TOWARDS A HOLISTIC FRAMEWORK FOR SOFTWARE ARTEFACT CONSISTENCY MANAGEMENT

Ildiko Pete

**A Thesis Submitted for the Degree of PhD
at the
University of St Andrews**

2017

Towards a Holistic Framework for Software Artefact Consistency Management

Ildiko Pete

University of St Andrews

This thesis is submitted in partial fulfilment for the degree of

Doctor of Philosophy

at the University of St Andrews

October 2016

Abstract

A software system is represented by different software artefacts ranging from requirements specifications to source code. As the system evolves, artefacts are often modified at different rates and times resulting in inconsistencies, which in turn can hinder effective communication between stakeholders, and the understanding and maintenance of systems. The problem of the differential evolution of heterogeneous software artefacts has not been sufficiently addressed to date as current solutions focus on specific sets of artefacts and aspects of consistency management and are not fully automated. This thesis presents the concept of holistic artefact consistency management and a proof-of-concept framework, ACM, which aim to support the consistent evolution of heterogeneous software artefacts while minimising the impact on user choices and practices and maximising automation. The ACM framework incorporates traceability, change impact analysis, change detection, consistency checking and change propagation mechanisms and is designed to be extensible. The thesis describes the design, implementation and evaluation of the framework, and an approach to automate trace link creation using machine learning techniques. The framework evaluation uses six open source systems and suggests that managing the consistency of heterogeneous artefacts may be feasible in practical scenarios.

Acknowledgements

Firstly, I would like to express my sincere gratitude to my supervisor, Dharini Balasubramaniam. You gave me the opportunity to challenge and improve myself, to focus my attention on exploring a problem, through which I did not only develop professional and technical skills; it has also shaped my world view and how I approach things in life. This has been one of the most valuable lessons. Thank you for your support and for providing me guidance throughout my time working with you. I would also like to say thank you to my secondary supervisor Juliana Bowles, and John Thomson, Alex Voss, Özgür Akgün, and Peter Nightingale, for their invaluable help.

I am fortunate to be surrounded by the most precious and supportive friends who have closely followed my work and helped me overcome obstacles by being there, giving me advice, and sharing their thoughts. My deepest thanks to Lakshitha de Silva, Chris Schneider, Percy Perez, Erőss Andrea, Bíró Dániel, Catherine Hollebrandse, Wagner da Silveira, Farnaz Ghajeri, Juan Mendoza Santana and Marco Caminati.

I would not be who I am and where I am without my parents' support, to whom I am eternally grateful. Szüleimnek: Hálásan köszönöm a sok segítséget, támogatást, iránymutatást, és a belém vetett fáradhatatlan hitet.

Declaration

Candidate's Declarations

I, Ildiko Pete, hereby certify that this thesis, which is approximately 55000 words in length, has been written by me, that it is the record of work carried out by me and that it has not been submitted in any previous application for a higher degree.

I was admitted as a research student and as a candidate for the degree of Doctor of Philosophy in October, 2012; the higher study for which this is a record was carried out in the University of St Andrews between 2012 and 2016.

Date:

Signature of candidate:

Supervisor's Declaration

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of Doctor of Philosophy in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

Date:

Signature of supervisor:

Permission for Electronic Publication

CONTENTS

List of Figures

List of Tables

LISTINGS

CHAPTER ONE

INTRODUCTION

1.1 Problem Statement

Evolution is an inherent characteristic of software systems. Software may be subject to modifications for a number of reasons including improving its performance, correcting and preventing faults and adapting to external changes [1]. Measuring the cost and effort of software maintenance has been the subject of studies as far back as the 1970s [2]. An often quoted figure shows that at least 60% of software development costs is spent on software maintenance [3]. Therefore the management of changes and the evolution of software systems have been extensively investigated through various disciplines with the aim of improving the maintainability of software. These research themes address different aspects of software evolution and include process models, tools supporting software evolution, and versioning systems [4] [5] [6]. Software evolution affects the complexity of the given system and has implications for software quality [7].

The effects of software evolution are exacerbated by the heterogeneous nature of entities representing software systems: the software development process produces artefacts expressed in various forms, such as source code, design diagrams, and requirement documents in natural language. All these artefacts represent the same system, at different levels of abstraction [8]. The evolution of software therefore can be described as the evolution of all of these artefacts [9]. In an ideal scenario, modifications to any of the artefacts will result in other related artefacts also being changed accordingly.

In practice however, software entities evolve at different paces and times. Research on software evolution has not yet adequately addressed the issue of **the differential evolution of software artefacts**, which is the focus of this work. The lack of synchronisation results in artefacts evolving inconsistently, where one representation reflects the latest changes, whilst related artefacts may mirror previous versions, and contain possibly invalid information. A common scenario involves changes being applied to source code while other artefacts are not updated. Such practices can result in an ever growing drift between the different representations. A simplified version of the problem is illustrated by Figure 1.1, which depicts a scenario where three specific types of artefacts - requirements specification, UML class diagram and Java source code - evolve inconsistently. Versions marked in red are inconsistent. The first consistency issue is presented by version 1 (V1) of the requirement specification, which is not modified following the creation of version 2 (V2) of the UML class diagram. Both version 2 (V2) and 3 (V3) of the Java source code conform to version 2 of the UML diagram. The second consistency issue is shown by version 2 of the UML class diagram, which is not updated to reflect Java source code changes, i.e. the creation of version 4 (V4) of the Java source code.

1.2 Motivation

The consequences of the differential evolution of software artefacts can be summarised as follows. Inconsistent artefacts do not accurately represent the software system, and consequently stakeholders may develop a lack of trust in them. Outdated artefacts also hinder effective communication and collaboration, which poses significant challenges in distributed development scenarios. Additionally, not maintaining the consistency of diverse representations impedes the effective understanding of the system. These issues reduce the evolvability [10] of systems and present obstacles to software maintenance.

Considering the efforts associated with maintaining software systems, it is apparent that the cost of such inconsistencies is not negligible. On the other hand, managing the consistency of artefacts is likely to lead to more easily maintainable systems fulfilling their intended purpose.

Various research areas within software engineering have contributed to addressing the problem of disconnected and inconsistently evolving artefacts from different perspectives. These areas include, for example, *Requirements Engineering, Software Processes, Software Change Management (Impact analysis, Software Configuration Management)*, and *Computer-Aided Software Engineering (CASE).*

CASE continues to improve the level of support for managing individual artefacts and software development tasks [11]. In some cases, such as integrated development environments (IDE), some support for keeping selected artefacts consistent may be in place, for example through automatic code generation, which assumes that these artefacts are created and used within the same environment. While CASE caters for some tasks relevant in managing the consistent evolution of artefacts, such as versioning, it falls short of providing a complete solution, which

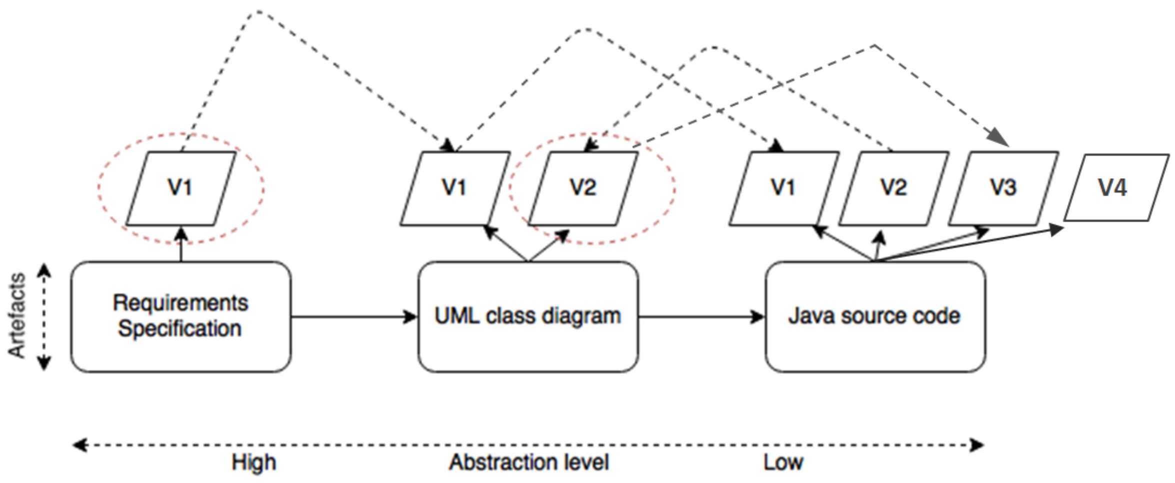


Figure 1.1: Differential evolution of software artefacts.

highlights the fact that the multitude of lifecycle tasks, tools and artefacts pose significant challenges.

Requirements engineering is the process of establishing what services the software should provide, and is concerned with analysing whether requirements are testable, properly understood and have clearly stated origins [12]. A key problem in requirements engineering is the management of inevitable changes. The ability to create and maintain links both between requirements themselves and other artefacts can be particularly useful in tackling this issue, which has not been adequately addressed to date.

Since changes cannot be eliminated, a viable solution that can accommodate them in a consistent manner is required. Several *software engineering process models* have emerged over the past decades aimed at providing a framework for supporting the steps of the development process. Incremental development strategies aim to eliminate the disadvantages of sequential models, such as late design breakage. Dividing the system into units of functionality based on subsets of requirements and delivering this functionality in increments form the core of this approach, providing the benefits of a more refined system at the end of the development process [6]. Being able to identify the impact of a change in one artefact on another requires establishing links between them. However, current processes do not enforce artefact linking and in most software development scenarios software artefacts stay disconnected and go through stages of refinement without considering dependent entities [13].

Besides minimising the challenges caused by modifications, an important aspect of effective *software change management* is the ability to control and coordinate software changes, which is carried out by configuration management [14] [15]. Policies and standards, such as Capability Maturity Model Integration (CMMI) steps for *configuration management* [16], aid change control tasks by specifying their main elements including the identification of configuration items (artefacts) or tracking change requests. Abiding by such practices facilitates the consistent evolution of software artefacts. However the individual tasks involved, which are often tedious and error-prone, in an ideal scenario should be supported in an automatic manner and by a single solution catering for all aspects of the problem.

1.3 Scope

This thesis investigates the feasibility of a holistic consistency management approach to handle the differential evolution of heterogeneous software artefacts. The central claim of this work is that the consistency of heterogeneous software artefacts can be managed in a single framework, independent of representations, tools and methodologies, based on an approach that supports the automation of traceability creation, change detection, change impact analysis, consistency checking and change propagation, and is guided by the principles of extensibility and minimal intrusion to user practices. The proposed approach aims to complement development methodologies such as spiral and agile, and to support artefacts produced in traditional and agile software development.

The approach is realised in a proof-of-concept system, the **Artefact Consistency Management (ACM) framework**. ACM provides semi-automatic traceability creation and change propagation, and automatic change detection, impact analysis and consistency checking functionality to manage the consistent evolution of requirements specification, UML class diagram, Java source code, JUnit test case, UML sequence diagram, UML use case diagram, and software architecture (conceptual and module view) artefacts. These artefacts are selected for the current implementation as they represent various lifecycle stages and abstraction levels. However, the ACM framework is designed to be extensible and to allow new artefacts to be added. Finally, the framework is evaluated using six open source systems.

Inconsistently evolving artefacts is a significant issue which has been discussed in both the *ViewPoint-oriented software development* literature and the *Model Driven Architecture (MDA)* community. While a summary of the main commonalities and differences between the approach presented in this work and these areas is provided, it is outwith the boundaries of this research to discuss their specifics in detail.

ViewPoint-oriented software development refers to the problem as the *multiple perspectives problem*, resulting from "many actors, sundry representation schemes, diverse domain knowledge, differing development strategies", and aims to provide a framework for managing these perspectives. The resulting *ViewPoints framework* offers an infrastructure for supporting multiple methods and views in a distributed, collaborative software development setting, through integrating existing software development tools and methods [17]. Integration constitutes the primary difference between the *ViewPoints framework* and this work, which aims to provide an artefact and tool independent solution. Additionally, multiple-perspectives software development requires a method engineering process to take place, for example, to construct ViewPoints templates through which additional ViewPoints can be created [17].

MDA is a framework for software development, in which different representations of a system are referred to as models [18]. The steps of software development include specifying Platform Independent and Platform Specific models, which are transformed to code [19]. The issue of inconsistency emerges as a result of changes. *MDA* specifies a methodology to carry out software development tasks, while the work presented in this thesis aims to explore an approach to provide support to manage the evolution of software artefacts as part of existing software development activities. However, the ACM framework does not preclude the use of MDA and MDA specific models.

1.4 Research Question and Hypotheses

This thesis poses the following research question:

Is it feasible to handle the differential evolution of heterogeneous software artefacts automatically without imposing specific methodologies or tools and in an artefact independent manner?

Answering this question entails the identification of the tasks involved in handling the evolution of heterogeneous artefacts, and the investigation of an achievable level of automation and a suitable representation of diverse software artefacts. To answer this research question, the following hypotheses are investigated:

H1. The differential evolution of heterogeneous software artefacts can be handled by one holistic consistency management framework. Currently no single approach offers a full solution to manage the consistency of software artefacts. The feasibility of this hypothesis is investigated through the design and implementation of such a holistic framework, the concept of which is introduced in Chapter 4.

H2. An artefact consistency management framework need not impose specific methodologies or CASE tools on the user. Existing solutions aimed at ensuring the consistency of artefacts often prescribe the processes to be followed and tools to be used. This is a restrictive approach considering the multitude of representations, tools and the diversity of projects.

H3. It is possible to automate all aspects of artefact consistency management. This work hypothesises that some aspects can be fully automated, while others may require manual effort, and hence can be partially automated.

H4. An artefact consistency management framework can be independent of specific artefacts: it can cater for heterogeneous software artefacts and can be extended to handle any new artefacts given that data contained in them can be accessed.

1.5 Novel Contributions

1. A survey and classification of approaches (see Chapter 3) relevant to the discussion of artefact consistency management. The survey reveals which research areas contribute to addressing the problem, and through an evaluation it identifies potential shortcomings. Based on the findings of the survey a set of challenges and characteristics of an ideal consistency management solution are derived (see Chapter 4) from which high level requirements of a consistency management framework are formed (see Chapter 4).
2. A holistic conceptual approach to support artefact consistency management incorporating traceability creation, change detection, impact analysis, consistency checking and change propagation (see Chapter 4).
3. The design, implementation and evaluation of a proof-of-concept prototype, the ACM framework[1], which provides:

   - An extensible, property-graph based representation of heterogeneous software artefacts
   - Automated support to transform heterogeneous representations to a unified graph-based format, using XSLT transformations
   - Change detection functionality to support

     a) The extraction of changes from an external repository

     b) The identification of changes at the property graph level using a graph-based change identification algorithm
   - Graph-based change impact analysis of heterogeneous software artefacts
   - Extensible XML consistency rule base to support consistency checking of heterogeneous software artefacts

[1]Source code can be found at: https://github.com/ACMFramework/ACMF

- A set of heterogeneous software artefacts obtained from six open source systems used for evaluation
  These contributions are discussed in Chapter 5, 6 and 7.

4. An approach to automate trace link creation between heterogeneous software artefacts using machine learning (see Chapter 8).
5. A data set containing 1100 data instances representing trace links between UML diagram (use case, sequence diagram, class diagram), Java source code, JUnit test case, and software architecture (module view and conceptual view) artefacts. The data set provides a foundation for conducting traceability experiments for heterogeneous artefacts using machine learning and can be found at: https://github.com/ACMFramework/ACMTraceability.
6. A trace link classification, which is utilised in the ACM framework (see Chapter 2).
7. A categorisation of heterogeneous software artefacts (see Chapter 2).

1.6 Thesis Organisation

Chapter II provides a discussion of concepts relevant in the discussion of artefact consistency management.

Chapter III presents a survey of related work. These solutions are classified and subsequently evaluated.

Chapter IV describes the proposed solution, and the concept of the holistic view of artefact consistency management.

Chapter V presents the overall architecture and design of the prototype (ACM) framework.

Chapter VI discusses the data representation strategy of ACM framework.

Chapter VII describes the implementation approach of each framework stage in the ACM framework.

Chapter VIII introduces a machine learning based approach to automate trace link creation between heterogeneous artefacts.

Chapter IX provides an evaluation of the proposed framework to assess feasibility and to demonstrate the effectiveness of the solution in fulfilling the requirements set out.

Chapter X concludes the thesis by discussing the strengths and potential shortcomings of the research and further areas of potential extension.

1.7 Publications

I. Pete and D. Balasubramaniam, "Handling the Differential Evolution of Software Artefacts : a Framework for Consistency Management," in Paper presented at 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, 2015, pp. 599-600.

Works in Progress

Towards a Holistic Artefact Consistency Management Framework
Article manuscript submitted for publication to *Automated Software Engineering* in October, 2016.

A Survey of Artefact Consistency Management
Article manuscript submitted for publication to *Journal of Systems and Software* in October, 2016.

CHAPTER TWO

BACKGROUND

This chapter introduces the terminology relevant in the discussion of artefact consistency management, which is derived from established research areas. Firstly, the concept of software artefacts is introduced and a classification of artefacts is provided. Subsequently, traceability is defined and a classification of trace links is supplied, which is followed by a description of change impact analysis. Lastly, definitions of consistency, consistency management, consistency checking and change propagation are discussed.

2.1 Software Artefacts

2.1.1 Definition

The term *artefact* is used both in industry and academia to refer to entities representing a software system. The context in which the concept primarily appears in industry is that of configuration management. Software artefacts can be thought of as information container units, and can be described by a number of attributes, such as actors creating or consuming them. Artefacts also have a state and a lifecycle, and a significant aspect of them is that they can be linked to other artefacts [20].

Finkelstein et al. define software artefacts as follows: "The (sub)-products and ’raw material’ of a process. An artefact produced by a process may later be used as a raw material by the same or a different process to produce other artefacts. Artefacts are often persistent and versioned. An aggregate of software artefacts to be delivered to a user is called a software product" [17]. Another definition by Beyer and Noack highlights the fact that there is a diverse range of software artefacts that constitute a software system ranging from documentation through test cases to source code [21].

In this work software artefacts are defined as products of the activities involved in software development. Due to the variety of these lifecycle activities the resulting artefacts show a high level of heterogeneity. Heterogeneous representations may also be managed by different tools, such as Microsoft Word[1], Axiom[2], Rational Rose[3], Microsoft Visual Studio[4], Eclipse[5], TestLink[6].

2.1.2 Artefact Classification

In the following section we propose a classification of software artefacts capturing the variety of dimensions through which a system can be represented. The categorisation is based on the following:

1. **Aim and scope of artefacts**: this aspect indicates whether the artefact is aimed at describing the system at a high level to allow stakeholders to communicate design decisions, such as an architecture diagram, or it captures low-level implementation details, such as source code. Thus, categories may include specification artefacts, communication artefacts, and implementation artefacts among others.
2. **Process methodology and life cycle stage**: each stage in the software life cycle has a well-defined purpose and produces specific artefacts. In a conventional waterfall life cycle [12], the analysis phase produces analysis artefacts, which can take multiple forms depending on the given project. For example, the results of analysis can be expressed through natural language requirements specifications, or equally, UML use cases. During design, typically architecture and design artefacts are created, such as design diagrams or architectures represented by Architecture Description Languages (ADL). The implementation phase may produce source code items or an executable (implementation artefacts). Testing may result in unit tests expressed in source code or test cases in a test management system (test artefacts). Finally, an artefact typically associated with the maintenance phase is an issue, a bug, or an item in a bug tracking system (maintenance artefacts).

   Agile practices [12] may result in new artefact types as they structure the software life cycle in a different manner. A notable difference when compared to the waterfall life cycle is the reduction of effort spent on upfront specification. For example, product backlogs produced during the project planning phase may capture requirements, and features, user stories or use

[1]https://products.office.com/en-gb/word
[2]http://www.iconcur-software.com
[3]http://www-03.ibm.com/software/products/en/ratirosefami
[4]https://www.visualstudio.com/
[5]https://eclipse.org/
[6]http://testlink.org/

cases may replace rigorous documentation (analysis artefacts). Each sprint produces source code and tests (implementation and test artefacts), and may also be accompanied by design diagrams capturing parts of the system (design artefacts).

3. **Specific artefact structure**: certain artefacts are characterised by a well-defined structure, such as source code built up from language constructs abiding by specified syntax rules. On the other hand, some artefacts, such as requirements specifications, do not necessarily exhibit a pre-defined structure to be rigorously followed across artefacts of the same type. Therefore, their structural elements may be shaped by authors' preferences and decisions. Another consideration is whether the structure of artefacts is hierarchical or flat.
4. **Abstraction level**: artefacts can be characterised by their abstraction level, which indicates the level of detail they capture. In this work, a relative scale is used to compare artefacts to identify whether they represent a lower or a higher abstraction level when compared to another one. The highest abstraction level is exemplified by requirement specifications, while an instance of the lowest abstraction level relevant in this discussion is source code, for example.

The variety of representations is further expanded by the different sub-types. For example, source code can take various forms; high-level programming languages can be grouped into categories based on the programming paradigm [22] they follow, and a single category may include multiple programming languages which differ in features they provide. Most artefact types can be categorised in multiple ways, which is illustrated by examples given in Table 2.1.

2.2 Traceability

In this section the definition of traceability used in this work, the description of traceability-related concepts, and a classification of trace links are presented.

2.2.1 Definition, Significance and Terminology

A key aspect of software change management is the ability to understand relationships among software artefacts. In a typical software project, a large number of artefacts may exist and the specification of relationships among them is a complex and challenging task. The area of traceability is concerned with the specification and maintenance of such links.

The need for traceability was first expressed at the end of the 1960s [25] and since then it has been most extensively researched in the requirements engineering community. Its most widely cited definition, which originates in this community, was coined by Gotel and Finkelstein: "the ability

Artefact Type		Categorisation and Examples
Source code	►	Based on programming paradigms [22], for example
		Imperative: C Declarative: SQL Event-driven: Javascript Functional: Haskell
UML design diagrams	►	Based on a classification proposed by Rumbaugh et al. [23], for example
		Structural: class diagram, collaboration diagram Dynamic: state machine diagram, activity diagram Physical: deployment diagram Model management: package diagram
Architecture diagrams	►	Based on architectural views [24], for example
		Logical view architecture Development view architecture Process view architecture

Table 2.1: Artefact Categorisation Examples.

to describe and follow the life of a requirement, in both a forwards and backwards direction" [26]. For the purposes of this work, a more generic definition of software traceability is proposed: *traceability refers to the ability to interrelate heterogeneous software artefacts representing a software system at different abstraction levels.*

Although traceability has been described as a quality attribute of software systems [27] and is required for compliance with certain industry standards [28] [29] [30], it is not characterised by broad acceptance and wide adoption in industry. In 1994, this problem was attributed to the lack of a common definition of traceability and the diverse user, project, task and information requirements [31]. The so called traceability problem [26] continues to exist in the present, where reasons for the lack of industry-wide adoption include costs and efforts associated with implementing traceability techniques. Current approaches may not offer a desirable level of accuracy and coverage, which also impedes their adoption.

Conversely, the importance of the area is well illustrated by the establishment of the International Centre for Excellence for Software Traceability (Coest), which was created to encourage research collaborations in the area of traceability [32]. Implementing traceability allows software projects to better abide by the various standards set for requirements engineering. Other advantages include software reuse, improving system comprehension, and the ability to assess completeness of implementation. A major contribution of traceability is in the area of software maintenance, where links connecting entities aid the identification of parts of a system affected by a change. Specifically, it supports software change impact analysis [33], which is discussed in detail in Section 2.3. As traceability relations indicate dependencies between software artefacts, these relationships also support a number of other tasks such as understanding rationale and design decisions [34].

Following is a brief summary of fundamental traceability concepts, which are used throughout this thesis. The definitions are adopted from the work of Gotel et al. [25].

Trace link. An association between two artefacts as denoted by Figure 2.1. The illustration also shows that trace links are effectively bidirectional as they can be traversed in two directions.

Besides the above definition, we define trace links as a pair *P* of a source and a target. Source and target are elements of the set of artefact elements.

Source and Target artefacts. The two ends of the association describing the origin and the destination of the trace link. In this work, artefacts may also be referred to as representations.

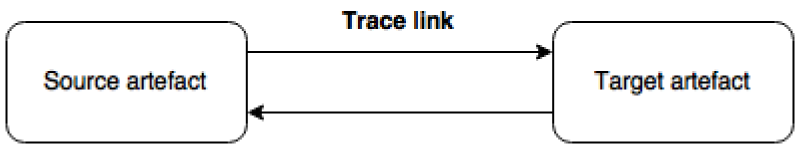


Figure 2.1: Trace link connecting a source and a target artefact.

Trace artefact type. As mentioned in Section 2.1.2, artefacts can be grouped based on similar characteristics they share.

Trace. "A specified triplet of elements", which consists of the source and target artefacts, and a trace link associating them.

Tracing. The process of establishing or using traces.

Trace granularity. The level of detail at which a trace captured, defined by the granularity of the source and the target.

Traceability creation. An activity that involves the:

a) definition of the source and target artefacts and their trace links,
b) representing of traces resulting from the tracing process,
c) storing of traces, and finally
d) validation of traces.

In this work the following synonyms are used to refer to this activity: establishing trace links, trace link creation.

Traceability maintenance. A process aimed at managing already established traces including the

a) retrieval,
b) analysis,
c) update, and
d) verification of traces.

Vertical tracing. The process of tracing artefacts, which are at different abstraction levels (for example tracing between source code and requirements).

Horizontal tracing. The process of tracing artefacts, which are at the same abstraction level (for example tracing between requirements).

2.2.2 Trace Link Classification

As Spanoudakis et al. point out, stakeholders in software projects utilise different links depending on their perspectives and goals [35]. Therefore, numerous link types exist, which can be categorised in various ways. Section 2.2.2.1 provides a summary of these classifications and Section 2.2.2.2 introduces a categorisation of trace links connecting heterogeneous artefacts.

2.2.2.1 Existing Classifications

A summary of trace link classification strategies is provided by Winkler and Pilgrim [36]. Additionally, Spanoudakis et al. present an extensive overview of trace links and survey existing classifications from the requirements engineering and software change management communities. The identified types of trace links include dependency, generalisation/refinement, evolution, satisfaction, overlap, conflicting, rationalisation, and contribution relations [35]. Dependency links imply that the existence of an element relies on the existence of another one. Generalisation/refinement links demonstrate how elements of a system can be broken down into components, how elements can be combined to form other entities and how elements can be refined by others. Evolution links capture the history of development in a structured manner. Satisfaction links indicate that requirements are satisfied by the system. Overlap links denote whether two requirements refer to the same feature. Conflicting links capture scenarios where two requirements conflict with each other. Rationalisation links allow the identification of the rationale behind creating the given artefact. Lastly, contribution relations are established between artefacts and the stakeholders who produced them.

Another categorisation of trace links is introduced through the reference model created by Ramesh et al. [31] who enhance their traceability meta model with four types of links, specifically, Satisfaction, Evolution, Rationale and Dependency links. Some of these links overlap with those specified above, as shown by their definitions:

A) Satisfaction links aim at ensuring that requirements of the system are satisfied.
B) Evolution links allow the recording of actions leading from existing objects to new objects.
C) Rationale links help identifying the rationale behind creating objects.
D) Dependency links aim to manage dependencies between objects.

2.2.2.2 Trace Link Classification of Heterogeneous Artefacts

To complement existing work in trace link classification and to provide a means to categorise link types used throughout this work, a generic classification of heterogeneous artefacts is proposed. The taxonomy differentiates two major trace link types, *inter* and *intra* links.

Intra Links

Intra artefact links exist among elements of the same artefact. Although they typically capture logical relationships in the application domain, there may be instances where developers may wish to add other links to aid understanding and maintenance. In this category, links may capture the following:

1. **Domain dependency**: to denote that relationships exist among elements to reflect the application domain. For example, composition, specialisation, use, instantiation, rationale and containment may be represented by such links.
2. **Development links**: to denote relationships that are identified during development although they may not be part of the application domain. For example, these links may be annotated to indicate that the two linked elements should co-evolve, or overlap, conflict or reinforce each other.
3. **Evolution**: to denote that an element is a new or revised form of another in an earlier version of the artefact. Such changes can happen as a result of corrections, changes to user or system requirements or other artefacts, and refinement as part of development activities.

Inter Links

Inter artefact links are those that exist between different artefacts. Within this category, links between elements in different artefacts may be created to capture the following:

1. **Identity**: to denote that the linked elements are the same but seen from different perspectives.

For example, an object in two different UML diagrams or a source code method and its corresponding unit test may need to be identified as the same entity.

2. **Satisfaction**: to denote that an element in one artefact satisfies, implements, evaluates or describes an element in another artefact, typically at a higher level of abstraction.

2.3 Change Impact Analysis

2.3.1 Definition and Terminology

Unmanaged modifications to a software system can have serious consequences. Therefore, a change management process is needed to control changes and their impact. An aspect of controlling changes is the activity of identifying the scope of changes, which is called impact analysis (IA). Bohner and Arnold describe the concept as the identification of the (ripple-) effects and consequences of a change [37]. Impact analysis is a recognised change management activity and it is part of the Change Management Process Framework described by Leffingwell and Widrig [38].

IA can be carried out prior to and following the implementation of a change [39]. Before a change is implemented, it aids the planning and estimation of costs associated with the given modification. IA techniques may also be applied to trace ripple effects of changes and to propagate changes following their implementation. The following concepts are related to the IA process and its evaluation as adopted from the work of Kama [40]:

Starting impact set (SIS). The first step of impact estimation is the identification of a set of entities initially affected by the change, which constitute the elements of SIS.

Candidate or Estimated impact set (CIS or EIS). The process of tracing from the elements contained within the SIS results in the establishment of a set of potentially impacted entities.

Actual Impact Set (AIS). This set contains elements, which are actually affected as a result of the change.

Discovered Impact Set (DIS). It represents a set of elements actually impacted by the modification although not identified and not in the CIS.

False Positive Impact Set (FPIS). These elements represent overestimation of impacts as it contains elements that were not actually impacted yet were identified.

In the light of these concepts, IA can be defined as an activity aimed at estimating a CIS, which coincides with the AIS. The most frequently applied accuracy metrics of the impact analysis process include precision and recall. Precision measures the percentage of candidate impacts that are actual impacts, whereas recall measures the percentage of actual impacts included in the CIS [41].

Precision can be calculated as follows.

$$Precision = (|EIS| \cap |AIS|)/(|EIS|) \tag{2.1}$$

while, recall is computed as follows.

$$Recall = (|EIS| \cap |AIS|)/(|AIS|) \tag{2.2}$$

2.3.2 Categorisation of Change Impact Analysis Techniques

Numerous methods have been proposed to estimate change impact. To conclude this section, a brief summary and categorisation of the most frequently used approaches is supplied; a detailed discussion of the topic is provided in literature [39] [40] [42].

One categorisation is based on the scope of analysis they provide. While the majority of approaches support IA within source code artefacts, some others facilitate heterogeneous artefacts. Ibrahim et al. refer to the first category as code-level impact analysis, while the latter is discussed as broader perspective impact analysis [43].

The categorisation provided by Wong et al. allows the derivation of the following IA concepts [44]:

1. **Structure-based IA.** These approaches utilise dependency structures of source code artefacts and trace links between heterogeneous artefacts in order to establish impact sets. A potential drawback of such approaches is that additional techniques are required to reveal semantic relationships.
2. **History-based IA.** To address shortcomings of structure-based IA solutions, these techniques utilise revision histories to identify logical couplings between entities. Discovering logical couplings (also referred to as evolutionary coupling) highlights how entities historically change together. Particular techniques include association rules (such as a $\implies$ b), to predict that an entity *b* is likely to change as the result of an entity *a* being changed.
3. **Probabilistic IA.** To complement structure-based and/or history-based IA approaches and to predict changes, probabilistic techniques can be leveraged. These include Bayesian networks and Markov processes.

2.4 Consistency, Consistency Management, Consistency Checking and Change Propagation

The concept of consistency is integral to the discussion of artefact consistency management. In the remainder of this chapter, consistency and consistency management are first discussed. Subsequently, the definition of artefact consistency management is derived and the concept of consistency checking is introduced.

2.4.1 Consistency

In the most generic sense of the word, consistency can be defined as the "condition of adhering together" [45]. It is a widely used term in various areas within software engineering, which handle different types of consistency related problems. However, at a high level, consistency in these diverse cases refers to the same notion, that is, two entities are consistent if they abide by some consistency definition, otherwise they are inconsistent.

A discussion of related work relevant to consistency is provided in Chapter 3. For example inconsistency is defined by Nuseibeh et al. as "a situation in which a set of descriptions do not obey some relationship that should hold between them" [46]. Zisman and Spanoudakis informally describe inconsistency as a "state in which two or more overlapping elements of different software models make assertions about the aspects of the system they describe which are not jointly satisfiable." [47]

Since this work focuses on the consistency of heterogeneous software artefacts during software evolution, the concept of consistency can be further narrowed to refer to this specific case. In the scope of this discussion a set of heterogeneous artefacts are consistent if they abide by conditions such that they represent the same state of a given software system.

2.4.2 Consistency Management

Similarly to consistency, the problem of consistency management is discussed in various software engineering areas including requirements engineering [48] and model-based software engineering [49]. Consistency management also lies at the core of solutions aiding the development of complex systems, such as object management systems [50], and the viewpoints system [17]. The discussion of consistency management in these areas is outside the scope of this work.

However, a generic definition of consistency management can be derived from these areas, and it involves the activities of 1) defining consistency conditions in relation to entities the consistency management is aimed at, 2) defining and identifying violations of consistency, and 3) re-establishing consistency following violations [50].

Specific areas of concern in consistency management include the way consistency conditions are formulated, for example, using constraints or rules, and the way consistency conditions are expressed, for example, using formal means or specification languages.

2.4.3 Artefact Consistency Management

The above definition of consistency management can be further refined to introduce the concept of artefact consistency management, which is used throughout this thesis. At a high level, artefact consistency management consists of a set of tasks aimed at keeping heterogeneous software artefacts consistent in the face of changes. *Software artefacts evolve consistently if changes applied to one artefact are reflected in all related artefacts and inconsistent representations before they are further used.* Based on this description it can be deduced that the consistency management activities defined in Section 2.4.2 form only a subset of the tasks artefact consistency management involves. The identified aspects of artefact consistency management are introduced in detail in Chapter 4.

2.4.4 Consistency Checking

Based on the definition of consistency management described in Section 2.4.2, consistency checking can be generally defined as a process that consists of the following activities:

a) definition of consistency conditions, and
b) detection of violations

Therefore consistency checking can be described as a subset of the activities involved in consistency management. Specifically, *consistency checking is the activity of assessing whether conditions defining consistency between or within selected artefacts hold following a change.* The topic of consistency checking between versions of specific artefacts has been widely researched, particularly in relation to the consistency of UML models [51]. Certain approaches are independent of artefacts and address generic consistency issues: Vierhauser et al. provide examples of consistency checking between specific models and generic approaches [52].

2.4.5 Change Propagation

Closely related to the concepts of impact analysis and consistency checking is change propagation, which is defined in the impact analysis literature as follows: "change to one part or element of an existing system configuration or design results in one or more additional changes to the system, when those changes would not have otherwise been required" [53]. This is not the definition adopted here since in this work, similarly to Han's research [54], change propagation is a separate activity.

Change propagation is a key activity in software maintenance and change management as it ensures that modifications are correctly applied to all dependent entities and no inconsistencies are left in the system as a result of the given change and its ripple effects. Change propagation can be applied within homogeneous artefacts, where a modification introduced to, for example, source code may result in further changes to other source code entities. This problem [55] [56] has been investigated in dependency analysis research [57]. Malik defines change propagation as "the process of propagating code changes to other entities in a software system to ensure the consistency of assumptions in the system after changing an entity" [58].

Based on the definition of consistency management described in Section 2.4.2 and related to Malik's definition, we define change propagation as *a consistency management activity aimed at enforcing consistency by re-establishing it following violations*.

Change propagation across heterogeneous artefacts is highly relevant in this work, where a modification to one software artefact results in inconsistencies in other entities and changes need to be propagated in order to resolve the consistency violation. To conclude, it is worth noting that change propagation also involves the discovery of the degree to which inconsistencies are tolerated and whether an optimistic or pessimistic approach is adopted to solve inconsistencies.

2.5 Conclusion

In summary, this chapter has defined the concepts of software artefacts, traceability, change impact analysis, consistency, consistency checking, consistency management and change propagation. This serves as a preamble to discussing related work and the proposed consistency management approach in Chapter 3 and 4, respectively.

CHAPTER THREE

LITERATURE REVIEW

This chapter explores related work relevant to artefact consistency management. The approaches, which span a number of research areas, are first classified. Following a detailed discussion of their characteristics, an evaluation is carried out to identify their benefits and shortcomings. This leads to establishing the characteristics of a potential solution in Chapter 4 addressing the identified shortcomings.

3.1 Introduction

State-of-the art solutions presented in this chapter contribute to advancing the field either by considering the problem in a holistic manner or by focusing on specific aspects of it. It is not within the scope of this work to outline the history of all related fields and to identify the main challenges of the individual research areas. Instead, the aim of the analysis is to compare solutions originating from diverse research areas based on common characteristics related to the discussion of artefact consistency management and to reveal the extent to which they advance artefact consistency management.

Specifically, the focus of the review is to identify to what degree the issue has been addressed and which research areas contribute to solving it. To obtain answers to these questions, relevant solutions were analysed and evaluated based on the questions described in Subsection 3.2.4.

The rest of the chapter is organised as follows. Firstly, the methodology used to carry out the review is presented in Section 3.2, followed by a classification (Section 3.3 and analysis of solutions (Section 3.4). The chapter concludes with the outcomes of the evaluation discussed in Section 3.5.

3.2 Methodology

The review process was guided by principles of systematic literature reviews in software engineering, based on a description of a methodology by Biolchini et al. [59] and a survey provided by Kitchenham et al. [60]. The aim of adopting a non ad hoc approach was to allow the reproducibility of the investigation. It is also worth noting, however, that despite the up front planning, the review process is an iterative one. Therefore, certain planned aspects of it, such as search terms and inclusion criteria were subject to change.

3.2.1 Planning

3.2.1.1 Research Questions

The following research questions are formulated to provide the groundwork for conducting the survey.

RQ1. What research areas contribute to artefact consistency management?
RQ2. Which aspects of the problem of artefact consistency management do these approaches address?
RQ3. Do these approaches implement the proposed solution and what level of automation do they provide?
RQ4. Which software artefacts do these solutions cater for?

3.2.1.2 Search Process

Firstly, Google, Google Scholar [1] and CiteSeerX [2] were selected to perform search. Secondly, the types of information sources were considered: conference proceedings, journals, PhD theses and websites dedicated to solutions. Finally, search terms were identified, which include: *software traceability, traceability creation, traceability maintenance, software artefact, software change impact analysis, software artefact consistency, software change propagation, heterogeneous software artefacts, software change management, software configuration management, consistency checking*.

Inclusion and Exclusion Criteria

At the outset, studies originating between 2000 and 2013 were selected to identify the latest results and approaches in each area. This was later extended to studies dating back prior to 2000

[1] https://scholar.google.co.uk/
[2] http://citeseerx.ist.psu.edu/index

to include works laying the foundations of these areas.

The above search terms also returned not directly relevant results, primarily due to two reasons. Some concepts, such as *consistency* or *consistency checking*, appear in various research areas. To filter such results and exclude irrelevant ones, specific search techniques were utilised, such as the exclusion search operator.

Besides, some of the search terms refer to large research areas, therefore studies directly relevant to artefact consistency management and the research questions had to be selected. To illustrate this process, the example of the term *software traceability* is taken. Firstly, the relevance of traceability in artefact consistency management is identified. Since traceability allows connections between heterogeneous software artefacts to be established, traceability solutions aimed at supporting the automatic linking of artefacts are considered. As software artefacts evolve, their connections are also impacted, therefore traceability maintenance solutions are also within the scope of potential solutions. Secondly, further questions were raised to select relevant solutions. For example:

- Are there any approaches that incorporate traceability to support software evolution?
- Are there any standalone traceability solutions that cater for heterogeneous software artefacts?

A similar process was applied with respect to each search term pertaining to larger research areas.

3.2.2 Execution

The original study was first carried out between 2012 and 2013 and was followed up periodically to identify unseen work. This approach proved to be effective as further examples of relevant work were found after 2013 until the submission of this thesis.

Data Collection

The following information was extracted from the studies in question:

- The aim and scope of the work
- Motivation
- Supported software artefacts and extensibility
- Level of automation: is the solution manual, or semi-automatic, or fully automated? Which aspects of the solution are automated, if any?
- Implementation details of prototype, if applicable, with respect to extraction and storage of software artefacts and trace links.

- Stages of artefact consistency management (introduced in Chapter 4) supported.

3.2.3 Results Analysis

Based on the findings of the search process, a classification of the investigated studies was established. The main premise of the classification is that some solutions are holistic and support multiple aspects of artefact consistency management, while others focus on solving a particular aspect of the problem. Since the boundary between the specific research areas is often blurred, the classification presents a major challenge in this review process.

Besides the categorisation, the findings also allow an evaluation of the individual solutions to be carried out, the criteria and results of which are discussed in Section 3.5.

3.2.4 Related Surveys

To the best of our knowledge, a comprehensive overview combining and assessing results achieved in areas that contribute to artefact consistency management, has not yet been produced.

Closely related to this survey is a review of traceability in requirements and model-driven development by Winkler et al. [36]. The classification approach they follow is based on grouping solutions around the hindering factors traceability practices face and strive to overcome. These are namely natural, technical, economic and social limitations. The authors classify research publications based on the artefacts among which traceability is established.

In "A Review of Software Change Impact Analysis", Lehnert presents the results of an investigation of change impact analysis and identifies five scopes of impact analysis [42]. Accordingly, impact analysis can be performed at the *source code*, *architectural model*, and *requirements model* level. Additionally, some solutions consider *miscellaneous artefacts*, such as documentation, configuration files, bug trackers. Finally, *combined scope approaches* provide a comprehensive analysis to trace impacts across different types of artefacts. Some of the solutions mentioned in Lehnert's survey are highlighted in this work for their contribution to artefact consistency management.

Sun et al. provide a survey of code-based change impact analysis techniques proposed between 1997 and 2010 in [39]. Their work constructs a framework to characterise and classify these techniques based on a set of criteria including the impact set, the type of analysis, intermediate representations, language support, and tool support. While this work is specific to source code artefacts, which is the focus of the majority of change impact analysis solutions, it provides useful information on the evaluation of source code change impact analysis techniques, which may be extensible to other artefacts.

Although not a survey paper, a study by Cleland-Huang et al. [61] presents a brief summary of work in the areas of trace creation, trace maintenance and trace integrity, the correctness of identified and maintained links. This work contributes to assessing current solutions. Similarly, Gotel and Finkelstein investigate the requirements traceability problem and reveal that one of the reasons contributing to the persistence of the issue is the lack of a common definition of *requirements traceability* and related concepts [26]. The authors also provide a framework for addressing the problem.

3.3 Classification

In this section, existing approaches are classified, which is followed by a detailed review in Section 3.4. As shown in Figure 3.1, solutions can be grouped into two primary categories based on the scope of the problem they aim to address. *Holistic solutions* are characterised by a comprehensive support for managing artefacts incorporating techniques from traceability, change impact analysis and various fields of software change management including configuration management. Conversely, *specific solutions* focus on techniques for the individual aspects - linking artefacts, assessing the ripple effects of changes and controlling the evolution of artefacts by various means.

3.4 Review of State-of-the-art Solutions

The review provides a means to identify which related areas have received greater attention and to articulate areas for improvement by assessing the individual solutions. It also aims to contribute to understanding the reasons behind the lack of automated tool support for managing software artefacts.

This section is organised as follows. Subsection 3.4.1 presents and categorises holistic solutions; subsection 3.4.2 introduces and groups approaches from various fields addressing specific aspects of the problem. Each solution is evaluated based on:

- its motivation and aims, and
- the stages of artefact consistency management (introduced in Chapter 4) and artefacts it supports.

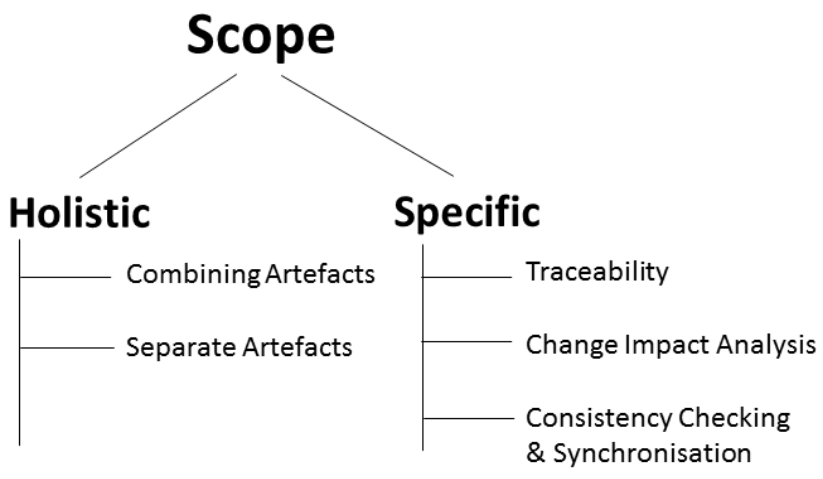


Figure 3.1: Classification of solutions contributing to artefact consistency management.

3.4.1 Holistic Solutions

This category contains approaches that attempt to cater for multiple aspects of artefact consistency management. Thus, they focus on managing the evolution of artefacts throughout the development process. These solutions recognise the importance of handling heterogeneous artefacts and the multidimensional nature of software [9]. They are further divided into subcategories based on their underlying approach for representing artefacts; while some approaches combine artefacts to embed one in another, others keep them separate and may utilise a unified representation for processing them. A subset of these solutions manages heterogeneous artefacts to support collaboration between stakeholders producing these artefacts.

3.4.1.1 Maintaining Separate Artefacts

Lehnert et al. present a rule-based approach for analysing the impact of changes across heterogeneous artefacts [62]. The solution provides a rule-based dependency analysis method to identify and record traceability links, and utilises a change taxonomy comprised of atomic and composite change operations. Additionally, change impact is determined based on the combination of change types and dependencies, and is expressed in the form of propagation rules. The supported artefacts, UML models, Java source code and JUnit test cases, are mapped to a unified format, to EMF-based models, and stored in the *EMFStore* model repository[3]. Through this mapping heterogeneous artefacts are supported. However, to extend the solution with additional artefacts, users are required to produce EMF-based meta-models. The feasibility of the approach is demonstrated through the *EMFTrace* prototype, which is an extension of the *EMFStore* model repository. The evaluation of the solution through a case study revealed that it is able to determine the impact of changes across heterogeneous artefacts and the rule-based impact analyses approach resulted in an average precision and recall of above 80%. Most features of *EMFTrace* are automated, thus it is a promising solution to assist users with their change impact analysis tasks. However, dependency detection and impact analysis rules are manually created and maintained, which may become a cumbersome task in cases where a large number of rules exist. Additionally, the prototype requires various dependencies such as *Eclipse Kepler Modeling Tools*, *EMFStore*, *EMF Client Platform*, *MoDisco*.

Olsson et al. [8] describe a conceptual architecture and a prototype tool for managing traceability and inconsistencies among software requirement descriptions, UML use case models and black-box test plans. Similarly to *EMFTrace*, their solution extracts key information from artefacts. Relationships among elements of these models are manually established. The prototype tool captures changes and displays them to developers who can take appropriate action based on these notifications. Thus, the majority of change propagation is carried out in a manual manner, while in some cases changes can be automatically propagated based on entity name similarity. The approach is closely related to work carried out in the area of inconsistency management in multiple-view software development environments, based on the notion that software can be viewed in analysis, design, code and test views, where inconsistencies between the views can be detected, monitored and presented to users who can resolve them once their causes are located [63].

Han takes a different approach to managing changes of heterogeneous artefacts [54]. Instead of using an extracted representation of artefacts, the change management activities, such as impact analysis and change propagation, are carried out on the original representations of software artefacts in the development environment. The approach incorporates the representation of relationships between artefacts, change impact analysis utilising change patterns and suggestions for carrying out change propagation based on rules and user intervention. The technique, however,

[3]http://eclipse.org/emfstore

has not been implemented in a proof of concept tool.

The fundamental premise of Reiss' constraint-based work [64] is that such a solution should be independent of tools and notations and should not rely on any artefacts being the primary representation. The approach is based on meta-constraints describing how entities in one type of artefact should be associated with entities in another type of artefact. For instance, the design can be thought of as constraints on the source code (for example a UML diagram can impose constraints about the existence of a class in the source code). Constraints are specified as predicate equations and are stored in a relational database. The concept was implemented in a prototype tool, *CLIME* and has been evaluated with the following artefacts: UML class diagrams, source files, design patterns and design constraints [9]. Independence from tools and artefacts is achieved by extracting relevant information from artefacts. However, the solution does not utilise a common representation of these. Instead, relevant information is abstracted from the original artefacts. Change detection is implemented through the update manager, which determines whether any of the artefacts in the project have changed since the last check. Then it notes any artefacts that have been deleted, added or modified. Change propagation is carried out in a manual manner: in case any inconsistencies appear, they are indicated to software developers, who can resolve them manually. The main strengths of the solution lie in its attributes of artefact and tool independence. However, *CLIME* currently only handles a limited set of artefacts and the maintenance of a large number of constraints may be problematic. Extending the prototype involves creating an information extractor specific to the artefact to be added.

Ensuring consistency between source code and design is the focus of the solution proposed by Hammad et al. [65] who aim to answer the question "Does a particular source code modification affect the design and if so, how should the design be changed". Firstly, an XML-based format, srcML[4] is utilised to represent source code, and another XML format, srcDiff, is used to capture source code changes. Source code modifications impacting the design (UML class diagrams) are identified using XPath queries, and results are presented to the user who can manually resolve the inconsistencies. Therefore, automatic change propagation is not implemented in the solution. The technique does not require explicit traceability links. However, it relies on the existence of a version of the source code and the class diagram when both artefacts were consistent with each other. The approach is implemented in the *srcTracer* prototype, which has been evaluated through a case study using C++ and UML class diagram artefacts and by comparing manual inspection results with that of the tool. The evaluation shows promising results as the tool has demonstrated the capability to reduce the effort required for the task. However, the approach is specific to source code and design artefacts, and therefore does not provide an artefact independent solution.

[4]http://www.srcml.org/

Zekkaoui et al. [66] recognise the multi-dimensional nature of software systems where representations evolve throughout the lifecycle and at different rates. The aim of their work is to facilitate the expression and management of consistency between all artefacts by proposing a unified approach to represent heterogeneous artefacts. This is achieved through a meta-model capable of representing all types of artefacts and with the help of a rule engine allowing users to define consistency rules as logical operations between artefacts. Their approach, partly implemented in a proof of concept tool and the *CMAC* Eclipse plugin under development, identifies elements that are not compliant with the specified consistency rules. Artefacts are represented using typed graphs and they are extracted manually by a domain expert. At the time of writing this thesis the approach was work in progress and no evaluation was provided. However, it is a promising solution as it ensures artefact independence by allowing users to extend existing consistency rules and artefacts (through the *Artifact Builder* architectural component).

ArchEvol [67], developed by Nistor et al., is aimed at maintaining an accurate architectural model consistently mapped to its corresponding implementation in order to support architecture-driven development and the evolution of the relationships between versions of the architecture and the implementation. The main premise of the work is to allow development and architecture design to be achieved separately with synchronisation taking place at certain points in time. The aim of the approach is to allow stakeholders to use specific tools for specific tasks, such as an IDE for implementation and an architecture development environment for creating and updating architectural models. To realise this and to enable the parallel evolution of architecture and source code, the authors propose the integration of ArchStudio [5], Eclipse [6] and SubVersion [7] by creating an additional layer of infrastructure. Specifically, the communication between tools is achieved by creating an Eclipse plugin and an ArchStudio component that are linked, whilst SubVersion is integrated through the Subclipse client. This makes it possible to record changes to architectural models, thereby providing versioning capabilities. While the solution allows stakeholders to perform their tasks in dedicated tools, it is not obvious how much effort would be required to extend the system to work with additional tools and artefacts.

Collaboration and Artefact Versioning

Some holistic solutions in the subcategory of *Maintaining Separate Artefacts* attempt to provide artefact versioning and create links between them to support their evolution, or they provide a collaboration environment for distributed teams handling heterogeneous artefacts. Software projects are increasingly carried out in a distributed manner and development efforts are

[5]http://isr.uci.edu/projects/archstudio/
[6]https://eclipse.org/
[7]https://subversion.apache.org/

distributed across multiple locations [68]. This presents further challenges in evolving software projects.

ADAMS [69] is a web-based environment to address problems arising in cooperative development and to support coordination and collaboration in distributed software engineering teams. It integrates both project management (e.g. schedule management) and artefact management features (e.g. artefact versioning, traceability management and artefact quality management). *ADAMS* is divided into a number of subsystems, out of which the following two are of relevant in the current discussion. The *Artefact Management* subsystem manages the lifecycle of artefacts and allows the tracking of artefact evolution. It is characterised by a fine-grained management of software artefacts, which means that artefacts can be managed as either atomic entities or composites. *ADAMS* has been extended with a traceability recovery tool, *Re-Trace* to support software engineers in defining trace links using information retrieval [70]. Therefore, it currently offers a semi-automatic approach to trace link creation. The *Event and Notification* subsystem makes it possible for software engineers to be notified about changes provided that they had subscribed to the specific event. Since in any given system a high number of artefacts can exist, the proliferation of notification messages presents scalability and usability issues. *ADAMS* provides excellent support for creating trace links between artefacts of different types and for managing the evolution of artefacts through versioning. However, a major consistency management task, change impact analysis is not explicitly supported, and the maintenance of trace links requires manual effort.

In the dynamic process of software development, one way to manage evolution is by using Software Configuration Management (SCM) techniques. As defined by Scott and Nisse [71], SCM is a "discipline to identify the configuration of a system at distinct points in time" to allow the systematic control of changes and to ensure its integrity and traceability. It manages the evolution of software systems by controlling changes to a product throughout its lifecycle. Furthermore, it ensures that product components are accessible, their consistency is maintained, and that change requests, components and their status are recorded [14]. Traditional SCM systems follow a file-oriented approach meaning that a software system is thought of as a collection of files residing in directories of a file system. However developers and other participants of the development process are likely to consider the system as a collection of interrelated high-level abstractions. SCM systems are particularly suited to handle source code, while higher-level artefacts, which can also be managed as files, may contain elements that can cause issues with certain operations, such as merging [72]. In addition, SCM systems lack traceability support.

Molhado [73] intends to provide a solution to these issues in an effort to reduce the gap between software design and SCM. It is an infrastructure that supports the building of object-oriented

SCM systems and allows developers to model their software systems in terms of logical objects and relationships. The *Molhado* architecture makes it possible to integrate tools used to create and manage different artefacts with the resulting SCM system. Therefore in this SCM-centred development environment, changes to objects (i.e. artefacts) will be recorded. The fine-grained evolution of links among artefacts is managed by a versioned hypermedia infrastructure. Nguyen et al. recognise the significance of managing the evolution of architectural elements and system architecture parallel with source code. Taking a different approach from *Architecture Description Languages (ADL)*[8], which handle the planned evolution of architectures, *MolhadoArch* [74], the architecture-based SCM infrastructure and framework, manages versions of architectural entities by capturing explicit revisions. Users of the environment can use built-in editors to specify the implementations of architectural elements. Alternatively, source code can be imported. This provides means to connect architecture with source code artefacts through hyperlinks and to ensure their consistent evolution. However, it ties the user to the environment for accomplishing both design and implementation related tasks, making it less likely to be adopted in non-architecture-centric project scenarios.

In a software project a large number of files may exist with a possibly high number of relationships between them. To capture these relationships, hypertext can be used. Accordingly, a software project can be thought of as a "complex information artefact" [75] as opposed to the software configuration management view, where a project is composed of a collection of files. Hypertext - nonlinear text - refers to a mechanism that allows the navigation from one textual chunk to another and the ability to establish relationships between them [76]. It is the underlying notion defining the World Wide Web. An earlier mentioning of using hypertext in a software engineering context was in 1986. Delisle et al. [77] stated that the hypertext technology naturally lends itself to capturing relationships in a software development project as it provides storage for all the information associated with a software project also including a version history while allowing simultaneous access to project information.

An example of a hypertext-based approach is described by Taylor et al. [78]. The authors propose an open hypermedia approach, called Chimera, to provide hypertext services to support software development activities and capturing relationships among information objects. The aim of this work is to augment existing software development environments (SDE) with hypermedia services to manage links between a wide variety of objects. Such objects represent different views of a software system, such as a requirement, design or source code. The solution aims to complement existing tools used in software development. However, the integration of the approach with different tools may demand significant effort.

[8]https://www.sei.cmu.edu/library/assets/Survey_of_ADLs.pdf

Scacchi et al. [79] present the hypertext-based system, *Document Integration Facility (DIF)* to facilitate the management of systems and the documents produced during their development, use and maintenance. The supported documents include requirements specifications, functional specifications, architectural designs, source code, testing information and user and maintenance manuals. *DIF* considers segments of documents as objects, while relationships between them are considered as links. Objects are stored in files and as nodes of hypertext, while relationships between objects are stored in a relational database. The approach is specific to the listed artefacts and uses a prescribed documentation method to produce them. Therefore it is not applicable in a wide variety of software development scenarios.

The main premise behind the work of Nguyen et al. [80] is that software documentation plays a significant role in program understanding. Thus, to eliminate some of the factors hindering productive software development and to improve software documentation, a better interoperability between source code and documentation should be provided. The lack of this interoperability is illustrated by effort spent on software maintenance and bug fixing during which many different tools are used. Their proposed solution, the *Software Concordance (SC)*, is an integrated development environment, which aims to improve software document management using hypermedia technology. Both Java source code and documentation are represented in a tree-based, XML-compatible format, which is built on Fluid Internal Representation (Fluid IR). This allows not only for inline multimedia documentation within source code but also for hyperlinks between elements of documents. Artefacts and hyperlinks are versioned using the Molhado hypertext versioning system [73], and the fine-grained version control services are constructed from the Molhado object-oriented SCM infrastructure.

The *Ophelia* project [81] [82] [83] [84] provides a platform to support software engineering in a distributed environment. With the help of *Ophelia*, heterogeneous tools used in software development are integrated in the environment by defining standard interfaces described using *CORBA IDL*[9] definitions. The project concluded in 2003 with the release of a beta version and supported requirements, modelling, bug tracking, and project management tools. The Integrator component of the Ophelia architecture supports extracting data from heterogeneous tools through which the solution can be extended. However, the task may require manual user intervention. As part of the project a traceability layer and prototype tool, *Traceplough*, were created to support traceability across all project artefacts and to manage the change process: stakeholders are notified about changes and inconsistent artefacts can be synchronised. While artefacts are represented as *CORBA* objects, relationships are modelled by a graph structure allowing navigation among artefacts. By means of the graph model links can be processed and

[9]Common Object Request Broker Architecture (CORBA) Interface Definition Language (IDL). CORBA makes it possible for separate pieces of software created in different languages to work together as a single application [85]

used for impact analyses. However automatic impact analysis is not explicitly supported. With *Traceplough*, users can define and visualise trace links, which are established at the artefact level, which is rather coarse grained when used for further analysis. Potential shortcomings of *Traceplough* arise from the maintenance of the traceability graph and the proliferation of events associated with notifications.

Bruegge et al. note that distributed development is increasingly common in global companies and propose *Sysiphus* [86], an environment for distributed modelling and collaboration in which system models, collaboration artefacts and organisational models reside in a shared repository. System models represent the different software artefacts (such as requirements, detailed design, architecture), collaboration artefacts provide information about system models, such as change requests, and finally, organisational models illustrate relationships among participants of the project, and between the models that they create. The main premise of the approach is that information captured as a side effect of development can be used to resolve issues and to facilitate collaboration between stakeholders. The solution incorporates traceability to support manual change impact analysis activities. Tracing among artefacts is realised using explicit links manually set up by users and visualised in the environment. Changes are recorded at the artefact level, and stakeholders can subscribe to be notified of them. Similarly to *Traceplough*, *Sysiphus* alleviates collaboration issues associated with distributed software development. However, trace links are captured in coarse-grained manner, which results in a less sophisticated impact analysis solution.

The *Open Source Component Artefact Repository (OSCAR)* system [87] [88] is the artefact management subsystem of the *GENESIS* environment, which was designed to support workflow processes and the management of work products in distributed software engineering environments. *OSCAR* handles different types of data including users, process models, for example, and through its APIs allows the integration of open source applications to provide configuration management functionality. It also defines the notion of an active artefact, an artefact aware of and capable of recording changes it undergoes. Artefacts are represented by XML documents which associate meta-data, such as linking information, with file data. This flexible approach allows the addition of new artefact types by extending the base Document Type Definition (DTD). Although *OSCAR* was released, its functionality was reduced to basic integration in the *GENESIS* environment and to provide version control and annotate artefacts with meta-data.

The Advanced Process Environment Research (Aper) [89] is an example of a Process-Centred Software Engineering Environment (PSEE) [10]. The aim of Process-Centred Software Engineering

[10]Terms used in literature to refer to PSEEs include: Process Sensitive Environments (PSE), Process Support

is to produce high quality software by focusing on the quality of the software processes [90]. The development process is represented in process models (written in a Process Modelling Language (PML)) specifying how the development activities have to be carried out. PSEEs provide services for the execution of these models. Typical PSEE features therefore include automation of routine tasks and rule enforcement. Furthermore, they also provide collaborative tools and project management capabilities. PSEEs are relevant in this discussion as one of their central requirements is the ability to manage artefacts, and they control not only the software processes themselves, but also software products, which continuously evolve.

Aper is composed of the *Aper* language, the *Aper* compiler, the *Aper* server, the object management system (*Aper OMS*) and multiple *Aper* clients. The solution supports consistency management, process evolution and heterogeneous artefacts including design diagrams and requirement specifications. In contrast with the more coarse grained approaches described above, one of the main contributions of *Aper* is the decomposition of software artefacts and the creation and maintenance of both inter and intra product relationships. However it is not explicitly stated whether intra and inter relationships are generated automatically or manually. Ripple effect management functionality in *Aper* is based on relationships. Firstly, artefacts affected by a change are identified by trigger mechanisms. Secondly, potentially impacted artefacts are traced using relationships. Finally, consistency conditions are monitored to flag up violations. Despite its holistic approach of managing software artefacts, the solution and its functionality being specific to process centred environments is a hindering factor for wider industry adoption.

3.4.1.2 Combining Artefacts

Some solutions attempt to create a continuous link between different types of artefacts by embedding one in another to ensure consistency and to manage their evolution.

Donald E. Knuth stated that the intent behind a design decision bears the same importance as code itself, and he proposed literate programming [91]. It is an approach that supports the parallel evolution of source code (written in C) and its corresponding documentation by combining the two artefacts in a single document. When a change is made, the literate program, which is a mix of explanations in natural language and implementation code, is to be updated first, following which the code will be re-generated [92].

The above concept was extended by the open source project, *Intent* [93]. The motivating factor behind this approach is that, although documentation is an important aspect of application lifecycle management as it is used to capture design choices and their rationale, keeping

Systems (PSS) and Process Centred Environments (PCE) [90] documentation consistent with code is a substantial effort. Intent is integrated in the Eclipse IDE, and provides an environment for creating and editing documentation mixing formal and non-formal syntax, and tools to synchronise documentation with development artefacts. Its purpose remained similar to that of literate programming, although it attempts to achieve this goal in a more flexible way. Firstly, *Intent* follows the philosophy that each task (modelling, specification, implementation, etc.) should be carried out with its dedicated tool. This also means that the solution tries to cater for the different lifecycle needs of various types of artefacts. This is supported by inconsistency markers which indicate synchronisation issues and allow users to take the appropriate action in a timely fashion. Secondly, the set of supported technical artefacts is not constrained to source code written in a single programming language. In the event of a change, which is automatically detected, synchronisation errors are placed in the parts of the documentation that are related to the changed artefacts. Users can apply changes when appropriate, therefore change propagation is a manual effort. Intent, similarly to other solutions in this category, makes it possible to synchronise related artefacts, and it also allows software development tasks to be undertaken using their designated tools. However, in a distributed software development team the solution may not provide the most efficient means to ensure consistency.

Aguiar et al. present an integrated solution [94], which is different from literate programming in that one artefact is not embedded in another one. However, it shows similarities as the approach focuses on weaving contents from heterogeneous sources together. The aim of the approach is to solve the semantic consistency problem of software documentation and heterogeneous contents including source code and diagrams by proposing *XSDoc*, a documentation infrastructure based on wiki and XML technologies. Utilising the wiki frontend, users can integrate heterogeneous contents in the web-based documentation using the mark-up language of the wiki and by means of inlining or linking. Various plugins allow the integration of XSDoc with IDEs (currently Eclipse) and wikis. Although the solution is extendible by implementing new plugins to work with additional tools, the authors did not specify if it supports change detection, which is essential for impact analyses. Additionally it is not mentioned how links between the heterogeneous representations are maintained.

ArchJava [95] incorporates architectural descriptions into Java source code and thereby the structure defined by architecture and the implementation are unified in one language. This allows the consistency property *communication integrity* to be satisfied between architecture and implementation in an effort to contribute to architecture conformance. *ArchJava* includes new language constructs such as components, connections and ports to support the specification of software architectures. Murta et al. describe this approach as equality by definition [96]. The main shortcoming of *ArchJava* is that it is not realistic to assume that developers and architects use the same tools and notations to accomplish their design or development tasks.

3.4.2 Solutions Addressing Specific Aspects of Consistency Management

Solutions in the following category focus on specific tasks that constitute artefact consistency management discussed in Chapter 2. They encompass traceability and change impact analysis techniques, and approaches enabling consistency checking and change propagation. In this section solutions from their respective research fields are presented. A more extensive review of the individual fields is provided in surveys discussed in Section 3.2.4.

3.4.2.1 Traceability Techniques

Since an understanding of relationships among software artefacts leads to numerous benefits including support for impact analysis tasks for predicting cost and effort and visualising dependencies, traceability is one of the key building blocks of managing artefact consistency. Relationships between artefacts can be created in a manual, semi-automatic or automatic manner. One of the best known manual techniques is using requirements traceability matrices [97], which associate requirements with other artefacts, such as test cases, using identifiers. A requirements matrix may take the form of a separate document. Several requirements management, life cycle and general-purpose tools support traceability tasks [25], such as *IBM Rational DOORS* [11], which allows the manual specification and management of traces. Since manual trace capture is a tedious and error prone activity, over the past couple of decades various solutions have been proposed in the research community to alleviate this problem. Despite the availability of (mostly) proprietary tools to support trace creation and maintenance, automated traceability remains an open issue.

Below is a brief summary of the research efforts and results achieved in the traceability community that contribute to managing artefact consistency. Providing a comprehensive survey of the field is beyond the scope of this thesis. Out of the large body of work produced in the area, this thesis investigates fundamental techniques of automated trace creation since support for establishing links brings the automation of artefact consistency management a step closer. This also applies to approaches aimed at managing the evolution of trace links. These solutions are grouped into the traceability maintenance category. Besides a categorisation of traceability techniques originating from before 2006 and criteria for their evaluation [98], Cleland-Huang et al. provide an overview of automated traceability approaches and group them into categories based on the intelligence level of algorithms utilised [99]. The authors argue that an ideal

[11]http://www-01.ibm.com/software/awdtools/doors/features/traceability.html

industrial strength traceability solution is dependent on these algorithms and how they can handle the complexities of real world requirements.

Automated Trace Creation A subcategory of automated trace creation approaches, traceability recovery techniques, identify candidate links retrospectively from existing artefacts to perform after-the-fact tracing [25]. Qusef et al. divides traceability recovery approaches into three groups [100]: information-retrieval-based, heuristic-based and data-mining based solutions.

Information retrieval (IR)-based techniques are used to recover trace links between software artefacts of different types. Numerous software artefacts, such as bug reports, user manuals, wikis, build logs, test scripts, design documents, provide textual content. Artefacts, which show high textual similarity, are good candidates for establishing relationships between them [33]. Various tools implement information retrieval techniques, these include *Poirot* [101], *DrTrace* [102], *ReqSimile* [103], *TRASE* [104] (which uses Latent Dirichlet Allocation), *RETRO* [105], *ADAMS Re-Trace* [70]. Similarly, Antoniol et al. [33] propose a traceability recovery method based on IR to create links between free text documents and source code. The method has been applied to C++ and Java source code, and functional requirements. Maletic et al. [106] present a solution that is primarily focused on trace link recovery. However, the technique, which is facilitated by a formal hypertext model, uses conformance analysis and a timestamp strategy to support the checking of trace link validity during system evolution. The link recovery process between source code and documentation uses latent semantic indexing and is partially automated as in some scenarios user input is required. The hypertext model allows software documents to be represented as nodes of a network, while their relationships are modelled by links. *Traceclipse* [107] is a traceability recovery tool implemented as an Eclipse IDE plug-in. It provides semi-automatic traceability recovery, which is based on IR using Lucene. The user interface is wizard-based and allows users to manage trace links by viewing, editing and manually specifying them. Trace links are stored in XML format. Similarly, the *TraceME* [108] Eclipse plug-in is aimed at supporting traceability recovery using the Lucene IR engine. Traceability information is stored and managed in XML files, and the solution also provides a traceability graph to help users with their impact analysis tasks.

The second group of traceability recovery solutions leverage data mining techniques on software configuration management repositories, combining traceability recovery with mining software repositories. Kagdi et al. [109] utilise sequential-pattern and itemset mining to recover trace links using version histories. Following the identification of files that were committed together, change patterns and trace links can be derived. The approach works with source code, user documents, build management documents, and release documents.

An example of heuristic solutions is the approach proposed by Qusef et al. [100]. In order to recover trace links between test cases and source code and to overcome deficiencies of existing heuristics, such as naming conventions, the authors apply Data Flow Analysis. Similarly, Egyed and Grunbacher also discuss a heuristic-based approach in [110].

Other techniques include ontology-based solutions that aim to improve recall and precision values achieved by other techniques. For example Zhang et al. [13] apply text mining and an ontology-based approach to automatically establish links between source code and natural language documents. Hayashi et al. [111] propose a technique to recover trace links between documentation written in natural language and source code.

As opposed to traceability recovery, in situ trace link capture aims to establish trace links prospectively while artefacts are generated or modified. Most importantly, the trace capture happens in the background while users perform their tasks. Asuncion et al. [25] present a prospective trace capture technique, supported by the ACTS tool implemented on top of ArchStudio. Their technique allows the capturing of trace links between heterogeneous artefacts using rules. Additionally, with the help of notification adapters, changes to artefacts are checked (whether the traced artefact has been moved, deleted or revised), and trace links can be updated. The solution can be used with tools providing public APIs or built-in history logs in order to capture trace links between artefacts created by these tools. While there are a vast number of open source tools that allow this, some proprietary ones do not. Furthermore, an expert technical user is required to create and manage rules.

Spanoudakis et al. [112] describe a rule-based approach to automatically generate relationships between requirement statement documents, use case documents and analysis object models. The authors have identified four types of trace links between the above mentioned artefacts: *overlap*, *requires execution of*, *requires feature in* and *can partially realise* relations. The links are automatically created assuming that they satisfy the traceability rules associated with them. Additionally, two types of rules have been specified: requirements-to-object-model rules are used to generate links between requirements and analysis models, while inter-requirement traceability rules create relationships between requirement statement, different parts of use cases or between requirement statements and use cases. Both artefacts and traceability rules are expressed in XML.

The *TraceM* [113] conceptual framework is aimed at allowing stakeholders to derive implicit relationships between artefacts from explicit and existing links through a relationship mapping service. The approach builds on the enabling services of open hypermedia and information integration to provide the means to create, maintain, view and navigate between relationships.

Finally, recent traceability research focuses on improving the overall performance of tracing by creating new techniques or performance enhancement techniques, and by analysing and mimicking the way humans perform tracing tasks [114]. The problem therefore has been approached from new angles such as expert systems and artificial intelligence [115].

Trace Link Maintenance Besides link creation, it is equally important to update existing relationships because as systems evolve trace links may degrade, which leads to traceability decay. This is illustrated by scenarios where trace links represent false dependencies and may point to non-existent artefacts. While much research attention has been directed towards establishing links, their maintenance has been less extensively investigated [25]. Maintenance techniques assume already established relationships which allows the focus to be shifted to adding, editing or removing links [96]. Solutions discussed below contribute to artefact consistency management by supporting the evolution of artefacts and relationships between them.

Cleland-Huang et al. present the concept of *event-based traceability (EBT)* [116], which supports the maintenance of artefacts and their trace links. The approach is based on the Event Notifier design pattern and that the evolution of requirements can be described as a series of change events, which can be automatically identified by an event recognition algorithm. Events are categorised into change primitives (create, inactivate, modify, merge, refine, decompose, replace) [117]. Artefacts can subscribe to the requirements that they are dependent on and upon changes the requirements manager component publishes an event notification message along with information about the change. This message is received by the event server and is forwarded to dependent artefacts. This is an effective mechanism for detecting changes and for assessing their impact on dependent artefacts. For example to identify artefacts impacted by a change to a requirement, the user can request a forward trace to all the artefacts that subscribe to this requirement. Utilising information contained in change logs, "owners" of artefacts can manually resolve inconsistencies. However, as the authors describe, a potential shortcoming is revealed in large software projects where, due to the potentially large number of fine-grained traceability links, the published event notification messages may become unmanageable.

Mäder et al. [34] [118] present a rule-based approach to reduce manual effort in maintaining an evolving set of existing relationships and to address traceability decay. The technique is demonstrated and evaluated through the *traceMaintainer* prototype tool, which supports the semi-automatic maintenance of trace links following changes to structural UML models. The solution assumes that during the object-oriented development activity a model-based development approach is followed, a CASE tool is used to capture UML diagrams and a traceability information model is also in place. The maintenance process can be split into two major phases. The recognition phase captures elementary changes to model elements and recognises the development activity that is composed of these changes. For this reason, integration with a CASE tool is required. The second phase is responsible for updating the trace links affected by the change event. Each development activity can be related to rules, which are expressed in XML format. Rules recognise development activities and provide options on how to perform a given update activity. The authors performed two experiments to evaluate the prototype. The second experiment, which was carried out using a library management system, revealed that the manual effort spent on maintaining trace links can be reduced by 71%. The integration with a CASE tool allows the automatic capturing of change events, and manual impact analysis is supported through a dialog listing all existing and potentially new trace links involved in the given activity. A limitation of the approach is that only pre-defined development activities can be recognised, which makes the use of *traceMaintainer* limited in some scenarios.

ArchTrace [96] is an extensible infrastructure for semi-automatically maintaining traceability between an architectural description and its corresponding implementation. It was designed to be independent from the specific tools that are used to create and manage architectural descriptions and source code. The solution is characterised by proactive consistency management and provides an instant update mechanism, where trace links are continuously updated after each change. The updates are managed by traceability management policies, which can act as rules, to help decide which actions to take, or constraints, limiting the actions that can be taken. The tool's pluggable architecture allows the addition of new policies, however currently it supports ten policies that map architectural components, connectors and interfaces to their respective source code elements. *ArchTrace* does not provide automatic support for trace capture; it assumes the existence of trace links. Changes to artefacts are detected through triggers inserted into external systems, such as CM systems and architectural design environments. The triggers monitor changes made by users, and fire when these changes are committed. Users can visualise all architectural elements that are related to the selected source code item, which provides a way to manually analyse change impact. Similarly, the propagation of changes to architecture and configuration items is manual.

Asuncion et al. [119] adopt a process-oriented approach to support traceability that spans the entire development lifecycle. The solution is based on weaving artefacts together in tandem with the different phases of the software lifecycle. To demonstrate the concept, a prototype tool has been developed, which provides a means for trace link creation and support for software lifecycle activities. Traceability creation is performed manually and is aimed at minimising the overhead associated with trace definition. The solution also allows the updating of trace links manually through data entry forms. The prototype tool has been evaluated in an industrial setting. It aids architects in high level design tasks and enables them to integrate traceability with less effort in their development tasks.

In summary, automated trace creation and maintenance techniques contribute to the management of artefact consistency as one of its key elements is the existence of relationships between artefacts. The majority of solutions are based on IR with recall values measuring coverage ranging between 90-100%, and precision values indicating accuracy ranging between 5-30% [114]. This presents a major hindering factor for industry adoption. Additionally, as pointed out by Bashir's study [98], automated techniques cater for a small subset of artefacts.

3.4.2.2 Change Impact Analysis (IA) Approaches

Automated support for assessing the consequences of modifications is one of the backbones of artefact consistency management, since identifying a potential impact set facilitates consistency checking and change propagation. A summary of the main concepts and some relevant IA solutions is presented below.

As reinforced by Lehnert's findings [42], most IA solutions concentrate on analysing the ripple effects of changes impacting source code elements. That is, determining which source code entities are likely to be affected when another source code element changes. To answer this question, techniques ranging from call graphs [120], dependency analysis [121] and probabilistic models [122] to history mining [123] and their combinations [124] have been proposed. These solutions are characterised by disparate accuracy and coverage values [125].

The abundance of techniques proposed to automate source code impact analysis provides a useful starting point for approaches handling heterogeneous artefacts. However, these techniques address only a part of the artefact consistency management problem. It is an interesting question waiting to be answered if and to what extent particular source code specific techniques could be used in the scope of diverse artefacts. For example, dependencies are extensively used in static source code IA techniques, which is similar to the notion of using trace links between different types of artefacts in estimating the consequences of a change. Equally, change coupling detection could be extended to non-source code artefacts assuming they are stored in version control.

An example of change impact analysis solutions catering for non-source code artefacts is the work proposed by Dantas and Werner [126]. The aim of the approach is to detect UML model elements that changed together in the past using mined association rules and the Apriori algorithm. For this reason a versioned UML repository is utilised and the authors present their work through a use case, a class and a component diagram. Kotonya et al. support change impact analysis at the requirement level using traceability information and probability values [127]. Lee et al. propose a goal-driven requirements traceability approach combined with an analysis of requirements change impact [128].

A number of solutions extend IA to multiple artefacts. For example, Briand defines a methodological framework and a rule-based approach for change detection and impact analysis to facilitate the change planning process [129]. The authors indicate that in order to provide support for impact analysis specific to UML models, the following prerequisites have to be satisfied: automatic change detection and classification, verification of the consistency of changed diagrams, suitable technique for impact analysis and an assessment and prioritisation of results. Thus, a change taxonomy consisting of 97 change categories has been created besides 120 consistency rules specified using OCL. The prototype tool, *iACMTool*, reads two versions of a UML model and produces an impact analysis report.

Finally, Ibrahim et al. recognise that traceability across high and low level software artefacts, such as design, test cases and code, and catering for impact analysis are essential factors in determining the ripple effects of changes following a change request [130]. Their approach is evaluated in a case study and by using the *Catia* prototype tool, which highlights the differences in change impact analysis results obtained for pairs of the different types of artefacts.

3.4.2.3 Consistency Checking and Change Propagation Approaches

A number of consistency checking solutions have been proposed in literature. This section briefly introduces the main concepts and groups of solutions contributing to artefact consistency management. Finkelstein discusses the technical challenges in consistency management [131], in an effort to analyse the requirements for constructing consistency management tools capable of handling heterogeneous and distributed information.

The discipline of Model Driven Engineering (MDE) aims to develop and maintain software through model transformations [132]. Using rules, a transformation translates a source artefact to a target artefact, both of which can be of different types ranging from concrete representations to more abstract models. Reviewing a taxonomy of transformations is beyond the scope of this thesis. However, it is important to mention that two specific types are directly relevant in the discussion of artefact consistency management. Firstly, synthesis turns a more abstract representation (e.g. design model) into a more concrete, lower level one (such as source code), a typical example being code generation, which can be classified as forward engineering. Secondly, reverse engineering provides an abstraction of higher-level artefacts from lower-level ones. Forward and reverse engineering together constitute round-trip engineering (RTE) [133]. Generative and transformational techniques are relevant since they implement means to handle the synchronisation (change propagation) of heterogeneous artefacts to re-establish consistency between them following a change [134]. Synchronisations also utilise relationships between artefacts, called mappings.

Examples of code generation facilities include the Visual Studio class designer [135], which was created with the intention to keep source code and class diagrams synchronised. The IDE [12] also provides the capability to generate source code from a class diagram and vice versa, where changes made to a class diagram are reflected in the source code, and alterations of the source code are also synchronised to the diagram. Additionally, the Eclipse plug-in, Objecteering [136] guarantees Java code-UML model consistency and Enterprise Architect's template driven code generation engine [137] allows its users to perform forward engineering tasks.

Furthermore, the open source framework and code generation facility, the Eclipse Modeling Framework (EMF) [138], facilitates the generation of different representations of an application, such as source code written in Java and UML. Relatedly, the main aim of Executable UML [139] [140] is to use models of the system to directly execute the system. These models are complete enough to be executed. The Foundational UML (fUML) standard has been created to specify precise semantics for an executable subset of standard UML. Another key ingredient of Executable UML is the Action Language for Foundational UML (Alf), which provides a textual notation for UML behaviours that can be attached to a UML model and allows the specification of detailed behaviour. Executable UML allows its user to program, at a higher level of abstraction. Finally, various commercial tools provide round trip engineering functionality. Poseidon for UML [141], which can also be run integrated within Eclipse, makes it possible to maintain consistency between UML models and Java code. A major benefit of code generation solutions is their support for software development activities and enhancing developer productivity by reducing time spent on routine programming tasks. However, outside the scope of some specific application domains, such as parser generators [24], applications of the approach are limited and generating a complete functioning program that does not require human intervention remains a distant goal. Another interesting question these solutions raise is the level of detail the various design models can capture, and how accurately models can be mapped to an implementation. Although ADLs and UML also support some code generation, these solutions are not widely adopted in industry.

Some consistency checking solutions address more general consistency issues, independent from the actual domain of the models and not specific to MDE [52]. A substantial amount of work has been done in checking the consistency of both heterogeneous and specific artefacts. As an example of the first category, Nentwich et al. have developed xlinkit [142], a consistency checking service that is based on the XML, XLink and XPath technologies. The framework is implemented

[12]https://www.visualstudio.com/en-us

as a web service and supports the management of the consistency of software specifications. This is achieved in a tolerant manner: the solution does not force the immediate resolution of inconsistencies as they are not always undesirable. The aim is to pinpoint inconsistencies so they can be handled by document owners when appropriate. Elements are connected by hyperlinks. One of the main contributions of this work is the creation of a set-based rule language, which is a restricted form of first-order logic expressing consistency constraints between distributed documents. Campbell et al. [143] provide an example of the second category of solutions. Their proposed approach checks inter-diagram structural and syntactical inconsistencies between UML diagrams of different types and detects structural problems within individual diagrams. The results of consistency checks are then presented to users who can address the issues manually. Additionally, Dimech and Balasubramaniam propose an automated approach to check conformance between Java source code and architectural models in UML [144]. The approach is implemented in an Eclipse plug-in, Card.

3.5 Evaluation

As mentioned at the beginning of this chapter, the aim of the evaluation of related approaches is to establish to what extent the issue of artefact consistency management has been addressed and to highlight which specific aspects have been in the focal point of research areas introduced in the survey.

The review of solutions and an analysis of the problem area revealed that the management of the consistency of software artefacts consists of multiple activities: a) managing changes requires mechanisms to detect modifications, b) it is essential that heterogeneous artefacts are linked and c) the impacts of any artefact being modified is assessed. Following these steps d) consistency checks can be carried and finally, e) consistency can be-re-established. The concept of this holistic view of artefact consistency management is discussed in detail in Chapter 4, however the approach is also utilised during evaluation, where these aspects allow the pinpointing of facets of the problem that have been more thoroughly investigated and automated compared to others.

A major component of the evaluation is a comparison, illustrated by Table B.1, B.2, B.3 and B.4 in Appendix B, which is carried out using a subset of the reviewed solutions. The approaches selected were primarily the ones where tool support is available and ones where more than one aspect of artefact consistency management is applicable. Thus, for example traceability recovery solutions were not considered. However, a review of such solutions contributes to drawing conclusions about automation levels and artefact coverage. The comparison provides information about automation levels by analysing which aspects of artefact consistency management are catered for. Additionally, the scope of solutions in terms of supported artefacts is highlighted, and it is also investigated if distributed development is supported. Finally, the analysis of artefact storage and link storage provide insights into implementation level specifics of the given solution.

The issue of inconsistently evolving artefacts has not yet been sufficiently addressed in its entirety. The solution space for managing artefact consistency is diverse and the individual solutions approach the problem from various angles, which is a hindrance to performing a direct comparison. Notably, solutions stem from different research areas and their motivation and aims are significantly disparate: some may focus on trace link creation for a better comprehension of the system, while others may have the priority to minimise the side effects of modifications.

The evaluation did not identify any solutions, which cater for all aspects of consistency management providing traceability creation and maintenance, change detection, impact analysis, consistency checking and change propagation. In most cases, a subset of these aspects is satisfied, either in an automated, semi-automatic (parts of the process can be automatic and mostly user input is required) or manual manner (the solution offers data or visualisation but the task is performed by the user).

In terms of supported artefacts, the evaluation revealed that most solutions concentrate on specific artefacts. This applies, in particular, to traceability-specific solutions and in cases where proof-of-concept tools have been developed to work with a subset of artefacts. A similar categorisation of solutions based on artefact types is discussed in *Software Traceability: A Roadmap* [35].

The automation of any aspect of artefact consistency management is a challenging problem and the reviewed solutions provide varying degrees of support for it. This problem is exacerbated when a given solution combines more than one aspect, and it is also impacted by which artefacts are catered for. It is apparent how strong of a link there is between the adoptability and success of an artefact management solution and how automated it is.

The implementation specifics analysed provide a further dimension to discovering differences between the approaches. Specifically, the results highlight that in terms of artefact representation solutions can be divided into two main categories: approaches that represent artefacts in an intermediary format, such as XML, and approaches that process artefacts in their original format.

3.6 Conclusions

This survey provides an overarching review of solutions that contribute to artefact consistency management and that are rooted in various research areas. The evaluation of related approaches and a summary of their limitations contribute to formulating the requirements of an artefact consistency management solution (presented in Chapter 4) and they also facilitate the definition of the directions of this research.

One of the focuses of this work is the exploration of possible ways to cater for all types of present and future artefacts, which was derived from the findings of the evaluation, namely that typically only a selected set of artefacts is supported by any given solution. As traceability stems from the requirements engineering community, traceability issues related to requirements have been extensively researched [25]. In MDA, the software development process is driven by modeling [19]. Agile and incremental software development processes emphasise the importance of coding or low level design. A possible solution that concentrates on a specific type of artefact cannot be comprehensive enough as it will not be applicable in a large number of scenarios and such a solution will not span the entire development lifecycle. Another area worth investigating is the level of automation as existing approaches automate some but not all aspects of artefact consistency management. Finally, since most solutions do not cater for all aspects of artefact consistency management, to more effectively address the challenges arising from the differential evolution of software artefacts, a holistic solution is required.

CHAPTER FOUR

HOLISTIC ARTEFACT CONSISTENCY MANAGEMENT FRAMEWORK

This chapter introduces the concept of a holistic artefact consistency management framework proposed in this thesis to address challenges arising from the differential evolution of software artefacts. It presents these challenges and describes the conceptual foundations of the framework including the stages of managing artefact consistency. The chapter concludes with a discussion of the selected data representation model, the property graph, and the individual framework stages.

4.1 Challenges of Artefact Consistency Management

Artefact consistency management is a complex problem due to the inherent characteristics of software development. The tasks involved in a software project are undertaken by various stakeholders who possess different skills, have exposure to different parts of the system, and carry out their work using different tools, potentially at different geographical locations. Each phase of the development produces various artefacts, which differ in their abstraction level and purpose. Additionally, the nature of the project determines the development process, the structure of teams, stakeholders involved and methodologies applied. Such diversity presents significant challenges to artefact consistency management efforts. These difficulties are considered below.

Diversity of Artefacts

Software development activities produce a number of artefacts, and any artefact at any abstraction level can take multiple forms. This is well illustrated by the different types of UML diagrams, such as class and sequence diagrams modelling different aspects of a system. This complexity in artefact types poses one of the key challenges associated with building an artefact consistency management framework.

Artefacts produced in software development are dependent on the adopted methodology. For example, agile projects, besides source code, produce artefacts that are mostly related to tracking work progress. These may include product backlogs, sprint backlogs (which in turn may contain user stories, goals and tasks associated with the given sprint), and product roadmaps [145]. On the other hand, traditional development methodologies are associated with another set of artefacts. For example, following a waterfall approach may produce a feasibility document, requirements specification, design documents, source code, and various tests [146]. Adopting a spiral model of software development may also produce documents highlighting the results of risk analysis [147].

Another factor influencing artefacts is the project domain. For example, the development process of mission critical systems may be regulated by safety regulations and industry standards [148], which results in rigorous documentation and formal specifications.

Finally, open source projects, which provide direct access to their artefacts, are also worth mentioning. Artefacts stored in open source repositories may include wikis or user manuals, as demonstrated by the systems discussed in Chapter 8.

Diversity of Tools

Software life cycle tasks are undertaken by stakeholders using a variety of CASE tools. Integrated development environments (IDEs) provide support to produce source code and tests. Higher-level artefacts are created using diagram and analysis tools. An ideal framework, to maximise its applicability in software projects, should not impose any specific application on the user and should be configurable to work with any tool.

Usability

Participants of software projects possess different skills and work with different artefacts. Some stakeholders, such as business analysts, may be more concerned with high-level representations, while others, such as developers and testers, may work with source code and tests. This results in different needs that must be accommodated by an artefact consistency management framework.

Customisability is thus an important characteristic. Examples of customisable properties include the level of detail and frequency of changes users want to be notified about.

Automation

Tasks associated with artefact consistency management, when performed manually, are error-prone, tedious and require substantial effort. Some aspects of artefact consistency management can be more easily automated, such as checking consistency violations, while others may present non-trivial challenges. For example identifying relationships between diverse representations is a complex task due to the heterogeneity of artefacts and the fact that semantics and intentions are not explicitly captured. The extent to which automation is possible is an open problem.

Distributed Software Development

Software development is increasingly carried out in a distributed manner [149] with stakeholders based in different geographical locations. Issues caused by this trend are related to knowledge management, quality control, synchronous collaboration, and risk, project and process management concerns. Specifically, global software development, where teams are distributed worldwide, introduces an additional level of complexity to artefact consistency management tasks. In this respect, areas of concern are multiple versions of artefacts, creating and maintaining links among distributed artefacts, and the availability and accessibility of the latest version of any given artefact. Artefact repositories and version control systems are often used to mitigate the effects of distribution.

4.2 An Ideal Consistency Management Framework

The attributes of an ideal consistency management solution and areas that have not yet been sufficiently addressed in research provide a basis for deriving the high level requirements of a viable artefact consistency management framework. Notably, catering for all aspects of artefact consistency management remains the main open challenge along with the lack of support for a diverse range of artefacts in one solution. Full automation is an ideal characteristic of such a framework. The discovery of a viable level of automation is highly desirable in supporting both artefact evolution and framework adoptability in real project scenarios.

Requirements of a workable framework for consistency management, such as extensibility and the ability to cater for a wide range of software representations, have already been expressed in literature [9]. Incorporating those characteristics, this work formulates the following key requirements:

R1 - Automated as far as possible. In order for the framework to be adopted in software projects and to reduce manual effort, it should provide automated support for consistency management tasks.

R2 - Artefact independent. The ideal framework should be able to cater for different types of artefacts instead of being limited to a particular subset of them.

R3 - Methodology and Tool independent. The ideal solution should take it into account that software artefacts are created and edited in a wide variety of tools. It should work with both new and existing tools and should support seamless integration into any environment. Stakeholders involved in the development process should be able to follow their usual work processes and the solution should be usable in different development settings. Both R2 and R3 express the requirement of extensibility.

R4 - Customisable and non-intrusive. An ideal framework should provide means for configuring and setting user preferences. Additionally, it should be non-intrusive to user practices and it should require minimal interaction.

R5 - Supports distributed development. With global teams being prevalent in software development projects, an ideal solution should provide support for addressing challenges arising from the distributed nature of software development.

R6 - Is able to handle variable numbers of artefacts and changes (Performance). Software systems differ in their size and complexity. An ideal solution takes this diversity into account and can effectively handle a varying number of artefacts and changes of different complexity.

4.3 Proposed Approach: Concept of a Holistic Artefact Consistency Management Framework

4.3.1 Definition

Following an in-depth review of state-of-the-art approaches addressing facets of the artefact consistency management problem, this work hypothesises that the problem of the differential evolution of heterogeneous software artefacts can be satisfied with a comprehensive framework which incorporates the following key stages:

- Traceability creation and maintenance,
- Change detection,
- Impact analysis,

- Consistency checking, and
- Change propagation

The holistic framework concept is depicted by Figure 4.1, where rounded rectangles denote the framework stages, the rectangle represents *Artefact Data*, block arrows show control flow, and line arrows capture data flow. *Artefact Data* represents elements of the different artefacts of the system and their interconnections. The *Trace Link Creation* stage encompasses functionality to create relationships between artefacts. The resultant trace links form an essential component of any consistency management solution as they allow the discovery of dependencies, which can be utilised by other stages of the process. Artefacts may be prone to frequent changes, therefore the *Change Detection* stage provides means to identify parts of the system that have been modified. This change data is subsequently passed to the *Change Impact Analysis* stage, which discovers the consequences of the identified change and builds a set of potentially affected artefacts by leveraging the established trace links. Potentially affected artefacts serve as an input to *Consistency Checking*, which establishes whether elements of the set are inconsistent as a result of the change. Lastly, *Change Propagation* is responsible for carrying out modifications to re-establish the consistency of affected software artefacts, which is similar to Han's definition of change propagation [54].

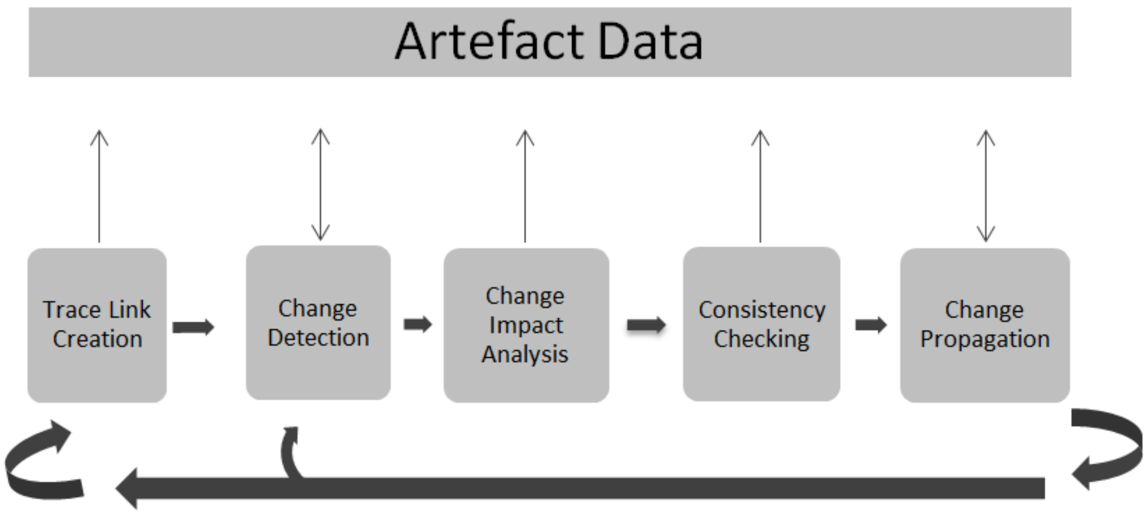


Figure 4.1: Holistic artefact consistency management theoretical framework process diagram.

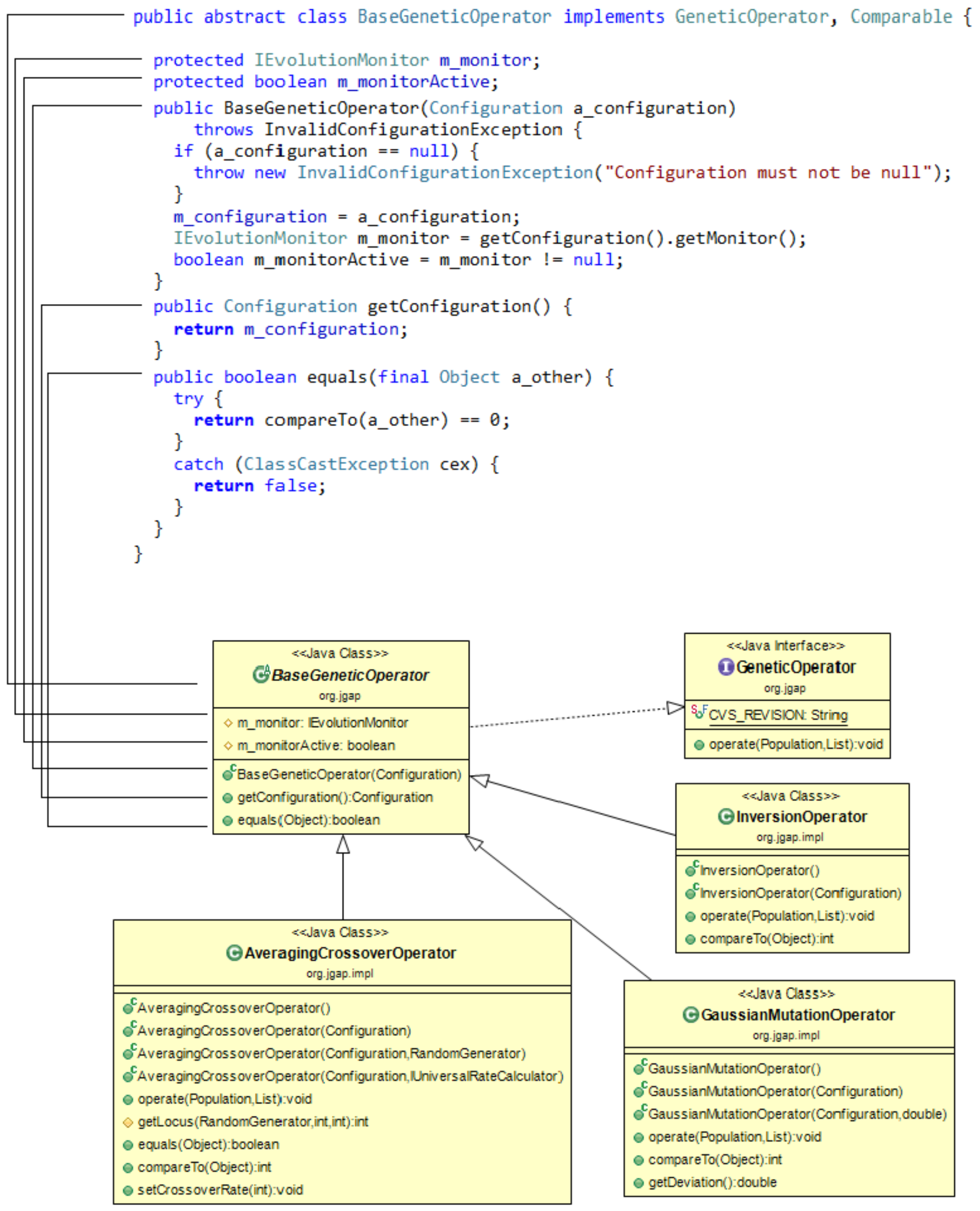
4.3.2 Illustrative Example

Following is a concrete example depicting a basic scenario to demonstrate aspects of the holistic framework. The example is extracted from an open source system, JGAP[1], which is introduced in detail in Chapter 8. Figure 4.2 shows an excerpt of the class diagram of JGAP, generated based on a subset of classes obtained from the code repository, and its corresponding Java source code. The *GeneticOperator* interface on the UML diagram represents an operation that takes place on a population of chromosomes during the evolution process. It is implemented by the *BaseGeneticOperator* class, which is extended by the *AveragingCrossoverOperator*, *GaussianMutationOperator* and *InversionOperator* classes. The Java source code excerpt in the image shows the corresponding implementation of the *BaseGeneticOperator* UML class, i.e. the *BaseGeneticOperator* Java class and its members, the *m_monitor* and *m_monitorActive* fields, and the *BaseGeneticOperator*, *getConfiguration* and *equals* methods.

The lines connecting Java source code and UML class diagram entities represent trace links between elements of these two artefact types. For the purpose of this illustration, trace links denote a connection between a UML diagram element and its corresponding source code element. For example, as shown in Figure 4.2, a link is established between the *getConfiguration()* method contained on the class diagram in the *BaseGeneticOperator* class, and its corresponding implementation, represented by the getConfiguration() Java method. Trace links are created by the *Trace Link Creation* framework stage shown on Figure 4.1. *Artefact Data* provides means to store artefact elements and trace links. In this example it comprises the classes, interfaces, methods, and fields outlined in the class diagram, and implemented in Java source code, coupled with their interconnections.

Let us presume that the *BaseGeneticOperator* Java class is updated with a new constructor, *BaseGeneticOperator*. The *Change Detection* stage is responsible for flagging up the modification and for providing details necessary for subsequent stages. *Change Detection* also involves the updating of the data store and *intra* trace links, that is, it also performs traceability maintenance. Based on the change, the *Change Impact Analysis* stage forms a set of potentially impacted elements linked to the modified entity. One such element is the *BaseGeneticOperator* class on the UML diagram, since it is connected to the *BaseGeneticOperator* Java class. *Consistency Checking* determines if the *BaseGeneticOperator* UML class is now inconsistent due to the modification. According to the present example it is inconsistent, therefore *Change Propagation* resolves the inconsistency by propagating the appropriate modification by adding the *BaseGeneticOperator* method to the UML diagram. This step also involves the maintenance of inter trace links. Thus, a link is established between the *BaseGeneticOperator* UML and Java

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



```java
public abstract class BaseGeneticOperator implements GeneticOperator, Comparable {

  protected IEvolutionMonitor m_monitor;
  protected boolean m_monitorActive;
  public BaseGeneticOperator(Configuration a_configuration)
      throws InvalidConfigurationException {
    if (a_configuration == null) {
      throw new InvalidConfigurationException("Configuration must not be null");
    }
    m_configuration = a_configuration;
    IEvolutionMonitor m_monitor = getConfiguration().getMonitor();
    boolean m_monitorActive = m_monitor != null;
  }
  public Configuration getConfiguration() {
    return m_configuration;
  }
  public boolean equals(final Object a_other) {
    try {
      return compareTo(a_other) == 0;
    }
    catch (ClassCastException cex) {
      return false;
    }
  }
}
```

<<Java Class>>
BaseGeneticOperator
org.jgap
m_monitor: IEvolutionMonitor
m_monitorActive: boolean
BaseGeneticOperator(Configuration)
getConfiguration():Configuration
equals(Object):boolean
<<Java Interface>>
GeneticOperator
org.jgap
CVS_REVISION: String
operate(Population,List):void
<<Java Class>>
InversionOperator
org.jgap.impl
InversionOperator()
InversionOperator(Configuration)
operate(Population,List):void
compareTo(Object):int
<<Java Class>>
AveragingCrossoverOperator
org.jgap.impl
AveragingCrossoverOperator()
AveragingCrossoverOperator(Configuration)
AveragingCrossoverOperator(Configuration,RandomGenerator)
AveragingCrossoverOperator(Configuration,IUniversalRateCalculator)
operate(Population,List):void
getLocus(RandomGenerator,int,int):int
equals(Object):boolean
compareTo(Object):int
setCrossoverRate(int):void
<<Java Class>>
GaussianMutationOperator
org.jgap.impl
GaussianMutationOperator()
GaussianMutationOperator(Configuration)
GaussianMutationOperator(Configuration,double)
operate(Population,List):void
compareTo(Object):int
getDeviation():double

Figure 4.2: JGAP system class diagram and source code excerpt.

methods. The framework then returns to the *Change Detection* stage.

4.3.3 Real World Applicability of the Holistic Approach

As mentioned in Section 4.1, software development is characterised by a great degree of diversity. As no universal methodologies and techniques exist that are suitable for all projects, a number of methodologies have emerged [12]. Although a key requirement of the proposed holistic approach is methodology and tool independence, considering the diversity of software development projects, some projects may benefit from adopting such an approach to a greater degree than others.

Considering project methodology and processes, the approach may provide benefits to stakeholders in projects where the need to create and maintain trace links is recognised, and is perhaps enforced by process improvement models, such as CMMI. Another influencing factor is the project domain, which also affects the development methodology. For example, in developing safety-critical control systems, which are characterised by formalised quality assurance and a rigorous analysis of the requirements, plan driven approaches are more suitable. In such systems, various documentation, design and test artefacts are produced, and change management processes and controlling the affects of changes are essential elements of the development process. In such scenarios adopting the consistency management approach contributes to alleviating change management efforts. Finally, projects can also be categorised based on their size. Large systems are also characterised by a diverse set of stakeholders who may produce and utilise different artefacts to understand the system. Such cases highlight the need to keep heterogeneous representations consistent, which provides software maintenance benefits.

4.4 Data Representation

The remainder of this chapter outlines overarching aspects of the holistic approach before the architecture and design of the prototype framework are discussed in Chapter 5. Such aspects include the data representation model and a discussion of each framework stage from a conceptual point of view.

Artefact Data, a core aspect of the framework, consists of artefacts and their trace links, supplying an input for stages of the framework. As defined in Chapter 2, software artefacts are products of activities involved in software development projects and are characterised by heterogeneity, which constitutes one of the main challenges for artefact consistency management. Based on the description of holistic artefact consistency management, it is apparent that one of the backbones of the approach is the existence of trace links between heterogeneous artefacts. Links allow subsequent framework stages, such as change impact analysis and consistency checking, to take place.

4.4.1 Conceptual Data Model

One of the pivotal aspects of the framework is selecting the most suitable model to represent *Artefact Data*, which contains heterogeneous artefacts and trace links connecting them. The aim of any representation is to capture information contained in artefacts and to allow operations to be performed on it in an effective manner. The first step of the investigation is the analysis of the characteristics of framework artefact and trace link data. These are summarised as follows.

- **Structure.** Some artefacts exhibit a hierarchical structure, while others do not. For example, in source code there may be a hierarchical relationship between elements. However, in a requirements specification document each requirement may bear the same significance and there may not be a root element.
- **Changeability.** Framework data is prone to continuous changes.
- **Significance of connections.** Framework data is highly connected and relationships between artefacts are as important as the elements connected by these relations.
- **Heterogeneity.** Numerous types of artefacts and artefact elements exist and are handled by the framework.
- **Specific element details**. Due to the heterogeneity of framework data, each element can be described by a varying number of attributes.

Additionally, at a high level, a data representation model should fulfil the following requirements:

- **Extensible.** In case new elements or specific details are added, the existing representation should be extendable to include these.
- **Can be queried.** Information about elements and their relationships should be available and mechanisms to reveal them through queries should be in place.
- **Modifiable.** Modifications of existing data can be carried out.

The listed attributes are the driving factors in choosing the most suitable representation. As part of the selection process a number of alternatives have been considered. These are introduced in the section below, along with the selected data model, the property graph structure.

4.4.1.1 Property Graph Structure

Graphs are applied in numerous fields, such as biological systems, neural networks, social networks, and the Internet, as they provide powerful means to represent interconnections. This ubiquity is illustrated by the emergence of network science [150] [151].

Specifically, graphs have been extensively used in computer science, and in particular in software engineering, in a number of problems: graph-based representations of object-oriented programs (such as class dependency networks, and subroutine call graphs) have been investigated to support the various phases and tasks involved in software development, such as software maintenance and indicating software quality [152] [153]. Other specific graph representations include control flow graphs and program dependence graphs developed in the fields of program analysis and compiler design [154]. Data flow graphs have been used in software testing [155], and the code property graph - combining abstract syntax trees, control flow graphs and program dependence graphs - has been applied in discovering software vulnerabilities [154]. In particular, out of the solutions discussed in Chapter 3, *MolhadoArch*, for example, applies a graph representation where the architectural structure of a system is modelled as a directed graph [74].

At a high level, graphs are collections of connected objects, which can be used to represent real world systems in the form of nodes (objects) and edges (their connections). A graph G is formally defined as a pair of sets *(V,E)*, where *V* is a set of vertices (nodes) and *E* is a set of edges between pairs of vertices [156]. Using a set based notation, an example set of vertices can be defined as *V = v1, v2, v3, v4, v5*, where the graph consists of five nodes. Edges are formed as follows: *E = (v1, v2), (v2, v3), (v3, v4), (v4, v1), (v5, v1).*

Heterogeneous artefacts and trace links between them in the framework are modelled as a graph structure. Artefact elements constitute graph nodes - each v node of the vertex set *V = v1,... vn* is an artefact element, where n is the number of these elements. Trace links are represented by edges of the graph. The set of edges, *E = e1, ... en* consists of connections between the n artefact elements. An important trait of the graph model used by the framework is that it associates attributes, i.e. a number of key/value properties, with both nodes and edges. This is called the **property graph**, which is a powerful knowledge representation model [157]. Edges have directions associated with them, however relationships can be navigated regardless of the direction. Applying a property graph model provides an approach both to represent parts of the data explicitly using the graph structure, and to infer implicit knowledge through traversals [158]. Figure 4.3 illustrates the structural elements (nodes, edges and properties with their values) of the property graph.

Given this description, a definition of framework data elements expressed as property graph concepts is now presented.

Artefacts: are the original representations of a system, which are high-level types. Artefact examples include source code, UML class diagram, unit tests. Artefacts are not explicit graph entities, but can be inferred from graph node properties.

Artefact elements: are entities that build up a given artefact (type). An example artefact element

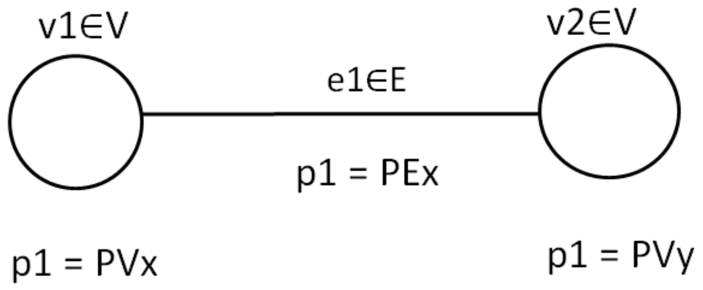


Figure 4.3: Illustration of the property graph model.

is a source code element, such as a *method* or an *interface*. These elements are represented by nodes of the graph, i.e. a set of vertices may contain the following specific artefact elements: Java class, Java method, UML interface, UML attribute, etc.

Artefact element attributes: are properties associated with graph nodes. The number and type of attributes are dependent on the given node. For example a Java method node will have a *parameters* property linked to it, whereas it is not applicable for a Java field, which can be described by other attributes, such as *variable_type*.

Trace link attributes: are properties associated with graph edges. This makes it possible to differentiate trace links in the framework. For example a link can have a *type* property, which may take the value *Inter_link* for an *inter* trace link.

4.4.1.2 Alternative Artefact and Trace Link Representations

Existing approaches in traceability and artefact consistency management research have introduced a number of artefact representations. Some solutions keep the original representation of artefacts [91][93], while others use XML [119] [142], hypertext [106], EMF-based models [62] and the Resource Description Framework (RDF) [159]. Trace links may be stored in numerous ways, including XML [62] or relational databases [9]. Some data representation models are specific to artefacts, whereas others, such as XML, can be generalised to heterogeneous artefacts. An XML-based solution satisfies the extensibility, modifiability and query requirements of an effective data model for representing artefact data. However, it is not the most practical solution as artefact data is prone to changes, and the resulting updates to the XML schema would necessitate further changes in the framework.

4.4.1.3 Evaluation of the Property Graph Model

The requirements of framework data are satisfied by the property graph model. Firstly, it supports the modelling of heterogeneous data, and it also captures the required level of detail, thus making it possible to express the richness of data. Most importantly, the graph representation reflects the nature of artefact and trace link data, i.e. that connections between elements are of vital importance in the framework. Finally, another advantage of a graph representation is the possibility of utilising graph algorithms. Table 4.1 summarises each aspect and discusses how the property graph satisfies it.

Requirement	Discussion
Structure	It is possible to represent hierarchical and non-hierarchical data using property graphs.
Changeability	Graph nodes and graph node property values can be added, modified and deleted.
Significance of connections	Graphs allow the sufficient modelling of both entities and relationships between them.
Heterogeneity	The property graph allows the modelling of heterogeneous entities through its properties.
Specific entity details	Each entity on a property graph can be described by different properties.

Table 4.1: Evaluation of the suitability of the property graph model for artefact data representation.

Table 4.2 shows that a wide range of artefacts can be modelled using property graphs, and provided that means to extract data from the original artefacts is available, property graphs present a viable solution to model any artefact. A detailed discussion of the specification of graph nodes and their properties pertaining to concrete artefacts is provided in Chapter 6.

4.4.2 Bridging the Gap Between Heterogeneous Artefacts and the Property Graph Model

Artefacts outside the framework are stored in their original tools in formats specific to the given tool. Artefacts inside the framework are represented by a property graph. To bridge the gap between diverse formats and the property graph, the approach of extracting artefact data from the original representations to a generic intermediary format was adopted. The implementation specifics of the approach are described in Chapter 6.

The intermediary format is used as a means to obtain artefact and trace link data and not as a means of representation. The extracted data at this stage is still heterogeneous due to structural

Artefact Type	Methodology
Java source code	Transform XML representation of source code
Python source code	Transform AST [2]
A, UML Behaviour diagrams: use case diagram, state machine diagrams	
B, UML Structure diagrams: class diagram, component diagrams	Transform underlying XML representation of UML diagrams
C, UML Interaction diagrams: sequence diagrams	
Architectures	Transform XML representation of architecture diagrams
Requirement specifications (natural language) stored in documents (e.g. Word)	Transform XML representation of documents [3], [4]
Requirement specification (natural language) stored in wikis	Use plugins available to extract content [5]
jUnit test classes	Transform XML representation of unit test classes
Issues in issue tracker	Use APIs to access issue trackers, such as [6]

Table 4.2: Artefacts to be represented in the framework by a property graph model.

differences between artefacts. Thus, a process is required to perform the transformation of data stored in the intermediary format to the unified representation, that is, to graph nodes, graph edges and their properties.

The problem is illustrated in Figure 4.4 and the proposed approach to solving it consists of three parts:

1. Data extraction from the original data source
2. Transformation of heterogeneous intermediary schemas to a uniform representation of artefact elements and trace links
3. Property graph persistence

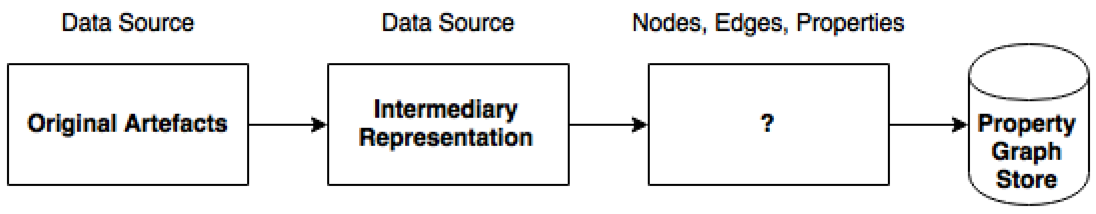


Figure 4.4: Bridging the gap between heterogeneous artefacts and the property graph model.

4.5 Framework Stages

The remainder of this chapter details each framework stage and the high-level approaches based on which framework design and, subsequently, implementation are carried out. The specifics of automatic *Traceability creation* are discussed separately, in Chapter 8.

	Change Detection	Change Impact Analysis	Consistency Checking	Change Propagation
Input(s)	Request to detect changes	Change Data,	1, Change Data 2, Impact set	1, Change Data 2, Impact set 3, Set of inconsistent elements
Output(s)	Change Data: details of the changed entity and the type of change	Impact set	Set of inconsistent elements	Suggestions to resolve inconsistencies

Table 4.3: Artefact consistency management process inputs and outputs.

At a high-level, each stage is summarised by listing the inputs it requires and the outputs it produces, which is shown in Table 4.3. For example, *Change Detection* is initiated by a user request to pinpoint changes and it generates *Change Data*, which contains details about the modification. *Change Data* is passed to the subsequent stage, *Change Impact Analysis*, which produces a set of potentially impacted elements. The following subsections examine the framework stages based on these inputs and outputs.

4.5.1 Change Detection

To provide artefact consistency management capabilities, an efficient and reliable mechanism to capture changes to heterogeneous artefacts has to be in place. The problem of change detection is manifold and it is concerned with the **identification of**

1. the changed artefact,
2. the change type, and
3. the changed elements within the artefact.

4.5.1.1 Artefact Change Classification

Identifying the type of change is an essential component of change detection and is also relevant for further stages of consistency management. The following considerations are to be taken into account when categorising changes.

- **Artefact type:** each artefact may be characterised by different changes and frequency of changes.
- **Change type:** any change can be grouped into an add, delete or edit category.

- **Structure:** composite changes are built up of a number of other changes.
- **Relevance:** some modifications are relevant for artefact consistency management, while others are not. For example modifying comments in source code is likely not to affect other related representations, such as UML class diagrams or requirement specifications.

A generic, graph-based taxonomy of changes is provided by Lehnert et al. [160] to support change impact analysis and software evolution. The taxonomy differentiates atomic and composite change types, where composite changes can be modelled as a sequence of atomic operations. In this work, a similar change categorisation approach is used and certain aspects of Lehnert's classification are adopted and customised, such as the notion of atomic and composite changes.

The first step of creating a change classification is to consider how artefacts may change at a high level. Existing artefacts can be edited or deleted and new artefacts may be added to the repository where they are stored. Edited artefacts can also change in a number of ways. Thus, a two-level change categorisation is proposed, which consists of *file level* and *artefact element level* changes.

1. **File level changes**
   - Add (new file)
   - Delete (existing file)
   - Edit (existing file)
2. **Artefact element level changes** (when an existing artefact is edited)
   - Edit by changing an existing element
   - Edit by adding a new element
   - Edit by deleting an existing element

The following scenarios exemplify these categories. For the purposes of the illustration three specific artefacts are selected, Java source code, UML class diagram and JUnit tests. However, the change categorisation is applicable to any artefact.

Scenario 1: while refactoring an existing software system, a new Java class is added to a repository.
Change type: at the file level, this change is an *add* type, which is not further decomposed.

Scenario 2a: the visibility of a method on a UML class diagram is modified.
Change type: at the file level this is an *edit* change type, and specifically, at the artefact element level, an existing element was edited.

Scenario 2b: a field is deleted from a UML class diagram.
Change type: at the file level this is an *edit* change type, and specifically, at the artefact element level, an existing element was deleted.

Scenario 2c: a new method is added to a UML class diagram.
Change type: at the file level this is an *edit* change type, and specifically, at the artefact element level, an element was added.

Scenario 3: a JUnit test class is deleted from a repository.
Change type: at the file level, this change is a *delete* type, which is not further decomposed.

4.5.1.2 Change Detection Output: Change Data

Following the identification of the modified artefact, the changed entity within the artefact and the change type, the change detection process produces a *Change Data* output, which is discussed in Chapter 7. The output provides a means to identify the modified entity, while also supplying detailed change information to subsequent framework stages.

4.5.2 Rule-based Traceability Maintenance

Although it is not a separate framework stage, traceability maintenance is integral to a successful consistency management process and is closely related to change detection and subsequent stages. Therefore it is discussed following an examination of change detection and prior to introducing the change impact analysis approach. Traceability maintenance encompasses the functionality to update inter trace links in the graph database following the detection of a modification and the identification of the changed entities and their corresponding change types. During change detection, the graph database is updated to reflect the latest state of artefacts, i.e. depending on the modification(s), new nodes may be added, and existing ones may be deleted or edited. Trace links may also be affected by the modification in question, such that new links are added or existing links are deleted.

The consequences of changes to inter trace links can be captured by formulating trace maintenance rules, which are described in Chapter 7. The rules depend on the specifics of the change provided by *Change Data*, and in summary, can be formulated based on the following:

a) File level change type,
In case of *edit* file level changes:
b) Artefact element level change type,
c) Fine-grained change type (signature or content change, discussed in Chapter 6),

d) Artefact element type from which the fine-grained artefact element type can be established (child or parent element, discussed in Chapter 6).

4.5.3 Change Impact Analysis

Succeeding the modification of an artefact element, the aim of the change impact analysis of heterogeneous software artefacts in the ACM framework is to determine further elements possibly affected by the same modification. Input to this process is provided by *Change Data*. The output of change impact analysis is therefore the set of elements deemed to be affected by the modification. As mentioned in Chapter 2, change impact analysis can be performed utilising various approaches, such as structure-based, history-based and probabilistic methods. Following is a discussion of the approach presented in this work.

Impact Analysis as Property Graph Traversals

The impact analysis strategy of the ACM framework utilises the property graph representation of heterogeneous artefact entities and specifics of the change obtained through change detection. Therefore it falls in the category of *structure-based* impact analysis solutions, which it extends to cater for multiple artefact types. Traversing the artefact graph allows the identification of connected entities at the specified level. The approach stems from the assumption that nodes connected to a modified node may also be impacted by the same modification, depending on the link connecting them. The traversal can be described as a *breadth-first search* and the approach currently caters for direct connections. Similarly to nodes, edges of the graph can also be described with a type property, which is a significant factor in traversals. Each node is connected to the start node through a specific link type expressed by the values of the type property, which define the scope of traversals.

The flow chart in Figure 4.5 describes the algorithm for traversals of *inter* links. The input of traversals is *Change Data*, which makes it possible to identify the modified node, which is the starting point of the traversal, and to express specifics of the change type, which determines the subsequent steps. In case the start node was edited or deleted, it is possible to identify nodes connected to it, which are then added to the set of potentially impacted nodes, i.e. the output of impact analysis. In case the node was newly added to the graph, further details are required and in certain cases there is no straight forward way of identifying connected elements. In case the newly added node represents a class, the impact analysis algorithm cannot determine which other nodes may be connected. Exceptions are Java source code and JUnit artefacts, in cases where they connect to other classes or interfaces. However, if a member element is added, it is already connected to a class through an *intra* link, hence it is possible to visit the node representing the container class. If the container class is connected to other nodes through *inter* links, those nodes are traversed and added to the set of potentially impacted nodes. In summary, the algorithm is based on the change type and the types of elements. Traversals are based on link types. These are the three factors constituting change impact analysis using the property graph.

4.5.4 Consistency Checking

An essential component of artefact consistency management is the functionality to establish whether an element affected by a modification remains consistent, and hence need not be changed, or as a result of the modification, it has become inconsistent. The identification of inconsistencies falls under the responsibility of consistency checking in the ACM framework.

Listing 1 shows the overview of the consistency checking approach, which is based on the interplay of the outputs of previous framework stages supplying data about the specifics of the change and its potential impact, and consistency rules. *Change Data*, provides details of the modification(s) through which the changed entity can be located in the graph database. Additionally, properties captured in Change Data also play a crucial role in formulating consistency checking rules. Specifically, the change type property separates three cases from each other; add, edit and delete scenarios are processed differently.

Algorithm 1 Consistency Checking Approach Overview.

```
1:  input: ChangeData, Impact set
2:  output: Artefact element is consistent, Artefact element is inconsistent, Artefact element is
    potentially inconsistent
3:  begin
4:  Evaluate the change type property value of the ChangeData object
5:      if change type = add, no consistency rules can be derived
6:      if change type = delete, evaluate changed and connected entity, and apply corresponding
7:      consistency rule
8:      if change type = edit, evaluate changed and connected entity, and the artefact element
9:      level change type, and apply corresponding consistency rule
10: Create ConsistencyResult
11: end
```

Consistency Rules

Consistency rules take the form of *if-then rule*s and are created based on the following information:

- Artefact level change type (add, edit, delete)
- Artefact type

- Artefact element type, which also denotes hierarchical relationships between elements found within a specific artefact; certain elements contain others, or some elements are contained within others
- Fine-grained artefact element type derived from Artefact element type
- Existing Inter and Intra trace links connecting elements

Specific examples of these aspects are given in Chapter 6.

Consistency checking is divided into two main categories. **Inter consistency checking** takes place between artefacts of different types, while **intra consistency checking** is performed between elements of the same artefact.

Finally, consistency rules represent three outcomes of consistency checking. An element connected to a modified element can be either consistent, inconsistent or potentially inconsistent. By flagging up potential inconsistencies, the framework applies a pessimistic approach to consistent checking, that is, if there is a chance of inconsistency, it is reported to the user who can decide about the course of action to take.

4.5.5 Change Propagation

Change Propagation is aimed at assisting users with re-establishing consistency by suggesting possible resolutions based on the output of consistency checking. Since the input includes information about the change and changed element, the potentially affected elements and their state of consistency, change propagation can provide users with further information on how to resolve inconsistencies. For example, in case a UML class is inconsistent as a result of editing a requirement in a requirement specification, the framework may suggest to edit the UML class.

In case the outcome of consistency checking is that no consistency violations have been detected, the users are given a summary about the change and the results of consistency management. In case there is a (potential) inconsistency, users are given suggestions on possible resolutions. The framework provides a summary of the change and the (potentially) inconsistent elements.

4.6 Conclusions

The problem of artefact consistency management may be more effectively tackled using a comprehensive framework incorporating *traceability creation and maintenance, change detection, change impact analysis, consistency checking* and *change propagation* techniques. This hypothesis constitutes the conceptual foundation of the proposed approach, which is aimed at fulfilling the high-level requirements discussed in 4.2. This chapter introduced the framework and its underlying stages, and provided an overview of data representation and the approaches taken in each framework stage. In the following chapter design decisions pertaining to the framework architecture, artefact and trace link data representation are described.

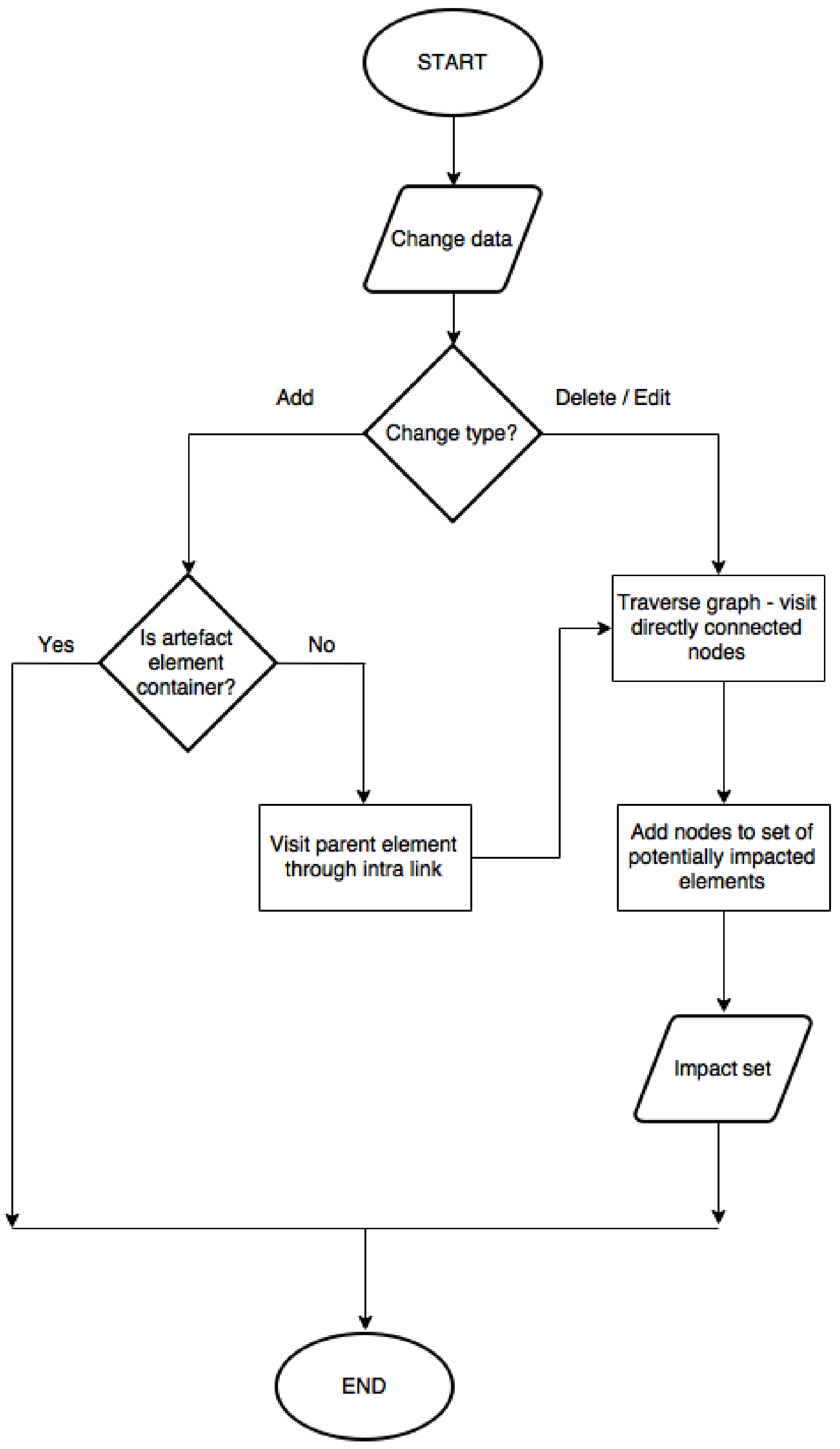

START
Change data
Add
Change type?
Delete / Edit
Yes
Is artefact element container?
No
Traverse graph - visit directly connected nodes
Visit parent element through intra link
Add nodes to set of potentially impacted elements
Impact set
END

Figure 4.5: Flow chart illustrating the algorithm for inter link traversals.

CHAPTER FIVE

ARCHITECTURE AND DESIGN

The proposed holistic approach to artefact consistency management is realised in the design and implementation of a proof-of-concept system, the **A**rtefact **C**onsistency **M**anagement (ACM) framework. Following the formulation of the characteristics of an ideal consistency management solution, the first step towards the realisation of the holistic framework is the translation of high-level requirements to an overall architecture of the framework. This chapter presents the design strategy, the framework architecture and decisions made during the design process.

5.1 Design Strategy

The design process started by specifying and analysing the following:

1. the functional requirements of the system,
2. design constraints, and
3. quality attributes.

The framework design, due to the nature of this work, adopted a lightweight version of *Attribute Driven Design (ADD)*, an approach for deriving software architectures characterised by a design process based on software quality attributes [161].

5.1.1 Functional Requirements

Functional requirements of the system were derived from the stages of the holistic framework, and were captured in the form of a use case document. The document can be found in the framework's GitHub repository[1].

5.1.2 Design Constraints

Design constraints establish assumptions that the implementation is required to fulfil. The design constraints of the ACM framework are as follows.

1. Persistent storage. The framework stores its data in a graph database.
2. Version control. The framework obtains artefact data from version control systems in which original artefacts are stored.

5.1.3 Architectural Tactics

When considering the quality attributes relevant for the framework, properties the framework shall exhibit, architectural tactics were utilised. Tactics provide the architectural means to control parameters relevant for a given quality attribute, such as modifiability, availability, performance, and security [162].

Quality attributes for the ACM framework are closely linked to a subset of the characteristics of an ideal framework, presented in Chapter 4. The remainder of this section describes the tactics, which are used to achieve the quality attributes.

1. **Modifiability**

   Modifiability tactics are linked to the requirements of artefact and tool independence, identified by *R2* and *R3*, respectively. To achieve this, the framework shall cater for heterogeneous artefacts and the addition of new ones. To support a changing number of artefacts and tools, the framework may require extensions: functionality to handle the consistency of new artefacts and their storage, capabilities to obtain artefact data from new sources represented by new file formats. Additionally, users may have their own preferences for persistence and version control systems. Finally, functionality offered by the framework can also be extended and modified. These *extensibility* requirements can be addressed by utilising *modifiability* tactics aimed at controlling changes including their time and cost implications. Such aims can be achieved through various means including localising modifications or preventing ripple effects of changes [163]. The specific modifiability tactics applied for each relevant architectural component are discussed in detail in Section 5.3.

[1]https://github.com/ACMFramework/ACMF/blob/master/ACMF_UseCaseDoc.docx

2. **Usability**

   Usability tactics are related to requirement *R4* (Customisable and non-intrusive). A framework that does not impose new methodologies and processes on the user, needs to seamlessly integrate with the user's current practices regardless of the role the user takes in a software development project. Closely linked is the ability to configure the framework to allow users to customise it to their own needs. Usability tactics help design a framework that does not require a high learning curve, is less error prone and provides a better user experience [164].

5.2 Framework Architecture

The overall architecture of the ACM framework, showing its boundaries and external components, and system components and their connectors, is depicted in Figure 5.1.

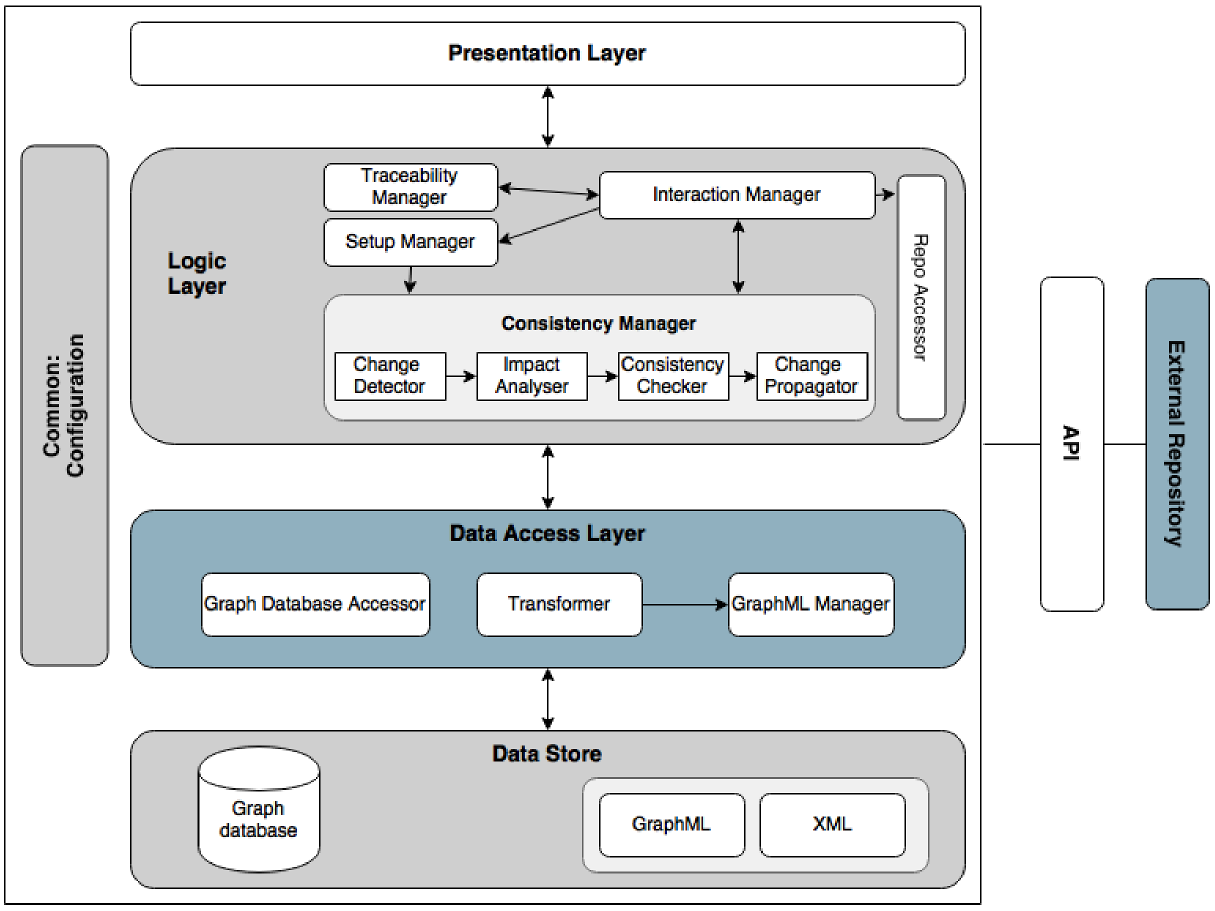


Figure 5.1: The overall architecture of the ACM framework.

The architecture adopts a combination of styles. Thus, it is best described as a heterogeneous architecture in which components are hierarchically decomposed, and can be organised using different styles at different levels. Figure 5.1 shows that at the topmost component level, structural elements are organised into layers, namely a *Presentation, Logic*, and *Data Access* constituting a layered style. This style allows the layers to evolve independently. The Logic Layer is further decomposed into an *Interaction Manager, Traceability Manager, Setup Manager* and *Consistency Manager* component. The latter exhibits a *Pipe-Filter* architectural style involving the *Change Detector, Impact Analyser, Consistency Checker* and *Change Propagator* components. The *External Repository* component, which is outwith the boundaries of the framework, interacts with the framework through its corresponding API. Finally, the *Configuration* component is identified as a common module used by various components of the framework. Following is a summary of the layers and architectural components within them with a brief description of the functional requirements they fulfil.

Presentation Layer

The Presentation Layer provides an interface between the system and users. It encapsulates the functionality of the frontend, and allows users to interact with the framework and view requested information. Its responsibilities include providing access to framework data visualisation, customisation and configuration capabilities.

Logic Layer

Artefact and link data stored in the framework is mainly processed and utilised in this layer. It encapsulates the consistency management functionality of the framework, through the *Consistency Manager* component, which supports users in re-establishing the consistency of artefacts across the system following a change: users can identify which artefacts have changed, which entities are potentially impacted by the same modification and whether these elements are in an inconsistent state. Functionality to support the semi-automatic establishment of trace links and the setting up of the framework - through the *Traceability Manager* and *Setup Manager* components - are also encapsulated by this layer. Framework setup, trace link creation and consistency management are initiated by the *Interaction Manager* component.

Data Access Layer

This layer consists of components that allow basic CRUD (Create, Retrieve, Update, and Delete) operations to be performed on the data store through the *GraphDatabaseAccessor* and *GraphMLManager* components. The *Transformer* component is responsible for mapping original artefacts to the framework's data representation model.

Data Store

Artefact Data, comprised of artefact elements and trace links, is stored using XML-based (GraphML and custom XML) formats and a graph database backend. Hence the *Data Store* consists of the *GraphML*, *XML* and *Graph database* components.

External artefact repository

Original artefacts are stored outside the framework in a version control system. This component is external to the system and can be accessed through APIs corresponding to the user's choice of version control system.

5.3 Detailed Design of Architectural Components

Following an overview of the framework's overall architecture, this section discusses the design decisions made with respect to each architectural component.

5.3.1 Data Access Layer and Data Store components

5.3.1.1 Data Store

The *Data store* component provides the means to store artefact data. Prior to saving data to the graph database however, XML and GraphML intermediate representations are required, which are managed by the *GraphML* and *XML* components.

5.3.1.2 Data Access Layer

GraphDatabaseAccess component

One of the aims of the *Data Access Layer (DAL)* is to provide a means to access and update the underlying data store through the *Graph Database* component. At a high level this involves adding new instances individually or in bulk, updating and deleting one or more nodes (or edges) and obtaining information about existing nodes (or edges). One of the principles applied in the design of the *DAL* is localising changes to the *DAL* in case the data store changes. Thus, the *Data Access Object (DAO)* design pattern [165] is adopted, which allows communication with the underlying persistence and the domain logic. In case a different graph database is selected, the conversion does not cause unintended changes to the rest of the framework. Concrete classes representing a specific database technology implement the *IGraphStoreDAO* interface, which provides signatures of basic graph database operations. Using the factory pattern and the *DaoFactory* class allows the instantiation of the required *Dao* implementation. The design of the *Graph Database* component draws inspiration from architectural tactics for modifiability. In particular, restricting changes to specific components reduces the cost of modifications and components prone to modifications can be identified to minimise the effects of changes.

Transformer and GraphMLManager components

The *Transformer* component is responsible for mapping the original artefacts to the framework's data representation model. The *GraphMLManager* component allows CRUD operations to be performed on *GraphML* representations saved in the *GraphML* data store. To perform mapping, the *Transformer* component uses services provided by *GraphMLManager*.

5.3.2 External Repository and Corresponding API Component

The change detection functionality requires details of modifications. As expressed by the *Version control* design constraint introduced in Section 5.1, the framework is designed to cater for artefacts originally hosted in a version control system. The aim of this decision is to ensure that the latest version of heterogeneous artefacts, including non-source code representations, is available. Should changes happen to any of these representations, they can be queried using the version control system or APIs providing programmatic access to them. Each major version control system provides open source APIs for accessing and manipulating their repositories, such as *hg4j* [2] for Mercurial, *SVNKit* [3] the Java Subversion library, or *JGit* [4] for Git. The *External repository* component represents such version control systems and is connected to the framework through the *API* component. During the design process the following concerns were considered.

1. **Heterogeneous Version Control Systems**

   Since a number of version control systems exist and they may vary across projects, to address this heterogeneity, the functionality to pull data from repositories is designed with *pluggability* in mind. The framework shall be able to obtain specifics of changes irrespective of the external system. The *IRepoQuery* interface exposes standard operations required to implement this functionality, and concrete classes provide repository specific implementations. Similarly to the design of database access, this design concern is also related to the modifiability attribute described in Section 5.1 Thus, the factory pattern and a *RepoFactory* class is used to create instances of the required class depending on the selected version control system.

[2]http://hg4j.com/
[3]http://svnkit.com/
[4]https://eclipse.org/jgit/

2. **Change Handling**

   Any artefact in any repository may be subject to modifications at any moment in time. Some artefacts may change more frequently than others and some artefacts may be more relevant to some users than others. Due to these variables, the idea of pulling changes as they occur was dismissed since it may potentially lead to a profusion of notifications, the problem of which is highlighted in literature [116]. Instead, users can configure the framework to pull changes from the repository at specified intervals.

5.3.3 Logic layer and its Components

5.3.3.1 Interaction Manager

The *Interaction Manager* component coordinates the functionality of multiple components within the Logic Layer and also provides a single point of interaction to the *Presentation* Layer.

5.3.3.2 Traceability Manager

The *Traceability Manager* component manages traceability related tasks. Firstly, it handles functionality to establish trace links between heterogeneous artefacts, which incorporates a machine learning approach, presented in Chapter 8, to automate the process as far as possible. Secondly, the component is responsible for the maintenance of trace links during the consistency management process. Link maintenance involves deleting and adding trace links at the graph database level.

5.3.3.3 Setup Manager

The *Setup Manager* is responsible for managing the initial framework setup. This involves initiating framework configuration using the *Configuration* component, following which the setting up of *Artefact Data* is performed: artefact elements are extracted from their original tools and are transformed to the unified representation of framework data (*Transformer* component), and trace links between them are established (*Traceability Manager* component). Finally, artefact and trace link data are saved to the *Data Store* using the *Data Access layer*.

The design of framework setup is based on the Observer pattern. A *SequentialExecutionManager* class provides orchestration functionality by registering all required functionality to be executed - in this case configuration, data extraction, transformation, link establishment and data import to the data store - and by executing them in the specified order. Figure 5.2 illustrates this design solution, where the *GraphMLImporter*, *ConfigurationHandler* and *TraceLinkCreator* classes provide an *execute()* method, which is registered with the *SequentialExecutionManager* class used by *SetupHandler* that performs the above mentioned sequential execution.

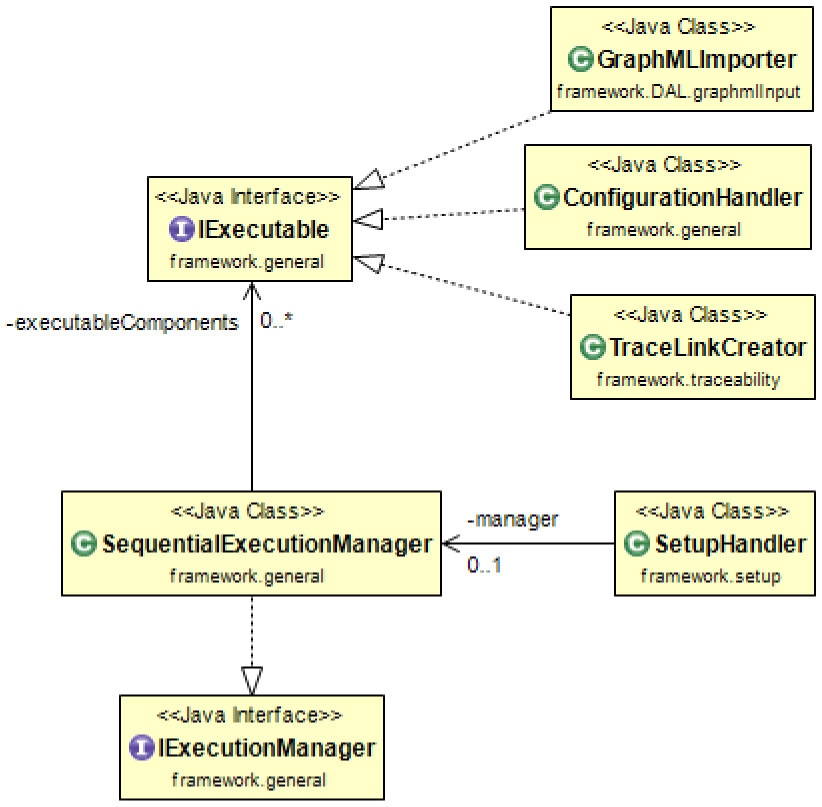


Figure 5.2: Design of setup management functionality, class diagram excerpt.

5.3.3.4 Consistency Manager

This composite component consists of four subcomponents that are responsible for the consistency management functionality in the framework. The output of each subcomponent is taken as an input and is processed by the subsequent subcomponent. Thus, these subcomponents are best described as a *Pipe-Filter* architectural style, where the tasks involved in consistency management are executed in a sequential manner.

The *Pipe-Filter* architectural style introduces modifiability tactics in the design process including *Use encapsulation*, which allows hiding the internal details of data processing as only inputs and outputs are visible, and *Restrict communication paths*, which defines the number of inputs and outputs each filter can have thus limiting dependencies between them [163]. Each filter is described in the following sections.

Change Detector Component

The *Change Detector* component provides change detection functionality to identify the file level and artefact element level change type. The following components are responsible for carrying out these tasks: file level changes are captured by the *External Repository* and *API* components. The artefact element level change type is identified by the *Change Detector* component. At the end of the process, the framework outputs change data capturing the details of the change. The *Change Detector* component also initiates updates to the data store using the Change Data output of change detection, and the maintenance of trace links in the database.

The orchestration of these responsibilities is achieved via the *SequentialExecutionManager* class described in Subsection 5.3.3.3. This class is responsible for registering components that are to be executed. Such components implement the *IExecutable* interface, which exposes an *execute()* method. In the current example these include the *ChangeExtractor* and *ChangeIdentifier* classes from the *Change Detector* component, and *TraceLinkMaintainer* class from the *Traceability* component. Functionality to be executed as part of change detection is provided by the execute methods implemented in the above mentioned classes, and registering them with the *SequentialExecutionManager* class makes it possible to perform their operations in a specified sequence.

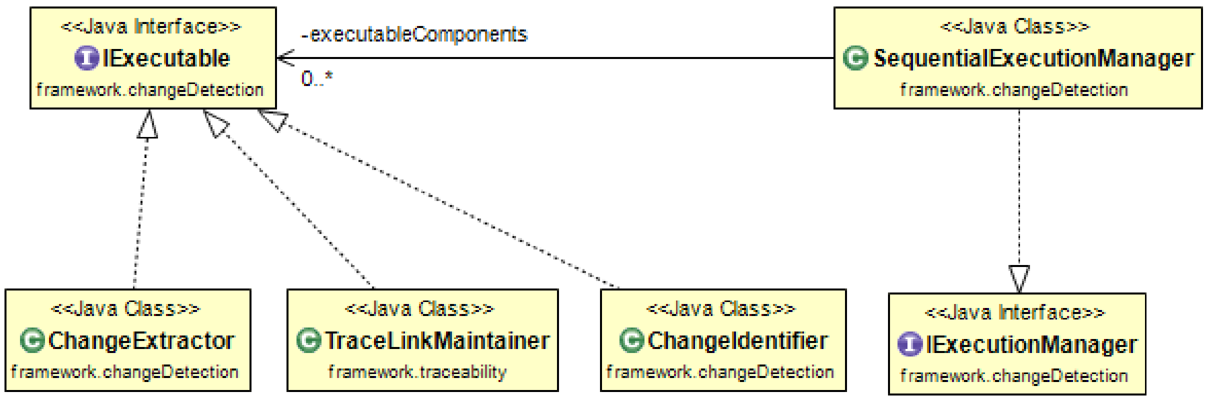


Figure 5.3: Design of the orchestration of change detection functionality, class diagram excerpt.

Change Impact Analyser Component

This component is responsible for establishing the set of potentially impacted artefact elements following detecting and identifying modifications. The input to this process is supplied by *Change Data* to identify the changed entity and its connections in the database. The approach is designed with artefact and tool independence in mind through performing impact analysis at the property graph level.

Functionality	Layer	Component
Support for Artefact Data	Data Access, Data Store	Transformer, Graph Database Accessor
Support for traceability	Logic	Traceability Manager
Support for change detection	Logic	Consistency Manager (Change Detector)
Support for change impact analysis	Logic	Consistency Manager (Impact Analyser)
Support for consistency checking	Logic	Consistency Manager (Consistency Checker)
Support for change propagation	Logic	Consistency Manager (Change Propagator)

Table 5.1: Mapping of functional areas to architectural components.

Consistency Checker Component

The functionality of checking the consistency of artefact elements following a change is represented by the *Consistency Checker* component, the input to which is suppied by the *Change Impact Analyser* component.

Change Propagator Component

The *Change Propagator* component, which is responsible for providing suggestions to users, receives its inputs from the *Consistency Checker* component.

5.4 Design Evaluation

To conclude the chapter, an assessment of the design is provided to reveal the extent to which it supports the functional and non-functional requirements set out in Chapter 4. Firstly, the functional requirements are evaluated by mapping functional areas to architectural components in Table 5.1. Lastly, non-functional properties of the framework are assessed.

Modifiability. As mentioned at the start of the chapter, this property is closely linked to the artefact and tool independence attributes of an ideal framework: a framework that can be extended to cater for new artefacts authored in different tools satisfies one aspect of the modifiability requirement. Components of the *Data Access Layer* and the *Data Store* provide means to extract and store heterogeneous artefacts in the framework. The second aspect is framework extensibility. The first step during design was the identification of variable components that may trigger changes to the framework. Such modifications are the result of adding new artefacts, tools, and changing user preferences with regards to version control systems or database backends. The identified components include *Data Store, Data Access Layer,* and *External Repository*. Thus, localising changes to the *DAL* layer ensures that modifications do not affect the rest of the framework.

Usability. User interactions are managed by the *Presentation Layer*. The architecture provides a *Configuration* component, which allows preferences to be set and modified. Users may interact with this component at setup time to perform an initial framework configuration. Preferences may include notification settings, change detection intervals, which allow the framework to provide its functionality in a non-intrusive manner.

Distributed development. The framework currently caters for distributed development by pulling artefact and link data from remote repositories. Since the architecture is designed to be modifiable, it is possible to extend it with components required for managing further functionality related to distributed development practices.

Automation. The architecture of the ACM framework allows the functionality of individual components to be carried out automatically. The level of automation provided for each functional area is dependent on the feasibility of the implementation, therefore automation is re-assessed in Chapter 6.

CHAPTER SIX

IMPLEMENTATION OF THE ACM FRAMEWORK: DATA REPRESENTATION

6.1 Introduction

The prototype, the ACM framework, is implemented in Java and supports the stages of artefact consistency management defined in Chapter 4, including *traceability creation, change detection, change impact analysis, consistency checking,* and *change propagation*. As mentioned in the same chapter, to automate these stages as far as possible, data was obtained from original artefacts and stored in a uniform representation, as a property graph. This mapping is handled by an overarching framework component, *Artefact Data*, which is an artefact and trace link store. *Artefact Data* is represented by the *Data Store* component and *Data Access Layer* at an architectural level. While this chapter presents the implementation specifics relevant to *Artefact Data* and discusses data representation, Chapter 7 describes the implementation of the framework stages.

Considerations relevant to *Artefact Data* include:

1. A description of the structural attributes of the original representations and their mapping to properties of the property graph model of the ACM framework,
2. The approach taken for mapping heterogeneous artefacts created by diverse tools to the property graph model, which involves

   a) the extraction of data from original artefacts and their transformation to a unified

representation, and
b) the storage of artefact and trace link data in the framework.

6.2 Artefact Selection

For this implementation, a set of concrete artefacts were selected to demonstrate the feasibility of the holistic consistency management approach and the functionality of the framework. The selected artefacts were obtained from a number of open source systems, which are introduced in detail in Chapter 8. These artefacts include:

- Requirement specifications,
- UML use case diagrams,
- UML class diagrams,
- UML sequence diagrams,
- Conceptual architecture,
- Module view architecture,
- Java source code, and
- JUnit test cases.

These artefacts are used to verify whether it is possible to obtain data from diverse formats and to convert it to uniform *Artefact Data* handled by the framework, and whether it is feasible to perform consistency management tasks leveraging this data. The level of automation achieved while carrying out these tasks is discussed in Chapter 9. Despite using a set of artefacts, the framework can be extended with additional representations.

The motivating factor for selecting these artefacts is the requirement to cover a diverse range of representations: artefacts with different structural characteristics, from various stages of the software lifecycle covering both lower and higher abstraction levels.

6.3 Property Graph Representation

Chapter 4 introduced the concepts of *artefacts, artefact elements* and *trace links* using the property graph-based representation. This section discusses the mapping of diverse artefacts and their relationships to artefact elements and trace links modelled by the property graph.

Firstly, the mapping of original artefacts to graph nodes is described in Subsection 6.3.1, which involves identifying the structural elements of the original representations. Based on these structural elements, relevant artefact elements can be derived and their properties can be specified. For example, in a requirements specification, each *functional* and *non-functional requirement* is represented by an artefact element, which in turn is mapped to a node in the graph. However, the *executive summary* structural element is outside the scope of this work as it is unlikely to directly contribute to consistency management tasks. Thus, such elements are not considered. This pattern applies to all artefacts. The derivation of artefact elements from original artefacts, their mapping to graph nodes and their properties are detailed in Table B.5 in Appendix B. For example, based on the *variable declaration* structural element of *JUnit test* artefact the *variable declaration* artefact element is derived. This artefact element is described, for example, by the *name, type*, and *modifier* properties and is represented by a graph node.

Secondly, the mapping of trace links between and within the original artefacts to graph edges is discussed in Subsection 6.3.2.

Common Properties: Name, Type, and Unique id

Despite the structural variance of heterogeneous artefacts, some properties are shared by all artefact types, such that each artefact element can be described by a *name* and a *type*. For example, an architectural component called *Service* can be associated with a name property that takes the value *Service*, and a type property, which is assigned the value *component*. Additionally, a *unique identifier* property is reserved to identify each element in a unique manner.

Unique identifiers are generated in the framework since an essential implementation level consideration is the identification of artefact elements. The motivation to generate identifiers stems from the fact that the graph database storing artefact elements and their trace links does not provide such a mechanism. Each node is assigned an identifier, although following database updates, these can be overwritten.

Another advantage of generating an identifier is that the information contained within can be customised. In the current implementation the unique id is composed of three parts: an *artefact prefix*, a *sequential number*, and the full *file path* of the transformed artefact representation. An example is as follows: *SC1D:LocalRepo/Account.graphml*. The artefact prefix specifies the type of the artefact. Thus, the artefact type is not stored in the form of an additional node property, instead, it can be inferred from the *unique id*. The prefix currently takes the following values and can be extended when new artefacts are added.

1. SC: source code artefact
2. UT: unit test artefact
3. DI: UML class diagram artefact

4. AR: Software architecture (Conceptual and Module view) artefact
5. SD: UML sequence diagram artefact
6. UC: UML use case artefact
7. RQ: requirement artefact

6.3.1 Specification of Graph Nodes and Properties

6.3.1.1 Requirement Specification

According to the classification described in Chapter 2, requirements specifications are high-level artefacts, mostly associated with the requirements engineering process [12]. Written typically in natural language, specifications can be structured in various ways. Therefore their building blocks may vary from document to document. A sample structure adopted from a template[1] is given in Table B.5 in Appendix B. The functionality and design constraints of the system are captured in functional and non-functional requirements, which constitute relevant artefact elements. Each requirement can be identified by its *name*, and can be described by a *title*, *contents*, *priority* and *type* attribute. These descriptors allow the specification of graph properties to describe artefact elements extracted from requirements specifications. To illustrate the format of the original artefact and the property graph representation, the following example is given: a single requirement in a requirements specification document may take the following form.

```
Name: R1
Title: Customer details
Description: The system shall record customer details: name, address,
    telephone number and account number.
Type: Functional
Priority: High
```

Figure 6.1 shows that the example requirement artefact element can be represented by graph node 1, and described by the *Name (R1), Title (Customer details), Contents (The system shall...), Type (Functional)*, *Priority (High)*, and *UniqueId (RQ0D/Users/I/f.graphml)* properties. Properties of the node are derived by transforming each descriptor of the requirement into a property. In this example, node 1 is connected to node 2 representing a Java class, which in turn is connected to node 3 denoting a Java method.

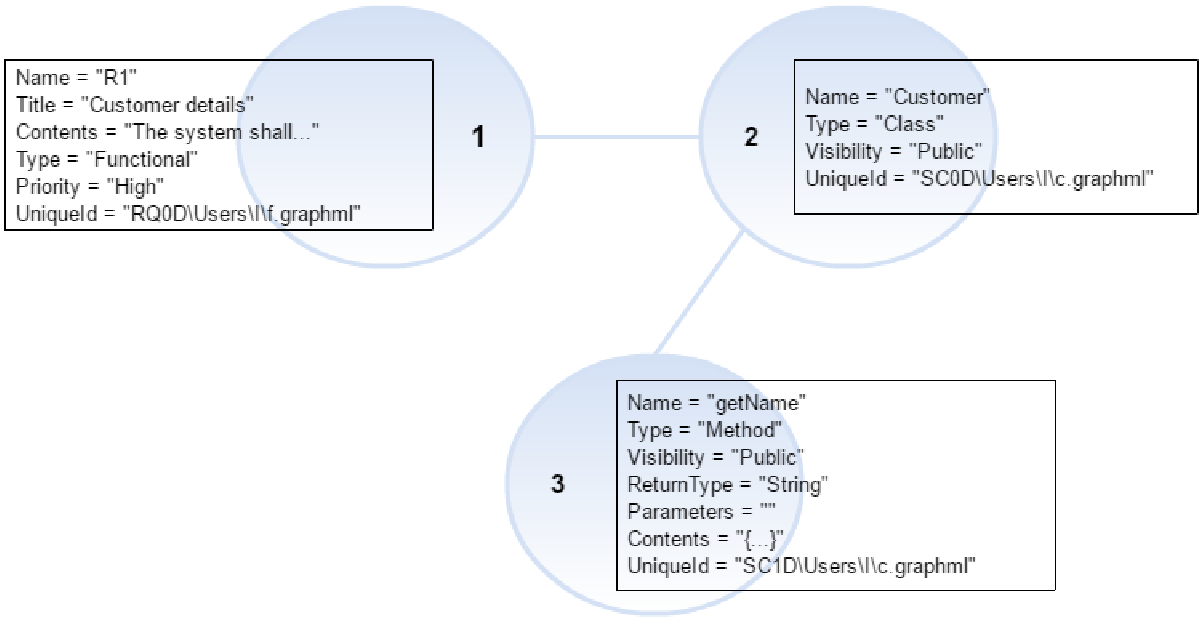


Figure 6.1: Property graph representation of a requirement, a Java source code class, and a Java method artefact element.

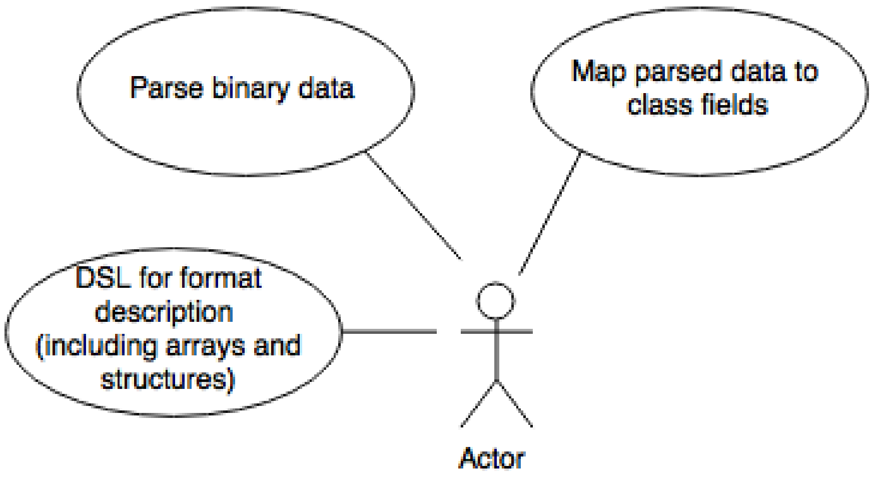

Parse binary data
Map parsed data to class fields
DSL for format description (including arrays and structures)
Actor

Figure 6.2: Binary Block Parser system - Use case diagram excerpt.

6.3.1.2 UML Design Diagram: Use Case diagram

The elements of a UML use case diagram are derived from its OMG specification [166]. Out of these elements, the framework captures the *use case* structural element. An excerpt of the use case diagram of the Binary Block Parser system[2] is shown in Figure 6.2. A single use case,

[1]http://csis.pace.edu/ marchese/CS775/Requirements%20Specification%20Template.doc
[2]https://github.com/raydac/java-binary-block-parser

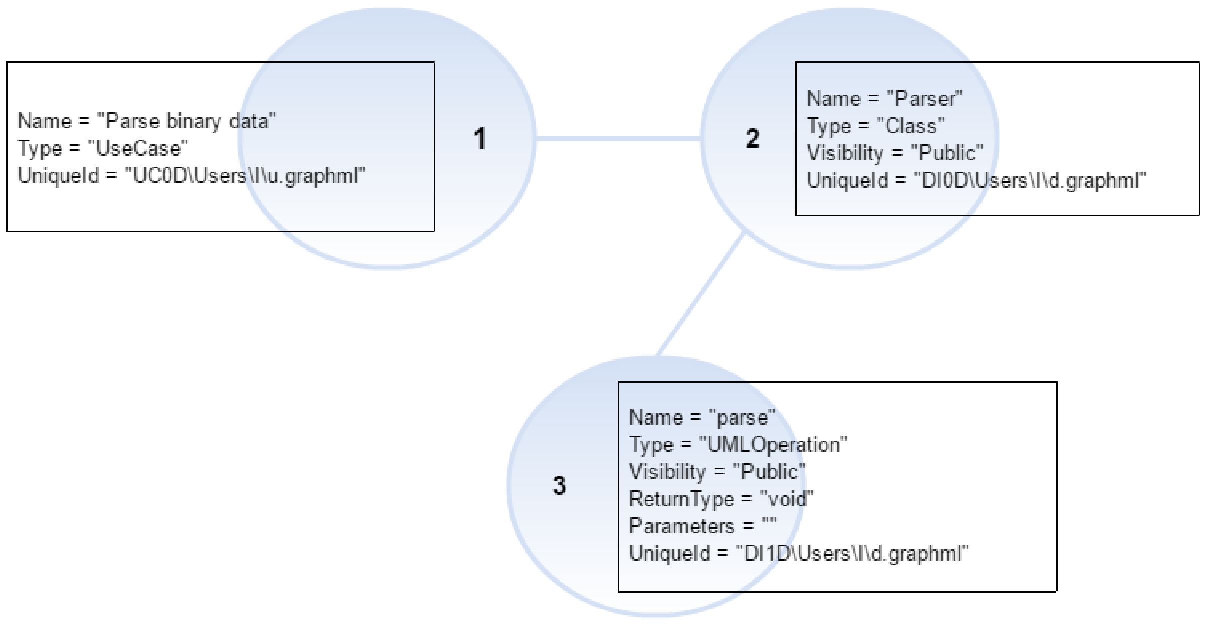


Figure 6.3: Property graph representation of a single use case, a UML class and a UML operation.

Parse binary data represented by graph node 1 and its properties is illustrated in Figure 6.3. It is connected to nodes 2 and 3, which depict a UML class and operation, respectively.

6.3.1.3 Software Architecture: Conceptual view

Software architectures capture elements encapsulating processing, data, and interaction [24]. The main structural elements include components and connectors. Figure 6.4 shows an excerpt of the architecture of the Titan graph database[3]. Each component is mapped to a graph node and can be described by a name and a type property. Figure 6.5 depicts node 1, which represents the *Storage and Index Interface layer* architectural component, and its connections, node 2 and node 3.

6.3.1.4 Software Architecture: Module view

Module view architectures represent the structure of a system as a set of code units [167]. Therefore, their main elements are modules, which can be described by a *name* and a *type* property. An excerpt of the module view architecture of Neo4j[4] is given in Figure 6.6, while a sample property graph representing three modules as nodes is shown in Figure 6.7.

[3]https://github.com/thinkaurelius/titan/tree/titan10/docs/static/images
[4]https://github.com/neo4j/neo4j/tree/3.1/docs/images

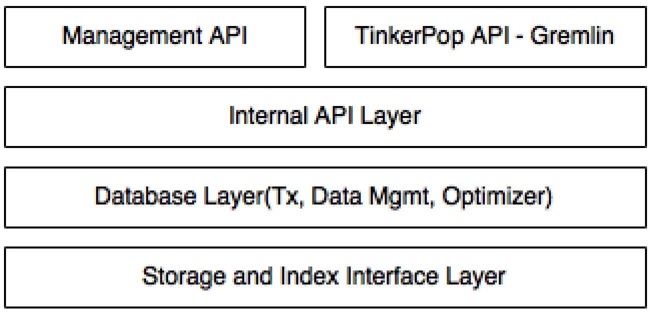

Management API
TinkerPop API - Gremlin
Internal API Layer
Database Layer(Tx, Data Mgmt, Optimizer)
Storage and Index Interface Layer

Figure 6.4: Architecture diagram of Titan

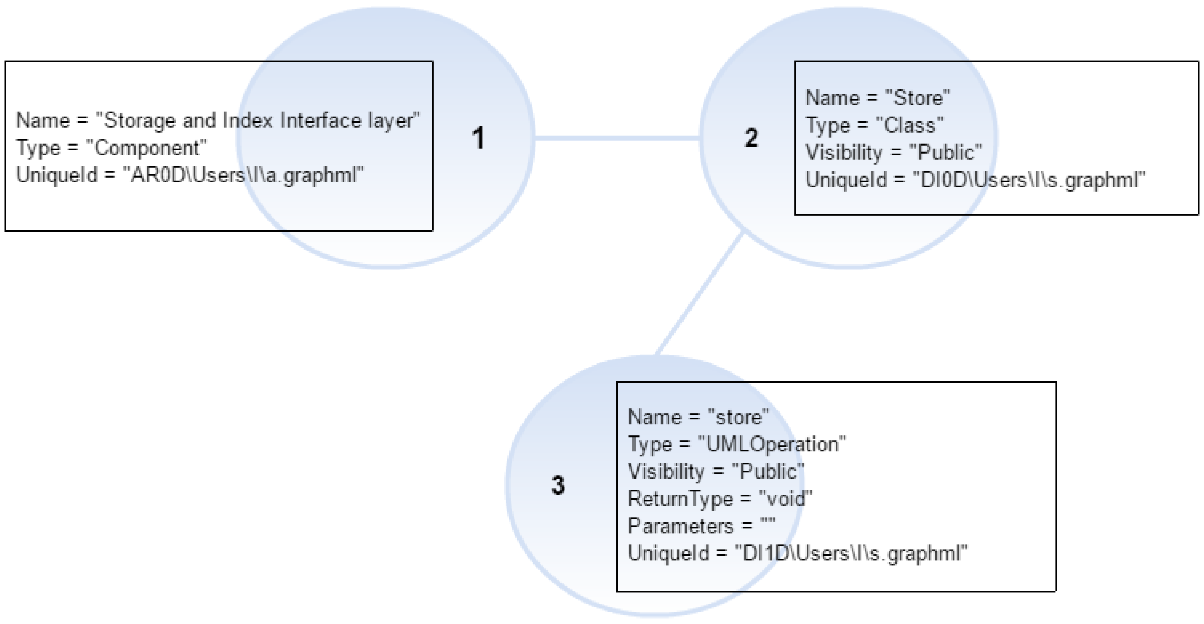


Figure 6.5: Property graph representation of a single architectural component, and a UML class and operation.

6.3.1.5 UML Design Diagram: Class diagram

The main structural elements of a UML class diagram can be derived from its formal specification [166] [168]. Artefact elements extracted from UML diagrams are either *container*, such as classes, or *member* elements, such as operations and attributes. Each element can be described by a *name*, *type* and *modifier* (such as visibility) attribute. Due to the diversity of element types, each set of UML elements may be associated with specific attributes. A UML class diagram excerpt containing a single UML class with an operation is provided in Figure 6.8 to highlight its structural elements. To represent the class and its member as a property graph, each artefact element contained in the diagram is transformed to a graph node, and each relevant descriptor of

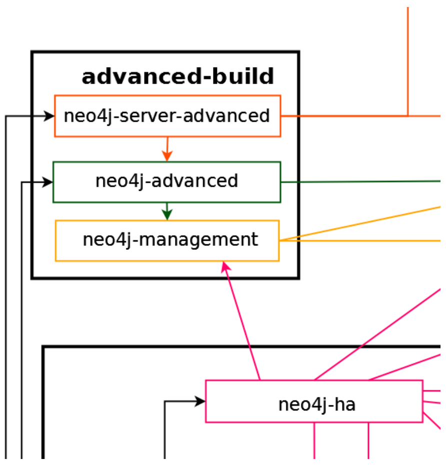


Figure 6.6: Module view architecture of Neo4j - Excerpt.

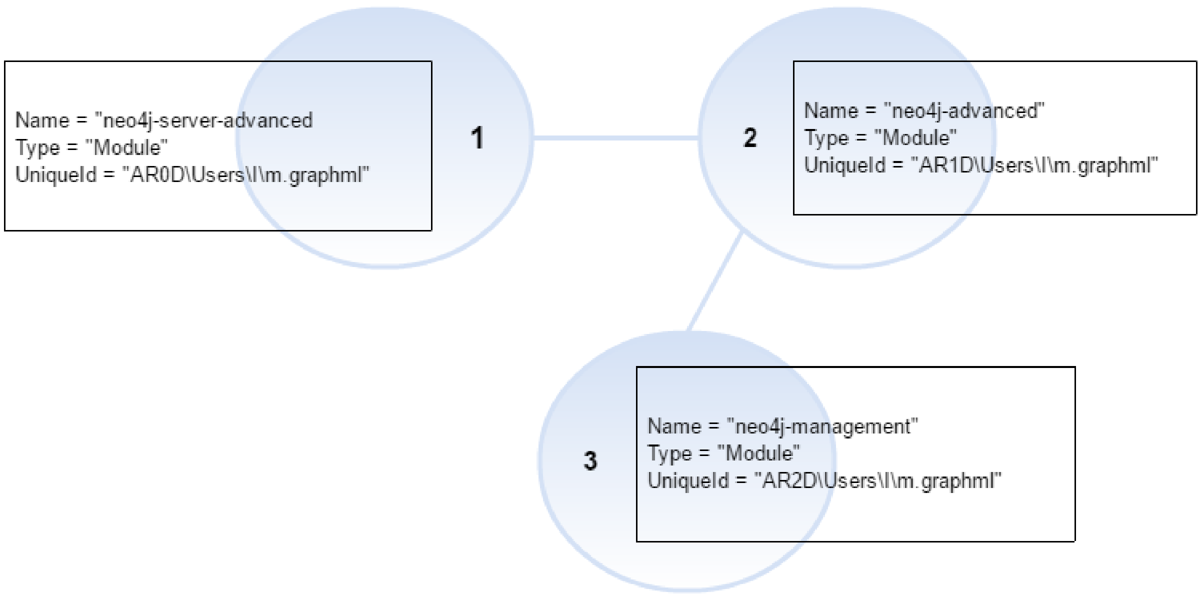

Name = "neo4j-server-advanced
Type = "Module"
UniqueId = "AR0D\Users\l\m.graphml"
1
2
Name = "neo4j-advanced"
Type = "Module"
UniqueId = "AR1D\Users\l\m.graphml"
3
Name = "neo4j-management"
Type = "Module"
UniqueId = "AR2D\Users\l\m.graphml"

Figure 6.7: Property graph representation of three architectural modules.

the given element is transformed to a property as shown by nodes 2 and 3 in Figure 6.3.

6.3.1.6 UML Design Diagram: Sequence diagram

UML sequence diagrams possess a number of structural elements as derived from OMG's formal specification [166]. For the current implementation the *lifeline* element is considered, which captures an object, a class or a use case. Figure 6.9 illustrates an example from the MyRobotLab

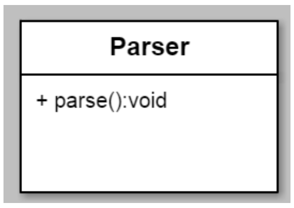


Figure 6.8: Example UML class *Parser* and its member method *parse*.

system[5], consisting of the *Service* and *Communication Manager* lifelines, and *send* messages. Lifeline elements are mapped to graph nodes and can be described by a *name* and *type* property. In the example shown in Figure 6.10, node 1 represents the *Service* lifeline, which is connected to a Java source code class, depicted by node 2.

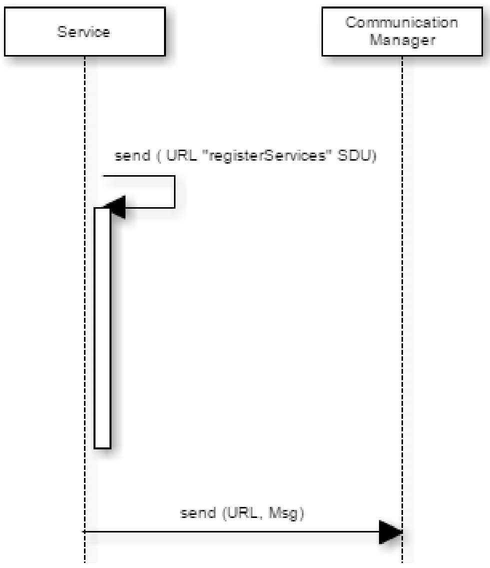


Figure 6.9: Sequence diagram fragment from the MyRobotLab system.

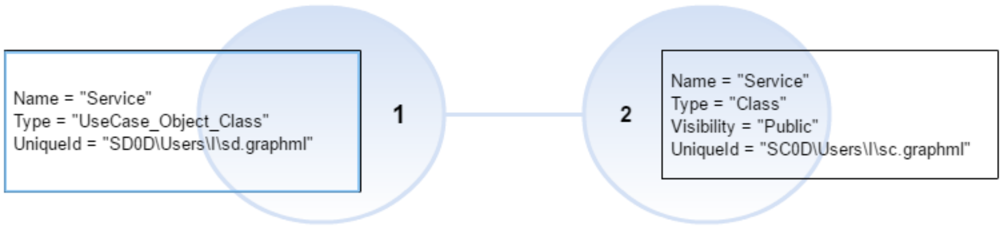

Name = "Service"
Type = "UseCase_Object_Class"
UniqueId = "SD0D\Users\I\sd.graphml"
1
2
Name = "Service"
Type = "Class"
Visibility = "Public"
UniqueId = "SC0D\Users\I\sc.graphml"

Figure 6.10: Property graph representation of the *Service* Lifeline element.

[5]https://github.com/MyRobotLab/myrobotlab

6.3.1.7 Java source code

Java source code shares common structural elements with UML class diagrams, such as *classes*, *interfaces*, *enums*, *methods*, and *fields*. Additionally, *.java* files contain implementation specific elements, which are detailed in the Java language specification [169]. The derivation of properties from Java source code follows the same process as that of UML class diagrams. The original and property graph representations are included in Listing 6.1 and Figure 6.1, respectively.

```java
package banking;
public class Account
{
  private float balance;
  public float getBalance()
  {
    return balance;
  }
}
```

Listing 6.1: Java source code excerpt of a class with a field and getter method.

6.3.1.8 JUnit test

The property graph representation of JUnit tests is identical to that of Java source code, since no additional elements are required to be captured for the purposes of the framework.

6.3.1.9 Element Hierarchy: Container and Member Elements

As noted earlier, some UML elements may contain further UML elements. For example, a class may contain a number of methods. This statement can be extended to other artefact types characterised by a hierarchical structure. Elements containing others are defined as *container (parent)* elements, whereas contained elements are called *members (children).* A summary of this hierarchical categorisation is given in Table 6.1. For the purposes of this work, requirements, use cases and UML sequence diagram lifelines are regarded as member elements. On the other hand, architectural components and modules may contain further components and modules.

6.3.2 Specification of Graph Edges and Properties

The classification of the two fundamental types of trace links, *intra* and *inter* links, is provided in Chapter 2. In a similar manner to deriving artefact elements and properties, firstly, the original format of trace links is discussed. Subsequently, their property graph model counterpart is introduced along with corresponding properties.

Element Type	Container	Member	Artefact Type
Field		Yes	Source code / UML class diagram / JUnit test case
Method		Yes	Source code / UML class diagram / JUnit test case
Class	Yes		Source code / UML class diagram / JUnit test case
Enum	Yes		Source code / UML class diagram / JUnit test case
Interface	Yes		Source code / UML class diagram / JUnit test case
Requirement		Yes	Requirement specification
Component	Yes		Conceptual architecture
Module	Yes		Module view architecture
Use case		Yes	UML use case diagram
Lifeline		Yes	UML sequence diagram

Table 6.1: Categorisation of artefact elements based on their hierarchical relationships.

Inter Links

An *inter* trace link can be illustrated, for example by a connection between a Java class and a requirement, as shown in Figure 6.1. Inter trace links are not explicitly present in the original representations of the artefacts as establishing them requires knowledge of the system. In case they are recorded, they are captured outside the artefacts. Thus, trace links are made explicit when they are modelled as graph edges. Each edge can be annotated with a number of properties. In the current implementation, links are assigned a *type* property, which for inter links takes the value *INTER_LINK*. Finally, links may also be characterised by their directionality, as discussed in Chapter 2. In the current implementation inter and intra trace links are bidirectional.

Intra Links

An example of *intra* links is the connection between a Java class and its member method. This relationship is implicit in the original format of the source code artefact, i.e. the *.java* file shown in Listing 6.1. However, through the property graph representation the relationship becomes explicit and is modelled as an edge between nodes corresponding to the given class and its member method. Similarly to *inter* links, *intra* links can also be described by properties.

6.3.3 Conclusions

In summary, the above examples demonstrate that property graphs offer a flexible means of modelling heterogeneous artefacts. Should it be required, the property graph can be extended to represent additional artefact types. In a similar manner to existing artefacts, when a new artefact type is considered, its structural elements and trace links can be represented as graph nodes and edges following the same steps.

Artefacts characterised by a hierarchical structure also map to a hierarchical representation in the property graph. However, in cases when intra connections are implicit or not recorded, such as a relationship between two requirements in a specification, the individual requirements are captured as independent nodes.

6.4 Bridging the Gap between Heterogeneous Artefacts and the Property Graph Model

A pivotal question of framework implementation concerns obtaining data from heterogeneous representations. The high-level approach is described in Chapter 4, while the next subsections introduce the steps taken to achieve the setting up of *Artefact Data* and bridge the gap between the original representations and the property graph model. Firstly, *artefact data extraction* is introduced in Subsection 6.4.1, followed by the implementation specifics of *transformation* in Subsection 6.4.2 and a discussion of the selected *graph database* in Subsection 6.4.3.

6.4.1 Artefact Data Extraction

The first step towards achieving artefact and tool independence in the framework is the extraction of artefact and trace link data from original representations. A framework prerequisite is a tool's capability to provide an XML-based representation of its data, which allows transformation to take place. Some tools and artefact types can be programmatically accessed to extract artefact data, while others involve manual aspects. Firstly, the tools used to create original artefacts are described, followed by the discussion of the export functionality.

6.4.1.1 Tools

The artefacts handled by the framework are originally created in a variety of tools. Since certain artefact types are not readily available from open source repositories, sample artefacts are needed to provide a wide variety of representations to demonstrate framework implementation, and subsequently its evaluation. For this reason, a sample natural language requirements specification and multiple UML class diagrams were created.

Requirements specifications can be written using a number of word processing applications and requirements management tools [170]. Since the main criterion of selecting a tool is the ease of access it provides to its artefact data and no other constraints are present, for the purposes of the framework, open source word processing solutions are considered. The selected tool for this work is *OpenOffice Write*[6], since a number of APIs can be used to access and manipulate documents created in *.odt* format. In particular, the *Apache ODF Toolkit*[7] returns text contained within the document as a single string, where the required elements can be selected and manipulated.

Other sample artefacts used in the framework are extracted from open source repositories, where their file format is already given. A wide variety of UML diagramming tools are available [171]. For the purposes of the framework, a suitable UML tool proved to be *Dia*[8], which supports exporting to various formats, including XML-based ones. Java source code can be written using a wide range of tools: from simple *text editors* to *IDEs*, including *NetBeans*[9], *IntelliJ IDEA*[10] and *Eclipse*[11]. *IDEs* can also be used to create JUnit test cases.

6.4.1.2 Extraction

The data extraction process involves exporting data from the selected tools manually, as is the case with *Dia* and UML class diagrams, or programmatically. Requirements specifications related artefact data can be obtained using the aforementioned *Apache ODT Toolkit API* from Java. Options for source code extraction are greater and various solutions were considered including *JavaML*[12], the XML vocabulary for representing Java source code, and *BeautyJ*[13], which converts Java source code to XJava XML. Eventually, the lightweight command line tool, *srcML*, was selected, which allows the creation of an XML representation of Java, C/C++ and C# source code by combining source code (text) and AST information (markup tags) [172]. Using the tool, it is possible to perform a one-to-one mapping of *.java* files to *.java.xml*. Each artefact is mapped to physical files in a repository or file system differently. While a UML class diagram may be represented by a single *.dia* file, Java source code and JUnit artefacts are a composite of multiple *.java* files and therefore are extracted to multiple *.java.xml* files.

The extraction process and the output for each artefact type are illustrated in Figure 6.11. The extracted files are stored in the framework folder (*ACMF*), which is specified when the framework is first setup. The *ACMF* folder contains the following sub-folders: *ArchitectureConceptual, ArchitectureModuleView, Requirement, SourceCode, UMLClass, UMLSequence, UMLUseCase,* and *UnitTests*. Each artefact is extracted to its corresponding folder based on its type.

[6]https://www.openoffice.org/
[7]http://incubator.apache.org/odftoolkit/simple/
[8]https://wiki.gnome.org/Apps/Dia
[9]https://netbeans.org/
[10]https://www.jetbrains.com/idea/
[11]https://eclipse.org/
[12]http://paginas.fe.up.pt/ aaguiar/javaml/
[13]https://sourceforge.net/projects/beautyj.berlios/

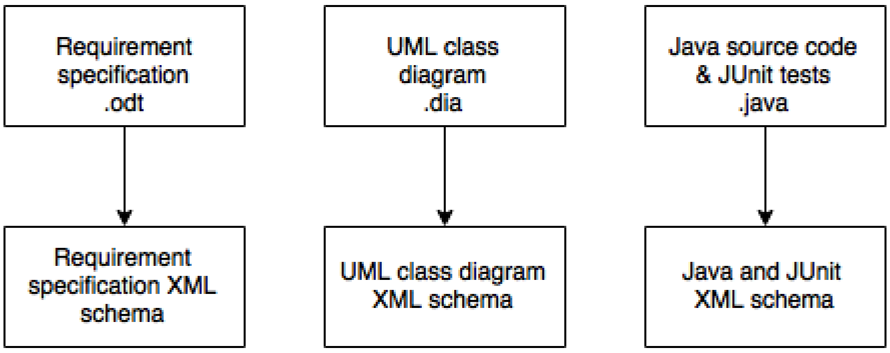

Requirement specification .odt
UML class diagram .dia
Java source code & JUnit tests .java
Requirement specification XML schema
UML class diagram XML schema
Java and JUnit XML schema

Figure 6.11: Artefact data extraction.

6.4.2 Transformation

Following the extraction, transformation aims to map heterogeneous XML documents to a uniform representation. Specifying a uniform representation allows artefact and link data to ultimately be saved in the data store, where they are represented as a property graph. The transformation functionality has raised numerous implementation-level considerations. Firstly, the format and the schema of the uniform representation is selected. Secondly, the strategy for establishing trace links using this representation is considered.

In terms of formats, the first alternative considered was a custom XML schema to represent artefacts of various types. According to this schema, both artefact elements and relations are uniquely identified and properties of both entities can be expressed through custom elements. The main advantages of this approach are the flexibility offered by XML and the freedom to specify the custom schema. However, adopting a custom XML-based solution involves handling issues that are already addressed by formats readily available to represent property graph concepts. These include the identification of elements, granularity of information, the ability to store generic data effectively, directionality of links, and most importantly, issues related to linking elements. One consideration is whether link information should be stored in the graph XML file or as a separate file.

6.4.2.1 Transformation: GraphML

A comprehensive survey of graph exchange formats reveals that a number of file formats are available to model, store and exchange graph data [173]. A few examples include the *Graph Modelling Language (GML)*[14], the *Graph eXchange Language (GXL)* [15], and the *LEMON Graph*

[14]https://www.fim.uni-passau.de/fileadmin/files/lehrstuhl/brandenburg/projekte/gml/gml-technical-report.pdf
[15]http://www.gupro.de/GXL/Introduction/intro.html

Format (LGF)[16]. To unify heterogeneous artefact and link data, *GraphML* [174] was chosen, which is used to describe graph structures and to represent application specific data.

```
<graphml>
<key attr.name="name" attr.type="string" for="node" id="d0"/>
<key attr.name="visibility" attr.type="string" for="node" id="d1"/>
<key attr.name="variableType" attr.type="string" for="node" id="d2"/>
<key attr.name="parameters" attr.type="string" for="node" id="d4"/>
<key attr.name="returnType" attr.type="string" for="node" id="d5"/>
<key attr.name="type" attr.type="string" for="node" id="d6"/>
<key attr.name="relType" attr.type="string" for="edge" id="d7"/>
<key attr.name="uniqueId" attr.type="string" for="node" id="d8"/>
  <graph edgedefault="undirected" id="DI">
    <node id="1">
    <data key="d0">Account</data>
    <data key="d1">Public</data>
    <data key="d2"/>
    <data key="d6">class</data>
    <data key="d8">Unique id value</data>
    </node>
    <node id="2">
    <data key="d0">getAccountNo</data>
    <data key="d1">Public</data>
    <data key="d5">String</data>
    <data key="d4"/>
    <data key="d6">UMLOperation</data>
    <data key="d8">di1/Users/ildikopete/Dropbox/PhD/SharedBackup/Evaluation/
        MazeSolver/Evaluation Files/UML/Revision19/XML/OldVersion/
        revision19Old.vdx</data>
    </node>
    <node id="3">
    <data key="d0">balance</data>
    <data key="d1">Private</data>
    <data key="d2">int</data>
    <data key="d6">UMLAttribute</data>
    <data key="d8">di22/Users/ildikopete/Dropbox/PhD/SharedBackup/Evaluation
        /MazeSolver/Evaluation Files/UML/Revision19/XML/OldVersion/
        revision19Old.vdx</data>
    </node>
    <edge id="diE2" source="1" target="2">
    <data key="d1">Parent_Child</data>
    </edge>
    <edge id="diE2" source="1" target="3">
    <data key="d1">Parent_Child</data>
    </edge>
  </graph>
</graphml>
```

Listing 6.2: Example GraphML file modelling a UML class diagram and its property graph representation.

[16]https://lemon.cs.elte.hu/trac/lemon

GraphML is best introduced through a concrete example. Listing 6.2 shows a *GraphML* file representing a UML class diagram, which is depicted in Figure 6.12. The properties of graph nodes and edges are derived based on the process introduced in Section 6.3.1. Artefact element properties are defined by the *key* element in the *GraphML* file. *Keys* have *identifiers*, *names*, *types* and a *domain attribute* specifying the element the given property is assigned to, as properties can be associated with edges, nodes or both. *Node* elements denote graph nodes, while *edge* elements stand for graph edges. The values of artefact element properties are defined by *data* elements nested in *Node* elements, whereas edge properties are specified in *data* elements inside *Edge* elements. Other artefact types with their corresponding elements, properties and connections are described in a similar manner in *GraphML* using the appropriate property keys and their values.

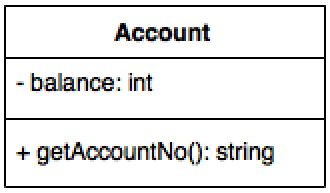


Figure 6.12: The Account UML class and its members.

Table 6.2 summarises the properties used in the framework describing the current set of selected artefacts. Should the framework be extended with new artefact types, further properties can be added. When adding new properties, a convention to be taken into account is that *GraphML key* element names are reserved to denote existing properties, and they should not be overridden by new ones. For example, regardless of the artefact type, *D8* should always stand for the *unique id* property.

6.4.2.2 Transformation: XSLT

The transformation functionality is implemented using XSLT transformations. Alternatives considered include the DOM[17], SAX[18] and JAXP Java parsers[19]. Each artefact type has a corresponding XSLT file, which transforms the XML-based extracted artefact data to the custom *GraphML* schema specified in Listing 6.2. The XSLT approach has proved to be a flexible one, as it allows the extension of the framework without major refactoring should new artefacts be added. In case a new artefact is introduced, its corresponding XSLT has to be supplied.

[17]https://docs.oracle.com/javase/tutorial/jaxp/dom/readingXML.html
[18]https://docs.oracle.com/javase/tutorial/jaxp/sax/parsing.html
[19]https://docs.oracle.com/cd/B28359_01/appdev.111/b28394/adx_j_parser.htm

Property	**Key Value**	**Description**	**Example Values**
Name	The name of the artefact element as specified in the original tool	d0	getAccountNo, Account, balance
Visibility	Visibility modifier	d1	Public, Private, Protected
VariableType	The data type of a variable (field / attribute)	d2	Int, String, Object
Parameters	Parameter list (methods / operations)	d4	(int a, String b)
ReturnType	Return value (methods / operations)	d5	Void, String, Int
Type	The type of the entity specified in the framework	d6	Method, Enum, Component
RelType	The type of the relationship specified in the framework	d7	Parent_Child, Uses
UniqueId	An identifier generated to uniquely identify artefact elements	d8	SC0D:/file.graphml
Content	The body of a member element (methods / requirements)	d3	return null; "The system shall..."
Title	The title of a requirement	d11	"User input"
Priority	The priority value associated with a requirement	d12	High
ReqType	The type of a requirement	d13	Functional

Table 6.2: Property key/value pairs used in the framework.

6.4.2.3 Transformation Output

The output of the transformation process is summarised in Table 6.3, which highlights the way original artefacts are mapped to the uniform *GraphML* format. Since a source code repository may contain hundreds of **.java** files, the mapping process of Java source code and JUnit test artefacts differs from other types. Each *.java* file is transformed to a *GraphML* representation as combining all Java source code or JUnit test artefact data to a single file may not be viable due to the possible overhead caused by handling large files. Despite the separate storage, the *GraphML* files are logically the same artefact.

6.4.2.4 Extracting and Transforming Trace Links

Besides artefact elements, *Artefact Data* also contains trace links, which connect these elements. *Intra* links are extracted from original artefacts when performing artefact data extraction and transformation. These links are denoted by *Edge* elements in the *GraphML* representation. *Inter*

Artefact	**GraphML Representation**
Java source code	A GraphML file represents a single Java class/interface/enum
UML class diagram	A GraphML file represents an entire UML class diagram that may consist of multiple classes/interfaces/enums
JUnit test	A GraphML file represents a single Java test class
UML sequence diagram	A GraphML file represents an entire UML sequence diagram consisting of multiple lifelines
Requirements specification	A GraphML file represents an entire document, which may contain multiple requirements
UML use case diagram	A GraphML file represents an entire UML use case diagram
Conceptual architecture	A GraphML file represents the entire architecture
Module view architecture	A GraphML file represents the entire architecture

Table 6.3: Transformation output summary.

links, however, require specific traceability creation mechanisms to be in place to establish them.

Intra Links Establishment

Domain dependency links are either explicitly specified in the original artefact or they can be derived. For example an association between two UML classes that is drawn on the diagram can be extracted using the same principle applied in extracting artefact elements: the XML representation of data obtained from the original artefacts also contains links between artefact elements. This class of links is annotated with a generic *Uses* label expressed through the *relType* key in the *GraphML* representation.

Implicitly stored relationships such as the ones between Java classes and their member methods are annotated with a *Parent-Child* label. The current implementation supports the extraction of such relationships from Java source code, UML, and JUnit test cases.

Inter Links Establishment

Establishing inter artefact trace links is currently achieved using machine learning techniques resulting in a semi-automatic process, which is described in Chapter 8. Since inter links are not represented by edges in the *GraphML* file, a custom representation is in place and inter trace links are stored in an XML file, where they are specified as follows.

```
<Relations>
  <Relation id="1">
  <SourceNode>x</SourceNode>
  <TargetNode>y</TargetNode>
```

```
    </Relation>
  </Relations>
```

Listing 6.3: An example inter trace link expressed in XML.

The motivation to store inter trace links separately from intra links and artefact elements relates to an implementation aspect, graph data persistence, which is discussed in the next subsection. The *Blueprints API* [20] allows *GraphML* files to be imported to graph databases through which *intra* links and artefact elements are saved. *Inter* links between nodes are established through an additional database update. However, adding *inter* links to the *GraphML* file would necessitate a custom logic to be built to save artefact data, *intra* links and *inter* links and accessing the database on multiple occasions. Additional repercussions may include the need to combine all *GraphML* files in a single file to avoid additional linking between files. Storing all artefact element data in a single file may present performance issues when dealing with large projects.

The following list outlines the strategy to establish *inter* trace links with respect to artefact types currently handled by the framework, and the granularity of linking. Creating inter links involves understanding the system and the concepts the given artefacts represent. For any given pair of source and target artefact, elements at the same level are connected. For example, container elements in Java source code are connected to container elements in UML class diagrams. On the other hand, member elements are linked to other members. The list also highlights that each artefact type is connected to all the other types. The motivation to cover inter links between all artefact types is to support artefact independence. Specific examples of artefact linking taken from open source systems are introduced in Chapter 8. An alternative linking strategy may utilise transitivity, where an identified link between element *a* and *b*, and a link between *b* and *c* allows the conclusion to be drawn that *a* and *c* are also connected without a link being created explicitly. This strategy may be implemented as part of future work.

1. Source artefact: *Java source code*
   Java source code container elements are connected to UML / JUnit container, UML sequence diagram lifeline, requirement, use case, architectural component, architecture module artefact elements. Java source code member elements are linked to UML / JUnit member elements.
2. Source artefact: *UML class diagram / JUnit tests / UML sequence diagram*
   UML / JUnit container and UML sequence diagram lifeline elements are connected to Java source code container, requirement, use case, architectural component and architecture module artefact elements. Additionally, UML / JUnit member elements are linked to Java source code and JUnit member elements.

[20]https://github.com/tinkerpop/blueprints/wiki

3. Source artefact: *Requirement specification*
   Requirement artefact elements are connected to use case, achitectural component, architecture module, UML sequence diagram lifeline, JUnit/ UML / Java container element(s).

4. Source artefact: *UML use case*
   Use case member elements are connected to requirement, achitectural component, architecture module, UML sequence diagram lifeline, JUnit/ UML / Java container element(s).
5. Source artefact: *Conceptual architecture*
   Architectural component elements are connected to requirement, use case, architectural module, sequence diagram lifeline, Java / UML / JUnit container element(s).
6. Source artefact: *Module view architecture*
   Module elements are connected to requirement, use case, architectural component, sequence diagram lifeline, Java / UML / JUnit container element(s).

6.4.2.5 Transformation Summary

In summary, at the end of the transformation process, the inputs are transformed to a uniform representation using *GraphML*. *Intra* links are stored within *GraphML* files, while *inter* links are captured in a separate XML file.

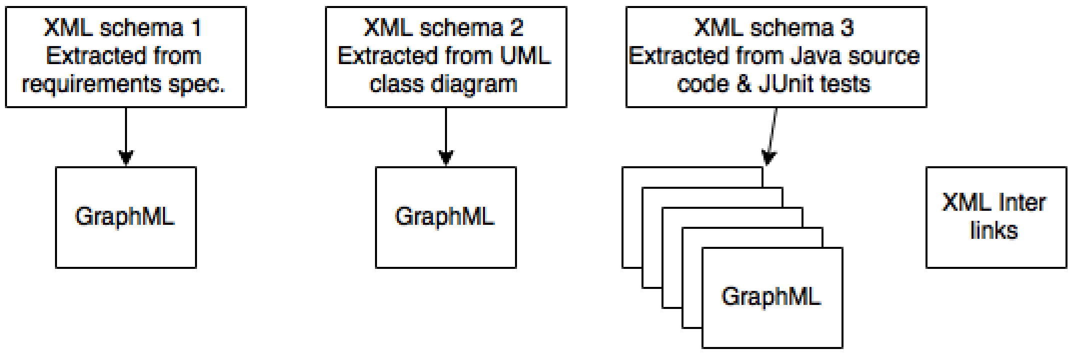


Figure 6.13: Mapping heterogeneous XML schemas to a uniform schema to represent artefact data.

6.4.3 Graph Data Persistence

The graph data required and utilised by the framework, possesses particular characteristics; the number of nodes may vary depending on the number of artefacts extracted from the given project. The number of edges varies similarly due to the types of artefacts and the complexity of relationships between them in a specific project. These attributes require a flexible and scalable storage solution to best fulfil the potential scenarios. Additionally, artefact and link data is prone to frequent modifications, which the selected persistence mechanism is required to handle effectively.

6.4.3.1 Graph Databases

One way of persisting a graph structure is by utilising graph databases, which is the selected approach in this work. This choice is evident given the representation of framework data. The decision is also underpinned by the numerous advantages offered by graph databases and conclusions drawn from a careful consideration of multiple alternatives. The emergence of big data and cloud computing has brought about requirements which are more adequately satisfied by NoSQL databases than with traditional relational database management systems [175]. Graph databases can be described as a specific type of NoSQL database tailored to handle large volumes of continually growing and highly connected data. Use cases of major graph databases include social networks, fraud detection and real-time recommendation engines [176].

A graph database is a database management system, which provides CRUD (Create, Read, Update, Delete) functionality and exposes a graph data model [177]. The graph data model, which dates back to the 1980s, is characterised by a graph data structure. Additionally, data manipulation is expressed by graph transformations, and it allows the definition of integrity constraints [178].

A graph data model is a particularly reasonable choice for data where the interconnectivity of data is as significant as the data itself, since connections are as important entities as nodes. In particular, the investigated graph databases support the property graph model, a directed, labelled multi-graph, which allows edges and nodes to maintain a key/value property map. Thus, it is possible to associate attribute metadata with nodes and edges enhancing their expressivity [157].

Graph databases offer numerous advantages and their use in the framework is underlined by the following key factors:

- **Property graph model.** This model allows efficient modelling of framework artefact and trace link data at a fine-grained level. The level of detail included on the graph through properties is customisable, which makes the model flexible to changes.
- **Framework functionality.** Besides providing storage for framework data, graph databases also assist with carrying out the consistency management tasks of the framework. Firstly, graph traversals make it possible to walk the graph based on pre-defined conditions and to establish subgraphs. This process is part of the change impact analysis functionality of the framework and is detailed in Chapter 7. To perform traversals, a number of query languages are available, such as Cypher[21] and Gremlin[22]. Other aspects of the framework require efficient querying of graphs, which is one of the most basic functionality of graph databases.

[21]http://neo4j.com/docs/stable/cypher-query-lang.html
[22]https://github.com/tinkerpop/gremlin/wiki

It is also important to note that the interoperability of graph databases with other technologies allows the framework to be easily extended, for example with visualisation techniques.

- **Performance considerations.** Graph databases provide index-free adjacency ("every element has a direct pointer to its adjacent element")[179] and represent an explicit graph where relationships are "first class citizens" [177]. These characteristics make it possible for graph databases to execute substantially more effective queries on the same data in comparison with relational databases, where graph data has to be mapped to tables and the queries require joins to be performed. These performance benefits guarantee that major performance-related issues are alleviated when reading and writing framework data. This is a significant consideration since a software project may contain a large number of connections.
- **Utilities and data maintenance.** Finally, in comparison with the *GraphML* file format, graph databases provide utilities aimed at the effective maintenance of the stored data, such as versioning, recovery, and backups to mention a few.

Possible disadvantages include the fact that most graph databases do not provide support for time-based versioning of property graphs [177]. Therefore, differences between two versions are captured using other means. Graph databases are a relatively new technology and different aspects of them are yet to reach a sufficient maturity level. Certain graph databases do not provide support for distributed storage, and query languages vary across databases. However, the area is actively growing, and various open source solutions aim to support interoperability between them.

6.4.3.2 Alternative Strategies

Besides utilising graph databases, various other means of modelling, storing graph data and managing its evolution are available.

Open source graph libraries allow the creation and manipulation of graphs. Additionally, capabilities to import, export, visualise and analyse graphs are supported. Examples of Java libraries include *GraphStream*[23] for directed and undirected multi-graphs, *Jung*[24] for directed and undirected multi-graphs providing entity annotations through metadata, *Grph*[25] providing a general model for graphs, and *JGraphT*[26] for various types of graphs. APIs written in other programming languages, such as *BGL*[27] in C++, are also widely used for solving graph-based problems.

[23]http://graphstream-project.org/
[24]http://jung.sourceforge.net/
[25]http://www.i3s.unice.fr/ hogie/grph/
[26]http://jgrapht.org/
[27]https://dl.acm.org/citation.cfm?id=504206

Besides graph exchange formats, non graph-specific formats can also be utilised to store and exchange graph data, such as a custom XML schema. Another alternative is the *Resource Description Framework (RDF)*, used in the semantic web, since information modelled in *RDF* can be viewed as a graph [159].

The libraries discussed above implement well-known graph algorithms and traversal mechanisms, which can be incorporated in the framework. These file formats allow graph data to be modelled. These properties alone are however not adequate in addressing the data description and storage requirements of the framework. Consequently, the combination of graph libraries and file formats raises questions regarding the handling of I/O operations, on-disk durability, memory management, change and transaction management of graph data, ease of querying large volumes of data, and the ability to scale to varying sizes of data. Graph exchange formats are predominantly concerned with the description of mostly static graph data [173], whereas graph databases, which are specifically aimed at managing dynamically changing graph data take care of such considerations and are better suited to model, store and query framework data.

Another aspect of vital importance is the specific graph model supported by any given solution. Accordingly, it is to be considered whether the model describes the data at the required level of detail and whether it is possible to modify it without significant effort should the data change. Notably, *GraphML* provides a suitable solution to model artefact data. However, when considering the number of artefacts and changes that may occur, querying and updating artefacts and links in a database seems to be a more robust approach.

6.4.3.3 Neo4j

Despite its relatively recent emergence, the field of graph databases is rapidly growing and numerous alternatives are available satisfying different requirements and usage scenarios. Graph databases include *Neo4j*[28], *Titan*[29], *Allegrograph* (proprietary)[30], *GraphDB*[31], *InfiniteGraph* (proprietary)[32], *OrientDB*[33], *InfoGrid*[34], *HypergraphDB*[35], and *Microsoft Trinity*[36]. Following a comparison and evaluation of the alternatives, *Neo4j* was selected as the storage backend for the ACM framework.

[28]http://neo4j.com/
[29]http://thinkaurelius.github.io/titan/
[30]http://franz.com/agraph/allegrograph/
[31]http://ontotext.com/products/graphdb/
[32]http://www.objectivity.com/products/infinitegraph/
[33]http://orientdb.com/orientdb/
[34]http://infogrid.org/trac/
[35]http://hypergraphdb.org/index
[36]http://research.microsoft.com/pubs/183710/Trinity.pdf

Since most graph databases provide full ACID transaction support, a graph model for data representation, indexing, and querying capabilities, a good basis for comparing them is their performance when being subjected to different data loads. Thorough performance introspections are available in literature discussing results for graph query languages and micro-operations (traversals and reading and writing individual elements to and from the database), graph operations (fundamental read/write graph operations, such as getting neighbours), and algorithms (shortest path, etc.) carried out using different graph databases [180] [181] [182] [183].

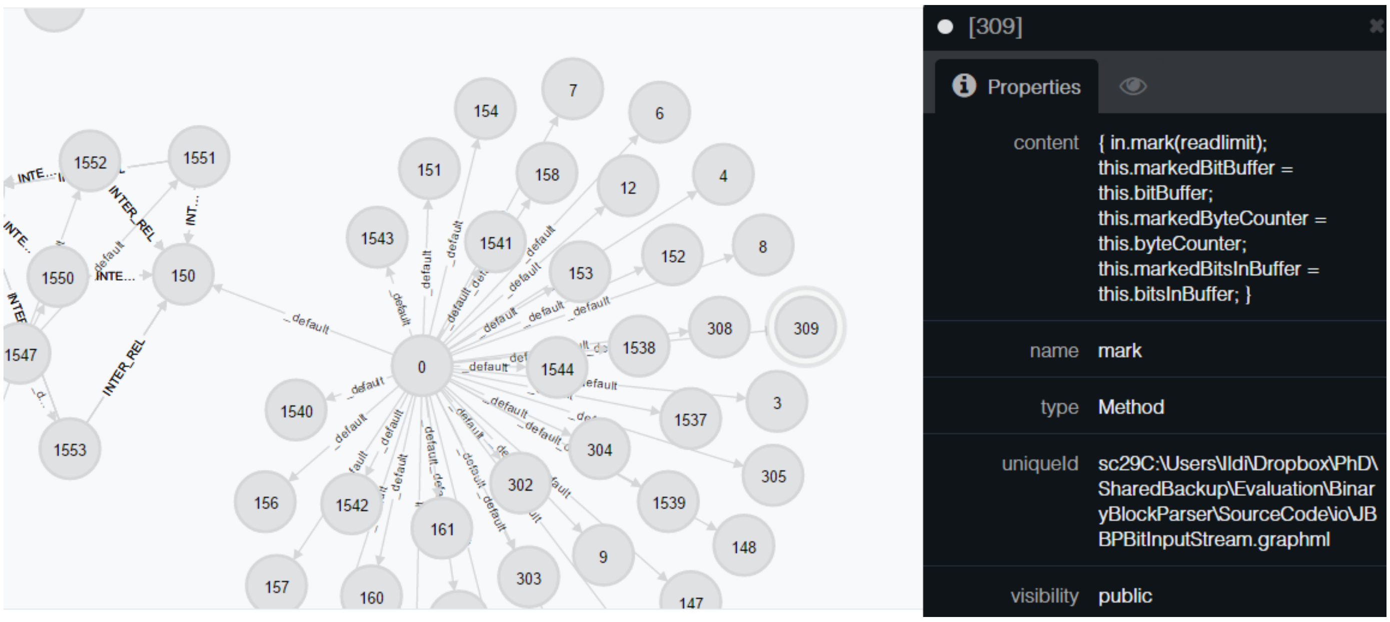
Besides results obtained from such studies, a performance comparison test of *Neo4j* and *Titan* was conducted: test artefacts and relationships were generated and the execution time of saving data in the databases was measured. The results highlight that *Neo4j* is a more suitable choice. However, *Titan* is part of a well-developed infrastructure of products that supports various aspects of solving problems using graphs [184].

Another aspect that contributed to the selection of *Neo4j* is the maturity of the project in comparison with other alternatives, and the support available through its community. *Cypher*, the graph query language offered by *Neo4j*, is a pattern-matching query language allowing traversals and graph operations. There are multiple ways of accessing and using the *Neo4j* database. At the outset, the *Neo4j* Server, to be accessed through its HTTP API, was considered, which was changed in favour of using *Neo4j* embedded in the JVM process. This allows the leveraging of the *Neo4j* core-Java-API providing high-speed traversals and a mapping of graph database concepts to Java objects [176]. Should current requirements change, the design of the *Data Access Layer* allows the current implementation to be substituted to alternative ones. Accessing the database through Java is a natural choice as the framework is implemented in Java.

Figure 6.14 shows a fraction of artefact and trace link data in *Neo4j*. Each node denotes an artefact element, each connection annotated with the keyword *_default* is an *intra* trace link, and each link labelled as *INTER_REL* is an *inter* trace link. The highlighted graph node stands for a public Java method, called *mark*. The method contents and unique identifiers are also illustrated.

6.5 Conclusions

The areas described in the preceding sections encompass functionality to establish *Artefact Data* in the holistic framework consisting of a property graph of nodes (artefact elements) and edges (trace links). At the outset, heterogeneous artefacts supply input for extraction. The output of extraction is an XML-based representation, which is transformed to a unified format to express artefact elements and their connections. Finally, artefact data is saved to the *Neo4j* graph database.


[309]

Properties

content	{ in.mark(readlimit); this.markedBitBuffer = this.bitBuffer; this.markedByteCounter = this.byteCounter; this.markedBitsInBuffer = this.bitsInBuffer; }
name	mark
type	Method
uniqueId	sc29C:\Users\Ildi\Dropbox\PhD\SharedBackup\Evaluation\BinaryBlockParser\SourceCode\io\JBBPBitInputStream.graphml
visibility	public

Figure 6.14: Artefact and trace link data in Neo4j.

This implementation approach takes various alternative solutions into account in each major step, along with possible advantages and drawbacks of the alternatives. The implementation is guided by high-level design principles and aims at maximising automation as far as possible. A notable shortcoming of the current implementation is that a layer of potential inconsistency is introduced due to the multiple representations. In the event of a change affecting an original artefact, the XML-based and *GraphML* representations have to be re-generated and updated. This issue is introduced by the gap that exists between heterogeneous formats. The property graph representation and the current implementation approach seem to be the most viable means to address this issue in comparison with alternatives considered.

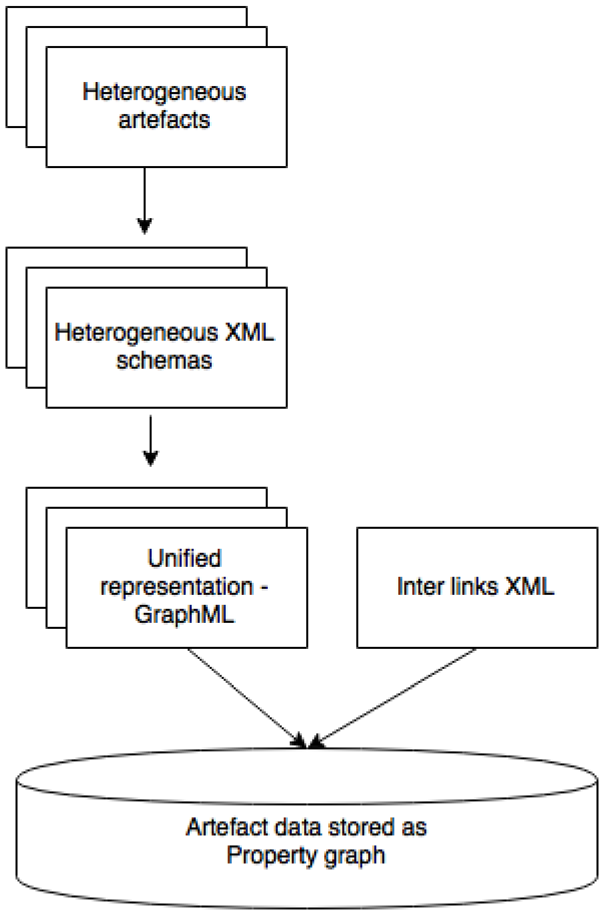

Heterogeneous artefacts
Heterogeneous XML schemas
Unified representation - GraphML
Inter links XML
Artefact data stored as Property graph

Figure 6.15: Framework setup functionality.

Chapter Seven

Implementation of the ACM Framework: Framework Stages

7.1 Introduction

Following the discussion of *Artefact Data* in Chapter 6, the implementation specifics of the framework stages including *Change Detection, Change Impact Analysis, Consistency Checking*, and *Change Propagation* are now presented. The discussion excludes the approach adopted for implementing the *Traceability creation* stage, which in turn is introduced in Chapter 8. This chapter concludes with an evaluation of the overall implementation of the ACM framework.

7.2 Change Detection

The aim of change detection is to reveal which original artefact have changed and how. This change information provides the required input for subsequent framework stages. Change detection is also responsible for carrying out further tasks to update the *XML*, *GraphML* representations and the graph database. This subsection firstly presents an overall summary of change detection. Next, details concerning the approach to identify the *file level* and *artefact element level* changes are given. A flow chart summarising the steps involved in change detection is shown in Figure 7.1.

The starting point of the process is the identification of changed artefacts and the file level change type. This information is extracted from the external repository where original artefacts are stored. Next, based on the *file level* change type, different actions are taken, which are summarised in Table 7.1.

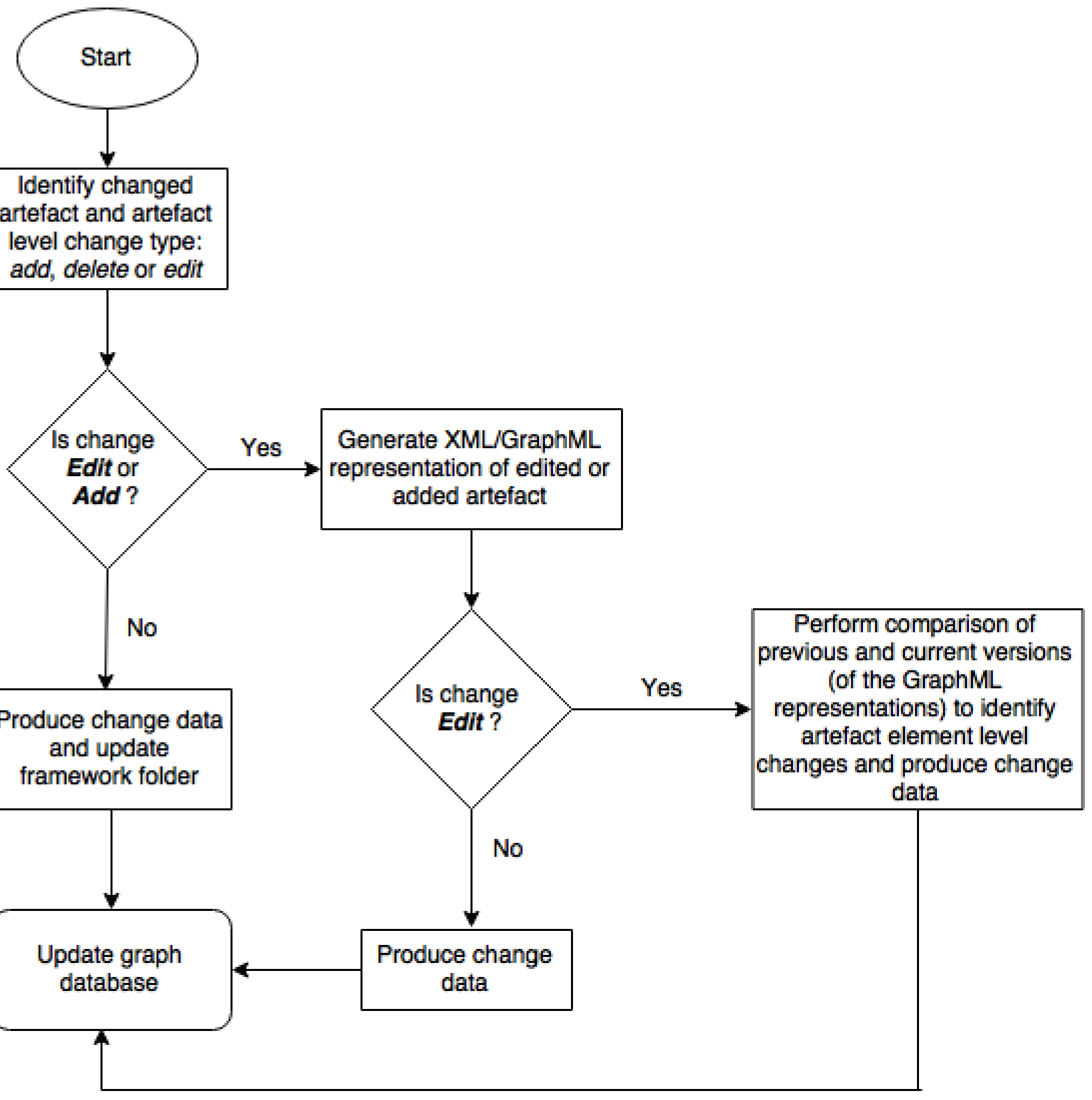

Start
Identify changed artefact and artefact level change type: *add*, *delete* or *edit*
Is change *Edit* or *Add* ?
Yes
No
Generate XML/GraphML representation of edited or added artefact
Is change *Edit* ?
Yes
No
Perform comparison of previous and current versions (of the GraphML representations) to identify artefact element level changes and produce change data
Produce change data and update framework folder
Produce change data
Update graph database

Figure 7.1: Change detection overview.

In case an artefact is *deleted*, all the necessary data for updating the graph database is in place and the update (labelling nodes to be removed) can be carried out. The output of the process is a list of *ChangeData* objects, which is passed to the subsequent framework stage. The framework folder and its subfolders are also updated accordingly: corresponding *XML* and *GraphML* representations are removed as they contain outdated artefact data and are no longer required by any of the framework stages.

Edit and *add* file level changes require additional processing. In both cases, an *XML* and *GraphML* representation are generated to represent the latest version of the given artefact. In

	Delete	**Add**	**Edit**
File System►	Remove artefact from file system	Add artefact to file system	Add new version of artefact to file system and label previous version with suffix
Graph database►	Label nodes as *toBeRemoved*	Import new artefact from GraphML file	Update property values of existing node, add new nodes, delete nodes

Table 7.1: Summary of actions taken depending on the file level change type.

case of an *add* change, ChangeData objects are produced and the graph database is updated by importing the new *GraphML* representations. For *edit* changes, both the previous and current versions of the *GraphML* representation are kept, which, using the proposed change identification algorithm, are compared. The result of the comparison is a list of *ChangeData* objects, which constitute the output of change detection and are used to perform the database update. In case of *add* and *delete artefact element level* changes, database nodes and edges representing the modified elements and their trace links are added and deleted, respectively. *Editing* artefact elements results in updating the properties of nodes representing the elements in the database. *Inter* links are updated based on inter trace link maintenance rules, which are discussed in Section 7.3.

7.2.1 Specifics of Changes

The change classification adopted in this work and example scenarios of *file level* changes are discussed in Chapter 4. In the following section some specifics of *file level* changes and a detailed discussion of *artefact element level* changes are presented.

In terms of *delete file level* changes, due to the unique characteristics of some artefacts, such as Java source code and JUnit tests, deleting a *.java* file equates to the *delete file level* change despite not all *.java* files representing the artefact are removed. On the other hand, an entire UML class diagram artefact can be removed by a single delete operation. The framework caters for such scenarios irrespective of their likelihood. Table B.6 in Appendix B shows specific examples of *delete* and *add* file level changes for each artefact type.

To identify *artefact element level* change types, different scenarios are considered based on the original artefacts and their main structural elements, which is highlighted in Table B.7 in Appendix B. It is important to note that some structural elements, such as comments in source code, are not relevant since managing their consistency is not in the scope of this work.

Based on the listing of concrete changes, the following conclusions can be drawn. Firstly, artefact element level changes can be further decomposed. For example, when editing Java source code and JUnit test cases, these modifications may either affect the contents of the artefact element, or its signature. Therefore *edit* type changes are classified as *signature*, *content* type changes or both. Secondly, the differentiation of *container* and *member* elements creates another dimension to categorising changes: each *artefact element level* change can be an *edit*, *add* or *delete* of a *container* or a *member* element.

Lastly, following the investigation of change types, it is concluded that apart from a generic categorisation of change types and the identification of the modified artefact, the framework requires a finer-grained change specification. Changes should be defined at the property graph level similarly to artefact and trace link data. By doing so, it is possible to express changed entities as nodes described by properties, and changes as addition or deletion of nodes, or as editing node properties.

7.2.2 Identification of the *File Level* Change Type

The framework identifies the *file level* change by pulling changes from external repositories. As noted earlier, the framework assumes that artefacts are checked into version control to provide access to them. In the current implementation changes are pulled from the repository by users invoking the change detection functionality. However, a number of alternative solutions present themselves in this context and may be implemented as part of future work. For example, change detection may be initiated automatically with the frequency of detection set by the user, similarly to specifying the types of changes, and artefacts of interest. The result of the identification of the *file level change* type is a list of files in the repository that have been added, deleted, or edited as part of the latest commit.

Reading and writing to a version control system can be implemented by utilising the APIs offering access to repositories. The first implementation concern concerns selecting a system that provides the required data in the most straightforward way. Popular open source version control systems include the centralised *SVN*[1], and the distributed *Git*[2] and *Mercurial*[3]. For the ACM framework, *Mercurial* was selected as it is widely used and it can be accessed either through the numerous available APIs, such as hg4j[4], or directly by using scripts. To return the list of added, modified and deleted files, the framework utilises a script. This information is sufficient to identify which original artefact was subject to a change and to establish the *file level* change

[1]https://subversion.apache.org/
[2]https://git-scm.com/
[3]https://www.mercurial-scm.org/
[4]http://hg4j.com/

type. In case of *delete* and *add file level* changes, file system and graph database operations take place. However, in case an artefact is modified through an *edit file level* change, the next step is the identification of the *artefact element level* change type, which is discussed in the next section.

7.2.3 Identification of the *Artefact Element Level* Change Type

Two pivotal questions relating to the identification of *artefact element level* changes include how deltas between two versions of an artefact are computed, and how these are represented and captured in the framework. The idea of utilising *difference algorithms* [185] used in a number of scenarios, including version control systems, was dismissed since a line or string based comparison is inadequate for the purposes of the framework where both the changed entity and the type of change are relevant and need to be identified. Hence, the delta produced by any approach should reflect the change type combined with the identification of an entity that is meaningful from the framework's perspective. The output of change detection allows information, such as the following statement, to be derived: "the signature of the Java class *Account* was edited by modifying its access modifier from public to protected". The level of detail captured by change detection corresponds to the level of detail captured by the property graph representation of artefact elements. For example, since import statements in Java source code are not mapped to a node in the property graph, their modifications are not recorded in the framework. Prior to introducing the graph-based approach, the next section discusses the alternatives.

7.2.3.1 Change Identification and Representation: XML

A potential approach for representing and capturing changed entities is utilising a generic XML solution. This seems to be a viable approach due to the fact that artefacts are stored in an XML-based format and XML is a flexible solution for representing concepts. XML change detection is used in a number of application areas, such as version management of documents, and various solutions are available to detect and represent changes. These are reviewed by Cobena et al. in depth [186]. Ones that are of particular importance are the algorithms that handle XML documents as tree data: XML documents can be represented as ordered, labelled trees. Thus, finding changes between two XML documents can be seen as the "tree-to-tree correction problem" for ordered labelled trees [187].

Existing tools for XML differencing include *DeltaXML* [5], which allows the storing of delta information in the form of change attributes in the original XML document. Another approach is

[5] http://www.deltaxml.com/products/core/

XyDelta [6], where each node in the XML document is given a unique identifier, while identifiers that are different between the two versions represent the corresponding operations. Further examples include the proprietary *TreeDiff* [7], *diffMK* [8] and *VM Tools* [9].

7.2.3.2 Change Identification and Representation: Graph-based Approach

For the purposes of this work, instead of XML differencing, a graph-based approach was chosen to identify and represent modified entities and their respective change types. The motivating factor is that since artefact data is already represented as graph nodes, no conversion to another representation is required. The higher the number of interim representations, the higher the potential for consistency issues to arise, should any of these representations change. Additionally, the graph-based approach makes it possible to express changes and differences between two versions of artefacts at the graph level. Since graph nodes model artefact elements, the translation of changes to the artefact element level is straightforward.

The idea of using graph-based differencing of versions of a software system is not new and has been expressed in the literature [188]. A graph-based approach presents a number of alternatives for implementation. One example is the format used for storing change data; the framework may utilise a custom XML-based format or the *GraphML* format can be extended to cater for expressing changes. Another consideration is the level at which graph differencing is performed, such as at the graph database or at the *GraphML* level, using Java APIs or custom objects.

This work adopts some aspects of the *XyDelta* change representation model to identify the change type and the changed entity. In *XyDelta* nodes in the original XML document are given unique identifiers, which are stored in the XidMap. The delta between two versions of the XML document is expressed through the operations of these identifiers: if an identifier is not found in the new version, it corresponds to a node that has been deleted [186].

GraphML File Differencing

In the approach adopted in this work, each node property is mapped to a *key-value* representation, which uniquely identifies property values of graph nodes, through which graph nodes and their changes can be identified. The approach is illustrated through an example shown in Listing 7.1. As mentioned in Chapter 6, each data element in the *GraphML* file has a key attribute with pre-defined values. These attributes can take the role of keys in the *key-value* pairs. Every data element has a value associated with it, which can be assigned a unique identifier such as $V + a$

[6]http://www.dia.uniroma3.it/ vldbproc/062_581.pdf
[7]http://www.xml.com/pub/r/536
[8]http://www.w3.org/2008/05/xmlspec-diff-generation/
[9]http://www.vmsystems.net/vmtools/doc/

sequential number. In case two data elements have the same value, they are assigned the same identifier, i.e. the same value in the *key-value* pair. In this example the *D1* key of both nodes has the same value (public), hence they are both assigned the value *V1*. Unique identifiers of nodes represented by *D8* data keys are excluded from differencing since they are not the same across two versions of the *GraphML* file. Hence, Every *D8* key is assigned a set value.

```
<node id="1">
  <data key="d0">BitIOCommonTest</data>-->V0
  <data key="d1">public</data>-->V1
  <data key="d6">class</data>-->V2
  <data key="d8">sc0Path</data>-->V3
  <data key="d9"/>-->V4
</node>
<node id="2">
  <data key="d0">rnd</data>-->V5
  <data key="d1">public</data>-->V1
  <data key="d2">Random</data>-->V6
  <data key="d6">Field</data>-->V7
  <data key="d8">sc1Path</data>-->V3
</node>
```

Listing 7.1: GraphML file, version *n* - mapping to key-value pairs

The above nodes shown in Listing 7.1 can be represented as collections of *key-value* pairs. A graph *G*, which consists of these two nodes, may therefore be defined as:

```
Let graph G:
N1 --> (D0 --> V0, D1 --> V1, D6 --> V2, D8 --> V3, D9 --> V4)
N2 --> (D0 --> V5, D1 --> V1, D2 --> V6, D6 --> V7, D8 --> V3)
```

Listing 7.2: Graph *G* defined as a collection of key-value pairs

The subsequent (modified) version of the same *GraphML* file is presented in Listing 7.4: the *D1* data key of the first node was edited from *public* to *protected*. Graph *G'*, which is version *n+1* of graph *G*, is therefore defined as:

```
Let graph G':
N1 --> (D0 --> V0, D1 --> V8, D6 --> V2, D8 --> V3, D9 --> V4)
N2 --> (D0 --> V5, D1 --> V1, D2 --> V6, D6 --> V7, D8 --> V3)
```

Listing 7.3: Graph *G'* defined as a collection of key-value pairs

```
<node id="1">
  <data key="d0">BitIOCommonTest</data>-->V0
  <data key="d1">protected</data>-->V8
  <data key="d6">class</data>-->V2
  <data key="d8">sc0Path</data>-->V3
  <data key="d9"/>-->V4
</node>
<node id="2">
  <data key="d0">rnd</data>-->V5
  <data key="d1">public</data>-->V1
  <data key="d2">Random</data>-->V6
  <data key="d6">Field</data>-->V7
  <data key="d8">sc1Path</data>-->V3
</node>
```

Listing 7.4: GraphML file, version *n+1* - mapping to key-value pairs

Listing 7.3 shows that the *D1-V1 key-value* pair was modified to *D1-V8*, therefore it can be concluded that the node was edited. The three basic types of changes can be defined as follows:

Add: if node exists in graph *G'* but it is not present in graph *G*, it was added.
Edit: if node exists in both graph *G* and *G'* and any of its key-value pairs are modified, it was edited.
Delete: if node exists in graph *G* but it is not present in graph *G'*, it was deleted.

Implementation of GraphML File Differencing

To implement the solution in Java, both previous and current versions of the graph (pre and post modification) obtained from *GraphML* files are parsed to a nested *hashmap* data structure. Nodes of the graph are identified by keys, while node property names and their values constitute values and are also stored as a *hashmap*, as illustrated by Figure 7.2. Change identification can also be realised using a constraint modelling system, such as Conjure [189], which allows the problem to be solved effectively. This option however, was dismissed for the current implementation, since it requires a constraint solver to be integrated in the framework resulting in further complexity. Furthermore, in case any changes are introduced to the underlying algorithm, utilising Java collections for implementing both the change and node identification problems provides a more flexible way of incorporating those changes.

Listing 2 describes the algorithm for identifying identical entities and fine-grained change types. While iterating the nested *hashmap* representations of the previous and current versions of the graph, *beforeMap* and *afterMap* respectively, the inner *hashmaps* inside both are compared. Inner

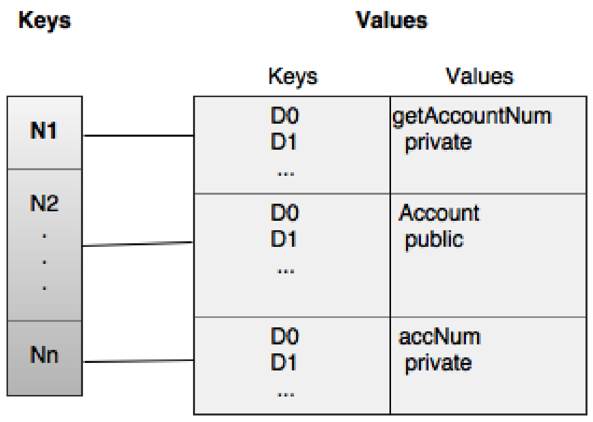


Figure 7.2: Nested *hashmap* representation of graph nodes and their properties.

hashmaps with the same number of keys are checked for matching values of *D0* keys. *D0* keys stand for the name property of graph nodes, and since nodes with the same name and number of properties are identical, to establish matching nodes, *hashmaps* with the same *D0* values are to be searched. In case values of all other keys are identical, the graph node is unchanged. If values of other keys mismatch, the node has been *edited*. Such keys are added to a list of edited entities. All matching keys are added to a list of matching entities. Should the values of *D0* keys be different, the inner *hashmaps* do not match, showing an *add* change. The same applies for inner *hashmaps* with different numbers of keys.

Deleted nodes are identified by differencing the sets of keys of *beforeMap* and matching keys. Added nodes can be obtained by differencing the sets of keys of *afterMap* and matching keys. *Edited* nodes can be established based on the list of edited entities.

A challenge revealed during implementation is differentiating *rename* changes from additions. Rename operations involve the modification of the *D0* property. However, at the same time, any other property may be modified. When comparing a node from the previous version with a node in the subsequent version, it cannot be stated with certainty whether the investigated nodes are the same in case the actual change was a *rename*. This is due to the fact that any two nodes can have the same number and type of properties, and their values are also subject to changes. For this reason, for any node where a match was not found, the node in the new graph is labelled as an *addition*.

Some modifications affect *intra* trace links, which are represented by edges in the *GraphML* files. To update *intra* links, they are re-generated following the identification of changed entities and change types. The approach for updating *inter* trace links is discussed separately in Section 7.3.

Algorithm 2 Change identification.

```
1:  Input:
2:      Nested hashmap representation of before and after graphs:
3:          a, outer hashmaps beforeMap and afterMap
4:          b, inner hashmaps beforeValue and afterValue
5:      Artefact change type: were the artefacts added, deleted or edited?
6:  Output: List of edited, added, deleted nodes
7:  begin
8:      For each key-value mapping in beforeValue and afterValue
9:          If beforeValue.keySet size = afterValue.keySet size
10:             If values corresponding to the D0 key are equal in beforeValue and afterValue
11:             beforeValue and afterValue represent the same graph entity
12:                 If beforeValue and afterValue are equal, the entity did not change.
13:                 Add beforeValue key and afterValue value to map of visited key-value pairs.
14:                 Else
15:                     The entity was edited. Add beforeValue key and afterValue value to
16:                     map of visited key-value pairs. Add beforeValue key and
17:                     afterValue to map of edited key-value pairs.
18:             Else beforeValue and afterValue are not the same graph entity. No action.
19:         Else
20:         beforeValue and afterValue are not the same graph entity. No action.
21:     End for
22:     Get added entities by differencing the keys of afterMap and visited key-value pairs.
23:     Get removed entities by differencing the keys of beforeMap and visited key-value pairs.
24:     Get edited entities from edited map keyset.
25: End
```

Element Identification Problem

A pivotal aspect of change detection is the identification of artefact elements across two subsequent versions of the given artefact. That is, how can it be established that *artefact element 1* in *version n* is identical to *artefact element 1* in *version n+1* regardless of the element being edited or remaining the same across the two versions. This issue exists regardless of the models selected to represent artefact elements and changes, and can be translated to the graph-based representation as follows. What methodology can be adopted to establish that *N1* on graph *G* is identical to *N1* on graph *G'*. The answer is straight forward in case the node was not modified since the key-value pairs are identical. Using the above example, it can be concluded that *N2* on graph *G* is the same as *N2* on graph *G'*. However, in case the node was edited, in the current implementation, domain-specific knowledge based on the artefact type is utilised. For example in a Java class, there cannot be two methods with the same name and signature.

Therefore, when comparing two nodes in two versions of a graph describing a Java source code artefact, if the two nodes have the same number of data elements, and their name is the same (*D1* data key), it can be concluded that they are identical even if other data key values have been modified. This rule can be applied in case of UML class diagrams and JUnit test cases. Further rules can be associated with other artefact types. For example, if node *N* stands for a requirement in a requirement specification artefact, it can be concluded that node *N* is the same across two versions of the file if their title properties (*D11*) are the same.

One specific case of this problem is the *rename* operation, which is also a type of *edit* change and it may be interpreted either as a *rename* or as a composite change consisting of a *delete* and an *add* operation. Another case is specific to Java source code, JUnit test and UML class diagram artefacts, where it is possible that multiple artefact elements with the same name and type exist. For example, there may be multiple Java methods with the same name and different parameters. Furthermore, for example, a Java class that contains multiple constructors poses the same challenge in identifying if elements are the same across two subsequent versions of the artefact.

7.2.4 Change Detection Output: Change Data object

The output of change detection is represented by the *ChangeData* object, which provides the following attributes of a modification and the modified artefact:

- **Change type.** As mentioned above, changes can be grouped as *add*, *delete*, or *edit* changes.
- **Name of the modified entity.** The name of the artefact element that was changed.
- **Specific artefact type.** It can take the values of artefacts handled by the framework, such as Java source code, UML class diagrams, etc.
- **Artefact element type.** Element types may take the value of *method*, *class*, *interface*, etc.
- **The scope of the change.** It shows whether the change is limited to the signature of the given artefact element or if it affects its contents or both.
- **UniqueId.** The unique identifier of an element.
- **Edits.** Property-level details of *artefact element level edit* changes.
- **Adds.** Property-level details of *artefact element level add* changes.

An example of the *ChangeData* object in Java and the granularity level it offers are shown by *change_1* in Listing 7.5. The example shows that the contents of the *MazeView* Java method were *edited*, while its signature remained the same.

```
Public ChangeData change_1 =
new ChangeData ("MazeView", ElementType.Method, false, ArtefactType.
    JAVA_SOURCE_CODE, ChangeType.EDIT);
```

Listing 7.5: Example *ChangeData* object

7.2.5 Conclusions

The main contribution of the change detection stage is the identification of changes of heterogeneous artefacts utilising a graph-based approach. The solution captures changes at the property graph level through the *ChangeData* object. While a graph-based representation of changes is required by subsequent stages of consistency management, it may also be useful in a number of other scenarios such as refactoring. To provide useful insights for refactoring tasks, the *GraphML* representation can be extended to capture finer-grained details of a method's body. The correctness of the change detection solution is evaluated through concrete examples obtained from open source systems, which is detailed in Chapter 9.

7.3 Rule-based Traceability Maintenance

The following section provides an analysis of the various change scenarios and their potential effects on *inter* trace links through which the traceability maintenance approach adopted in this work is revealed. Three main change scenarios can be established based on the *file level* change type. Thus, this section is divided into three corresponding subsections. Figure 7.3 presents a property graph denoting Java source code (SC1, SC2, SC3) and UML class diagram (DI1, DI2, DI3) artefact elements, which will be used to demonstrate the different change scenarios in each corresponding subsection. The figure depicts a Java class (SC1) with its members (SC2 and SC3), a field and a method, respectively. *SC1* is connected to *DI1* through an inter trace link, which models a UML class, and *DI2* and *DI3* denote its members.

7.3.1 Delete File Level Change

As described in Subsection 7.2.1, the *delete file level* change equates to removing files representing artefacts in the repository. Since Java source code and JUnit test artefacts are stored in multiple *.graphml* files, deleting any of the corresponding artefact files results in partially removing these artefacts.

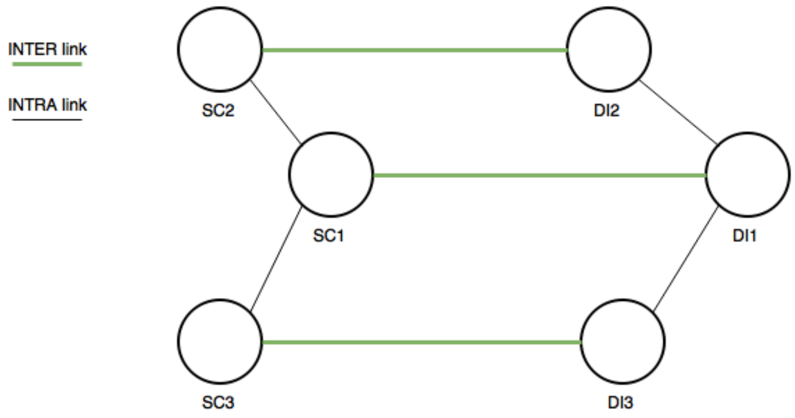


Figure 7.3: Graph data representing Java source code and UML class diagram nodes.

The effects of *delete file level* changes and the derivation of inter trace link maintenance rules are demonstrated using a Java source code artefact as depicted by Figure 7.4. In this scenario a *.java* file is deleted from the code repository. The change results in removing nodes *SC1*, *SC2* and *SC3*, marked in red, from the graph database. Since the deleted nodes are required for change impact analysis and consistency checking, they are first labelled in the graph database with a *to be removed* annotation. The update of the database is performed during the *Change Propagation* stage. Based on this scenario, the following rule can be formulated: if a Java source code artefact element type is removed, all of its relationships, including both intra and inter trace links, should be deleted. After considering other artefact types, this rule can be extended to all artefacts currently handled by the framework. Therefore the derived *delete trace maintenance rule* is as follows:

If an artefact element of any type is removed, all of its relationships, including intra and inter trace links, should be deleted.

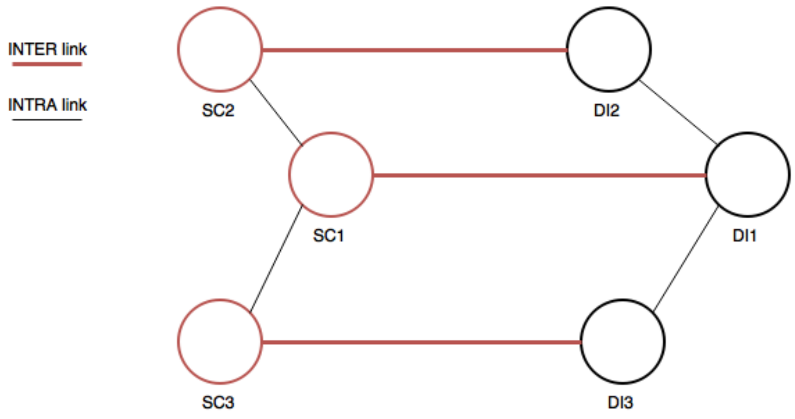


Figure 7.4: Delete file level change.

A summary table of the delete file level change scenario for each artefact type is provided in Appendix B in Table B.16.

7.3.2 Add File Level Change

It is the inverse operation of the *delete file level* change, which results in adding new artefacts to the repository. In case of Java source code and JUnit test artefacts, in most cases fragments of the artefact are added by creating new *.java* files. Further examples include adding a new UML class diagram, sequence diagram or conceptual architecture diagram. In most cases, *inter* trace link maintenance cannot take place as rules are not sufficient enough to infer new relationships. However, inter trace link creation techniques can be used to automate the process, as described in Chapter 8.

However, in the specific case of Java source code artefacts and when creating a new Java class or interface, inheritance information, that is, an existing *intra* link between the newly added class and an existing class or interface, may be used to infer potential *inter* links. Specifically, in case the *added* class extends an existing class or implements an interface:

- it should be connected to the requirement, architectural component, module or use case artefact element that class / interface is connected to through an *inter* link,
- it may potentially be connected to the UML class, JUnit class, UML sequence diagram that class / interface is connected to through an inter link.

7.3.3 Edit File Level Change

Edit file level changes describe situations when existing artefacts are modified either by *deleting*, *editing* or *adding* new elements. Therefore, the edit scenario is further grouped into three *artefact element level* change types. *Edit file level* changes and the formulation of trace maintenance rules are demonstrated through the same example introduced above using a Java source code artefact.

7.3.3.1 Delete Artefact Element Level Change

Figure 7.5 illustrates deleting the *SC3* node, which is a member element of the Java class represented by *SC1*. The example highlights that in this case the *delete trace maintenance rule* is applicable, which can be generalised to the other artefacts handled by the framework. A summary of the change scenarios specific to each artefact is provided in Appendix B in Table B.17.

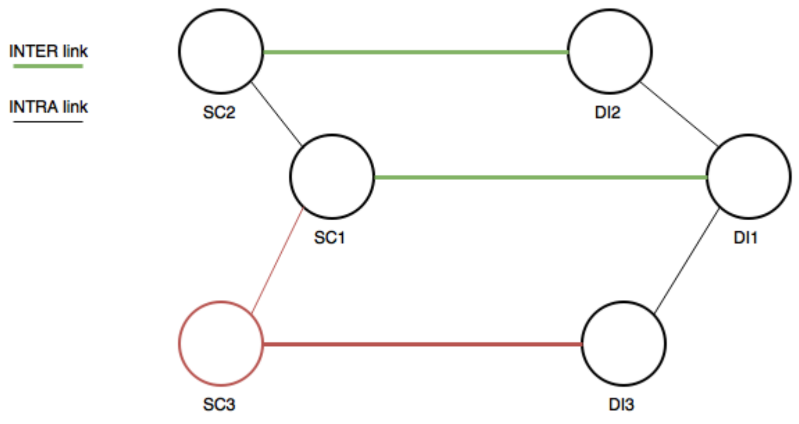


Figure 7.5: Delete element from artefact.

7.3.3.2 Add Artefact Element Level Change

Adding a new member element is shown on Figure 7.6. The addition of *SC4* results in an *intra* trace link being established between *SC1* and *SC4*. However, any potential *inter* trace links cannot be simply inferred by trace maintenance rules, and fall under the *inter* trace link creation problem. This scenario overlaps with consistency checking in certain cases, as based on the findings of the consistency checking stage, the addition of *inter* trace links may be suggested. For example, if consistency checking identifies an inconsistency by adding *SC4* based on a rule, it may suggest the addition of a *DI4* node, which should be connected to *SC4*.

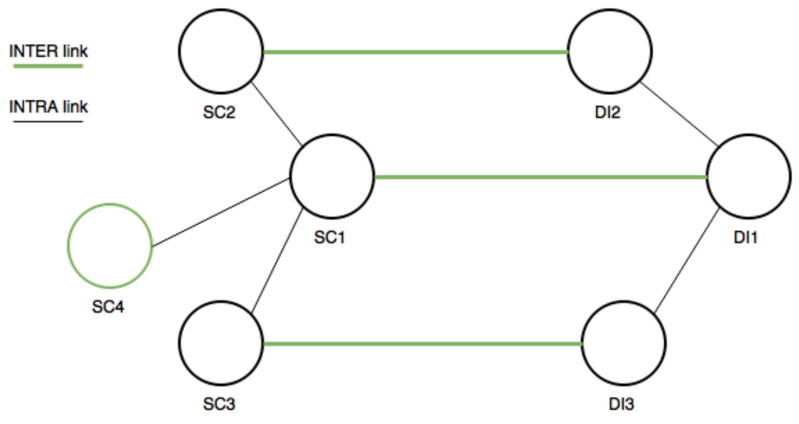


Figure 7.6: Add new element to artefact.

7.3.3.3 Edit Artefact Element Level Change

The edit scenario is depicted on Figure 7.7, which shows that the visibility property value of the *SC3* node was changed to *public*. The *SC3* node is a member of *SC4*, which is a container node. The edit scenario in general can represent editing both *container* and *member* elements, which is applicable to Java source code, UML class diagram and JUnit test artefacts. In the example, the *inter* trace link connecting *SC3* and *DI3* remains unmodified. Further edit change scenarios may include renaming *SC3*, where the entity may still remain connected to *DI3*. However, in ambiguous situations the identity of the node may change as it now may represent a new artefact element. Additionally, the contents of some artefact elements may also be subject to updates, which may or may not affect their *inter* and *intra* links. Thus, the framework takes an optimistic approach to trace link maintenance for edit artefact element level changes, which is based on the granularity of artefact elements. The following *edit trace maintenance* rule is specified:

Following an edit artefact element level change, existing intra and inter trace links are not affected.

Table B.18 provides a summary of *edit artefact element level* change scenarios for each artefact.

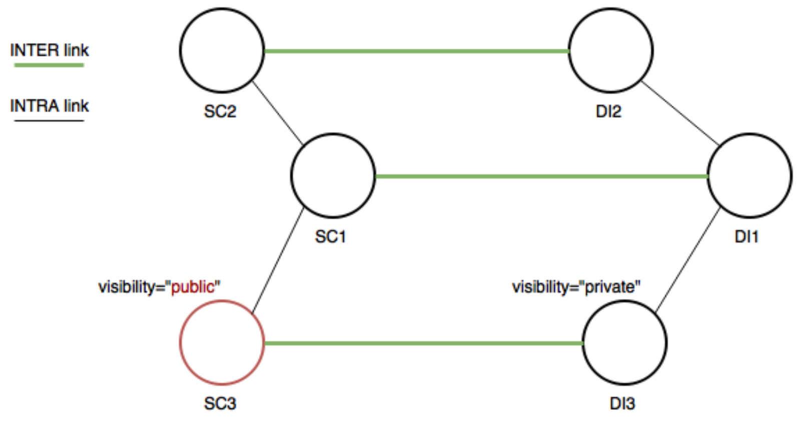


Figure 7.7: Edit property of an existing element.

7.4 Change Impact Analysis

The impact analysis algorithm, which is introduced in Chapter 4, is implemented using the *Neo4j Traverser Framework*[10]. The main components for realising the approach are the following:

- A start node, which is supplied by the *ChangeData* object
- Edges (relationship types and directions) to traverse, which is specified by the algorithm
- A stop criterion to finish traversing (stop evaluator)
- A selection criterion to establish the set of nodes to return (returnable evaluator)

[10]http://neo4j.com/docs/stable/tutorial-traversal-java-api.html

7.4.1 Illustrative Example

The following example shown in Figure 7.8 highlights how the impact analysis algorithm is realised using the *Traverser Framework*.

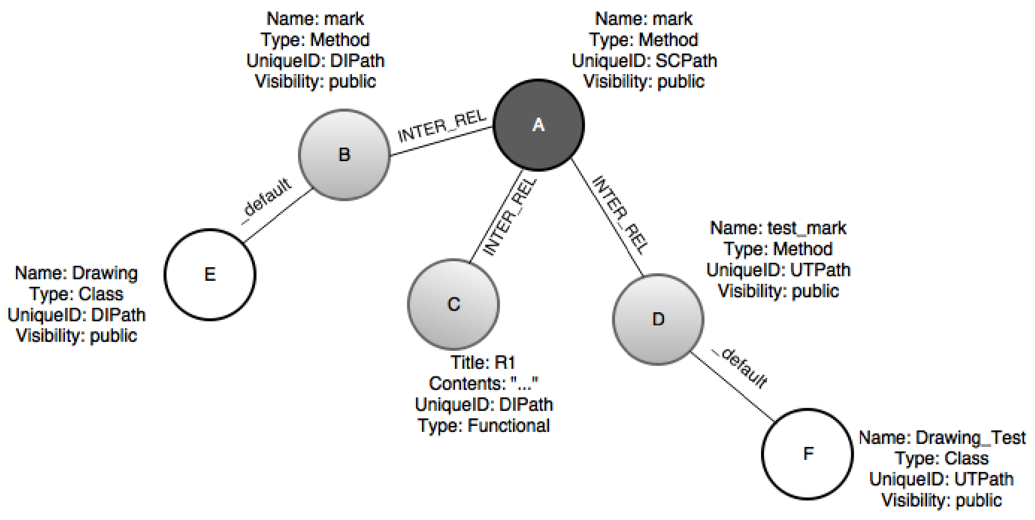


Figure 7.8: Example artefact property graph to illustrate the change impact analysis approach.

Inputs

- Property graph of artefact elements: the graph consists of nodes *A, B, C, D, E* and *F*. Node *A*, which denotes a Java method called *mark*, is directly connected to *B, C* and *D*, through an *inter* link. Node *B* is a UML method called *mark*, node *C* stands for a requirement in a specification document expressing the functionality of the *mark* method. Finally, node *D* is a unit test for the *mark* Java method. Node *D* is also connected to *F* through an *intra* link, while node *B* has another connection, namely node *E.*
- The *ChangeData* object, which specifies the following:
  - *The name of element:* mark
  - *The type of element:* method
  - *The fine-grained change type:* false - it is not a signature change
  - *The artefact type:* source code
  - *The artefact element level change type:* edit
  - *The file level change type:* edit
  - *The specifics of the edit artefact level change:* previous and current contents
  - *The uniqueId:* SCPath

These details suggest that the contents of the *mark* method were edited. The aim of impact analysis using this concrete example is to establish which nodes may be impacted by the same modification.

Processing

Based on the *ChangeData* object, the modified node (*A*) is identified and is set as the starting point of the traversal. The *Neo4j Traverser* object is responsible for performing and customising the traversal. For example, it defines the depth and type (breadth or depth first) of the traversal. The API also makes it possible to define rules for the traversals by specifying them in the overridden *Evaluator* object of the *Traverser Framework*.

Output

At the end of the process, the *impact_set{B, C, D}* consisting of node *B, C* and *D* is returned. A string representation of the output may take the following form:

```
(A)--[INTER_REL]-->(B)
(A)--[INTER_REL]-->(C)
(A)--[INTER_REL]-->(D)
```

Listing 7.6: String representation of impact analysis output.

The output is represented by an *IAResult* object in the framework, which encapsulates the changed entity, the specifics of the change, and the results of inter and intra traversals in the form of lists of *Node* and *Path* objects.

7.5 Rule-based Consistency Checking

The realisation of consistency checking is detailed in the following section. Firstly, *inter* consistency checking is introduced, then the particulars of *intra* consistency checking are presented. As described in Chapter 4, the basic premise of the consistency checking approach is that following change detection and impact analysis, the consistency of potentially impacted entities can be analysed by applying consistency rules. The components of consistency rules, which are identified in Chaper 4, are obtained at implementation level in the following manner:

- *File level* change type (add, edit, delete) obtained from the *getFileLevelChangeType* property of the *ChangeData* object
- Artefact type obtained from the *getArtefactType* property of the *ChangeData* object. It takes

the values of types currently handled by the framework.

- Artefact element type derived from the *type* property of the artefact element in question.
- Fine-grained artefact element type denoting hierarchical relationships between elements of a specific artefact, which can take the value of *container* or *member* element and is derived from the artefact element type. This categorisation is applicable to Java source code, UML class diagram and JUnit test artefacts due to their structural similarities.
- Existing *inter* and *intra* trace links connecting elements derived from the property values of edges connecting the given elements.

7.5.1 Inter Consistency Checking

In the following section the consistency management approach between heterogeneous artefacts is discussed. To derive inter consistency rules, three basic scenarios are considered based on the *file level* change type. Therefore this section is divided into three subsections, which represent the three categories of *file level* changes. In each subsection a number of scenarios, based on the artefact type, are introduced.

Delete File Level Change

A *delete file level* change results in removing multiple artefact elements at the same time, thus at the graph database level multiple nodes are affected.

Tables B.8, B.9 and B.10 in Appendix B show the derivation of consistency rules and that the rules are dependent on the specifics of the changed entity, the elements it is connected to, and the type of the *inter* trace link between them. In the current discussion, *identity* and *satisfaction* links are relevant. An *identity inter* trace link between two entities shows that they represent the same concept. On the other hand, a *satisfaction* type *inter* link does not guarantee a one-to-one mapping between the two entities. Therefore, in case a Java class is deleted, a requirement may remain consistent. However, the same rule may not be applicable if the changed entity is a requirement, and the connected entity is a Java class.

Add File Level Change

The *add file level* change is the inverse operation of the *delete file level* change and the same example scenarios are applicable. In case new elements are added, no connections are in place between these and existing elements. Since consistency checking requires the existence of trace links, rules cannot be defined to cater for this scenario. However, in particular cases, which are specific to Java source code, UML class diagram and JUnit test artefacts, *intra* connections may be utilised. For example a newly added class may implement an existing interface, which results in the creation of an *intra* link between them.

Edit File Level Change

Edit file level changes result in modifications within existing artefacts and are summarised in Table B.7 in Appendix B. Since further three *edit file level* change scenarios exist - *add, delete and edit artefact element level* changes -, consistency rules are derived accordingly. Firstly, *delete artefact element level* changes are introduced in Table B.11 in Appendix B.

Establishing *inter* rules for the *add new artefact element* change scenario in Java, JUnit and UML class diagram artefacts requires the *inter* trace links of the parent of the newly added artefact element. For example, when a new UML method is added to a UML interface, the elements connected to the UML interface through an *inter* link may be inconsistent. As shown in Table B.12 in Appendix B, deriving rules in such a manner is applicable to artefacts which provide both member and container elements. Furthermore, in the specific case of adding a UML container element, existing intra connections do not provide sufficient information to create *inter* consistency rules.

Edit artefact element level changes are summarised in Tables B.13 and B.14 in Appendix B. The types of edited elements show that specific artefacts can be edited in a number of ways. For example Java member artefact elements can be changed by modifying their *signature*, such as renaming a method, or by editing the *contents*, such as changing a method's body.

7.5.2 Intra Consistency Checking

Intra rules are applicable in case of *edit artefact element level* changes since in case of *add artefact element level* and *delete artefact element level* changes, *intra* links are added and deleted, respectively. Table B.15 in Appendix B shows the intra rules applicable to Java source code, UML class diagram and JUnit test artefacts. Since *intra* trace links between requirement elements in a requirement specification, use case elements in a use case document, and lifeline elements in a sequence diagram artefact are not recorded in the current implementation of the framework, no *intra* rules are applicable to these artefacts. Furthermore, despite the fact that conceptual architecture and module view architecture artefacts are characterised by a structure in which certain elements may contain others, such as components and subcomponents, the framework presently handles these artefact elements individually.

7.5.3 Rule Implementation

Consistency checking rules are captured in a rule-base in an XML format and are parsed using the Java DOM API. XML was selected as a means of representing the different scenarios since it allows new rules to be added should the framework be extended with additional artefacts. Such flexibility provides further benefits from an implementation point of view, as adding further rules does not require the implementation to be altered. Listing 7.7 shows an excerpt of the rule-base, which represents a *delete file level* change. In this example a Java source code container artefact is removed, and the corresponding rule indicates that the connected UML sequence diagram member element is inconsistent.

```
<DeleteFileLevelChangeRule id="rule3">
      <ChangedEntity>
        <ArtefactType>JAVA_SOURCE_CODE</ArtefactType>
        <FineGrainedElementType>Container</FineGrainedElementType>
      </ChangedEntity>
      <ConnectedEntity>
        <ArtefactType>SEQUENCE_DIAGRAM</ArtefactType>
        <FineGrainedElementType>Member</FineGrainedElementType>
      </ConnectedEntity>
      <InterLinkType>Identity</InterLinkType>
      <StateOfConsistency>Connected entity is inconsistent.</
          StateOfConsistency>
    </DeleteFileLevelChangeRule>
```

Listing 7.7: Excerpt of XML rules capturing consistency rules.

7.5.4 Output

The results of consistency checking are captured in a *ConsistencyCheckResult* class, which associates a pair of changed artefact element with a potentially impacted element and an applicable consistency rule.

7.6 Change Propagation

In the current implementation of the ACM framework, change propagation is responsible for suggesting resolutions to users based on the results of consistency checking. Additionally, it is the last stage of the consistency management process, and therefore the final graph database update takes place in this stage prior to pulling new changes from the repository. Finally, users are presented with the results of consistency management, including a summary of the changes, their potential impact and consistency issues.

Similarly to consistency checking, change propagation can also be divided into two broad categories. Intra change propagation follows intra consistency checking when inconsistencies are resolved within an artefact. On the other hand, inter change propagation refers to the activity of resolving inconsistencies in heterogeneous artefacts.

7.6.1 Graph Database Update

The majority of database update operations take place in the *Change detection* stage as described in Subsection 6.3.1.5. However, *inter* trace link maintenance, that is adding, removing and editing edges of the graph, is performed during change propagation. Moreover, nodes that have been previously labelled with the *tobeRemoved* annotation are removed from the database.

7.6.2 Inconsistency Resolution

The *ConsistencyCheckResult* object provides sufficient data to suggest resolutions to users. It contains details about the changed artefact element, and the database nodes that may be inconsistent as a result. Based on this input, the framework differentiates two cases:

a, (Potential) inconsistencies

In this case, the framework suggests the user to apply the same change to inconsistent and potentially inconsistent elements that was previously performed on the changed artefact element.

b, No consistency issues

In this case a message is displayed to the user stating that the framework did not identify consistency issues following the specific change. However, the elements of the impact set are listed and the user can confirm these results by checking these elements either at the database or at the original artefact level.

7.6.3 Final Output of Consistency Management

The final output of the consistency management cycle is presented to the user at the end of change propagation. The output consists of a summary of the changes and (potential) inconsistencies. An ideal change propagation solution is automated as far as possible and therefore involves applying changes to inconsistent artefacts as automatically as possible. Additionally, the user should be able to configure which changes may be automatically applied and which modifications require manual input. This level of change propagation support is not within the scope of this work.

7.7 Implementation Evaluation and Conclusions

To conclude Chapter 6 and Chapter 7, an evaluation of framework implementation is provided. The assessment is carried out by considering system prerequisites, the functionality offered by the current implementation and by analysing the level of support provided for high-level requirements set out.

The current implementation of the ACM framework enables users to perform *framework setup* to unify heterogeneous artefacts to a property graph representation, and *consistency management*. In order to invoke the functionality of the framework certain prerequisites are required. These assumptions and the system requirements for running the framework can be summarised as follows.

- **Version Control.** The current implementation assumes that the original artefacts are stored in a version control system. This makes it possible for the framework to pull artefacts from the repository and perform operations on them.
- **Database backend.** It is assumed that the user has a graph database installed. The current implementation supports *Neo4j* and can be extended to allow migrating to other graph databases.
- **User supplied data.** In order to extend the framework with new artefacts users are required to supply XSLT files to allow the transformation of their custom XML-based representation to the custom *GraphML* structure.
- **Tools.** The tools used to create and store original artefacts are required to provide functionality to export artefact data to an XML-based representation.
- **System requirements.** The framework currently runs on a *Windows platform.* Further prerequisites include a version control system of the user's choice (in the current implementation Mercurial is required) and the Java platform[11].

Following is a summary of the framework functionality and an analysis of how the current implementation achieves the high level requirements set out in Chapter 4.

- **Tool independence.** The framework caters for any tool assuming that it allows the extraction of artefact data to an XML-based format. For the present version of the implementation the following heterogeneous tools and formats are selected: OpenOffice Write *(.odt)*, DIA *(.dia)* and Eclipse *(.java).* Additionally, the framework can be extended to cater for artefacts created in further tools.

[11]http://www.oracle.com/technetwork/java/javase/overview/index.html

- **Artefact independence.** The property graph model allows the representation of any entity characterised by any structural attributes and abstraction level assuming artefact data can be extracted as mentioned above. Apart from the ability to handle any artefact regardless of its type, another important aspect to consider is that the framework should prioritise artefacts which are the most widely used in software development projects. Representations selected for the current implementation include requirement specifications written in natural language, UML class diagrams, Java source code, JUnit test cases, UML use cases, UML sequence diagrams, software architectures (conceptual and module view), subsets of which are used in traditional and agile software projects.
- **Automation.** A major goal of the ACM framework is the discovery of approaches that allow automation across all stages of the consistency management process. The framework makes it possible to automatically extract and transform heterogeneous artefacts to a uniform format assuming data in the original tools can be accessed. The creation of *inter* trace links is currently semi-automated and the approach is presented in the following chapter. Change detection is carried out in an automatic manner and is currently invoked manually by the user. Change impact analysis and consistency checking are automatic. Change propagation automatically suggests resolutions to inconsistencies, however, the propagation of changes is carried out manually by user at their discretion.
- **Configuration.** The ability to configure the framework during the setup process addresses the "Customisable and non-intrusive" requirement. Users can currently perform framework configuration at startup. However, future work remains to be done in the area to allow users to customise other aspects of the framework, which is discussed in Chapter 10.
- **Performance.** The ability to effectively handle a varying number of artefacts and changes of different complexity is evaluated in Chapter 9.

CHAPTER EIGHT

AUTOMATING TRACEABILITY CREATION USING MACHINE LEARNING

An integral aspect of the ACM framework is *Traceability creation*, which lays the foundations for subsequent stages. Therefore, its automation plays a pivotal role in providing an effective solution. This chapter introduces an approach based on machine learning to automate trace link creation, which is identified as a classification problem. It then discusses data collection, and feature and model selection. Finally, the trained models are evaluated, and an assessment of the approach and the strategy used to integrate it in the framework are provided.

8.1 Introduction

Traceability creation aims to establish inter trace links between software artefacts. Since the stages of consistency management, which are discussed in Chapter 4, rely on the existence of correct and complete trace links, a mechanism for creating them is intrinsic to the ACM framework. In accordance with the high level requirements of the framework, any traceability approach should be independent of artefacts and tools, and be as automatic as possible. Automatic link creation is also central to the adoption of the framework in real world scenarios, where establishing links manually in a potentially large number of artefacts may not be feasible.

Automating trace link creation is a well-established research problem and various techniques have been proposed to develop more intelligent algorithms to automatically identify links or to complement and improve the accuracy of existing solutions. As described in Chapter 3, these can be categorised in different ways including information retrieval [70], heuristic [100], data mining [109], ontology [190], and rule-based [112] techniques. Despite the number of approaches, providing a solution to accurately and automatically establish trace links among a set of heterogeneous representations remains an open problem. The aim of the approach discussed in this chapter is to provide a machine learning based semi-automated solution to create inter trace links and to cater for diverse artefacts. Prior to discussing the specifics of the approach, basic concepts of machine learning are introduced.

8.2 Machine Learning

8.2.1 Basic Concepts

Mitchell defines machine learning as a field concerned with the construction of "computer programs that automatically improve with experience" [191]. Machine learning allows the discovery of knowledge from data by devising algorithms that draw inspiration from a number of fields. Such areas include artificial intelligence, probability and statistics, computational complexity, information theory, psychology and neurobiology, control theory, and philosophy [191]. The impact of these fields is manifested in the core ideas behind machine learning algorithms and models. For example *Neural Networks* are modelled based on the biological brain, and *Bayesian Networks* learning is based on principles originating in probability and statistics.

Machine learning algorithms have proven to be useful in a substantial number of application domains. One example is data mining problems where the aim is to discover implicit correlations and novel patterns in large-scale data [192]. Other areas include speech recognition, computer vision, and robot control.

Machine learning problems can be categorised into various groups, such as *classification*, *regression* or *clustering* problems. The aim of both classification and regression is to predict a target (output) based on some predictors (inputs) [193]. However, the two differ in the type of the target. While the target in classification is a nominal variable, in regression it is numeric. Classification, under which the approach presented here falls, is introduced in detail in Subsection 8.2.4. The main premise of clustering is to assign observations into groups based on some similarity. A notable clustering method is the *K-means algorithm*, which is aimed at finding user-specified number of clusters represented by their centroids [192].

Depending on the learning approach, four main types of machine learning scenarios can be differentiated [194]. *Supervised learning* involves the use of labelled instances, that is, the algorithm is provided with a training set that contains the desired output values. On the other hand, in *unsupervised learning* the training data does not contain the desired outputs, whereas *semi-supervised learning* may involve a few desired outputs. Finally, in *reinforcement learning* the algorithm learns through trial and error. The work presented here falls under the area of supervised learning.

8.2.2 Relevant Machine Learning Usage Scenarios

Machine learning has been applied in a number of software development and software maintenance problems; as Zhang points out, requirements engineering, rapid prototyping, component reuse, cost/effort prediction, defect prediction, test oracle generation, validation, reverse engineering and change impact prediction are just a few areas that can benefit from the potential machine learning techniques offer [195] [196]. However, due to the data requirements of such techniques, one of the hindering factors of applying machine learning algorithms is the availability and accessibility of relevant software engineering specific data from software projects [197].

In the field of traceability, a number of solutions rely on machine learning techniques to complement other automated trace generation techniques and to improve their results. A few examples include the Multi-strategy Learning approach to recover trace links between Java programs and Use Case elements [198], and a custom classification algorithm to improve the quality of traces between regulatory code and product level requirements [199]. Additionally, work has been done to evaluate the applicability and performance of clustering in automated tracing [200], to combine the Vector Space Model with Regular Expressions, Key Phrases and Clustering using a modified K-means algorithm to automatically recover links between text documents and source code [201], and to investigate the use of clustering to improve tracing between high-level requirements and low-level design elements [202]. Finally, reinforcement learning has been used to identify common textual segments between documents and to suggest links between them [203]. These solutions focus on specific artefacts and on automating tracing between these representations. In comparison, the approach presented in this work applies supervised learning to establish trace links between heterogeneous artefacts and hereby aims at providing a more generic solution applicable in different development scenarios.

8.2.3 Motivation to Use Machine Learning

Automatically creating trace links is a complex problem. Inter artefact relationships cannot simply be inferred from a set of rules describing correlations between artefact elements without imposing very restrictive practices on developers, such as strict naming conventions, or manually creating mappings between artefacts. The heterogeneity of artefacts and artefact elements, which differ in their naming, structure and abstraction levels, exacerbate this complexity. It cannot be guaranteed that software projects follow standardised coding practices such as naming conventions, which means all aspects of artefacts can be variable. The complexity of this problem makes a simple heuristic approach unlikely to succeed. Thus, a way of capturing and leveraging the fundamental complexity of the interactions between artefacts is required and machine learning is particularly suited to modelling complex non-linear spaces.

8.2.4 Traceability Creation as a Classification Problem

The premise of our approach is that establishing trace links can be thought of as a binary classification problem. That is, a pair of source and target artefact elements can be categorised into a given set of categories, related or unrelated, based on existing and already categorised pairs. As described by Domingos [204], a classification is a system which, given a vector of feature values, outputs a single discrete value called the *class*. The problem, specifically in the context of classification, can be defined as approximating a boolean-valued function from training examples, i.e. given examples labelled as members and non-members of a class. Each instance X - a pair of source and target artefact elements - is represented by attributes (selected features, which are discussed in Section 8.5). The target concept - whether or not a trace link exists for X - can be denoted by:

$$c : X \rightarrow 0, 1 where c(X) = 1 \tag{8.1}$$

if there is a link between source and target, and

$$c : X \rightarrow 0, 1 where c(X) = 0 \tag{8.2}$$

if there is no link between source and target.

The learner is presented with negative (c(X) = 0) and positive (c(X) = 1) examples and the aim of the classification is to find an estimation (h) such that h(X) = c(X). The outcome of the learning process is successful if following the approximation of the target function over training examples, the approximation on unobserved examples yields sufficiently accurate results [191]. In the next few sections, the methodology for data collection, preparation and feature selection is outlined, followed by model selection, training and an evaluation of results.

8.3 Data Collection

A prerequisite of successfully implementing machine learning algorithms in the ACM framework is the availability and accessibility of relevant software engineering data for training and evaluation. This is often a challenge due to the nature of software projects; data from proprietary products is typically unavailable [197] and artefacts other than source code are not always available or complete for open source systems. The experiments presented in this work utilise data from six different open source systems hosted in online repositories.

8.3.1 Criteria for Candidate System Selection

The primary criteria for selecting candidates include:

Artefacts available in repository. The main criterion was the availability of a variety of artefacts to represent different combinations of traceability scenarios. It was also considered which artefacts are most widely used in projects. According to a survey, the most widely used non source code artefact is the UML class diagram, followed by sequence diagrams and use cases [205]. Examining a number of online repositories reveals similar patterns.

Implementation language: Java. The framework currently handles Java source code, and therefore the search was limited to systems implemented in this language.

System size. Systems of varying sizes were selected for experiments. Smaller systems are easier to comprehend and allow the establishment of trace links across the entire system instead of having to focus on individual components to manage complexity. Conversely, larger systems may offer more complicated links of different types between artefacts.

The criterion of the availability of various artefact types proved to be a challenge since only a small proportion of systems provide documentation, such as requirement or architecture specifications. To maximise the chance of finding candidate systems, an extensive search took place on popular source code repositories using a list of available hosts [206]. Out of the listed repositories, candidate systems were found on GitHub[1], SourceForge[2] and Google Code[3]. A further challenge is the non-uniform metrics these hosts provide for comparing project size. GitHub, for example, does not disclose lines of code metrics; therefore, where such information

[1]https://github.com/

[2]https://sourceforge.net/

[3]https://code.google.com/

is not available, the metric was calculated. Following is a brief summary of the candidate systems.

8.3.2 Candidate Systems

Table 8.1 provides a summary of the functionality, origin and size metrics of the selected systems.

Micro Mouse Simulator (MMS)[4] is a micro-mouse maze editor and simulator that leverages various maze solving algorithms. It has been implemented using Java and Python. MMS provides Java source code and UML class diagram type artefacts.

JGAP[5] is a Java framework that can be used as a means to solve problems applying evolutionary principles. JGAP offers extensive documentation and approximately 1400 test cases, which makes it a suitable candidate for extracting source code and unit test artefacts.

Neo4j[6], the popular graph database, was selected because of the size of its codebase and because it provides Java source code, unit test and module view architecture artefacts.

Myrobotlab[7] is a framework for robotics and creative machine control providing services for machine vision, speech recognition, servo control, GUI control and microcontroller communication. Since Myrobotlab offers extensive documentation in the form of architectural diagrams, as well as some test cases covering certain areas of its functionality, it provides data for setting up architecture-source code and unit test-source code links.

The Java Binary Block Parser (JBBP)[8] is a framework for parsing binary block data in Java supporting various data types. JBBP was selected due to the variety of artefacts it contains: most Java classes are covered by test cases and the system also allowed the extraction of a use case diagram providing another dimension to artefact data used in trace link establishment.

Finally, **Titan**[9] is an open source distributed graph database designed to support complex and real-time traversal queries on large graphs and concurrent transactions. The project provides test cases, Java source code, as well as a conceptual architecture artefact for extraction.

The various metrics provided by the repositories, such as lines of code, number of contributors and commits allow the comparison of the size of the candidate systems. It is concluded that MMS and JBBP represent one end of the spectrum characterised by a smaller size, JGAP and

[4]https://code.google.com/p/maze-solver/
[5]http://jgap.sourceforge.net
[6]https://github.com/neo4j
[7]https://github.com/MyRobotLab/myrobotlab
[8]https://github.com/raydac/java-binary-block-parser
[9]https://github.com/thinkaurelius/titan

System	Description	Source Repository	Lines of Code (LOC)	Number of Contributors / Commits
MazeSolver	Micro-mouse maze editor	Google Code	9223	4/139
JGAP	Java framework for Genetic Algorithms	SourceForge	57200	-
Neo4j	Graph database	GitHub	152139	118 / 34995
MyRobotLab	Java framework for robotics	GitHub	133247	11 / 665
Java Binary Parser	Java binary block data parser	GitHub	27677	1 / 194
Titan	Distributed graph database	GitHub	107792	32 / 4422

Table 8.1: Comparison of candidate systems.

	ARTEFACT TYPES						
SYSTEMS	**UML Use Case Diagram**	**Module View Architecture Diagram**	**Conceptual Architecture Diagram**	**UML Class Diagram**	**UML Sequence Diagram**	**Java Source Code**	**JUnit tests**
MazeSolver				X		X	
JGAP						X	X
Neo4j		X				X	X
Myrobotlab					X	X	X
Java Binary Block Parser	X					X	X
Titan			X			X	X

Table 8.2: Extracted artefacts.

MyRobotLab are larger systems, followed by Titan, while Neo4j is the largest of the candidate systems.

Table 8.2 shows the types of artefacts extracted from the systems. It can be seen that Java source code was available in all repositories and most repositories allowed the extraction of unit test artefacts, while every other artefact type was found only in single repositories.

8.4 Data Preparation

The aim of data preparation and feature selection is to establish a training set for supervised learning, which is characterised by the desired outputs being specified for each data instance. The data from available artefacts for each system was extracted using the extraction functionality of the ACM framework. The extracted data was transformed to the GraphML format, which provided input for generating the training data. Artefacts across different systems include Java source code, UML class diagram, UML use case diagram, logical and development architecture, UML sequence diagram, and JUnit test cases.

As described in Chapter 6, in a GraphML representation each artefact element is denoted by a node element with nested data elements describing its properties.

8.4.1 Establishing Positive Instances - Trace Links

The *Traceability* component of the framework processes the values of the data elements in the GraphML files to produce numeric feature values, which are explained in Section 8.5. Following the extraction of artefact data, trace links were established manually between artefact elements to provide training examples for machine learning. As described in Chapter 6, a trace link is a pair of source and target elements identified by unique ids.

The process of trace link establishment varies between artefact combinations. However, one of the conditions holds across all systems: in order to establish all the correct links and arrive at an adequate trace link coverage, domain knowledge and experience with the given system is required. The smaller the system, the easier it is to comprehend its overall architecture and discover links. The number of links can be correlated not only to project size, it is also determined by the type of the source and target artefact. For example numerous connections can be established between unit tests and source code due to their proximity in abstraction levels. However, when connecting a high level conceptual architecture to source code, the number of connections is lower. Trace link coverage is also dependent on the level of detail extracted from the original representations. The framework allows the extraction of class level and member level entities in case of Java source code, JUnit test cases and UML class diagrams. For architectures, the framework considers components, while other architectural features are part of future work. For UML sequence and use case diagrams extracted elements include lifelines, messages, and use cases, respectively. In the remainder of this section, the methodology to establish links between heterogeneous artefacts is described. Links were created between representations that were available from the given system. Each system provides source code artefacts. On the other hand, other representations can be found in specific systems. Thus, trace links were established between source code and other available artefact types.

Java Source Code - JUnit Test Links

Firstly, trace links were created at class level between source code and test classes, or other relevant class level elements. It may be possible that the functionality of multiple classes is tested in a single test class, in which case all connections were recorded separately. Secondly, trace links were established at member level between source code and test class methods and fields. A single test case may connect to multiple Java methods depending on the complexity of the test case.

The strategy taken to establish trace links involved analysing *assert* statements, which mainly test a single unit, and are usually mapped to a single method. Besides assertions, test cases are likely to contain calls to various other methods. In such cases, as mentioned above, connections were established between the called methods and the calling test case. This may also include links between test cases and Java fields, since the modification of fields may also lead to changing the given test case. This case highlights another strategy used during tracing, which is based on analysing the impact of changes should any of the entities change.

Java Source Code - UML Class Diagram Links

In this case links were established at class level between source code and UML classes, or other relevant class level elements, and at member level between source code and UML methods and fields. Both scenarios denote which UML entities are mapped to which implementation classes. The tracing process involved the identification of mappings from UML class level entities to source code class level entities based on name similarity as a first step, followed by the discovery of links between member elements.

Java Source Code - UML Sequence Diagram Links

The level of detail captured in this scenario is dependent on the given sequence diagram. In the systems used for data extraction, available sequence diagrams contain higher-level use case entities instead of object or class entities. This allowed tracing to multiple source code classes, which embody the functionality of the use case.

Java Source Code - UML Use Case Diagram Links

Use cases were mapped to a number of Java classes depending on the functionality described in the use case, the specificity of the wording of the use case and the design of the system. The approach taken to establish links was based on understanding the functionality described in the use case and then searching for corresponding implementations at the class level. Tracing in this specific case did not involve member level entities.

Java Source Code - Software Architecture (Conceptual View) Links

Depending on the architecture of the given system, components may encompass larger or smaller areas of functionality, which affects the number of implementation entities they connect to. The strategy of setting up relationships was to identify the functionality offered by the given component either based on the documentation or the description provided on the diagram and to determine which source code entities implement this functionality. Trace links were established at class level.

Java Source Code - Software Architecture (Module View) Links

Such as in case of conceptual view architectures, modules and subsystems in a module view architecture are mapped to multiple class level implementation artefact elements. Given the systems available for data extraction, the name similarity between module names and the physical directories in which source code files are saved was used to establish trace links.

In summary, the tracing strategies and artefact combinations described above resulted in the following links:

- Test Class-Source Code Class
- Test Case-Source Code Method
- Test Case-Source Code Field
- Test Class Field-Source Code Field
- UML Class-Source Code Class
- UML Operation-Source Code Method
- UML Attribute-Source Code Field
- Sequence Diagram Use Case-Source Code Class
- Sequence Diagram Message-Source Code Method
- UML Use Case-Source Code Class
- Architecture Component-Source Code Class, and
- Module-Source Code Class

8.4.2 Establishing Negative Instances - Generating Data for Representing Non-Relations

Subsequent to establishing trace links, training data representing unrelated artefact elements was generated. Elements within the extracted artefacts were correlated with target elements that they do not form a trace link with. An example scenario is when a use case models functionality that is not implemented by the selected Java class. The result is training data that contains:

- Architecture module source elements and unrelated source code target elements
- Architecture component source elements and unrelated source code target elements
- UML Sequence diagram source elements and unrelated source code target elements
- Unit test source elements and unrelated source code target elements
- UML class diagram and unrelated source code target elements
- UML use case source elements and unrelated source code target elements

In summary, it can be concluded that the characteristics of the training data are determined by attributes of the systems the data is extracted from. Key factors include the size of the system, the available artefacts and their complexity.

8.5 Feature Selection

Features form one of the essential components of creating an accurate predictive model. A feature is the specification of an attribute of a data instance, which may either be represented by a categorical or a continuous variable [207]. Features were chosen based on application domain knowledge to capture generic attributes applicable across heterogeneous artefacts independent of their type. This includes features based on the names and types of artefact elements since all artefacts can be described by these. The selected features can also be extended to new artefacts should they be added to the model. Data extracted from the original representations is textual. Therefore, a major component of the feature engineering process is the conversion of textual data to numeric variables. Following is a description of how this was achieved for specific features.

Name Similarity

Since each artefact element has a name, source and target elements can be compared through a similarity measure. To compute similarity, various algorithms can be used. For the purpose of this work, the Levenshtein (edit) distance was selected [208]. The similarity score is expressed using a 0.0 to 1.0 scale, where 0.0 denotes a 0% match, while 1.0 stands for a 100% match.

Atomic vs. Container Elements

Artefact elements can either be composite or atomic. For example, in the current dataset a Java method element is atomic, while a UML interface element may contain fields and methods, thus it is composite. This feature is defined as follows: container elements are assigned the value *1*, atomic elements take the value *0*.

Enumerating Abstraction Levels

Artefacts can also be described by their abstraction levels. Requirement specifications represent high abstraction level artefacts, source code and unit tests are low abstraction level artefacts, while architectures and diagrams are in between the two. The abstraction level of both source and target elements are expressed in a numeric format, such that a high abstraction level is represented by 0, and a low abstraction level equals to 2. Artefacts that are in between the two abstraction levels, are assigned the value 1. This scale is applicable to any artefact including ones from a traditional software process or agile projects. The feature is represented by a numeric value based on the absolute value of the difference between the source and the target.

Artefact Type

Every artefact can be grouped into a type category. At present, the categories the framework caters for include: requirement, architecture, design, source code, test, documentation, API, and configuration files. These cover the types of artefacts currently identified and can be easily extended to include new categories.

The feature values are expressed as follows: the values 1 or 0 are assigned to every source and target artefact element depending on their type. To indicate that a given source artefact is of architecture type, the following feature values are specified: *IsSourceArchitecture* = 1, *IsSourceRequirement* = 0, *IsSourceDiagram* = 0, *IsSourceSourceCode* = 0, *IsSourceDiagram* = 0.

Class

The class feature can take two possible values: a true or false value denoting a relation or a non-relation between the source and target artefact element.

Source and target id

These additional features allow the identification of the original source and target artefact elements. Unique ids are extracted from the GraphML representation of artefacts and take the same form as described in Section 8.4.

Table 8.3 illustrates a data instance as described by feature vectors. The data instance denotes a pair of source and target artefact element related by a trace link, hence the class feature takes the value 1. The source element represents a Java class called Maze, while the target element stands for a UML class which is also named Maze. The name similarity feature indicates a 100% match between the names of the source and the target. Since both the source and target elements are container types, the *IsSourceContainer* and *IsTargetContainer* features are assigned the value 1. The fact that source and target represent different abstraction levels is expressed by the value of the *AbstractionLevelSeparation* feature, which is set to 1. The features show the type of the source and target elements: since the source element is a Java class, the *IsSourceSoureCode* feature is set to 1, while the values of the remaining type features are set to 0. The same is applicable to representing the type of the target element.

8.6 Model Selection

The next step in devising the approach was selecting models, which were firstly trained on the training data and were subsequently validated. To the best of our knowledge this process has not been carried out in the context of heterogeneous artefact data. Therefore, a number of experiments were conducted using various models to compare their performance. Initially, in the exploratory

Feature	Value
NameSimilarity	1
IsSourceContainer	1
IsTargetContainer	1
AbstractionLevelSeparation	1
IsSourceRequirement	0
IsSourceSourceCode	1
IsSourceDiagram	0
IsSourceArchitecture	0
IsSourceUnitTest	0
IsTargetRequirement	0
IsTargetSourceCode	0
IsTargetDiagram	1
IsTargetArchitecture	0
IsTargetUnitTest	0
Related	1

Table 8.3: Feature vectors.

phase of the experiments, Matlab[10] was used. However, for the purposes of integrating the approach in the ACM framework, which is discussed in Section 8.9, the Waikato Environment for Knowledge Analysis (Weka)[11] was selected. Firstly, a simple linear technique, *the perceptron*, and a non-linear technique, *multilayer perceptron with backpropagation*, were explored. The initial selection of these two models was based on the premise that the potential failure of the linear technique to classify trace links demonstrates the inherent complexity of the system being modelled, and justifies a more complex approach. Subsequently, further standard classifiers, such as *J48*, *Naive Bayes* and *Support Vector Machines (SMO)*, were selected. Additionally, *ZeroR* was utilised to establish a baseline. The aim of the experiments was to compare the performance of the selected models, which are introduced in the subsection below.

Single Layer Network, the Perceptron

Artificial neural networks provide "a robust approach to approximating real-valued, discrete-valued, and vector-valued target functions" [191]. The first selected classifier, the perceptron, is the simplest form of artificial neural networks consisting of an input and an output layer. Inspired by the biological nervous system, it is an abstract model of the biological neuron dating back to the 1940s [209]. The basic premise of the model is that supplied with a vector of real-valued inputs, it performs a linear combination of them. Based on a threshold value, the model outputs

[10]http://uk.mathworks.com/products/matlab/index.html?s_tid=gn_loc_drop
[11]http://www.cs.waikato.ac.nz/ml/weka/

1 if the result is greater than the threshold, and outputs 0 otherwise. The perceptron can be used for problems where the data is linearly separable, i.e. there exists a hyperplane that perfectly separates positive and negative examples of the class [191].

Multilayer Perceptron with Backpropagation

To represent non-linear decision surfaces, multilayer networks are required, consisting of a number of neurons, which can be input, output or hidden units. Such networks are capable of solving non-linearly separable problems. Multilayer networks are most commonly used in conjunction with the backpropagation algorithm, which has been successfully applied in a large number of areas including speech recognition, pattern recognition and computer vision [210].

ZeroR

The simplest classifier relies on the target class and ignores all the other features. That is, it possesses no predictive power and merely identifies the majority class in the dataset. Therefore ZeroR can be leveraged as a benchmark for other classifiers [211].

Decision Trees - J48

The decision tree classifier builds classification models in a tree structure and by breaking the data set into smaller subsets. The output is a tree consisting of decision nodes and leaf nodes representing the classification result [192]. In this work, J48, an implementation of the C4.5 algorithm is used [212].

Naive Bayes

Naive Bayes is a probabilistic classifier based on Bayes Theorem. It is characterised by independence assumptions: according to the model all features contribute to the result equally and they are independent from each other [192].

Support Vector Machines - Sequential Minimal Optimisation (SMO)

Support Vector Machines (SVMs) date back to the 1970s. In its simplest form an SVM is a hyperplane "that separates a set of positive examples from a set of negative examples with maximum margin." [213] Finding the maximum margin is a quadratic programming problem. One way of solving this problem and a method of training support vector machines is SMO, which was applied in this work.

	All Systems	MazeSolver	JGAP	Neo4j	MyRobotLab	Java Binary Parser	Titan
No. of Relations (Positive instances)	512	57	93	127	52	106	77
No. of Non-Relations (Negative instances)	649	62	192	83	119	62	131

Table 8.4: Training data.

8.7 Methodology

The following section discusses the training methodology and provides a description of the implementation of model evaluation.

8.7.1 Training

As shown in Table 8.4, a total of *1161* data instances were generated across all systems. *512* data instances represent related artefacts, while the remaining *649* constitute unrelated pairs. Table 8.2 highlights which systems provide which artefacts. Since test cases are offered by most systems, this is reflected in the number of unit test - source code relationships contained in the data set.

Initially, the building, training and evaluation of the single and multilayer neural networks were performed using the Matlab Neural Network Toolbox[12]. Input training data was imported from CSV input files to Matlab and it was loaded into an *1161x14 feature matrix* (1161 instances with 14 features). Following the data input and the construction of the network architecture, the model was trained using *the perceptron*, and *the scaled conjugate gradient backpropagation* algorithm.

Following that, the Weka environment was used to build the ZeroR, J48, Multilayer perceptron, Naive Bayes and Support Vector Machines (SMO) classifiers. Weka provides an Experimenter tool to design and run experiments to allow the comparison of the accuracy of selected algorithms. Additionally, the models can also be trained and evaluated using the Weka Workbench or its Java API. The specifics of model evaluation are discussed in the next section.

8.7.2 Model Evaluation

The accuracy of a classifier is the percentage of test set data instances that are correctly classified [211]. In order to evaluate model performance and to find the model that most accurately represents both current and future data, various evaluation methods were utilised.

[12]http://uk.mathworks.com/products/matlab/

MazeSolver	JGAP	Neo4j	MyRobotlab	Java Binary P.	Titan
MazeSolver	JGAP	Neo4j	MyRobotlab	Java Binary P.	Titan
MazeSolver	JGAP	Neo4j	MyRobotlab	Java Binary P.	Titan
MazeSolver	JGAP	Neo4j	MyRobotlab	Java Binary P.	Titan
MazeSolver	JGAP	Neo4j	MyRobotlab	Java Binary P.	Titan
MazeSolver	JGAP	Neo4j	MyRobotlab	Java Binary P.	Titan

Table 8.5: Cross-validation: Systems highlighted in white are used for testing.

Firstly, **hold-out methods** [214] as part of a preliminary evaluation, randomly separate data instances into training, validation and test sets to avoid overfitting. In case of the multilayer perceptron, the training set is used for updating the network weight and biases, the validation set allows the tuning of the parameters of the classifier, and the test set is not used during training and is utilised for evaluating model performance on unseen data [215]. 70% of the data is assigned to the training set, while the remaining 30% is split into two equal parts which make up the validation and test sets. For the entire data set this means that the model was trained on 813 samples, and it was validated and tested using 174 and 174 instances, respectively.

Secondly, **k-fold cross-validation** [214] was performed to assess the accuracy and validity of the models. In the validation process Weka's default cross-validation settings were applied, which split the dataset into ten subsets, i.e. folds. Each fold is held out for testing while training is performed on the rest of the data. Results following training and cross-validation are presented in Figure 3.

Thirdly, in order to select the features with the most predictive power, the models were trained and evaluated on data with **different feature combinations**. This step is also a means to evaluate how data from different systems affects the performance of the models. The combinations are shown in Table 8.6, and are as follows:

- Exclude one specific feature from the feature space
- Combine two specific features
- Apply all features

Finally, the dataset was split into **six subsets**, each of which represents one of the systems. In the first iteration the models were trained on a training set that excludes one of the subsets, which in turn is used for testing. In each iteration, a new system was selected for testing. The process and the selection of test sets are illustrated in Table 8.5. The subsets, as shown in Table 8.4, are not of equal size, they differ in the number of positive and negative instances, and they represent different artefact types.

8.8 Results and Discussion

In the following section the results of training and validation are presented. Using the perceptron algorithm, which was trained using Matlab with default settings and was supplied with the entire dataset, convergence was not reached in 1000 iterations. The experiment highlights that the perceptron is not able to separate positive and negative classes in the specified number of steps.

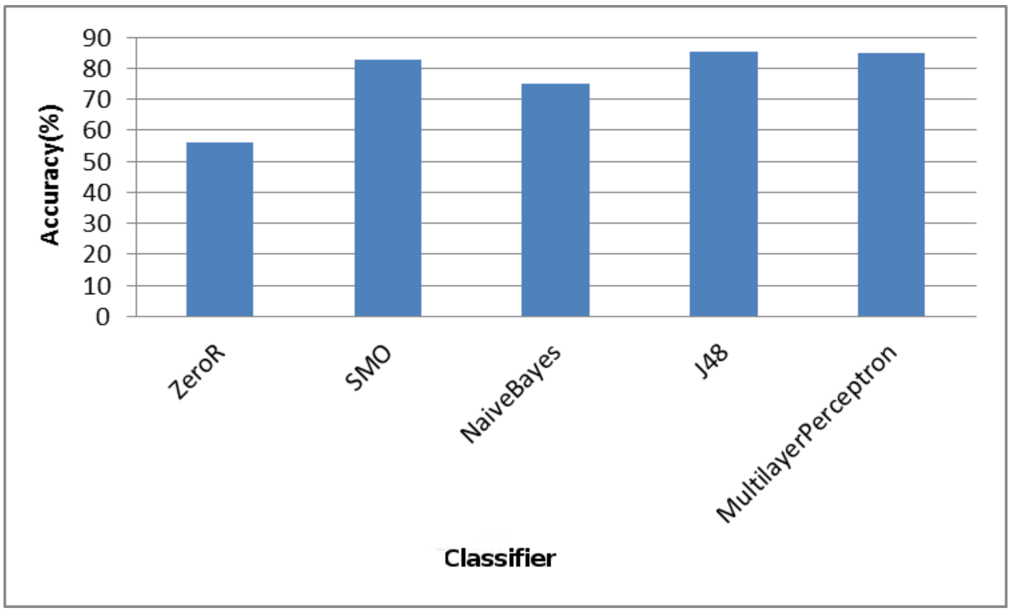


Figure 8.1: Accuracy results of the J48, Multilayer Perceptron, Naive Bayes, SMO, and ZeroR classifiers.

Figure 8.1 compares the classification performance following the training and 10-fold cross-validation of the J48, Naive Bayes, SMO, Multilayer Perceptron and the Zero R classifiers. These results were obtained by using all the features and the entire dataset consisting of all systems. The accuracy value represents the percentage of correctly classified instances. The results show that J48 and the Multilayer perceptron classify the data instances with similar accuracy values and are the best performing models compared to the Naive Bayes and SMO algorithms. Finally, the ZeroR classifier sets the baseline at 55.9% accuracy.

The J48, Multilayer Perceptron, SMO and Naive Bayes classifiers were also evaluated using different feature combinations on all systems. The results are presented in Table 8.6; the first column shows features and their combinations, while subsequent columns represent the classifiers. The values in the intersections are accuracy values obtained from Weka. The results highlight the differences between systems and the weight of specific features and their combinations. Certain features show significantly worse performance than others. For example, training the Multilayer Perceptron on the *AbstractionLevelSeparation* feature results in 59.1% accuracy, while excluding

Selected Features	Multilayer Perceptron	J48	SMO	Naive Bayes
AbstractionLevelSeparation	59.1	61.2	61.2	61.2
All features excluding AbstractionLevelSeparation	85.9	85.2	82.6	77.4
All features excluding IsSourceContainer & IsTargetContainer	80.8	81.3	81.2	78.8
All features excluding NameSimilarity	67.7	70.1	70	65.3
All features excluding Type features	84.2	85.5	79.8	76.4
All features	85.7	85.2	82.6	75.1
IsSourceContainer & IsTargetContainer and AbstractionLevelSeparation	67	69.1	69	66.2
IsSourceContainer & IsTargetContainer and Type features	68.3	70.1	70.3	65.5
IsSourceContainer & IsTargetContainer	69.4	69.1	69	66.2
NameSimilarity & AbstractionLevelSeparation	77.95	79.5	76.5	74.3
NameSimilarity & IsSourceContainer and IsTargetContainer	81.3	81.7	79.8	79.8
NameSimilarity & Type features	80.8	81.3	81.2	79.5
NameSimilarity	73.6	73.4	72.6	73.2
Type features & AbstractionLevelSeparation	62.1	63.3	63.3	63.3
Type features	62.1	63.3	63.3	58.6

Table 8.6: Accuracy of the Multilayer Perceptron, J48, SMO and Naive Bayes classifiers using different feature combinations.

the feature and training and validating on the remaining features shows an accuracy value of 85.9%. In comparison, excluding *NameSimilarity* results in 67.7% accuracy, and *NameSimilarity* on its own also performs poorly at 73.6%. The findings of the Multilayer perceptron suggest that accuracy of the model does not rely on a single feature. However, certain combinations of *NameSimilarity*, *AbstractionLevelSeparation*, *IsSourceContainer*, *IsTargetContainer*, and the *Type* features yield better accuracy values. The other classifiers, J48, SMO and Naive Bayes, do not show significantly different results and allow the same conclusions to be drawn.

Finally, Figure 8.2 shows the accuracy values obtained following the cross-validation of the models using each system as a test set. Validating the performance of the Multilayer Perceptron on MazeSolver results in 96.6% of the instances being correctly classified, while JBBP shows a considerably lower accuracy value, 47%. The results could be explained by the differences between the systems in terms of artefact types and the number of inter links connecting artefact elements. The features describing heterogeneous artefacts may be better suited to a set of artefacts, for example provided by MazeSolver, where there is not a wide gap between the abstraction levels of artefacts. An additional factor may be the strategy used for establishing trace links. Finally, the systems also differ in the number of data instances they provide. Thus, the trained model may not in all cases be applicable to a specific system, and discovering the degree to which it can be generalised to all systems, forms part of future work.

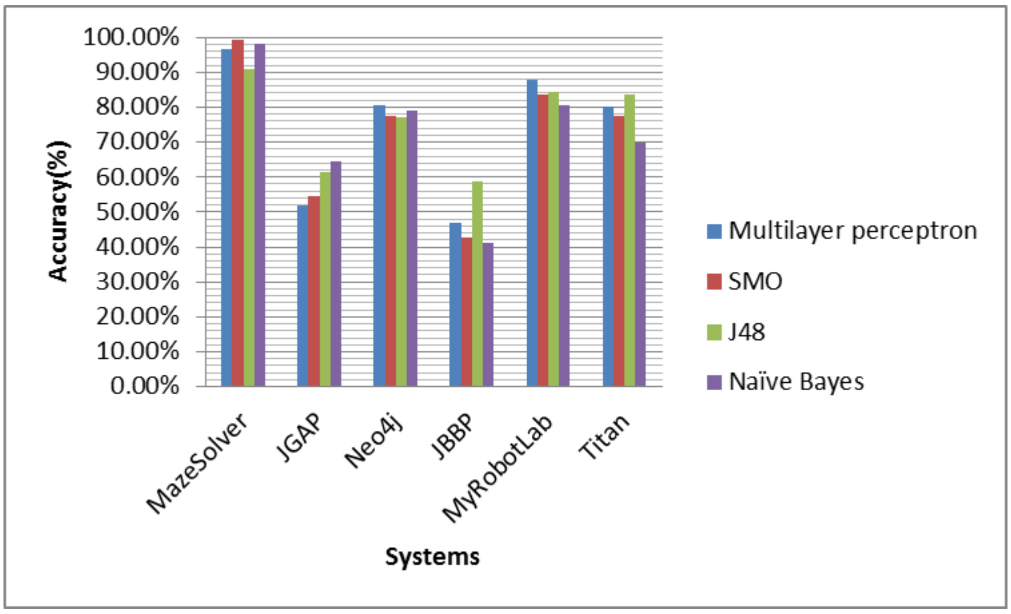


Figure 8.2: Cross-validation results: each system is used as a test set.

8.9 Integration in the Framework

The final step of implementing the approach is integrating it in the ACM framework to arrive at a semi-automatiac trace link creation approach. The integration can be achieved using a wide variety of machine learning libraries including Weka, Encog[13], PyBrain[14] and scikit[15]. For the current implementation the Weka API was selected since the framework is written in Java.

The aim of the *Traceability creation* stage of the framework is to assist users with their inter trace link creation tasks as part of setting up Framework Data. The input to inter trace link creation is supplied by users in the form of candidate links stored in a *.xml* links file, which is discussed in Chapter 6. Following the training of the selected classifier, the framework runs the model on test data obtained from the *.xml* links file and classifies each data instance. The user is presented with a final *.xml* links file, which contains trace links as returned by the selected model.

8.10 Conclusions

The poor performance of the linear model, the perceptron, empirically confirms the widely held belief about the complexity of this problem. Using the multilayer perceptron the experiments show a prediction accuracy of 85.7% in cross validation, which is closely followed by the J48 algorithm at 85.2% accuracy. The differences in the accuracy values of the models obtained following cross-validation can be explained by the diversity of the modelled systems. However, the accuracy results prove that using machine learning to aid the automation of trace link creation is a viable approach and it is worth further investigating. Besides observing the benefits of applying the approach in this complex problem domain, which does not assume the use of specific artefacts or development conventions, the shortcomings machine learning approaches may present are also to be discovered.

Potential improvements to the current implementation include extending the breadth of programs considered by utilising different systems providing further artefact types. Further experiments are required to analyse the correlation between systems and features. The solution also forms a basis for further work in various other areas, such as visualising results, and allowing users to edit trace links predicted by the model. Since each classifier is trained with default settings, an investigation into fine-tuning the parameters of classifiers may provide further classification accuracy gains. Specifically, since the J48 classifier shows promising results, pruning methods could be utilised [216].

[13]http://www.heatonresearch.com/encog
[14]http://pybrain.org
[15]http://scikit-learn.org/stable

Chapter Nine

Evaluation

This chapter describes the evaluation strategy to analyse the applicability of the framework in software development scenarios and to test the functionality of its individual components. It starts with summarising the hypotheses presented in Chapter 1, and the requirements of the proposed approach discussed in Chapter 4. Thereby, the objectives of the evaluation are established and the evaluation questions are formulated. Subsequently, appropriate research methods pertaining to each question are selected. This provides the basis for evaluating the proposed approach and its implementation, the ACM framework. Table 9.1 shows the relationships between hypotheses and requirements, and the process of deriving corresponding evaluation questions and research methods. The *Evaluation aim* column highlights the evaluation concerns. Next, the evaluation process is designed and implemented. These steps involve data collection, data analysis and testing the framework against the collected data. Finally, the results are analysed and conclusions are drawn while taking into account validity considerations.

9.1 Evaluation Objectives

The aim of the evaluation is threefold: the main objective is to provide a verification of the hypotheses using *empirical software engineering* methods to critically evaluate the degree to which the proposed solution addresses the problem at hand. Through evaluation, it is revealed whether the stated hypotheses are realistic, and the strengths and weaknesses, and areas of potential improvements are also identified. Secondly, the correctness of the results achieved at each stage of the consistency management process is tested using software engineering *validation* and *verification* methods. Lastly, the performance of the solution is analysed. A solution is suitable for wider adoption if, besides other criteria, it meets demands to scale under varying workloads. Performance is measured using appropriate *metrics*. Table 9.1 also highlights that *H1* is not mapped to a requirement and an evaluation question. The reason for this choice is the fact

Hypotheses	**Requirements**	**Evaluation Aim**	**Evaluation Questions**	**Evaluation Methods**
H1		Hypothesis		Design & implementation of proposed approach
H2	R3, R4	Hypothesis	Q1	*Case study*
H3	R1	Hypothesis	Q2	Design & implementation of proposed approach, *case study*
H4	R2, R4	Hypothesis	Q3	*Case study*
	R6	Performance	Q4	*Performance metrics*
	Func. Req. & R4	Correctness	Q5	Validation & verification
	R5	Outside the scope of this thesis		

Table 9.1: Derivation of evaluation questions and methods from the hypotheses and requirements.

that the feasibility of the proposed approach expressed in *H1* is investigated through the design and implementation of the ACM framework.

9.2 Evaluation Questions

The evaluation is based on the following questions:

Q1: Is the ACM framework independent of methodologies and tools? This questions investigates whether the framework imposes any additional tasks on the user that are not part of their usual work, such as including annotations in any of the artefacts, and whether it imposes additional tools on users instead of their usual CASE tools. This question can be answered through **qualitative** methods.

Q2: Can the stages of consistency management identified in Chapter 4 be automated and to what degree? The goal of this question is to discover the level of automation that can be attained for each stage, and possible limitations. This question can be evaluated **qualitatively**.

Q3: Is the ACM framework independent of artefacts? This question is aimed at assessing whether the framework can handle heterogeneous artefacts, and if it can be extended to cater for additional artefact types. This question can be evaluated **qualitatively**.

Q4: Performance: does the ACM framework support varying workloads when subjected to
a, a varying number of artefacts (system size),
b, a varying number of artefact nodes (artefact size),
c, a varying number of changes? (change complexity)
This question can be investigated **quantitatively**.

Q5: Does the ACM framework fulfil its functionality requirements and produce the expected output in each stage of the consistency management process? This question also entails discovering the degree to which the framework can be configured and customised to suit users' preferences. This final evaluation question is assessed by comparing results produced by the framework with expected results.

Questions *Q1* to *Q3* are investigated using *empirical software engineering* methods. On the other hand, *Q5* is answered using *validation* and *verification*. Finally, analysing *Q4* requires the use of *metrics*.

9.3 Evaluation Design

The design of the evaluation consists of two steps. Firstly, the most suitable methodology to investigate each evaluation question was selected. Secondly, the data collection strategy was planned and specific systems were selected.

9.3.1 Research Method Selection

9.3.1.1 Evaluation of Hypotheses

As described by Sjøberg et al., empirical research "seeks to explore, describe, predict, and explain natural, social, or cognitive phenomena by using evidence based on observation or experience" [217]. Empirical software engineering research provides an extensive toolset to achieve these goals. After a careful consideration of the evaluation questions and the alternatives, such as controlled experiments and surveys [218], the method of case studies was selected.

Case studies are widely used in software engineering and can be defined as a method "aimed at investigating contemporary phenomena in their context" [219]. The primary motivation for using a case study approach is to prove that the hypotheses hold in real project scenarios, which is a significant consideration for a software engineering solution. Additionally, through a case study a deeper understanding of the problem can be gained and potential shortcomings of the proposed approach can be discovered.

Criteria for Success

A pivotal aspect of the design of the case study is the specification of success and failure criteria. Success criteria were established per evaluation question.

Q1 - Methodology and Tool independence
Success: The consistency management process does not require the changing of the methodology that is used to create the original artefacts. Additionally, users are not required to utilise further tools to create and edit artefacts. The outcome is a true or false statement that accepts or rejects the corresponding hypothesis.

Q2 - Level of automation
Success: One or more aspects of artefact consistency management can be carried out without manual intervention. The outcome of the investigation is a list of automatic, semi-automatic and manual steps.

Q3 - Artefact independence
Success: The framework can be extended to handle any software artefact. The outcome is a true or false statement that accepts or rejects the corresponding hypothesis.

9.3.1.2 Correctness testing

Framework correctness is evaluated through software engineering *validation* and *verification*. Specifically, each functional area is tested and expected outputs are compared to actual outputs. Both individual functional units (unit testing) and collections of a number of functional units (integration testing) are assessed. The high level functionality areas, inputs and expected outputs are summarised in Table 9.2. The tests can be found in the ACM framework's GitHub repository[1]. Test packages are named as follows: *framework.X.Tests*, where *X* stands for the given framework component or functionality area to be tested.

Criteria for Success

Besides identifying the expected outputs of each functionality area, the following *success criteria* were defined determining if a given test passes or fails.

- **Artefact Extraction.** XML representation of original artefacts is saved to the specified framework folder in the form of *.xml* files.

[1]https://github.com/ACMFramework/ACMF

Functionality	Inputs	Outputs
Artefact Extraction	Original artefacts in their original format	XML representation of original artefacts
Artefact Transformation	XML representation of original artefacts, XSLT stylesheets	GraphML representation of artefacts
Traceability Creation	GraphML representation of artefacts converted to feature data for classification	XML representation of trace links, containing pairs of ids of connected GraphML nodes
Data Storage	GraphML representation of artefacts, XML representation of trace links	Graph database populated with data consisting of nodes and edges
Change Detection	A change in an external repository	ChangeData and updated graph database
Change Impact Analysis	ChangeData	Set of potentially impacted artefact elements expressed as graph nodes
Consistency Checking	ChangeData, set of potentially impacted artefact elements	Artefact element is consistent, potentially inconsistent, or inconsistent
Change Propagation	ChangeData, list of inconsistent elements	Update suggestions for each element to re-establish consistency
Configuration	User input (database location, external repository location, XSLT file path and framework root folder)	The framework configuration file is populated with values specified by the user

Table 9.2: Functionality areas, inputs and expected outputs.

- **Artefact Transformation.** GraphML representation of XML inputs is saved to the specified framework folder in the form of *.graphml* files. GraphML representation captures required artefact data: artefact elements and their *intra* trace links.
- **Traceability Creation.** XML links file is produced and contains correct trace links between artefact elements obtained from GraphML representation.
- **Data Storage.** Graph database is populated with data obtained from the *.graphml* and *.xml* links files.
- **Change Detection.**
  A) File level changes in external repository are extracted and identified.
  B) Artefact element level changes, if any, are identified.
  C) Local representations in the framework folder are updated.
  D) Graph database nodes, properties, and specific edges are updated.

- **Impact Analysis.** The framework returns a set of potentially impacted elements based on graph traversals. This includes graph nodes directly connected to the changed node through *inter* and *intra* links.
- **Consistency Checking.** The framework returns one of the following results: *consistent*, *inconsistent* or *potentially inconsistent* based on consistency checking rules.
- **Change Propagation.** The framework suggests resolutions to each identified (potential) inconsistency.

9.3.1.3 Performance Evaluation

Performance measurements reveal how the framework performs when subjected to specific workloads [220] in terms of the number of artefacts, size of artefacts and the number of changes. This question can be analysed through a *case study* approach and specific *metrics*. For this evaluation, *execution time (s)* was selected. The objective of scenarios 1 and 2 is to reveal the correlation between execution times and the number of artefacts. In scenario 1 the steps involved in *Framework Setup* are tested, while scenario 2 measures execution times of the *Consistency Management* steps. Scenarios 3 and 4 investigate the correlation between artefact size and execution times. Finally, scenarios 5 and 6 measure the performance of the change identification algorithm and change detection, respectively.

Scenario 1. Measure execution times of *Framework Setup* with a system consisting of the smallest number of artefacts (MazeSolver), the largest number of artefacts (MyRobotLab), and a system in between the two (JBBP).

Scenario 2. Measure execution times of *Consistency Management* with a system consisting of the smallest number of artefacts (MazeSolver) and a system consisting of the largest number of artefacts (MyRobotLab).

Scenario 3. Measure execution times of *Framework Setup* with artefacts consisting of the largest and smallest number of nodes. The inputs include:

- GraphML file representing the *Service* Java class of the *MyRobotLab* system, which contains *171* nodes. This artefact contains the highest number of nodes out of the artefacts used in this evaluation.
- GraphML file representing the *Owner* interface from the *Neo4j* system. This interface represents the other end of the spectrum with *2* nodes, which is the lowest number in the data set.
- GraphML file representing the *JBBPToken* class from the *JBBP* system. The number of nodes in this class (37) fall between the number of nodes in the *Service* class and *Owner* interface.

Scenario 4. Measure execution times of *Consistency Management* with artefacts consisting of the largest and smallest number of nodes (*Service* class, *Owner* interface).

Scenario 5. Measure execution times of the *change identification* algorithm, which is part of the Change Detection framework stage. This test is aimed at measuring the performance of the algorithm with the largest and smallest number of nodes and all *artefact element level* change types. The inputs of this test scenario are the same as described in Test scenarios *3* and *4*.

Scenario 6. Measure execution times of *Change Detection* with different change types. This test is aimed at measuring the impact of change types on the performance of change detection.

9.3.2 Data Collection

9.3.2.1 Selecting a Data Collection Technique

The selection of the most suitable data collection method was driven by the evaluation objectives and questions. An additional factor was the volume of data required for carrying out the evaluation. The technique that best fulfils these requirements was chosen from a number of data collection methods for software field studies. For the purposes of this evaluation, second degree techniques were considered, which are characterised by an indirect involvement of software engineers. Such techniques include *Static and Dynamic Analysis of a System* and *Documentation Analysis*. In this work, the *Analysis of Electronic Databases of Work Performed* technique was selected, which took the form of extracting artefact data from online version control systems [221]. The aim of obtaining data from existing open source software development projects hosted in online repositories was to allow the evaluation of the solution in realistic project scenarios. This technique is also extensively used in research related to mining software repositories [124]. Comments in code were not considered as their investigation is outside the scope of this thesis.

9.3.2.2 Selecting Particular Open Source Systems

The next step was the identification of subject systems. The selection criteria are described in Chapter 8, and for convenience a brief summary is provided here. Principally, candidate systems are required to provide a wide range of artefacts to assess artefact independence. Therefore, this requirement stems from question *Q3*. Another aspect is system size. The evaluation of question *Q4* requires different system sizes to model different levels of complexity in terms of the number of artefacts. Challenges encountered and specifics of the selected systems are described in Chapter 8. The particular systems used in different steps of the evaluation process are specified in each corresponding step.

9.3.2.3 Change Selection

An integral part of evaluating the framework was introducing changes to artefacts. For this purpose, existing changesets from the selected repositories were taken. The changesets provide changes of varying sizes and complexity. The main motivation for selecting existing changesets was to capture realistic project scenarios. Since changes in open source repositories are to a large extent constrained to source code, custom modifications to other representations, such as UML class diagrams, were also introduced.

9.3.2.4 Artefacts Obtained from Open Source Systems

From each system a subset of artefacts were obtained. This is due to the challenge of establishing *inter* trace links in larger systems, where relationships between entities may potentially be complex. The task therefore requires domain and expert knowledge of the given system to ensure that the property graph representation of the system is accurate. Hence the problem was constrained to a subset of artefacts. The number of artefacts obtained from each system is shown on Table 9.3. The types of artefacts and the methodology of establishing trace links between them are discussed in detail in Chapter 8.

9.4 Methodology and Results

In the following section a description of the evaluation methodology and a report of results are provided. As mentioned in Section 9.1, the overall evaluation serves three purposes. Accordingly, this section is split into three subsections. Firstly, the steps involved in carrying out testing for correctness are detailed along with a discussion of results. Next, the implementation of the case study and its results are described. Lastly, the methodology applied in assessing performance is introduced, followed by a discussion of results.

9.4.1 Methodology: Testing Correctness

Q6 was evaluated using the tests introduced in Section 9.3.1.2. These tests also demonstrate basic usage scenarios of the framework and utilise data obtained from the JGAP system. Additionally, tests are grouped into two distinct scenarios, *Framework Setup* and *Consistency Management*. The organisation of this section follows these scenarios.

9.4.1.1 Framework Setup Scenario

The first step was carrying out setup, in which the framework was configured, followed by data extraction and transformation. Subsequently, trace links were established. Lastly, artefact

System	**Number of artefacts**
MazeSolver	Java source code: 10 JUnit test: 1 UML class diagram:2 Total: 13
JGAP	Java source code: 61 JUnit test: 47 UML class diagram: 1 Total: 109
Neo4j	Java source code: 77 JUnit test: 39 Module view architecture: 1 Total: 117
MyRobotLab	Java source code: 334 JUnit test: 6 UML sequence diagram: 1 Total: 341
Java Binary Block Parser	Java source code: 76 JUnit test: 55 UML Use case diagram: 1 Total: 132
Titan	Java source code: 133 JUnit test: 34 Conceptual architecture: 1 Total: 168

Table 9.3: The number of artefacts obtained from each open source system.

element and link data were saved to the graph data store.

To perform framework setup, the following prerequisites are required:

- Neo4j (version 2.1.3 minimum) is installed
- Mercurial is installed, and the path of the local and remote repositories are available
- Java (version 7) is installed
- Original artefacts are in local Mercurial repository
- Artefacts can be exported to an XML-based representation (The *src2ml* tool can be used to extract Java, C, C# and C++ source code files. Automatic extraction is also available for

SQL[2], JavaScript[3] and Python[4].)

- Additional XSLT files for new artefact types are supplied.

1. Configure Framework

Firstly, the framework root directory, *ACMF*, and its subfolders were created in the local file system. The subfolders *SourceCode* and *UnitTests* were populated with executables necessary for automating artefact data extraction. The second step involved editing the framework configuration file to setup the database path, the framework root folder path, and the local and remote repository paths.

The JGAP system utilised in these tests provides Java source code, JUnit test, and UML class diagram artefacts. Since the available UML diagrams are not consistent with the latest version of the source code, additional diagrams were generated using the code generation functionality of *Eclipse*. All the original artefacts were placed in the external repository the framework has been configured to use.

2. Extract Artefact Data

The input to this step was provided by the original artefacts located in the external repository. These include *.java* and *.dia* files. To extract Java source code and JUnit tests the framework calls the *src2ml* script, which takes all *.java* files from the specified local repository and produces a *.java.xml* representation. Requirement specifications, when available, can be extracted using the *Apache ODF Toolkit*. Obtaining artefact data from UML class diagrams is currently achieved by manually invoking the export functionality in *DIA*, which outputs a *.vdx* file. At the end of the extraction process, the following outputs were produced: *.vdx* files representing UML class diagram artefacts, and *.java.xml* files representing Java source code and JUnit test artefacts. The outputs are stored in the specified framework subfolders.

3. Transform XML Representations to GraphML

Next, transformation was performed. All *.java.xml* and *.vdx* files located in the subdirectories of the framework folder were transformed to a GraphML representation. The output of the process was a set of *.graphml* files placed in the framework's subfolders.

[2]https://crmbusiness.wordpress.com/2011/10/27/how-to-convert-an-sql-statement-into-a-xml-file/
[3]https://www.npmjs.com/package/js2xmlparser
[4]http://pythonhosted.org/pyRegurgitator/

Test scenario	Result
Configure Framework	►Passed. The configuration file is populated with the specified values.
Extract Artefact Data	►Passed. Specified folder is populated with .java.xml and .xml files extracted from original .dia, .java and .odt representations.
Transform to GraphML	►Passed. Specified folder is populated with .graphml files. .graphml files contain correct artefact data.
Setup Inter Trace Links	►This step is manually performed. The result is an .xml file containing trace links.
Import to Graph Database	►Passed. Neo4j is populated with nodes and edges.

Table 9.4: Summary of Framework Setup results.

4. Setup *Inter* Trace Links

During correctness testing this step was performed manually: based on the input *.graphml* files, an *.xml* links file was produced. However, the framework provides an approach, described and evaluated in Chapter 8, to automate *inter* trace link setup. The output of this approach can be incorporated during setup and can be approved by the user.

5. Import Artefact Elements and Trace Links to Graph Database

In this step all the GraphML representations were automatically imported to the *Neo4j* database, and edges were established between nodes based on the contents of the *.xml* links file. At the end of the process the database was populated with artefact element data represented as nodes, and trace links connecting artefact elements in the form of edges. This concludes the framework setup process.

6. Framework Setup Results and Discussion

Testing the Framework Setup scenario confirms that assuming the prerequisites highlighted at the beginning of this section are satisfied, following artefact data extraction and transformation, artefact elements and trace links are correctly saved to the data store.

9.4.1.2 Consistency Management Scenario

Following setup, the framework is ready to perform consistency management. Firstly, change detection was initiated by introducing changes to existing artefacts. Subsequently, change impact analysis, consistency checking and change propagation were performed. Consistency management correctness tests utilised the *Salesman* and *SalesmanFitnessFunction* Java source code artefacts and their UML class counterparts from the JGAP system.

1. Introduce Changes to Original Artefacts

The input to consistency management was supplied through updating the repository by adding new artefacts, and deleting or editing existing ones. The aim of generating changes was to span both *file* and *artefact element level* modifications and to cover a number of change combinations. The following section provides a summary of the applied changes.

Scenario 1. Add new artefact. This change at the *file level* equates to creating a new *.java, .dia* or *.odt* file and it may represent adding a Java or JUnit class, or other container types such as interfaces or enums. Additionally, it may stand for changes that involve creating a new requirement specification document or a new UML class diagram.

Scenario 2. Delete existing artefact. This change represents the delete counterpart of Scenario 1. and utilises the *Salesman* class.

Scenario 3. Edit existing artefact. This *file level* change can be broken down into further *artefact element level* modifications. It represents a scenario in which contents of an existing artefact are updated by either adding new elements to it, or editing or deleting existing ones. Table 9.5 highlights the *artefact element level* changes and their combinations pertaining to each artefact type available in the JGAP system. Due to the structural similarity of Java source code, JUnit test case and UML class diagram artefacts, the same tests are applicable. The *artefact level changes* summarised in Table 9.5 are performed using the *SalesmanFitnessFunction* class, its *m_salesman* field, its *SalesmanFitnessFunction* constructor and *evaluate* method.

2. Perform Change Detection

Change detection was initiated by invoking the *manageChangeDetection()* method in the *ChangeDetectionManager* class. Firstly, the *file level* change was identified. Next, depending on the change type, different courses of action were taken. Ultimately, all changes resulted in the graph database and GraphML representations being updated. However, edit changes also required the identification of *artefact element level* change types. The output of the change detection process was a list of *ChangeData* objects. The aim of this step was to reveal if the output produced was correct when compared to manual change detection. The correctness of the

Artefact Element Level Change	Affected Structural Element
Edit	Class/Enum/Interface/Method/Field name, modifier, and specific properties, such as parameters, contents, return type, and their combinations
Add	Field, Method
Delete	Field, Method
Add and Delete	Field, Method
Add and Edit	Add Field, Method Edit Class/Enum/Interface/Method/Field name, modifier, and specific properties, such as parameters, contents, return type
Edit and Delete	Delete Field, Method Edit Class/Enum/Interface/Method/Field name, modifier, and specific properties, such as parameters, contents, return type

Table 9.5: Summary of artefact element level change combinations in Java source code, JUnit test, and UML class diagram artefacts.

output is critical as subsequent framework stages rely on it.

3. Perform Change Impact Analysis

Checking impact analysis results for correctness was achieved through the comparison of results obtained through manual change impact analysis with those produced by the framework. Firstly, *Change Data* was supplied to the *ChangeImpactAnalyser* class. Change impact analysis functionality was initiated by invoking the *execute()* method, which created an *IAResult* object for each change. The list of *IAResult* objects contains the set of potentially impacted graph database nodes for each change.

4. Perform Consistency Checking

Based on the list of *IAResult* objects, consistency checking was invoked by calling the *execute()* method of the *ConsistencyChecker* class. The output was a list of *ConsistencyCheckResult* objects indicating whether the elements in question are consistent, potentially inconsistent, or inconsistent.

Test Scenario	Result
Change Detection	► Passed. A) File level changes are extracted and correctly identified B) Artefact element level changes are correctly identified. C) Local representations (.xml and .graphml) in the specified framework folder are correctly updated D) Graph database nodes, properties, and specific edges are correctly updated.
Change Impact Analysis	► Passed. The framework correctly returns nodes connected to the changed node.
Consistency Checking	► Passed. The framework identifies (potential) inconsistencies based on the specified rules.
Change Propagation	► Passed. The framework provides suggestions to resolve (potential) inconsistencies.

Table 9.6: Summary of Consistency Management test scenarios and results.

5. Perform Change Propagation

Utilising the list of *ChangeData* objects and the output of consistency checking, change propagation suggested resolutions of inconsistencies and it was invoked through the *execute()* method of the *ChangePropagator* class. Since this is the last step of consistency management, it is also responsible for removing nodes labelled for deletion from the graph database.

6. Results and Discussion

The results of consistency management test scenarios are summarised in Table 9.6.

Change Detection

Change detection returned the expected results in case of *add*, *edit* and *delete file level* changes. It catered for both individual *artefact element level* changes and their combinations described in 9.5. However, it raised a number of issues as a result of the particulars of the *change identification* algorithm. As mentioned in Chapter 6, the *change identification* algorithm considers *rename* operations as a combination of *delete* and *add* artefact element level changes. For example, in case the *evaluate()* method is renamed to *evaluate_m()*, the modification is flagged up as a *delete* operation where *evaluate()* is removed, and an *add* operation where *evaluate_m()* is added. Since the results of change detection provide input to change impact analysis, the output of rename operations has implications on the results of impact analysis and further subsequent stages.

Another feature of the *change identification* algorithm concerns handling multiple constructors and methods in Java source code artefacts. In this specific case, multiple entities share the same name and differ in their parameters. In case any of them changes, all the entities with the same name are identified as being subject to modifications. This is the default behaviour of the algorithm as parameters are a property that can also change and hence cannot be used to identify entities across two versions of the same artefact.

Besides returning a list of *ChangeData* objects, change detection is also responsible for correctly updating the graph database and the file system. Specifically, in case of *add file level* changes, the XML and GraphML representations of the new artefacts are generated and are imported to Neo4j. In case of *delete file level* changes, the corresponding XML and GraphML representations are removed from the framework folder and the relevant graph database nodes are labelled for deletion. Should an artefact be *edited*, its XML and GraphML representation are re-generated and correct unique ids are assigned. Finally, property values of nodes in the graph database are correctly updated.

Change Impact Analysis

Impact analysis returns elements directly connected through both *inter* and *intra* trace links. These elements constitute the *Estimated Impact Set (EIS)*. As per the change impact analysis algorithm, no impact set is returned for *add file level* changes. In case Java source code, JUnit test and UML class diagram artefacts (member elements) are affected by *add artefact element level* changes, impact analysis returns inter traversal results for the parent of the modified entity.

As shown by the following example, the impact set returned by the framework can differ from manual results and the *Actual Impact Set (AIS)*. This is due to the *pessimistic* impact analysis approach of the framework, which deems all connected elements potentially affected. Currently, this is achieved by considering all direct connections on the graph. On the other hand, since indirect connections are not catered for, the EIS may not include elements that are actually part of the AIS. Upon multiple invocations of the consistency management functionality, indirect connections may also be identified to be potentially affected. However, considering them within a single consistency management iteration is part of future work.

As mentioned in Chapter 4, precision and recall are two metrics that are used to measure the accuracy of change impact analysis. However, precision and recall metrics are dependent on the types of changes, the artefacts, the level of human expertise in the given system, which pose threats to validity and are discussed in Subsection 9.4.4.

- **Edit** the signature of the *m_salesman* field in the *SalesmanFitnessFunction* class. This change affects the *m_salesman* field specified in the UML class diagram due to an inter link that exists between them and it was correctly added to the impact set. The field is connected to the *SalesmanFitnessFunction* class through an *intra* link, therefore the class and all its members were added to the impact set. Manual impact analysis showed that the modification of the field affected the *SalesmanFitnessFunction* constructor and the *evalute* method of the class, while remaining members were not affected. Therefore in this case, the EIS was larger than the AIS.

- **Delete** the *SalesmanFitnessFunction* constructor of the *SalesmanFitnessFunction* class. The modification impacts the *SalesmanFitnessFunction* method in the UML class diagram that is connected through an inter link. Similarly to the *edit* scenario, the *SalesmanFitnessFunction* class and all its members were flagged up as potentially impacted elements due to their intra links. Thus, in this case, the EIS was larger than the AIS.
- **Add** new method *newMethod* to the *SalesmanFitnessFunction* class. Since the *SalesmanFitnessFunction* Java class is connected to the *SalesmanFitnessFunction* UML class through an *inter* link, this was returned as a potentially impacted element. Similarly to the *edit* scenario, the *SalesmanFitnessFunction* class and all its members were flagged up as potentially impacted elements due to their intra links. Thus, in this case, the EIS was larger than the AIS.

Additionally, correctness tests also reveal that errors can introduce unexpected and incorrect results: the framework heavily relies on trace links being correctly identified. For example if there is a missing *inter* link between the *SalesmanFitnessFunction* class and the *SalesmanFitnessFunction* UML class, the addition of a UML member entity does not flag the container as being potentially affected by the change.

Consistency Checking

The consistency checking mechanism takes the elements of the impact set and based on the specified rules evaluates them to be consistent, inconsistent or potentially inconsistent. The results suggest that the framework can identify potentially inconsistent elements within the same artefact and can make a decision if another related artefact is also inconsistent due to the modification. Since not all identified possibly inconsistent elements are actually inconsistent, the framework produces a number of false positives, which is due to its pessimistic approach to consistency management. In cases where it cannot be said with certainty that the potentially impacted entities are consistent, the frameworks flags up potential inconsistencies. On the other hand, if not all dependencies are modelled by explicit trace links, other potentially inconsistent entities may not be identified.

To test the correctness of consistency checking results, the same examples were used as above. Editing the signature of the *m_salesman* field in the *SalesmanFitnessFunction* class results in an inconsistency of the *SalesmanFitnessFunction* UML class. Deleting the *SalesmanFitnessFunction* constructor of the *SalesmanFitnessFunction* class was correctly identified as an inconsistency, which affects the *SalesmanFitnessFunction* UML class. Adding a new method to the *SalesmanFitnessFunction* class results in an inconsistency of the the *SalesmanFitnessFunction* UML class, which was correctly identified. Since establishing a comprehensive set of trace links using this system requires expert knowledge of it, missing dependencies may result in undiscovered inconsistencies. Missing fine-grained intra links may also impact the results of consistency checking. For example, links are not established between the contents of a method and the entities referenced inside. The introduction of additional, finer-grained consistency checking rules and the extension of impact analysis to indirect dependencies are potential avenues for improving results.

The ACM framework currently analyses changes separately from each other, which results in connections between modifications not being captured: some changes require further modifications in order to manage inconsistencies within the given artefact. Consequently, the framework may flag up the impact of a change up as potential inconsistency, even though these are resolved by subsequent changes. The understanding of such connections is an area for future work including the establishing of different granularity levels of impact analysis and consistency checking either to handle each change individually or the change set as a whole. This feature could potentially be customised by the user.

Change Propagation

Based on the results of consistency checking, change propagation correctly displays suggestions. In case of a (potential) inconsistency, the user is recommended to apply the same *file level* change to the potentially impacted and inconsistent artefact element as the original change. In case of *Add file level* changes a message is displayed to the user and no change propagation is carried out.

9.4.2 Methodology: Evaluation of Hypotheses

9.4.2.1 Q1 - Tool and methodology independence

The assessment of whether the use of the framework imposes specific tools or methodologies involves answering the following questions:

1. Is it possible to extract data from any artefact authoring tool?
2. Does the framework impose any specific methodologies on its user?

System	Tools Used in Design or Development
JGAP	JBuilder
MMS	Eclipse
JBBP	NetBeans
MyRobotLab	Eclipse, NetBeans, Dia

Table 9.7: Tools used in the design or development of selected subject systems.

The first question requires carrying out the steps of *Framework Setup*. Firstly, the original artefacts were obtained. Subsequently XML data extraction was carried out. Table 9.7 highlights the tools used in the development or design of the case study systems where such information is available. It shows that a variety of CASE tools are utilised when producing artefacts. To answer the second question, the steps of *Consistency Management* were carried out on the *JGAP* system. These steps reveal whether using the framework involves changing the currently applied software methodology.

Results and Discussion

The ability to extract artefact data from any tool in the form of an XML document depends on the format of the files produced by these tools and the tools' export capabilities. The evaluation reveals that *.java* files obtained from the *JBuilder*[5], *Eclipse* and *NetBeans*[6] IDEs can be converted to an XML format using available tools. Additionally, *.dia* files produced by *Dia* can be exported to a *.vdx* format, which can be processed in the framework. Finally, the underlying XML contents of *.odt* files can be programmatically accessed. Performing change detection, change impact analysis, consistency checking and change propagation on the *JGAP* system, for example, does not result in changes in the user's current methodology.

9.4.2.2 Q2 - Automation

Q2 investigates the automation level of each framework stage. It is also assessed whether setting up *Artefact Data* is an automatic process. To accomplish these goals, the *JGAP* system was utilised and the same steps were performed as during correctness testing. This is due to the fact that framework stages and *Artefact Data* coincide with the main functional areas of the framework. In particular, automation levels pertaining to *Artefact Data* can be assessed by performing the *Framework Setup* tests. Lastly, *Consistency Management* tests can be used to investigate automation levels of the framework stages.

[5]www.embarcadero.com/products/jbuilder
[6]www.netbeans.org

Scenario	Level of automation
Setup Artefact Data	Automatic - assuming prerequisites are satisfied.
Create Trace Links	Semi-automatic using machine learning technique presented in Chapter 8.
Detect Changes	Automatic.
Analyse Change Impact	Automatic.
Check Consistency	Automatic.
Propagate Changes	Base implementation is automatic. Changes are applied manually.

Table 9.8: Automation level of Framework stages and Artefact Data setup.

Results and Discussion

The evaluation highlights that one of the main hindrances to automation is the diversity of representations. However, storing heterogeneous artefacts and their trace links in a uniform format in the framework allows the consistency management steps to be carried out in an automated manner. To facilitate setting up *Artefact Data* as automatically as possible, some prerequisites were defined. Provided that these conditions are satisfied, establishing *Artefact Data* is automatic. Specifically, following framework configuration, which involves the manual creation of the framework folder and the preparation of XSLT files, the extraction process is automatic in case of source code, unit test and requirement specification artefacts. However, UML diagrams are currently manually exported from their tools. Transformation is an automatic process. Automating trace link creation between heterogeneous artefacts remains the main challenge of this work. However, effort has been made to automate the process by utilising machine learning techniques. Although change detection is currently manually invoked by the user, the process is automatic. Additionally, configuring the framework to automatically pull changes from a repository at specified intervals can be implemented as part of future work. Change impact analysis and consistency checking are automatic steps. The base implementation of change propagation is automatic. However, carrying out changes is performed manually by the user at this stage.

Overall, as shown in Table 9.8, automating certain stages of the holistic framework is more straightforward, while others pose challenges. Additionally, manual and semi-automatic stages create obstacles in efforts to fully integrate each aspect in an automated manner.

9.4.2.3 Q3 - Artefact Independence

Artefact independence was evaluated primarily based on two qualities. Firstly, it was analysed whether the framework handles heterogeneous artefacts. Secondly, it was investigated whether the framework can be extended with additional artefacts. Specifically, the following questions were answered:

- Is it possible to extract data to an XML format from any artefact independent of its type?
- Is it possible to transform any XML representation to the custom GraphML format?

To answer the first question, five systems, *MazeSolver, JGAP, Neo4j, Titan,* and *JBBP*, were selected, which cover Java source code, Junit test, UML class diagram, module view architecture, conceptual architecture and UML use case diagram artefacts. To test if the framework can be extended with additional artefacts, the sixth system, *MyRobotLab*, was selected, as it provides a UML sequence diagram artefact, which previously had not been used with the framework. The evaluation was achieved by carrying out the steps of *Framework Setup* using each of the five open source systems. To test new artefacts, such as UML sequence diagrams provided by the *MyRobotLab* system, new XSLT transformation files were created and the same process was followed.

Results and Discussion

Any artefact providing an XML-based representation can be used with the framework, which is demonstrated by the fact that the artefacts mentioned above are successfully extracted. The level to which extraction can be automated may vary depending on the ease of access to the underlying XML data, which is discussed in the previous subsection. Another component required to achieve artefact independence is the availability of XSLT files to perform transformation to the uniform property graph format. An associated challenge is the complexity of the XML schema, which determines whether the required level of detail can be accessed. While data from the XML representation of Java source code and JUnit tests can easily be obtained, extracting relevant elements for example from UML sequence diagrams involves manual aspects.

Artefact independence also entails the ability to carry out consistency management tasks on any artefact. Following transformation and data being saved to the database, *Consistency Management* is performed on the property graph representation. Therefore, it can be stated that consistency management of heterogeneous artefacts can be carried out in the framework, which is independent of artefacts.

9.4.3 Methodology: Performance Tests

The performance evaluation was performed on a test environment characterised by properties given in Table 9.9. The methodology of carrying out performance tests in each test scenario is described in Table 9.10.

Operating system	Windows 8.1
System type	64-bit Operating System, x64-based processor
Processor	Intel ®Core ^{TM} i5-3210M CPU @ 2.50 GHz
Memory	6.00 GB
Hard disk	1 TB HDD
Java runtime	Version 1.8

Table 9.9: Test environment properties.

Results and Discussion

The tests can be found in the *framework.PerformanceTests* package in the *src* folder of the ACM framework. To reproduce the results of performance evaluation, users are required to setup the framework with artefact and trace link data. Once the framework is setup, consistency management can be carried out by initiating changes to the original artefacts in the repository. The remainder of this section describes the summary of results, while details are provided by the performance tables in Appendix A.

Scenario 1

Framework setup was firstly performed using *54* classes / interfaces / enums and *1* UML class diagram obtained from the *MazeSolver* system. At the graph database level, these artefacts equate to *846* nodes and *727* edges (intra relationships). Secondly, the *MyRobotlab* system provided *643* Java and JUnit classes / enums / interfaces and *1* sequence diagram. Using this data, *12502* nodes and *11794* edges were established in the database. Finally, the database was populated with *1326* nodes and *1175* edges obtained from the JBBP system from *76* Java classes / interfaces / enums, *1* use case diagram and *55* unit tests. *Framework setup* takes an average of *20.917* seconds using the *MazeSolver*, an average of *182.058* seconds using the *MyRobotLab*, and an average of *46.007* seconds using data from the *JBBP* system as illustrated by Figure 9.1.

Scenario 2

Besides change detection, impact analysis, consistency checking and change propagation, consistency management involves database and file system operations. This is reflected in the performance results. Specifically, editing the *attachGUI* and *detachGUI* methods of the *CalibratorGUI* class took 31.968 seconds, while deleting the *setRobotLocation* method of the *RobotBase* class was performed in 32.381 seconds. In the first example, the XML and GraphML representations are re-generated prior to database updates. In the second case, the XML and GraphML representations are deleted from the file system, following which corresponding nodes are labelled in the database. Additionally, change impact analysis also involves opening and

Test Scenario	Methodology
Scenario 1	Carry out Framework Setup on the *MazeSolver*, *MyRobotLab*, and *JBBP* systems. In all scenarios measure elapsed execution time using the *System.nanoTime()* method .
Scenario 2	Carry out Consistency Management on the *MazeSolver* system and on the *MyRobotLab* system. Applied changes: a, *MyRobotLab* system: edit the *attachGUI* and *detachGUI* methods of the *CalibratorGUI* class. b, *MazeSolver* system: delete the *setRobotLocation* method of the *RobotBase* class.
Scenario 3	Perform Framework Setup using the *Service* class of the *MyRobotLab* system, the *Owner* interface of the *Neo4j* system, and the *JBBPToken* class of the *JBBP* system.
Scenario 4	Perform Consistency Management using the *Service* class of the *MyRobotLab* system and the *Owner* interface from the *Neo4j* system. Applied changes: a, Edit the *getMethodToolTip* method of the Service class. b, Add method *newMethod* to the Owner interface.
Scenario 5	Run the change identification algorithm with the following inputs: a, *Service* class b, *Owner* interface. Applied changes: a, Edit signature of class / interface b, Delete existing method c, Add a new method.
Scenario 6	Perform Change Detection using the JGAP system. Changes: a, *Edit file level* changes taken from *Revision 1.24.* affecting the *BestChromosomesSelector* class: Edit *m_chromosomes* field and the *selectChromosomes* method. Delete the *returnsUniqueChromosomes* method. Add a new method. b, *Delete file level* change manually invoked: Delete the *AveragingCrossoverOperator* class. c, Add file level change manually invoked: Add the *BestChromosomesSelector* class.

Table 9.10: Performance tests - methodology.

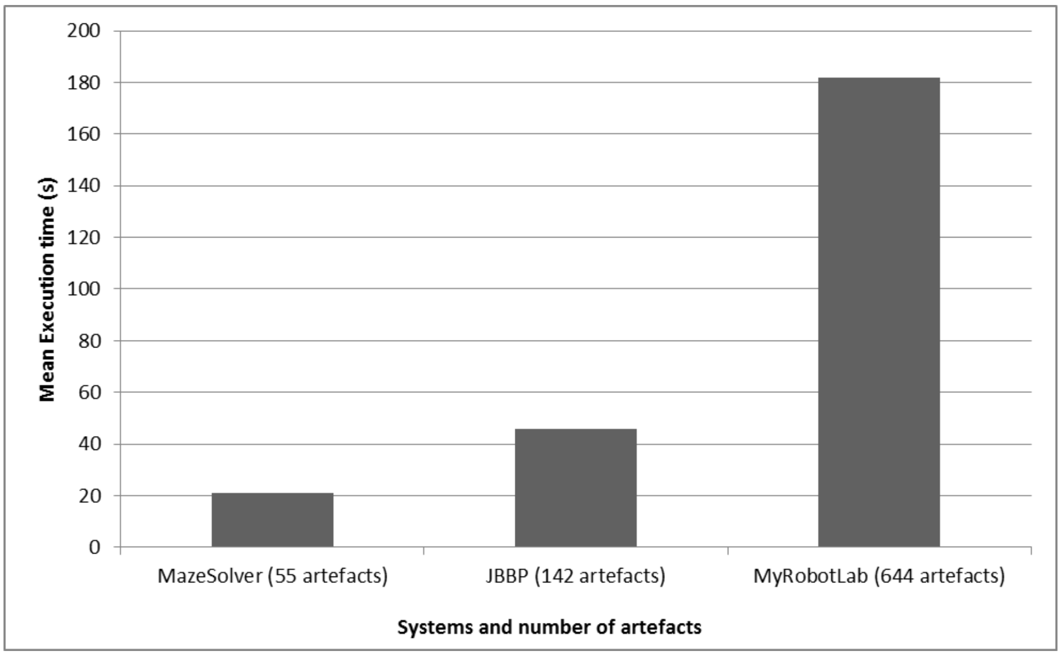


Figure 9.1: Framework Setup execution time (s) and system size.

closing the database connection while performing graph traversals. The duration of consistency management varies depending on the number of file level and graph database level operations. Further influencing factors include the size of artefacts, and the number of graph database nodes and edges, which determine the duration of traversals and file level operations.

Scenario 3 and **Scenario 4**

The results show that *Framework setup* took an average of *12.827* seconds using the *Service* class, an average of *5.5* seconds using the *Owner* class, and an average of *8.9171* seconds using the *JBBP* class, which is depicted by Figure 9.2. The execution times of consistency management carried out on the *Owner* interface (*30.449* seconds) and the *Service* class (*39.291*) show a similar trend seen in scenario 2. Consistency management, as mentioned in the previous section, involves additional operations and may also vary depending on the applied changes, as suggested by the results of scenarios 5 and 6.

Scenario 5

The change identification algorithm is realised in a number of steps: firstly, the input is converted to a nested hashmap representation, which is passed to the *compareMaps* method that performs the comparison of the inputs. Lastly, the *getChangeData* method returns the output in the form of a list of *ChangeData* objects. In this test scenario the execution time of all of these steps was measured. In assessing the performance of the change identification algorithm, two Java

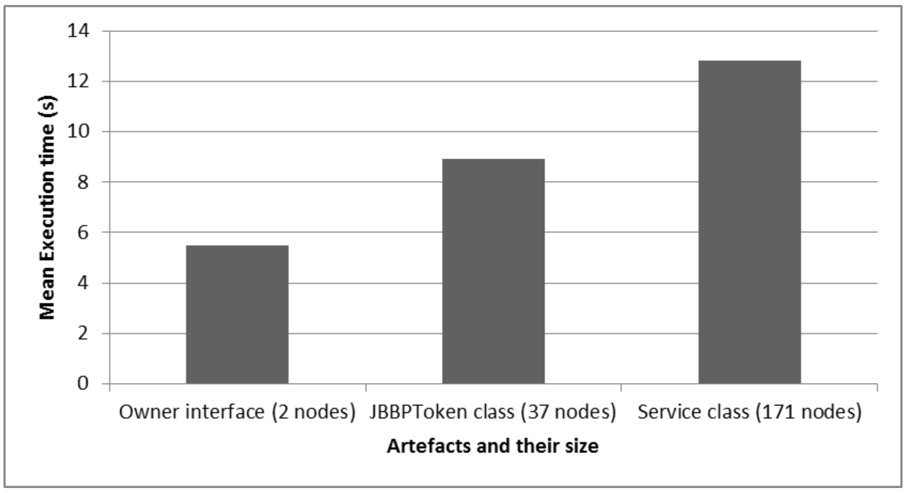


Figure 9.2: Framework Setup execution time (s) and artefact size.

container elements were used to reveal the correlation between artefact size and execution times, which is depicted by Figure 9.3. The results show that running the algorithm using the *Owner* interface, which contains *2* nodes, took *0.08* seconds on average. However, when running the algorithm in a significantly larger *GraphML* file, which contains *171* nodes, the average execution time was *5.6* seconds.

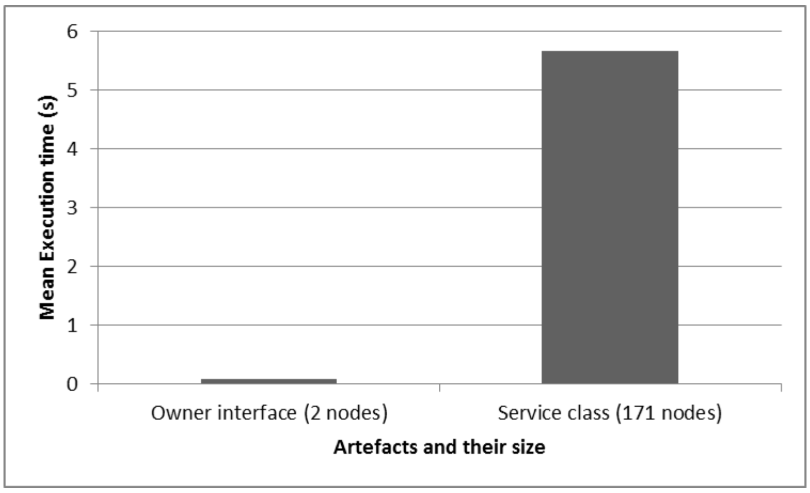


Figure 9.3: Change Detection execution time (s) and artefact size.

Scenario 6

Change detection entails different tasks depending on the *file level change* type as described in Chapter 6. All changes involve database updates, and file level operations such as generating new *GraphML* files or deleting existing ones. Additionally, *edit file level* changes also involve running the change identification algorithm. The different operations that take place in each individual case explain the differences in the execution times shown by results.

Performance results show that artefact size, the number of trace links and change type determine the duration of *Framework setup* and *Consistency management* tasks. Therefore, the framework may exhibit different performance values in specific projects. Artefact size and the complexity of trace links are factors external to the framework. However, performance may be improved by handling large *GraphML* files more effectively by reducing the number of save operations, and the number of times database connections are opened and closed, through connection pooling, for example. Finally, in case of large datasets, clustering Neo4j may provide further performance benefits.

9.4.4 Limitations and Threats to Validity

The evaluation of the ACM framework is characterised by the following limitations that may pose threats to the validity of the results.

Firstly, a threat to validity exists in that all aspects of the evaluation were carried out by one person. This includes selecting candidate systems, data collection and establishing trace links between heterogeneous artefacts obtained from open source systems. Thus, it cannot be guaranteed that all relevant links from the selected subset of the systems have been identified. This affects the validity of results due to the framework stages being dependent on trace links.

Secondly, some limitations of the evaluation arise due to characteristics of open source systems used and artefacts provided by these systems. Each open source system provides different artefacts and is characterised by different levels of complexity, which can be expressed through the number of inter trace links and dependencies, and the number of artefacts. This raises the question whether evaluating the ACM framework on the selected systems and case studies allows the generalisation of findings to any other systems and cases. Therefore, additional systems and artefacts are required to broaden this work to further projects.

Finally, the validity of results is also affected by the selected changesets and manual changes applied during evaluation. It raises the question whether these changes are representative of the most common changes in real project scenarios and how results can be generalised to other systems. Particularly, in case of change impact analysis, some selected changes may provide different results from others in terms of precision and recall. In some cases there may be larger differences between the EIS and AIS.

9.5 Conclusions

The evaluation presented in this chapter shows that the ACM framework is a viable option to manage the consistency of heterogeneous artefacts in a holistic manner. It supports artefacts created using a variety of tools and it is possible to automate framework stages provided specific pre-requisites are satisfied. The framework can be extended to handle additional artefacts. Potential issues and areas of improvement are highlighted in each corresponding section and are further discussed in Chapter 10.

The evaluation was guided by the principles of transparency and reproducibility to allow the tests to be duplicated. This is supported by the availability of the source code and tests.

CHAPTER TEN

CONCLUSIONS

10.1 Summary

Since the evolution of modern software systems is inevitable, artefacts representing the system go through frequent refinements. Consequently, the different representations may become inconsistent with one another. The differential evolution of artefacts may hinder effective software maintenance and stakeholders may develop a lack of trust in them as they do not accurately represent the system. Frequent modifications characterising incremental and agile development may aggravate this problem.

Various solutions have emerged in requirements engineering, traceability, change management and impact analysis research to establish links between artefacts, to assess the impact of modifications and to provide mechanisms to control changes in software development projects. A survey of these approaches leads to the conclusion that the problem remains to be addressed. Based on the findings of the survey, the requirements of an ideal software artefact consistency management solution were derived. This thesis proposes a holistic consistency management approach and investigates its feasibility. The concept is realised in the design and implementation of the ACM framework, which aims to fulfil a subset of the proposed requirements of an ideal consistency management solution discussed in Chapter 4. It combines traceability, change detection, change impact analysis, consistency checking and change propagation. Finally, the thesis provides an evaluation of the framework, which shows promising results in supporting artefact consistency management tasks.

The next section presents an overall assessment of the holistic approach and the ACM framework.

10.2 Assessment and Limitations

First, an assessment is provided in light of evaluation results and based on the requirements presented in Chapter 4. Next, limitations of the work are discussed. These are divided into three main categories: limitations 1) of the approach, 2) of the current implementation, and 3) attributed to external factors.

10.2.1 Assessment in the Context of Requirements

R1 - Automated as far as possible.
A critical element of any consistency management solution ready for wider adoption is automation. The implementation of each framework stage strives to provide automation as far as possible. This work demonstrates that some stages are easier to automate than others. Particularly challenging areas include establishing and maintaining trace links, which is a well-recognised problem in traceability research. However, with trace links in place, further framework stages can be successfully automated.

Therefore, all stages of consistency management are automated to varying degrees. Trace link creation is supported by machine learning techniques, which has yielded a semi-automated solution. Change detection, impact analysis, consistency checking are fully automated. On the other hand, besides providing suggestions, actual change propagation and applying changes to inconsistent artefacts remains a manual task. Additionally, the consistency management process currently requires user intervention at certain points, such as when initiating change detection.

Although full automation remains a challenge to be addressed, the ACM framework has significantly contributed to achieving this aim.

R2 - Artefact Independent and R3 - Methodology and Tool Independent.
Since both requirements relate to extensibility and the framework's ability to handle different representations authored in different tools, they are discussed together. Evaluation results show that the framework is capable of handling heterogeneous software artefacts and it does not require users to change their methodologies and tools, assuming the prerequisite to generate an XML-based representation of the original artefact is met.

R4 - Customisable and Non-Intrusive.
The framework currently allows users to configure the framework during the setup stage to specify the database backend, the external repository and XSLT files used for transformation. However, there is a wide scope for further work in this area as other stages of the framework are also configurable. These are discussed in future work.

R5 - Supports Distributed Development.
Distributed software development is a key aspect of modern software projects. The ACM framework presently contributes to achieving this aim by connecting to external repositories, which are used in distributed settings. However, in the scope of the current implementation, explicit support for distributed development remains an area for future work.

R6 - Is able to handle variable numbers of artefacts and changes (Performance).
A pivotal aspect of the adoptability of the framework is its performance. The evaluation shows that the ACM framework is able to handle artefacts of different sizes and changes of different types. However, further work is required to fully assess the level to which this requirement is satisfied. Specifically, in case of large systems with a large number of trace links, expert knowledge is required to establish links. Providing trace link data at this level of detail was not possible in the scope of the current work.

10.2.2 Limitations

- **Traceability.** The main limitation of the approach is that the correctness of each consistency management stage is heavily reliant upon the correctness of trace links. The dependence on trace links raises the question of how framework stages could be more tolerant to errors introduced during trace creation. This is a significant issue considering that the current approach to creating trace links using machine learning, by nature, is not likely to provide 100% accuracy. Thus, user intervention is required to ensure correct links are established prior to consistency management.
- **Granularity of artefact elements.** One of the limitations of the current implementation arises from the level of granularity of artefact elements. Each artefact type is different in terms of the level of detail they capture and this is reflected in their property graph representation in the framework. Specifically, in case of Java source code and JUnit test artefacts, although the contents of methods are captured, the method body is considered as an atomic entity. Naturally, this limits change detection, change impact analyis and consistency checking and provides a more coarse-grained approach. This limitation can be addressed by implementation level changes and additional consistency rules need to be added to the rule base capturing the required level of detail.
- **Extraction and transformation.** A pre-requisite of the *artefact independence* requirement is the availability of an XML-based representation of artefacts, which is an external limitation. The XML format ensures that through XSLT transformations a GraphML representation of the original artefact can be produced. Each authoring tool provides different XML representations and in some cases some elements and *intra* trace links cannot be automatically extracted.

For example the XML schema of UML class diagrams extracted from Dia does not allow capturing the connections between UML classes that are on the same diagram. In such cases the automatically generated *.graphml* file is manually edited to add the required trace links.

- **Change detection.** Change detection has presented the problem of identifying an entity across two versions of the same artefact, which exists regardless of change and artefact representation, and is discussed in Chapter 6. The current solution applies domain knowledge to identify entities, which can be extended to the artefacts supported by the framework. However, an artefact-independent approach may provide a more robust solution.
- **Impact analysis.** In the current implementation when a new artefact element is added without a trace link connecting it to existing elements, no impact analysis and consistency checking take place. In this particular case the problem falls under the scope of trace link creation. The current workaround to this issue involves flagging up the change to the user in the change propagation stage. However, a trace maintenance mechanism is required to handle such cases in the current or the next consistency management iteration.
- **Consistency checking.** As mentioned in Chapter 9, the current consistency checking approach analyses each change in isolation. Thus, in cases where there are connections between modifications they are not captured and inconsistencies may be flagged up even if they are resolved by other changes. This issue may potentially be resolved by considering a changeset as a whole.

10.3 Future Work

Due to the complexity and multi-faceted nature of the research problem, there is a wide scope for further work and extending the framework. Each framework stage is a research area on its own and each of these fields may benefit from extending relevant aspects of the framework. This research can be continued in different directions. Future work can be based on the challenges introduced in Chapter 4, as well as on optimising and extending the framework, and further automating the stages of consistency management.

Artefact Data representation

The property graph-based representation has proven to be a flexible approach to model software artefacts and their connections. Graph traversals are utilised in the change impact analysis process. Open questions include whether the graph representation can be further utilised to support other consistency management tasks, such as consistency checking, and whether further optimisations or automation can be enabled based on the structure of the property graph.

Framework Stages

- **Trace link Creation and Maintenance.** In the current implementation machine learning techniques are applied in an effort to automate trace link creation. The approach seems promising based on the preliminary cross-validation results. One potential avenue for further exploration is extending machine learning techniques to the overall framework to contribute to further automating the stages or improving their results. In particular, trace maintenance, which is currently rule-based, could be supported by machine learning.
- **Change Detection.** An important avenue of future work is the identification of entities across two versions of the same artefact, mentioned in Section 10.2. Additionally, since change detection is manually invoked by the user, a useful enhancement would be to provide configuration options for pulling change data from an external repository at specified intervals. Adapting to the frequency of changes may present an interesting extension to current functionality. Finally, the entire consistency management process would benefit from a finer-grained change detection approach. For example, in the current implementation changes to the contents of Java methods or JUnit field values are not captured, which limits the granularity of further consistency management stages.
- **Impact Analysis.** The impact analysis process of the framework currently considers only direct dependencies. In order to minimise the number of false positives and false negatives in the set of potentially impacted artefacts, indirect dependencies should also be included. Techniques currently applied mostly in conjunction with source code artefacts to discover logical dependencies [42] present another avenue for investigation. Further work may also include exploring whether it is possible adopt historical co-change analysis to be used with heterogeneous artefacts. Finally, source code impact analysis utilises probabilistic methods to improve the accuracy of change impact analysis results, which is also worth investigating in the context of heterogeneous representations.
- **Consistency Checking.** The present consistency checking approach is based on pre-defined consistency rules, which are stored in an XML rule base. Each scenario, defined by a changed element and a potentially affected element, is mapped to a corresponding rule. Further work may explore another approach where hierarchical relationships may reduce the number of explicit rules. Another potential direction may be associating probability values with identified inconsistencies, which may provide further benefits to consistency checking.

  Finally, the current implementation assumes that the initial state is a consistent one. It does not flag up inconsistencies between artefacts that are saved to the framework folder and the graph database. Instead it checks for inconsistencies as a result of a change introduced to one of the artefacts. A potential enhancement may be decoupling consistency checking from the rest of the framework stages to ensure that consistency management always starts from a consistent state.

- **Change Propagation.** Change propagation lends itself to a substantial amount of further work. Propagating and applying a change to an inconsistent artefact is a non-trivial task, especially when the heterogeneity of artefacts is considered. In an ideal scenario the task is carried out in an automatic manner. Automation may be feasible in cases, such as when propagating changes from source code to UML diagrams. However, in other scenarios, user intervention may be required. A more immediate enhancement to the current implementation is the functionality to pinpoint which original artefact is inconsistent. This is currently expressed using graph nodes in the database, which the user can locate based on a unique id.

Usability and Visualisation

The ACM framework presently does not provide a user interface. Thus, users are required to be familiar with invoking framework functionality and some internal implementation details. To transition the framework from a proof-of-concept implementation to a tool ready for wider adoption, a user interface is a high priority enhancement. Besides abstracting implementation level details, a further element of providing a user interface is managing user interactions, such as displaying appropriate messages when errors occur, and setting preferences.

Providing a user interface also allows framework data to be visualised. Viewing trace links between heterogeneous entities may help users in maintaining and understanding their systems. Visualising a selected set of impact nodes, changed nodes and inconsistencies aids impact analysis and consistency checking and assists in the maintenance of the system.

Customisation capabilities are also closely associated with providing a user interface and usability. The ability to configure the framework would provide a greater degree of flexibility and would contribute to fulfilling the requirement of offering a non-intrusive solution. With a user interface in place users may specify which artefacts are relevant to them, the intervals at which consistency management is initiated, and the level of detail they would like to view in each framework stage.

Visualising the results of consistency checking may present an approach to automating change propagation. For example, inconsistent artefact elements could be accessed through the user interface and users could apply the required change.

Work has been done at the University of St Andrews in the form of a Master's project to provide visualisation of artefact and trace link data [222]. The integration of this functionality in the framework provides a starting point to visualisation efforts.

Distributed Software Development

Due to the ubiquity of distributed software development, for the framework to be useful in a large number of projects, future work is required to investigate how a consistency management solution can function in such a setting and how specific requirements arising from distributed development can be met. In this respect, areas of concern include multiple versions of artefacts, creating and maintaining links among distributed artefacts, and the availability and accessibility of the latest version of any given artefact.

Evaluation of Applicability

Finally, the framework lends itself to further work in evaluating its applicability to explore the effort involved in using the framework on a project of realistic complexity, and assessing the benefits it provides to users in consistency management tasks. Such an analysis, similarly to the evaluation that was carried out in the scope of this work, may utilise a case study approach. A key aspect of the evaluation is the identification of an industrial project that provides multiple artefacts produced by different stakeholders using different tools, which also constitutes the main challenge involved in such an evaluation due to the accessibility of such information. Following the deployment of the framework, users could assess its usefulness while performing their tasks during the development process, the effort involved in using it and its ease of use. While the study is pre-dominantly qualitative, a quantitative analysis of artefact numbers, artefact types, project size would provide further insights into the applicability of the framework.

10.4 Concluding Remarks

This thesis has presented a novel holistic approach to support managing the consistency of heterogeneous software artefacts and demonstrated that it is a viable and promising solution to an open problem.

The ACM framework may provide benefits to various stakeholders during both software development and maintenance. The challenges software changes pose, in the context of diverse teams, tools, and artefacts, may be alleviated by performing consistency management. The framework provides support for change detection, impact analysis, consistency checking and change propagation to reduce manual effort while allowing users to perform development, analysis and testing tasks following their normal processes. Stakeholders may also benefit from utilising specific framework stages, for example to assist them with analysing the impact of changes to other representations. Overall, the framework reduces maintenance efforts, and contributes to better comprehension and sustainability of a software system.

The ACM framework, which is available in an online repository, can be extended with any artefacts and is thus applicable in different software development scenarios. Due to the complexity of the problem, much work remains to be done, presenting interesting challenges and new directions in traceability, change impact analysis and consistency checking research, particularly in the context of heterogeneous software artefacts.

APPENDIX - A PERFORMANCE RESULTS

Test Scenario 1: MazeSolver, MyRobotlab & JBBP			
Run	**Execution time(s) (MazeSolver)**	**Execution time(s) (MyRobotlab)**	**Execution time(s) (JBBP)**
1	32.3	211.797	55.703
2	19.061	164.071	44.25
3	18.860	171.234	43.005
4	19.18	178.374	45.389
5	21.528	180.487	46.847
6	19.510	161.096	52.208
7	19.671	198.66	44.669
8	18.797	165.785	43.029
9	21.048	205.497	39.937
10	19.218	183.584	45.039
Mean:	**20.917**	**182.058**	**46.007**

Table A.1: Framework Setup execution times using the MazeSolver, MyRobotlab and JBBP systems.

Test Scenario 3: *Service* class, *Owner* interface, *JBBPToken* class			
Run	**Execution time(s) Service class**	**Execution time(s) Owner interface**	**Execution time(s) JBBPToken class**
1	13.838	5.075	9.816
2	12.427	5.292	8.698
3	12.604	5.609	8.873
4	12.833	5.653	9.238
5	13.180	5.366	8.842
6	12.678	5.957	8.315
7	12.605	5.147	8.528
8	13.064	5.679	8.775
9	12.565	5.66	9.081
10	12.482	5.568	9.005
Mean:	**12.827**	**5.5**	**8.9171**

Table A.2: Framework Setup execution times using the Service class (MyRobotLab), the Owner interface (Neo4j), and the JBBPToken class (JBBP).

Test Scenario 5: Owner interface			
Run	**Execution time (s) Edit interface signature**	**Execution time(s) Add method**	**Execution time (s) Delete method**
1	0.084	0.094	0.089
2	0.084	0.1	0.085
3	0.091	0.086	0.09
4	0.084	0.115	0.082
5	0.083	0.082	0.084
6	0.084	0.087	0.077
7	0.082	0.92	0.087
8	0.085	0.076	0.081
9	0.081	0.07	0.078
10	0.068	0.086	0.076
Mean:	**0.082**	**0.088**	**0.082**

Table A.3: Change identification algorithm execution times using the Owner interface of the Neo4j system.

Test Scenario 5: Service class			
Run	**Execution time (s) Edit class signature**	**Execution time(s) Add method**	**Execution time (s) Delete method**
1	6.095	5.75	5.64
2	5.86	5.46	5.47
3	5.89	5.44	5.72
4	5.79	5.46	5.62
5	5.74	5.75	5.76
6	6.02	5.38	5.89
7	5.71	5.53	5.65
8	5.48	5.63	5.68
9	5.68	5.35	5.85
10	5.59	5.59	5.65
Mean:	**5.78**	**5.53**	**5.69**

Table A.4: Change identification algorithm execution times using the Service class of the MyRobotlab system.

Test Scenario 6: Change Detection					
	Add File Level Change	**Delete File Level Change**		**Edit File Level Change**	
			Edit Artefact Element Level Change	**Add Artefact Element Level Change**	**Delete Artefact Element Level Change**
Run					
1	11,71	24.751	14.055	13.273	15.967
2	10.744	22.098	17.049	14.144	14.354
3	10.777	22.242	13.943	12.985	19.067
4	10.325	22	14.054	13.291	15.112
5	10.44	22.273	13.709	14.455	15.888
6	10.436	21.697	13.679	14.306	15.126
7	10.307	22.013	14.593	14.884	18.387
8	10.326	22.931	13.858	14.33	14.795
9	10.314	22.562	14.001	15.027	15.635
10	10.238	21.961	13.679	14.406	14.904
Mean:	**10.561**	**22.452**	**14.262**	**14.11**	**15.923**

Table A.5: Change detection execution times using the JGAP system.

APPENDIX B - SUMMARY TABLES

Solutions	Traceability Creation	Traceability Maintenance	Change Detection	Impact Analysis	Consistency Checking	Change Propagation	Supported Artefacts	Supports Distributed Dev.	Artefact Representation	Link Storage & Representation
MolhadoArch	Not provided by the tool. Manual or automatic processes can be incorporated.	Automatic	Versioning and differencing are provided.	Not covered	Automatic	Not covered	Specific artefacts: architecture and source code	-	Document tree	Architectural relationship graph
Ophelia	Semi-automatic; relationhips can be defined in the TracePlough prototype	Not covered	Not covered	Not covered	Not covered	Not covered	Heterogeneous artefacts: specifically documentation and code, code and test cases	Yes	CORBA objects	Graph
Sysiphus	Semi-automatic; implicit dependencies can also be identified	Not covered	Automatic through a notification system	Automatic	Not covered	Not covered	Heterogeneous artefacts: problem statements, requirements, architecture, detailed design, test cases	Yes	Graph-based	Not specified
Roundtrip engineering solutions: Microsoft Visual Studio	NA	NA	Automatic	Automatic	Automatic	Automatic	Specific artefacts: design diagrams and source code	Yes	Na	NA
Literate programming	NA	NA	Manual	Manual	Manual	Manual	Specific artefacts: documentation and C code	-	Original	NA
Intent	Automatic by combining representations	Automatic by combining representations	Automatic	Automatic	Automatic based on constraints	Manual	Specific artefacts: documentation and technical artefacts	-	Original	Central repository

Table B.1: Comparison summary, Part I. The reviewed solutions are listed by their names (where the name of the implemented system or project title is in place) or by the authors' names.

Solutions	Traceability Creation	Traceability Maintenance	Change Detection	Impact Analysis	Consistency Checking	Change Propagation	Supported Artefacts	Supports Distributed Dev.	Artefact Representation	Link Storage & Representation
Aper	Links are present; it is not specified whether their creation is automatic	Not covered	Automatic through a trigger process	Automatic	Automatic	Manual: users are notified	Heterogeneous artefacts: not specified	-	Software products are decomposed	Not specified
Olsson & Grundy	Manual	Not covered	Semi-automatic: change data can be imported to the tool	Automatic	Automatic	Manual; some changes can be automatically applied	Specific artefacts: requirements, use case models, black-box test plans	-	Abstracted representation model capturing key information from original artefacts	Not specified
ArchEvol	Automatic by integrating existing tools	Automatic by integrating existing tools	Manual	Manual	Manual	Manual	Specific artefacts: architecture and source code	-	Mappings are defined between architecture and source code elements; original artefacts are stored in a WebDAV repository	Not specified
Clime	Manual, links are defined through meta-constraints	-	Automatic using the Activity monitor	Not covered	Automatic using constraints	Manual	Heterogeneous artefacts: prototype works with Java source code, UML class diagrams	-	Abstract information from original artefacts (common framework, not common representation)	Relations in a database
Software Concordance	Semi-automatic	Manual through versioned hypermedia	Automatic	Not covered	Semi-automatic	Manual	Specific artefacts: Java source code and multimedia documentation	Through collaborative documentation process	Fluid Internal Representation	Hyperlinks
Maletic et al.	Semi-automatic, based on LSI	Automatic through conformance analysis	Not covered	Not covered	Not covered	Not covered	Specific artefacts: source code and documentation	-	Hypertext	Hypertext

Table B.2: Comparison summary, Part II. The reviewed solutions are listed by their names (where the name of the implemented system or project title is in place) or by the authors' names.

Solutions	Traceability Creation	Traceability Maintenance	Change Detection	Impact Analysis	Consistency Checking	Change Propagation	Supported Artefacts	Supports Distributed Dev.	Artefact Representation	Link Storage & Representation
ACTS	Automatic	Semi-automatic through notification adapters	Automatic through recording adapters	Not covered	Not covered	Not covered	Heterogeneous artefacts, differentiation between primary (architecture) and other artefacts	Yes	Original artefacts; tool-specific adapters are utilised	XML-based
Asuncion et al.	Manual	Manual, updates are side affects of trace utilisation	Not covered	Manual, bidirectional updates between documents and artefacts using data entry forms	Not covered	Not covered	Heterogeneous artefacts; specifically features, use cases, functional requirements	-	Document representation stored in artefact repository (MS SQL)	Trace repository
ADAMS	Automatic	Trace link versioning provided	Automatic	Semi-automatic through event notification	Not covered	Manual	Specific artefacts: UML diagrams and textual documentation	Yes	Metadata about original artefacts is stored in a relational database	Not specified
OSCAR	Supported, automation level not specified	-	Not covered	Not covered	Not covered	Not covered	Heterogeneous artefacts; no specific artefacts mentioned in evaluation of the working system	Yes	XML-based	XML-based
EBT	Supported, automation level not specified	Automatic	Automatic through change events - integrated with DOORS	Automatic	Not covered	Manual	Specific artefacts: requirements, class and sequence diagrams, Java code, test cases	Yes	Original	Event server

Table B.3: Comparison summary, Part III. The reviewed solutions are listed by their names (where the name of the implemented system or project title is in place) or by the authors' names.

Solutions	Traceability Creation	Traceability Maintenance	Change Detection	Impact Analysis	Consistency Checking	Change Propagation	Supported Artefacts	Supports Distributed Dev.	Artefact Representation	Link Storage & Representation
ArchJava	Automatic by integrating the two representations	Automatic by integrating the two representations	Automatic by integrating the two representations	Automatic by integrating the two representations	Automatic	NA	Specific artefacts: architecture and source code	-	Implicit mappings	NA
traceMaintainer	-	Semi-automatic, rule-based	Not covered	Not covered	Not covered	Not covered	Specific artefacts: structural UML models	-	Original artefacts	Traceability relation repository
ArchTrace	Manual	Automatic, policy-based	Automatic	Manual with visualisation support	Not covered	Not covered	Specific artefacts: source code, architecture	Yes	Original - architecture represented by xADL, source code stored in SubVersion	Stored in architecture description or CM system
EMFTrace	Semi-automatic, Rule-based	Semi-automatic	Based on change taxonomy	Automatic, Rule-based	Not covered	Not covered	Heterogeneous artefacts; specifically: UML models, Java source code , Junit test cases	-	EMF-based models stored in EMFStore	XML-based
iACMTool	Intra-artefact traceability	-	Automatic	Automatic, Rule-based	Automatic, Rule-based (using OCL)	-	Specific artefact: UML model	-	Original	Not specified

Table B.4: Comparison summary, Part IV. The reviewed solutions are listed by their names (where the name of the implemented system or project title is in place) or by the authors' names.

Artefact	Structural Elements	Properties Derived	Artefact Elements	Graph Nodes
Requirement spec.	Executive summary, Product Description, Functional Requirements, Non-Functional Requirements, Recommendations, References, Appendices	Common: name, type descriptor, contents, title, priority	Requirement	A graph node represents a single Requirement
UML class d.	Class / Interface / Enum, Attribute, Operation	Common: name, type descriptor, modifier, Attribute specific: attribute type, Operation specific: input parameters, return type	Class / Interface / Enum, Attribute, Operation	A graph node represents a single Class / Interface / Enum, Attribute, or Operation
Java s. c.	Package statement, Import statements, Interface / Class / Method / Enum / Variable declaration, Comments, Annotations, Other language constructs within methods	Common: name, type descriptor, modifier, Field specific: field type, Method specific: input parameters, return types, throws clause, method body, Class / Interface / Enum specific: extends, implements clause	Interface / Class / Method / Enum / Variable declaration	A graph node represents a single Interface / Class / Enum, Method, or Variable declaration
JUnit	Artefact elements and graph nodess derived are identical to those of Java source code			
Module view arch.	Module	Common: name, type	Module	A graph node represents a single Module
Conceptual arch.	Component, Connector	Common: name, type	Component	A graph node represents a single Component
UML use case d.	Use case, Actor	Common: name, type	Use case	A graph node represents a single Use case
UML sequence d.	Frame, Lifeline, TimeConstraint, Message, etc.	Specific to Lifeline and Message elements: name, type	Object/Use case/ Class represented by lifeline	A graph node represents a single Lifeline

Table B.5: Summary of artefacts handled by the framework and the derivation of property graph nodes and their attributes from structural elements.

Artefact	Delete and Add file level changes and examples
Java source code	A .java file is deleted from the repository or added to the repository. The .java file contains a single class and its members. However, in some cases the .java file may contain nested classes.
UML class diagram	A .dia file is deleted from the repository or added to the repository. The .dia file represents a UML class diagram that contains multiple container and member elements.
JUnit test	A .java file is deleted from the repository or added to the repository. The .java file contains a single test class and its members.
UML sequence diagram	A .dia file is deleted from the repository. The .dia file represents a UML sequence diagram with multiple lifelines.
Requirement specification	A .odt file is deleted from the repository or added to the repository. The .odt file contains multiple requirements.
UML Use case diagram	A .dia file is deleted from the repository or added to the repository. The .dia file represents a UML use case diagram with multiple use cases.
Conceptual Architecture diagram	A .dia file representing a conceptual architecture diagram is deleted or added to the repository.
Module view architecture diagram	A .dia file representing a module view architecture diagram is deleted or added to the repository.

Table B.6: *Delete* and *Add File Level* changes - specific examples.

Type of artefact	**Change**
Requirement spec.	Edit requirement name / priority / description
Requirement spec.	Delete requirement
Requirement spec.	Add requirement
UML class diagram	Add class / interface / enum / operation / attribute
UML class diagram	Edit class / interface / enum / attribute / operation signature / operation return type
UML class diagram	Delete class / interface / enum / attribute / operation
Java s.c. / JUnit tests	Add class / interface / enum / method / field
Java s.c. / JUnit tests	Edit class / interface / enum / field / method signature / method return type / method body
Java s.c. / JUnit tests	Delete class / interface / enum / field / method

Table B.7: *Artefact element level* changes.

Changed Entity	Inter Trace Link Type	Connected Entity	Inter Consistency Rule
Artefact type:Java source code, Artefact element type: Java class, Fine-grained element type: Container element, Example scenario: delete a single or multiple Java classes	Identity	UML/JUnit container element, UML sequence diagram lifeline	Connected entity is inconsistent.
	Satisfaction	Requirement, use case	Connected entity is potentially inconsistent.
	Satisfaction	Architectural component, architectural module	Connected entity is potentially inconsistent.
Artefact type:UML class diagram Delete entire UML class diagram	Identity	Java/JUnit container element, UML sequence diagram lifeline	Connected entity is inconsistent.
	Satisfaction	Requirement, use case	Connected entity is potentially inconsistent.
	Satisfaction	Architectural component, architectural module	Connected entity is potentially inconsistent.
Artefact type: JUnit test, Artefact element type: JUnit test class, Fine-grained element type: Container element, Example scenario: Delete a single or multiple test files	Identity	UML/Java container element, UML sequence diagram lifeline	Connected entity is inconsistent.
	Satisfaction	Requirement, use case	Connected entity is potentially inconsistent.
	Satisfaction	Architectural component, architectural module	Connected entity is potentially inconsistent.

Table B.8: *Delete file level* change - Derivation of consistency rules based on the modified entity, the connected entity, and *inter* link type. Part I.

Changed Entity	Inter Trace Link Type	Connected Entity	Inter Consistency Rule
Artefact type: UML sequence diagram, Delete an entire UML sequence diagram	Identity	UML/JUnit/Java container element	Connected entity is inconsistent.
	Satisfaction	Requirement, use case	Connected entity is potentially inconsistent.
	Satisfaction	Architectural component, architectural module	Connected entity is potentially inconsistent.
Artefact type: Requirement specification, Delete an entire requirement specification document.	Satisfaction	UML/JUnit/Java container element, UML sequence diagram lifeline	Connected entity is inconsistent.
	Identity	Use case	Connected entity is inconsistent.
	Satisfaction	Architectural component, architectural module	Connected entity is inconsistent.
Artefact type: Use case diagram, Delete an entire use case document.	Satisfaction	UML/JUnit/Java container element, UML sequence diagram lifeline	Connected entity is inconsistent.
	Identity	Requirement	Connected entity is inconsistent.
	Satisfaction	Architectural component, architectural module	Connected entity is inconsistent.

Table B.9: *Delete file level* change - Derivation of consistency rules based on the modified entity, the connected entity, and *inter* link type. Part II.

Changed Entity	Inter Trace Link Type	Connected Entity	Inter Consistency Rule
Artefact type: Conceptual architecture, Delete an entire architecture diagram.	Satisfaction	UML/JUnit/Java container element, UML sequence diagram lifeline	Connected entity is inconsistent.
	Satisfaction	Use case, Requirement	Connected entity is inconsistent.
	Identity	Architectural module	Connected entity is inconsistent.
Artefact type: Module view architecture, Delete an entire architecture diagram.	Satisfaction	UML/JUnit/Java container element, UML sequence diagram lifeline	Connected entity is inconsistent.
	Satisfaction	Use case, Requirement	Connected entity is potentially inconsistent.
	Identity	Architectural component	Connected entity is potentially inconsistent.

Table B.10: *Delete file level* change - Derivation of consistency rules based on the modified entity, the connected entity, and *inter* link type. Part III.

Deleted Entity	Connected Entity	Inter Trace Link Type	Inter Rules
Java member element	UML/JUnit member	Identity	Connected entity is inconsistent.
UML member element	Java/JUnit member	Identity	Connected entity is inconsistent.
UML container	Java/JUnit container / UML sequence diagram lifeline	Identity	Connected entity is inconsistent.
UML container	Requirement, Use case, Architectural component, Architectural module	Satisfaction	Connected entity is potentially inconsistent.
JUnit member element	UML/Java member	Identity	Connected entity is inconsistent.
UML sequence diagram lifeline	UML/JUnit/Java container	Identity	Connected entity is inconsistent.
	Requirement, Use case, Architectural component, Architectural module	Satisfaction	Connected entity is potentially inconsistent.
Requirement	UML/JUnit / Java container /UML sequence diagram lifeline / Architectural component / Architectural module	Satisfaction	Connected entity is inconsistent.
	Use case	Identity	Connected entity is inconsistent.
Use case	UML/JUnit / Java container /UML sequence diagram lifeline / Architectural component / Architectural module	Satisfaction	Connected entity is inconsistent.
	Requirement	Identity	Connected entity is inconsistent.
Architectural component	UML/JUnit / Java container /UML sequence diagram lifeline	Satisfaction	Connected entity is inconsistent.
	Architectural module	Identity	Connected entity is inconsistent.
	Requirement / Use case	Satisfaction	Connected entity is potentially inconsistent.
Architectural module	UML/JUnit / Java container /UML sequence diagram lifeline	Satisfaction	Connected entity is inconsistent.
	Architectural component	Identity	Connected entity is inconsistent.
	Requirement / Use case	Satisfaction	Connected entity is potentially inconsistent.

Table B.11: *Edit file level* change - *Delete artefact element level* change - Derivation of inter rules.

Added Entity	Entity Connected to Parent	Parent Inter Trace Link	Inter Rule
Java member element	UML/JUnit container	Identity	Connected entity is inconsistent.
UML member element	Java/JUnit container	Identity	Connected entity is inconsistent.
JUnit member element	UML/Java container	Identity	Connected entity is potentially inconsistent.
UML sequence diagram lifeline, requirement, use case, architecture component, architecture module	No connected entity as no parents exist	NA	No consistency rule can be established.

Table B.12: *Edit file level* change - *Add artefact element level* change - Derivation of inter rules.

Edited Entity	**Connected Entity**	**Inter Trace Link Type**	**Inter Rules**
Java source code member / container element, Change type: signature change	UML/JUnit member/container	Identity	Connected entity is inconsistent.
	Requirement / UML use case / architectural component / architectural module	Satisfaction	Connected entity is consistent.
	UML sequence diagram lifeline	Identity	Connected entity is consistent.
Java source code member element, Change type: content change	UML member	Identity	Connected entity is consistent.
	JUnit member	Identity	Connected entity is potentially inconsistent.
	Requirement / UML use case / architectural component / architectural module	Satisfaction	Connected entity is consistent.
	UML sequence diagram lifeline	Identity	Connected entity is consistent.
UML member /container element, Change type: signature change	Java source code / JUnit member/container	Identity	Connected entity is inconsistent.
	Requirement / UML use case / architectural component / architectural module	Satisfaction	Connected entity is consistent.
	UML sequence diagram lifeline	Identity	Connected entity is consistent.
JUnit member /container element, Change type: signature / content change	Java source code / UML class diagram,member / container / UML sequence diagram lifeline	Identity	Connected entity is consistent.
	Requirement / UML use case / architectural component / architectural module	Satisfaction	Connected entity is consistent.
UML sequence diagram lifeline	Java source code / UML class diagram / JUnit container	Identity	Connected entity is consistent.
	Requirement / UML use case / architectural component / architectural module	Satisfaction	Connected entity is consistent.

Table B.13: *Edit file level* change - *Edit artefact element level* change - Derivation of inter rules part I.

Edited Entity	Connected Entity	Inter Trace Link Type	Inter Rules
Requirement	Java source code / UML class diagram / JUnit,container / UML sequence diagram lifeline	Satisfaction	Connected entity is potentially inconsistent.
	UML use case	Identity	Connected entity is inconsistent.
	Architectural component / architectural module	Satisfaction	Connected entity is potentially inconsistent.
UML Use case	Same consistency rules apply as in case of requirement artefact elements.		
Architectural component	Java source code / UML class diagram / JUnit container / UML sequence diagram lifeline	Satisfaction	Connected entity is potentially inconsistent.
	Requirement / Use case	Satisfaction	Connected entity is consistent.
	Architectural module	Satisfaction	Connected entity is potentially inconsistent.
Architectural module	Same consistency rules apply as in case of architectural component artefact elements.		

Table B.14: *Edit file level* change - *Edit artefact element level* change - Derivation of inter rules part II.

Changed Entity	Connected Entity	Intra Trace Link Type	Intra Rules
Edit UML member element (signature)	UML container element	Domain dependency	Connected entity is consistent.
Delete UML member element	UML container element	Domain dependency	Connected entity is consistent.
Add UML member element	UML container element	Domain dependency	Connected entity is consistent.
Edit UML container element (signature)	UML container element	Domain dependency	Connected entity is potentially inconsistent.
Delete UML container element	UML container element	Domain dependency	Connected entity is potentially inconsistent.
Add UML container element	UML container element	Domain dependency	Connected entity is consistent.
Edit Java / JUnit member element (signature)	Java / JUnit container element	Domain dependency	Connected entity is potentially inconsistent.
Edit Java / JUnit member element (content)	Java / JUnit container element	Domain dependency	Connected entity is consistent.
Delete Java / JUnit member element	Java / JUnit container element	Domain dependency	Connected entity is potentially inconsistent.
Add Java / JUnit member element	Java / JUnit container element	Domain dependency	Connected entity is consistent.
Edit Java / JUnit container element (signature)	Java / JUnit container element	Domain dependency	Connected entity is inconsistent.

Table B.15: Derivation of intra consistency rules.

Artefact	Change Description	Corresponding Rule
Java source code	Delete a .java file from the repository.	Delete trace maintenance rule
UML class diagram	Delete an entire UML class diagram (.dia) from the repository.	Delete trace maintenance rule
JUnit test	Delete a .java file from the repository.	Delete trace maintenance rule
UML sequence diagram	Delete an entire UML sequence diagram (.dia) from the repository.	Delete trace maintenance rule
Requirement specification	Delete an entire .odt file, from the repository.	Delete trace maintenance rule
UML use case diagram	Delete an entire UML use case diagram (.dia) from the repository.	Delete trace maintenance rule
Software architecture (conceptual view)	Delete an entire architecture diagram (.dia) from the repository.	Delete trace maintenance rule
Software architecture (module view)	Delete an entire architecture diagram (.dia) from the repository.	Delete trace maintenance rule

Table B.16: Trace maintenance rules in the delete file level change scenario for each artefact type.

Artefact	Change Description	Corresponding Rule
Java source code	Delete a Java member / container element.	Delete trace maintenance rule
UML class diagram	Delete a UML member / container element.	Delete trace maintenance rule
JUnit test	Delete a JUnit member / container element.	Delete trace maintenance rule
UML sequence diagram	Delete a lifeline element.	Delete trace maintenance rule
Requirement specification	Delete a requirement.	Delete trace maintenance rule
UML use case diagram	Delete a use case.	Delete trace maintenance rule
Software architecture (conceptual view)	Delete a component.	Delete trace maintenance rule
Software architecture (module view)	Delete a module.	Delete trace maintenance rule

Table B.17: Trace maintenance rules in the delete artefact element level change scenario for each artefact type.

Artefact	Change Description	Corresponding Rule
Java source code	Edit a Java member / container element.	Edit trace maintenance rule
UML class diagram	Edit a UML member / container element.	Edit trace maintenance rule
JUnit test	Edit a JUnit member / container element.	Edit trace maintenance rule
UML sequence diagram	Edit a lifeline element.	Edit trace maintenance rule
Requirement specification	Edit a requirement.	Edit trace maintenance rule
UML use case diagram	Edit a use case.	Edit trace maintenance rule
Software architecture (conceptual view)	Edit a component.	Edit trace maintenance rule
Software architecture (module view)	Edit a module.	Edit trace maintenance rule

Table B.18: Trace maintenance rules in the edit artefact element level change scenario for each artefact type.

REFERENCES

[1] T. Mens, *Software Evolution*, ch. Introduction and Roadmap: History and Challenges of Software Evolution, pp. 1–11. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[2] B. W. Boehm, “Software Engineering,” *IEEE Transactions on Computers*, vol. C-25, no. 12, pp. 1226–1241, 1976.

[3] P. Marounek, “Simplified approach to effort estimation in software maintenance,” *Journal of Systems Integration*, vol. 3, no. 3, pp. 51–63, 2012.

[4] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri, “Challenges in software evolution,” pp. 13–22, 2005.

[5] *Extreme Chaos*, 2001. Available at `http://www.cin.ufpe.br/{~}gmp/docs/papers/extreme{_}chaos2001.pdf`. Accessed January 2016.

[6] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition).* Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004.

[7] S. D. Suh and I. Neamtiu, “Studying software evolution for taming software complexity,” *Proceedings of the Australian Software Engineering Conference, ASWEC*, pp. 3–12, 2010.

[8] T. Olsen and J. Grundy, “Supporting traceability and inconsistency management between software artefacts,” 2002.

[9] S. P. Reiss, “Incremental maintenance of software artifacts,” *IEEE Transactions on Software Engineering*, vol. 32, no. 9, pp. 682–697, 2006.

[10] H. P. Breivold, I. Crnkovic, and P. Eriksson, “Evaluating software evolvability,” *Software Engineering Research and Practice in Sweden*, vol. 96, 2007.

[11] P. Loucopoulos and V. Karakostas, *C . A . S . E . Technology.* New York, NY, USA: McGraw-Hill, Inc., 1995.

[12] I. Sommerville, *Software Engineering: (8th Edition) (International Computer Science)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[13] Y. Zhang, R. Witte, J. Rilling, and V. Haarslev, "Ontological approach for the semantic recovery of traceability links between software artefacts," *IET software*, vol. 2, no. 3, pp. 185–203, 2008.

[14] S. Dart, "Concepts in configuration management systems," in *Proceedings of the 3rd international workshop on Software configuration management*, pp. 1–18, ACM, 1991.

[15] *Configuration Management*. Available at `http://www.sei.cmu.edu/productlines/frame{_}report/config.man.htm`. Accessed January 2016.

[16] *Configuration Management*. Available at `http://www.sei.cmu.edu/productlines/frame{_}report/config.man.htm`. Accessed March 2016.

[17] B. Nuseibeh, J. Kramer, and A. Finkelstein, "A framework for expressing the relationships between multiple views in requirements specification," *IEEE Transactions on Software Engineering*, vol. 20, no. 10, pp. 760–773, 1994.

[18] T. Stahl, M. Völter, J. Bettin, A. Haase, and S. Helsen, *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

[19] A. G. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[20] J. Bettin, *Software, Engineering, Artefacts, Language*, 2010. Available at `http://semat.org/documents/20181/27952/SEMAT_position_Bettin.pdf/665484b8-ef4b-4c0c-a77a-891fd9d8cb46`. Accessed January 2016.

[21] D. Beyer and A. Noack, "Clustering software artifacts based on frequent common changes," in *13th International Workshop on Program Comprehension (IWPC'05)*, pp. 259–268, IEEE, 2005.

[22] *Programming Paradigms*. Available at `http://cs.lmu.edu/{~}ray/notes/paradigms/`. Accessed January 2016.

[23] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified Modeling Language Reference Manual*. Pearson Higher Education, 2004.

[24] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[25] J. Cleland-Huang, O. Gotel, and A. Zisman, *Software and Systems Traceability*. London: Springer London, 2012.

[26] O. C. Z. Gotel and C. W. Finkelstein, "An analysis of the requirements traceability problem," in *Requirements Engineering, 1994., Proceedings of the First International Conference on*, pp. 94–101, Apr 1994.

[27] B. Ramesh, C. Stubbs, T. Powers, and M. Edwards, "Requirements traceability: Theory and practice," *Annals of Software Engineering*, vol. 3, no. 1, pp. 397–415, 1997.

[28] *MATLAB and Simulink - MathWorks*. Available at `http://www.mathworks.co.uk/discovery/requirements-traceability.html`. Accessed March 2016.

[29] *IEEE Standard Dictionary of Measures to Produce Reliable Software*, 1989. Available at `https://standards.ieee.org/findstds/standard/982.1-1988.html`. Accessed March 2016.

[30] S. Engineering and S. Committee, "IEEE Standard for Software Maintenance," 1998. Available at `http://www.cs.uah.edu/~rcoleman/CS499/CourseTopics/IEEE_Std_1219-1998.pdf`. Accessed March 2016.

[31] B. Ramesh and M. Jarke, "Toward reference models for requirements traceability," *IEEE transactions on software engineering*, vol. 27, no. 1, pp. 58–93, 2001.

[32] *Center of Excellence for Software Traceability*. Available at `http://www.coest.org/`. Accessed March 2016.

[33] G. Antoniol, G. Canfora, G. Casazza, a. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Transactions on Software Engineering*, vol. 28, no. 10, pp. 970–983, 2002.

[34] P. Mäder, O. Gotel, and I. Philippow, "Rule-based maintenance of post-requirements traceability relations," in *2008 16th IEEE International Requirements Engineering Conference*, pp. 23–32, IEEE, 2008.

[35] G. Spanoudakis and A. Zisman, "Software Traceability: A Roadmap," in *Handbook of Software Engineering and Knowledge Engineering*, pp. 395–428, World Scientific Publishing, 2004.

[36] S. Winkler and J. Pilgrim, "A survey of traceability in requirements engineering and model-driven development," *Software & Systems Modeling*, vol. 9, no. 4, pp. 529–565, 2009.

[37] R. Arnold and S. Bohner, “Impact analysis-Towards a framework for comparison,” *1993 Conference on Software Maintenance*, pp. 292–301, 1993.

[38] D. Leffingwell and D. Widrig, *Managing Software Requirements: A Unified Approach*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[39] B. Li, X. Sun, H. Leung, and S. Zhang, “A survey of code-based change impact analysis techniques,” vol. 23, pp. 613–646, Wiley Online Library, 2013.

[40] N. Kama, “Change Impact Analysis for the Software Development Phase : State-of-the-art,” *International Journal of Software Engineering & Its Applications*, vol. 7, no. 2, pp. 235–244, 2013.

[41] S. 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*, pp. 41–50, ACM, 2011.

[42] S. Lehnert, “A Review of Software Change Impact Analysis,” tech. rep., Ilmenau University of Technology, Department of Software Systems / Process Informatics, Ilmenau, 2011.

[43] S. Ibrahim, N. B. Idris, M. Munro, and A. Deraman, “Integrating software traceability for change impact analysis.,” *Int. Arab J. Inf. Technol.*, vol. 2, no. 4, pp. 301–308, 2005.

[44] S. Wong, Y. Cai, and M. Dalton, “Change impact analysis with stochastic dependencies,” *Drexel University Philadelphia, PA, USA, Tech. Rep*, 2011.

[45] *Definition of Consistency*. Available at `http://www.merriam-webster.com/dictionary/consistency`. Accessed March 2016.

[46] B. Nuseibeh, S. Easterbrook, and A. Russo, “Leveraging inconsistency in software development,” *Computer*, vol. 33, no. 4, pp. 24–29, 2000.

[47] G. Spanoudakis and A. Zisman, “Inconsistency management in software engineering: Survey and open research issues,” in *Handbook of Software Engineering and Knowledge Engineering*, pp. 329–380, World Scientific, 2001.

[48] M. Kamalrudin and S. Sidek, “A review on software requirements validation and consistency management,” *International Journal of Software Engineering and its Applications*, vol. 9, no. 10, pp. 39–58, 2015.

[49] S. J. I. Herzig, U. States, and A. Reichwein, “A Conceptual Framework for Consistency Management in Model-Based Systems Engineering,” *Proceedings of the ASME 2011 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference*, pp. 1–11, 2011.

[50] P. Tarr and L. Clarke, “Consistency management for complex applications,” *Proceedings of the 20th International Conference on Software Engineering*, pp. 230–239, 1998.

[51] M. Elaasar and L. Briand, “An overview of uml consistency management,” *Carleton University, Canada, Technical Report SCE-04-18*, 2004.

[52] M. Vierhauser, P. Grünbacher, W. Heider, G. Holl, and D. Lettner, “Applying a consistency checking framework for heterogeneous models and artifacts in industrial product lines,” in *Proceedings of the 15th International ACM/IEEE Conference on Model Driven Engineering Languages & Systems (MODELS)*, 2012.

[53] M. Giffin, O. de Weck, G. Bounova, R. Keller, C. Eckert, and P. J. Clarkson, “Change Propagation Analysis in Complex Technical Systems,” *Journal of Mechanical Design*, vol. 131, no. 8, p. 081001, 2009.

[54] J. Han, “Supporting Impact Analysis and Change Propagation in Software Engineering Environments,” in *International Workshop on Software Technology and Engineering Practice*, no. 96, pp. 172–182, 1997.

[55] V. Rajlich, “A model for change propagation based on graph rewriting,” pp. 84–91, 1997.

[56] P. Jönsson, “Impact analysis: Organisational views and support techniques,” 2005.

[57] H. Malik and A. E. Hassan, “Supporting software evolution using adaptive change propagation heuristics,” in *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pp. 177–186, IEEE, 2008.

[58] A. E. Hassan and R. C. Holt, “Predicting change propagation in software systems,” in *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pp. 284–293, IEEE, 2004.

[59] J. Biolchini, P. G. Mian, A. Candida, and C. Natali, “Systematic Review in Software Engineering,” *Engineering*, vol. 679, pp. 165–176, 2005.

[60] B. Kitchenham, P. Brereton, D. Budgen, M. Turner, J. Bailey, and S. G. Linkman, “Systematic literature reviews in software engineering - A systematic literature review,” *Information & Software Technology*, vol. 51, no. 1, pp. 7–15, 2009.

[61] J. Cleland-Huang, O. Gotel, J. H. Hayes, P. Mäder, and A. Zisman, “Software traceability: trends and future directions,” in *Proceedings of the on Future of Software Engineering, FOSE*, pp. 55–69, 2014.

[62] S. Lehnert, Q. Farooq, and M. Riebisch, “Rule-based impact analysis for heterogeneous software artifacts,” in *17th European Conference on Software Maintenance and Reengineering, CSMR*, pp. 209–218, 2013.

[63] J. Grundy, J. Hosking, and W. B. Mugridge, “Inconsistency management for multiple-view software development environments,” *IEEE Transactions on Software Engineering*, vol. 24, no. 11, pp. 960–981, 1998.

[64] S. P. Reiss, “Constraining software evolution,” in *18th International Conference on Software Maintenance (ICSM*, pp. 162–171, 2002.

[65] M. Hammad, M. L. Collard, and J. I. Maletic, “Automatically identifying changes that impact code-to-design traceability during evolution,” *Software Quality Journal*, vol. 19, no. 1, pp. 35–64, 2010.

[66] M. Zekkaoui and A. Fennan, “Unified approach for building heterogeneous artifacts and consistency rules,” *Journal of Emerging Technologies in Web Intelligence*, vol. 6, no. 1, 2014.

[67] E. C. Nistor, J. R. Erenkrantz, S. A. Hendrickson, and A. van der Hoek, “Archevol: versioning architectural-implementation relationships,” in *Proceedings of the 12th International Workshop on Software Configuration Management, SCM*, pp. 99–111, 2005.

[68] F. Lanubile, “Collaboration in distributed software development,” in *Software Engineering, International Summer Schools, ISSSE*, pp. 174–193, 2008.

[69] F. Fasano, “Fine-grained management of software artefacts,” in *23rd IEEE International Conference on Software Maintenance (ICSM*, pp. 507–508, 2007.

[70] A. De Lucia, R. Oliveto, and G. Tortora, “Adams re-trace: Traceability link recovery via latent semantic indexing,” in *Proceedings of the 30th International Conference on Software Engineering*, ICSE ’08, pp. 839–842, 2008.

[71] C. Hapter, J. A. Scott, and D. Nisse, “Software Configuration Management,” in *Guide to the Software Engineering Body of Knowledge* (R. D. Alain Abran, James W. Moore, Pierre Bourque, ed.), ch. 7, Los Alamitos: IEEE Computer Society, 2001.

[72] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora, “Fine-grained management of software artefacts: the adams system,” *Software Practice & Experience*, vol. 40, no. 11, pp. 1007–1034, 2010.

[73] T. N. Nguyen, E. V. Munson, J. T. Boyland, and C. Thao, “An infrastructure for development of object-oriented, multi-level configuration management services,” in *27th International Conference on Software Engineering (ICSE*, pp. 215–224, 2005.

[74] T. N. Nguyen, E. V. Munson, J. Boyland, and C. Thao, “Architectural software configuration management in molhado,” in *20th International Conference on Software Maintenance (ICSM), pages = 296–305, year = 2004, crossref = DBLP:conf/icsm/2004, url = http://dx.doi.org/10.1109/ICSM.2004.1357815, doi = 10.1109/ICSM.2004.1357815, biburl = http://dblp2.uni-trier.de/rec/bib/conf/icsm/NguyenMBT04, bibsource = dblp computer science bibliography, http://dblp.org*.

[75] E. J. Whitehead Jr, *An analysis of the hypertext versioning domain*. PhD thesis, University of California, Irvine, 2000.

[76] J. Conklin, “Hypertext: An introduction and survey,” *Computer*, vol. 20, no. 9, pp. 17–41, 1987.

[77] N. M. Delisle and M. D. Schwartz, “Neptune: a hypertext system for CAD applications,” in *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, pp. 132–143, 1986.

[78] K. M. Anderson, R. N. Taylor, and E. J. W. Jr., “Chimera: Hypertext for heterogeneous software environments,” in *ECHT ’94: European Conference on Hypertext Technology*, pp. 94–107, 1994.

[79] P. K. Garg and W. Scacchi, “A hypertext system to manage software life cycle documents,” in *Proceedings of the Twenty-First Annual Hawaii International Conference on Software Track*, pp. 337–346, 1988.

[80] T. N. Nguyen and E. V. Munson, “The software concordance: A new software document management environment,” in *Proceedings of the 21st Annual International Conference on Documentation*, pp. 198–205, 2003.

[81] R. G. Dewar, L. M. MacKinnon, R. J. Pooley, A. D. Smith, M. J. Smith, and P. A. Wilcox, “The OPHELIA project: Supporting software development in a distributed environment,” in *Proceedings of the IADIS International Conference WWW/Internet 2002, ICWI*, pp. 568–571, 2002.

[82] P. A. Wilcox, C. R. Russell, M. J. Smith, A. D. Smith, R. J. Pooley, L. M. Mackinnon, R. G. Dewar, and D. Weiss, "A CORBA-Oriented Approach to Heterogeneous Tool Integration ; OPHELIA," in *In The Workshop on Tool Integration in System Development (TIS 2003), 2003. the 9th European Software Engineering Conference and 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2003.

[83] M. Hapke, A. Jaszkiewicz, K. Kowalczykiewicz, D. Weiss, and P. Zielniewicz, "Ophelia: Open platform for distributed software development," in *Open Source for an Information and knowledge society: Proceedings of the Open Source International Conference.*, Citeseer, 2004.

[84] M. Smith, D. Weiss, P. Wilcox, and R. Dewar, "The ophelia traceability layer," in *Cooperative Method and Tools for Distributed Softwar Processes, Volume 380.222*, pp. 88–464, 2003.

[85] *CORBA*. Available at `http://www.omg.org/gettingstarted/corbafaq.htm`. Accessed March 2016.

[86] B. Bruegge, A. H. Dutoit, T. Wolf, T. Universit, and D. Garching, "Sysiphus : Enabling informal collaboration in global software development," in *IEEE International Conference on Global Software Engineering (ICGSE'06)*, 2006.

[87] D. Nutter, C. Boldyreff, S. Rank, *et al.*, "An artefact repository to support distributed software engineering," 2003.

[88] C. Boldyreff, D. Nutter, and S. Rank, "Active artefact management for distributed software engineering," in *Computer Software and Applications Conference, Proceedings. 26th Annual International*, pp. 1081–1086, 2002.

[89] J.-Y. Chen and S.-C. Chou, "Consistency management in a process environment," *Journal of Systems and Software*, vol. 47, pp. 105 – 110, 1999.

[90] R. Matinnejad and R. Ramsin, "An analytical review of process-centered software engineering environments," in *IEEE 19th International Conference and Workshops on Engineering of Computer-Based Systems, ECBS*, pp. 64–73, 2012.

[91] D. E. Knuth, "Literate programming," *The Computer Journal*, vol. 27, no. 2, pp. 97–111, 1984.

[92] D. Kupfer, *Eclipse Intent: Being Agile does not mean being short-sighted*, 2012. Available at `http://jaxenter.com/`

`eclipse-intent-being-agile-does-not-mean-being-short-sighted-45856.html`. Accessed March 2016.

[93] *Mylyn Intent.* Available at `http://www.eclipse.org/proposals/mylyn.docs.intent/`. Accessed March 2016.

[94] A. Aguiar and G. David, "Wikiwiki weaving heterogeneous software artifacts," in *Proceedings of the 2005 International Symposium on Wikis*, pp. 67–74, 2005.

[95] J. Aldrich, C. Chambers, and D. Notkin, "Archjava: connecting software architecture to implementation," in *Proceedings of the 24th International Conference on Software Engineering, ICSE*, pp. 187–197, 2002.

[96] L. G. Murta, A. van der Hoek, and C. M. Werner, "Continuous and automated evolution of architecture-to-implementation traceability links," *Automated Software Engineering*, vol. 15, no. 1, pp. 75–107, 2008.

[97] F. A. C. Pinheiro, *Perspectives on Software Requirements*, ch. Requirements Traceability, pp. 91–113. Boston, MA: Springer US, 2004.

[98] M. F. Bashir and M. A. Qadir., "Traceability techniques: A critical study," in *2006 IEEE International Multitopic Conference*, pp. 265–268, 2006.

[99] J. Cleland-Huang and J. Guo, "Towards more intelligent trace retrieval algorithms," in *3rd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, RAISE*, pp. 1–6, 2014.

[100] A. Qusef, R. Oliveto, and A. De Lucia, "Recovering traceability links between unit tests and classes under test: An improved method," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pp. 1–10, IEEE, 2010.

[101] J. Lin, C. C. Lin, J. Cleland-Huang, R. Settimi, J. Amaya, G. Bedford, B. Berenbach, O. B. Khadra, C. Duan, and X. Zou, "Poirot: A distributed tool supporting enterprise-wide automated traceability," in *14th IEEE International Conference on Requirements Engineering (RE)*, pp. 356–357, 2006.

[102] B. Liang, G. V. Wilson, and J. Aranda, "Tracing requirements to tests: An information retrieval approach,"

[103] J. N. och Dag, B. Regnell, P. Carlshamre, M. Andersson, and J. Karlsson, "A feasibility study of automated natural language requirements analysis in market-driven development," *Requirements Engineering*, vol. 7, no. 1, pp. 20–33, 2002.

[104] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor, “Software traceability with topic modeling,” in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, pp. 95–104, 2010.

[105] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram, “Advancing candidate link generation for requirements tracing: The study of methods,” *IEEE Transactions on Software Engineering*, vol. 32, no. 1, pp. 4–19, 2006.

[106] J. I. Maletic, E. V. Munson, A. Marcus, and T. N. Nguyen, “Using a hypertext model for traceability link conformance analysis,” in *In Proc. of the 2nd Int. Workshop on Traceability in Emerging Forms of Software Engineering*, pp. 47–54, 2003.

[107] S. Klock, M. Gethers, B. Dit, and D. Poshyvanyk, “Traceclipse: an eclipse plug-in for traceability link recovery and management,” in *TEFSE’11, Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering*, pp. 24–30, 2011.

[108] G. Bavota, L. Colangelo, A. D. Lucia, S. Fusco, R. Oliveto, and A. Panichella, “Traceme: Traceability management in eclipse,” in *28th IEEE International Conference on Software Maintenance, ICSM*, pp. 642–645, 2012.

[109] H. Kagdi, J. I. Maletic, and B. Sharif, “Mining Software Repositories for Traceability Links,” in *Proceedings of the 15th IEEE International Conference on Program Comprehension*, pp. 145–154, IEEE Computer Society, 2007.

[110] A. Egyed and P. Grünbacher, “Automating requirements traceability: Beyond the record & replay paradigm,” in *17th IEEE International Conference on Automated Software Engineering(ASE)*, pp. 163–171, 2002.

[111] S. Hayashi, T. Yoshikawa, and M. Saeki, “Sentence-to-code traceability recovery with domain ontologies,” in *2010 Asia Pacific Software Engineering Conference*, pp. 385–394, 2010.

[112] G. Spanoudakis, A. Zisman, E. Pérez-miñana, P. Krause, and B. P. D. Systems, “Rule-based generation of requirements traceability relations,” *Journal of Systems and Software*, vol. 72, pp. 105–127, 2004.

[113] S. A. Sherba, K. M. Anderson, and M. Faisal, “A framework for mapping traceability relationships,” in *2 nd International Workshop on Traceability in Emerging Forms of Software Engineering at 18th IEEE International Conference on Automated Software Engineering*, pp. 32–39, 2003.

[114] A. Mahmoud, N. Niu, and S. Xu, "A semantic relatedness approach for traceability link recovery," in *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pp. 183–192, 2012.

[115] H. Sultanov, J. H. Hayes, and W.-K. Kong, "Application of swarm techniques to requirements tracing," *Requirements Engineering*, vol. 16, no. 3, pp. 209–226, 2011.

[116] J. Cleland-Huang, C. K. Chang, and M. Christensen, "Event-based traceability for managing evolutionary change," *IEEE Transactions on Software Engineering*, vol. 29, no. 9, pp. 796–810, 2003.

[117] J. Cleland-Huang, C. K. Chang, and Y. Ge, "Supporting event based traceability through high-level recognition of change events," in *26th International Computer Software and Applications Conference (COMPSAC)*, pp. 595–602, 2002.

[118] P. Mäder and O. Gotel, "Towards automated traceability maintenance," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2205–2227, 2012.

[119] H. U. Asuncion, F. François, and R. N. Taylor, "An end-to-end industrial software traceability tool," in *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 115–124, 2007.

[120] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, "Chianti: A tool for change impact analysis of java programs," in *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pp. 432–448, 2004.

[121] M. Lee, A. J. Offutt, and R. T. Alexander, "Algorithmic analysis of the impacts of changes to object-oriented software," in *Technology of Object-Oriented Languages and Systems. TOOLS 34. Proceedings. 34th International Conference on*, pp. 61–70, 2000.

[122] A. R. Sharafat and L. Tahvildari, "A probabilistic approach to predict changes in object-oriented software systems," in *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pp. 27–38, 2007.

[123] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *Proceedings of the 26th International Conference on Software Engineering*, pp. 563–572, 2004.

[124] H. Kagdi, "Improving change prediction with fine-grained source code mining," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, pp. 559–562, 2007.

[125] G. Tóth, P. Hegedűs, A. Beszédes, T. Gyimóthy, and J. Jász, "Comparison of different impact analysis methods and programmer's opinion: An empirical study," in *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, pp. 109–118, 2010.

[126] C. Dantas, L. Murta, and C. Werner, "Mining change traces from versioned uml repositories," in *Brazilian Symposium on Software Engineering (SBES)*, pp. 236–252, 2007.

[127] S. Lock and G. Kotonya, "An integrated, probabilistic framework for requirement change impact analysis," *The Australian Journal of Information Systems*, vol. 6, no. 2, pp. 38–63, 1999.

[128] W.-T. Lee, W.-Y. Deng, J. Lee, and S.-J. Lee, "Change impact analysis with a goal-driven traceability-based approach," *International Journal of Intelligent Systems*, vol. 25, no. 8, pp. 878–908, 2010.

[129] L. C. Briand, Y. Labiche, and L. O'Sullivan, "Impact analysis and change management of uml models," in *Software Maintenance, (ICSM). Proceedings. International Conference on*, pp. 256–265, Sept 2003.

[130] S. Ibrahim, N. B. Idris, M. Munro, and A. 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.

[131] A. Finkelstein, "A foolish consistency: Technical challenges in consistency management," in *International Conference on Database and Expert Systems Applications*, pp. 1–5, Springer, 2000.

[132] T. Mens and P. Van Gorp, "A taxonomy of model transformation," *Electronic Notes in Theoretical Computer Science*, vol. 152, pp. 125–142, 2006.

[133] S. Sendall and J. Küster, "Taming model round-trip engineering," in *Proceedings of Workshop on Best Practices for Model-Driven Software Development*, p. 1, 2004.

[134] M. Antkiewicz and K. Czarnecki, “Design space of heterogeneous synchronization,” in *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE*, pp. 3–46, 2007.

[135] *Class Designer - keep code and class model in sync*. Available at `http://blogs.msdn.com/b/classdesigner/archive/2005/05/12/417000.aspx`. Accessed March 2016.

[136] *Overview of Objecteering for Eclipse*. Available at `http://support.objecteering.com/objecteering6.1/help/us/objecteering{_}for{_}eclipse/intro/overview.htm`. Accessed March 2016.

[137] *Enterprise Architect*. Available at `http://www.sparxsystems.com.au/resources/demos/`. Accessed March 2016.

[138] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd ed., 2009.

[139] *Executable UML | Modeling Languages*. Available at `http://modeling-languages.com/executable-uml/`. Accessed March 2016.

[140] D. Milicev, *Model-Driven Development with Executable UML*. Wrox Press Ltd., 1st ed., 2009.

[141] *Gentleware - model to business: uml tool - professional edition*. Available at `http://www.gentleware.com/uml-software-pe.html`. Accessed March 2016.

[142] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelsteiin, “xlinkit: A consistency checking and smart link generation service,” *ACM Transactions on Internet Technology (TOIT)*, vol. 2, no. 2, pp. 151–185, 2002.

[143] A. L. Campbell, C. B. H. Cheng, E. W. McUmber, and K. R. E. Stirewalt, “Automatically detecting and visualising errors in uml diagrams,” *Requirements Engineering*, vol. 7, no. 4, pp. 264–287, 2002.

[144] C. Dimech and D. Balasubramaniam, “Maintaining architectural conformance during software development: A practical approach,” in *European Conference on Software Architecture*, pp. 208–223, Springer, 2013.

[145] S. Sheuly, *A Systematic Literature Review on Agile Project Management*. PhD thesis, 2013.

[146] E. M. Simão, *Comparison of software development methodologies based on the SWEBOK*. PhD thesis, 2011.

[147] B. W. Boehm, “A spiral model of software development and enhancement,” *Computer*, vol. 21, no. 5, pp. 61–72, 1988.

[148] K. Fowler, *Mission-critical and safety-critical systems handbook: Design and development for embedded applications*. Newnes, 2009.

[149] J. D. Herbsleb and D. Moitra, “Global software development,” *IEEE software*, vol. 18, no. 2, pp. 16–20, 2001.

[150] S. Boccaletti, V. Latora, Y. Moreno, M. Chavez, and D.-U. Hwang, “Complex networks: Structure and dynamics,” *Physics Reports*, vol. 424, pp. 175 – 308, 2006.

[151] *Network science*. Available at `http://www.network-science.org/`. Accessed March 2016.

[152] L. Šubelj and M. Bajec, “Software systems through complex networks science: Review, analysis and applications,” in *Proceedings of the First International Workshop on Software Mining*, SoftwareMining ’12, pp. 9–16, 2012.

[153] A. Chatzigeorgiou, N. Tsantalis, and G. Stephanides, “Application of graph theory to oo software engineering,” in *Proceedings of the 2006 International Workshop on Workshop on Interdisciplinary Software Engineering Research*, pp. 29–36, 2006.

[154] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, “Modeling and discovering vulnerabilities with code property graphs,” in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, pp. 590–604, 2014.

[155] C. Wenjing and X. Shenghong, “A software function testing method based on data flow graph,” in *International Symposium on Information Science and Engineering*, vol. 2, pp. 28–31, 2008.

[156] K. Ruohonen, *Graph Theory*. Tampere University of Technology, 2008.

[157] M. A. Rodriguez and P. Neubauer, “Constructions from dots and lines,” *Bulletin of the American Society for Information Science and Technology*, vol. 36, no. 6, pp. 35–41, 2010.

[158] M. Rodriguez, *Knowledge Representation and Reasoning with Graph Databases*, 2011. Available at `http://markorodriguez.com/2011/02/23/knowledge-representation-and-reasoning-with-graph-databases/`. Accessed March 2016.

[159] J. Tauberer, *What is RDF and what is it good for?*, 2008. Available at `https://github.com/JoshData/rdfabout/blob/gh-pages/intro-to-rdf.md`. Accessed March 2016.

[160] S. Lehnert, Q. u. a. Farooq, and M. Riebisch, “A taxonomy of change types and its application in software evolution,” in *Engineering of Computer Based Systems (ECBS), IEEE 19th International Conference and Workshops on*, pp. 98–107, 2012.

[161] R. Wojcik, F. Bachmann, L. Bass, P. Clements, P. Merson, R. Nord, and B. Wood, “Attribute-driven design (add), version 2.0,” tech. rep., DTIC Document, 2006.

[162] L. Bass, *Architectural tactics*, 2006. Available at `http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=31149`. Accessed March 2016.

[163] F. Bachmann, L. Bass, and R. Nord, “Modifiability tactics,” tech. rep., DTIC Document, 2007.

[164] L. J. Bass and B. E. John, “Linking usability to software architecture patterns through general scenarios,” *Journal of Systems and Software*, vol. 66, no. 3, pp. 187–197, 2003.

[165] *Core J2EE Patterns - Data Access Object*. Available at `http://www.oracle.com/technetwork/java/dataaccessobject-138824.html`. Accessed March 2016.

[166] O. M. Group, *OMG Unified Modeling Language*, 2015. Available at `http://www.omg.org/spec/UML/2.5/`. Accessed March 2016.

[167] F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, and J. Stafford, *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2011.

[168] K. Fakhroutdinov, *Class and Object Diagrams Overview*. Available at `http://www.uml-diagrams.org/class-diagrams-overview.html`. Accessed March 2016.

[169] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, *The Java® Language Specification*, 2015. Available at `http://docs.oracle.com/javase/specs/jls/se8/jls8.pdf`. Accessed March 2016.

[170] G. H. Andreas Birk, *List of Requirements Management Tools*. Available at `http://makingofsoftware.com/resources/list-of-rm-tools`. Accessed March 2016.

[171] *UML Directory*. Available at `http://uml-directory.omg.org/`. Accessed March 2016.

[172] D. M. L. Collard, D. J. I. Maletic, M. Decker, B. Bartman, D. Guarnera, C. Newman, H. Michaud, B. Kovacs, and Tessandra Sage, *srcML*. Available at `http://www.srcml.org/about-srcml.html`. Accessed March 2016.

[173] M. Roughan and J. Tuke, "Unravelling graph-exchange file formats," *arXiv preprint arXiv:1503.02781*, 2015.

[174] M. Eiglsperger, U. Brandes, J. Lerner, and C. Pich, "Graph Markup Language (GraphML) 16.1," *Handbook of Graph Drawing and Visualization*, pp. 517–541, 2013.

[175] J. Pokorny, "New Database Architectures: Steps towards Big Data Processing," *IADIS European Conference Data Mining*, pp. 3–10, 2013.

[176] *Neo4j - The World's Leading Graph Database*. Available at `http://www.neo4j.org/`. Accessed March 2016.

[177] I. Robinson, J. Webber, and E. Eifrem, *Graph Databases*. O'Reilly Media, Inc., 2013.

[178] R. Angles and C. Gutierrez, "Survey of graph database models," *ACM Computing Surveys*, vol. 40, no. 1, pp. 1–39, 2008.

[179] M. Rodriguez, *The Graph Traversal Programming Pattern*, 2010. Available at `http://www.slideshare.net/slidarko/graph-windycitydb2010?related=1`. Accessed March 2016.

[180] P. Macko, D. Margo, and M. Seltzer, "Performance introspection of graph databases," *Proceedings of the 6th International Systems and Storage Conference on - SYSTOR '13*, p. 1, 2013.

[181] M. Ciglan, A. Averbuch, and L. Hluchy, "Benchmarking Traversal Operations over Graph Databases," *IEEE 28th International Conference on Data Engineering Workshops*, pp. 186–189, 2012.

[182] N. Martínez-bazan, V. Muntés-mulero, and S. Gómez-villamor, "DEX : High-Performance Exploration on Large Graphs for Information Retrieval," *Artificial Intelligence*, pp. 573–582, 2007.

[183] F. Holzschuher and R. Peinl, "Performance of Graph Query Languages: Comparison of Cypher, Gremlin and Native Access in Neo4J," in *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, (New York, NY, USA), pp. 195–204, 2013.

[184] M. Rodriguez, *Solving Problems with Graphs*, 2012. Available at http://www.slideshare.net/slidarko/yow-australia2012?related=1. Accessed March 2016.

[185] E. W. Myers, "An O(ND) difference algorithm and its variations," *Algorithmica*, vol. 1, no. 1-4, pp. 251–266, 1986.

[186] G. Cobéna, T. Abdessalem, and Y. Hinnach, "A comparative study for xml change detection.," in *BDA*, 2002.

[187] K.-C. Tai, "The Tree-to-Tree Correction Problem," *Journal of the ACM*, vol. 26, no. 3, pp. 422–433, 1979.

[188] J. Stanek, S. Kothari, and K. Gui, "Method of comparing graph differencing algorithms for software differencing," *IEEE International Conference on Electro/Information Technology*, pp. 482–487, 2008.

[189] O. Akgun, I. P. Gent, C. Jefferson, I. Miguel, and P. Nightingale, "Breaking Conditional Symmetry in Automated Constraint Modelling with CONJURE," *The 21st European Conference on Artificial Intelligence*, vol. 263, pp. 3–8, 2014.

[190] Y. Zhang, R. Witte, J. Rilling, and V. Haarslev, "An Ontology-based Approach for the Recovery of Traceability Links," in *3rd International Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering (ATEM)*, 2006.

[191] T. M. Mitchell, *Machine Learning*. New York, NY, USA: McGraw-Hill, Inc., 1 ed., 1997.

[192] M. J. Zaki and W. M. Jr, *Data Mining and Analysis: Fundamental Concepts and Algorithms*. New York, NY, USA: Cambridge University Press, 2014.

[193] L. Schmidt-Thieme, *Linear Regression*, 2007. Available at http://www.ismll.uni-hildesheim.de/lehre/ml-07w/skript/ml-2up-01-linearregression.pdf. Accessed March 2016.

[194] M. Mohri, A. Rostamizadeh, and A. Talwalkar, *Foundations of Machine Learning*. The MIT Press, 2012.

[195] D. Zhang and J. J. P. Tsai, "Machine learning and software engineering.," *Software Quality Journal*, vol. 11, no. 2, pp. 87–119, 2003.

[196] V. Musco, A. Carette, M. Monperrus, and P. Preux, "A Learning Algorithm for Change Impact Prediction," in *5th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, 2016.

[197] M. S. Twala, Bhekisipho, Michelle Cartwright, “Applying Rule Induction in Software Prediction,” *Advances in Machine Learning Applications in Software Engineering*, pp. 265–286, 2007.

[198] M. Grechanik, K. S. McKinley, and D. E. Perry, “Recovering and using use-case-diagram-to-source-code traceability links,” in *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 95–104, 2007.

[199] J. Cleland-Huang, A. Czauderna, M. Gibiec, and J. Emenecker, “A machine learning approach for tracing regulatory codes to product specific requirements,” in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pp. 155–164, 2010.

[200] C. Duan and J. Cleland-Huang, “Clustering support for automated tracing,” in *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 244–253, 2007.

[201] X. Chen and J. C. Grundy, “Improving automated documentation to code traceability by combining retrieval techniques,” in *26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 223–232, 2011.

[202] B. T. Armstrong, “Can clustering improve requirements traceability? a tracelab-enabled study,” 2013.

[203] H. Sultanov and J. H. Hayes, “Application of reinforcement learning to requirements engineering: requirements tracing,” in *21st IEEE International Requirements Engineering Conference, RE*, pp. 52–61, 2013.

[204] P. Domingos, “A few useful things to know about machine learning,” *Communications of the ACM*, vol. 55, no. 10, pp. 78–87, 2012.

[205] B. Dobing and J. Parsons, “How uml is used,” *Communications of the ACM*, vol. 49, no. 5, pp. 109–113, 2006.

[206] Drue Placette, *45 of the Top Source Code Repository Hosts*. Available at `https://blog.profitbricks.com/top-source-code-repository-hosts/`. Accessed March 2016.

[207] *Glossary of Terms*.

[208] M. J. Atallah and S. Fox, eds., *Algorithms and Theory of Computation Handbook*. Boca Raton, FL, USA: CRC Press, Inc., 1st ed., 1998.

[209] T. Munakata, *Fundamentals of the New Artificial Intelligence - Neural, Evolutionary, Fuzzy and More, Second Edition*. Texts in Computer Science, Springer, 2007.

[210] K. Kawaguchi, *Backpropagation Neural Networks*. Available at `http://www.ece.utep.edu/research/webfuzzy/docs/kk-thesis/kk-thesis-html/node18.html`. Accessed March 2016.

[211] C. Nasa and Suman, "Article: Evaluation of different classification techniques for web data," *International Journal of Computer Applications*, vol. 52, no. 9, pp. 34–40, 2012.

[212] *Class J48*. Available at `http://weka.sourceforge.net/doc.dev/weka/classifiers/trees/J48.html`. Accessed March 2016.

[213] J. C. Platt, "Sequential minimal optimization: A fast algorithm for training support vector machines," *Advances in Kernel Methods. Support Vector Learning*, vol. 208, pp. 1–21, 1998.

[214] R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI*, pp. 1137–1145, 1995.

[215] *Mathworks Documentation. Divide Data for Optimal Neural Network Training*. Available at `http://uk.mathworks.com/help/nnet/ug/divide-data-for-optimal-neural-network-training.html`. Accessed March 2016.

[216] S. Drazin, "Decision Tree Analysis using Weka," *Machine Learning-Project II, University of Miami*, pp. 1–3, 2010.

[217] D. I. K. Sjoberg, T. Dyba, and M. Jorgensen, "The future of empirical methods in software engineering research," in *Future of Software Engineering*, pp. 358–378, IEEE Computer Society, 2007.

[218] S. Easterbrook, "Empirical research methods for software engineering," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, pp. 574–574, 2007.

[219] P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case Study Research in Software Engineering: Guidelines and Examples*. Wiley Publishing, 1st ed., 2012.

[220] H. H. Liu, *Software Performance and Scalability*. Wiley-Blackwell, 2009.

[221] T. C. Lethbridge, S. E. Sim, and J. Singer, “Studying software engineers: Data collection techniques for software field studies,” *Empirical software engineering*, vol. 10, no. 3, pp. 311–341, 2005.

[222] R. Carvalho, “Visualising and analyzing software artefacts and their relationships.” unpublished thesis, 2016.