A Declarative Foundation for Querying the History of Software Projects

Reinout Stevens

May 2017

Abstract

The use of a Version Control System (VCS) is an industry best practice for developing software projects. VCS's enable developers to share and integrate changes made to the source code. As a side-effect, a VCS stores the entire development history of the versioned software project. This history is of interest to several stakeholders. Developers can use it to find answers to questions that might arise during the development. Examples of such questions are "Who introduced this piece of code?" or "Who has contributed to this failing class?". Researchers in the domain of mining software repositories can leverage the same history to analyze, spot trends, and retrieve actionable information about the development of software projects.

Although very insightful, software histories are too large to inspect manually. Support from a *general-purpose* history querying tool that satisfies the needs of the different stakeholders is in order. Such a tool automatically identifies the elements from a project's history that exhibit the characteristics specified in a given history query. We identify several criteria for such a general-purpose history querying tool. Among others, history querying tools should support the following characteristics in their queries:

Version Characteristics concern the different elements of a particular version of a project. Examples of these characteristics are the author, commit message, or the state of the source code of a particular version.

Temporal Characteristics concern the temporal quantification over the versions of the software project. Examples include quantifiers such as "for every version" or "after the first version that . . . ".

Change Characteristics concern the fine-grained edits made between two versions of the source code. Examples of these are the source code elements that have been inserted, moved, updated or removed.

Evolution Characteristics concern the effect of applying a (sub)sequence of changes to source code. Examples are all change sequences that have the same effect as a known refactoring or earlier transformation of the code.

We propose a declarative foundation to history querying that supports these different characteristics. Characteristics are expressed in logic queries, while a logic proof procedure identifies history elements exhibiting the specified characteristics. We create and design the general-purpose history querying tool QwalKeko that supports the characteristics in a uniform declarative language. We validate our work through different means. We validate the support for version and temporal characteristics through several usage scenarios stemming from the aforementioned stakeholders. We validate the support for change characteristics by conducting a large-scale empirical study in which we investigate the co-evolution of functional tests with the system under test. We conduct this study twice; once using our approach to history querying and once using a general-purpose programming language. This enables us to compare both implementations on the different concerns of the study. Finally, we evaluate the support for evolution characteristics by identifying instances of refactorings in different change sequences stemming from open-source projects, and ensuring that the identified change sequences are correct, minimal and executable.

Samenvatting

Het gebruik van versiebeheersystemen is standaard om softwareprojecten te ontwikkelen en onderhouden. Ze laten ontwikkelaars toe om gemaakte wijzigingen aan de broncode te verdelen onder ontwikkelaars. Als bijproduct bevat een versiebeheersysteem ook de geschiedenis van de broncode van het softwareproject. Deze geschiedenis is interessant voor verscheidene belanghebbenden. Softwareontwikkelaars kunnen deze geschiedenis gebruiken om antwoord te vinden op vragen zoals "Wie heeft deze code geïntroduceerd?" of "Wie heeft er bijgedragen aan deze falende klasse?". Onderzoekers actief in het domein van mining software repositories gebruiken deze geschiedenis om het ontwikkelingsproces te analyseren, om trends te vinden en toepasbaar advies te verkrijgen.

Zulke geschiedenissen zijn te groot en complex om efficiënt manueel te doorzoeken. Er is nood aan een *algemene* history querying tool die voldoet aan de verschillende noden van de belanghebbenden. Zulk een tool identificeert automatisch de elementen in de geschiedenis van een project die voldoen aan karakteristieken die gespecificeerd zijn in een history query. We identificeren verscheidene criteria voor zulk een algemene history querying tool. History querying tools moeten onder andere de volgende karakteristieken in hun queries ondersteunen:

Versiekarakteristieken betreffen de verschillende elementen van een versie. Voorbeelden van zulke elementen zijn de auteur, commit message of de staat van de broncode van een bepaalde versie.

Temporele karakteristieken betreffen de temporele quantificatie van elementen van verschillende versies. Voorbeelden zijn "voor elke versie" of "na de eerste versie die . . . ".

Wijzigingskarakteristieken betreffen de wijzigingen tussen twee versies van de broncode. Voorbeelden zijn de toevoeging, verplaatsing, aanpassing of verwijdering van broncode-elementen.

Evolutiekarakteristieken betreffen het effect van de toepassing van een sequentie van wijzigen op de broncode. Voorbeelden zijn de verzameling van change sequenties die hetzelfde effect als een gekende refactoring of eerder geziene transformatie van de broncode hebben.

We stellen een declaratieve fundering voor om de geschiedenis te bevragen die de verschillende karakteristieken ondersteunt. Karakteristieken worden uitgedrukt in logische queries, terwijl een logic proof procedure de geschiedenis-elementen identificeert die de gespecificeerde karakteristieken omvatten. We ontwikkelen en ontwerpen de algemene history querying tool QwalKeko die de verschillende karakteristieken ondersteunt in een uniforme taal. We valideren de ondersteuning van versie- en temporele karakteristieken door middel van verschillende gebruiksscenarios. We valideren de ondersteuning van wijzigingskarakteristieken door het uitvoeren van een empirische studie die de co-evolutie van functionele testen met het geteste systeem onderzoekt. We implementeren deze studie tweemaal; eenmaal in QwalKeko en eenmaal in een algemene programmeertaal. We evalueren beide implementaties door de verschillende onderdelen van de studie te vergelijken met elkaar. Tot slot valideren we de ondersteuning van evolutiekarakteristieken door verschillende instanties van refactorings te detecteren in verscheidene change sequenties, verkregen van open-source projecten, en te verzekeren dat de verkregen change sequenties correct, minimaal en uitvoerbaar zijn.

Acknowledgements

I want to thank the members of my jury (Julia Lawall, Xavier Blanc, Ann Dooms, Jan Hidders, Peter Vranckx and Theo D'Hondt) for their insightful comments and feedback.

I want to thank Coen for the support, guidance and feedback during the last five years (six if you count my master's thesis). I apologize for all of my poorly written drafts you had to crawl through, correct (I should have invested in red marker stocks), but that you turned into Shakespearean masterpieces.

I want to thank all my current and former colleagues. Laure and Eline, for being good friends. Simon, for having the nicest hairdo (next to Coen). Carlos, for being the flyest of dudes (and helping me with Eclipse). Florian, for making me look like a better person. Mattias and Jens, for being part of the holy trinity known as the IWT triangle, back when research was still hard and challenging. Everybody who distracted me: Dries, Kevin, Nathalie and Laure for the daily coffee, Tim, Kevin, Laurent and Joeri for the occasional swim and Ward for running with me.

I want to thank my friends for providing the necessary venting opportunities. Sean and Johan for our bi-weekly rhum&coke, retro-gaming solving-the-world's-problems evenings accompanied by top-notch music. Not many people can say that they have been friends since crèche and kindergarten. Karl and Karel, for the irregularly planned games of "kleurenwies" and king. Annelies for losing these games with the consistency that would make a Swiss clockmaker proud.

I want to thank my family of engineers that never did a PhD. Finally I have two extra characters in front of my name that (hopefully) compensate for my engineering degree from "den Aldi".

Even though you asked me not to put you in the acknowledgments, I want to thank Frederiekje for being an awesome girlfriend. I am sorry for all the deadlines, the lack of holidays, being grumpy from time to time, etc.

Finally, I want to thank Gustaaf and Nori for being little balls of fluff. Just stop eating my plants.

CONTENTS

1 INTRODUCTION

Contemporary software projects are not developed by an individual developer. Creating and maintaining a software project often requires many developers working in parallel on different tasks. One developer may be implementing new features, a different developer may be resolving bugs while still another developer may be implementing an experimental algorithm to improve the performance of the project. All these developers need to share their modifications among each other.

A project's source code is stored in a centralized, shared repository. A Version Control System (VCS) supports the management of such modifications. Developers have a copy of the project's source code on their computer. Developers can *commit* their local modifications, hereby recording the changes to their local copy of the repository — the structure holding the data about the versioned files —, and *push* them to a centralized copy on a server. Applying these changes to the source code results in a new *revision* of the source code. As a side effect, a VCS contains the history of the source code of the versioned software project.

1.1 Context: Potential Uses of History Information

The history information stored in a VCS can be used by several stakeholders. This section introduces these stakeholders and existing tool-supported approaches to querying this information.

1.1.1 Stakeholders of History Information

We introduce two stakeholders of history information; more specifically software developers and researchers working in the field of Mining Software Repositories.

Software developers Software developers need answers to a wide series of questions during the development of a software project [36, 51, 68]. Some of these questions can be answered using the history the project. Examples of questions from these studies are:

1. Who is the expert of this class?
2. Who has made changes to my classes?
3. What is the evolution of the code?
4. Who made a particular change and why?
5. Who is working on the same classes as I am?
6. What has changed between two builds and who has made these changes?

For example, the question "Who has made changes to my classes?" can be answered by retrieving the revisions in which the developer introduced one or more classes, and subsequently determining whether another developer modified any of those classes in a later revision.

Researchers Researchers working in the field of Mining Software Repositories (MSR) [7] aim to provide tangible recommendations about the development process of software by spotting trends and correlations in the evolution of the source code of software systems. The history information residing in version control systems is invaluable to their research. For example, Chen et. al [12] studied how Object-Relational Mapping (ORM) Code is maintained in Java systems, found that compatibility, performance, and security problems are more common to result in ORM code changes, and recommend better tooling support for such ORM systems. Christophe et al. [13] studied which parts of automated functional tests for web applications were most prone to changes over time, and recommend using the `PageObject` design pattern in such tests.

Both groups of stakeholders can leverage the information stored in Version Control Systems. This information can be extensive[1]. It is not feasible to manually inspect a VCS to retrieve the elements of interest (e. g., classes, methods, authors, code changes, etc.) and their history and evolution. The querying facilities of most Version Control Systems only support querying the history of the versioned source code. For instance, they support querying a textual representation of the

[1]At the time of writing, the project with the largest number of commits stored in GHTorrent [41] contains 1043116 commits, while the median number of commits of the top 50 largest projects is 151489.

source code using regular expressions. Regular expressions are useful for developers searching for elements featuring a particular character string. A regular expression cannot account for variations of code constructs (e. g., whitespace, formatting, or equivalent `for`/`while` loops, etc.). These querying facilities are insufficient for the aforementioned stakeholders. In general, they cannot be used to answer *general-purpose* program or history queries.

1.1.2 Tool Support for Querying Program and History Information

Program querying tools enable users to query a single revision of a software project. History querying tools enable users to query multiple revisions of a versioned software project. In what follows we introduce both.

Program Querying Tools Finding source code elements using the tools provided by an integrated development environment (IDE) would be cumbersome. They provide support for some commonly required information (e. g., the callers of a method, the type hierarchy of a class), but they do not provide a more general-purpose means to launch queries against source code. This problem is addressed by a program querying tool. Program querying tools (e. g., [15, 20, 43, 55]) feature a dedicated language for users to specify the characteristics of the elements of a program's source code they are interested in, and return the elements that adhere to this specification. For example, a developer could specify the characteristics of methods that may return a `null` object, or that access a certain variable (directly or indirectly) and have these methods returned by the tool.

The following example query, expressed using EKEKO [18, 19], returns all `return` statements that return `null`.

```
(ekeko* [?return ?value]
  (ast :ReturnStatement ?return)
  (has :expression ?return ?value)
  (ast :NullLiteral ?value))
```

Solutions to the query consist of pairs of a `return` statement and the `null` value it returns; the bindings for variables `?return` and `?value` respectively. In each solution, variable `?return` is bound to a return statement that returns the value `?value`, which must be a `null` literal.

However, program querying tools are still limited to querying a single revision of a software project. They cannot be used to query the history of a software project: users needs to find an ad-hoc solution to retrieve and combine results stemming from different revisions. They do form a good starting point as a foundation for history querying; users need only specify the characteristics of the sought-after code elements, and are not encumbered with implementing an operational search through the code.

History Querying Tools A history querying tool (e. g., [25, 46, 58, 75]) extends the notion of a Program Querying Tool with support for history information. To this end, it reifies the information stored by a VCS so that this information can be queried. Queries no longer only concern code elements but also the versions of these elements. For example, the following QwalKeko [71–73] (the history querying tool presented in this dissertation) query returns all classes named `Evaluator` and the revision in which such a class has been introduced by a developer:

```
1 (qwalkeko* [?class ?end]
2   (qwal vcs start ?end []
3     ;;skip all versions in which the class is absent
4     (q=>*
5       (in-source-code [version]
6         (class-named|absent "Evaluator")))
7     ;;class is present
8     (in-source-code [version]
9       (class-named ?class "Evaluator"))))
```

The query describes a path of successive revisions in the history of the versioned software project represented by the variable `vcs`, starting in revision `start` and ending in revision `?end`. Line 4 skips an arbitrary number of revisions in which the `Evaluator` class is absent until a revision is encountered in which the `Evaluator` class is present. This class is bound to `?class` and, together with the revision `?end`, returned as a solution to the user.

1.2 Problem Statement

Querying a *single* revision of a software project has long since been supported by program querying tools. Querying *multiple* revisions of a software project is only supported to a *limited* extent by existing history querying tools. Most existing history querying tools only support querying a coarse-grained representation of the history of the queried software project. They do not support querying the history of the complete source code of a software project (i. e., up to the level of individual abstract syntax tree nodes), they do not feature a uniform language that supports query reuse, abstraction and composition, and they do not support querying the edit operations that have been made between two revisions.

We examine the source code information required by our stakeholders for performing existing Mining Software Repositories studies and for answering developers' questions regarding the evolution of a software project. From this examination we discern four kinds of characteristics that must be supported by a *general-purpose* history querying tool. For instance, in a Mining Software Repositories study Ray et al. [64] define, identify and study changes to the source code that are unique in the history of a software project. Thus, performing this study using a history querying tool requires support for specifying the characteristics of changes made to the abstract syntax tree between two revisions (i. e., change characteristics). In a study regarding the questions of software developers Latoza et al. [51] conducted a survey about hard-to-answer questions that developers ask about their code. The survey includes questions regarding code changes (e. g., debugging, implementing features, code history), code elements (e. g., intent of code, method properties, performance) and their element relationships (e. g., contracts, control flow). Answering these questions using a history querying tool requires, at a minimum, support to specify the characteristics of code elements (i. e., revision characteristics) and their temporal relationships (i. e., temporal characteristics).

The four discerned characteristics are the following:

Revision Characteristics Revision characteristics concern elements from a single revision. Examples of these elements are the author of a revision, a class or an if-statement. Examples of these characteristics are the name of the author, the parent class of a class or the expression of an if-statements. These characteristics can be used in queries to express conditions such as "a revision introduced by the author named Bob", "a class inheriting from `Person`" or "an if-statement that performs a null check".

Temporal Characteristics Temporal characteristics concern the temporal quantification between elements from different revisions. Examples of these characteristics are "after a particular revision", "before a revision in which. . . ", "in every revision", "until a revision in which. . . ". These characteristics can be used in queries to express conditions such as "in what pair of successive revisions is the class `Person` present and then absent?" or "does every class have a corresponding unit test in every version?".

Change Characteristics Change characteristics concern fine-grained changes that have occurred between two revisions of a file. Examples of these changes are the insertion or removal of an AST node. Examples of such characteristics are the index at which an AST node has been inserted, the AST node that has been removed from the code, an update to the modifier of a method. These characteristics can be used in queries to express conditions such as "what if-tests have been introduced?" or "what methods have been removed?".

Evolution Characteristics Evolution characteristics concern code transformations that are implemented by change sequences. Examples of such code transformations are refactorings or systematic edits. Examples of their characteristics are the intermediate states of the source code as it undergoes the transformation. These characteristics can be used in queries to express conditions such as "do these changes implement a Remove Unused Method refactoring?", or "what additional changes have been applied next to this code clone removal?".

These characteristics need to be supported for a history querying tool to be general-purpose. Existing history querying tools do *not* support all these characteristics. Without support for revision characteristics, sought-after revision elements cannot be specified by a user and need to be found manually by inspecting the revision. Without support for temporal characteristics, a user needs to manually configure a program querying tool to retrieve revision elements in the correct revision. Without support for change characteristics, a user cannot specify the changes to be found among those that occurred between two revisions of the code. They then need to be found manually by inspecting the change sequences or edit scripts. Without support for evolution characteristics, users need to account for the different possible change sequences (i. e., the different possible change operations and their permutations) that implement the same source code transformation to ensure detecting all possible instances of that source code transformation. Evolution characteristics are difficult to support because a source code transformation can be implemented by different change sequences that, when applied, have the same end result.

Both groups of stakeholders require support in the form of a tool that enables specifying the characteristics of sought-after history elements, and that subsequently returns the corresponding elements adhering to the given specification. This tool needs to support these characteristics in a *uniform* language to lower the learning curve for the stakeholders.

1.3 Contributions

This dissertation makes the following contributions:

1. We identify and motivate the different criteria for a *general-purpose* history querying tool that serves the needs of stakeholders in information about the history of a software project.

2. We design and implement a history querying tool called QwalKeko that satisfies these criteria. It has a declarative foundation: characteristics are expressed in logic queries, while a logic proof procedure identifies history elements exhibiting the specified characteristics. Unique to the approach is the use of regular path expressions [17] for specifying paths through different graph structures, and the use of logic conditions within such a regular path expression specifying the characteristics nodes along this path must exhibit. Temporal characteristics are specified using paths through a revision graph. In this graph nodes correspond to revisions and edges connect successive revisions. Evolution characteristics are specified using paths through an evolution state graph. In this graph a node corresponds to the intermediate abstract syntax tree that can be constructed from applying a subsequence of

changes, an edge corresponds to a single change, and two nodes are connected via an edge when the application of that edge transforms its source's abstract syntax tree into its target's syntax tree. This particular specification enables our logic proof procedure to recognize instances of the same code transformation in change sequences of which the order, number and operations of changes differ.

3. We validate that QwalKeko satisfies the different criteria for a general-purpose history querying tool, and thus serves the needs of the history information stakeholders, through example queries and empirical studies that are representative for its intended use. For some studies, the results are scientific contributions by themselves.

1.4 Outline of the Dissertation

Chapter 2 introduces and motivates the different criteria for a general-purpose history querying tool by looking at the history information required by recent studies in the domain of mining software repositories and the history information required to answer developers' questions regarding the history of a software project.

Chapter 3 discusses the state of the art in querying software, its history and its evolution. We motivate why existing approaches do not satisfy all the criteria for a general-purpose history querying tool.

Chapter 4 provides a high-level overview of our approach to history querying. It introduces the different components of QwalKeko, which are then discussed in detail in the subsequent chapters.

Chapter 5 introduces the declarative foundation of QwalKeko. It integrates the graph querying language Qwal [72] with the logic program querying tool Ekeko [19]. Ekeko supports querying Java projects in an Eclipse workspace. Qwal supports querying paths through a graph-based representation of a VCS using regular path expressions [17]. The combination of both supports revision and temporal characteristics in history queries. We evaluate this foundation by providing queries for several scenarios of the different stakeholders. We have published about Qwal at several conferences [50, 72, 74].

Chapter 6 extends QwalKeko with support for change characteristics using ChangeNodes. ChangeNodes implements a change distilling algorithm. Such an algorithm takes as input two revisions of a source code file, and outputs a sequence of changes that, when applied, transforms the first revision of the file into the second revision. We extend QwalKeko with support for change characteristics by providing a declarative API on top of ChangeNodes and its output. We evaluate this extension by performing a mining software repositories study on the evolution of functional automated tests for web applications. We have two implementations of this experiment; one using Clojure and one using QwalKeko. This enables us to compare both implementations regarding the different concerns of the study. The results of the Selenium study are published in a conference paper [13] and is accompanied by a tool demonstration paper [72].

Chapter 7 extends QwalKeko with support for evolution characteristics. The chapter discusses the problems when specifying a sought-after source code transformation in terms of change characteristics. The main problem is the change equivalence problem, which stipulates that different change sequences can implement the same conceptual transformation. To support evolution characteristics, we transform the output of ChangeNodes into a graph of all possible ASTs that can be constructed by applying the changes in a change sequence. This enables specifying source code transformations as a path through this graph of potential intermediate ASTs, and the conditions ASTs along this path must adhere to. To ensure the uniformity of our history querying language, these paths are, just like the paths through the revision graph, specified using regular path expressions. QwalKeko returns a *minimal, executable* subsequence of changes that implement the desired source code transformation. We evaluate this approach by identifying instances of the same refactoring and the changes implementing this refactoring in several commits to different open-source projects. This approach to supporting evolution characteristics in a history querying tool is published in a conference paper [73].

1.5 Publications Supporting this Dissertation

Supporting Publications

1. R. Stevens, C. De Roover, C. Noguera, A. Kellens, and V. Jonckers. A logic foundation for a general-purpose history querying tool. Elsevier Journal on Science of Computer Programming, 2014.

   This journal paper presents a prototype of our logic foundation to history querying, supporting coarse-grained revision characteristics and temporal characteristics. It supports Chapter 5.

2. L. Christophe, R. Stevens, C. De Roover and W. De Meuter. Prevalence and maintenance of automated functional tests for web applications. In Proceed-

ings of the 30th International Conference on Software Maintenance and Evolution (ICSME14), 2014.

This conference paper presents a mining software repositories study regarding the evolution of automated functional unit tests performed using our history querying tool. The study validates our support for change characteristics in history querying. This paper supports Chapter 6.

3. R. Stevens, C. De Roover. Extracting Executable Transformations from Distilled Code Changes. In the Proceedings of the 24th International Conference on Software Analysis, Evolution, and Reengineering (SANER17), 2017.

   This conference paper presents our approach to supporting evolution characteristics in history querying. It supports Chapter 7.

Related Publications

1. A. Kellens, C. De Roover, C. Noguera, R. Stevens, and V. Jonckers. Reasoning over the evolution of source code using quantified regular path expressions. In Proceedings of the 18th Working Conference on Reverse Engineering (WCRE11), 2011.

2. R. Stevens, C. De Roover, C. Noguera, and V. Jonckers. A history querying tool and its application to detect multi-version refactorings. In Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR13), 2013.

3. C. De Roover and R. Stevens. Building development tools interactively using the Ekeko meta-programming library. In Proceedings of the 18th European Conference on Software Maintenance and Reengineering (CSMR14), Tool Demo Track, 2014.

4. R. Stevens and C. De Roover. Querying the history of software projects using QwalKeko. In Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME14), Tool Demo Track, 2014.

5. R. Stevens. A declarative foundation for comprehensive history querying. In Proceedings of the 37th International Conference on Software Engineering, Doctoral Symposium Track (ICSE15), 2015.

2 BACKGROUND ON HISTORY QUERYING

The goal of this chapter is to discern the criteria of a general-purpose history querying tool. First, Section 2.1 introduces version control systems, which contain the history of a software project. Next, Section 2.2.1 discusses examples of existing history querying approaches. Section 2.2.2 discusses the different characteristics that should be supported by a general-purpose history querying tool These characteristics are *revision*, *temporal*, *change* and *evolution* characteristics. Section 2.3 introduces two application domains for history querying. The first application is program comprehension. In this application the history of a software project is leveraged to provide a better understanding of its current state. Existing research on program comprehension has already identified the kind of questions developers ask during the development and maintenance of a software project [14, 36, 68]. A subset of those questions can only be answered using information about the history of the software system under development. The second application consists of empirical studies, which can make use of the history of software projects to confirm existing or identify new best practices in software engineering. Section 2.4 defines, based on the application domains, the different criteria a history querying approach must adhere to for it to be general-purpose.

2.1 Version Control Systems

We begin this chapter with an introduction to version control systems, as they are the source of historical information of our approach. Version Control Systems (VCS) enable developers to work in parallel on a software project. A wide series of VCS exist, each with their own specialized features. In this section we discuss the elements shared by most of these VCS. Figure 2.1 depicts a meta-model of the information stored by most VCS. This representation is based on the one used by Evolizer [37] (cf. Section 3.2). Several commits are stored for a project. A commit contains a set of changes a single *author* made to the project. These changes can either be the addition, modification or removal of a file. Changes made to an existing file are represented in terms of addition or removal of lines of text. A line is represented by combining an addition and removal. In order to develop a software system collaboratively, an initial commit is created. We call this the *root* revision of the system. A developer working on the project can *checkout* a revision. As a result, he gets a local copy of that revision on his working station. Every revision is assigned a unique identifier so that developers can specify what revision they want to checkout. He can modify the code and *commit* or *push* his modifications to the VCS. To document the intent of the revision he may add a *commit message* to the commit. Finally, other developers can retrieve his changes. In case two people have modified the same file simultaneously a *merge conflict* may occur. In this case, the developer has to manually inspect both revisions of the code and commit the correct revision in which all changes have been integrated.

VCS enable developers to implement experimental features in a separate *branch*. Branches enable developers to develop different variants of the system in parallel. Two branches can be *merged*, so that all the modifications made in both branches are present. In case both branches modified the same files a merge conflict may occur as well.

2.2 Querying Version Control Systems for History Characteristics

In this section we define the term "history query" and provide several example queries supported by existing approaches. A history query describes elements from the history and evolution of a software project. These elements, and thus the answer to a history query, are either *VCS* meta-elements (i. e., the author, the commit message, the timestamp of a revision), *revision* elements (i. e., a class, a method or any arbitrary Abstract Syntax Tree (AST) node of a revision) or *change* elements (i. e., the insertion or removal of an AST node). A history query describes the characteristics these returned elements must exhibit in a dedicated language.

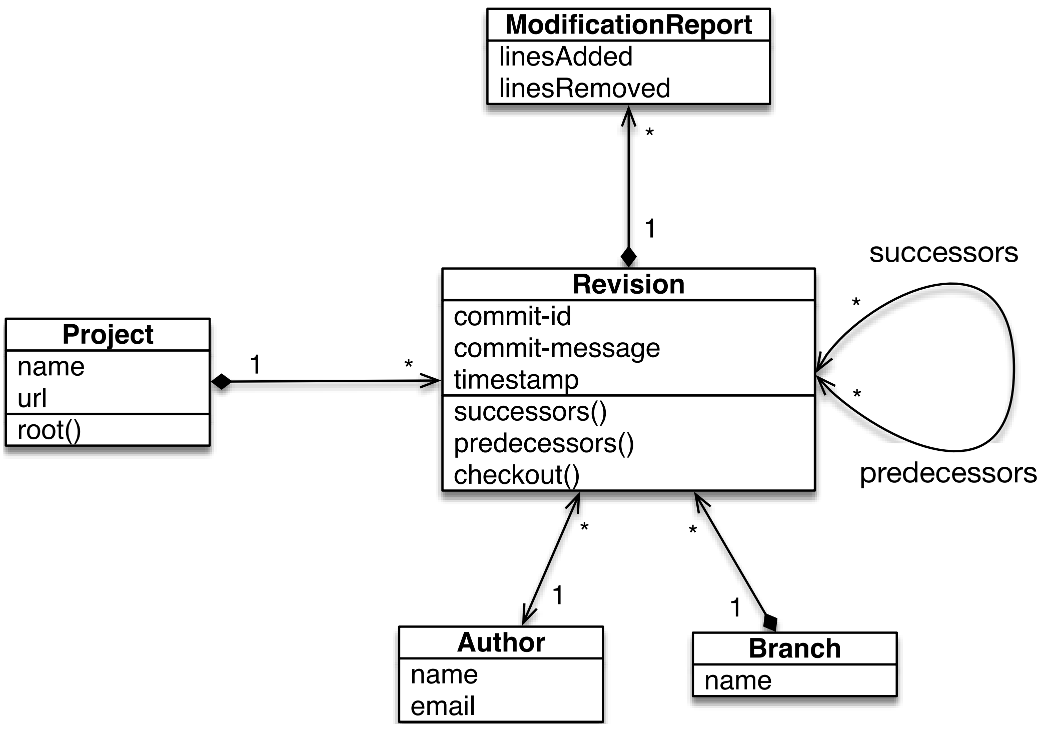


Figure 2.1: UML diagram depicting a meta-model of a Version Control System, based on the meta-model of Evolizer [37].

The identification of these elements is done by the history query tool, which accepts history queries from the user. This dissertation focuses on tool support for querying the history of the *source code* of software projects. Other sources of information, such as bug trackers, mailing lists, etc. are not considered.

2.2.1 Examples of History Querying

Figure 2.2 depicts an example query expressed in the Boa language [25]. Boa enables users to query the history of multiple software projects using the Visitor pattern. Boa runs this query in parallel across multiple projects using MapReduce [21]. The depicted example computes the number of authors the queried software projects have. For each project, a task is created with its own variable space. The first line introduces a task-local variable `p` that is bound to the corresponding project of that task. The second line defines an output variable `counts`. Such a variable is shared across the different tasks, and is the result of the query. Each project is assigned a unique index, and this array stores the number of authors for a project at this index. Lines 5–12 implement a Visitor that visits every revision of the queried software projects. This Visitor checks whether the author of the visited revision is a new author for that project, and if so increments the number of authors for that project. Boa is an example of an imperative history query language.

```
p: Project = input;
counts: output sum[string] of int;

committers: map[string] of bool;

visit(p, visitor {
        before node: Revision ->
                if (!haskey(committers, node.committer.username)) {
                        committers[node.committer.username] = true;
                        counts[p.id] << 1;
                }
})
```

Figure 2.2: History query expressed in Boa for computing the number of authors that worked on each software project. This query is taken from the Boa example page.

Figure 2.3 depicts a SCQL [46] query that identifies whether an author `a` has only modified files created by author `b`. To this end, it uses a declarative specification language that returns the different solutions for every logic variable. The language features the existential quantifier `E` and the universal quantifier `A` as well as a large library of predicates that can be used in a query. The first two lines specify that two authors `a` and `b` must exist. Line 3 specifies that these authors must be different. Line 4–7 specify that for all revisions `r` created by author `a`, all files `f` of these revisions must be created by author `b` in a revision `r2` that was created before revision `r`.

```
1  E(a, Author) {
2    E(b, Author) {
3      a!=b &&
4      A(r, a.revisions) {
5        A(f, r.file) {
6          Ebefore( r2, f.revisions, r) {
7            isAuthorOf( b, r2)
8          }
9        }
10     }
11   }
12 }
```

Figure 2.3: History query expressed in SCQL that computes the different authors A that only modify files introduced by author B. This query is taken from [46].

Figure 2.4 depicts a query written in QwalKeko [72], our declarative history querying language. A QwalKeko query describes a sequence of commits, and the properties of elements that reside in these commits. The query retrieves all methods that have ever been present in any revision of the software project. Solutions to the query consist of all bindings for the logic `?method` and `?version` that make the query's conditions succeed. Line 2 launches a history query over the representation of a software project, bound to `project`, starting in version `root` and ending in `?version`. Line 3 skips an arbitrary number of revisions without specifying any revision characteristics. Lines 4–5 state that `?method` must unify with a method declaration present in the current revision `curr` of the software.

```
1 (qwalkeko* [?method ?version]
2   (qwal project root ?version []
3     (q=>*)
4     (in-source-code [version]
5       (ast :MethodDeclaration ?method))))
```

Figure 2.4: History query expressed in QwalKeko that retrieves all methods that have ever been present in any version of a project.

2.2.2 Types of History Characteristics

A general-purpose history querying tool should facilitate its users in expressing a wide variety of characteristics of the sought-after solution. We first introduce these characteristics, and then discuss them in detail. We discern the following characteristics, based on the applications of history querying discussed in Section 2.3.

Revision Characteristics Revision characteristics concern the properties elements of a single revision must exhibit. Examples of such elements are the author of the revision, a class, or any arbitrary AST node. Examples of such characteristics are the name of an author, the class hierarchy to which a class belongs etc. These characteristics can be used in a history query to find, for instance, modification records by a particular author, a class that implements a particular design pattern, all methods that are recursive etc.

Temporal Characteristics Temporal characteristics concern quantification over elements from different revisions. Examples of such quantification are "for every revision", "after a particular revision", "for every pair of successive revisions", etc. These characteristics can be used in queries to express conditions such as "is the class `Person` present in every revision?" or "does every class have a corresponding unit test in every revision?".

Change Characteristics Change characteristics concern fine-grained source code changes that occurred between two revisions. Source code changes can be caused by edits or IDE interactions performed by the developers. Examples of such changes are the insertion, deletion, move or update of AST nodes. Examples of such characteristics are the index at which an AST node has been inserted, the AST node that has been removed from the code, an update to the modifier of a method. These characteristics can be used in queries to express conditions such as "a method declaration node that was inserted", "an if test that was removed" or "a method body that was moved to a different location".

Evolution Characteristics Evolution characteristics concern high-level change patterns that occurred between two revisions of a file. The difference between change characteristics and evolution characteristics is that the first concern individual changes, whereas the latter concern the effect of a sequence of changes. Examples of such characteristics are code transformations, such as a refactoring or a systematic edit, that are implemented by a sequence of changes. These characteristics can be used in queries to express conditions such as "does this change sequence implement a field rename refactoring?", or "does this change sequence implement a code clone removal?".

We will argue that a general-purpose history querying tool must enable expressing these different characteristics in a uniform language. Among these characteristics, the revision and temporal characteristics are orthogonal. The revision characteristics concern elements of a single revision, which must be retrieved from a particular revision. The temporal characteristics concern from what revision the elements are retrieved, but do not concern the characteristics of these elements. Change and evolution characteristics concern operations between two successive revisions of the same code. As these operations are applied to concrete source code, the characteristics describing these operations also concern the affected source code. As such, specifications of change characteristics and source code characteristics (which are part of the revision characteristics) are necessarily tangled. *We envision a history querying tool that offers dedicated features for each of the aforementioned characteristics.*

In what follows we discuss the different characteristics in detail.

Revision Characteristics

The revision characteristics concern the revision or VCS elements that reside in a single revision of the queried software project. We discern three categories, namely: revision meta-data characteristics, coarse-grained revision characteristics and fine-grained revision characteristics.

Revision Meta-Data Characteristics Revision Meta-Data Characteristics (RMC) concern elements that stem from the meta-data kept by the underlying VCS. Examples of these elements are the author, the timestamp, the commit message, the modified files etc. of a revision. Examples of these characteristics are the name of the author, the words of a commit message or the path of a modified file. These characteristics can be used in queries to express conditions such as "a commit pushed by the author Alice", "a commit after the second of June", "a commit message containing the word 'fix' or 'bug"' or "a commit modifying a file in the 'tests' directory".

Coarse-grained Revision Characteristics Coarse-grained Revision Characteristics (CRC) concern elements that stem from a coarse-grained representation of the source code. For object-oriented programming languages, such a representation could be a Hismo [39] model of the versioned software project. A Hismo model represents the history of a project up until the level of methods. For instance, the classes each package is composed of and the methods or fields within each class. Relations between these entities are included as well. For instance, the callees and callers of each method. The key point is that the complete source code is not available, and information that is not stored in the model cannot be retrieved. Examples of coarse-grained characteristics are the name of a class, the methods present in a class, or the signature of a method. These characteristics can be used in queries to express conditions such as "a class with at least 10 methods", "a method named `foo`" or "a method that calls itself recursively".

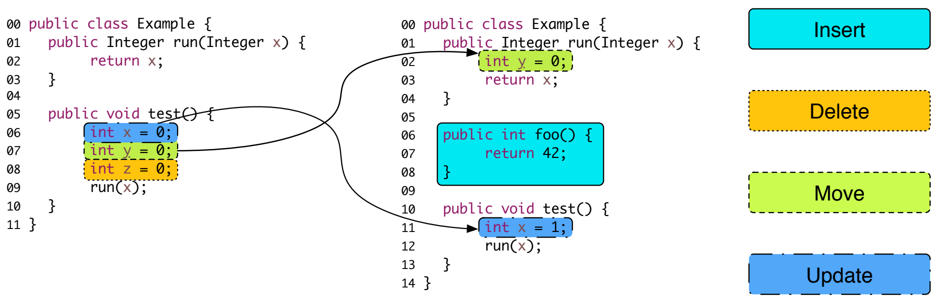
Fine-Grained Revision Characteristics Fine-grained Revision Characteristics (FRC) concern the individual source code elements in a revision. Examples of these elements are any type of AST node, such as a method invocation, an if test, a return statement etc. Examples of their characteristics are the name of the invoked method, the expression of the if test or the value of the return statement. These characteristics can be used in queries to express conditions such as "a method invocation of the method named `launchMissile`", "an if test performing a null-check" or "a return statement returning `0`".

Temporal Characteristics

Temporal Characteristics (TC) concern the temporal quantification over elements from different revisions. Examples are "for every revision", "after a particular revision", "for every pair of successive revisions", "until a revision in which...", "for every revision after a particular revision" etc. These characteristics can be used in queries to express conditions such as "for every revision the class `Person` must be present", "a bug fix must be introduced eventually after the revision that introduced that bug", "until a class is removed it must have a corresponding unit test" or "every revision after the release of Java must enumerate elements of a collection using internal rather than external iteration constructs (i. e., a `forEach()` rather than a `for`-loop)."

Change Characteristics

Change Characteristics (CC) concern changes to source code. Examples of these changes are the operations that were performed by a developer in order to transform the source code from one revision into the code of a different revision. Examples of their characteristics are the type of the operation such as an insert, a delete



```
00 public class Example {
01   public Integer run(Integer x) {
02       return x;
03   }
04
05   public void test() {
06       int x = 0;
07       int y = 0;
08       int z = 0;
09       run(x);
10   }
11 }
```

```
00 public class Example {
01   public Integer run(Integer x) {
02       int y = 0;
03       return x;
04   }
05
06   public int foo() {
07       return 42;
08   }
09
10   public void test() {
11       int x = 1;
12       run(x);
13   }
14 }
```

Figure 2.5: Figure depicting changes on two revisions of the same source code

or a move. These characteristics can be used in queries to express conditions such as “a field that was introduced by an insert”, “a method of which the body was moved to the consequent of an if-test” or “an unused method that was deleted”.

Figure 2.5 depicts two revisions of the same source code, and the change operations performed on those ASTs. It depicts two revisions of the class `Example`. The variable declaration `int x = 0` is updated to a new initial value. The variable declaration `int y = 0` is moved to a different method. The variable declaration `int z = 0` is no longer present, and thus removed. Finally, a new method `foo` is inserted.

We outline three different ways to obtain information about source code changes: *line differencing*, *change logging* and *change distilling*.

Line Differencing Line Differencing computes the changes between two versions of a file in terms of the addition and removal of lines of text. One well-known line differencing tool is `diff`, which takes as input two files, and outputs what lines need to be added or removed to transform the first file into the second file. The same functionality is also provided by most VCS. Line differencing would result in the move of line 7 to line 2 of Figure 2.5 to be represented as a combination of addition and removal instead of a move.

Change Logging A change logger, such as FLUORITE [77], SpyWare [66], CodingTracker [60], and Syde [44], records the different operations a developer performs inside his IDE *while* he is developing the project. Change are represented by the keystrokes the developer performed, as well as interactions with the IDE (such as invoking a refactoring tool). A change logger returns accurate edit operations, including edits that are invisible in the final revision (e. g., adding and later removing a piece of code). Changes also span the source code of the complete project, such as moving pieces of code from one file to a different one.

Change Distilling A change distiller, such as CHANGEDISTILLER [33], CHANGENODES [72] or GUMTREE [30], is an algorithmic approach to procure changes *post factum*. The algorithm takes as input two revisions of a file, and outputs a sequence of changes. To this end, it uses heuristics, such as the textual similarity of two AST nodes, to determine whether nodes were modified. The different possible change types are inserts, moves, deletes and updates of AST nodes.

Line differencing, change logging and change distilling differ in how well the resulting changes correspond to the actual actions the developer performed. Change logging produces an accurate representation of how the developer implemented the modifications to the source code. It is, however, not readily accessible for most projects as the change logger must be installed prior to the changes being performed. There are also privacy concerns as it includes invisible edits. A change distiller can be used for any versioned software project, but its use of heuristics can sometimes produce non-minimal change sequences, nor do these edit scripts match with the actual modifications the developer performed. Line differencing tools such as `diff` work at the level of complete lines rather than individual source code elements on these lines, and do not provide `move` operations. Thus, they produce the least accurate changes.

Evolution Characteristics

Evolution characteristics concern instances of patterns that are the result of applying change sequences. Examples of such evolution patterns or code transformations are refactorings or systematic edits. Examples of their characteristics are the intermediate states of the source code as it undergoes the transformation. These characteristics can be used in queries to express conditions such as "do these changes implement a Remove Unused Method pattern?", or "what additional changes are applied next to this code clone removal?".

The difference between change characteristics and evolution characteristics is that change characteristics concern *individual* changes, whereas evolution characteristics concern the behavior of a change sequence as a whole. We illustrate this by means of an example. Listing 2.1 depicts a source code extract from the ANT project, in which methods `setIncludes` and `setExcludes` contain a piece of cloned code. Listing 2.2 depicts the next revision of this source code, in which the cloned code has been extracted to a new method `normalizePattern`, and each clone instance has been replaced by an invocation to that method. This "extract method refactoring in order to remove a code clone" has occurred in several projects. Instances of this pattern have concrete change sequences that do not necessarily consist out of identical change operations. As such, describing these patterns using only change characteristics is hard as one needs to take these differences into account.

```java
public void setIncludes(String[] includes) {
  if (includes == null) {
    this.includes = null;
  } else {
    this.includes = new String[includes.length];
    for (int i = 0; i < includes.length; i++) {
      String pattern;
      pattern =
        includes[i].replace('/', File.separatorChar).replace(
              '\\', File.separatorChar);
      if (pattern.endsWith(File.separator)) {
        pattern += "**";
      }
      this.includes[i] = pattern;
    }
  }
}

public void setExcludes(String[] excludes) {
  if (excludes == null) {
    this.excludes = null;
  } else {
    this.excludes = new String[excludes.length];
    for (int i = 0; i < excludes.length; i++) {
      String pattern;
      pattern =
        excludes[i].replace('/', File.separatorChar).replace(
              '\\', File.separatorChar);
      if (pattern.endsWith(File.separator)) {
        pattern += "**";
      }
      this.excludes[i] = pattern;
    }
  }
}
```

Listing 2.1: Source containing cloned code. This code is present in commit `6bdc259c2e818e1c86f944cbd8950e670294d944`

```java
public void setIncludes(String[] includes) {
  if (includes == null) {
    this.includes = null;
  } else {
    this.includes = new String[includes.length];
    for (int i = 0; i < includes.length; i++) {
      this.includes[i] = normalizePattern(includes[i]);
    }
  }
}

public void setExcludes(String[] excludes) {
  if (excludes == null) {
   this.excludes = null;
  } else {
    this.excludes = new String[excludes.length];
    for (int i = 0; i < excludes.length; i++) {
      this.excludes[i] = normalizePattern(excludes[i]);
    }
  }
}

private static String normalizePattern(String p) {
  String pattern = p.replace('/', File.separatorChar)
                 .replace('\\', File.separatorChar);
  if (pattern.endsWith(File.separator)) {
    pattern += "**";
  }
  return pattern;
}
```

Listing 2.2: Source after extracting the cloned code. This code is present in commit `28d39b09a766fbb0dd3ca9b65ac06edf89075e8e`

2.3 Applications of History Querying

In this section we provide two application domains that require reasoning about the history of a software project. The first domain we discuss is the domain of program comprehension. In this domain, the history of a software project is leveraged to answer questions developers ask. Next, we discuss the domain of mining software repositories (MSR). According to the Mining Repositories Conference this domain focuses on the ways in which mining these repositories "can help to understand software development, to support predictions about software development, and to plan various aspects of software projects" [23].

2.3.1 Program Comprehension

Developers need answers to a variety of questions regarding the software system they are developing and maintaining, as indicated by several studies [14, 36, 51]. From this research we discern the developer questions that could be answered using a history querying tool. Table 2.1 summarizes these studies by depicting the

Table 2.1: Table depicting an overview of questions that developers ask that can be answered by looking at the history of a software project, taken from [14, 36, 51]. The columns indicate the question number, paper from which it was taken, the question itself, and whether it requires expressing VCS Meta-Data characteristics (VMC), coarse-grained revision characteristics (CGRC), fine-grained revision characteristics (FGRC) or change characteristics at the level of line changes (LiC), distilled changes (DC) or logged changes (LoC).

No.	Ref	Question	VMC	CGRC	FGRC	LiC	DC	LoC
Rationale								
QR1	[51]	Why was it done this way?	✓	✗	✗	✗	✗	✗
QR2	[51]	Why wasn't it done this other way?	✓	✗	✗	✗	✗	✗
QR3	[51]	Was this intentional, accidental, or a hack?	✓	✗	✗	✗	✗	✗
QR4	[36]	Who made a particular change and why?	✓	✗	✗	✓	✓	✓
QR5	[14]	Why is this "this way"? [Recover the rationale behind a snippet of code]	✓	✗	✗	✗	✗	✓
Evolution								
QV1	[51]	When, how, by whom, and why was this code changed or inserted?	✓	✗	✗	✓	✓	✓
QV2	[51]	What else changed when this code was changed or inserted?	✓	✗	✓	✓	✓	✓
QV3	[51]	How has it changed over time?	✓	✗	✗	✗	✓	✓
QV4	[51]	Has this code always been this way?	✓	✗	✗	✓	✓	✓
QV5	[51]	What recent changes have been made?	✓	✗	✗	✓	✓	✓
QV6	[36]	Who has made changes to my classes?	✓	✓	✓	✗	✗	✗
QV7	[36]	What classes have been changed?	✓	✓	✓	✗	✗	✗
QV8	[36]	Which code has recently changed that is related to me?	✓	✗	✗	✓	✓	✓
QV9	[36]	What is the most popular class? [Which class has been changed most?]	✓	✓	✓	✗	✗	✗

No.	Ref	Question	VMC	CGRC	FGRC	LiC	DC	LoC
QV10	[51]	Did my teammates do this?	✓	✗	✓	✓	✓	✓
Code Expertise								
QX1	[36]	Who created the API that I am about to change?	✓	✗	✓	✗	✗	✗
QX2	[36]	Who owns this piece of code? Who modified it the latest?	✓	✗	✓	✓	✓	✓
QX3	[36]	Who owns this piece of code? Who modified it most?	✓	✗	✓	✓	✓	✓
QX4	[36]	Who to talk to if you have to work with packages you have not worked with?	✓	✓	✓	✗	✗	✗
QX5	[51]	Who is the owner or expert for this code?	✓	✗	✓	✓	✓	✓
Branching								
QB1	[51]	How can I move this code to this branch?	✓	✗	✓	✗	✓	✓
QB2	[51]	Have changes in another branch been integrated into this branch?	✓	✗	✓	✗	✓	✓

questions that could be answered using a history querying tool and the characteristics required to answer such question. We denote whether the question can be answered using line-level changes (LiC) as their change representation, distilled changes (DC) or logged changes (LoC). The table does not include temporal characteristics, as they are required for every question, nor does it contain evolution characteristics as we found no question requiring these. Evolution characteristics are used for empirical studies, which we discuss in Section 2.3.2.

In 2010 Fritz et al. [36] conducted an empirical survey, asking eleven professional developers what questions they have while working on a project, for which there is no tool support. To this end, the authors first demonstrated a small tool that provided information about source code, change sets, work items and team members. Given such a tool, they asked developers what kind of questions they would want answered. From these interviews handwritten notes were taken, and from these notes the developer questions were distilled. The study resulted in 46 different questions (including questions not related to the history). They subsequently categorized these questions as follows: questions related to source code, to change sets, to teams, to work items, to comments, to web/wiki, to stack traces and to test cases.

In 2010 Latoza et al. [51] conducted a survey about hard-to-answer questions that developers ask about their code. They invited a random sample of around 2000 developers at Microsoft, to which 469 responded. The survey includes questions regarding code changes (e. g., debugging, implementing features, code history), code elements (e. g., intent of code, method properties, performance), element relationships (e. g., contracts, control flow), and an open response item about other hard-to-answer questions. The referenced paper focuses on answers to this free response item, which was filled in by 179 developers. In order to analyze responses, the authors clustered the questions into categories using the underlying intent of the question. We analyzed the different questions and identified the ones that require history information in order to answer them.

In 2015 Codoban et al. [14] conducted a study about the motivations of developers to use and examine the history of their software projects. To this end, 14 developers from 11 different companies were interviewed in a semi-structured manner. In order to extend their findings from these interviews a survey was created and posted on popular social media channels. This survey was filled in by 217 respondents. Although the paper does not immediately provide concrete questions developers ask, it does provide insights in how and why the history of a software project is used by developers.

Table 2.1 contains all the questions from [14, 36, 51] that could be answered by inspecting the history of the software project. For example, QV5 "Who has made changes to my classes?" can be answered by first identifying all classes a specific author introduced, followed by finding the authors of revisions that modify these classes that are different from the original author. QX3 "Who owns this piece of code? Who modified it most?" can be answered in different ways. The owner of a piece of code could be defined as the person who first introduced it. Depending on the granularity of the used information, this can be the person who introduced the file (requiring VCS information), the class (requiring CGSC) or that specific AST node (requiring DC or LoC). The person who modified it the most could also be the person who modified that file the most, or the person that modified that specific AST node the most often. Finally, some information is subsumed by more detailed information. For example, a coarse-grained representation of the source code is subsumed by a fine-grained representation. The same holds for line changes, distilled changes and logged. Table 2.1 depicts the different sources of information that could answer the question. More precise or fine-grained information should result in more precise answers.

Manually browsing through the history of a software project, which can contain thousands of revisions, is not a task any developer wants to undertake. Tool support is required to deal with the size of the data, the diversity of questions and the information required to answer these questions.

2.3.2 Empirical Studies

The history of software projects is the subject of several empirical studies. For example, the mining software repositories (MSR) community focuses "on the ways in which mining these repositories can help to understand software development, to support predictions about software development, and to plan various aspects of software projects" [23]. To this end, the community analyzes the information present in software repositories, as well as other sources of information such as bug trackers, mailing lists, etc. We focus on studies that primarily use Version Control Systems (VCS) as their source of information.

Table 2.2 depicts a summary of several empirical studies and the information that each study required. To this end, we have looked at the relevant tracks from several recent major conferences, identifying papers that mainly use VCS as their source of information for their study. We indicate whether the study uses VCS meta-data characteristics (VMC), coarse-grained revision characteristics (CGRC), fine-grained revision characteristics (FGRC) and change characteristics at the level of line changes (LiC), distilled changes (DC) or logged changes (LoC). All studies require temporal characteristics, and are not indicated. We found no study that uses evolution characteristics, even though studies would benefit from these. Instead, current studies manually identify change sequences that implement a code transformation (e. g., Negara et al. [61]) or they make use of an already known set of systematic edits and their changes (e. g., Meng et al. [56]).

Table 2.2: Table summarizing several empirical studies and the information the study required. The columns indicate the paper, a summary of that paper and whether the study required VCS meta-data (VMD), Coarse-Grained Revision Characteristics (CGRC), Fine-Grained Revision Characteristics (FGRC) or change characteristics at the level of line changes (LiC), distilled changes (DC) or logged changes (LoC).

Paper	**Summary**	**VMD**	**CGRC**	**FGRC**	**LiC**	**DC**	**LoC**
Ray et al. [64]	Identifies which changes are unique in the history of the project. The study explores how prevalent unique changes are and determines where they occur in the architecture of the project.	✓	✗	✗	✓	✗	✗
Mondal et al. [57]	Identifies code clones and studies how bug-prone the different types of code clones are. To this end, the study identifies bug-fixing changes made to these clones.	✓	✗	✓	✓	✗	✗
Meng et al. [56]	Presents a tool Lase that creates an executable edit script from systematic code changes. To this end, the tool generalizes several edit examples, represented by fine-grained code changes.	✓	✗	✓	✗	✓	✗
Zaidman et al. [78]	Studies the co-evolution between source code and their corresponding unit tests. For example, one of the research questions answered is whether this evolution happens synchronously or phased.	✓	✓	✓	✗	✗	✗
Cacho et al. [9]	Studies the evolution of exception handlers and their robustness. The latter is done by computing the places in which an exception can be thrown, and which exceptions are potentially uncaught.	✓	✗	✓	✓	✗	✗
Kawrykow and Robillard [49]	Identifies changes that are essential and non-essential. Examples of non-essential changes are local variable refactorings, or changes caused by a method rename. The study concludes that between 2.6% and 15.5% of all method updates consists entirely of non-essential modifications.	✓	✗	✓	✗	✓	✗
Giger et al. [38]	Studies whether fine-grained source code changes are more effective than coarse-grained changes in order to predict bugs.	✓	✗	✗	✓	✓	✗

Paper	Summary	VMD	CGRC	FGRC	LiC	DC	LoC
Dyer et al. [26]	Studies how and when new language features are adapted by developers, and whether there is code in which a new feature potentially could have been used, but was not. In order to determine whether existing code was adapted they compare the number of potential feature usages with the concrete number of usages between two successive revisions.	✓	✗	✓	✗	✗	✗
Negara et al. [61]	Finds unknown change patterns from fine-grained source code changes, represented by logged developer actions. A data-mining algorithm is applied to transactions of changes in order to detect patterns.	✗	✗	✗	✗	✗	✓
Steidl and Deissenboeck [70]	Studies how JAVA methods grow over time. To this end, the size of a method across every revision of a software project is compared.	✓	✗	✓	✗	✗	✗
Christophe et al. [13]	Studies the co-evolution between SELENIUM tests and the production code. It also studies which parts of SELENIUM tests are most prone to change.	✓	✗	✓	✗	✓	✗

There are several examples of such studies [38, 64, 78]. Study [78] investigates whether production code and its accompanying unit tests co-evolve throughout the history of the software project. To this end, its authors visualize how the commit behavior of developers, the growth of the project and the test quality evolve across time. In order to further substantiate their findings the authors manually analyze commit messages and interview the developers of the studied software projects.

Ray et al. [64] define, identify and study unique source code changes. Some changes are repetitive, idiomatic or frequent, while others are unique in the history of a project. In order to retrieve changes they use the meta-data of GIT, which stores source code modifications at the level of a line. Changes are collected for all the revisions of the analyzed software project. The authors bundle successive modified lines of code in a hunk of code. Next, they compare modified lines across hunks in order to find which changes are unique and which are not. After categorizing these changes, the authors analyze the results, for example to find how frequent unique changes occur, who introduces them and where they occur.

The work of Giger et al. [38] compares the use of fine-grained source code changes to the use of modified lines of code in bug prediction algorithms. To this end, they use the meta-data present in a VCS to retrieve the modified lines of successive revisions. In order to retrieve fine-grained source code changes they use ChangeDistiller [33]. ChangeDistiller is an algorithm that computes fine-grained between two revisions of a file. They find that source code changes outperform line changes for bug prediction models.

Looking at the research methods of the studies in Table 2.2, we notice that researchers combine existing analysis tools for a single revision with scripts to apply these tools on multiple revisions. Although the data used and produced by these studies is frequently available, the scripts are not. The latter is required to reproduce existing studies.

A general-purpose history querying tool could alleviate these problems. Such a tool enables expressing in a domain-specific language the wide variety of characteristics of the sought-after history elements. The tool automatically finds the history elements exhibiting these characteristics. The resulting queries can be reused on different projects and across different studies.

2.4 Criteria for General-Purpose History Querying Support

Based on the different types of history querying applications of Section 2.3 and Section 2.2.2, we identified six criteria that a general-purpose history querying tool must adhere to.

C1–C4: Support revision, temporal, change and evolution characteristics in a uniform language. For each of these characteristics, a user must express conditions over that characteristic in a history query. The history querying tool must identify the history elements that exhibit these characteristics. This is especially challenging for the evolution characteristics; the same code evolution can be implemented using different change sequences, which must all be identified by the history querying tool.

To this end, it must support characteristics concerning VCS meta-data, coarse-grained revision elements and fine-grained revision elements. Revision characteristics have been discussed in Section 2.2.2.

C5: Support a means for query abstraction, reuse and composition Query abstraction, reuse and composition enable users to write small queries that can be tested individually. These queries can be composed to form larger ones. This results in improved readability and maintainability of large queries. Without query reuse and composition, queries become convoluted, hard to understand and hard to debug.

C6: Provide solutions in an on-demand manner Looking at the different history applications we notice that they do not always need to have a complete set of answers. In the case of program comprehension retrieving some but not all solutions to a query can provide sufficient insight. During the prototyping phase of an empirical study, queries need to be tested. To this end, retrieving *some* solutions and verifying their correctness is desired over retrieving all the solutions, which may be time consuming. As such, a history querying tool should provide solutions in an on-demand manner.

2.5 Conclusion

In this chapter we have provided an overview of two history querying applications: program comprehension and empirical studies of software repositories. For each application, we have summarized existing research and the kind and granularity of history information required for that application. From these applications we distilled the different history characteristics that need to be supported by a history querying tool for it to be general-purpose.

These characteristics are classified as follows. First, *revision characteristics* which are further divided into version control meta-data, coarse-grained and fine-grained source code characteristics. Revision characteristics concern elements of a single revision. Second, *temporal characteristics*, which concern the temporal quantifications over revision elements. Third, *change characteristics* concern individual change operations and the affected source code. Finally, *evolution characteristics* concern change sequences and their effect on the source code.

We have discerned the different criteria to which a history querying tool needs to adhere in order to cater to the different needs of the history applications.

3
STATE OF THE ART IN HISTORY AND SOFTWARE QUERYING

In this chapter we provide an overview of the state of the art in querying software, its history and its evolution. To this end, we have divided this overview into three different parts. First, Section 3.1 discusses tool-supported approaches to querying a single revision of a software project. Second, Section 3.2 discusses tool-supported approaches to querying multiple revisions of a software project. Third and finally, Section 3.3 discusses tool-supported approaches to querying source code changes. We link all these approaches back to the criteria stipulated in Section 2.4.

Table 3.1 lists the discussed approaches and whether they adhere to the different criteria stipulated in Section 2.4. We divide criterion C1 into two subcriteria, expressing coarse-grained and fine-grained revision characteristics, to better illustrate the capabilities of the approaches. A plus denotes that the tool adheres to the criteria, a minus that it does not adhere, and a plus-minus that it partially fulfills the requirement. This is often the case when the tool provides the data (e. g., source code changes), but no actual query language (e. g., a change query language) for that data.

3.1 Querying a Single Revision

In this section we discuss existing approaches for querying a single revision of a software project. Section 3.1.1 briefly discusses approaches that query text. Section 3.1.2 discusses logic program querying.

Table 3.1: Table depicting an overview of existing history querying approaches, and whether they adhere to our criteria stipulated in Section 2.4

Tool	$C1_{coarse}$	$C1_{fine}$	C2	C3	C4	C5	C6
PQL	$+$	$+$	$-$	$-$	$-$	$-$	$-$
SOUL	$+$	$+$	$-$	$-$	$-$	$+$	$-$
CodeQuest	$+$	$-$	$-$	$-$	$-$	$+$	$-$
JTL	$+$	$+$	$-$	$-$	$-$	$-$	$-$
SCQL	$+$	$-$	$\pm$	$-$	$-$	$-$	$+$
V-Praxis	$+$	$-$	$\pm$	$-$	$-$	$-$	$+$
Harmony	$+$	$+$	$\pm$	$-$	$-$	$-$	$-$
Absinthe	$+$	$-$	$\pm$	$-$	$-$	$+$	$+$
Boa	$+$	$+$	$\pm$	$-$	$-$	$-$	$\pm$
Evolizer	$\pm$	$\pm$	$\pm$	$\pm$	$-$	$-$	$-$
CheOPSJ	$+$	$-$	$\pm$	$+$	$\pm$	$+$	$-$

3.1.1 Text Querying

Regular expressions [35] are suited to query large files of text. They enable users to express a sequence of characters that have to be present in a string, using operators such as repetition, optional characters, wildcards etc. For example, both the commandline `grep` tool and modern IDEs support searching through text files using regular expressions. Regular expressions are useful for developers searching for elements featuring a particular character string. A regular expression cannot account for variations of expressions (e. g., whitespace, formatting, or equivalent `for`/`while` loops, etc.). Thus, complex code patterns and all their possible variations become very hard to specify correctly and concisely.

3.1.2 Logic Program Querying

Logic program querying has frequently been proposed for querying a single revision of a software program. It advocates using formulas in an executable logic to specify the characteristics of the sought-after source code. This is done by reifying the program under investigation such that the logic variables can range over the elements of the program. Executing a proof procedure establishes whether certain program elements exhibit the characteristics specified by the formula.

Logic programming allows querying source code in a declarative style. Logic formulas enable users to describe the characteristics source code has to exhibit, while the proof procedure for the logic identifies the corresponding source code adhering to these characteristics.

There has been extensive research around the use of logic programming for querying a single version of a software project. In what follows we provide an overview of some of the work done in this domain.

PQL Program Query Language (PQL) [55] is a program query language mainly used in the context of finding application errors and security flaws. A PQL query is a pattern that is matched against the execution trace of the program. Potential matches for the pattern are detected statically by means of a static analyzer. These matches are further refined using dynamic analysis. The static analysis narrows the locations of the application that have to be instrumented and verified by the dynamic analysis. To this end, the program under investigation is transformed into a logic fact base, and a PQL query is translated to a Datalog [10] query consulting this fact base. The dynamic analysis will then catch all matches during the execution of the program. A user can define actions that are executed when such a match occurs. Examples of such actions are the insertion of a breakpoint, or writing the occurrence of a match to a log.

Figure 3.1 depicts a PQL query that verifies that `InputStream` resources are managed properly. It detects methods that create an `InputStream in` which is not closed before the end of that method. Should it escape that method an action is executed that closes the `InputStream`. This example is adapted from Martin et al. [55].

```
query forceClose()
uses object InputStream in;
within _ . _ ();
matches {
  in = new InputStream();
  ~in.close();
}
executes in.close();
```

Figure 3.1: PQL example that enforces every opened stream to be eventually closed. This example is adapted from Martin et al. [55].

CodeQuest CodeQuest [43] enables a developer to write logic queries over Java programs using Datalog. To this end, it creates a fact base representating the software system under investigation. This fact base reifies relations between selected source code entities, such as the reading from and writing to fields and calling of methods. It does not provide submethod information or ASTs.

Figure 3.2 depicts a CodeQuest query, adapted from Hajiyev et al. [43]. This query finds all methods that implement an abstract method. To this end, it unifies logic variable `Abstract` with an abstract method, `Implements` with a method overriding this abstract method, and ensures that this method is not abstract itself.

JTL The Java Tools Language (JTL) [15] is a program query language to reason about Java code. JTL represents the program under investigation in a Datalog fact base. The query language is a Datalog language with a syntax resembling Java. As such, JTL features predicates that enable users to write queries that superficially resemble the corresponding source code. These queries are then evaluated using

```
implementation_of_abstract(Abstract, Implements) :-
  modifier(Abstract, abstract),
  overrides(Implements, Abstract),
  not(modifier(Implements, abstract)).
```

Figure 3.2: Finding a method that implements an abstract method using CodeQuest, adapted from Hajiyev et al. [43].

```
boolean_return_recommendation :=
if[_] then[S1] else[S2],
S1.return[V1],
S2.return[V2],
V1.literal["true"],
V2.literal["false"];
```

Figure 3.3: A JTL example detecting unneeded if-tests that either return `true` or `false`. This example is taken from Cohen et al. [15].

Datalog. JTL uses the bytecode representation of the Java program instead of the actual source code.

Figure 3.3 depicts a JTL query, taken from Cohen et al. [15]. It detects unneeded if-tests that look like `if(test) return true; else return false;`. The second line introduces two new variables `S1` and `S2` that substitute respectively for the consequent and alternative of an if-test. Both `S1` and `S2` must be a return-statement, returning `V1` and `V2`. `V1` must be the literal value `true`, while `V2` must be the literal value `false`. This query illustrates the combination of a grammar resembling Java as well as the use of declarative programming.

SOUL The Smalltalk Open Unification Language [20], known as SOUL, is a declarative program query language implemented in Smalltalk. SOUL queries support three kind of conditions: regular logic conditions, Smalltalk conditions and template conditions. Smalltalk conditions are any Smalltalk expression that evaluates to either `true` or `false`. Template conditions are conditions in a logic query that resemble the sought-after source code. Its main motivation is that code queries expressed using only logic predicates can become convoluted. For example, `if jtStatement(st) ?x = (?type)?e;` is an example of such a template query. The template contains logic variables that unify with the corresponding source code element matching the template.

Figure 3.4 depicts an example person class written in Java and a SOUL query using a code template, adapted from the SOUL website[1]. This template matches all the accessors for the age variable. Imagine for a moment that the name of the field and the operand of the return statement are reified as strings. The unification would unify the shadowed variable used on line 4 with the instance variable. The method on line 3 would not match, as `age` does not unify with `this.age`. The domain-

[1] `http://soft.vub.ac.be/SOUL`

specific unification used in SOUL treats reified objects differently. This allows the unification of the two variables on lines 2–-3, while the variable on line 4 does not unify. SOUL uses static analyses similar to PQL to gather this information.

```
1  class Person {
2    private Integer age;
3    public Integer getAge() { return this.age; }
4    public Integer notGettingAge(Integer age) { return age; }
5  }
6
7  if jtClassDeclaration(?c){
8       class ?className {
9         private ?fieldDeclarationType ?fieldName;
10        ?modifierList ?returnType ?methodName(?parameterList) {
11          return ?fieldName;
12      }
13    }
14  }
```

Figure 3.4: A Person class with an accessor for the age in Java. The other method does not return the instance variable, as it is shadowed. This example is adapted from the SOUL website.

3.1.3 Conclusion

Logic program querying has proven a popular choice for querying the source code of a software project. It enables users to describe the sought-after source code in a high-level language, while the identification of the program elements that adhere to the logic specification is performed by the logic engine. Existing program querying tools cannot be readily used as a history querying tool, as they do not enable users to express temporal characteristics nor change characteristics. They do form a starting point for supporting revision characteristics in a history query language. To this end, they must be adapted significantly to enable querying VCS meta-data next to the source code of individual revisions. Change and evolution characteristics would not be supported as VCS do not provide (distilled or logged) change information.

3.2 Querying Multiple Revisions

In the following section we discuss existing history querying approaches that enable users to query multiple revisions of a software project.

SCQL SCQL [46] is one of the very first history querying approaches. It converts a VCS into a graph: nodes in this graph correspond to revisions, modification requests, authors and files. A modification request is a set of modified files by a specific author that have been submitted to the repository, but that are not accepted

```
1 E(a, Author) {
2   A(f, author.files) {
3     A(f2, author.files) {
4       eq(f.directory, f2.directory)
5     }
6   }
7 }
```

Figure 3.5: An SCQL query detecting authors that only modified files that are in a single directory. This example is taken from Hindle and German [46].

yet. Once accepted, a new revision is created. Edges correspond to the relations between these elements. A modification request is linked to the successive modification request, the author of its revisions and each of its revisions. SCQL provides a declarative language for expressing relationships such as `previous`, `after`, `always` or `never`.

We already provided an example SCQL query in Section 2.2.1. Figure 3.5 depicts another SCQL query that detects authors who only modified files residing in the same directory. This example is taken from Hindle and German [46]. The first line uses the existential quantifier `E` to state that an author `a` must exist. Lines 2–4 uses the universal quantifier `A` to state that *all* pair of files `f` and `f2` modified by author `a` must reside in the same directory.

Although SCQL represents one of the first approaches to history querying, it has several shortcomings. First and foremost, no actual source code (C1) nor source code changes (C3, C4) can be queried as the information is limited to the file level. This limits the kind of questions that can be answered with this approach. Basic temporal relationships can be expressed (e. g., `EBefore`, `ABefore`, `EAfter`, `AAfter`), but more complex relationships (e. g., `while`, within three revisions, . . .) can only be expressed by combining the provided predicates, which leads to heavily nested queries (C2). SCQL does not provide any means for query abstraction, reuse and composition, and thus these combinations cannot be abstracted away into new predicates (C5).

V-Praxis V-Praxis [58] converts the history of a software project to a relational database. This database contains a coarse-grained representation of every revision. V-Praxis does not provide a dedicated query language. Instead, it provides an interface for several query languages to query the database. Examples of such query languages are Prolog, XQuery and SQL.

Figure 3.6 depicts a logic predicate `changedOperations(Me,OpIds,R)` implemented in Prolog. This predicate succeeds for a list of methods `OpIds`, introduced by one author `Me`, that call methods that are modified by a different author in a recent revision `R`. The introduced methods must call a method that was recently (in this case within 10 commits of this revision) modified by a different author. Lines 3–5 find all methods that are introduced by `Me` and binds them to `MyOpIdList`, which is

```
1  changedOperations(Me,OpIds,R) :-
2  % Store the IDs of the method created by me in MyOpIdSet
3    findall(MyOpId,
4           (create(MyOpId,'method',_,Me)),
5            MyOpIdList),
6    list_to_set(MyOpIdList,MyOpIdSet),
7  % Store the IDs of the called method in CalledOpIdSet
8    findall(CalledOpId,
9            (parcours(CurrentMyOpId,MyOpIdSet),
10            addReference(CurrentMyOpId,'calls',CalledOpId,_,_)),
11            CalledOpIdList),
12   list_to_set(CalledOpIdList,CalledOpIdSet),
13 % Store result of the query in OpIds
14   findall(CurrentCalledOpId,
15           (parcours(CurrentCalledOpId,CalledOpIdSet),
16           action(_,CurrentCalledOpId,_,_,V,Other),
17           Other \= Me,
18           V >= R-10),
19           OpIds).
```

Figure 3.6: An example of a V-Praxis predicate, taken from its wiki. This predicate detects methods that are introduced by an author `Me` that call methods that were recently modified by a different author.

turned into a set on line 6. Lines 8–12 find all methods called by the methods in `MyOpIdList` and binds them to the set `CalledOpIdSet`. Finally, lines 14–19 find methods that are modified in a revision `V` that is at most 10 revisions removed from `R`.

V-Praxis does not feature a dedicated query language. Users are free to choose from Prolog, SQL and other general-purpose query languages for expressing revision and temporal characteristics. However, such general-purpose languages lack dedicated features that facilitate expressing these characteristics. For instance, there is no dedicated temporal specification language. Instead, the facts representing the program have a revision argument that needs to be accessed by the chosen query language (C2). Next, it is hard to state whether the query language provides means for query abstraction, reuse and composition (C5) and results in an on-demand fashion (C6). Prolog enables abstracting queries into predicates that can be reused, and computes results only if needed. SQL on the other hand can only abstract queries using stored procedures, which are cumbersome to implement. Finally, change information is missing from the fact base, and thus change and evolution characteristics cannot be queried for (C3, C4).

Harmony Harmony [29] is a history querying tool aimed at performing MSR studies. To this end, the tool provides data extractors for a wide array of industry-strength VCS. The history is represented by a model similar to the one used by V-Praxis. This model stores revisions, authors, files and modifications (create, edit or delete) made to files. Source code is available in an on-demand fashion for more fine-grained studies. To query this information, users need to subclass from an abstract analysis class. This class provides access to the stored data. Relevant

```
@Override
public void runOn(Source repo) {
  HashMap <Item , HashMap <Author , Integer >> HashMap <Author , Integer >>();
  for (Item it : repo.getItems()) {
    HashMap <Author , Integer > authors = new HashMap<Author, Integer>();
    ownership.put(it, authors);
    for (Action a : it.getActions()) {
      for (Author at : a.getEvent().getAuthors()) {
        Integer own = new Integer(1);
        if (authors.containsKey(at)){
          own = authors.get(at)+1;
        }
        authors.put(at, own);
      }
    }
  }
}
```

Figure 3.7: An example of a Harmony analysis, taken from Falleri et al. [29]. This analysis computes ownership of files.

data for the MSR study needs to be navigated to and inspected imperatively using Java expressions.

Figure 3.7 depicts an analysis implemented using Harmony. This example is taken from Falleri et al. [29]. The example computes the ownership of source files by counting how frequently each author modified each file. The method `runOn`, implementing an abstract method provided by the analysis class, imperatively iterates over the different files in a data source (i. e., an imported repository). Each file is associated with a list of actions (i. e., modifications to that file), which in itself is associated with a list of authors. By nesting loops the analysis updates a map that associates how frequently different authors updated that file.

Harmony imports a repository and makes its data more accessible for the user. Finding the relevant data must be done manually in an imperative manner, as depicted by Figure 3.7 (C1, C2). This makes queries convoluted, for example by having multiple nested loops. Composition and reuse of an analysis is difficult as it is constrained to the abstract analysis class (C5). Finally, change information is unavailable, and thus change and evolution characteristics cannot be queried for (C3, C4).

Absinthe Absinthe [75] is a history query language that is the conceptual predecessor of QwalKeko. It enables reasoning about the history of a software project, represented by a Hismo [39] model (cf., Section 2.2.2). To this end, a query describes a path throughout a revision graph using regular path expressions [17]. Sought-after revision characteristics are expressed using the declarative program query language SOUL, described earlier (cf., Section 3.1). Characteristics concern either elements residing in a Hismo model or the concrete source code of a method. The first is stored in memory, while the latter requires retrieving the source code from the VCS. This source code is queried using the regular predicates

```
1  ?first isOldestVersion,
2  e(manyOf>([true]),
3  and(?class isClass,
4  ?class wasChanged,
5  ?author isAuthorOfVersion),
6  manyOf<(not(?class wasChanged))) matches:
7  graph(versionTrans, ?first, ?last, <?class, ?author>),
8  ?last isEndVersion
```

Figure 3.8: Absinthe query that detects the author who last modified a class.

provided by SOUL. The model of Absinthe ensures that an entity (e. g., a class) stemming from in one revision unifies with the corresponding entity in another revision, even when that entity underwent some modifications. This allows users to use the same variable bound to an entity across different revisions as long as that entity is present in those revisions. The temporal characteristics are expressed as a path throughout a graph representation of the VCS. The used fact base changes throughout a history query depending on the current state of the path. As a result, predicates concerning revision elements do not require an argument specifying the revision of the sought-after element. Instead, this is handled by the temporal specification of the query.

Figure 3.8 depicts an Absinthe query. This query finds the author that last modified a class. To this end, the query describes a path through the revision graph. Along this path, there must be a revision in which the author modified the class, followed by revisions in which that class is no longer modified up to the terminal revision is encountered. The first line unifies `?first` with the starting revision of the VCS. Lines 2–7 describe a path throughout the revision graph, starting in `?first` and ending in `?last`. Line 2 uses the existential quantifier `e` to state that the query must hold for at least one path. The universal quantifier `a` would imply that the query must hold for all the paths between `?first` and `?last`. Line 2 also skips an arbitrary number of revisions along the path by using the `manyOf>` temporal operator with a body that always succeeds. `manyOf>` is a regular path expression operator that succeeds when its goal succeeds an arbitrary, including zero, amount of times. It also implicitly skips one revision after each succession of its goals. `manyOf>` is greedy, meaning it tries to succeed as much as possible (or in this case skips as many versions as possible). The lazy variant `manyOf<` would skip zero versions first. Both operators have the same semantics, but produce results in a different order. Lines 3–5 state that in the current revision of the expression a class `?class` must be present that was modified by `?author`. Next, line 6 states that for the remainder of the path that class cannot be changed. Note that the same variable `?class` is reused across different revisions. Finally, the last line of the query states that `?last` must be a terminal version, ensuring that the path ends in a terminal revision in the graph.

Absinthe provides a clean separation of temporal and revision concerns. Predicates are evaluated against a dynamic fact base that changes depending on the temporal specification. Nonetheless, it has several shortcomings. First, it is focused on querying a coarse-grained representation of the history of a software project. It enables users to retrieve the source code of a method, but does this inefficiently by fetching the code from a remote repository (C1). Such a coarse-grained representation can only be built efficiently when the underlying VCS has knowledge of the grammar of the stored programming language. If not, the complete representation of the source code is required. Next, Absinthe comes with a limited, predefined set of graph navigation operators that are not easily extended (C2). Finally, source code changes are not available, and thus change (C3) and evolution (C4) characteristics cannot be expressed.

Boa Boa [25] is a language for querying large software histories. A query is matched against a (predefined) set of software histories. Queries are written in a MapReduce [21] style: a mapping function retrieves the correct data elements, while an aggregator (such as max, min, mean, sum, . . .) combines these data elements into a single result. MapReduce ensures that queries are evaluated efficiently against a large number of repositories.

We already provided a Boa example in Section 2.2.1. Figure 3.9 depicts a different example, taken from Boa's website[2]. It depicts a query that detects whether a fixing revision added a null-check to a file. A fixing revision is identified by a commit message containing the word "fix" To this end, a visitor is defined that visits every queried project. It visits the different modified files for every revision. Whenever a previous revision exists of the modified file (in case it is introduced there is no previous revision), it visits both revisions of the file. This time the visitor will visit nodes of type `Statement`. It will count the number of if-statements that check for a null-statement. This is done for both revisions of the file, and in case the number of null-checks is increased it must have been added in the fixing revision.

Boa is aimed at supported MSR studies across large repositories. To this end, it uses MapReduce as an efficient way to query the different revisions across different repositories in parallel. Temporal and revision characteristics are expressed using the Visitor pattern. The temporal language is constrained as users can only express that all revisions of a project must be visited (C2). A visitor pattern also does not allow composing nor easy reuse of existing queries (C5). Next, Boa is only available through an online platform, restricting users to query only a predefined set of software projects. Finally, source code changes are not available, and thus change (C3) and evolution characteristics (C4) cannot be expressed.

[2]http://boa.cs.iastate.edu/

```
p: Project = input;
AddedNullCheck: output sum of int;

isfixing := false;
count := 0;
# map of file names to the last revision of that file
files: map[string] of ChangedFile;

visit(p, visitor {
        before node: Revision -> isfixing = isfixingrevision(node.log);
        before node: ChangedFile -> {
                # if this is a fixing revision and
                # there was a previous version of the file
                if (isfixing && haskey(files, node.name)) {
                        # count how many null checks were previously in the file
                        count = 0;
                        visit(getast(files[node.name]));
                        last := count;

                        # count how many null checks are currently in the file
                        count = 0;
                        visit(getast(node));

                        # if there are more null checks, output
                        if (count > last)
                                AddedNullCheck << 1;
                }
                # if file is deleted
                if (node.change == ChangeKind.DELETED)
                        remove(files, node.name);
                else
                        files[node.name] = node;
                stop;
        }
        before node: Statement ->
                # increase the counter if there is an IF statement
                # where the boolean condition is of the form:
                #        null == expr OR expr == null OR null != expr OR expr != null
                if (node.kind == StatementKind.IF)
                        visit(node.expression, visitor {
                                before node: Expression ->
                                        if (node.kind == ExpressionKind.EQ || node.kind == ExpressionKind.NEQ)
                                                exists (i: int; isliteral(node.expressions[i], "null"))
                                                        count++;
                        });
});
```

Figure 3.9: Boa query that computes whether a fixing revision added a null-check to a file. This example is adapted from the Boa documentation.

3.2.1 Conclusion

Early approaches, such as SCQL, V-Praxis and Absinthe, only provide a coarse-grained representation of the history. Absinthe provides a complete representation of the source code in an on-demand fashion, at the cost of a performance penalty. Later approaches such as Harmony focus on providing history information, but lack a dedicated specification language. Boa is aimed at supporting MSR studies over multiple large repositories. Boa specifications adhere to the MapReduce paradigm, which is an efficient way to collect and process data of every revision in the system. A downside of visitor pattern is that queries are difficult to abstract from, reuse and compose.

3.3 Querying Source Code Changes

In the following section we discuss existing approaches that enable users to query source code changes.

CheOPS CheOPSJ [69] advocates change-centric software engineering using a first-class representation of changes. It is a Java-centric continuation of CheOPS [27]. CheOPSJ is created to work with the Eclipse IDE, whereas CheOPS does so for Smalltalk. CheOPSJ provides changes stemming from both a change logger and change distiller. CheOPS models revision entities, changes and their dependencies in a graph. Examples of these change dependencies are that the subject (i. e., the affected AST node) of a change must be present or that a method can only be removed after all its callers are also removed. CheOPSJ uses GROOVE [65] to describe patterns in these graphs.

Figure 3.10 depicts a GROOVE query over a CheoPSJ change graph. The query detects instances of a "move method" pattern. This pattern requires the removal of a single method in one class, and the addition of a method with the same name and signature in a different class. This query is taken from Soetens [69]. The query consists out of two class nodes (labeled `Class`) and two method nodes (labeled `Method`). Four addition nodes (labeled `Add`) and a single removal node (labeled `Rem`) are also present. The class nodes must be different, indicated by the red edge. Both methods must have the same name and signature, indicated by the shared nodes. One of the methods must have been removed, indicated by it being the subject of a remove change. Finally, both addition and removal must happen within a timespan of 10000ms. To this end, the query ensures that the time stamps of the operations are within this time span.

CheOPSJ provides a change-oriented representation of the history of a software project. Its program representation is based on a Famix [67], which does not provide submethod information. As a result, fine-grained revision data is not avail-

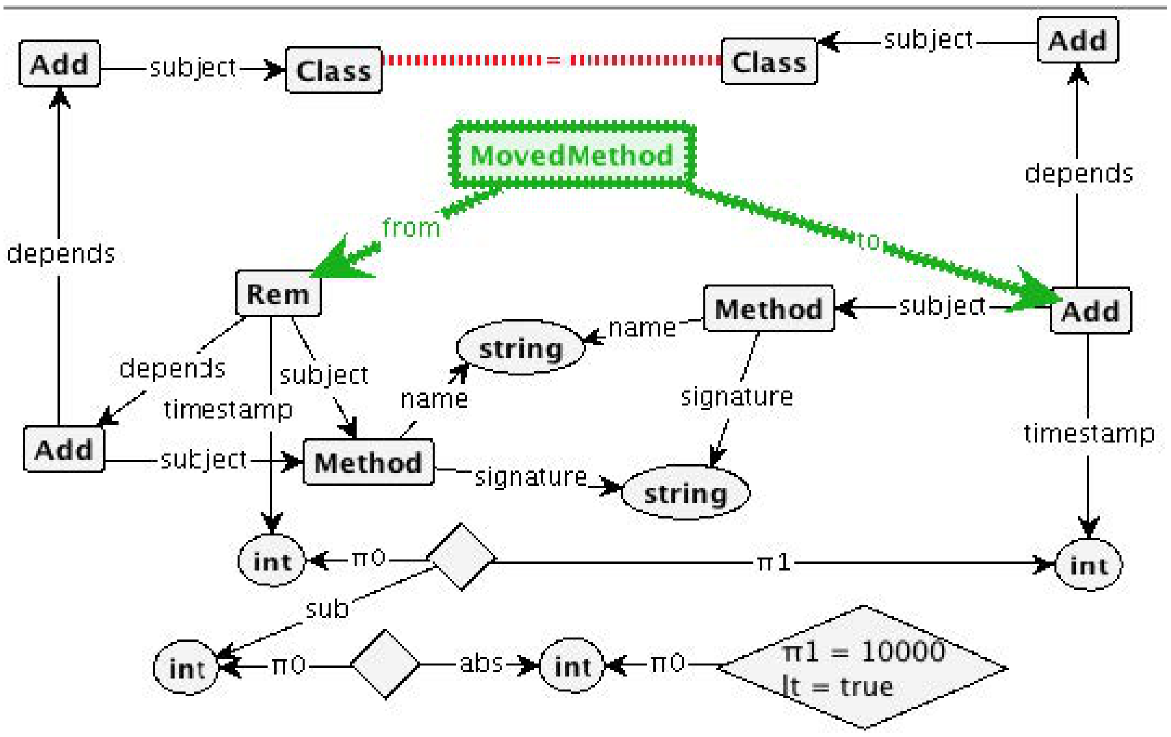


Figure 3.10: A GROOVE query that detects a "move method" pattern in the change graph of CheOPSJ. This example is taken from Soetens [69].

able (C1). The authors focus on changes stemming from a logging approach. These changes have a time stamp, but the revision elements themselves do not. There is no notion of a revision present, only changes that were applied on an initial starting state. This makes expressing temporal characteristics hard (C2). CheOPSJ enables expressing change characteristics (C3). The authors circumvent some of the problems of specifying evolution characteristics by relying on logged changes and by not providing move operations, which are represented by the removal and addition of the element. Nonetheless, the "Change Equivalence" problem (detailed in Chapter 7) still exists. For example, a method can be introduced by inserting a new method, but also by renaming an existing method. These different sequences must be manually accounted for. The query language GROOVE supports query abstraction, composition and reuse (C5).

Evolizer Evolizer [37] is a platform to perform studies on the evolution of source code. To this end, the authors enable importing data from the version control systems CVS and SVN and the issue tracking software Bugzilla. Evolizer stores the imported data in a database that can be accessed to perform MSR studies. Several tools are built on top of Evolizer. Among these is ChangeDistiller [33]. ChangeDistiller computes fine-grained source code changes between files of two successive revisions. To this end, it uses the facilities provided by Evolizer to retrieve the source code of modified files between two revisions, and stores the computed changes in a database as well.

Evolizer facilitates accessing the source code from a repository, but, to the best of our knowledge, does not provide a query language for this data. Source code can be queried using the facilities provided by Eclipse, for example by implementing a visitor (C1). ChangeDistiller provides fine-grained source code changes, but does not provide a dedicated query language for these changes. As such, Evolizer does not allow expressing temporal (C2) characteristics, nor change (C3) or composable source code characteristics (C5). It could provide a solid foundation for a history query language, as it exposes the historical data required to support each of the desired criteria.

3.3.1 Conclusion

Existing approaches that provide source code changes do not focus on querying these changes and the revision elements. They do not suffice as general-purpose history querying tools. Evolizer forms a starting point by providing the different data sources needed to create such a history querying tool. CheOPSJ represents changes and their dependencies as a graph. This enables users to query changes and the affected code using a graph query language. Unfortunately, CheOPSJ does not facilitate in expressing fine-grained revision and temporal characteristics.

3.4 Conclusion

In this chapter we have discussed the state of the art in querying software, its history and evolution. First, we have discussed several approaches that query a single revision of a software project. Logic program query languages represent their programs as a fact base and provide predicates to query this fact base. Their declarative nature facilitates expressing the characteristics of the sought-after source code in a terse query, while the identification of the elements adhering to these characteristics is done by the underlying reasoning engine. Next, we have discussed several existing history querying approaches. Early approaches provide but a coarse-grained representation of each revision of the system. Later approaches include the complete source code, but lack a general-purpose specification language or are focused on performing MSR studies. Finally, we have discussed existing tools and approaches that work with changes. Tools such as CheOPS organize changes to help developers browse a sequence of changes. These approaches are not suited to perform automated studies across different repositories. In general, there exists no tool that adheres to all the criteria listen in Section 2.4.

4 OVERVIEW OF THE APPROACH

In this chapter we introduce our approach to history querying that satisfies the criteria in Chapter 2. We have instantiated this approach in a domain-specific language called QwalKeko. QwalKeko enables its users to query the history and evolution of Java projects stored in git. To this end, it converts a git repository into a directed acyclic graph, of which the nodes corresponds to revisions, and the edges correspond to the successor relation between revisions. A QwalKeko query describes a path throughout this graph, and the characteristics revision elements need to exhibit in revisions along this path. QwalKeko is a *declarative* query language. Users specify the different characteristics of the sought-after elements, while identifying these elements is left to an underlying search mechanism.

QwalKeko integrates three smaller domain-specific languages; the graph query language Qwal, the program query language Ekeko, and the change query language ChangeNodes. Figure 4.1 depicts an overview of the architecture of QwalKeko. Qwal is a general-purpose graph query language that enables specifying paths through any graph and the properties that have to hold in nodes along this path. In the context of QwalKeko, Qwal is used to specify paths through the revision graph. Such a path describes the temporal relations of a history query, and supports the second criterion. We introduce Qwal in Section 4.2, and discuss it in detail in Chapter 5. Ekeko is a declarative programming language that enables querying Java projects in the Eclipse IDE. It is used to express the revision characteristics that elements need to exhibit in revisions along the path specified using Qwal. It supports the first criterion. ChangeNodes provides an implementation of the change differencing algorithm presented by Chawathe et al. [11]. ChangeNodes features predicates that reify the output of this distiller and enables users to express change characteristics. This supports the third criterion. We introduce these predicates in Section 4.4 and discuss them in detail in Chapter 6. To enable users to specify evolution characteristics, QwalKeko provides a graph representation of intermediate ASTs, where each node is an AST that can be created by

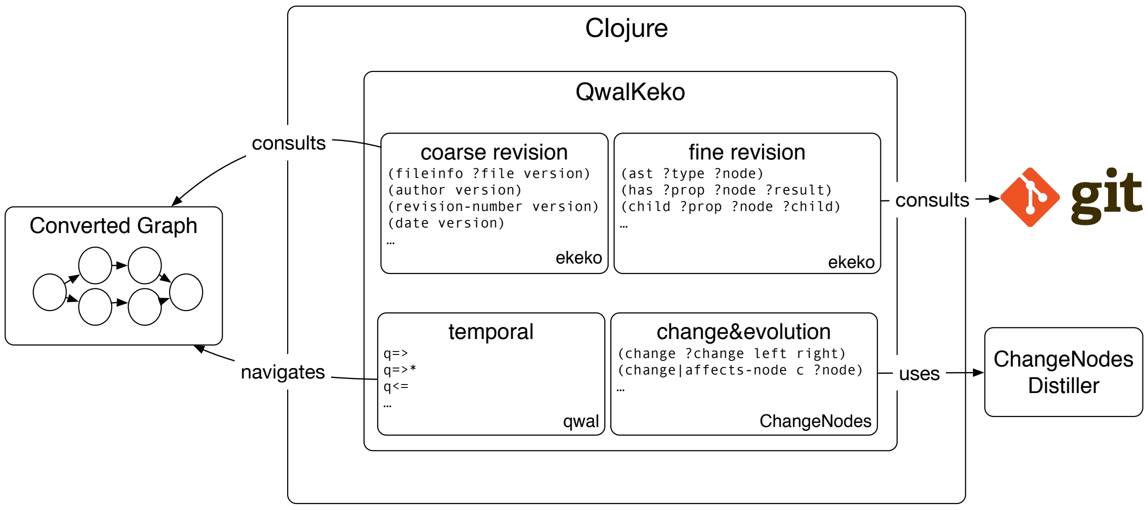


Figure 4.1: Overview of QwalKeko's integration of ChangeNodes, Qwal and Ekeko.

applying a subset of changes. Evolution characteristics concern paths throughout such an "Evolution Graph" and the source code characteristics of its intermediate states. This supports the fourth criterion. QwalKeko is a declarative language that supports query abstraction, reuse and composition, hereby supporting the fifth criterion. Its declarative nature supports the sixth criterion. As it relies so heavily on declarative programming, we start this chapter with an introduction to declarative programming in Clojure.

4.1 An Introduction to Declarative Programming in Clojure

Throughout this thesis we make use of the dynamic programming language Clojure. Clojure is a Lisp-dialect that compiles to Java Virtual Machine (JVM) bytecode. Clojure features a symbiosis with Java enabling calling Java code from Clojure and vice versa. This thesis makes extensive use of a declarative reasoning engine called `core.logic`[1]. This engine is based on miniKanren [8], which provides a declarative library for the Scheme programming language.

Logic queries are launched using the special forms `run` or `run*`. The first returns a given number of solutions, while the latter computes all solutions to the query. Throughout this thesis we will use the naming convention that logic variables start with a question mark. We explain the different aspects of `core.logic` by means of several examples that gradually increase in complexity.

[1]https://github.com/clojure/core.logic

```
1 (run* [?x]
2   (== ?x 1))
3
4 ;;[1]
```

The above example depicts how a logic query can be launched. The query returns all solutions to the logical variable `?x`. The first line launches a logic query using `run*`. It takes as its first argument a list of result variables, and computes all possible bindings for these variables that satisfy the query's logic conditions. These are given as the second argument to `run*`. The special form `run` has a similar interface, except that it takes the number of desired solutions as an additional first argument. Line 2 *unifies* `?x` with the number 1 using the predicate `==/2`.[2] `==/2` is the basic unification predicate of `core.logic`. Syntactically there is no difference between a predicate and a Clojure function. Semantically, a predicate does not return any value, but provides values for its arguments. In declarative programming these values are called *terms*. A term can be one of the three following things:

1. A primitive value, such as a number, a string or a Java object.

2. A logic variable.

3. A collection of which each element is another term.

We use the work of Flach to explain unification [32]. We first introduce the concept of *substitution*. A substitution is a mapping from variables to terms. For example, `{?x → 1}` and `{?x → ?y}` both are substitutions. *Unification* is the process of making terms equal by means of a substitution. This substitution is called a *unifier*. Several unifiers may exist to unify both terms, where one unifier replaces more variables by terms than strictly necessary. The most general unifier is the one that only substitutes the necessary variables. Such a unifier is unique, up to renaming of variables. Unification is the process of identifying such unifier. If no such unifier exist we say that the terms are not unifiable.

```
(run* [?x]
  (membero ?x [1 2 3]))

;;[1 2 3]
```

The above example illustrates the glue of declarative programming, namely backtracking. Backtracking tries to find additional solutions, either when the user requested additional solutions or when a later condition fails, by trying the different possible values for a variable. The example calls the `membero/2` predicate, a logic implementation of `member`. It unifies its first argument with the different elements of its second argument. Note that `?x` will only have a single value at a single time. Additional solutions are computed using backtracking.

[2]The number indicates the arity of the predicate

```
1 (run* [?x]
2   (membero ?x [1 2 3])
3   (membero ?x [2 3 4]))
4
5 ;;[2 3]
```

The above example further illustrates backtracking and conjunction. First, line 2 unifies `?x` with the value 1. Next, line 3 tries to unify `?x` with the value 2. This fails, so the reasoning engine backtracks to line 2, which had more possible solutions. This time, `x` is unified with the value 2. Line 3 now succeeds, and a solution is found. As `run*` computes all solutions, `?x` will be unified with the value 3 as well, which is also a solution.

```
(run* [?x]
  (conde
    [(== ?x 1)]
    [(== ?x 2)]))

;;[1 2]
```

The above example illustrates the logical disjunction `conde`. `conde` takes an arbitrary number of clauses, and each clause consists out of an arbitrary number of conditions. `conde` succeeds if at least one of its clauses succeeds. When multiple clauses succeed, backtracking ensures the different succeeding clauses are used to compute solutions. Thus, `?x` either takes the value 1 or the value 2.

```
(run* [?x ?y]
  (membero ?x [1 2 3])
  (membero ?y [3 4 5]))

;;[[1 3] [2 3] [3 3] [1 4] [2 4] [3 4] [3 3] [3 4] [3 5]]
```

The above example illustrates queries with multiple solution variables. Instead of outputting a collection of possible values for a single variable, `core.logic` returns a collection of vectors. In each vector the first element corresponds to a solution for the first variable, the second element with a solution for the second variable etc.

```
(run* [?x ?y]
  (membero ?x [1 2 3])
  (membero ?y [3 4 5])
  (!= ?x ?y))

;;[[1 3] [2 3] [1 4] [2 4] [3 4]...]
```

The above example illustrates the predicate `!=`/2, which states that its arguments cannot unify. When both arguments are a primitive value, this predicate succeeds by verifying that both arguments are not equal. When one of the arguments is an unbound variable it places a constraint on the variable that is verified when the variable is ground. `core.logic` features a complete library to perform constraint-based programming. We do not discuss this library as we only make use of the `!=` predicate in this dissertation.

```
1 (defn membero [?x ?list]
2   (fresh [?head ?tail]
3     (conso ?head ?tail ?list)
4     (conde
5      [(== ?head ?x)]
6      [(membero ?x ?tail)])))
```

The final example illustrates how new predicates can be implemented. They are defined like a regular CLOJURE function, except that their body consists of an `all` or `fresh` special form. Both implement a logic conjunction of conditions. `fresh`, in addition, introduces new, local, logic variables. In contrast to PROLOG, `core.logic` requires variables to be declared. The code depicts a possible implementation of `membero/2`. On line 2 it introduces two new logic variables `?head` and `?tail` using `fresh`. Line 3 uses predicate `conso` which is the declarative version of CLOJURE's regular `cons` function. It unifies its first argument with the head of the third argument and its second argument with the rest of the third argument. Lines 3–5 state that either `?x` must unify with `?head` or that `?x` must be a member of `?tail`.

4.1.1 Negation As Failure

The final concept we would like to introduce is the concept of negation as failure (NAF). Negation is used in queries to state that conditions are not allowed to succeed. To this end, the special form `fails` is used, which takes a single goal that must fail. For example, in the following query we state that a variable should not be a member of the list on line 3 but of the list on line 4:

```
1 (run* [?x]
2   (fails
3     (membero ?x [1 2 3]))
4   (membero ?x [2 3 4]))
5
6 ;;'()
```

Counter-intuitively, this query does not yield any results. The reason for this is that `fails` implements NAF. As such, the predicate succeeds only when its conditions do not yield a solution. If no solution exists, `fails` succeeds, otherwise it fails. For the given example, the condition on line has several solutions, namely `?x` either being 1, 2 or 3. Variable `?x` not being ground can be a member of the first list. As a result, the condition on lines 2–3 fails. Switching around both statements yields the expected result, as depicted in the following figure:

```
1 (run* [?x]
2   (membero ?x [2 3 4])
3   (fails
4     (membero ?x [1 2 3])))
5
6 ;;[[4]]
```

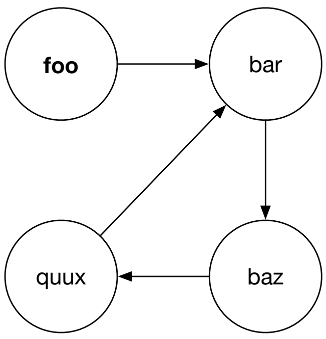


Figure 4.2: Example Graph containing four nodes `foo`, `bar`, `baz` and `quux`. The root node is bound to `foo`.

The following code depicts the implementation of `fails`. It uses `condu`, a logical disjunction that, once the first condition of a clause succeeds, does not backtrack. Thus, if `goals` on line 3 succeeds the second branch on line 4 will never be considered. If it does not succeed the second branch is considered, which always succeeds.

```
1 (defmacro fails [& goals]
2   `(condu
3      [(all ~@goals) fail]
4      [succeed]))
```

Throughout this dissertation we frequently use NAF to ensure a code element is absent from a revision. For example, a query that wants to find in what revision a class is introduced must find a revision in which that class is *not* present, and that the class is present in the next revision. In order to correctly implement such queries it is important to fully grasp the semantics of NAF with regards to variables that are ground or not.

4.2 Querying Graphs with Qwal

Qwal[3] is a declarative, domain-specific language for querying graphs hosted by Clojure's `core.logic` library. It enables navigating a graph using regular path expressions [17, 54]. Regular path expressions (RPE) are akin to regular expressions, except that they match nodes throughout a graph instead of characters in a string. We illustrate Qwal by means of several examples. In Chapter 5 we discuss Qwal in detail.

Figure 4.2 depicts an example graph of four nodes, `foo`, `bar`, `baz` and `quux`. The nodes of this graph are represented by Clojure symbols and do not contain any information. We have bound the Clojure variable `graph` to this graph, and the

[3]https://github.com/ReinoutStevens/damp.qwal

Clojure variable `start` to the node `foo`. We illustrate the basic concepts of Qwal by means of several simple queries.

```
1  (let [graph {:nodes ... :successors ...}
2        start 'foo]
3    (run* [?end]
4      (qwal graph start ?end []
5        (current-node [node]
6          (== node 'foo))
7        q=>
8        (current-node [node]
9          (== node 'bar))
10       q=>
11       (current-node [node]
12         (== node 'baz))
13       q=>
14       (current-node [node]
15         (== node 'quux)))))
16
17 ;;[quux]
```

The above example navigates a specific path through the graph; from `foo` over `bar` and `baz` to `quux`. To this end, line 3 uses `run*` to launch a query that returns all solutions for `?end`, which will be the end node of our RPE. Line 4 launches a RPE using the `qwal` special form. It takes as input a graph and starting node, a logic variable that will be bound with the end node of the RPE, and a list of local logical variables. In this example we start our path expression in `start`, which is bound to `foo`. Lines 5–6 make use of the special form `current-node` to bind `node` to the node of the graph up to which the RPE has navigated so far. Note that the scope of the Clojure variable `node` is limited to the body of the `current-node` special form. Line 6 unifies `node` with `'foo`. As our path expression started in the start node, and the expression has not yet navigated to a different one, the unification succeeds. Line 7 makes use of the navigation operator `q=>`, which moves the current node to one of its successors. The node `foo` only has a single successor, `bar`, which is now the current node of the expression. Line 8–9 ensure that this is the case. Lines 10–15 follow a similar pattern, changing the current node to respectively `baz` and finally `quux`. The latter will be the end node of our expression, and is unified with `?end`.

```
1 (run* [?end]
2   (qwal graph start ?end []
3     (q=>*)
4     (current-node [node]
5       (== node 'quux))))
6
7 ;;[quux]
```

The above example depicts a more complex navigation. It starts in a similar manner as before, launching a new logic query followed by a RPE starting in `start` and finishing in `?end`. Line 3 makes use of the `q=>*` operator, which skips an arbitrary, including zero, number of nodes. It is implemented using a logic disjunction, which either succeeds, or which applies `q=>` and a recursive call of itself. Backtracking ensures that all possible paths are explored. Lines 4–5 then state that the current node, and thus the end node, must be `quux`. Note that the graph contains an infinite path due to a cycle between `bar`, `baz` and `quux`. Nonetheless, Qwal detects this cycle and does not compute results indefinitely. The nodes in the cycle are visited only once. If the query would contain an additional `q=>*` nodes in the cycle would be visited twice.

```
 1 (run* [?end]
 2   (qwal graph start ?end []
 3     (q=>*)
 4     (current-node [node]
 5       (== node 'foo))
 6     (q=>*)
 7     (current-node [node]
 8       (== node 'absent))))
 9
10 ;;[]
```

The above example determines whether a path exists through graph `graph` in which the node `foo` is encountered, *eventually* followed by the node `absent`. Such a sequence of nodes does not exist in our example graph. This example further illustrates that Qwal can handle infinite paths, and does not take the same path indefinitely when it will not yield new results.

```
 1 (run* [?end]
 2   (qwal graph start ?end [?state]
 3     (q=>*)
 4     (current-node [node]
 5       (== node ?state))
 6     (q=>+)
 7     (current-node [node]
 8       (== node ?state))))
 9
10 ;;[bar baz quux]
```

The final example detects cycles in the graph by checking whether the same node is encountered twice along a single path. To this end, it introduces a logic variable `?state` in the scope of the path expression. Line 3 skips an arbitrary number of nodes using the `q=>*` operator. Lines 4–5 unify `?state` with the current node of the path expression. Line 6 skips an arbitrary, non-zero, number of nodes. The final two lines detect whether the node `?state` is encountered again. The query returns the three nodes that are part of the cycle, `bar` `baz` and `quux`. The `q=>*` operator on line 3 will first skip zero nodes, meaning `?state` is bound to `foo`. The `q=>*` operator on line 6 navigates to a successor, in this case `bar`. Lines 7–8 now fail, and upon backtracking line 6 navigates to `baz`. These, and further backtracking on line 6, will not yield any solutions. It is only when the reasoning engine backtracks on line 3 that solutions are found. Once again Qwal will not go into an infinite loop: the reasoning engine detects whether, during the execution of a single goal, the same state is encountered twice. In this case a cycle exists and it will not further explore this state.

We further discuss regular path expressions in the context of QwalKeko in Chapters 5 and 7. They are used to express temporal and evolution characteristics, as stipulated by Criteria 1 and 2.

4.3 Querying Code with Ekeko

Ekeko[4] [19] is another domain-specific language hosted by Clojure, used to query Java projects in the Eclipse IDE. To this end, it features a predicate library that enables users to query the AST representation of the source code, as well as control and data flow information of the program under investigation. Users can launch queries directly from Eclipse as Ekeko is integrated via an Eclipse plugin that provides a read-eval-print loop.

Ekeko predicates reason about the Abstract Syntax Tree (AST) representation provided by Eclipse. A basic understanding of this Eclipse representation is therefore required. All AST nodes inherit from an abstract class `ASTNode`. For each AST node type a corresponding class exists, such as `MethodDeclaration`, `TypeDeclaration`, `ReturnStatement` etc. Every node type has a predefined set of properties. For example, the body of a method declaration is accessed using the `BodyProperty`. Three kinds of properties exist; child properties have a single AST node as their value, child list properties have a collection of nodes as their value, and simple properties have a regular Java object as their value. Ekeko reifies these AST nodes and their properties. We illustrate Ekeko by means of several examples.

```
1 (run* [?method]
2   (ast :MethodDeclaration ?method))
```

The above query detects all method declarations present in the queried software projects. The query finds all solutions for `?method`. On line 2 it uses the Ekeko predicate `ast`, which unifies its first argument with the type of the sought-after AST node,[5] and unifies its second argument with an instance of that type from the AST of the program under investigation. In this case we already provided the value for the desired type, namely a `:MethodDeclaration`. As such, `?method` will be unified with a method declaration in the program. Types are specified using the Clojure keyword that corresponds to the name of the sought-after class.

```
1 (run* [?method ?body]
2   (ast :MethodDeclaration ?method)
3   (has :body ?method ?body))
```

The above query retrieves all method declarations and their corresponding body. To this end, line 3 uses the `has/3` predicate which accesses the value of a particular

[4] https://github.com/cderoove/damp.ekeko
[5] This argument is a Clojure keyword, which are prefixed by a colon.

property of a code. It unifies its first argument with the name of the property, and its second argument with the owner of that property, and its third argument with the value of that property. In case of a simple property this value is a wrapper for a Java obect that keeps a link to its owner. In case of a child property the value is another AST node. In case of a child list property, the value is a list of nodes.

```
(run* [?method ?body ?statement]
  (ast :MethodDeclaration ?method)
  (has :body ?method ?body)
  (child :statements ?body ?statement))
```

The above query retrieves all method declarations, their corresponding body and the statements inside that body. The query starts from the same conditions as the previous one, but continues with a condition that retrieves a statement from the method's body. We use the Ekeko predicate `child`, which is similar to `has` except that it is specialized for child list properties. It unifies its first argument with the keyword representation of a child list property, unifies the second argument with the owner of the list, and the third argument with a single element from the list located at that property.

```
1 (run* [?method]
2   (fresh [?return ?value]
3     (ast :MethodDeclaration ?method)
4     (child+ ?method ?return)
5     (ast :ReturnStatement ?return)
6     (has :expression ?return ?value)
7     (ast :NullLiteral ?value)))
```

The final example detects all methods that explicitly return a null value. We introduce two local variables `?return` and `?value` using `fresh`. Line 3 binds `?method` to a method declaration. Line 4 uses `child+`, a recursive version of `child`, and binds `?return` to a descendant of `?method`. Line 5 ensures that this descendant `?return` is a return statement. Lines 6 retrieves the expression of the return statement using `has` and binds it to `?value`. Finally, line 7 ensures that `?value` is a null literal.

Chapter 5 discusses Ekeko and how it can be used to express revision characteristics, stipulated by Criterion 2, in the context of QwalKeko in more detail.

4.4 Querying Changes with ChangeNodes

The final part of QwalKeko enables users to query the fine-grained evolution of the source code of the software project under investigation. To this end, it provides the following three parts. First, it provides an algorithm to distill changes between two revisions of a file. Second, it provides a declarative language that reifies these changes, enabling users to express the characteristics of an individual change. Third and finally, it provides an evolution query language that enables users to express how source code could have evolved, and that identifies the changes responsible for this evolution.

Retrieving AST Changes

Most VCS only store changes at the level of modified lines of text. They do not have a language-aware representation of the versioned source code, and as such cannot provide fine-grained source code changes. Fine-grained changes are operations that can be applied on an AST to transform it from one revision into the AST of the next revision.

QwalKeko features a change distiller implementation called ChangeNodes. ChangeNodes computes fine-grained source code changes (i. e., insert, move, delete or update of an AST node) between two revisions of a file. It is based on the distilling algorithm of Chawathe et al. [11], which is also used by ChangeDistiller [33], a widely used change distiller. The output of a distilling algorithm is a sequence of changes that must be applied *in order* and that transform the original AST into the target AST. A distilling algorithm tries to output a minimal edit script that reuses as many existing nodes as possible. The main difference between ChangeDistiller and ChangeNodes is that ChangeNodes represents its changes using the AST representation of Eclipse, while ChangeDistiller uses a language agnostic representation. The advantage of using a differencing algorithm is that it can compute source code changes for any two source code files. The disadvantage is that these algorithmically retrieved changes do not necessarily correspond to the changes a developer made.

Supporting Change Characteristics

As ChangeNodes uses the Eclipse AST representation, we can use Ekeko to query these changes. In addition to the distiller, ChangeNodes provides a predicate library, which extends Ekeko with predicates for reasoning about the distilled changes. It enables a user to express characteristics of an individual change, its subject and its effect. The specification of the AST nodes of a change is done using Ekeko. We illustrate this idea further by means of an example:

```
(let [rev-graph ...
      root (first (:roots rev-graph))]
  (qwalkeko* [?change ?end]
    (qwal rev-graph root ?end [?left ?right ?inserted]
      (in-source-code [curr]
        (ast :CompilationUnit ?left))
      q=>
      (in-source-code [curr]
        (compilationunit-compilationunit|corresponding ?left ?right)
        (change ?left ?right ?change)
        (change|insert ?change)
        (change|insert-node|node ?change ?inserted)
        (ast :MethodDeclaration ?inserted)))))
```

The above example query finds changes that insert a method declaration in the source code, between the root revision and a direct successor of the source code.

The third line launches a QwalKeko query that finds all such inserts in version `?end`. It uses the special form `qwalkeko*`, which is the QwalKeko equivalent of `run*`. It enables to modify the fact base used by Ekeko throughout the execution of a query. Line 4 launches a regular path expression over the revision graph `rev-graph`, representing the history of a software project, starting in the root revision `root`, and ending in `?end`. It introduces three local variables, `?left`, `?right` and `?inserted`. These will get bound respectively to the original source code of one file, the source code of that file in the next revision and the method declaration that was inserted in between. Lines 5–6 bind `?left` to a compilation unit (i. e., the root node of a file) from the root revision. The `in-source-code` special form extracts the source code of the current revision, and evaluates the conditions in its body against that revision. This enables users to express source code characteristics for that revision. Line 7 changes the current revision to a direct successor using the Qwal predicate `q=>`. Line 9 uses `compilationunit-compilationunit|corresponding` to find the same compilation unit in the current revision, and unifies it with `?right`. To this end, it takes as its first argument a compilation unit, and looks for a compilation unit in the same package that contains the same class declaration. Line 10 uses `change`, which distills the different changes between both compilation units, and unifies `?change` with a single one of these changes. Line 11 ensures that `?change` is an insert. Line 12 unifies `?inserted` with the effect of the insert, i. e., the node that is inserted. Line 13 uses the Ekeko predicate `ast` to ensure that a method declaration is added.

Chapter 6 discusses the ChangeNodes distiller and its change query language in detail. The language is used to express change characteristics, as stipulated by Criterion 3.

4.4.1 Supporting Evolution Characteristics

The query from the previous example identified changes that introduce a method declaration. Individual change characteristics suffice when the user knows beforehand what specific changes he is looking for. Evolution characteristics are required when the change sequence is unknown beforehand or when a high-level code transformation must be described. One of the problems with specifying changes is that multiple change sequences can implement the same source code transformation, even when the underlying source code files are similar. This is due to the heuristic nature of the distilling algorithm. As a result, users must take *all* possible change sequences into account when writing generic change queries. For example, the query from the previous example only takes insert operations into account, even though a method can be introduced by updating the name of an existing method, or by moving a name to the `name` property of a method declaration.

This is where source code evolution characteristics come into the picture. Instead of trying to account for the different possible change sequences that can implement the sought-after code evolution a user describes the source code before and after the evolution using Ekeko. The search mechanism of QwalKeko identifies the concrete changes implementing this code evolution and returns them to the user. To this end, a so-called *Evolution Graph* (EG) is constructed, which consists out of all the possible intermediate ASTs that can be constructed from a sequence of changes.

Evolution characteristics concern the evolution of source code along the path through this ESG. We use Qwal to describe paths throughout this graph. To this end, extend Qwal with change navigation predicates that control the number of applied changes. The following example illustrates such a query, that returns an evolution state in which a method is introduced. An evolution state contains an intermediate AST that is constructed from applying a subsequence of changes. The example assumes the evolution graph is already created.

```
1 (let [evolution-graph ...
2       source ...]
3   (qwalkeko* [?end]
4     (query-changes evolution-graph ?end [?inserted]
5       (change->*)
6       (in-current-es [ast es]
7         (ast-ast-method|introduced source ast ?inserted)))))
```

The second line launches a QwalKeko query that finds all evolution states `?end` in which a method was added. Line 4 launches a query over the evolution graph using the `query-changes` special form. It starts in the "source" evolution state, that is the state in which no changes have been applied. Line 5 uses `change->*` to apply an arbitrary number of changes. Applying a change moves the current evolution state to a successive evolution state. A new intermediate AST is constructed by applying the change on the intermediate AST of the previous evolution state. Line 6 uses special form `in-current-es` to express source code characteristics of the AST `ast` of the current evolution state `es`. Line 7 uses `ast-ast-method|introduced`, which takes as input 2 bound ASTs and unifies its third argument with a method declaration that is introduced in the second AST. We assume `source` is bound to the original source code. `ast` is a local variable introduced by `in-current-es`, and is bound to the AST of the current evolution state. Finally, `?inserted` is unified with the newly introduced method.

Chapter 7 discusses the support for evolution characteristics in detail, as stipulated by Criterion 4.

4.5 Applicability of the Approach

QwalKeko supports querying Java projects that are stored in git. The discussed concepts can be generalized to other programming languages and version control systems.

Modern general-purpose VCS (e. g., GIT, DARCS, SVN or MERCURIAL) have a notion (either directly supported or by convention) of branches and revisions. Such a VCS can be represented as a graph, and temporal characteristics can be expressed as paths through this graph.

A coarse-grained representation of the versioned source code is required to support coarse-grained revision characteristics. For object-oriented languages a HISMO [39] model can be used. An instance of such a model cannot be created for any revision of a software project stored in language-agnostic VCS without checking out the complete source code of that revision first. Language-aware VCS (e. g., MONTICELLO, a VCS used for PHARO) do enable creating such a model efficiently [52]. QWALKEKO does not feature such a representation as creating it for the combination of JAVA and GIT requires checking out the complete source code anyway. Thus, QWALKEKO focuses on querying a fine-grained revision representation.

A structured reification of the source code is required to support fine-grained characteristics. QWALKEKO combines the representation provided by the ECLIPSE JDT with the logic program query language EKEKO to support revision characteristics. The granularity of this information can differ. The representation may contain a reification of the source code (i. e., an abstract syntax tree), but it may also contain control and data-flow information. A parser is required to create such an abstract syntax tree that adheres to an abstract grammar. The level of information that is stored in such AST depends on the used parser.

An AST representation is required to support change characteristics. A change distiller takes as input two ASTs, and outputs a sequence of changes. A distiller does not rely on the semantics of the programming language, but works purely on a syntactical level.

An AST representation that has knowledge of the abstract grammar of the represented programming language is required to support evolution characteristics. This is required to support change subsequences that, when applied, result in syntactically legal source code. The absence of such an AST representation would result in support for evolution characteristics that may also return solutions that yield syntactically illegal source code.

4.6 Conclusion

In this chapter we have provided an overview of QWALKEKO and its components; the graph query language QWAL, the program query language EKEKO and the change query language CHANGENODES. QWAL is used to express temporal characteristics (C1). Users describe a path using regular path expressions through a graph of revisions. This graph represents the history of the software project under investigation. EKEKO is used to express revision characteristics of the revisions along such a path (C2). To this end, it provides its users with a predicate library that enable expressing characteristics over the AST of a revision, as well as control and data flow graphs of the software project. QwalKeko supports change characteristics (C3) using ChangeNodes. ChangeNodes provides a change distilling algorithm to compute fine-grained source code changes between two revisions of a file. QwalKeko supports evolution characteristics (C4) by transforming the output of ChangeNodes into an evolution graph. Nodes in this graph are represented by intermediate ASTs that are the effect of applying a subset of the distilled changes. Evolution characteristics describe a path throughout an evolution graph and the characteristics intermediate AST states need to exhibit. These intermediate AST states are described using Ekeko. The *declarative* nature of QwalKeko ensures that the different components are extensible (C5) and provide solutions in an on-demand fashion (C6).

In the following chapters we discuss these different components in detail. In Chapter 5 we discuss the combination of Qwal and Ekeko, and provide queries for several history related questions. Chapter 6 discusses the change distilling algorithm used by ChangeNodes and how change characteristics can be expressed. Finally, Chapter 7 discusses the problems of expressing evolution patterns using only change characteristics and how evolution characteristics are required to express these patterns.

5 SUPPORTING TEMPORAL AND REVISION CHARACTERISTICS

In the previous chapters we motivated the need for a general-purpose history querying tool and the criteria such a tool needs to adhere to. In this chapter we introduce a version of QwalKeko, our query language, that allows expressing temporal and revision characteristics (C1 and C2). QwalKeko combines the general-purpose graph query language Qwal with the declarative program query language Ekeko. A graph query language enables expressing temporal characteristics by navigating a revision graph, while a program query language enables expressing revision characteristics that have to hold in specific nodes of that graph.

Section 5.1 motivates the need for dedicated support for specifying temporal characteristics. Section 5.2 discusses Qwal; our approach to supporting temporal characteristics. Qwal is a general-purpose graph query language that enables specifying paths through a graph. A path is specified using regular path expressions, which are akin to regular expressions. They consist of logic conditions to which regular expression operators have been applied. Rather than matching a sequence of characters in a string, they match paths through a graph along which their conditions holds. In the context of history querying, Qwal enables specifying temporal characteristics as paths through a graph representation of a version control system.

Section 5.3 motivates the need for dedicated support for specifying revision characteristics. Section 5.4 discusses Ekeko; our approach to supporting revision characteristics. The declarative program query language Ekeko enables expressing revision characteristics of Java programs inside the Eclipse IDE.

Section 5.5 evaluates the combination of Qwal and Ekeko by answering history-related questions.

5.1 The Need for Dedicated Support for Specifying Temporal Characteristics

As discussed in Chapter 2 and stipulated by criterion **C2**, a history query language must enable expressing temporal characteristics. **Temporal characteristics** concern the temporal quantification over elements from different revisions. **Revision characteristics** concern the revision or VCS elements of these revisions. The temporal specification language must overcome several challenges.

The main challenge is that the temporal specification language must be compatible with the revision query language. Even though both concern a different aspect of a history query, the temporal specification determines in what revisions certain revision characteristics must hold. Thus, the temporal and revision specification language must be compatible with each other. For example, combining a declarative temporal specification with an imperative revision query language may result in unclear or hard to grasp semantics due to backtracking. The temporal specification language also influences whether some revision characteristics become easy or hard to express. For example, using a declarative revision query language, specifying whether a code element has been introduced in a revision is easily done by finding a revision in which the element is present, and ensuring that that element is absent in a predecessor. The other way around is harder due to negation-as failure, as there is no way to express that any element must be absent without knowing what the sought-after element is. A similar example is specifying whether an element was removed. Thus, depending on the chosen temporal specification language such revision characteristics become easy or hard to express.

A second challenge is the memory usage and performance. A version control system may contain thousands of revisions which cannot all be loaded in memory at the same time. Thus, proving that a certain property holds for all revisions may result in thousands of revisions being simultaneously open. An alternative approach is to automatically close revisions after they have been used. This can lead to the same revision being loaded several times throughout the execution of a history, a costly process. Additionally, it may also lead to the loss of object identity. Elements that are retrieved from a revision may still exist in memory, and reopening that revision results in building a new representation of that revision. As a result, multiple instances of the same code element can exist in parallel. This may result in undesired behavior. A way to combat these problems is to provide a coarse-grained and fine-grained representation of a revision, as discussed in Section 2.2.2. A coarse-grained representation can be fully loaded into memory, and does not require checking out any revisions.

5.1.1 Representing a Version Control System as a Revision Graph

To support temporal characteristics QwalKeko represents the revisions of a VCS and their successor relation as a directed acyclic graph (DAG). Every node of this graph corresponds to an individual revision, and successive revisions are connected via a directed edge. Note that a node can have multiple outgoing edges. This is the case for revisions that initiate a new branch in the software's history. Nodes can also have multiple incoming edges. This is the case for revisions that resulted from the merge of different branches.

Nodes also provide a representation of the meta-data of a revision, as discussed in Section 2.2.2. This representation consists of the commit message, author, time stamp, revision number and modified files (i. e., added, changed and removed files). Table 5.1 depicts the functional Clojure interface for this graph representation. Table 5.2 depicts the logic interface that is written on top of the functional one. Note that most revision predicates have an already bound revision as their revision argument instead of a logic variable.

Table 5.1: Functional interface for the revision graph.

Procedure	**Description**
Revision Procedures	
`(successors revision)`	returns the direct successors of revision
`(predecessors revision)`	returns the direct predecessors of revision
`(revision-number revision)`	returns the revision number of revision
`(author revision)`	returns the author of revision
`(commit-message revision)`	returns the commit message of revision
`(date revision)`	returns the timestamp of revision
`(files revision)`	returns the modified files of revision
`(root? revision)`	returns a boolean indicating whether revision is a root revision
`(terminal? revision)`	returns a boolean indicating whether revision is a terminal revision
Graph Procedures	
`(revisions graph)`	returns the revisions of a graph
`(roots graph)`	returns the root revisions of a single graph

5.2 Supporting Temporal Characteristics through Qwal

Regular path expressions (RPE) are akin to regular expressions [1], except that they consist of logic conditions to which regular expression operators have been applied. Rather than matching a sequence of characters in a string, they match paths through a graph along which their conditions holds. In the context of history querying, RPEs match sequences of nodes in the revision graph (cf. Section 5.1.1). Conditions are specified using revision characteristics. These conditions quantify over the entities in a single revision of the software project. RPEs have been used in different software engineering domains as well, such as compiler optimizations, where they match paths through control-flow graphs [17].

A regular path expression uses operators that navigate a revision graph, hereby changing the revision against which revision characteristics are verified. Some of

Table 5.2: Declarative interface for the revision graph.

Predicate	**Description**
Revision Predicates	
`(graph-revision graph ?rev)`	unifies `?rev` with a revision stemming from the revision graph `graph`.
`(revision|root rev)`	succeeds if `rev` is a root revision (i. e., a revision without any predecessors).
`(revision|terminal rev)`	succeeds if `rev` is a terminal revision (i. e., a revision without any successors).
`(revision|branching rev)`	succeeds if `rev` is a branching revision (i. e., a revision with multiple successors).
`(revision|merging rev)`	succeeds if `rev` is a merging revision (i. e., a revision with multiple predecessors)
`(revision|non-branching rev)`	succeeds if `rev` is a non-branching revision (i. e., a revision without multiple successors)
`(revision|non-merging rev)`	succeeds if `rev` is a non-merging revision (i. e., a revision without multiple predecessors)
`(revision-number rev ?num)`	unifies ?num with the revision number of `rev`. Note that `rev` is not a logic variable, but must already be bound to a revision.
`(revision-author rev ?author)`	unifies `?author` with the author of revision `rev`.
`(revision-date rev ?date)`	unifies `?date` with the time stamp of revision `rev`.
`(revision-message rev ?message)`	unifies `?message` with the commit message of revision `rev`.
`(revision-file|modified rev ?file)`	unifies `?file` with modified file of revision `rev`.
`(revision-file|unmodified rev file)`	succeeds if `file` is not modified in revision `rev`. This is the equivalent of `(fails (revision-file|modified rev file))`.
Graph Predicates	
`(graph-revision graph ?rev)`	unifies ?rev with a revision of the revision graph `graph`. Note that `graph` is not a logic variable, but must already be bound to a revision graph.

these operators also enable expressing revision characteristics that have to hold in the current revision of the path expression. We provide an overview of the available operators. In the used notation ϕ is a sequence consisting of either temporal operators or logic conditions that need to hold in the current state of the path expression.

`q=>` The `q=>` operator moves the state of the regular path expression from the current node in the revision graph to one of its successor nodes. Upon backtracking a different successor node is used if available.

`q<=` The `q<=` operator moves the state of the regular path expression from the current node in the revision graph to one of its predecessor nodes. Upon backtracking a different predecessor node is used if available.

`(q*` ϕ`)` The `q*` operator indicates that ϕ has to hold an arbitrary, including zero, number of times. The `q*` operator is lazy, meaning it immediately succeeds. Upon backtracking, ϕ must hold another time. If ϕ does not move the state of the regular path expression no more solutions will be returned. A greedy variant is not available as it is not supported by the underlying logic implementation of `core.logic`.

`(q=>*` ϕ`)` The `q=>*` operator indicates that ϕ followed by a `q=>` has to hold an arbitrary, including zero, number of times. This is equivalent to `(q*` ϕ `q=>)`.

`(q<=*` ϕ`)` The `q<=*` operator indicates that ϕ followed by a `q<=` has to hold an arbitrary, including zero, number of times. This is equivalent to `(q*` ϕ `q<=)`.

`(q+ ϕ)` The `q+` operator indicates that ϕ has to hold an arbitrary, non-zero, number of times. This is equivalent to `(all ϕ (q* ϕ))`.

`(q=>+ ϕ)` The `q=>+` operator indicates that ϕ followed by a `=>` has to hold an arbitrary, non-zero, number of times. This is equivalent to `(q+ ϕ q=>)`.

`(q<=+ ϕ)` The `q<=+` operator indicates that ϕ followed by a `q<=` has to hold an arbitrary, non-zero, number of times. This is equivalent to `(q+ ϕ q<=)`.

`(q? ϕ)` The `q?` operator indicates that ϕ has to hold zero or one time.

`(q=>? ϕ)` The `q=>?` operator indicates that ϕ followed by a `=>` has to hold zero or one time. This is equivalent to `(q? ϕ q=>)`.

`(q<=? ϕ)` The `q<=?` operator indicates that ϕ followed by a `q<=` has to hold zero or one time. This is equivalent to `(q? ϕ q<=)`.

`(in-git-info [rev] ψ)` The `in-git-info` special form indicates that ψ has to hold in the current node of the revision graph. `rev` is a newly introduced local variable that can be used by ψ. ψ is a sequence of Ekeko conditions that concern meta-revision data.

`(in-source-code [rev] ψ)` The `in-source-code` special form is similar to `in-git-info`, except that ψ is a sequence of Ekeko conditions that concern meta-revision, coarse-grained and fine-grained revision data. As a side-effect, the current node of the revision graph is imported as a separate Eclipse project.

5.2.1 Example Queries

We illustrate the use of Qwal through three example queries. The first, second and third query identify authors that committed two successive revisions within 10 minutes, co-changing files and potential merge conflicts respectively.

Successive Commits In the first example we are interested in finding authors that committed two successive revisions within the time span of 10 minutes. The following query retrieves these authors:

```
(qwalkeko* [?author ?corrected]
  (qwal graph root ?corrected [?date ?end-date]
    (q=>*)
    (in-git-info [original]
      (version-author original ?author)
      (version-date original ?date))
    q=>
    (in-git-info [corrected]
      (version-author corrected ?author)
      (version-date corrected ?end-date)
      (within-10-minutes ?date ?end-date))))
```

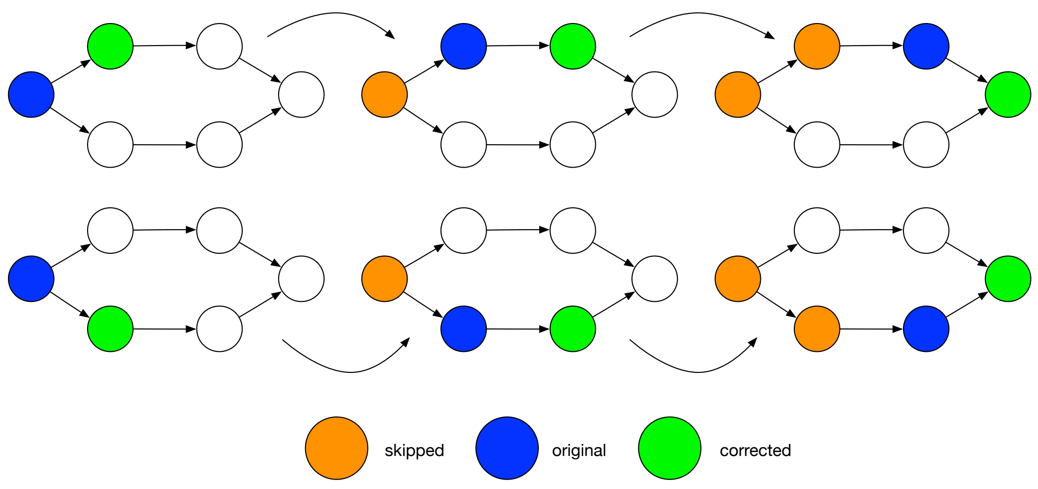


Figure 5.1: Overview of the different considered paths throughout the execution of the "Successive Commits" query.

The first line launches a history query that returns an author `?author` and a revision `?end` that was committed 10 minutes after the previous revision by the same author. The second line launches a Qwal expression over a revision graph `graph`, starting in `root` and ending in `?corrected`. We assume that the `graph` and `root` variables were defined earlier. The Qwal expression introduces two new local logic variables `?date` and `?end-date`. Line 3 uses the `q=>*` operator to skip an arbitrary number of versions. Its operands are empty, meaning only the `q=>` operator is applied an arbitrary number of times. Lines 4–6 specify some conditions that need to hold in the current revision of the path expression. They unify `?author` with the author of the revision, and `?date` with the timestamp of which it was committed. These conditions are wrapped in the special form `in-git-info`, which introduces a local variable `original` that is bound to the current revision, and takes an arbitrary number of logic conditions that have to hold in that revision. `in-git-info` enables expressing coarse-grained characteristics of information stored in the graph (see Table 5.1). Next, line 7 uses the `q=>` operator to move the current revision to a direct successor. The final 4 lines state that this revision needs to be modified by the same author within 10 minutes of the previous commit. This is also the end revision of the path expression, and is unified with `?corrected`. Note that Qwal will backtrack when the author of the revision on line 9 does not unify with the author of the revision on line 5.

Figure 5.1 depicts, for a simple example graph, a graphical overview of the different possible paths that are verified throughout the execution of this query. The revision graph consists of 6 revisions and two branches. The `original` and `corrected` nodes in the figure correspond to their namesake variables. `skipped` are the revisions that are skipped by line 3 of the query. The top row depicts the

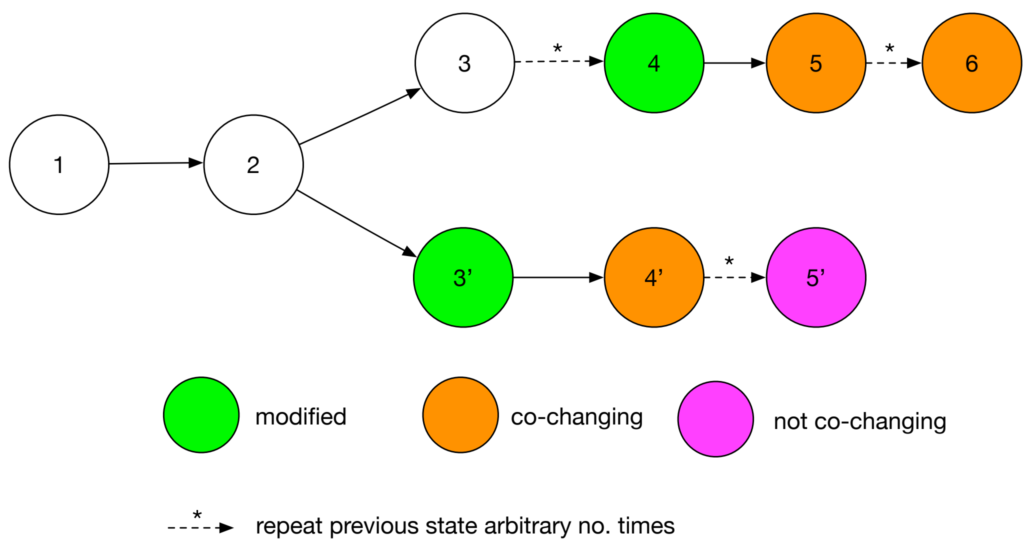


Figure 5.2: Two possible paths of the "co-changing files" query. The path on the top succeeds, while the path on the bottom contains no co-changing files until a terminal revision is encountered.

considered paths in the top branch, while the bottom row depicts the considered paths in the bottom branch.

Co-changing Files The following example illustrates the use of QwalKeko for finding co-changing files. These are files that, from a certain point in the revision graph, are always modified together. The following query retrieves such files:

```
(qwalkeko* [?fileA ?fileB ?end]
  (terminal graph ?end)
  (qwal graph root ?end []
    (q=>*)
    (in-git-info [modified]
      (revision-file|modified modified ?fileA)
      (revision-file|modified modified ?fileB)
      (!= ?fileA ?fileB))
    (q=>+
      (in-git-info [cochanging]
        (conde
          [(revision-file|modified cochanging ?fileA)
           (revision-file|modified cochanging ?fileB)]
          [(revision-file|unmodified cochanging ?fileA)
           (revision-file|unmodified cochanging ?fileB)])))))
```

Figure 5.2 depicts two paths that are considered during the execution of this query. The path on the top succeeds as there is a revision in which two files are modified, and these files co-change until a terminal revision is encountered.[1] The path on the bottom does not yield any results, as no files co-change until a terminal

[1]The history query accounts for multiple terminal revisions in a revision graph through backtracking over the different possible paths

revision is encountered. The first line launches a history query that returns all pairs of files `?fileA` and `?fileB` and end revision `?end` of the path expression. The second line states that `?end` must be a terminal revision (i. e., a revision without any successors). It ensures that the files co-change until a terminal revision is encountered. Line 3 launches the actual path expression over the revision graph `graph`, starting in `root` and ending in `?end`. Line 4 skips an arbitrary number of versions using the `q=>*` operator. Lines 5–8 state that two different files `?fileA` and `?fileB` in the current revision must have been modified. Lines 9–15 state that these files need to co-change an arbitrary, non-zero, number of times. To this end, they use the temporal operator `q=>+` in combination with the logic disjunction `conde` to state that either both files were modified, or both files remained unmodified. These files must remain co-changing until a terminal revision is encountered and the path expression terminates.

Potential Merge Conflict The final example illustrates the use of QwalKeko to detect potential merge conflicts by finding a file that was modified in two parallel branches that are merged in a later stage. The following query detects such files:

```
1  (qwalkeko* [?file ?end]
2    (qwal graph root ?end [?branch ?prev ?modified]
3      (q=>*)
4      (in-git-info [branching]
5        (revision|branching branching)
6        (== branching ?branch))
7      (q=>+
8        (in-git-info [curr]
9          (revision|non-merging curr)))
10     (in-git-info [modified]
11       (revision-file|modified curr ?file))
12     (q=>*
13       (in-git-info [curr]
14         (revision|non-merging curr)))
15     (in-git-info [curr]
16       (== ?prev curr))
17     q=>
18     (in-git-info [merging]
19       (revision|merging merging))
20     q<=
21     (in-git-info [curr]
22       (!= ?prev curr))
23     (q<=*
24       (in-git-info [modified]
25         (!= curr ?branch)
26         (revision-file|modified curr ?modified)
27         (file-file|same ?file ?modified)))))
```

Figure 5.3 depicts a successful path that is considered during the execution of the query. This query works by finding a branching version `?branch`, and a modified file `?file` in one of its branches. This is implemented by lines 3–11. Line 3 skips an arbitrary number of revisions. Lines 4–6 state that the current revision `?branch` must be a branching revision. Lines 7–11 skip an arbitrary number of non-merging

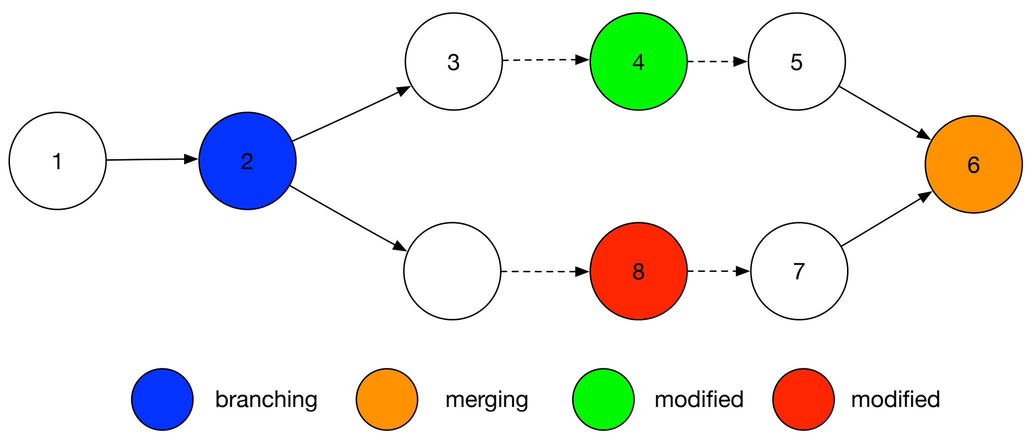


Figure 5.3: A successful path that is considered during the execution of the "Potential Merge Conflict" query.

revisions and unify `?file` with a modified file in such a revision. Next, the query continues skipping revisions until a merging revision is encountered. It will also capture the revision just before the merging one. This is needed so the query can move backwards and take a different path. This is a pattern that frequently occurs when querying branches. All of this is done on lines 12–19. Lines 12–14 skip an arbitrary number of non-merging revisions. Lines 15–16 unify `?prev` with the revision right before a merging revision. Line 17 moves to the next revision, which must be a merging one. This is stated by lines 18–19. Now, the query will move backwards along a different branch and verify whether the same file is modified. Line 20 moves one version back. To ensure that the path expression does not simply moves backwards on the same branch lines 21-22 state that the current revision must be different from `?prev`. Finally, lines 24–27 ensure that `?file` is modified in this branch as well. We cannot use unification as the file may be modified in a different manner (i. e., it may be removed in one branch and edited in the other one, so the modification type does not unify). Finally, line 25 ensures that the path expression does not go further than the initial branching version.

5.2.2 Supporting User-defined Temporal Operators

Qwal provides an extensible implementation that can be extended with user-defined operators. Users may want to implement their own operators to abstract over domain-specific patterns, or to query graph structures for which the regular Qwal operators do not suffice. For example, we define domain-specific operators in Chapter 7 to query so-called evolution graphs. In order to support existing Qwal operators a graph structure needs to provide a `successors` function that returns a list of successors for a given node.

Every operator in QWAL is implemented by a logic rule that takes three arguments: a graph object, a bound current node and an unbound next node. These rules need to bind the `?next` variable to another node of graph for their operator to succeed. We illustrate th extensible implementation by means of three examples.

Implementing `q=>` The following code snippet shows the implementation of `q=>`:

```
(defn q=> [graph current ?next]
  (all
    (membero ?next (successors graph current))))
```

The first line corresponds to the interface operators must adhere to. The `graph` and `current` variables will be bound to the queried graph and the current state of the path expression respectively, while the `?next` variable is an unbound logic variable that must be ground by the operator. `q=>` itself is implemented using `membero`, a logic implementation of `member`, over the successors of the current node of the path. `successors` calls the graph-specific successor function.

Implementing `q*` The following example shows the implementation of `q*`:

```
(defn q* [& goals]
  (def q*loop
    (tabled
     [graph current end goals]
     (conde
      [(fresh [next]
         (solve-goals graph current next goals)
         (q*loop graph next end goals))]
      [(== current end)])))
  (fn [graph current next]
    (q*loop graph current next goals)))
```

Operator `q*` is implemented in terms of a local tabled rule `q*loop` that either proves all the logic goals, hereby arriving in a new state `next`, and calling itself recursively; or that finishes the loop by unifying the `next` state with the `current` state. A tabled rule is proven using tabled resolution (see below), ensuring that proving the same goal multiple times does not result in an infinite loop (e. g., when a query moves back and forth between the same revision). Finally, the `q*`-rule returns a new function that just calls the internal function.

Implementing `branch=><=` In the final example we implement a new temporal operator `branch=><=` that we could have used in the "Potential Merge Conflict" example (cf., Section 5.2.1). This operator moves forwards one revision, and backwards into a different branch.

```
(defn branch=><= [graph current ?next]
  (fresh [?branch]
    (q=> graph current ?branch)
    (revision|branching ?branch)
    (q<= graph ?branch ?next)
    (!= ?next ?branch)))
```

The code reuses the Qwal operators `q=>` and `q<=` to navigate the graph, while ensuring a branching revision is encountered and a different branch is taken. Such an operator is not provided by Qwal as it is specific to history querying, but can easily be implemented by a user. This is stipulated by Criterion **C2**.

5.2.3 Qwal Compared to Graph Query Languages

In this section we briefly discuss existing graph query languages. We discuss languages stemming from two domains. The first domain is the formal verification of computer programs. The second domain are graph databases.

Formal Verification Linear Temporal Logic (LTL) [6, 62] and Computation Tree Logic (CTL) [6, 28] have both been used for the formal verification of computer programs. LTL is a temporal logic; a form of logic specifically tailored for formal statements that involve the notion of order in time. In the context of history querying this notion of time is provided by the successor relationship between the different revisions. An LTL query describes properties of an (infinite) path through a graph structure. CTL is a branching-time logic. It reasons about multiple paths in parallel, resulting in a tree-like structure instead of a single path. It features a similar set of operators to LTL, but these are accompanied with either an existential or universal quantifier.

We have chosen regular path expressions (RPE) as the foundation for Qwal. LTL, CTL and RPE share a subset of properties that can be expressed in each formalism, and have a disjoint set of properties that cannot be expressed by the other formalisms. The shared subset contains frequently occurring patterns [24]. The differences in expressivity between the three formalisms are apparent in cyclic graphs that contain infinite paths. In the context of history querying, the queried revision graph is always acyclic.

Graph Databases A graph database represents its data as a graph using nodes, edges, and their properties. Several languages exist to query such graph databases. We focus on three major query languages Cypher, SPARQL and Gremlin.

Cypher is a declarative language to create, update and query the graph database Neo4j.[2] Neo4j represents its stored data as either a node, an edge or an property. Nodes and edges can be labeled. For example, imagine we want to store people and their friendship relations. A person would be represented as a node, his name would be a property, and a friend would be represented via an edge to another person.

[2]`https://neo4j.com/`

A Cypher query describes a path through such a graph using a syntax that resembles the specified path. The following query returns all the "friends of a friend" of the person named "Alice":

```
1 MATCH (alice name: "Alice")-[:friend*2]->(foaf)
2 RETURN alice.name, foaf.name
```

Characteristics of a node are specified between parentheses (resembling a drawn node), while characteristics of an edge are specified in arrows. Line 1 specifies that there must be a node with name "Alice, bound to `alice`. It follows the edge labeled `:friend` twice. This is indicated by `*2`. The end node of this path is a friend-of-a-friend node, bound to `foaf`. Line 2 returns the results back to the user. Replacing `*2` by `*2..` would result in all indirect friends of Alice. Replacing `*2` with a `*` would result in all friends of Alice. These operators are similar to the operators of RPE.

SPARQL is a query language for Resource Description Framework (RDF). RDF is a directed, labeled graph format that is used in the context of the semantic web [45]. RDF represents data as triplets.

SPARQL supports property path expressions, which are similar to RPE. The following query returns all the "friends of a friend" of a person named "Alice:

```
1 {
2   ?x foaf:name "Alice",
3   ?x foaf:friend/foaf:friend/foaf:name ?name .
4 }
```

Line 2 retrieves a person named "Alice". Line 3 follows its `friend` property twice, and then unifies `?name` with the name of the friend-of-a-friend. The `/` construct enables chaining properties. The following query finds all the indirect friends of Reinout:

```
1 {
2   ?x foaf:name "Alice",
3   ?x foaf:friend/foaf:friend+/foaf:name ?name .
4 }
```

It adds a `+` construct to indicate that the `friend` property must be followed an arbitrary, non-zero, number of times.

Gremlin is a graph traversal language provided by Apache Tinkerpop. Gremlin enables users to implement graph traversals. According to the official website[3], every traversal is composed of a sequence of steps, where a step performs an atomic operation on a data stream. Each step of a traversal is either a `map`-step (transforming the data in the data stream), a `filter`-step (filtering data from the stream) or a `side effect`-step (used to compute statistics). These steps are then composed into a larger query.

The following query finds all the "friends of a friend" of a person named "Alice":

[3]https://tinkerpop.apache.org/gremlin.html

```
1 g.V().match(
2   as("alice").has("name","Aeinout"),
3   as("alice").out("friend").as("f"),
4   as("f").out("friend").as("foaf")).
5     select("foaf").by("name")
```

The first line launches a query over a graph `g`. It selects all the vertices that match the characteristics specified on lines 2–4. Line 2 creates a variable `"alice"` bound to a node that has a `"name"` property equal to `"alice"`. Line 3 follows the `"friend"` edge from the variable `"alice"`, and unifies it with `"f"`. Line 4 follows the `"friend"` edge from the node `"f"`, and stores the resulting node in variable `"foaf"`. Finally, line 5 selects the name of all the friends-of-a-friend. Selecting all the indirect friends of Alice can be done by through a `repeat` directive.

Cypher and SPARQL both enable specifying paths through a graph to retrieve elements of interest in terms of regular path expressions. Gremlin is more similar to functional programming. Cypher and SPARQL use similar operators as RPE to support expressing paths through a graph. This motivates the choice for RPE to support temporal characteristics in the context of history querying. We have optied to implement our own RPE over reusing Cypher or SPARQL as we aim to create a uniform language with a declarative foundation for history querying, as motivated by Chapter 3. This facilitates combining the support for temporal characteristics with the support for revision characteristics in the form of the declarative program querying language Ekeko.

5.2.4 Conclusion

Qwal implements regular path expressions, which enable users to specify paths throughout a graph, and the conditions that have to hold in nodes along that path. Path expressions are akin to regular expressions. In the context of history querying Qwal is used to navigate a graph representation of a VCS, in which nodes correspond to a revision and successive revisions are connected via an edge. The rest of the chapter focuses on expressing the conditions that have to hold in a revision.

5.3 The Need for Dedicated Support for Specifying Revision Characteristics

Most Version Control Systems provide support for querying their stored contents. This enables retrieving the stored meta-data, such as the author or commit-message, of a particular commit. They also support retrieving the data contained in each commit, such as the files or lines that were modified. The state of a file at a particular point in time (e. g., after a given commit) can also be reconstructed, after which it can be queried using separate tools for searching through text, such as `grep`[4].

For example, the `git log` command enables users to query the log of projects versioned in GIT[5]. This allows querying the author, commit message, date, changed files and changed lines of any revision. Unfortunately, filtering this data to find a revision with the interesting revision elements of that revision is left to the user. Similarly, the `git show` command can be used to reconstruct the contents of a particular file in a given revision. Unfortunately, most VCS have *no knowledge* of the syntax of the stored contents. As such, there is no difference between a plain text file and for example a Java source file. Thus, users need to resort to text querying tools to find source code exhibiting characteristics of interest.

A well-known example of such a tool is `grep`[6]. `grep` enables searching through text using regular expressions. This limits the kind of queries that can be expressed. For example, even a regular expression which detects whether a particular file implements a method with a specific signature is already cumbersome if one wants to account for the different white space that may be present. With more constraints added, such as requiring the method to perform a null-check, one can quickly see that regular expressions are not suited to account for all different ways such a method may be implemented. Next to the problems in accounting for the different legal syntactical ways to write source code is the problem of data- and control-flow information that is not available using a textual representation of the source code.

In general, fine-grained revision characteristics cannot be expressed using the query facilities provided by a VCS alone. Coarse-grained characteristics can be expressed in a limited manner, but may require combining several tools. As such, the query facilities of a VCS do not fulfill criterion **C1**: a history tool must enable expressing revision characteristics.

Program Query Languages (PQLs, cf., Section 3.1) are designed for querying the source code of a single revision of a software project. They relieve the user from implementing a search through code files themselves. Instead, source code characteristics are specified in a dedicated language, and the detection of code exhibiting these characteristics is done by the underlying implementation. We have provided several examples of such PQL in Section 3.1.

[4]http://www.gnu.org/software/grep/
[5]https://git-scm.com
[6]http://www.gnu.org/software/grep/

Table 5.3: Table depicting the commonly used Ekeko predicates

Predicate	Description
`(ast ?type ?node)`	Unifies `?node` with an AST node of type `?type`. `?type` is a keyword corresponding to the name of the class of the AST node (e. g., `:TypeDeclaration`, `:IfStatement`, `:MethodInvocation`, . . .).
`(has ?property ?node ?value)`	Unifies `?value` with a child node of `?node`, located at property `?property`. In case the property is a `ChildListProperty` `?child` is unified with the collection itself. `?property` is a keyword representing the identifier of the property (e. g., `:name`, `:body`, `:arguments`, . . .).
`(child ?property ?node ?child)`	Similar to `has`, except that property must be a `ChildListProperty`. `?child` is unified with an element of the collection located at `?property`.
`(child+ ?node ?child)`	Unifies `?child` with any descendant of `?node`.
`(parent ?node ?parent)`	Unifies `?parent` with the direct parent of `?node`
`(parent+ ?node ?parent)`	Unifies `?parent` with an ancestor of `?node`
`(ast-compilationunit|encompassing ?node ?cu)`	Unifies `?cu` with the surrounding compilation unit of the arbitrary AST node `?node`.
`(ast-type|encompassing`	Unifies with the surrounding type declaration of the arbitrary AST node `?node`.
`(method-method|caller ?method ?caller)`	Unifies `?caller` with a method that calls `?method`

5.4 Supporting Revision Characteristics through Ekeko

Ekeko [19] is a declarative program query language, implemented in Clojure, enabling users to query Eclipse projects written in Java. Just like Qwal, it makes use of `core.logic` as its declarative engine. Characteristics of the sought-after source code are specified declaratively by means of a logic query. The conditions of such a query quantify over the source code of the program. Solutions to the query, computed by Ekeko, correspond to the sought-after code.

We have opted to use a declarative PQL and more specifically Ekeko to support revision characteristics in QwalKeko. Declarative PQL have proven themselves successful in querying source code, as discussed in Section 3.1. We have opted for Ekeko as it already supports criteria **C1** — expressing revision characteristics — **C5** — query reuse, abstraction and composition — and **C6** — providing solutions in an on-demand manner —.

Most declarative program query languages specify source code by transforming it to logic terms. This way, the terms can be bound to logic variables. Ekeko foregoes such a transformation, and instead leaves the reified version of an AST node as the AST node itself (i.e., an instance of `org.eclipse.jdt.core.dom.ASTNode`). Such *identify-based* reification prevents queries from returning stale results that no longer reflect the current state of the workspace. It also brings some practical advantages, such as allowing the reuse of existing infrastructure provided by Eclipse, and facilitates interoperability with other Eclipse plugins.

Ekeko provides a library of predicates that can be used to query programs. These predicates reify basic structural, control flow and data flow relations about the queried program, as well as higher-level relations that are derived from the basic ones. Table 5.3 depicts the predicates used most commonly in this thesis.

They are the ones reifying structural relations maintained by the Eclipse Java development tools (JDT)[7]. Binary predicate (ast ?type ?node), for instance, reifies the relation of all AST nodes of a particular type. Here, ?type is a Clojure keyword denoting the capitalized, unqualified name of ?node's class. Solutions to the query (ekeko [?inv] (ast :MethodInvocation ?inv)) therefore comprise all method invocations in the source code. The ternary predicate (has ?property ?node ?value) reifies the relation between an AST node and the value of one of its properties. Here, ?property is a Clojure keyword denoting the decapitalized name of the property (e.g., :modifiers). In general, ?value is either another AST node or a wrapper for primitive values (e. g., integers), null, or collections. This wrapper ensures the relationality of the predicate. The following query retrieves nodes that have null as the value for their :expression property (i. e., receiver-less invocations):

```
(ekeko [?node]
  (fresh [?exp]
    (nullvalue ?exp)
    (has :expression ?node ?exp)))
```

The child/3 predicate reifies the relation between an AST node and one of its immediate AST node children. Solutions to the following query therefore consist of pairs of a method invocation and one of its arguments:

```
(ekeko [?inv ?arg]
  (ast :MethodInvocation ?inv)
  (child :arguments ?inv ?arg))
```

Next to facilitating the use of query results in tools and preventing queries from returning stale results that no longer reflect the current state of the workspace, our identity-based reification of AST nodes also brings along some practical implementation advantages. Many predicates call out to Java whenever this is more convenient than a purely declarative implementation:

```
(defn ast-parent [?node ?parent]
  (fresh [?kind]
    (ast ?node ?ast)
    (!= null ?parent)
    (equals ?parent (.getParent ?node))))
```

Here, the last line uses equals/2, which first unwraps any logic variables in its second argument, so that the method ASTNode.getParent() can be called on it. This result is unified with the first argument ?parent. The before-last line ensures that the predicate fails for CompilationUnit instances that function as AST roots.

[7] http://www.eclipse.org/jdt/

5.4.1 Integrating a PQL into a History Query Language

As a PQL, Ekeko enables querying one or more Eclipse projects, as indicated by the user, against which logic queries are evaluated. In general, a PQL is used to query a single, fixed, revision of a software project. History queries, in contrast, span multiple revisions of a versioned software project. Integrating a PQL into a history query language such that existing predicates can be reused is challenging.

V-Praxis [58](cf., Section 3.2) uses an approach in which every predicate of the PQL receives an additional argument that is bound to the queried revision. Binding this argument to the correct revision is left to the history query language, and more specifically, its temporal specification language. The reification of the source code is also adapted – every fact gets the same extra revision argument. As a result, there is but a single data structure representing the history of the software project under investigation.

An alternative solution exists when the PQL can be configured to change the data it queries at run time. In the context of history querying a separate data structure could be used for each revision. Throughout the execution of a history query the PQL is configured so that it queries the correct data structure. The difficulty lies in ensuring that, upon backtracking, this configuration is updated accordingly. Done correctly, the predicates provided by Ekeko can be used in history queries without requiring modification.

Figure 5.4 illustrates the low-level implementation of `in-source-code`. It is implemented as a macro that takes as its first argument a singleton vector containing a variable bound to the node of the revision graph representing the current revision, and an arbitrary number of goals that have to hold in that revision. It returns a goal adhering to the specifications of Qwal (cf., Section 5.2), and thus it can be used in a Qwal query. This predicate will be called by Qwal with its first argument bound to the VCS graph, its second argument to the current revision node and the predicate must unify the third argument with the next revision of the RPE. As `in-source-code` does not navigate the revision graph, the next revision can remain the same revision as the current one. Lines 5–6 ensure that the current revision is configured and built as a separate Eclipse project, and that Ekeko predicates quantify over this project. Next, line 7 evaluates all the goals provided by the user. Line 8 unifies the next variable with the current revision. Finally, lines 9–11 ensure that, upon backtracking, the current revision is restored correctly. The low-level interface of `core.logic`, which exposes the substitution map, is used to implement this side-effect within the backtracking. A lambda is created that takes as argument the current variable substitutions, and that restores the correct revision. Unlike line 6 this must be done in this manner to enforce the evaluation of `set-current` upon backtracking.

PQLs, and thus Ekeko, are tailored towards querying a single revision of a software project. A history query language on the other hand retrieves source code

```
(defmacro in-source-code [[revision] & goals]
  `(fn [graph# ~revision next#]
     (project [~revision]
       (all
         (ensure-checkout ~revision)
         (set-current ~revision)
         ~@goals
         (== ~version next#)
         (fn [subs#]
           (set-current ~revision)
           subs#)))))
```

Figure 5.4: Code listing depicting the implementation of `in-source-code` and how the current version is updated.

elements from different revisions. Thus, a history query language must provide facilities to retrieve the same code entity (e. g., a class or a method) across different revisions.

Absinthe [50](cf., Section 3.2) uses an approach in which a single code entity has a single representation, regardless of the current revision of the query. The values that are returned from accessing properties of this representation do depend on the current revision. This approach has as an advantage that the same logic variable can be used throughout a history query for a single code entity. The disadvantage is that Absinthe builds a Hismo [39] model beforehand which tracks the location of these code entities.

QwalKeko does not opt for such an approach, as building such a model is time and memory consuming. Absinthe can build a Hismo representation efficiently as the used VCS has knowledge of the semantics of the program language of the versioned project. Building such a model in QwalKeko would require importing the source code of every modified file of each revision. Instead, QwalKeko provides predicates that enable users to retrieve an element found in one revision in the current revision. These predicates rely on the name of an entity to find it in the current revision. For example, the predicate `type-type|corresponding` unifies its second argument with the corresponding type declaration of the first argument. To this end, it finds a type declaration within the same package and the same name as the original type declaration. Table 5.4 shows an overview of these predicates. This name-based tracking has as a disadvantage that it cannot handle renames, nor can it be used for any arbitrary code element, such as if statements, method invocations etc. In Chapter 7 we discuss a change-based approach which provides tracking of arbitrary code elements.

Table 5.4: Predicates that find the corresponding element of elements retrieved from a different revision.

Predicate	Description
`(compilationunit-compilationunit|corresponding orig ?corr)`	unifies `?corr` with the compilation unit corresponding to `orig`, based on the location of the original compilation unit.
`(type-type|corresponding orig ?corr)`	unifies `?corr` with the type declaration (i. e., a class declaration or an interface) corresponding to `orig`, based on name of the original type declaration.
`(method-method|corresponding orig ?corr)`	unifies `?corr` with the method corresponding to `orig`, based on the signature and corresponding compilation unit of the original method.

5.5 Evaluation: Answering History Questions using QwalKeko

Section 5.2 introduced Qwal and Section 5.4 introduced Ekeko. In this section, we illustrate the expressiveness and applicability of the combination of both in the history querying tool QwalKeko. To this end, we provide queries that answer several history-related questions. We provide queries for three sets of questions: the first set is related to questions developers ask regarding the history of a system's source code. The second set is related to questions managers might be interested in about the development process of their team. The diversity of these queries demonstrates the broad applicability of a general-purpose history querying tool. The final set is related to questions a researcher working in the field of mining software repositories might want answered.

Using these queries, we evaluate QwalKeko on the criteria for a general-purpose history querying tool introduced in Section 2.4.

5.5.1 History Queries for Answering Questions Developers Ask

We begin by providing queries that answer questions that developers commonly ask, as identified by two surveys [36, 51]. We have discussed some of these questions in Section 2.3.

Identifying Co-Authors of a Class Owned by a Developer

In our first example we introduce a query that retrieves all developers who have modified classes owned by a particular developer. A developer owns a class when he or she introduced that class.

We build this query *incrementally*. First, we create a query that finds the version in which a specific class named `Evaluator` was introduced. Second, we create a query that finds all the classes introduced by a certain developer. We finish with the query that answers the history-related question.

Find the revision in which `Evaluator` was introduced The following query finds the revision in which a class named `Evaluator` was introduced:

```
1  (qwalkeko* [?end]
2    (qwal graph root ?end [?classA ?classB ?nameA ?nameB]
3      (q=>*)
4      (in-source-code [curr]
5        (fails
6          (ast :TypeDeclaration ?classA)
7          (has :name ?classA ?nameA)
8          (name|simple-string ?nameA "Evaluator")))
9      q=>
10     (in-source-code [curr]
11       (ast :TypeDeclaration ?classB)
12       (has :name ?classB ?nameB)
13       (name|simple-string ?nameB "Evaluator"))))
```

Line 1 launches the history query using the `qwalkeko*` special form, returning all revisions in which the class `Evaluator` is introduced. Multiple solutions are possible when the class is introduced in multiple branches, or removed and reintroduced at a later point. Next, line 2 launches a regular path expression using the `qwal` special form over the predefined revision `graph`, starting in `root` and ending in `?end`. It introduces several new logical variables, namely `?classA`, `?classB`, `?nameA` and `?nameB`. Line 3 skips an arbitrary number of revisions using the `q=>*` operator with an empty body. Lines 4–8 specify that the current revision should not contain the `Evaluator` class. They use the `in-source-code` special form to specify revision characteristics in the current revision should exhibit. This current revision is bound to `curr`. Lines 6–8 try to find a type declaration `?classA` that is named `Evaluator`. By wrapping these in a `fails` primitive, line 5 ensures that no such class is present in the current revision. Line 9 skips a single revision. Next, the query requires the `Evaluator` class to be present in the current revision. This means that the current revision is the one that introduced the class as it was not present in the previous revision. To this end, lines 10–13 contain the same code as the fails body, and look for this class. The path expression succeeds if such a pair of successive revisions can be found, and binds `?end` to the final revision.

One problem with this query is that it only works when the name of the class is known beforehand. We use negation-as-failure to require that a class with a specific name is absent. As a result, replacing the class' name with an unbound logic variable would result in finding a revision in which *no* class is present. Alternatively, the revision introducing an element can be retrieved by finding a revision in which the element is present, that is preceded by a revision in which it is not. This would eliminate the problem caused by our use of negation-as-failure as follows:

```
 1 (qwalkeko* [?introduced]
 2   (fresh [?end]
 3     (qwal graph root ?end [?classA ?classB]
 4       (q=>*)
 5       (in-source-code [curr]
 6         (ast :TypeDeclaration ?classB)
 7         (type-name|string ?classB "Evaluator")
 8         (== ?introduced curr))
 9       q<=
10       (in-source-code [curr]
11         (fails
12           (ast :TypeDeclaration ?classA)
13           (type-name|string ?classA "Evaluator")))))
```

The query first identifies a revision in which the class `Evaluator` is present, and then verifies that this class is not present in any of the revision's predecessors using the `q<=` operator. It uses some higher-level predicates out of succintness concerns. For example, the predicate `type-name|string/2` binds its second argument to the string representation of the name of the class passed as its first argument. Note that the regular path expression no longer ends in the revision in which the element was introduces. To reproduce the behavior of the previous query, a fresh variable is introduced on line 2 that will contain the end revision of the path expression. The query returns all solutions for `?introduced`, bound on line 8 to the current revision of the path expression at that stage.

Finding which classes are introduced by a specific author The previous query identified the version in which the `Evaluator` class was introduced. The following query generalizes the previous one such that it identifies *any* class introduced by Bob. We also introduce a new predicate `type|absent` that takes as input a bound variable, and succeeds when the current revision does not contain a type with the same name.

```
 1 (defn type|absent [type]
 2   (fresh [?type ?name]
 3     (type-name|string type ?name
 4     (fails
 5       (ast :TypeDeclaration ?type)
 6       (type-name|string ?type ?name)))))
 7
 8 (qwalkeko* [?class]
 9   (fresh [?end]
10     (qwal graph root ?end []
11       (q=>*)
12       (in-source-code [curr]
13         (ast :TypeDeclaration ?class)
14         (version-author curr "Bob"))
15       q<=
16       (in-source-code [curr]
17         (type|absent ?class)))))
```

Lines 1–6 introduce a new, reusable predicate `type|absent`, to which the code from the original path expression is extracted. This predicate is used on line 17. Line 14 uses the predicate `version-author/2` , which succeeds when its second argument unifies with the author of the given version.

Finding who changed classes introduced by a specific author Our final query answers the original question: "who changed classes introduced by Bob?".

```
1  (defn type-type|corresponding [original ?corresponding]
2    (fresh [?name]
3      (type-name|string original ?name)
4      (type-name|string ?corresponding ?name)))
5
6  (defn type|absent [type]
7    (fresh [?corresponding]
8      (fails
9        (type-type|corresponding type ?corresponding))))
10
11 (qwalkeko* [?author]
12   (fresh [?end]
13     (qwal graph root ?end
14       [?classA ?classB]
15       (q=>*)
16       (in-source-code [curr]
17         (ast :TypeDeclaration ?classA)
18         (version-author curr "Bob"))
19       q<=
20       (in-source-code [curr]
21         (type|absent ?classA))
22       (q=>+)
23       (in-source-code [curr]
24         (type-type|corresponding ?classA ?classB)
25         (type|modified ?classB)
26         (version-author curr ?author)
27         (!= ?author "Bob")))))
```

We introduce another predicate `type-type|corresponding/2` that further extracts common code. This predicate is now used by `type|absent/1` to further reuse query code. A path expression is launched on line 11, with a similar beginning to the previous one. Lines 16–21 find a class that was introduced by "Bob". Lines 23–27 look for a revision in which a class with the same name was modified by a different author. To this end, the predicate `type|modified/1` unifies its argument with a type declaration that resides in a file that is modified in the current revision. Finally, we verify that the author of that revision is different from "Bob". This query can easily be generalized to find classes that were modified by a person different from the original author by replacing "Bob" by a logic variable.

Identifying Re-introduced Methods Our second example identifies methods deleted from a class at one particular point in time, but added again afterwards to the same class —possibly under a different name but with the same body. The following query identifies such methods:

```
 1 (qwalkeko* [?method]
 2   (fresh [?end]
 3     (qwal graph root ?end
 4       [?classA ?classB ?body ?introduced ?introbody]
 5       (q=>*)
 6       (in-source-code [curr]
 7         (ast :MethodDeclaration ?method)
 8         (ast-type|encompassing ?method ?classA)
 9         (has :body ?method ?body))
10       q=>
11       (in-source-code [curr]
12         (method|absent ?method))
13       (q=>*)
14       (in-source-code [curr]
15         (ast :MethodDeclaration ?introduced)
16         (ast-type|encompassing ?introduced ?classB)
17         (type-type|corresponding ?classA ?classB)
18         (has :body ?introduced ?introbody)
19         (ast-ast|same ?body ?introbody))
20       q<=
21       (in-source-code [curr]
22         (method|absent ?introduced)))))
```

Lines 3–4 launch a RPE over the revision graph, starting in `root` and ending in `?end`. A number of logic variables used by the regular path expression are also introduced. Line 5 skips an arbitrary number of revisions. Next, lines 6–9 retrieve a method `?method` from the current revision, its encompassing class `?classA` and the body of that method. Line 10 advances a single revision. Lines 11–12 verify that the method `?method` is no longer present by using `method|absent` (cf., Section 5.4). So far, the query has identified a method that is removed in a later revision. The next part of the query identifies a newly introduced method in the same class that has the same body as the earlier removed method. Line 13 skips an arbitrary number of revisions, until a revision is encountered in which a method `?introduced` with the same body is introduced. Finally, lines 20–22 ensure that the method was introduced by moving back a single revision and failing to find a corresponding method.

5.5.2 History Queries for Verifying a Development Process

We now shift our focus to history-related questions managers of a development team might want answered. These questions concern the development process of the software project.

Finding Violations against Test-Driven Development

In the first example, a manager needs to find violations against the principle of test-driven development (TDD) which advocates writing tests prior to implementing the tested functionality. The manager wants to assert that the team actually adhered to TDD. The question that the manager needs answered is "what are the methods in the system, for which a unit test was added after the method's introduction?".

The following QwalKeko query assumes that a predicate `test-tested|method/2` exists. Although we do not provide a definition for this predicate, this predicate links a unit test to a method through a naming convention or program analysis.

```
1  (qwalkeko* [?methodA ?end]
2    (qwal graph root ?end
3      [?test ?methodB ?method-test]
4      (q=>*)
5      (in-source-code [curr]
6        (ast :MethodDeclaration ?methodA)
7        (fails
8          (test-tested|method ?test ?methodA)))
9      q=>
10     (in-source-code [curr]
11       (method-method|corresponding ?methodA ?methodB)
12       (test-tested|method ?method-test ?methodB))))
```

The query launches a Qwal query, starting in `root` and ending in `?end`. On line 4 it starts by skipping an arbitrary number of revisions using the `q=>*` operator. Next, lines 5–8 state that a revision must exist, in which a method `?methodA` has no corresponding unit test. Line 9 skips an arbitrary, non-zero number of revisions. Lines 10–12 state that in the current revision a unit test is added. To this end, `method-method|corresponding` retrieves the corresponding method in the current revision. The query returns all methods `?methodA` that violate the test-driven development principle, for which a unit test is present in a later revision `?end`.

5.5.3 History Queries for Mining Software Repositories

We now provide examples, from the domain of mining software repositories. As discussed in Section 2.3 the domain of mining software repositories analyzes information in software repositories to better understand software development.

Identifying "Zombie Code"

In the following example we illustrate the use of QwalKeko to detect a, what we coin, temporal bad smell. This is a bad smell [34] that only becomes apparent when analyzing the evolution of the source code of a system. In particular, we introduce the concept of *zombie code*. We define this temporal bad smell as methods in the system that were stopped being used in a particular version but that are not removed (i. e., become dead code), and that are used again in a later version. While the presence of zombie code is not necessarily problematic, instances of this bad smell might point at uses of implicitly deprecated code.

The following QwalKeko query retrieves instances of zombie methods:

```
1  (qwalkeko* [?zombie]
2    (fresh [?end]
3      (qwal graph root ?end []
4        (q=>*)
5        (in-source-code [curr]
6          (fresh [?caller]
7            (ast :MethodDeclaration ?zombie)
8            (method-method|caller ?zombie ?caller)))
9        q=>
10       (q=>+
11         (in-source-code [curr]
12           (fresh [?caller ?corresponding]
13             (method-method|corresponding ?zombie ?corresponding)
14             (fails
15               (method-method|calls ?zombie ?caller)))))
16       (in-source-code [curr]
17         (fresh [?caller ?corresponding]
18           (method-method|corresponding ?zombie ?corresponding)
19           (method-method|caller ?zombie ?caller))))))
```

This query begins with the usual configuration of QwalKeko and Qwal. Line 4 skips an arbitrary number of revisions, until a revision is encountered in which a method `?zombie` is called by `?caller`. Next, line 9 skips a single revision using the `q=>` operator. Lines 10–15 state that the zombie method is no longer being called for an arbitrary, non-zero number of revisions. Remember that the `q=>+` operator will first prove the goals passed as its arguments before skipping a single revision, which is why line 9 skipped a revision. Finally, the last 4 lines specify that the zombie method should be called again.

5.6 Discussion

In the following section we discuss the currently described state of QwalKeko and how it adheres to the criteria stipulated in Section 2.4.

C1: Revision Characteristics QwalKeko converts a VCS into a graph-based representation, which contains a coarse-grained representation of every revision (cf., Section 5.1.1). By reifying this data and providing a declarative layer, QwalKeko enables expressing *coarse-grained* characteristics. Declarative program query languages have proven themselves successful in querying a single revision of a software project. By integrating Ekeko into QwalKeko (cf., Section 5.4). Ekeko supports expressing *fine-grained* revision characteristics. We have shown this expressiveness through the different example queries, which are representative of the different application domains stipulated in Section 2.3.

QwalKeko supports expressing coarse and fine-grained revision characteristics.

C2: Temporal Characteristics QwalKeko features the graph query language Qwal (cf., Section 5.2). Qwal enables users to express regular path expressions through the aforementioned revision graph. Qwal provides an extensible implementation that enables users to provide their own navigation predicates. We have shown through the examples that Qwal supports expressing temporal characteristics. Its bi-directional navigation eases the expression of some revision characteristics, such as the introduction or removal of code elements.

QwalKeko supports expressing temporal characteristics.

C3–4: Change and Evolution Characteristics QwalKeko as it has been introduced up until now does not feature support for fine-grained source code changes, and does not support expressing change and evolution characteristics. We introduce these in Chapters 6 and 7.

The currently described state of QwalKeko does not support expressing change and evolution characteristics.

C5: Query Abstraction, Reuse and Composition QwalKeko enables query abstraction, reuse and composition in several ways. Users can define their own predicates that can be used in queries and other predicates. Users can define their own temporal operators, either by combining existing operators or by implementing a logic goal that adheres to Qwal's interface.

QwalKeko supports query abstraction, reuse and composition.

C6: On-demand Solutions QwalKeko combines a *declarative* graph query language with a *declarative* revision query language. This declarative nature enables computing solutions in an on-demand fashion.

QwalKeko supports the computation of solutions in an on-demand manner.

5.7 Conclusion

In this chapter we discussed the *declarative* programming query language Ekeko. It enables users to query Java projects stored in Eclipse by reifying the AST representation of Eclipse, and combines this with core.logic, a declarative engine for Clojure. We combine Ekeko with Qwal, a graph query language implementing regular path expressions, to query the history of a software project. Regular path expressions enable specifying a path throughout a graph of revisions, while Ekeko enables specifying the source code characteristics that have to hold in revisions along this path. The declarative nature of Ekeko enables users to describe revision characteristics in a high-level language, while the identification of elements adhering to these characteristics is performed by the underlying reasoning engine. We have demonstrated the use of QwalKeko in answering history-related questions, stemming from different application domains such as program comprehension as well as the mining software repositories.

The currently described state of QwalKeko adheres to criteria C1, C2, C5 and C6. The missing ingredient is support for expressing change and evolution characteristics (see Section 2.2.2). Chapter 6 discusses how fine-grained changes can be retrieved, and how change characteristics can be supported. Chapter 7 discusses the final concept of QwalKeko, the support for evolution characteristics.

6 SUPPORTING CHANGE CHARACTERISTICS

In the previous chapter we introduced a version of QwalKeko that supports the expression of temporal and revision characteristics. In this chapter we focus on the third criterion for a general-purpose history querying tool (cf., Section 2.4); support expressing change characteristics. First, Section 6.1 motivates the need for dedicated support specifying change characteristics. Next, Section 6.2 discusses how to procure fine-grained AST changes. Two approaches exist, namely change logging and change distilling. A change logger produces changes by recording the actions of developers *while* they are developing the software project in an IDE. A change distiller produces changes between any two files in an algorithmic manner. We focus on some of the implementation details of change distilling algorithms. Understanding these details is needed to fully grasp the challenges of querying their output.

Section 6.3 discusses existing approaches to querying distilled and/or logged source code changes. Section 6.4 introduces a declarative API to query the output of our own distilling algorithm called ChangeNodes. We extend QwalKeko with a declarative API that enables expressing change characteristics in queries. We validate this extension using an empirical study in which we investigate which parts of Selenium[1] tests in mature open-source projects are most prone to change. We implement this study twice; once using QwalKeko and once using Clojure in combination with the raw output of ChangeNodes. We compare both implementations and assess how they handle the different concerns of the study.

[1]`http://www.seleniumhq.org/`

6.1 The Need for Dedicated Support for Specifying Change Characteristics

As discussed in Section 2.3, fine-grained code changes are needed to perform studies concerning the evolution of source code. A general-purpose history querying tool must procure these changes and, as stipulated by criterion C3, support specifying their characteristics. Fine-grained changes cannot be retrieved directly from most version control systems, which store changes at the level of line changes (i. e., the addition or removal of a single line). As line changes do not convey sufficiently fine-grained information about the modifications made to the source code between two commits, it does not suffice for a general-purpose history querying tool to support querying them. Two other approaches exist, which are detailed in Section 6.2, change logging and change distilling. The former records the interactions of a developer inside the IDE *during* the development of the software project. The latter is an algorithmic approach that takes as input two versions of a file, and outputs a sequence of changes that when applied in order transforms the original version of the file into the new one.

The specification of change characteristics and specification of revision characteristics are entangled. Fine-grained source code changes operate on source code, and thus the specification of change characteristics (e. g., "Does a change exist that adds a new method?", or "Does a change exist that moves a return statement to an if statement?") also involves the description of source code characteristics. To facilitate satisfying criterion C5 – supporting query reuse, abstraction and composition – we argue that both kinds of characteristics should be specified in the same language. A single uniform specification language for all characteristics also lowers the learning curve for users of the tool.

6.2 Retrieving Fine-grained AST Changes

We first discuss from where to obtain fine-grained change information. Fine-grained change operations are the low-level operations that were applied to the individual nodes of an AST in order to transform it from one revision to the next revision. VCS generally store changes at the level of line modifications. These line modifications only consider the addition and removal of lines, but not a line being moved. These moves are represented by the removal and addition of a line. Thus, VCS do not suffice as a source for fine-grained source code changes. We discuss two approaches that do provide fine-grained changes; change logging and change distilling.

Change Logging

A change logger records the operations a developer performs inside the IDE *while* developing an application. Several such logging approaches exist, such as SpyWare [66], CodingTracker [60], Fluorite [77] and Syde [44]. The recorded operations can also include invocations of refactoring tools, interactions with the underlying VCS, copy-pasting code etc. Change loggers track changes that are invisible to a VCS. For example, a developer may make several attempts to implement a certain feature, but only commit the final result. These attempts are invisible when looking at the code in the VCS, yet a change logger provides sufficient information to uncover these.

The main advantage of change loggers is that they provide a complete, accurate representation of the actions the developer performed. The main disadvantage is that a change logger must be installed inside the IDE of every developer working on the project. As a result, logging data is not readily available for most projects. Second, replaying the logged operations will result in many intermediate states that cannot be parsed. For these reasons we opt for a different approach, namely change distilling.

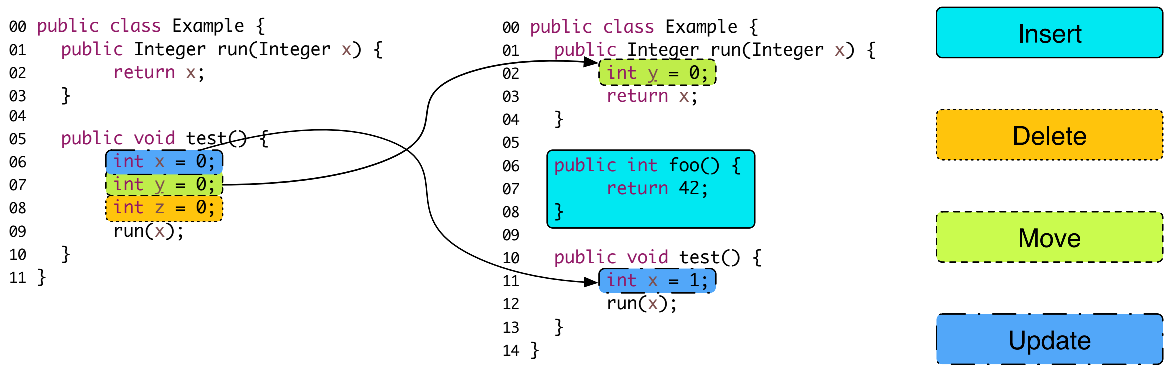
Change Distilling

A change distiller implements an algorithmic approach to retrieving changes from any pair of source files. To this end, a change distiller takes as input two ASTs, called the *source* and *target* AST, and outputs a minimal sequence of change operations that, when applied *in order* to the source AST, transforms it into the target AST. These changes can either be an insert of a node, an update of its properties, its removal or its moving to a different location. A change distiller tries to output a minimal sequence of changes that reuses as many nodes as possible from the source AST.

ChangeDistiller [33] implements such a distilling approach, computing changes between Java source code files. It is based on the differencing algorithm of Chawathe et. al [11], originally intended for LaTeX files. Gumtree [30] improves upon ChangeDistiller with more fine-grained changes by operating upon a more detailed representation of the source code. In general, these distilling approaches provide similar operations.

The main advantage of using a change distiller is that it can be used on any pair of source files. As such, it can be used for any software project stored in a VCS. The main disadvantage is that the outputted change sequence does not necessarily match the concrete edits a developer performed. The algorithm only works for a pair of files, and does not track source code that was moved to a different file.

In this dissertation we focus on a distilling approach as it can, unlike change logging, be applied to any versioned software project. We have implemented a change



```
public class Example {
    public Integer run(Integer x) {
        return x;
    }

    public void test() {
        int x = 0;
        int y = 0;
        int z = 0;
        run(x);
    }
}
```

```
public class Example {
    public Integer run(Integer x) {
        int y = 0;
        return x;
    }

    public int foo() {
        return 42;
    }

    public void test() {
        int x = 1;
        run(x);
    }
}
```

Figure 6.1: Example of the output of a distilling algorithm.

distiller called ChangeNodes.[2] ChangeNodes consists of two parts. First, it enables users to distill changes between two Eclipse Java source files. Second, it provides a declarative API for these changes that is used by QwalKeko. At its heart lies the same algorithm as ChangeDistiller. The main difference between both implementations is that ChangeDistiller uses its own AST representation instead of the one provided by Eclipse. ChangeNodes represents its changes using Eclipse AST nodes. This facilitates its integration with Ekeko and QwalKeko. The AST representation of Eclipse is also more fine-grained, resulting in more fine-grained change information.

6.2.1 The Inner Workings of a Change Distilling Algorithm

The distilling algorithm described by Chawathe et al. [11], used by all of the aforementioned change distillers, works in two phases. First, the *matching* phase establishes which nodes are considered to be equal between the source and the target AST. Second, the *differencing* phase makes use of the matching to output change operations. Chawathe et al. assume such a matching already exists, and only describe the differencing phase. The work of Fluri et al. [33] focuses on heuristics that provide an adequate matching between two ASTs. The more accurate the matching, the fewer redundant and unneeded change operations are output.

Figure 6.1 depicts a simple example, in which an `Example` class underwent some changes. A new method `foo` has been added, the initializer expression of variable `x` has been updated, variable `y` has been moved to a new location and variable `z` was removed. Modified nodes that match are indicated with arrows. The matching of unmodified nodes is not depicted. As newly added method `foo` has no matching nodes in the original source code, an insert operations is outputted by the distiller. As variable `z` has no matching node in the target AST, a removal operation is outputted.

[2] `https://github.com/ReinoutStevens/ChangeNodes`

The algorithm's matching phase takes as input two ASTs, and outputs a mapping between nodes of the source AST to nodes of the target AST that match. Two matching nodes (and their subtrees) are not necessarily equal – they are just sufficiently similar. For example, line 6 in Figure 6.1 depicts a node that is not completely identical to its matching counterpart. We do not detail the heuristics used to determine such matching, as they have been discussed in other work [33]. Examples of these heuristics include the Levenshtein distance between the string representation of two nodes or the ratio of matching and non-matching children between two nodes, which is also used by ChangeNodes.

The differencing phase takes as input two ASTs and a matching between nodes in these ASTs. Next, it navigates the target AST using a breadth-first traversal, outputting the correct change operations. For example, an insert operation is created when a node is encountered that has no a matching node in the source AST. Whenever a change operation is created, it is also applied immediately to the current state of the source AST and the matching is updated to reflect this change. This is required to maintain the assumption that, at every step of the traversal, the parent of the current node has a matching counterpart. To prevent modifications to the original source code, a copy of the source AST is taken. In this dissertation we call this copy `source'`, and nodes residing in this AST are also indicated by a quote. Note that, even though `source'` starts as a copy of the original source code, it will be identical to the target AST after executing the algorithm. The representation of the changes that are output therefore refers to one of three ASTs, `source`, `source'` or `target`.

6.2.2 Detailed Change Definitions

We now provide a definition of the different kinds of change operations that are produced by ChangeNodes. Eclipse nodes can have three different kinds of children, accessed via so-called properties: child properties, list properties and simple properties. The difference lies in the values they can assume. A child property has another AST node as its value, a list property has a list of AST nodes as its value, and a simple property has a regular Java object that is not an AST node as its value (e.g., a String). More importantly, some properties are mandatory, meaning that the AST node must always have a non-null value for them. The "name" property of a `MethodDeclaration` node is an example of a mandatory property.

We now define the semantics of the different change operations that are output by ChangeNodes. These definitions are more detailed than those from the literature (i.e., [30, 33]), but we found that the additional detail is implicitly present in most existing distiller implementations (and could hence be made explicit in their supporting publications). As a distilling algorithm applies changes during its execution, thereby modifying the AST, a copy of the source AST is taken.

`insert(node',original,parent',removed',property,index)`
: A `node'` is inserted at location `property` of node `parent'`. In case `property` is a child list property, the node is inserted at index `index`. Applying the insert will only add a minimal representation of `node'` to `parent'`. Node `original` is the parent in the original AST, and can be `null` if the insert is part of the subtree introduced by another change operation. Finally, `removed'` is the node that the insert operation overwrote in case a node was already present in `original` at location `property`. If such a node was present its child nodes are added to `node'` so that later change operations can use them. If no such node was present this value is `null`. During the execution of the algorithm the removed node is still accessible through the matching data structure. For example, the overwritten node can still be moved to a different location.

`move(node',original,parent',preparent',property,index)`
: A `node'` is moved from `preparent'` to location `property` of node `parent'`. In case `property` is a list property, the node is moved to index `index`. Only a minimal representation of the node will be moved. Its original location is replaced by a placeholder node that still contains the subtree located at `node'`. Thus, only the node is moved, and not the node and its complete subtree. `original` is the representation of `node'` in the original AST.

`update(node',original,property,value)`
: The value of node `node'` at location `property` is updated to `value`. `property` must be a simple property. As such, the value will be a Java object, and not an AST node. `original` is the representation of `node'` in the original AST.

`delete(node',original,parent',property,index)`
: A node `node'` and its complete subtree are removed at the value of `property` in `parent'`. In case `property` is a list property, `index` indicates the index of `node'` in its list. Note that `node'` will not be present in source′ as the change has already been applied. As such, `node'` will not have a parent node. The parent node before the application of the delete is captured by `parent'`. `original` is the representation of `node'` in the original AST.

Note that moves and inserts produce minimal representations of an AST node (as defined above), even if that AST node has a large subtree. This subtree will be introduced by subsequent change operations. For example, a minimal method declaration node can be inserted, after which its body can be created by moving a pre-existing piece of code. Looking back at our initial example from Figure 6.1, the depicted introduced method `foo(){...}` will actually be distilled as several seperate insert operations, each one introducing new parts of the method.

6.3 Working with Source Code Changes

In the following section we describe existing approaches that work with or query fine-grained source code changes. These approaches do not necessarily operate in the context of history querying.

JET JET [40] is a tool for analyzing subsequences of changes and characterizes dependencies between these changes. It is used in the context of change integration, where changes made in one branch of a system need to be applied to a different branch. This requires integrators to manually inspect changes made in either branch to see how they can be integrated. JET supports integrators in recovering dependencies between changes, in assessing how the integration of changes would impact the target branch and in detecting whether the same source code entity is modified multiple times by different changes. Just as CheOPS (cf., Section 3.3), JET models these changes based on modifications made to a FAMIX [67] representation of the source code. It provides several views of the changes and their dependencies to aid integrators.

JET targets a single group of users, namely integrators. As such, it supports them by helping with the exploration and manual inspection of change sequences. It does not help with programmatically querying and analyzing changes, which is needed in the context of MSR studies.

Lase Lase [56] is a tool that generates a single context-aware edit script that automates multiple systematic edits, provided by means of examples. To this end, Lase distills changes between modified methods, identifies common subsequences among the corresponding change sequences, generalizes identifiers in these sequences to meta-variables and finally extracts the common edit context. This approach requires comparing individual changes, but not reasoning about the change script as a whole. This is identified by the authors as one of the weak points of the approach. Reasoning about the change script as a whole is difficult for two reasons. First, a change distilling algorithm only returns but one of many possible orderings between changes (e. g., changes that modify different parts of the code can be interchanged in the change sequence). Lase assumes that similar code edits result in similar change sequences where changes appear in the same order. Second, the identification of common subsequences does not account for different change sequences implementing the same transformation. It is assumed a distilling algorithm returns the same sequence for the same code transformation.

The limitations of Lase further motivate the need for a dedicated and general-purpose history query language.

```
1  @r@
2  expression x;
3  flexible expression list es;
4  identifier f != pci_enable_msix;
5  @@
6  x =
7  - pci_enable_msix
8  + f
9      (es)
10
11 @script:python@
12 f << r.f;
13 @@
14 print f
```

Figure 6.2: A Prequel query searching for replacements of `pci_enable_msix`. This example is taken from [53].

Prequel Prequel [53] is a tool for querying line changes in the history of C projects. Prequel is tailored towards identifying commits that (partially) contain a source code transformation Figure 6.2 depicts a Prequel query that detects replacement of a function call of `pci_enable_msix`. This example is taken from Lawall et al. [53]. The query consists out of a pattern matching rule `r` (lines 1–9) and a Python script (lines 11–14). Lines 2–5 introduce meta-variables `x`, an expression representing the return value, `es`, an expression list representing the arguments of the function call and `f`, an identifier representing the name of the function that replaces `pci_enable_msix`. Lines 6–9 specify the form of the sought-after pattern, where a call to `pci_enable_msix` is replaced by an arbitrary function `f`. The `flexible` keyword indicates that the variable `es` can match one sequence of arguments before the patch application, and a different sequence of arguments afterwards. Prequel prints the identifiers of matching commits and the text generated by the Python script at the end of the query.

Prequel is an interesting tool that enables querying large change sequences by describing the surrounding and affected source code. They use a line-based change representation and not fine-grained code changes operating on individual AST nodes, although there is no requirement that the requested changes need to align with complete lines.

6.4 Supporting Change Characteristics through ChangeNodes

Criterion C3 (cf., Section 2.4) stipulates that a general-purpose history querying tool must support change characteristics. These characteristics concern the individual changes (cf., Section 6.2.2) outputted by a change distilling algorithm. ChangeNodes distills changes between two versions of a Java AST, and outputs a sequence of changes that, when applied *in order*, transforms the original source

Table 6.1: Table depicting the libary of logic predicates to retrieve describing the characteristics of a single change.

Predicate	Description
Retrieving Changes	
`(changes ?changes source target)`	Unifies `?changes` with the change sequence distilled between the compilation units `source` and `target`.
`(change ?change source target)`	Unifies `?change` with a single change of the change sequence distilled between the compilation units `source` and `target`.
Change Characteristics	
`(change|insert change)`	Succeeds if `change` is an insert operation.
`(change|move change)`	Succeeds if `change` is a move operation.
`(change|update change)`	Succeeds if `change` is an update operation.
`(change|delete change)`	Succeeds if `change` is a delete operation.
`(change-node|original change ?original)`	Unifies the `original` node of `change` with `?original`.
`(change-property|property change ?property)`	Unifies the property of `change` with `?property`.
`(change|insert-node|parent insert ?parent)`	Unifies the parent of `insert` with `?parent`
`(change|insert-node|removed insert ?removed)`	Unifies the removed node of `insert` with `?removed`
`(change|move-node|parent move ?parent)`	Unifies the parent of `move` with `?parent`
...	Similar accessors exist for the different change operations.
Matching	
`(changes-node-node|matching changes ?nodeA ?nodeB)`	Unifies `?nodeA` with a node in the source AST and `?nodeB` with its matching node in the target AST.

```
1 (qwal graph root ?end [?left-cu ?right-cu ?change]
2   (in-source-code [curr]
3     (ast :CompilationUnit ?left-cu))
4   q=>
5   (in-source-code
6     (compilationunit-compilationunit|corresponding ?left-cu ?right-cu)
7     (change ?change ?left-cu ?right-cu)
8     (change|insert ?change)))
```

Figure 6.3: Code depicting how `change/3` is used to retrieve a single insert in the sequence of distilled changes.

AST into the target AST. In order to satisfy Criterion C3 a declarative API is provided that enables using the output of ChangeNodes in QwalKeko queries.

Table 6.1 depicts this API. The purpose of this API is two-fold. First, it facilitates retrieving changes between two revisions of a file. These predicates are depicted in the first part of the table. Second, it provides predicates enabling expressing change characteristics (e. g., a change must be an insert) and source code characteristics of the AST nodes affected by a change (e. g., a `MethodDeclaration` was inserted). These predicates are depicted in the middle part of the table. An auxiliary predicate `changes-node-node|matching/3` is also provided that exposes the matching computed by ChangeNodes between the source and target AST.

Figure 6.3 illustrates how the `change/3` predicate can be used to retrieve fine-grained changes made between two revisions of the same Java file. The query uses `change/3` to find all newly inserted AST nodes.

On line 3 `?left-cu` is bound to a compilation unit in the root revision of the graph. It moves to one of the successors of that revision on line 4. The compilation unit `?right-cu` corresponding to `?left-cu` is retrieved on lines 5–6 from that revision using `compilationunit-compilationunit|corresponding`, which looks for a compilation unit in the same package that defines the same type (cf., Section 5.4) . On line 7 the changes between these two compilation units are computed using the predicate `change/3`. The final line specifies that `?change` needs to be an insert operation. We can use Ekeko to specify more complex characteristics of the surrounding source code.

In the example we define a predicate `change-method|affected` that reifies the relation of changes affecting a particular method. It is depicted in Figure 6.4. Lines 1–4 implement `change-method|affected`, which calls the more general predicate `change-node|affected`, and verifies that the affected node is indeed a method declaration. Predicate `change-node|affected`, implemented on lines 6–15, checks what kind of change operation is provided as argument, and calls the corresponding specialized predicate. Such a predicate is depicted on lines 17–23. Predicate `change|insert-node|affected` unifies its second argument with an AST node that is affected by an insert. To this end, it introduces a local variable `?affected` that is unified with either the subject of the insert (residing in the `source` AST) or the node resulting from applying the insert (residing in the `source'` AST). Nodes affected by the insert are all the ancestors of that node, as their subtree is modified by the insert. These ancestors are retrieved using `child+`. Predicates for the different change operations are implemented using a similar pattern. For example, a delete refers to a node in the original source code, as well as a (removed) node and its parent in source′.

The example also illustrates some of the problems with querying changes directly; it requires a deep understanding of how the different changes are represented, and in what AST the nodes involved in a change resides. These problems are discussed in detail in Chapter 7, as they motivate the need for supporting evolution characteristics in history queries.

6.5 Evaluation: Expressing Change Characteristics using ChangeNodes

In the following section we evaluate whether QwalKeko extended with ChangeNodes can be used to perform studies regarding the evolution of a software project. We want to answer the following research questions:

RQ1 Can QwalKeko be used to conduct MSR studies on real-world software projects?

RQ2 How does the expressiveness and conciseness of QwalKeko compare to the general-purpose programming language Clojure's?

```
(defn change-method|affected [change ?method]
  (all
    (change-node|affected change ?method)
    (ast :MethodDeclaration ?method)))

(defn change-node|affected [change ?node]
  (all
    (conde
      [(change|update change)
       (change|update-node|affected change ?node)]
      [(change|delete change)
       (change|delete-node|affected change ?node)]
      [(change|insert change)
       (change|insert-node|affected change ?node)]
      ...)))

(defn change|insert-node|affected [insert ?node]
  (fresh [?affected ?prop]
    (conde
      [(change|insert-node|original insert ?affected)]
      [(change|insert-node|inserted’ insert ?affected)])
    (child+ ?prop ?node ?affected)))

(defn change|delete-node|affected [delete ?node]
  (fresh [?affected ?prop]
    (conde
      [(change|delete-node|original delete ?affected)]
      [(change|delete-node|parent’ delete ?affected)])
    (child+ ?prop ?node ?affected)))
```

Figure 6.4: Code listing depicting a predicate `change-node|affected` that retrieves all changes affecting a method.

To answer these research questions, we revisit a study we conducted earlier [13]. In this study we investigated how frequently automated functional tests for web applications tests are modified, and what parts of such tests are most prone to change. We answer our first research question by detailing the different steps of the study. These give an indication how QwalKeko can be used to perform other MSR studies. We answer the second research question by replicating this study without QwalKeko in Clojure, the raw output of ChangeNodes and some Java to interact with Eclipse. To this end, we directly query the AST representation of Eclipse and the output of ChangeNodes. Checking out and importing revisions is done using a Java helper class. We compare both implementations of the same study. To this end, we classify each line of code as related to one of the following concerns:

1. Expressing revision characteristics
2. Expressing temporal characteristics
3. Expressing change characteristics
4. Interacting with the VCS
5. Interacting with Eclipse

6. Interacting with the database
7. Processing the results

6.5.1 Context of the Study

Functional GUI testing has recently seen the arrival of test automation tools such as HP Quick Test Pro, SWTBot[3], Robotium[4] and Selenium[5]. These tools execute so-called *test scripts* which are executable implementations of the traditional requirements scenarios. Test scripts consist of commands that simulate the user's interactions with the GUI (e.g., button clicks and key presses) and of assertions that compare the observed state of the GUI (e.g., the contents of its text fields) with the expected one.

Although test automation allows repeating tests more frequently, it also brings about the problem of *maintaining test scripts*: as the system under test (SUT) evolves, its test scripts are bound to break. Assertions may start flagging correct behavior and commands may start timing out thus precluding the test from being executed at all. Little is known about the kind of repairs that developers perform in practice. Insights about manually performed repairs are therefore of vital importance to researchers in automated test repair.

To address this, we have analyzed open-source projects using Selenium. Selenium is a functional testing framework in which a developer scripts browser actions, and for example verifies that an application's user interface displays the desired elements. In the original paper we answered the following research questions:

SQ1 How prevalent are Selenium-based functional tests for open-source web applications? To what extent are they used within individual applications?

SQ2 Do Selenium-based functional tests co-evolve with the web application? For how long is such a test maintained as the application evolves over time?

SQ3 How are Selenium-based functional tests maintained? Which parts of a functional test are most prone to changes?

To distinguish between the research questions of our validations and the research questions of the original study we shall call the latter study questions (SQ). In this dissertation we focus on the second and third study questions. In the original paper [13] we answered the first question by querying GitHub for open-source projects that use Selenium. We identified 4287 candidate repositories that use

[3]http://eclipse.org/swtbot/
[4]https://code.google.com/p/robotium/
[5]http://docs.seleniumhq.org/

Selenium. We narrowed the number of repositories down to 287 by selecting projects that adhere to the following criteria:

1. A project needs to be at least 1 year old.
2. A project needs to have over 100 commits in the last year.
3. A project needs to be larger than 500 KBytes.

In order to answer the second and third SQ we narrowed down these projects to a corpus of 47 repositories that uses Selenium extensively. From these 47 repositories, we manually selected 8 projects. These 8 projects form our high-quality corpus, based on the number of Selenium tests present in the system. This selection excludes web frameworks and test frameworks built on top of Selenium. It also excludes test-only project repositories. As such, the high-quality corpus consists of repositories that version true web applications and their Selenium-based functional tests. Table 6.2 describes these repositories. The first two columns depict some general information about the project; the *GitHub Repository* column depicts the name of the GitHub project and the *Description* column describes the general purpose of the project. Next, the # *Commit* column depicts the total number of commits in the project. The # *Sel. Commit* column depicts the total number of commits that modified or introduced Selenium files. The *Java LoC* column depicts the total number of lines of Java code in the final revision of the project. Finally, the *Sel. LoC* depicts the total number of lines of code in Selenium files in the final revision of the project.

These projects form the basis for both the study performed in our paper as well as the validation of this chapter. These are large, mature projects that are representative for projects used in other MSR studies. In what follows we reconstruct the different steps performed for the study. We provide two versions of the source code; one version written using QwalKeko and one using Clojure, Java and the raw output of ChangeNodes. We do provide a small layer on top of the Eclipse JDT AST representation that enables describing the source code using functional variants of Ekeko's `ast`, `has` and `child`. These only provide a thin wrapper around the property descriptors used by Eclipse, but increase the readability of the code. To interact with git we make use of JGit[6], a library providing Java bindings for the git API. Performing the study using QwalKeko depicts how similar MSR studies could be performed. Comparing both versions of the code enables us to evaluate the expressiveness of QwalKeko.

The different steps are as follows: first, we need to identify Selenium files for every revision. Once we have identified these we can answer SQ2 by determining how frequently these files change. Next, we need to detect revisions in which these

[6]https://eclipse.org/jgit/

Table 6.2: The 8 repositories in the high-quality corpus.

GitHub Repository	Description	# Commit	# Sel. Commit	Java LoC	Sel. LoC
gxa/atlas	Portal for sharing gene expression data	2118	358	32,375	5,374
INCF/eeg-database	Portal for sharing EEG/ERP portal clinical data	854	17	68,262	7,158
mifos/head	Portfolio management for micro-finance institutions	7977	505	338,705	18,735
motech/TAMA-Web	Front office application for clinics	2358	239	62,034	2,815
OpenLMIS/open-lmis	Logistics management information system	4714	1153	72,275	19,195
xwiki/xwiki-enterprise	Enterprise-level wiki	688	164	28,405	13,506
zanata/zanata-server	Software for translators	3430	81	111,698	3,509
Zimbra-Community/zimbra-sources	Enterprise collaboration software	377	243	1,025,410	189,413

```
1 (qwalkeko* [?file ?cu ?end]
2   (qwal graph root ?end [ ]
3     (q=>*)
4     (in-git-info [curr]
5       (file|add ?file curr))
6     (in-source-code [curr]
7       (file|compilationunit ?file ?cu curr)
8       (compilationunit|selenium ?cu))))
```

Figure 6.5: QwalKeko code to identify Selenium tests in the history of a project.

files are modified, and compute the changes made to the test scripts. Finally, we need to classify these changes in order to detect which parts of the test code are most prone to change.

6.5.2 Identifying Selenium Files using QwalKeko

First of all, we need to identify which files of each project are Selenium scripts. To this end, we write a QwalKeko query that loops over all the revisions of the queried software project. For each revision, it inspects the newly added files and identifies whether it is a Selenium script. The latter is done by looking whether the file imports a package of which name contains "selenium". Albeit a simple heuristic we have not encountered any incorrectly identified files.

Figure 6.5 depicts a query that returns all the Selenium tests, the compilation unit and the revision of the script of the queried software project. First, an arbitrary, including zero, number of revisions is skipped using the `q=>*` operator on line 3. Next, the query binds `?file` to a newly added file. The `in-git-info` special form is evaluated without checking out code to avoid unnecessary operations. Finally, the last three lines check out the corresponding compilation unit of the added file and verify whether it is a Selenium script.

```
1 (defn compilationunit|selenium [?cu]
2   (fresh [?imp ?impname ?str]
3     (ast :CompilationUnit ?cu)
4     (child :imports ?cu ?imp)
5     (has :name ?imp ?impname)
6     (name|qualified-string ?impname ?str)
7     (string-contains ?str ".selenium")))
```

Figure 6.6: QwalKeko code that detects whether a compilation unit imports a Selenium package

Identifying whether a compilation unit is a Selenium script is done purely using Ekeko. Figure 6.6 depicts the implementation of predicate `compilationunit|selenium`. Line 2 defines three new logic variables using `fresh`. Line 3 verifies whether `?cu` unifies with a compilation unit. Line 4 unifies `?imp` with one of the import statements on that compilation unit. Lines 5–6 retrieve the name of the imported package. The last line verifies whether the name contains the string ".selenium".

The results of this query are written to a database so that they do not need to be recomputed by other predicates. We introduce a new predicate `(file|selenium file version)` which verifies whether a file corresponds to a Selenium script for a certain revision. This predicate consults this database and will be used in further examples.

6.5.3 Identifying Selenium Files using Clojure

Implementing the functionality in plain Clojure requires some scaffolding. We cannot rely on the facilities provided by QwalKeko to navigate the different commits, nor to import these as an Eclipse project. The latter is needed to have access to the different ASTs. Even though our heuristic approach to identify Selenium files based on their import statements could be done using a text search tool such as grep, we will still parse the source code using the Eclipse parser. Our motivation is two-fold. First, an AST representation is required for the change classification step. Second, parsing the source code would be required in more complex studies.

Figure 6.7 depicts this scaffolding. The function `read-git-repo` imports a repository residing at a certain location. The function `get-walker` returns an iterator for a given repository that can be used to navigate the different commits. Finally, functions `checkout-commit` and `delete-commit` ensure that an Eclipse project is created or removed containing the code of the given commit. To this end, it uses a helper Java class `ClojureMetaVersion` that handles the low-level git checkout command and importing the project in Eclipse.

This scaffolding facilitates importing git repositories, checking out a single commit and importing this commit as a separate Eclipse project. Next, we need

```
(defn read-git-repo [location]
  (let [builder (new FileRepositoryBuilder)]
    (.setGitDir builder (file location))
    (.build builder)))

(defn get-walker [repo]
  (let [git (new Git repo)
        walker (-> git .log .all .call)]
    walker))

(defn repo-name [repo]
  (.getName (.getParentFile (.getDirectory repo))))

(defn checkout-commit [repo rev-commit]
  (let [version (ClojureMetaVersion. rev-commit (repo-name repo))]
    (.openAndCheckoutIfNeeded version)))

(defn delete-commit [repo rev-commit]
  (let [version (ClojureMetaVersion. rev-commit (repo-name repo))]
    (.closeAndDeleteIfNeeded version)))
```

Figure 6.7: Clojure code to import a git repository and import an individual commit into Eclipse.

to implement the functionality that detects whether a file implements a Selenium test. This is depicted by Figure 6.8. Similar to the QwalKeko implementation, the Clojure implementation checks for an import of a Selenium package. The code uses the function `has-clj`, a Clojure re-implementation of Ekeko's `has/3`. `has-clj` is used to retrieve the import statements of a parsed compilation unit. Next, it verifies whether a single import statement includes a Selenium package. Unlike the QwalKeko version, in which the declarative engine backtracks over the different import statements, the Clojure version must perform this explicitly. Luckily Clojure features high-level list processing functions, such as `some`. This higher-order function succeeds when a given function succeeds for at least a single element of a collection. To convert a qualified name (e. g., `org.eclipse.jdt.core.dom.ASTNode`) into a list of individual parts (e. g., `(org eclipse jdt core dom ASTNode)`) we implement and use `split-qualified-name`.

We can now identify compilation units that implement a Selenium test. As a last step we need to verify for every commit whether any newly added file is a Selenium test. Figure 6.9 depicts this. Line 27 defines a function `process-repo`, which takes as input a repository, creates a corresponding walker and iterates over every commit using `map`. In contract to the QwalKeko implementation, the Clojure implementation is limited in how commits are traversed. JGit can only provide the parents of a given commit, and not the successors. For this example we just need to perform a single operation for every commit, and the order in which commits are traversed is irrelevant. As such, a walker suffices. A walker enables a user to set a start commit, to filter out some commits and then visit all the commits. Function `process-commit` processes such a single commit. It uses `get-modified-files` to retrieve the modified files of that commit. Internally this function uses JGit to

```
(defn split-qualified-name
  ([name]
   (split-qualified-name name '()))
  ([name res]
   (if (.isSimpleName name)
     (conj res name)
     (recur (has-clj :qualifier name) (conj res (has-clj :name name))))))

(defn contains-selenium-import? [cu]
  (let [imports (has-clj :imports cu)]
    (some
      (fn [import]
        (let [qualname (has-clj-unwrapped :name import)
              names (split-qualified-name qualname)
              ids (map #(has-clj-unwrapped :identifier %) names)]
          (some
            #"selenium"
            ids)))
      imports)))
```

Figure 6.8: Clojure code to detect whether a compilation unit imports a Selenium package.

compute the differences with every parent commit. It returns a list of `DiffEntry` objects, which contain, among others, the path of the file and what kind of modification was performed (i. e., the file was added, moved, deleted or modified). From this list we select all added Java files. When such a file exists, line 15 performs a checkout of the commit and call `process-file`, passing the name of the Eclipse project representing that commit as well as the local path of the file. After processing all the modified files, line 22 ensures the commit is removed as well. Finally, the results are written to a database. Note that Clojure has lazy sequences. In this case we must ensure that the results are computed before the project is removed. Thus, all elements of the collection are realized using `doall`.

6.5.4 Classification of Changes using QwalKeko

Having successfully identified the Selenium scripts in the queried software project, we now need to compute and categorize changes made to these files. To this end, whenever a change was made to a Selenium script we will use ChangeNodes to compute the differences between that revision of the file and its predecessor. The classification we use is based on code elements frequently found in Selenium test scripts. These elements are either **locators**, which retrieve elements in the DOM, **commands**, such as clicking on a button, **demarcator**, such as test specific annotations, **asserts** and **constants**.

Figure 6.10 depicts a QwalKeko query that computes and classifies changes made to Selenium scripts. On line 4 it skips an arbitrary, non-zero number of versions using the `q=>+` predicate. Next, it binds `?file` to a modified file in the current revision. It ensures this file is a Selenium script. If no Selenium scripts are modified in this revision no code is checked out. Next, it binds `?right-cu` to the

```
(defn process-file [eclipse file]
  (let [cu (get-compilation-unit eclipse file)]
    (when (contains-selenium-import? cu)
      [file (count-lines cu)])))

(project-name [repo commit]
  (str (repo-name repo) "-" (commit-id commit)))

(defn process-commit [repo walker commit]
  (let [modified-files (get-modified-files repo walker commit)
        added-files (map get-path (filter diffentry-added? modified-files))
        java-files (filter #(.endsWith % ".java") added-files)
        eclipse-name (project-name repo commit)]
    (when-not (empty? java-files)
      (checkout-commit repo commit))
    (let [results (doall (remove nil?
                           (map
                             (fn [file]
                               (process-file eclipse-name file))
                             java-files)))]
      (when-not (empty? java-files)
        (delete-commit repo commit))
      (doall
        (map (fn [[file loc]] (add-changed-file repo commit file loc)) results))
      results)))

(defn process-repo [repo]
  (let [walker (get-walker repo)]
    (doall (map #(process-commit repo walker %) (seq walker)))))
```

Figure 6.9: CLOJURE code that detects SELENIUM files in every commit of a VCS.

```
1  (qwalkeko*
2    [?left-cu ?right-cu ?file ?end ?change ?category]
3    (qwal graph version ?end []
4      (q=>+)
5      (in-git-info [curr]
6        (file|edit ?file curr)
7        (file|selenium ?file curr))
8      (in-source-code [curr]
9        (file|compilationunit ?file ?right-cu curr))
10     q<=
11     (in-source-code [curr]
12       (compilationunit|corresponding ?right-cu ?left-cu)
13       (change ?change ?left-cu ?right-cu)
14       (classify-change ?change ?category))))
```

Figure 6.10: QWALKEKO code that computes and classifies changes made to SELENIUM.

compilation unit of that SELENIUM script. On line 10 of the query it moves to one of the predecessors of the current revision using the `q<=` predicate. On line 12 it retrieves the corresponding compilation unit of `?right-cu` and binds it to `?left-cu`. Note that the way the query is written, `?left-cu` is bound to the original script, while `?right-cu` contains the more recent revision of the script. On the last 2 lines it computes the changes made to these files. The predicate `classify-change` is responsible for classifying a single change.

```
(defn change|affects-findBy [change ?find-by]
  (all
    (change|affects-node change ?find-by)
    (conde
      [(methodinvocation|by ?find-by)]
      [(annotation|findBy ?find-by)])))

(defn methodinvocation|by [?x]
  (fresh [?name]
    (ast :MethodInvocation ?x)
    (child :expression ?x ?name)
    (name|simple-string ?name "By")))

(defn annotation|findBy [?x]
  (all
    (ast :NormalAnnotation ?x)
    (annotation-name|equals ?x "FindBy")))
```

Figure 6.11: Classification of changes using QwalKeko

Figure 6.11 depicts how a change can be classified as a modification to a locator. To this end it uses `(change|affects-node change ?node)`, which unifies `?node` to any parent node of both the original and the target AST node of the change. This predicate works purely on an AST level, and does not use other sources of information. It then verifies whether one of the affected nodes resides inside either a method invocation with the name "By" or an annotation named "FindBy". These predicates are once again written using Ekeko. Predicates for the remaining categories are analogous.

6.5.5 Classification of Changes using Clojure

We now implement the same functionality in plain Clojure. Figure 6.12 implements the function `change-get-affected-nodes`, mimicking the behavior of predicate `change|affects-node/2`. This function returns a collection of all the AST nodes that are affected by a single change. We define a helper function `node-get-parent-nodes` that returns the path from a node to the root node of the AST. For each change type a specialized function is created. For example, `insert-get-affected-nodes` returns the nodes affected by an insert operation. Unlike the declarative implementation we must manually return a collection containing all the solutions. These solutions are also computed in an on-demand fashion due the Clojure's lazy collections.

Next, we need to implement the functionality to detect whether a change affects certain Selenium elements, such as `FindBy` statements. Figure 6.13 implements the function `classify-findby`, which returns whether a change affects a `FindBy` statement. The function `classify-change` returns the different categories that a single change affects. The declarative QwalKeko implementation handles this in a more intuitive manner by backtracking. Several type checks are also scattered throughout the code; the AST can return different types of AST nodes for a single property, while we are only interested in a single type. This type checking is present to a

```
(defn node-get-parent-nodes [node]
  (take-while #(not (nil? %))
    (iterate #(.getParent %) node)))

(defn insert-get-affected-nodes [insert]
  (let [original (.getOriginal insert)
        inserted (.getCopy insert)]
    (concat
      (node-get-parent-nodes original)
      (node-get-parent-nodes inserted))))

(defn move-get-affected-nodes [move]
  (let [source (.getOriginal move)
        target (.getCopy move)]
    (concat
      (node-get-parent-nodes source)
      (node-get-parent-nodes target))))

(defn delete-get-affected-nodes [delete]
  (let [removed (.getOriginal delete)
        prime-parent (.getPrimeParent delete)]
    (concat
      (node-get-parent-nodes removed)
      (node-get-parent-nodes prime-parent))))

(defn update-get-affected-nodes [update]
  (let [node (.getOriginal update)]
    (node-get-parent-nodes node)))

(defn change-get-affected-nodes [change]
  (cond
    (.isInsert change) (insert-get-affected-nodes change)
    (.isMove change) (move-get-affected-nodes change)
    (.isDelete change) (delete-get-affected-nodes change)
    (.isUpdate change) (update-get-affected-nodes change)))
```

Figure 6.12: Clojure implementation to detect what nodes are affected by a change

lesser extent in the QwalKeko implementation due to its use of `ast/3`. Unlike the Clojure implementation, the declarative engine just fails and backtracks whenever an element does not have a certain property, whereas the Clojure variant throws an exception. Thus, a more careful checking for `nil` values is required in the Clojure implementation.

The final part of the implementation must iterate over every pair of revisions, compute the changes and classify these changes using the aforementioned functions. Figure 6.14 implements the function `classify-all-changes`. This function loops over every commit, calling `process-commit-changes`. This function introduces a local function `process-commit-parent`, which processes a commit and its predecessor. To this end, it retrieves all the modified Selenium files on line 12 and 13. If such a file exists a checkout of the commit and its predecessor is performed, and the modified Selenium files are processed using `process-file`. This function takes as input a commit, its predecessor and a file path. It parses both revisions of the file, computes the changes between both revisions and classifies these changes. It returns three values; the changes, a collection of changes and their corresponding

```
(defn method-findby? [node]
  (when (ast-methodinvocation? node)
    (let [exp (ast/has-clj-unwrapped :expression node)]
      (when (ast-simplename? exp)
        (= (.getIdentifier exp) "FindBy")))))

(defn annotation-findby? [node]
  (when (ast-normalannotation? node)
    (let [name (.getIdentifier (ast/has-clj-unwrapped :typeName node))]
      (= name "FindBy"))))

(defn classify-findby [change]
  (let [affected (change-get-affected-nodes change)]
    (some (fn [node]
            (or
              (annotation-findby? node)
              (method-findby? node)))
      affected)))

(defn classify-change [change]
  (remove nil?
    (list
      (if (classify-assert change) :assert)
      (if (classify-findby change) :findby)
      (if (classify-pageobject change) :pageobject)
      (if (classify-constantupdate change) :constant)
      (if (classify-driver change) :driver)
      (if (classify-command change) :command))))
```

Figure 6.13: CLOJURE implementation to classify a single change

classification and the file location that was passed as input. These results are then passed to `add-change-result`, which writes them to a database. Finally, the opened projects are closed again.

6.5.6 Results of our Evaluation

In the following section we classify each line of the study according to one of the following concerns:

- Checking **Revision** characteristics, for example whether a compilation unit implements a SELENIUM test.
- Checking **Temporal** characteristics, for example getting the successor of a revision.
- Checking **Change** characteristics, for example whether a change affects an assert statement.
- Interacting with the **VCS**, for example checking out a specific revision or getting all the commits of the versioned project.
- Interacting with **Eclipse**, for example importing a commit into ECLIPSE or retrieving parsed code from a file.

```
(defn process-file [commit parent file]
  (let [left (get-compilation-unit (project-name parent) file)
        right (get-compilation-unit (project-name commit) file)
        changes (get-java-changes left right)
        classified (map classify-change changes)
        partitioned (remove #(empty? (second %)) (partition 2 (interleave changes classified)))]
    [changes partitioned file]))

(defn process-commit-changes [repo walker commit]
  (letfn
    [(process-commit-parent [commit prev]
       (let [modified-files (filter diffentry-modified? (get-modified-files-parent repo walker commit prev))
             seleniums (filter is-selenium-file? (map get-path modified-files))]
         (when (not (empty? seleniums))
           (checkout-commit repo commit)
           (checkout-commit repo prev))
           (let [results (map #(process-file commit prev %) seleniums)]
             (doall
               (map
                 (fn [[changes interleaved file]]
                   (add-change-result repo commit file changes interleaved))
                 results)))))]
     (let [preds (get-parents repo walker commit)]
       (doall
         (map #(process-commit-parent commit %) preds))
       (doall
         (map #(delete-commit repo %) preds))
       (delete-commit repo commit))))

(defn classify-all-changes [repo]
  (let [walker (get-walker repo)]
    (doall
      (map
        (fn [commit]
          (process-commit-changes repo walker commit))
      (seq walker)))))
```

Figure 6.14: Clojure implementation that computes and classifies changes to made Selenium files

- Interacting with the **Database**, for example to store computed results so that they can later be reused or shared.
- **Processing** the results, for example grouping all the changes affecting assert statements together.

Table 6.3 depicts the results of this classification. We have split the code of both implementations into three groups; code to perform the identification of Selenium files, code to perform the change classification and code that is shared between both steps. The table only depicts the lines of Clojure code. For the Clojure implementation we make use of a Java class that facilitates importing projects into Eclipse or to retrieve a specific commit from the git repository. This Java class contains 160 lines of code. The Clojure code provides a thin wrapper around this class. Figure 6.15 provides a different view on this data; it depicts the main concerns (i. e., all but the database and processing of results) of both implementations as a bar plot.

Table 6.3: Table depicting the lines of code spent on the different concerns for both the Clojure and QwalKeko implementation of the Selenium experiment.

	Clojure				**QwalKeko**			
Category	**Shared**	**Sel.**	**Classify**	**Total**	**Shared**	**Sel.**	**Classify**	**Total**
VCS	30	7	7	44	0	1	2	3
Eclipse	14	1	2	17	0	1	2	3
Revision	12	27	124	163	0	22	106	128
Temporal	0	4	7	11	0	1	2	3
Change	0	0	53	53	0	0	72	72
Processing	0	4	41	45	0	3	72	75
Database	9	14	8	31	9	5	13	27
Other	24	0	0	24	15	0	0	15
Total				388				326

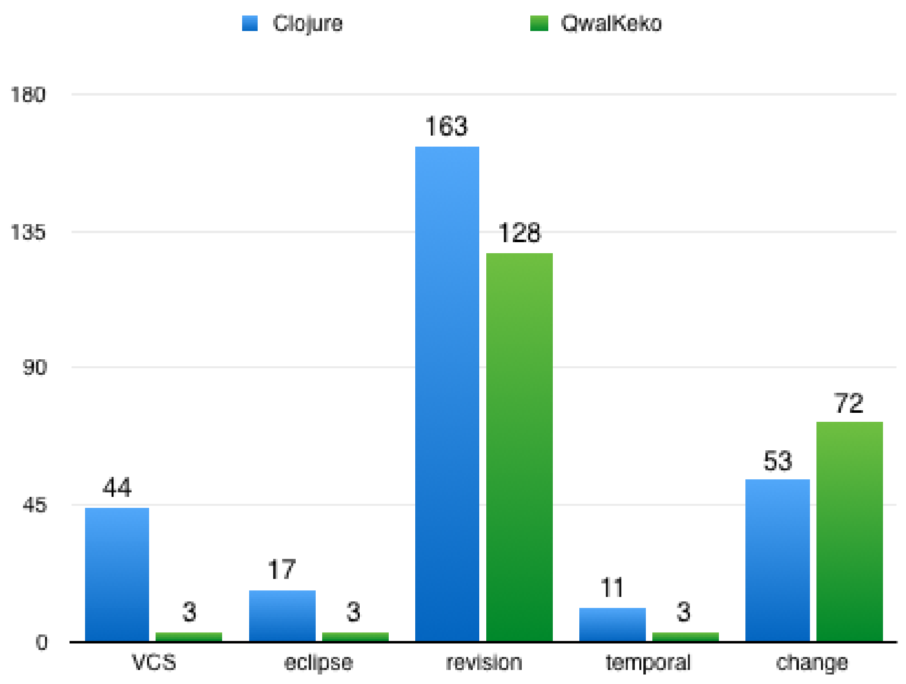


Figure 6.15: Plot depicting the different concerns in both implementations of the study.

Results for the shared code Looking at the results we notice that the Clojure implementation has a lot more shared code than the QwalKeko variant. This is because we needed to implement functionality that is provided by QwalKeko. This is mainly interacting with the VCS to retrieve a specific commit, as well as Eclipse to retrieve the parsed source code. Also note that the depicted data does not include an Eclipse class that handles the low-level commands of checking out a commit and importing it into Eclipse as a separate project. These concerns are handled by QwalKeko by the graph representation of a VCS and by using the `in-source-code` construct. The shared code in QwalKeko concerns the setup of the database and importing the different namespaces of QwalKeko.

Results for the Selenium identification For the identification of Selenium files we note that both implementations have an almost identical number of lines. Ekeko features a very explicit style of source code querying that is easily mimicked in Clojure. Thus, both implementations have a similar number of lines. The main difference is that Clojure needs to implement its own version of backtracking; its functions need to return all results which are filtered in later stages. For example, function `contains-selenium-import?` first needs to retrieve all import statements before it can filter them. This code is quite terse due to the functional nature of Clojure and its higher-order functions that operate on collections. This is also noticeable in the temporal specification in Clojure. We need to perform the same operation for every revision of the queried project. Thus, we use `map` over the collection of commits. This is similar to QwalKeko, in which we use `q=>*`.

Results for the change classification The first thing to notice is that the expression of change characteristics takes up more lines-of-code in the QwalKeko implementation. This is due to an implementation of several short predicates that detect whether a change affects a certain node type. The QwalKeko implementation introduces some logic variables that are inlined in the Clojure implementation, thus resulting in more lines-of-code. Both implementations do not require that much effort to express temporal characteristics, although the Clojure implementation requires a bit more. This is due to the nature of the change classification which, similar to the Selenium identification, requires to perform an operation on every revision pair. The Clojure implementation can only easily retrieve the predecessors of a revision, and not the successors. This limits the way the commits of the VCS can be navigated.

We can positively answer our first research question RQ1. We have used QwalKeko to perform an MSR study on software projects of industrial scale. The study requires expressing temporal characteristics, fine-grained revision characteristics and change characteristics. In Chapter 2 we have provided an overview of existing studies and the granularity of information they require. As such, this

Selenium study is representative of a whole series of studies regarding the history and evolution of a software project.

For the second research question RQ2 we conclude that QwalKeko provides a cleaner and more concise way to implement the Selenium experiment. Clojure does not feature constructs for specifying temporal characteristics. The resulting code is harder to maintain and read than the equivalent QwalKeko queries.

6.5.7 Visualizing Commit Histories

To answer SQ2 of the original study, we also visualized the commit histories of the high quality corpus that provide some insight into the commit histories of individual projects from our high-quality corpus. We present these visualizations in this dissertation as well to provide some more examples of what QwalKeko can aid in achieving. Figure 6.16 depicts a variant of the *Change History Views* introduced by Zaidman et al. [78] for three randomly selected projects. For each commit in a project's history, our variant visualizes whether a Selenium (rather than a unit test file) or application file was added, removed or edited. The X-axis depicts the different commits ordered by their timestamp. The Y-axis depicts the files of the project. To this end, we assign a unique identifier to each file. We ensure that Selenium files get the lowest identifiers by processing them first. As a result, they are located at the bottom of the graph.

Figure 6.16 clearly demonstrates that **Selenium tests are modified as the projects evolve**. However, the modifications do not appear to happen in a coupled manner. This is to be expected as Selenium tests concern the user interface of an application, while application commits also affect the application logic underlying the user interface. Any evolutionary coupling will therefore be less outspoken than, for instance, between a unit test and the application logic under test.

Note that many files are removed and added around revision 1000 of the `xwiki-enterprise` project. The corresponding commit message[7] reveals that this is due to a change in the project's directory structure. We see this occur in several other projects.

6.5.8 Results of the Change Classification

To answer SQ3 we have classified the changes made to Selenium tests into categories. These categories are created from the typical components of a Selenium-based functional test. We define the following categories:

Driver-related Expressions opening and closing a connection to the web browser; e. g., `new ChromeDriver()`, `driver.close()`

[7]Commit `74feec18b81dec12d9d9359f8fc793587b4ed329`

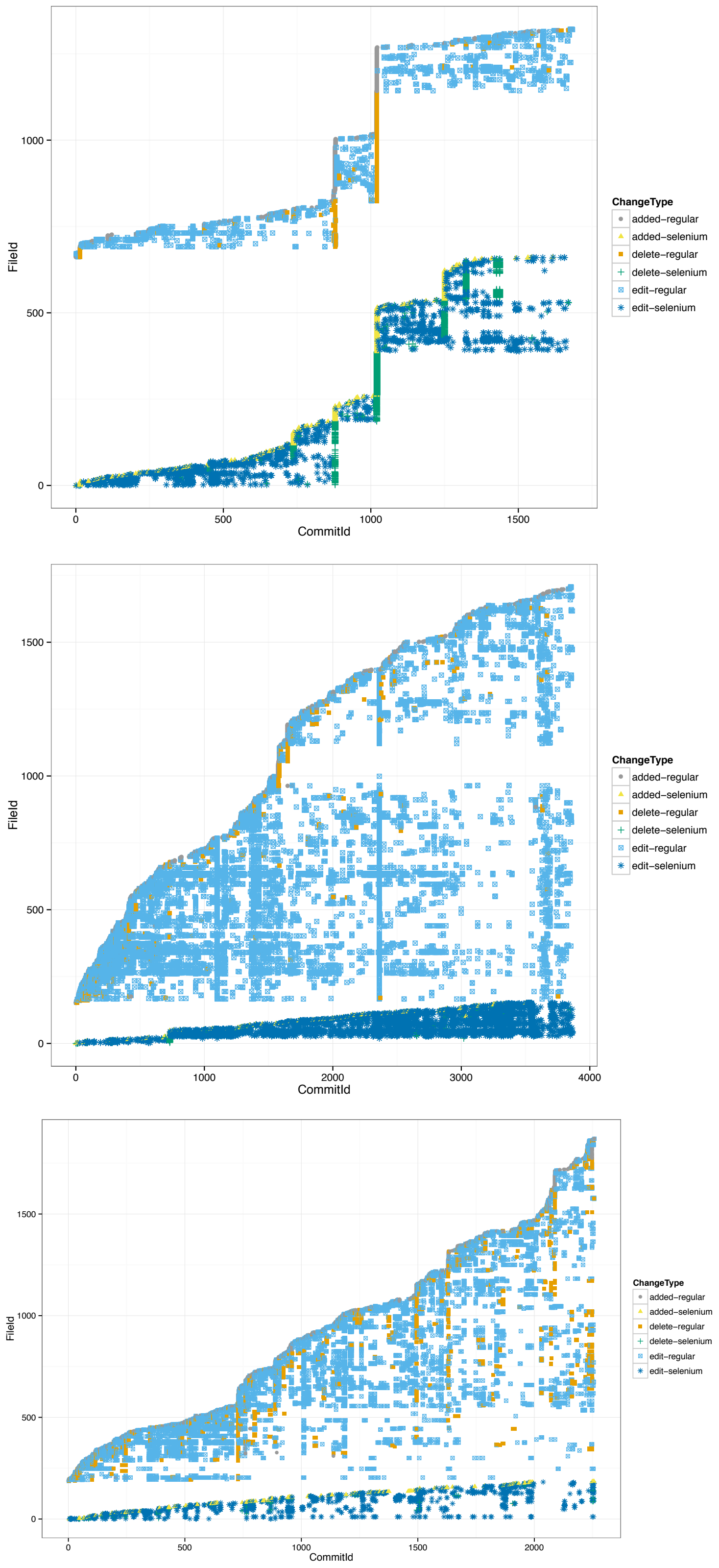


Figure 6.16: Change histories of the XWIKI-ENTERPRISE (top), OPEN-LMIS (middle), and ATLAS projects (bottom).

Locators Expressions locating specific DOM elements; e. g., `driver.findElements(By.ByName("..."))`, `webElement.findElement(By.ById("..."))`

Inspectors Expressions retrieving properties of a DOM element; e. g., `element.getAttribute("...")`, `element.isDisplayed()`

Commands Navigation requests and interface interactions; e. g., `driver.get("...")`, `navigation.back()`, `element.click()`, `element.sendKeys("...")`

Demarcators Means for demarcating actual tests and setting up or tearing down their fixtures, typically provided by a Unit Testing framework; e. g., `Test`, `BeforeClass`

Assertions Predicates over values, typically provided by a Unit Testing framework ; e. g., `assertTrue(element.isDisplayed())`, `assertEquals(element.getText(), value)`

Exception-related Means for handling exceptions that stem from interacting with a separate process. e. g., `StaleElementReferenceException`, `TimeOutException`

Constants Constants specific to web pages such as identifiers and classes of DOM element.

Figure 6.17 depicts the ratio of changes that are classified in a specific category for our high-quality corpus. The Y-axis has a box plot summarizing the number of changes that were classified in each category, divided by the total number of changes made to Selenium files. Table 6.4 lists project-scoped results.

The most frequently made changes are those to constants and asserts. These are the two test components that are most prone to changes. Constants occur frequently in locator expressions to retrieve DOM elements from a web page and in assert statements as the values compared against.[8] Focusing future tool support for test maintenance on these areas might therefore benefit test engineers most. Existing work about repairing assertions in unit tests [16], and about repairing references to GUI widgets in functional tests for desktop applications [42] suggests that this is not infeasible. Note that existing work also targets repairing changes in test command sequences [47], but such repairs do not seem to occur much in practice.

[8]Our tool classifies such changes also in the locator or assertion category.

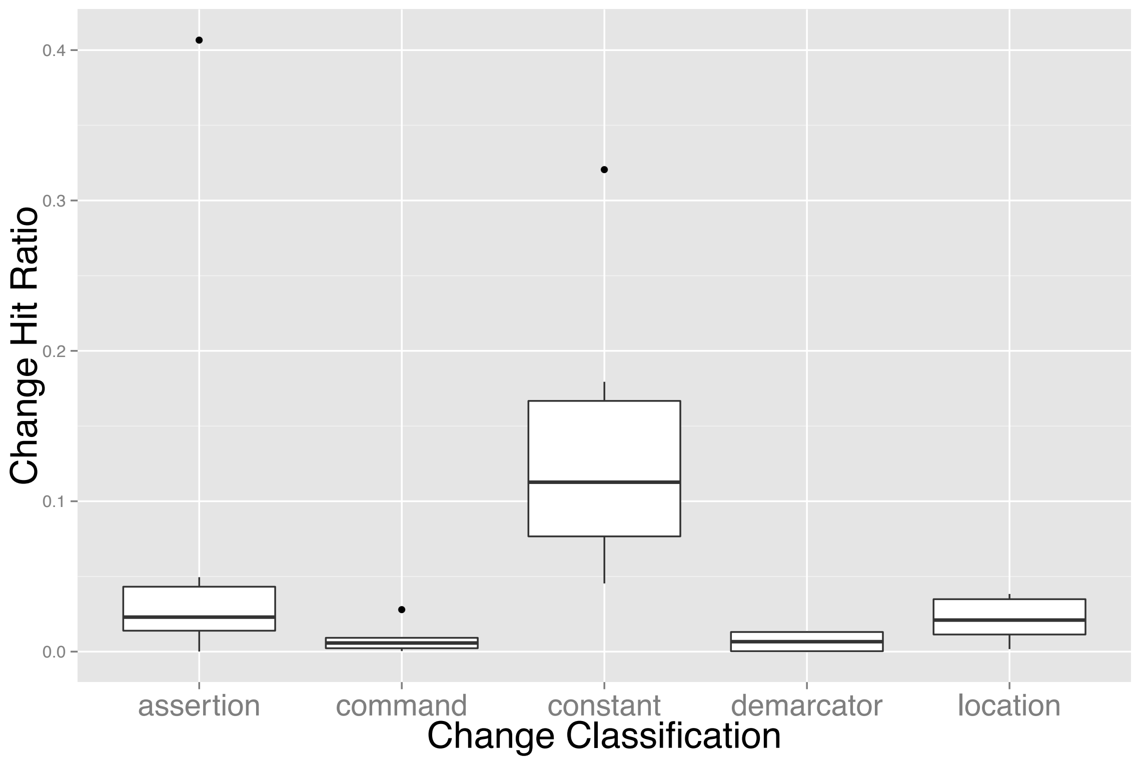


Figure 6.17: Summary of the corpus-wide change classification.

Table 6.4: Project-scoped change classification.

Project	Total	Locator	Command	Demarcator	Asserts	Constants
Atlas	8068	90	3	104	3282	2586
XWiki	68665	115	154	24	1490	3114
Tama	31821	95	89	43	36	571
Zanata	12959	497	119	0	1	906
EEG/ERP	248	3	0	0	6	24
OpenLMIS	69792	2550	401	8	3454	8972

Both outliers in our results stem from the ATLAS project. Its test scripts contain hardcoded genome strings inside assert statements that are frequently updated.

6.6 Discussion

Chapter 5 discusses which criteria of a general-purpose history querying tool QwalKeko fulfills. We concluded that support for change and evolution characteristics is missing. By integrating ChangeNodes into QwalKeko we aim to solve these shortcomings. This section discusses whether the Clojure implementation fulfills the criteria, and whether QwalKeko supports the specification of change characteristics through the integration with ChangeNodes.

C1: Revision Characteristics Both the Clojure and QwalKeko implementation support the specification of revision characteristics. QwalKeko supports these through Ekeko. We implemented similar functionality in Clojure. The major difference is that the backtracking must be implemented manually in Clojure. This is done by returning all possible solutions that are filtered later on. QwalKeko on the other hand relies on the underlying declarative engine.

C2: Temporal Characteristics QwalKeko enables users to specify temporal characteristics through Qwal. Our Clojure implementation uses JGit to query the meta-data of a git repository. git only stores the parent (or predecessor) of a commit. Thus, navigating the commits in an arbitrary order is difficult. A walker class, specifying a walk through the revision graph, can be implemented by the user. This is not sufficient for history querying, as the walk depends on the elements found in certain revisions, and cannot be known beforehand. For both the Selenium identification and classification the temporal characteristics are very simple. Thus, this shortcoming is only partially visible in our experiment.

C3–4: Change and Evolution Characteristics Both implementations illustrate that change characteristics are cumbersome to express. In this study we were only interested in classifying *individual* changes, yet we already notice that we need to implement specialized predicates or functions for every change type. Different change types are represented by different ASTs – the source AST, the target AST and the source′ AST – and thus require specialized behavior. Our current change specification language is suited to describe a single change, but does not scale to describing larger multi-change patterns. We describe the problems of querying changes and how to overcome them through a dedicated specification language in Chapter 7.

QwalKeko extended with ChangeNodes supports the specification of change characteristics. Support for specifying evolution characteristics is introduced in Chapter 7.

C5: Query abstraction, reuse and composition Both implementations enable abstracting, reusing and composing queries. QwalKeko enables users to define new predicates, introduce new temporal operators, etc. Providing a declarative interface on top of the output of ChangeNodes enables the description of change characteristics. The AST-based representation of changes enables reusing existing Ekeko predicates to describe the source code affected by a change. Clojure advocates a functional approach to write queries, in which collections are iterated, filtered, etc. Writing and composing small functions is key to such an approach. The Clojure functionality is based on the functionality provided by QwalKeko, resulting in similar query abstraction, reuse and composition.

C6: On-Demand Solutions Clojure is a functional language with lazy evaluation. Some special care is needed in the case of side-effects, such as importing or removing a revision. It is left to the user to perform this operation carefully and ensure that the query benefits as much as possible from the delayed evaluation. QwalKeko suffers from a similar problem; it relies on backtracking to provide more solutions. To this end, it keeps every revision open until a query has finished (or the query must explicitly remove a revision). Solutions are provided in an on-demand manner, but memory consumption can be high.

6.7 Conclusion

In this chapter we extended QwalKeko with a change distilling algorithm called ChangeNodes. It is based on the algorithm of Chawathe et al. [11] that is also used by other change distilling approaches such as ChangeDistiller [33]. ChangeNodes implements an algorithm to retrieve fine-grained changes between two revisions of a file. These changes do not necessarily correspond to the concrete edits a developer performed. They do form a source of information to see what elements were potentially modified during development.

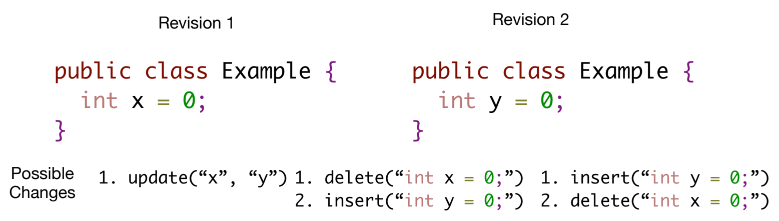
This extension is done through a declarative API that reifies these changes. This layer enables users to retrieve changes and specify these characteristics. We have evaluated the current state of QwalKeko by performing an empirical study regarding the evolution of Selenium tests. In this study we identified mature open-source projects using Selenium and detected which parts of Selenium tests are most prone to change. We implement this study twice; once using Clojure and once using QwalKeko. We discerned the different concerns of this study, and classified the code of each implementation according to these concerns. From these we conclude that QwalKeko's dedicated support results in a more concise implementation. We also note that, even though change characteristics can be expressed, that these are expressed in a cumbersome manner. In Chapter 7 we introduce support for evolution characteristics, which enables specifying the effect of multiple changes in a concise manner.

7 SUPPORTING EVOLUTION CHARACTERISTICS

Chapter 6 extended QwalKeko with support for change characteristics (C3) through the use of ChangeNodes. The chapter concluded that, although QwalKeko supports specifying *individual* change characteristics, specifying the behavior of multiple changes is cumbersome. This chapter discusses these challenges in detail and extends QwalKeko with support for evolution characteristics. Evolution characteristics concern patterns of interest in a sequence of changes. As an answer QwalKeko returns a *minimal, executable* subsequence of changes that, when applied, introduce the sought-after pattern.

Specifying such patterns in terms of single change characteristics is difficult; different change sequences can implement the same code transformation. These must be accounted for when specifying evolution characteristics in terms of individual change characteristics. In this chapter we extend QwalKeko with support for evolution characteristics in a change-agnostic manner, in which a source code transformation can be detected in multiple change sequences regardless of the concrete distilled changes. Evolution characteristics concern instances of source code transformations that are the result of applying change sequences. A source code transformation is specified as a path in a graph of intermediate states called the *Evolution Graph*. The paths are described using Qwal, and the source code characteristics of the states using Ekeko. An auxiliary graph called the *Change Dependency Graph* is constructed, in which the syntactical dependencies between changes are made explicit. The change dependency graph is used to construct the evolution graph.

We evaluate the support for evolution characteristics in QwalKeko by specifying and detecting several evolution patterns, and verifying that the returned solutions are minimal and executable. To this end, we use several data sets from other studies that detected refactorings in the history of open-source projects. We show that QwalKeko can identify multiple instances of the same refactoring using a



```
public class Example {
  int x = 0;
}
```

```
public class Example {
  int y = 0;
}
```

```
1. update("x", "y")
```

```
1. delete("int x = 0;")
2. insert("int y = 0;")
```

```
1. insert("int y = 0;")
2. delete("int x = 0;")
```

Figure 7.1: Three different change sequences that each rename the field x of revision 1 into the field y of revision 2.

single query. We show that the returned solutions to these queries are minimal and executable.

7.1 The Need for a Dedicated Support for Specifying Evolution Characteristics

Figure 7.1 depicts an example code transformation, in which a field of the class `Example` is renamed between two revisions. Three change sequences that can be returned by ChangeNodes, or any distilling algorithm, are shown at the bottom. Their differences illustrate one of the problems with querying the output of a distilling algorithm directly. It requires enumerating all possible change sequences that perform the sought-after code transformation.

When querying distilled changes directly using change characteristics, one needs to account for the fact that the algorithm distills but one of the possible change sequences that could transform the source AST into the target AST. Although the distilling algorithm strives to output a minimal change sequence, the concrete sequence that will be distilled is hard to predict. As a result, one needs to enumerate all possible change sequences in the query explicitly. Figure 7.2 depicts such an enumeration in a query for our field renaming problem.

Line 1 defines a function `rename-field`, which takes as input a sequence of distilled changes, and outputs a sequence of changes that implement the field rename. Solutions to this query consist of bindings for the logic variable `?sequence` that satisfy each condition. Line 3 features a logical disjunction `conde`, of which each disjunct corresponds to a potentially distilled change sequence. The first sequence is specified on lines 4–14, the second sequence on lines 15–28. We discuss the former lines first. They retrieve singleton sequences from `changes` that consist of a change that updates the name of a field. Line 4 introduces a number of logic variables through the `fresh` keyword. Next, variable `?update` is bound to one of the distilled changes using `member`. The logic engine will bind `?update` to the next change in the distilled change sequence `changes` upon backtracking. The sought-after sequence in this case consists of a single change, so line 6 unifies our solution `?sequence` with a singleton list. Next, line 7 ensures that the change is an update (rather than an insert or a delete). Line 8 retrieves the value that was updated and verifies that it is a `SimpleName` AST node. This name needs to reside in a `VariableDeclarationFragment`, verified through the `parent` predicate on line 10. Finally, line 14 verifies that the new value differs from the original one. Lines 15–28 describe both the second and third change sequence that could implement a field rename; an insert followed by a delete, or vice versa. Their approach is similar to the one for the first sequence. Lines 17–18 retrieve two changes `?insert` and `?delete` from the distilled change sequence `changes`, while line 19 unifies our solution `?sequence` with a list consisting of these two changes. The remaining lines restrict the bindings for these variables further through conditions on the source code affected by each change.

```
1  (defn rename-field [changes]
2    (qwalkeko* [?sequence]
3      (conde
4        [(fresh [?update ?val|source ?parent|source ?val|target]
5           (member ?update changes)
6           (== ?sequence (list ?update))
7           (change|update ?update)
8           (change|update-original ?update ?val|source)
9           (ast :SimpleName ?val|source)
10          (parent ?val|source ?parent|source)
11          (ast :VariableDeclarationFragment ?parent|source)
12          (change|update-node|newval ?update ?val|target)
13          (ast :SimpleName ?val|target)
14          (name-name|different ?val|source ?val|target))]
15       [(fresh [?insert ?delete ?insert|source’
16                 ?delete|original ?i-name ?d-name]
17          (member ?insert changes)
18          (member ?delete changes)
19          (== ?sequence (list ?insert ?delete))
20          (change|insert ?insert)
21          (change|delete ?delete)
22          (insert-node|inserted ?insert ?insert|source’)
23          (ast :VariableDeclarationFragment ?insert|source’)
24          (change|delete-node ?delete ?delete|original)
25          (ast :VariableDeclarationFragment ?delete|source)
26          (has :name ?insert|source’ ?i-name)
27          (has :name ?delete|source ?d-name)
28          (name-name|different ?i-name ?d-name))]))))
```

Figure 7.2: Querying a distilled change sequence for a field rename.

Reasoning about a sequence of changes in this manner gives rise to the following problems:

- Different change sequences can implement the same transformation of the source code. This was illustrated by Figure 7.1. The corresponding problem is two-fold. On the one hand, there is no straightforward way to know beforehand what change sequence will be output by a distilling algorithm. Distilling algorithms make use of heuristics to determine whether nodes from the source AST are sufficiently equal to nodes from the target AST. As a result, very different change sequences can be distilled for similar modifications to similar files. On the other hand, it is not practical for a user to enumerate all the change sequences that could possibly be distilled. This was illustrated by the large disjunction in Figure 7.2. We call this problem the *Change Equivalence Problem*.

- Change distilling algorithms immediately apply each change as it is distilled. In order not to modify the original source code, they make a copy. As such, a change refers to potentially three different ASTs; the original source code, the target source code, and a copy of the original source code (denoted source′) that will look identical to the target source code after the execution of the distilling algorithm. For example, a delete can only be represented using nodes from the source AST, as the node is not present in the target AST (otherwise it would not have been removed), nor is it present in source′ as the delete has already been applied. An insert on the other hand only has nodes that are present in source′ and target, but not in source (otherwise it would not have been inserted). As a user, it is cumbersome to switch between the different ASTs to describe the surrounding source code affected by a change. Lines 7, 8, 12, 21 and 23 of Figure 7.2 are indicative of this problem. The nodes affected by individual changes reside in one of three different ASTs. The names for variables such as `?insert|source'` hint at the AST in which its binding resides. Users need to be aware of the AST in which a node resides, as comparing nodes from different ASTs for equality produces incorrect results.[1] We call this problem the *Change Representation Problem*.

7.2 Change Characteristics Compared to Evolution Characteristics

Chapter 6 extended QwalKeko with support for change characteristics. This support enables specifying the characteristics of *individual* changes residing in a distilled change sequence. Section 6.5 used this support to classify changes made to Selenium tests. Our support for evolution characteristics enables specifying a sought-after code transformation, and as a solution returns a *minimal*, *executable* subset of changes stemming from a *single* change sequence. The returned subset contains the changes that implement this code transformation. The original returned change sequence may contain other changes that do not contribute to the code transformation. Executable means that the solution can be applied to the original source code, and that the resulting source code contains the effect of the sought-after code transformation. Minimal means that the solution only consists of three groups of changes. First, *evolution implementing* changes are changes

[1]This motivated our use of `name-name|different`, which copes with nodes from different ASTs, while unification cannot.

that perform the code transformation. Second, *evolution supporting* changes are changes that are depended on by one of the evolution implementing changes. Finally, *evolution linking* changes are neither evolution implementing or evolution supporting, but are depended on by one of the non-solution changes, and applying the evolution implementing or supporting changes would render those non-solution changes inexecutable. Section 7.5.3 further illustrates these three groups of changes.

The support for evolution characteristics forms the basis for future research. We envision using this support for different applications, such as commit untangling, porting features from one branch to a different branch or automating systematic edits. For example, commit untangling is the process of splitting a single tangled commit into multiple atomic commits. An atomic commit is a commit that only contains changes resulting from a single development activity. Provided that the different development activities of a tangled commit are identified, a user could write a history query that extracts a minimal, executable change sequence for an activity. By applying this change sequence a new atomic commit would be created. A minimal sequence is needed, if not the resulting commit would contain changes that are part of another development activity. An executable sequence is needed to create a new atomic commit containing only the extracted activity.

Specifying a sought-after code transformation and retrieving a minimal, executable solution in terms of change characteristics is difficult. A *minimal, inexecutable* solution could be retrieved by specifying the changes implementing a sought-after transformation. Such specification must enumerate all possible change sequences that implement the sought-after code transformation due to the change equivalence problem (cf., Section 7.1). Extracting an *executable* solution may prove to be difficult. Change distillers return a sequence that must be applied *in order*. Every change potentially depends on any preceding change. Retrieving a *executable* subsequence of changes by expressing a code transformation in terms of change characteristics requires manually verifying these dependencies. The support for evolution characteristics renders these dependencies implicit. They are navigated using change operators. These dependencies and the corresponding change navigaton operators are detailed in Section 7.4.

7.3 Supporting Evolution Characteristics

We propose support for evolution characteristics to specify and detect code transformations in distilled sequences of changes. Unique to this support is that it enables specifying a code transformation in terms of source code characteristics; those that a sequence of intermediate AST states could evolve through were the distilled changes replayed. This shields users from the problems that specifying code transformations in terms of distilled changes gives rise to.

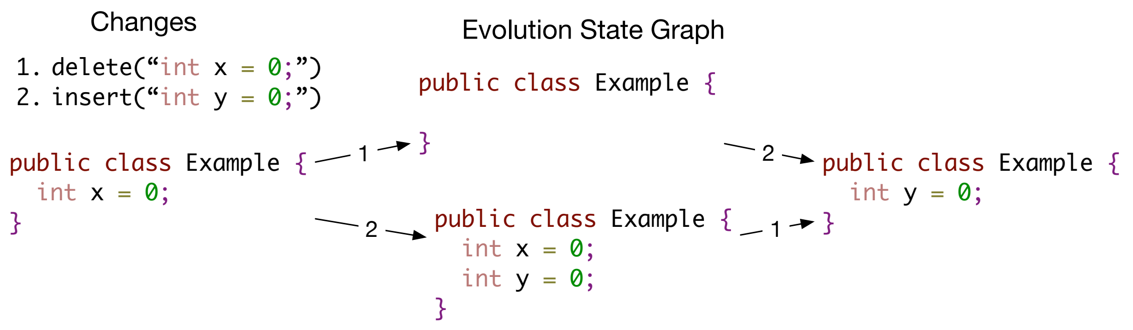


Figure 7.3: Evolution state graph (ESG) for a sequence of distilled changes. Edge labels correspond to applied changes.

For specifying said code transformation, we provide specialized QWAL predicates. Whereas we previously used QWAL to match paths through a graph of revisions (cf., Chapter 5), we now match against a sequence of nodes in a so-called Evolution State Graph (ESG) constructed from a distilled change sequence. Figure 7.3 depicts the ESG for the distilled sequence in the middle of Figure 7.1. The nodes of the ESG, called Evolution States (ES), contain an AST state and the specific changes that transformed the source AST into this state.

Figure 7.4 depicts an overview our approach to supporting evolution characteristics in a history querying language. There are two revisions of the same file, called `Rev1` and `Rev2`. The goal is to detect instances of a user-specified pattern in the differences between them. The distiller's output is a sequence of changes needed to transform the AST of the source file into the AST of the target file. We convert this sequence into an auxiliary Change Dependency Graph (CDG) that makes the dependencies among individual changes explicit. For instance, the CDG encodes the fact that an AST node cannot be inserted by a change operation if its parent node has not been inserted by a preceding change operation. Section 7.4.1 details all change dependencies. Next, the Evolution State Graph (ESG) is constructed. The process starts from a single Evolution State (ES) node containing the original AST of the source file. Other nodes are created in an on-demand fashion; when the user-specified evolution characteristics require navigating to an unexplored region in the graph. This requires determining which changes from the distilled sequence remain and are applicable dependency-wise. To this end, the ESG consults the CDG —a process that is detailed in Section 7.4.2. The solution to such a query is an ES, containing an *executable* script of changes (i. e., the changes can be applied to the original source code) that implements the evolution pattern specified by the user. This script consists of the minimal number of changes needed to implement the source code transformation, and any changes that would no longer be executable if the rest of the solution were executed. As such, a solution ensures that the remainder of the change sequence remains executable as well.

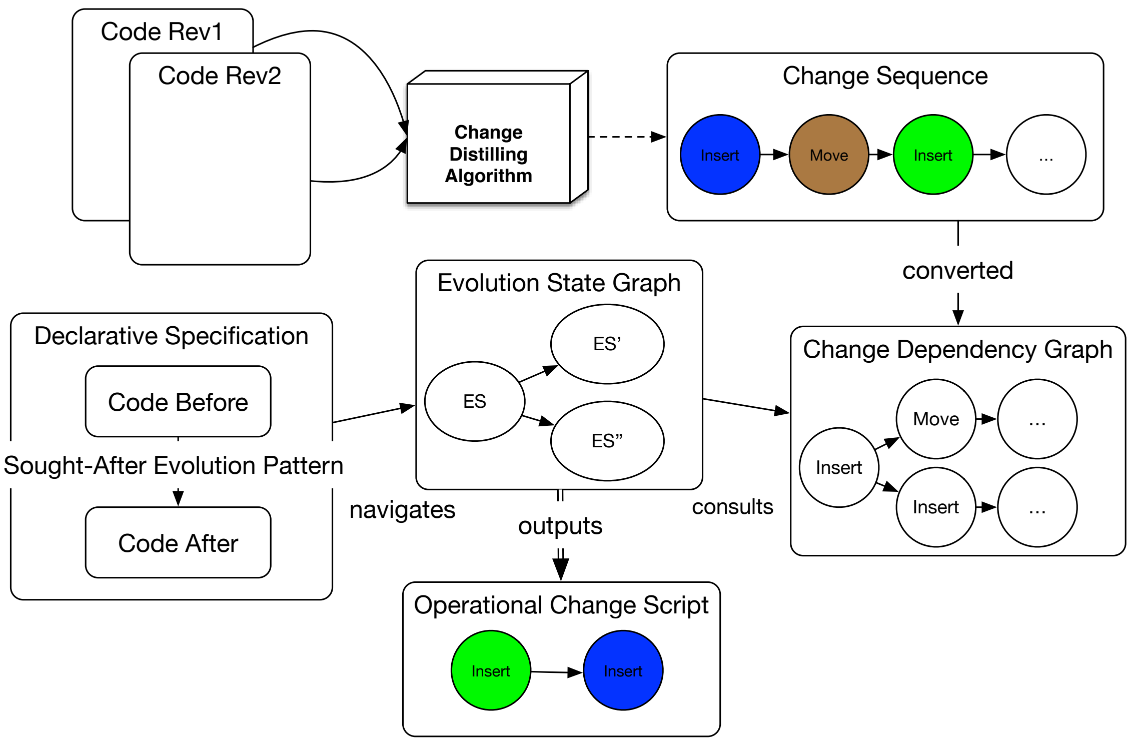


Figure 7.4: Graphical overview of the approach.

7.3.1 Motivating Example Revisited: Querying the ESG

Figure 7.5 illustrates the specification language of our approach for supporting evolution characteristics. Depicted is a logic query that finds instances of a field rename by navigating the ESG. To this end, line 3 launches a QWAL expression through the `query-changes` construct. Internally `query-changes` calls the `qwal` construct, with the starting node bound to the root ES. It takes as input an ESG and unifies its second argument with the end state of a matching path. Its third argument is a collection of logic variables, made available to the remainder of its arguments. These comprise a sequence of instructions that either verify that the current ES adheres to the given logic conditions, or navigate to another ES in the ESG. Lines 4–6 describe the initial state using `in-current-es`, which introduces two variables `es` and `ast`. The first is bound to the current ES of the query, the latter is bound to the AST of that ES.[2] Lines 5–6 describe the source code of that AST, in which a `?field` needs to be present at some depth. Next, line 7 applies an arbitrary, non-zero, amount of changes using the regular path expression operator `change->+`. This will change the current ES for the remainder of the expression. Finally, lines 8–12 state that the current ES needs to have a newly field `?renamed`. To this end, lines

[2]These are variables only visible in the body of `in-current-es`. If these variables need to be available in other parts of the query a user needs to explicitly bind them to a logic variable.

11–12 ensure that ?field is not present in the current AST, and that ?renamed is not present in the original AST. This is done based on the name of the field using the predicate ast-field|absent.

Figure 7.2 depicted an equivalent logic query that quantifies directly over a sequence of distilled changes without support for evolution characteristics. There, the user specifying the query was burdened with describing the characteristics of the sought-after changes and the code that they affect. The query with support for evolution characteristics, in contrast, merely required the user to describe source code characteristics. The changes resulting in this source code are returned as part of the query's result. Users can therefore abstract away from the concrete changes that were distilled. This solves both of the aforementioned problems. First, by having users describe ASTs instead of changes, users are shielded from the change equivalence problem (cf., Section 7.1). Second, it is always clear in which AST a node resides, shielding the user from the change representation problem (cf., Section 7.1). Where necessary, auxiliaries are provided to retrieve the corresponding node in a different intermediate AST.

```
 1 (defn field-rename [esg]
 2   (qwalkeko* [?es]
 3     (query-changes esg ?es [?orig-ast ?field]
 4       (in-current-es [es ast]
 5         (== ?orig-ast ast)
 6         (ast-field ast ?field))
 7       change->+
 8       (in-current-es [es ast]
 9         (fresh [?renamed ?new-name]
10           (ast-field ast ?renamed)
11           (ast-field|absent ast ?field)
12           (ast-field|absent ?orig-ast ?renamed)))))
```

Figure 7.5: Querying an ESG for changes renaming a field.

7.3.2 Example Applications and the Corresponding Queries

We illustrate the advantages of supporting evolution queries in history queries through two example applications. In the first example, we are tasked with determining whether and which changes are responsible for introducing a new method in between two revisions. In the second, more complex example, we detect which changes from a distilled change sequence are responsible for eliminating a code clone.

Introduction of a Method

We first consider the problem of identifying the changes in a change sequence that are responsible for introducing a new method in between two revisions of a file. At first sight, it might suffice to query the change sequence for a single insert operation that added a MethodDeclaration. Inadvertently, however, a change sequence will be encountered in which the name of an existing MethodDeclaration has been changed by an update or a move operation. Before long, the query will have evolved into a large enumeration of potential change operations with a similar effect. Operations that change the signature of the method, for instance, might also have to be accounted for.

Instead, it is much easier to find an intermediate AST in which a new method is present, and retrieve the changes that led to the creation of this AST. Figure 7.6 depicts a function that launches such an evolution query. The function takes as input an ESG for a particular change sequence, and returns pairs of a method that has been introduced and the corresponding evolution state. The function launches a query on line 2 that will find solutions for a pair of variables ?method and ?node. Lines 3–9 describe a path through the ESG that ends in an evolution state ?es. Lines 3–5 describe the initial state on this path, for which a logic variable ?absent is introduced. Line 5 binds this variable to the source AST, as so far no changes have been executed on the path. Line 4 introduces two new variables es and ast, bound to the current ES and its corresponding AST, using special form in-current-es. Line 6 executes an arbitrary, non-zero amount of changes using change->+. Lines 7–9 verify that a new method is added to the current ES. Line 8 binds ?method to any method declaration in the current ES, and verifies that that method was not present in the original AST using ast-method|absent. The query returns all different ES that exhibit these characteristics.

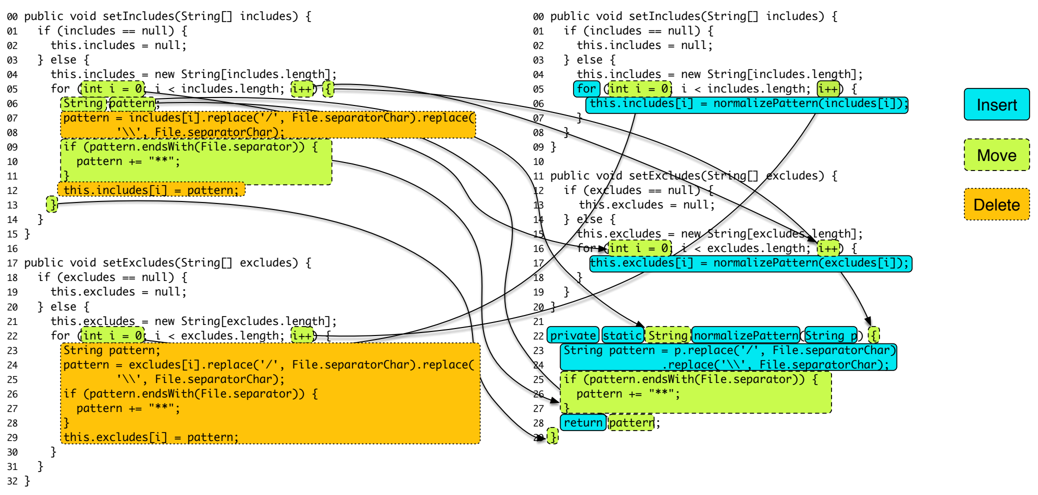
```
1 (defn introduced-method [esg]
2   (qwalkeko* [?method ?es]
3     (query-changes esg ?es [?absent]
4       (in-current-es [es ast]
5         (== ?absent ast))
6       change->+
7       (in-current-es [es ast]
8         (ast-method ast ?method)
9         (ast-method|absent ?absent ?method)))))
```

Figure 7.6: Querying an ESG for changes introducing a method.

Code Clone Elimination

For the final example application, we are tasked with finding the changes in between two revisions that resulted in the removal of a code clone. We will look for evidence of a removal technique involving, but not limited to, the *extract method* refactoring: the cloned code is extracted to a new method, and each clone instance is replaced by a method invocation to the newly introduced method. A concrete example of such a clone removal exists in the APACHE ANT[3] project. Commit

[3] https://ant.apache.org/



```
public void setIncludes(String[] includes) {
  if (includes == null) {
    this.includes = null;
  } else {
    this.includes = new String[includes.length];
    for (int i = 0; i < includes.length; i++) {
      String pattern;
      pattern = includes[i].replace('/', File.separatorChar).replace(
              '\\', File.separatorChar);
      if (pattern.endsWith(File.separator)) {
        pattern += "**";
      }
      this.includes[i] = pattern;
    }
  }
}

public void setExcludes(String[] excludes) {
  if (excludes == null) {
    this.excludes = null;
  } else {
    this.excludes = new String[excludes.length];
    for (int i = 0; i < excludes.length; i++) {
      String pattern;
      pattern = excludes[i].replace('/', File.separatorChar).replace(
              '\\', File.separatorChar);
      if (pattern.endsWith(File.separator)) {
        pattern += "**";
      }
      this.excludes[i] = pattern;
    }
  }
}
```

```
public void setIncludes(String[] includes) {
  if (includes == null) {
    this.includes = null;
  } else {
    this.includes = new String[includes.length];
    for (int i = 0; i < includes.length; i++) {
      this.includes[i] = normalizePattern(includes[i]);
    }
  }
}

public void setExcludes(String[] excludes) {
  if (excludes == null) {
     this.excludes = null;
  } else {
    this.excludes = new String[excludes.length];
    for (int i = 0; i < excludes.length; i++) {
      this.excludes[i] = normalizePattern(excludes[i]);
    }
  }
}

private static String normalizePattern(String p) {
  String pattern = p.replace('/', File.separatorChar)
                   .replace('\\', File.separatorChar);
  if (pattern.endsWith(File.separator)) {
    pattern += "**";
  }
  return pattern;
}
```

Figure 7.7: Two revisions of a class from the ANT project in between which a code clone is extracted into a separate method `normalizePattern`, to which two invocations are added. Overlaid is the output of our ChangeNodes change distilling algorithm.

`6bdc259...` removes a code clone from file `DirectoryScanner.java`. Figure 7.7 depicts the changes distilled from this commit. We only show a small snippet of the original source file, which is slightly over 1500 lines of code. We assume that the code clone has already been detected using an existing tool such as CCFinder [48], and that the ESG has been created. We are only tasked with finding the specific changes that implemented the clone removal.

The first line of Figure 7.8 defines a function that takes as input an ESG, the names of two methods containing cloned code, and the extracted method AST node. The body of the function launches a logic query on line 2 returning a collection of all possible bindings for `?end`, which is the end node of a path throughout the ESG. Line 3 describes the shape of this path through a regular path expression. Line 4 introduces the logic variables that are available to the path expression. Lines 5–9 bind `cloneA` and `cloneB` to the clones detected in the source AST (i.e., `setIncludes` and `setExcludes` in the *left revision* in Figure 7.7). the `child+` construct requires the binding for each variable to stem from the AST of the first node of the ESG. Line 10 navigates to a different node of the ESG by applying an arbitrary, non-zero amount of changes. Lines 11–19 specify a strict implementation of the extract method refactoring. Lines 12–13 require the presence of a method `?extracted` in the current ES that is identical to the method given as the `extracted` parameter to the function (i.e., `normalizePattern` in the *right revision* in Figure 7.7). This ensures that an ES node of the ESG has been reached in which all the changes extracting the cloned code have been applied. If not, line 10 will be backtracked to and another change will be applied. Next, lines 14–15 retrieve the version in that ES of the methods in which the two instances of the cloned code resided originally (i.e., `setIncludes` and `setExcludes` in the *right revision* in Figure 7.7). They use the auxiliary construct `ast-method-method|corresponding` to this end, which returns the corresponding method from an ast for a given method. Finally, lines 16–19 ensure that the methods previously containing cloned code now contain a method invocation invoking the method to which the clone was extracted (i.e., the invocations on line 6 and on line 17 in the *right revision* in Figure 7.7).

```
 1 (defn clone-elimination [esg nameA nameB extracted]
 2   (qwalkeko* [?end]
 3     (query-changes esg ?end
 4       [?cloneA ?cloneB ?extracted ?aInvoc ?bInvoc ?aCurr ?bCurr]
 5       (in-current-es [es ast]
 6         (child+ ast ?cloneA)
 7         (child+ ast ?cloneB)
 8         (method-string|named ?cloneA nameA)
 9         (method-string|named ?cloneB nameB))
10       change->+
11       (in-current-es [es ast]
12         (ast :MethodDeclaration ?extracted)
13         (ast-ast|same ?extracted extracted)
14         (ast-method-method|corresponding ast ?cloneA ?aCurr)
15         (ast-method-method|corresponding ast ?cloneB ?bCurr)
16         (child+ ?aCurr ?aInvoc)
17         (child+ ?bCurr ?bInvoc)
18         (method-invocation|invokes ?extracted ?aInvoc)
19         (method-invocation|invokes ?extracted ?bInvoc)))))))
```

Figure 7.8: Querying an ESG for extract method refactorings.

7.4 Conceptual Implementation

Having presented the necessary background on changes and change distilling, we are ready to present the support for evolution characteristics in a more detailed manner. We target the problem of identifying executable subsequences in a distilled change sequence that implement an evolution pattern of interest. QwalKeko recalls different subsequences that implement the same evolution pattern, specified as paths through a graph of intermediate AST states. This spares users the “Change Equivalence" and “Change Representation" problems identified in Section 7.1.

Section 7.4.2 will detail the Evolution State Graph (ESG) against which our approach evaluates evolution queries. We provide specialized Qwal change predicates to navigate this graph. Table 7.1 shows an overview of the different available predicates. We provide operators for navigating an ESG, such as `change->`, which moves evaluation to a successor of the current node in the ESG. We also provide operators such as `in-current-es` for evaluating logic conditions against the current node of the ESG. Such embedded conditions comprise the primary means for describing the source code characteristics that need to hold along a path of the ESG.

Table 7.1: Language for specifying evolution patterns through ESG-navigating regular path expressions.

Navigation through the ESG	
`change->`	`change->` is an operator that moves the current state to the next one by applying one of the applicable changes.
`change->?`	`change->?` is an operator that either stays in the current state or that moves to the next one by applying one of the applicable changes.
`change->*`	`change->*` is an operator that changes the current state by applying an arbitrary, including zero, number of changes.
`change->+`	`change->+` is similar to `change->*`, except that at least one change will be applied.
`change==>`	`change==>` is an operator that moves the current state to a successive one by applying one of the applicable changes and all of its dependent changes.
`change==>*`	`change==>` is an operator that changes the current state by applying an arbitrary, including zero, number of changes and their dependent changes.
Characteristics of an ES	
`(in-current-es [es ast] & conditions)`	`in-current-es` binds `es` to the current ES of the query, and `ast` to the intermediate AST of that state. It verifies whether the logic conditions `conditions` hold in this intermediate state. These conditions can be any Ekeko predicate.
Launching a Query	
`(query-changes esg ?end [&vars] & conditions)`	`query-changes` launches a path query over `esg` and binds `?end` to the end node of that query. Logic variables `vars` are introduced and available in the scope of the path query. `conditions` is a sequence of the aforementioned operators that are proven for the given ESG.

7.4.1 Construction of a Change Dependency Graph

Section 7.4.2 will detail an algorithm for constructing the Evolution State Graph (ESG) against which our approach evaluates a query. The algorithm relies on a model of the order dependencies among the changes in a distilled change sequence. Even though such a sequence is by definition ordered (i.e., the distiller guarantees the sequence transforms the source AST into the target AST when the changes are executed in order), additional order dependencies are required to identify (possibly non-contiguous) *sub*sequences that implement a pattern of interest. Individual changes in such a subsequence, can depend on any change in the distilled sequence.

A dependency $A \rightarrow B$ between changes A and B denotes that in order to execute change B, one needs to execute change A first. We gather all dependencies among the changes in a change sequence in a Change Dependency Graph (CDG), of which the nodes correspond to changes and the directed edges to dependencies. Recall the definition of the different changes discussed in Section 6.2.2. We compute the following kinds of dependencies:

Parent Dependency

There is a parent dependency $A \rightarrow_p B$ between changes A and B if the subject of B is introduced by the application of A. Nodes can be introduced either by an insert or by a move operation. We determine this dependency by checking whether the subject of change B is part of the subtree created by the application of change A. To this end, we verify whether `parent'` of change B is part of the minimal representation of `node'` of change A. The left-most situation in Figure 7.9 illustrates a parent dependency between a change introducing node α and a change introducing node β.

Move Dependency

There is a move dependency $A \rightarrow_m B$ between a move A and a change B if B removes part of the code-to-be-moved of A, rendering it impossible to move its node. This can happen either by a delete or by an insert that overwrites the part of the AST in which the node-to-be-moved resides. To detect this dependency, we check whether the `preparent'` (the parent′ of node′ before applying the move) of A is part of the AST of the `removed'` (the node′ that is overwritten by applying the insert) of an insert operation or `node'` of a delete operation. Note that without `preparent'` or `removed'` it would be impossible to detect whether the node of a move was overridden by an insert, as both the move and the insert have already been applied in `source'`, and as such the original location would be lost. An example move dependency is depicted in the middle in Figure 7.9, where node α is moved, while node β is inserted at the original location of α.

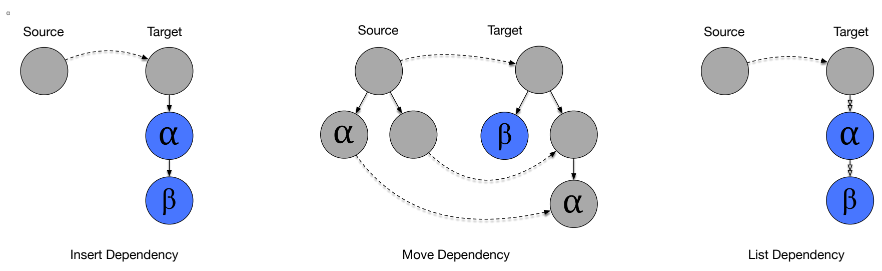


Figure 7.9: Different kinds of change dependencies illustrated among the changes distilled between a source and target AST. Dotted lines connect matching AST nodes. Single-arrowed lines connect an AST node with a child AST node. Double-arrowed lines connect an AST node with an element of one of its list-valued properties. Blue nodes are added to the target AST.

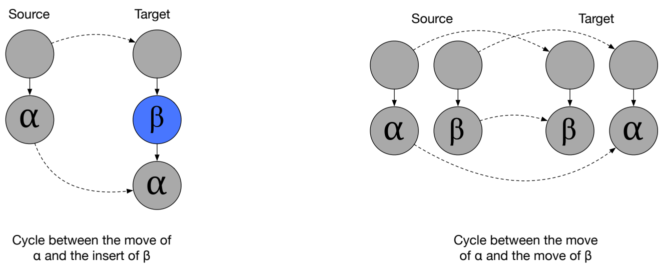

Source
Target
α
β
α
Cycle between the move of α and the insert of β
Source
Target
α
β
β
α
Cycle between the move of α and the move of β

Figure 7.10: Different combinations of dependencies can induce cycles. The example of the left depicts a cycle between the insert of *beta* an the move of *alpha*. The example on the right depicts a cycle between two moves.

List Dependency

There is a list dependency $A \rightarrow_l B$ between changes A and B if they operate on elements of the same list, but the element of B has a lower index than the element of A. Although changes A and B can be applied independently of each other, the index of A will change depending on whether B has already been applied. For example, change A inserts a new method at index 3 in a type declaration, after which change B inserts a new method at index 4 of the same type declaration. An example of such a dependency is depicted on the right in Figure 7.9.

The subsequent ESG construction algorithm will require the CDG to be acyclic. Particular combinations of the above dependencies can, however, induce cycles in the graph. Figure 7.10 depicts dependencies that induce cycles. The left example depicts a move dependency and a parent dependency, which form a cycle. Node β is inserted on the same location where node α is residing. Node α is moved as a child of node β. As such, the move of node α can only be applied after the insertion of node β, while inserting node β overwrites node α. To solve this, our CDG construction algorithm detects such cycles and replaces the move operation of α by an insert operation.

Cycles can also occur across moves. For example, two moves that swap the location of two nodes α and β result in the first move overwriting the node of the other move. This can also occur across multiple moves, in which move A overwrites move B, move B overwrites move C and move C overwrites move A. These cycles can also be removed by replacing one of the moves with an insert operation. The right example in Figure 7.10 depicts a cycle between two moves.

7.4.2 On-demand Construction of the Evolution State Graph

We now explain how the Change Dependency Graph (CDG) from the previous section enables constructing a Evolution State Graph (ESG) that is navigated through by a regular path expression. The ESG represents the possible ASTs that can be constructed by applying subsets of the distilled changes in different orderings. A single ESG node wraps a syntactically legal AST and an ordered sequence of changes that have been applied to construct that AST. Two ESG nodes are connected if there exists an unapplied change that transforms the AST of one into the AST of the other. The resulting edge is labeled by the applied change. A single change can appear on multiple edges in the graph, but only once along a single path.

The ESG has one source node (i.e., the node containing the original source code with no applied changes) and one sink node (i.e., the node containing the target source code and no unapplied changes). The graph is constructed using the information provided by the CDG. Initially, the source node is constructed from the source AST. Successors of the source node are constructed by applying a change without dependencies. The CDG facilitates the retrieval of applicable changes given a set of applied changes. It also enables computing the correct index for changes modifying a list, where the index changes depending on whether list dependencies have already been applied. The ESG is constructed on-demand; nodes and their ASTs are only created as needed.

7.4.3 Minimizing Solutions to an Evolution Query

A solution is computed by navigating the ESG until an ES is encountered of which the source satisfies the declarative specification provided by the user. This ES, together with bindings for all the logical variables, is returned as a solution to the user. ES nodes returned in this manner are not necessarily a minimal solution, i.e., an ES node that wraps the smallest number of applied changes. Multiple ES may exist that satisfy the declarative specification, each with a different number of applied changes.

We define a minimal solution as the smallest operational script of changes that implements the specified evolution pattern.

There are different ways to compute such a minimal solution:

Brute-force Path Exploration A brute-force path exploration approach considers all the different paths in the ESG. To this end, we can simply rely on the declarative proving mechanism to compute all the solutions for the query for that ESG. A minimal solution can easily be retrieved from all those solutions. The major drawback from this approach is the performance. It requires considering all the different possible paths in the ESG. A single ES can be reached along different paths (i. e., permutations of the same subsequence), while these different paths will not yield new solutions in *future* ES. Thus, this brute-force approach will revisit the same ES multiple times, even when it has already shown that further exploring that ES will not yield any results.

Brute-force Subset Generation Another brute-force approach is to generate all the applicable subsets of changes. A change can be included in a subset if all of its dependencies are also in that subset. This can be done by consulting the CDG. An ESG′ can be constructed for a single subset by topologically sorting the changes. Determining whether such an ESG′ exhibits the specified history characteristics can be done more efficiently as there is only a single path through the ESG′. A minimal solution can be found by ordering the subsets by their size, and considering the smaller sets first. The advantage of this approach is that only a single path leading to a specific ES is considered, improving the performance. The disadvantage is that a different ordering of the same set of changes result in different ES (but the same final ES). Thus, potential solutions are missed.

Minimizing a Solution A final approach is to compute a single non-minimal solution, and to minimize its changes to a minimal solution. A solution ES contains a single change sequence that realizes the sought-after evolution pattern. This change sequence can be transformed into an ESG′, in which every ES only has a single successor, constructed by applying the next change in the solution sequence. Determining whether such an ESG′ exhibits the specified history characteristics can be done more efficiently as there is only a single path through the ESG′. In order to minimize a given solution we verify whether the removal of a single change and all its dependents still results in an ESG′ exhibiting the evolution pattern. If so, that change is not part of the minimal solution. Otherwise, that change must have been part of the solution, and must be kept. We can do this for every change in a non-minimal solution to transform it into a minimal one. While this approach turns a solution into a minimal one, the *worst case* performance of finding a single solution still requires enumerating all the different paths through the ESG. QwalKeko provides more coarse-grained navigation predicates that apply a series of changes to quickly find a single solution (e. g., `change==>` applies a change and all of its dependent changes), at the risk of missing an ES containing the solution. Finally, note that when the sought-after evolution pattern is present in the target AST, a starting solution can be constructed by simply executing all distilled changes.

We have opted for the last approach as the sought-after source code transformation is frequently present in the final ES. Finding a solution can be done by executing all the changes, and this solution can in turn be minimized. Section 8.3 discusses detecting patterns that are only present in intermediate ES.

7.5 Evaluation: Extracting Executable Transformations from Distilled Code Changes

The examples in Section 7.3.2 served to demonstrate the expressiveness of the specification language of our approach. We now seek to answer the following research questions:

RQ1 Can a *single* query identify instances of the same evolution pattern in *different* change sequences?

RQ2 Is a *minimal* and *executable* change script returned, and can the *remaining distilled changes* still be executed after the change script has been executed?

RQ3 How does support for evolution characteristics compare to directly querying the output of a distilled change sequence with respect to solution size, precision and the number of changes that need be executed?

To answer these questions, we will use a data set of commits from open-source repositories that each contain —among many other changes— one of three well-known refactorings (Section 7.5.1). We aim to extract the exact changes contributing to each refactoring among all of the changes distilled for each commit.

For each refactoring, we will attempt to specify the state of the source code before and after the refactoring by means of a history query (Section 7.5.2). Each solution to these queries is an executable script of changes. When executed, this script will transform the source code from *before the commit* to a state that matches the specified state of the code *after the refactoring*. In other words, the extracted changes will perform the specified refactoring. The remaining changes distilled for the commit will, when executed, in turn transform the state of the code *after the refactoring* to the state of the code *after the commit*.

QwalKeko computes a minimal solution — that is the smallest subset of changes that implements the code evolution. To this end, it retrieves the ES that exhibits the evolution characteristics with the smallest amount of applied changes. We manually verify the solutions depicted in Table 7.3 on their minimality. We also compute various metrics pertaining to each research question (cf., Section 7.5.3).

7.5.1 Data Set of Commits Containing Refactorings

Our evaluation proceeds on a data set of commits that each contain, among other changes, a *"Replace Magic Constant"*, *"Remove Unused Method"* or *"Rename Field"* refactoring. This random selection of refactorings is sufficiently varied in the number of changes required to perform them, as well as in the types of AST nodes affected by them. Tables 7.2 and 7.3 list the identifier of each commit, the open source project repository it originates from, the name of the refactoring it contains, the name of the class affected by the refactoring, and the oracle according to which the commit contains the refactoring. The oracle is indicated by the subscript in the first column. We have used two such oracles:

- The first oracle, denoted by the subscript 1, corresponds to a data set[4] produced by the REF-FINDER [63] tool which recognizes refactorings in commit histories using coarse-grained change information (e.g., changes in the subtyping relation). We manually inspected all occurrences to the *"Replace Magic Constant"* refactoring in this data set, discarded the false positives, and —without loss of generality— discarded the commits that span multiple files. The latter because our prototype implementation is currently limited to querying changes between two revisions of the same file. The commits listed in Table 7.3 are all such commits in the RF data set.

- The second oracle, denoted by the subscript 2, corresponds to a data set[5] originating from a study by Murphy-Hill et. al [59] of logs of interactions of developers with the refactoring functionality of their IDE. Each commit in this data set has already been cross-checked by the authors with the interaction logs. After filtering commits that span multiple files, we are left with 3 instances of the *"Remove Unused Method"* refactoring and 4 instances of *"Field Rename"* refactoring in Table 7.3.

Note that the commit identifiers listed in Table 7.2 differ depending on the data set the commit stems from. For commits with subscript 1, the short identifier from the project's GitHub repository is used. For commits with subscript 2, we use the same identifier as the authors of the original study.

[4]`http://web.cs.ucla.edu/~miryung/inspected_dataset.zip`
[5]`http://multiview.cs.pdx.edu/refactoring/experiments/`

```
1  (query-changes esg ?es
2    [?not-present ?method ?literal value
3     ?cmethod ?field ?field-access]
4     (in-current-es [es ast]
5       (== ast ?absent)
6       (ast-method ast ?method)
7       (child+ ?method ?literal)
8       (literal-value ?literal ?value))
9     change->*
10    (in-current-es [es ast]
11      (ast-ast-field|introduced ?absent ast ?field)
12      (field-value|initialized ?field ?value)
13      (ast-method-method|corresponding ast ?method ?cmethod)
14      (child+ ?cmethod ?field-access)
15      (field-name|accessed ?field ?field-access)))
```

Figure 7.11: Evolution query for those changes in a commit that implement a "*Replace Magic Constant*" refactoring.

7.5.2 Queries for Changes Implementing Refactorings

We describe the queries used to identify the exact changes contributing to the "replace magic constant", "remove unused method", and "rename field" refactorings. The query results will be discussed in the next section.

Query for "Replace Magic Constant" Figure 7.11 depicts the query that identifies changes from a commit that implement a "*Replace Magic Constant*" refactoring. This refactoring extracts a literal string or number from the body of a method to a field, and updates the method such that it references the newly introduced field. The first line of Figure 7.11 launches the query for a path ending in an Evolution State `?es` through Evolution State Graph `esg`. Lines 2–3 introduce additional logic variables used throughout the query. Lines 4–8 describe the initial Evolution State of the source code. Line 5 unifies the original AST with `?absent`, so that it can be used later to determine whether a fresh field has been introduced. Lines 6–8 identify a method `?method` that contains a constant value `?value`. Line 9 uses the `change->*` operator to apply an arbitrary number of changes. Lines 10–15 describe a future Evolution State, in which a new field has been introduced to replace the constant value. Line 11 uses `ast-ast-field|introduced` to ensure that `?field` is absent from its first AST argument, but present in its second. Line 12 ensures that this field features the constant `?value` as its initializer expression. Line 13 uses `ast-method-method|corresponding` to retrieve a method `?cmethod` in the current Evolution State that corresponds to `?method` in the original one. The names and signatures of the methods are required to match, but not their bodies. Finally, lines 14–15 ensure that this method now accesses the newly introduced field.

Query for "Remove Unused Method" Figure 7.12 depicts the query that identifies changes implementing a "*Remove Unused Method*" refactoring. The query de-

```
1  (query-changes esg ?end
2    [?method]
3    (in-current-es [es ast]
4      (child+ ast ?method)
5      (ast :MethodDeclaration ?method)
6      (method|unused ?method))
7    change->*
8    (in-current-es [es ast]
9      (ast-method|absent ast ?method)
10     (method|unused ?method)))
```

Figure 7.12: Evolution query for those changes in a commit that implement a *"Remove Unused Method"* refactoring.

scribes an initial Evolution State containing an unused method, and a later Evolution State in which the method is no longer present. Lines 3–6 describe the initial ES, in which method `?method` is unused. Line 6 uses `method|unused/1`, which implements a straightforward name-based resolution mechanism to verify that ES does not contain an invocation of this method. Line 7 applies an arbitrary number of changes using `change->*`. Lines 8–10 describe a successive ES, in which no method with the same name as the name of `?method` can be found. It also ensures that there is no call introduced to the removed method.

Query for "Rename Field" Figure 7.13 depicts the query that identifies changes implementing a *"Rename Field"* refactoring. Lines 4–9 describe an initial ES in which a field is present. Lines 11–21 then describe a later ES in which that field and its accesses are no longer present, and in which a new field has been introduced that has the same number of accesses. Line 5 unifies `?original` with the AST of the starting ES, so that it can be used in future ES. Lines 6–7 unify `?field` with a field declaration of that AST. Lines 8–9 collect all the uses of that field in a list `?accesses` with length `?count`. Line 10 uses `change->*` to apply an arbitrary number of changes. Lines 11–18 describe the later ES in which the refactoring has been completed. Lines 12–13 unify `?renamed` with a field declaration. Line 14 uses `ast-field|absent` to ensure that `?renamed` is absent from the original AST, while line 15 ensures that `?field` is absent from the current AST. Next, lines 16–17 verify that this new field is used as often as the original variable. Finally, the last line ensures that no accesses to the original field are present in the AST.

7.5.3 Query Results

Table 7.2 and Table 7.3 depict the results of our validation. The first part of Table 7.2 describes the detected refactoring and the data set from which this refactoring stems. The second part of the table depicts metrics about the distilled changes and corresponding CDG. Column *#Ch* shows the total number of distilled changes for the file. Next, column *LP* shows the length of the longest path through the CDG.

```
(query-changes esg ?es
  [?original ?field ?accesses
   ?count ?renamed ?renamed-accesses]
  (in-current-es [es ast]
    (== ast ?original)
    (child+ ast ?field)
    (ast-field ast ?field)
    (ast-field-list|accesses ast ?field ?accesses)
    (length ?accesses ?count))
  change->*
  (in-current-es [es ast]
    (child+ ast ?renamed)
    (ast-field ast ?renamed)
    (ast-field|absent ?original ?renamed)
    (ast-field|absent ast ?field)
    (ast-field-list|accesses ast ?renamed ?renamed-accesses)
    (length ?renamed-accesses ?count)
    (ast-field|unaccessed ast ?field)))
```

Figure 7.13: Evolution query for those changes in a commit that implement a *"Rename Field"* refactoring.

Column *MP* shows the median length of the paths through CDG. Both indicate, using our approach, how many changes need to be applied before a given change becomes applicable. Were the output of a change distiller used directly, this would be all of the preceding changes in the distilled sequence. The last columns show how connected the graph is. Column *#Co* shows the number of connected components in the CDG. Changes from one component can only be connected with changes from the same component. Column *#Single* shows the number of components that contain only a single node. Columns *MaxIn* and *MIn* show respectively the maximum and median in-degree of the CDG (i. e., the number of changes depending on a change). Finally, Columns *MaxOut* and *MOut* show respectively the maximum and median out-degree the CDG (i. e., the number of changes a change depends on).

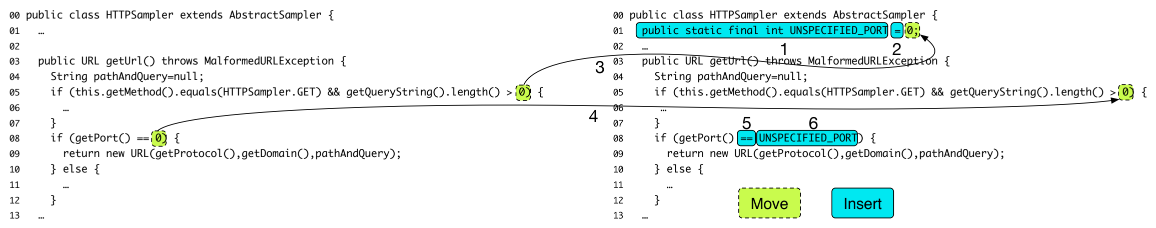
Table 7.3 describes the solution and its changes. Column *#Sol* depicts the number of changes in the minimal, executable solution returned by the query. Next, Columns *LS* and *MS* depict respectively the longest and median span, indicating the number of changes that separate two successive changes in the solution. Column *#DS* depicts the total number of changes that would need to be applied were the distilled output queried directly, before the described evolution pattern would be recognized. Thus, the columns *LS*, *MS* and *#DS* indicate how many irrelevant changes would need to be applied when **not** using our approach, while *#Sol* indicates the total number of changes that actually need to be applied using our approach. The next three columns depict a manual classification of the solution into either *Evolution Implementing* (*#EI*), *Evolution Supporting* (*#ES*) and *Evolution Linking* (*#EL*) ones:

Table 7.2: Table describing the data and computed CDG of our evaluation. The first column shows the identifier that links Table 7.3 and this table. The next four columns describe the used data and detected refactoring. The rest of the columns describe the distilled changes and their corresponding CDG.

Id	Ref.	Project	Commit	Class	#Ch	LP	MP	#Co	#Single	MaxIn	MIn	MaxOut	MOut
1	Constant_1	ant	d97f4f3	`WeblogicDeploymentTool`	202	14	8	10	4	16	1	10	1
2	Constant_1	ant	34dc512	`Jar`	74	10	3	5	4	33	1	6	2
3	Constant_1	ant	a794b2b	`FixCRLF`	1244	20	6	2	1	235	1	49	2
4	Constant_1	JMeter	b57a7b3	`AuthPanel`	245	10	4	17	15	106	1	11	1
5	Constant_1	JMeter	3a53a0a	`HTTPSampler`	149	7	2	36	18	11	1	6	1
6	Constant_1	JMeter	8275917	`HTTPSampler`	25	5	2	4	0	5	1	4	1
7	Method_2	jdt.ui	678	`JavaEditor`	11	2	1	10	9	1	1	1	1
8	Method_2	jdt.ui	2910	`JavaNavigatorContentProvider`	5	1	1	5	5	0	0	0	0
9	Method_2	jdt.ui	2722	`StubUtility2`	291	8	2	28	24	91	1	10	1
10	Field_2	jdt.ui.test	0277	`MarkerResolutionTest`	10	4	2.5	5	4	1	1	3	1
11	Field_2	jdt.ui	2810	`SourceAnalyzer`	63	7	2	25	22	1	1	6	2
12	Field_2	jdt.ui	2810	`SourceProvider`	27	6	2	12	10	3	1	3	2
13	Field_2	jdt.ui	2810	`InlineMethodRefactoring`	221	14	3	41	33	41	1	9	2

Table 7.3: Table depicting the solutions of our evaluation. The first column shows the identifier. The next seven columns describe the minimal solution and its changes. The last column describes the running time of the computation of the result.

Id	#Sol	LS	MS	#DS	#EI	#ES	#EL	Time(s)
1	8	29	8	82	4	4	0	33
2	4	11	7	23	4	0	0	19
3	10	300	68	1000	4	6	0	2054
4	5	135	23	199	4	1	0	106
5	5	96	8.5	118	4	1	0	1985
6	6	4	3	14	4	1	1	285
7	2	1	0.5	1	1	0	1	748
8	1	0	0	0	1	0	0	3
9	39	22	4.5	264	2	0	37	1536
10	3	3	2	9	3	0	0	53
11	13	13	3	57	13	0	0	8842
12	15	3	1	24	10	5	0	3133
13	6	77	29	213	6	0	0	5757



```
00 public class HTTPSampler extends AbstractSampler {
01   …
02
03   public URL getUrl() throws MalformedURLException {
04     String pathAndQuery=null;
05     if (this.getMethod().equals(HTTPSampler.GET) && getQueryString().length() > 0) {
06       …
07     }
08     if (getPort() == 0) {
09       return new URL(getProtocol(),getDomain(),pathAndQuery);
10     } else {
11       …
12     }
13   …
```

```
00 public class HTTPSampler extends AbstractSampler {
01   public static final int UNSPECIFIED_PORT = 0;
02   …
03   public URL getUrl() throws MalformedURLException {
04     String pathAndQuery=null;
05     if (this.getMethod().equals(HTTPSampler.GET) && getQueryString().length() > 0) {
06       …
07     }
08     if (getPort() == UNSPECIFIED_PORT) {
09       return new URL(getProtocol(),getDomain(),pathAndQuery);
10     } else {
11       …
12     }
13   …
```

Figure 7.14: Code snippet from the `HTTPSampler` class, in which a new field is introduced (1,2). This field is initialized via a move of a constant (3), which itself is replaced by a different move (4). Move 4 is an EL change as its node-to-be-moved is overwritten by a later insert (6). Insert 5, unnecessarily, overwrites the parent location of change 6, and is classified as an ES change.

Evolution Implementing An EI change is an integral part of the sought-after evolution pattern. In the *"Rename Field"* refactoring, for example, the change modifying the name of the field is considered as evolution implementing.

Evolution Supporting An ES change is not an integral part of the sought-after evolution pattern, but is depended on by one of its EI changes. Without the ES change, the EI change would no longer be executable. For example, an EI change inserting a field access into a method body depends on ES changes preparing that method's body.

Evolution Linking An EL change is included in the minimal solution, but is neither an EI nor an ES change. EL changes ensure that the remainder of the distilled changes can still be executed after each change in the solution has been executed. As such, they link the solution to the rest of the distilled changes. For example, when parts of a method that fell victim to the *"Remove Method"* refactoring are moved and subsequently changed elsewhere, the minimal solution will include these moves as EL changes. These changes could be removed from the solution by our approach.

Finally, the last column indicates the total running time in seconds for distilling the changes, constructing the CDG, and finding a single minimal solution. This is the running time of a single execution, and only serves to provide the general order of magnitude of the running time.

Figure 7.14 illustrates this classification of the changes in the solution to the *"Replace Magic Constant"* query against commit `8275917`. Before the commit, method `getUrl()` contained the constant 0 twice: once as a magic constant on line 8, and once as part of a check for an empty list on line 5. In the depicted solution, change 1 inserts a new field "`private static int UNSPECIFIED_PORT;`", change 2 inserts an initializer expression "`...= 0;`" into the field, and change 3 moves the latter 0 to replace the `null` in the initializer, leaving a copy of the value behind on line 5 as it is a mandatory node (cf. the minimal representation of AST nodes discussed in Section 6.2.2). Change 4 then moves the former 0 from line 8 to replace the one on line 5. Change 5 overwrites the infix expression on line 8 as its textual representation differs too much between both revisions. The left hand side is kept, while change 6 inserts a new field access in the right hand side. Changes 1, 2, 3 and 6 in the solution are EI changes implementing the actual sought-after refactoring. Change 5 is an ES change as it is depended upon by change 6. Change 4 is an EL change as it would no longer be applicable after the application of change 6, which overwrites the node-to-be-moved. It is not required for the sought-after refactoring, Note that we performed this classification manually, ensuring that the sum of *#EI*, *#ES*, and *#EL* is always *#Sol*.

Results for "Replace Magic Constant" For each of the refactoring commits from projects `ant` and `JMeter`, the evolution query depicted in Figure 7.11 reports a minimal solution consisting of the 4 sought-after EI changes: two for inserting a new field declaration and its name, one for copying the magic constant to the field initializer, and one for replacing the constant with an access to the inserted field. The remaining ES changes always prepare a parent node for this field access. We already explained the EL change in the minimal solution for commit `8275917` above using Figure 7.14.

Note that the size of the change sequence distilled for the different commits containing this refactoring varies wildly, as does their complexity. As demonstrated by columns #Sol and #DS, the CDG reduces the number of changes that need to be applied for any given change significantly. Thus, the returned minimal solutions always consist out of a very small number of changes. The minimal solution for commit `a794b2b`, for example, includes only 11 of the 1244 changes distilled in total. More than doubling class `FixCRLF` from 429 to 972 lines of code, this commit contains many changes unrelated to the sought-after ones. Here, we also find the largest span in the distilled change sequence between any two solution changes: 300 successive changes would have to be searched through to find the next change that is part of the solution, and 1000 changes would be applied in total, compared to 10 changes using our approach.

Each of the minimal solutions can be replayed on the source code before the commit. However, doing so might not eliminate every copy of the magic constant. This is because our specified query is somewhat too relaxed. Its minimal solution only needs to encompass the changes that eliminate a single copy of the constant, leaving the changes that eliminate the remaining copies behind. The query could be improved by, for instance, requiring that the final evolution state has as many accesses to the newly introduced field as there were copies of the constant. The danger of making queries this strict is that some instances of the evolution pattern will no longer be recognized. This would already be the case for the commit in Figure 7.14, where the same constant is used for two different purposes.

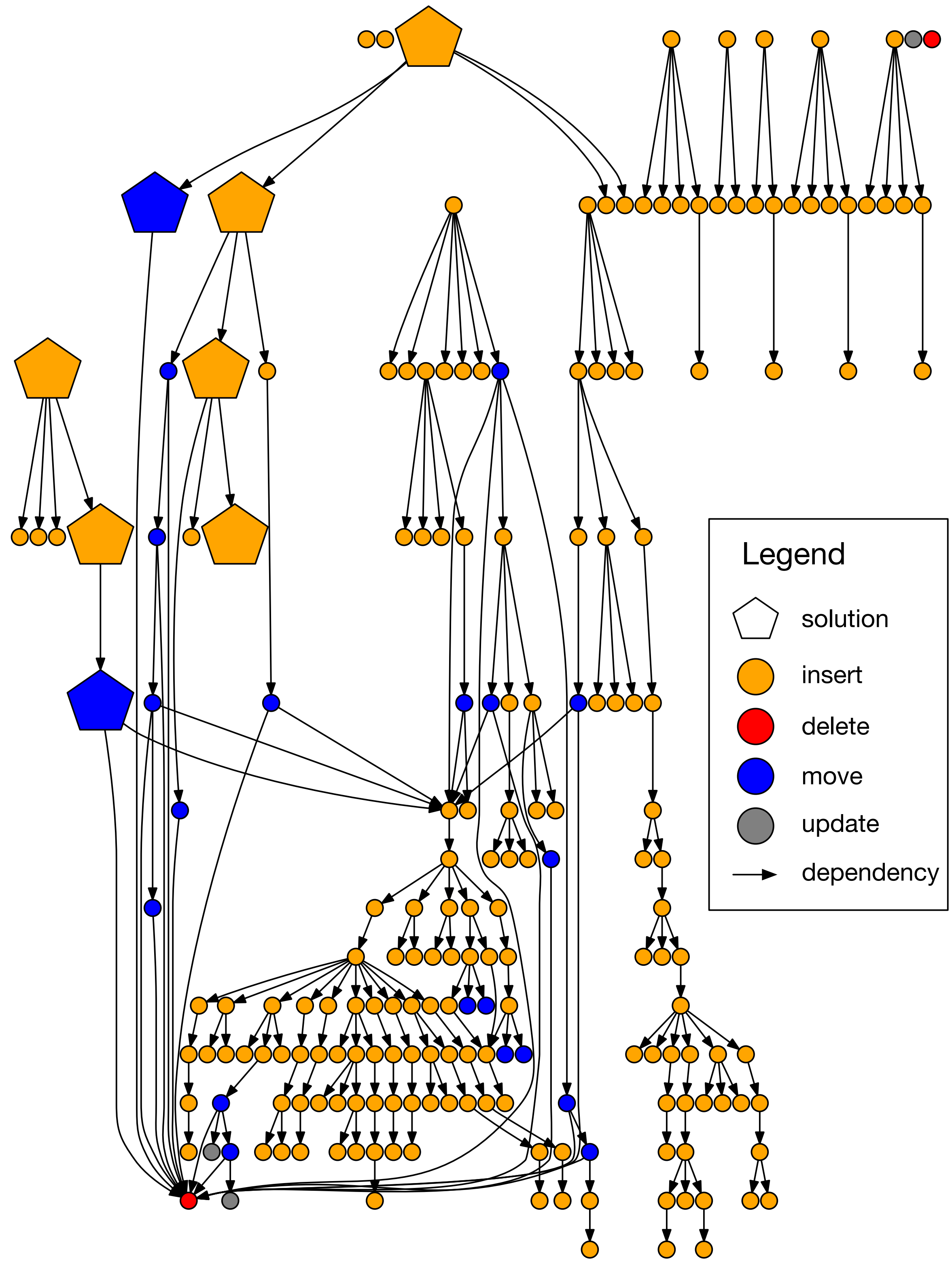


Figure 7.15: Figure depicting the CDG created for the commit with identifier 2 in Table 7.2.

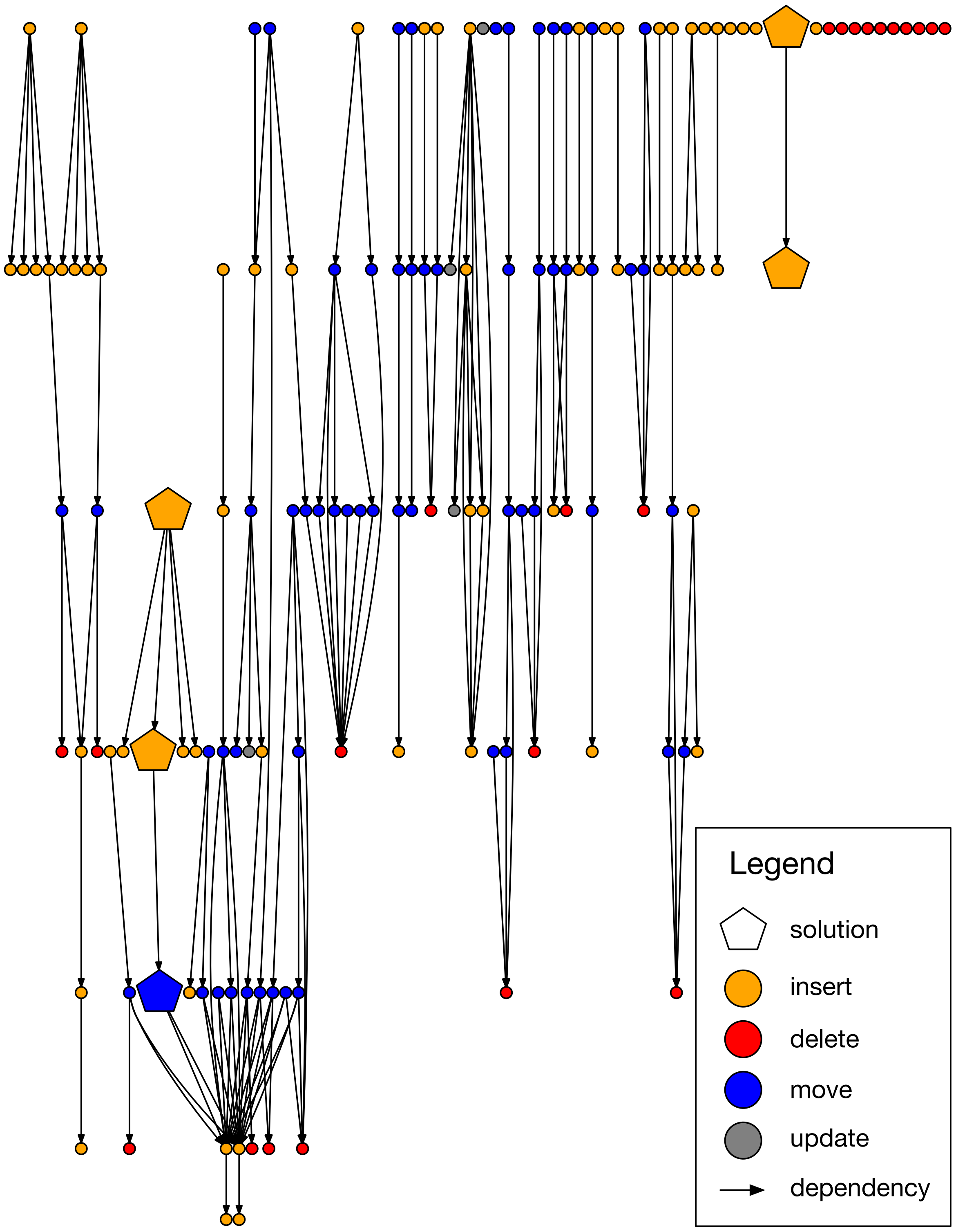


Figure 7.16: Figure depicting the CDG created for the commit with identifier 5 in Table 7.2.

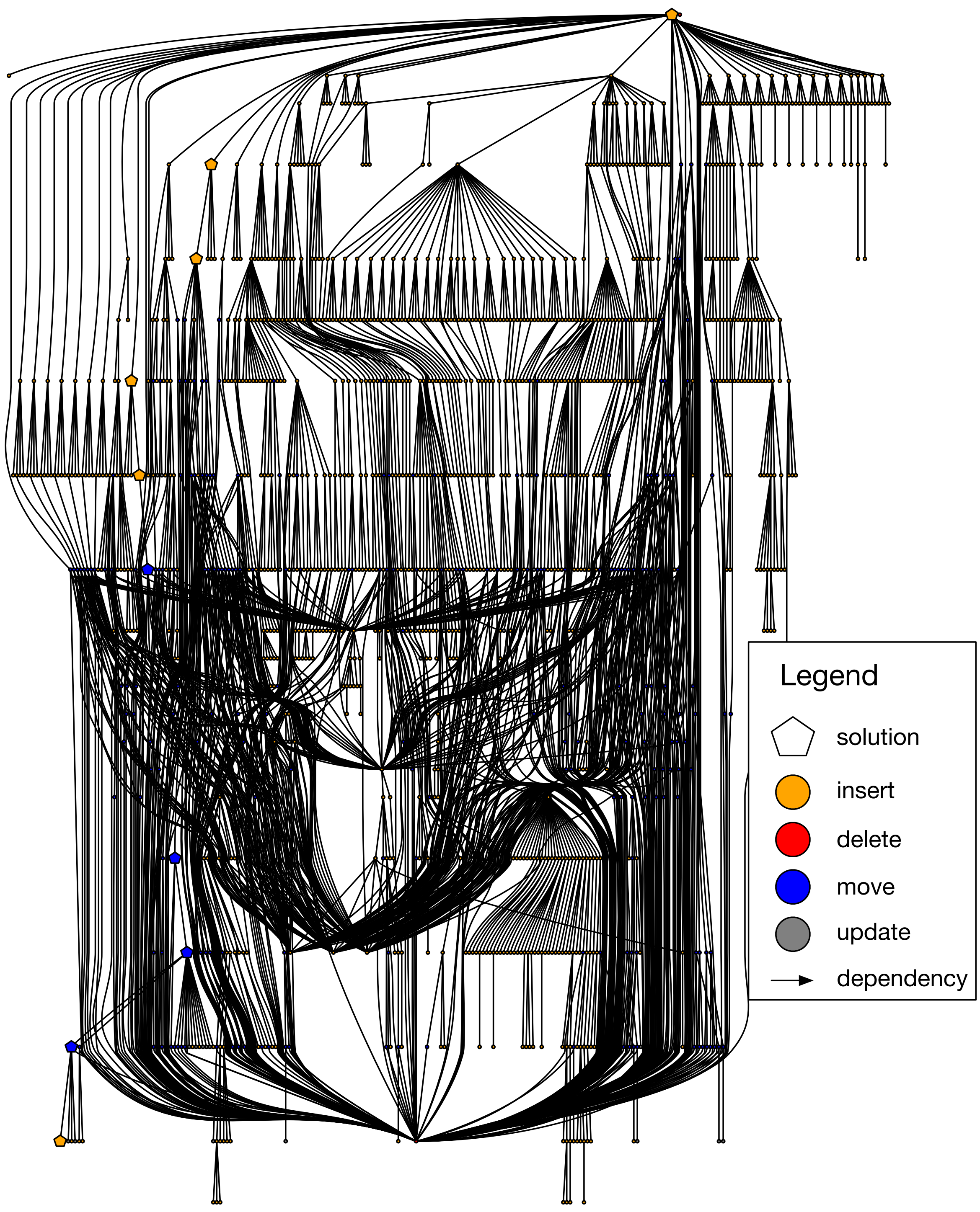


Figure 7.17: Figure depicting the CDG created for the commit with identifier 3 Table 7.2.

Figure 7.15 depicts the CDG created for commit `34dc512` of the `ant` project. Figure 7.16 depicts the CDG created for commit `3a53a0a` of the `JMeter` project. Every distilled change corresponds to a node in the graph. Dependent changes are connected through an edge. The colors of the node indicate the change type. Changes that are part of the minimal solution are depicted as a pentagon. Figure 7.15 demonstrates that inserting blocks of code results in many parent dependencies. This is due to the minimal representation of change operations (cf., Section 6.2.2). Both figures illustrate that the CDGs has several connected components, with changes that can be applied independently from each other.

Figure 7.17 depicts the CDG created for commit `a794b2b` of the `ant` project. This figure illustrates the complexity of some of the CDGs. The high number of distilled changes results in a complex CDG, where a manual detection of the dependencies is not feasible.

Results for "Remove Unused Method" The sought-after refactoring can be performed by a single change, namely a delete of the unused method. Inspecting the results we note that this only holds for a single case. The returned solution solution for `StubUtility2` even contains 39 changes in total. This is due to parts of the removed method being moved to different locations by other changes. These moves are part of the solution, and are classified as EL changes. We also note that there are 2 EI changes: two methods with the same name are removed. This is due to the declarative specification, requiring that no methods with the same name are present. A stricter specification would prevent this from happening.

Figure 7.18 depicts the CDG created for the `StubUtility2` class. This figure clearly illustrates the evolution linking move operations (i. e., the blue pentagons), and the single evolution implementing delete operation (i. e., the red pentagon). The nodes-to-be-moved are all part of the subtree that will be removed by the delete. Thus, the moves must be applied before the delete. Two similar delete operations are present; one on the left side of the figure that is not part of the solution and one in the top right corner of the figure that is part of the solution. Each delete that is part of the solution removes one of the two methods that share their name.

Results for "Rename Field" The final results are for the "Rename Field" refactoring. The number of changes in the solution differ across the different instances. This is due to the nature of the refactoring, as it requires that every access is updated to reflect the name change. Implementing this query without our approach, but by directly querying the distilled changes, would be hard as the number of changes is not known beforehand. These changes can also span the entire change sequence, as the accesses can happen throughout the whole AST. We note that the running times for all but one example are high compared to the other refactorings. This can be attributed to the nature of the declarative description of the source

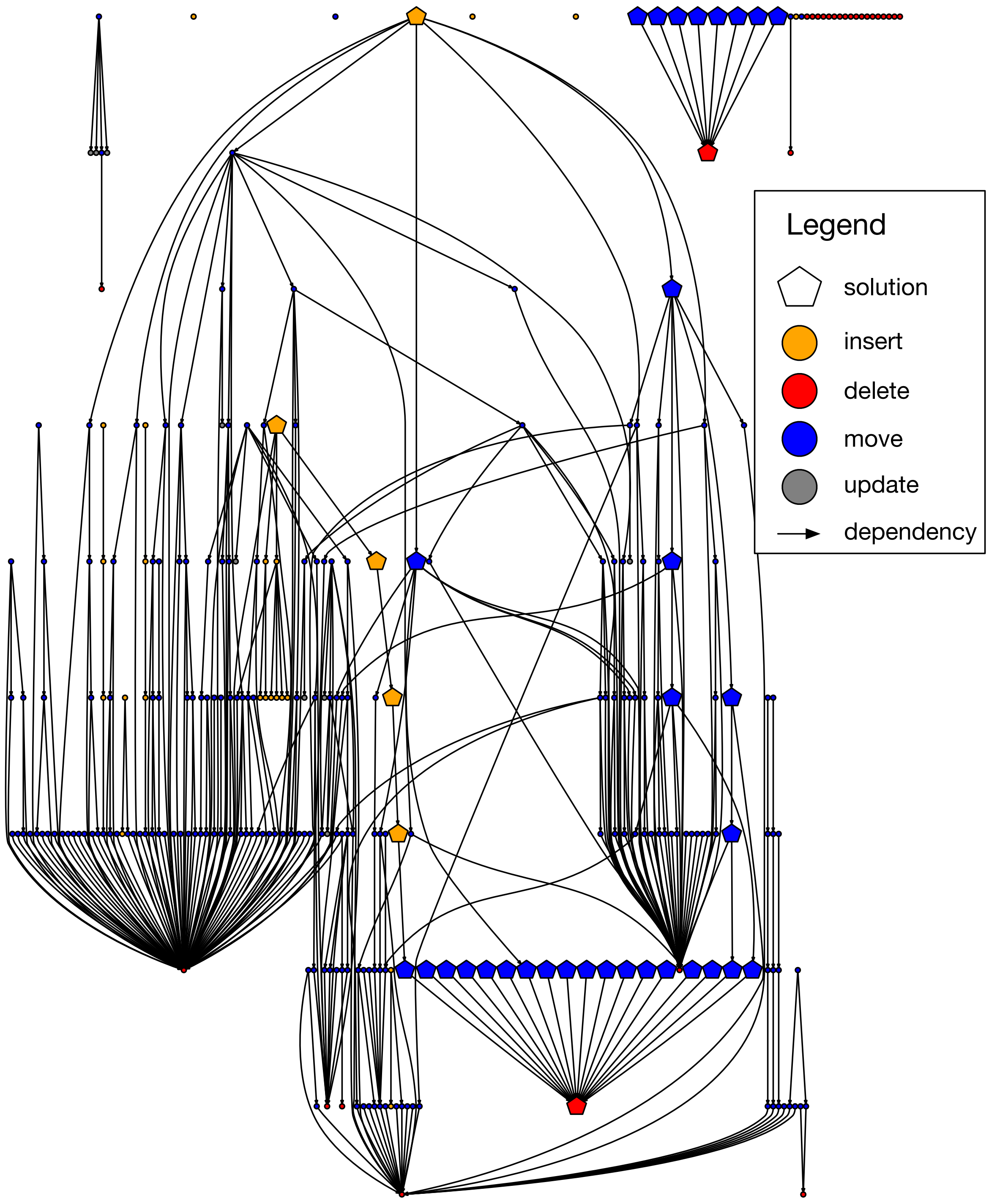


Figure 7.18: Figure depicting the CDG created for the commit with identifier 9 in Table 7.2.

code, which takes several seconds to run on a single ES. Detecting the absence of an element requires visiting all the nodes in the AST to ensure that the element is not present, which is a slow process.

Figure 7.19 depicts the CDG created for the `SourceProvider` class. This figure depicts the different evolution implementing updates that update the accesses to the the renamed field.

From this validation we can answer the three RQ, introduced in Section 7.5:

RQ1: Identifying instances of the same evolution pattern We have successfully used a single query to identify different instances of the same pattern in different change sequences created for commits containing the sought-after pattern. As such, we can affirmatively answer RQ1.

RQ2: Minimal and executable change script We have manually inspected the returned change sequences and classified their constituent changes. Some solutions may still contain evolution linking changes that are, strictly speaking, not part of the minimal solution. Evolution linking changes ensure that the remainder of the distilled changes can still be executed after each change in the solution has been executed. It is left to the user to discard the unwanted EL changes from a solution. The returned solutions are relatively small compared to the solutions that would be returned by directly querying the distilled changes, rendering a manual inspection feasible. Our approach moreover guarantees that changes can still be applied after applying the solution. The remaining changes distilled for the commit will, when executed, in turn transform the state of the code *after the refactoring* to the state of the code *after the commit*. As such, we can positively answer RQ2.

RQ3: Solution size, precision and number of executed changes We have computed several metrics about the solutions returned by QwalKeko, the distilled changes and the CDG. From metrics LS, MS and #DS – indicating how solution changes are interspersed between non-solution ones – we deduce that replaying the distilled change sequence until a desired ES is found would result in much larger solutions. Thus, without the support for evolution characteristics many more changes would need to be applied before the sought-after transformation is present. We do note that our returned solutions may still contain EL changes that a user, if desired, wants to filter out. Nonetheless, our approach lowers the number of changes that its users need to inspect. We also want to stress that our approach focuses on finding *minimal* solutions; if users merely desire to know whether a change sequence implements a certain code evolution, simply replaying the distilled changes until a desired state is encountered suffices.

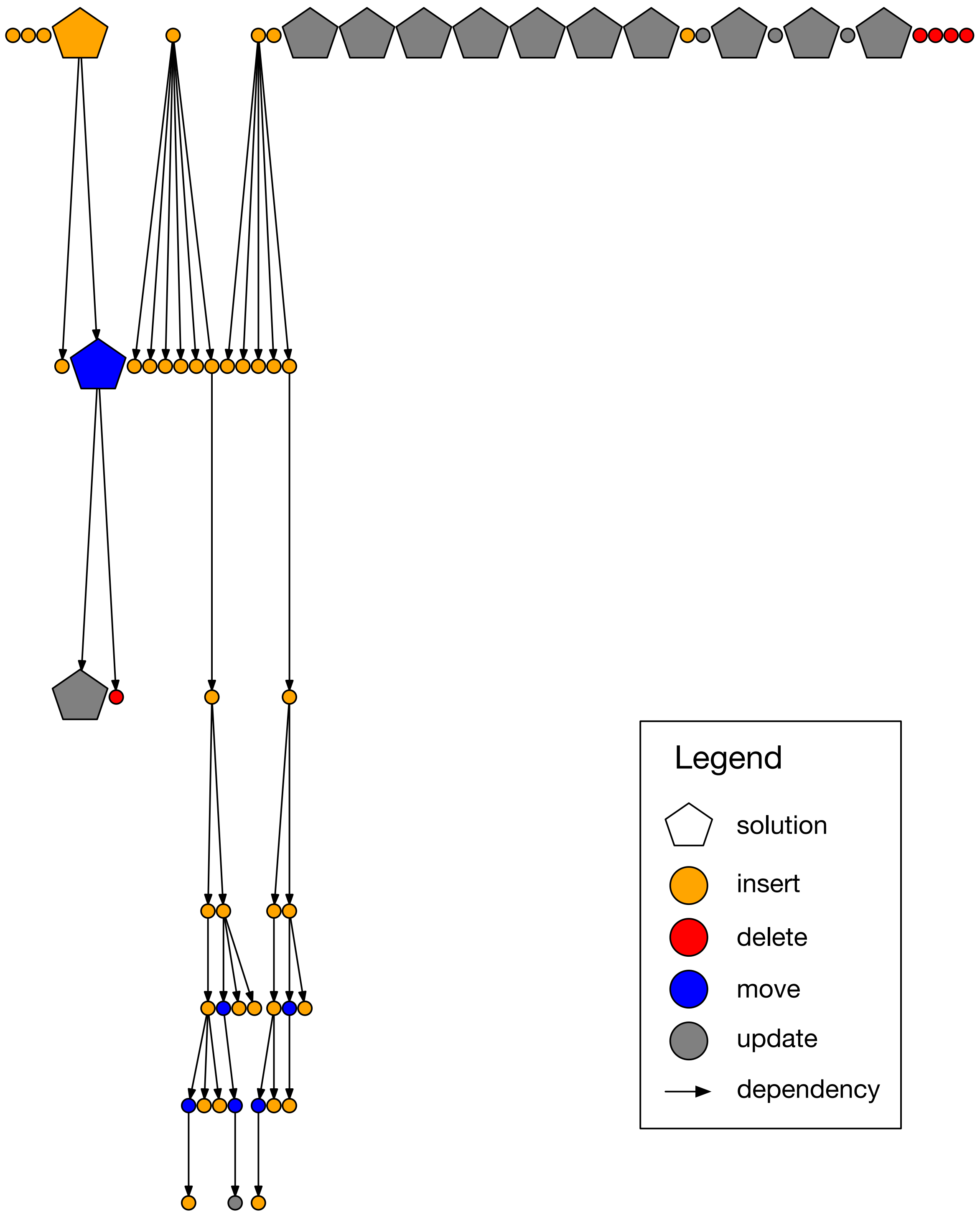


Figure 7.19: Figure depicting the CDG created for the commit with identifier 12 in Table 7.2.

7.6 Discussion

In Chapter 6 we concluded that QwalKeko supported Criteria C1–C3, expressing revision, temporal and change characteristics, but did not yet support C4, expressing evolution characteristics. In this chapter we extended QwalKeko with support for evolution characteristics.

C4: Support Evolution Characteristics Section 7.5 provides an evaluation of the support for evolution characteristics in our history query language. By answering RQ1 and RQ2 affirmatively we can state that QwalKeko supports evolution characteristics in its history query. A solution consists of an executable, minimal, (possibly non-contiguous) subsequence of changes.

Users can express evolution characteristics by reusing existing concepts of QwalKeko. The characteristics of an evolution state are expressed in terms of the same support for revision characteristics that its users are already familiar with. The navigation of the evolution state graph is specified using Qwal through the use of specialized operators. Change characteristics, although not desired due to the change equivalence problem (cf., Section 7.1), can also be specified.

We have opted to introduce specialized "change navigation" operators over reusing existing Qwal operators (cf., Table 7.1) to make the distinction between navigating a revision graph and an evolution state graph more clear. For some of the change navigation operators, there is a corresponding revision navigation operator, such is the case for `change->` and `q=>`. Other change navigation operators, such as `change==>`, don't. Thus, we have opted for a dedicated set of operators over reusing the existing, generic ones.

QwalKeko supports evolution characteristics in its history queries.

C5: Support Query Reuse, Abstraction and Composition Logic conditions expressing evolution characteristics can be abstracted into predicates. QwalKeko supports the different characteristics in a uniform, declarative language. This uniformity facilitates the abstraction, reuse and composition of queries.

QwalKeko supports query abstraction, reuse and composition.

C6: On-demand Answers The declarative nature of QwalKeko's support for evolution characteristics supports finding *non-minimal*, executable solutions in an on-demand manner. Backtracking over the ESG may yield additional ES exhibiting the specified evolution characteristics. Solutions can in turn be minimized using the approach discussed in Section 7.4.3.

QwalKeko provides solutions in an on-demand manner.

7.7 Conclusion

In this chapter we discussed the challenges a user faces when specifying the characteristics of a sought-after source code transformation in terms of the characteristics of multiple changes. First, the *change equivalence* problem states that a single source code transformation can be implemented by different change sequences. Specification of a transformation must account for all different possible change sequences. Second, the *change representation* problem stipulates that changes returned from a distiller involve three different ASTs. This renders specifications complex as comparing nodes from different ASTs for equality produces incorrect results.

As a solution for these two problems we extended QwalKeko with dedicated support for evolution characteristics, enabling users to specify the sought-after transformation in terms of a path through the evolution state graph, and the characteristics of the code of the ASTs of the evolution states along this path. As a solution, a *minimal*, *executable* subsequence of changes is returned. This solves both the change equivalence and representation problem as the specification does not rely on change characteristics. Implementation-wise, constructing the evolution state graph requires consulting a secondary auxiliary graph called the change dependency graph. This graph models dependencies between changes, and facilitates retrieving the applicable changes for any given evolution state.

We evaluated our support for evolution characteristics on its ability to identify multiple instances of the same evolution pattern in different change sequences (RQ1), whether it returns solutions that are minimal and executable (RQ2), and how it compares to directly querying the output of a distilled change sequence with respect to solution size, precision and the number of changes that need to be executed (RQ3). To this end, we detected instances of known refactorings in open-source projects using history queries in which we specified the evolution characteristics of the refactoring, and verified that the returned solutions were minimal and executable. We concluded that QwalKeko supports evolution characteristics, hereby fulfilling to all the criteria for a general-purpose history querying tool stipulated in Section 2.4.

8 CONCLUSION AND FUTURE WORK

8.1 Summary of the Dissertation

Version Control Systems enable developers to share changes, undo changes or create branches that do not affect the main branch of the software project. The history stored in such a VCS can be leveraged by developers to answer questions regarding the evolution of the software project, or by researchers in the domain of mining software repositories. Both stakeholders require support in the form of a tool that enables specifying the characteristics of sought-after history elements, and that subsequently returns the corresponding elements adhering to the given specification.

This dissertation makes the following contributions:

1. We identify and motivate the different criteria for a general-purpose history querying tool that serves the needs of stakeholders in obtaining information about the history of a software project.

2. We have designed and implemented a declarative history querying tool QwalKeko that satisfies these criteria. It has a declarative foundation: characteristics are expressed in logic queries, while a logic proof procedure identifies history elements exhibiting the specified characteristics. Unique to the approach is the use of regular path expressions for specifying paths through different graph structures, a revision graph and an evolution state graph, and the use of logic conditions within such a regular path expression specifying the characteristics nodes along this path must exhibit. Most importantly, evolution characteristics are expressed in terms of a regular path expression and source code characteristics. This change-agnostic specification solves the change equivalence problem (cf., Section 7.1). The specification of a code transformation in terms of a path through an evolution state graph and the characteristics of the ASTs along this path supports identifying different instances of the same code transformation in different distilled change sequences.

3. We have validated that QwalKeko adheres to the different criteria for a general-purpose history querying tool, and serves the needs of the stakeholders through example queries and empirical studies that are representative for its intended use. For some studies, the results are scientific contributions by themselves.

8.2 Criteria for General-Purpose History Querying Support

Section 2.4 introduces the different criteria for a general-purpose history querying tool, based on the applications of the stakeholders of history information. We define the following criteria:

C1–C4 A history querying tool must support the specification of revision (C1), temporal (C2), change (C3) and evolution (C4) characteristics. Revision characteristics concern the properties elements of a single revision must exhibit. Temporal characteristics concern quantification over elements from different revisions. Change characteristics concern individual source code changes that occurred between two revisions. Evolution characteristics concern code transformations that are implemented by change sequences.

C5 A history querying tool must support a means for query abstraction, reuse and composition.

C6 A history querying tool must provide solutions in an on-demand manner.

Chapter 3 provides an overview of the state of the art in source code querying tools. Program querying tools such as CodeQuest [43], JTL [15] and SOUL [20], do not support criteria C2, C3 and C4. Early history querying tools such as SCQL [46] only support expressing coarse-grained revision characteristics, and none of criteria C3 and C4. Later history querying tools such as Boa [25] do support fine-grained revision characteristics, but do not fulfill criteria C3 and C4, nor do they support query abstraction, reuse and composition. Queries, for instance, are specified as imperative visitors over ASTs which need to be composed manually. Other tools such as CheOPSJ [69] support querying changes, but lack a dedicated query language for specifying the characteristics of the sought-after history information.

Chapter 4 presents a high-level overview of the different components of QwalKeko. The declarative foundation facilitates integrating these components into a uniform history query language that support query abstraction, reuse and composition (C5) and computes solutions in an on-demand manner (C6). These components are detailed in the subsequent chapters. Chapter 5 provides the foundation of our approach to support history querying. It combines the declarative program querying tool Ekeko with the graph query language Qwal to support revision (C1) and temporal (C2) characteristics. Chapter 6 extends QwalKeko with support for change characteristics (C3) by reifying the output of ChangeNodes. ChangeNodes computes a sequence of fine-grained source code changes between two files of Java code through a change differencing algorithm. Chapter 7 extends QwalKeko with support for evolution characteristics (C4).

C1: Supporting Revision Characteristics using Logic Queries Section 5.4 introduced Ekeko, a declarative program querying tool that enables users to query the Java projects in an Eclipse workspace. Ekeko supports querying the AST of a single revision using logic queries of which the conditions specify revision characteristics. To this end, we reified the meta-data of a revision, such as the author, commit message, timestamp, etc., and extended Ekeko with new predicates for this data.

C2: Supporting Temporal Characteristics through Regular Path Expressions Embedded in Logic Queries Section 5.2 introduced Qwal, a regular path expression [17] language that enables users to specify paths through arbitrary graphs. QwalKeko converts the information stored in a VCS to a graph, in which each node corresponds to a revision, and successive revisions are connected by an edge. Section 5.2.1 provides several example queries depicting the usage of Qwal. A Qwal query consists of navigation predicates that specify a path through a revision graph, and the conditions that have to hold in revisions along this path.

Qwal provides a generic set of navigation predicates to specify paths through any graph. The extensible implementation of Qwal enables users to define their own domain-specific navigation predicates if needed. This extensible implementation is used in Chapter 7 to implement support for evolution characteristics which are specified as paths through an evolution state graph.

Section 5.5 evaluates the foundation of our approach to history querying. It provides queries for several scenarios of the stakeholders of history information. These queries illustrate how regular path expressions, implemented by Qwal, are used to express temporal characteristics by navigating the revision graph, while the declarative program query language Ekeko is used to express revision characteristics that have to hold in revisions along the specified path.

We conclude that combination of Qwal and Ekeko, forming the foundation of QwalKeko, satisfies criteria C1–2 and C5–6. Criteria C5 and C6 are supported due to the declarative nature of the chosen specification formalisms

C3: Supporting Change Characteristics through a Change Distiller Chapter 6 extended QwalKeko with support for change characteristics. To this end, it introduced ChangeNodes, a change distiller that computes fine-grained source code changes between two Java source files. We extended QwalKeko with a declarative layer (cf., Section 6.4) that reifies the output of ChangeNodes, hereby supporting change characteristics.

Section 6.5 evaluated QwalKeko on its support for change characteristics. To this end, we performed a mining software repositories study [13] with the goal of finding what parts of automated functional tests are most prone to change throughout the history of web applications. We implemented this experiment in QwalKeko and in Clojure. We concluded that the dedicated support of QwalKeko for revision, temporal and change characteristics (C1–C3) results in a more concise implementation than Clojure.

C4: Supporting Evolution Characteristics through an Evolution State Graph Chapter 7 discussed the two major problems of querying the output of a change distilling algorithm, such as ChangeNodes, directly to detect complex patterns. The *change representation* problem stipulates that changes involve AST nodes that stem from three ASTs; the source AST, the target AST, and a copy of the source AST that is transformed into the target AST during the execution of the algorithm. This encumbers specifying change characteristics as the user needs to track from which AST the subject of a change stems. The *change equivalence* problem stipulates that different change sequences can implement the same conceptual source code transformation. A user must account for these different possible sequences in every specification.

To address these problems we extended QwalKeko with support for evolution characteristics. A change sequence is transformed into an evolution state graph, containing intermediate ASTs that can be constructed by applying subsets of changes. A source code transformation is described as a path, using regular path expressions, through this evolution state graph, along with the source code characteristics of the intermediate ASTs. This change-agnostic specification solves the change representation and equivalence problems. As a solution, a *minimal, executable* subsequence of changes is returned. Section 7.5 evaluated this approach by detecting multiple instances of the same refactoring across different open-source projects using a single history query.

We conclude that QwalKeko satisfies all the stipulated criteria of a general-purpose history querying tool.

8.3 Limitations of the Approach

8.3.1 Performance

We did not focus on the performance and memory usage of QwalKeko. History queries may take a long time to complete, depending on the size of the project and whether all possible answers must be computed. This is acceptable when performing an empirical study, but not for developers who want near-instant answers to a question they have during the development.

There are several slow parts in QwalKeko. First, importing the source code of a revision from a VCS into Eclipse may take several seconds due to the data being written to disk and the construction of a model of that revision. Next, distilling changes between two versions has a performance of $O(n^2)$ [30], where n is the maximum number of AST nodes in either the source or target AST. Finally, for a given set of changes with size N, $N!$ different sequences with length N can be constructed in case no change has a dependency. The corresponding evolution state graph will contain all these different permutations. Computing all the solutions for such a graph is not feasible for large change sequences.

QwalKeko already mitigates some of these issues:

The declarative nature of QwalKeko supports computing results in an on-demand manner. This enables writing queries incrementally which can be tested quickly. QwalKeko features different representations of the history information (cf., Section 2.2.2). From coarse-grained to fine-grained these are a coarse-grained revision representation (e.g., the modified files or the author of a revision), a fine-grained revision representation (e.g., the source code of a revision), the fine-grained changes made between two revisions of a file and finally an evolution state graph created from such changes. An initial query could only specify coarse-grained revision characteristics (e. g., to find all the revisions by a specific author). A follow-up query could use these results to limit the number of checkouts that must be performed.

To reduce the memory usage an incremental representation of the source code could be used [2]. In such a representation an initial AST is created for the first revision containing that AST. Later revisions in which that file is modified reuse the initial AST, but also store the modifications the AST underwent. Thus, unchanged nodes are reused across revisions, reducing the memory usage.

To mitigate the problem of querying a high number of possible evolution states, we have introduced coarse-grained navigation predicates, such as `change==>` (cf., Section 7.4), that apply multiple changes at once. These predicates limit the search space of a query, at the cost of removing intermediate states that may contain the solution.

8.3.2 Detecting Patterns in Intermediate Evolution States

Currently, QwalKeko has only been used to detect source code transformations for which the sought-after transformation is present in the target AST. For example, Section 7.5 detected refactorings for which the resulting code was present in the AST of the final ES. In theory, a sought-after transformation can be present in some ES, but not in the final ES. It is ill-advised to detect intermediate ES in which the transformed code is present, but that is absent in the final ES. The construction of the ESG depends on the distilled change sequence. As such, there is no way to know beforehand whether the desired ES will actually be present, as an unexpected change sequence may be generated. Detecting patterns in intermediate evolution states does make sense in the context of *logged* changes. Such changes may uncover initial attempts of the developer when modifying the source code. An ESG can be constructed using logged changes. However, whereas an ESG for distilled changes contains all the different ASTs that could be constructed, an ESG for logged changes could use the temporal relation between changes. For example, a developer introduced two new methods. We could create two groups of changes, one that affect the first method and one that affect the second one. The order in which the complete group of changes is replayed can differ, but replaying a single group could follow the actions of the developer. This reduces the number of possible ES in the graph, and improves the performance. This temporal information is unavailable when using distilled changes. Even though similar groupings could be made, the order of changes inside a group would still depend on the output of the distiller, and suffer from the change equivalence problem.

8.4 Future Research

In this section we discuss several potential avenues of future research for querying the history of software projects. For us, the main focus lies on extending the facilities to specify evolution characteristics. Section 8.3 already discussed using logged changes instead of distilled changes as the input to our evolution state graph.

8.4.1 Other Sources of Information

This dissertation focused on querying the history and evolution of *source code*. However, software development does not include source code alone. Bug trackers, mailing lists, a continuous integration pipeline, etc., are all used to develop and maintain a project. All these are also sources of relevant information for the history information stakeholders, but are currently not integrated into QwalKeko. Mailing lists can provide insights into the design decisions of the project. Bug trackers can provide the raison d'être of a commit. Continuous integration data provides insights into the build status of a revision (i. e., whether the revision compiled successfully, what unit tests did or did not pass, ...). For example, TravisTorrent [5] collects TravisCI (a continuous integration system) information from open-source projects and makes it accessible to researchers. Existing work [3, 31] links bug tracker information to its related revision. This extra information from external sources could be incorporated in QwalKeko as additional facts in the revision graph.

8.4.2 Coarse-Grained Source Code Changes

We could use QwalKeko's support for evolution characteristics to transform fine-grained changes into coarse-grained changes, such as the ones provided by CheOPSJ (cf., Section 3.3). Examples of these coarse-grained changes are the introduction of a method and its body, a field rename, etc. Coarse-grained changes better convey the meaning of what happened between two revisions. QwalKeko ensures that the coarse-grained changes are correctly identified, regardless of the concrete distilled changes. One could add coarse-grained change characteristics to query these changes as they should not suffer from the change equivalence problem.

8.4.3 Semantic Dependencies

A next step is to incorporate semantic dependencies in our support for evolution characteristics. Currently, the change dependency graph only includes syntactical dependencies. Using data flow information one could add semantic dependencies (e. g., a method can only be called if it is introduced, a field can only be removed if it is no longer accessed, etc.). Semantic dependencies have several potential uses. First, they ensure evolution states are semantically correct. The resulting evolution state graph would be smaller as fewer semantically correct evolution states exist. Second, the semantic change dependency graph could be used to untangle commits [22, 76]. Ideally, a single commit implements a single concern (e. g., a bug fix, a feature addition, etc.). By analyzing the connectivity of the change dependency graph we hope to find subgraphs that implement such a single concern. The dependency graph ensures that the changes of the subgraph can be applied, and thus be extracted into a separate commit. We want to leverage the history of the software project to apply an edit script on similar source code, such as a different branch of the project. To this end, the history of the software project contains the information to see how both branches have diverged. This information can be used to modify the patch so that it can be applied on the different branch.

8.4.4 Empirical Studies

Finally, we want to use QwalKeko in more empirical studies. For example, our original motivation for the support for evolution characteristics was to perform a study about code clones [4, 57] that are removed by performing a refactoring (e. g., an extract method refactoring), and what additional operations developers perform besides the refactoring. The evaluation of the support for evolution characteristics (cf., Section 7.5) shows that QwalKeko can identify minimal change sequences that implement a refactoring. We want to continue this study for a larger data set using QwalKeko.

8.5 Concluding Remarks

We started this dissertation motivating the need for a general-purpose history querying tool based on the potential information about the history of a software project. We discerned the different criteria for such a general-purpose history querying tool. Throughout the remainder of the dissertation we instantiated our own history querying tool QwalKeko that satisfies these criteria.

QwalKeko is a declarative history querying tool. As its foundation it combines the program query language Ekeko with the graph query language Qwal, which implements regular path expressions. We extended QwalKeko with support for change characteristics through ChangeNodes. Finally, we extended QwalKeko with support for evolution characteristics by transforming the output of ChangeNodes into an evolution state graph. Unique to the approach is the use of regular path expressions within history queries to express paths through the different graph structures — the revision graph and the evolution state graph — and the characteristics nodes along this path must exhibit.

We evaluated QwalKeko through example queries, an empirical study regarding the evolution of automated functional tests performed once in Clojure and once in QwalKeko, and a study regarding the specification of refactorings using the support for evolution characteristics, and the identification of the subset of changes in different change sequences that implement these refactorings. These studies illustrate that QwalKeko satisfies the different criteria for a general-purpose history querying tool.

QwalKeko can be improved in different aspects, such as its performance. The evolution state graph can be extended with semantic dependencies. Such dependencies might ensure that the generated ES would be semantically correct. Next to that, they would decrease the number of evolution states that need to be generated, reducing the size of the evolution state graph.

In conclusion, QwalKeko provides a solid, declarative foundation for a general-purpose history querying tool that serves the needs of the history information stakeholders.

REFERENCES

[1] A. Aho. *Algorithms for finding patterns in strings*. MIT Press, 1990. 63

[2] Carol V. Alexandru and Harald C. Gall. Rapid multi-purpose, multi-commit code analysis. In *Proceedings of the 37th International Conference on Software Engineering (ICSE15)*, 2015. 160

[3] Adrian Bachmann, Christian Bird, Foyzur Rahman, Premkumar Devanbu, and Abraham Bernstein. The missing links: Bugs and bug-fix commits. In *Proceedings of the 18th International Symposium on Foundations of Software Engineering (FSE10)*, 2010. 161

[4] Saman Bazrafshan and Rainer Koschke. An empirical study of clone removals. In *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*, pages 50–59, 2013. 162

[5] Moritz Beller, Georgios Gousios, and Andy Zaidman. Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In *Proceedings of the 14th Working Conference on Mining Software Repositories (MSR17)*, 2017. 161

[6] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and Ph. Schnoebelen. *System and Software Verification, Model-Checking Techniques and Tools*. Springer, 2001. ISBN 9783540415237. 71

[7] Christian Bird, Tim Menzies, and Thomas Zimmermann, editors. *The Art and Science of Analyzing Software Data*. Morgan Kaufmann, 2015. 2

[8] William E. Byrd. *Relational Programming in miniKanren: Techniques, Applications, and Implementations*. PhD thesis, Indiana University, Bloomington, IN,, September 30, 2009. 46

[9] Nélio Cacho, Eiji Adachi Barbosa, Juliana Araujo, Frederico Pranto, Alessandro F. Garcia, Thiago César, Eliezio Soares, Arthur Cassio, Thomas Filipe, and Israel García. How does exception handling behavior evolve? an exploratory study in java and c# applications. In *Proceedings of the 30th International Conference on Software Maintenance and Evolution(ICSME)*, 2014. 26

[10] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989. 33

[11] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD96)*, pages 493–504, 1996. 45, 55, 91, 92, 118

[12] Tse-Hsun Chen, Weiyi Shang, Jinqiu Yang, Ahmed E. Hassan, Michael W. Godfrey, Mohamed Nasser, and Parminder Flora. An empirical study on the practice of maintaining object-relational mapping code in java systems. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR16)*, 2016. 2

[13] Laurent Christophe, Reinout Stevens, Coen De Roover, and Wolfgang De Meuter. Prevalence and maintenance of automated functional tests for web applications. In *Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSMe14)*, 2014. 2, 8, 27, 99, 100, 158

[14] Mihai Codoban, Sruti Srinivasa Ragavan, Danny Dig, and Brian Bailey. Software history under the lend: A study on why and how developers examine it. In *31st International Conference on Software Maintenance and Evolution (ICSME15)*, 2015. 11, 21, 22, 24

[15] Tal Cohen, Joseph (Yossi) Gil, and Itay Maman. JTL: the Java Tools Language. In *Proceedings of the 21st Annual SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA06)*, pages 89–108, 2006. 3, 33, 34, 157

[16] Brett Daniel, Vilas Jagannath, Danny Dig, and Darko Marinov. Reassert: Suggesting repairs for broken unit tests. In *Proceedings of the International Conference on Automated Software Engineering (ASE09)*, 2009. 115

[17] Oege de Moor, David Lacey, and Eric Van Wyk. Universal regular path queries. *Higher-Order and Symbolic Computation*, pages 15–35, 2002. 6, 7, 38, 50, 63, 157

[18] Coen De Roover and Katsuro Inoue. The ekeko/x program transformation tool. In *Proceedings of the 14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM14), Tool Demo Track*, 2014. 3

[19] Coen De Roover and Reinout Stevens. Building development tools interactively using the ekeko meta-programming library. In *Proceedings of the 18th European Conference on Software Maintenance and Reengineering (CSMR14)*, 2014. 3, 7, 53, 75

[20] Coen De Roover, Carlos Noguera, Andy Kellens, and Viviane Jonckers. The SOUL tool suite for querying programs in symbiosis with Eclipse. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ11)*, pages 71–80, 2011. 3, 34, 157

[21] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI04)*, 2004. 13, 40

[22] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. Untangling fine-grained code changes. *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER15)*, 2015. 162

[23] Stephan Diehl, Harald C. Gall, Martin Pinzger, and Ahmed E. Hassan. Introduction to MSR 2006. In *Proceedings of the 2006 International Workshop on Mining Software Repositories (MSR06)*, pages 1–2, 2006. 21, 25

[24] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In *Proceedings of the Second Workshop on Formal Methods in Software Practice (FMSP'98)*, 1998. 71

[25] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE13)*, 2013. 4, 13, 40, 157

[26] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. Mining billions of ast nodes to study actual and potential usage of java language features. In *36th International Conference on Software Engineering (ICSE14)*, 2014. 27

[27] Peter Ebraert, Jorge Vallejos, Pascal Costanza, Ellen Van Paesschen, and Theo D'Hondt. Change-oriented software engineering. In *Proceedings of the 2007 International Conference on Dynamic languages (ICDL07)*, 2007. 42

[28] E. A. Emerson and Joseph Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Computer and System Sciences*, 30(1):1–24, February 1985. 71

[29] Jean-Rémy Falleri, Cédric Teyton, Matthieu Foucault, Marc Palyart, Floréal Morandat, and Xavier Blanc. The harmony platform. *CoRR*, 2013. 37, 38

[30] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Montperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th International Conference on Automated Software Engineering (ASE14*, 2014. 18, 91, 93, 159

[31] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance (ICSM '03)*, 2003. 161

[32] Peter Flach. *Simply Logical, Intelligent Reasoning by Example*, chapter Logic and Logic Programming, pages 26–35. John Wiley & Sons, 1994. 47

[33] Beat Fluri, Michael Würsch, Martin Pinzger, and Harald C. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *Transactions on Software Engineering*, 33(11), 2007. 18, 28, 43, 55, 91, 92, 93, 118

[34] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., 1999. 84

[35] Jeffrey E.F. Friedl. *Mastering Regular Expressions, 3rd Edition*. O'Reilly Media, 2006. 32

[36] Thomas Fritz and Gail C. Murphy. Using information fragments to answer the questions developers ask. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE10)*, pages 175–184, 2010. 2, 11, 21, 22, 23, 24, 79

[37] Harald C. Gall, Beat Fluri, and Martin Pinzger. Change analysis with evolizer and changedistiller. *IEEE Software*, 26(1):26–33, January 2009. 12, 13, 43

[38] Emanuel Giger, Martin Pinzger, and Harald C. Gall. Comparing fine-grained source code changes and code churn for bug prediction. In *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR11)*, pages 83–92, 2011. 26, 28

[39] Tudor Gîrba and Stéphane Ducasse. Modeling history to analyze software evolution. *Journal of Software Maintenance: Research and Practice (JSME06)*, 18: 207–236, 2006. 17, 38, 58, 78

[40] Verónica Uquillas Gómez, Stéphane Ducasse, and Andy Kellens. Supporting streams of changes during branch integration. *Science of Computer Programming*, 96:84–106, 2014. 95

[41] Georgios Gousios. The ghtorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR13)*, MSR '13, pages 233–236, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-2936-1. 2

[42] Mark Grechanik, Qing Xie, and Chen Fu. Maintaining and evolving gui-directed test scripts. In *Proceedings of the 31st International Conference on Software Engineering (ICSE09)*, 2009. 115

[43] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. Codequest: Scalable source code queries with datalog. In *Proceedings of the 20th European conference on Object-Oriented Programming (ECOOP06)*, 2006. 3, 33, 34, 157

[44] Lile Hattori and Michele Lanza. Syde: A tool for collaborative software development. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE10)*, pages 235–238, 2010. 18, 91

[45] John Hebeler, Matthew Fisher, Ryan Blace, and Andrew Perez-Lopez. *Semantic Web Programming*. Wiley Publishing, 2009. 72

[46] Abram Hindle and Daniel M. German. SCQL: A formal model and a query language for source control repositories. In *Proceedings of the 2005 Working Conference on Mining Software Repositories (MSR05)*, pages 100–105, 2005. 4, 14, 35, 36, 157

[47] Si Huang, Myra B. Cohen, and Atif M. Memon. Repairing gui test suites using a genetic algorithm. In *Proceedings of the 3rd Internal Conference on Software Testing, Verification and Validation (ICST10)*, 2010. 115

[48] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 2002. 130

[49] David Kawrykow and Martin P. Robillard. Non-essential changes in version histories. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE11)*, 2011. 26

[50] Andy Kellens, Coen De Roover, Carlos Noguera, Reinout Stevens, and Viviane Jonckers. Reasoning over the evolution of source code using quantified regular path expressions. In *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE11)*, pages 389–393, 2011. 7, 78

[51] Thomas D. LaToza and Brad A. Myers. Hard-to-answer questions about code. In *Evaluation and Usability of Programming Languages and Tools (PLATEAU10)*, pages 8:1–8:6, 2010. 2, 5, 21, 22, 23, 24, 79

[52] Jannik Laval, Simon Denier, Stéphane Ducasse, and Jean-Rémy Falleri. Supporting simultaneous versions for software evolution assessment. *Journal of Science of Computer Programming*, 2010. 58

[53] Julia Lawall, Quentin Lambert, and Gilles Muller. Prequel: A Patch-Like Query Language for Commit History Search. Research Report RR-8918, Inria Paris, 2016. 96

[54] Yanhong A. Liu, Tom Rothamel, Fuxiang Yu, Scott D. Stoller, and Nanjun Hu. Parametric regular path queries. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI04)*, pages 219–230, 2004. 50

[55] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using pql: a program query language. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA05)*, 2005. 3, 33

[56] Na Meng, Miryung Kim, and Kathryn S. McKinley. Lase: Locating and applying systematic edits by learning from examples. In *Proceedings of the 35th International Conference on Software Engineering (ICSE13)*, 2013. 25, 26, 95

[57] Manishankar Mondal, Chanchal K. Roy, and Kevin A. Schneider. A comparative study on the bug-proneness of different types of code clones. In *Proceedings of the 31th International Conference on Software Maintenance and Evolution (ICSME15)*, 2015. 26, 162

[58] Alix Mougenot, Xavier Blanc, and Marie-Pierre Gervais. D-Praxis: A peer-to-peer collaborative model editing framework. In *Proceedings of the 9th International Conference on Distributed Applications and Interoperable Systems (DAIS09)*, pages 16–29, 2009. 4, 36, 77

[59] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. *Transactions on Software Engineering*, 38:5–18, 2012. 138

[60] Stas Negara, Mohsen Vakilian, Nicholas Chen, Ralph E. Johnson, and Danny Dig. Is it dangerous to use version control histories to study source code evolution? In *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP12)*, 2012. 18, 91

[61] Stas Negara, Mihai Codoban, Danny Dig, and Ralph E. Johnson. Mining fine-grained code changes to detect unknown change patterns. In *Proceedings of the 36th International Conference on Software Engineering (ICSE14)*, 2014. 25, 27

[62] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (SFCS77)*, SFCS '77, pages 46–57, 1977. 71

[63] Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim. Template-based reconstruction of complex refactorings. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM10)*, 2010. 138

[64] Baishakhi Ray, Meiyappan Nagappan, Christian Bird, Nachiappan Nagappan, and Thomas Zimmermann. The Uniqueness of Changes: Characteristics and Applications. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR15)*, 2015. 4, 26, 28

[65] Arend Rensink. The groove simulator: A tool for state space generation. In *Applications of Graph Transformations with Industrial Relevance (AGTIVE04)*, pages 479–485, 2004. 42

[66] Romain Robbes and Michele Lanza. Spyware: A change-aware development toolset. In *Proceedings of the 30th international conference on Software engineering (ICSE08)*, pages 847–850, 2008. 18, 91

[67] S. Tichelaar Serge Demeyer and P. Steyaert. The FAMOOS information exchange model. Technical report, University of Berne, 1999. 42, 95

[68] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of Software Engineering (FSE06)*, 2006. 2, 11

[69] Quinten David Soetens. *Change-Based Software Engineering Using Reified Changes for Test Selection and Refactoring Reconstruction*. PhD thesis, Universiteit Antwerpen, 2015. 42, 43, 157

[70] Daniela Steidl and Florian Deissenboeck. How do java methods grow? In *Proceedings of the 15th International Working Conference on Source Code Analysis and Manipulation (SCAM15)*, 2015. 27

[71] Reinout Stevens. A declarative foundation for comprehensive history querying. In *Proceedings of the 37th International Conference on Software Engineering, Doctoral Symposium Track (ICSE15)*, 2015. 4

[72] Reinout Stevens and Coen De Roover. Querying the history of software projects using QWALKEKO. In *Proceedings of the 30th International Conference on Software Maintenance and Evolution*, 2014. 7, 8, 14, 18

[73] Reinout Stevens and Coen De Roover. Extracting executable transformations from distilled code changes. In *Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering (SANER17)*, 2017. 4, 8

[74] Reinout Stevens, Coen De Roover, Carlos Noguera, and Viviane Jonckers. A history querying tool and its application to detect multi-version refactorings. In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR13)*, 2013. 7

[75] Reinout Stevens, Coen De Roover, Carlos Noguera, Andy Kellens, and Viviane Jonckers. A logic foundation for a general-purpose history querying tool. *Elsevier Journal on Science of Computer Programming*, 2014. 4, 38

[76] Yida Tao and Sunghun Kim. Partitioning composite code changes to facilitate code review. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR15)*, 2015. 162

[77] YoungSeok Yoon and Brad A. Myers. Capturing and analyzing low-level events from the code editor. *Proceedings of the 3rd Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 11)*, 2011. 18, 91

[78] Andy Zaidman, Bart Van Rompaey, Serge Demeyer, and Arie van Deursen. Mining software repositories to study co-evolution of production & test code. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation (ICST08)*, pages 220–229, 2008. 26, 28, 113