modelRelatedTo(e1, 'Contains', e3), modelRelatedTo(e2, 'Contains', e3)))
Action	reportImpact(e1, e2, 'Move artifact to other component', e3)
IR_Cmp_020	**Move the ports to the merged component**
Elements	Component e1, Component e2, Port e3, CompositeChangeType e4
Conditions	AND(valueEquals(e4::name, 'Merge components'), OR(modelRelatedTo(e1, 'Defines', e3), modelRelatedTo(e2, 'Defines', e3)))
Action	reportImpact(e1, e2, 'Move port to other component', e3)
IR_Cmp_021	**Merge the lifelines that are instances of the merged components**
Elements	Component e1, Component e2, Lifeline e3, Lifeline e4, CompositeChangeType e5
Conditions	AND(valueEquals(e5::name, 'Merge components'), modelRelatedTo(e3, 'Is-Instance-Of', e1), modelRelatedTo(e4, 'Is-Instance-Of', e2))
Action	reportImpact(e1, e2, 'Merge lifelines', e3, e4)
IR_Cmp_022|23	**Move the refining class to the new package**
Elements	Component e1, Class|Component e2, Package e3, CompositeChangeType e4
Conditions	AND(valueEquals(e4::name, 'Move component to other package'), modelRelatedTo(e2, 'Refines', e1))
Action	reportImpact(e1, e3, 'Move [class|component] to other package', e3, e3)
IR_Cmp_024|25	**Split the refining [component|class]**

Elements	Component e1, Class|Component e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split component'), modelRelatedTo(e2, 'Refines', e1))
Action	reportImpact(e1, 'Split [class|component]', e2)
IR_Cmp_026|29	**Split the implemented [use case system|actor]**
Elements	Component e1, Model|Actor e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split component'), modelRelatedTo(e1, 'Implements', e2))
Action	reportImpact(e1, 'Split [use case system|actor]', e2)
IR_Cmp_027	**Move provided interfaces when splitting components**
Elements	Component e1, Interface e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split component'), modelRelatedTo(e1, 'Provides', e2))
Action	reportImpact(e1, 'Move provided interface to other component', e2)
IR_Cmp_028	**Move ports when splitting components**
Elements	Component e1, Interface e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split component'), modelRelatedTo(e1, 'Defines', e2))
Action	reportImpact(e1, 'Move port to other component', e2)
IR_Cmp_030	**Split lifelines that are instances of the component**
Elements	Component e1, Lifeline e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split component'), modelRelatedTo(e2, 'Is-Instance-Of', e1))
Action	reportImpact(e1, 'Split lifeline', e2)
IR_Cmp_031	**Add the required interface to the refined component as well**
Elements	Component e1, Interface e2, Component e3, CompositeChangeType e4
Conditions	AND(valueEquals(e4::name, 'Add required interface to component'), modelRelatedTo(e1, 'Requires', e2), modelRelatedTo(e3, 'Refines', e2), NOT(modelRelatedTo(e3, 'Requires', e2)))
Action	reportImpact(e1, e2, 'Add required interface to component', e3, e2)
IR_Cmp_032	**Add the provided interface to the refined component as well**
Elements	Component e1, Interface e2, Component e3, CompositeChangeType e4
Conditions	AND(valueEquals(e4::name, 'Add provided interface to component'), modelRelatedTo(e1, 'Provides', e2), OR(modelRelatedTo(e3, 'Refines', e1), modelRelatedTo(e3, 'Is-Part-Of', e1)))
Action	reportImpact(e1, e2, 'Add provided interface to component', e3, e2)
IR_Cmp_033	**Delete the required interface from subcomponents and from the refined component**
Elements	Component e1, Interface e2, Component e3, CompositeChangeType e4
Conditions	AND(valueEquals(e4::name, 'Delete required interface from component'), modelRelatedTo(e1, 'Requires', e2), OR(modelRelatedTo(e3, 'Is-Part-Of', e1), modelRelatedTo(e1, 'Refines', e3)))
Action	reportImpact(e1, e2, 'Add required interface to component', e3, e2)
IR_Cmp_034	**Delete empty ports which became obsolete after removing the required interface**
Elements	Component e1, Interface e2, Port e3, AtomicChangeType e4
Conditions	AND(valueEquals(e4::name, 'Delete required interface from component'), modelRelatedTo(e1, 'Requires', e2), modelRelatedTo(e1, 'Defines', e3), modelRelatedTo(e3, 'Requires', e2))
Action	reportImpact(e1, e2, 'Delete port from component', e3)
IR_Cmp_035	**Remove the provided interface from the refined component**
Elements	Component e1, Interface e2, Component e3, AtomicChangeType e4
Conditions	AND(valueEquals(e4::name, 'Delete provided interface from component'), modelRelatedTo(e1, 'Provides', e2), modelRelatedTo(e1, 'Refines', e3), modelRelatedTo(e3, 'Provides', e2))
Action	reportImpact(e1, e2, 'Delete provided interface from component', e3, e2)
IR_Cmp_036	**Delete empty ports which became obsolete after removing the provided interface**
Elements	Component e1, Interface e2, Port e3, AtomicChangeType e4
Conditions	AND(valueEquals(e4::name, 'Delete provided interface from component'), modelRelatedTo(e1, 'Provides', e2), modelRelatedTo(e1, 'Defines', e3), modelRelatedTo(e3, 'Provides', e2))
Action	reportImpact(e1, e2, 'Delete port from component', e3)
IR_Cmp_037	**Add the new port to refining component as well**
Elements	Component e1, Port e2, Component e3, AtomicChangeType e4
Conditions	AND(valueEquals(e4::name, 'Add port to component'), modelRelatedTo(e1, 'Refines', e3))
Action	reportImpact(e1, e2, 'Add port to component', e2, e3)
IR_Cmp_038	**Add the new artifact to refining component as well**
Elements	Component e1, Artifact e2, Component e3, AtomicChangeType e4
Conditions	AND(valueEquals(e4::name, 'Add artifact to component'), modelRelatedTo(e3, 'Refines', e1))
Action	reportImpact(e1, e2, 'Add port to component', e2, e3)
IR_Cmp_039	**Remove the port from the refined component**
Elements	Component e1, Port e2, Component e3, AtomicChangeType e4
Conditions	AND(valueEquals(e4::name, 'Delete port from component'), modelRelatedTo(e1, 'Defines', e2), modelRelatedTo(e1, 'Refines', e3))
Action	reportImpact(e1, e2, 'Delete port from component', e3, e2)
IR_Cmp_040|43	**Move the [port|artifact] to the refined component**
Elements	Component e1, Component e2, Component e3, Component e4, Port e5, CompositeChangeType e6
Conditions	AND(valueEquals(e6::name, 'Move [port|artifact] to other component'), modelRelatedTo(e2, 'Defines', e5), modelRelatedTo(e1, 'Refines', e2), modelRelatedTo(e3, 'Defines', e4))
Action	reportImpact(e5, e4, 'Move [port|artifact] to to other component', e5, e3)
IR_Cmp_041	**Remove the artifact from the refined component as well**
Elements	Component e1, Artifact e2, Component e3, AtomicChangeType e4
Conditions	AND(valueEquals(e4::name, 'Delete artifact from component'), modelRelatedTo(e2, 'Is-Part-Of', e1), modelRelatedTo(e1, 'Refines', e3))

Action	reportImpact(e1, e2, 'Delete artifact from component', e3, e2)
IR_Cmp_042	**Rename the refined artifact**
Elements	Artifact e1, Artifact e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename artifact'), modelRelatedTo(e2, 'Refines', e1))
Action	reportImpact(e1, 'Rename artifact', e2)
IR_Pck_001	**Rename the corresponding UML/Java package**
Elements	Package e1, Package e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename package'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2))
Action	reportImpact(e1, 'Rename package', e2)
IR_Pck_002	**Add the new package to the corresponding UML/Java package as well**
Elements	Package e1, Package e2, Package e3, AtomicChangeType e4
Conditions	AND(valueEquals(e4::name, 'Add package'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2))
Action	reportImpact(e1, e3, 'Add package', e2, e3)
IR_Pck_003	**Add the class to the corresponding UML/Java package as well**
Elements	Package e1, Package e2, Class|ClassDeclaration e3, AtomicChangeType e4
Conditions	AND(valueEquals(e4::name, 'Add class'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2))
Action	reportImpact(e1, e3, 'Add class', e2, e3)
IR_Pck_004	**Add the interface to the corresponding UML/Java package as well**
Elements	Package e1, Package e2, Interface|InterfaceDeclaration e3, AtomicChangeType e4
Conditions	AND(valueEquals(e4::name, 'Add interface'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2))
Action	reportImpact(e1, e3, 'Add interface', e2, e3)
IR_Pck_005	**Remove all sub-packages of a deleted package**
Elements	Package e1, Package e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete package'), modelRelatedTo(e2, 'Is-Part-Of', e1))
Action	reportImpact(e1, 'Delete package', e2)
IR_Pck_006	**Delete the corresponding UML/Java package**
Elements	Package e1, Package e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete package'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2))
Action	reportImpact(e1, 'Delete package', e2)
IR_Pck_007	**Remove all classes contained by deleted package**
Elements	Package e1, Class|ClassDeclaration e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete package'), modelRelatedTo(e2, 'Is-Part-Of', e1))
Action	reportImpact(e1, 'Delete class', e2)
IR_Pck_008	**Remove all components contained by deleted package**
Elements	Package e1, Component e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete package'), modelRelatedTo(e2, 'Is-Part-Of', e1))
Action	reportImpact(e1, 'Delete component', e2)
IR_Pck_009	**Remove all interfaces contained by deleted package**
Elements	Package e1, Interface|InterfaceDeclaration e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete package'), modelRelatedTo(e2, 'Is-Part-Of', e1))
Action	reportImpact(e1, 'Delete interface', e2)
IR_Pck_010	**Move the corresponding UML/Java package to the new package**
Elements	Package e1, Package e2, Package e3, Package e4, CompositeChangeType e5
Conditions	AND(valueEquals(e5::name, 'Move package to other package'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2), modelUndirectedRelatedTo(e3, 'Is-Equivalent-To', e4))
Action	reportImpact(e1, e3, 'Move package to other package', e2, e4)
IR_Pck_011	**Merge the corresponding UML/Java packages**
Elements	Package e1, Package e2, Package e3, Package e4, CompositeChangeType e5
Conditions	AND(valueEquals(e5::name, 'Merge packages'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2), modelUndirectedRelatedTo(e3, 'Is-Equivalent-To', e4))
Action	reportImpact(e1, e2, 'Merge packages', e3, e4)
IR_Pck_012	**Move all components into the merged package**
Elements	Package e1, Package e2, Component e3, Package e4, CompositeChangeType e5
Conditions	AND(valueEquals(e5::name, 'Merge packages'), OR(modelRelatedTo(e3, 'Is-Part-Of', e1), modelRelatedTo(e3, 'Is-Part-Of', e2)))
Action	reportImpact(e1, e2, 'Move component to other package', e3)
IR_Pck_013	**Move all classes into the merged package**
Elements	Package e1, Package e2, Class|ClassDeclaration e3, Package e4, CompositeChangeType e5
Conditions	AND(valueEquals(e5::name, 'Merge packages'), OR(modelRelatedTo(e3, 'Is-Part-Of', e1), modelRelatedTo(e3, 'Is-Part-Of', e2)))
Action	reportImpact(e1, e2, 'Move class to other package', e3)
IR_Pck_014	**Move all interfaces into the merged package**
Elements	Package e1, Package e2, Interface|InterfaceDeclaration e3, Package e4, CompositeChangeType e5
Conditions	AND(valueEquals(e5::name, 'Merge packages'), OR(modelRelatedTo(e3, 'Is-Part-Of', e1), modelRelatedTo(e3, 'Is-Part-Of', e2)))
Action	reportImpact(e1, e2, 'Move interface to other package', e3)
IR_Pck_015	**Split the corresponding UML/Java package**
Elements	Package e1, Package e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split package'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2))
Action	reportImpact(e1, 'Split package', e2)

IR_Pck_016	**Move the content of the split package to the other package**
Elements	Package e1, Component e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split package'), modelRelatedTo(e2, 'Is-Part-Of', e1))
Action	reportImpact(e1, 'Move component to other package', e2)
IR_Pck_017	**Move the content of the split package to the other package**
Elements	Package e1, Interface|InterfaceDeclaration e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split package'), modelRelatedTo(e2, 'Is-Part-Of', e1))
Action	reportImpact(e1, 'Move component to other package', e2)
IR_Pck_018	**Move the content of the split package to the other package**
Elements	Package e1, Class|ClassDeclaration e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split package'), modelRelatedTo(e2, 'Is-Part-Of', e1))
Action	reportImpact(e1, 'Move component to other package', e2)
IR_Pck_019	**Adjust the import statements according to the new name of the package**
Elements	Package e1, ImportDeclaration e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename package'), modelRelatedTo(e2, 'Imports', e1))
Action	reportImpact(e1, 'Modify statement', e2)
IR_Pck_020	**Delete import statements if the imported package was deleted**
Elements	Package e1, ImportDeclaration e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete package'), modelRelatedTo(e2, 'Imports', e1))
Action	reportImpact(e1, 'Delete statement', e2)
IR_Int_001	**Add the implementation of a method declaration**
Elements	MethodDeclaration|Operation e1, Interface|InterfaceDeclaration e2, Class|ClassDeclaration e3, AtomicChangeType e4
Conditions	AND(valueEquals(e4::name, 'Add method'), modelRelatedTo(e3, 'Implements', e2))
Action	reportImpact(e2, e1, 'Add method', e3)
IR_Int_002	**Add the method to the equivalent UML/Java class/interface**
Elements	MethodDeclaration|Operation e1, Class|ClassDeclaration|Interface|InterfaceDeclaration e2, Class|ClassDeclaration|Interface|InterfaceDeclaration e3, AtomicChangeType e4
Conditions	AND(valueEquals(e4::name, 'Add method'), modelUndirectedRelatedTo(e3, 'Is-Equivalent-To', e2))
Action	reportImpact(e2, e1, 'Add method', e3)
IR_Int_003	**Add the attribute to the corresponding UML/Java classes/interfaces**
Elements	FieldDeclaration|Property e1, Class|ClassDeclaration|Interface|InterfaceDeclaration e2, Class|ClassDeclaration|Interface|InterfaceDeclaration e3, AtomicChangeType e4
Conditions	AND(valueEquals(e4::name, 'Add method'), modelUndirectedRelatedTo(e3, 'Is-Equivalent-To', e2))
Action	reportImpact(e2, e1, 'Add attribute', e3)
IR_Int_004	**Rename the equivalent UML/Java interface and the refined required/provided interfaces**
Elements	Interface|InterfaceDeclaration e1, Interface|InterfaceDeclaration e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename interface'), OR(modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2), modelUndirectedRelatedTo(e1, 'Refines', e2)))
Action	reportImpact(e1, 'Rename interface', e2)
IR_Int_005	**Rename the "inherits" property of the subclass to match the new name of the superclass**
Elements	Interface|InterfaceDeclaration e1, Interface|InterfaceDeclaration e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename interface'), modelRelatedTo(e1, 'Is-A', e2))
Action	reportImpact(e1, 'Add superclass to interface', e2)
IR_Int_006	**Update the "implements" property of the class to match the new name of the interface**
Elements	Interface|InterfaceDeclaration e1, Class|ClassDeclaration e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename interface'), modelRelatedTo(e1, 'Implements', e2))
Action	reportImpact(e1, 'Add implemented interface', e2)
IR_Int_007	**Update the data type of the attribute to match the new name of the interface**
Elements	Interface|InterfaceDeclaration e1, Property|FieldDeclaration e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename interface'), modelRelatedTo(e2, 'Is-Instance-Of', e))
Action	reportImpact(e1, 'Change attribute data type', e2)
IR_Int_008	**Update the data type of the variable to match the new name of the interface**
Elements	Interface|InterfaceDeclaration e1, VariableDeclarationFragment e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename interface'), modelRelatedTo(e2, 'Is-Instance-Of', e1))
Action	reportImpact(e1, 'Change attribute data type', e2)
IR_Int_009	**Update the data type of the variable to match the new name of the interface**
Elements	Interface|InterfaceDeclaration e1, Parameter|SingleVariableDeclaration e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename interface'), modelRelatedTo(e2, 'Is-Instance-Of', e1))
Action	reportImpact(e1, 'Change methodparameter data type', e2)
IR_Int_010	**Update the return type of the method to match the new name of the interface**
Elements	Interface|InterfaceDeclaration e1, Operation|MethodDeclaration e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename interface'), modelRelatedTo(e2, 'Is-Type-Of', e1))
Action	reportImpact(e1, 'Change methodparameter data type', e2)
IR_Int_011	**Update the name of the lifeline to match the new name of the interface**
Elements	Interface|InterfaceDeclaration e1, Lifeline e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename interface'), modelRelatedTo(e2, 'Is-Instance-Of', e1))
Action	reportImpact(e1, 'Rename lifeline', e2)
IR_Int_012	**Delete the corresponding and refining UML/Java interface**
Elements	Interface|InterfaceDeclaration e1, Interface|InterfaceDeclaration e2, AtomicChangeType e3

Conditions	AND(valueEquals(e3::name, 'Delete interface'), OR(modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2), modelUndirectedRelatedTo(e1, 'Refines', e2)))
Action	reportImpact(e1, 'Delete interface', e2)
IR_Int_013	**Delete the implementing class**
Elements	Interface|InterfaceDeclaration e1, Class|ClassDeclaration e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete interface'), modelRelatedTo(e2, 'Implements', e1))
Action	reportImpact(e1, 'Delete class', e2)
IR_Int_014|15	**Delete the [methods|attributes] defined by the interface**
Elements	Interface|InterfaceDeclaration e1, Operation|MethodDeclaration|Property|FieldDeclaration e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete interface'), modelRelatedTo(e1, 'Defines', e2))
Action	reportImpact(e1, 'Delete [method|attribute]', e2)
IR_Int_016	**Delete the method which returns an instance of the deleted interface**
Elements	Interface|InterfaceDeclaration e1, Operation|MethodDeclaration e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete interface'), modelRelatedTo(e2, 'Is-Type-Of', e1))
Action	reportImpact(e1, 'Delete method', e2)
IR_Int_017	**Delete attributes that are instances of deleted interfaces**
Elements	Interface|InterfaceDeclaration e1, Property|FieldDeclaration e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete interface'), modelRelatedTo(e2, 'Is-Instance-Of', e1))
Action	reportImpact(e1, 'Delete attribute', e2)
IR_Int_018	**Delete variable that are instances of deleted interfaces**
Elements	InterfaceDeclaration e1, VariableDeclarationFragment e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete interface'), modelRelatedTo(e2, 'Is-Instance-Of', e1))
Action	reportImpact(e1, 'Delete variable', e2)
IR_Int_019	**Delete method-parameters that are instances of deleted interfaces**
Elements	Interface|InterfaceDeclaration e1, Parameter|SingleVariableDeclaration e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete interface'), modelRelatedTo(e2, 'Is-Instance-Of', e1))
Action	reportImpact(e1, 'Delete methodparameter', e2)
IR_Int_020	**Delete lifelines which represent deleted interfaces**
Elements	Interface e1, Lifeline e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete interface'), modelRelatedTo(e2, 'Is-Instance-Of', e1))
Action	reportImpact(e1, 'Delete lifeline', e2)
IR_Int_021	**Move the corresponding UML/Java interface to the new package**
Elements	Interface|InterfaceDeclaration e1, Interface|InterfaceDeclaration e2, Package e3, Package e4, CompositeChangeType e5
Conditions	AND(valueEquals(e5::name, 'Move interface to other package'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2), modelUndirectedRelatedTo(e3, 'Is-Equivalent-To', e4))
Action	reportImpact(e1, e3, 'Move interface to other package', e2, e4)
IR_Int_022	**Move the implementing class to the new package**
Elements	Interface|InterfaceDeclaration e1, Class|ClassDeclaration e2, Package e3, Package e4, CompositeChangeType e5
Conditions	AND(valueEquals(e5::name, 'Move interface to other package'), modelRelatedTo(e2, 'Implements', e1), modelUndirectedRelatedTo(e3, 'Is-Equivalent-To', e4))
Action	reportImpact(e1, e3, 'Move class to other package', e2, e4)
IR_Int_023	**Add the superclass to the corresponding UML/Java interface**
Elements	Interface|InterfaceDeclaration e1, Interface|InterfaceDeclaration e2, Interface|InterfaceDeclaration e3, Interface|InterfaceDeclaration e4, AtomicChangeType e5
Conditions	AND(valueEquals(e5::name, 'Add superclass to interface'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2), modelUndirectedRelatedTo(e3, 'Is-Equivalent-To', e4))
Action	reportImpact(e1, e3, 'Add superclass to interface', e2, e4)
IR_Int_024	**Add implementations for methods inherited from the new superclass to the implementing class**
Elements	Interface|InterfaceDeclaration e1, Interface|InterfaceDeclaration e2, Class|ClassDeclaration e3, Operation|MethodDeclaration e4, AtomicChangeType e5
Conditions	AND(valueEquals(e5::name, 'Add superclass to interface'), modelRelatedTo(e2, 'Defines', e4), modelRelatedTo(e3, 'Implements', e1))
Action	reportImpact(e1, e3, 'Add method', e4, e2)
IR_Int_025	**Delete the superclass from the corresponding UML/Java interface**
Elements	Interface|InterfaceDeclaration e1, Interface|InterfaceDeclaration e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Add superclass to interface'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2))
Action	reportImpact(e1, 'Remove superclass from interface', e2)
IR_Int_026	**Delete the implementations of methods inherited from the deleted superclass of the implementing interface**
Elements	Interface|InterfaceDeclaration e1, Interface|InterfaceDeclaration e2, Class|ClassDeclaration e3, Operation|MethodDeclaration e4, Operation|MethodDeclaration e5, AtomicChangeType e6
Conditions	AND(valueEquals(e5::name, 'Delete superclass from interface'), modelRelatedTo(e1, 'Is-A', e2), modelRelatedTo(e3, 'Implements', e1), modelRelatedTo(e3, 'Defines', e4), modelRelatedTo(e2, 'Defines', e5), modelRelatedTo(e4, 'Implements', e5))
Action	reportImpact(e1, 'Delete method', e3, e4)
IR_Int_027	**Update the superclass property of interfaces who's superclass was split**
Elements	Interface|InterfaceDeclaration e1, Interface|InterfaceDeclaration e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split interface'), modelRelatedTo(e2, 'Is-A', e1))

Action	reportImpact(e1, 'Add superclass to interface', e2)
IR_Int_028	**Split the corresponding UML/Java interfaces**
Elements	Interface|InterfaceDeclaration e1, Interface|InterfaceDeclaration e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split interface'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2))
Action	reportImpact(e1, 'Split interface', e2)
IR_Int_029	**Update the the implemented interface of a class after splitting the interface**
Elements	Interface|InterfaceDeclaration e1, Class|ClassDeclaration e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split interface'), modelRelatedTo(e2, 'Implements', e1))
Action	reportImpact(e1, 'Add implemented interface', e2)
IR_Int_030|31	**Move [attributes|methods] to other interface when splitting an interface**
Elements	Interface|InterfaceDeclaration e1, Property|FieldDeclaration|Operation|MethodDeclaration e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split interface'), modelRelatedTo(e1, 'Defines', e2))
Action	reportImpact(e1, 'Move [attribute|method] to other interface', e2)
IR_Int_032|33|34	**Update the type of [attributes|variables|parameters] after splitting interfaces**
Elements	Interface|InterfaceDeclaration e1, Property|FieldDeclaration|VariableDeclarationFragment|SingleVariableDeclaration|Parameter e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split interface'), modelRelatedTo(e2, 'Is-Instance-Of', e1))
Action	reportImpact(e1, 'Change [attribute|variable|methodparameter] data type', e2)
IR_Int_035	**Update the return type of methods after splitting interfaces**
Elements	Interface|InterfaceDeclaration e1, Operation|MethodDeclaration e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split interface'), modelRelatedTo(e2, 'Is-Type-Of', e1))
Action	reportImpact(e1, 'Change return type', e2)
IR_Int_036	**Split the lifelines that are instances of a split interface**
Elements	Interface e1, Lifeline e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split interface'), modelRelatedTo(e2, 'Is-Instance-Of', e1))
Action	reportImpact(e1, 'Split lifeline', e2)
IR_Int_037	**Merge the corresponding UML/Java interfaces**
Elements	Interface|InterfaceDeclaration e1, Interface|InterfaceDeclaration e2, Interface|InterfaceDeclaration e3, Interface|InterfaceDeclaration e4, CompositeChangeType e5
Conditions	AND(valueEquals(e5::name, 'Merge interfaces'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e3), modelUndirectedRelatedTo(e2, 'Is-Equivalent-To', e4))
Action	reportImpact(e1, e2, 'Merge interfaces', e3, e4)
IR_Int_038	**Update the superclass reference of all subclasses of the merged interfaces**
Elements	Interface|InterfaceDeclaration e1, Interface|InterfaceDeclaration e2, Interface|InterfaceDeclaration e3, CompositeChangeType e4
Conditions	AND(valueEquals(e4::name, 'Merge interfaces'), OR(modelRelatedTo(e1, 'Is-A', e3), modelRelatedTo(e2, 'Is-A', e3)))
Action	reportImpact(e1, e2, 'Add superclass to interface', e3)
IR_Int_039	**Update the implemented-class of all classes implementing one of the merged interfaces**
Elements	Interface|InterfaceDeclaration e1, Interface|InterfaceDeclaration e2, Class|ClassDeclaration e3, CompositeChangeType e4
Conditions	AND(valueEquals(e4::name, 'Merge interfaces'), OR(modelRelatedTo(e3, 'Implements', e1), modelRelatedTo(e3, 'Implements', e2)))
Action	reportImpact(e1, e2, 'Add implemented interface', e3)
IR_Int_040|44	**Move all [attributes|methods] to the merged interface**
Elements	Interface|InterfaceDeclaration e1, Interface|InterfaceDeclaration e2, FieldDeclaration|Property|MethodDeclaration|Operation e3, CompositeChangeType e4
Conditions	AND(valueEquals(e4::name, 'Merge interfaces'), OR(modelRelatedTo(e2, 'Defines', e3), modelRelatedTo(e1, 'Defines', e3)))
Action	reportImpact(e1, e2, 'Move [attribute|method] to other interface', e3)
IR_Int_041|42|43	**Update the type of the [attributes|variables|parameters] to match the merged interfaces**
Elements	Interface|InterfaceDeclaration e1, Interface|InterfaceDeclaration e2, FieldDeclaration|Property|VariableDeclarationFragment|Parameter|SingleVariableDeclaration e3, CompositeChangeType e4
Conditions	AND(valueEquals(e4::name, 'Merge interfaces'), OR(modelRelatedTo(e3, 'Is-Instance-Of', e1), modelRelatedTo(e3, 'Is-Instance-Of', e2)))
Action	reportImpact(e1, e2, 'Change [attribute|variable|methodparameter] data type', e3)
IR_Int_045	**Update all methods who returned one of the merged interfaces**
Elements	Interface|InterfaceDeclaration e1, Interface|InterfaceDeclaration e2, Operation|MethodDeclaration e3, CompositeChangeType e4
Conditions	AND(valueEquals(e4::name, 'Merge interfaces'), OR(modelRelatedTo(e3, 'Is-Type-Of', e1), modelRelatedTo(e3, 'Is-Type-Of', e2)))
Action	reportImpact(e1, e2, 'Change return type', e3)
IR_Int_046	**Merge lifelines that are instances of one the merged interfaces**
Elements	Interface e1, Interface e2, Lifeline e3, Lifeline e4, CompositeChangeType e5
Conditions	AND(valueEquals(e5::name, 'Merge interfaces'), modelRelatedTo(e3, 'Is-Instance-Of', e1), modelRelatedTo(e4, 'Is-Instance-Of', e2))
Action	reportImpact(e1, e2, 'Merge lifelines', e3, e4)
IR_Cls_001	**Delete the corresponding UML/Java class**
Elements	Class|ClassDeclaration e1, Class|ClassDeclaration e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete class'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2))

Action	reportImpact(e1, 'Delete class', e2)
IR_Cls_002	**Delete interfaces whose only implementing class was deleted and which is not superclassed by any class**
Elements	Class|ClassDeclaration|Interface|InterfaceDelcaration e1, Interface|InterfaceDelcaration e2, Class|ClassDeclaration|Interface|InterfaceDelcaration e3, AtomicChangeType e4
Conditions	AND(OR(valueEquals(e4::name, 'Delete class'), valueEquals(e4::name, 'Delete interface')), OR(modelRelatedTo(e1, 'Implements', e2), modelRelatedTo(e2, 'Is-A', e1)), NOT(modelRelatedTo(e2, 'Is-A', e3)), NOT(modelRelatedTo(e3, 'Implements', e2)), NOT(modelEquals(e3, e1)))
Action	reportImpact(e1, 'Delete interface', e2)
IR_Cls_003	**Delete the superclass property of a UML/Java class**
Elements	Class|ClassDeclaration e1, Class|ClassDeclaration e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete class'), modelRelatedTo(e2, 'Is-A', e1))
Action	reportImpact(e1, 'Delete superclass from class', e2)
IR_Cls_004	**Delete the test class when the corresponding class is removed**
Elements	Class|ClassDeclaration e1, Class|ClassDeclaration e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete class'), modelRelatedTo(e2, 'Tests', e1))
Action	reportImpact(e1, 'Delete class', e2)
IR_Cls_005	**Delete the component that was refined by the deleted class**
Elements	Class|ClassDeclaration e1, Component e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete class'), modelRelatedTo(e1, 'Refines', e2))
Action	reportImpact(e1, 'Delete component', e2)
IR_Cls_006	**Delete the use case system that was implemented by the deleted class**
Elements	Class|ClassDeclaration e1, Model e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete class'), modelRelatedTo(e1, 'Implements', e2))
Action	reportImpact(e1, 'Delete system', e2)
IR_Cls_007	**Delete the methods that were defined by the deleted class**
Elements	Class|ClassDeclaration e1, Operation|MethodDeclaration e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete class'), modelRelatedTo(e1, 'Defines', e2))
Action	reportImpact(e1, 'Delete method', e2)
IR_Cls_008	**Delete the attributes that were defined by the deleted class**
Elements	Class|ClassDeclaration e1, Property|FieldDeclaration e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete class'), modelRelatedTo(e1, 'Defines', e2))
Action	reportImpact(e1, 'Delete attribute', e2)
IR_Cls_009	**Delete the method which returns an instance of the deleted class**
Elements	Class|ClassDeclaration e1, Operation|MethodDeclaration e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete class'), modelRelatedTo(e2, 'Is-Type-Of', e1))
Action	reportImpact(e1, 'Delete method', e2)
IR_Cls_010|11|12	**Delete [attributes|variables|parameters] which type is a deleted class**
Elements	Class|ClassDeclaration e1, FieldDeclaration|Property|VariableDeclarationFragment|Parameter| SingleVariableDeclaration e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete class'), modelRelatedTo(e2, 'Is-Instance-Of', e1))
Action	reportImpact(e1, 'Delete [attribute|variable|methodparameter]', e2)
IR_Cls_013	**Delete the use case actor implemented by the class**
Elements	Class|ClassDeclaration e1, Actor e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete class'), modelRelatedTo(e1, 'Implements', e2))
Action	reportImpact(e1, 'Delete actor', e2)
IR_Cls_14|15|16	**Delete [lifelines|swimlanes|object nodes] that were instances of the deleted class**
Elements	Class|ClassDeclaration e1, Lifeline|ActivityPartition e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete class'), modelRelatedTo(e2, 'Is-Instance-Of', e1))
Action	reportImpact(e1, 'Delete [lifeline|swimlane|object node]', e2)
IR_Cls_017	**Rename the equivalent UML/Java class**
Elements	Class|ClassDeclaration e1, Class|ClassDeclaration e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename class'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2))
Action	reportImpact(e1, 'Rename class', e2)
IR_Cls_018	**Rename the test class when the actual class is renamed**
Elements	Class|ClassDeclaration e1, Class|ClassDeclaration e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename class'), modelRelatedTo(e2, 'Tests', e1))
Action	reportImpact(e1, 'Rename class', e2)
IR_Cls_019	**Rename the "inherits" property of the subclass to match the new name of the superclass**
Elements	Class|ClassDeclaration e1, Class|ClassDeclaration e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename class'), modelRelatedTo(e2, 'Is-A', e1))
Action	reportImpact(e1, 'Add superclass to class', e2)
IR_Cls_020	**Rename the refined component**
Elements	Class|ClassDeclaration e1, Component e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename class'), modelRelatedTo(e1, 'Refines', e2))
Action	reportImpact(e1, 'Rename component', e2)
IR_Cls_021|26	**Rename the implemented use case [system|actor]**
Elements	Class|ClassDeclaration e1, Model|Actor e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename class'), modelRelatedTo(e1, 'Implements', e2))
Action	reportImpact(e1, 'Rename [system|actor]', e2)
IR_Cls_022|23|24	**Change the type of the [attributes|variables|parameters] to match to new name of the class**

Elements	Class|ClassDeclaration e1, FieldDeclaration|Property|VariableDeclarationFragment|Parameter| SingleVariableDeclaration e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename class'), modelRelatedTo(e2, 'Is-Instance-Of', e1))
Action	reportImpact(e1, 'Change [attribute|variable|methodparameter] data type', e2)
IR_Cls_025	**Change the return type of the method to match the new name of the class**
Elements	Class|ClassDeclaration e1, MethodDeclaration|Operation e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename class'), modelRelatedTo(e2, 'Uses', e1))
Action	reportImpact(e1, 'Change return type', e2)
IR_Cls_027|28|29	**Rename all [lifelines|swimlanes|object nodes] that are instances of the renamed class**
Elements	Class|ClassDeclaration e1, Lifeline|ActivityPartition e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename class'), modelRelatedTo(e2, 'Is-Instance-Of', e1))
Action	reportImpact(e1, 'Rename [lifeline|swimlane|object node]', e2)
IR_Cls_030	**Move the corresponding UML/Java class to the new package**
Elements	Class|ClassDeclaration e1, Class|ClassDeclaration e2, Package e3, Package e4, CompositeChangeType e5
Conditions	AND(valueEquals(e5::name, 'Move class to other package'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2), modelUndirectedRelatedTo(e3, 'Is-Equivalent-To', e4))
Action	reportImpact(e1, e3, 'Move class to other package', e2, e4)
IR_Cls_031	**Move the refined component to the new package**
Elements	Class|ClassDeclaration e1, Component e2, Package e3, CompositeChangeType e4
Conditions	AND(valueEquals(e4::name, 'Move class to other package'), modelRelatedTo(e1, 'Refines', e2))
Action	reportImpact(e1, e3, 'Move class to other package', e2, e3)
IR_Cls_032	**Add the superclass to the corresponding UML/Java class**
Elements	Class|ClassDeclaration e1, Class|ClassDeclaration e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Add superclass to class'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2))
Action	reportImpact(e1, e3, 'Add superclass to class', e2, e4)
IR_Cls_033	**Delete the superclass from the corresponding UML/Java class**
Elements	Class|ClassDeclaration e1, Class|ClassDeclaration e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Remove superclass from class'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2))
Action	reportImpact(e1, e3, 'Remove superclass from class', e2, e4)
IR_Cls_034	**Change the implmented interface of the corresponding UML/Java class**
Elements	Class|ClassDeclaration e1, Class|ClassDeclaration e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Add implemented interface'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2))
Action	reportImpact(e1, 'Add implemented interface', e2)
IR_Cls_035	**Add implementations for all methods of the new interface**
Elements	Class|ClassDeclaration e1, Interface|InterfaceDeclaration e2, MethodDeclaration|Operation e3, AtomicChangeType e4
Conditions	AND(valueEquals(e4::name, 'Add implemented interface'), modelRelatedTo(e2, 'Defines', e3))
Action	reportImpact(e1, e2, 'Add method', e3, e1)
IR_Cls_036	**Change the implmented interface of the corresponding UML/Java class**
Elements	Class|ClassDeclaration e1, Class|ClassDeclaration e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete implemented interface'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2))
Action	reportImpact(e1, 'Delete implemented interface', e2)
IR_Cls_037	**Remove all method-implementations of the former interface**
Elements	Class|ClassDeclaration e1, Interface|InterfaceDeclaration e2, MethodDeclaration|Operation e3, MethodDeclaration|Operation e4, AtomicChangeType e5
Conditions	AND(valueEquals(e5::name, 'Delete implemented interface'), modelRelatedTo(e1, 'Implements', e2), modelRelatedTo(e2, 'Defines', e3), modelRelatedTo(e1, 'Implements', e5), modelRelatedTo(e5, 'Implements', e3))
Action	reportImpact(e1, 'Delete method', e3)
IR_Cls_038	**Add the abstract modifier to the corresponding UML/Java class**
Elements	Class|ClassDeclaration e1, Class|ClassDeclaration e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Add abstract modifier to class'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2))
Action	reportImpact(e1, 'Add abstract modifier to class', e2)
IR_Cls_039	**Delete the abstract modifier from the corresponding UML/Java class**
Elements	Class|ClassDeclaration e1, Class|ClassDeclaration e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete abstract modifier from class'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2))
Action	reportImpact(e1, 'Delete abstract modifier from class', e2)
IR_Cls_040	**Split corresponding classes**
Elements	Class|ClassDeclaration e1, Class|ClassDeclaration e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split class'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2))
Action	reportImpact(e1, 'Split class', e2)
IR_Cls_041	**Update the superclass property of classes who's superclass was split**
Elements	Class|ClassDeclaration e1, Class|ClassDeclaration e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split class'), modelRelatedTo(e2, 'Is-A', e1))
Action	reportImpact(e1, 'Add superclass to class', e2)

IR_Cls_042	**Split the test class when split the class**
Elements	Class|ClassDeclaration e1, Class|ClassDeclaration e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split class'), modelRelatedTo(e2, 'Tests', e1))
Action	reportImpact(e1, 'Split class', e2)
IR_Cls_043	**Split the refined component when split the class**
Elements	Class e1, Component e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split class'), modelRelatedTo(e1, 'Refines', e2))
Action	reportImpact(e1, 'Split component', e2)
IR_Cls_044	**Split the implemented interface when splitting classes**
Elements	Class|ClassDeclaration e1, Interface|InterfaceDeclaration e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split class'), modelRelatedTo(e1, 'Implements', e2))
Action	reportImpact(e1, 'Split interface', e2)
IR_Cls_045	**Split the implemented use case actor when splitting classes**
Elements	Class|ClassDeclaration e1, Actor e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split class'), modelRelatedTo(e1, 'Implements', e2))
Action	reportImpact(e1, 'Split actor', e2)
IR_Cls_046	**Move methods to other class when splitting a class**
Elements	Class|ClassDeclaration e1, Operation|MethodDeclaration e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split class'), modelRelatedTo(e1, 'Defines', e2))
Action	reportImpact(e1, 'Move method to other class', e2)
IR_Cls_047	**Update the return type of methods after splitting classes**
Elements	Class|ClassDeclaration e1, Operation|MethodDeclaration e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split class'), modelRelatedTo(e2, 'Is-Type-Of', e1))
Action	reportImpact(e1, 'Change return type', e2)
IR_Cls_048|50|51	**Update the type of [attributes|variables|methodparamters] after splitting classes**
Elements	Class|ClassDeclaration e1, Property|FieldDeclaration|VariableDeclarationFragment|Parameter|SingleVariableDeclaration e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split class'), modelRelatedTo(e2, 'Is-Instance-Of', e1))
Action	reportImpact(e1, 'Change [attribute|variable|methodparamter] data type', e2)
IR_Cls_049	**Move attributes to other class when splitting a class**
Elements	Class|ClassDeclaration e1, Property|FieldDeclaration e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split class'), modelRelatedTo(e1, 'Defines', e2))
Action	reportImpact(e1, 'Move attribute to other class', e2)
IR_Cls_052|53|54	**Split [lifelines|swimlanes|object nodes] that are instances of split classes**
Elements	Class|ClassDeclaration e1, Lifeline|ActivityPartition|ObjectNode e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split class'), modelRelatedTo(e2, 'Is-Instance-Of', e1))
Action	reportImpact(e1, 'Split [lifeline|swimlane|object node]', e2)
IR_Cls_055	**Merge the corresponding UML/Java classes**
Elements	Class|ClassDeclaration e1, Class|ClassDeclaration e2, Class|ClassDeclaration e3, Class|ClassDeclaration e4, CompositeChangeType e5
Conditions	AND(valueEquals(e5::name, 'Merge classes'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e3), modelUndirectedRelatedTo(e2, 'Is-Equivalent-To', e4))
Action	reportImpact(e1, e2, 'Merge classes', e3, e4)
IR_Cls_056	**Update the superclass reference of all subclasses of the merged classes**
Elements	Class|ClassDeclaration e1, Class|ClassDeclaration e2, Class|ClassDeclaration e3, CompositeChangeType e4
Conditions	AND(valueEquals(e4::name, 'Merge classes'), OR(modelRelatedTo(e3, 'Is-A', e1), modelRelatedTo(e3, 'Is-A', e2)))
Action	reportImpact(e1, e2, 'Add superclass to class', e3)
IR_Cls_057	**Merge test classes when merging classes**
Elements	Class|ClassDeclaration e1, Class|ClassDeclaration e2, Class|ClassDeclaration e3, Class|ClassDeclaration e4, CompositeChangeType e5
Conditions	AND(valueEquals(e5::name, 'Merge classes'), modelRelatedTo(e3, 'Tests', e1), modelRelatedTo(e4, 'Tests', e2))
Action	reportImpact(e1, e2, 'Merge classes', e3, e4)
IR_Cls_058	**Merge the refined components when merging classes**
Elements	Class|ClassDeclaration e1, Class|ClassDeclaration e2, Component e3, Component e4, CompositeChangeType e5
Conditions	AND(valueEquals(e5::name, 'Merge classes'), modelRelatedTo(e1, 'Refines', e3), modelRelatedTo(e2, 'Refines', e4))
Action	reportImpact(e1, e2, 'Merge components', e3, e4)
IR_Cls_059	**Merge the implemented actors when merging classes**
Elements	Class|ClassDeclaration e1, Class|ClassDeclaration e2, Actor e3, Actor e4, CompositeChangeType e5
Conditions	AND(valueEquals(e5::name, 'Merge classes'), modelRelatedTo(e1, 'Implements', e3), modelRelatedTo(e2, 'Implements', e4))
Action	reportImpact(e1, e2, 'Merge actors', e3, e4)
IR_Cls_060	**Move all methods to the merged class**
Elements	Class|ClassDeclaration e1, Class|ClassDeclaration e2, Operation|MethodDeclaration e3, CompositeChangeType e4
Conditions	AND(valueEquals(e4::name, 'Merge classes'), OR(modelRelatedTo(e1, 'Defines', e3), modelRelatedTo(e2, 'Defines', e3)))
Action	reportImpact(e1, e2, 'Move method to other class', e3)
IR_Cls_061	**Update all methods who returned one of the merged classes**

Elements	Class|ClassDeclaration e1, Class|ClassDeclaration e2, Operation|MethodDeclaration e3, CompositeChangeType e4
Conditions	AND(valueEquals(e4::name, 'Merge classes'), OR(modelRelatedTo(e3, 'Is-Type-Of', e1), modelRelatedTo(e3, 'Is-Type-Of', e2)))
Action	reportImpact(e1, e2, 'Change return type', e3)
IR_Cls_062	**Update the types of method-parameters to match the merged classes**
Elements	Class|ClassDeclaration e1, Class|ClassDeclaration e2, Parameter|SingleVariableDeclaration e3, CompositeChangeType e4
Conditions	AND(valueEquals(e4::name, 'Merge classes'), OR(modelRelatedTo(e3, 'Is-Instance-Of', e1), modelRelatedTo(e3, 'Is-Instance-Of', e2)))
Action	reportImpact(e1, e2, 'Change methodparameter data type', e3)
IR_Cls_063	**Move all attributes to the merged class**
Elements	Class|ClassDeclaration e1, Class|ClassDeclaration e2, Property|FieldDeclaration e3, CompositeChangeType e4
Conditions	AND(valueEquals(e4::name, 'Merge classes'), OR(modelRelatedTo(e1, 'Defines', e3), modelRelatedTo(e2, 'Defines', e3)))
Action	reportImpact(e1, e2, 'Move attribute to other class', e3)
IR_Cls_064|65	**Update the types of the [attributes|variables] to match the merged classes**
Elements	Class|ClassDeclaration e1, Class|ClassDeclaration e2, Property|FieldDeclaration| VariableDeclarationFragment e3, CompositeChangeType e4
Conditions	AND(valueEquals(e4::name, 'Merge classes'), OR(modelRelatedTo(e3, 'Is-Instance-Of', e1), modelRelatedTo(e3, 'Is-Instance-Of', e2)))
Action	reportImpact(e1, e2, 'Change [variable|attribute] data type', e3)
IR_Cls_066|67|68	**Merge [lifelines|swimlanes|activity nodes] that are instances of the merged classes**
Elements	Class e1, Class e2, Lifeline|ActivityPartition|ObjectNode e3, Lifeline|ActivityPartition|ObjectNode e4, CompositeChangeType e5
Conditions	AND(valueEquals(e5::name, 'Merge classes'), modelRelatedTo(e3, 'Is-Instance-Of', e1), modelRelatedTo(e4, 'Is-Instance-Of', e2))
Action	reportImpact(e1, e2, 'Merge [lifelines|swimlanes|object nodes] data type', e3, e4)
IR_Mth_001	**Remove the method from the corresponding UML/Java class/interface**
Elements	MethodDeclaration|Operation e1, MethodDeclaration|Operation e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete method'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2))
Action	reportImpact(e1, 'Delete method', e2)
IR_Mth_002	**Delete the implementation of a deleted method declaration**
Elements	MethodDeclaration|Operation e1, MethodDeclaration|Operation e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete method'), modelRelatedTo(e2, 'Implements', e1))
Action	reportImpact(e1, 'Delete method', e2)
IR_Mth_003	**Delete the test method of a deleted method**
Elements	MethodDeclaration|Operation e1, MethodDeclaration|Operation e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete method'), modelRelatedTo(e2, 'Tests', e1))
Action	reportImpact(e1, 'Delete method', e2)
IR_Mth_004|5|6|7|8	**Delete the [use case|activity|activity node] realized by this method**
Elements	MethodDeclaration|Operation e1, UseCase|Activity|CallBehaviorAction|StateMachine|Message e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete method'), modelRelatedTo(e1, 'Realizes', e2))
Action	reportImpact(e1, 'Delete [use case|activity|activity node|state machine|message]', e2)
IR_Mth_009	**Delete code statements containbed by the deleted method**
Elements	MethodDeclaration e1, ExpressionStatement|IfStatement|ForStatement|WhileStatement|DoStatement e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete method'), modelRelatedTo(e1, 'Contains', e2))
Action	reportImpact(e1, 'Delete statement', e2)
IR_Mth_010	**Delete code statements calling the deleted method**
Elements	MethodDeclaration e1, ExpressionStatement|IfStatement|ForStatement|WhileStatement|DoStatement e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete method'), modelRelatedTo(e1, 'Calls', e2))
Action	reportImpact(e1, 'Delete statement', e2)
IR_Mth_011	**Delete attributes that were either get or set by the deleted method**
Elements	MethodDeclaration|Operation e1, FieldDeclaration|Property e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete method'), OR(modelRelatedTo(e1, 'Modifies', e2), modelRelatedTo(e1, 'Uses', e2)))
Action	reportImpact(e1, 'Delete attribute', e2)
IR_Mth_012	**Change the return type of the corresponding UML/Java method**
Elements	MethodDeclaration|Operation e1, MethodDeclaration|Operation e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Change return type'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2))
Action	reportImpact(e1, 'Change return type', e2)
IR_Mth_013	**Change the return type of the related method declaration/implementation**
Elements	MethodDeclaration|Operation e1, MethodDeclaration|Operation e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Change return type'), modelUndirectedRelatedTo(e1, 'Implements', e2))
Action	reportImpact(e1, 'Change return type', e2)
IR_Mth_014	**Change the return type of the related method declaration/implementation**
Elements	MethodDeclaration|Operation e1, FieldDeclaration|Property e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Change return type'), modelRelatedTo(e1, 'Uses', e2))

Action	reportImpact(e1, 'Change attribute data type', e2)
IR_Mth_015	**Change the code statements calling the method**
Elements	MethodDeclaration e1, ExpressionStatement|IfStatement|ForStatement|WhileStatement|DoStatement e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Change return type'), modelRelatedTo(e1, 'Calls', e2))
Action	reportImpact(e1, 'Modify statement', e2)
IR_Mth_016	**Rename the corresponding UML/Java method**
Elements	MethodDeclaration|Operation e1, MethodDeclaration|Operation e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename method'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2))
Action	reportImpact(e1, 'Rename method', e2)
IR_Mth_017	**Rename the implementation of the method**
Elements	MethodDeclaration|Operation e1, MethodDeclaration|Operation e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename method'), modelUndirectedRelatedTo(e1, 'Implements', e2))
Action	reportImpact(e1, 'Rename method', e2)
IR_Mth_018	**Rename the test method of the method**
Elements	MethodDeclaration|Operation e1, MethodDeclaration|Operation e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename method'), modelRelatedTo(e2, 'Tests', e1))
Action	reportImpact(e1, 'Rename method', e2)
IR_Mth_019|20|21|22|23	**Rename the [use case|activity|activity node] realized by this method**
Elements	MethodDeclaration|Operation e1, UseCase|Activity|CallBehaviorAction|StateMachine|Message e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename method'), modelRelatedTo(e1, 'Realizes', e2))
Action	reportImpact(e1, 'Rename [use case|activity|activity node|state machine|message]', e2)
IR_Mth_024	**Add the static modifier to the corresponding and implementing methods**
Elements	MethodDeclaration|Operation e1, MethodDeclaration|Operation e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Add static modifier to method'), OR(modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2), modelUndirectedRelatedTo(e1, 'Implements', e2)))
Action	reportImpact(e1, 'Add static modifier to method', e2)
IR_Mth_025	**Delete the static modifier from the corresponding and implementing methods**
Elements	MethodDeclaration|Operation e1, MethodDeclaration|Operation e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete static modifier from method'), OR(modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2), modelUndirectedRelatedTo(e1, 'Implements', e2)))
Action	reportImpact(e1, 'Delete static modifier from method', e2)
IR_Mth_026	**Update the code statement calling the now non-static method**
Elements	MethodDeclaration e1, ExpressionStatement|IfStatement|ForStatement|WhileStatement|DoStatement e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete static modifier from method'), modelRelatedTo(e1, 'Calls', e2))
Action	reportImpact(e1, 'Modify statement', e2)
IR_Mth_027	**Change the final modifier of the corresponding and implementing methods**
Elements	MethodDeclaration|Operation e1, MethodDeclaration|Operation e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Add final modifier to method'), OR(modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2), modelUndirectedRelatedTo(e1, 'Implements', e2)))
Action	reportImpact(e1, 'Add final modifier to method', e2)
IR_Mth_028	**Change the final modifier of the corresponding and implementing methods**
Elements	MethodDeclaration|Operation e1, MethodDeclaration|Operation e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete final modifier from method'), OR(modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2), modelUndirectedRelatedTo(e1, 'Implements', e2)))
Action	reportImpact(e1, 'Delete final modifier from method', e2)
IR_Mth_029	**Update the visibility of the corresponding and implementing UML/Java method to "public"**
Elements	MethodDeclaration|Operation e1, MethodDeclaration|Operation e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Change method visibility to public'), OR(modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2), modelUndirectedRelatedTo(e1, 'Implements', e2)))
Action	reportImpact(e1, 'Change method visibility to public', e2)
IR_Mth_030	**Update the visibility of the corresponding and implementing UML/Java method to "protected"**
Elements	MethodDeclaration|Operation e1, MethodDeclaration|Operation e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Change method visibility to protected'), OR(modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2), modelUndirectedRelatedTo(e1, 'Implements', e2)))
Action	reportImpact(e1, 'Change method visibility protected', e2)
IR_Mth_031	**Change the code statement that is now calling a "protected" method**
Elements	MethodDeclaration e1, ExpressionStatement|IfStatement|ForStatement|WhileStatement|DoStatement e2, AtomicChangeType e3, MethodDeclaration e4, ClassDeclaration e5, ClassDeclaration e6
Conditions	AND(valueEquals(e3::name, 'Change method visibility to protected'), modelRelatedTo(e2, 'Calls', e1), modelRelatedTo(e4, 'Contains', e2), modelRelatedTo(e5, 'Defines', e1), modelRelatedTo(e6, 'Defines', e4), NOT(modelRelatedTo(e6, 'Is-A', e5)))
Action	reportImpact(e1, 'Modify statement', e2)
IR_Mth_032	**Update the visibility of the corresponding and implementing UML/Java method to "private"**
Elements	MethodDeclaration|Operation e1, MethodDeclaration|Operation e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Change method visibility to private'), OR(modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2), modelUndirectedRelatedTo(e1, 'Implements', e2)))
Action	reportImpact(e1, 'Modify statement', e2)

IR_Mth_033	**Change the code statement that is now calling a “private” method**
Elements	MethodDeclaration e1, ExpressionStatement|IfStatement|ForStatement|WhileStatement|DoStatement e2, AtomicChangeType e3, MethodDeclaration e4, ClassDeclaration e5, ClassDeclaration e6
Conditions	AND(valueEquals(e3::name, ’Change method visibility to private’), modelRelatedTo(e2, ’Calls’, e1), modelRelatedTo(e4, ’Contains’, e2), modelRelatedTo(e5, ’Defines’, e1), modelRelatedTo(e6, ’Defines’, e4), NOT(modelRelatedTo(e6, ’Is-A’, e5)))
Action	reportImpact(e1, ’Modify statement’, e2)
IR_Mth_034	**Move the corresponding UML/Java method to the UML/Java equivalent of the new class**
Elements	MethodDeclaration|Operation e1, MethodDeclaration|Operation e2, CompositeChangeType e3, Class|ClassDeclaration e4, Class|ClassDeclaration e5
Conditions	AND(valueEquals(e3::name, ’Move method to other class’), OR (modelUndirectedRelatedTo(e1, ’Is-Equivalent-To’, e2), modelUndirectedRelatedTo(e4, ’Is-Equivalent-To’, e5)))
Action	reportImpact(e1, e4, ’Move method to other class’, e2, e5)
IR_Mth_035	**Move the UML/Java method declaration to the UML/Java interface implemented by the new class**
Elements	MethodDeclaration|Operation e1, MethodDeclaration|Operation e2, CompositeChangeType e3, Interface|InterfaceDeclaration e4, Class|ClassDeclaration e5
Conditions	AND(valueEquals(e3::name, ’Move method to other class’), modelRelatedTo(e1, ’Implements’, e2), modelRelatedTo(e5, ’Implements’, e4))
Action	reportImpact(e1, e5, ’Move method to other interface’, e2, e4)
IR_Mth_036	**Move the corresponding UML/Java test method to the corresponding UML/Java test class of the new class**
Elements	MethodDeclaration|Operation e1, MethodDeclaration|Operation e2, CompositeChangeType e3, Class|ClassDeclaration e4, Class|ClassDeclaration e5
Conditions	AND(valueEquals(e3::name, ’Move method to other class’), modelRelatedTo(e2, ’Tests’, e1), modelRelatedTo(e5, ’Tests’, e4))
Action	reportImpact(e1, e4, ’Move method to other class’, e2, e5)
IR_Mth_037	**Move the attribute accessed by the getter/setter to the new class**
Elements	MethodDeclaration|Operation e1, FieldDeclaration|Property e2, CompositeChangeType e3, Class|ClassDeclaration e4, ImpactReport e5
Conditions	AND(valueEquals(e3::name, ’Move method to other class’), OR(modelRelatedTo(e1, ’Uses’, e2), modelRelatedTo(e1, ’Modifies’, e2)), NOT(AND(valueEquals(e5::changetype::name, ’Move attribute to other class’), modelEquals(e5::impactsources, e4), modelEquals(e5::impactsources, e2))))
Action	reportImpact(e1, e4, ’Move method to other class’, e2, e5)
IR_Mth_038	**Adjust code statements calling a method that has been moved to another class**
Elements	MethodDeclaration e1, ExpressionStatement|IfStatement|ForStatement|WhileStatement|DoStatement e2, AtomicChangeType e3, Class|ClassDeclaration e4, Class|ClassDeclaration e5
Conditions	AND(valueEquals(e3::name, ’Move method to other class’), modelRelatedTo(e2, ’Calls’, e1), modelDirectParentOf(e5,e1), NOT(modelRelatedTo(e4, ’Is-A’, e5)))
Action	reportImpact(e1, e4, ’Modify statement’, e2)
IR_Mth_039	**Move the corresponding UML/Java method to the UML/Java equivalent of the new interface**
Elements	MethodDeclaration|Operation e1, MethodDeclaration|Operation e2, CompositeChangeType e3, Interface|InterfaceDeclaration e4, Interface|InterfaceDeclaration e5
Conditions	AND(valueEquals(e3::name, ’Move method to other interface’), OR (modelUndirectedRelatedTo(e1, ’Is-Equivalent-To’, e2), modelUndirectedRelatedTo(e4, ’Is-Equivalent-To’, e5)))
Action	reportImpact(e1, e4, ’Move method to other interface’, e2, e5)
IR_Mth_040	**Move the implementation of the method to the implementation of the interface**
Elements	MethodDeclaration|Operation e1, Interface|InterfaceDeclaration e2, MethodDeclaration|Operation e3, CompositeChangeType e4, Class|ClassDeclaration e5
Conditions	AND(valueEquals(e4::name, ’Move method to other interface’), modelRelatedTo(e3, ’Implements’, e1), modelRelatedTo(e5, ’Implements’, e2), NOT(modelRelatedTo(e5, ’Defines’, e3)))
Action	reportImpact(e1, e2, ’Move method to other interface’, e3, e5)
IR_Mth_041	**Move the attribute along with the method to the new interface**
Elements	MethodDeclaration|Operation e1, Property|FieldDeclaration e2, CompositeChangeType e3, Interface|InterfaceDeclaration e4
Conditions	AND(valueEquals(e3::name, ’Move method to other interface’), modelRelatedTo(e1, ’Uses’, e2), NOT(modelRelatedTo(e4, ’Defines’, e2)))
Action	reportImpact(e1, e4, ’Move method to other interface’, e2, e4)
IR_Mth_042	**Move the attribute along with the method to the new interface**
Elements	MethodDeclaration|Operation e1, Property|FieldDeclaration e2, CompositeChangeType e3, Interface|InterfaceDeclaration e4
Conditions	AND(valueEquals(e3::name, ’Move method to other interface’), modelRelatedTo(e1, ’Uses’, e2), NOT(modelRelatedTo(e4, ’Defines’, e2)))
Action	reportImpact(e1, e4, ’Change attribute visibility to public’, e2)
IR_Mth_043	**Merge the corresponding UML/Java methods**
Elements	MethodDeclaration|Operation e1, MethodDeclaration|Operation e2, MethodDeclaration|Operation e3, MethodDeclaration|Operation e4, CompositeChangeType e5
Conditions	AND(valueEquals(e5::name, ’Merge methods’), modelUndirectedRelatedTo(e1, ’Is-Equivalent-To’, e3), modelUndirectedRelatedTo(e2, ’Is-Equivalent-To’, e4))
Action	reportImpact(e1, e2, ’Merge methods’, e3, e4)
IR_Mth_044	**Merge the implementations of both methods**

Elements	MethodDeclaration|Operation e1, MethodDeclaration|Operation e2, MethodDeclaration|Operation e3, MethodDeclaration|Operation e4, CompositeChangeType e5
Conditions	AND(valueEquals(e5::name, 'Merge methods'), modelUndirectedRelatedTo(e1, 'Implements', e3), modelUndirectedRelatedTo(e2, 'Implements', e4))
Action	reportImpact(e1, e2, 'Merge methods', e3, e4)
IR_Mth_045	**Merge the test methods of both methods**
Elements	MethodDeclaration|Operation e1, MethodDeclaration|Operation e2, MethodDeclaration|Operation e3, MethodDeclaration|Operation e4, CompositeChangeType e5
Conditions	AND(valueEquals(e5::name, 'Merge methods'), modelRelatedTo(e3, 'Tests', e1), modelRelatedTo(e4, 'Tests', e2))
Action	reportImpact(e1, e2, 'Merge methods', e3, e4)
IR_Mth_046	**Modify code statements calling merged methods**
Elements	MethodDeclaration e1, MethodDeclaration e2, ExpressionStatement|IfStatement|ForStatement|WhileStatement|DoStatement e3, CompositeChangeType e4
Conditions	AND(valueEquals(e4::name, 'Merge methods'), OR(modelRelatedTo(e3, 'Calls', e1), modelRelatedTo(e3, 'Calls', e2)))
Action	reportImpact(e1, e2, 'Modify statement', e3)
IR_Mth_047	**Add the method-parameter to the corresponding UML/Java method**
Elements	Parameter|SingleVariableDeclaration e1, MethodDeclaration|Operation e2, MethodDeclaration|Operation e3, AtomicChangeType e4
Conditions	AND(valueEquals(e4::name, 'Add methodparameter'), modelUndirectedRelatedTo(e2, 'Is-Equivalent-To', e3))
Action	reportImpact(e2, e1, 'Add methodparameter', e3, e1)
IR_Mth_048	**Add the method-parameter to the implementation of the method as well**
Elements	Parameter|SingleVariableDeclaration e1, MethodDeclaration|Operation e2, MethodDeclaration|Operation e3, AtomicChangeType e4
Conditions	AND(valueEquals(e4::name, 'Add methodparameter'), modelUndirectedRelatedTo(e2, 'Implements', e3))
Action	reportImpact(e2, e1, 'Add methodparameter', e3, e1)
IR_Mth_049	**Change the code statements which call methods where new method-parameters were introduced**
Elements	SingleVariableDeclaration e1, MethodDeclaration e2, ExpressionStatement|IfStatement|ForStatement|WhileStatement|DoStatement e3, AtomicChangeType e4
Conditions	AND(valueEquals(e4::name, 'Add methodparameter'), modelRelatedTo(e3, 'Calls', e2))
Action	reportImpact(e1, e2, 'Modify statement', e3)
IR_Par_001	**Delete the method-parameter from the corresponding or implementing UML/Java method**
Elements	Parameter|SingleVariableDeclaration e1, Operation|MethodDeclaration e2, Operation|MethodDeclaration e3, AtomicChangeType e4
Conditions	AND(valueEquals(e4::name, 'Delete methodparameter'), modelRelatedTo(e2, 'Defines', e1), OR (modelUndirectedRelatedTo(e2, 'Is-Equivalent-To', e3), modelUndirectedRelatedTo(e2, 'Implements', e3)))
Action	reportImpact(e1, 'Delete methodparameter', e3)
IR_Par_002	**Change the code statements which call methods where method-parameters were deleted**
Elements	SingleVariableDeclaration e1, MethodDeclaration e2, ExpressionStatement|IfStatement|ForStatement|WhileStatement|DoStatement e3 , AtomicChangeType e4
Conditions	AND(valueEquals(e4::name, 'Delete methodparameter'), modelRelatedTo(e2, 'Defines', e1), modelRelatedTo(e3, 'Calls', e2))
Action	reportImpact(e1, 'Modify statement', e3)
IR_Par_003	**Rename the method-parameter in the method declaration/implementation or the corresponding parameter**
Elements	Parameter|SingleVariableDeclaration e1, Parameter|SingleVariableDeclaration e2, Operation|MethodDeclaration e3, Operation|MethodDeclaration e4 , AtomicChangeType e5
Conditions	AND(valueEquals(e5::name, 'Rename methodparameter'), OR(modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2), AND(valueEquals(e1::name, e2::name), modelRelatedTo(e3, 'Defines', e1), modelRelatedTo(e4, 'Defines', e2), OR(modelUndirectedRelatedTo(e3, 'Implements', e4), modelUndirectedRelatedTo(e3, 'Is-Equivalent-To', e4)))))
Action	reportImpact(e1, 'Rename methodparameter', e2)
IR_Par_004	**Change the data type of the method-parameter in the method declaration/implementation**
Elements	Parameter|SingleVariableDeclaration e1, Parameter|SingleVariableDeclaration e2, Operation|MethodDeclaration e3, Operation|MethodDeclaration e4 , AtomicChangeType e5
Conditions	AND(valueEquals(e5::name, 'Change methodparameter data type'), OR(modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2), AND(valueEquals(e1::name, e2::name), modelRelatedTo(e3, 'Defines', e1), modelRelatedTo(e4, 'Defines', e2), OR(modelUndirectedRelatedTo(e3, 'Implements', e4), modelUndirectedRelatedTo(e3, 'Is-Equivalent-To', e4)))))
Action	reportImpact(e1, 'Change methodparameter data type', e2)
IR_Par_005	**Change the code statements which call methods where the type of method-parameters was changed**
Elements	SingleVariableDeclaration e1, MethodDeclaration e2, ExpressionStatement|IfStatement|ForStatement|WhileStatement|DoStatement e3 , AtomicChangeType e4
Conditions	AND(valueEquals(e4::name, 'Change methodparameter data type'), modelRelatedTo(e2, 'Defines', e1), modelRelatedTo(e3, 'Calls', e2))
Action	reportImpact(e1, 'Modify statement', e3)
IR_Par_006	**Add the final modifier to the corresponding method-parameter or of the implementing method**
Elements	SingleVariableDeclaration|Parameter e1, SingleVariableDeclaration|Parameter e2, MethodDeclaration|Operation e3, MethodDeclaration|Operatio e4 , AtomicChangeType e5

Conditions	AND(valueEquals(e5::name, 'Add final modifier to methodparameter'), OR(modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2), AND(valueEquals(e1::name, e2::name), modelRelatedTo(e3, 'Defines', e1), modelRelatedTo(e4, 'Defines', e2), OR(modelUndirectedRelatedTo(e3, 'Implements', e4), modelUndirectedRelatedTo(e3, 'Is-Equivalent-To', e4)))))
Action	reportImpact(e1, 'Add final modifier to methodparameter', e2)
IR_Par_007	**Change the code statements which call methods where the method-parameter changed to final**
Elements	SingleVariableDeclaration e1, MethodDeclaration e2, ExpressionStatement|IfStatement|ForStatement|WhileStatement|DoStatement e3 , AtomicChangeType e4
Conditions	AND(valueEquals(e4::name, 'Add final modifier to parameter'), modelRelatedTo(e2, 'Defines', e1), modelRelatedTo(e3, 'Calls', e2))
Action	reportImpact(e1, 'Modify statement', e3)
IR_Par_008	**Delete the final modifier from the corresponding method-parameter or of the implementing method**
Elements	SingleVariableDeclaration|Parameter e1, SingleVariableDeclaration|Parameter e2, MethodDeclaration|Operation e3, MethodDeclaration|Operatio e4 , AtomicChangeType e5
Conditions	AND(valueEquals(e5::name, 'Delete final modifier from methodparameter'), OR(modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2), AND(valueEquals(e1::name, e2::name), modelRelatedTo(e3, 'Defines', e1), modelRelatedTo(e4, 'Defines', e2), OR(modelUndirectedRelatedTo(e3, 'Implements', e4), modelUndirectedRelatedTo(e3, 'Is-Equivalent-To', e4)))))
Action	reportImpact(e1, 'Delete final modifier from methodparameter', e2)
IR_Par_009	**Move the method-parameter to the equivalent and implementing methods**
Elements	SingleVariableDeclaration|Parameter e1, SingleVariableDeclaration|Parameter e2, MethodDeclaration|Operation e3, MethodDeclaration|Operatio e4, MethodDeclaration|Operatio e5, MethodDeclaration|Operatio e6, CompositeChangeType e7
Conditions	AND(valueEquals(e7::name, 'Move parameter to other method'), valueEquals(e1::name, e2::name), OR(modelUndirectedRelatedTo(e3, 'Is-Equivalent-To', e4), modelUndirectedRelatedTo(e3, 'Implements', e4)), OR(modelUndirectedRelatedTo(e5, 'Implements', e6), modelUndirectedRelatedTo(e5, 'Is-Equivalent-To', e6)))
Action	reportImpact(e1, e5, 'Move parameter to other method', e2, e6)
IR_Par_010	**Change the code statements which call methods where the method-parameters were moved to other methods**
Elements	SingleVariableDeclaration e1, MethodDeclaration e2, ExpressionStatement|IfStatement|ForStatement|WhileStatement|DoStatement e3 , CompositeChangeType e4
Conditions	AND(valueEquals(e4::name, 'Move parameter to other method'), modelRelatedTo(e2, 'Defines', e1), modelRelatedTo(e3, 'Calls', e2))
Action	reportImpact(e1, 'Modify statement', e3)
IR_Att_001	**Delete the attribute from the equivalent UML/Java class/interface**
Elements	FieldDeclaration|Property e1, FieldDeclaration|Property e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete attribute'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2))
Action	reportImpact(e1, 'Delete attribute', e2)
IR_Att_002	**Remove the getter and setter for a deleted attribute**
Elements	FieldDeclaration|Property e1, MethodDeclaration|Operation e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete attribute'), OR(modelRelatedTo(e2, 'Uses', e1), modelRelatedTo(e2, 'Modifies', e1))
Action	reportImpact(e1, 'Delete method', e2)
IR_Att_003	**Modify code statements that access a deleted attribute**
Elements	FieldDeclaration e1, ExpressionStatement|IfStatement|ForStatement|WhileStatement|DoStatement e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete attribute'), OR(modelRelatedTo(e2, 'Uses', e1), modelRelatedTo(e2, 'Modifies', e1))
Action	reportImpact(e1, 'Modify statement', e2)
IR_Att_004	**Change the attribute type of the corresponding UML/Java attribute**
Elements	FieldDeclaration|Property e1, FieldDeclaration|Property e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Change attribute data type'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2))
Action	reportImpact(e1, 'Change attribute data type', e2)
IR_Att_005	**Change the return type of the method to match the new attribute type**
Elements	FieldDeclaration|Property e1, MethodDeclaration|Operation e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Change attribute data type'), modelRelatedTo(e2, 'Uses', e1))
Action	reportImpact(e1, 'Change return type', e2)
IR_Att_006	**Modify code statements that access attributes who's datatype was changed**
Elements	FieldDeclaration e1, ExpressionStatement|IfStatement|ForStatement|WhileStatement|DoStatement e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Change attribute data type'), OR(modelRelatedTo(e2, 'Uses', e1), modelRelatedTo(e2, 'Modifies', e1))
Action	reportImpact(e1, 'Modify statement', e2)
IR_Att_007	**Rename corresponding UML/Java attribute**
Elements	FieldDeclaration|Property e1, FieldDeclaration|Property e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename attribute'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2))
Action	reportImpact(e1, 'Rename attribute', e2)
IR_Att_008	**Rename the getter and setter to match the new name of the attribtute**
Elements	FieldDeclaration|Property e1, MethodDeclaration|Operation e2, AtomicChangeType e3

Conditions	AND(valueEquals(e3::name, ’Rename attribute’), OR(modelRelatedTo(e2, ’Uses’, e1), modelRelatedTo(e2, ’Modifies’, e1)))
Action	reportImpact(e1, ’Rename method’, e2)
IR_Att_009	**Modify code statements that access renamed attributes**
Elements	FieldDeclaration e1, ExpressionStatement|IfStatement|ForStatement|WhileStatement|DoStatement e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, ’Rename attribute’), OR(modelRelatedTo(e2, ’Uses’, e1), modelRelatedTo(e2, ’Modifies’, e1)))
Action	reportImpact(e1, ’Modify statement’, e2)
IR_Att_010	**Add a final modifier to the corresponding UML/Java attribute**
Elements	FieldDeclaration|Property e1, FieldDeclaration|Property e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, ’Add final modifier to attribute’), modelUndirectedRelatedTo(e1, ’Is-Equivalent-To’, e2))
Action	reportImpact(e1, ’Add final modifier to attribute’, e2)
IR_Att_011	**Add the final modifier to the getter method**
Elements	FieldDeclaration|Property e1, MethodDeclaration|Operation e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, ’Add final modifier to attribute’), modelRelatedTo(e2, ’Uses’, e1))
Action	reportImpact(e1, ’Add final modifier to method’, e2)
IR_Att_012	**Remove the setter method for the now final attribute**
Elements	FieldDeclaration|Property e1, MethodDeclaration|Operation e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, ’Add final modifier to attribute’), modelRelatedTo(e2, ’Modifies’, e1))
Action	reportImpact(e1, ’Delete method’, e2)
IR_Att_013	**Delete code statements that assign values to final attributes**
Elements	FieldDeclaration e1, ExpressionStatement|IfStatement|ForStatement|WhileStatement|DoStatement e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, ’Add final modifier to attribute’), modelRelatedTo(e2, ’Modifies’, e1))
Action	reportImpact(e1, ’Delete statement’, e2)
IR_Att_014	**Delete the final modifier from the corresponding UML/Java attribute**
Elements	FieldDeclaration|Property e1, FieldDeclaration|Property e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, ’Delete final modifier from attribute’), modelUndirectedRelatedTo(e1, ’Is-Equivalent-To’, e2))
Action	reportImpact(e1, ’Delete final modifier from attribute’, e2)
IR_Att_015	**Delete the final modifier from the getter method**
Elements	FieldDeclaration|Property e1, MethodDeclaration|Operation e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, ’Delete final modifier from attribute’), modelRelatedTo(e2, ’Uses’, e1))
Action	reportImpact(e1, ’Delete final modifier from method’, e2)
IR_Att_016	**Add the static modifier to the corresponding UML/Java attribute**
Elements	FieldDeclaration|Property e1, FieldDeclaration|Property e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, ’Add static modifier to attribute’), modelUndirectedRelatedTo(e1, ’Is-Equivalent-To’, e2))
Action	reportImpact(e1, ’Add static modifier to attribute’, e2)
IR_Att_017	**Add the static modifier to the getter and setter methods**
Elements	FieldDeclaration|Property e1, MethodDeclaration|Operation e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, ’Add static modifier to attribute’), OR(modelRelatedTo(e2, ’Uses’, e1), modelRelatedTo(e2, ’Modifies’, e1)))
Action	reportImpact(e1, ’Add static modifier to method’, e2)
IR_Att_018	**Delete the static modifier from the corresponding UML/Java attribute**
Elements	FieldDeclaration|Property e1, FieldDeclaration|Property e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, ’Delete static modifier from attribute’), modelUndirectedRelatedTo(e1, ’Is-Equivalent-To’, e2))
Action	reportImpact(e1, ’Delete static modifier from attribute’, e2)
IR_Att_019	**Delete the static modifier from the getter and setter methods**
Elements	FieldDeclaration|Property e1, MethodDeclaration|Operation e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, ’Delete static modifier from attribute’), OR(modelRelatedTo(e2, ’Uses’, e1), modelRelatedTo(e2, ’Modifies’, e1)))
Action	reportImpact(e1, ’Delete static modifier from method’, e2)
IR_Att_020|21|22	**Change the visibility of the corresponding Java/UML attribute to [public|protected|private]**
Elements	FieldDeclaration|Property e1, FieldDeclaration|Property e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, ’Change attribute visibility to [public|protected|private]’), modelUndirectedRelatedTo(e1, ’Is-Equivalent-To’, e2))
Action	reportImpact(e1, ’Change attribute visibility to [public|protected|private]’, e2)
IR_Att_023|24	**Modify code statements that try to access [protected|private] attributes**
Elements	FieldDeclaration e1, ExpressionStatement|IfStatement|ForStatement|WhileStatement|DoStatement e2, MethodDeclaration e3, ClassDeclaration e4, ClassDeclaration e5, AtomicChangeType e6
Conditions	AND(valueEquals(e6::name, ’Change attribute visibility to [protected|private]’), modelRelatedTo(e3, ’Contains’, e2), modelRelatedTo(e4, ’Defines’, e3), modelRelatedTo(e5, ’Defines’, e1), OR(modelRelatedTo(e2, ’Uses’, e1), modelRelatedTo(e2, ’Modifies’, e1)), NOT(modelEquals(e4, e5)))
Action	reportImpact(e1, ’Modify statement’, e2)
IR_Att_025	**Move the corresponding UML/Java attribute to the corresponding UML/Java class**
Elements	FieldDeclaration|Property e1, FieldDeclaration|Property e2, Class|ClassDeclaration e3, Class|ClassDeclaration e4, CompositeChangeType e5

Conditions	AND(valueEquals(e5::name, 'Move attribute to other class'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2), modelUndirectedRelatedTo(e3, 'Is-Equivalent-To', e4), NOT(modelRelatedTo(e4, 'Defines', e2))
Action	reportImpact(e1, e3 'Move attribute to other class', e2, e4)
IR_Att_026	**Move the getter and setter to the new class**
Elements	FieldDeclaration|Property e1, MethodDeclaration|Operation e2, Class|ClassDeclaration e3, CompositeChangeType e4
Conditions	AND(valueEquals(e4::name, 'Move attribute to other class'), OR(modelRelatedTo(e2, 'Uses', e1), modelRelatedTo(e2, 'Modifies', e1)))
Action	reportImpact(e1, e3 'Move method to other class', e2, e4)
IR_Att_027	**Modify code statements that access attributes that were moved to other classes**
Elements	FieldDeclaration e1, ExpressionStatement|IfStatement|ForStatement|WhileStatement|DoStatement e2, CompositeChangeType e3, ClassDeclaration e4, ClassDeclaration e5
Conditions	AND(valueEquals(e3::name, 'Move attribute to other class'), modelDirectParentOf(e4, e1), OR(modelRelatedTo(e2, 'Uses', e1), modelRelatedTo(e2, 'Modifies', e1)), NOT(modelIndirectlyRelatedTp(e4, 'Is-A', e5)))
Action	reportImpact(e1, e5 'Modify statement', e2)
IR_Att_028	**Move the corresponding UML/Java attribute to the corresponding UML/Java interface**
Elements	FieldDeclaration|Property e1, FieldDeclaration|Property e2, Interface|InterfaceDeclaration e3, Interface|InterfaceDeclaration e4, CompositeChangeType e5
Conditions	AND(valueEquals(e5::name, 'Move attribute to other interface'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2), modelUndirectedRelatedTo(e3, 'Is-Equivalent-To', e4), NOT(modelRelatedTo(e4, 'Defines', e2))
Action	reportImpact(e1, e3 'Move attribute to other interface', e2, e4)
IR_Att_029	**Move the getter and setter to the new interface**
Elements	FieldDeclaration|Property e1, MethodDeclaration|Operation e2, Interface|InterfaceDeclaration e3, CompositeChangeType e4
Conditions	AND(valueEquals(e4::name, 'Move attribute to other interface'), OR(modelRelatedTo(e2, 'Uses', e1), modelRelatedTo(e2, 'Modifies', e1)))
Action	reportImpact(e1, e3 'Move method to other interface', e2, e4)
IR_Att_030	**Modify code statements that access attributes that were moved to other interfaces**
Elements	FieldDeclaration e1, ExpressionStatement|IfStatement|ForStatement|WhileStatement|DoStatement e2, CompositeChangeType e3, InterfaceDeclaration e4, InterfaceDeclaration e5
Conditions	AND(valueEquals(e3::name, 'Move attribute to other interface'), modelDirectParentOf(e4, e1), OR(modelRelatedTo(e2, 'Uses', e1), modelRelatedTo(e2, 'Modifies', e1)), NOT(modelIndirectlyRelatedTp(e4, 'Is-A', e5)))
Action	reportImpact(e1, e5 'Modify statement', e2)
IR_Var_001	**Modify code statements that access deleted variables**
Elements	VariableDeclarationFragment e1, ExpressionStatement|IfStatement|ForStatement|WhileStatement|DoStatement e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete variable'), OR(modelRelatedTo(e2, 'Uses', e1), modelRelatedTo(e2, 'Modifies', e1)))
Action	reportImpact(e1, 'Delete statement', e2)
IR_Var_002	**Modify code statements that access renamed variables**
Elements	VariableDeclarationFragment e1, ExpressionStatement|IfStatement|ForStatement|WhileStatement|DoStatement e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename variable'), OR(modelRelatedTo(e2, 'Uses', e1), modelRelatedTo(e2, 'Modifies', e1)))
Action	reportImpact(e1, 'Modify statement', e2)
IR_Var_003	**Modify code statements that access variables who's value was changed**
Elements	VariableDeclarationFragment e1, ExpressionStatement|IfStatement|ForStatement|WhileStatement|DoStatement e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Change variable value'), OR(modelRelatedTo(e2, 'Uses', e1), modelRelatedTo(e2, 'Modifies', e1)))
Action	reportImpact(e1, 'Modify statement', e2)
IR_Var_004	**Modify code statements that access variables who's data type was changed**
Elements	VariableDeclarationFragment e1, ExpressionStatement|IfStatement|ForStatement|WhileStatement|DoStatement e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Change variable data type'), OR(modelRelatedTo(e2, 'Uses', e1), modelRelatedTo(e2, 'Modifies', e1)))
Action	reportImpact(e1, 'Modify statement', e2)
IR_Var_005	**Delete code statements that change final variables**
Elements	VariableDeclarationFragment e1, ExpressionStatement|IfStatement|ForStatement|WhileStatement|DoStatement e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Add final modifier to variable'), modelRelatedTo(e2, 'Modifies', e1))
Action	reportImpact(e1, 'Delete statement', e2)
IR_Seq_001	**Delete the use case that is realized by a sequence**
Elements	Interaction e1, UseCase e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete sequence'), modelRelatedTo(e1, 'Realizes', e2))
Action	reportImpact(e1, 'Delete use case', e2)
IR_Seq_002|3	**Delete the [statemachine|activity] that overlaps with a sequence**
Elements	Interaction e1, StateMachine|Activity e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete sequence'), modelUndirectedRelatedTo(e1, 'Overlaps-With', e2))

Action	reportImpact(e1, 'Delete [state machine|activity]', e2)
IR_Seq_004	**Rename the use case that is realized by a sequence**
Elements	Interaction e1, UseCase e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename sequence'), modelRelatedTo(e1, 'Realizes', e2))
Action	reportImpact(e1, 'Rename use case', e2)
IR_Seq_005|6	**Rename the [statemachine|activity] that overlaps with a sequence**
Elements	Interaction e1, StateMachine|Activity e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename sequence'), modelUndirectedRelatedTo(e1, 'Overlaps-With', e2))
Action	reportImpact(e1, 'Rename [state machine|activity]', e2)
IR_Seq_007	**Split the use case that is realized by a sequence**
Elements	Interaction e1, UseCase e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split sequence'), modelRelatedTo(e1, 'Realizes', e2))
Action	reportImpact(e1, 'Split use case', e2)
IR_Seq_008|9	**Split the [statemachine|activity] that overlaps with a sequence**
Elements	Interaction e1, StateMachine|Activity e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split sequence'), modelUndirectedRelatedTo(e1, 'Overlaps-With', e2))
Action	reportImpact(e1, 'Split [state machine|activity]', e2)
IR_Seq_010	**Merge the use cases that are realized by merged sequences**
Elements	Interaction e1, Interaction e2, UseCase e3, UseCase e4, CompositeChangeType e5
Conditions	AND(valueEquals(e5::name, 'Merge sequences'), modelRelatedTo(e1, 'Realizes', e3), modelRelatedTo(e2, 'Realizes', e4))
Action	reportImpact(e1, 'Merge use cases', e2)
IR_Seq_011|12	**Merge the [statemachines|activities] that overlap with merged sequences**
Elements	Interaction e1, Interaction e2, StateMachine|Activity e3, StateMachine|Activity e4, CompositeChangeType e5
Conditions	AND(valueEquals(e5::name, 'Merge sequences'), modelUndirectedRelatedTo(e1, 'Overlaps-With', e3), modelUndirectedRelatedTo(e2, 'Overlaps-With', e4))
Action	reportImpact(e1, 'Merge [state machines|activities]', e2)
IR_Lin_001|2	**Delete [classes|components] that are equivalent to deleted lifelines**
Elements	Lifeline e1, Class|ClassDeclaration|Component e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete lifeline'), modelRelatedTo(e1, 'Is-Instance-Of', e2))
Action	reportImpact(e1, 'Delete [class|component]', e2)
IR_Lin_003|4	**Rename [classes|components] that are equivalent to renamed lifelines**
Elements	Lifeline e1, Class|ClassDeclaration|Component e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename lifeline'), modelRelatedTo(e1, 'Is-Instance-Of', e2))
Action	reportImpact(e1, 'Rename [class|component]', e2)
IR_Lin_005|6	**Merge [classes|components] that are equivalent to merged lifelines**
Elements	Lifeline e1, Lifeline e2, Class|ClassDeclaration|Component e3, Class|ClassDeclaration|Component e4, CompositeChangeType e5
Conditions	AND(valueEquals(e5::name, 'Merge lifelines'), modelRelatedTo(e1, 'Is-Instance-Of', e3), modelRelatedTo(e2, 'Is-Instance-Of', e4))
Action	reportImpact(e1, 'Merge [classes|componentes]', e2)
IR_Lin_007|8	**Split [classes|components] that are equivalent to split lifelines**
Elements	Lifeline e1, Class|ClassDeclaration|Component e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split lifeline'), modelRelatedTo(e1, 'Is-Instance-Of', e2))
Action	reportImpact(e1, 'Split [class|component]', e2)
IR_Msg_001|2|3	**Delete the [method|activity|activity node] that is equivalent to a deleted message**
Elements	Message e1, Operation|Activity|CallBehaviorAction e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete message'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2))
Action	reportImpact(e1, 'Delete [method|activity|activity node]', e2)
IR_Msg_004|5|6	**Rename the [method|activity|activity node] that are equivalent to a deleted message**
Elements	Message e1, Operation|Activity|CallBehaviorAction e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename message'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2))
Action	reportImpact(e1, 'Rename [method|activity|activity node]', e2)
IR_Msg_007|8|9	**Merge the [methods|activities|activity nodes] that are equivalent to merged messages**
Elements	Message e1, Message e2, Operation|Activity|CallBehaviorAction e3, Operation|Activity|CallBehaviorAction e4, CompositeChangeType e5
Conditions	AND(valueEquals(e5::name, 'Merge message'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e3), modelUndirectedRelatedTo(e2, 'Is-Equivalent-To', e4))
Action	reportImpact(e1, 'Rename [method|activity|activity node]', e2)
IR_Msg_010|11|12	**Split the [methods|activities|activity nodes] that are equivalent to split messages**
Elements	Message e1, Operation|Activity|CallBehaviorAction e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split message'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2))
Action	reportImpact(e1, 'Split [method|activity|activity node]', e2)
IR_Act_001|2	**Delete the [state machine|sequence] that overlaps with a deleted activity**
Elements	Activity e1, StateMachine|Interaction e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete activity'), modelUndirectedRelatedTo(e1, 'Overlaps-With', e2))
Action	reportImpact(e1, 'Delete [state machine|sequence]', e2)
IR_Act_003	**Delete the use case that is realized by deleted activity**
Elements	Activity e1, UseCase e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete activity'), modelRelatedTo(e1, 'Realizes', e2))

Action	reportImpact(e1, 'Delete use case', e2)
IR_Act_004|5	**Rename the [state machine|sequence] that overlaps with a deleted activity**
Elements	Activity e1, StateMachine|Interaction e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename activity'), modelUndirectedRelatedTo(e1, 'Overlaps-With', e2))
Action	reportImpact(e1, 'Rename [state machine|sequence]', e2)
IR_Act_006	**Rename the use case that is realized by deleted activity**
Elements	Activity e1, UseCase e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename activity'), modelRelatedTo(e1, 'Realizes', e2))
Action	reportImpact(e1, 'Rename use case', e2)
IR_Act_007|8	**Split the [state machine|sequence] that overlaps with a split activity**
Elements	Activity e1, StateMachine|Interaction e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split activity'), modelUndirectedRelatedTo(e1, 'Overlaps-With', e2))
Action	reportImpact(e1, 'Split [state machine|sequence]', e2)
IR_Act_009	**Split the use case that is realized by the merged activities**
Elements	Activity e1, UseCase e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split activity'), modelRelatedTo(e1, 'Realizes', e2))
Action	reportImpact(e1, 'Split use case', e2)
IR_Act_010|11	**Merge the [state machines|sequences] that overlap with merged activities**
Elements	Activity e1, Activity e2, StateMachine|Interaction e3, StateMachine|Interaction e4, CompositeChangeType e5
Conditions	AND(valueEquals(e5::name, 'Merge activities'), modelUndirectedRelatedTo(e1, 'Overlaps-With', e3), modelUndirectedRelatedTo(e2, 'Overlaps-With', e4))
Action	reportImpact(e1, e2, 'Merge [state machines|sequences]', e3, e4)
IR_Act_012	**Merge the use case that is realized by the merged activities**
Elements	Activity e1, Activity e2, UseCase e3, UseCase e4, CompositeChangeType e5
Conditions	AND(valueEquals(e3::name, 'Split activity'), modelRelatedTo(e1, 'Realizes', e3), modelRelatedTo(e2, 'Realizes', e4))
Action	reportImpact(e1, e2, 'Merge use cases', e3, e4)
IR_Act_013	**Delete the method that is equivalent to a deleted activity node**
Elements	CallBehaviorAction e1, Operation e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete activity node'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2))
Action	reportImpact(e1, 'Delete method', e2)
IR_Act_014	**Delete the activity node that refines a deleted activity node**
Elements	CallBehaviorAction e1, CallBehaviorAction e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete activity node'), modelUndirectedRelatedTo(e1, 'Refines', e2))
Action	reportImpact(e1, 'Delete activity node', e2)
IR_Act_015	**Rename the method that is equivalent to a renamed activity node**
Elements	CallBehaviorAction e1, Operation e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename activity node'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2))
Action	reportImpact(e1, 'Rename method', e2)
IR_Act_016	**Rename the activity node that refines a renamed activity node**
Elements	CallBehaviorAction e1, CallBehaviorAction e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename activity node'), modelUndirectedRelatedTo(e1, 'Refines', e2))
Action	reportImpact(e1, 'Rename activity node', e2)
IR_Act_017	**Merge the methods that are equivalent to merged activity nodes**
Elements	CallBehaviorAction e1, CallBehaviorAction e2, Operation e3, Operation e4, CompositeChangeType e5
Conditions	AND(valueEquals(e5::name, 'Merge activity nodes'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e3), modelUndirectedRelatedTo(e2, 'Is-Equivalent-To', e4))
Action	reportImpact(e1, e2, 'Merge methods', e3, e4)
IR_Act_018	**Merge the methods that are equivalent to merged activity nodes**
Elements	CallBehaviorAction e1, CallBehaviorAction e2, CallBehaviorAction e3, CallBehaviorAction e4, CompositeChangeType e5
Conditions	AND(valueEquals(e5::name, 'Merge activity nodes'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e3), modelUndirectedRelatedTo(e2, 'Is-Equivalent-To', e4))
Action	reportImpact(e1, 'Merge methods', e2)
IR_Act_019	**Split the method that is equivalent to a split activity node**
Elements	CallBehaviorAction e1, Operation e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split activity node'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2))
Action	reportImpact(e1, 'Split method', e2)
IR_Act_020	**Split the activity node that is equivalent to a split activity node**
Elements	CallBehaviorAction e1, Operation e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split activity node'), modelUndirectedRelatedTo(e1, 'Refines', e2))
Action	reportImpact(e1, 'Split activity node', e2)
IR_Act_021	**Delete code statements that are equivalent to deleted decision nodes**
Elements	DecisionNode e1, IfStatement|ForStatement|WhileStatement|DoStatement e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete decision node'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2))
Action	reportImpact(e1, 'Delete statement', e2)
IR_Act_022	**Modify code statements that are equivalent to modified decision nodes**
Elements	DecisionNode e1, IfStatement|ForStatement|WhileStatement|DoStatement e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Change decision node condition'), modelUndirectedRelatedTo(e1, 'Is-Equivalent-To', e2))

Action	reportImpact(e1, 'Modify statement', e2)
IR_Stm_001|3	**Delete [activities|sequences] that overlap with deleted state machines**
Elements	StateMachine e1, Activity|Interaction e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete state machine'), modelUndirectedRelatedTo(e1, 'Overlaps-With', e2))
Action	reportImpact(e1, 'Delete [activity|sequence]', e2)
IR_Stm_002	**Delete use cases that are realized by a deleted state machine**
Elements	StateMachine e1, UseCase e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete state machine'), modelRelatedTo(e2, 'Realizes', e1))
Action	reportImpact(e1, 'Delete use case', e2)
IR_Stm_004|5	**Rename [activities|sequences] that overlap with a renamed state machine**
Elements	StateMachine e1, Activity|Interaction e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename state machine'), modelUndirectedRelatedTo(e1, 'Overlaps-With', e2))
Action	reportImpact(e1, 'Rename [activity|sequence]', e2)
IR_Stm_006	**Rename use cases that are realized by a renamed state machine**
Elements	StateMachine e1, UseCase e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Rename state machine'), modelRelatedTo(e2, 'Realizes', e1))
Action	reportImpact(e1, 'Rename use case', e2)
IR_Stm_007|8	**Split [activities|sequences] that overlap with a split state machine**
Elements	StateMachine e1, Activity|Interaction e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split state machine'), modelUndirectedRelatedTo(e1, 'Overlaps-With', e2))
Action	reportImpact(e1, 'Split [activity|sequence]', e2)
IR_Stm_009	**Split use cases that are realized by a split state machine**
Elements	StateMachine e1, UseCase e2, CompositeChangeType e3
Conditions	AND(valueEquals(e3::name, 'Split state machine'), modelRelatedTo(e2, 'Realizes', e1))
Action	reportImpact(e1, 'Split use case', e2)
IR_Stm_010|11	**Merge [activities|sequences] that overlap with merged state machines**
Elements	StateMachine e1, StateMachine e2, Activity|Interaction e3, Activity|Interaction e4, CompositeChangeType e5
Conditions	AND(valueEquals(e5::name, 'Merge state machines'), modelUndirectedRelatedTo(e1, 'Overlaps-With', e3), modelUndirectedRelatedTo(e2, 'Overlaps-With', e4))
Action	reportImpact(e1, e2, 'Merge [activities|sequences]', e3, e4)
IR_Stm_012	**Merge use cases that are realized by merged state machines**
Elements	StateMachine e1, StateMachine e2, UseCase e3, UseCase e4, CompositeChangeType e5
Conditions	AND(valueEquals(e5::name, 'Merge state machines'), modelRelatedTo(e1, 'Realizes', e3), modelRelatedTo(e2, 'Realizes', e4))
Action	reportImpact(e1, e2, 'Merge use cases', e3, e4)
IR_Src_001	**Move variables to other methods if the code-statements accessing them were moved**
Elements	ExpressionStatement|IfStatement|ForStatement|WhileStatement|DoStatement|VariableDeclarationStatement e1, VariableDeclarationFragment e2, MethodDeclaration e3, CompositeChangeType e4, ImpactReport e5
Conditions	AND(valueEquals(e4::name, 'Move statement to other method'), OR(modelRelatedTo(e1, 'Modifies', e2), modelRelatedTo(e1, 'Uses', e2)), NOT(valueEquals(e5::changetype::name, 'Move variable to other method'), modelEquals(e5::impactsources, e2), modelEquals(e5::impactsources, e3)))
Action	reportImpact(e1, e3, 'Move variable to other method', e2, e3)
IR_Src_002	**Move attributes to other classes if code-statements using them were moved to methods of other classes**
Elements	ExpressionStatement|IfStatement|ForStatement|WhileStatement|DoStatement|VariableDeclarationStatement e1, FieldDeclaration e2, MethodDeclaration e3, ClassDeclaration e4, CompositeChangeType e5, ImpactReport e6
Conditions	AND(valueEquals(e5::name, 'Move statement to other method'), modelDirectParentOf(e4, e3), NOT(modelDirectParentOf(e4, e2)), NOT(valueEquals(e6::changetype, 'Move attribute to other class'), modelE-quals(e6::impactsources, e2), modelEquals(e6::impactsources, e4)))
Action	reportImpact(e1, e3, 'Move variable to other method', e2, e4)
IR_Src_003	**Delete variables if the statements declaring them were deleted**
Elements	VariableDeclarationStatement e1, VariableDeclarationStatement e2, AtomicChangeType e3
Conditions	AND(valueEquals(e3::name, 'Delete statement'), modelRelatedTo(e1, 'Defines', e2))
Action	reportImpact(e1, 'Delete variable', e2)

C. Evaluation Material

C.1. Case Study Data

Due to the size of all the case study data it is not attached to this thesis in a printed form. Instead, the source code of the EMFTrace application can be obtained from the project's website. The original version of EMFTrace before the application of the change scenarios is located here:
`https://sourceforge.net/p/emftrace/code/HEAD/tree/tags/EMFTrace_0.3.2/`

The revised version after applying the changes is located here:
`https://sourceforge.net/p/emftrace/code/HEAD/tree/tags/EMFTrace_0.4.0/`

C.2. Scenario 1: Refactoring of the Impact Analyzer Components

S1.C001: add("AbstractImpactAnalyzer", "impactanalyzer")

S1.C002: update("AbstractImpactAnalyzer::implementedinterface", "IImpactAnalyzer")

S1.C003: update("AbstractImpactAnalyzer::superclass", "TraceComponent")

S1.C004: update("TypeBasedImpactAnalyzer::superclass", "AbstractImpactAnalyzer")

S1.C005: update("DistanceBasedImpactAnalyzer::superclass", "AbstractImpactAnalyzer")

S1.C006: update("TypeBasedImpactAnalyzer::implementedinterface", "null")

S1.C007: update("DistanceBasedImpactAnalyzer::implementedinterface", "null")

S1.C008: move("TypedBasedImpactAnalyzer::reportManager", "AbstractImpactAnalyzer")

S1.C009: delete("DistanceBasedImpactAnalyzer::reportManager")

S1.C010: move("TypedBasedImpactAnalyzer::disconnectReportManager", "AbstractImpactAnalyzer")

S1.C011: delete("DistanceBasedImpactAnalyzer::disconnectReportManager")

S1.C012: move("TypedBasedImpactAnalyzer::registerReportManager", "AbstractImpactAnalyzer")

S1.C013: delete("DistanceBasedImpactAnalyzer::registerReportManager")

S1.C014: add("AbstractImpactAnalyzerTest", "impactanalyzer")

S1.C015: update("AbstractImpactAnalyzerTest::superclass", "EMFTraceBaseTest")

S1.C016: add("testRegisterReportManager", "AbstractImpactAnalyzerTest")

S1.C017: add("testDisconnectReportManager", "AbstractImpactAnalyzerTest")

C.3. Scenario 2: Extraction of a Cache Component

S2.C001: add("ModelElementCache", "accesslayer")

S2.C002: move("AccessLayer::modelCacheHeader", "ModelElementCache")

S2.C003: move(“AccessLayer::modelCacheTable”, “ModelElementCache”)

S2.C004: move(“AccessLayer::projects”, “ModelElementCache”)

S2.C005: delete(“AccessLayer::dirtyFlags”)

S2.C006: add(“insert”, “ModelElementCache”)

S2.C007: add(“get”, “ModelElementCache”)

S2.C008: add(“remove”, “ModelElementCache”)

S2.C009: add(“clear”, “ModelElementCache”)

S2.C010: add(“clearEntireCache”, “ModelElementCache”)

S2.C011: add(“containsProject”, “ModelElementCache”)

S2.C012: add(“addProject”, “ModelElementCache”)

S2.C013: add(“getProjects”, “ModelElementCache”)

S2.C014: add(“modelCache”, “AccessLayer”)

Changes applied on *AccessLayer::invalidateCache()*:

S2.C015: move(“if(projectIdx <modelCacheTable.size()) modelCacheTable.get(projectIdx).clear();”, “ModelElementCache::clear”)

S2.C016: move(“if(projectIdx <modelCacheHeader.size()) modelCacheHeader.get(projectIdx).clear();”, “ModelElementCache::clear”)

S2.C017: add(“ModelElementCacheTest”, “accesslayer”)

S2.C018: update(“ModelElementCacheTest::superclass”, “EMFTraceBaseTest”)

S2.C019: add(“testInsert”, “ModelElementCacheTest”)

S2.C020: add(“testGet”, “ModelElementCacheTest”)

S2.C021: add(“testRemove”, “ModelElementCacheTest”)

S2.C022: add(“testClear”, “ModelElementCacheTest”)

S2.C023: add(“testClearEntireCache”, “ModelElementCacheTest”)

S2.C024: add(“testAddProject”, “ModelElementCacheTest”)

S2.C025: add(“testGetProjects”, “ModelElementCacheTest”)

S2.C026: add(“testContainsProject”, “ModelElementCacheTest”)

Changes applied on *AccessLayer::init()*:

S2.C027: move(“projects = new ArrayList<Project>();”, “ModelElementCache()”)

S2.C028: delete(“dirtyFlags = new ArrayList<Boolean>();”)

S2.C029: move(“modelCacheTable = new ArrayList<ArrayList<ArrayList<EObject>>>();”, “ModelElementCache()”)

S2.C030: move(“modelCacheHeader = new ArrayList<ArrayList<String>>();”, “ModelElementCache()”)

S2.C031: delete(“projects.clear();”)

S2.C032: delete(“dirtyFlags.clear();”)

S2.C033: delete(“projectSpaces.clear();”)

S2.C034: delete(“modelCacheTable.clear();”)

S2.C035: delete(“modelCacheHeader.clear();”)

Changes applied on *AccessLayer::getElements()*:

S2.C036: replace(“if(!projects.contains(project))”, “if(!cache.containsProject(project))”)

S2.C037: replace(“projects.add(project);”, “cache.addProject(project);”)

S2.C038: add(“cache.insert(project, element);”)

S2.C039: delete(“dirtyFlags.add(true);”)

S2.C040: delete(“modelCacheTable.add(new ArrayList<ArrayList<EObject>>());”)

S2.C041: delete(“modelCacheHeader.add(new ArrayList<String>());”)

S2.C042: delete(“int idx = projects.indexOf(project);”)

S2.C043: delete(“modelCacheTable.get(idx).add(new ArrayList<EObject>());”)

S2.C044: delete(“modelCacheTable.get(idx).get(0).add(element);”)

S2.C045: delete(“modelCacheHeader.get(idx).add(element.eClass().getName());”)

S2.C046: delete(“int projectIdx = projects.indexOf(project);”)

S2.C047: delete(“int modelIdx = modelCacheHeader.get(projectIdx).indexOf(classname);”)

S2.C048: delete(“if(modelIdx != -1)”)

S2.C049: delete(“return modelCacheTable.get(projectIdx).get(modelIdx);”)

S2.C050: delete(“modelCacheHeader.get(projectIdx).add(classname);”)

S2.C051: delete(“modelIdx = modelCacheHeader.get(projectIdx).indexOf(classname);”)

S2.C052: delete(“modelCacheTable.get(projectIdx).add(new ArrayList<EObject>());”)

S2.C053: replace(“modelCacheTable.get(projectIdx).get(modelIdx).addAll(result);”, “cache.insert(project, list.get(i));”)

Changes applied on *AccessLayer::notifyProject()*:

S2.C054: replace(“if(!projects.contains(project))”, “if(!cache.containsProject(project))”)

S2.C055: replace(“projects.add(project);”, “cache.addProject(project);”)

S2.C056: delete(“dirtyFlags.add(true)”)

S2.C057: delete(“modelCacheTable.add(new ArrayList<ArrayList<EObject>>());”)

S2.C058: delete(“modelCacheHeader.add(new ArrayList<String>());”)

S2.C059: delete(“int idx = projects.indexOf(project);”)

S2.C060: delete(“modelCacheTable.get(idx).add(new ArrayList<EObject>());”)

S2.C061: delete(“modelCacheTable.get(idx).get(0).add(element);”)

S2.C062: delete(“modelCacheHeader.get(idx).add(element.eClass().getName());”)

S2.C063: delete(“int projectIdx = projects.indexOf(project);”)

S2.C064: delete(“dirtyFlags.set(projectIdx, true);”)

S2.C065: delete(“int modelIdx = modelCacheHeader.get(projectIdx).indexOf(element.eClass().getName());”)

S2.C066: delete(“if(modelIdx != -1)”)

S2.C067: replace(“if(delete) modelCacheTable.get(projectIdx).get(modelIdx).remove(element);”,
“if(delete) cache.remove(project, element);”)

S2.C068: replace(“else modelCacheTable.get(projectIdx).get(modelIdx).add(element);”, “else cache.insert(project, element);”)

S2.C069: delete(“modelCacheHeader.get(projectIdx).add(element.eClass().getName());”)

S2.C070: delete(“modelIdx = modelCacheHeader.get(projectIdx).indexOf(element.eClass().getName());”)

S2.C071: delete(“modelCacheTable.get(projectIdx).add(new ArrayList<EObject>());”)

S2.C072: delete(“modelCacheTable.get(projectIdx).get(modelIdx).add(element);”)

C.4. Scenario 3: Replacement of the Logging Features

S3.C001: add(“java.util.logging.Level”, “TraceComponent”)

S3.C002: add(“java.util.logging.Logger”, “TraceComponent”)

S3.C003: add(“logger”, “TraceComponent”)

S3.C004: replace(“System.out.println(logEntry);”, “logger.log(Level.INFO, logEntry.toString());”)

S3.C005: replace(“System.out.println(logEntry);”, “logger.log(Level.INFO, logEntry.toString());”)

S3.C006: replace(“System.out.println(logEntry);”, “logger.log(Level.INFO, logEntry.toString());”)

S3.C007: replace(“System.out.println(logEntry);”, “logger.log(Level.INFO, logEntry.toString());”)

C.5. Scenario 4: Migration to EMFStore/ECP 1.2.x

S4.C001: delete(“removeElement”, “IAccessLayer”)

S4.C002: delete(“getElement”, “IAccessLayer”)

S4.C003: delete(“registerProjectSpace”, “IAccessLayer”)

S4.C004: update(“IAccessLayer::addElement::project::type”, “ECPProject”)

S4.C005: update(“IAccessLayer::removeElement::project::type”, “ECPProject”)

S4.C006: update(“IAccessLayer::getElements::project::type”, “ECPProject”)

S4.C007: update(“IAccessLayer::getElements::project::type”, “ECPProject”)

S4.C008: update(“IAccessLayer::getAllElements::project::type”, “ECPProject”)

S4.C009: update(“IAccessLayer::invalidateCache::project::type”, “ECPProject”)

S4.C010: update(“IAccessLayer::notifyProject::project::type”, “ECPProject”)

S4.C011: update(“IAccessLayer::notifyProject::project::type”, “ECPProject”)

S4.C012: update(“IAccessLayer::commitProject::project::type”, “ECPProject”)

S4.C013: delete(“projectSpaces”, “AccessLayer”)

S4.C014: update(“AccessLayer::projects::type”, “ArrayList<ECPProject>”)

Changes applied on *AccessLayer::addElement()*:

S4.C015: replace(“project.addModelElement(element);”, “project.getContents().add(element);”)

Changes applied on *AccessLayer::commitProject()*:

S4.C016: replace(“project == null || !projects.contains(project)) return;”, “if(project != null && project.hasDirtyContents())”)

S4.C017: delete(“int index = projects.indexOf(project);”)

S4.C018: replace(“projectSpaces.get(index).commit(null, null, null);”, “project.saveContents();”)

Changes applied on *AccessLayer::commitProjects()*:

S4.C019: replace(“for(int i = 0; i <projectSpaces.size(); i++)”, “for(int i = 0; i <projects.size(); i++)”)

S4.C020: replace(“projectSpaces.get(i).commit(null, null, null);”, “projects.get(i).saveContents();”)

Changes applied on *AccessLayer::getAllChildren()*:

S4.C021: replace(“TreeIterator<EObject >it = element.eAllContents();”, “result.addAll(element.eContents());”)

S4.C022: replace(“while(it.hasNext())”, “for(int i = 0; i <element.eContents().size(); i++)”)

S4.C023: replace(“*while*-body”, “result.addAll(getAllChildren(element.eContents().get(i)));”)

S4.C024: update(“IImpactAnalyzer::performImpactAnalysis::project::type”, “ECPProject”)

S4.C025: update(“TypeBasedImpactAnalyzer::addToClosedListAndPrepareNewSIS::project::type”, “ECPProject”)

S4.C026: update(“TypeBasedImpactAnalyzer::getChangeType::project::type”, “ECPProject”)

S4.C027: delete(“ILinkManager::deleteTrace”)

S4.C028: delete(“ILinkManager::validateTrace”)

S4.C029: delete(“ILinkManager::validateLink”)

S4.C030: update(“ILinkManager::checkIfLinkExists::project::type”, “ECPProject”)

S4.C031: update(“ILinkManager::createLink::project::type”, “ECPProject”)

S4.C032: update(“ILinkManager::deleteLink::project::type”, “ECPProject”)

S4.C033: update(“ILinkManager::createTrace::project::type”, “ECPProject”)

S4.C034: update(“ILinkManager::deleteTrace::project::type”, “ECPProject”)

S4.C035: update(“ILinkManager::validateLink::project::type”, “ECPProject”)

S4.C036: update(“ILinkManager::validateTrace::project::type”, “ECPProject”)

S4.C037: update(“ILinkManager::addToTrace::project::type”, “ECPProject”)

S4.C038: update(“ILinkManager::removeFromTrace::project::type”, “ECPProject”)

S4.C039: update("ILinkManager::performTransitivityAnalysis::project::type", "ECPProject")

S4.C040: update("IProjectCleaner::cleanUpProject::project::type", "ECPProject")

S4.C041: update("IProjectCleaner::cleanUpRuleOrphans::project::type", "ECPProject")

S4.C042: update("IProjectCleaner::cleanUpLinkTypeOrphans::project::type", "ECPProject")

S4.C043: update("IProjectCleaner::cleanUpViolationTypeOrphans::project::type", "ECPProject")

S4.C044: update("IProjectCleaner::cleanUpChangeTypeOrphans::project::type", "ECPProject")

S4.C045: update("IProjectCleaner::updateLinkTypeCatalogs::project::type", "ECPProject")

S4.C046: update("IProjectCleaner::updateViolationTypeCatalogs::project::type", "ECPProject")

S4.C047: update("IProjectCleaner::updateRuleCatalogs::project::type", "ECPProject")

S4.C048: update("IProjectCleaner::updateChangeTypeCatalogs::project::type", "ECPProject")

S4.C049: update("IProjectCleaner::updateLinkContainer::project::type", "ECPProject")

S4.C050: update("IProjectCleaner::updateReportContainer::project::type", "ECPProject")

S4.C051: update("IReportManager::createImpactReport::project::type", "ECPProject")

S4.C052: update("IReportManager::createConsistenceReport::project::type", "ECPProject")

S4.C053: update("IReportManager::checkIfImpactReportAlreadyExists::project::type", "ECPProject")

S4.C054: update("IReportManager::checkIfConsistenceReportAlreadyExists::project::type", "ECPProject")

S4.C055: update("EMFTraceBaseTest::project::type", "ECPProject")

S4.C056: delete("EMFTraceBaseTest::projectSpace")

S4.C057: update("IProcessingComponent::run::project::type", "ECPProject")

S4.C058: update("IResultProcessor::processCreateLinkResult::project::type", "ECPProject")

S4.C059: update("IResultProcessor::processReportViolationResult::project::type", "ECPProject")

S4.C060: update("IResultProcessor::processReportImpactResult::project::type", "ECPProject")

S4.C061: update("IRuleEngine::applyRule::project::type", "ECPProject")

S4.C062: update("IRuleValidator::checkActionDefinition::project::type", "ECPProject")

S4.C063: update("IRuleValidator::validateRule::project::type", "ECPProject")

S4.C064: update("JoinProcessor::run::project::type", "ECPProject")

S4.C065: update("IElementProcessor::retrieveElements::project::type", "ECPProject")

S4.C066: update("IElementProcessor::retrieveElements::project::type", "ECPProject")

S4.C067: update("IElementProcessor::run::project::type", "ECPProject")

S4.C068: update("ConditionProcessor::project::type", "ECPProject")

S4.C069: update("IElementProcessor::run::project::type", "ECPProject")

Changes applied on *ElementProcessor::retrieveElements()*:

S4.C070: replace("list = new ArrayList<EObject>(project.getAllModelElements())",
"list = new ArrayList<EObject>(accessLayer.getAllElements(project))")

C.6. Scenario 5: Miscellaneous Changes

Component-changes:

S5.C001: update("QueryOptimizer::name", "RuleOptimizer")

S5.C002: delete("QueryOptimizer")

S5.C003: add("FastOptimizer", "QueryOptimizer")

S5.C004: update("QueryOptimizer::providedinterface", "IFastOptimizer")

S5.C005: update("QueryOptimizer::requiredinterface", "IFastOptimizer")

S5.C006: delete("IQueryOptimizer")

S5.C007: split("QueryOptimizer", "BaseConditionOptimizer", "LogicConditionOptimizer")

S6.C008: merge(“QueryOptimizer”, “RuleValidator”)

S5.C009: move(“QueryOptimizer”, “RuleValidator)

Package-changes:

S5.C010: update(“AccessLayer::name”, “RepositoryLayer”)

S5.C011: add(“TestPackage”, “AccessLayer”)

S5.C012: delete(“AccessLayer”)

S5.C013: add(“AccessLayerChangeListener”, “AccessLayer”)

S5.C014: delete(“AccessLayerImpl”, “AccessLayer”)

S5.C015: split(“AccessLayer”, “AccessLayer”, “AccessLayerTest”)

S5.C016: merge(“LinkManager”, “ProjectCleaner”)

Class/Interface-changes:

S5.C017: update(“ITraceComponent::name”, “EMFTraceBaseComponent”)

S5.C018: add(“dumpComponentStats”, “ITraceComponent”)

S5.C019: delete(“disconnectAccessLayer”, “ITraceComponent”)

S5.C020: add(“msgBuffer”, “TraceComponent”)

S5.C021: delete(“accessLayer”, “TraceComponent”)

S5.C022: update(“ElementProcessor::ImplementedInterface”, “IProcessingComponent”)

S5.C023: update(“ElementProcessor::SuperClass”, “Object”)

S5.C024: split(“TraceComponent”, “BaseComponent”, “LogComponent”)

S5.C025: move(“TraceComponent”, “accessLayer”)

S5.C026: merge(“QueryOptimizer”, “RuleValidator”)

Method-changes:

S5.C027: update(“ITraceComponent::getName::type”, “ComponentID”)

S5.C028: update(“ITraceComponent::getName::final”, “true”)

S5.C029: update(“ITraceComponent::getName::static”, “true”)

S5.C030: add(“newParameter”, “ITraceComponent::getName”)

S5.C031: delete(“status”, “ITraceComponent::enableLogging”)

S5.C032: update(“ITraceComponent::getName::visibility”, “private”)

S5.C033: update(“ITraceComponent::getName::name”, “getComponentName”)

S5.C034: move(“ITraceComponent::getName”, “ILinkManager”)

Parameter-changes:

S5.C035: update(“IAccessLayer::getParent::element::type”, “Object”)

S5.C036: update(“IAccessLayer::getParent::element::final”, “true”)

S5.C037: update(“IAccessLayer::getParent::element::static”, “true”)

S5.C038: update(“IAccessLayer::getParent::element::name”, “modelElement”)

Attribute-changes:

S5.C039: update(“TraceComponent::componentName::type”, “ComponentIdentifier”)

S5.C040: update(“TraceComponent::componentName::final”, “true”)

S5.C041: update(“TraceComponent::componentName::static”, “true”)

S5.C042: update(“TraceComponent::componentName::visibility”, “private”)

S5.C043: update(“TraceComponent::componentName::name”, “componentID”)

S5.C044: move(“TraceComponent::componentName”, “LinkManager”)

UseCase-changes:

S5.C045: update(“LinkManager::name”, “DependencyManager”)

S5.C046: update(“Create trace::name”, “Create transitive dependency relations”)

S5.C047: delete(“LinkManager”)

S5.C048: delete(“Create trace”)

Eidesstattliche Versicherung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Dissertationsschrift selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.
I hereby declare, on oath, that I have written the present dissertation by my own and have not used other than the acknowledged resources and aids.

Hamburg, den	Unterschrift
city and date	*signature*