Vrije Universiteit Brussel

Faculty of Science and Bio-Engineering Sciences
Department of Computer Science

Mining Change Histories for Unknown Change Patterns

Dissertation for the degree of Master of Science in Applied Science and Engineering: Computer Science

Arvid De Meyer

Promotor: prof. dr. Coen De Roover
Advisors: ir. Reinout Stevens

AUGUST 2015

Vrije Universiteit Brussel

Faculteit Wetenschappen en Bio-ingenieurswetenschappen
Departement Computerwetenschappen

Mining Change Histories for Unknown Change Patterns

Proefschrift ingediend met het oog op het behalen van de titel Master in de Ingenieurswetenschappen: Computerwetenschappen

Arvid De Meyer

Promotor:	prof. dr. Coen De Roover
Begeleiders:	ir. Reinout Stevens

AUGUSTUS 2015

Abstract

During software evolution similar but not necessarily identical code changes are often applied to multiple code locations, thus forming frequent patterns. Finding all locations requiring modification and performing necessary changes manually is tedious, laborious and error-prone. Although tools for automating specific well-known change patterns such as refactorings exist, developers can benefit from automating previously unknown change patterns. To the best of our knowledge, no such general-purpose tool for finding and automating previously unknown change patterns from distilled changes exists.

In this thesis, the problem of finding previously unknown automatable change patterns in the history of software projects is formulated as a data mining problem. More concretely, we applied a closed frequent itemset mining algorithm, operating on generalizations of changes distilled from successive version control system snapshots.

As an implementation of the proposed approach, we have written an Eclipse plugin mining automatable Java change patterns from successive git snapshots. The implementation then proposes code locations which can be transformed according to mined change patterns. We have evaluated the implementation by first investating the ability to recall change patterns in small code fragments modified using similar code changes. Second, we investigated the performance on open-source projects. Mined change patterns did correspond with high-level program transformations.

The contribution of this thesis is twofold. First, a general-purpose method and tool for finding previously unknown automatable change patterns from distilled changes is proposed. Second, the impact of change generalization on change pattern mining is investigated and evaluated.

Samenvatting

Gedurende de evolutie van software worden vaak gelijkaardige doch niet noodzakelijk identieke codewijzigingen toegepast op diverse codelocaties, resulterend in frequente codewijzigingspatronen. Het manueel zoeken van locaties die aanpassing vereisen en het toepassen van de nodige wijzigingen is vervelend, omslachtig en foutgevoelig. Hoewel gereedschappen bestaan voor het automatiseren van specifieke, welgekende patronen zoals refactorings, kunnen ontwikkelaars voordeel halen uit de automatisatie van nog ongekende codewijzigingspatronen. Voor zover wij weten bestaat geen algemeen inzetbaar gereedschap voor het vinden en automatiseren van nog ongekende codewijzigingspatronen uit gedistilleerde wijzigingen.

In deze thesis wordt het probleem van het zoeken van eerder ongekende automatiseerbare codeveranderingspatronen in de geschiedenis van softwareprojecten geformuleerd als een datamining-probleem. Meer concreet pasten we een algoritme voor het graven van gesloten frequente itemverzamelingen toe, werkend op generalisaties van wijzigingen gedistilleerd uit opeenvolgende momentopnames van versiebeheersystemen.

Als implementatie van de voorgestelde aanpak schreven we een Eclipse-plugin die graaft naar automatiseerbase Java-codewijzigingspatronen in opeenvolgende git momentopnames. De implementatie is vervolgens in staat codelocaties voor te stellen die getransformeerd kunnen worden overeenkomstig gegraven codewijzingspatronen. We hebben de implementatie geëvalueerd door ten eerste het onderzoeken van het vermogen codewijzingspatronen te herkennen in relatief kleine codefragmenten waarop gelijkaardige codewijzigingen zijn toegepast. Ten tweede onderzochten we de performantie op open-source projecten. Gegraven codewijzingspatronen correspondeerden met high-level programmatransformaties.

De bijdrage van dit proefschrift is tweeledig. Ten eerste wordt een algemeen inzetbare methode en gereedschap voor het zoeken van nog ongekende automatiseerbare codewijzigingspatronen voorgesteld. Ten tweede wordt de impact van generalizatie op het graven van codewijzigingspatronen onderzocht en geëvalueerd.

Acknowledgements

This thesis would not have existed without the help and support of many people. I would like to express my sincere gratitude to all direct and indirect contributors of this thesis.

First, and foremost, I offer my enduring gratitude to Prof. dr. Coen De Roover for promoting my thesis. I owe particular thanks to Prof. dr. Coen De Roover and ir. Reinout Stevens for their advisory role during the accomplishment of this work. Those weekly meetings involving constructive and valuable feedback, guidance, and the answers to all my questions provided me the necessary insights to finish this work succesfully.

Besides my promotor and advisors, a big thanks goes out to the members of the Software Language Lab and especially all CAMP members for their insightful comments and hard questions during the half-yearly thesis presentations. In particular, I want to thank dr. Tim Molderez for introducing me to Ekeko/X.

Furthermore, I would like to thank my friends for their never-ending support and contribution to my mental health. Especially, I want to mention Jorgen Blommaert, Wout De Rooms, Stijn Van Asbroeck and Kristof Van Miegem.

Last but not the least, special thanks are owed to my parents Hans De Meyer and Carina Van Gendt, my brother Jannik De Meyer, my grandparents, Patrick Schellekens and Marian Hemerijckx who all have supported me throughout my numerous years of education, and this morally, financially and gastronomically.

Contents

List of Figures

List of Tables

List of Code Listings

1 Introduction

1.1. Context

Software evolution, the process which includes the initial development of software as well as its maintenance afterwards, implies performing changes to source code. Similar nonidentical code changes are often applied to multiple code locations. For example, whenever a method receives a new parameter, all invocations of this method should receive an additional argument. We will call such code changes *systematic-repetitive*.

1.1.1. Systematic-Repetitive Nature of Changes

The main reason behind this systematic-repetitive nature of changes is software reuse, which Krueger (1992) defines as “the process of creating software systems from existing software rather than building software systems from scratch”, and which includes the reuse of ideas, code and other artifacts. Especially code reuse results in similar code and thus similar code changes. Code reuse can manifest itself in different ways, including cross-cutting concerns, program forking, the use of common algorithms or programming idioms and the shared use of frameworks and/or libraries.

Cross-cutting concerns are design decisions falling outside the natural units of abstraction, such as logging, caching and transactions. Cross-cutting concerns are scattered across the source code (Kiczales et al., 1997), introducing code duplication. Related design decision modifications require similar but sometimes nonidentical changes to all occurrences throughout the entire code base. Although aspect-oriented programming (AOP) addresses the modularization of cross-cutting concerns by means of aspects, AOP is not widely adopted. Munoz, Baudry, Delamare and Le Traon (2009) found that less than 1% of all Java projects created on SourceForge[1], a service provider of online project hosting, between 2001 and 2008 integrate aspects.

Project forking, the process of developing a variant of a product by starting of a copy of the product’s source code, introduces code duplication which may result in systematic-repetitive changes between different software products. For example, bug fixes and

[1]http://sourceforge.net/

new features may have to be ported to other product variants. Ray and M. Kim (2012) studied cross-system porting in the products of the BSD product family (e.g., FreeBSD, NetBSD and OpenBSD), finding that cross-system porting requires significant effort.

But cross-cutting concerns and project forking are not the only form of code reuse resulting in code duplication, also known as *code cloning*. In fact, code reuse can be as simple as copy-pasting existing code segments (e.g., implementations of algorithms), possibly modifying them along the way. A substantial amount of cloned code is present in many code bases (Ducasse, Rieger & Demeyer, 1999). Krinke (2007) showed that clones often have more non-common changes than systematic-repetitive changes. However fixing bugs or improving performance still often involves systematic-repetitive changes in all clones.

Finally, frequently used functionality can be collected in *libraries* or *frameworks*. As libraries and frameworks evolve to accommodate new feature requests, so do their Application Programming Interfaces (APIs). Despite all efforts to limit API version incompatibilities, API evolution often requires migration of all applications to the new API. This once again results in systematic-repetitive changes.

1.1.2. Change Patterns

Code changes that are systematic-repetitive form *change patterns*. Change patterns which do not change the program's behavior are called refactorings, in contrast to semantics-modifying change patterns.

Fowler (1999) defines a *refactoring* as "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior". The process of refactoring as an answer to the maintenance problem related with code aging was introduced by Griswold (1991). Opdyke (1992) presented a set of refactorings for object-oriented programs. All refactorings are program transformations, ensuring behavior preservation whenever their preconditions are met. Note that although the general definition of refactoring includes all *semantics-preserving* program transformations, often only the well-known fixed set of transformations are seen as refactorings.

Unlike refactorings, change patterns can also be *semantics-modifying*. For example, bug fixes are often recurring (Nguyen, Nguyen, Pham, Al-Kofahi and Nguyen (2010), S. Kim, Pan and Whitehead (2006)), i.e. similar bug fixes are often applied at multiple code locations within one or more software versions.

1.1.3. Retrieving Changes

Change data can be retrieved in two ways: change logging and change distilling.

The first method involves using a *change logger*, reporting changes as they are performed by the developer in the Integrated Development Environment (IDE). A second option of retrieving change data, called *change distilling*, involves inferring fine-grained changes algorithmically based on two code snapshots.

The latter approach of change distilling goes well with the best practice of recording changes to source code and other artifacts in a Version Control System (VCS). Developers alter source code files and commit after task completion. As such, VCSs track and store the history of files in a software project at a commit granularity, i.e. for each commit they provide two source code snapshots: one containing the original code and one containing the resulting code. The use of VCS during software development and maintenance is common for closed-source as well as open-source projects, making the use of VCS data as primary source of code evolution convenient.

1.1.4. Finding and Automating Previously Unknown Change Patterns

Many approaches to detect and automate well-known change patterns such as the set of refactorings discussed by Fowler (1999) exist. However, learning and automating *previously unknown change patterns* benefits developers and scientists (Negara, Codoban, Dig & Johnson, 2014). Indeed, as manually editing code according to change patterns is laborious and error prone, scientists can implement tools to automate this transformation of code.

To the best of our knowledge, no general-purpose tool for finding and automating previously unknown change patterns from distilled changes exists.

1.2. Problem Statement

In this thesis, we formulate the problem of finding previously unknown automatable change patterns in the history of software projects as a data mining problem. More concretely, we apply the frequent itemset mining algorithm, operating on the changes between two successive VCS snapshots, looking for change patterns in open-source software repositories provided by GitHub.

- *Defining change patterns.* In the context of this work, change patterns should be automatable, i.e. it should be possible to use the output of the mining algorithm as input of a program transformation tool.
- *Reconciling change patterns and standard data mining techniques.* Standard data mining techniques mine frequent substructures called patterns (e.g. subitemsets, subsequences, episodes, substrings, subtrees) inside one or more larger structures. This requires a way to generate all candidate patterns as well as a method for quantifying candidate patterns in terms of interestingness. A way for mapping code changes to mineable structures is necessary.

- *Effectiveness of change mining.* Since all changes are distinct, a form of generalization based on certain change commonalities is needed in order to allow the detection of change patterns. With potentially many changes applied between successive VCS snapshots, naive generalization approaches quickly result in many uninteresting change patterns. The amount of required commonalities determines the amount of mined patterns but also the information mined patterns provide.

- *Limitations of Version Control Systems.* A VCS provides coarse-grained data at commit granularity, i.e. for each commit it provides two source code snapshots: one containing the original code and one containing the resulting code. Negara, Vakilian, Chen, Johnson and Dig (2012) concluded that data obtained from VCS is incomplete (changes may be invisible because changed code can be rechanged multiple times within a single commit) as well as imprecise (within a single commit, multiple unrelated changes may be performed to the same source code entity). For example, when considering a software history, semantics-preserving refactorings are often interleaved with semantics-modifying changes aiming to add functionality or fix bugs, and this within a single VCS commit (Murphy-Hill, Chris & Black, 2009). Also, change distilling from VCS data does not give information about the time-dimension of performed changes.

1.3. Brief Overview of the Approach

In this section we briefly introduce our approach to mining VCS data for automatable change patterns. The phases of our sequential approach are depicted in Figure 1.1.

We start by extracting two versions of a modified source code file from the software history contained within a VCS. We then extract fine-grained changes made between these two versions by a differencing process called change distilling.

To allow the use of a data mining technique called frequent itemset mining, preprocessing of distilled changes is necessary: a preprocessing step groups and generalizes the changes. During grouping changes are classified in disjoint sets by the subtree in which they occur, e.g. we can group changes at by the method in which they occur. Generalization is about considering certain changes equivalent if and only if they share certain commonalities, e.g. considering two literal values 3 and 4 equivalent because both are numeric literals.

After preprocessing, a frequent itemset mining algorithm mines automatable change patterns. These patterns can then serve as input for a program transformation tool, allowing the automatic transformation of source code according to mined patterns.

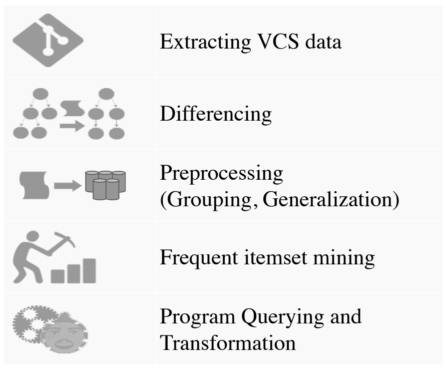

Figure 1.1: Overview of the Approach

1.4. Roadmap

This dissertation is organized as followed.

Chapter 2 introduces our approach to mining VCS data for automatable change patterns. This introduction starts by a general outline of the sequential approach. Then, each phase in the approach is discussed at a high level. Concretely, we explain how changes are represented and how they are collected from VCS repositories by a process called change distilling. Then, we discuss frequent itemset mining as a mining method matching the inherent characteristics of distilled changes and change patterns. Next, we focus on how to apply frequent itemset mining to VCS data by preprocessing distilled changes in the form of grouping and generalization. We also mention a post-processing of change patterns, which allows automatic identification of new source code segments which can be transformed according to the corresponding patterns.

Chapter 3 focusses on the first two phases of our approach, the detection of source code changes performed during software evolution. First, we focus on the differences between change logging, capturing changes directly as they are performed, and change inference from VCS snapshots. Then, we focus on the latter approach and discuss the general problem of change detection. Next, we apply these principles to source code, which also allows us to come up with a representation of source code changes. Finally we discuss ChangeNodes, the change distiller used in this work, in greater detail.

In Chapter 4 we discuss the selection of a mining algorithm suitable for mining change patterns from distilled changes. First, we discuss the inherent characteristics of distilled changes. We then give an overview of a representative subset of the field of data mining (i.e. frequent substring mining, frequent episode mining, frequent itemset mining and frequent sequence mining), for each data mining approach we evaluate input, output and assumptions in terms of distilled change characteristics. Finally, we motivate the use of frequent itemset mining as the fourth phase of our approach for mining VCS data for previously unknown change patterns.

In Chapter 5 we investigate how to apply frequent itemset mining to a sequence of changes reported by a change distiller, i.e. the process of building a mineable transaction database from a sequence of distilled changes. For reasons that will become clear later on, this process consists of two steps: grouping and generalization. The last step, generalization, is driven by an equivalence relation defined over changes, considering certain changes equivalent if and only if they share certain commonalities. The commonalities required by the equivalence relation not only determines which patterns are found but also influences what information change patterns yield. For this reason, we finish the chapter by studying this equivalence relation more thoroughly. We discuss the change operation equality, change subject equality and context equality as constituents of the equivalence relation and discuss the relation's requirements to yield valid results. We give examples and discuss their impact on the meaning of mined change patterns.

As mentioned, developers can use tool support to identify new source code segments to which change patterns can be applied and to apply change patterns automatically to these source code segments. Chapter 6 focusses on tool support to meet the first objective, i.e. finding new code locations which can be transformed according to mined change patterns. After a general discussion, we give a template-based approach for specifying change pattern subject characteristics, allowing querying other source code for code segments matching these characteristics.

We evaluate our approach in Chapter 7. In this evaluation, we answer two research questions. We first investigate whether the algorithm can recall change patterns from small code fragments with known systematic-repetitive changes. We then focus on the performance of mining useful previously unknown patterns from open source projects.

Finally, we list related work in Chapter 8 and conclude in Chapter 9. At the end of the conclusion we state our contribution and provide an overview of possible improvements and extensions as future work.

2 Overview of the Approach

In this chapter we introduce our approach to mining VCS data for frequent change patterns.

The phases of our sequential approach are depicted in Figure 1.1 of Chapter 1. We start by extracting two versions of a modified fragment of source code from the software history contained within a Version Control System. We then extract changes made between these two versions by a process called change distilling. Afterwards, a preprocessing step groups and generalizes the changes, which allows a mining algorithm to mine change patterns. As an example of tool support, a program querying solution allows the identification of candidate change pattern instances in other source code.

The remainder of this chapter is organized as follows. We first discuss all phases in our sequential approach in greater detail. Section 2.1 discusses how to collect changes from a Version Control System. Section 2.2 introduces the data mining approach used in this work, after which Section 2.3 covers the application of this approach to changes distilled from VCS data. Section 2.4 presents a program querying solution allowing the identification of candidate change pattern instances in other new code. After discussing all phases in the approach, we define the notion of a change pattern in Section 2.5. and give implementation details in Section 2.6.

2.1. Representing and Retrieving Changes

During software evolution, it is a best practice to record changes to source code and other artifacts in a VCS. Developers alter source code files and commit to the VCS after task completion. As such, VCSs track and store the history of files in a software project at a commit granularity, i.e. for each commit they provides two source code snapshots: one containing the original code and one containing the resulting code.

A differencing tool for tree-structured information calculates an edit script which lists the operations to transform a given tree into another given tree. Such an edit script is a sequence of insertions, updates, removals and moves. Performing all operations in the sequence in order transforms the former tree into the latter.

To represent and find performed source code changes, consider the abstract syntax trees (ASTs) corresponding with the original source code (before committing the changes) and the resulting source code after code edits were committed. The former we call the source AST, the latter is called the target AST. We can now use a differencing tool for tree-structured information to report source code changes made between the source AST and the target AST, a process which is known as *change distilling*.

Consecutively, source code changes are represented by edit script operations (inserts, updates, removals and moves of AST nodes).

Chapter 3 covers the representation and the process of retrieving changes in greater detail.

2.2. Frequent Pattern Mining

Frequent pattern mining is the subdiscipline in data mining involving finding frequent patterns in data. Such frequent patterns are patterns appearing in a database no less than a user-specified threshold. Mining algorithms exist for a large variety of data structures, including sets, strings, sequences and trees and graphs.

To determine the kind of patterns to mine, consider two inherent characteristics of distilled code changes and change patterns. First, unlike change logging, involving capturing changes as they are performed by the developer by use of IDE functionality, change distilling is an algorithmic process reconstructing fine-grained changes from two code snapshots. The resulting sequence of changes does not necessarily correspond to the sequence of changes as performed by the developer. In fact *order of distilled changes is only partially defined*. Second, within a single commit the developer may have performed *changes interleave changes forming a pattern*. For example, he might have modified two methods in a systematic-repetitive way, but he might as well have performed other unrelated modifications to these same methods, and this within a single commit.

As we will discuss in more detail in Chapter 4, sets, being unordered collections, match the inherent characteristics of distilled code changes. Frequent itemset mining is a frequent pattern mining approach mining frequent sets. It was introduced in the context of market basket analysis for finding products that are frequently bought together. It is built around two kinds of objects: items and baskets. The set of possible items (e.g. *Bread*, *Milk*) is fixed. Each of the possibly many baskets contains a subset of all items. The aim is to find sets of items with a high support, i.e. items that appear frequently together in baskets.

In formal terminology, sets of items are called itemsets: baskets (subsets of the set of items) as well as mined patterns (sets of items with high support) are itemsets. The bag of baskets taken as input is called the transaction database, baskets are called transactions.

Chapter 4 motivates the usage of frequent itemset mining and formalizes its concepts.

2.3. Preprocessing

Frequent itemset mining requires a transaction database, i.e. a bag of itemsets in which frequent patterns are mined. However, change distilling reports a sequence of changes represented by concrete AST operations. For this reason, using frequent itemset mining for finding patterns in distilled changes requires an answer to two questions: 1) what is our set of possible items? 2) how do we form a transaction database given a single sequence of changes? To address both questions, we introduce a preprocessing of changes, a two-step process involving grouping and generalization.

A first preprocessing step involves grouping changes. Since we can not rely on change order within the sequence of distilled changes, we group changes, represented by concrete AST operations, into sets by the subtree in which they occur, rooted at a specific AST type of node. For example, we can group changes by the method in which they occur.

Change grouping alone results in sets of changes, all of which are distinct. Since no two sets contain the same change, the mining algorithm can not find frequent patterns: frequent itemset mining mines sets of items that appear frequently together in multiple(!) transactions. For this reason, we convert changes into so-called change generalizations by dropping certain information. This results in similar changes being represented by the same generalization.

As discussed here, preprocessing results in the set of items being all change generalizations, and the transaction database being the bag of sets resulting from first grouping then generalizing the changes.

Preprocessing is discussed in large detail in Chapter 5.

2.4. Tool Support

Identifying change patterns by mining VCS histories mainly benefits researchers, who can learn a lot from changes made to source code. In practice, developers need tool support as manual implementation of systematic-repetitive changes to all necessary code locations is laborious, tedious and error prone.

To be concrete, developers can use tools to 1) identify new source code segments to which change patterns can be applied and 2) apply change patterns automatically to new source code (i.e. automatic code transformation).

We have investigated the first objective, i.e. the postprocessing of change patterns to find new code segments to which the pattern can be applied. This topic is discussed in Chapter 6.

2.5. Defining Change Patterns

As mentioned earlier, systematic-repetitive changes tend to form change patterns. As such, change patterns represent recurring code modifications. Before continuing, we have to refine this notion.

The mining algorithm outputs frequent itemsets, which are called change patterns in the context of this approach. Technically, a change pattern is nothing more than a set of change generalizations occurring in a number of groups (e.g. methods) above a user-specified minimal support threshold.

More important is the information delivered by a change pattern. This information is implicitly defined by the way changes are grouped and generalized: the information is the accumulation of all change generalizations occurring within the change pattern. Sufficient information should be provided for a change pattern to be automatable.

2.6. Implementational Details

The written ECLIPSE plugin is usable via the Clojure REPL. In this section we highlight important implementational considerations.

As a source of VCS data we use git, which we interface using the JGIT[1] library. We use a differencing tool called CHANGENODES[2], which operates similar to the algorithm implemented in the well-known tool CHANGEDISTILLER (Fluri, Wursch, Pinzger & Gall, 2007).

Use of candidate generation data mining algorithms does not scale: even simple code modifications quickly result in many code changes. For this reason we use an algorithm without candidate generation, called CHARM (Mohammed J. Zaki & Hsiao, 2002): we use the CHARM-implementation of SPMF[3], an open-source data mining library written in Java.

We construct EKEKO/X templates to allow querying other source code for candidate transformation subjects. EKEKO/X[4], (De Roover & Inoue, 2014) is a template-driven program transformation tool implemented on top of the meta-programming library EKEKO. It allows specifying code segment characteristics by means of code templates, after which new source code can be queried for code segments matching the specified characteristics.

To allow customization of the algorithm's behavior, a "strategy" must be specified. Such a strategy defines the way of grouping, the way of generalization and the template generation approach.

[1]https://eclipse.org/jgit
[2]https://github.com/ReinoutStevens/ChangeNodes
[3]http://www.philippe-fournier-viger.com/spmf/
[4]https://github.com/cderoove/damp.ekeko.snippets

2.7. Summary

In this chapter we have introduced our sequential approach to mining VCS data for applicable change patterns. We work in 5 successive phases.

We start by extracting two versions of a modified fragment of source code from the software history contained within a VCS. In the second phase we apply a differencer for tree-structured information on the ASTs corresponding both versions, a process which is known as change distilling. As such, we extract a sequence of fine-grained changes (represented by concrete AST operations, e.g. node insert or subtree move) made between these two versions.

In the third phase we preprocess the sequence of distilled changes, in order to render it suitable for data mining. This preprocessing involves two substeps: grouping and generalization. During grouping changes are classified in disjoint sets by the subtree in which they occur, e.g. the grouping of changes by the method in which they occur. Generalization is about considering certain changes equivalent if and only if they share certain commonalities.

After preprocessing, a frequent itemset mining algorithm mines automatable change patterns. These change patterns can then serve as input for tools. As an example of such tool support, we provide a program querying solution allowing the identification of candidate change pattern instances in new source code.

3 Representing and Retrieving Changes

This chapter focusses on representing and retrieving source code changes performed during software evolution. Section 3.1 compares two main approaches for retrieving necessary data: on the one hand source code changes can be captured as they are performed and on the other hand changes can be inferred from VCS snapshots. Afterwards, the algorithms used by the latter approach, which is used in this thesis, are discussed in greater detail. Section 3.2 focusses on the general problem of change detection, the application to source code is discussed in Section 3.3. Finally, Section 3.4 focusses on ChangeNodes, the change distilling tool used in this work.

3.1. Retrieving Changes

As stated earlier code changes are continously applied during software evolution. Two approaches are common to collect these changes.

The first approach, called *change logging*, captures changes as they are performed by the developer. For example, CodingTracker (Negara et al., 2012) is an Eclipse plugin capturing source code edits as well as other developer actions e.g., running tests and performing automated refactorings. As such, change logging provides detailed time-stamped information of performed changes. Unfortunately, due to change logging not being a common practice and the potential privacy concerns related to its use, logged data is not readily available.

An alternative approach called *change distilling* uses data provided by a *VCS* to infer source code changes. During software evolution, it is a best practice to record changes to source code and other artifacts in a VCS. Popular VCSs such as git[1], Apache Subversion (SVN)[2], Concurrent Version System (CVS)[3] or Mercurial[4] record file changes and as

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

such, VCSs track and store the history of files in a software project. Changes to files are called *revisions*. Developers are said *to commit* sets of revisions of files to the VCS, *a commit* is a set of committed file revisions. Typically, developers commit changes after completing a certain task. The VCS provides coarse-grained data at commit granularity, i.e. for each commit it provides two source code snapshots: one containing the original code and one containing the resulting code. Using those two snapshots of the source code, we can infer source code changes in the form of AST edits by a process called change distilling by tree differencing. Change distilling is an algorithmic process based on coarse-grained VCS data. Consecutively, distilled changes do not necessarily correspond with actual changes performed by developers. As a consequence, Negara et al. (2012) concluded that change distilling is incomplete (changes may be invisible because changed code can be rechanged multiple times within a single commit) as well as imprecise (within a single commit, mutiple unrelated changes may be performed to the same source code entity).

The use of VCS during software development and maintenance is common for closed-source as well as open-source projects. For instance, the usage of Github, a service provider of online git project hosting, increased exponentially since it was founded in 2008 with around 45000 public repositories in February 2009, 1 million repositories in 2010 and 10 million repositories in 2013[5]. Because of the amount of information available in VCS repositories we follow the latter approach of change distilling, i.e. we use VCS data as primary source of code evolution.

3.2. Change Detection and Representation

Change detection approaches can be classified according to the type of data on which they operate. In this section we focus on early flat-file based approaches and later approaches for tree-structured data.

3.2.1. Flat-File Change Detection

Wagner and Fischer (1974) defines the *string-to-string correction problem* as the problem of determining the distance between two strings as measured by the minimum cost *edit script* (or *delta*), i.e. the sequence of edit operations needed to change the one input string into the other input string. In this definition, an *edit operation* can be either the insertion, deletion or update of a single symbol in the input string. An algorithm is presented which solves this problem in time proportional to the product of the lengths of the two strings, i.e. $O(nm)$ with n and m the length of the input strings.

Over time several improvements and alternative algorithms for the string-to-string correction problem and its variants were found.

Especially, E. W. Myers (1986) presents an algorithm for a variant of the string-to-string correction problem, with an $O(ND)$ time complexity where N is equal to the

[5]https://github.com/blog/1724-10-million-repositories

sum of the lengths of the input strings and D the size of the minimum edit script. The presented shortest edit script algorithm, which is implemented in the GNU diff utility[6], is limited to symbol insertions and deletions: it does not consider other edit script commands such as updates and moves. Thus, an edit script for input symbol sequences A and B is a set of insertions and deletions that transform A into B.

3.2.2. Finding Changes in Structured Documents

In a hierarchical context, the notion of edit script can be redefined as a sequence of edit operations that transform one tree into another. The *change detection problem* can then be defined as the problem of finding an edit script given two trees.

Chawathe, Rajaraman, Garcia-Molina and Widom (1996) studies the problem of detecting and representing changes to tree-structured information, in contrast to the discussed text-based approaches. It does not limit edit operations to node inserts and node deletes: it also supports node update and subtree movement operations. The presented algorithm first finds a matching between nodes of both trees. Then, given the performed matching, an edit script transforming the first tree into the other is generated.

3.3. Program Differencing

In this section we will discuss the problem of *program differencing*, i.e. finding an edit script between two versions of a piece of source code. Although textual deltas can operate on source-code representations of programs, text-based change detection for program differencing surely has its limitations; mining source code changes often requires one to operate on more meaningful representations of software changes. The fact that source code can be represented by tries (i.e. its abstract syntax tree) enables the usage of change detection approaches for tree-structured data. Such a tool then reports a sequence of AST edits (AST node insertions, AST node removals, AST node updates and AST subtree movement operations) transforming one AST into another. Source code changes thus can be represented by these AST edits.

Fluri et al. (2007) discusses an approach for fine-grained source code change extraction. The approach is based on the algorithm of Chawathe et al. (1996), but the matching phase is improved to extract changes in source code. The discussed algorithm is implemented in ChangeDistiller[7].

ChangeNodes[8] (Stevens, 2015), part of the history querying tool QwalKeko developed at the Vrije Universiteit Brussel, is similar to ChangeDistiller. The main difference between ChangeDistiller and ChangeNodes lies in the used AST representation: ChangeDistiller uses its own language-agnostic AST representation while

[6]http://www.gnu.org/software/diffutils/

[7]https://bitbucket.org/sealuzh/tools-changedistiller/wiki/Home

[8]https://github.com/ReinoutStevens/ChangeNodes, https://github.com/ReinoutStevens/damp.qwalkeko

ChangeNodes uses language-aware JDT nodes (being instances of `org.eclipse.jdt-.core.dom.ASTNode`), which the Eclipse JDT provides.

A novel alternative for change distilling is the Gumtree algorithm (Falleri, Morandat, Blanc, Martinez & Montperrus, 2014). Gumtree also works at the AST level, but unlike change distilling, it works at a lower granuarity: instead of representing a statement as a single node with a significant amount of text it encodes a statement as raw ASTs, as multiple nodes. Edit scripts in Gumtree thus are more fine-grained. Furthermore, the essential premise at the heart of the Gumtree algorithm is to compute en edit-script script reflecting the developers intent.

Unlike Gumtree, ChangeNodes integrates with the programming querying tool Ekeko, whose facilities are useful for this work (e.g. change generalization, . . .).

3.4. ChangeNodes

ChangeNodes is a Clojure library whose main interface expects an original AST (heceforth called the source AST) and the AST after modifications were performed (henceforth called the target AST) and returns a sequence of changes represented by Clojure records. In this section, the output of ChangeNodes is discussed in greater detail.

3.4.1. Change Definitions

The field `:operation` represents the type of change, and is one of `:delete` (the removal of a node from the AST), `:update` (the replacement of a node in the AST), `:move` (the movement of a node to a different location) or `:insert` (the insertion of a node in the AST).

`delete(original, property, index)`

A node X and its subtree are deleted from the AST. The field `:original` refers to the deleted node X in the source AST. Furthermore, a Clojure keyword identifying the `StructuralPropertyDescriptor` of the structural property whose value is the deleted AST node X is stored in the field `:property`. If this structural property descriptor is a `ChildListPropertyDescriptor`, the field `:index` contains the index of the deleted node within the corresponding list, otherwise the field's value is `nil`.

`update(original, property, copy)`

Update operations are emitted whenever the value A of a simple structural property of a node N in the source AST is replaced by another value B. The field `:original` refers to the replaced value A in the source AST, and the field `:property` contains a Clojure keyword identifying the corresponding structural simple property descriptor of node N. The field `:copy` refers to the value B in the target AST.

`insert(original, property, index, copy)`

A minimal representation of a node X is inserted in a node Y in the AST. The field `:copy` refers to the inserted node X in the target AST. The field `:original` refers to the node Y in the source AST in which the insertion took place, the field `:property` contains a Clojure keyword identifying the corresponding structural simple property descriptor of this node. If this structural property descriptor is a `ChildListProperty-Descriptor`, the field `:index` contains the index specifying the position of the insertion in the list, otherwise the field's value is `nil`.

First of all note that by minimal representation we mean the insertion of a node and the value of all its mandatory properties. Insertions of subtrees with non-mandatory nodes will generate several changes of operation `:insert`. As a consequence, such changes will not have field `:original` set: these insertions have no corresponding node X in the source AST.

Second, an insertion may overwrite an already existing node present at Y's structural property identified by field `:property`.

`move(original, rightParent, property, index, copy)`

A minimal representation of a node X is moved to a different location in a node Y. The field `:original` refers to the node X in the source AST. The field `:rightParent` refers to the node Y in the target AST in which the insertion took place, the field `:property` contains a Clojure keyword identifying the corresponding structural simple property descriptor of this node. If this structural property descriptor is a `ChildListProperty-Descriptor`, the field `:index` contains the index specifying the position of the moved node X in the list, otherwise the field's value is `nil`.

Like insert operations, change operations only apply to a minimal representation of the node. Moving a larger subtree may produce multiple move operations.

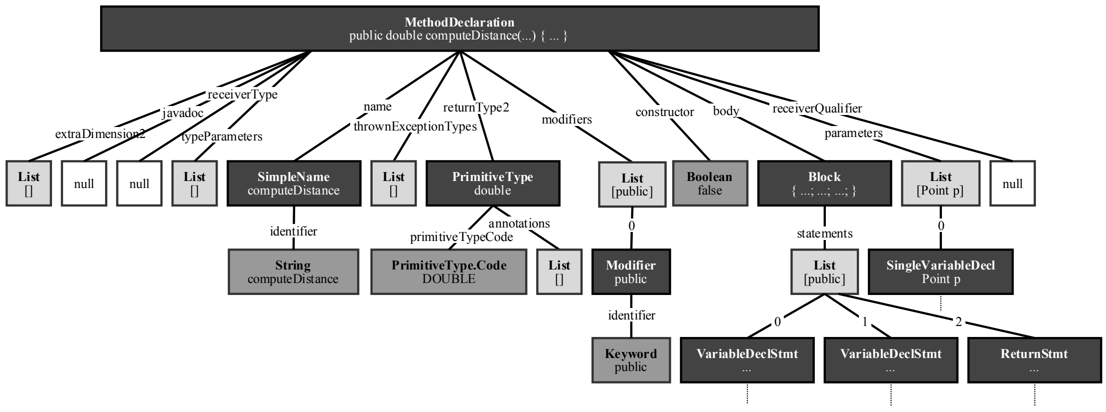

Figure 3.1: AST of the `MethodDeclaration` of `computeDistance()` of Example "Return If Current Object is equal to Passed Argument"

3.4.2. Illustrative Example

In order to demonstrate the output of ChangeNodes, consider Code Listing 3.1 showing a fragment of source code in which highlighted lines were added in an attempt to improve performance in the case that the current `Point` object is equal to the passed argument.

Code Listing 3.1: Example: "Return If Current Object is equal to Passed Argument"

```
public class Point {

  private int x;
  private int y;

  public double computeDistance(Point p) {
+ if (this.equals(p)) return 0;
    double deltaX = this.computeDeltaX(p);
    double deltaY = this.computeDeltaY(p);
    return Math.sqrt(Math.pow(deltaX, 2) + Math.pow(deltaY, 2));
  }

  public double computeDirection(Point point) {
+ if (this.equals(point)) return 0;
    double deltaX = this.computeDeltaX(point);
    double deltaY = this.computeDeltaY(point);
    return Math.atan2(deltaY, deltaX) * 180 / Math.PI;
  }

  ...

}
```

Although only 2 statements were added, change distilling using ChangeNodes gives 8 changes: an insert operation does not insert an entire subtree but only a minimal representation of an AST node. Consecutively, insertion of larger subtrees do produce multiple changes. Table 3.1 lists the reported change sequence. All changes are node insertions. The column Copy refers to the inserted node in the target AST. The column Original refers to the node in the source AST in which the insertion occurred and the column Property identifies the corresponding structural property of this node. If this structural property's value is a list, the column `Index` contains the position of the inserted node. As such, change 1 should be read as: "insert a minimal representation of the statement `if (this.equals(p)) return 0;` in the block's structural property statements at position 0". Immediatly note that insertions within a node inserted by another change do not have an original node: no corresponding node exist in the source AST.

We now discuss the change in `computeDistance()` of this example at the level of the abstract syntax tree itself, and in doing so further illustrate the notion of minimal representation of a node.

Figure 3.1 partially shows the source AST of this method, i.e. the subtree rooted at the `MethodInvocation` node before the `IfStatement` was inserted. The `MethodDeclaration`

contains a structural property `body`, whose value is a `Block`. This `Block` contains a structural property `statements`, whose value is a list with 3 statements. The `IfStatement` as shown in Figure 3.2 is inserted inside this list, at the first position (this is, before the first statement).
Please note that in this figure full lines are used for mandatory structural properties, dotted lines represent optional structural properties (`ChildListProperties` are considered mandatory since their value can be the empty list, items inside the list are considered optional).
According to the discussion above, the minimal representation of a node consists of the node itself and all its straight-line descendants. The minimal representation of the `IfStatement` does not include the NumberLiteral, the `SimpleName p` and the `ThisExpression` nodes and their descendants. For this reason, 3 additional insertions are necessary to insert these one by one, which corresponds with outputted changes as listed in Table 3.1.

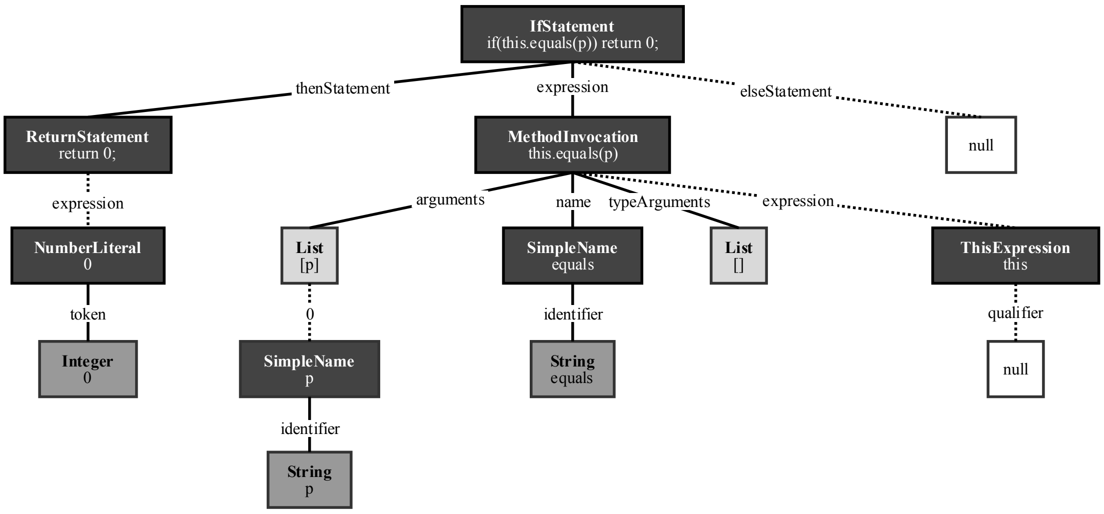

Figure 3.2: AST of the `IfStatement` of Example "Return If Current Object is equal to Passed Argument"

3.5. Summary

This chapter discussed the representation and retrieval of source code changes performed during software evolution. The first section started by comparing two ap-

Table 3.1: Changes in Example "Return If Current Object is equal to Passed Argument"

Change	Op.	Original	Property	Idx	Copy
1	insert	#<Block { ... }>	statements	0	#<IfStatement if(this.equals(p)) return 0;>
2	insert	#<Block { ... }>	statements	0	#<IfStatement if(this.equals(point)) return 0;>
3	insert	-	expression	-	#<ThisExpression this>
4	insert	-	arguments	0	#<SimpleName p>
5	insert	-	expression	-	#<NumberLiteral 0>
6	insert	-	expression	-	#<ThisExpression this>
7	insert	-	arguments	0	#<SimpleName point>
8	insert	-	expression	-	#<NumberLiteral 0>

proaches for data retrieval: change logging and change distilling. Despite distilling fine-grained changes from VCS data being an algorithmic process shown to be inprecise and incomplete, it still is a convenient source for code evolution. Indeed, unlike logged data which is not readily available, VCS usage is a common best-practice during software development.

In Section 3.2 we focussed on the general problem of change detection. Given two datastructures, the problem is to find an edit script of operations that, when performed in order, transforms the first datastructure into the other. Early approaches are text-based, in contrast to later approaches for tree-structured data. Section 3.3 covers program differencing, which is an application of these change detection approaches to source code.

In the last section we focussed on ChangeNodes, the program differencing tool used in this work. As such, given two ASTs, ChangeNodes outputs an edit script, which is a sequence of AST changes (inserts, updates, deletes and moves) transforming the first AST into the other.

4 Pattern Mining

Frequent patterns are patterns appearing in a database no less than a user-specified threshold. Frequent patterns include itemsets, subsequences and other substructures (subtrees, subgraphs,. . .).

This chapter discusses frequent pattern mining approaches for a representative variety of patterns we have considered for mining VCS data for unknown change patterns. Furthermore, we motivate the use of frequent itemset mining for mining VCS data for unknown change patterns.

To this end, we first discuss the characteristics of distilled changes in Section 4.1. The next 4 sections give an overview of a representative subset of the field of data mining, and evaluate their input, output and assumptions in terms of these characteristics: in Section 4.2 we will focus on frequent itemset mining and association rule learning, Section 4.3 covers frequent substring mining, Section 4.4 discusses frequent episode mining and Section 4.5 focusses on sequential pattern mining. Finally, in Section 4.6 we give an overview of seen data mining approaches and conclude with the use of frequent itemset mining for mining VCS data for change patterns.

4.1. Problem Characteristics

In this section we look at the characteristics of our problem domain, i.e. changes represented by AST nodes, distilled from successive VCS snapshots.

A first aspect to consider for mining approach selection is the *order of change*s in the distilled change sequence. When using a change logger, such as CodingTracker (Negara et al., 2012), changes are reported in the order in which they are performed by the developer. The version control system on the other hand only provides coarse-grained data at a commit-level granularity and as such it only allows us to retrieve an AST before modifications and one after the commit was made.

Given two abstract syntax trees (a source AST and a target AST), the change distiller reports a sequence of changes that, when applied in order, transform the source AST into the target AST. During this process, changes are applied sequentially, and each

Table 4.1: Changes in method `computeDistance()` of Example “Return If Current Object is equal to Passed Argument”

Change	Op.	Original	Property	Idx	Copy
A	insert	#<Block { ... }>	statements	0	#<IfStatement if(this.equals(p)) return 0;>
B	insert	-	expression	-	#<ThisExpression this>
C	insert	-	arguments	0	#<SimpleName p>
D	insert	-	expression	-	#<NumberLiteral 0>

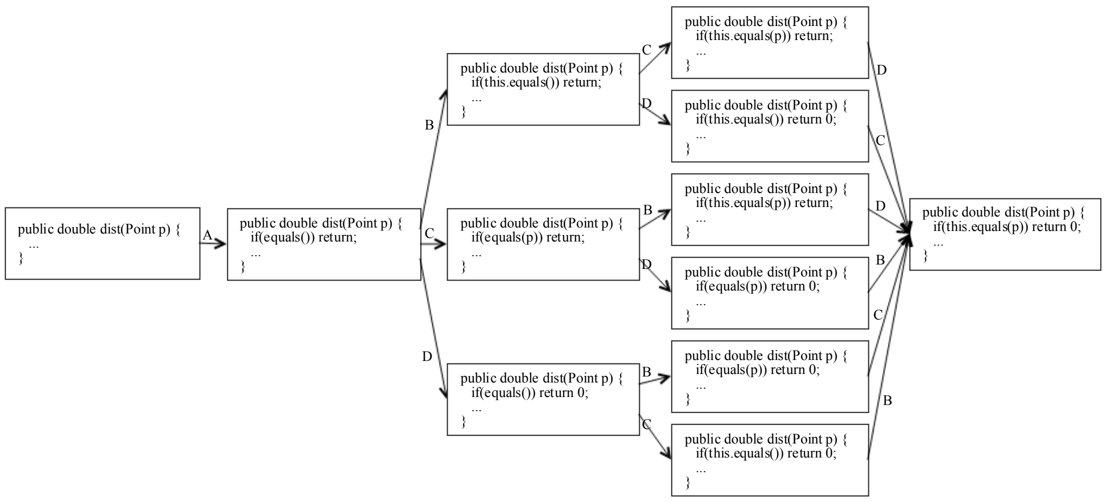
public double dist(Point p) { ... }
A
public double dist(Point p) { if(equals()) return; ... }
B
C
D
public double dist(Point p) { if(this.equals()) return; ... }
public double dist(Point p) { if(equals(p)) return; ... }
public double dist(Point p) { if(equals()) return 0; ... }
public double dist(Point p) { if(this.equals(p)) return; ... }
public double dist(Point p) { if(this.equals()) return 0; ... }
public double dist(Point p) { if(this.equals(p)) return; ... }
public double dist(Point p) { if(equals(p)) return 0; ... }
public double dist(Point p) { if(equals(p)) return 0; ... }
public double dist(Point p) { if(this.equals()) return 0; ... }
public double dist(Point p) { if(this.equals(p)) return 0; ... }

Figure 4.1: Possible Change Orderings of `computeDistance()` of Example “Return If Current Object is equal to Passed Argument”, Represented by a Graph

time a change is applied the current AST is transformed into a new AST. After all changes are applied, the resulting AST will be equal to the target AST.

However, the order of the changes within this sequence of distilled changes is often only one of many possible orderings of those changes. In fact, we can build a rooted directed graph in which each path corresponds with a possible ordering of changes. The root of this graph is a node representing the source AST, each node represents an AST which can be obtained by applying changes. Two nodes are connected by an arc if and only if there exists a change transforming the AST represented by the head of the arc AST into the AST represented by the tail of the arc. All paths through the graph eventually reach a node representing the target AST, a node without any successors.

As an illustration, consider the modification of the method `computeDistance()` as given in Code Listing 3.1 of Chapter 3: the insertion of an if-statement to return if the current object is equal to the passed argument. Table 4.1 lists the distilled change sequence, as provided by ChangeNodes. Figure 4.1 shows the resulting graph. As can be seen, the change sequence $ABCD$ corresponds to only one of the available paths from source AST to target AST within this graph. The change distiller could as well have reported another change sequence, such as $ABDC$ or $ADCB$.

Another aspect to consider is the possibility of *unrelated changes interleaving changes forming a change pattern.* For example, a developer might have changed two methods in a systematic-repetitive way, but he may have performed other unrelated modifications to one of both methods. Depending on the position where these other changes occurred, these change may interleave changes in the distilled change sequence forming a pattern.

4.2. Frequent Itemset Mining

Frequent itemset mining was introduced in the context of market basket analysis for finding products that are frequently bought together. It is built around two kinds of objects: items and baskets. The number of items (e.g. *Bread*, *Milk*) is fixed. Each of the possibly many baskets contains a subset of all items. The aim is to find items with a high support, i.e. items that appear frequently together in baskets.

4.2.1. Formal Problem Statement

More formally, let the *item base* $I = \{i_1, ..., i_m\}$ be a set of items. Any subset T of I is called an *itemset*. A *transaction* t over an item base I is a tuple $\langle tid, T\rangle$ where tid is a unique transaction identifier and T is an itemset. A transaction $t = \langle tid, T\rangle$ *contains* an itemset S if $S \subseteq T$. A *transaction database* D is a set of transactions. The *support* of an itemset in D is the percentage s of transactions in D that contain that itemset. A *frequent itemset* is an itemset whose support is above a certain threshold c. The problem of *frequent itemset mining* can now be defined as: given a transaction database and a threshold, identify all frequent itemsets.

For demonstrative purposes, consider the items *Bread*, *Milk*, *Butter* and *Bear* and the example transaction database given in Table 4.2, containing 5 transactions. Also, consider a frequency threshold of 3, i.e. an itemset has to occur in three or more transactions in order to be considered frequent. Frequent itemsets are $\emptyset$, (support 5), $\{Bread\}$ (support 3), $\{Milk\}$ (support 3), $\{Butter\}$ (support 5), $\{Bread, Butter\}$ (support 3) and $\{Milk, Butter\}$ (support 3). See also Table 4.3, listing all candidate frequent itemsets and their support.

Table 4.2: Frequent Itemset Mining Example: Transaction Database

Transaction Identifier	Itemset
1	{Bread, Butter}
2	{Beer, Milk, Butter}
3	{Bread, Milk, Butter}
4	{Beer, Butter}
5	{Bread, Milk, Butter}

A notable variant of frequent itemset mining results from contraposition of the apriori property (discussed below): all subsets of a frequent itemset are frequent. This principle brings us to the notion of *closed frequent itemsets*: an itemset is closed if it is frequent and there are no superitemsets with the same support. Solutions to the problem of closed itemset mining provide more compact results while preserving completeness.

Table 4.3: Frequent Itemset Mining Example: Itemsets

Itemset	Support	Frequent	Closed	Maximal
$\{\}$	5	✓		
$\{Bread\}$	3	✓		
$\{Milk\}$	3	✓		
$\{Butter\}$	5	✓	✓	
$\{Beer\}$	2			
$\{Bread, Milk\}$	2			
$\{Bread, Butter\}$	3	✓	✓	✓
$\{Bread, Beer\}$	0			
$\{Milk, Butter\}$	3	✓	✓	✓
$\{Milk, Beer\}$	1			
$\{Butter, Beer\}$	2			
$\{Bread, Milk, Butter\}$	2			
$\{Bread, Milk, Beer\}$	0			
$\{Milk, Butter, Beer\}$	1			
$\{Bread, Milk, Butter, Beer\}$	0			

Another variant involves mining maximal frequent itemsets. A *maximal frequent itemset* is a frequent itemset included in no other frequent itemset.

To revisit the example from table 4.2, the frequent itemset $\{Bread\}$ is not closed since there exists a super itemset $\{Bread, Butter\}$ with the same support. In fact, there are only 3 closed itemsets: $\{Butter\}$, $\{Bread, Butter\}$ and $\{Milk, Butter\}$. Similarly, the frequent itemset $\{Butter\}$ with support 5 is not a maximal frequent itemset, since it is included in the frequent itemsets $\{Bread, Butter\}$ and $\{Milk, Butter\}$, both having support 3. See also Table 4.3.

4.2.2. Association Rule Mining

Closely related to the problem of frequent itemset mining is the problem of association rule mining: association rules can be derived after mining of frequent itemsets. Let X, Y be itemsets. An *association rule* $X \to Y$ is an implication where $X \cap Y = \emptyset$. The *support* of an association rule $X \to Y$ in the transaction database is the percentage s of the transactions in the database that contain $X \cup Y$. The *confidence* of an association rule $X \to Y$ can be defined as support$(X \to Y)/$ support(X), or in other words: the percentage of transactions that contain X that also contain Y. The problem of *association rule mining* can now be defined as: identify all association rules that have support and confidence greater than a certain threshold.

As an example, the association rule $\{Milk\} \to \{Bread\}$ has a support of $2/5 = 40\%$ and a confidence of $(2/5)/(3/5) = 66\%$. This means: whenever $\{Milk\}$ is bought, there is a probability of 66% that $\{Bread\}$ is bought too, i.e. someone who bought milk is likely to buy bread. And this was the case for 40% of the transactions.

4.2.3. Approaches

Early Approaches

Agrawal, Imieliński and Swami (1993) is the pioneering work about association rule mining. It shows that association rule mining can be decomposed in identification of frequent itemsets on one hand and association rule generation on the other hand. It proposes a first algorithm for association rule mining called the AIS algorithm. Another early algorithm for association rule mining is the SETM algorithm (Houtsma & Swami, 1995). Both algorithms do not allow consequents of association rules to consist of multiple items.

Mining with Candidate Generation ("Apriori-like Algorithms")

Agrawal and Srikant (1994) proposes three new algorithms: Apriori, AprioriTid and AprioriHybrid (the latter combining the best features of Apriori and AprioriTid by using Apriori for the initial pass after which AprioriTid is used). The work observed the downward closure property, now well-known as the *apriori property*, stating that any subset of a frequent itemset must also be frequent (or in other words: no superset of an infrequent item set can be frequent). The Apriori algorithm proceeds by scanning the transactional database for the identification of frequent individual items (converted to frequent 1-itemsets). It then extends frequent 1-itemsets one at a time (a process called "candidate generation") to generate candidate frequent 2-itemsets, after which the database is scanned again to see which of these candidate frequent itemsets appear sufficiently often, and thus actually are frequent. This two-step process is repeated until no further extensions can be found.

Pasquier, Bastide, Taouil and Lakhal (1999) not only introduces the idea of closed itemset mining, it also proposes a breadth-first Apriori-like algorithm called A-Close.

Mining Without Candidate Generation

Candidate generation ("apriori-based") approaches perform a costly scan over the complete database in every step and generate many candidate itemsets which are later found to be infrequent. These two drawbacks are inherent to the so-called "candidate set generation and test"-concept of these approaches, which lead to alternative approaches without candidate generation. Algorithms for frequent itemset mining without candidate generation fall apart in two categories: those using the *horizontal database format* (in which the database is a set of tuples $\langle tid, itemset \rangle$) and those using the *vertical database format* (in which the database is a set of tuples $\langle tidset, item \rangle$ with *tidset* the set of transaction identifiers of transactions that contain the item *item*). Since transaction databases can be huge, database compression methods were developed. FP-trees are a notable representation for databases in a horizontal format, DiffSets allows compression of databases in the vertical database format.

The FP-GROWTH algorithm (Han, Pei, Yin & Mao, 2004) is a first algorithm for frequent itemset mining without candidate generation, using a horizontal database format. The idea is to construct an FP-tree out of 1-itemsets, after which this tree is recursively grown. Pei, Han and Mao (2000) discusses CLOSET, an FP-tree based mining algorithm for closed itemset mining.

M. J. Zaki (2000) too performs frequent itemset mining without candidate generation, but uses a vertical database format. The main idea of the ECLAT algorithm is to compute n-itemsets form $(n-1)$-itemsets by intersecting the tidsets, this until no more frequent itemsets can be found. Mohammed J. Zaki and Hsiao (2002) introduces CHARM, a miner for closed itemsets using the vertical data format, which works around a data structure called an itemset-tidset tree. M. J. Zaki and Gouda (2003) introduces the concept of DiffSets and gives variants of ECLAT and CHARM called DECLAT and DCHARM using this compact representation.

The more recent Wang, Han and Pei (2003) describes a systematic study on those search strategies, which lead to the development of a 'winning algorithm' called CLOSET+.

4.2.4. Discussion

Frequent itemset mining mines frequent itemsets inside a transaction database. The itemsets in the transaction database are unordered collections of distinct items, and as a result frequent itemset mining is not order-sensitive to the input. Also, mined patterns are itemsets too, and thus their elements must not be a continuous sequence.

4.3. Frequent Substring Mining

The *frequent substring mining* problem is the problem of enumerating all substrings appearing more frequently than some threshold *in a given string.* This is: the mining of continuous subsequences ("substrings") in a single sequence ("string"). Lee and De Raedt (2005) discusses a variant of the frequent substring mining problem in which substring patterns are mined from a string database being *a bag of strings.*

4.3.1. Approaches

Typically, a special kind of tries called *suffix trees* are used as main data structure in frequent substring mining. The concept of suffix trees was introduced by Weiner (1973). Improvements in tree construction followed (McCreight, 1976; Ukkonen, 1995; Farach, 1997). After a linear-time construction of a suffix tree, frequent substring mining too is a linear-time process. Unfortunately, memory footprint limits scalability. *Suffix arrays* (Manber & G. Myers, 1990) are a more recent and space-efficient alternative for suffix trees. Tsuboi, Y. (2003) discusses an algorithm similar to PREFIXSPAN (see Section 4.5), which generates an incomplete suffix array, representing the appearance positions of frequent substrings.

Lee and De Raedt (2005) mines frequent substring patterns from a string database consisting of multiple strings. The work uses a data structure based on suffix trees called *version space trees* and introduces two algorithms: VST and VFAST. VST is a simple APRIORI-like algorithm, VFAST works without candidate generation and thus only performs a single database scan.

4.3.2. Discussion

Both definitions mine frequent substrings. Substrings are ordered sequences, making frequent substring mining order-sensitive. Furthermore, frequent substrings are continuous. Order-sensitivity and pattern continuity do render frequent substring mining unusable for change pattern mining.

4.4. Frequent Episode Mining

In frequent episode mining a single sequence, called the event sequence, is mined for frequent episodes. As an abstract example, consider the event sequence given in Figure 4.2. It is easy to see that whenever both A and B occur in whatever order, C will also occur. This is, the episode "after the occurrence of both A and B, C occurs" is frequent. As you can see, an episode is a partially ordered collection of events occurring relatively close together.

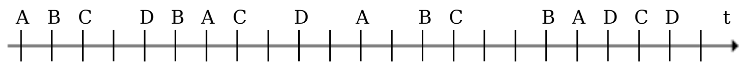

Figure 4.2: Frequent Episode Mining Example: Event Sequence

The problem of frequent episode mining is closely related to frequent substring mining: the major difference lies in the fact that frequent episode mining allows interleaved events within an occurrence of an episode, while frequent substring mining focusses on mining frequent continuous patterns.

4.4.1. Formal Problem Statement

Let E be a finite set of *event types*. An *event* is a tuple $\langle A, t\rangle$ where $A \in E$ and t is an integer, being the time of the event. An *event sequence* on E is a triple $\langle s, T_s, T_e\rangle$. T_s is an integer called the starting time of the event sequence, T_e is an integer called the ending time of the event sequence. s is a by time ordered sequence of events $(\langle A_1, t_1\rangle, \langle A_2, t_2\rangle, \ldots, \langle A_n, t_n\rangle)$, such that $A_i \in E$ and $T_s \leq t_i < T_e$ for all $i \in [1..n]$.

Episodes are partially ordered collections of events. A *serial episode* occurs in a sequence if its events occur relatively close together in the sequence (the events should occur in a given order). A *parallel episode* gives no constraints on the relative order (the order of the events does not matter). Finally, there are *hybrid episodes*, like the one from the example sequence given in Figure 4.2.

The problem of *frequent episode mining* can now be defined as: given a class of episodes (e.g., serial, parallel) and an input sequence of events, find all episodes that occur frequently in the event sequence. Note that practically all algorithms are limited to serial and parallel episodes.

Similar to closed frequent itemset mining, there is a variant called closed frequent episode mining. An episode is *closed* if it is frequent and there exists no super episode with the same support.

4.4.2. Approaches

The problem was first stated by Mannila, Toivonen and Verkamo (1995), as follows: "given a class of episodes and an input sequence of events, a window width and a frequency threshold, find all episodes of the class that occur frequently enough". The work also introduces a first general APRIORI-like algorithm, which is based a reformulation of the apriori-property for episodes (episodes are only considered when all its subepisodes are frequent). These methods were later extended by Manilla and Toivonen (1996).

The influential work Mannila, Toivonen and Verkamo (1997) introduces MINEPI and WINEPI, two APRIORI-like algorithms, which are given for both serial and parallel episodes. The user has to specify window width (the input sequence is seen through *partially overlapping windows*) and a support threshold.

WINEPI uses "frequency", being defined as the fraction of all sliding fixed-length windows in which the episode occurs, as interestingness measure (support). MINEPI uses an alternative interestingness measure: support is defined as the number of minimal windows that contain it ("minimal occurrences"). This is later found by Tatti (2009) to not be monotonically decreasing, a property which is required for APRIORI-like level-wise mining. Laxman, Sastry and Unnikrishnan (2007) solves the problem by defining support as the maximal number of non-overlapping minimal windows.

Casas-Garriga (2003) improves the method of windows-based frequency counting as seen in WINEPI. It replaces the fixed window width by a more intuitive notion called the *gap constraint*. In essence, it allows interesting episodes to grow during mining by increasing the window width accordingly. Laxman, Sastry and Unnikrishnan (2004) further improves frequency counting.

Zhou, Liu and Cheng (2010) discusses mining closed episodes based on minimal occurrences. It addresses the following problem: given an event sequence, a minimum support threshold, and a maximum window size threshold, find the complete set of closed serial episodes. To this purpose a MINEPI-based method called CLO_EPISODE is proposed. CLO_EPISODE combines a breadth first search with pruning techniques to prune non-closed episodes. Note that CLO_EPISODE did not address the non-monotonicity issue of MINEPI.

Tatti and Cule (2011) proposes a depth-first algorithm for mining frequent closed episodes that properly handles simultaneous events (occurring at exactly the same time). To address problem-specific issues a more conservative definition of closure is used during search: instance-closure is used to reduce the search space. Since instance-closed episodes are not always closed, a post-processing step filters closed episodes. The given approach uses frequency based on a sliding window, as defined for WINEPI, but it allows adoption for other monotonically decreasing measures.

4.4.3. Discussion

Although frequent episode mining is more flexible than frequent substring mining by allowing interleaved events within an occurrence of an episode, and thus fixes the continuity problem, order sensitivity remains.

Indeed, frequent episode mining approaches operate by using a fixed-length or variable length (gap-constraint based approaches) sliding window over the input sequence. As the relative order of changes is not fixed, the mining result is not fixed either.

Note that this sliding window technique is also used by Negara et al. (2014) to mine for unknown change patterns in a continuous sequence of changes in the order in which they are performed, provided by a change logger. In this work, the continuous input sequence is divided into distinct transactions, which partially overlap to account for patterns crossing transaction boundaries. Those transactions are then mined using a variant of frequent itemset mining, which tracks each item's occurrences to allow duplicate items (i.e. itembag mining) and which uses a special definition of minimal support to accomodate overlapping transactions and multiple item occurrences. The usage of this technique is justified by the use of change logging and the assumption that developers perform related changes are performed in close neighborhood from a time-perspective. However, the technique is not applicable to the output sequence of a change distiller.

4.5. Sequential Pattern Mining

Sequential pattern mining is an extension of frequent itemset mining to the case of temporal data: the transaction database makes place for a sequence database, which contains sequences, being ordered lists of itemsets. It is also closely related to frequent episode mining, but instead of mining episodes occurring sufficiently many times in a single long sequence, sequential patterns occurring in sufficiently many sequences are discovered.

As finding products frequently bought together is an application of frequent itemset mining, sequential pattern mining can be applied to entire customer shopping sequences, predicting what customers will buy based on items they bought earlier. In this application, the sequence database contains a sequence for each customer, containing his baskets.

4.5.1. Formal Problem Statement

Let the *item base* $I = \{i_1, \ldots, i_m\}$ be a set of items. Any subset T of I is called an *itemset*. A *sequence* $S = (T_1, T_2, \ldots, T_N)$ is an ordered list of itemsets. Items in the same itemset occur at the same time, items in different itemsets occur at different times. Note that an item can occur only once in an itemset, but it can occur multiple times in a sequence consisting of multiple itemsets. A sequence $S_a = (A_1, A_2, \ldots, A_n)$ is a *subsequence* of a sequence $S_b = (B_1, B_2, \ldots, B_m)$ (denoted $S_a \sqsubseteq S_b$) if there exist n integers $i_1 < \ldots < i_n$ such that, for all A_j holds: $A_j \subseteq B_{i_j}$.

A *sequence database* D is a set of tuples $\langle sid, S \rangle$ where sid is a unique sequence identifier and S is a sequence. A tuple $\langle sid, S \rangle$ in D is said to *contain* a sequence S_a if $S_a \sqsubseteq S$. The *support* of a sequence S_a in D is the percentage of tuples in D that contain that sequence S_a. A *frequent sequence* is a sequence whose support is above a certain threshold c. A *closed frequent sequence* is a frequent sequence for which there are no supersequences with the same support. The problem of *frequent sequential pattern* mining can now be defined as: given a sequence database and a threshold, identify all frequent sequences.

For example, a customer buys a PC, later he buys some books about photography. He becomes interested and buys a digital single-lens reflex camera and SD memory card. This results in the sequence: $\langle \{PC\}, \{Book\}, \{DSLR, SD\} \rangle$. The sequence $\langle \{PC\}, \{SD\} \rangle$ is a subsequence of this sequence since $\{PC\} \subseteq \{PC\}$ and $\{SD\} \subseteq \{DSLR, SD\}$.

Note that in literature *alternative definitions* are sometimes used. For example, sometimes sequences are defined as ordered lists of items, not as ordered list of itemsets. Furthermore, other definition take into consideration the time of occurrence of sequence elements.

4.5.2. Approaches

The apriori property can be reformulated for sequential patterns: no supersequence of an infrequent sequence can be frequent. Agrawal and Srikant (1995) introduces the concept of sequential pattern mining and gives APRIORI-like algorithms for frequent sequence mining: APRIORIALL, APPRIORISOME and DYNAMICSOME. Other APRIORI-based approaches include GSP (Srikant & Agrawal, 1996) and SPADE (M. J. Zaki, 2001), the latter using a vertical database format. Pattern-growth based methods include FREESPAN (Han et al., 2004) and PREFIXSPAN (Pei et al., 2001). CLOSPAN (Yan, Han & Afshar, 2003) is an algorithm for mining closed sequential patterns. Wang and Han (2004) discusses BIDE, which improves CLOSPAN.

4.5.3. Discussion

The order-sensitivity problem of frequent episode mining persists for frequent sequence mining. Like frequent episode mining, sequential pattern mining uses a sliding window.

4.6. Algorithm Selection

Table 4.4 shows a comparison of the investigated algorithms in function of the characteristics of the output of the change distiller. Frequent itemset mining is order insensitive and allows discontinuous patterns. We will use this technique.

For now we conclude on using frequent itemset mining. In the next chapter we discuss the construction of a transaction database from a sequence of distilled changes. We also determine whether mining all frequent itemsets is necessary or whether a subset of these itemsets (e.g. all closed frequent itemsets, all maximal frequent itemsets) is preferred.

	Allowing discontinuity	Order insensitive
Frequent Itemset Mining	✓	✓
Frequent Sequence Mining	✓	
Frequent Episode Mining	✓	
Frequent Substring Mining		

Table 4.4: Data Mining Algorithms: Overview

4.7. Summary

In Section 4.1 we started with a discussion of the inherent characteristics of our problem domain, i.e. changes represented by AST nodes, distilled from successive VCS snapshots. First, the order of the changes within the sequence of distilled changes is only one of many possible orderings of those changes, making it impossible to rely on change order for change pattern mining. Next, there is a possibility of *unrelated changes interleaving changes forming a change pattern*

We then covered a representative variety of "mineable types of patterns" we have considered for mining VCS data for unknown change patterns: we have discussed 4 frequent pattern mining approaches and evaluated their input, output and assumptions in terms of problem domain characteristics. In Section 4.2 we covered frequent itemset mining and association rule learning, Section 4.3 focussed on frequent substring mining, Section 4.4 discussed frequent episode mining and Section 4.5 focussed on sequential pattern mining.

Finally, in Section 4.6 we gave a final comparison of frequent pattern mining approaches. Due to its order insensitivity and the fact that discontinuous patterns are allowed, frequent itemset mining is selected as the preferred approach for this thesis.

5 Edit Scripts as Itemsets

Frequent pattern mining algorithms mine frequent substructures (frequent patterns) inside a larger structure. Mining approaches have two common requirements: first, they require a way to generate candidate patterns and second, a method for quantifying candidate patterns in terms of interestingness is necessary.

When performing frequent itemset mining, these substructures are sets of items ("itemsets"), the larger structure is a bag of itemsets ("transaction database"). Recall from the previous chapter that frequent itemset mining was introduced in the context of market basket analysis for finding products that are frequently bought together. Given a fixed set of items (e.g. *Bread*, *Milk*) and a number of baskets ("transactions"), each containing a subset of all items, the aim is to find items that appear frequently together in baskets.

As such, frequent itemset mining provides answers to both requirements. First, generation of candidate frequent itemsets can be done iteratively by starting with the empty set, then generating candidate n-itemsets from candidate $(n-1)$-itemsets by adding all possible items. Next, itemsets can be quantified in terms of interestingness by means of their support, i.e. the number of transactions containing all items in the itemset.

In this chapter we investigate how we can apply frequent itemset mining to a sequence of changes reported by a change distiller, i.e. the process of building a transaction database from a sequence of distilled changes. In doing so, we address two main questions: 1) how do we form a transaction database given a single sequence of changes? and 2) what is our set of possible items? Section 5.1 discusses how preprocessing in the form of grouping and generalization of the sequence of distilled changes answers both questions. Section 5.2 then further discusses change grouping, Section 5.3 focusses on change generalization. In Section 5.4 we revisit frequent itemset mining and apply it to the formed groups of change generalizations. Finally, in the last section we study the equivalence relation used during change generalization more thoroughly.

5.1. Preprocessing Data

First, remark that change distilling reports a sequence of changes, which when performed in order, transform the source AST into the target AST. Also recall that only one ordering of many possible orderings is reported. To avoid change order dependent mining results, we group changes into (unordered) sets by the subtree in which they occur, rooted at a specific type of node. For example, we can group changes at by the method in which they occur.

A second observation is that all distilled changes are distinct. When defining our set of items I as the set of distinct changes and building the transaction database D using the sets resulting from grouping our changes, all itemsets in the transaction database will be disjoint, and no patterns will be found (at least, when the minimal support threshold is greater than 1). Instead, we have to generalize the changes, i.e. consider certain changes equal based on certain commonalities. In other words: we convert changes into change generalizations by dropping all information not required to be common. The set of items I is then equal to set of change generalizations, the transaction database D then contains sets of change generalizations.

To conclude, two-step *preprocessing* transforms the sequence of distilled changes into a mineable transaction database: *grouping* and *generalization*. During grouping, the single sequence of changes is transformed into multiple sets of changes. Then, generalization transforms these sets of changes into sets of change generalizations. These sets of change generalizations correspond to itemsets in the transaction database.

5.2. Grouping Changes

Recall from Chapter 4 that a transaction database consists of multiple transactions. A first step involves grouping all changes in the distilled change sequence into multiple sets: all these sets will later, at mining time, correspond to transactions in the transaction database.

5.2.1. Grouping Strategies

The strategy used to form groups determines the number of groups and thus the number of transactions in the transaction database. Furthermore, it directly determines the maximal scope of change patterns. Grouping strategies include:

- *By change.* Each change receives its own private group. All itemsets in the transaction database will be singletons. A change pattern will never consist of multiple changes.

- *By containing AST node type* (e.g. `Statement`, `MethodDeclaration`, `TypeDeclaration`, `CompilationUnit`). Changes are grouped by the statement, method, type, file,. . . in which they occur. Maximum pattern size will be limited to the chosen grouping granularity. For example, grouping by statement does not allow finding multi-statement patterns, and grouping by type allows finding renames of fields or class variables as well as the systematic insertion of new methods.

- *Intra-commit or inter-commit.* Changes are grouped by the commit in which they occur or even by multiple commits.

- *No grouping.* All changes belong to the same group. Unless the minimal support threshold is set to 1, no patterns will be found. When the minimal support threshold is set to 1, only 1 pattern will be found, and this pattern will encompass all change generalizations.

In our approach, changes are grouped at a specific granularity according to the second strategy, i.e. changes are grouped by the subtree in which they occur, rooted at a specific type of node. This top node of the subtree is called the *container* of the group. As such, groups are pairs $\langle container, changes\rangle$ where *container* is an AST node and *changes* is a set of distinct changes, provided by a change distiller and thus represented by concrete AST operations.

5.2.2. Example

To continue our practical example from previous chapters, again consider the ChangeNodes change sequence in Table 3.1. Suppose we are grouping changes at a `MethodDeclaration` granularity: inserts are grouped by the method in which they occur. Classification of change 1 and change 2 is easy: simply look for the ancestor of type `MethodDeclaration` of the original node of the change. This way, change 1 belongs to the group with as container the `MethodDeclaration` of method `computeDistance()`, since it's original node is the `Block` embodying the body of this `MethodDeclaration`. Similarly, change 2 belongs to the group corresponding with the method `computeDirection()`. Classification of the other changes is more complex, since they do not have an original node in the source AST. However, we know that one of the ancestors of the copy of a change is in fact the copy of another insert which has an original node and which happened earlier in the change sequence. For example, change 1's copy (an `IfStatement`) is an ancestor of the copy node (the `SimpleName p`) of change 4, and since change 1 belonged to the group corresponding with container the `MethodDeclaration` of method `computeDistance()`, so does change 4. To finish grouping of the changes in our example, change 1, 3, 4 and 5 belong to the group with container the `MethodDeclaration` of method `computeDirection()` and changes 2, 6, 7 and 8 belong to the group with container the `MethodDeclaration` of method `computeDistance()`. See also Table 5.1

Table 5.1: Groups in Example "Return If Current Object is equal to Passed Argument"

Change	Op.	Original	Property	Idx	Copy
Group #<MethodDeclaration computeDistance>					
1	insert	#<Block { ... }>	statements	0	#<IfStatement if(this.equals(p)) return 0;>
3	insert	-	expression	-	#<ThisExpression this>
4	insert	-	arguments	0	#<SimpleName p>
5	insert	-	expression	-	#<NumberLiteral 0>
Group #<MethodDeclaration computeDirection>					
2	insert	#<Block { ... }>	statements	0	#<IfStatement if(this.equals(point)) return 0;>
6	insert	-	expression	-	#<ThisExpression this>
7	insert	-	arguments	0	#<SimpleName point>
8	insert	-	expression	-	#<NumberLiteral 0>

5.3. Generalizing Changes

Although each group will correspond to a transaction in the transaction database, as mentioned further preprocessing of the changes is necessary. To see why, remember that frequent itemset mining mines itemsets which are contained in a number of transactions greater than a user-specified threshold. However, since all changes are distinct, considering changes as items will render all itemsets disjoint. Consecutively, mining will not yield any frequent itemsets (unless the frequency threshold is set to 0 or 1, but this corresponds to mining infrequent patterns).

During grouping we have formed disjoint sets of distinct changes. However, since frequent itemset mining looks for items occurring in more than one itemset, we have to generalize changes in such a way that certain changes in different groups are considered equivalent if and only if they share certain commonalities; such equivalent changes will then be represented by the same item. In other words, we disregard certain information from changes (we generalize changes and only retain certain properties), such that two changes can result in the same generalization.

The central question in generalization is: when are two changes equivalent, i.e. which commonalities do we require? With potentially many changes applied between successive VCS snapshots, naive approaches quickly result in many uninteresting change patterns. Due to the impact of required commonalities on both the amount of mined patterns and the information mined patterns provide, will study the equivalence relation in greater detail in the last section of this chapter.

For now, once again consider the example from Table 3.1. We have chosen to generalize changes in such a way that two changes are considered equal if and only if the same kind of operation is performed on the same type of AST node at the same location within its container; this definition disregards structural properties of the change subject as well as any other context. In this approach, a generalized change is a triple $\langle operation, nodeType, location\rangle$, where $operation \in \{Insert, Update, Delete, Move\}$, *nodeType* is a subclass of `ASTNode` and *location* is the path locating the change within its container. Table 5.2 shows the resulting groups of change generalizations. For brevity, the last column gives a character representation of each generalized change.

Table 5.2: Preprocessing Result of Example "Return If Current Object is equal to Passed Argument"

Change	Operation	Type of Node	Location	Item
Group #<MethodDeclaration computeDistance>				
1	Insert	#<Class IfStatement>	[body statements 0]	*A*
3	Insert	#<Class ThisExpression>	[body statements 0 expression expression]	*B*
4	Insert	#<Class SimpleName>	[body statements 0 expression arguments 0]	*C*
5	Insert	#<Class NumberLiteral>	[body statements 0 thenStatement expression]	*D*
Group #<MethodDeclaration computeDirection>				
2	Insert	#<Class IfStatement>	[body statements 0]	*A*
6	Insert	#<Class ThisExpression>	[body statements 0 expression expression]	*B*
7	Insert	#<Class SimpleName>	[body statements 0 expression arguments 0]	*C*
8	Insert	#<Class NumberLiteral>	[body statements 0 thenStatement expression]	*D*

This simple approach will obviously not always suffice: this kind of generalization certainly is too generic and it will result in mining results which are often not useful in practice. It will be useful to study alternative ways of generalization, which also take into account more context and other AST node properties. We leave this study for the end of this chapter.

5.4. Frequent Itemset Mining: Revisited

Recall that we had defined a transaction as a tuple $\langle tid, itemset \rangle$ with *tid* a unique transaction identifier and *itemset* a set of items. The preprocessing result as discussed in previous sections can be used to form such transactions: for each group, form a transaction with *tid* equal to the container of the group and *itemset* equal to the set of change generalizations within the group. We can then apply closed frequent itemset mining to mine change patterns.

To finish our example, preprocessing resulted in a database with two transactions:

$$\langle \#\text{<MethodDeclaration computeDistance>}, \{A, B, C, D\} \rangle$$

$$\langle \#\text{<MethodDeclaration computeDirection>}, \{A, B, C, D\} \rangle$$

Applying closed frequent itemset mining with minimum support threshold 2 results in one closed frequent itemset or change pattern: $\{A, B, C, D\}$ with support 2.

5.4.1. Why Mining Closed Frequent Itemsets?

An attentive reader might notice that we perform closed frequent itemset mining. To see why, we discuss the following question: do we mine all frequent itemsets, or is a subset of these itemsets (e.g. all closed frequent itemsets, all maximal frequent itemsets) preferred?

Recall the apriori property stating that any subset of a frequent itemset must also be frequent. Standard frequent itemset mining includes such frequent subsets, closed frequent itemset mining does not: closed frequent itemsets are frequent itemsets for which there are no superitemsets with the same support. We have already illustrated the difference between frequent itemset mining and closed frequent itemset mining in the example given in Table 4.3, in which, for instance $\{Bread\}$ is not a closed frequent itemset due to frequent superitemset $\{Bread, Butter\}$ having the same support. Since we want to mine full code change patterns rather than all combinations of their constituents, we want to perform closed frequent itemset mining. Furthermore, some of these partial code change patterns are even illegal due to the possible presence of nested change operations. For example, consider again the illustrative example from Code Listing 3.1, involving the insertions of a statement to return if the current object is equal to the passed argument. With the minimal support threshold set to 2, $\{A, B, C, D\}$ (Table 5.2) is a closed frequent itemset. Subitemset $\{B, C, D\}$ is frequent too, but all three changes involve inserts nested inside the insertion of the `IfStatement` by change A, which is not part of the subitemset, and which is thus never inserted.

In maximal frequent itemset mining, mining results are limited to frequent itemsets included in no other frequent itemset. For instance, consider the mining results given in Table 4.3. The frequent itemset $\{Butter\}$ with support 5 is not a maximal frequent itemset, since it is included in the frequent itemsets $\{Bread, Butter\}$ and $\{Milk, Butter\}$, both having support 3. In other words, when performing maximal frequent itemset mining longer superitemsets with possibly lower support are preferred above smaller highly supported itemsets. When mining repositories for change patterns, this approach may exclude interesting change patterns with possibly many occurrences from the mining result.

Code Listing 5.1: Example: "Protected Stack"

```
public class Stack {

  private int[] stackArray;
  private int top;
  public Stack() {
    stackArray = new int[100];
    top = 0;
  }
- public void clear() {
+ public Stack clear() {
    top = 0;
+ return this;
  }
- public void push(int value) throws Exception {
+ public Stack safePush(int value) throws Exception {
    if(top == 100) throw new Exception();
    stackArray[top++] = value;
+ return this;;
  }
- public void poke(int value) {
+ public Stack poke(int value) {
    stackArray[top] = value;
+ return this;;
  }
- public void zap() throws Exception {
```

```
+ public Stack safeZap() throws Exception {
    if(top == 0) throw new Exception();
    --top;
+ return this;
  }
- public int pop() throws Exception {
+ public int safePop() throws Exception {
    if(top == 0) throw new Exception();
    return stackArray[--top];
  }

}
```

For example, consider Code Listing 5.1 showing modifications to a protected stack implementation. The user has renamed methods `push()`, `zap()` and `pop()` to `safePush()`, `safeZap()` and `safePop()` respectively. Furthermore, to allow chaining of stack modifications he has modified methods `clear()`, `push()`, `poke()` and `zap()` to return the `Stack` itself. When performing grouping at a method level and generalizing changes only by the kind of operation and the type of `ASTNode`, the preprocessing result is given in Table 5.3. To summarize, insertions of `SimpleName`-nodes correspond with method renames, insertions of `SimpleType`-nodes correspond with changes of a method's return type and insertions of nodes of type `ReturnStatement` and `ThisExpression` correspond to the insertion of a statement `return this;`. When performing closed frequent itemset mining, three patterns are found: pattern $\{A\}$ (support 3), pattern $\{B, C, D\}$ (support 4) and pattern $\{A, B, C, D\}$ (support 2). Indeed, three methods have their name changed, four patterns have been modified for chaining purposes, and of all these methods and 2 methods had both changes. Of these patterns, only pattern $\{A, B, C, D\}$ (support 2) is maximal. In other words, either only changing the method to allow chaining or only renaming the method would is not considered to be a pattern: only the composite change is.

5.5. Comparing Changes

We have grouped changes by the subtree in which they occur, rooted at a specific type of node. Each group of entirely distinct changes corresponds to an itemset. Using changes directly as items in these itemsets results in itemsets which are all disjoint. However, frequent itemset mining with a frequency threshold higher than 1 looks for items occurring in more than one itemset. This is never the case and thus no patterns will be found.

As a solution, we opted to perform change generalization in such a way that certain changes in different groups (itemsets) are considered equivalent if and only if they share certain commonalities. Changes which are considered equivalent will be represented by the same item in the mining database. In other words, we disregard certain information from changes (we generalize changes and only retain certain properties), and the resulting data structures, which we call *change generalizations*, are used to populate the itemsets in the transaction database.

Table 5.3: Preprocessing Result of Example “Protected Stack”

Change	Operation	Type of Node	Item
Group #<MethodDeclaration clear>			
1	Insert	#<Class ThisExpression>	*B*
2	Insert	#<Class ReturnStatement>	*C*
3	Insert	#<Class SimpleType>	*D*
Group #<MethodDeclaration push>			
4	Insert	#<Class ThisExpression>	*B*
5	Insert	#<Class ReturnStatement>	*C*
6	Insert	#<Class SimpleName>	*A*
7	Insert	#<Class SimpleType>	*D*
Group #<MethodDeclaration poke>			
8	Insert	#<Class ThisExpression>	*B*
9	Insert	#<Class ReturnStatement>	*C*
10	Insert	#<Class SimpleType>	*D*
Group #<MethodDeclaration zap>			
11	Insert	#<Class ThisExpression>	*B*
12	Insert	#<Class ReturnStatement>	*C*
13	Insert	#<Class SimpleName>	*A*
14	Insert	#<Class SimpleType>	*D*
Group #<MethodDeclaration pop>			
15	Insert	#<Class SimpleName>	*A*

To implement change generalization, we define an equivalence relation $\sim$ over the set of distillable changes, which considers two changes equivalent if and only if they share certain (but not necessarily all) commonalities. The used equivalence relation not only impacts the found change patterns, it only has an impact on the information mined patterns provide. Even more important, as we shall see, equivalence relations have to fullfil certain requirements in order to yield valid results. For example, because itemsets are sets, which are collections of distinct items, the equivalence relation may not consider two changes within the same group equivalent: this would result in itemsets with duplicate items.

5.5.1. Defining an Equivalence Relation over Changes

We now define a binary relation $\sim$ over the set of distillable changes. In order for this relation to be an *equivalence relation*, three properties must hold. Given changes a, b and c:

- *Reflexivity* ($a \sim a$). Each change is equal to itself.
- *Symmetry* ($a \sim b \rightarrow b \sim a$). If a change a is equal to another change b, then this other change b is equal to a.
- *Transitivity* ($a \sim b \wedge b \sim c \rightarrow a \sim c$). If a change a is equal to a change b and b is in turn equal to a change c, then change a is equal to change c.

For reasons that will become clear in the following subsections, we will define $\sim$ as a conjunction of three predicates: change operation equality, change subject equality and context equality. Before discussing $\sim$'s constituents in greater detail, for now lets consider two extreme cases in which $\sim$ can be defined:

- *All changes are distinct.* $\sim$ considers no 2 changes equal. As no change properties are disregarded, change generalization is the identity function, i.e. all changes generalize to themselves. Although this is by itself perfectly legal, all itemsets will be disjoint, and no change patterns will be found (at least, when the minimal support threshold is greater than 1).
- *All change are equal.* As no change properties are taken into consideration to discriminate between changes, all changes generalize to some constant value representing "any change occurred". This is legal as long as no two changes occur within the same group (e.g. two changes within the same method when grouping at a method granularity): this would result in itemsets with duplicate elements, which is not allowed since sets are collections of distinct objects.

The last definition of $\sim$ illustrates a more general problem with certain equivalence relations: changes within the same group should never generalize to the same value. We will come back to this important issue in Subsection 5.5.3.

5.5.2. Comparing Change Operations and Change Subjects

A first aspect to consider when comparing changes is equality of *change operations* (insert, update, move and delete).

Next to change operation equality we can define an equality over *change subjects.* The change subject of an insert operation corresponds to the inserted node, for a delete operation the change subject is the deleted node. Similarly, the for move operations the change subject refers to the moved node and for update operations, the change subject refers to the node that we have updated. As such, some change subjects are nodes in the source AST (deletes and updates), others are nodes in the target AST (inserts, moves). Several change subject comparison methods are given below. As an example of each method, consider Figure 5.1.

- *Full equality.* Both change subject AST trees are traversed recursively (deep equality). This requires full equality of node types, list node children, AST node properties and unwrapped primitive values. This is illustrated in Figure 5.1's top-left AST.
- *Structural equality.* Similar to full equality, but values for simple properties (variable names, literal values,. . .) are not compared. This is illustrated in Figure 5.1's top-right AST.
- *Depth-limited structural equality.* See also Figure 5.1's middle row ASTs: the left tree shows depth-limited structural equality with depth 2, the right tree shows a depth limitation of 3.
- *Structural equality of mandatory descendants.* Figure 5.1's bottom-left AST illustrates this: only straight-line arrows, representing mandatory properties, are followed in the recursion.
- *AST node type equality.* A special case of depth-limited structural equality, limited to depth 1: a change subject's children (list elements, AST node property values) are not checked. See Figure 5.1, bottom-right.
- *Do not check.* Change subjects are not taken into consideration.

5.5.3. Introducing Context

By only considering change operations and change subject characteristics when comparing changes, mined change patterns only give information about *what* (subject) was changed *how* (operation), but they give no information about *where* the change occurred within the group. For this reason, we discuss taking contextual information into consideration when comparing changes. In doing so, we reduce the number of change patterns, and increase the information contained by each change pattern. Furthermore, information about where changes occurred can help to automate the transformation of code according to change patterns.

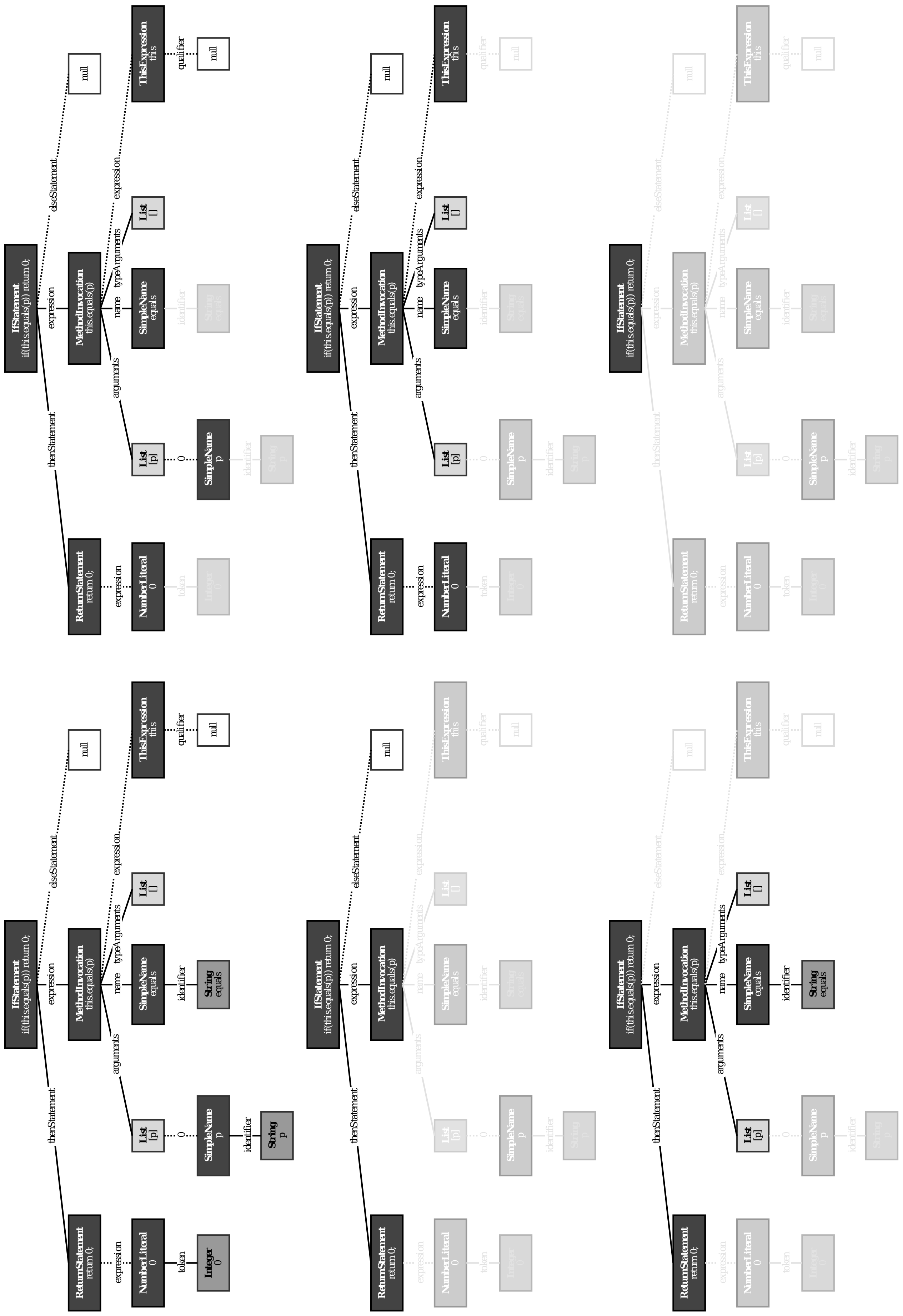

Figure 5.1: Examples of Several Context Equalities

Implications of Frequent Itemset Mining

In Subsection 5.5.1 we already discussed an equality considering all changes equal, resulting in changes generalizing to the same constant value indicating that “a change was performed”. As mentioned, this equivalence relation is only legal as long as no two changes are performed in the same group. To see why, recall that groups are used to form itemsets, with as items being the change generalizations occurring within the group. If all changes generalize to the same value, this corresponding itemset would contain duplicate elements. Since itemsets are collections of distinct objects, a form of data loss would occur.

When only comparing changes by requiring change operation and change subject commonalities, and thus disregarding all contextual characteristics, the same problem can occur. For example, an equal statement (same subject) can be inserted (same operation) twice within the same group, resulting in two changes within the same group having the same generalization. This implication of frequent itemset mining can be solved by using a contextual equality taking into consideration the location where changes occurred within the group: if this is done in such a way that no two changes are performed at the same locations, changes with an equivalent change operation and change subject are still distinct due to their different location and thus different context.

One way to locate changes within the group is by considering their exact location. We have already used this approach in the example given in Table 5.2, in which the path, through the abstract syntax tree, from group container to the node where a change occurred is used to compare context. This path includes structural properties through which do descend in the case of AST nodes, as well as list indices through which to descend in the case of list nodes.

Example Contextual Equalities

To finish our discussion of contextual equality, the following list gives a discussion of several contextual equality methods:

- *Full context equality.* Two changes are considered equivalent if and only if their context is equal, i.e. a recursive traversal of their container ASTs requires that both change subjects are at the same location, for other nodes equality of node types, list node children, AST node properties and unwrapped primitive values is required. In the case of grouping at a method granularity, this context equality only considers two changes equal if they occur at the same location inside two methods, in two different classes, who are completely identical including their name.

- *Structural context equality.* Similar to full context equality, but values for simple properties (variable names, literal values,...) are not compared.

- *Exact location equality.* Two changes are considered equivalent if and only if they have the exact same location within the group. This location includes structural properties through which to descend in the case of AST nodes, as well as exact list indices through which to descend in the case of list nodes.

- *Relative location equality.* All of the above contextual equality methods require 2 changes to have exactly the same location in the group order to be considered equivalent. This exact same location includes exact list indices in list nodes. Relative path equality replaces this exact list index equivalency by a less strict notion, using criteria relative to other members of the list the form (e.g. "before the first node", "after the last node" or "before/after node X"). Note that special considerations have to be taken, for example a node might be inserted after another inserted node, in which case the previous node is not in the AST yet. This approach still uniquely identifies changes within the container, but allows the detection of patterns

- *No equality.* Context is not taken into consideration when comparing changes.

5.6. Summary

In this chapter we have investigated the application of frequent itemset mining to a sequence of distilled changes. In the first section we discussed how preprocessing in the form of grouping and generalization supports the contruction of a mineable database.

During grouping (discussed in greater detail in Section 5.2), the single sequence of changes is transformed into multiple sets of changes. To avoid change sequence order dependent mining results, changes are grouped by the subtree in which they occur rooted at a specific type of node (called the group container).

Although each group will correspond to a transaction in the transaction database, considering changes as items will render all itemsets disjoint, because all change are distinct. For this reason we perform change generalization (covered in Section 5.3), which involves transforming the sets of changes into sets of change generalizations by disregarding certain information from changes such that multiple changes can result in the same generalization. These sets of change generalizations then correspond to itemsets in the transaction database.

In Section 5.4 we revisited frequent itemset mining and applied it to the formed groups of change generalizations. We proposed closed frequent itemset mining instead of regular and maximal frequent itemset mining.

The central question in generalization is: when are two changes equivalent, i.e. which commonalities do we require? To this end we defined an equivalence relation over changes, considering changes sharing certain commonalities equivalent. We discussed this equivalence relation as a conjunction of change operation equality, change subject equality and context equality in the last section.

6 Identifying Candidate Transformation Subjects

Identifying change patterns by mining VCS histories mainly benefits researchers, who can learn a lot from changes made to source code. In practice, however, developers need tool support. Indeed, manual implementation of systematic-repetitive changes to all necessary code locations is laborious, tedious and error prone.

In the context of this approach, patterns are similar modifications (with the same generalization) of code occurring within a number of groups above a certain user-specified threshold. Developers can use tool support to 1) identify new source code segments to which change patterns can be applied and 2) apply change patterns automatically to new source code. In this chapter we discuss the processing of change patterns to meet the first objective, i.e. finding new groups which can be transformed according to a change pattern.

In Section 6.1 we cover tool support and its benefits. Next, Section 6.2 presents a practical approach, applicable for change patterns derived using an equivalence relation requiring full equality of change operations, structural equality of subjects and a exact path contextual equality. Finally, Section 6.3 discusses required change pattern characteristics to allow querying new source code for candidate transformation subjects by means of templates.

6.1. Tool Support

Two meta-programming activities related to change patterns are program querying and program transformation. In this introductory section we first introduce meta-programming, then discuss both activities.

6.1.1. Meta-programming

Meta-programming refers to the act of writing programs that create or manipulate other programs. Such so-called *metaprograms* (e.g. compilers, interpreters, pretty printers, refactoring tools and code generation tools) treat programs as data: they take programs as input and might output modified programs.

The language meta-programs are written in is called the *meta-language*, the language of the programs being created or manipulated is called the *object language*.

This metalanguage can be identical to the object language. For example, in Lisp dialects (e.g. Common Lisp, Scheme), expressions are represented by lists, the primary Lisp data structure. This representation of programs as data has many benefits, including the ease of development of metaprograms and the ability of language extension by means of macros.

On the other hand, meta-programming is often done by means of *domain-specific languages* (DSLs). Typical examples are Lex and Yacc, which are often used in conjunction during compiler development. Lex[1] is a lexical analyzer generator. It reads a specification of the lexical analyzer written in a domain-specific language called Lex Source to generate a lexical analyzers ("scanner"). Yacc[2] is a parser generator working similarly: an input specification containing a collection of grammar rules is read. Generated lexical analyzers and parsers are written in C.

6.1.2. Program Querying

Program querying languages are metaprograms aimed at the extraction of information from programs to answer certain questions, such as the identification of interesting source code segments and the computation of program statistics.

One application of program querying in the context of change pattern mining is the identification of source code segments which can be transformed according to a mined change pattern.

6.1.3. Beyond Program Querying: Towards Program Transformation

Where program querying is limited to retrieving information from programs, *program transformation* refers to the act of modifying a program by means of modification of its source code.

Programs can be transformed manually, in fact developers do this all the time while writing new code. To aid certain program transformations, it is also often possible to use *program transformation tools*. Those tools provide an automated way to transform old source code into new source code.

[1] http://dinosaur.compilertools.net/lex/
[2] http://dinosaur.compilertools.net/yacc/

A straight-forward program transformation tool is the search-and-replace functionality provided by most text editors. This system works at a text level, more advanced tools often work at an AST level. For example, many Integrated Development Environments (IDEs) provide automated ways to perform a fixed set of semantics-preserving program transformations called refactorings (Opdyke, 1992). Performing a refactoring modifies the source code's AST, and thus also modifies the underlying textual representation.

Change patterns represent systematic-repetitive changes of source code, many of which are not part of some fixed set of well-known semantics-preserving refactorings. To avoid repetitive manual modification of source code, reducing the risk of human error, developers can benefit from a method to automatically apply mined change patterns to new source code, similar to the automatic application of refactorings. Note that by "applying a mined change pattern to new source code", we mean the transformation of this new source code, in a way that is similar to the systematic modification as described by the pattern.

6.2. An Algorithm for Candidate Transformation Subject Identification

In this section we present a program querying approach identifying candidate transformation subjects, applicable for change patterns derived using an equivalence relation requiring full equality of change operations, structural equality of subjects and a exact path contextual equality. Concretely, we will construct an Ekeko/X template, specifying all characteristics of candidate transformation subjects. The Ekeko/X library then allows finding ASTs in other source code which match the template, i.e. finding code locations which can be transformed as described by the pattern.

6.2.1. The Ekeko/X Program Transformation Tool

Ekeko/X is a template-driven program transformation tool developed on top of the meta-programming library Ekeko. We first briefly introduce Ekeko, afterwards we shift our focus to the Ekeko/X program transformation tool and its templates. Finally, we discuss the template construction algorithm.

The Ekeko Meta-Programming Library

Ekeko[3] (De Roover & Stevens, 2014) is a Clojure library offering meta-programming facilities: it offers logic meta-programming facilities to allow querying of Eclipse workspaces as well as applicative meta-programming facilities enabling program manipulation.

On the one hand, Ekeko offers logic meta-programming facilities to support querying programs. At the heart of these facilities lies the `ekeko` special form, which launches

[3]https://github.com/cderoove/damp.ekeko/

a logic query: it takes a vector of logic variables and a sequence of logic goals and its solutions consist of the bindings for its variables such that all logic goals succeed.

EKEKO owes its logic meta-language to Clojure's `core.logic`. As a result `core.logic`'s functionality is freely available, next to a library of predicates allowing the program querying: provided basic predicates reify the basic structural, control flow and data flow relations of the program being queried. For convenience, several higher-level predicates are derived from these basic predicates.

On the other hand, EKEKO offers applicative meta-programming facilities to enable program transformation, to associate error markers and to compute statistics. For example, it provides a functional interface to the `org.eclipse.jdt.core.dom.rewrite.ASTRewrite` API and provides functionality to associate and disassociate problem markers with AST nodes.

Although EKEKO allows specifying queries gradually, refining the corresponding program queries incrementally, and although it offers extensive documentation and code completion features, EKEKO requires significant expertise regarding functional and logic programming as well as the Eclipse JDT.

The EKEKO/X Program Transformation Tool

EKEKO/X[4] (De Roover & Inoue, 2014), also developed at the Software Languages Lab of the Vrije Universiteit Brussel, is a template-driven program transformation tool implemented on top of EKEKO.

EKEKO/X-*transformations* consist of two templates: the *left-hand side template* identifies transformation subjects and the *right-hand side template* defines changes to perform to these subjects. We will only focus on construction of a left-hand side template.

Left-hand side templates specify the characteristics of the candidate transformation subjects; EKEKO/X provides functionality to match other ASTs against a left-hand side template, reporting all nodes/subtrees matching the given template. A template is a data structure similar to an AST, but allows annotation of its nodes with *directives*, which specify the matching strategy of other nodes against the given node.

Template construction always starts from a given AST. EKEKO/X offers an extensive library of *operators*, which can then be applied the template. Such operators manipulate template nodes (e.g. remove the node on which it is applied) or add directives to template nodes. For example, applying the operator `replace-by-wildcard` on a node of a template adds a `directive-replacedbywildcard` to this node. Adding such a directive makes any node of the queried program match.

[4]https://github.com/cderoove/damp.ekeko.snippets

6.2.2. Template Construction Algorithm

We present an algorithm applicable for change patterns derived using an equivalence relation requiring full equality of change operations, structural equality of subjects and a exact path contextual equality.

A mined pattern is technically nothing more than a bag of change generalizations, together with a list of all instances of the change patterns, i.e. all groups in which the change pattern occurs (note that the number of items in this list is equal to the support of the change pattern). As templates are constructed from a specific given AST node, which is then modified by the use of operators, we start from one specific change pattern instance (any instance can be used). We then take the group container of this instance as a basis for template construction. Next, we apply operators to generalize the nodes in the group according to the commonalities required by the equivalence relation.

Before discussing the main algorithm, consider helper function `structural-generalizer` given in Code Listing 6.1 (note that given code is simplified, and not following the real Ekeko/X API). The function takes a template and a node inside this template. The template is recursively traversed, nodes for primitive values receive a `replacedbywildcard` directive.

Code Listing 6.1: Structural generalization in Ekeko/X templates

```
(defn
  structural-generalizer
  [template current]
  (cond (is-list? current)
          (doall (map (fn [child] (structural-generalizer template child))
                      current)
        (is-node? current)
          (doall (map (fn [child] (structural-generalizer template child))
                      (node-children current)))
        (is-primitive? current)
          (replace-by-wildcard! template current))))
```

The main algorithm is given in Code Listing 6.2. It is to be called with the current template (created from the container AST of the group of the change pattern instance) and initially the root node of this template. Once again, the algorithm traverses the AST, introducing wildcards where necessary.

Boolean flags returned by the function are used throughout the recursion to indicate whether container nodes (lists and AST nodes) should be replaced by wildcards: only nodes having no descendants on which changes are involved are generalized. As a result, the path from the root node to the changes (and only this path) is maintained. Change subjects which are present in the LHS (i.e. updates, deletes and moves) are generalized structurally using the helper function discussed above.

Code Listing 6.2: Algorithm for candidate subject identification of change patterns derived using an equivalence relation requiring full equality of change operations, structural equality of subjects and a contextual equality

```
(defn
  generalize
  [template current]
  (let [change (get-change-at current)]
    (cond (and change (not (insert? change)))
            (do (structural-generalizer template current)
                true)
          (is-nil? current)
            (do (replace-by-wildcard! template current)
                false)
          (is-list? current)
            (if (or (some true?
                          (doall (map (fn [child] (generalize template child))
                                      current)))
                    (not (nil? change)))
                true
                (do (replace-by-wildcard! template current)
                    false))
          (is-node? current)
            (if (or (some true?
                          (doall (map (fn [child] (generalize template child))
                                      (node-children current))))
                    (not (nil? change)))
                true
                (do (replace-by-wildcard! template current)
                    false))
          (is-primitive? current)
            (do (replace-by-wildcard! template current)
                false)))))
```

6.3. Specifying Pattern Subject Characteristics using Templates

In the previous section we covered a template construction algorithm applicable for change patterns derived using an equivalence relation requiring full equality of change operations, structural equality of subjects and a exact path contextual equality. As mentioned, template construction always starts from a given AST. As a consequence, we start from a specific change pattern instance by building a template given its group container AST, which encompasses all change generalizations occurring within this group. We then use operators to modify the template according to the generalization approach, i.e. we only retain the commonalities the equivalence relation requires for the source AST.

The sole requirement to allow subject identification lies in the fact that necessary operators should exist to allow modification of templates according to the generalization approach: it should be possible to build templates based on only the commonalities required by the equivalence relation.

Given the information of our example from Code Listing 3.1, it is possible to build a template. As an illustration, the resulting template's string representation is given in Code Listing 6.3 ("..." represents a node with a `replacedbywildcard` directive, "*" represents cardinality zero or more). This corresponds to: any `MethodDeclaration` whose body is a `Block`. Please note that the `IfStatement` being inserted is not included in the template: the template identifies candidate transformation subjects, which are locations where the change pattern could be applied. However, the `Block` is required since the `IfStatement` is inserted inside the `Block`'s structural property `statements` at position zero.

Code Listing 6.3: Code Template of the Pattern in Example: "Return If Current Object is equal to Passed Argument"

```
...* ... ...(...*) {
  ...*
}
```

6.4. Summary

Manual implementation of systematic-repetitive changes to all necessary code locations is laborious, tedious and error prone. For this reason, developers can use tool support.

In the first section of this chapter we started with a discussion of meta-programming and 2 practical applications, related to change pattern mining, which can help developers: program querying for candidate transformation subjects and automated program transformation. Candidate transformation subject querying relates to the identification of new source code segments which can be transformed according to a change pattern, while automated program transformation goes a step further by automatically modifying source code according to the change pattern.

In the rest of the chapter we focus on the former problem, i.e. program querying for candidate transformation subjects. We presented an approach applicable for change patterns derived using an equivalence relation requiring full equality of change operations, structural equality of subjects and a exact path contextual equality in Section 6.2. This approach is based on code templates specifying necessary code characteristics.

In the last section we discussed required change pattern characteristics to allow querying new source code for candidate transformation subjects by means of templates. This boils down to the fact that operators should exist to allow modification of templates according to the generalization approach.

7 Evaluation

In this evaluation, we would like to answer these questions:

- **RQ1 (Recall Known): can the approach recall known change patterns?** By answering this question we wish to address whether the approach can derive change patterns from relatively small code fragments to which known systematic-repetitive changes are applied.
- **RQ2 (Find Unknown): how does the approach perform on open-source projects?** Next to determining performance on relatively small code fragments with known systematic-repetitive changes we want to evaluate the performance of our approach on open-source projects, i.e. determine its ability to derive useful previously unknown change patterns given a large database of code and code changes.

All experiments are performed on the Exapus source code[1], a web application for exploring the usage of APIs within a single project and across a corpus of projects along the dimensions of where, how much and in what manner. We have chosen Exapus due to familiarity with the project and the knowledge that many systematic-repetitive changes were made during development. At time of writing the project's git repository contains 263 commits made by 3 authors between October 5th, 2012 and April 2nd, 2014. The last commit contains 189 files, of which 156 are text files together containing 17240 lines of code. Of these text files, 129 files together containing 13198 lines of code are java source code files.

7.1. Can the Approach Recall Known Change Patterns?

To answer the first question, we did an empiric evaluation. We applied the algorithm to a set of 4 relatively small code fragments with known systematic-repetitive changes. For each code fragment, we first identified intuitively which change pattern(s) are present.

[1]https://github.com/cderoove/exapus

We then applied the mining algorithm under a variety of representative groupings and generalizations. Mined change patterns are then verified manually by comparison with the change patterns present intuitively. We manually counted true positives TP (mined change pattern instances which are an instance of the intuitive pattern), false positives FP (mined change pattern instances which are not an instance of the intuitive pattern) and false negatives FN (instances of the intuitive patterns which are not an instance of the mined pattern). This allowed us to compute precision as $TP/(TP + FP)$ and recall as $TP/(TP + FN)$.

All code fragments come from the Exapus project mentioned above. Code fragments are mined in isolation by the compilation unit in which they occur.

Code Fragment 1

Code Listing 7.1 depicts the a modification of the file FactForestTreeContentProvider.java in commit cad2c96 ("Moved FactForest to exapus.model.forest package"). The file contains 129 lines, of which only the listed 9 import declarations were modified. Changes are the result of the movement of several files to a subpackage within the same commit. For illustrative purposes, Figure 7.1 shows the underlying source AST structure of the first modified import declaration. An `ImportDeclaration` has a structural property `name` whose value is a `Name`, i.e. a `SimpleName` or a `QualifiedName`. A `QualifiedName` is of the form `qualifier.name` with structural property `qualifier` another `Name` and structural property `name` a `SimpleName`. As such, differencing source AST and target AST of the compilation unit under investigation results in 9 changes, one insert for each import declaration, representing the insertion of the `QualifiedName` `exapus.model.forest` in structural property `qualifier` of the structural property `name` the import declaration.

Code Listing 7.1: Exapus, commit cad2c96 ("Moved FactForest to exapus.model.forest package."), modifications to ExapusRAP/src/exapus/gui/views/forest/FactForestTreeContentProvider.java

```
...

import com.google.common.collect.Iterables;

- import exapus.model.DeltaEvent;
- import exapus.model.FactForest;
- import exapus.model.ForestElement;
- import exapus.model.IDeltaListener;
- import exapus.model.Member;
- import exapus.model.MemberContainer;
- import exapus.model.PackageLayer;
- import exapus.model.PackageTree;
- import exapus.model.Ref;
+ import exapus.model.forest.DeltaEvent;
+ import exapus.model.forest.FactForest;
+ import exapus.model.forest.ForestElement;
+ import exapus.model.forest.IDeltaListener;
+ import exapus.model.forest.Member;
+ import exapus.model.forest.MemberContainer;
+ import exapus.model.forest.PackageLayer;
+ import exapus.model.forest.PackageTree;
+ import exapus.model.forest.Ref;

public class FactForestTreeContentProvider implements ITreeContentProvider, IDeltaListener {

...
```

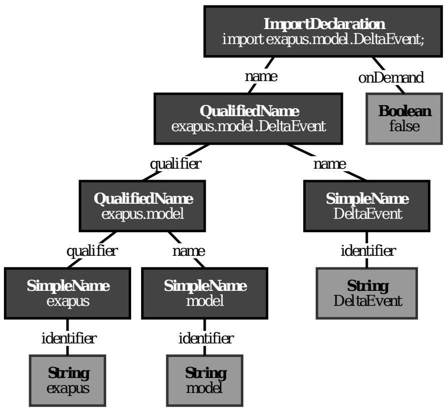

Figure 7.1: AST of the First Modified `ImportStatement` of Code Listing 7.1

Intuitively, the corresponding change pattern involves changing qualifiers `exapus.model` of names of import declarations to `exapus.model.forest`.

Results are given in Table 7.1. First of all, note that no pattern can be detected when grouping at a `Statement` or `MethodDeclaration` granularity, no matter which equivalence relation is used. Indeed, import declarations are not statements, nor are they wrapped inside a method declaration. As a result, grouping at `Statement` or `MethodDeclaration` granularity results in one group with group container undefined, containing all changes. Due to the absence of a group container, the mining algorithm does not consider changes in this group. To conclude, under both configurations there are 9 false negatives, 0 true positives and 0 false positives with respect to the intuitive pattern, resulting in a undefined precision (division by zero) and zero recall.

Grouping changes at an `ImportDeclaration` granularity, using a full change operation equality, full change subject equality and exact location contextual equality yields the change pattern "insert of `QualifiedName exapus.model.forest` at position `[:name :qualifier]` in `ImportDeclarations`" with support 9. As such, the pattern can be applied at all AST nodes that are `ImportDeclarations` with their `name` being a `QualifiedName`. This includes all 15 import declarations in the source AST; for example, the pattern can be applied to `import com.google.common.collect.Iterables;`, resulting in `import exapus.model.forest.Iterables;`. Thus, the pattern can be applied to 15 AST nodes, where the intuitive change pattern could only be applied to 9 AST nodes. To conclude, there are 0 false negatives, 9 true positives and 6 false positives with respect to the intuitive pattern, resulting in a recall of $9/(9+0) = 100\%$ and a precision of $9/(9+6) = 60\%$.

Grouping changes at an `ASTNode` granularity using the same equivalence relation yields another change pattern "insert of `QualifiedName exapus.model.forest` at structural

Table 7.1: Overview of the Evaluation Results of Code Fragment 1

Grouping	Ch. op. eq.	Ch. subj. eq.	Cont. eq.	Nr. of generalizations	Nr. of groups	Pattern	TP	FN	FP	Precision	Recall
ImportDeclaration	full	full	exact location	1	9	1	9	0	6	60%	100%
MethodDeclaration	full	full	exact location	0	0	-	0	9	0	N/A	0
Statement	full	full	exact location	0	0	-	0	9	0	N/A	0
ASTNode	full	full	exact location	1	9	1	9	0	48	16%	100%

property `qualifier` of any type of `ASTNode` having this structural property (e.g. `QualifiedName`, `NameQualifiedType`) ". Note that this change pattern can be applied to all `QualifiedName`'s, including nested ones. The compilation unit under investigation contains 57 of such nodes. To conclude, there are 0 false negatives, 9 true positives and 48 false positives with respect to the intuitive pattern, resulting in a recall of $9/(9+0) = 100\%$ and a precision of $9/(9+48) = 16\%$.

The poor precision even when grouping at an `ImportDeclaration` granularity is the result of ChangeNodes reporting insert-operations instead of update-operations: as seen in Chapter 3 update-operations are only emitted when replacing the value of simple property values, which the `qualifier` of a `QualifiedName` is not. An equality-definition taking into consideration characteristics of the overwritten source AST node of insert-operations would improve precision to 100%.

Code Fragment 2

Second, Code Listing 7.2 depicts the rename of the file World.java to Store.java in commit 333ad46 ("Renamed World to Store."). The file contains 76 lines of source code, of which only the shown 5 lines were modified. Differencing source AST and target AST of the compilation unit under investigation results in 5 changes, listed in Table 7.2. Change 1 represents the change of the class name, change 3 corresponds to the corresponding update of the class constructor's name. Change 2 refers to the modification of the static accessor `getCurrent()`. Change 4 corresponds with the change to the static variable `current` and change 5 with the change to the `ClassInstanceCreation` in the static initialization block.

Code Listing 7.2: Exapus, commit 333ad46 ("Renamed World to Store."), modifications to ExapusRAP/src/exapus/model/world/Store.java

```
...

import exapus.model.forest.ExapusModel;
import exapus.model.forest.FactForest;

- public class World {
+ public class Store {

-   private static World current;
+   private static Store current;

    static {
-       current = new World();
+       current = new Store();
    }

-   public static World getCurrent() {
```

```
+   public static Store getCurrent() {
      return current;
    }

-   private World() {
+   private Store() {
      registry = new HashMap<String, FactForest>();
      workspaceModel = new ExapusModel();
      registerForest("apis", workspaceModel.getAPICentricForest());

...
```

Table 7.2: Exapus, commit 333ad46 ("Renamed World to Store."), changes in Exapus-RAP/src/exapus/model/world/Store.java

Change	Op.	Original	Property	Idx	Copy
1	insert	#<TypeDeclaration public class World ... >	name	-	#<SimpleName Store>
2	insert	#<MethodDeclaration public static World getCurrent() ... >	returnType2	-	#<SimpleType Store>
3	insert	#<MethodDeclaration private World() ... >	name	-	#<SimpleName Store>
4	update	#<String World>	identifier	-	#<String Store>
5	update	#<String World>	identifier	-	#<String Store>

The change pattern present intuitively involves renaming the `Name` of the class, its constructor and all referencing `Type`s from `World` to `Store`.

Results are given in Table 7.4. First of all, note that no pattern can be detected when grouping at an `ImportDeclaration` granularity. Since no change is part of a subtree rooted at a node of type `ImportDeclaration`, the constructed transaction database is empty and no patterns are mined.

One important realization regarding the actual changes performed by the developer is that all AST edits are updates of the `identifier` structural property of `SimpleName` nodes. The fact that we receive 3 different kinds of changes (we retrieve 2 insertions of a `SimpleName`, 1 insertion of a `SimpleType` and two updates of `SimpleName` identifiers) is only a consequence of implementational details of the used change distiller. All these changes correspond to the same AST modification, each time applied at a different node.

Grouping at a method declaration granularity results in two groups: a first group with container `#MethodDeclaration getCurrent()>` containing change 2 and a second group with container `#<MethodDeclaration World()>` containing change 3. All other changes are not encompassed by a method declaration and thus not taken in consideration. Since both changes are not equivalent according to the equivalence relation, both singleton groups are disjoint and no pattern is found. Similarly, when grouping at a statement granularity, only one group is formed, containing change 5, the only change performed inside a statement. No patterns will be mined.

Grouping at an AST node granularity and using a full change operation equality, full change subject equality and exact location contextual equality leads to a preprocessing result as shown in Table 7.3. Mining results in two change patterns, both with support 2. The first one indicates the insertion of a `SimpleName Store` at structural property `name` of any `ASTNode`. This pattern covers all 5 nodes to which the intuïtive pattern

Table 7.3: Preprocessing Result of changes in ExapusRAP/src/exapus/model/world/-Store.java when grouping at an AST node granularity and using a full change operation equality, full change subject equality and exact location contextual equality

Change	Operation	Subject	Location	Item
Group #<TypeDeclaration>				
1	Insert	#<SimpleName Store>	[name]	*B*
Group #<getCurrent>				
2	Insert	#<SimpleType Store>	[returnType2]	*C*
Group #<World>				
3	Insert	#<SimpleName Store>	[name]	*B*
Group #<SimpleName>				
4	Update	#<String World>	[identifier]	*A*
Group #<SimpleName>				
5	Update	#<String World>	[identifier]	*A*

Table 7.4: Overview of the Evaluation Results of Code Fragment 2

Grouping	Ch. op. eq.	Ch. subj. eq.	Cont. eq.	Nr. of generalizations	Nr. of groups	Pattern	TP	FN	FP	Precision	Recall
ImportDeclaration	full	full	exact location	0	0	-	0	5	0	N/A	0%
MethodDeclaration	full	full	exact location	0	0	-	0	5	0	N/A	0%
Statement	full	full	exact location	0	0	-	0	5	0	N/A	0%
ASTNode	full	full	exact location	3	5	1	5	0	74	5%	100%
						2	5	0	0	100%	100%

can be applied (0 false negatives), but also all other 95 `ASTNodes` having a structural property `name` (i.e. 5 true positives and 95 false positives). The second one involves updating the structural property `identifier` containing value `World` of any `ASTNode`. This pattern can only be applied to the 5 nodes to which the intuïtive pattern can be applied.

Code Fragment 3

A third code fragment is given in Code Listing 7.3. It depicts the modification of View.java in commit 10c60e3 ("Setters of a view only mark the view's forest and graph as dirty if their argument differs from the current value."). The body of three procedures is wrapped in an `IfStatement` comparing a field variable with the formal parameter with the same name. Furthermore, in method `setMetrics()` an additional call to method `makeDirty()` is inserted. The file under investigation contains 173 lines of code.

Differencing source AST and target AST of the compilation unit results in 12 changes, listed in Table 7.5. Changes 1, 4 7, 8 relate to the modification of method `setRender-able()`, changes 2, 5, 9 and 10 are related to method `setPerspective()` and change 3, 6, 11 and 12 are changes performed to method `setMetrics()`. Please note that the insertion of the call to method `makeDirty()` is represented by a change of type insert for the last method, in the other methods a call was already present resulting in a change of type move.

Code Listing 7.3: Exapus, commit 10c60e3 ("Setters of a view only mark the view's forest and graph as dirty if their argument differs from the current value."), modifications to ExapusRAP/src/exapus/model/view/View.java

```
...
```

```
  public abstract class View {

    ...

    public void setRenderable(boolean renderable) {
-       this.renderable = renderable;
-       makeDirty();
+       if(renderable = this.renderable) {
+         this.renderable = renderable;
+         makeDirty();
+       }
    }

    public void setPerspective(Perspective perspective) {
-       this.perspective = perspective;
-       makeDirty();
+       if(perspective = this.perspective) {
+         this.perspective = perspective;
+         makeDirty();
+       }
    }

    ...

    public void setMetrics(Metrics metrics) {
-       this.metrics = metrics;
+       if(this.metrics = metrics) {
+         this.metrics = metrics;
+         makeDirty();
+       }
    }

    ...
  }
```

Table 7.5: Exapus, commit 10c60e3 ("Setters of a view only mark the view's forest and graph as dirty if their argument differs from the current value.."), changes in Exapus-RAP/src/exapus/model/view/View.java

Change	Op.	Original	rightParent	Property	Idx	Copy
1	insert	#<MethodDeclaration setRenderable()>	-	body	-	#<Block if (renderable != ...) ... >
2	insert	#<MethodDeclaration setPerspective()>	-	body	-	#<Block if (perspective != ...) ... >
3	insert	#<MethodDeclaration setMetrics()>	-	body	-	#<Block if (this.metrics != ...) ... >
4	insert	-	-	statements	0	#<IfStatement if (renderable != ...) ...>
5	insert	-	-	statements	0	#<IfStatement if (perpective != ...) ...>
6	insert	-	-	statements	0	#<IfStatement if (this.metrics != ...) ...>
7	move	#<ExpressionStatement this.renderable=...>	#<Block this.renderable = ...; ...; >	statements	0	#<ExpressionStatement this.renderable=...>
8	move	#<ExpressionStatement makeDirty()>	#<Block this.renderable = ...; ...; >	statements	1	#<ExpressionStatement makeDirty()>
9	move	#<ExpressionStatement this.perspective=...>	#<Block this.perspective = ...; ...; >	statements	0	#<ExpressionStatement this.perspective=...>
10	move	#<ExpressionStatement makeDirty()>	#<Block this.perspective = ...; ...; >	statements	1	#<ExpressionStatement makeDirty()>
11	move	#<ExpressionStatement this.metrics=...>	#<Block this.metrics = ...; ...; >	statements	0	#<ExpressionStatement this.metrics=...>
12	insert	-	-	statements	1	#<ExpressionStatement makeDirty()>

Since the last method from the other methods in two aspects (the left and right operand of the `InfixExpression` are switched and a `makeDirty()` was added and not moved), we intuitively see 2 change patterns: the pattern applied to the first 2 methods, and a second, less complex, pattern applied to all three methods. In the calculations below, we focus on the second pattern.

Results are given in Table 7.6. Once again no pattern can be detected when grouping at an `ImportDeclaration` granularity since no change is part of a subtree rooted at a node of type `ImportDeclaration`, resulting in an empty transaction database.

Table 7.6: Overview of the Evaluation Results of Code Fragment 3

Grouping	Ch. op. eq.	Ch. subj. eq.	Cont. eq.	Nr. of generalizations	Nr. of groups	Pattern	TP	FN	FP	Precision	Recall
ImportDeclaration	full	structural	exact location	0	0	-	0	3	0	N/A	0%
MethodDeclaration	full	full	exact location	10	3	1 (supp. 2)	3	0	3	50%	100%
MethodDeclaration	full	structural	exact location	6	3	1 (supp. 3)	2	1	2	50%	66%
						2 (supp. 2)	2	1	0	100%	66%

Grouping at a method declaration granularity with full change operation equality, full change subject equality and exact location contextual equality yields only one change pattern with support 2, the act of “moving an `ExpressionStatement makeDirty();` from index 1 in the list at structural property `statements` of structural property `body` of a `MethodDeclaration`”. It is expected that full subject equality would not find the entire change pattern performed, since change subjects of changes in the three methods are only structurally equivalent. Six methods have a call to `makeDirty()` as second statement, resulting in 3 true positives and 3 false positives. There are 0 false negatives.

Using a structural subject equality yields two patterns: the first pattern with support 3, is the act “moving an `ExpressionStatement this.<aSimpleName> = <aSimpleName>;` from index 0 in the list at structural property `statements` of structural property `body` of a `MethodDeclaration`”. 4 methods have such a statement as first statement in their body. There are 2 true postiives, 2 false positives and 1 false negative. The other pattern has support 2 and records all modifications performed to both methods `setRenderable()` and `setPerspective()`: as such it matches the following construction: `... ... ...(...) { this.<aSimpleName> = <aSimpleName>; <aSimpleName>(); }`.

Grouping at a `Statement` granularity yields two patterns: the move of an `Expression-Statement this.<aSimpleName> = <aSimpleName>;` at the root of a `Statement` and the move of an `ExpressionStatement makeDirty()` at the root of a `Statement`. Since both patterns are only constituents of a larger pattern at the method declaration level, we have excluded grouping at a Statement granularity from the overview table. The same measurement is taken for grouping at an `ASTNode` granularity.

Code Fragment 4

Finally, a fourth code fragment is given in Code Listing 7.4. It depicts the modification of PackageLayer.java in commit 5e82120 (“Incorporated partial program analysis...”). The return type of three methods is modified from `void` to `Member`, with accompanying introduction of return statements where necessary. Furthermore an unrelated modification is performed: a method `processPartialCompilationunit()` is added. The file under investigation contains 261 lines of code. Differencing source AST and target AST of the compilation unit results in 61 changes, mainly due to the presence of nested inserts and nested moves.

Intuitively, the pattern involves changing the return type of a method declaration from `void` to `Member`, and moving invocations of the method `getOrAddMember()` to the `expression` of a `ReturnStatement`.

Code Listing 7.4: Exapus, commit 5e82120 ("Incorporated partial program analysis..."), modifications to ExapusRAP/src/exapus/model/forest/PackageLayer.java

```
 ...

 public class PackageLayer extends MemberContainer implements ILayerContainer {
   ...

-   public void addBodyDeclaration(BodyDeclaration bd, Stack<ASTNode> scope) {
-     getOrAddMember(UqName.forNode(bd), Element.forNode(bd), scope.iterator());
+   public Member addBodyDeclaration(BodyDeclaration bd, Stack<ASTNode> scope) {
+     return getOrAddMember(UqName.forNode(bd), Element.forNode(bd), scope.iterator());
   }

-   public void addMethodDeclaration(MethodDeclaration md, Stack<ASTNode> scope, IMethodBinding mb) {
+   public Member addMethodDeclaration(MethodDeclaration md, Stack<ASTNode> scope, IMethodBinding mb) {
     if(mb == null) {
-       getOrAddMember(new UqName(md.getName()), Element.forNode(md), scope.iterator());
-       return;
+       return getOrAddMember(new UqName(md.getName()), Element.forNode(md), scope.iterator());
     }
-     getOrAddMember(UqName.forBinding(mb), Element.forNode(md), scope.iterator());
+     return getOrAddMember(UqName.forBinding(mb), Element.forNode(md), scope.iterator());
   }

-   public void addAnonymousClassDeclaration(AnonymousClassDeclaration bd, Stack<ASTNode> scope) {
-     getOrAddMember(UqName.forNode(bd), Element.forNode(bd), scope.iterator());
+   public Member addAnonymousClassDeclaration(AnonymousClassDeclaration bd, Stack<ASTNode> scope) {
+     return getOrAddMember(UqName.forNode(bd), Element.forNode(bd), scope.iterator());
   }

   ...

   public void processPartialCompilationunit(CompilationUnit cu, Set<String> sourcePackageNames) {
     cu.accept(new PartialLayerPopulatingVisitor(this, sourcePackageNames));
   }

   ...
 }
```

Evaluation results are shown in Table 7.7. Grouping at a method declaration granularity with full change operation equality, structural change subject equality and exact location contextual equality yields 2 patterns. The first pattern has support 3, and is based on the change of the insertion of a `SimpleType` at the `returnType2` structural property of a `MethodDeclaration`. Again, due to ChangeNodes reporting an insert, no information on the return type being replaced is provided. As such the pattern can be applied to all 32 method declarations, which leads to 3 true positives, 29 false positives and 0 false negatives. The other pattern is build from generalizing all changes performed to the methods `addBodyDeclaration()` and `addAnonymousClassDeclaration()`: these include not only changing the return type, but also the move of the existing body statement to the `expression` structural property of an inserted `returnStatement`. As such, it requires the statement to exist at the first position in the method body.

Using full change subject equality yields the same results. Indeed, all patterns found using structural subject equality are based on change subjects that are fully equal. Note that the patterns are more complete. For example, the first pattern with support 3 now specifies that the `SimpleType Member` is inserted.

Table 7.7: Overview of the Evaluation Results of Code Fragment 4

Grouping	Ch. op. eq.	Ch. subj. eq.	Cont. eq.	Nr. of generalizations	Nr. of groups	Pattern	TP	FN	FP	Precision	Recall
MethodDeclaration	full	structural	exact location	34	3	1 (supp. 3)	3	0	29	9%	100%
						2 (supp. 2)	2	1	0	100%	66 %
MethodDeclaration	full	full	exact location	34	3	1 (supp. 3)	3	0	29	9%	100%
						2 (supp. 2)	2	1	0	100%	66 %

7.2. How Does the Approach Perform on Open-Source Projects

To answer this question we have mined change patterns in the entire history (all 263 commits) of the open source Exapus project. We grouped changes at a `MethodDeclaration` granularity, using a full change operation equality, structural change subject equality and exact location contextual equality.

The experiment was performed on a standard 13-inch MacBook Pro (Mid 2014) with Retina display, 8GB RAM and 128GB flash storage. Although not a high-end computer, it has 4 logical CPU's (due to Intel Hyper-Threading Technology): it contains an Intel(R) Core(TM) i5-4278U CPU clocked at 2.60GHz. Its level 2 cache provides 256 KB (per core) and it has a 3 MB shared level 3 cache. The central memory consists of 2 4GB 1600MHz DDR3 memory modules, together providing 8GB of memory. Eclipse Luna SR2 (4.4.2), Clojure 1.6 and Java 1.8.0 were used. The benchmark system ran Mac OS X Yosemite version 10.10.4. No other special CPU-intensive software was running. The algorithm mined 403 patterns after distilling 22670 changes in 7 minutes and 42 seconds.

Table 7.8 depicts patterns size in function of support, while Table 7.9 depicts support in function of pattern size. Clearly, pattern size tends to decrease with increasing support: the more complex change patterns are, the less frequent they occur. Furthermore, many patterns with a pattern size of 1 exist. Such patterns involve very simple operations such as changing a `SimpleValue`'s identifier. Based on these observations and due to the size of the mining result, we ordered mined change patterns by frequency times size. We then investigated results manually, starting from the top of the list.

Table 7.8: Pattern Size in Function of Support

		Size					
Support	Count	Mean	Min	Q2	Median	Q3	Max
2	209	4.14	1	1.0	2.0	5.0	28
3	95	3.2	1	1.0	1.0	4.0	24
4	42	3.5	1	1.0	2.0	4.0	23
5	23	4.35	1	1.0	2.0	4.0	22
6	13	3.0	1	1.0	2.0	3.5	14
7	10	1.6	1	1.0	2.0	2.0	2
8	1	2.0	2	2.0	2.0	2.0	2
9	1	1.0	1	1.0	1.0	1.0	1
10	2	1.0	1	1.0	1.0	1.0	1
11	1	1.0	1	1.0	1.0	1.0	1
13	1	1.0	1	1.0	1.0	1.0	1
14	2	1.0	1	1.0	1.0	1.0	1
39	3	1.0	1	1.0	1.0	1.0	1

Table 7.9: Support in Function of Pattern Size

		Support					
Size	Count	Mean	Min	Q2	Median	Q3	Max
1	175	3.91	2	2.0	3.0	4.0	39
2	73	3.16	2	2.0	2.0	4.0	8
3	34	2.91	2	2.0	2.0	4.0	6
4	32	2.97	2	2.0	2.5	4.0	6
5	21	2.57	2	2.0	2.0	3.0	6
6	16	2.19	2	2.0	2.0	2.0	3
7	10	2.6	2	2.0	2.0	3.25	4
8	1	4.0	4	4.0	4.0	4.0	4
9	3	2.0	2	2.0	2.0	2.0	2
10	4	2.25	2	2.0	2.0	2.75	3
11	3	2.33	2	2.0	2.0	3.0	3
12	6	3.0	2	2.0	2.0	5.0	5
13	4	3.0	2	2.0	3.0	4.0	4
14	3	4.0	3	3.0	3.0	6.0	6
15	1	2.0	2	2.0	2.0	2.0	2
17	2	2.0	2	2.0	2.0	2.0	2
18	3	3.0	2	2.0	2.0	5.0	5
21	2	2.0	2	2.0	2.0	2.0	2
22	1	5.0	5	5.0	5.0	5.0	5
23	2	3.5	3	3.0	3.5	4.0	4
24	1	2.0	2	2.0	2.0	2.0	2
25	1	3.0	3	3.0	3.0	3.0	3
26	2	2.0	2	2.0	2.0	2.0	2
28	1	2.0	2	2.0	2.0	2.0	2

The first pattern in the list is mined from commit 7f9fa73 ("Huge speed improvement in view calculation"). Here, systematic-repetitive changes were applied to 5 methods in file ExapusRAP/src/exapus/model/visitors/FastSelectiveCopyingForestVisitor.java. Code Listing 7.5 lists diff output for one of these methods, all other methods are modified in a similar way.

Code Listing 7.5: Exapus, commit 7f9fa73 ("Huge speed improvement in view calculation"), modifications to `applyTags(Member copy)` of ExapusRAP/src/exapus/model/visitors/FastSelectiveCopyingForestVisitor.java

```
- private boolean applyTags(Member copy) {
+ private boolean applyTags(Member copy, FactForest sourceForest, Member original) {
-   Iterator<Selection> i = selections.iterator();
-   while(i.hasNext()) {
-     Selection selection = i.next();
+   for(Selection selection :  selections) {
      if(selection.hasTag()) {
-       if(selection.matches(copy)) {
-         if(copy.addTag(selection.getTag()))
-           //re-iterate when a new tag has been added
-           //i = selections.iterator();
-           ;
+       if(selection.matches(original,sourceForest)) {
+         forestCopy.addTag(copy, selection.getTag());
        }
      }
    }
    return true;
  }
```

The second pattern in the list is mined from commit ee819f7 ("changed API/Project selection field of View to list"). Here, systematic-repetitive changes were applied to 6 methods over the two files ExapusRAP/src/exapus/model/view/evaluator/API-CentricEvaluator.java and ExapusRAP/src/exapus/model/view/evaluator/ProjectCentricEvaluator.java. Code Listing 7.6 lists diff output for one of these methods, once again all other methods are modified in a similar way.

Code Listing 7.6: Exapus, commit ee819f7 ("changed API/Project selection field of View to list"), modifications to `select(PackageLayer packageLayer)` of ExapusRAP/src/exapus/model/view/evaluator/APICentricEvaluator.java

```
     @Override
-    protected boolean select(PackageLayer packageLayer) {
+    protected boolean select(final PackageLayer packageLayer) {
-      return APISelection.matchAPIPackageLayer(packageLayer);
+      return Iterables.any(selections, new Predicate<Selection>() {
+        @Override
+        public boolean apply(Selection selection) {
+          return selection.matchAPIPackageLayer(packageLayer);
+        }
+      });
     }
```

Since both top patterns have a relatively low frequency, we continue the investigation through the list only considering patterns with a support greater than 10. Only 7 patterns fullfil this requirement and all these patterns are based on only one change generalization. We especially focus on the first 2 patterns with support 39.

The first pattern occurs in commit 7f9fa73 ("Huge speed improvement in view calculation"), where 39 methods spread over 12 files receive a second formal parameter (e.g. `FactForest source`). To illustrate, Code listing 7.7 depicts the changes performed to one of these files. The other pattern can be found in commit 4060e8e ("Revert to Kates' unmerged commit"). Here, all 39 modifications from commit 7f9fa73 are undone: the inverse operation was applied to exactly the same set of methods.

Code Listing 7.7: Exapus, commit 7f9fa73 ("Huge speed improvement in view calculation"), modifications to ExapusRAP/src/exapus/model/view/UniversalSelection.java

```
  import javax.xml.bind.annotation.XmlRootElement;
  import javax.xml.bind.annotation.XmlType;

+ import exapus.model.forest.FactForest;
  import exapus.model.forest.ForestElement;
  import exapus.model.forest.Member;
  import exapus.model.forest.PackageLayer;

  ...

  @XmlRootElement
  public class UniversalSelection extends Selection {
    ...

    @Override
-   public boolean matches(Ref ref) {
+   public boolean matches(Ref ref, FactForest source) {
      return true;
    }

    @Override
-   public boolean matches(PackageTree packageTree) {
+   public boolean matches(PackageTree packageTree, FactForest source) {
      return true;
    }

    @Override
-   public boolean matches(PackageLayer packageLayer) {
+   public boolean matches(PackageLayer packageLayer, FactForest source) {
      return true;
    }

    @Override
-   public boolean matches(Member member) {
+   public boolean matches(Member member, FactForest source) {
      return true;
    }

    @Override
-   public boolean matches(ForestElement e) {
+   public boolean matches(ForestElement e, FactForest source) {
      return true;
    }

    ...
  }
```

7.3. Summary

First, we investigated whether our approach can recall known change patterns from small code fragments to which systematic-repetitive changes are applied. To this extend, we first identified 4 code fragments, then defined intuitively which change pattern is present. We then investigated the mining result under a variety of grouping granularities and equivalence relations. To conclude, the approach was able to identify all patterns with a precision and recall highly dependent on the used algorithm parameters.

Second, we analyzed performance of the approach on the entire history of open-source projects, i.e. we have determined its ability to derive useful previously unknown change patterns given a large database of code and code changes. We have mined 403 change patterns in an open source project containing 263 commits, with the last commit containing 13198 lines of java code. Due to the size of the mining result and the dominance of relatively small change patterns occurring often, we sorted change patterns by size times support. All mined change patterns did correspond with systematic-repetitive high-level program transformations.

8 Related Work

This work is located in the field of software evolution, more specifically the discipline of software repository mining. In this chapter, we briefly explore related work. We especially present an overview of other approaches to change pattern mining, listing differences with our approach and discussing advantages and disadvantages.

This dissertation is mainly based upon Negara et al. (2014), which uses frequent closed itembag mining with overlapping transactions to identify previously unknown frequent code change patterns from a fine-grained sequence of code changes.

A change logger captures change data from the IDE as changes are performed, and provides this data to a mining algorithm. This algorithm is based on CHARM (Mohammed J. Zaki & Hsiao, 2002), an algorithm for closed frequent itemset mining. In order to allow code change pattern mining, the CHARM algorithm was however modified to allow overlapping transactions, making it possible to process a continuous sequence of code changes ordered by timestamp, without any previous knowledge of where the boundaries between patterns of transformations are. Furthermore, the algorithm was updated to allow itembags instead of itemsets. This was necessary since a high-level program transformation may contain several instances of the same kind of code changes.

The algorithm is able to identify repetitive code change patterns that may correspond to some previously unknown high-level program transformations. Evaluation of the algorithm showed effectiveness, usefulness and scalability by running the algorithm on previously collected data involving 23 participants with 1520 hours of development. More precisely, CodingTracker recorded IDE data and an AST node operations inference algorithm represented code changes as add, delete and update operations on the underlying AST. Feeding the miner with this data resulted in a set of change patterns together with all occurrences of each pattern. They ordered the patterns by frequency and size and performed a manual investigation of a small fraction the detected patterns, involving filtering out certain results. As such, they have identified 10 kinds of previously unknown high-level program transformations.

Instead of relying on a change logger to provide change data, we collect change data by comparing version control snapshots. Consecutively, we use standard frequent itemset mining instead of frequent closed itembag mining with overlapping transactions. Also, our approach allows automatic transformation of source code according to a change pattern.

Negara et al. (2014) is not the only work applying standard data mining approaches to source code and source code changes. Both Ying, Murphy, Ng and Chu-Carroll (2004) and Zimmerman, Weißgerber, Diehl and Zeller (2004) address the difficulty developers face to find relevant source code fragments for a certain modification. In Ying et al. (2004), change patterns are defined as files that have changed together frequently enough. A recommender tool tracks files changed by the developer: if a developer changes a set f_S of files, the approach recommends a set of files f_R that will likely need modification too. To allow recommendation, association rule mining is used over pre-processed CVS data. Zimmerman et al. (2004) uses the same idea but allows working below file-level granularity. Mulder and Zaidman (2010) proposed a method for identifying cross-cutting concerns in software systems by use of software repository mining. As cross-cutting concerns are scattered throughout the code base, modification of cross-cutting concerns requires changes in multiple files. This allows the use of frequent itemset mining for finding files that are often changed together, i.e. that were frequently committed simultaneously. Next to this file-level mining a more expensive but more fine-grained method is proposed for mining methods that are frequently changed together. Li, Lu, Myagmar and Zhou (2004) proposes CP-miner, a tool for finding copy-paste and related bugs in operating system code. CP-Miner first identifies copy-pasted code and then performs bug finding. To achieve the former, the program is parsed, resulting in a stream of tokens. Tokens are then assigned numerical values, which allows formulation of the problem of identifying copy-pasted code as a sequential pattern mining problem on this stream of numerical values.

All of these approaches provide identification and recommendation but do not allow automatically modifying source code based on earlier code changes.

9 Conclusion and Future Work

9.1. Summary

Software evolution implies performing changes to source code. Code duplication due to code reuse leads to similar but not necessarily identical code, resulting in similar but not necessarily identical code changes. Such systematic-repetitive code changes form change patterns. As manually editing code systematically is laborious and error-prone, scientists and developers can both benefit from learning and automating previously unknown change patterns.

In this thesis, we formulated the problem of finding previously unknown automatable change patterns in the history of software projects as a data mining problem. More concretely, we applied the frequent itemset mining algorithm, operating on the changes distilled from successive VCS snapshots, looking for automatable change patterns in software repositories such as open-source GitHub repositories.

To this end we use a sequential approach, consisting of five consecutive phases. We start by extracting two versions of a modified fragment of source code from the software history contained within a VCS. In the second phase we extract fine-grained changes, represented by concrete AST operations, made between these two version by applying a change distiller on the ASTs corresponding with both versions. Preprocessing of distilled changes in the form of grouping and equivalence relation-driven generalization builds a mineable transaction database: during grouping changes are classified in disjoint sets by the subtree in which they occur, while generalization is about considering certain changes equivalent if and only if they share certain commonalities. After preprocessing, a frequent itemset mining algorithm mines automatable change patterns. These patterns can then allow program querying or serve as input of program transformation tools.

As an implementation of the proposed approach, we have written an Eclipse plugin, usable via the Clojure REPL. This implementation uses ChangeNodes to distill changes in successive git snapshots of Java source code. After preprocessing, the CHARM closed frequent itemset mining algorithm is used to mine automatable change patterns. As an illustration of tool support, we provided a program querying solution based on Ekeko/X templates, allowing the identification of candidate change pattern instances in new source code.

We have evaluated our implementation by answering two research questions. The first question relates to whether the approach can recall known change patterns? To conclude, the algorithm is able to recall known change patterns, with performance highly dependent on the used grouping granularity and equivalence relation. The second question is about performance on open-source projects. Mined change patterns did correspond with systematic-repetitive high-level program transformations. However, most mined change patterns are small in size.

9.2. Contribution

The contribution of this thesis is twofold. First, we proposed a general-purpose method and tool for finding previously unknown automatable change patterns from distilled changes. Second, we investigated and evaluated the impact of change generalization on automatable change pattern mining.

9.3. Future Work

Improved Tool Support

We have presented a method for finding new source code fragments to which change patterns can be applied. However, although we focus on automatable change patterns in this work, developers still have to manually transform identified code fragments, a process which is highly repetitive and thus prone to human error. Developers can benefit from a method to automatically apply mined change patterns to new source code, similar to the automatic application of refactorings provided by modern IDEs. A first line of future work involves improving tool support to support automatic transformation of program fragments as described by mined change patterns. As a practical application, library and framework developers can provide transformations whenever the API evolves, automatically updating client applications when applied.

Due to late awareness of refactoring, almost all refactorings are still performed by hand (Murphy-Hill et al., 2009). To reduce tool underuse, Ge, Dubose and Murphy-Hill (2012) proposes BeneFactor, a proof-of-concept refactoring tool that detects when the developer is refactoring code, thus solving the late awareness problem of refactoring. This idea can be extended to previously unknown change patterns: although the current approach mines change patterns between successive VCS snapshots, the same algorithm can be applied to any two snapshots of source code and as such, it should be possible to incorporate on-the-fly change pattern mining in tools, informing the developer whenever he is performing systematic-repetitive changes manually. Such tools can then guide the developer by highlighting other code fragments which might still need modification or by checking correctness of performed manual modifications. Furthermore, it may be possible to automate the remaining code modifications.

Explore other languages, VCSs and Change Distillers

In order to achieve a reduction of engineering effort, we have only implemented the mining of git repositories for change patterns in Java source code. Future work can cover other VCSs (e.g. Apache SVN, CVS) and other programming languages. Especially, a language-agnostic representation of ASTs can help to achieve support for multiple programming languages.

Furthermore, we used ChangeNodes as differencing tool. The impact of the use of other program differencing algorithms such as Gumtree is left as future work.

Towards Change Pattern Taxonomies

A third line of future works relates to a further study of mined change patterns, classifying systematic-repetitive changes and investigating their popularity on a cross-project basis. This helps scientists to get a better understanding of how code evolves and allows IDE developers and tool builders to provide built-in functionality to automate popular transformations beyond well-known refactorings.

Bibliography

Agrawal, R., Imieliński, T. & Swami, A. (1993). Mining Association Rules between Sets of Items in Large Databases. In P. Buneman & S. Jajodia (Eds.), *Proceedings of the ACM SIGMOD International Conference on Management of Data 1993* (pp. 207–216). May 26–28, 1993, Washington, D.C., US.

Agrawal, R. & Srikant, R. (1994). Fast Algorithms for Mining Association Rules. In J. B. Bocca, M. Jarke & C. Zaniolo (Eds.), *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB) 1994* (pp. 487–499). September 12–15, 1994, Santiago, CL.

Agrawal, R. & Srikant, R. (1995). Mining Sequential Patterns. In P. S. Yu & A. L. P. Chen (Eds.), *Proceedings of the 11th International Conference on Data Engineering (ICDE) 1995* (pp. 3–14). March 6–10, 1995, Taipei, TW.

Casas-Garriga, G. (2003). Discovering Unbounded Episodes in Sequential Data. In N. Lavrac, D. Gamberger, H. Blockeel & L. Todorovski (Eds.), *Proceedings of the 7th European Conference on the Principles and Practice of Knowledge Discovery in Databases (PKDD) 2003* (pp. 83–94). September 22–26, 2003, Cavtat-Dubrovnik, HR.

Chawathe, S. S., Rajaraman, A., Garcia-Molina, H. & Widom, J. (1996). Change Detection in Hierarchically Structured Information. In P. Jalote, L. C. Briand & A. van der Hoek (Eds.), *Proceedings of the ACM SIGMOD International Conference on Management of Data 1996* (pp. 493–504). June 4–6, 1996, Montreal, QC, CA.

De Roover, C. & Inoue, K. (2014). The Ekeko/X Program Transformation Tool. In *Proceedings of the 14th International Working Conference on Source Code Analysis and Manipulation (SCAM) 2014* (pp. 53–58). September 28–29, 2014, Victoria, BC, CA.

De Roover, C. & Stevens, R. (2014). Building Development Tools Interactively using the EKEKO Meta-Programming Library. In S. Demeyer, D. Binkley & F. Ricca (Eds.), *Proceedings of the IEEE conference on Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week* (pp. 429–433). February 3–6, 2014, Antwerp, BE.

Ducasse, S., Rieger, M. & Demeyer, S. (1999). A Language Independent Approach for Detecting Duplicated Code. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM) 1999* (pp. 109–118). August 30 – September 3, 1999, Oxford, England, UK.

Falleri, J., Morandat, F., Blanc, X., Martinez, M. & Montperrus, M. (2014). Fine-Grained and Accurate Source Code Differencing. In *Proceedings of the ACM/IEEE 29th International Conference on Automated Software Engineering (ASE) 2014* (pp. 313–324). September 15–19, 2014, Vasteras, SE.

Farach, M. (1997). Optimal Suffix Tree Construction with Large Alphabets. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS) 1997* (pp. 137–143). October 19–22, 1997, Miami Beach, FL, USA.

Fluri, B., Wursch, M., Pinzger, M. & Gall, H. C. (2007). Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering*, *33*(11), 725–743.

Fowler, M. (1999). *Refactoring. Improving the Design of Existing Code.* Addison-Wesley.

Ge, X., Dubose, Q. L. & Murphy-Hill, E. R. (2012). Reconciling Manual and Automatic Refactoring. In M. Glinz, G. C. Murphy & M. Pezzè (Eds.), *Proceedings of the 34th International Conference on Software Engineering (ICSE) 2012* (pp. 211–221). June 2–9, 2012, Zurich, CH.

Griswold, W. G. (1991). *Program Restructuring as an Aid to Software Maintenance* (Doctoral dissertation, Department of Computer Science and Engineering, University of Washington).

Han, J., Pei, J., Yin, Y. & Mao, R. (2004). Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach. *Data Mining and Knowledge Discovery*, *8*(1), 53–87.

Houtsma, M. A. W. & Swami, A. N. (1995). Set-Oriented Mining for Association Rules in Relational Databases. In P. S. Yu & A. L. P. Chen (Eds.), *Proceedings of the 11th International Conference on Data Engineering (ICDE) 1995* (pp. 25–33). March 6–10, 1995, Taipei, TW.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., … Irwin, J. (1997). Aspect-Oriented Programming. In M. Aksit & S. Matsuoka (Eds.), *Proceedings of the European Conference on Object-Oriented Programming (ECOOP) 1997* (pp. 220–242). June 9–13, 1997, Jyväskylä, FI.

Kim, S., Pan, K. & Whitehead, J., E. J. (2006). Memories of Bug Fixes. In *Proceedings of the ACM SIGSOFT 14th International Symposium on the Foundations of Software Engineering (FSE) 2014* (pp. 35–45). November 5–11, 2006, Portland, Oregon, US.

Krinke, J. (2007). Changes to Code Clones in Evolving Software. *Softwaretechnik-Trends*, *27*(2), 131–183.

Krueger, C. W. (1992). Software Reuse. *ACM Computing Surveys*, *24*(2), 131–183.

Laxman, S., Sastry, P. S. & Unnikrishnan, K. P. (2004). Fast Algorithms for Frequent Episode Discovery in Event Sequences. In *Proceedings of the 3rd Workshop on Mining Temporal and Sequential Data (TDM) 2004*. August 22–24, 2004, Seattle, WA, US.

Laxman, S., Sastry, P. S. & Unnikrishnan, K. P. (2007). A Fast Algorithm for Finding Frequent Episodes in Event Streams. In P. Berkhin, R. Caruana & X. Wu (Eds.), *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge*

Discovery and Data Mining (KDD) 2007 (pp. 410–419). August 12–15, 2007, San Jose, CA, US.

Lee, S. D. & De Raedt, L. (2005). An Efficient Algorithm for Mining String Databases Under Constraints. In B. Goethals & A. Siebes (Eds.), *Proceedings of the 3rd International Workshop on Knowledge Discovery in Inductive Databases (KDID) 2004* (pp. 108–129). September 20, 2004, Pisa, IT.

Li, Z., Lu, S., Myagmar, S. & Zhou, Y. (2004). CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In E. A. Brewer & P. Chen (Eds.), *Proceedings of the 6th Symposium on Operating System Design and Iimplementation (OSDI) 2004* (pp. 289–302). December 6–8, 2004, San Francisco, CA, US.

Manber, U. & Myers, G. (1990). Suffix Arrays: A New Method for On-Line String Searches. In D. S. Johnson (Ed.), *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms 1990* (pp. 319–327). January 22–24, 1990, San Francisco, CA, US.

Manilla, H. & Toivonen, H. (1996). Discovering Generalized Episodes using Minimal Occurrences. In E. Simoudis, J. Han & U. M. Fayyad (Eds.), *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining (KDD) 1996* (pp. 146–151). August 2–4, 1996, Portland, OR, US.

Mannila, H., Toivonen, H. & Verkamo, A. I. (1995). Discovering Frequent Episodes in Sequences. In U. M. Fayyad & R. Uthurusamy (Eds.), *Proceedings of the 1st International Conference on Knowledge Discovery and Data Mining (KDD) 1995* (pp. 210–215). August 20–21, 1995, Montreal, CA.

Mannila, H., Toivonen, H. & Verkamo, A. I. (1997). Discovery of Frequent Episodes in Event Sequences. *Data Mining and Knowledge Discovery*, *1*(3), 259–289.

McCreight, E. M. (1976). A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM*, *23*(2), 262–272.

Mulder, F. & Zaidman, A. (2010). Identifying Cross-cutting Concerns using Software Repository Mining. In A. Capiluppi, A. Cleve & N. Moha (Eds.), *Proceedings of the joint ERCIM Workshop on Software Evolution (EVOL) and Internation Workshop on Principles of Software Evolution (IWPSE) 2010* (pp. 23–32). September 20–21, 2010, Antwerp, BE.

Munoz, F., Baudry, B., Delamare, R. & Le Traon, Y. (2009). Inquiring the Usage of Aspect-Oriented Programming: An Empirical Study. In *Proceedings of the 25th IEEE International Conference on the Software Maintainance (ICSM) 2009* (pp. 137–146). September 20–26, 2012, Edmonton, AB, CA.

Murphy-Hill, E. R., Chris, P. & Black, A. P. (2009). How We Refactor, and How We Know It. In *Proceedings of the 31st International Conference on Software Engineering (ICSE) 2009* (pp. 287–297). May 16–24, 2009, Vancouver, BC, CA.

Myers, E. W. (1986). An O(ND) Difference Algorithm and Its Variations. *Algorithmica*, *1*(1–4), 251–266.

Negara, S., Codoban, M., Dig, D. & Johnson, R. E. (2014). Mining Fine-Grained Code Changes to Detect Unknown Change Patterns. In P. Jalote, L. C. Briand & A. van der Hoek (Eds.), *Proceedings of the 36th International Conference on Software Engineering (ICSE) 2014* (pp. 803–813). May 31 – June 7, 2014, Hyderabad, India.

Negara, S., Vakilian, M., Chen, N., Johnson, R. E. & Dig, D. (2012). Is It Dangerous to Use Version Control Histories to Study Source Code Evolution? In J. Noble (Ed.), *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP) 2012* (pp. 79–103). June 11–16, 2012, Beijing, CN.

Nguyen, T. T., Nguyen, H. A., Pham, N. H., Al-Kofahi, J. & Nguyen, T. N. (2010). Recurring Bug Fixes in Object-Oriented Programs. In J. Kramer, J. Bishop, P. T. Devanbu & S. Uchitel (Eds.), *Proceedings of the ACM/IEEE 32nd International Conference on Software Engineering (ICSE) 2010 - volume 1* (pp. 315–324). May 1–8, 2010, Cape Town, ZA.

Opdyke, W. F. (1992). *Refactoring Object-Oriented Frameworks* (Doctoral dissertation, University of Illinois at Urbana-Champaign).

Pasquier, N., Bastide, Y., Taouil, R. & Lakhal, L. (1999). Discovering Frequent Closed Itemsets for Assocation Rules. In C. Beeri & P. Buneman (Eds.), *Proceedings of the 7th International Conference on Database Theory (ICDT) 1999* (pp. 398–416). January 10–12, 1999, Jerusalem, IL.

Pei, J., Han, J. & Mao, R. (2000). CLOSET: An Efficient Algorithm for Mining Frequent Closed Itemsets. In U. M. Fayyad & R. Uthurusamy (Eds.), *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD) 2000* (pp. 21–30). May 14, 2000, Dallas, TX, US.

Pei, J., Han, J., Mortazavi-Asl, B., Pinto, H., ... Hsu, M. (2001). PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth. In D. Georgakopoulos & A. Buchmann (Eds.), *Proceedings of the 17th International Conference on Data Engineering (ICDE) 2001* (pp. 215–224). April 2–6, 2001, Heidelberg, DE.

Ray, B. & Kim, M. (2012). A Case Study of Cross-System Porting in Forked Projects. In W. Tracz, M. P. Robillard & T. Bultan (Eds.), *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE) 2012* (53:1–53:11). November 11–16, 2012, Cary, NC, US.

Srikant, R. & Agrawal, R. (1996). Mining Sequential Patterns: Generalizations and Performance Improvements. In P. M. G. Apers, M. Bouzeghoub & G. Gardarin (Eds.), *Proceedings of the 5th International Conference on Extending Database Technology (EDBT) 1996* (pp. 3–17). March 25–29, 1996, Avignon, FR.

Stevens, R. (2015). A Declarative Foundation for Comprehensive History Querying. In *ACM/IEEE 37th International Conference on Software Engineering (ICSE) 2015*. May 16–24, 2015, Florence, IT.

Tatti, N. (2009). Significance of Episodes Based on Minimal Windows. In W. Wang, H. Kargupta, S. Ranka, P. S. Yu & X. Wu (Eds.), *Proceedings of the 9th IEEE International Conference on Data Mining (ICDM) 2009* (pp. 513–522). December 6–9, 2009, Miami, FL, US.

Tatti, N. & Cule, B. (2011). Mining Closed Episodes with Simultaneous Events. In C. Apté, J. Ghosh & P. Smyth (Eds.), *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD) 2011* (pp. 1172–1180). August 21-24, 2011, San Diego, CA, US.

Tsuboi, Y. (2003). *Mining Frequent Substring Patterns with Ternary Partitioning.* IBM.

Ukkonen, E. (1995). Onine Construction of Suffix Trees. *Algorithmica*, *14*(3), 249–260.

Wagner, R. A. & Fischer, M. J. (1974). The String-to-String Correction Problem. *Journal of the ACM*, *21*(1), 168–173.

Wang, J. & Han, J. (2004). BIDE: Efficient Mining of Frequent Closed Sequences. In Z. M. Özsoyoglu & S. B. Zdonik (Eds.), *Proceedings of the 20th International Conference on Data Engineering (ICDE) 2004* (pp. 79–90). March 30 – April 2, 2004, Boston, MA, US.

Wang, J., Han, J. & Pei, J. (2003). CLOSET+: Searching for the Best Strategies for Mining Frequent Closed Itemsets. In L. Getoor, T. E. Senator, P. M. Domingos & C. Faloutsos (Eds.), *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD) 2003* (pp. 236–245). August 24–27, 2003, Washington, D.C., US.

Weiner, P. (1973). Linear Pattern Matching Algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (SWAT) 1973* (pp. 1–11). October 15–17, 1973, Iowa City, IA, USA.

Yan, X., Han, J. & Afshar, R. (2003). CloSpan: Mining Closed Sequential Patterns in Large Datasets. In D. Barbará & C. Kamath (Eds.), *Proceedings of the 3th SIAM International Conference on Data Mining (SDM) 2003* (pp. 166–177). May 1–3, 2003, San Francisco, CA, US.

Ying, A. T. T., Murphy, G. C., Ng, R. T. & Chu-Carroll, M. (2004). Predicting Source Code Changes by Mining Change History. *The IEEE Transactions on Software Engineering (TSE)*, *30*(9), 574–586.

Zaki, M. J. [M. J.]. (2000). Scalable Algorithms for Association Mining. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, *12*(3), 372–390.

Zaki, M. J. [M. J.]. (2001). SPADE: An Efficient Algorithm for Mining Frequent Sequences. *Machine Learning*, *42*(1/2), 31–60.

Zaki, M. J. [M. J.] & Gouda, K. (2003). Fast Vertical Mining Using Diffsets. In L. Getoor, T. E. Senator, P. M. Domingos & C. Faloutsos (Eds.), *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD) 2003* (pp. 326–335). August 24–27, 2003, Washington, D.C., US.

Zaki, M. J. [Mohammed J.] & Hsiao, C.-j. (2002). CHARM: An Efficient Algorithm for Closed Itemset Mining. In R. L. Grossman, J. Han, V. Kumar, H. Mannila & R. Motwani (Eds.), *Proceedings of the 2nd SIAM International Conference on Data Mining (SDM) 2002* (pp. 457–473). April, 11–13, 2002, Arlington, VA, US.

Zhou, W., Liu, H. & Cheng, H. (2010). Mining Closed Episodes from Event Sequences Efficiently. In M. J. Zaki, J. X. Yu, B. Ravindran & V. Pudi (Eds.), *Proceedings of the 14th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining (PAKDD) 2010* (pp. 310–318). June 21–24, 2010, Hyderabad, IN.

Zimmerman, T., Weißgerber, P., Diehl, S. & Zeller, A. (2004). Mining Version Histories to Guide Software Changes. In A. Finkelstein, J. Estublier & D. S. Rosenblum (Eds.), *Proceedings of the 26th International Conference on Software Engineering (ICSE) 2004* (pp. 563–572). May 23–28, 2004, Edinburgh, UK.