Analyzing and Inferring the Structure of Code Changes

Miryung Kim

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

2008

Program Authorized to Offer Degree: Computer Science and Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Miryung Kim

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Chair of the Supervisory Committee:

_____

David Notkin

Reading Committee:

_____

David Notkin

_____

Daniel Grossman

_____

Robert DeLine

Date: _____

In presenting this dissertation in partial fulfillment of the requirements for the doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with ?fair use? as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to Proquest Information and Learning, 300 North Rebe Road, Ann Arbor, MI 48106-1346, 1-800-521-0600, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature_____

Date_____

University of Washington

Abstract

Analyzing and Inferring the Structure of Code Changes

Miryung Kim

Chair of the Supervisory Committee:
Professor David Notkin
Computer Science & Engineering

Programmers often need to reason about how a program evolved between two or more program versions. Reasoning about program changes is challenging as there is a significant gap between how programmers think about changes and how existing program differencing tools represent such changes. For example, even though modification of a locking protocol is conceptually simple and systematic at a code level, *diff* extracts scattered text additions and deletions per file.

To enable programmers to reason about program differences at a high-level, this dissertation proposes an approach that automatically discovers and represents systematic changes as first order logic rules. This rule inference approach is based on the insight that high-level changes are often systematic at a code level and that first order logic rules can represent such systematic changes concisely. There are two similar but separate rule-inference techniques, each with its own kind of rules. The first kind captures systematic changes to application programming interface (API) names and signatures. The second kind captures systematic differences at the level of code elements (e.g., types, methods, and fields) and structural dependencies (e.g., method-calls and subtyping relationships).

Both kinds of rules *concisely represent systematic changes* and *explicitly note exceptions* to systematic changes. Thus, software engineers can quickly get an overview of program differences and identify potential bugs caused by inconsistent updates. The viability of this approach is demonstrated through its application to several open source projects as well as

a focus group study with professional software engineers from a large e-commerce company.

This dissertation also presents empirical studies that motivated the rule-based change inference approach. It has been long believed that code clones—syntactically similar code fragments—indicate poor software design and that refactoring code clones improves software quality. By focusing on the *evolutionary aspects* of clones, this dissertation discovered that, in contrast to conventional wisdom, programmers often create and maintain code duplicates with clear intent and that immediate and aggressive refactoring may not be the best solution for managing code clones. The studies also contributed to developing the insight that a high-level change operation comprises systematic transformations at a code level and that identification of such systematicness can help programmers better understand code changes and avoid inconsistent updates.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

My graduate career would have not been possible without support from my family, mentors, colleagues, and friends.

First of all, I am grateful to my parents for their love and support. They taught me the importance of passion and persistence in everything I do. I thank my sisters for their love and encouragement.

I thank my adviser, David Notkin, for giving me a chance to select research problems and pursue them. David believed in my potential and saw creativity in me. Through the process of discussing many research proposals with David, I learned how to assess the research value of each proposal and how to be selective. He is a patient and thoughtful mentor; he *listens* to his students, not only what they are saying but also what they are not saying. I am proud to be one of his students and I am grateful to know that I can continue to rely on him.

I thank Dan Grossman for his support and advocacy. His advice as a programming language researcher was very valuable to this dissertation and he challenged me to hold high standards in conveying algorithms and formal definitions.

I thank Gail Murphy for her advice and encouragement. She always made time to listen to my research ideas. I learned so much from her insights into software engineering practices. I admire her diligence and organizational skills.

I thank Rob DeLine for his valuable comments on my research ideas from industry research perspectives.

I thank Andreas Zeller for his advice and advocacy. I would like to emulate his passion for software engineering research.

I thank the members of my committee: David Notkin, Dan Grossman, Rob DeLine and Ken Bube.

Vibha Sazawal has been a wonderful mentor and friend for me. I was lucky to have the opportunity to learn from her as a junior graduate student. More importantly, I am always inspired by her passion for computer science education and gender equality. Vibha taught me that diversity is a vital ingredient for creativity and thus a catalyst for moving our field forward.

I thank Tessa Lau and Larry Bergman for giving me a chance to do such a cool research internship with them at IBM. I learned a great deal about how to push a research agenda from Tessa. I thank them for being my mentors and friends.

I thank Tao Xie, Jonathan Aldrich, Mike Ernst, Kevin Sullivan, and Bill Griswold for discussing research with me and advising me throughout my career. I am proud to belong to Notkin's academic family.

I thank Jonathan Beall for implementing LSDiff tool, Marius Nita for helping me with the focus group study and many fun discussions, and Stanley Kok for helping me use Alchemy.

I thank Annie Ying, Sunghun Kim, Thomas Zimmermann, Reid Holmes, Beat Fluri, Martin Robillard, Tao Xie, and Jim Whitehead, and Andreas Zeller. I always enjoy discussing and brainstorming research ideas with you.

I thank Jennifer Bevan, Sunghun Kim, Peter Weißgerber, Zhenchang Xing, and Kris De Volder for their tools and data sets.

Thanks to all my office mates and UW CSE friends throughout my graduate career, including Yongchul Kwon, Stephen Friedman, Pradeep Shenoy, Zizhen Yao, Jiwon Kim, Jayant Madhavan, Tammy VanDeGrift, Vibha Sazawal, Tao Xie, Andrew Petersen, Marius Nita, Kate Moore, Ben Lerner, Laura Effinger-Dean, and countless others for their friendship, encouragement, and help. I learned a great deal from your comments during all my practice talks.

I thank everyone who helped me during my job interviews in Spring 2008: David Notkin, Dan Grossman, Gail Murphy, Andreas Zeller, Rob DeLine, Vibha Sazawal, Hank Levy, James Landay, Craig Chambers, Jaeyeon Jung, James Fogarty, and many others.

I thank Doo-Hwan Bae for his advice throughout my undergraduate days.

I thank Cathy Dong, Jackie Yang, Sammy Lee and many NCBC friends who prayed for me. I thank Rob Lendvai and SSHS friends for all the fun we had together.

I thank my husband, Alan Ho. He is the source of my happiness, love, strength and courage. My life is so much richer because of his love and support. (Thanks go to Alan for also proofreading this dissertation.) Finally, I thank God for His love.

Chapter 1

INTRODUCTION

Software evolution plays an ever-increasing role in software development. Programmers constantly update existing software to provide new features to customers or to fix defects. As software evolves, programmers often need to inspect program differences between two versions or its generational changes over multiple versions. For example, for a team lead to check whether the intended change is implemented correctly, she needs to review the modifications done by her team members. As another example, when a program behaves differently from expected behavior after several modifications, programmers inspect past code changes. In addition to these scenarios from a programmer's perspective, software engineering researchers also need to reason about program differences for software version merging, profile propagation, regression testing, change impact analysis, and software evolution analysis.

When inspecting program differences, programmers or researchers may ask the following kinds of high-level questions about code changes: "What changed?" "Is anything missing in that change?" and "Why did this set of code fragments change together?"

To enable programmers to reason about software changes at a high-level, this dissertation proposes a novel approach that extracts program differences in the form of concise, rule-oriented descriptions. The novelty of the approach is best seen in the context of four existing approaches that can be used to reason about software changes.

The first approach records program edit operations in an editor or an integrated development environment (Section 2.3.1). A key shortcoming of this approach is that it locks programmers into a specific editor or an environment, which is rarely an acceptable trade-off.

The second approach is a programming language-based approach, exemplified by source code transformation languages and tools (Section 2.3.2). These tools let programmers spec-

ify a high-level change using the syntax of transformation languages and automatically update a program using the specified change script. Though software change can be described explicitly at a high-level, this language-based approach has a *high adoption cost* and *is not compatible with exploratory programming practices*. Programmers need to plan software changes in advance and write a change script accordingly using the syntax of a source transformation language.

The third approach is to use a check-in comment or a change log that programmers manually write in a natural language. Though it is easy to understand the high-level change intent associated with a program change, these natural language descriptions tend to be *incomplete* and may not reflect actual code changes faithfully.

The fourth approach is an automatic program differencing approach that takes two program versions as input and computes differences between them (Section 2.2). This approach is *practical* as it can be applied to any software projects with a version control system or a release archive. Unfortunately, most existing differencing approaches produce low-level differences individually without any structure even when this collection of low-level differences has a latent structure because the programmer applied a high-level operation such as a refactoring or a crosscutting modification; existing approaches do not identify nor leverage systematic relationships created by the programmer's implementation of the high-level change.

These limitations of existing program differencing approaches make it difficult for programmers to reason about software changes at a high-level. For example, the ubiquitous program differencing tool *diff* computes differences per file, obliging the programmer to read changed-lines file by file, even when those cross-file changes were done systematically. Similarly other differencing tools that work at different levels of abstraction (e.g., abstract syntax trees [304] and control flow graphs [7]) report individual differences without structure. Some approaches attempt to mitigate this problem by grouping the differences by physical locations (directories and files) [138], by logical locations (packages, classes, and methods) [302], by structural dependencies (define-use and overriding) [50], or by similarity of names. However, they generally do not capture systematic changes along other dimensions. For example, Eclipse *diff* and UMLDiff [302] organize differences by logical locations

but do not group changes that are orthogonal to a program's containment hierarchy. In contrast, crosscutting change identification techniques (Section 2.1.2) do not find regularities within a program's containment hierarchy such as adding similar fields to the same class. As existing approaches do not recognize regularities in code changes, subsequently they are unable to detect inconsistency in code changes, leaving it to a programmer to discover potential bugs.

This dissertation presents an *automatic program differencing* approach that infers *concise high-level change descriptions*. In contrast to existing program differencing techniques, this approach focuses on helping programmers reason about software changes as opposed to reconstructing a new program version given the old version and a delta. Our approach discovers and represents high-level change operations explicitly using rule-based representations (first order logic rules). This rule inference approach is based on the observation that high-level changes are often *systematic*. For example, moving a set of related classes from one package to another package consists of a set of similar *move class* refactorings. Updating an application programming interface (API) often requires updating all calls to the API. Adding secondary design decisions such as logging or synchronization consists of making similar updates throughout a program. Our approach concisely describes such systematic changes by using universally quantified logic variables in first order logic rules.

There are two similar but separate change-rule inference techniques, each of which captures a different kind of change. The first kind of change-rules capture changes to API names and signatures (Chapter 5). The second kind of change-rules capture changes to code elements and structural dependencies (Chapter 6). Both kinds of rules *concisely represent systematic changes* and *explicitly note exceptions* to systematic change patterns. Thus, software engineers can quickly get an overview of program differences and use the noted exceptions to avoid inconsistent code changes.

We demonstrate the viability of this approach by applying our techniques to several open source projects and by conducting a focus group study with professional software engineers at a large e-commerce company.

This rule-based program differencing approach is partially motivated by studies of code clones—syntactically similar code fragments. By studying the *evolutionary* aspects of clones,

we discovered the needs for a program differencing tool that extracts high-level change descriptions. We also developed the insight that identification of systematicness in code changes can help programmers better reason about software changes.

This chapter describes empirical studies of code clones (Section 1.1); describes the insights into systematic program changes (Section 1.2); introduces a rule-based change inference approach (Section 1.3); lists some uses of inferred change-rules (Section 1.4); lists the contributions of the dissertation (Section 1.5); and gives a road map to the remainder of the document (Section 1.6).

1.1 Empirical Analyses of Code Clone Evolution

Code clones are syntactically identical or similar code snippets that are often created by copying and pasting code. There is no consistent or precise definition of code clones, and they are often operationally defined by individual clone detectors.

It has been long believed that code clones are inherently bad and they indicate poor software quality. The rationale behind this conventional wisdom is that programmers often need to update code clones similarly. If programmers neglect to update related code clones consistently, this missed update could lead to a potential bug during software evolution. In addition, a latent bug can be propagated to multiple places in a code base through unknowingly copying and pasting the buggy code.

This view has directed previous research efforts about code clones. Many efforts have focused on automatically identifying code clones and using clone detection results for refactoring. Software engineering researchers and practitioners have advised programmers not to create code clones and to remove existing clones in software by factoring out the commonality as a separate procedure and invoking the procedure instead. For example, Fowler [92] argues that code duplicates are bad smells of poor design and programmers should aggressively use refactoring techniques. The Extreme Programming (XP) [27] community has integrated frequent refactoring as a part of the development process.

We investigated how code clones are actually created and maintained using two analysis approaches. Our studies suggest that programmers often create and maintain code clones with clear intent and that refactoring may not always improve software with respect to

clones. The following two subsections briefly summarize the studies.

1.1.1 An Ethnographic Study of Copy and Paste Programming Practices

Programmers often copy and paste (C&P) code from various locations: documentation, someone else's code, or their own code. However, the use of copy and paste as a programming practice has bad connotations, because this practice has the potential to create unnecessary duplicates in a code base. Earlier studies have formed a few informal hypotheses about how programmers reuse code using C&P [181, 261]; however, these studies did not focus on identifying and solving potential problems caused by C&P during software evolution.

To understand common C&P usage patterns and associated implications, we conducted an ethnographic study at IBM T.J. Watson Research Center [161]. We developed a logger that records key strokes and editing operations such as copy, cut, paste, redo, undo, and delete. We built a replayer that plays back the edit logs. In addition to replaying the edit logs, we carried out manual analysis and semi-structured interviews to discover high-level change patterns and associated intentions of a programmer. Using an affinity process [34], we created a taxonomy of common C&P patterns. This study found that skilled programmers often create and manage code clones with clear intent:

- Limitations of programming languages designs may result in unavoidable duplicates in a code base. Though we observed C&P practices in only object-oriented program languages, we suspect that the cloning issue is independent of a choice of programming language as no language can support all kinds of abstraction.

- Programmers often discover a shared abstraction of similar code through the process of copying, pasting, and modifying code. They keep and maintain clones for some period of time before they decide how to abstract the common part of the clones.

- Copied text is often customized in the pasted context and reused as a structural template. Current software engineering tools have poor support for identifying structural templates or maintaining them during software evolution.

- Programmers often apply a similar change to clones from the same origin. In other words, after they create clones, they tend to modify the structural template embedded in the clones consistently.

Based on these insights into C&P patterns, we proposed tools that could reduce C&P related problems.

1.1.2 An Empirical Study of Code Clone Genealogies

In the copy and paste study, we explored an approach of capturing and replaying edits in an IDE. This approach is limited in a number of ways. First, most projects do not retain archives of editing logs, but rather have version control systems or software release archives. Second, while most projects are developed by more than one developer in a collaborative work environment, capturing edits in an IDE is limited to a single programmer workspace. Third, a longitudinal analysis is not feasible due to the high cost of analyzing edits.

To extend this type of change-centric analysis to software projects without edit logs, we developed a tool that automatically reconstructs the history of similar code fragments from a source code repository [165]. The core of this tool is a representation that captures clone change patterns over a sequence of program versions. We named this representation a clone genealogy because, like a human genealogy, it shows when new clones were introduced, how long the clones stayed in the system, and whether the clones were updated consistently or not. In a clone genealogy, a group to which the clone belongs is traced to its origin clone group in previous versions. In addition, clone groups that have originated from the same ancestor clone group are connected by a clone evolution pattern.

Using this tool, we studied how long clones stay in a system and how often and in which way clones change in two Java open source projects, *carol* and *dnsjava*. Our study indicates that refactoring (merging code clones) is not always beneficial or even applicable to many clones for two reasons:

- Many clones are short-lived, diverging clones: 48% and 72% of clones disappear in a very short amount of a time (within an average of eight check-ins out of over at least 160 check-ins). 26% and 34% of these clones are no longer considered as clones because

they changed differently from other clones in the same group. Refactoring such short-lived clones may not be necessary and can be counterproductive if a programmer has to undo the refactoring.

- Refactoring cannot remove many long-lived clones: 49% and 64% of clones cannot be easily removed using standard refactoring techniques [92]. The longer clones survive in the system, the higher percentage of them consist of unfactorable, consistently changing clones.

Instead of measuring the extent of clones over time quantitatively, we focused on how individual clones change over the life time of software. By focusing on the *evolutionary* aspects, we advanced the understanding of code clones and contributed to shifting research focus from automatic clone detection to clone management support.

1.2 Systematicness of Code-Level Change

Based on the two empirical studies of code clones, we developed the insight that high-level changes are often systematic—consisting of similar transformations at a code level. The same insight arises from numerous other research efforts, primarily within the domain of refactorings and crosscutting concerns.

Refactoring is the process of changing a software system that does not alter the external behavior of the code, yet improves the internal structure and the quality of software (e.g., extensibility, modularity, reusability, complexity, maintainability, efficiency) [92, 112, 211, 234]. Meaningful refactorings often consist of one or more elementary behavior preserving transformations, which comprise a group of similar transformations such as "moving the print method in ***each*** Document subclass" or "introduce ***three*** abstract `visit*` methods."

Crosscutting concerns represent secondary design decisions—e.g., performance, error handling, and synchronization—that are generally scattered throughout a program [158, 279]. Modifications to these design decisions involve similar changes to every occurrence of the design decision. To cope with evolution of crosscutting concerns, aspect-oriented programming languages provide language constructs that allow these concerns to be updated in a modular fashion [157]. Information transparency based techniques [111] use naming

conventions, formatting styles, and ordering of code in a file to locate and document cross-cutting concerns.

Consistent maintenance of *code clones* is another kind of systematic change. Our C&P study explains why programmers create and maintain clones. (1) The limitation of programming languages do not allow all levels of design decisions to be isolated in a single module. (2) As programmers deal with volatile design decisions, they often create code clones temporarily until they discover a right level of abstraction for similar programming logic. To support consistent update of code clones, simultaneous editing [215] and linked editing [281] allow programmers to edit related clones with a single stream of editing commands.

1.3 Automatic Inference of High-Level Change Descriptions

To discover and represent systematic changes, we developed rule-based change representations and corresponding algorithms. There are two kinds of change-rules that we developed. The first kind of change-rules capture changes to API names and signatures to match method-headers between two program versions. The second kind of change-rules capture changes to code elements and structural dependencies to discover a logical structure in program differences.

1.3.1 Changes to API Name and Signature

Code matching is the underlying basis for various software engineering tools. Version merging tools identify possible conflicts among parallel updates by analyzing matched code elements [208]. Regression testing tools prioritize or select test cases that need to be re-run by analyzing matched code elements [117, 262, 275]. Profile propagation tools use matching to transfer execution information between versions [294]. Emerging interest in mining software repositories—studying program evolution by analyzing existing software project artifacts—demands more effective matching techniques.

Existing matching techniques individually compare code elements at particular granularities such as functions and files using certain similarity measures (Section 2.2). Though this approach is intuitive, it has three limitations: First, these techniques produce a long

list of matches or refactorings without any structure and do not recognize a general pattern of changes witnessed by the low-level code matches. Second, as these techniques do not recognize systematic change patterns, it is generally difficult for programmers to spot inconsistent or incomplete changes to a general pattern. For example, only after looking at three matches [`(Foo.mA()` $\mapsto$ `Foo.mA(float))`, `(Foo.mB()` $\mapsto$ `Foo.mB(float))`, and `(Foo.mC()` $\mapsto$ `Foo.mC())`], one can understand that `mC()` method did not change its input signature unlike other methods in the `Foo` class. Third, because existing techniques do not consider the global context of changes, in part due to the lack of explicit high-level change representations, these techniques cannot easily disambiguate which match is more accurate than other potential matches; this limitation results in either low precision or low recall of matches.

As a first step, we addressed the problem of matching Java method-headers. Our change-rule representation concisely describes systematic changes to API names and signatures and explicitly notes anomalies to a systematic change pattern. Our rule inference algorithm takes two program versions as input; extracts a set of method-headers from each version; and finds seed-matches using token-level name similarity between each possible pair of method-headers. By generalizing the scope and transformation in the seed matches, our algorithm systematically enumerates candidate change-rules. It then evaluates and selects these rules using a greedy algorithm. In each iteration, each rule's accuracy is computed by counting its matches and its exceptions. When the rule's accuracy is above the chosen threshold, the rule is selected and its matches are removed from a set of remaining method-headers to be matched. This algorithm can be seen as a practical, domain-specific rule inference algorithm in the sense that it finds a restricted subset of first order logic rules—a horn clause with a single literal in the antecedent.

Consider an example where a programmer reorganizes a chart drawing program by moving axis-drawing classes from the package `chart` to the package `chart.axis`. To allow toggling of tool tips by the user, the programmer appends a `boolean` parameter to a set of chart-creation interfaces. Even though the goals of these transformations can be stated concisely in natural language, existing code matching techniques (Section 2.2) would enumerate each moved method and each modified interface. The programmer may have to examine hundreds or thousands of matches or refactorings before discovering that a few simple high-

level changes took place. Moreover, if the programmer neglected to move one axis drawing class, this missed update would be hard to detect. The following two change-rules concisely describe these changes. (Chapter 5 discusses the syntax and semantics of API change-rules in detail.)

• for all x:method-header in chart.*Axis*.*(*)

packageReplace(x, chart, chart.axis)

[Interpretation: All methods with a name "chart.*Axis*.*(*)" moved from the chart package to the chart.axis package.]

• for all x:method-header in chart.Factory.create*Chart(*Data)

except createGanttChart, createXYChart

argAppend(x, [boolean])

[Interpretation: All methods with a name "chart.Factory.create*Chart(*Data)" changed their input signature by appending an argument with boolean type.]

We applied our API change-rule inference tool to the change history of five open source projects (*JFreeChart*, *JHotDraw*, *JEdit*, *Tomcat*, and *ArgoUML*) and compared it with three competing tools (S. Kim et al.'s method-header matching tool [167], Weißgerber and Diehl's refactoring reconstruction tool [295], and Xing and Stroulia's UMLDiff [302]). Our comparative evaluation shows that our technique makes matching results smaller and more readable, and finds more matches without loss of precision.

1.3.2 Changes to Program Structure

The success of API change-rule inference led to extending this approach to complement some existing uses of *diff*. Programmers often use *diff* to inspect detailed differences between program versions. When its output involves hundreds of lines across multiple files, programmers find it difficult to understand code changes because there is no structure that groups related line-level differences. We hypothesize that, by identifying shared structural characteristics of changed code, we can discover a logical structure that groups related line-level differences.

Logical Structural Diff (LSDiff) abstracts a program as code elements (packages, types,

methods, and fields) and their structural dependencies (method-calls, field-accesses, subtyping, overriding, and containment). For example, LSDiff extracts facts such as "class `C` has a field `f` of type `t`" and "class `C`'s method `m` overrides class `D`'s method `m`."Once LSDiff represents each version as a set of logic facts, it uses a set-differencing operator to compute fact-level differences between two versions. To condense the delta, LSDiff infers Datalog rules [284] that imply fact-level differences. The rule inference algorithm is a top-down inductive logic programming algorithm that enumerates domain specific Datalog rules up to a certain length specified by a user. Remaining non-systematic differences are presented as logic facts.

This rule-based change inference approach is similar to the previous section's API change-rule inference. However, LSDiff differs from it in several ways: First, while API change-rules focus on changes at the level of method-headers, LSDiff accounts for changes within method-bodies as well as at a field level. Second, from a change representation perspective, while API change-rules rely on a regular expression to group related code elements, LSDiff uses conjunctive logic literals to allow programmers to understand shared structural characteristics of systematically changed code, not only a shared naming convention, e.g. "all `setHost` methods in `Service`'s subclasses" instead of "all methods with name `*Service.setHost(*).`" Finally, from a rule-inference technique perspective, while our API change-rule inference algorithm finds rules in an open system, LSDiff's algorithm learns rules in a closed system by first computing structural differences and then enumerating all rules within the rule search space set by the input parameters.[1]

Suppose that a programmer deleted method-calls to the `SQL.exec` method from all `setHost` methods in `Service`'s subclasses. The following rule concisely captures regularities among five method-call deletions and also reports an exception, indicating a potential missing change. The detailed syntax and semantics of LSDiff rules is described in Chapter 6.

• past_subtype("Service", t) ∧ past_method(m, "setHost", t) ⇒ deleted_calls(m, "SQL.exec") except t="NameSvc"

[Interpretation: All setHost methods in Service's subclasses deleted calls to the SQL.exec method except the NameSvc class.]

[1]Learning rules in a closed system means that the facts in the fact-bases cannot be altered during the rule-inference process.

In a study of three software projects' histories, LSDiff outputs are on average 9.3 times more concise than structural differences without rules. What this means in practice is that, when *diff* output consists of 997 lines of change scattered across 16 files on average, LSDiff summarizes structural differences using only 7 rules and 27 facts.

To gain insights into when and how LSDiff can complement existing program differencing tools, we conducted a focus group study with professional developers from a large e-commerce company. Overall, our focus group participants were very positive about LSDiff and asked us when they can use it for their work. They believed that LSDiff can help programmers reason about related changes effectively by allowing top-down reasoning of code changes, as opposed to reading *diff* outputs without having a high-level context. Some quotations from the focus-group study were:

"This is cool. I'd use it if we had one."

"This is a definitely winner tool."

"You can't infer the intent of a programmer, but this is pretty close."

1.4 Uses of Inferred Change-Rules

This section lists several scenarios in which change-rules can help programmers in reasoning about software changes. These scenarios are based on real examples found in the *carol* open source project as well as the observation carried out by Ko et al. [170].

Understanding the rationale of others' change. Alice and Bill work in the same team. When Alice tried to commit her bug fix, she got an error message that her change conflicted with Bill's last change. To understand what he changed and why, she started reading Bill's last check-in comment, *"Common methods go in an abstract class. Easier to extend/maintain/fix,"* and the associated *diff* output. However, she could not easily understand whether his change was indeed an *extract superclass* refactoring, which classes were involved, and whether the refactoring was completed. Browsing the *diff* output, she was overwhelmed by the many files to examine.

The following LSDiff output helps Alice understand the rationale of Bill's change: Bill created `AbsRegistry` by pulling up `host` fields and `setHost` methods from the classes implementing the `NameSvc` interface; however, he did not complete the refactoring on `LmiRegistry`,

that also implements the `NameSvc` interface. After reading them, Alice decided to double check with Bill why `LmiRegistry` was left out.[2]

• Fact 1. added_type("AbsRegistry")

[Interpretation: AbsRegistry is a new class.]

• Rule 1. current_inheritedmethod(m, "AbsRegistry", t)⇒added_inheritedmethod(m, "AbsRegistry", t)

[Interpretation: Many classes inherit method implementations from the AbsRegistry class.]

• Rule 2. past_subtype("NameSvc", t) ∧ past_field(f, "host", t) ⇒ deleted_field(f, "host", t) (except t = "LmiRegistry")

[Interpretation: All NameSvc's subtypes' host fields were removed except LmiRegistry's host field.]

• Rule 3. past_subtype("NameSvc", t) ∧ past_method(m, "setHost", t) ⇒ deleted_method(m, "setHost", t) (except t = "LmiRegistry")...

[Interpretation: All NameSvc's subtypes' setHost methods were removed except the LmiRegistry's setHost method.]

Reviewing a patch before its submission. To simplify the usage of constants in her program, Alice decided to put all constants in the `Context` class. While implementing this change, she ported the constant accesses to use `Context`'s constants instead. After finishing edits, she reviewed the *diff* output but could not easily verify the correctness of constant porting because some constants were accessed from many methods.

The following LSDiff output helps her confirm that `DefaultValues`' constants were deleted and that all methods that once used `DefaultValues`' constants use the `Context` class instead.[3]

• Fact 1. deleted_field("DefaultValues.FACTORY")

[Interpretation: The DefaultValues.FACTORY field was deleted.]

• Fact 2. deleted_field("DefaultValues.URL")

[Interpretation: The DefaultValues.URL field was deleted.]

• Rule 1. past_accesses("DefaultValues.URL", m)⇒ added_accesses("Context.URL", m)

[Interpretation: All methods that accessed the DefaultValues.URL field in the old version added accesses to the Context.URL field.]

---

[2]Source: carol revision 430

[3]Source: carol revision 389

• Rule 2. past_accesses("DefaultValues.FACTORY", m) ⇒ added_accesses("Context.FACT", t)...

[Interpretation: All methods that accessed the DefaultValues.FACTORY field in the old version added accesses to the Context.FACT field.]

Writing change documentation. To write a check-in comment, Alice ran the *diff* tool to examine her modification. By looking at the list of changed files, she suspected that two different logical changes got mixed up: a design change request and a configuration bug fix. However, she could not remember which changed code fragments correspond to which logical change.

By examining the following LSDiff output, she recalled that she added the `lmi` package and the `LmiDelegate` class for the design change request and added the `loadCfg` method in several classes to fix the bug.[4]

• Fact 1. added_package("jndi.lmi")

[Interpretation: The jndi.lmi package is a new package.]

• Fact 2. added_type("LmiDelegate", "rmi.multi")

[Interpretation: The LmiDelegate class was added to the rmi.multi package.]

• Rule 1. current_method("loadCfg", t) ⇒ added_method("loadCfg", t)

[Interpretation: All loadCfg methods are newly added methods.]

Mining software repositories research. In the last several years, researchers in software engineering have begun to analyze programs together with their change history. For example, Nagappan and Ball's algorithm [229] finds line-level changes between two consecutive versions, counts the total number of changes per binary module, and infers the characteristics of frequently changed modules. As another example, a signature change pattern analysis [167, 168] traces how the name and the signature of functions change. Finally, visualization techniques were applied to change history data to identify evolution trends, unstable components, coherent entities, design and architectural evolution, and fluctuations in team productivity [19, 20, 53, 78, 100, 183, 245, 265]. As these mining software repositories research techniques require matching code elements across versions, we conjecture that our change-rules can benefit these research efforts in two ways: (1) By automatically

[4]Source: carol revision 63

identifying code renaming and moving, software evolution history can be modeled without discontinuation. (2) By grouping related atomic transformations, researchers can reason about software changes at a high-level.

1.5 Thesis and Contributions

- We demonstrated the benefits of change-centric analysis approaches in the context of studying clone evolution. By focusing on the *evolutionary aspects* of clones, our studies found that immediate and aggressive refactoring of code clones are often not necessary nor beneficial.

- Based on the studies of code clones, we developed an insight that high-level software changes are often systematic at a code level, consisting of similar transformations.

- Based on this insight, we invented an approach that automatically infers high-level change descriptions as logic rules by discovering and representing systematicness in code changes.

- Our assessments and focus-group study show that our approach finds concise change descriptions, identifies inconsistent changes, and complements existing program differencing tools.

1.6 Outline

Chapter 2 surveys related work. This can be divided into three main categories: (1) a survey of systematic software changes, (2) techniques that take two program versions as input and infer changes between them automatically, (3) techniques that can record program changes, and (4) background on code clones and clone evolution studies. Chapter 3 and Chapter 4 describe empirical analyses of clone evolution: an ethnographic study of copy and paste programming practices and an empirical study of code clone evolution in open source projects. Chapter 5 and Chapter 6 describe our rule-based change inference approach at the level of method-headers and at the level of code elements and structural dependencies respectively.

The assessment of inferred change-rules is discussed respectively in each chapter. Chapter 7 presents future work and conclusions.

Chapter 2

RELATED WORK

Section 2.1 reviews several kinds of systematic program changes because systematicness of high-level changes is a basis for our rule-based change inference approach. Section 2.2 and Section 2.3 describe approaches that can help programmers reason about software change. Out of the four approaches discussed in Chapter 1, three approaches are described in detail: A program differencing approach automatically infers change operations from two program versions (Section 2.2). An edit capture and replay approach explicitly records change operations. Change operations can be also captured in source transformation tools (Section 2.3). Using check-in comments or change logs is excluded because natural language descriptions tend to be incomplete and may not reflect code changes faithfully. Section 2.4 discusses the background related to our empirical studies of code clones. Section 2.5 summarizes related work in the area of software evolution analyses, because these analyses can benefit from explicit, semantic change descriptions provided by our change-rule inference techniques.

2.1 Systematic Code Change

This section discusses several kinds of systematic program changes, namely refactorings and crosscutting concerns. By inspecting these changes' characteristics, we show that high-level software changes are often systematic, consisting of similar transformations at a code-level. Consistent maintenance of *code clones* is another kind of systematic change and related work on code clones is described in Section 2.4.

2.1.1 Refactoring

As a software system is enhanced, modified, and adapted to new requirements, the code becomes more complex and drifts away from its original design, thereby lowering the qual-

ity of the software. *Refactoring (Restructuring)* [92, 112, 234, 211] copes with increasing software complexity by transforming a program from one representation to another while preserving the program's external behavior (functionality and semantics).

Griswold's dissertation [112] discusses one of the first refactoring tools that automate repetitive, error-prone, non-local transformations. Griswold's tool supports a number of restructuring operations: replacing an expression with a variable that has its value, swapping the formal parameters in a procedure's interface and the respective arguments in its calls, etc. It is important to note that many of these restructurings are systematic in the sense that they involve repetitive non-local transformations.

Opdyke's dissertation [234] distinguishes the notion of low-level refactorings from high-level refactorings. High-level refactorings (i.e., composite refactorings) reflect more complex behavior-preserving transformations while low-level refactorings are primitive operations such as creating, deleting, or changing a program entity or moving a member variable. Opdyke describes three kinds of complex refactorings in detail: (1) creating an abstract superclass, (2) subclassing and simplifying conditionals, and (3) capturing aggregations and components. All three refactorings are systematic in the sense that they contain ***multiple*** similar transformations at a code level. For example, creating an abstract superclass involves moving ***multiple*** variables and functions common to ***more than one*** sibling classes to their common superclass. Subclassing and simplifying conditionals consists of creating ***several*** classes, each of which is in charge of evaluating a different conditional. Capturing aggregations and components usually involves moving ***multiple*** members from a component to an aggregate object.

Fowler's refactoring book [92] describes 72 types of common refactorings. Most refactorings are systematic in the sense that (1) ***multiple*** objects participate in the refactoring; (2) these objects often share ***similar*** structural characteristics such as inheriting the same superclass or using the same object; and (3) they often undergo ***similar*** primitive refactorings.

2.1.2 Crosscutting Concerns

As programs evolve over time, they may suffer from the *the tyranny of dominant decomposition* [279]. They can be modularized in only one way at a time. Concerns that are added later may end up being scattered across many modules and tangled with one another. Logging, performance, error handling, and synchronization are canonical examples of such secondary design decisions. These can be seen as another kind of systematic change that involves inserting similar code throughout a program. Aspect-oriented programming languages provide language constructs to allow concerns to be updated in a modular fashion [157]. A number of other approaches instead leave the crosscutting concerns in a program while providing mechanisms to manage related but dispersed code fragments. Griswold's information transparency technique uses naming conventions, formatting styles, and ordering of code in a file to provide indications about code that should change together [111].

There are several tools that allow programmers to automatically or semi-automatically locate crosscutting concerns. Robillard et al. [260] allow programmers to manually document crosscutting concerns using structural dependencies in code. Similarly, the Concern Manipulation Environment [116] allows programmers to locate and document different types of concerns. Van Engelen et al. [285] use clone detectors to locate crosscutting concerns. Shepherd et al. [271] locate concerns using natural language program analysis. Breu et al. [41] mine aspects from version history by grouping method-calls that are added together. Dagenais et al. [61] automatically infer and represent structural patterns among the participants of the same concern as rules in order to trace the concerns over program versions. In general, our change-rule inference techniques differ from these tools by inferring general kinds of systematic changes, which may or may not be crosscutting concerns, and by detecting anomalies from systematic changes.

2.2 Inferring Change

Inferring changes between two program versions is the same problem as matching corresponding unchanged code elements between two versions. Section 2.2.1 describes existing code matching (program differencing) techniques that are built for software version merging,

program differencing, profile propagation, and regression testing. Most matching techniques are different from our change-rule inference techniques (Chapter 5 and Chapter 6) in that they compute low-level program differences without structure and cannot help programmers reason about software changes at a high-level. Section 2.2.2 describes techniques that infer refactorings from two program versions. These techniques are closely related to our change-rule inference techniques in that both use code matching information to infer refactorings. Section 2.2.3 describes techniques that group related code changes. These techniques are similar to our change-rule inference techniques in that they cluster related changes and can identify potential missed updates.

2.2.1 Code Matching

Suppose that a program P' is created by modifying P. Determine the difference Δ between P and P'. For a code fragment $c' \in P'$, determine whether $c' \in \Delta$. If not, find c''s corresponding origin c in P.

A code fragment in the new version either contributes to the difference or comes from the old version. If the code fragment has a corresponding origin in the old version, it means that it does not contribute to the difference. Thus, finding the delta between two versions is the same problem as finding corresponding code fragments between two versions. Suppose that a programmer inserts if-else statements in the beginning of the method `m_A` and reorders several statements in the method `m_B` without changing semantics (see Table 2.1). An intuitively correct matching technique should produce [(s1-s1'), (s2-s2'), (s3-s4'), (s4-s3'), and (s5-s5')] and identify that s0' is added.

Matching code across program versions poses several challenges. First, previous studies [167] indicate that programmers often disagree about the origin of code elements; low inter-rater agreement suggests that there may be no ground truth in code matching. Second, renaming, merging, and splitting of code elements make the matching problem non-trivial. Suppose that a file `PElmtMatch` changed its name to `PMatching`; a procedure `matchBlck` is split into two procedures `matchDBlck` and `matchCBlck`; and a procedure `matchAST` changed its name to `matchAbstractSyntaxTree`. The intuitively correct matching technique should produce

Table 2.1: Example code change

before	after

mA (){
  if (pred_a) { //s1
  foo(); //s2
  }
}
mB (b){
  a= 1; //s3
  b= b+1; //s4
  fun(a,b); //s5
}
	mA (){
	  if (pred_a0) { //s0'
	  if (pred_a) { //s1'
	  foo(); //s2'
	  }
	  }
	}
	mB (b){
	  b= b+1; \\s3'
	  a= 1; \\s4'
	  fun(a,b); \\s5'
	}

[(`PElmtMatch, PMatching`), (`matchBlck, matchDBlck`), (`matchBlck, matchCBlck`), and (`matchAST, matchAbstractSyntaxTree`)], while simple name-based matching will consider `PMatching`, `matchDBlck`, `matchCBlck`, and `matchAbstractSyntaxTree` added and consider `PElmtMatch`, `matchBlck`, and `matchAST` deleted.

Existing code matching techniques usually employ syntactic and textual similarity measures to match code. They can be characterized by the choices of (1) an underlying program representation, (2) matching granularity, (3) matching multiplicity, and (4) matching heuristics. This section explains how the choices impact applicability, effectiveness, and accuracy of each matching method by creating an evaluation framework.

Entity Name Matching The simplest matching method treats code elements as immutable entities with a fixed name and matches the elements by name. For example, Zimmermann et al. model a function as a tuple, *(file name, FUNCTION, function name)*, and a field as a tuple, *(function name, FIELD, field name)* [312]. Similarly, Ying et al. [308] model a file with its full path name.

String Matching When a program is represented as a string, the best match between two strings is computed by finding the longest common subsequence (LCS) [9]. The LCS problem is built on the assumption that (1) available operations are addition and deletion, and (2) matched pairs cannot cross one another. Thus, the longest common subsequence does not necessarily include all possible matches when available edit operations include copy, paste, and move. Tichy's *bdiff* [280] extended the LCS problem by relaxing the two assumptions above: permitting crossing block moves and not requiring one-to-one correspondence.

The line-level LCS implementation, *diff* [139] is fast, reliable, and readily available. Thus, it has served as a basis for popular version control systems such as CVS.[1] or Subversion[2] Many evolution analyses are based on *diff* because they use version control system data as input. Our clone genealogy extractor tracks code snippets by their file name and line number (Chapter 4). Identification of fix-inducing code snippets [273] is also based on tracking *(file name:: function name:: line number)* backward from the moment that a bug is fixed.

Reiss [253] evaluated practical LCS-based source line tracking techniques. His investigation shows that the *W_BEST_LINE* method—a variation of LCS algorithm that considers k number of contextual lines—is about as effective as any other method but is faster and requires only a small amount of storage. This method compares each line to derive a normalized match value between zero (no match) and one (exact match); looks at a context consisting of $k/2$ lines before and after the line; and counts the number of these lines that match the corresponding line in the new version.

Recently, Canfora et al. [46] developed a source line technique that takes differencing results from *diff*-based version control systems as input and identifies changed-lines in addition to added- and deleted-lines. This technique first computes hunk similarity between every possible hunk pair using a vector space model and then computes the Levenstein distance [186] to map source lines within the mapped hunk pairs. In contrast to *diff*, this approach detects changed-lines in addition to deleted- and added-lines.

---

[1] http://www.cvshome.org

[2] http://subversion.tigris.org

Syntax Tree Matching For software version merging, Yang [304] developed an AST differencing algorithm. Given a pair of functions (f_T, f_R), the algorithm creates two abstract syntax trees T and R and attempts to match the two tree roots. Once the two roots match, the algorithm aligns T's subtrees $t_1, t_2, ..., t_i$ and R's subtrees $r_1, r_2, ...r_j$ using the LCS algorithm and maps subtrees recursively. This type of tree matching respects the parent-child relationship as well as the order between sibling nodes, but is very sensitive to changes in nested blocks and control structures because tree roots must be matched for every level.

For dynamic software updating, Neamtiu et al. [230] built an AST-based algorithm that tracks simple changes to variables, types, and functions. Neamtiu's algorithm assumes that function names are relatively stable over time. It traverses two ASTs in parallel; matches the ASTs of functions with the same name; and incrementally adds one-to-one mappings as long as the ASTs have the same shape. In contrast to Yang's algorithm, it cannot compare structurally different ASTs.

Fluri et al.'s Change Distiller [90] uses an improved version of Chawathe et al.'s hierarchically structured data comparison algorithm [48]. Change Distiller takes two abstract syntax trees as input and computes basic tree edit operations such as *insert, delete, move* or *update* of tree nodes. It uses *bi-gram string similarity* to match source code statements such as method invocations and uses *subtree similarity* to match source code structures such as if-statements. After identifying tree edit operations, Change Distiller maps each tree-edit to an atomic AST-level change type.

Cottrell et al.'s Breakaway [58] automatically identifies detailed structural correspondences between two abstract syntax trees to help programmers generalize two pieces of similar code. Its two-pass greedy algorithm is applied to ordered child list properties (statements in a block) then to unordered nodes (method declarations).

Finally, the following two techniques do not directly compare ASTs but use syntactic information to guide string level differencing. Hunt and Tichy's 3-way merging tool [136] parses a program into a language neutral form; compares token strings using the LCS algorithm; and finds syntactic changes using structural information from the parse. Raghavan et al.'s Dex [249] locates the changed parts in C source code files using *patch* file information and feeds the changed parts into a tree differencing algorithm to output the edit operations.

Control Flow Graph Matching Laski and Szermer [184] first developed an algorithm that computes one-to-one correspondences between CFG nodes in two programs. This algorithm reduces a CFG to a series of single-entry, single-exit subgraphs called hammocks and matches a sequence of hammock nodes using a depth first search (DFS). Once a pair of corresponding hammock nodes is found, the hammock nodes are recursively expanded in order to find correspondences within the matched hammocks.

Jdiff [7] extends Laski and Szermer's (LS) algorithm to compare Java programs based on an enhanced control flow graph (ECFG). *Jdiff* is similar to the LS algorithm in the sense that hammocks are recursively expanded and compared, but is different in three ways: First, while the LS algorithm compares hammock nodes by the name of a start node in the hammock, *Jdiff* checks whether the ratio of unchanged-matched pairs in the hammock is greater than a chosen threshold in order to allow for flexible matches. Second, while the LS algorithm uses DFS to match hammock nodes, *Jdiff* only uses DFS up to a certain look-ahead depth to improve its performance. Third, while the LS algorithm requires hammock node matches at the same nested level, *Jdiff* can match hammock nodes at a different nested level; thus, *Jdiff* is more robust to addition of while loops or if-statements at the beginning of a code segment. *Jdiff* has been used for regression test selection [236] and dynamic change impact analysis [8].

CFG-like representations are commonly used in regression test selection research. Rothermel and Harrold's algorithm [262] traverses two CFGs in parallel and identifies a node with unmatched edges, which indicates changes in code. In other words, the algorithm stops parallel traversal as soon as it detects changes in a graph structure; thus, this algorithm does not produce deep structural matches between CFGs. However, traversing graphs in parallel is still sufficient for the regression testing problem because it conservatively identifies affected test cases. In practice, regression test selection algorithms [117, 236] require that syntactically changed classes and interfaces are given as input to the CFG matching algorithm.

Program Dependence Graph Matching There are several program differencing algorithms based on a program dependence graph [133, 37, 143].

Horwitz [133] presents a semantic differencing algorithm that operates on a program representation graph (PRG) which combines features of program dependence graphs and static single assignment forms. In her definition, semantic equivalence between two programs $P1$ and $P2$ means that, for all states σ such that $P1$ and $P2$ halt, the sequence of values produced at $c1$ is identical to the sequence of values produced at $c2$ where $c1$ and $c2$ are corresponding locations. Horwitz uses Yang's algorithm [305] to partition the vertices into a group of semantically equivalent vertices based on three properties, (1) the equivalence of their operators, (2) the equivalence of their inputs, (3) the equivalence of the predicates controlling their evaluation. The partitioning algorithm starts with an initial partition based on the operators used in the vertices. Then by following flow dependence edges, it refines the initial partition if the successors of the same group are not in the same group. Similarly, it further refines the partition by following control dependence edges. If two vertices in the same partition are textually different, they are considered to have only a *textual change*. If two vertices are in different partitions, they have a *semantic change*. After the partitioning phase, the algorithm finds correspondences between $P1$'s vertices and $P2$'s vertices that minimize the number of semantically or textually changed components of $P2$.

Binkley et al. [37] presents a 3-way merging algorithm that is based on semantic differences. This algorithm does not find corresponding elements between two versions of a program, but rather makes an assumption that a special editor is used to tag each PDG node to identify added nodes, deleted nodes and changed nodes. Given PDG node level correspondence among three input programs A, B, and Base, the integration algorithm produces a program M that integrates the difference A from Base, the difference B from Base, and the preserved behavior among A, B, and Base. The behavior differences between A and B are approximated by the slice of $AP_{A,Base}$ in G_A where $AP_{A,Base}$ is a set of vertices of G_A whose program slice is different from G_{Base}'s slice. Although the problem of determining whether G_M corresponds to some program is NP-complete, Binkley et al. presented a backtracking algorithm that behaves satisfactorily on actual programs.

In general, PDG-based algorithms are not applicable to popular modern program languages because they can run only on a limited subset of C-like languages without global variables, pointers, arrays, or procedures.

Binary Code Matching *BMAT* [294] matches two versions of a binary program without knowledge of source code changes. *BMAT* was used for profile propagation and regression test prioritization [275]. BMAT's algorithm matches blocks in three steps: (1) matching procedures by their names and code contents, (2) matching data blocks using hash functions, and (3) matching code blocks using hash functions and control flow equivalence.

BMAT has been improved [293] because its hashing based algorithm was too sensitive to instruction reordering due to compiler optimization. Its new algorithm uses control tree representations and matches code blocks (leaf-nodes) in two phases: In the bottom-up phase, code blocks are matched first and matching criteria is relaxed to accommodate minor changes in the tree structure or node contents while still using the hierarchical tree structure to identify correct matches. In the top-down phase, the tree is traversed in depth-first order and block matches are finalized recursively, using higher level matches to guide lower-level matches.

Text Document Several tools find a similar document (or a group of similar documents) in a directory of files using chunking and approximate fingerprints [91, 148, 198]. *Siff* [198] is characteristic of these tools. First, *siff* breaks each file into a sequence of chunks using sliding windows and computes fingerprints for each chunk incrementally. After this step, each file is represented as a list of fingerprints. Second, for a given file, *siff* finds a similar document using the *agrep* algorithm. For each fingerprint, *siff* lists all files that include the fingerprint. Then it sorts the list and computes the shared number of fingerprints for each distinctive set of files. Third, it uses a threshold to discard matches with a small number of shared fingerprints.

Clone Detection A clone detector is simply an implementation of an arbitrary equivalence function. The equivalence function defined by each clone detector depends on a program representation and a comparison algorithm. Most clone detectors are heavily dependent on (1) hash functions to improve performance, (2) parametrization to allow flexible matches, and (3) thresholds to remove spurious matches. A clone detector can be considered as a many-to-many matcher based solely on content similarity heuristics. Section 2.4

discusses clone detection techniques in detail.

Others The matching problem is one of the most fundamental and general problems in computer science. For example, schema matching [250] is a very well studied area in database research. This problem is similar to code matching in that (1) the problem can be solved at a different granularity (attributes or tables); (2) typed structures are often compared to find a matching; and (3) splitting, merging, and renaming of attributes often make the problem more challenging. However, it is difficult to adapt schema matching techniques to the code matching problem, because many techniques—instance based matching, linguistics based matching, key characteristics based matching, and uses of join queries—are only pertinent to schema matching.

Selective recompilation (incremental compilation [47]) assumes an initial set of changes and focuses only on how to efficiently identify compilation units to be recompiled using structural dependencies (e.g., def-use).

Translation validation (or checking comparison) [231, 311] checks whether an optimized version of a program still preserves the same semantics of an unoptimized version of a program. Solving this problem requires matching equivalent program elements. Necula [231] uses simple heuristics to match control flow graphs and uses symbolic evaluation to check equivalence of simulation relations (predicates which are provided by the compiler for particular points of the programs). Necula's algorithm [231] compares control predicates and function call sequences to match CFG nodes. Zhang and Gupta [311] address the same problem at a binary level by matching instructions based on program execution histories given the same input. Based on WET representation [310], a static representation of the program labeled with dynamic profile information, the algorithm matches instructions that dynamically behave the same even though statically they appear to be different.

Comparison Table 2.2 compares these surveyed matching techniques. The first column shows which underlying program representation each technique is based on. The second columns includes references to each surveyed technique. The third column shows at what granularity each technique matches code. The fourth column shows at what granularity

each technique assumes correspondences. Many techniques assume correspondences at a certain granularity no matter whether this assumption is explicitly stated or not; *diff* and *bdiff* match source lines assuming that input files are already matched; *cdiff* matches AST nodes assuming that enclosing functions are matched by the same name; and Tichy et al. [197, 136] assume that corresponding files are mapped. The fifth column shows matching multiplicity. Only *bdiff* allows one to many mappings at a line level and BMAT allows many to one mappings at a block level. Other techniques all assume one to one mappings. The column six, seven, and eight describe matching heuristics employed by each technique. All matching techniques heavily rely on heuristics to reduce the scope of potential matches, and the heuristics are categorized into three categories that are not comprehensive or mutually exclusive.

1. Name-based heuristics match entities with similar names.

2. Position-based heuristics match entities with similar positions. If entities are placed in the same syntactic position or surrounded by already matched pairs, they become a matched pair.

3. Similarity-based heuristics match entities that are nearly identical; they often rely on parametrization and a hash function to find near identical entities. All clone detectors can be viewed as similarity-based matchers.

The three different heuristics complement one another. For example, when hash values collide or parametrization results in spurious matches, position-based heuristics will select a matched pair that preserves linear ordering or structural ordering by checking neighboring matches. *Diff* and *bdiff* use similarity-based heuristics to map source lines. *cdiff* uses name-based heuristics to map AST nodes by their labels. Neamtiu's algorithm uses both name-based and position-based heuristics: It traverses two ASTs in parallel, matches AST nodes by the same label, and matches variable nodes placed in the same syntactic position regardless of their labels. *Jdiff* matches CFG nodes by the same label and matches hammock nodes by the hammock's content (the ratio of unchanged-matched pairs in the hammock

nodes). BMAT uses a name-based heuristic to match procedures in multiple phases: by the same globally qualified name (e.g., `System.out.println`), by the same hierarchical name, by the same signature, and by the same procedure name. BMAT also uses a position-based heuristic to remove unmatched pairs.

Evaluation Matching techniques are often inadequately evaluated, in part due to a lack of agreed evaluation criteria or representative benchmarks. To evaluate matching techniques uniformly, we take a scenario-based evaluation approach; We describe how well matching techniques will perform for the hypothetical program change scenarios described in the beginning of this section (page 21).[3]

The third column of Table 2.3 summarizes how well each technique will work in the scenario of Table 2.1. *Diff* can match lines of `m_A` but cannot match reordered lines in `m_B` because the LCS algorithm does not allow crossing block moves. *Bdiff* can match reordered lines in `m_B` because crossing block moves are allowed. Neamtiu's algorithm will perform poorly in both `m_A` and `m_B` because it does not perform a deep structural match. *Cdiff* cannot match unchanged parts in `m_A` correctly because *cdiff* stops early if roots do not match for each level. *Jdiff* will be able to skip the changed control structure, map unchanged parts in `m_A`, and match reordered statements in `m_B` if the look-ahead threshold is greater than the depth of nested controls. *BMAT* cannot track code blocks in `m_B` because BMAT's hashing algorithms are instruction order sensitive. In conclusion, *Jdiff* will work best for changes within procedures at a statement or predicate level.

The fourth column of Table 2.3 summarizes how each technique will work in case of renaming and splitting at a file or procedure level. Most name-based matching techniques will do poorly with renaming events. *Diff* and *bdiff* will be able to track each line only if file names do not change. Both *cdiff* and Neamtiu's algorithm will perform poorly if procedure names change. *BMAT* will perform well because it relies on multiple passes of hash functions and multiple phases of name matching. The remaining columns of Table 2.3 describe how well each matching technique will work in case of restructuring tasks at a

[3]PDG-based matching techniques are excluded as these techniques only work for limited C-like languages and do not support modern programming languages.

Table 2.2: Comparison of code matching techniques

Program Representation	Citation	Granularity	Assumed Correspondence	Multiplicity	Heuristics			Application
					N	P	S	
A set of entities	[109, 77, 229]	Module		1:1	√			Fault proneness
	Bevan et al. [31]	File		1:1	√			Instability
	Ying et al. [308]	File		1:1	√			Co-change
	Zimmermann et al. [312]	File Procedure Field		1:1	√			
String	*diff* [139]	Line	File	1:1			√	Merging Clone genealogy [165] Fix inducing code [273]
	bdiff [280]	Line	File	1:n			√	Merging
AST	*cdiff* [304]	AST Node	Procedure	1:1	√			
	Neamtiu et al.[230]	Type, Var		1:1	√	√		Type change
	Hunt, Tichy [136, 197]	Token	File	1:1		√	√	Merging
CFG	*Jdiff* [7]	CFG node		1:1	√		√	Regression testing Impact analysis
Binary	BMAT [294]	Code block		1:1 (procedure) n:1 (block)	√	√	√	Profile propagation Regression testing

N: Name-based heuristics, P: Position-based heuristics, S: Similarity-based heuristics

procedure level or at a file level. Based on Table 2.2 and 2.3, we conclude the following:

- Most matching techniques produce low-level differences without any structure; thus, matching results are difficult for programmers to inspect and reason about at a high-level.

- AST or CFG based techniques produce matches at fine-grained levels but are only applicable to a complete and parsable program. Researchers must consider the trade-off between matching granularity, requirements, and cost.

- Many techniques employ the LCS algorithm and thus inherit the assumptions of LCS: (1) one-to-one correspondences between matched entities and (2) linear ordering among matched pairs. Implicit assumptions like these must be carefully examined before implementing a matcher.

- Most techniques support only one-to-one mappings at a fixed granularity. Therefore, they will perform poorly when merging or splitting occurs.

- The more heuristics are used, the more matches can be found by complementing one another. For example, name-based matching is easy to implement and can reduce matching scope quickly, but it is not robust to renaming events. In this case, similarity-based matching can produce matches between renamed entities, and position-based matching can leverage already matched pairs to infer more matches.

2.2.2 Refactoring Reconstruction

Refactoring reconstruction (RR) techniques compare two program versions and look for a predefined set of refactoring patterns: move a method, rename a class, add an input parameter, etc. While code matching tools stop at mapping corresponding code elements, RR tools infer refactorings that explain the identified correspondences at the level of a function, class, or file.

Table 2.3: Evaluation of the surveyed code matching techniques

Program Representation	Citation	Scenario		Transformations				Strength and Weakness
				Split/Merge		Rename		
		1	2	Proc	File	Proc	File	
String	*diff* [139]	M	P	P	P	M	P	− sensitive to file name changes
	bdiff [280]	G	P	M	P	M	P	+ can trace copied blocks
AST	*cdiff* [304]	P	P	P	P	P	P	− sensitive to nested level change − require procedure level mappings
	Neamtiu et al. [230]	P	P	P	P	P	P	− partial AST matching
	Hunt, Tichy [136, 197]	M	P	P	P	G	P	− require file level mappings + can identify procedure renaming
CFG	*Jdiff* [7]	G	M	P	P	M	M	+ robust to signature changes − sensitive to control structure changes
Binary	BMAT [294]	P	G	P	P	G	G	+ robust to procedure renaming − assume 1:1 procedure correspondence − only applicable to binary programs

G: good M: mediocre P: poor

Demeyer et al. [65] first proposed the idea of inferring refactoring events by comparing the two programs. Demeyer et al. used a set of ten characteristics metrics such as LOC and the number of method calls within a method (i.e., fan-out) and inferred refactorings based on the metric values and a class inheritance hierarchy.

Zou and Godfrey's origin analysis [316] matches procedures using multiple criteria (e.g., names, signatures, metric values, callers, and callees) and infers merging, splitting, and renaming events. It is semi-automatic in the sense that a programmer must manually tune matching criteria and select a match among candidate matches.

Kim et al. [167] automated Zou and Godfrey's procedure renaming analysis. In addition to matching criteria used by Zou and Godfrey, Kim et al. used clone detectors such as CCFinder [149] and Moss [2] to calculate content similarity between procedures. An overall similarity is computed as a weighted sum of each similarity metric, and a match is selected if the overall similarity is greater than a certain threshold.

A renaming detection tool by Malpohl et al. [197] aligns tokens using *diff* and infers a function or variable renaming when distinct tokens are surrounded by mapped token pairs. Similarly, Neamtiu et al.'s analysis [230] detects a function or variable renaming based on the syntactic position of tokens.

Rysselberghe and Demeyer [263] use a clone detector, Duploc [75], to detect moved methods. Another similar approach by Antoniol et al. [5] identifies class-level refactorings using a vector space information retrieval approach.

Xing and Stroulia's UMLDiff [302] matches packages, classes, interfaces, fields and blocks based on their name and structural similarity metrics in a top-down order. After matching code elements, UMLDiff infers refactorings as well as other structural changes.

Dig et al.'s *Refactoring Crawler* identifies refactorings in two stages. First it finds a list of code element pairs using *shingles* (a metric-based fingerprint) and performs a semantic analysis based on reference relationships (calls, instantiations, or uses of types, import statements). The second part of the algorithm is an iterative, fix point algorithm that considers refactorings in a top-down order.

Weißgerber and Diehl's technique [295] identifies refactoring candidates using only names and signatures then uses clone detection results to rank the refactoring candidates.

Fluri et. al.'s Change Distiller [90] compares two versions of abstract syntax trees; computes tree-edit operations; and maps each tree-edit to atomic AST-level change types (e.g., parameter ordering change) [89]. The identified change types are largely categorized into two: (1) body-part changes and (2) declaration-part changes that include refactorings such as access modifier changes, final modifier changes, attribute declaration changes, method declaration changes, etc. Change Distiller was built for in-depth investigation of why change coupling occurs in the evolution of open source projects.

Table 2.4 and Table 2.5 compare these RR techniques and our rule-based change inference techniques (Chapters 5 and 6) in terms of matching granularity, matching multiplicity, matching heuristics and the type of inferred change operations.

The main difference between our change-rule representations and the representations used by RR techniques is that none of the RR techniques explicitly represent systematic changes in a formal syntax nor identify exceptions to systematic change patterns. RR techniques report a list of refactorings, leaving it to the programmer to identify emerging change patterns by discovering insights about the relationships among the particular set of refactorings. In addition to this inherent limitation, many RR techniques have the following two limitations that result from their implementation and algorithm choices.

First, many RR techniques do not consider the global context of transformation (in part due to their lack of explicit high-level change representations), and thus they either find too many refactoring candidates, leading to many false positives or find too few refactoring candidates, leading to many false negatives. To cope with a large number of false positives, RR techniques often need to post-process their results. For example, Weißgerber and Diehl's technique initially finds a large number of false positive refactorings; Hence it uses clone detection results to rank the refactoring candidates. S. Kim et al.'s technique requires users to reduce false positives by optimizing each similarity factor's weight and tuning the threshold value. Second, many RR techniques cannot find multiple refactorings that affect the same code elements as they impose a certain order in mapping code elements or do not consider the possibility of overlapping transformations. For example, UMLDiff cannot identify changes that involve both renaming and moving such as *"move Foo class from package A to package B and rename Foo to Bar"* as UMLDiff maps code in a top down

order.

2.2.3 Identification of Related Change

Several techniques use historical change data or program structure information to identify code elements that frequently change together. These techniques are similar to our rule-based change inference techniques in that they group a set of related low-level changes to discover a high-level logical change.

Gall et al. [93] were the first to use release data to detect logical coupling between modules. They developed a change sequence analysis and applied this analysis to a 20-release history of a large telecommunication system. Zimmermann et al. [312] applied association rule learning to version history data to discover which files, classes, methods, and fields frequently change together. Similarly, Ying et al. [308] applied a similar technique to the Eclipse and Mozilla change history. Like our change-rule inference techniques, these approaches suggest a potential missing change. However, they do not explicitly group systematic changes nor report their common structural characteristics, leaving it to programmers to figure out why some code fragments change together. For example, Rose [312] may report that methods `foo` and `bar` are likely to change together but does not report that both methods are called by the same method `fun`.

Hassan and Holt [119] proposed several heuristics that predict which code elements should change together and evaluated these heuristics using open source projects' evolution histories. For example, the *Historical Co-change* heuristic recommends code elements that changed together in previous change sets, the *Code Structure* heuristic recommends all code elements related by static dependencies (calls, uses, and defines), the *Developer-Based* heuristic recommends all code elements that were previously modified by the same developer, and the *Code Layout* heuristic recommends all entities that are defined in the same file as the changed entity. Surprisingly, their study found that the history-based or code layout based heuristic outperform code structure based heuristics.

Crisp [50] groups code changes using four predefined rules. While Crisp's goal is to create a compilable intermediate version for fault localization, the goal of this dissertation

Table 2.4: Comparison of refactoring reconstruction techniques (1)

Authors	Citation	Granularity	Multiplicity	Heuristics	Change operations
Demeyer at al.	[65]	Method Class	1:1, 1:n, n:1	Method size Class size Inheritance metric	Split into superclass (or subclass) Merge with superclass (or subclass) Move to super, sub or sibling class Split method Factor out common functionality
Zou and Godfrey	[316]	Function File Subsystem	1:1, 1:n, n:1	Metric values Callers and callees Function name and signature	Move, rename Merge, split, replacement
S. Kim et al.	[167]	Function	1:1	Function name and signature Callers and callees Clone detection results (CCFinder, Moss) Function body diff Complexity metrics	Move and rename
Antoniol et al.	[5]	Class	1:1, 1:n, n:1	Vector space model similarity	Merge, split, replacement
Malpohl et al.	[197]	Token	1:1	Syntactic position similarity	Rename
Neamtiu et al.	[230]	Type, Var	1:1	Name similarity Syntactic position similarity	Rename

Table 2.5: Comparison of refactoring reconstruction techniques (2)

Authors	Citation	Granularity	Multiplicity	Heuristics	Change Operations
Rysselberghe and Demeyer	[263]	Line	n:n	Clone detection results (Dup)	Move method
Xing and Stroulia (UMLDiff)	[302]	Package Class Method Field Block	1:1	Name similarity Containers References	Rename and move Signature changes Changes to structural dependencies (Inheritance, calls, data accesses, containment)
Dig et al. (Refactoring Crawler)	[70]	Package Class Method	1:1, 1:n	Clone detection results (Shingles) References	Rename package (or class, method) Pull up method Push down method Move method Change method signature
Weißgerber and Diehl	[295]	Class Method Field	1:1	Clone detection results (CCFinder) Name and signature	Move and rename Change method signature Visibility change
Fluri et al. (Change Distiller)	[90]	AST node	1:1	bi-gram string similarity subtree similarity	16 change types
Kim et al. (Chapter 5)	[164]	Method	1:1 or n:1	Name and signature similarity	Rename and move Signature changes
Kim et al. (Chapter 6)		Package Class Method Field	1:1	Name and signature similarity	Changes to structural dependencies (inheritance, calls, data accesses containment, overriding)

is to help programmers understand code changes by recovering a latent systematic structure in program differences.

2.3 Recording Change

Recorded change operations can be used to help programmers reason about software changes. Section 2.3.1 describes techniques that capture change operations in an editor or an integrated development environment. Section 2.3.2 describes source code transformation languages, which can serve as a basis for capturing high-level semantic transformations.

2.3.1 Edit Capture and Replay

Several editors or integrated development environment (IDE) extensions capture and replay keystrokes, editing operations, and high-level update commands to use the recorded change information for intelligent version merging, studies of programmers' activities, and automatic updates of client applications.

For example, Dig et al.'s MolhadoRef [71] automatically resolves merging conflicts that a regular *diff*-based merging algorithm cannot resolve by taking into account the semantics of recorded move and rename refactorings. This algorithm extends Lippe's operation-based merging [190] by defining a model of merging conflicts in case of rename and move refactorings. While Lippe's operation-based merging only defined abstract change operations and did not have a means of recording change operations in IDE, MolhadoRef implements refactoring-aware version merging by recording refactoring commands in the Eclipse IDE.[4]

Henkel and Diwan's CatchUp [123] captures API refactoring actions as a developer evolves an API and allows the users of the API to replay the refactorings to bring their client software up to date.

Robbes [257] extended a small talk IDE to capture AST-level change operations (creation, addition, removal and property change of an AST node) as well as refactorings. He used the recorded changes to study when and how programmers perform refactorings.

[4]Refactoring-aware version merging is one instance of version merging algorithms. A survey of version merging algorithms and tools is described in [208].

Evans et al. [83] collected students' programming data by capturing keystroke, mouse and window focus events generated from the Windows operating system and used this data to observe programming practices. Likewise, in our copy and paste study (Chapter 3), we recorded keystrokes and edit operations in an Eclipse IDE to study copy and paste programming practices.

When recorded change operations are used for helping programmers reason about software changes, this approach's limitation depends on the granularity of recorded changes. If an editor records only keystrokes and basic edit operations such as cut and paste, it is a programmer's responsibility to raise the abstraction level by grouping keystrokes. If an IDE records only high-level change commands such as refactorings, programmers cannot retrieve a complete change history. In general, capturing change operations to help programmers reason about software change is *impractical* as this approach constrains programmers to use a particular IDE.

2.3.2 Source Code Transformation Tools

Source transformation tools allow programmers to author their change intent in a formal syntax and automatically update a program using the change script. Most source transformation tools automate repetitive and error-prone program updates. The most ubiquitous and the least sophisticated approach to program transformation is text substitution. More sophisticated systems use program structure information. For example, A* [179] and TAWK [113] expose syntax trees and primitive data structures. Stratego/XT[288] is based on algebraic data types and term pattern matching. These tools are difficult to use as they require programmers to understand low-level program representations. TXL [56] attempts to hide these low-level details by using an extended syntax of the underlying programming language. Boshernitsan et al.'s iXJ [39] enables programmers to perform systematic code transformations easily by providing a visual language and a tool for describing and prototyping source transformations. Their user study shows that iXj's visual language is aligned with programmers' mental model of code changing tasks. Erwig and Ren [82] designed a rule-based language to express systematic updates in Haskell. Coccinelle [238] allows

programmers to safely apply crosscutting updates to Linux device drivers.

While these tools focus on applying systematic changes to a program, our approach focuses on recovering systematic changes from two versions. Despite the significant difference in their goals, both approaches' change-representations capture systematic changes concisely and explicitly. In theory, one can build a program differencing tool using a source transformation tool's change-representation by (1) automatically enumerating potential transformations, (2) applying the transformations to the old program version, and (3) checking whether the updated program is the same as the new program version. However, the change-representation's granularity and expressive power will affect its use for high-level reasoning of program differences. With a specific focus on high-level reasoning of software change, we compare our rule-based change representations (Chapters 5 and 6) with two representative source transformation languages and tools, TXL [56] and iXj [39].

Comparison with TXL TXL is a programming language and rapid prototyping system specifically designed to support structural source transformation. TXL's source transformation paradigm consists of parsing the input text into a structure tree, transforming the tree to create a new structure tree, and unparsing the new tree to a new output text. Source text structures to be transformed are described using an unrestricted ambiguous context free grammar in extended Backus-Nauer (BNF) form. Source transformations are described by example, using a set of context sensitive structural transformation rules from which an application strategy is automatically inferred.

Each transformation rule specifies a *target type* to be transformed, a *pattern* (an example of the particular instance of the type that we are interested in replacing), and a *replacement* (an example of the result we want when we find such an instance). In particular, the pattern is an actual source text example expressed in terms of tokens (terminal symbols) and variables (non-terminal types). When the pattern is matched, variable names are bound to the corresponding instances of their types in the match. Transformation rules can be composed like function compositions. Figure 2.1 shows an example TXL rule that replaces `(1+1)` expressions with `2`.

TXL's transformation rules in general require programmers to obtain the knowledge of

```
rule addOnePlusOne % target structure
replace [expression] % pattern to search for
1+1
by  2
% replacement to make
end rule
```

Figure 2.1: Example TXL rule

syntax trees. Though it is well suited for systematic changes at an expression level, it is less suited for expressing systematic changes at a higher abstraction level such as moving a set of classes from one package to another package. In addition, as our change-rules abstract a program at the level of code elements and structural dependencies, our approach finds systematic change patterns even when the constituent transformations are not exactly the same; For example, adding call dependencies to a particular method is grouped as a single rule even if the input parameters in the call invocation statements vary.

Comparison with iXj iXj's pattern language consists of a *selection pattern* and a *transformation action*. A selection pattern is similar to our rules' antecedent, and a transformation action is similar to our rules' consequent. Similar to our API change-rules, iXj's transformation language allows grouping of code elements using a wild-card symbol `*`. Figure 2.2 shows an example selection pattern and a transformation pattern.

To reduce the burden of learning the iXj pattern language syntax, iXj's visual editor scaffolds this process through from-example construction and iterative refinement; When a programmer selects an example code fragment to change, iXj automatically generates an initial pattern from the code selection and visualizes all code fragments matched by the initial pattern. The initial pattern is presented in a pattern editor, and a programmer can modify it interactively and see the corresponding matches in the editor. A programmer may edit the transformation action and see the preview of program updates interactively.

Selection pattern: `* expression instance of java.util.Vector (:obj).removeElement(:method)(* expressions(:args))`

[Interpretation: Match calls to the removeElement() method where the obj expression is a subtype of java.util.Vector.]

Transformation action: `$obj$.remove($obj$.indexOf($args$))`

[Interpretation: Replace these calls with with calls to the remove() method whose argument is the index of an element to remove.]

Figure 2.2: Example iXj transformation

Similar to TXL, iXj's transformation language works at the level of syntax tree nodes, mostly at an expression level. Thus, it is not effective for expressing higher-level transformation such as moving a set of related classes from one package to another package. Its transformation actions are more expressive than our change-rules in that they support free-form text edits.

2.4 Code Clones

This section describes automatic clone detection techniques, studies of clone coverage, clone re-engineering techniques, empirical studies about code cloning practices, and clone management techniques.

2.4.1 Automatic Code Clone Detection

Although most consider code clones to be identical or similar fragments of source code [23, 149], code clones have no consistent or precise definition in the literature. Indeed, a "clone" has been defined operationally based on the computation of individual clone detectors.

Clone detectors can be grouped into four basic approaches, each of which uses a different representation of source code and different algorithms for comparing the representation of potential clones.

First, some detectors are based on lexical analysis. For instance, Baker's [12] *Dup* uses a

lexer and a line-based string matching algorithm. *Dup* removes white spaces and comments; replaces identifiers of functions, variables, and types with a special parameter; concatenates all files to be analyzed into a single text file; hashes each line for comparison; and extracts a set of pairs of longest matches using a suffix tree algorithm. Kamiya et al. improved *Dup*'s algorithm and developed CCFinder [149], which transforms tokens of a program according to a language-specific rule and performs a token-by-token comparison. CCFinder is recognized as a state of the art clone detector that handles industrial size programs; it is reported to produce higher recall although its precision is lower than some other tools [44]. *CP-Miner* [187] identifies a similar sequence of tokenized statements using a frequent subsequence mining technique.

Second, Baxter et al. developed *CloneDr* [26], which parses source code to build an abstract syntax tree (AST) and compares its subtrees by characterization metrics (hash functions). Jiang et al. [146] and Koschke et al. [175] also developed AST-based clone detection algorithms.

Third, some detectors find clones by identifying an isomorphic program dependence graph (PDG). Komondoor and Horwitz's clone detector finds isomorphic PDG subgraphs using program slicing [173]. Krinke uses a k-length patch matching to find similar PDG subgraphs [176]. PDG-based clone detection is robust to reordered statements, code insertion and deletion, intertwined code, and non-contiguous code, but it is not scalable to large programs.

Finally, metric-based clone detectors [148, 202, 212] compare various software metrics called fingerprinting functions. These clone detectors find clones at a particular syntactic granularity such as a class, a function, or a method, because fingerprinting functions are often defined for a particular syntactic unit.

Comparative studies of clone detectors have been done by Burd and Bailey [44], Ryssel-berghe and Demeyer [264], and Walenstein et al. [291].

2.4.2 Studies of Clone Coverage

Several studies have investigated the extent of code clones in software. Comparing the result of these studies is difficult because the definition of a clone depends on the computation of individual clone detectors and many detection algorithms take adjustable parameters. Nearly as much as 10% to 30% of the code in many large scale projects was identified as code clones (e.g., *gcc*-8.7% [75], *JDK*-29% [149], *Linux*-22.7% [187], etc). Antoniol et al. [6] and Li et al. [187] studied changes in clone coverage (the ratio of cloned code to the total lines of code) in Linux and found that clone coverage increased early in development but stabilized over time. They interpreted these data as showing that the design of Linux is not deteriorating due to copy and paste practices. These quantitative studies of clones cannot answer questions about why programmers create and maintain code clones and how clones actually evolve over time—for example, how often do clones actually require consistent updates during evolution? Our copy and paste programming study and clone genealogy study answer such questions by focusing on the evolutionary aspects of clones (Chapter 3 and Chapter 4).

2.4.3 Clone Reengineering

Researchers have also used the output of a clone detector as a basis for refactoring. For example, Balazinska et al. developed a clone reengineering tool, called *CloRT* [16, 15]. *CloRT* finds clones using software metrics and a dynamic pattern matching algorithm, determines whether the *Strategy* or *Template* design pattern applies to these clones, factors out the common parts of methods, and parametrizes the differences with respect to the design patterns. As another example, Komondoor and Horwitz developed a semantics-preserving procedure extraction algorithm that runs on PDG-based clones [173, 174]. Finally, *CCShaper* [125] filters the output of CCFinder to find candidates for the *extract method* and *pull up method* refactoring patterns. *CCShaper*'s improved version, *Aries* [126] suggests which refactoring method to use to remove code clones based on the positional relationship of code clones in the class hierarchy and the coupling between code clones and their surrounding code.

Language-based approaches such as multiple inheritance, mixins, use of delegation,

traits, and parametrization can remove a certain type of code duplication. For example, Murphy-Hill et al. [227] removed similar method implementations in the Java.io library using traits, a modularity mechanism that allows sharing of pure method implementations across the inheritance hierarchy. Jarzabek et al. [145] and Basit et al. [23] used a meta language XVCL to reduce code duplication. XVCL allows parametrization via meta-variables and meta-expressions, insertions of code at designated break points, selection among given options based on conditions, code generation by iterating over selections of meta-components, etc. Jarzabek et al. [145] report that XVCL reduced 68% of code duplication in the Java buffer library. Basit et al. [23] report that C++ standard libraries (STL) made heavy use of generics but duplication still existed because similar functionality had to be replicated in a slightly different way.

2.4.4 Studies about Cloning Practice

Several studies share a view to ours (Chapter 3 and Chapter 4) that code cloning is not necessarily a harmful software engineering practice.

Cordy [55] notes that cloning is a common method of risk minimization used by financial institutions because modifying an abstraction can introduce risks of breaking existing code. Fixing a shared abstraction is both costly and time consuming as it requires any dependent code to be extensively tested. On the other hand, clones increase the degrees of freedom in implementing and maintaining each new application or module—each is free to refine its view of the data. Cordy noted that propagating bug fixes to code clones is not always a desired practice because the risk of changing an already properly working module is too high.

Godfrey et al. [103] conducted a preliminary investigation of cloning in Linux SCSI drivers and found that super-linear growth in the Linux system is largely caused by cloning of Linux drivers. Kapser and Godfrey [152] extended this work and further studied cloning practices in several open source projects (Linux operating system kernel, Postgresql relational database management systems, Apache httpd web server, and Gnumeric spreadsheet) and found that clones are not necessarily harmful. Developers create new features by start-

ing from existing similar ones, as this cloning practice permits the use of stable, already tested code. In fact, they report that about 71% of the clones could be considered to have a positive impact on the maintainability of the software system. Their study also cataloged four major reasons of why programmers clone code: *Forking* is used to bootstrap development of similar solutions with the expectation that clones will evolve somewhat independently at least in the short term to accommodate hardware variations or platform variations. *Templating* is used to directly copy the behavior of existing code when appropriate abstraction mechanisms such as inheritance or generics are unavailable. *Customization* occurs when currently existing code does not adequately meet a new set of requirements. *Exact match* duplication is typically used to replicate simple solutions or repetitive concerns within the source code.

While interviewing and surveying developers about how they develop software, LaToza et al. [185] uncovered six patterns of why programmers create clones: repeated work, example, scattering, fork, branch, and language. For each pattern, less than half of the developers interviewed thought that the cloning pattern was a problem. LaToza et al.'s study confirms that most cloning is unlikely to be created with ill intentions. This study also reports that programmers are concerned with large scale cloning such as copying a module created by another team (forking), rather than copying example code fragments, subclasses, or code fragments that are hard to refactor.

In a recent study, Rajapakse et al. [251] experimented with using Server page techniques to eliminate code clones in a web application. They found that reducing duplication in a web application had negative effects on the extensibility of an application: After significantly reducing the size of the source code, a single change often required testing a vastly larger portion of the system. Their study also suggests that avoiding cloning during initial development could contribute to a significant overhead.

2.4.5 Clone Management

Miller and Myers [215] proposed simultaneous text editing to automate repetitive text editing. After describing a set of regions to edit, a user can edit any one record and see

equivalent edits simultaneously applied to all other records. A similar editing technique, called Linked-Editing [281] applies the same edits to a set of code clones specified by a user. Though these tools were developed prior to our clone genealogy study in 2005 (Chapter 4), our study results provide the reasons why these tool-based approaches are useful for evolving code clones and suggest when and how often these tools are complementary to using refactoring techniques.

Our clone genealogy study (Chapter 4) motivated clone tracking and management techniques. Duala-Ekoko and Robillard [74] developed an Eclipse plug-in that takes the output of a clone detector as input and automatically produces an abstract syntax-based clone region descriptor for each clone. Using this descriptor, it automatically tracks clones across program versions and identifies modifications to the clones. Similar to the Linked-Editing tool by Toomim et al. [281], it uses the longest common subsequence algorithm to map corresponding lines and to echo edits in one clone to other counterparts upon a developer's request.

2.4.6 Studies of Copy and Paste in Programming

Similar to our copy and paste (C&P) study in Chapter 3, there are several studies that address how code clones are entered into a system or why programmers duplicate code. To understand code reuse strategies in object oriented programming, Lange et al. [181] conducted a week long observation of a single subject programming in Objective-C. The investigators noted that the subject often copied a super-class or a sibling-class as a template for a new class then edited the copied class. Rosson et al. [261] observed four subjects programming in Smalltalk; when the subjects were interested in reusing a particular target class, they copied the usage protocol of the target class and used it as example code without deeply comprehending the behavior of the target class. Although they considered C&P as one strategy of source code reuse, they did not focus on the implication of C&P. In our study, we not only observed the same code reuse behavior but also analyzed why some code snippets are chosen as a template.

2.5 Software Evolution Analysis

Belady and Lehman [28] are the first to study evolving software systems. They proposed laws of software evolution after analyzing change data from the evolution of the OS/360 operating system. Since their work, many researchers have developed analysis and visualization techniques to study study evolution. Based on a brief survey of software evolution analysis and visualization techniques, we conclude that these techniques can benefit from our rule inference techniques' ability to extract concise, high-level change descriptions.

2.5.1 Empirical Studies of Software Evolution

Kemerer and Slaughter [154] manually coded over 25000 change logs to classify each change event to 6 types of corrective, 6 types of adaptive, and 6 types of perfective changes. Their analysis used phase mapping and gamma sequence analysis methods originally developed in social psychology to identify and understand the phases of software evolution.

Eick et al. [77] developed a process for analyzing the change history of the code, which is assumed to reside in a version management system, calculating code-decay indices, and predicting the fault potential and change effort through regression analysis. The objective of this research is to support project management so that code decay is delayed.

Hassan and Holt [118] studied the chaos of software systems in terms of information entropy—the amount of uncertainty related to software products. Intuitively, in the context of software evolution, if a software system is being modified across all its modules, it has high entropy, and the software maintainers will have a hard time keeping track of all the changes. Their work relies on maintenance documentation to keep track of software modifications in order to compute information entropy of files that evolved over a period of time.

The major drawback of this line of research is that it requires developers' comments recorded in the version management system. In most real-world software projects, comments are inconsistent in their detail and they often do not even exist.

Most refactoring reconstruction techniques are applied to some software evolution history to study software evolution. This type of study is limited by the accuracy of refactoring reconstruction and the kinds of refactorings targeted by each technique. Section 5.3 provides

more details on the accuracy and the kinds of refactorings.

2.5.2 Visualization of Software Evolution

There are several visualization techniques that focus on software evolution, in particular, changes in software-process statistics, source code metrics, static dependence graphs, *diff*-based deltas and their derivatives, etc.

Ball et al. [19] developed the one of the first systems that [19] explored visualizing software evolution data, in particular, the age of individual code lines as a color. Their work also visualizes program differences between two versions, which are calculated using *diff*.

Holt and Pak [130] visualized structural changes between two program versions by explicitly modeling code elements and their structural dependencies. Their visualization focuses on which structural dependencies are common, which dependencies are new, and which dependencies are deleted between two versions at the subsystem level.

Eick et al. [78] developed a number of views (matrix, cityscape, bar and pie charts, data sheets, and network) that facilitate rapid exploration of high-level structure in software evolution data and also serve as a powerful visual interface to the data details as needed. These visualization tools explicitly model logical software changes as their visualization is built upon proprietary evolution history data, where a set of related program deltas are grouped to a logical software change called a modification request (MR) and its change type is manually written by developers as an adaptive, corrective, or perfective change.

Pinzger et al.'s RelVis approach [245] condenses multi-dimensional software evolution metric data into two graphs. The first graph visualizes modules and their metrics over time. The second graph visualizes relationships between source code modules. In both graphs, the evolution of metrics is visualized using a Kiviat diagram where annual rings indicate metric values for each release.

Lanza and Ducasse's Polymetric views [183] is a lightweight software visualization technique enriched with software metrics information. Polymetric views help to understand the structure and detect problems of a software system in the initial phases of a reverse engi-

neering process by combining software visualization and software metrics. Lanza applied this general visualization technique to metric values over multiple program versions, and named this view an evolution matrix [182]. This view is instantiated at two granularity levels (a system level or a class level) and can help programmers understand how the system size grows, when and where classes are added or deleted, etc.

Girba et al. [100] introduced the *Yesterday's Weather* metric that can further condense historical change patterns at a class granularity. This metric is designed to help a programmer identify a candidate for further reverse engineering based on the observation that classes that changed most in the recent past are likely to undergo important changes in the near future.

Rysselberghe et al. [265] proposed a dot plot visualization of change data extracted from a version control system to identify unstable components, coherent entities, productivity fluctuations, etc.

These visualization techniques assume a substantial interpretation effort on behalf of their users and do not scale well. They become unreadable for a long evolution history of large systems with numerous components. In addition, many of these techniques are inherently limited by the source of history data—most version control systems consider a software system as a set of files containing lines of texts and consequently they report changes at the lexical level and are unaware of the high-level logical structural changes of the software system. We believe that our change-rule inference techniques (Chapter 5 and Chapter 6) can enable these visualization techniques to model software evolution more semantically and structurally.

2.6 Other Related Work

Logic-based Program Representation. Our change-rule inference technique in Chapter 6 models a program as a set of logic facts at a program structure level. Representing a program's code elements and structural dependencies as a set of logic facts (or a relational database) has been used for decades [189]. Approaches such as JQuery [144] or CodeQuest [115] automatically evaluate logic queries specified by programmers to assist program investigation. Mens et al.'s intentional view [205] allows programmers to specify concerns or

design patterns using logic rules. Eichberg et al. [76] use Datalog rules to continuously enforce constraints on structural dependencies as software evolves. Our change-rule inference techniques differ from these by (1) focusing on systematic differences between two versions, as opposed to regularities within a single version and (2) inferring rules without requiring the programmers to specify them explicitly.

One could apply fact extractors such as grok [131] to each of two program versions and use a set-difference operator to compute fact-level differences. Section 6.4 shows that although this approach computes accurate structural differences, those deltas would be quite large (often hundreds of facts) and thus more demanding on the programmer than our condensed rule representation.

Framework Evolution. We discuss several approaches that were developed to assist framework evolution as these approaches are similar to our LSDiff (Chapter 6) in that they model program changes at a structure level and identify systematic change patterns. However, these approaches differ from LSDiff by focusing on only API usage replacement patterns.

Dagenais and Robillard's SemDiff [62] monitors adaptive changes within a framework to recommend similar changes to its clients. SemDiff and LSDiff are similar in that both identify additions and deletions of methods and method-calls. SemDiff carries out a partial program analysis to find changes in the callers of a particular deleted API, consistent with its focus on framework evolution. In contrast, LSDiff uses the full logic-based program representation of two versions to infer change rules. Schäfer et al. proposed an approach that infers API usage replacement patterns as change-rules to assist framework evolution [270]. Although LSDiff infers a broader class of systematic changes, their underlying technology, developed independently, is similar to ours. At a more detailed level, LSDiff rules are more expressive than theirs. First, we infer first order logic rules with variables as opposed to association rules (propositional rules without variables). Variables in our rule representation allow explicit references to the same code elements, removing the need for context-based filtering. Second, their predefined rule patterns limit discovery of systematic changes that exhibit a combination of different types of structural characteristics such as subtyping and

method-calls.

Chapter 3

AN ETHNOGRAPHIC STUDY OF COPY AND PASTE PROGRAMMING PRACTICES

This chapter describes an ethnographic study of copy and paste programming practices. As a result of this study, we developed a taxonomy of common copy and paste patterns. This taxonomy has been useful in understanding systematic changes and building change-rule inference techniques that leverage systematicness at a code level.

The rest of this chapter is organized as follows. Section 3.1 details the observational study settings and the analysis method used. Sections 3.2, 3.3 and 3.4 describe the resulting taxonomy, focusing respectively on (1) *programmers' intentions*, (2) *design decisions* that cause programmers to copy and paste, and (3) the associated *maintenance tasks*. Section 3.5 presents how often and which granularity of text programmers copy and paste. Section 3.6 discusses possible flaws in the validity of this study and shares our conjectures about copy and paste patterns in different study settings. Section 3.7 summarizes the insights.

3.1 Study Method

We observed programmers performing coding tasks first by watching them directly and then by having them use an instrumented editor that logs their editing operations. In the latter case, we conducted follow-up interviews to understand programmers' tasks at a high level and to confirm our interpretation of their actions. We applied the affinity process [34] to the collected data to develop a taxonomy of copy and paste (C&P) patterns.

3.1.1 Observation

First, we observed participants by watching them program directly. We occasionally interrupted the participants' programming flow and asked them to explain what they were copying and pasting and why. Most participants voluntarily explained their intentions of

Table 3.1: Copy and paste observation study setting

	Direct Observation	Observation using a logger and a replayer
Subjects	Researchers at IBM T.J. Watson	
No. of Subjects	4	5
Total Coding Hours	About 10 hrs	About 50 hrs
Interviews	Questions asked during observation	Twice after analysis (30 minutes to 1 hour each)
Programming Languages	Java, C++, Jython	Java

C&P.

In order to enable participants to program in a more natural setting and to log editing operations with greater precision, we developed a logger and a replayer. Using the logger, we recorded coding sessions, then observed the participants' actions off-line by replaying the captured editing operations. For both types of observation, the participants were researchers at IBM T.J. Watson Research Center. They were expert programmers and were involved in small team research projects. In total, nine subjects participated in the study, and we observed about 60 hours of coding in object-oriented programming languages, mainly in Java. Observational study settings are summarized in Table 3.1.

3.1.2 Logger and Replayer

The logger efficiently records the minimal information required to reconstruct document changes performed by a programmer. The logger was developed by extending the text editor of the Eclipse IDE[1] and instrumenting text editing operations.[2] It records the initial contents of all documents opened in the workbench and logs changes in the documents. It records the type of editing operations, the file names of edited documents, the range of selected text and the length and offset of text entry, as well as editing operations such as copy, cut, paste, delete, undo, and redo. It also captures document changes triggered by

[1] http://www.eclipse.org

[2] The logger was built for Eclipse version 2.1.

automated operations such as refactoring and organizing import statements. Appendix A shows a sample edit log file in an XML format.

The replayer regenerates the programming context involved in document changes from the low level editing events captured by the logger. It displays documents and highlights changes and selected text. It has a few controls such as play, stop, and jump. While a videotape analysis of coding behavior normally takes ten times as long as the actual coding, by using the instrumented text editor and the replayer, replaying and reviewing the logs took only 0.5 to 1 times of the actual coding tasks.

3.1.3 Analysis Method

By replaying the editing logs, we documented individual instances of copy and paste operations. An instance consists of one copy (or cut) operation followed by one or more paste operations of the copied (or cut) text. It also includes other modifications performed on the original text or the cloned text. We categorized each instance with a focus on the procedural steps and the structural entity of copied (or cut) content. Since we observed multiple C&P instances that share similar editing steps, we generalized the editing procedures to identify C&P usage patterns. For example, one frequent C&P pattern was to repeatedly change the name of a variable. The renaming procedure consists of selecting a variable, copying the variable, pasting the variable n times, and optionally searching for the variable n times (where n is the number of appearances of the variable within its scope). For each generalized copy and paste ensemble, we inferred the associated programmer's intention. Inferring a programmer's intention was often straightforward. For example, *changing the name of a variable consistently* is the intention associated with the renaming procedure described above. For each C&P instance, we also investigated the relationship between a copied code fragment and code elsewhere in the code base, and analyzed the evolutionary aspect of the C&P instances by observing how duplicated code fragments were updated during our study.

After producing detailed notes for each C&P instance, we met with subjects to confirm our interpretation of their actions. Appendix B shows an example analysis note that we created by replaying one of the coding session logs and interviewing the participant. In

total, we analyzed 460 C&P instances.

To build a taxonomy of C&P patterns, we used the affinity process [34]. The affinity process is often used to gather insights or discover new patterns from large amounts of language data (ideas, opinions, and issues) by grouping them based on their natural relationships. Affinitizing is an interactive process often performed by a group or a team. First, the ideas or issues are written on post-its and are displayed on a wall. The team members start by looking for ideas that seem related in some way and place them together. They create header-cards that capture the essential link among the ideas contained in a group of cards. The product of the affinity process is a diagram which shows the groupings and header-cards.

We wrote post-it notes based on the detailed description of each C&P instance. Then we grouped related C&P instances and created header-cards for the identified groups of cards. As a result, we created an affinity diagram, which is shown in Figure 3.1. Appendix C includes each sub-part of the affinity diagram in detail.[3]

The following three sections present the resulting taxonomy, focusing respectively on the intention, design, and evolutionary aspects of C&P operations. Section 3.2 (Intention view) describes the categorization of programmers' intentions involved in copy and paste operations. Section 3.3 (Design view) describes the categorization of design decisions that caused programmers to copy and paste in particular patterns. Section 3.4 (Maintenance view) discusses maintenance tasks associated with copy and paste operations.

3.2 Intention View

The categorization of programmers' intentions was constructed by inferring intentions associated with common C&P patterns and by directly asking questions of the subjects. During the interviews, the inferred intentions were confirmed or corrected.

One use of C&P is to relocate, regroup, or reorganize code from one place to another according to the programmers' mental model of the program's structure. Programmers

[3]In the resulting diagram, there is no one-to-one mapping between C&P instances and post-it notes because the notes that represent similar intentions, design decisions, or maintenance tasks were merged during the affinity process.

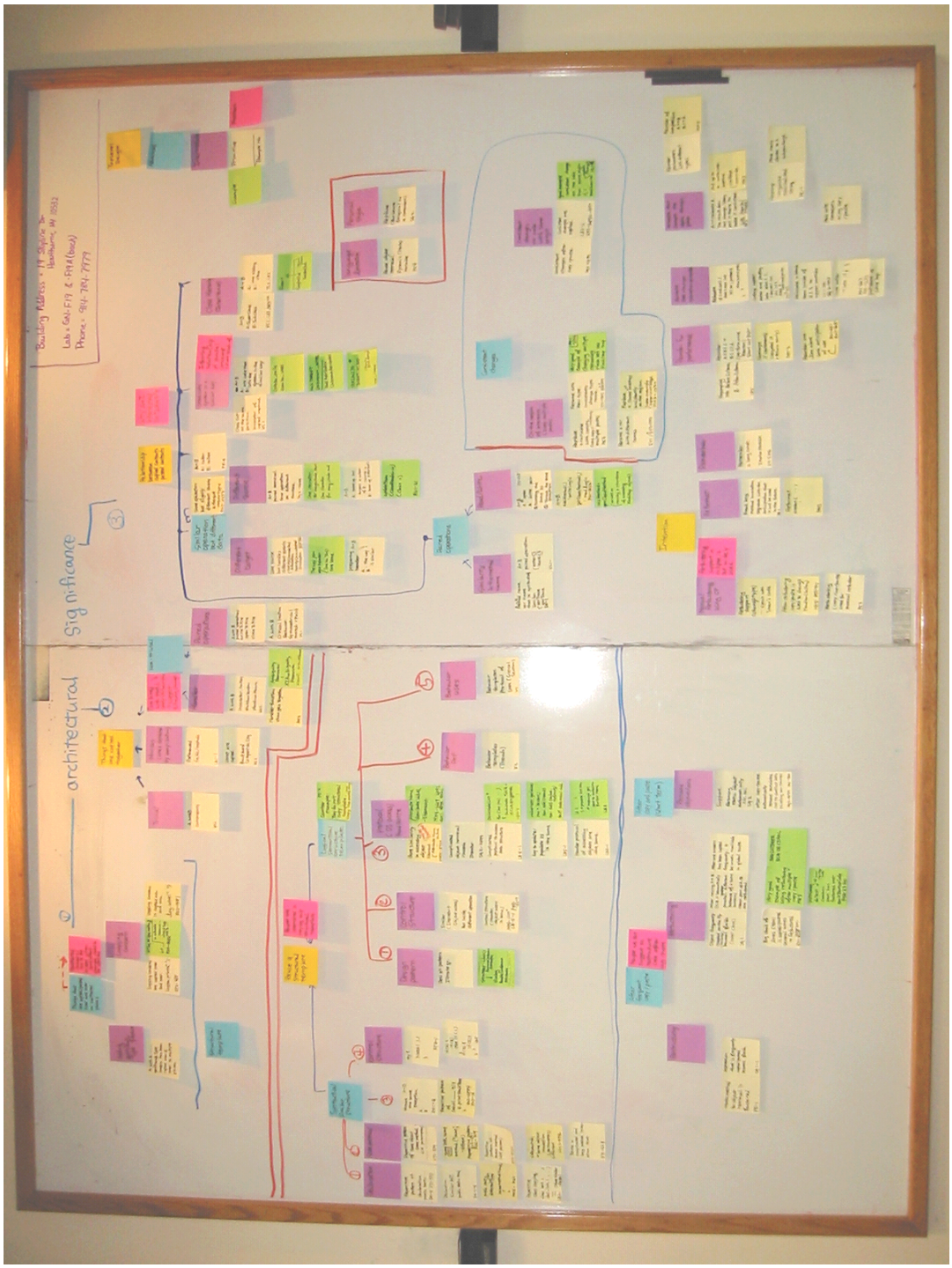

Figure 3.1: Affinity diagram representing copy and paste patterns. Detail diagrams appear in Appendix C.

```
static{
  protectedClasses.add("java.lang.Object");
  protectedClasses.add("java.lang.ref.Reference$ReferenceHandler");
  protectedClasses.add("java.lang.ref.Reference");
  protectedMethods.add("java.lang.Thread.getThreadGroup");
}
```

Figure 3.2: An example syntactic template

also use copy and paste to reorder code fragments. For example, a Boolean expression $(A \parallel B \parallel C)$ could be reordered as the equivalent expression $(B \parallel C \parallel A)$ to improve performance, or several if-blocks could be reordered so that negated if-statements return early. Programmers also use copy and paste to restructure (or refactor) their code manually. The most common copy and paste intention in our study was to use a copied code snippet as a *structural template* for another code snippet. Programmers often copied the entire code snippet and removed code that was irrelevant to the pasted context. The structural templates can be either reusable syntactic elements of code snippets (*syntactic templates*) or reusable programming logic (*semantic templates*). The usage of syntactic templates is explained through the example in Figure 3.2. The statement `protectedClasses.add("java.lang.Object")` was copied multiple times. The duplicates were modified after they were pasted. The programmer intended to reuse `protectedXXX.add("java.lang.YYY")` as a template for other statements in the static method initialization. The lack of functionality in today's IDEs and/or and limitations in language constructs also increase the need for copying syntactic templates. For example, the absence of repetitive text editing support in an IDE or the lack of the `enum` construct in Java 1.4 causes programmers to copy and paste a particular phrase frequently. In the following examples, copied text is represented as copied text (with wavy underline), pasted text is represented as pasted text *(italic)*, deleted text is represented as deleted text (~~with strike-through~~), and cut text is represented as cut text (~~with strike-through and underline~~). Modifications performed on top of pasted text are represented as modified text (with solid underline).

The following four subsections categorize the use of semantic templates.

```
DOMNodeList *children = doc->getChildNodes();
int numChildren = children->getLength();

for (int i=0; i<numChildren; ++i){
  DOMNode *child = (children->item(i));
  if (child->getNodeType() ==
          DOMNode.ELEMENT_NODE)
        {
            DOMElement *element = (DOMElement*)child
```

Figure 3.3: Code fragment: traversing over element nodes in a DOM document in C++

Design Pattern In our study, we observed a case where a programmer copied the usage of a Strategy design pattern [94]. The programmer used a concrete instantiation of the Strategy pattern as a template, because it is easier than writing code from an abstract description of that design pattern.

Usage of a Module (Class) Programmers often copy a code snippet to reuse the usage protocol of a target module [261]. We observed many cases where a code snippet was copied because it contains logic for accessing a frequently used data structure. Programmers are often required to know the usage protocol for data structures that they intend to use. For example, in order to traverse keys in a `Hashtable`, a programmer needs to get a reference for a key set by invoking the `keySet()` method on the hashtable object and then obtain an iterator for the key set. We observed a number of similar cases in our study. For example, the code snippet in Figure 3.3 was copied because it contains code for traversing over Element nodes in a DOM Document in C++.

Implementation of a Module Programmers often copy a code snippet that contains a definition of particular behavior—the signature and some partial implementation. This duplication can be removed by inheriting abstract classes or interfaces.

```
for (Iterator it=messages.iterator();it.hasNext();) {
  Message curr= (Message) it.next();
        IFile markFile=
        WorkspaceUtils.getFile(curr.getFirstLocation().getClassName());
        ...
}
for (Iterator it=messages.iterator();it.hasNext();) {
        Message curr= (Message) it.next();
        MessageLocation loc = curr.getLastLocation();
        IFile markFile=
        WorkspaceUtils.getFile(loc.getLocation().getClassName());
        ...
}
```

Figure 3.4: Copying a loop construct and modifying the inner logic

Control Structure Programmers frequently reuse complicated control structures (e.g., a nested `if then else` or a loop construct). When programmers intend to write code that has the same control structure but different operations inside the control structure, they tend to copy the code with the outer control structure and modify its inner logic. Figure 3.4 shows a for-loop that was modified after copy and paste.

3.3 Design View

Unlike the intention view where we analyzed code snippets involved in each C&P instance in isolation, in the design view, we analyzed the code snippets in relation to other code snippets in the system. We asked several questions to understand the architectural (or design) context of copy and paste operations. Each subsection discusses why we chose each question and describes the categorization of answers to the question.

3.3.1 Why is Text Copied and Pasted Repeatedly in Multiple Places?

The underlying premise of the Aspect Oriented Programming is that primary design decisions that are already in a system sometimes do not allow the secondary design decisions

```
if (logAllOperations) {
        try{
                PrintWriter w = getOutput();
                w.write("$$$$$");
                ...
        } catch (IOException e) {
        }
```

Figure 3.5: Code fragment: logging concern

to be modularized when they are added to the system [158, 279]. The lack of modularity leads programmers to insert similar code snippets across a code base—which we observed in our study. For example, the logging concern in Figure 3.5 was copied four times within one file, and many more times across the code base. Because it is difficult to generalize the list of arguments for the factored logging function, refactoring this code snippet is often less preferable than copying the code snippet. In addition, even if the programmer chooses to refactor it, the dependencies between the logging module and the other modules would remain entangled in Java unless an aspect-oriented language such as AspectJ [158] is used. For the same reason, adding a feature sometimes requires making changes in scattered places across a code base. In one project that we observed, a programmer added a feature to display a user-friendly type for internal objects instead of the internally used XML type for the objects in his software. First, he wrote the body of `getFriendlyTypeName()` and duplicated it in four different classes. When he realized that it was better to refactor the code into a separate method, he copied the body of `getFriendlyTypeName()` and pasted it into the `MiscOps` class. He then copied and pasted the invocation statement of `MiscOps.getFriendlyTypeName()` four times to call the refactored method.

3.3.2 Why are Blocks of Text Copied Together?

When a code snippet is copied from A and pasted to B, related code snippets are also often copied from A and pasted to B. Code snippets that are often copied together belong to the same functionality or concern:

Comments A comment is copied when its related code is copied.

Referenced fields/constants Programmers copy referenced fields and constants when they copy a method that refers to them.

Caller method and Callee method Programmers copy a referenced method when they copy a method or a class that invokes the method. Similarly, a caller method is copied when its called method is copied. In our study, a programmer copied the contents of the `sender.cpp` file to `heartbeat.cpp` in order to create a heartbeat thread that has similar behavior to the sender thread. After he finished modifying `heartbeat.cpp`, he copied the invocation statement of `start_sender()` and pasted it as the invocation statement of `start_heartbeat()` in the test driver file. He also copied the invocation of `shutdown_sender()` and pasted it as the invocation of `shutdown_heartbeat()`.

Paired operations Programmers copy and paste paired operations together. For example, when a programmer copies `writeToFile()`, he also copies `openFile()` and `closeFile()`. Likewise, when `enterCriticalSection()` is copied, `leaveCriticalSection()` is copied as well.

3.3.3 What is the Relationship between Copied and Pasted Text?

We raised this question to understand why programmers choose a code fragment as a template.

Similar Operations but Different Data Sources This category is a special case of semantic templates where the duplicated code snippets manipulate different data sources. In our study, error messages were sent from one stage to the next stage by calling method A. The same error messages are also sent to a user by invoking method B. A is copied and used as a template for B, because A and B contain logic for reading the same header, only differing in the targets to which they direct error messages.

In Figure 3.6, the `updateFrom (Class c)` method is used as a template for the `updateFrom (ClassReader cr)`. Both methods contain logic for populating the same data structure. While

```
public void updateFrom  (Class c ) {
        String cType = Util.makeType(c.getName());
        if (seenClasses.contains(cType)) {
              return;
        }
        seenClasses.add(cType);
        if (hierarchy != null) {
              addToHierarchyViaReflection(c);
        }
        if (methods != null) {
               Method[] ms = c.getDeclaredMethods();
               for (int i = 0; i < ms.length; i++) {
                      Method m = ms[i];
                      methods.addMethod(cType,
                      m.getName(),
                      Util.computeSignature (m.getParameterTypes(),
                      m.getReturnType()),
                      m.getModifiers());
... }
public void updateFrom  (ClassReader cr ){
        String cType = CTDecoder.convertClassToType(c.getName());
        if (seenClasses.contains(cType)) {
              return;
        }
        seenClasses.add(cType);
        if (hierarchy != null) {
              CTUtils.addClassToHierarchy(hierarchy, cr);
        }
        if (methods != null) {
              int count = cr.getMethodCount();
              for (int i = 0; i < count; i++) {
                      Method m = ms[i];
                      methods.addMethod(cType,
                      cr.getMethodName(i),
                      cr.getMethodType(i),
                      cr.getMethodAccessFlags(i));
... }
```

Figure 3.6: Code fragments: `updateFrom(Class c)` and `updateFrom (ClassReader cr)`

one method reads from a class object through Java reflection, the other reads from Java byte code.

Semantically Parallel Concerns We define semantically parallel concerns as design decisions that crosscut a system in a similar way. While aspects often refer to similar code appearing in multiple places, semantically parallel concerns refer to a group of concerns that bears similarity to another group of concerns. Semantically parallel concerns are related to Griswold's information transparency principle [111] in the sense that these concerns are often encoded with the same signature, such as the use of particular variables, data structures, language features or even similar comments.

We observed one project that involves extending a compiler to support processing XML DOM objects. At the time of the observation, the compiler already had code related to the `serialize` concern and the subject wanted to insert code related to the `appendChildren` concern. Although these two concerns are independent, the `appendChildren` concern should be inserted into the same places where the `serialize` concern appears. The programmer identified all the code related to the `serialize` concern by searching the code base with the keyword `serialize`. The programmer then copied the identified code snippets and modified them as necessary for the `appendChildren` concern. When we asked the programmer about why he programmed in such way, he answered that those concerns crosscut the same places in the compiler architecture and it helped him to keep track of which part of the system to extend. Griswold observed a similar case when C-Star was retargeted to Ada [111]; the pipeline architecture of C-Star guided the programmer to identify all the code related to C syntax specific support and convert it to Ada syntax specific support.

Paired Operations In Section 3.3.2, we mentioned paired operations that are copied together frequently. In this section, we discuss paired operations as a special case of sharing the usage of the same data structure.

For example, in Figure 3.7 the `addMethod()` method was used as a template for the `getClassMethod()` method, because the `addMethod()` and the `getClassMethod()` access a *hashmap* where each value of *(key,value)* pairs can be either a single object or an array list of multiple

```
public void addMethod {
        // retrieve a map
        if (map == null) {
                // create a map
        }
        // get an entry o
        if (o == null) {
                // add that method into a map and return
        }
        if (o instanceof ArrayList) {
                // cast
        } else {
                // create an array list and add it to a map
        }
        // add a method to the array list
public MethodInfo getClassMethod(
        // retrieve map
        if (map == null) {
                // return null
        }
        // get an entry o
        if (o == null) {
                // return null
        }
        if (o instanceof ArrayList) {
                // traverse each method "m" in the array list, and if matches, return "m"
        } else {
                // if signature matches, return that method
        }
```

Figure 3.7: Code fragments: write/read logic

objects. `getClassMethod()` contains read logic that pairs with write logic in `addMethod()`.

Inheritance In several cases, a superclass was used as a template for subclasses and a sibling class was used as a template for other sibling classes.

Conclusions Based on our analysis of C&P dependencies, we conclude that explicitly maintaining C&P dependencies is worthwhile, because these dependencies reflect important design decisions such as crosscutting concerns, feature extensions, paired operations, semantically parallel concerns, and type dependencies (inheritance).

3.4 Maintenance View

We investigated maintenance tasks for duplicated code, because failing to perform such tasks may create defects in software. Although this ethnographic study was not a longitudinal study, we approached the maintenance problems associated with copy and paste by raising questions such as (1) what does a programmer do immediately after C&P? and (2) how does a programmer modify code duplicates created by C&P?

Short term We noticed that cautious programmers modify the portion of pasted code that is specific to the current intended use immediately after they copy and paste. For example, they modify the name of a variable to prevent identifier naming conflicts or remove the portion of the pasted code that is not part of the structural template.

Long term Programmers refactor code after copying and pasting the same code multiple times. For example, after one code snippet is copied and pasted multiple times, the code snippet may be refactored as a separate method. Another example is that after frequently defining an anonymous class and instantiating objects of the class on the fly, a programmer may define an inner class and create a member variable that holds the object. By observing how programmers handle code duplicates, we noted that programmers tend to apply consistent changes to code from the same origin. In other words, after they create structural clones, they modify the structural template embedded in the clones consistently when the

template evolves. This observation is symmetric to the information transparency principle [111] that code elements that change together must look similar.

3.5 Statistics

This section presents statistics about C&P operations in our study. With the instrumented editor, we observed 460 C&P instances. We measured the number of C&P instances per hour as a frequency measure because each session lasted about a few hours. The average number of C&P instances per hour is 16 instances per hour and the median is 12 instances per hour. Figure 3.9 shows how many C&P instances each subject performed per hour.

In order to understand how often C&P operations of different size occurred, we grouped C&P instances into four different syntactic units and counted them (Figure 3.8). About 74% of C&P instances fall into the category of copying text less than a single line such as a variable name, a type name or a method name. Copying in this category saves typing. However, about 25% C&P instances involved copying and pasting a block or a method. Copying in this category often creates structural clones and reflects design decisions in a program. When we multiply this percentage (25%) by the average 16 instances per hour, it means that a programmer produces four interesting C&P dependencies per hour on average. Figure 3.10 shows how many lines of code are copied per instance.

3.6 Threats to Validity

The scope of our study was limited to object oriented programming languages (OOPL). Thus, some results that involve OOPL-specific features may not apply to other programming languages. For example, higher order functions in functional programming languages may remove the need to copy a complicated control structure. Nevertheless, OOPLs are widely used and our study results provide valuable insights for the design of software engineering tools for OOPLs. Participants in our study were researchers at the IBM T.J. Watson Research Center. They were expert programmers and were involved in small team research projects. Our results may not be applicable to larger projects or novice programmers. We conjecture that novice programmers may copy and paste more to learn programming language syntax or employ less of their knowledge about C&P history when they maintain

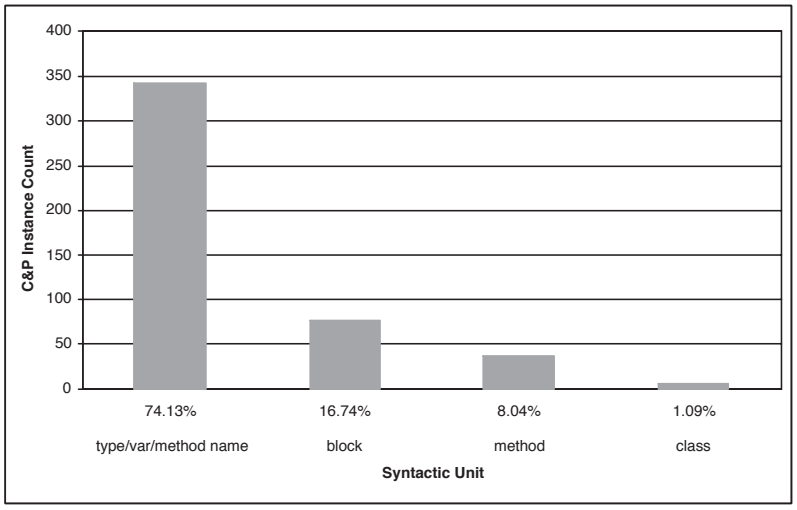

Figure 3.8: Distribution of C&P instances by different syntactic units

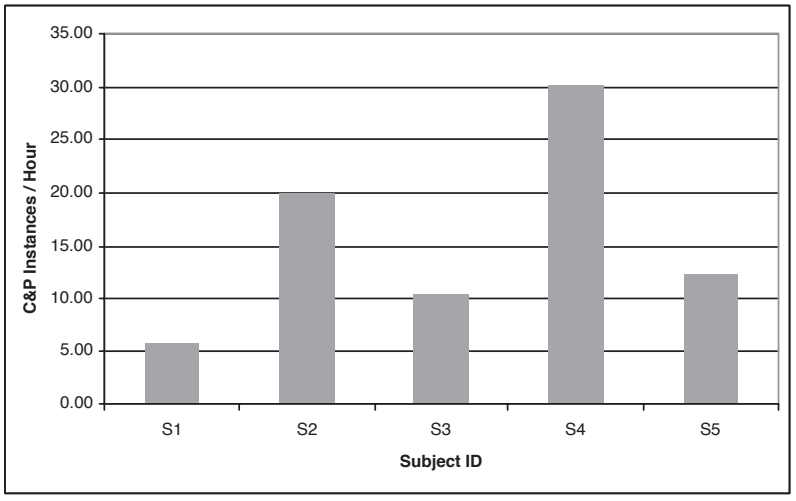

Figure 3.9: C&P frequency per subject

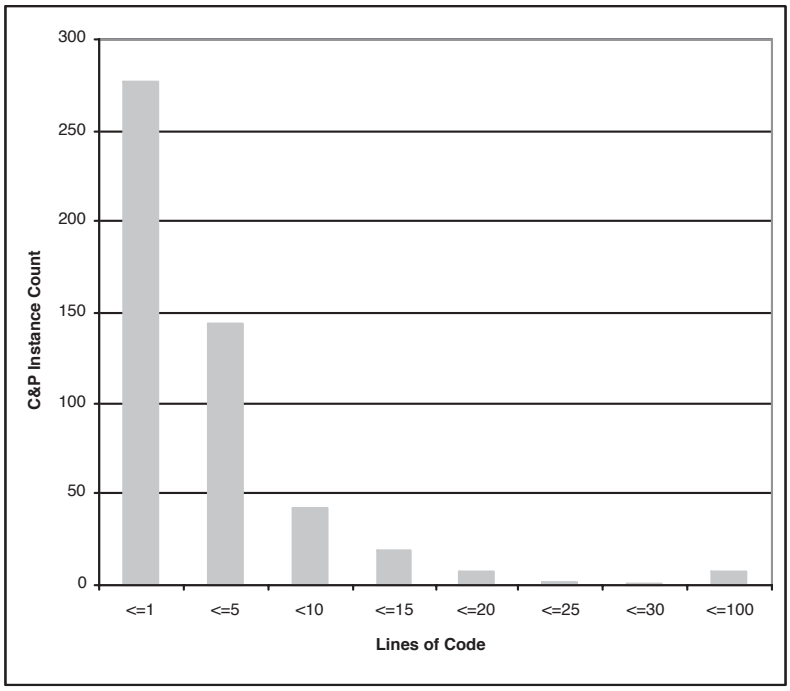

Figure 3.10: Distribution of C&P instances by the number of source lines

software.

For direct observation, the experimenter's presence in the room may have affected the participants' coding behavior because they knew that we were analyzing the intention of each C&P operation. Sometimes the participants did not copy and paste unless they thought they had a valid reason and they seemed to have pressure to write code continuously, which was unnatural for them. In addition, the accuracy of the direct observation may be lower than the logger-based observation as manual logging was extremely difficult.

3.7 Key Insights

- Limitations of particular programming languages produce unavoidable duplicates in a code base. For example, the lack of multiple inheritance in Java induces code duplicates. During the interviews with the subjects, one subject told us that the absence of `enum` construct in Java 1.4 caused him to copy the `public static final String` phrase repetitively.

- Programmers use their memory of C&P history to determine when to restructure code. They deliberately delay code restructuring until they C&P several times, because such reuse helps them discover the right level of abstraction.

- C&P dependencies are worth maintaining explicitly because they reflect design decisions such as aspects, semantically parallel concerns, and paired operations. Programmers often rely on their memory of C&P dependencies when they apply consistent changes to duplicated code.

- Programmers often copy code to reuse structural templates. Thus, it is desirable to learn structural code templates and to support reuse of the learned templates. Additionally, identifying frequently used structural templates will provide input for better programming language design.

These insights served as a basis for proposing software engineering tools that address problems associated with common C&P patterns. The proposed tools are described in

Section 4.7 together with clone maintenance tools that leverage clone genealogy information.

3.8 Conclusions from the Copy and Paste Study

Common wisdom dictates that good programmers do not use C&P operations because it tends to produce maintenance problems. Our ethnographic study has shown that programmers nevertheless use C&P very frequently, producing up to four architecturally significant C&P instances per hour. Rather than viewing this as a drawback, we instead take this as an opportunity to identify and develop software engineering tool support for existing practices. We discovered that C&P information is useful for program understanding and that programmers actively make use of C&P history to decide when to restructure code. We cataloged common C&P patterns and maintenance problems associated with them.

Chapter 4

AN EMPIRICAL STUDY OF CODE CLONE GENEALOGIES

The previous chapter's C&P study suggests that the practice of creating and managing clones is not necessarily bad. To check whether code clones indeed pose challenges during software evolution, we need to answer the following questions: "How often do code clones require consistent updates during software evolution?" "How often do programmers create clones by copying existing code?" "Can refactoring indeed improve software quality with respect to clones?" and "How long do code clones stay in the system before they get removed or refactored?"

While there have been a number of studies on clone evolution [6, 103, 187], these studies measured only the changes in clone coverage (the ratio of code clones to the total size of a program). This type of quantitative analysis does not answer the questions above. For example, when clone coverage increased, it could be due to copying existing code or introducing a group of completely new code fragments that are similar to one another. When clone coverage decreased, it could be due to removing clones through refactoring or updating clones inconsistently.

To study clone evolution structurally and semantically, we defined a formal model of clone evolution that tracks individual clones and their changes over multiple program versions. The core of this model is a clone genealogy representation that describes how each member in a group of clones has changed with respect to other members in the same group. Then based on the model, we developed a tool that automatically extracts clone genealogies from a sequence of program versions. Using this tool, we studied clone evolution in two Java open source projects, *carol* and *dnsjava*. In particular, we analyzed (1) how often clones were updated consistently, (2) how long they stayed in the system, and (3) to what extent refactoring removed clones.

The rest of this chapter is organized as follows. Section 4.1 formally defines the model

of clone evolution, which serves as the basis of the clone genealogy extractor described in Section 4.2. Section 4.3 describes the study procedure, and Section 4.4 presents an analysis of clone evolution patterns. Section 4.5 discusses the study limitations, and Section 4.7 proposes clone maintenance tools based on the clone genealogy study results as well as the C&P study in Chapter 3.

4.1 Model of Clone Genealogy

A clone genealogy describes how groups of code clones change over a sequence of program versions. In a clone's genealogy, a group to which the clone belongs is traced to its origin clone group in the previous version. The model associates related clone groups that have originated from the same ancestor clone group, and represents how each element in a group of clones has changed with respect to other elements in the same group.

The basic unit in our model is a Code Snippet, which has two attributes, Text and Location. Text is an internal representation of code that a clone detector uses to compare code snippets. For example, when using CCFinder [149], Text is a parametrized token sequence, whereas when using *CloneDr* [26] Text is an isomorphic AST. A Location is an identifier for matching code across versions. Every code snippet in a particular version of a program has a unique location. To determine how much the text of a code snippet has changed across versions, a TextSimilarity function measures the similarity between two texts $t1$ and $t2$ ($0 \leq$ TextSimilarity$(t1, t2) \leq 1$). To trace a code snippet across versions, a LocationOverlapping function measures how much two locations $l1$ and $l2$ overlap each other ($0 \leq$ LocationOverlapping$(l1, l2) \leq 1$).

A Clone Group is a set of code snippets with identical Text. In other words, a Clone Group refers to a group of code snippets that are considered equivalent by a clone detector. $CG.text$ is syntactic sugar for the text of any code snippet in a clone group CG. A Cloning Relationship is defined between two clone groups CG_1 and CG_2 if and only if TextSimilarity$(CG_1.text, CG_2.text) \geq sim_{th}$, where sim_{th} is a constant between 0 and 1. An Evolution Pattern is defined between an old clone group OG in the $k-1^{th}$ version and a new clone group NG in the k^{th} version such that there exists a cloning relationship between NG and OG.

We initially defined five types of evolution patterns based on our insights from the C&P study in Chapter 3. We defined the *Add* pattern to describe introduction of a new clone by copying existing code in the old version; the *Subtract* pattern to describe clone removal through refactoring or deleting code; the *Consistent Change* pattern to describe application of similar edits to clones; the *Inconsistent Change* pattern to describe not updating some clones while updating others; and the *Same* pattern to model no changes to a clone group.

In our model, different kinds of evolution patterns may overlap. For example, combination of the *Consistent Change* pattern and the *Add* pattern represents applying a similar edit to clones and also copying one of the clones. To clarify the relationship among the five evolution patterns above and to check whether they can describe *all* possible changes to a clone group, we wrote our model in the Alloy modeling language [142]. The relationship among evolution patterns is described in the Venn diagram in Figure 4.1. In our initial attempt, we discovered an additional pattern, the *Shift* pattern, which is a somewhat unintuitive but necessary pattern to cover all possible changes to a clone group.[1] The six evolution patterns are described below in Alloy syntax and the entire model is available in Appendix D.

- *Same*: all code snippets in NG did not change from OG.
  TextSimilarity(NG.text,OG.text) = 1
  all cn:CodeSnippet | some co:CodeSnippet | cn in NG ⇒ co in OG && LocationOverlapping(cn,co) = 1
  all co:CodeSnippet | some cn:CodeSnippet | co in OG ⇒ cn in NG && LocationOverlapping(cn,co) = 1

- *Add* : at least one code snippet in NG is newly added. For example, programmers added a new code snippet to NG by copying an old code snippet in OG.
  TextSimilarity(NG.text,OG.text) $\geq sim_{th}$
  some cn:CodeSnippet | all co:CodeSnippet | co in OG ⇒ cn in NG && LocationOverlapping(cn,co) = 0

- *Subtract* : at least one code snippet in OG does not appear in NG. For example, programmers refactored or removed a code clone.

[1] Checking the ALL_EXHAUSTIVE assert statement in Appendix D without the *Shift* pattern generated a counterexample.

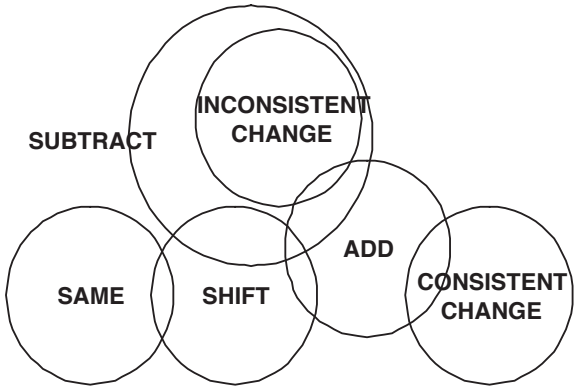

Figure 4.1: The relationship among evolution patterns

TextSimilarity(NG.text,OG.text) $\geq sim_{th}$

some co:CodeSnippet | all cn:CodeSnippet | cn in NG $\Rightarrow$ co in OG && LocationOverlapping(cn,co) $= 0$

- *Consistent Change*: all code snippets in OG have changed consistently; thus they belong to NG together. For example, programmers applied the same change consistently to all code clones in OG.
  $sim_{th} \leq$TextSimilarity(NG.text,OG.text)< 1
  all co:CodeSnippet | some cn:CodeSnippet | co in OG $\Rightarrow$ cn in NG && LocationOverlapping(cn,co) > 0

- *Inconsistent Change*: at least one code snippet in OG changed inconsistently; thus it does not belong to NG anymore. For example, a programmer forgot to change one code snippet in OG.
  $sim_{th} \leq$TextSimilarity(NG.text,OG.text)< 1
  some co:CodeSnippet | all cn:CodeSnippet | cn in NG $\Rightarrow$ co in OG && LocationOverlapping(cn,co) $= 0$

- *Shift*: at least one code snippet in NG partially overlaps with at least one code snippet in OG.
  TextSimilarity(NG.text,OG.text) $= 1$
  some cn:CodeSnippet | some co:CodeSnippet | cn in NG && co in OG && (1 >LocationOverlapping(cn,co) > 0)

A Clone Lineage is a directed acyclic graph that describes the evolution history of a sink node (clone group). A clone group in the k^{th} version is connected to a clone group in the

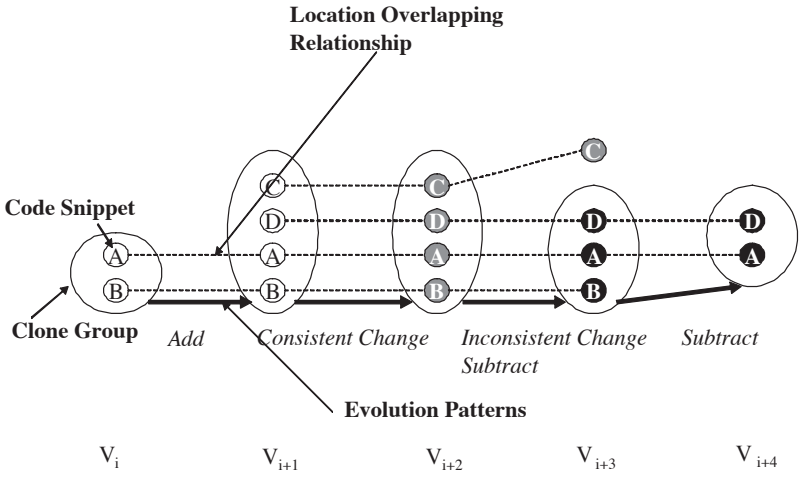

Figure 4.2: An example clone lineage

$k-1^{th}$ version by an evolution pattern. For example, Figure 4.2 shows a clone lineage including the *Add*, *Consistent Change*, *Inconsistent Change*, and *Subtract* patterns. In the figure, code snippets with the same text are filled with the same shade.

A Clone Genealogy is a set of clone lineages that have originated from the same clone group. A clone genealogy is a connected component where every clone group is connected by at least one evolution pattern.[2] A clone genealogy approximates how programmers create, propagate, and evolve code clones. Figure 4.3 shows an example clone genealogy that comprises two clone lineages. Ovals in the figure represent clone groups. Appendix E describes an example genealogy stored in an XML format.

4.2 Clone Genealogy Extractor

Based on the model in Section 4.1, we built a tool that automatically extracts clone genealogies over a project's lifetime. Our clone genealogy extractor (CGE) requires three inputs: (1) multiple versions of a program in chronological order, $\{V_k \mid 1 \leq k \leq n\}$, (2) a clone

[2] A clone genealogy is a connected component in the sense that there exists an undirected path for every pair of clone groups. Although a clone genealogy is often an inverted tree in practice, it is a connected component in theory because the in-degree of a new clone group can be greater than one when it is ambiguous to determine the most likely origin of a new clone group.

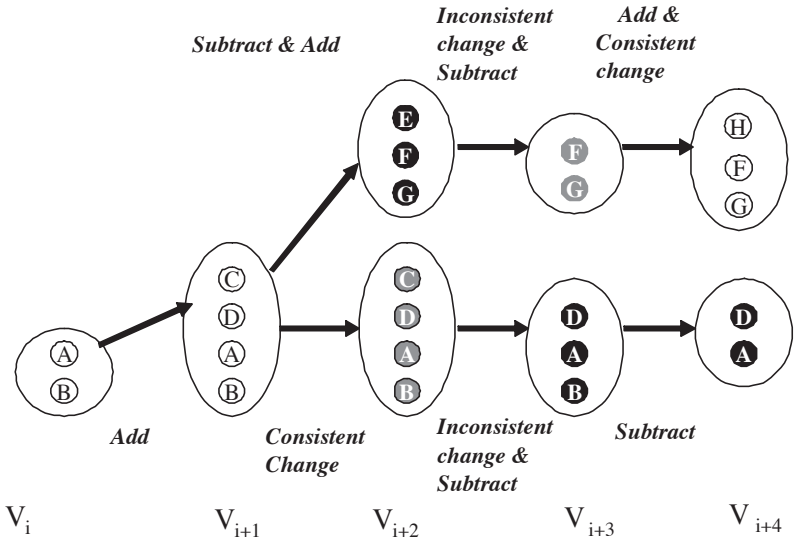

Figure 4.3: An example clone genealogy

detector, and (3) a location tracker that traces a code snippet's location across versions.[3]

To assist a user of CGE to prepare multiple versions of a program, CGE automatically extracts check-in level program snapshots from a source code repository (CVS).[4] Because CVS records individual file revisions but not which files were changed together, CGE uses *Kenyon*'s [32] front-end to identify CVS check-in transactions and to check out the source code that corresponds to each check-in. Depending on the granularity of evolution analysis, a user can select a subset of versions. For example, a user can select all versions corresponding to all check-ins or only the versions that increased (or decreased) the total number of lines of code clones (LOCC).

CGE identifies clone groups in each version V_k using a clone detector. Currently we use CCFinder [149] described in Section 2.4, but any clone detector can be used. Using a clone detector, CGE implements the TextSimilarity function. CGE currently identifies the common part between two texts $t1$ and $t2$ using CCFinder and calculates the common part's

[3]A location tracker can be considered as a code matching technique (see Section 2.2).

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

Table 4.1: Line number mappings generated using *diff*

A.txt	B.txt	*diff*'s meta data	Line mappings
1: a	1: a		1↦ 1
2: b	2: d	`2c2`	2↦ 2
3: c	3: c		3↦ 3
4: f	4: e	`3a4,5`	3↦ 4
	5: e		3↦ 5
	6: f		4↦ 6

relative proportion to the size of $t1$ and $t2$.

$$TextSimilarity(t1, t2) = \frac{2|t1 \bigcap t2|}{|t1| + |t2|} \tag{4.1}$$

where $|t|$ is the size of text t and $t1 \bigcap t2$ is the common part of $t1$ and $t2$. Using a different type of clone detector may require reimplementing the TextSimilarity function.

CGE uses a location tracker to implement the LocationOverlapping function, which computes an overlapping score between a location L_i in V_{k-1} and a location L_j in V_k. Currently we use a file and line based location tracker based on *diff*. The tracker first finds a corresponding file in the next (or previous) version using the same hierarchical name (e.g., `/org/xbill/DNS/CertRecord.java`). It runs *diff* on the mapped files, parses the meta information in *diff*'s output, and creates line number mappings. For example, by comparing `A.txt` with `B.txt` in Table 4.1, *diff* generates the meta information in the third column, which is used to create line mappings in the fourth column. By converting the line numbers of L_j to old line numbers in the same file f in V_{k-1}, the tracker computes the LocationOverlapping score—a relative proportion of an overlapped region between L_i and the calibrated region L_j.

$$LocationOverlapping(L_i, L_j) = \frac{min(n_e, o_e) - max(n_s, o_s)}{n_e - n_s} \tag{4.2}$$

where L_i spans from the line o_s to the line o_e, and the calibrated location of L_j in V_{k-1} spans from the line n_s to the line n_e. Using a different type of location tracker or a code matching technique may require reimplementing the LocationOverlapping function. Please

refer to Section 2.2.1 for other line matching techniques.

Using the same clone detector, CGE finds Cloning Relationships between the clone groups in V_{k-1} and the clone groups in V_k for $1 < k \leq n$. For each clone group in V_k, a clone detector may find several cloning relationships to clone groups in V_{k-1}. CGE applies the following heuristic to remove less interesting cloning relationships; For each clone group in V_k, CGE selects both a cloning relationship with the best LocationOverlapping score and a cloning relationship with the best TextSimilarity score. Usually, these are the same cloning relationship.

After applying the heuristic, CGE separates each connected component of cloning relationships (i.e., a clone genealogy) and then labels evolution patterns in it. CGE visualizes a genealogy graph using the *Graphviz* package [80] and allows a user to browse code relevant to a selected genealogy.

4.3 Study Procedure

Our CGE captures various kinds of clone evolution patterns and thus allows us to explore a wide variety of research questions about clone evolution. In this study, we focused on the following questions: (1) how often do programmers update clones consistently? (2) how long do clones live in the system? and (3) what are evolutionary characteristics of clones that cannot be easily removed with refactoring techniques?

To determine these characteristics, we chose two subject programs with a significant evolutionary history. We extracted clone genealogies from those programs and then computed the age of the genealogies and the kinds of evolution patterns they include.

Because our C&P study in Chapter 3 primarily focused on Java programs, we focused on subject programs written in Java. *Carol* and *dnsjava* met this condition and both had a version history for over two years. In addition, their code size allowed us to manually inspect genealogies if necessary. *Carol* is a library that allows clients to use different RMI (remote method invocation) implementations and has evolved over 26 months from August 2002 to October 2004. *Dnsjava* is an implementation of DNS (domain name system) in

Java that has evolved over 74 months from September 1998 to November 2004.[5] Table 4.2 describes the programs' size in lines of code (LOC), the period that we studied, and the number of CVS check-ins during the period.[6] We studied the history of *dnsjava* from March 1999 (the first release) because many directories were duplicated for file back up and they were not cleaned up until the first release.

For our analysis, we focused on versions of the programs in which the LOCC (the total number of Lines Of Code Clones) increased or decreased from the preceding version; this identifies the set of program versions that added or deleted code clones or made changes to code clones. For our target programs, this resulted in studying 37 versions out of 164 versions of *carol* and 224 versions out of 905 versions of *dnsjava*.

CCFinder can be tuned using a number of input parameters. We used the default settings, most noticeably the minimum token length default of 30 tokens. Setting the minimum at 30 tokens resulted in an average clone size of seven lines in *carol* and *dnsjava*. With this setting, CCFinder found an average clone coverage ratio of 10.6% in *carol* and 10.5% in *dnsjava*.

We set the threshold sim_{th} of TextSimilarity function to be 0.3 because we empirically found that 0.3 neither underestimates nor overestimates the size and the length of genealogies. We discuss how sim_{th} affects our results in detail in Section 4.5.

CCFinder occasionally detects false positive clones that are similar in a token sequence, although common sense says that they are not clones. We used our previously defined concept of a *syntactic template* to identify which clones are false positives. The idea of a syntactic template comes from our C&P study in Chapter 3. A syntactic template is a template of repeated code in a series of syntactically similar code fragments. For example, a set of field declarations often appear in a row in a class, a series of method invocation statements appear together in a static initializer, or case statements appear in a row in a switch-case block. We manually removed 13 out of 122 genealogies in *carol* and 15 out of 140 genealogies in *dnsjava* because they consist of only syntactic templates (see Table

[5] This study was conducted in early 2005.

[6] A *check-in* in our analysis corresponds to a single logical CVS transaction that commits a set of revisions together within a time window of 3 minutes [313].

Table 4.2: Description of two Java subject programs for clone genealogy study

Program	*carol*	*dnsjava*
URL	carol.objectweb.org	www.dnsjava.org
LOC	7878 ∼ 23731	5756 ∼ 21188
duration	26 months	68 months
# of check-ins	164	905

4.4). Example false positive clones that consist of syntactic templates are shown in Table 4.3. Although there could be false negative clones that CCFinder cannot find, we do not think that there are many because a previous comparison of clone detectors [44] suggests that CCFinder has a much higher recall than *CloneDr* (AST-based) [26] or *Covet* (metric-based) [202], although its precision is lower than the *CloneDr*.

4.4 Study Results

This section presents the evolution patterns of code clones in *carol* and *dnsjava* and answers the questions raised in Section 4.3.

4.4.1 Consistently Changing Clones

To determine how often code clones change consistently with other clones in the same clone group, we measured the number of genealogies with a consistent change pattern. Throughout this chapter, we use a genealogy instead of a lineage as our measurement unit for two reasons. (1) Lineages in the same genealogy stem from the same clone group, thus inheriting the same evolution patterns. (2) A clone group's location in one lineage may overlap with that of other lineages in the same genealogy.

We say that a clone genealogy includes a consistent change pattern if and only if all lineages in the clone genealogy include at least one *consistent change* pattern. Out of 109 genealogies in *carol*, 41 genealogies (38%) include a consistent change pattern. Out of 125 genealogies in *dnsjava*, 45 genealogies (36%) include a consistent change pattern. Fewer than the half of the clones undergo consistent updates during evolution.

Table 4.3: Example of false positive clones. Clones are marked in blue.

```
/**
 * Converts rdata to a String
 */
public String
rdataToString() {
  StringBuffer sb = new StringBuffer();
  sb.append(footprint & 0xFFFF);
    sb.append(" ");
    sb94.append(alg & 0xFF);
    sb.append(" ");
    sb.append(digestid & 0xFF);
    if (digest != null) {
        sb.append(" ");
        sb.append(base16.toString(digest));
    }
     return sb.toString(); }
```

```
/**
 * Converts rdata to a String
 */
public String
rdataToString() {
  StringBuffer sb = new StringBuffer();
  if (key != null || (flags & (FLAG_NOKEY)) == (FLAG_NOKEY) ) {
    if (!Options.check("nohex")) {
      sb.append("0x");
      sb.append(Integer.toHexString(flags & 0xFFFF));
    }
    else
      sb.append(flags & 0xFFFF);
      sb.append(" ");
      sb.append(proto & 0xFF);
      sb.append(" ");
      sb.append(alg & 0xFF);
      if (key != null) {
        sb.append(" (n");
        sb.append(base64.formatString(key, 64, "t", true));
        sb.append(" ; key_tag = ");
        sb.append(getFootprint() & 0xFFFF);
      }
   }
   return sb.toString(); }
```

Table 4.4: Clone genealogies in *carol* and *dnsjava*(min_{token} =30, $sim_{th} = 0.3$)

# of genealogies	carol	dnsjava
total	122	140
false positive	13	15
true positive	109	125

4.4.2 Volatile Clones

To understand how long clones survive in the systems, we measured the age of a clone genealogy—how many versions a genealogy spans. In our analysis, we classified genealogies in two groups, *dead* genealogies that do not include clone groups of the final version and *alive* genealogies that include clone groups of the final version. We differentiate a dead genealogy from an alive genealogy because only dead genealogies provide information about how long clones stayed in the system before they disappeared. On the other hand, for an alive genealogy, we cannot tell how long its clones will survive because they are still evolving. At the end point of our analysis, in *carol*, out of 109 clone genealogies, 53 of them are dead and 56 of them are alive. In *dnsjava*, out of 125 clone genealogies, 107 of them are dead and 18 of them are alive.

To reason about how long genealogies survived in terms of absolute time as well as in the number of versions used in our analysis, we define k-volatile genealogies (clone genealogies that disappeared within k versions) and measure the average lifetime of k-volatile genealogies, i.e., k-volatile genealogies = $\{g|g$ is a dead genealogy and $0 \leq g.age \leq k\}$. Figure 4.4 shows the average lifetime of k-volatile genealogies in the number of check-ins (left axis) and the number of days (right axis). Let $f(k)$ be the number of genealogies with the age k and $f_{dead}(k)$ be the number of dead genealogies with the age k. $CDF_{dead}(k)$ is a cumulative distribution function of $f_{dead}(k)$ and it means the ratio of k-volatile genealogies among all dead genealogies. Let $R_{volatile}(k)$ be the ratio of k-volatile genealogies among all genealogies in the system.

$$CDF_{dead}(k) = \frac{\sum_{i=0}^{k} f_{dead}(i)}{\sum_{i=0}^{n} f_{dead}(i)} \tag{4.3}$$

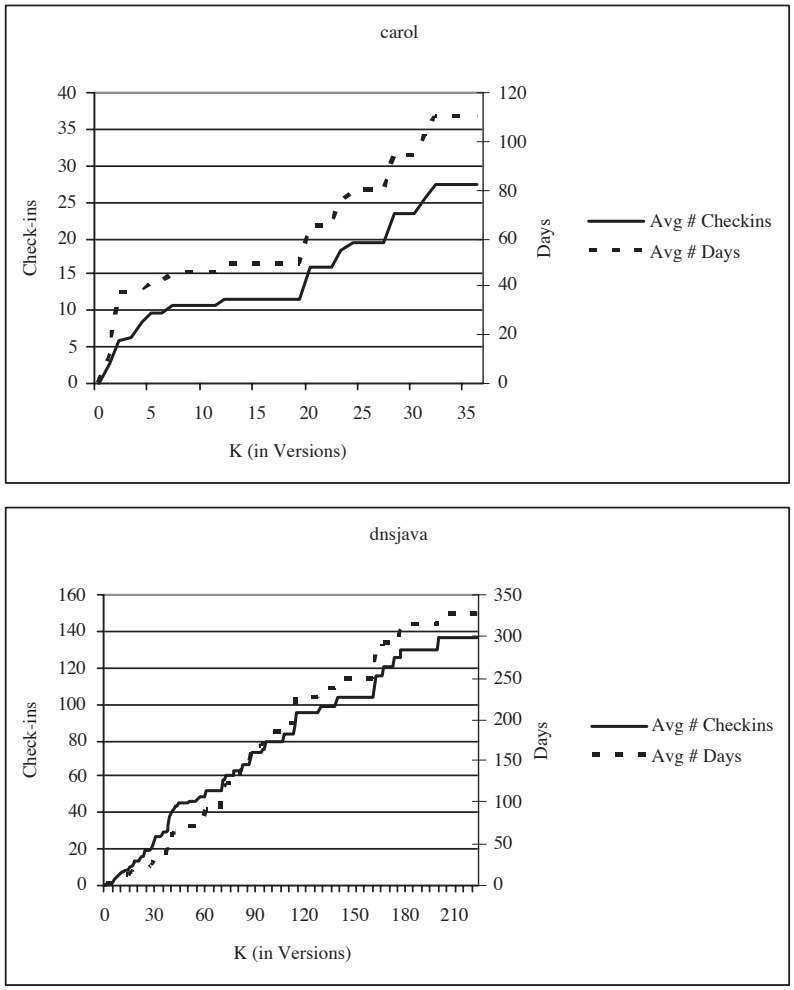

Figure 4.4: The average lifetime of k-volatile clone genealogies

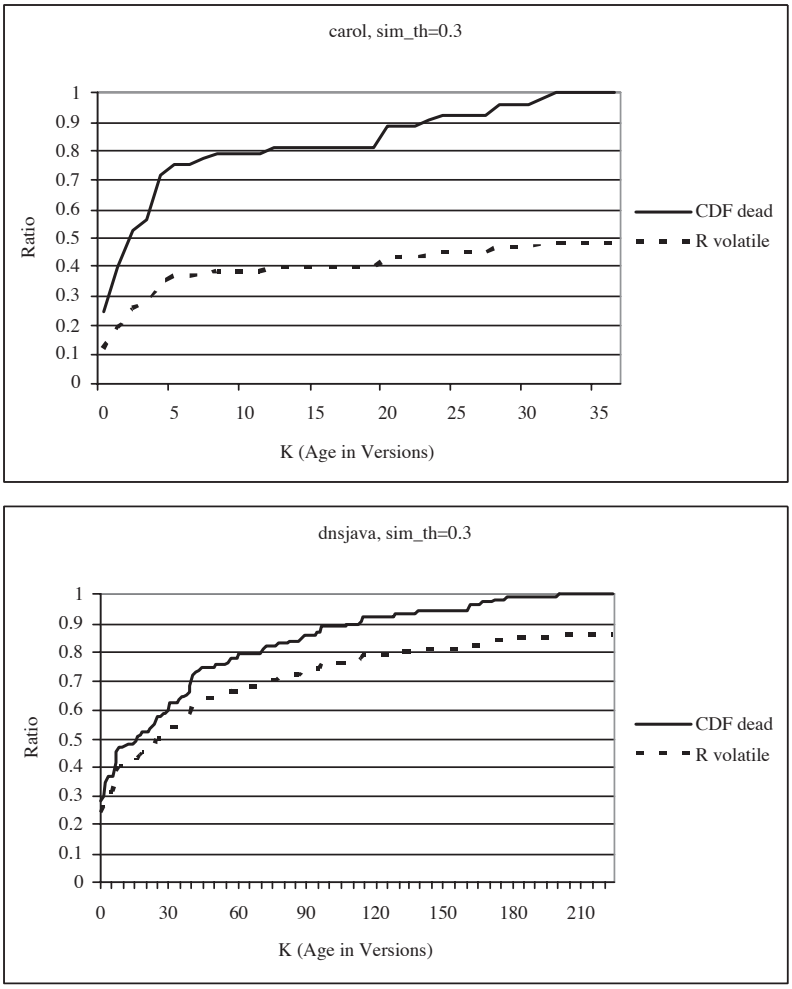

Figure 4.5: $CDF_{dead}(k)$ and $R_{volatile}(k)$ of *carol* and *dnsjava*

$$R_{volatile}(k) = \frac{\sum_{i=0}^{k} f_{dead}(i)}{\sum_{i=0}^{n} f(i)} \tag{4.4}$$

Figure 4.5 presents $CDF_{dead}(k)$ and $R_{volatile}(k)$ for *carol* and *dnsjava*. In *carol*, 37% of all genealogies (75% of dead genealogies) disappeared within 5 versions, and 39% of all genealogies (79% of dead genealogies) disappeared within 10 versions. When we interpret these data in the number of check-ins or in the number of days by referring to Figure 4.4, they mean that 37% of all genealogies lasted an average of 9.6 check-ins and 41.7 days and 39% lasted an average of 10.8 check-ins and 45.6 days during the evolution period of 164 check-ins and 800 days in *carol*.

In *dnsjava*, 31% of all genealogies (36% of dead genealogies) disappeared within 5 versions, and 41% of all genealogies (48% of dead genealogies) disappeared within 10 versions. These data mean that 31% of all genealogies lasted an average of 1.48 check-ins and 1.48 days and 41% lasted an average of 7.35 check-ins and 11.05 days during the evolution period of 905 check-ins and 2051 days in *dnsjava*. So in both systems, a large number of clones were *volatile*. The large extent of volatile clones suggests that a substantial amount of the work done by a developer applying a strategy of aggressive, immediate refactoring may not be cost-effective.

To investigate why code clones disappear during evolution, we built a user interface that allows tracking code of interest. The same location tracking technique in Section 4.2 was used to map corresponding source lines between consecutive versions. By comparing clones at the time of disappearance (lineage death) and the corresponding source lines in the next version, we categorized the reasons why clone lineages disappear into three categories: *Divergent Change* means that the clones in the same clone group changed differently enough that they were no longer considered as clones. *Removal* means that the clones were deleted by merging the commonality between the clones (refactoring) or by removing the code that contains them (e.g., removing files that included the clones). *Cut-off* means that the length of each clone in the clone group became shorter than 30 tokens so that they were not found by CCFinder or the TextSimilarity between clone groups of consecutive versions is less than the chosen sim_{th} threshold. Table 4.5 shows the percentage for each category.

We found that 26% (*carol*) to 34% (*dnsjava*) of clone lineages were discontinued because

Table 4.5: How do lineages disappear?

Reasons	*carol*	*dnsjava*
Divergent Change	26%	34%
Removal	67%	45%
Cut-off	7%	21%

of divergent changes in the clone group. Refactoring of such volatile clones may not be necessary and can be counterproductive if a programmer sometimes has to undo refactoring.[7]

4.4.3 Locally Unfactorable Clones

We define that a clone group is *locally refactorable* if a programmer can remove duplication with standard refactoring techniques, such as *pull up a method*, *extract a method*, *remove a method*, *replace conditional with polymorphism*, etc. [92]. On the other hand, (1) if a programmer cannot use standard refactoring techniques to remove clones, (2) if a programmer must deal with cascading non-local changes in the design to remove duplication (for example, modifications to public interfaces), or (3) if a programmer cannot remove duplication due to programming language limitations, we consider the clone group as locally unfactorable.[8]

Table 4.6 presents a code example of a locally unfactorable clone group found in *carol*. In this example, `exportObject` and `unexportObject` are paired operations that have identical control logic (if-else logic, iterator logic, and exception handling logic) but throw different types of exceptions, pass different messages to the tracing module, and invoke different methods. It is difficult to remove this duplication because Java 1.4 does not provide a unit of abstraction that encapsulates similar logic involving different types or different method invocations inside the logic. Although it is possible to remove this duplication by using

[7]This observation is consistent with the general notion that delaying some design decisions in software development may at times add value [278].

[8]Chapter 3 describes a taxonomy of locally unfactorable clones that are often created by copy and paste in Java. Basit et al. also summarize the characteristics of locally unfactorable clones that are difficult to remove using abstractions in C++ [23].

Table 4.6: Example of locally unfactorable clones

```
public void exportObject(Remote obj)
throws RemoteException{
  if (TraceCarol.isDebugRmiCarol()) {
    TraceCarol.debugRmiCarol(
      "MultiPRODelegate.exportObject(" ... .
  }
  try {
    if (init) {
     for (Enumeration e = activePtcls.elements(); ...
      ((ObjDlgt)e.nextElement()).exportObject(obj);
      }
    } else {
     initProtocols();
     //iterate protocol elements and export obj
     }
    }
  }catch (Exception e) {
    String msg = "exportObject(Remote obj) fail";
    TraceCarol.error(msg,e);
    throw new RemoteException(msg);
  }
}
```

```
public void unexportObject(Remote obj)
throws NoSuchObjectException {
  if (TraceCarol.isDebugRmiCarol()) {
    TraceCarol.debugRmiCarol(
      "MultiPRODelegate.unexportObject(" ... .
  }
  try {
    if (init) {
     for (Enumeration e = activePtcls.elements(); ...
      ((ObjDlgt)e.nextElement()).unexportObject(obj);
      }
    } else {
     initProtocols();
     //iterate protocol elements and unexport obj
     }
    }
  } catch (Exception e) {
    String msg = "unexportObject(Remote obj) fail";
    TraceCarol.error(msg,e);
    throw new NoSuchObjectException(msg);
  }
}
```

a generic type in Java 5.0, by transforming the code to use the strategy design pattern (p. 315, [94]) and changing public interfaces, or by changing the external library's APIs (e.g., `exportObject` and `unexportObject`), these transformations either incur non-local changes or require using language constructs that were not available in Java 1.4.[9] This limitation is not specific to Java only. There is no programming language that provides all possible levels of abstraction.

A locally unfactorable genealogy means that a programmer cannot discontinue any of its clone lineages by local refactoring. In other words, a clone genealogy is locally unfactorable if and only if all of its clone lineages end with a clone group that is locally unfactorable.

In the two subject programs, we inspected all clone lineages to find those that are locally unfactorable. 70 genealogies (64%) in *carol* and 61 genealogies (49%) in *dnsjava* are locally unfactorable.

4.4.4 Long-Lived Clones

Programmers would get a good return on their refactoring investment if clones live for a long time and if they tend to change with other clones. But our data show that even when refactoring looks attractive, it may not be feasible given the significant number of clones that are locally unfactorable.

Out of 37 genealogies that lasted more than half of *carol*'s software history (18 versions out of 37 versions), 29 of them include consistent change patterns, 24 of them comprise locally unfactorable clones, and 19 of them include both consistent change patterns and locally unfactorable clones. Out of 18 genealogies that lasted more than half of *dnsjava*'s software history (112 versions out of 224 versions), 15 of them include consistent change patterns, 13 of them comprise locally unfactorable clones, and 11 of them include both consistent change patterns and locally unfactorable clones.

Figure 4.6 shows the cumulative fraction of (1) consistently changed genealogies, i.e., $\frac{\sum_{i=0}^{k} f_{consistent}(i)}{\sum_{i=0}^{k} f(i)}$, (2) locally unfactorable genealogies, and (3) locally unfactorable genealogies that include consistent change patterns. As k increases (meaning that as the genealogies

[9]Note that Java 5.0 became available in September 2004 and this code was written prior to that.

get older), the more genealogies include a consistently changing pattern and comprise locally unfactorable clones. This result suggests that the current programming languages and conventional refactoring techniques cannot improve many troublesome clones—those that are long-lived and consistently changing.

4.5 Discussion

This section discusses how the text similarity threshold affects our analysis results and describes limitations of our study.

4.5.1 Impact of Similarity Threshold

The text similarity threshold (sim_{th}) sets the bar for defining a cloning relationship. A low sim_{th} can find a consistent change pattern between OG and NG while a high sim_{th} will consider that OG's lineage is discontinued. Sim_{th} affects the size and length (age) of clone genealogies and the number of consistent change patterns.

Table 4.7 shows that, when 0.1 is used, CGE finds fewer genealogies with a larger size and a longer length because it finds more cloning relationships and thus combines several genealogies to one. When 0.5 is used, CGE finds more genealogies with a smaller size and a shorter length because it divides a long clone genealogy into many short and small genealogies. When sim_{th} is 0.1, the ratio of consistently changed genealogies is 2% higher in *dnsjava* and 26% higher in *carol* than using 0.3. Figure 4.7 shows $CDF(k)$ in *carol* and *dnsjava* when sim_{th} is 0.1, 0.3, and 0.5. $CDF(k)$ of 0.1 shows a coarse-grained distribution because sim_{th} 0.1 reduces the total number of genealogies. Figure 4.7 shows that our choice of sim_{th} 0.3 generates a finer-grained distribution than using 0.1 and estimates the number of volatile genealogies more conservatively than using 0.5.

4.5.2 Study Limitations

Clone Detection Technique CGE incorrectly counts the number of consistent change patterns in some cases because CCFinder detects only a contiguous token string as a clone.[10]

[10]Gemini, a clone analysis and visualization tool, identifies gapped clones (i.e., non-contiguous code clones) by post-processing CCFinder's output [283].

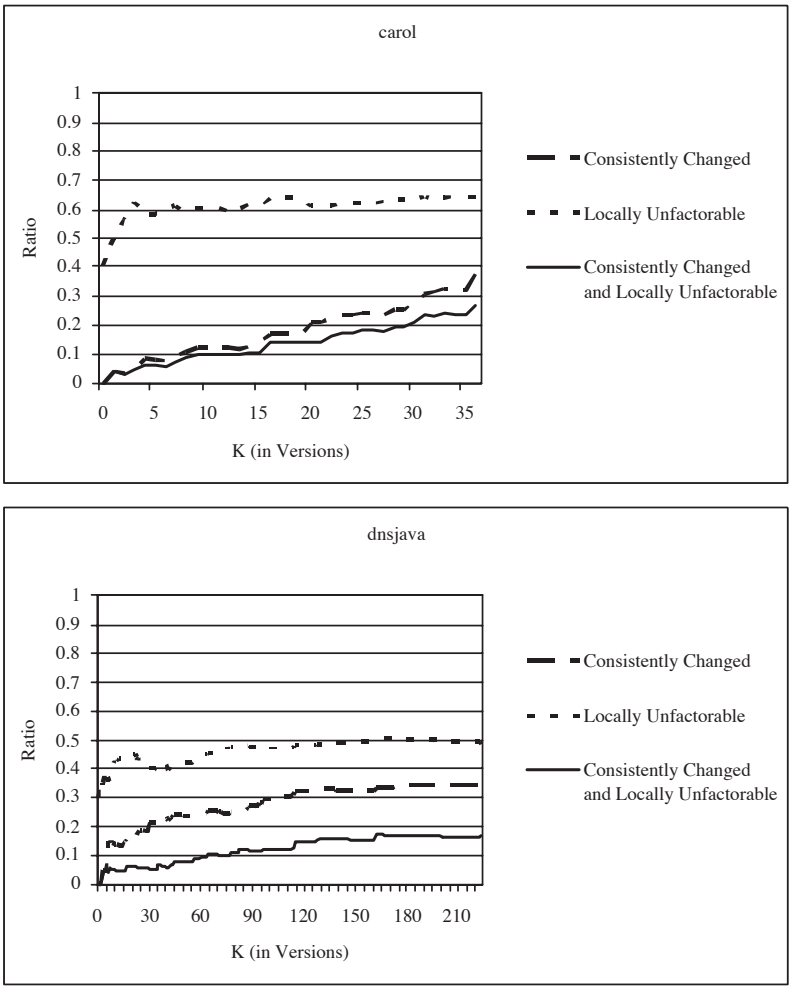

Figure 4.6: Cumulative fraction of consistently changed genealogies, locally unfactorable genealogies, and consistently changed and locally unfactorable genealogies

Table 4.7: Average size and length of genealogies with varying sim_{th}

		sim_{th}		
		0.1	0.3	0.5
# of genealogies	*carol*	27	122	153
including false positives	*dnsjava*	63	140	180
# of consistently	*carol*	16	41	53
changed genealogies	*dnsjava*	21	45	52
avg size	*carol*	117.52	26.01	20.74
(in # of clone groups)	*dnsjava*	233.73	105.17	81.81
avg age	*carol*	25.19	12.57	12.56
(length)	*dnsjava*	63.49	46.16	40.93

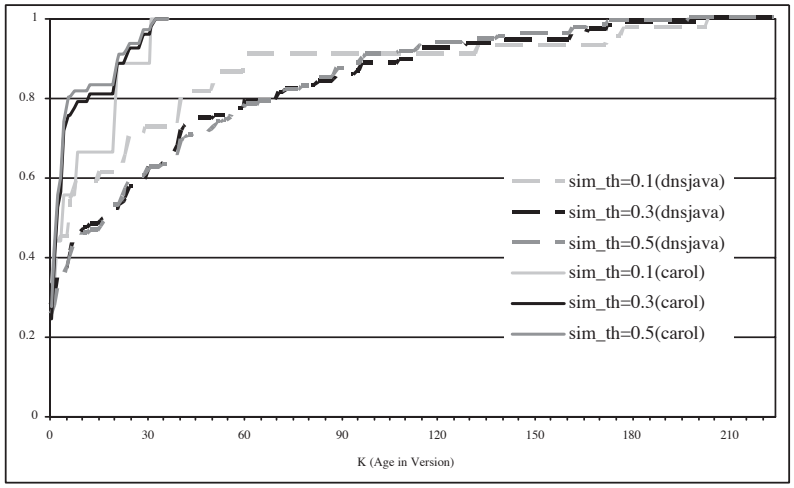

Figure 4.7: Cumulative distribution function of dead genealogies with varying sim_{th}

For example, when code is inserted in the middle of one clone in a clone group, the existing clone group is broken into two new clone groups with shorter contiguous text, causing identification of two consistent patterns rather than one inconsistent change pattern. As another example, even if a programmer consistently modified OG to create NG, CCFinder does not find a cloning relationship between OG and NG if they do not share a contiguous token string greater than the size of $sim_{th}(|OG.text| + |NG.text|)/2$. The absence of a cloning relationship can be mistakenly interpreted as a discontinuation of a lineage. This limitation can be overcome by plugging in clone detectors that find non-contiguous code clones, such as *CP-Miner* [187], PDG-based detectors [176, 173], and metric-based detectors [148, 202, 212]. We speculate that using these clone detectors will not change our key findings as other follow-up studies that use different clone detectors support the same key findings (see Section 4.6).

Location Tracking Technique We implemented a file and line based location tracker based on the *diff* algorithm; thus our location tracking algorithm is limited in two ways. First, it depends on *diff* to resolve ambiguity in finding a corresponding line. For example, when a file A contains abc in the $k-1^{th}$ version and contains cba in the k^{th} version, *diff* considers that ab is deleted before c and ba is inserted after c, even if a programmer replaced a to c and c to a. Second, our algorithm considers that two files are related only when their hierarchical file names match. For example, when a file A is renamed to B or A is split into two files $B1$ and $B2$, the evolution patterns between A and B or A and $B1(B2)$ would be identified as the *Add* and *Subtract* patterns. We speculate that code matching techniques and refactoring reconstruction techniques in Section 2.2 can improve our location tracker by inferring how classes were renamed, split, or merged.

Subject Programs Our two subject programs both comprise about 20,000 lines of code. The clone coverage ratio of these programs was smaller than many programs reported in the literature. We speculate that *carol* and *dnsjava* may have fewer locally unfactorable and consistently changing clones than larger programs whose duplication is difficult to remove without compromising many existing design decisions. Both *carol* and *dnsjava* have been

maintained by a small number of people: two developers for *dnsjava* and six developers for *carol*. The small team size may have affected our study results.

The granularity of our analysis was a check-in that changed the total number of lines of code clones (LOCC); thus, we did not observe the changes between each check-in, or the changes that did not result in $\Delta LOCC \neq 0$ even if the clones' text changed. Our study also does not model clones that do not live long enough to make it to the revision history. Based on our experience of observing programmers copy and paste, we suspect that programmers create more clones temporarily before finding an appropriate level of an abstraction.

Our definition of locally unfactorable clones is Java language dependent; thus our claims about the locally unfactorable clones may not apply to other languages. We speculate that, in other programming languages, different types of locally unfactorable clones would be found.

4.6 Comparison with Clone Evolution Analyses

This section describes several clone evolution analysis techniques—many of which were developed after our study in 2005—and compares them with ours.

Evolution of code clones was studied for the first time by Laguë et al. [180]. They studied clones in six versions of a large telecommunication software system and found that a significant number of clones were removed but the overall number of clones increased over time in the system. Their approach traces code clones in consecutive versions using a metric-based clone detector and classifies clones into four categories: new clones, modified clones, never modified clones, and deleted clones. However, their analysis does not address how elements in a group of code clones change with respect to other elements in the group.

Geiger et al. [95] studied the relation of code clones and change couplings (files which are committed at the same time by the same author with the same modification description) in the Mozilla project. Our study explains why Geiger et al. did not find a strong correlation between clones and change coupling: the amount of consistently changing clones is lower than may have been believed. Similarly, Lozano et al. [194] examined the relation between code clones and change coupling at a finer granularity (method-level) in *dnsjava*.

Aversano et al. [11] slightly refined our clone evolution model by further categoriz-

ing the *Inconsistent Change* pattern into the *Independent Evolution* pattern and the *Late Propagation* pattern. Independent evolution means that programmers intentionally applied inconsistent updates to clones to implement different pieces of functionality. Late propagation means that programmers must have accidentally applied inconsistent updates to clones because programmers later propagated the same change to them. They used the SimScan clone detector[11] and analyzed clone evolution in *dnsjava* and *ArgoUML*. In their study, only 45% of clone groups underwent consistent changes, 32% underwent independent evolution, and 18% underwent late propagation. Through manual analysis, they found that, if the clones contain bugs, developers always consistently update clones.

Krinke [177] also extended our clone genealogy analysis and independently studied clone evolution patterns in five open source projects (*ArgoUML*, *carol*, *jdt.core*, *Emacs*, and *FilZilla*). Krinke came to similar conclusions. Only roughly 45% to 55% of the clones change consistently and many clones disappear in a short amount of time due to divergent changes. Furthermore, it is rare for inconsistently changed clone groups to become consistently changed clones later by applying missed changes, contradicting Aversano et al.'s *Late Propagation* result.

Balint et al. [18] developed a visualization tool that shows the evolution of code clones. Their *Clone Evolution View* visualizes code clones at a line granularity and also shows four types of additional information: (1) who created and modified code clones, (2) the time of the modifications, (3) the location of clones in the system, and (4) the size of code clones. Balint et al. applied this tool to three open source projects (*Ant*, *ArgoUML* and *Ptolemy2*) and discovered several common cloning patterns such as *Inconsistent Line Cloning Fixed by the Same Author*. Though their visualization explicitly represents clone evolution, it differs by focusing on meta information instead of code change.

To the best of our knowledge, our clone genealogy extractor is the first tool that systematically analyzes clone evolution patterns by monitoring how a clone group evolves. Our study is one of the first to report that code cloning is not necessarily harmful and refactoring is not always the best solution for clones. Many researchers [11, 74, 95, 102, 152, 194]

[11] http://www.blue-edge.bg/simscan

independently reproduced similar results from different subject programs.

4.7 Proposed Tools

Our clone genealogy study results indicate that the problems of code clones are not so black and white as previous research has assumed and that refactoring would not help several types of clones. Based on our clone studies (Chapter 3 and Chapter 4), we propose new clone maintenance approaches. For each tool, we describe what it does, what kinds of potential benefit it provides, and the mechanism through which it can be implemented.

Cloning Dependency Tracking We propose a tool that maintains and visualizes cloning dependencies. This tool can help programmers locate related code clones. Suppose that a programmer copied example code and then modified a small part of it. When the example code requires some adaptive changes—e.g., a library used in the example is updated to a new version—all instances of the copied code must be located. The programmer may need to consult the original author to figure out how to adapt the copied code appropriately, if he does not fully understand the logic of the copied code.

We found a concrete example from the Mozilla project. One bug required a programmer to fix bugs that had been propagated to 12 different places by copy and paste. This bug was created by invoking the `appendFrames` method instead of `insertFrames` method. (See Figure 4.8.) The code snippet was copied twice within the same method and the method itself was copied three times. Ultimately, 12 structural clones containing that faulty code snippet were produced. The programmer who fixed the bug had to lexically search the code base for comments starting with `XXX` in order to apply the appropriate modifications consistently. Lexical search will fail if `XXX` did not exist in the copied comment, or if the structural template evolved very differently.

This cloning dependency tracker can be implemented either by capturing copy and paste operations in IDEs or using the *Add* pattern in the extracted clone genealogy.

Structural Template Inference We propose a tool that learns a structural template from the frequently copied code and assists programmers in reusing the template in a safe

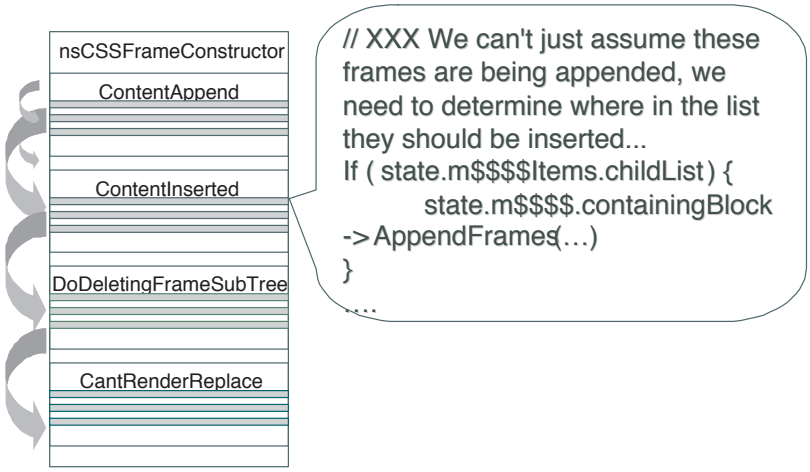

Figure 4.8: Mozilla bug id: 217604

manner. This tool can save repetitive edits by providing advanced block completion or by removing the code that is irrelevant to the pasted context. This tool can be implemented by monitoring subsequent edits on copied code in an IDE or by extracting the common internal representation of clones. Duala-Ekoko and Robillard's Clone Tracker [74] extracts a AST-based structural template descriptor by comparing clones found by the SimScan clone detector.[12]

Cloning Related Bug Detection and Prevention When a programmer fails to change clones consistently, this missed update could lead to a bug. We propose a tool that detects and prevents cloning related bugs by identifying a missed update. This tool can be implemented by detecting the *Consistent Change* pattern followed by the *Inconsistent Change* pattern in a clone genealogy as consistently changed clones are likely to change similarly in the future.

There are several motivating examples for such a tool. Li et al. reported that errors in Linux were created when a programmer copied code but failed to rename identifiers correctly in the pasted code [187]. Ying *et al.*, also reported a cloning related bug in Mozilla [308]. A web browser using the gtk UI toolkit and the version using the xlib UI toolkit were code

[12] http://blue-edge.bg/download.html

clones. When a developer changed the version using gtk but did not update the version using xlib, this inconsistent update led to a serious defect, "huge font crashes X Windows."

Refactoring Recommendation Although many modern IDEs provide automatic refactoring features, they do not suggest what to refactor or when to refactor code. We believe that clones are good candidates for refactoring and there's a right time to refactor them. If programmers refactor clones too early, they may have to undo the refactoring when the clones diverge. On the other hand, if programmers wait too long before they restructure code, they would get only marginal benefit on their investment. This proposed refactoring recommender can leverage clone genealogy information—the age of clones and the frequency of consistent changes—to suggest which code and when to refactor.

Simultaneous Text Editing Recommendation Simultaneous text editing is proposed and prototyped by Miller and Myers to automate repetitive text editing [215]. After describing a set of regions to edit, a user can edit any one record and see equivalent edits applied simultaneously to all other records. A similar editing technique, called *linked editing*, applies the same editing change to a set of code clones specified by a user [281]. These editing techniques require a user to manually specify what must be simultaneously edited. We propose a tool that recommends clones suitable for simultaneous text editing by monitoring the frequency of the *Consistent Change* pattern and by assessing the cost of the refactoring. This tool can effectively maintain many long-lived, consistently changing, locally unfactorable clones.

4.8 Conclusions from the Clone Genealogy Study

Our clone genealogy study found that refactoring may not always improve software with respect to clones for two reasons. First, many code clones exist in the system for only a short time; extensive refactoring of such short-lived clones may not be worthwhile if they are likely to diverge from one another very soon. Second, many clones, especially long-lived clones that have changed consistently with other elements in the same group, are not easily refactorable due to programming language limitations. These insights show that

refactoring will not help in dealing with some types of clones and open up opportunities for complementary clone maintenance tools that target these other classes of clones. In addition, our analysis of consistent clone updates helped us build insights into systematic changes. This motivated our rule-based change inference approach that leverages the systematicness of code-level changes.

Chapter 5

INFERRING CHANGES TO API NAME AND SIGNATURE

In many situations, programmers need to comprehend differences between two program versions: They often review code changes done by other developers. They also inspect their own code changes when writing documentation or check-in comments. In these situations, programmers may ask the following questions about code changes: "What changed between the two program versions?" "Is anything missing in that change?" and "Why was this set of files changed together?" Our goal is to help programmers answer these kinds of questions by building a tool that extracts high-level change descriptions. Achieving this goal is also important for enabling various kinds of mining software repository research that analyzes program history by matching corresponding code elements across versions.

Based on the insight that high-level changes are often systematic at a code level, our approach uses rule-based representations to concisely describe systematic changes and to note exceptions to systematic change patterns. This chapter in particular describes how we instantiated this rule-based change inference approach at a *method-header level* (API-level). Our API-change rule concisely describes systematic API changes, and our algorithm automatically infers such rules from two program versions.

Section 5.1 describes the representation of API change-rules. Section 5.2 describes the API change-rule inference algorithm. Section 5.3 presents the results on five open source projects and compares our approach to three other approaches.

5.1 Definition of API Change-Rule

As our survey in Section 2.2 shows, many existing techniques take two program versions as input and automatically infer changes from one version to the other. These techniques usually match code at a fixed granularity and represent the inferred changes as an unstructured, often lengthy, list of code matches. Although this unstructured representation is adequate

for conventional uses (e.g., transferring code coverage information in profile propagation tools), it is not effective in helping programmers reason about software changes at a high level. Programmers are usually left with the burden of reading all code matches one by one and discovering emerging patterns from them.

Consider an example where a programmer reorganizes a chart drawing program by the type of a rendered object, moving axis-drawing classes from the package `chart` to the package `chart.axis`. To add tool-tip on/off information, she appends a `boolean` parameter to a set of chart-creation interfaces. A method-level matching tool [167, 316] would report a long list of matches by individually enumerating each moved method and each modified interface. One may have to examine hundreds or thousands of matches before discovering that a few simple transformations took place. Moreover, if the programmer neglected to move one axis drawing class, this error would be hard to detect.

It is important to note that both changes consist of applying a similar transformation to a set of related code elements. One change moves a group of classes that match the *Axis pattern from package `chart` to package `chart.axis`. The other adds a `boolean` parameter to a group of methods that match the create*Chart pattern.

Our change vocabulary represents a group of similar transformations explicitly in a rule-based representation. This chapter discusses a change-rule representation *at the level of a method-header* (i.e., API level). Given two program versions (P_1, P_2), our goal is to find a set of *API change-rules*, in turn generating a set of method-header level matches. Each change-rule maps a subset of method-headers in P_1 to a subset of method-headers in P_2.

API change-rules model a program only at the level of method-headers and do not model changes within method bodies, control logic, temporal logic, etc. Hence, applying the inferred change-rules to an old program version does not reconstruct a new version.

Transformation. Our API change-rules support transformations at the level of a method-header. A method-header is defined as a tuple, (`package:String, class:String, procedure:String, input_argument_list:List[String], return_type:String`). In pseudo code descriptions, a method header is represented as `(pack, cls, pr, sig, ret)`. For presentation purposes, a Java method-

Table 5.1: Comparison between programmer's intent and existing tools' results

Programmer's Intent	Matching Tool Results [167]	Refactoring Reconstruction Results [295]

Move classes that draw axes from `chart` package to `chart.axis` package	`[chart.DateAxis..., chart.axis.DateAxis...]`
	`[chart.NumberAxis..., chart.axis.NumberAxis...]`
	`[chart.ValueAxis..., chart.axis.ValueAxis...]`
		Move class `DateAxis` from `chart` to `chart.axis`
		Move class `NumberAxis` from `chart` to `chart.axis`
		Move class `ValueAxis` from `chart` to `chart.axis`

Widen the APIs of chart factory methods by adding a `boolean` type argument	`[createAreaChart(Data), createAreaChart(Data, boolean)]`
	`[createLChart(IntData), createLChart(IntData, boolean)]`
	`[createPieChart(PieData), createPieChart(PieData, boolean)]`
		Add `boolean` parameter to `createAreaChart`
		Add `boolean` parameter to `createLChart`
		Add `boolean` parameter to `createPieChart`

header is represented as return_type package.class.procedure(input_argument_list).[1] Each transformation is a tuple modifying operation.

- *packageReplace(x:Method, f:String, t:String)*:
  change x's package name from f to t:
  ```
  if(x.pack==f) x.pack:= t;
  ```

- *classReplace(x:Method, f:String, t:String)*:
  change x's class name from f to t:
  ```
  if(x.cls==f) x.cls:= t;
  ```

- *procedureReplace(x:Method, f:String, t:String)*:
  change x's procedure name from f to t:
  ```
  if(x.pr==f) x.pr:= t;
  ```

- *returnReplace(x:Method, f:String, t:String)*:
  change x's return type from f to t:
  ```
  if(x.ret==f) x.ret:= t;
  ```

- *inputSignatureReplace(x:Method,f:List[String], t:List[String])*:
  change x's input argument list from f to t:[2]
  ```
  if(x.sig.equals(f)) x.sig:= t;
  ```

- *argReplace(x:Method, f:String, t:String)*:
  change argument type f to t in x's input argument list:
  ```
  for (0<=index<x.sig.size()) {if (x.sig.get(index)==f) x.sig.set(index,t)}
  ```

- *argAppend(x:Method, t:List[String])*:
  append all of the argument types in t to the end of x's input argument list:
  ```
  x.sig.addAll(t)
  ```

---

[1] The return_type is sometimes omitted for presentation purposes.

[2] *inputSignatureReplace* subsumes *argReplace*, *argAppend*, and *argDelete*.

- *argDelete(x:Method, t:String)*:

  delete every occurrence of type t in the x's input argument list:

  ```
  x.sig.removeAll(new List[t])
  ```

- *typeReplace(x:Method, f:String, t:String)*:

  change every occurrence of type f to t in x:

  ```
  if (x.cls==f) x.cls:= t;
  if (x.cls==x.pr && x.cls==f) x.pr:= t; /* constructor methods */
  if (x.ret==f) x.ret:= t;
  for (0<=index<x.sig.size()) {if (x.sig.get(index)==f) x.sig.set(index,t)}
  ```

Rule. A change-rule consists of a scope, exceptions and a transformation.

• for all x:method-header in (scope)

  except (exceptions)

  transformation(x)

The only method-headers transformed are those in the scope but not in the exceptions. When a group of method-headers have similar names, we summarize these method-headers as a scope expression using a wild-card pattern matching operator. For example, *.*Plot.get*Range() describes methods with any package name, any class name that ends with `Plot`, any procedure name that starts with `get` and ends with `Range`, and an empty argument list.[3] This use of a wild card pattern is based on the observation that programmers tend to name code elements similarly when they belong to the same concern [111].

For example, the following rule means that all methods that match the chart.*Plot.get*Range() pattern take an additional `ValueAxis` argument.

• for all x:method-header in chart.*Plot.get*Range()

  argAppend(x, [ValueAxis])

  [Interpretation: All methods with a name "chart.*Plot.get*Range()" appended an input argument with the

[3] Wild-card pattern matching is implemented using java.util.regex.Pattern in Java 1.4. A Java method header m matches a scope expression s if Pattern.compile(s).matcher(m.toString()) returns true.

ValueAxis type.]

Rules explicitly note exceptions that violate systematic change patterns. For the preceding example, the following rule describes that the `getVerticalRange` method did not change its input signature similarly as other methods.

• for all x:method-header in chart.*Plot.get*Range()

except {chart.MarkerPlot.getVerticalRange}

argAppend(x, [ValueAxis])

[Interpretation: All methods with a name "chart.*Plot.get*Range()" added an input argument with the ValueAxis type except the chart.MarkerPlot.getVerticalRange method.]

In addition, to discover emerging transformation patterns, a scope can have disjunctive scope expressions. The following rule means that all methods whose class name includes `Plot` or `JThermometer` changed their package name from `chart` to `chart.plot`.

• for all x:method-header in chart.*Plot*.*(*)

or chart.*JThermometer*.*(*)

packageReplace(x, chart, chart.plot)

[Interpretation: All methods with a name "chart.*Plot*.*(*)" or "chart.*JThermometer*.*(*)" moved from the chart package to the chart.plot package.]

Rule-based Matching. We define a matching between two versions of a program by a set of change-rules. The methods that are not matched by any rules are either deleted or added methods.[4] For example, the five rules in Table 5.2 explain seven matches. The unmatched method O2 is considered deleted. The scope of one rule may overlap with the scope of another rule as some methods undergo more than one transformation. Our algorithm ensures that we infer a set of rules such that the application order of rules does not matter.

5.2 Inference Algorithm

Our algorithm accepts two versions of a program and infers a set of API change-rules. Our algorithm has four parts: (1) generating seed matches, (2) generating candidate rules based

[4]Method headers that are identical in both versions are excluded from a rule-based matching.

Table 5.2: Rule-based matching example

A set of method-headers in P_1	A set of method-headers in P_2
O1. `chart.VerticalPlot.draw(Grph,Shp)`	N1. `chart.plot.VerticalPlot.draw(Grph)`
O2. `chart.VerticalRenderer.draw(Grph,Shp)`	N2. `chart.plot.HorizontalPlot.range(Grph)`
O3. `chart.HorizontalPlot.range(Grph,Shp)`	N3. `chart.axis.HorizontalAxis.getHeight()`
O4. `chart.HorizontalAxis.height()`	N4. `chart.axis.VerticalAxis.getHeight()`
O5. `chart.VerticalAxis.height()`	N5. `chart.ChartFactory.createAreaChart(Data, boolean)`
O6. `chart.ChartFactory.createAreaChart(Data)`	N6. `chart.ChartFactory.createGanttChart(Interval, boolean)`
O7. `chart.ChartFactory.createGanttChart(Interval)`	N7. `chart.ChartFactory.createPieChart(PieData, boolean)`
O8. `chart.ChartFactory.createPieChart(PieData)`	

Rule			Matches Explained
scope	*exceptions*	*transformation*	
*chart.*Plot.*(*)*		*packageReplace(x, chart, chart.plot)*	[O1, N1], [O3, N2]
*chart.*Axis.*(*)*		*packageReplace(x, chart, chart.axis)*	[O4, N3], [O5, N4]
*chart.ChartFactory.create *Chart(*)*		*argAppend(x, [boolean])*	[O6, N5], [O7, N6], [O8, N7]
chart..*(Grph, Shp)*	{O2}	*argDelete(x, Shp)*	[O1, N1], [O3, N2]
*chart.*Axis.height()*		*procedureReplace(x, height, getHeight)*	[O4, N3], [O5, N4]

on the seeds, (3) iteratively selecting the best rule among the candidate rules, and (4) post-processing the selected candidate rules to output a set of API change-rules. We first describe a naïve version of our algorithm, followed by a description of essential performance improvements for the second and third parts of the algorithm. Finally, we summarize key characteristics of our algorithm.

Part 2 is a bottom-up approach in the sense that hypotheses about high-level changes are generated from seeds. Part 3 is a top-down approach in the sense that rules with a large number of matches are found before finding rules with fewer number of matches.

Part 1. Generating Seed Matches. We start by searching for method headers that are similar on a textual level, which we call *seed matches*. Seed matches provide initial hypotheses about the kind of changes that occurred. Given the two program versions (P_1, P_2), we extract a set of method headers O and N from P_1 and P_2 respectively. Then, for each method header x in $O - N$, we find the closest method header y in $N - O$ in terms of the token-level name similarity, which is calculated by dividing x and y into a list of tokens starting with capital letters and then computing the longest common subsequence at a token level [139]. Our seed match generation algorithm is summarized in Algorithm 1. If the name similarity is over a threshold γ, the pair is added to the initial set of seed matches. In our study, we found that thresholds in the range of 0.65-0.70 (meaning 65% to 70% of tokens are the same) gave good empirical results. The seeds need not all be correct matches, as our rule selection algorithm (Part 3) rejects bad seeds and leverages good seeds. Seeds can instead come from other sources such as check-in comments, recorded refactorings, or other matching tools' results. Many algorithms match code elements based on string similarity using n-gram or bi-gram matching of source code [90, 302]. We speculate that, compared to these, our LCS-based algorithm is more sensitive to reordering of tokens but more effective for code written in the CamelCase naming convention.[5]

[5]CamelCase naming is the practice of writing compound words or phrases in which words are joined without spaces and are capitalized within the compound.

Algorithm 1: Seed Generation

Input:

P_1 `/* an old program version */`

P_2 `/* a new program version */`

γ `/* a seed similarity threshold */`

Output:

S ={(d,c) | d ∈ D, c ∈ C where `tokenSim` (d,c) > γ} `/* a set of seed matches */`

`/* A Java method-header is defined as a tuple of` $(pack, cls, pr, sig, ret)$ `where` $pack$ `is a package name,` cls `is a class name,` pr `is a procedure name,` sig `is a list of input argument types (i.e.,` $[arg1, arg2, \ldots]$`), and` ret `is a return type. */`

O:= `extractMethodHeaders(`$P1$`);`

N:= `extractMethodHeaders(`$P2$`);`

D := O − N `/* domain */`

C := N − O`/* codomain */`

S := ∅ ;

foreach *d in* D **do**
  bestMatch:= null;
  bestSimToken:= 0;
  foreach *c in* C **do**
    simToken:= `tokenSim` (d, c);
    if *(simToken* > γ*)* ∧ *(simToken > bestSimToken)* **then**
      bestSimToken:= simToken;
      bestMatch:= c;
    end
  end
  S := S ∪ {(d, bestMatch)};
end

Part 2. Generating Candidate Rules. For each seed match $[x, y]$, we build a set of *candidate rules* in three steps. A candidate rule may include one or more transformations $t_1, \ldots t_i$ such that $y = t_1(...t_i(x))$, unlike a change-rule, where for every match $[x, y]$, y is the result of applying a single transformation to x.

We write candidate rules as "for all x:method-header in scope, $t_1(x) \wedge ... \wedge t_i(x)$." This representation allows our algorithm to find a match $[x, y]$ where x undergoes multiple transformations to become y.

Step 1. We compare x and y to find a set of transformations $T = \{t_1, t_2, \ldots, t_i\}$ such that $t_1(t_2(\ldots t_i(x))) = y$. We then create T's power set 2^T. For example, a seed [`chart.VerticalAxis.height()`, `chart.plot.VerticalAxis.getHeight()`] produces the power set of packageReplace(x, chart, chart.plot) and procedureReplace(x, height, getHeight). The pseudo code of function `extractTransformations(seed)` and function `extractSignatureTransformations(lsig,rsig)` is detailed in pages 110 and 111.

Step 2. We conjecture scope expressions from a seed match $[x, y]$. We divide x's full name to a list of tokens starting with capital letters. For each subset, we replace every token in the subset with a wild-card operator to create a candidate scope expression. As a result, when x consists of n tokens, we generate a set of 2^n scope expressions based on x. For the preceding example seed, our algorithm finds $S = \{$*.*.*(*), chart.*.*(*), chart.Vertical*.*(*), ..., *.*Axis.height(), ..., chart.VerticalAxis.height()$\}$.

Step 3. We generate a candidate rule with scope expression s and compound transformation t for each (s, t) in $S \times 2^T$. We refer to the resulting set of candidate rules as CR. Each element of CR is a generalization of a seed match.

Part 3. Evaluating and Selecting Rules. Our goal is to select a small subset of candidate rules in CR that explain a large number of matches. While selecting a set of candidate rules, we enforce candidate rules to have a limited number of exceptions.

The inputs are a set of candidate rules (CR), a domain ($D = O - N$), a codomain ($C = N$), and an exception threshold ($0 \leq \epsilon < 1$). The outputs are a set of selected candidate rules (R), and a set of found matches (M). For a candidate rule r, "for all x in scope, $t_1(x) \wedge ... \wedge t_i(x)$":

1. r has a **match** $[a, b]$ if $a \in$ scope, $t_1, ..., t_i$ are applicable to a, and $t_1(...t_i(a)) = b$.

Function `extractTransformations(`*seed*`)`

```
l:= seed.left;
r:= seed.right;
T := ∅ ;   /* a set of transformations t1, t2, ...ti such that t1(...(ti(l))) = r.
*/
if l.pack ≠ r.pack then
  | T := T ∪ packageReplace(x, l.pack, r.pack)
end
if l.cls ≠ r.cls then
  | if l.cls is an inner class name then
  |   | T := T ∪ classReplace(x, l.cls, r.cls);
  | else
  |   | T := T ∪ typeReplace(x, l.cls, r.cls);
  | end
end
if (l.pr ≠ r.pr) ∧ (l.pr is not a constructor) then
  | T := T ∪ procedureReplace(x, l.pr, r.pr);
end
if l.ret ≠ r.ret then
  | T := T ∪ returnReplace(x, l.ret, r.ret);
end
if l.sig ≠ r.sig then
  | T := T ∪ extractInputSignatureTransformation (l.sig, r.sig);
end
2^T := createPowerSet (T);
return 2^T;
```

Function `extractInputSignatureTransformation(`*lsig, rsig*`)`

```
Transforamtion sigt := null;
if lsig.size() = rsig.size() then
    n :=0;
    for 0≤i<lsig.size() do
        if lsig.get(i)≠ rsig.get(i) then
            sigt := argReplace(x, lsig.get(i), rsig.get(i);
            n:= n+1;
        end
    end
    if n≠ 1 then
        /* Allow an argReplace transformation only if applying a single
           argReplace transformation to lsig generates rsig.                */
        sigt := null;
    end
end
else if (lsig.size() < rsig.size()) ∧ lsig.subList(0, lsig.size()).equals(rsig.subList(0, lsig.size())
then
    /* Allow an argAppend transformation only if lsig is a prefix of rsig   */
    appendOperand := rsig.subList(lsig.size()+1, rsig.size());
    sigt := argAppend(x, appendOperand)
end
else if (rsig.convertListToSet() ⊂ lsig.convertListToSet()) ∧ (|rsig.convertListToSet() −
lsig.convertListToSet()| = 1) then
    /* Allow an argDelete transformation only if {rsig} is a subset of {lsig}
       and |{rsig}−{lsig}|=1.                                               */
    sigt := argDelete(x, rsig.convertListToSet() − lsig.convertListToSet);
end
if sigt == null then
    sigt := inputSignatureReplace(x, lsig, rsig);
end
return sigt;
```

2. a match $[a, b]$ **conflicts** with a match $[a', b']$ if $a = a'$ and $b \neq b'$
3. r has a **positive** match $[a, b]$ given D, C, and M, if $[a, b]$ is a match for r, $[a, b] \in \{D \times C\}$, and none of the matches in M conflict with $[a, b]$
4. r has a **negative** match (an exception) $[a, b]$, if it is a match for r but not a positive match for r.
5. r is a **valid** rule if the number of its positive matches is at least $(1-\epsilon)$ times the number of its matches. For example, when ϵ is 0.34 (our default), r's negative matches must be fewer than roughly one third of its matches.

Our algorithm greedily selects one candidate rule at each iteration such that the selected rule maximally increases the total number of matches. Initially we set both R and M to the empty set. In each iteration, for every candidate rule $r \in CR$, we compute r's matches and check whether r is valid. Then, we select a valid candidate rule s that maximizes $|M \cup P|$ where P is s's positive matches. After selecting s, we update $CR := CR - \{s\}$, $M := M \cup P$, and $R := R \cup \{(s, P, E)\}$ where P and E are s's positive and negative matches respectively, and we continue to the next iteration. The iteration terminates when no remaining candidate rules can explain any additional matches. The naïve version of this greedy algorithm has $O(|CR|^2 \times |D|)$ time complexity.

Part 4. Post Processing. To convert a set of candidate rules to a set of change-rules, for each transformation t, we find all candidate rules that contain t and then create a new scope expression by combining these rules' scope expressions. Then we find exceptions to this new rule by enumerating negative matches of the candidate rules and checking if the transformation t does not hold for each match.

Optimized Algorithm. Two observations allow us to improve the naïve algorithm's performance. First, if a candidate rule r can add n additional matches to M at the i^{th} iteration, r cannot add more than n matches on any later iteration. By storing n, we can skip evaluating r on any iteration where we have already found a better rule s that can add

more matches than r. Second, candidate rules have a subsumption structure because the scopes can be subsets of other scopes (e.g., *.*.*(*Axis) $\subset$ *.*.*(*)).

The pseudo code of our optimized algorithm is described in Algorithm 4 and function `selectTheBestRule` on pages 114–115.

Our optimized algorithm behaves as follows. Suppose that the algorithm is at the i^{th} iteration, and after examining $k-1$ candidate rules in this iteration, it has found the best valid candidate rule s that can add N additional matches. For the k^{th} candidate rule r_k,
(1) If r_k could add fewer than N additional matches up to i-1^{st} iteration, skip evaluating r_k as well as candidate rules with the same set of transformations but a smaller scope, as our algorithm does not prefer r_k over s.
(2) Otherwise, reevaluate r_k.

(2.1) If r_k cannot add any additional matches to M, remove r_k from CR.

(2.2) If r_k can add fewer than N additional matches regardless of its validity, skip evaluating candidate rules with the same set of transformations but a smaller scope.

(2.3) If r_k is not valid but can add more than N additional matches to M, evaluate candidate rules with smaller scope and the same set of transformations.
(3) Update s and N as needed and go to step (1) to consider the next candidate rule in CR.

By starting with the most general candidate rule for each set of transformations and generating more candidate rules on demand only in step (2.3) above, the optimized algorithm is much more efficient that the naïve algorithm. Running our tool currently takes only a few seconds for the usual check-ins and about seven minutes in average for a program release pair. Though optimized, our algorithm is not optimal in the sense that it does not guarantee finding the smallest number of rules for the same set of matches. This optimized algorithm's worst case complexity is the same as the naïve algorithm, $O(|CR|^2 \times |D|)$. In the case of expanding only one rule to investigate its children rules at each level of the subsumption lattice, its time complexity is $O(|log_n(CR)|^2 \times |D|)$ where n is the number of tokens in $seed.left$.

Key Characteristics of Our Algorithm. First, our algorithm builds insight from seed matches, generalizes the scope that a transformation applies to, and validates this insight.

Algorithm 4: Rule Generation and Selection - Optimized Algorithm

```
Input:
S /* a set of seed matches                                                    */
ϵ /* an exception threshold                                                  */
D /* domain: extractMethodHeaders(P1) − extractMethodHeaders(P2)            */
C /* codomain: extractMethodHeaders(P2)                                      */
Output:
R /* a set of selected rules                                                 */
M /* a set of found matches                                                  */
/* Initialize R, M, and CR                                                   */
R := ∅, M := ∅, CR := ∅;
/* Create an initial set of rules                                            */
foreach seed ∈ S do
    2^T := extractTransformations (seed);
    foreach trans ∈ 2^T do
        scope:= findTheMostGeneralScope (seed.left, trans);
        rule:= createNewRule (scope, trans);
        CR := CR ∪ {rule};
    end
end
cont := true;
while cont do
    n := |M |;
    s = selectTheBestRule (CR, D, C, M, ϵ);
    R := R ∪ {s};
    CR := CR − {s};
    M := M ∪ s.positive;
    if (|M|=n) then
        cont := false;
    end
end
```

Function `selectTheBestRule(`*CR, D, C, M,* ϵ`)`

```
/* numRemainingPositive (r) returns the number of |r.positive.getLefts() -
   M.getLefts()|.                                                          */
/* isValid (r, D, C, M, ε) returns true if the number of r's positive matches is
   at least (1-ε) times the number of r'matches.                           */
/* Scan rules in CR and update N                                           */
N := 0, s := null;
foreach r_k ∈ CR do
    if (numRemainingPositive (r_k) > N) ∧ (isValid (r_k, D, C, M, ε)) then
        N = numRemainingPositive (r_k);
        s = r_k;
    end
end
/* If an invalid rule r_k in CR can find more than N matches, expand its
   children rules.                                                         */
toBeRemoved := ∅; toBeAdded := ∅;
foreach r_k ∈ CR do
    if numRemainingPositive (r_k)=0 then
        toBeRemoved := toBeRemoved ∪ {r_k};
    end
    else if (numRemainingPositive (r_k) > N) ∧ (isValid (r_k, D, C, M, ε))= false then
        toBeRemoved := toBeRemoved ∪ {r_k}; children = createChildrenRules (r_k,N);
        foreach c ∈ children do
            if (isValid (c, D, C, M, ε)) ∧ (numRemainingPositive (c) > N) then
                N := numRemainingPositive (c); s := c;
            end
        end
        toBeAdded := toBeAdded ∪ children;
    end
end
/* Add toBeAdded to CR and remove toBeRemoved from CR.                     */
CR := CR ∪ toBeAdded;
CR := CR − toBeRemoved;
return s ;
```

Second, it prefers a small number of general rules to a large number of specific rules. Third, when there are a small number of exceptions that violate a general rule, our algorithm allows these exceptions but remembers them.

5.3 Evaluation

We assess the benefits of our API change-rule inference technique in two ways. First, we evaluate its accuracy by measuring the precision and recall of the inferred rules. Second, we measure our inferred rules' effectiveness in helping programmers reason about program changes by comparing the conciseness of our results with other approaches' results.

To evaluate our inferred rules, we compared the method-header level matches found by our inferred rules (M) with the ground truth—a set of correct matches (E). We identified a set of correct matches in two steps. First, we used our own inference algorithm on each version pair in both directions (which can find additional matches) and computed the union of those matches with the matches found by other approaches. Second, we labeled correct matches through a manual inspection. For this manual inspection, we developed a viewer that shows each rule along with the corresponding method-header level matches (Figure 5.1). Our inferred rules are shown in the top pane; the corresponding method-header level matches are shown in the bottom pane along with the matches found by other approaches. To help with inspection, matches that are found by one approach but not the other are marked in color.[6]

Our quantitative evaluation is based on the three following criteria.

Precision: the percentage of our matches that are correct, $\frac{|E \cap M|}{|M|}$.

Recall: the percentage of correct matches that our tool finds, $\frac{|M \cap E|}{|E|}$.

Conciseness: the measure of how concisely a set of rules explains matches, represented as a M/R ratio = $\frac{|M|}{|Rules|}$. A high M/R ratio means that using rules instead of plain matches significantly reduces the size of results.

Our evaluations are based on released versions as well as check-in snapshots, i.e., internal, intermediate versions. The primary difference is that there tends to be a much

[6] Determining the correctness of matches is a subjective process. Having independent coders and measuring inter-rater agreement among the coders could increase confidence about our evaluation data sets.

larger delta between successive program releases than between successive check-in snapshots. To demonstrate our tool's effectiveness for cases where existing approaches produce overwhelmingly large results, our data set includes a release-granularity data on purpose.

Section 5.3.1 presents rule-based matching results for three open source release archives. Sections 5.3.2 presents comparison with two refactoring reconstruction tools [295, 302] and a method-level origin analysis tool [167]. Section 5.3.3 discusses the impact of the seed generation threshold (γ) and the exception threshold (ϵ). Section 5.3.4 discusses threats to the validity of our evaluation.

5.3.1 API Change-Rule Based Matching Results

Subject Programs. We chose three open source Java programs that have release archives on *sourceforge.net* and contain one thousand to seven thousand methods. The moderate size lets us manually inspect matches when necessary. *JFreeChart* is a library for drawing different types of charts, *JHotDraw* is a GUI framework for technical and structured graphics, and *jEdit* is a cross platform text editor. On average, release versions were separated by a two-month gap in *JFreeChart* and a nine-month gap in *JHotDraw* and *jEdit*.

Results. Table 5.3 and Table 5.4 summarize results for the projects (γ=0.7 and ϵ=0.34). $|O|$ and $|N|$ are the number of methods in an old version and a new version respectively. $|O \cap N|$ is the number of methods whose name and signature did not change. Running time is described in minutes.

The precision of our tool is generally high in the range of 0.78 to 1.00, and recall is in the range 0.70 to 1.00. The median precision and the median recall for each set of subjects is above, often well above, 0.90.

The M/R ratio shows significant variance not only across the three subjects but also for different release pairs in the same subject. The low end of the range is at or just over 1 for each subject, representing cases where each rule represents roughly a single match. The high end of the range varies from 2.39 (for *JEdit*) to nearly 244.26 (for *JHotDraw*). We observed, however, that most matches are actually found by a small portion of rules (recall our algorithm finds rules in descending order of the number of matches). Figure 5.2 plots

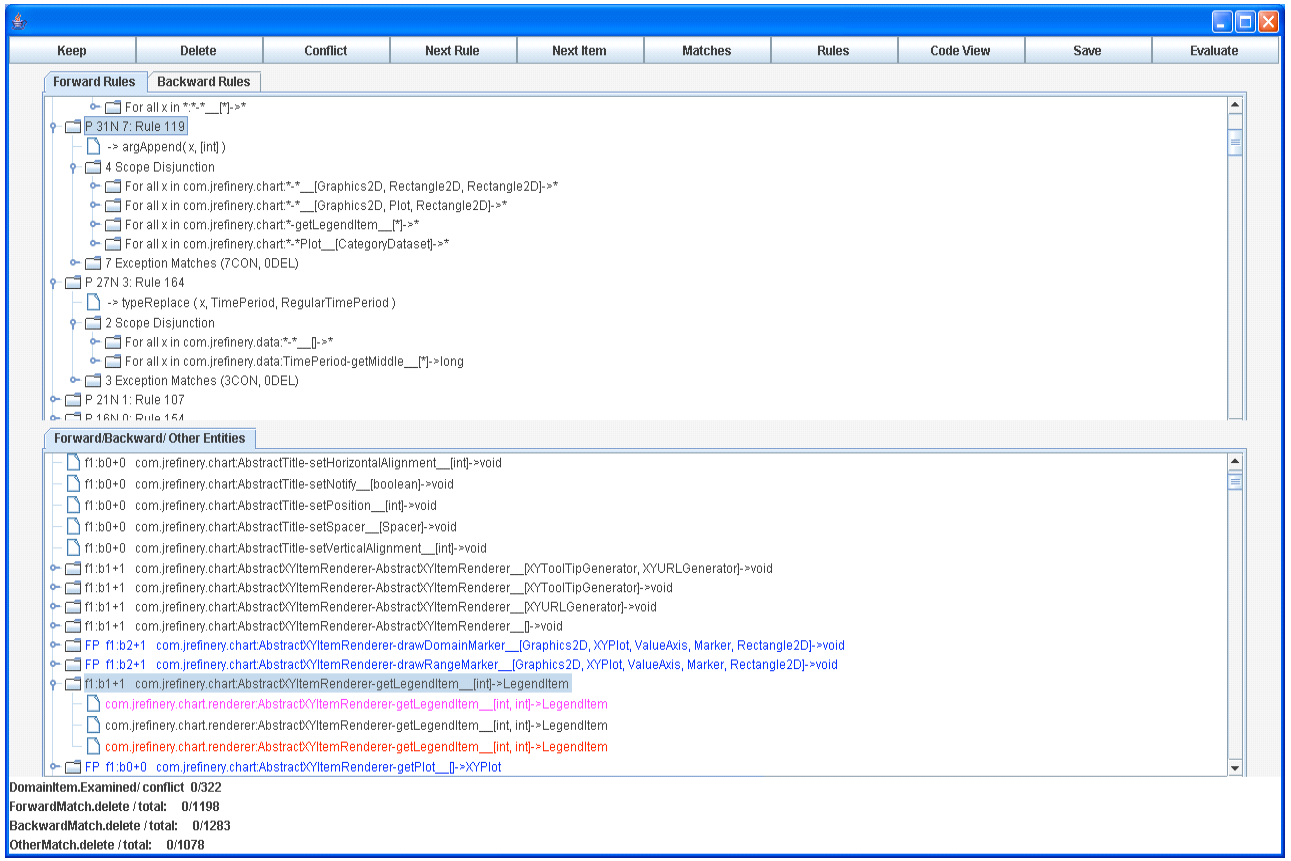

Figure 5.1: A viewer that presents each rule with corresponding method-header matches

Table 5.3: Rule-based matching results (1)

JFreeChart (www.jfree.org/jfreechart) The actual release numbers are prefixed with 0.9.									
Versions	$\|O\|$	$\|N\|$	$\|O \cap N\|$	**Rule**	**Match**	**Prec.**	**Recall**	**M/R**	**Time**
4→5	2925	3549	1486	178	1198	0.92	0.92	6.73	21.01
5→6	3549	3580	3540	5	6	1.00	1.00	1.20	<0.01
6→7	3580	4078	3058	23	465	1.00	0.99	20.22	1.04
7→8	4078	4141	0	30	4057	1.00	0.99	135.23	43.06
8→9	4141	4478	3347	187	659	0.91	0.90	3.52	22.84
9→10	4478	4495	4133	88	207	0.99	0.93	2.35	0.96
10→11	4495	4744	4481	5	14	0.79	0.79	2.80	<0.01
11→12	4744	5191	4559	61	113	0.78	0.79	1.85	0.40
12→13	5191	5355	5044	10	145	1.00	0.99	14.50	0.11
13→14	5355	5688	5164	41	134	0.94	0.86	3.27	0.43
14→15	5688	5828	5662	9	21	0.90	0.70	2.33	0.01
15→16	5828	5890	5667	17	77	0.97	0.86	4.53	0.32
16→17	5890	6675	5503	102	285	0.91	0.86	2.79	1.30
17→18	6675	6878	6590	10	61	0.90	1.00	6.10	0.08
18→19	6878	7140	6530	98	324	0.93	0.95	3.31	1.67
19→20	7140	7222	7124	4	14	1.00	1.00	3.50	<0.01
20→21	7222	6596	4454	71	1853	0.99	0.98	26.10	62.99
MED						**0.94**	**0.93**	**3.50**	**0.43**
MIN						**0.78**	**0.70**	**1.20**	**0.00**
MAX						**1.00**	**1.00**	**135.23**	**62.99**

Table 5.4: Rule-based matching results (2)

Versions	$\|O\|$	$\|N\|$	$\|O \cap N\|$	**Rule**	**Match**	**Prec.**	**Recall**	**M/R**	**Time**
JHotDraw (www.jhotdraw.org)									
5.2→5.3	1478	2241	1374	34	82	0.99	0.92	2.41	0.11
5.3→5.41	2241	5250	2063	39	104	0.99	0.98	2.67	0.71
5.41→5.42	5250	5205	5040	17	17	0.82	1.00	1.00	0.07
5.42→6.01	5205	5205	0	19	4641	1.00	1.00	244.26	27.07
MED						**0.99**	**0.99**	**2.54**	**0.41**
MIN						**0.82**	**0.92**	**1.00**	**0.07**
MAX						**1.00**	**1.00**	**244.26**	**27.07**
jEdit (www.jedit.org)									
3.0→3.1	3033	3134	2873	41	63	0.87	1.00	1.54	0.13
3.1→3.2	3134	3523	2398	97	232	0.93	0.98	2.39	1.51
3.2→4.0	3523	4064	3214	102	125	0.95	1.00	1.23	0.61
4.0→4.1	4064	4533	3798	89	154	0.88	0.95	1.73	0.90
4.1→4.2	4533	5418	3799	188	334	0.93	0.97	1.78	4.46
MED						**0.93**	**0.98**	**1.73**	**1.21**
MIN						**0.87**	**0.95**	**1.23**	**0.61**
MAX						**0.95**	**1.00**	**2.39**	**4.46**

the cumulative distribution of matches for the version pairs with the median M/R ratio from each of the three projects. The x axis represents the percentage of rules found after each iteration, and the y axis represents the recall and precision of matches found up to each iteration.

In all three cases, the top 20% of the rules find over 55% of the matches, and the top 40% of the rules find over 70% of the matches. In addition, as the precision plots show, the matches found in early iterations tend to be correct matches evidenced by a systematic change pattern. The fact that many matches are explained by a few rules is consistent with the view that a single conceptual change often involves multiple low level transformations, and it confirms that leveraging a systematic change structure is a good matching approach.

Our tool handled the major refactorings in the subject programs quite well. For example, consider the change from release 4 to 5 of *JFreeChart*. Although nearly half of the methods cannot be matched by name, our tool finds 178 rules and 1198 matches. The inferred rules indicate that there were many package-level splits as well as low-level API changes. As presented below, these kind of high-level change patterns are not detected by other tools we analyzed. Examples of the inferred rules in *JFreeChart* include:

- for all x:method-header in chart.*Plot.*(CategoryDataSet)

  or chart.*.*(Graph, Rect, Rect2D)

  or chart.*.*(Graph, Plot, Rect2D)

  argAppend(x, [int])

  [Interpretation: All methods with a name "chart.*Plot.*(CategoryDataSet)," "chart.*.*(Graph, Rect, Rect2D)" or "chart.*.*(Graph, Plot, Rect2D)" appended an int argument.]

- for all x:method-header in int renderer.*.draw*(*, Graph, Rect)

  returnReplace(x, int, AxisState)

  [Interpretation: All methods with a name "renderer.*.draw*(*, Graph, Rect)" changed their return type from int to AxisState.]

Table 5.5: Comparison: number of matches and size of result

Other Approach				**Our Approach**		**Improvement**			
Xing and Stroulia (XS)		**Match**	**Refactoring**	**Match**	**Rules**				
jfreechart	(17 release pairs)	8883	4004	9633	939	8%	more matches	77%	decrease in size
Weigerber and Diehl (WD)		**Match**	**Refactoring**	**Match**	**Rules**				
jEdit	*RCAll*	1333	2133	1488	906	12%	more matches	58%	decrease in size
(2715 check-ins)	*RCBest*	1172	1218	1488	906	27%	more matches	26%	decrease in size
Tomcat	*RCAll*	3608	3722	2984	1033	17%	fewer matches	72%	decrease in size
(5096 check-ins)	*RCBest*	2907	2700	2984	1033	3%	more matches	62%	decrease in size
S. Kim et al (KPW)		**Match**		**Match**	**Rules**				
jEdit	(1189 check-ins)	1430		2009	1119	40%	more matches	22%	decrease in size
ArgoUML	(4683 check-ins)	3819		4612	2127	21%	more matches	44%	decrease in size

Table 5.6: Comparison: precision

Comparison of Matches			**Match**	**Precision**
Xing and Stroulia (XS)		XS ∩ Ours	8619	1.00
jfreechart		Ours − XS	1014	0.75
(17 release pairs)		XS − Ours	264	0.75
Weißgerber and Diehl (WD)		WD ∩ Ours	1045	1.00
jEdit (2715 check-ins)	*RCAll*	Ours − WD	443	0.94
		WD − Ours	288	0.36
	RCBest	WD ∩ Ours	1026	1.00
		Ours − WD	462	0.93
		WD − Ours	146	0.42
Tomcat (5096 check-ins)	*RCAll*	WD ∩ Ours	2330	0.99
		Ours − WD	654	0.66
		WD − Ours	1278	0.32
	RCBest	WD ∩ Ours	2251	0.99
		Ours − WD	733	0.75
		WD − Ours	656	0.54
S. Kim et al. (KPW)		KPW ∩ Ours	1331	1.00
jEdit		Ours − KPW	678	0.89
(1189 check-ins)		KPW − Ours	99	0.75
ArgoUML (4683 check-ins)		KPW ∩ Ours	3539	1.00
		Ours − KPW	1073	0.78
		KPW − Ours	280	0.76

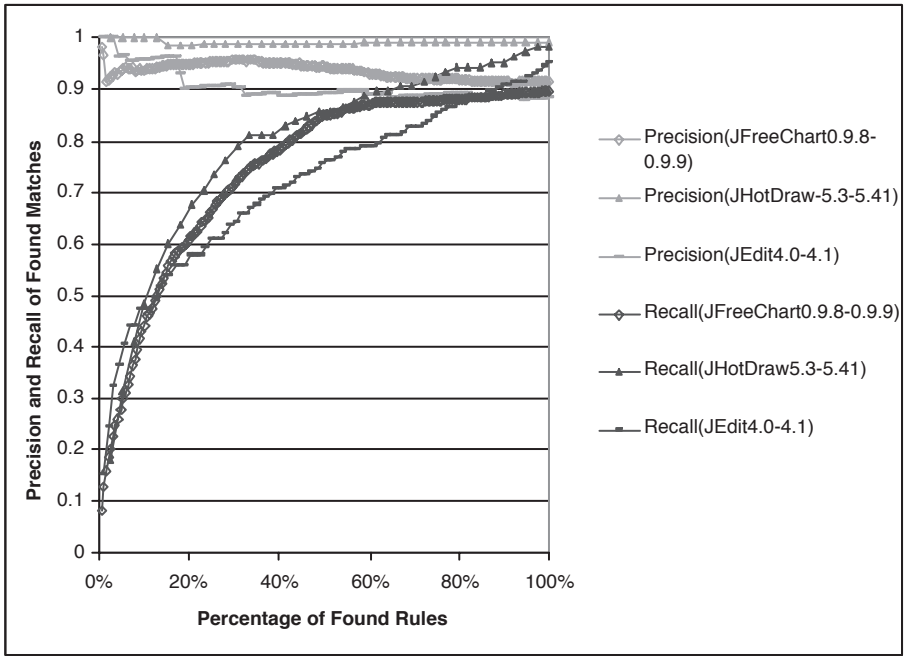

Figure 5.2: Recall and precision vs. percentage of found matches

5.3.2 Comparison with Refactoring Reconstruction Tools

Refactoring reconstruction tools (Section 2.2.2) compare two versions of a program and look for code changes that match a predefined set of refactoring patterns [92]. Among these tools, we compared our matching results with Xing and Stroulia's approach (XS) [302], Weißgerber and Diehl's approach (WD) [295], and S. Kim et al.'s approach (KPW) [167]. We chose these three approaches for comparison because they are representative of many refactoring tools. They were developed and published around the same time (from 2005 to 2007), and their results on open source project data were available to us. XS provided their results on *JFreeChart* release archives, WD provided their results on *jEdit* and *Tomcat* check-in snapshots, and KPW provided their results on *jEdit* and *ArgoUML* check-in snapshots.

In all three cases, we uniformly compared the precision and recall of method-level matches found by each approach. This required us building a tool that deduces method-level matches from XS and WD's inferred refactorings. We also compared the size of inferred

changes (the number of rules in our approach, the number of relevant refactorings in XS and WD's approach, and the number of method-level matches in KPW's approach).

Comparison with Xing and Stroulia's UMLDiff. XS's tool UMLDiff extracts class models from two versions of a program; traverses the two models; identifies corresponding entities based on their name and structure similarity; and reports additions and removals of these entities and inferred refactorings. XS can find most matches that involve more than one refactoring; however, to reduce its computational cost, it does not handle combinations of move and rename refactorings such as 'move `CrosshairInfo` class from `chart` to `chart.plot` package' and 'rename it to `CrosshairState`.' In contrast, our tool finds the following two rules that explain the combination of move and rename.

- for all x:method-header in *.*.*(*)

  typeReplace(x, CrosshairInfo, CrosshairState)

  [Interpretation: The CrosshairInfo type was renamed to the CrosshairState type.]

- for all x:method-header in chart.*Marker.*(*)

  packageReplace(x, chart, chart.plot)

  [Interpretation: All methods with a name "chart.*Marker.*(*)" moved from the chart package to the chart.plot package.]

The comparison results are summarized in Tables 5.5 and 5.6. Overall, XS's precision is about 2% (=8807/8883-9369/9633) higher. However, our tool finds 761 (=1014×0.75) correct matches not found by XS while there are only 199 (=264×0.75) correct matches that our tool failed to report. More importantly, our tool significantly reduces the result size by 77% by describing results as rules. Matches missed by XS often involve both rename and move refactorings. Matches missed by our tool often had a very low name similarity, indicating a need to improve our current seed generation algorithm.

Comparison with Weißgerber and Diehl's Work. WD's tool extracts added and deleted entities (fields, methods, and classes) by parsing deltas from a version control system and then compares these entities to infer various kinds of structural and local refactorings: move class, rename method, remove parameter, etc. The tool finds redundant refactor-

ing events for a single match. For example, if the `Plot` class were renamed to `DatePlot`, it would infer "rename class `Plot` to `DatePlot`" as well as move method refactorings for all methods in the `Plot` class. When it cannot disambiguate all refactoring candidates, it uses the clone detection tool CCFinder [149] to rank these refactorings based on code similarity. For example, if `VerticalPlot.draw(Graph)` is deleted and `VerticalPlot.drawItem(Graph)` and `VerticalPlot.render(Graph)` are added, it finds both "rename method `draw` to `drawItem`" and "rename method `draw` to `render`," which are then ordered.

We compared our results both with (1) all refactoring candidates RC_{all} and (2) only the top-ranked refactoring candidates RC_{best}. The comparison results with RC_{best} and RC_{all} (γ=0.65 and ϵ=0.34) are shown in Table 5.5 and 5.6. Compared to RC_{best}, our approach finds 27% more matches yet decreases the result size by 26% in *jEdit*, and finds 3% more matches yet decreases the result size by 62% in *Tomcat*. This result shows our approach achieves better matching coverage while retaining concise results. We manually inspected 50 sample check-ins to estimate precision for the matches missed by one tool but not the other as well as the matches found by both tools. For *jEdit*, our approach found 462 matches not identified by WD's RC_{best}, and RC_{best} found just over 146 matches that we failed to report. When combined with the precision, this means our approach found about 430 (=462×0.93) additional useful matches, and their approach found about 61 (=146×0.42) additional useful matches. *Tomcat* shows roughly similar results. WD's tool missed many matches when compound transformations were applied. Our tool missed some matches because γ=0.65 did not generate enough seeds to find them.

Comparison with S. Kim et al.'s Origin Analysis. For comparison, both our tool and KPW's tool were applied to *jEdit* and *ArgoUML*'s check-in snapshots. Table 5.5 and 5.6 shows the comparison result (γ=0.65 and ϵ=0.34). For *jEdit*, our approach finds 40% more matches yet reduces the result size by 22%. For *ArgoUML*, it finds 21% more matches yet reduces the result size by 44%.

We also compared our matches to KPW's matches and inspected the matches from 50 sample check-ins to measure precision. For *jEdit*, we found over 678 matches not identified by KPW's approach. KPW's approach found about 100 matches that we did not. When

combined with the precision of sampled matches, this means our approach found over 600 (=678×0.89) useful matches and that KPW's approach found about 75 (=99×0.75) useful matches. *ArgoUML* shows roughly similar results. This result is noteworthy because KPW's approach considers more information such as calling relationships as well as clone detection results in addition to name similarity. We suspect that it is because of KPW's approach's two limitations: It cannot accept correct matches when their overall similarity score is lower than a certain threshold, and it cannot easily prune incorrect matches once their overall similarity score is over a certain threshold and is the highest among potential matches. On the other hand, our algorithm tends to reject matches whose transformation is an isolated incident even if the similarity score is high. Our tool's incorrect matches usually come from bad seeds that coincidentally have similar names. Overall, our approach finds more matches without sacrificing its precision and represents results more concisely than KPW's approach.

5.3.3 Impact of Seed and Exception Thresholds

Seed Threshold. Our results in part depend on the quantity and quality of seeds. Figure 5.3 shows how our algorithm behaves when we change the seed generation threshold γ for *JFreechart* (0.9.4→0.9.5). We varied γ from 0.9 to 0.5 and measured recall of seeds, precision, recall, and the ratio of rejected seeds to the total number of seeds. When γ is set high in the range of 0.9 to 0.8, the name matching technique finds a relatively small number of seeds, but the seeds tend to be all good seeds. So our algorithm rejects very few seeds and leverages the good seeds to quickly reach recall of 0.65 to 0.85. However, the recall is still below 0.85 as the seeds do not contain enough transformations. As γ decreases, more seeds are produced, and a higher percentage of them are bad seeds that our algorithm later rejects. Using a low threshold (< 0.6) generally leads to higher recall (above 0.9) but it lowers precision and increases the running time since there are more candidate rules based on bad seeds. For the results in Figure 5.3, we observed a roughly linear increase from 6 minutes (γ=0.9) to 26 minutes (γ=0.5).

In general, when the precision and recall of seed matches are low, our algorithm improves

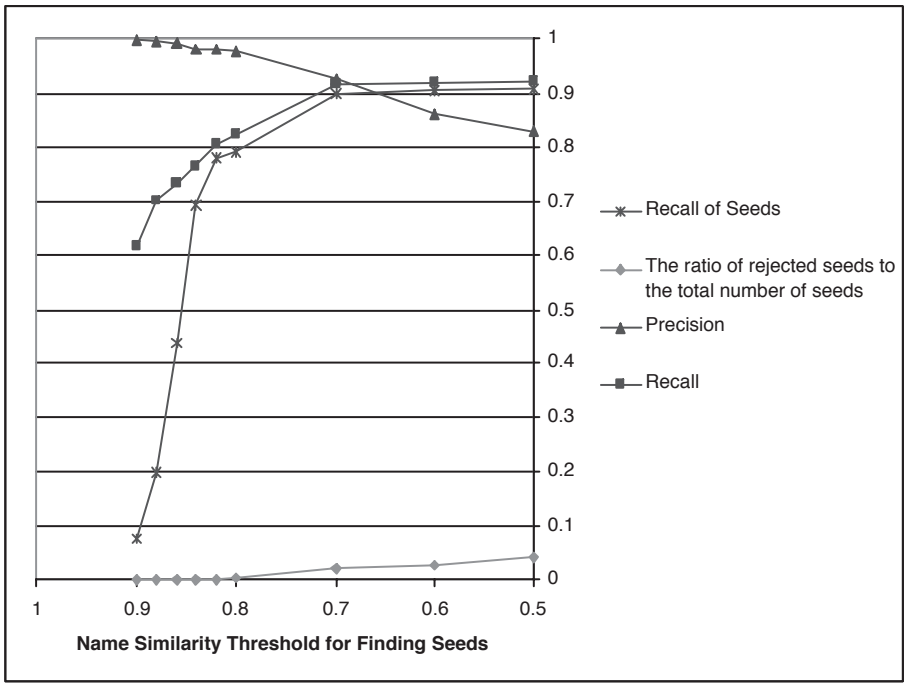

Figure 5.3: Impact of seed threshold γ

both measures significantly. When the seed matches already have precision and recall over 0.9, the algorithm still improves both measures, although less so because the seeds are already very good. However, even in this case, our algorithm significantly improves the conciseness measure. Effective seed generation and its interaction with our candidate rule selection algorithm needs additional research.

Exception Threshold. Table 5.7 shows the matching results with different exception thresholds: 0.25, 0.34, and 0.5. Using a low threshold increases running time and slightly decreases the M/R ratio. Surprisingly we found that changing exception thresholds does not affect precision and recall much. We suspect that it is because most exceptions come from deleted entities.

Table 5.7: Impact of exception threshold

Exception Threshold	Precision	Recall	M/R ratio	Running Time (Min)
0.25	0.94	0.91	14.12	10.44
0.34	0.94	0.91	14.14	9.19
0.50	0.93	0.91	14.33	7.54

All values are average values for *JFreeChart* data set.

5.3.4 Threats to Validity

To measure precision, we manually inspected the matches generated by our tool and by other tools. Manual labeling is subject to evaluator bias. All data are publicly available,[7] so other researchers can independently assess our results or use our data.

Our effort to date is limited in a number of ways. First, we have not explored other (non-Java) programming languages and even different naming conventions, all of which could have consequences. Second, we have not explored possible ways to exploit information from programming environments such as Eclipse that support higher-level refactoring operations.

5.3.5 Limitations

Our API change-rules are transformation *functions*. Thus, our approach cannot concisely capture the following scenarios that require mapping a single method-header in an old version to multiple method-headers in a new version: (1) An old, deprecated API is kept for backward compatibility purposes, and (2) An old method is split to two new methods, e.g., a modified old method and a new helper method. It is possible to mitigate this $1 : n$ matching multiplicity problem by running our tool backward and finding rules that transform a new program version to an old program version.

Our API change-rules do not leverage the name of input arguments. Thus, our algorithm could accidentally aggregate API-level changes that are syntactically homogeneous but semantically different. For example, when a programmer adds an `age` input argument

[7] www.cs.washington.edu/homes/miryung/matching

of type `int` to the `printAge()` method and also adds a `fileID` argument of type `int` to the `printFile()`, our approach could find the following rule that summarizes unrelated low-level changes.

- for all x:method-header in *.*.print*(*)

  argAppend(x, [int])

  [Interpretation: All methods with a name "*.*.print*(*)" appended an int argument.]

5.4 Summary of API Change-Rule Inference

Based on the insight that high-level changes are often systematic at a code level, we developed a rule-based change inference approach. The core of this approach is (1) a rule representation that explicitly captures systematic changes and (2) a corresponding inference algorithm that automatically finds such rules from two program versions.

In particular, this chapter describes an instantiation of this approach at a method-header level. Our comparative evaluation on open source projects shows that our approach produces 22 to 77% more concise results than other refactoring reconstruction tools. In terms of producing method-level matches, our approach has a high precision (median 98 to 99%) and recall (median 93 to 94%). The next chapter discusses how we extended this rule-based approach to discover systematic changes within method-bodies as well as at a field level.

Chapter 6

INFERRING CHANGES TO PROGRAM STRUCTURE

Software engineers often inspect program differences when reviewing others' code changes, when writing check-in comments, or when determining why a program behaves differently from expected behavior. For example, suppose Bill asks Alice, his team lead, to review his most recent software change. Bill's check-in message, *"Common methods go in an abstract class. Easier to extend/maintain/fix,"* suggests some questions to Alice: "Was it indeed an *extract superclass* refactoring?" "Did Bill make some other changes along the way?" and "Did Bill miss any parts of the refactoring?" Alice is left to answer these questions by a tedious investigation of the associated *diff* output, which comprises 723 lines across 9 files. She may need to check even more code, perhaps the entire code-base or at least some surrounding unchanged code, for potential missed updates.[1]

Existing program differencing approaches (Section 2.2) generally try to help programmers answer these kinds of high-level questions by returning numerous lower-level changes. In many cases, this collection of lower-level changes has a latent structure because a high-level change operation was applied to the program. Existing approaches do not identify regularities in code changes created by a high-level change and subsequently cannot detect inconsistency in code changes, leaving it to a programmer to discover potential bugs.

To complement existing differencing approaches, we designed Logical Structural Diff (LSDiff) that discovers a latent structure in low-level changes. LSDiff abstracts a program as code elements (packages, types, methods, and fields) and their structural dependencies (method-calls, field-accesses, subtyping, overriding, containment) based on two premises. First, programmers often look for structural information when inspecting program differences: which code elements changed and how their structural dependencies are affected by

[1]This scenario is based on a real example found in our evaluation (*carol* revision 430) and Ko et al.'s study [170].

the change. (See Section 6.3.) Second, code elements that change similarly together often share common structural characteristics such as using the same field or implementing the same interface. Finding such shared characteristics in changed code is useful for discovering systematic changes.

LSDiff infers logic rules to discover and represent systematic structural differences. Remaining non-systematic differences are output as logic facts. For the preceding scenario, the following LSDiff output can help Alice understand the rationale of Bill's change; Bill created the `AbsRegistry` class by pulling up `host` fields and `setHost` methods from the classes implementing the `NameSvc` interface; however, he did not complete the refactoring on the `LmiRegistry` class that also implements the `NameSvc` interface. In addition to the *extract superclass* refactoring, Bill made changes that seem to alter program behavior by deleting calls to the `SQL.exec` method. After reading this, Alice decided to double check with Bill why `LmiRegistry` was left out and why he deleted calls to the `SQL.exec` method.

Fact 1. added_type(AbsRegistry)

[Interpretation: AbsRegistry is a new class.]

Rule 1. past_subtype("NameSvc", t) ∧ past_field(f, "host", t) ⇒ deleted_field(f, "host", t) except t="LmiRegistry"

[Interpretation: All host fields in NameSvc's subtypes got deleted except LmiRegistry class.]

Rule 2. past_subtype("NameSvc", t) ∧ past_method(m, "setHost", t) ⇒ deleted_method(m, "setHost", t) except t="LmiRegistry"

[Interpretation: All setHost methods in NameSvc's subtypes got deleted except LmiRegistry class.]

Rule 3. past_subtype("NameSvc", t) ∧ past_method(m, "getHost", t) ⇒ deleted_calls(m, "SQL.exec") except [m="LmiRegistry.getHost",t="LmiRegistry"]

[Interpretation: All getHost methods in NameSvc's subtypes deleted calls to SQL.exec except LmiRegistry class ...]

The core idea of discovering and representing systematic changes is similar to Chapter 5's API change-rule inference. Specifically, LSDiff is an instantiation of our rule-based approach at the level of a program structure. LSDiff's rule representation and inference algorithm differ in several ways: First, the goal of API change-rule inference was to match method-headers across program versions to enable program analysis over multiple versions. LSDiff aims to discover a logical structure that relates line-level differences to complement existing

diff. Second, while API change-rules focus on changes above the level of method-headers, LSDiff accounts for changes within method-bodies as well as at a field level. Third, from a change representation perspective, while API change-rules rely on a regular expression to group related code elements, LSDiff uses conjunctive logic literals to allow programmers to understand shared structural characteristics of systematically changed code, not only a shared naming convention, e.g. "all `setHost` methods in `Service`'s subclasses" instead of "all methods with name `*Service.setHost(*)`." Finally, from a rule-inference technique perspective, our API change-rule inference algorithm finds rules in an open system because there is no ground truth for a code matching problem. On the other hand, LSDiff's algorithm learns rules in a closed system by first computing structural differences and then enumerating all rules within the rule search space set by the input parameters.

To evaluate LSDiff, we conducted a focus group study with professional software engineers in a large E-commerce company. The participants' comments show that LSDiff is promising both as a complement to *diff*'s file-based approach and also as a way to help programmers discover potential bugs by identifying exceptions to inferred systematic changes. We also compared our results with (1) structural differences that an existing differencing approach would produce at the same abstraction level for an evenhanded comparison and (2) textual deltas computed by *diff*. These quantitative assessments show that, on average, LSDiff produces 9.3 times more concise results by identifying 75% of structural differences as systematic changes. LSDiff finds an average of 9.7 more contextual facts that cannot be found in the delta such as `LmiRegistry`'s `host` field not being deleted.

The rest of this chapter is organized as follows. Section 6.1 and Section 6.2 describe how LSDiff represents and identifies structural differences. Section 6.3 discusses the focus group study. Section 6.4 describes quantitative and qualitative assessments. Section 6.5 discusses LSDiff's limitations and future work. Section 6.6 reinforces the benefits of inferred change-rules by demonstrating applications that can be built using our approach. Example change-rules are drawn from our empirical evaluation of LSDiff as well as API-level change inference. Section 6.7 concludes.

6.1 Definition of Logical Structural Delta

LSDiff represents each program version using a set of predicates that describe code elements and their dependencies. Some of the predicates (1-7 below) describe code elements and their containment relationships. The others describe structural dependencies (field-access, method-call, subtyping, and overriding). To support grouping of code elements with the same simple name, our predicates include both the simple and the fully qualified name. Table 6.1 shows the fact-base representation (a database of facts) of an example program. LSDiff captures only structural differences and does not capture differences in control logic, temporal logic, etc. Thus, unlike *diff*, its output cannot be used to reconstruct a new version by applying the delta to an old version. Appendix F describes the predicates in the Tyruba logic programming language [289].

1. package (packageFullName:string):

   There is a package `packageFullName`.

2. type (typeFullName:string, typeShortName:string, packageFullName:string):

   There is a class or an interface `typeShortName` in the `packageFullName` package.

3. method (methodFullName:string, methodShortName:string, typeFullName:string):

   There is a method `methodShortName` in the `typeFullName` type.

4. field (fieldFullName:string, fieldShortName:string, typeFullName:string):

   There is a field `fieldShortName` in the `typeFullName` type.

5. return (methodFullName:string, returnTypeFullName:string):

   The `methodFullName` method returns an object with the `returnTypeFullName` type.

6. fieldoftype (fieldFullName:string, declaredTypeFullName:string):

   The `fieldFullName` field is declared with the type `declaredTypeFullName`.

7. typeintype (innerTypeFullName:string, outerTypeFullName:string).

   The `innerTypeFullName` type is contained in the `outerTypeFullName` type.

8. accesses (fieldFullName:string, accessorMethodFullName:string).

   The `fieldFullName` is accessed by the `accessorMethodFullName` method.

9. calls (callerMethodFullName:string, calleeMethodFullName:string).

   The `callerMethodFullName` method calls the `calleeMethodFullName` method.

Table 6.1: A fact-base representation of two program versions and their differences

P_o (an old version)	P_n (a new version)	FB_n (a fact-base of the new version)	ΔFB

`class BMW implements Car`
`void start (Key c) {`
`...`
	`class BMW implements Car`
	`void start (Key c) {`
	`Key.chk (null); ...`
		subtype("Car","BMW"), ...
		method("BMW.start", "start", BMW)
		calls("BMW.start", "Key.chk")...
			
			+calls("BMW.start", "Key.chk")

`class GM implements Car`
`void start (Key c ) {`
`if (c.on) { ....`
`...`
	`class GM implements Car`
	`void start (Key c ) {`
	`Key.chk (c );`
	...
		subtype("Car","GM"), ...
		method("GM.start", "start", "GM")
		calls("GM.start", "Key.chk")
		...
			
			-accesses("Key.on", "GM.start")
			+calls("GM.start", "Key.chk")

`class Kia implements Car`
`void start (Key c ) {`
`c.on = true; ....`
	`class Kia implements Car`
	`void start (Key c ) {`
	`...`
		subtype("Car","Kia"), ...
		method("Kia,start", "start", "Kia")
		...
			
			-accesses("Key.on", "Kia.start")

`class Bus {`
`void start (Key c) {`
`c.on = false;} }`
	`class Bus {`
	`void start (Key c);`
	`log(); } }`
		type("Bus")
		method("Bus,start", "start", Bus)
		calls("Bus.start", "log")
			
			-accesses("Key.on", "Bus.start")
			+calls("Bus.start", "log")

`class Key {`
`boolean on = false;`
`void chk (Key c) { ...`
`void out () { ...`
	`class Key {`
	`boolean on = false;`
	`void chk (Key c) {`
	`void output (Key c){ ...`
		type ("Key")
		field("Key.on", "on", "Key")
		method ("Key.chk", "chk", "Key")
		method ("Key.output", "output", "Key") ...	

• For presentation purposes, fully qualified names are shortened, and the added and deleted facts in ΔFB are noted with + and − sign respectively. FB_o is omitted to save space; it can be inferred based on FB_n and ΔFB.

Table 6.2: LSDiff rule inference example

$\Delta FB'$	$\Delta FB''$
1. past_accesses("Key. on", m) ⇒ deleted_accesses("Key.on", m) 2. added_calls("BMW.start", "Key.chk") 3. added_calls("GM.start", "Key.chk") 4. added_calls("Bus.start", "log")	1. past_accesses("Key. on", m) ⇒ deleted_accesses("Key.on", m) 2. past_method(m, "start", t) ∧ past_subtype("Car",t) ⇒ added_calls(m, "Key.chk") except t = Kia 3. added_calls("Bus.start", "log")

10. subtype (superTypeFullName:string, subTypeFullName:string).

The `subTypeFullName` type either inherits or implements the type `superTypeFullName`.

11. inheritedfield (fieldShortName:string, superTypeFullName:string, subTypeFullName:string).

The `subTypeFullName` type inherits the `fieldShortName` field from the `superTypeFullName` type.

12. inheritedmethod (methodShortName:string, superTypeFullName:string, subTypeFullName:string).

The `subTypeFullName` type inherits the `methodShortName` method from the `superTypeFullName` type. [2]

Applying a set-difference operator to the fact-base of an old version (FB_o) and the fact-base of a new version (FB_n) produces structural differences, ΔFB. (See Table 6.1.) However, using ΔFB as a program delta has two weaknesses. First, because it lists fact-level differences without any high level structure, it is time-consuming to read and understand when it contains a large number of facts. Second, it describes only the structural dependencies in changed code fragments but not those of their surrounding context. For example, suppose that a program change involves removing all accesses to `Key.on` and invoking the `Key.chk` method from `Car`'s subtypes' `start` methods. ΔFB lists the three deleted accesses facts separately and does not include contextual information that new method-calls to `Key.chk` occurred in the context of `Car`'s subtypes.

Our approach overcomes these two weaknesses by inferring logic rules from the union of

[2]It is possible to add more predicates or modify existing predicates. For example, one can add throwException(methodFullName:string, typeName:string) to model exception handling or modify the type, method and field predicates to model changes to their visibility: `public, private` and `protected`.

all three fact-bases: FB_o, FB_n, and ΔFB. To distinguish which fact-base each fact belongs to, we prefix past_ and current_ to the facts in FB_o and FB_n respectively and deleted_ and added_ to the corresponding facts in ΔFB. Inferring rules from all three fact-bases has two advantages: First, our rule-based delta is concise because a single rule can imply a number of related facts. Second, by inferring rules from not only the delta but also from unchanged code, our approach finds contextual facts such as subtype("Car", "Kia"), which is not in ΔFB but signals a potential missed update, +calls("Kia.start", "Key.chk").

LSDiff Rule. Just as logic rules describe the relationship among groups of related logic facts, LSD rules describe high-level systematic changes by relating groups of facts in the three fact-bases.

To represent a group of similar facts at once, we create a logic literal by binding some of a predicate's arguments to variables. For example, subtype("Foo", t) represents all subtype facts that have Foo as a first argument.

Rules relate groups of facts by connecting literals with boolean logic operators. In particular, LSDiff rules are horn clauses where the conjunction of one or more literals in the antecedent implies a single literal in the conclusion, i.e., $A(x) \wedge B(x,y) \ldots \wedge C(x,z) \Rightarrow D(x,z)$. LSDiff rules are in the subset of Datalog queries, which ensures termination and fast evaluation.[3] All variables are universally quantified and variables do not appear in the conclusion unless they are bound in the antecedent. LSDiff Rules are either ungrounded rules (rules without constant bindings) or partially grounded rules (rules with constant bindings).

A rule r has a **match** f in ΔFB if f is a fact created by grounding r's conclusion with constants that satisfy r's antecedent given FB_o, FB_n, and ΔFB. A rule r has an **exception** if there is no match in ΔFB implied by a true grounding of its antecedent. For example, a rule $A(x) \Rightarrow B(x)$ has a match $B(c_1)$ and an exception $x{=}c_2$ if $A(c_1)$, $A(c_2)$, and $B(c_1)$ are in the three fact-bases, but $B(c_2)$ is not in ΔFB. We explicitly encode exceptions as a part of a rule to note anomalies to a systematic change.

LSDiff supports a restricted set of Datalog rule styles. The styles are shown in Table 6.3: Only deleted_* or added_* can appear in the consequent of a rule. The antecedent of

[3]Datalog is a subset of Prolog that ensures termination. Because of its good run-time efficiency compared to full Prolog, Datalog is argued to be a good choice for program queries [115].

a rule cannot have predicates with different prefixes. These rule styles force LSDiff to learn regularities about changes between the two versions, not the regularities in the old or new version itself. These rule styles are effective in expressing general kinds of systematic changes, including:

• dependency removal or feature deletion by stating that all code elements with similar characteristics in the old version were removed. (e.g., past_* ⇒ deleted_*),

• consistent updates to clones by stating that all code elements with similar characteristics in the old version added similar code (e.g., past_* ⇒ added_*),

• replacement of API usages by relating deletions and additions of dependencies (e.g., deleted_* ⇒ added_*),

• feature addition by stating that all code elements with particular characteristics in the new version are added by the change (e.g., current_* ⇒ added_*), etc.

Example. Suppose that a programmer removes all accesses to the `Key.on` field and adds calls the `Key.chk` method from `Car`'s subtypes' `start` methods. Table 6.1 presents the fact-bases and Table 6.2 shows the rule inference and ΔFB reduction process. By inferring a rule, "all accesses to the Key.on field are removed from the old version (#1 in ΔFB′)," ΔFB is reduced to ΔFB′ by replacing the three deleted_accesses facts with the rule. By inferring a rule, "all Car's subtypes' start methods added calls to the Key.chk method (#2 in ΔFB″)," ΔFB′ is reduced to ΔFB″ by winnowing out the two added_calls facts. This rule also signals inconsistency that `Kia` did not change similarly.

The remaining added_calls("Bus.start", "log") is output as is, because it does not form a systematic change pattern.

6.2 Inference Algorithm

Our algorithm accepts two versions of a program and outputs a logical structural delta that consists of logic rules and facts. Our algorithm has three parts: (1) generating fact-bases, (2) inferring rules from the fact-bases, and (3) post-processing the inferred rules.

Part 1. Fact-base Generation. We create FB_o and FB_n from the old and new version respectively by extracting logic facts using JQuery [144], a logic query-based program investigation tool. JQuery analyzes a Java program using the Eclipse JDT Parser; thus,

Table 6.3: LSDiff rule styles and example rules

Rule Styles Antecedent ⇒ Conclusion	High-Level Change Patterns	Example Rule and Its Interpretation
past_* ⇒ deleted_*	dependency removal, feature deletion, etc.	past_calls(m, "DB.exec") ⇒ deleted_calls(m, "DB.exec") All methods that called `DB.exec` in the old version deleted a call dependency to `DB.exec`.
past_* ⇒ added_*	consistent maintenance, etc.	past_accesses("Log.on",m) ⇒ added_calls(m, "Log.trace") All methods that accessed `Log.on` in the old version added a call dependency to `Log.trace`.
current_* ⇒ added_*	dependency addition, feature addition, etc.	current_method(m, "getHost", t) ∧ current_subtype("Svc", t) ⇒ added_method(m, "getHost", t) All `getHost` methods in the `Svc`'s subclasses are newly added ones.
deleted_* ⇒ added_* added_* ⇒ deleted_*	related code change, API usage change, etc	deleted_method(m, "getHost", t) ⇒ added_inheritedfield("getHost", "Service", t) All types that deleted `getHost` method inherit `getHost` from `Service` instead.

its precision depends on the Eclipse's static analysis capability. We then compute ΔFB using a set-difference operator and remove spurious added_ and deleted_ facts caused by code renaming or moving using inferred method-header level refactorings from Chapter 5. Note that ΔFB in Table 6.1 does not contain −method("Key.out", ...) and +method("Key.output", ...) by accounting for the renaming.

Part 2. First Order Logic Rule Learning. Our goal is to infer rules each of which corresponds to a high-level systematic change and thus explains a group of added_ and deleted_ facts. This step takes the three fact-bases and outputs inferred rules and remaining unmatched facts in ΔFB. Some rules refer to groups of past_ and current_ facts, providing structural characteristics about changed code that cannot be found in ΔFB only.

Three input parameters define which rules to be considered in the output: (1) m, the minimum number of facts a rule must match, (2) a, the minimum accuracy of a rule, where accuracy = # matches / (# matches + # exceptions), and (3) k, the maximum number of literals in a rule's antecedent. A rule is considered **valid** if the number of matches and exceptions is within the range set by these parameters.

Our rule learning algorithm is a bounded-depth search algorithm that enumerates rules up to a certain length. The depth is determined by k. Increasing k allows our algorithm to find more contextual information from FB_o and FB_n. Evaluating all possible rules with k literals in the antecedent has the same effect as examining surrounding contexts that are roughly k dependency hops away from changed code fragments. Our algorithm enumerates rules incrementally by extending rules of length i to create rules of length $i + 1$. In each iteration, we extend the ungrounded rules from the previous iteration by appending each possible literal to the antecedent of the rules. Then for each ungrounded rule, we try all possible constant substitutions for its variables. After selecting valid rules in this iteration, we winnow out the selected rules' matches from U (a set of unmatched facts in ΔFB) and proceed to the next iteration.

Some rules are always true regardless of change content and do not provide any specific information about code change. For example, deleting a package deletes all contained types in the package, and deleting a method implies deleting all structural dependencies involving the method. To prevent learning such rules, we have written 30 *default winnowing rules* by

hand and winnow out the facts from U in the beginning of our algorithm.

For the rest of this section, we explain two subroutines in detail: (1) extending ungrounded rules from the previous iteration and (2) generating a set of partially grounded rules from an ungrounded rule. Then we discuss a beam search heuristic that we use to tame the exponential growth of the rule search space. Our rule inference algorithm is summarized in Algorithm 6.

Subroutine 1. Extending Ungrounded Rules. For each ungrounded rule from the previous iteration, we identify all possible predicates that can be appended to its antecedent. For each of those predicates, we create a set of candidate literals by enumerating all possible variable assignments. After we create a new rule by appending each candidate literal to the ungrounded rule's antecedent, we check two conditions: (1) we have not already generated an equivalent rule, and (2) it matches at least m facts in U. If the rule has fewer than m matches, we discard it because adding a literal to its antecedent or grounding its variables to constants can find only fewer matches. If the two conditions are met, we add the ungrounded rule to the list of new ungrounded rules to try constant substitutions for its variables and to pass to the next iteration. The pseudo code of this subroutine is described on page 144.

Subroutine 2. Generating Partially Grounded Rules. To create partially grounded rules from an ungrounded rule, we consider each variable in turn and try substituting each possible constant for it as well as leaving it alone. At each step within this process, we evaluate the rule to check how many matches it finds in U. If it finds fewer than m matches, we discard the rule and do not explore further substitutions, as more specific rules can find only fewer matches than m. The pseudo code of this subroutine is described on page 145.

Beam Search. As the size of the rule search space increases exponentially with the number of variables in ungrounded rules, enumerating rules quickly becomes infeasible for longer rules. To tame this exponential growth, we use a beam search heuristic: in each iteration, we save only the best β number of ungrounded rules and pass them to the next iteration. The beam search is a widely used heuristic in first order logic rule learning [171]. As our tests found no improvement when β was increased beyond 100, we used this as a default. To select the best β rules, we first rank rules by their number of matches. When there's a tie, we prefer rules with fewer number of exceptions, as these rules are worth refining

Algorithm 6: LSDiff Rule Inference Algorithm

Input:

FB_o /* a fact-base of an old program version */

FB_n /* a fact-base of a new program version */

ΔFB /* fact-level differences between FB_o and FB_n */

m /* the minimum number of facts a rule must match to be selected */

a /* the minimum accuracy of a rule */

k /* the maximum number of literals in a rule's antecedent */

β /* beam search window size */

Output:

L /* a set of valid learned rules */

/* Initialize R (a set of ungrounded rules), L, and U (a set of facts in ΔFB that are not covered by L). */

R := $\emptyset$, L := $\emptyset$, U := ΔFB;

U := reduceDefaultWinnowingRules (ΔFB, FB_o, FB_n) ;

/* reduce ΔFB using default winnowing rules. */

foreach $i = 0 \ldots k$ **do**

  if *($i = 0$)* **then**

    R := createInitialUngroundedRules (m) ; /* create ungrounded rules with an empty antecedent by enumerating all possible consequents. */

  else

    R := extendUngroundedRules (R) ; /* extend all ungrounded rules in R by adding all possible literals to their antecedent. */

  foreach $r \in$ R **do**

    G := createPartiallyGroundedRules (r) ; /* try all possible constant substitutions for r's variable. */

    foreach g *in* G **do**

      if isValid *(g)* **then**

        L :=L $\cup$ {g};

        U :=U $-$ {g.matches};

  R :=selectRules (R, β) ; /* select the best β rules in R */

end

```
Function createInitialUngroundedRules
  conclusions = ∅;
  /* for each predicate that could be in the conclusion,                    */
  foreach p ∈ DELTA_PREDICATES do
      /* create a literal l by instantiating p with new variable bindings.  */
      /* create a new rule r and set its consequent to l                    */
      l:= createLiteral(p, freshvariables());
      r:= new Rule();
      r.setConsequent(l);
      if |r.matches| ≥ m then
          conclusions:= conclusions ∪ {r};
      end
  end
  return conclusions;
```

further. If there is still a tie, we prefer rules whose variables are more general in terms of Java containment hierarchy: package > type > field = method > name.

Part 3. Post Processing. Rules with the same length may still have overlapping matches after Part 2. To avoid outputting rules that cover the same set of facts in the ΔFB, we select a subset of the rules using the greedy version of the SET-COVER algorithm [14]. In this step, we use the same ranking order as in our beam search. We then output the selected rules and the remaining unmatched facts in ΔFB.

6.3 Focus Group Study

To understand our target users' perspectives on LSDiff, we conducted a focus group study with professional software engineers from a large E-commerce company. We selected this study method for several reasons. First, *LSDiff* is a prototype tool in an early stage and we need to assess its potential benefits before investing our efforts in user interface development. A focus group study fits our needs, as it is typically carried out in an early stage of product development to gather target users' opinions on new products, concepts, or messages. Second, we needed to develop concrete hypotheses about how programmers use *diff*

Function `extendUngroundedRules(`R`)`

```
NR := ∅;
foreach r ∈ R do
    /* for each antecedent predicate candidate,                          */
    foreach p ∈ ANTECEDENT_PREDICATES do
        bindings := enumerateBindingsForPredicate (r, p);   /* enumerate variables
        bindings for the predicate p.  */
        foreach b ∈ bindings do
            /* create a new antecedent literal l by instantiating p with b.   */
            /* create a new rule by copying r and add l to the new rule's
               antecedent.                                                    */
            r:= new Rule(r);
            r.addAntecedentLiteral(l);
            if |r.matches| ≥ m ∧ !(r ∈ NR) then
                NR := NR ∪ {r};
            end
        end
    end
end
return NR;
```

Function `createPartiallyGroundedRules(`r`)`

```
NR:= ∅;
S = new Stack() ;                          /* a stack of partially-grounded rules */
S.push(r) ;            /* add the initial ungrounded rules with no constants.  */
/* for each head of the stack,                                                  */
while !S.isEmpty() do
    pr = S.pop();
    foreach variable ∈ pr.remainingVariables() do
        constants := getReplacementConstants(pr, variable);     /* use Tyruba engine to
        find all possible constant substitutions for the variable.  */
        foreach constant ∈ constants do
            n = substitute(pr, variable, constant);          /* create a new copy of pr,
            substitute the corresponding variable with constant */
            if |n.matches| ≥ m ∧ accuracy(n) ≥ a then
                NR := NR ∪ {n };
            end
            if n.remainingVariables.size() > 0 then
                S.push(n);    /* add the new rule to the stack to continue constant
                substitutions.  */
            end
        end
    end
end
return NR;
```

Table 6.4: Focus group participant profile

	Title	Years in SW Industry	*diff* or P4	code change review
P1	SDE	10+	Daily	Daily
P2	Senior SDE	14+	Daily	Daily
P3	Senior Principal SDE	30+	Daily	Weekly
P4	SDET	6+	Weekly	Daily
P5	SDET	6+	Weekly	Daily

and what are the difficulties of using *diff* for understanding code changes. A focus group is good for developing hypotheses that can be further tested by quantitative studies such as survey and experiments. Third, compared to other qualitative study methods, a focus group has a relatively low cost.

The goal of the focus group was to answer: (1) In which task contexts do programmers need to understand code changes? (2) What are difficulties of using program differencing tools such as *diff*? and (3) How can LSDiff complement existing uses of program differencing tools?

With the help of a liaison at the company, we identified a target group consisting of software development engineers (including those in testing), technical managers, and software architects. A screening questionnaire asked the target group about their programming and software industry experience, their familiarity with Java, how frequently they use *diff* and *diff*-based version control systems, and the size of code bases that they regularly work with. Appendix I describes the screener questionnaire. All sixteen participants responded to the questionnaire and five out of them attended the focus group: each had primary development responsibilities; each had industry experience ranging from 6 to over 30 years; each used related tools at least weekly; and each reviewed code changes daily except one who did only weekly. Table 6.4 shows their profile.

Once the focus group participants were targeted, we created a discussion guide to allow a moderator to thoroughly cover all the necessary topics and questions. The discussion guide consists of a 5 minute introduction, a 10 minute discussion on the current practice of using *diff*, a 10 minute introduction and demonstration of *LSDiff*, a 5 minute initial evaluation of

LSDiff, and a 10 minute hands-on trial of reviewing a sample *LSDiff* output followed by a 15 minute in-depth evaluation of *LSDiff*. Appendix J shows the entire discussion guide.

The hands-on trial used a sample LSDiff output on *carol* project revision 430.[4] We chose this change because it is a conceptually simple change based on dispersed textual modifications of 723 lines across 9 files. LSDiff identified the systematic nature of the change, inferring 12 rules and 7 facts.

We used CSDiff[5] to prepare a regular word-level differencing result as an HTML document, in which each modified file is presented as a hyperlink to the new version's source file, deleted words are presented with red strike-through, and added words are highlighted in yellow. We then manually augmented the HTML *diff* document by creating an overview of systematic changes using the inferred rules. Each inferred rule was directly translated to an English sentence and presented as a hyperlink. (See Figure 6.1.) Upon clicking the hyperlink, a programmer can see the rule's accuracy, which code elements support the rule, and which code elements violate the rule. (See below.)

"All `host` fields in the classes that implement `NameService` interface got deleted except in the `LmiRegistry` class."

past_subtype("NameService",t) ∧ past_field(f,"host",t)

⇒ deleted_field(f,"host",t) except t="LmiRegistry"

Accuracy: (5/6)

deleted_field("CmiRegistry.host","host","CmiRegistry")

deleted_field("IIOPCosNaming.host","host","IIOPCosNaming")

deleted_field("JRMPRegistry.host","host","JRMPRegistry")

deleted_field("JacCosNaming.host","host","JacCosNaming")

deleted_field("JeremieRegistry.host","host","JeremieRegistry")

Exception: [t="LmiRegistry", f="LmiRegistry.host"]

In addition, by clicking each match, a programmer can navigate to corresponding word-level differences. (See Figure 6.2.) When the word-level differences are a part of systematic

---

[4] http://www.cs.washington.edu/homes/miryung/LSDiff/carol429-430.htm

[5] http://www.componentsoftware.com/Products/CSDiff/

Carol Revision 430.
SVN check-in message: Common methods go in an abstract class. Easier to extend/maintain/fix
Author: benoif @ Thu Mar 10 12:21:46 2005 UTC
723 lines of changes across 9 files (2 new files and 7 modified files)

Inferred Rules		
1	(50/50)	By this change, six classes inherit many methods from AbsRegistry class.
2	(32/32)	By this change, six classes implement NameService interface.
3	(6/8)	All methods that are included in JacORBCosNaming class and NameService interface are deleted except start and stop methods.
4	(5/6)	All host fields in the classes that implement NameService interface got deleted except LmiRegistry class.
5	(5/6)	All port fields in the classes that implement NameService interface got deleted except LmiRegistry class.
6	(5/6)	All getHost methods in the classes that implement NameService interface got deleted except LmiRegistry class.
7	(5/6)	All getPort methods in the classes that implement NameService interface got deleted except LmiRegistry class.
8	(5/6)	All setConfigProperties methods in the classes that implement NameService interface got deleted except LmiRegistry class.
9	(5/6)	All setHost methods methods in the classes that implement NameService interface got deleted except LmiRegistry class.
10	(5/6)	All setPort methods in the classes that implement NameService interface got deleted except LmiRegistry class.
11	(3/3)	All configurationProperties fields got deleted.
12	(3/4)	All DEFAULT_PORT_NUMBER fields are added by this change except JacORBCosNaming class.

Remaining Change Facts	
Added Class	AbsRegistry
Added Class	DummyRegistry
Added Method	JRMPRegistry.getRegistry
Deleted Field	IIOPCosNaming.DEFAUL_PORT
Deleted Field	JacORBCosNaming.started
Added Field Access	CmiRegistry's constructor added accesses to ClusterRegistry.DEFAULT_PORT field.
Added Field Access	JacORBCosNaming's constructor added accesses to JacORBCosNaming.DEFAULT_PORT_NUMBER field.

Figure 6.1: Overview based on *LSDiff* rules

changes, the corresponding rule description is inserted as a hyperlink so that a programmer can navigate to other related code changes. (See line 49 and 50 in Figure 6.2.)

During the focus group, I worked as the moderator of the focus group discussion. We audio-taped the discussion and had a note-taker transcribe the conversation. Appendix K shows the transcript of the focus-group discussion.

Our key findings are organized by the questions asked by the moderator.

When do programmers use diff? Programmers often use *diff* when reviewing other engineers' code changes or when resolving a problem report. When the program's execution behavior is different from their expectation or when investigating unfamiliar code, programmers examine the *evolutionary context* of the involved code: how the code changed over time and why it was changed.

"The one that comes up the most frequently is a code review. . . Multiple times a day, someone makes changes and sends them out so that everybody can see it."

"In troubleshooting, you get an error and I think the code should be doing this. . . , this variable is not being set or they did not anticipate these situations. 'Is it that I got a bad input?' or 'Are they not handled correctly or what?' . . . The only documentation you have is the code that you are staring at right there. So you wanna know how the code got to the state that it is at. "

". . . When I'm troubleshooting and trying to figure out what's wrong with a piece of code, usually I'd like to know some context about when it changed, cause when something broke on a certain day, it is nice to find out about what changed at that time."

". . . You need to see generational changes; not just this file and that file but how it has changed over time . . . how it went through a series of change motivations. . . "

"It's hard to change something without knowing how it evolved and it is in the state that it is at."

What would you like to have in an ideal program differencing tool? Programmers would like to see program-wide, explicit, semantic relationships between different changed files. Many complained that *diff*'s file-based organization is inadequate for rea-

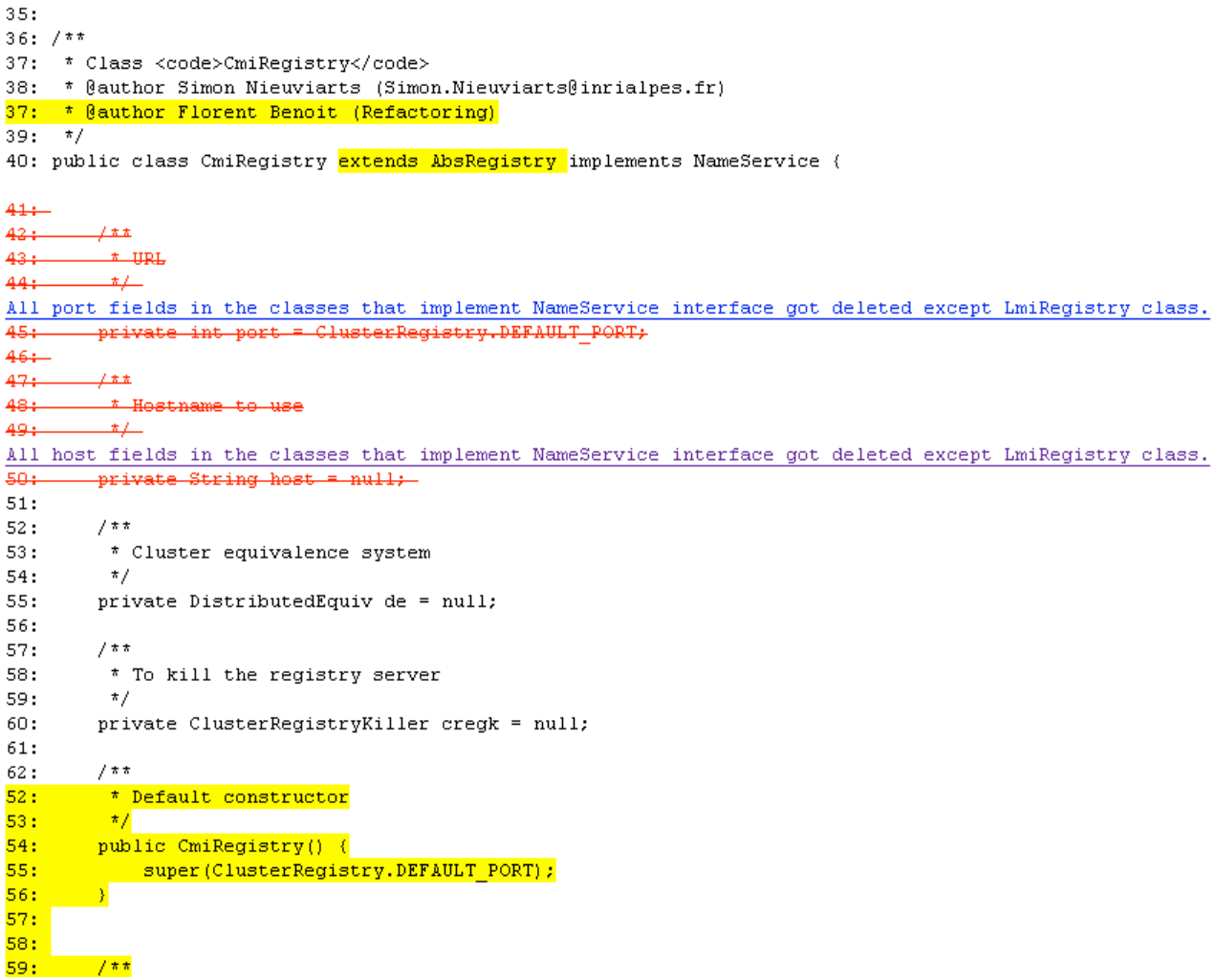

Figure 6.2: Sample HTML *diff* output augmented with *LSDiff* rules

soning about related changes. Though organizing changes based on containment hierarchy information—for example, Eclipse *diff*'s tree view—is useful to some degree, they believe it is still inadequate for global changes such as a refactoring that affects multiple files.

"The *diff* tools that I use, they are all file-oriented. They don't have notions, which I think you are trying to address is that, they don't have semantic relationships between different files. I want to say 'What did I change due to this problem?' It might have changed over 300 different files. I'd like to see not just one file but all 300 files that were included as a part of that. It is scaling up from a single source file to into spacing in which correlated change took place."

"... You may want to group similar methods together... *Diff* doesn't help with that and if a tool was aware of that, that would be great. Not just simple methods added here and there."

"Let's say that somebody refactored something, and they took a big chunk of code and moved it from this file to that file. Looking at this one file, you have no idea about its history and how it evolved. It evolved over here and then (it) got cut over here and pasted over here. So it's like you have no idea, and you have just lost all the contexts. "

"P4(Perforce) seems to be smart about language-level diffs. It's not like a typical *diff* where it is just a line-level ... especially when you are doing a merge, it figures out, 'this is a method encapsulated here, not just a collection of lines.' It's not perfect about recognizing related changes and it would be nice if it does better."

In general, the participants thought explicitly representing code elements is important. Their questions often focused on whether LSDiff accounts for Java language syntax.

"Does it use type information?"

"There goes to my scoping question. All the `ints` go to `longs` in a particular class, or a method, or a package?"

In which task contexts would you use *LSDiff*? The participants believed that LSDiff can be used in the situations where they are already using *diff* such as code reviews, in particular, when there is a large amount of changes. One testing engineer said he would like to use LSDiff to understand the evolution of the component that he is writing test cases for [62, 270].

"This is definitely good for code review. It gives you a lot more context behind the actual

change."

"This is definitely a winner tool. It lets you do things that would be so tough to do with diff that you don't even try."

"I write tests for the new XXXXXX SDK, an E-commerce platform SDK. They released a PR1 and now a PR2. We wrote all our tests against PR1 and now we have to move them to PR2. How do we figure out those differences? Specifically with testing, this is where this can be really powerful. You don't have to go by line by line... This will make the tester's time much more efficient."

Strengths of LSDiff The participants believed that LSDiff's ability to discover exceptions can help programmers find missing updates and better understand design decisions.

"This 'except' thing is great, because there's always the situation that you are thinking, 'why is this one different?'"

"I think a lot of times you're going to have 50 and 50. These 4 were changed and these 4 weren't. It just gives you more context. It just tells you that this thing is different for some reason. You can't infer the intent of a programmer, but this is pretty close..."

"Tell me if this tool does, it would be really useful if 9 out of 10 got changed, but one got forgotten. It doesn't know it got certainly forgotten, but with a high probability that this instance is kinda against the other ones. In fact, it is a missing change."

"If I'm going to assume that this was a correct change, it might be interesting for me to look at the the exceptions and contrast with those. Then I would understand better why these ones need to change. It's just more context."

The participants thought that the change overview based on the inferred rules would reduce change investigation time. Programmers can start from rules and drill down to details in a top down manner as opposed to reading changed-lines file by file without having the context of what they are reading about.

"I guess it is much a higher level of abstraction... You may start with the summary of changes and dive down to detail using a tool like *diff*. *Diff* will print out details and this will give you overall things. It is complementary in different levels. "

"This and *diff* have a very little overlap actually. Because this is a different level of abstrac-

tion, so this differencing is contextual. It is much more complementary to *diff*. It gives you condensed information. "

"And then when you click through to drill down, you know what you're looking at."

"You know what to expect. You can minimize the time that you are looking at code."

"If I am following a code base, I'd like to read the change list or read something like this to see how it's changed."

Limitations of LSDiff The participants were concerned that LSDiff does not identify cross-language systematic changes such as changing a Java program and subsequently changing XML configuration files. Some were concerned that LSDiff would not provide much additional benefits for non-systematic, random, or small changes and that LSDiff may find uninteresting systematic changes. For example, "all newly added constructors are contained in the types with a toString() method" is a valid systematic pattern but may be uninteresting to programmers.

"LSDiff has some awareness of what the file has in it. And you cannot do that with config files. In fact, I think this would be a great improvement."

"This will look for relationships that do not exist."

"This wouldn't be used if you were just working with one file. If you don't have rules about the structure of the file, it does not make sense to use it."

"This looks great for big architectural changes but I am wondering what it would give you if you had lots of random changes."

Overall, our focus group participants were very positive about LSDiff and asked us when they can use it for their work. They believed that LSDiff can help programmers reason about related changes effectively and it can allow top-down reasoning of code changes, as opposed to reading *diff* outputs without having a high-level context.

"This is cool. I'd use it if we had one."

"This is a definitely winner tool."

6.4 Assessments

Subject Programs. We applied LSDiff to two open source projects, *carol* and *dnsjava*, and to LSDiff itself. We selected these programs because their medium code size (up to 30 KLOC) allowed us to manually analyze changes in these programs in detail. *Carol* is a library that allows clients to use different remote method invocation implementations. From its version control system, we selected 10 version pairs with check-in comments that indicate non-trivial changes. Its size ranged from 10800 LOC to 29050 LOC and from 90 files to 190 files. *Dnsjava* is an implementation of domain name services in Java. From its release archive, we selected 29 version pairs. Its program size ranged from 5080 LOC to 14500 LOC and from 40 files to 83 files. We also selected LSDiff's first 10 versions pairs—revisions that are at least 8 hours apart and committed by different authors. Its program size ranged from 15651 LOC to 16897 LOC and from 93 files to 101 files.

6.4.1 Comparison with Structural Delta

We compared LSDiff's result (LSD) with ΔFB because ΔFB represents what an existing program differencing approach would produce at the same abstraction level. The goal of this comparison is to answer the following questions:

(1) How often do individual changes form systematic change patterns? LSDiff is based on the observation that high-level changes are often systematic at a code level. To understand how often this observation holds true in practice, we measured *coverage*, the percentage of facts in ΔFB explained by inferred rules: # of facts matched by rules / ΔFB. For example, when 10 rules explain 90 facts out of 100 facts in ΔFB, the coverage of rules is 90%.

(2) How concisely does LSDiff describe structural differences by inferring rules in comparison to an existing differencing approach that computes differences without any structure? We measured *conciseness* improvement: ΔFB / (# rules + # facts). For example, when 4 rules and 16 remaining facts explain all 100 facts in ΔFB, LSD improves conciseness by a factor of 5.

(3) How much contextual information does LSDiff find from unchanged code fragments? We believe that analyzing the entire snapshot of both versions instead of only deleted and

Table 6.5: Comparison with ΔFB

	FB_o	FB_n	ΔFB	Rule	Fact	Coverage	Conciseness	Additional Facts
Carol								
Min	3080	3452	15	1	3	59%	2.3	0.0
Max	10746	10610	1812	36	71	98%	27.5	19.0
Median	9615	9635	97	5	16	87%	5.8	4.0
Avg	8913	8959	426	10	20	85%	9.9	5.5
dnsjava								
Min	3109	3159	4	0	2	0%	1.0	0.0
Max	7200	7204	1500	36	201	98%	36.1	91.0
Median	4817	5096	168	3	24	88%	4.8	0.0
Avg	5144	5287	340	8	37	73%	8.4	14.9
LSDiff								
Min	8315	8500	2	0	2	0%	1.0	0.0
Max	9042	9042	396	6	54	97%	28.9	12.0
Median	8732	8756	142	1	11	91%	9.8	0.0
Avg	8712	8783	172	2	17	68%	11.2	2.3
Median	6650	6712	132	2	17	89%	7.3	0.0
Avg	6632	6732	302	7	27	75%	9.3	9.7

added text can discover relevant contextual information, reducing a programmer's burden of examining code that surrounds deleted or added text. We measured how many *additional facts* LSDiff finds by analyzing all three fact-bases as opposed to only ΔFB: # facts in FB_o and FB_n that are mentioned by the rules but are not contained in ΔFB. For example, the second rule in Table 6.2 refers to three additional facts subtype("Car","BMW"), subtype("Car", "GM") and subtype("Car", "Kia").

Table 6.5 shows the results for the three data sets with the default parameter settings m=3, a=0.75, k=2. (Section 5.3.4 describes how varying these parameters affects the results.) On average, 75% of facts in ΔFB are covered by inferred rules; this implies that 75% of structural differences form higher-level systematic change patterns. Inferring rules improves the conciseness measure by a factor of 9.3 on average. LSDiff finds an average of 9.7 more facts than ΔFB.

Table 6.6: Comparison with textual delta (1)

	Textual Delta							LSD	
	Files				CLOC	Hunk	%	Rule	Fact
Version	+	-	X	Total			Touched		
carol (carol.objectweb.org)									
62-63	7	1	13	21	2151	44	19%	12	71
128-129	0	0	10	10	164	11	7%	1	4
289-290	0	0	1	1	67	9	1%	2	3
387-388	0	0	12	12	528	107	7%	7	21
388-389	0	0	12	12	90	31	7%	3	4
421-422	3	0	11	14	4313	131	7%	36	30
429-430	2	0	7	9	723	71	4%	12	7
480-481	6	4	25	35	3032	132	17%	24	29
547-548	1	0	5	6	90	11	3%	1	10
576-577	4	2	4	10	1133	27	4%	1	25
MED	**2**	**0**	**11**	**11**	**626**	**38**	**7%**	**5**	**16**
AVG	**2**	**1**	**10**	**13**	**1229**	**57**	**8%**	**10**	**20**
LSDiff									
3-4	2	0	6	8	747	33	7%	3	23
4-13	5	0	5	10	563	13	8%	0	13
13-20	1	0	5	6	276	10	5%	0	8
20-21	0	5	6	11	637	37	9%	6	19
21-26	0	0	6	6	60	10	6%	1	6
26-27	0	0	3	3	31	3	3%	0	0
27-28	0	0	2	2	96	17	2%	0	2
28-34	0	0	4	4	178	28	4%	1	54
34-36	1	0	7	8	344	39	8%	2	8
36-39	0	0	2	2	9	2	2%	0	0
MED	**0**	**0**	**5**	**6**	**227**	**15**	**6%**	**1**	**8**
AVG	**1**	**1**	**5**	**6**	**294**	**19**	**5%**	**1**	**13**

Table 6.7: Comparison with textual delta (2)

	Textual Delta							LSD	
	Files				CLOC	Hunk	%	Rule	Fact
Version	+	-	X	Total			Touched		
dnsjava (www.dnsjava.org)									
0.1-0.2	1	0	5	6	137	17	14%	1	17
0.2-0.3	8	0	28	36	1120	134	73%	3	21
0.3-0.4	1	1	24	26	711	45	52%	3	35
0.4-0.5	3	2	25	30	978	95	57%	31	37
0.5-0.6	0	0	9	9	272	45	18%	6	29
0.6-0.7	6	0	10	16	1052	40	25%	5	53
0.7-0.8	6	1	16	23	1354	78	34%	23	46
0.8-0.8.1	0	0	3	3	27	3	5%	1	2
0.8.1-0.8.2	0	0	42	42	1519	344	70%	19	55
0.8.2-0.8.3	0	0	6	6	307	40	10%	1	45
0.9-0.9.1	1	2	6	9	553	30	13%	0	13
0.9.1-0.9.2	58	56	3	117	15915	115	100%	21	55
0.9.2-0.9.3	0	0	1	1	5	1	2%	0	0
0.9.3-0.9.4	0	0	1	1	9	1	2%	0	0
0.9.4-0.9.5	0	0	4	4	307	16	7%	0	5
0.9.5-1.0	3	0	61	64	1181	105	100%	9	43
1.0-1.0.1	0	0	6	6	52	11	9%	0	10
1.0.1-1.0.2	0	0	13	13	457	47	20%	4	36
1.0.2-1.1	16	2	35	53	3362	264	62%	29	174
1.1-1.1.1	1	0	13	14	413	29	17%	4	13
1.1.1-1.1.2	0	0	5	5	26	6	6%	0	6
1.1.2-1.1.3	2	0	2	4	240	10	4%	0	10
1.1.3-1.1.4	0	0	3	3	47	11	4%	0	5
1.1.4-1.1.5	0	0	8	8	354	41	10%	11	24
1.1.5-1.1.6	1	0	8	9	271	14	10%	0	7
1.1.6-1.2.0	2	1	21	24	2150	208	27%	36	201
1.2.0-1.2.1	0	0	28	28	323	56	34%	10	23
1.2.1-1.2.2	0	0	14	14	436	72	17%	3	31
1.2.2-1.2.3	0	0	4	4	36	8	5%	0	4
MED	**0**	**0**	**8**	**9**	**354**	**40**	**17%**	**3**	**23**
AVG	**4**	**2**	**14**	**20**	**1159**	**65**	**28%**	**8**	**34**
AVG	**3**	**2**	**11**	**16**	**997**	**54**	**19%**	**7**	**27**

Table 6.8: Extracted rules and associated change descriptions (1)

Source	Rules and Their Interpretation	Excerpt from Change Description
carol		
62-63	current_field(f,n,"CarolConfiguration") ∧ current_accesses(f,"CarolConfiguration.loadCarolConfiguration()") ⇒ added_field(f,n,"CarolConfiguration") All fields in the `CarolConfiguration` class that are accessed from the `loadCarolConfiguration` method are newly added.	A new simplified configuration mechanism. (with bug id references)
	past_field(f,n,"CarolDefaultValues") ∧ past_fieldoftype(f,"Properties") ⇒ deleted_field(f,n,"CarolDefaultValues") All `Properties` type fields in the `CarolDefaultValues` class got deleted.	
128-129	current_method(m,"getPort()",t) ⇒ added_method(m,"getPort()",t) All `getPort` methods are new methods.	Port number trace problem.
421-422	current_calls(m,"NamingExceptionHelper.create(Exception)") ⇒ added_calls(m,"NamingExceptionHelper.create(Exception)") past_calls(m,"JNDIRemoteResource.getResource()") ⇒ deleted_calls(m,"Throwable.printStackTrace()") All calls to the `NamingExceptionHelper.create` method are new. All methods that called the `getResource` method no longer call the `printStackTrace` method.	Refactoring of the spi package... (247 words long)
	current_inheritedmethod(m,"AbsContext",t) ⇒ added_inheritedmethod(m,"AbsContext",t) past_method(m,n,"JRMPContext") =¿ deleted_method(m,n,"JRMPContext")... Many methods inherit implementation from the `AbsContext` class. All methods in the `JRMPContext` class were deleted.	
429-430	added_type("AbsRegistry") current_inheritedmethod(m, "AbsRegistry", t) ⇒ added_inheritedmethod(m, "AbsRegistry", t) past_subtype("NameSvc", t) ∧ past_field(f, "host", t) ⇒ deleted_field(f, "host", t), except t="LmiRegistry" past_subtype("NameSvc", t) ∧ past_method(m, "getHost()", t) ⇒ deleted_method(m, "getHost()", t), except t="LmiRegistry" `AbsRegistry` is a new class. Many methods inherit implementation from the `AbsRegistry` class. All `host` fields in the `NameSvc`'s subtypes were deleted. All `getHost` methods in the `NameSvc`'s subtypes were deleted.	Common methods go in an abstract class, easier to extend/maintain/fix.

Table 6.9: Extracted rules and associated change descriptions (2)

Source	Rules and Their Interpretation	Excerpt from Change Description
480-481	past_calls(m, "CarolCurrentConfiguration.setRMI(String)") ⇒ deleted_calls(m, "Enumeration.nextElement())" current_accesses("MultiContext.contextsOfConfigurations",m) ⇒added_calls(m, "Iterator.next()") All methods that called the `CarolCurrentConfiguration.setRMI` method no longer call the `Enumeration.nextElement` method. All methods that access the `MultiContext.contextsOfConfigurations` field added calls to the `Iterator.next` method.	Change the configuration process of Carol as discussed... (139 words long.)
dnsjava		
0.6-0.7	current_method(m,n, "RRset") ∧ current_calls("Cache.addRRset(RRset,byte,Object)",m) ⇒ added_method(m,n, "RRset") `RRset`'s methods that are called by the `Cache.addRRset` method are new.	DNS.dns uses Cache
1.0.2-1.1	past_method(m, "sendAsync()",t) ⇒ added_return(m, "Object") past_method(m, "sendAsync()",t) ⇒ deleted_return(m, "int") All `sendAsync` methods return an object with the `Object` type. All `sendAsync` methods no longer return an object with the `int` type.	Resolver.sendAsync returns an Object instead of an int.
1.1.4-1.1.5	past_calls(m, ".update.parseRR(Tokenizer,short,int)") ⇒ deleted_calls(m, "String.equals(Object)") past_calls(m, ".update.parseSet(Tokenizer,short)") ⇒ deleted_calls(m, ".update.parseRR(Tokenizer,short,int)") past_calls(m, ".update.parseSet(Tokenizer,short)") ⇒ added_calls(m, "String.startsWith(String)") All methods that called the `update.parseRR` method no longer call the `String.equals` method. All methods that called the `update.parseRR` method no longer call the `update.parseRR` method. All methods that called the `update.parseRR` method no longer call the `String.startsWith` method.	update client syntax enhancement (add/delete/require/prohibit/glue) no longer require -r, -s, or -n.
LSDiff		
20-21	past_type(t1,n, "edu.uw.cs.lsd") ∧ past_type(t2,n, "edu.uw.cs.lsd.jquery") ⇒ deleted_type(t1,n, "edu.uw.cs.lsd") All classes in the `lsd` package that have the same name class in the `jquery` package got deleted.	*no corresponding comment*

6.4.2 Comparison with Textual Delta.

In practice, programmers often use *diff* and read programmer-provided descriptions such as check-in comments or change logs. It is infeasible to directly compare LSDiff results (LSD) with traditional *diff* results (TD) and change descriptions. *Diff* computes textual differences while LSDiff computes only structural differences, and change descriptions are often missing, hard to trace back to a program, and in free-form. Thus, our goal is not to directly compare them but to understand when LSDs complement TDs and change descriptions. For this investigation, we built a viewer that visualizes each rule with *diff* outputs, similar to what is shown in Figure 6.1 and Figure 6.2.

Table 6.6 and Table 6.7 show quantitative comparison results. *CLOC* represents the number of added, deleted, and changed lines. *Hunk* represents the number of blocks with consecutive line changes, and *% Touched* represents the percentage of files that programmers must inspect to examine the change completely out of the total number of files in both versions. It is computed as (# added files + # deleted files + 2 × # changed files) / (total # files in both versions). The more hunks there are and the higher the percentage of touched files is, generally the harder it is to inspect a TD.

While the average TD for *carol* has over 1200 lines of changes across 13 different files, LSD represents these changes as roughly 10 rules and 20 facts. While the average TD for *dnsjava* has over 1100 lines across 20 different files, the average LSD has 8 rules and 34 facts. For our own program, while the average TD has about 300 lines of changes across 6 files, the average LSD has 1 rule and 13 facts. Overall, while an average textual delta consists of 997 lines of change scattered across 16 files, our LSD reports an average of 7 rules and 27 facts, relatively smaller than an equivalent textual delta.

The benefits of LSD appear to depend heavily on how systematic the change is. (See Table 6.8). When changes are structurally systematic—e.g., refactoring, feature addition and removal, dependency addition and removal, constant pool migration—LSDs contain only a few rules and facts even if TDs contain a large number of hunks scattered across many files. Consider the change in *carol* 429-430, *"Common methods go in an abstract class, Easier to extend/maintain/fix."* If a programmer intends to understand whether this

change is truly an *extract superclass* refactoring and whether the refactoring was completed, she needs to examine over 700 lines across 9 files. On the other hand, LSD summarizes this change using only 12 rules and 7 facts and provides concrete information about the refactoring—`AbsRegistry` was created by pulling up `host` related fields and methods from the classes implementing `NameSvc` interface except for `LmiRegistry`. Consider another change in *carol* 128-129, *"Bug fix, port number trace problem."* To understand how the bug was fixed, a programmer needs to read over 150 lines scattered across 10 files. Our LSD represents the same change with only 1 rule and 4 facts—`getPort` methods were added to six different classes and they were invoked from a tracer module, `TraceCarol`. If a programmer examines the LSD before reading the TD, upon inspecting one corresponding file, she can probably skip five other files that include `getPort`.

When several different systematic changes are mixed with many random non-systematic changes, LSDs tend to contain many rules and facts. Despite a large amount of information in those LSDs, we believe LSDs can still complement scattered and verbose TDs by providing an overview of systematic changes, helping programmers focus on remaining non-systematic changes instead. For instance, a programmer may find the TD for *carol* 421-422 overwhelming since it includes more than 4000 lines of changes across 14 files. In this case, LSD rules can help programmers quickly understand the systematic changes—modifying exception handling to use `NamingExceptionHelper` and creating a superclass `AbsContext` by extracting common methods from `Context` classes—and focus on other changes instead.

In several cases, TD shows some changes but LSD is empty because LSD does not model differences in comments, control logic, and temporal logic. For example, the LSD for *dnsjava* 0.9.2-0.9.3 is empty because the code change includes only one added *if* statement and does not incur changes in structural dependencies.

Overall, our comparison shows that the more systematic code changes are, the smaller number of rules and facts LSDs include. On the other hand, TDs may be scattered across many files and hunks even if the change is structurally homogeneous and systematic. Consistent with our focus on systematic changes, when the change is very small or completely random, LSDiff provides little additional benefits.

6.4.3 Comparison with Programmer-Provided Change Descriptions.

Programmers often write check-in comments or update a change log file to convey their change intentions. To understand how LSDs and change descriptions complement each other, we compared LSDs with check-in comments (*carol* and LSDiff) and change logs (*dnsjava*). For this comparison, we examined and interpreted all LSD rules and facts and then traced them to corresponding sentences in the change description. Table 6.8 and Table 6.9 show the comparison results.

In many cases, although change descriptions hint at systematic changes, they do not provide much detail. For example, the check-in comment for *carol* 62-63—*"a new simplified configuration mechanism"*—does not indicate which classes implement the new configuration mechanism. LSD rules show that `CarolConfiguration` added many fields to be used by `loadCarolConfiguration`, and `CarolDefaultValues` deleted all `Properties` type fields.

In some cases, change comments and LSDs agree on the same information with a similar level of detail. For example, in *dnsjava* 1.0.2-1.1, both the LSD and the change log describe that `sendAsync` methods return `Object` instead of `int`. In some other cases, LSDs and change descriptions discuss different aspects of change. For instance, the change comments for *carol* 480-481 refer to email discussions on the design of new APIs and include code examples while LSD provides implementation details such as the use of `Iterator` instead of `Enumeration`.

Because change descriptions are free-form, they can contain any kind of information at any level of detail; however, it is often incomplete or too verbose. More importantly, it is generally hard to trace back to a program. We believe that LSDs can complement change descriptions by providing concrete information that can be traced to code.

6.5 Discussion

6.5.1 Impact of Input Parameters.

The input parameters, m (the minimum number of facts a rule must match), a (the minimum accuracy), and k (the maximum number of literals a rule can have in its antecedent) define which rules should be considered in the output. To understand how varying these parameters affects our results, we varied m from 1 to 5, a from 0.5 to 1 with an increment of 0.125, and

Table 6.10: Impact of varying input parameters

		Rule	Fact	Cvrg.	Csc.	Ad'l.	Time(Min)
m	1	39.6	0	100%	7.4	10.1	2.0
	2	14.6	13.1	92%	10.6	7.4	11.2
	3	9.9	20.4	85%	9.9	5.5	9.1
	4	7.7	25.7	82%	9.1	5.4	8.7
	5	5.7	30	80%	8.5	3.5	7.8
a	0.5	11.1	15.6	89%	10.6	2.1	6.8
	0.625	9.7	17.2	88%	11.0	4.0	7.3
	0.75	9.9	20.4	85%	9.9	5.5	9.0
	0.875	10.8	24.2	78%	8.6	9.1	12.7
	1	13.3	26.2	78%	7.9	12.5	16.5
k	1	7.5	33.8	78%	7.2	0.4	0.7
	2	9.9	20.4	85%	9.9	5.5	9.1

k from 1 to 2. Table 6.10 shows the results in terms of average for the *carol* data set.

When m is 1, all facts in ΔFB are covered by rules by definition. As m increases, fewer rules are found and they cover fewer facts in ΔFB.

As a increases, a smaller proportion of exceptions is allowed per rule; thus, our algorithm finds more rules each of which covers a smaller proportion of the facts, decreasing the conciseness and coverage measures.

Changing k from 1 to 2 allows our algorithm to find more rules and improves the additional information measure from 0.4 to 5.5 by considering unchanged code fragments that are further away from changed code. With our current tool, we were not able to experiment with k greater than 2 because the large rule search space led to a very long running time. In the future, we plan to explore using *Alchemy*—a state-of-the-art first order logic rule learner developed at the University of Washington [171]—to find rules more efficiently.

6.5.2 Threats to Validity.

In terms of our focus group study, though it is common to commission an external, professional research vendor, I designed the discussion guide and took a moderator role due to

the difficulty of finding a vendor with similar expertise in program differencing tools. The moderator's intimate knowledge and bias towards *LSDiff* may have led the participants to support the moderator's views. Though conducting multiple focus groups and contrasting them is encouraged, we conducted only a single focus group. Furthermore, the participants' view may be biased to practices in their organization—where code reviews are often done by emails.

In terms of internal validity, the inferred rules are incomplete in nature as they depend on both input parameter settings and the predefined rule styles (Table 6.3). We need to investigate further about what other types of systematic changes that *LSDiff* does not cover and how frequent they are.

As some participants in the focus group pointed out, LSDiff may report systematic changes that are not of interest to the programmer. Determining the frequency and cost of such false positives is beyond the scope of our work-to-date largely because it is heavily dependent on tasks, projects and programmers. Resolving this will likely take in-depth evaluations of LSDiff in the context of real development tasks; these evaluations will need to consider how to distinguish uninteresting patterns from unanticipated but interesting patterns.

In terms of external validity, although our assessment in Section 6.4 provides a valuable illustration of how LSDiff can complement existing uses of *diff*, our findings may not generalize to other data sets. We need further investigations into how a program size and the gap between program versions affect LSDiff results.

6.6 Application of Change-Rules

Our inferred change-rules represent program differences in a concise and comprehensible form and also make it easier for programmers to note inconsistencies in code changes. This allows them to serve as a basis for many software engineering applications. We sketch several such applications and include motivating examples from our study. Some of the example rules below are slightly modified for presentation purposes.

Finding Bugs While examining the inferred API change-rules, we found that the exceptions of change-rules often signal a bug arising from incomplete or inconsistent changes. For example, the rule

- for all x:method-header in J*.addTitle(*)

  except JThermometer.addTitle(Title)

  procedureReplace(x, addTitle, addSubtitle)

  [Interpretation: All methods with a name "J*.addTitle(*)" changed their procedure name from addTitle to addSubtitle except the JThermometer.addTitle(Title) method.]

has one exception, which indicates that a programmer misspelled `addSubtitle` to `addSubitle` when modifying the `addTitle` method of `JThermometer`, which is a subclass of `JFreeChart`. This misspelling causes dynamic dispatching to `JThermometer` not to function properly because `addSubtitle` is no longer overridden.

As another example, consider the following two API change-rules, in which the second one is found one release after the first one. We suspect that a programmer fixed only two out of the three problems, leaving one bug.

- for all x:method-header in *.draw(*, Key, *)

  except { HorizontalBar, VerticalBar, StatBar }

  argReplace(x, Key, Category)

  [Interpretation: All methods with a name "*.draw(*, Key, *)" changed a Key type input argument to a Category type argument.]

- for all x:method-header in *Bar.draw(*, Key, *)

  except { VerticalBar }

  argReplace(x, Key, Category)

  [Interpretation: All methods with a name "*Bar.draw(*, Key, *)" changed a Key type input argument to a Category type argument.]

A similar idea that detects potential errors from inferred refactorings has been explored by Görg and Weißgerber [106]. However, they check only a predefined set of refactoring consistency patterns.

Empirical Software Evolution Analysis Our inferred API change-rules can be used for understanding API evolution. For example, the following rule describes that `Shape` type in some APIs were removed to hide unnecessary details from clients.

• for all x:method-header in chart.*.*(Graphic, *, Shape)

argDelete(x, Shape)

[Interpretation: All methods with a name "chart.*.*(Graphic, *, Shape)" deleted Shape type arguments in their input signature.]

Furthermore, inferred change-rules may reveal volatility of some API changes. In the following example, the first rule shows that the use of `Category` type was replaced by `[Key, int]` type. In the next release, the same change was quickly reversed.

• for all x:method-header in *.*.*(Category)

inputSignatureReplace(x,[Category],[Key, int])

[Interpretation: All methods with a name "*.*.*(Category)" changed their input signature from one Category type argument to a tuple of (Key, int).]

• for all x:method-header in *.*.*(Key, int)

inputSignatureReplace(x,[Key, int],[Category])

[Interpretation: All methods with a name "*.*.*(Key, int)" changed its input signature from a tuple of (Key, int) to a Category type argument.]

Checking Dependency Creation and Removal When team leads review a patch, they often wonder whether a new dependency is unexpectedly introduced or whether existing dependencies are completely removed as intended. In our study, we have found many inferred LSDiff rules clearly show such dependency creation and removal; for example, the following two rules show that all call dependencies to `NamingHelper` are newly introduced and that all accesses to `JNI.URL` in the old version are completely removed.

• current_calls(m, "NamingHelper()") ⇒ added_calls(m, "NamingHelper()")

[Interpretation: All method-calls to the NamingHelper method in the new version are new.]

• past_accesses("JNI.URL", m)⇒deleted_accesses("JNI.URL", m)

[Interpretation: All methods that accessed the JNI.URL field deleted the accesses to the JNI.URL field.]

Identifying Related Changes Programmers often need to sort out mixed logical changes because some programmers commit unrelated changes together. Our inferred LSDiff rules can help identify related changes by showing the common characteristics of co-changed code. Consider dnsjava release 0.6-0.7; there are two added classes, `Cache` and `CacheResponse`, and three added methods in `RRSet`. Despite its change comment, *"DNS.dns uses Cache,"* it is not clear whether all added code fragments implement the cache feature. The following rule shows that the three methods are indeed a part of cache feature because they are called by `Cache.addRRSet`.

• current_calls("Cache.addRRSet", m)⇒added_method(m,"RRset")

[Interpretation: All methods that are called from the Cache.addRRSet method are new methods.]

6.7 Summary of Logical Structural Diff

LSDiff discovers and represents systematic structural differences as logic rules. Each rule concisely describes a group of changes that share similar structural characteristics and notes anomalies to systematic change patterns.

Through a focus group study with professional software engineers, we assessed LSDiff's potential benefits and studied when and how LSDiff can complement existing program differencing tools. Our study participants believe that the grouping of related systematic changes can complement *diff*'s file-based organization and the detection of anomalies can help programmers discover potential missed updates.

In addition, we quantitatively compared LSDiff results with what an existing program differencing approach would produce at the same abstraction level; LSDiff produces 9.3 times more concise results and finds 9.7 additional structural facts that cannot be found by looking at the code that changed between versions.

Chapter 7

CONCLUSIONS AND FUTURE WORK

Section 7.1 summarizes this dissertation's contributions, and Section 7.2 describes future research ideas.

7.1 Summary of Contributions

To help programmers reason about software changes at a high-level, this dissertation introduced a program differencing approach that extracts high-level change descriptions.

Based on the insight that high-level changes often require systematic code-level changes, our approach discovers and represents systematic code changes as first order logic rules. The core of this approach is novel rule-based representations that explicitly capture systematic changes and corresponding rule-inference algorithms that automatically discover such rules from two program versions. This approach is instantiated at two abstraction levels: first at a method-header level and then at the level of code elements and their structural dependencies.

This rule-based change inference approach has been assessed both quantitatively and qualitatively through its application to multiple open source projects' change history and through a focus group study with professional developers from a large E-commerce company. The participants' comments show that our approach is promising both as a complement to *diff*'s file-based approach and also as a way to help programmers discover potential bugs by identifying exceptions to inferred systematic changes. The quantitative assessments show that our API change-rule inference produces 22–77% more concise matching results compared to other method level matching tools and refactoring reconstruction tools. Our LSDiff produces 9.3 times more concise results by identifying 75% of structural differences as systematic changes compared to an existing program differencing approach.

7.2 Future Work

Improvement of Logical Structural Diff Logic rules are not always easy to read by developers; as it is fairly mechanical to translate inferred rules to English sentences, we plan to build a rule translator.[1]

Our empirical studies of rules indicate that there is a need for even higher-order representations beyond rules; for example, complex refactorings and design pattern changes are often described as a collection of related rules. We intend to build a higher-order representation and a clustering algorithm that discovers sets of related rules.

Exploiting Change Semantics and Structures We plan to use change-rules as a foundation to approach problems in release planning, regression testing, and cost estimation. In particular, we are interested in research questions including the following: By reasoning about temporal and structural dependencies among program deltas, can we assist software engineers in identifying the unit of logical software changes. How can we better select and prioritize regression tests by exploiting the homogeneity and heterogeneity found in program changes? Can we better estimate software change cost using the semantic structure of past similar changes?

Past empirical studies of software evolution have primarily focused on quantitative and statistical analyses of a program over multiple versions, largely ignoring the structure and semantics of software changes between versions. How do such studies compare with qualitative studies of software evolution using inferred rules? We believe that the homogeneity and heterogeneity found in program changes may shed light on the problem of assessing software quality. In particular, we plan to use LSDiff to build a better understanding of the frequency and characteristics of systematic changes. For example, it may be feasible to run experiments to assess how often systematic changes are aligned with a containment hierarchy, with crosscutting changes, with refactorings, or indeed with none of these known kinds of systematic change.

---

[1] In this dissertation, english rule descriptions were produced manually.

Beyond Source Code Level Change-Rules We will investigate software changes from various angles such as abstraction level, time, behavior, and type of software artifact. (1) The notion of a program delta must be extended from "between two versions" to "across a series of versions" as programmers often complete a single logical change in phases or submit several unrelated changes in a single check-in. (2) How should we identify and represent differences in run-time behavior caused by a program change? While we have a common vocabulary for describing different types of source code changes, such as 'refactoring' and 'crosscutting changes,' there is a lack of common vocabulary for run-time behavior changes. How can we categorize and describe run-time behavior changes? Is there also structure in behavioral differences? Can we extend existing run-time behavior capturing techniques such as dynamic invariants or path spectra? (3) We plan to investigate changes in requirement models and architecture diagrams. Automatically checking consistency between different abstractions of software changes may bridge the gap between how software architects plan changes and how programmers implement source-level changes.

Actively Leveraging Historical Information Explicitly capturing the semantics of a program delta will not only help programmers in their daily tasks but also enable software engineering research to leverage historical information. Our goal is to avoid redundancies in re-analyzing and re-testing changed programs. The envisioned analysis technique will be more efficient than traditional program analyses by leveraging two types of information, which are not actively used in software engineering research: (1) history-based approximation of a program's behavior and (2) program delta semantics. For example, suppose that there is 90% confidence that a variable `foo` never pointed to `bar` in the past revisions and the code modification does not involve a direct assignment to `foo`; when time is limited for a whole program analysis, our analysis algorithm will use historical knowledge about aliasing relations and delta semantics to produce an approximated result.

History can help us learn which decisions are good and which decisions are bad. We see tremendous potential in using change history to help programmers make better decisions. First, inspired by our clone genealogy analysis, we plan to build a software economics model and a refactoring reminder that can suggest to programmers when to refactor duplicated

code to maximize their return on refactoring investment. Second, we envision a tool that suggests how to redesign software by identifying design decisions that were intended to be encapsulated but were later inadvertently exposed to other modules.

Coping with Redundancies in Software Evolution There are many redundancies in software development: redundant code edits, redundant bug reports and subsequent efforts of triaging them, redundant efforts in deployment and configuration, etc. We believe that such redundancies are exacerbated in collaborative software development, where one developer's effort is not captured and reused by other developers. We would like to study in which task contexts redundancies occur frequently, which types of redundancies are inherent, and which types of redundancies can be avoided. Based on these studies, we will build software engineering tools that actively capture developer efforts, recognize their redundancies, and save the reusable efforts for later uses.

BIBLIOGRAPHY

[1] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216, New York, NY, USA, 1993. ACM.

[2] Alex Aiken. Moss: A system for detecting software plagiarism. *Stanford University, See http://theory.stanford.edu/ãiken/moss/*, 2005.

[3] Glenn Ammons, Rastislav Bodik, and James R. Larus. Mining specifications. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16, New York, NY, USA, 2002. ACM Press.

[4] Giuliano Antoniol, Gerardo Casazza, Massimiliano Di Penta, and Ettore Merlo. Modeling clones evolution through time series. In *ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance*, page 273, Washington, DC, USA, 2001. IEEE Computer Society.

[5] Giuliano Antoniol, Massimiliano Di Penta, and Ettore Merlo. An automatic approach to identify class evolution discontinuities. In *IWPSE '04: Proceedings of the Principles of Software Evolution, 7th International Workshop*, pages 31–40, Washington, DC, USA, 2004. IEEE Computer Society.

[6] Giuliano Antoniol, Umberto Villano, Ettore Merlo, and Massimiliano Di Penta. Analyzing cloning evolution in the Linux kernel. *Information & Software Technology*, 44(13):755–765, 2002.

[7] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. A differencing algorithm for object-oriented programs. In *ASE '04: Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, pages 2–13, Washington, DC, USA, 2004. IEEE Computer Society.

[8] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 432–441, New York, NY, USA, 2005. ACM.

[9] Alberto Apostolico and Zvi Galil, editors. *Pattern matching algorithms*. Oxford University Press, Oxford, UK, 1997.

[10] Darren C. Atkinson and William G. Griswold. Effective pattern matching of source code using abstract syntax patterns. *Software–Practice & Experience*, 36(4):413–447, 2006.

[11] Lerina Aversano, Luigi Cerulo, and Massimiliano Di Penta. How clones are maintained: An empirical study. In *CSMR '07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, pages 81–90, Washington, DC, USA, 2007. IEEE Computer Society.

[12] Brenda S. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, 24:49–57, 1992.

[13] Brenda S. Baker and Udi Manber. Deducing similarities in Java sources from bytecodes. In *Proceedings of Usenix Annual Technical Conference*, pages 179–190, 1998.

[14] Egon Balas and Manfred W. Padberg. Set Partitioning: A Survey. *SIAM Review*, 18:710–760, 1976.

[15] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, and Kostas Kontogiannis. Measuring clone based reengineering opportunities. In *METRICS '99: Proceedings of the 6th International Symposium on Software Metrics*, page 292, Washington, DC, USA, 1999. IEEE Computer Society.

[16] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, and Kostas Kontogiannis. Partial redesign of java software systems based on clone analysis. In *WCRE '99: Proceedings of the Sixth Working Conference on Reverse Engineering*, page 326, Washington, DC, USA, 1999. IEEE Computer Society.

[17] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lagüe, and Kostas Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *WCRE '00: Proceedings of the Seventh Working Conference on Reverse Engineering*, page 98, Washington, DC, USA, 2000. IEEE Computer Society.

[18] Mihai Balint, Radu Marinescu, and Tudor Girba. How developers copy. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 56–68, Washington, DC, USA, 2006. IEEE Computer Society.

[19] Thomas Ball and Stephen G. Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, 1996.

[20] Thomas Ball, Jung min Kim, Adam A. Porter, and Harvey P. Siy. If your version control system could talk. In *ICSE '97 Workshop on Process Modelling and Empirical Studies of Software Engineering*, May 1997.

[21] Elisa L. A. Baniassad and Gail C. Murphy. Conceptual module querying for software reengineering. In *ICSE '98: Proceedings of the 20th International Conference on Software Engineering*, pages 64–73, Washington, DC, USA, 1998. IEEE Computer Society.

[22] Elisa L. A. Baniassad, Gail C. Murphy, and Christa Schwanninger. Design pattern rationale graphs: linking design to source. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 352–362, Washington, DC, USA, 2003. IEEE Computer Society.

[23] Hamid Abdul Basit and Stan Jarzabek. Detecting higher-level similarity patterns in programs. In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 156–165, New York, NY, USA, 2005. ACM.

[24] Hamid Abdul Basit, Damith C. Rajapakse, and Stan Jarzabek. Beyond templates: a study of clones in the stl and some general implications. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 451–459, New York, NY, USA, 2005. ACM.

[25] Samuel Bates and Susan Horwitz. Incremental program testing using program dependence graphs. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 384–396, New York, NY, USA, 1993. ACM Press.

[26] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 368, Washington, DC, USA, 1998. IEEE Computer Society.

[27] Kent Beck. *extreme Programming explained, embrace change*. Addison-Wesley Professional, 2000.

[28] Laszlo A. Belady and M.M. Lehman. A Model of Large Program Development. *IBM Systems Journal*, 15(3):225–252, 1976.

[29] Thomas Berlage and Andreas Genau. A framework for shared applications with a replicated architecture. In *UIST '93: Proceedings of the 6th annual ACM symposium on User interface software and technology*, pages 249–257, New York, NY, USA, 1993. ACM.

[30] Valdis Berzins. Software merge: semantics of combining changes to programs. *ACM Transactions on Programming Languages and Systems*, 16(6):1875–1903, 1994.

[31] Jennifer Bevan and Jr. E. James Whitehead. Identification of software instabilities. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*, page 134, Washington, DC, USA, 2003. IEEE Computer Society.

[32] Jennifer Bevan, Jr. E. James Whitehead, Sunghun Kim, and Michael Godfrey. Facilitating software evolution research with kenyon. In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 177–186, New York, NY, USA, 2005. ACM.

[33] Dirk Beyer, Andreas Noack, and Claus Lewerentz. Efficient relational calculation for software analysis. *IEEE Transactions on Software Engineering*, 31(2):137–149, 2005.

[34] Hugh Beyer and Karen Holtzblatt. *Contextual design*. Morgan Kaufmann, 1999.

[35] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas Webster. The concept assignment problem in program understanding. In *ICSE '93: Proceedings of the 15th International Conference on Software Engineering*, pages 482–498, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[36] David Binkley. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering*, 23(8):498–516, 1997.

[37] David Binkley, Susan Horwitz, and Thomas Reps. Program integration for languages with procedure calls. *ACM Transactions on Software Engineering and Methodology*, 4(1):3–35, 1995.

[38] Barry W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. *Proceedings of the 2nd International Conference on Software Engineering*, pages 592–605, 1976.

[39] Marat Boshernitsan, Susan L. Graham, and Marti A. Hearst. Aligning development tools with the way programmers think about code changes. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 567–576, New York, NY, USA, 2007. ACM.

[40] Gilad Bracha and William Cook. Mixin-based inheritance. In *OOPSLA/ECOOP '90: Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages, and Applications*, pages 303–311, New York, NY, USA, 1990. ACM.

[41] Silvia Breu and Thomas Zimmermann. Mining aspects from version history. In *International Conference on Automated Software Engineering*, pages 221–230, 2006.

[42] Magiel Bruntink, Arie van Deursen, Tom Tourwe, and Remco van Engelen. An evaluation of clone detection techniques for identifying crosscutting concerns. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 200–209, Washington, DC, USA, 2004. IEEE Computer Society.

[43] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. Towards a taxonomy of software change: Research articles. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(5):309–332, 2005.

[44] Elizabeth Burd and John Bailey. Evaluating clone detection tools for use during preventative maintenance. In *SCAM '02: Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation*, pages 36–43, Washington, DC, USA, 2002. IEEE Computer Society.

[45] Scott Burson, Gordon B. Kotik, and Lawrence Z. Markosian. A program transformation approach to automating softwarere-engineering. In *COMPSAC 90: Fourteenth Annual International Computer Software and Applications Conference*, pages 314–322, 1990.

[46] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. Identifying changed source code lines from version repositories. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 14, Washington, DC, USA, 2007. IEEE Computer Society.

[47] Craig Chambers, Jeffrey Dean, and David Grove. A framework for selective recompilation in the presence of complex intermodule dependencies. In *ICSE '95: Proceedings of the 17th International Conference on Software Engineering*, pages 221–230, New York, NY, USA, 1995. ACM.

[48] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 493–504, New York, NY, USA, 1996. ACM.

[49] Annie Chen, Eric Chou, Joshua Wong, Andrew Y. Yao, Qing Zhang, Shao Zhang, and Amir Michail. Cvssearch: Searching through source code using cvs comments. In *ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance*, page 364, Washington, DC, USA, 2001. IEEE Computer Society.

[50] Ophelia C. Chesley, Xiaoxia Ren, and Barbara G. Ryder. Crisp: A debugging tool for java programs. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 401–410, Washington, DC, USA, 2005. IEEE Computer Society.

[51] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson R. Engler. An empirical study of operating system errors. In *SOSP*, pages 73–88, 2001.

[52] Kingsum Chow and David Notkin. Semi-automatic update of applications in response to library changes. In *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*, page 359, Washington, DC, USA, 1996. IEEE Computer Society.

[53] Christian Collberg, Stephen Kobourov, Jasvir Nagra, Jacob Pitts, and Kevin Wampler. A system for graph-based visualization of the evolution of software. In *SoftVis '03: Proceedings of the 2003 ACM Symposium on Software Visualization*, pages 77–88, New York, NY, USA, 2003. ACM.

[54] Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. *ACM Computing Survey*, 30(2):232–282, 1998.

[55] James R. Cordy. Comprehending reality " practical barriers to industrial adoption of software maintenance automation. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, page 196, Washington, DC, USA, 2003. IEEE Computer Society.

[56] James R. Cordy. The txl source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.

[57] James R. Cordy, Thomas R. Dean, Andrew J. Malton, and Kevin A. Schneider. Source transformation in software engineering using the txl transformation system. *Journal of Information and Software Technology*, 44:827–837, 2002.

[58] Rylan Cottrell, Joseph J. C. Chang, Robert J. Walker, and Jörg Denzinger. Determining detailed structural correspondence for generalization tasks. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 165–174, New York, NY, USA, 2007. ACM.

[59] Davor Cubranic and Gail C. Murphy. Hipikat: recommending pertinent software development artifacts. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 408–418, Washington, DC, USA, 2003. IEEE Computer Society.

[60] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 25–35, New York, NY, USA, 1989. ACM Press.

[61] Barthélémy Dagenais, Silvia Breu, Frédéric Weigand Warr, and Martin P. Robillard. Inferring structural patterns for concern traceability in evolving software. In *ASE '07: Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering*, pages 254–263, New York, NY, USA, 2007. ACM.

[62] Barthélémy Dagenais and Martin P. Robillard. Recommending adaptive changes for framework evolution. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 481–490, New York, NY, USA, 2008. ACM.

[63] Michael L. Van de Vanter. The documentary structure of source code. *Information & Software Technology*, 44(13):767–782, 2002.

[64] Robert DeLine, Amir Khella, Mary Czerwinski, and George Robertson. Towards understanding programs through wear-based filtering. In *SoftVis '05: Proceedings of the 2005 ACM Symposium on Software Visualization*, pages 183–192, New York, NY, USA, 2005. ACM Press.

[65] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 166–177, New York, NY, USA, 2000. ACM.

[66] Marcus Denker, Tudor Gîrba, Adrian Lienhard, Oscar Nierstrasz, Lukas Renggli, and Pascal Zumkehr. Encapsulating and exploiting change with changeboxes. In *ICDL '07: Proceedings of the 2007 International Conference on Dynamic Languages*, pages 25–49, New York, NY, USA, 2007. ACM.

[67] Premkumar Devanbu. GENOA: a customizable language-and front-end independent code analyzer. *Proceedings of the 14th International Conference on Software Engineering*, pages 307–317, 1992.

[68] Premkumar Devanbu, Ronald J. Brachman, Peter G. Selfridge, and Bruce W. Ballard. LaSSIE: a knowledge-based software information system. *Communications of the ACM*, 34(5):34–49, 1991.

[69] Danny Dig and Ralph Johnson. The role of refactorings in api evolution. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 389–398, Washington, DC, USA, 2005. IEEE Computer Society.

[70] Danny Dig and Ralph Johnson. Automated detection of refactorings in evolving components. In *ECOOP '06: Proceedings of European Conference on Object-Oriented Programming*, pages 404–428. Springer, 2006.

[71] Danny Dig, Kashif Manzoor, Ralph Johnson, and Tien N. Nguyen. Refactoring-aware configuration management for object-oriented programs. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 427–436, Washington, DC, USA, 2007. IEEE Computer Society.

[72] Pedro Domingos, Stanley Kok, Hoifung Poon, Matthew Richardson, and Parag Singla. Unifying logical and statistical AI. In *AAAI '06: Proceedings of the Twenty-First National Conference on Artificial Intelligence, Boston, MA, July*, 2006.

[73] Stephen Drape, Oege de Moor, and Ganesh Sittampalam. Transforming the .net intermediate language using path logic programming. In *PPDP '02: Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 133–144, New York, NY, USA, 2002. ACM Press.

[74] Ekwa Duala-Ekoko and Martin P. Robillard. Tracking code clones in evolving software. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 158–167, Washington, DC, USA, 2007. IEEE Computer Society.

[75] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, page 109, Washington, DC, USA, 1999. IEEE Computer Society.

[76] Michael Eichberg, Sven Kloppenburg, Karl Klose, and Mira Mezini. Defining and continuous checking of structural program dependencies. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 391–400, New York, NY, USA, 2008. ACM.

[77] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J. S. Marron, and Audris Mockus. Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.

[78] Stephen G. Eick, Todd L. Graves, Alan F. Karr, Audris Mockus, and Paul Schuster. Visualizing software changes. *IEEE Transactions on Software Engineering*, 28(4):396–412, 2002.

[79] Sebastian G. Elbaum, David Gable, and Gregg Rothermel. The impact of software evolution on code coverage information. In *ICSM*, pages 170–179, 2001.

[80] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz-Open Source Graph Drawing Tools. *Graph Drawing*, pages 483–485, 2001.

[81] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. Ph.D. Disseratation, University of Washington, Seattle, Washington, August 2000.

[82] Martin Erwig and Deling Ren. A rule-based language for programming software updates. In *RULE '02: Proceedings of the 2002 ACM SIGPLAN workshop on Rule-based programming*, pages 67–78, New York, NY, USA, 2002. ACM.

[83] Huw Evans, Malcolm Atkinson, Margaret Brown, Julie Cargill, Murray Crease, Steve Draper, Phil Gray, and Richard Thomas. The pervasiveness of evolution in grumps software. *Software–Practice & Experience*, 33(2):99–120, 2003.

[84] L. Feijs, R. Krikhaar, and R. Van Ommering. A relational approach to support software architecture analysis. *Software–Practice & Experience*, 28(4):371–400, 1998.

[85] F. Fiorvanti, G. Migliarese, and P. Nesi. Reengineering analysis of object-oriented systems via duplication analysis. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 577–586, Washington, DC, USA, 2001. IEEE Computer Society.

[86] Michael Fischer, Johann Oberleitner, Jacek Ratzinger, and Harald Gall. Mining evolution data of a product family. In *MSR '05: Proceedings of the 2005 International Workshop on Mining Software Repositories*, pages 1–5, New York, NY, USA, 2005. ACM.

[87] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 23, Washington, DC, USA, 2003. IEEE Computer Society.

[88] Marc E. Fiuczynski, Robert Grimm, Yvonne Coady, and David Walker. patch (1) considered harmful. In *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems*, pages 16–16, Berkeley, CA, USA, 2005. USENIX Association.

[89] Beat Fluri and Harald C. Gall. Classifying change types for qualifying change couplings. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 35–45, Washington, DC, USA, 2006. IEEE Computer Society.

[90] Beat Fluri, Michael Würsch, Martin Pinzger, and Harald C. Gall. Change distilling—tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):18, November 2007.

[91] George Forman, Kave Eshghi, and Stephane Chiocchetti. Finding similar files in large document repositories. In *KDD '05: Proceeding of the eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, pages 394–400, New York, NY, USA, 2005. ACM Press.

[92] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 2000.

[93] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 190, Washington, DC, USA, 1998. IEEE Computer Society.

[94] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1994.

[95] Reto Geiger, Beat Fluri, Harald C. Gall, and Martin Pinzger. Relation of code clones and change couplings. In *Proceedings of the 9th International Conference of Funtamental Approaches to Software Engineering (FASE)*, Lecture Notes in Computer Science, pages 411–425, Vienna, Austria, March 2006. Springer.

[96] Thomas Genssler and Volker Kuttruff. *Source-to-Source Transformation in the Large*, volume 2789. Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2003.

[97] Daniel M. German, Peter C. Rigby, and Margaret-Anne Storey. Using evolutionary annotations from change logs to enhance program comprehension. In *MSR '06: Proceedings of the 2006 International Workshop on Mining Software Repositories*, pages 159–162, New York, NY, USA, 2006. ACM Press.

[98] Shahram Ghandeharizadeh, Richard Hull, and Dean Jacobs. Heraclitus: elevating deltas to be first-class citizens in a database programming language. *ACM Transactions on Database Systems*, 21(3):370–426, 1996.

[99] Tudor Gîrba and Stéphane Ducasse. Modeling history to analyze software evolution: Research articles. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(3):207–236, 2006.

[100] Tudor Girba, Stéphane Ducasse, and Michele Lanza. Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 40–49, Washington, DC, USA, 2004. IEEE Computer Society.

[101] Michael Godfrey, Xinyi Dong, Cory Kapser, and Lijie Zou. Four interesting ways in which history can teach us about software. In *MSR '04: Proceedings of 2004 International Workshop on Mining Software Repositories*, 2004.

[102] Michael Godfrey and Qiang Tu. Tracking structural evolution using origin analysis. In *IWPSE '02: Proceedings of the International Workshop on Principles of Software Evolution*, pages 117–119, 2002.

[103] Michael W. Godfrey, Davor Svetinovic, and Qiang Tu. Evolution, growth, and cloning in linux, a case study. In *2000 CASCON workshop on Detecting Duplicated and Near Duplicated Structures in Large Software Systems*, 2000.

[104] Nicolas Gold and Andrew Mohan. A framework for understanding conceptual changes in evolving source code. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 431, Washington, DC, USA, 2003. IEEE Computer Society.

[105] Simon Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 385–402, New York, NY, USA, 2005. ACM Press.

[106] Carsten Gorg and Peter Weisgerber. Detecting and visualizing refactorings from software archives. In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 205–214, Washington, DC, USA, 2005. IEEE Computer Society.

[107] Carsten Görg and Peter Weißgerber. Error detection by refactoring reconstruction. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM Press.

[108] Judith E. Grass and Yih-Farn Chen. The c++ information abstractor. In *C++ Conference*, pages 265–278, 1990.

[109] Todd L. Graves, Alan F. Karr, J. S. Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, 2000.

[110] Todd L. Graves and Audris Mockus. Inferring change effort from configuration management databases. In *METRICS '98: Proceedings of the 5th International Symposium on Software Metrics*, page 267, Washington, DC, USA, 1998. IEEE Computer Society.

[111] W.G. Griswold. Coping with crosscutting software changes using information transparency. In *Reflection 2001: The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 250–265. Springer, 2001.

[112] William G. Griswold. *Program Restructuring as an Aid to Software Maintenance*. PhD thesis, University of Washington, 1991.

[113] William G. Griswold, Darren C. Atkinson, and Collin McCurdy. Fast, flexible syntactic pattern matching and processing. In *WPC '96: Proceedings of the 4th International*

Workshop on Program Comprehension, page 144, Washington, DC, USA, 1996. IEEE Computer Society.

[114] Bjorn Gulla, Even-Andre Karlsson, and Dashing Yeh. Change-oriented version descriptions in epos. *Software Engineering Journal*, 6(6):378–386, 1991.

[115] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. Codequest: Scalable source code queries with datalog. In *ECOOP'06: Proceedings of the 20th European Conference on Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 2–27, Berlin, Germany, 2006. Springer.

[116] W. Harrison, H. Ossher, S. Sutton, and P. Tarr. Concern modeling in the concern manipulation environment. In *Proceedings of the 2005 Workshop on Modeling and Analysis of Concerns in Software*, pages 1–5. ACM Press New York, NY, USA, 2005.

[117] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. Regression test selection for java software. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 312–326, New York, NY, USA, 2001. ACM.

[118] Ahmed E. Hassan and Richard C. Holt. The chaos of software development. In *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, page 84, Washington, DC, USA, 2003. IEEE Computer Society.

[119] Ahmed E. Hassan and Richard C. Holt. Predicting change propagation in software systems. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 284–293, Washington, DC, USA, 2004. IEEE Computer Society.

[120] Yasuhiro Hayase, Makoto Matsushita, and Katsuro Inoue. Revision control system using delta script of syntax tree. In *SCM '05: Proceedings of the 12th International Workshop on Software Configuration Management*, pages 133–149, New York, NY, USA, 2005. ACM Press.

[121] Paul Heckel. A technique for isolating differences between files. *Communications of the ACM*, 21(4):264–268, 1978.

[122] Scott A. Hendrickson and Andre van der Hoek. Modeling product line architectures through change sets and relationships. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 189–198, Washington, DC, USA, 2007. IEEE Computer Society.

[123] Johannes Henkel and Amer Diwan. Catchup!: capturing and replaying refactorings to support api evolution. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 274–283, New York, NY, USA, 2005. ACM.

[124] James D. Herbsleb, Audris Mockus, Thomas A. Finholt, and Rebecca E. Grinter. An empirical study of global software development: distance and speed. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 81–90, Washington, DC, USA, 2001. IEEE Computer Society.

[125] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Refactoring support based on code clone analysis. In *PROFES '04: Proceedings of 5th International Conference on Product Focused Software Process Improvement, Kausai Science City, Japan, April 5-8, 2004*, pages 220–233, 2004.

[126] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Aries: refactoring support tool for code clone. In *3-WoSQ: Proceedings of the third workshop on Software quality*, pages 1–4, New York, NY, USA, 2005. ACM.

[127] Yoshiki Higo, Yasushi Ueda, Shinji Kusumoto, and Katsuro Inoue. Simultaneous modification support based on code clone analysis. In *APSEC '07: Proceedings of the 14th Asia-Pacific Software Engineering Conference*, pages 262–269, Washington, DC, USA, 2007. IEEE Computer Society.

[128] Abram Hindle and Daniel M. German. Scql: a formal model and a query language for source control repositories. In *MSR '05: Proceedings of the 2005 International Workshop on Mining Software Repositories*, pages 1–5, New York, NY, USA, 2005. ACM.

[129] Reid Holmes and Gail C. Murphy. Using structural context to recommend source code examples. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 117–125, New York, NY, USA, 2005. ACM Press.

[130] R. Holt and J. Y. Pak. Gase: visualizing software evolution-in-the-large. In *WCRE '96: Proceedings of the 3rd Working Conference on Reverse Engineering*, page 163, Washington, DC, USA, 1996. IEEE Computer Society.

[131] Richard C. Holt. Structural manipulations of software architecture using tarski relational algebra. In *WCRE '98: Proceedings of the Working Conference on Reverse Engineering*, page 210, Washington, DC, USA, 1998. IEEE Computer Society.

[132] Richard C. Holt. Software architecture abstraction and aggregation as algebraic manipulations. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 5. IBM Press, 1999.

[133] Susan Horwitz. Identifying the semantic and textual differences between two versions of a program. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 1990. ACM.

[134] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, 1989.

[135] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Professional, 2000.

[136] James J. Hunt and Walter F. Tichy. Extensible language-aware merging. In *ICSM '02: Proceedings of the International Conference on Software Maintenance (ICSM'02)*, page 511, Washington, DC, USA, 2002. IEEE Computer Society.

[137] James J. Hunt, Kiem-Phong Vo, and Walter F. Tichy. Delta algorithms: an empirical analysis. *ACM Transactions on Software Engineering and Methodology*, 7(2):192–214, 1998.

[138] James W. Hunt and M.D. Mcilroy. An algorithm for differential file comparison. *Technical report*, 1976.

[139] James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, 1977.

[140] IClaudia-Lavinia Ignat and Moira C. Norrie. Operation-based versus State-based Merging in Asynchronous Graphical Collaborative Editing. In *Proceedings of the 6th International Workshop on Collaborative Editing Systems, Chicago, November*, 2004.

[141] Clemente Izurieta and James Bieman. The evolution of freebsd and linux. In *ISESE '06: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, pages 204–211, New York, NY, USA, 2006. ACM.

[142] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering Methodology*, 11(2):256–290, 2002.

[143] Daniel Jackson and David A. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *ICSM '94: Proceedings of the International Conference on Software Maintenance*, pages 243–252, Washington, DC, USA, 1994. IEEE Computer Society.

[144] Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In *International Conference on Aspect Oriented Software Development*, pages 178–187, 2003.

[145] Stan Jarzabek and Li Shubiao. Eliminating redundancies with a "composition with adaptation" meta-programming technique. In *ESEC/FSE-11: Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 237–246, New York, NY, USA, 2003. ACM.

[146] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society.

[147] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. Context-based detection of clone-related bugs. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 55–64, New York, NY, USA, 2007. ACM.

[148] J. Howard Johnson. Identifying redundancy in source code using fingerprints. In *CASCON '93: Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research*, pages 171–183. IBM Press, 1993.

[149] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[150] Cory Kapser and Michael W. Godfrey. Aiding comprehension of cloning through categorization. In *IWPSE '04: Proceedings of the Principles of Software Evolution, 7th International Workshop*, pages 85–94, Washington, DC, USA, 2004. IEEE Computer Society.

[151] Cory Kapser and Michael W. Godfrey. Improved tool support for the investigation of duplication in software. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 305–314, Washington, DC, USA, 2005. IEEE Computer Society.

[152] Cory Kapser and Michael W. Godfrey. "cloning considered harmful" considered harmful. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 19–28, Washington, DC, USA, 2006. IEEE Computer Society.

[153] Yoshio Kataoka, David Notkin, Michael D. Ernst, and William G. Griswold. Automated support for program refactoring using invariants. *ICSM*, 00:736, 2001.

[154] Chris F. Kemerer and Sandra Slaughter. An empirical approach to studying software evolution. *IEEE Transactions on Software Engineering*, 25(4):493–509, 1999.

[155] Mik Kersten and Gail C. Murphy. Mylar: a degree-of-interest model for ides. In *AOSD '05: Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, pages 159–168, New York, NY, USA, 2005. ACM Press.

[156] Roni Khardon. Learning horn expressions with logan-h. In *ICML '00: Proceedings of the Seventeenth International Conference on Machine Learning*, pages 471–478, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[157] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[158] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-oriented Programming*, volume 1241, pages 220–242. Lecture Notes in Computer Science 1241, 1997.

[159] Christoph Kiefer, Abraham Bernstein, and Jonas Tappolet. Mining software repositories with isparol and a software evolution ontology. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 10, Washington, DC, USA, 2007. IEEE Computer Society.

[160] Jung-Min Kim and Adam Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 119–129, New York, NY, USA, 2002. ACM.

[161] Miryung Kim, Lawrence Bergman, Tessa Lau, and David Notkin. An ethnographic study of copy and paste programming practices in oopl. In *ISESE '04: Proceedings of the 2004 International Symposium on Empirical Software Engineering*, pages 83–92, Washington, DC, USA, 2004. IEEE Computer Society.

[162] Miryung Kim and David Notkin. Using a clone genealogy extractor for understanding and supporting evolution of code clones. In *MSR '05: Proceedings of the 2005 International Workshop on Mining Software Repositories*, pages 1–5, New York, NY, USA, 2005. ACM.

[163] Miryung Kim and David Notkin. Program element matching for multi-version program analyses. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 58–64, 2006.

[164] Miryung Kim, David Notkin, and Dan Grossman. Automatic inference of structural changes for matching across program versions. In *ICSE '07: Proceedings of the 29th*

International Conference on Software Engineering, pages 333–343, Washington, DC, USA, 2007. IEEE Computer Society.

[165] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 187–196, New York, NY, USA, 2005. ACM.

[166] Sunghun Kim and Michael D. Ernst. Which warnings should i fix first? In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 45–54, New York, NY, USA, 2007. ACM.

[167] Sunghun Kim, Kai Pan, and Jr. E. James Whitehead. When functions change their names: Automatic detection of origin relationships. In *WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering*, pages 143–152, Washington, DC, USA, 2005. IEEE Computer Society.

[168] Sunghun Kim, E. James Whitehead, and Jennifer Bevan. Analysis of signature change patterns. In *MSR '05: Proceedings of the 2005 International Workshop on Mining Software Repositories*, pages 1–5, New York, NY, USA, 2005. ACM.

[169] Günter Kniesel, Jan Hannemann, and Tobias Rho. A comparison of logic-based infrastructures for concern detection and extraction. In *LATE '07: Proceedings of the 3rd Workshop on Linking Aspect Technology and Evolution*, page 6, New York, NY, USA, 2007. ACM.

[170] Andrew J. Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *International Conference on Software Engineering*, pages 344–353, 2007.

[171] Stanley Kok and Pedro Domingos. Learning the structure of markov logic networks. In *ICML '05: Proceedings of the 22nd international conference on Machine learning*, pages 441–448, New York, NY, USA, 2005. ACM.

[172] Raghavan Komondoor and Susan Horwitz. Semantics-preserving procedure extraction. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 155–169, New York, NY, USA, 2000. ACM Press.

[173] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *SAS*, pages 40–56, 2001.

[174] Raghavan Komondoor and Susan Horwitz. Effective, automatic procedure extraction. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, page 33, Washington, DC, USA, 2003. IEEE Computer Society.

[175] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 253–262, Washington, DC, USA, 2006. IEEE Computer Society.

[176] Jens Krinke. Identifying similar code with program dependence graphs. In *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering*, page 301, Washington, DC, USA, 2001. IEEE Computer Society.

[177] Jens Krinke. A study of consistent and inconsistent changes to code clones. In *WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering*, pages 170–178, Washington, DC, USA, 2007. IEEE Computer Society.

[178] David Lacey and Oege de Moor. Imperative program transformation by rewriting. In *CC '01: Proceedings of the 10th International Conference on Compiler Construction*, pages 52–68, London, UK, 2001. Springer-Verlag.

[179] David A. Ladd and J. Christopher Ramming. A*: A language for implementing language processors. *IEEE Transactions on Software Engineering*, 21(11):894–901, 1995.

[180] Bruno Lague, Daniel Proulx, Jean Mayrand, Ettore M. Merlo, and John Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *ICSM '97: Proceedings of the International Conference on Software Maintenance*, page 314, Washington, DC, USA, 1997. IEEE Computer Society.

[181] B. M. Lange and T. G. Moher. Some strategies of reuse in an object-oriented programming environment. In *CHI '89: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 69–73, New York, NY, USA, 1989. ACM.

[182] Michele Lanza. The evolution matrix: recovering software evolution using software visualization techniques. In *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*, pages 37–42, New York, NY, USA, 2001. ACM.

[183] Michele Lanza and Stéphane Ducasse. Polymetric views-a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, 2003.

[184] Janusz Laski and Wojciech Szermer. Identification of program modifications and its applications in software maintenance. In *ICSM 1992: Proceedings of International Conference on Software Maintenance*, 1992.

[185] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, pages 492–501, New York, NY, USA, 2006. ACM.

[186] V. I. Levenstein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady 10*, 10(8):707–710, 1966.

[187] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI*, pages 289–302, 2004.

[188] A. Lie, R. Conradi, T. M. Didriksen, and E.-A. Karlsson. Change oriented versioning in a software engineering database. *SIGSOFT Software Engineering Notes*, 14(7):56–65, 1989.

[189] Mark A. Linton. Implementing relational views of programs. In *SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 132–140, New York, NY, USA, 1984. ACM.

[190] Ernst Lippe and Norbert van Oosterom. Operation-based merging. *SIGSOFT Software Engineering Notes*, 17(5):78–87, 1992.

[191] Simone Livieri, Yoshiki Higo, Makoto Matsushita, and Katsuro Inoue. Analysis of the linux kernel evolution using code clone coverage. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 22, Washington, DC, USA, 2007. IEEE Computer Society.

[192] Simone Livieri, Yoshiki Higo, Makoto Matushita, and Katsuro Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 106–115, Washington, DC, USA, 2007. IEEE Computer Society.

[193] Benjamin Livshits and Thomas Zimmermann. Dynamine: finding common error patterns by mining software revision histories. In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 296–305, New York, NY, USA, 2005. ACM.

[194] Angela Lozano, Michel Wermelinger, and Bashar Nuseibeh. Evaluating the harmfulness of cloning: A change based experiment. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 18, Washington, DC, USA, 2007. IEEE Computer Society.

[195] Carine Lucas, Patrick Steyaert, and Kim Mens. Managing software evolution through reuse contracts. In *CSMR '97: Proceedings of the 1st Euromicro Working Conference on Software Maintenance and Reengineering (CSMR '97)*, page 165, Washington, DC, USA, 1997. IEEE Computer Society.

[196] Boris Magnusson, Ulf Asklund, and Sten Minör. Fine-grained revision control for collaborative software development. In *SIGSOFT '93: Proceedings of the 1st ACM SIGSOFT symposium on Foundations of Software Engineering*, pages 33–41, New York, NY, USA, 1993. ACM Press.

[197] Guido Malpohl, James J. Hunt, and Walter F. Tichy. Renaming detection. *Automated Software Engineering*, 10(2):183–202, 2000.

[198] Udi Manber. Finding similar files in a large file system. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 1–10, San Fransisco, CA, USA, 17–21 1994.

[199] David Mandelin, Doug Kimelman, and Daniel Yellin. A bayesian approach to diagram matching with application to architectural models. In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, pages 222–231, New York, NY, USA, 2006. ACM.

[200] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the api jungle. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 48–61, New York, NY, USA, 2005. ACM.

[201] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using pql: a program query language. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 365–383, New York, NY, USA, 2005. ACM.

[202] Jean Mayrand, Claude Leblanc, and Ettore Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*, page 244, Washington, DC, USA, 1996. IEEE Computer Society.

[203] Akhil Mehra, John Grundy, and John Hosking. A generic approach to supporting diagram differencing and merging for collaborative design. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering*, pages 204–213, New York, NY, USA, 2005. ACM.

[204] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. Schmidt, and B. Natarajan. Skoll: Distributed continuous quality assurance. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 459–468, Washington, DC, USA, 2004. IEEE Computer Society.

[205] Kim Mens, Tom Mens, and Michel Wermelinger. Maintaining software through intentional source-code views. In *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 289–296, New York, NY, USA, 2002. ACM.

[206] Kim Mens, Isabel Michiels, and Roel Wuyts. Supporting software development through declaratively codified programming patterns. *Expert Systems With Applications*, 23(4):405–413, 2002.

[207] Kim Mens, Roel Wuyts, and Theo D'Hondt. Declaratively codifying software architectures using virtual software classifications. In *TOOLS '99: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 33, Washington, DC, USA, 1999. IEEE Computer Society.

[208] Tom Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002.

[209] Tom Mens, Gabriele Taentzer, and Olga Runge. Detecting structural refactoring conflicts using critical pair analysis. *Electronic Notes in Theoretical Computer Science*, 127(3):113–128, 2005.

[210] Tom Mens and Tom Tourwe. A declarative evolution framework for object-oriented design patterns. In *ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 570, Washington, DC, USA, 2001. IEEE Computer Society.

[211] Tom Mens and Tom Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[212] Ettore Merlo, Giuliano Antoniol, Massimiliano Di Penta, and Vincenzo Fabio Rollo. Linear complexity object-oriented similarity for clone detection and software evolution analyses. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 412–416, Washington, DC, USA, 2004. IEEE Computer Society.

[213] Amir Michail. Data mining library reuse patterns using generalized association rules. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 167–176, New York, NY, USA, 2000. ACM.

[214] Lilyana Mihalkova and Raymond J. Mooney. Bottom-up learning of markov logic network structure. In *ICML '07: Proceedings of the 24th International Conference on Machine Learning*, pages 625–632, New York, NY, USA, 2007. ACM Press.

[215] Robert C. Miller and Brad A. Myers. Interactive simultaneous editing of multiple text regions. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 161–174, Berkeley, CA, USA, 2001. USENIX Association.

[216] Florian Mitter. Tracking source code propagation in software system via release history data and code clone detection. Master's thesis, Technical University of Vienna, 2006.

[217] Audris Mockus and James D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 503–512, New York, NY, USA, 2002. ACM Press.

[218] Audris Mockus and Lawrence G. Votta. Identifying reasons for software changes using historic databases. In *ICSM '00: Proceedings of the International Conference on Software Maintenance*, page 120, Washington, DC, USA, 2000. IEEE Computer Society.

[219] Audris Mockus and David M. Weiss. Globalization by chunking: A quantitative approach. *IEEE Software*, 18(2):30–37, 2001.

[220] Stephen Muggleton and Cao Feng. Efficient induction of logic programs. In *Proceedings of the 1st Conference on Algorithmic Learning Theory*, pages 368–381. Ohmsma, Tokyo, Japan, 1990.

[221] Stephen Muggleton, Luc, and De Raedt. Raedt. inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19:629–679, 1994.

[222] Bjrn P. Munch, Jens otto Larsen, Bjrn Gulla, Reidar Conradi, and Even andre Karlsson. Uniform versioning: The change-oriented model. In *Proceedings of the 4th International Workshop on Software Configuration Management (Preprint*, pages 188–196, 1993.

[223] Jonathan P. Munson and Prasun Dewan. A flexible object merging framework. In *CSCW '94: Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work*, pages 231–242, New York, NY, USA, 1994. ACM.

[224] Andrzej S. Murawski. About the undecidability of program equivalence in finitary languages with state. *ACM Transactions on Computational Logic*, 6(4):701–726, 2005.

[225] Gail C. Murphy, Mik Kersten, Martin P. Robillard, and Davor Cubranic. The emergent structure of development task. In *ECOOP '05: Proceedings of the 19th European Conference on Object-Oriented Programming*, 2005.

[226] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: bridging the gap between source and high-level models. In *SIGSOFT '95: Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 18–28, New York, NY, USA, 1995. ACM.

[227] Emerson R. Murphy-Hill, Philip J. Quitslund, and Andrew P. Black. Removing duplication from java.io: a case study using traits. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 282–291, New York, NY, USA, 2005. ACM.

[228] Eugene W. Myers. An o(ND) difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

[229] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 284–292, New York, NY, USA, 2005. ACM.

[230] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. In *MSR'05*, pages 2–6, 2005.

[231] George C. Necula. Translation validation for an optimizing compiler. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 83–94, New York, NY, USA, 2000. ACM.

[232] Eric Nickell and Ian Smith. Extreme programming and software clones. In *International Workshop on Detection of Software Clones*, 2003.

[233] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of moose: an agile reengineering environment. *SIGSOFT Software Engineering Notes*, 30(5):1–10, 2005.

[234] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, Urbana-Champaign, IL, USA, 1992.

[235] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. Leveraging field data for impact analysis and regression testing. In *ESEC/FSE-11: Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 128–137, New York, NY, USA, 2003. ACM Press.

[236] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. Scaling regression testing to large software systems. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth International Symposium on Foundations of Software Engineering*, pages 241–251, New York, NY, USA, 2004. ACM.

[237] Andy Ozment and Stuart E. Schechter. Milk or wine: does software security improve with age? In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, pages 7–7, Berkeley, CA, USA, 2006. USENIX Association.

[238] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 247–260, New York, NY, USA, 2008. ACM.

[239] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. Understanding collateral evolution in linux device drivers. *SIGOPS Operating Systems Review*, 40(4):59–71, 2006.

[240] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. Smpl: A domain-specific language for specifying collateral evolutions in linux device drivers. *Electronic Notes Theoretical Computer Science*, 166:47–62, 2007.

[241] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[242] David Lorge Parnas. Software aging. In *ICSE '94: Proceedings of the 16th International Conference on Software Engineering*, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[243] Santanu Paul and Atul Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475, 1994.

[244] Ulf Pettersson and Stan Jarzabek. Industrial experience with building a web portal product line using a lightweight, reactive approach. In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 326–335, New York, NY, USA, 2005. ACM.

[245] Martin Pinzger, Harald Gall, Michael Fischer, and Michele Lanza. Visualizing multiple evolution metrics. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 67–75, New York, NY, USA, 2005. ACM.

[246] Ranjith Purushothaman and Dewayne E. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, 31(6):511–526, 2005.

[247] J. Ross Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, 1990.

[248] Luc De Raedt and Luc Dehaspe. Clausal discovery. *Machine Learning*, 26(2-3):99–146, 1997.

[249] Shruti Raghavan, Rosanne Rohana, David Leon, Andy Podgurski, and Vinay Augustine. Dex: A semantic-graph differencing tool for studying changes in large code bases. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 188–197, Washington, DC, USA, 2004. IEEE Computer Society.

[250] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal: Very Large Data Bases*, 10(4):334–350, 2001.

[251] Damith C. Rajapakse and Stan Jarzabek. Using server pages to unify clones in web applications: A trade-off analysis. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 116–126, Washington, DC, USA, 2007. IEEE Computer Society.

[252] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Sieve: A tool for automatically detecting variations across program versions. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 241–252, Washington, DC, USA, 2006. IEEE Computer Society.

[253] Steven P. Reiss. Tracking source locations. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 11–20, New York, NY, USA, 2008. ACM.

[254] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: a tool for change impact analysis of java programs. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 432–448, New York, NY, USA, 2004. ACM.

[255] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *ESEC '97/FSE-5: Proceedings of the 6th European Conference held jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 432–449, New York, NY, USA, 1997. Springer-Verlag New York, Inc.

[256] Matthias Rieger, Stéphane Ducasse, and Michele Lanza. Insights into system-wide code duplication. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering*, pages 100–109, Washington, DC, USA, 2004. IEEE Computer Society.

[257] Romain Robbes. Mining a change-based software repository. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 15, Washington, DC, USA, 2007. IEEE Computer Society.

[258] Martin P. Robillard. Automatic generation of suggestions for program investigation. In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 11–20, New York, NY, USA, 2005. ACM.

[259] Martin P. Robillard and Gail C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 406–416, New York, NY, USA, 2002. ACM.

[260] Martin P. Robillard and Gail C. Murphy. Feat: a tool for locating, describing, and analyzing concerns in source code. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 822–823, Washington, DC, USA, 2003. IEEE Computer Society.

[261] Mary Beth Rosson and John M. Carroll. Active programming strategies in reuse. In *ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming*, pages 4–20, London, UK, 1993. Springer-Verlag.

[262] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, 1997.

[263] Filip Van Rysselberghe and Serge Demeyer. Reconstruction of successful software evolution using clone detection. In *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, page 126, Washington, DC, USA, 2003. IEEE Computer Society.

[264] Filip Van Rysselberghe and Serge Demeyer. Evaluating clone detection techniques from a refactoring perspective. In *ASE '04: Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, pages 336–339, Washington, DC, USA, 2004. IEEE Computer Society.

[265] Filip Van Rysselberghe and Serge Demeyer. Mining version control systems for facs(frequently applied changes). In *MSR '04: 2004 Internationl Workshop on Mining Software Repositories*, 2004.

[266] Filip Van Rysselberghe and Serge Demeyer. Studying software evolution information by visualizing the change history. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 328–337, Washington, DC, USA, 2004. IEEE Computer Society.

[267] Filip Van Rysselberghe and Serge Demeyer. Studying versioning information to understand inheritance hierarchy changes. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 16, Washington, DC, USA, 2007. IEEE Computer Society.

[268] Vibha Sazawal. *Connecting Software Design Principles to Source Code for Improved Ease of Change*. PhD thesis, University of Washington, 2005.

[269] Vibha Sazawal, Miryung Kim, and David Notkin. A study of evolution in the presence of source-derived partial design representations. In *IWPSE '04: Proceedings of the Principles of Software Evolution, 7th International Workshop*, pages 21–30, Washington, DC, USA, 2004. IEEE Computer Society.

[270] Thorsten Schäfer, Jan Jonas, and Mira Mezini. Mining framework usage changes from instantiation code. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 471–480, New York, NY, USA, 2008. ACM.

[271] David Shepherd, Zachary P. Fry, Emily Hill, Lori Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *AOSD '07: Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, pages 212–224, New York, NY, USA, 2007. ACM.

[272] Saurabh Sinha and Mary Jean Harrold. Analysis and testing of programs with exception handling constructs. *IEEE Transactions on Software Engineering*, 26(9):849–871, 2000.

[273] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM.

[274] Richard Snodgrass and Karen Shannon. Fine grained data management to achieve evolution resilience in a software development environment. In *SDE 4: Proceedings of the fourth ACM SIGSOFT symposium on Software development environments*, pages 144–156, New York, NY, USA, 1990. ACM Press.

[275] Amitabh Srivastava and Jay Thiagarajan. Effectively prioritizing tests in development environment. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 97–106, New York, NY, USA, 2002. ACM.

[276] Maximilian Stoerzer, Barbara G. Ryder, Xiaoxia Ren, and Frank Tip. Finding failure-inducing changes in java programs using change classification. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 57–68, New York, NY, USA, 2006. ACM.

[277] Margaret-Anne Storey, Casey Best, Jeff Michaud, Derek Rayside, Marin Litoiu, and Mark Musen. Shrimp views: an interactive environment for information visualization and navigation. In *CHI '02: Extended abstracts on Human factors in Computing Systems*, pages 520–521, New York, NY, USA, 2002. ACM Press.

[278] K.J. Sullivan, P. Chalasani, S. Jha, and V. Sazawal. *Software Design as an Investment Activity: A Real Options Perspective in Real Options and Business Strategy: Applications to Decision Making.* Risk Books, November 1999.

[279] Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *ICSE '99: Proceedings of the 21st International Conference on Software Engineering*, pages 107–119, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[280] Walter F. Tichy. The string-to-string correction problem with block moves. *ACM Transactions on Computer Systems*, 2(4):309–321, 1984.

[281] Michael Toomim, Andrew Begel, and Susan L. Graham. Managing duplicated code with linked editing. In *VLHCC '04: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, pages 173–180, Washington, DC, USA, 2004. IEEE Computer Society.

[282] Qiang Tu and Michael W. Godfrey. An integrated approach for studying architectural evolution. In *IWPC '02: Proceedings of the 10th International Workshop on Program Comprehension*, page 127, Washington, DC, USA, 2002. IEEE Computer Society.

[283] Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. On detection of gapped code clones using gap locations. In *Proc. of the Asia-Pacific Software Engineering Conference*, pages 327–336, 2002.

[284] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I.* Computer Science Press, 1988.

[285] Remco van Engelen. On the use of clone detection for identifying crosscutting concern code. *IEEE Transactions on Software Engineering*, 31(10):804–818, 2005. Student Member-Magiel Bruntink and Member-Arie van Deursen and Member-Tom Tourwe.

[286] Michael VanHilst and David Notkin. Decoupling change from design. In *SIGSOFT '96: Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 58–69, New York, NY, USA, 1996. ACM.

[287] Mathieu Verbaere, Ran Ettinger, and Oege de Moor. JunGL: a scripting language for refactoring. In Dieter Rombach and Mary Lou Soffa, editors, *ICSE'06: Proceedings of the 28th International Conference on Software Engineering*, pages 172–181, New York, NY, USA, 2006. ACM Press.

[288] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. *Domain-Specific Program Generation*, 3016:216–238, 2004.

[289] Kris De Volder. *Type-Oriented Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, 1998.

[290] Vera Wahler, Dietmar Seipel, Jurgen Wolff v. Gudenberg, and Gregor Fischer. Clone detection in source code by frequent itemset techniques. In *SCAM '04: Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE International Workshop*, pages 128–135, Washington, DC, USA, 2004. IEEE Computer Society.

[291] Andrew Walenstein, Nitin Jyoti, Junwei Li, Yun Yang, and Arun Lakhotia. Problems creating task-relevant clone detection reference data. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*, page 285, Washington, DC, USA, 2003. IEEE Computer Society.

[292] Jason T. L. Wang, Kaizhong Zhang, and Chung-Wei Chirn. The approximate graph matching problem. In *IEEE International Conference on Pattern Recognition*, pages 284–288, 1994.

[293] Zheng Wang. *Progressive profiling: a methodology based on profile propagation and selective profile collection*. PhD thesis, Harvard University, 2001. Adviser-Michael D. Smith.

[294] Zheng Wang, Ken Pierce, and Scott McFarling. BMAT - a binary matching tool for stale profile propagation. *Journal of Instruction-Level Parallelism*, 2, 2000.

[295] Peter Weissgerber and Stephan Diehl. Identifying refactorings from source-code changes. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 231–240, Washington, DC, USA, 2006. IEEE Computer Society.

[296] Bernhard Westfechtel. Structure-oriented merging of revisions of software documents. In *Proceedings of the 3rd International Workshop on Software configuration management*, pages 68–79, 1991.

[297] Deborah L. Whitfield and Mary Lou Soffa. An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems*, 19(6):1053–1084, 1997.

[298] Jingwei Wu, Richard C. Holt, and Ahmed E. Hassan. Exploring software evolution using spectrographs. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering*, pages 80–89, Washington, DC, USA, 2004. IEEE Computer Society.

[299] Sun Wu and Udi Manber. Agrep – a fast approximate pattern-matching tool. In *Proceedings USENIX Winter 1992 Technical Conference*, pages 153–162, San Francisco, CA, 1992.

[300] Roy Wuyts. Declarative reasoning about the structure of object-oriented systems. In *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, pages 112–124, 1998.

[301] Zhenchang Xing. Analyzing the evolutionary history of the logical design of object-oriented software. *IEEE Transactions on Software Engineering*, 31(10):850–868, 2005. Member-Eleni Stroulia.

[302] Zhenchang Xing and Eleni Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 54–65, New York, NY, USA, 2005. ACM.

[303] Tetsuo Yamamoto, Makoto Matsushita, Toshihiro Kamiya, and Katsuro Inoue. Measuring similarity of large software systems based on source code correspondence. In *Proceedings of 2005 Product Focused Software Process Improvement*, pages 530–544, 2005.

[304] Wuu Yang. Identifying syntactic differences between two programs. *Software – Practice & Experience*, 21(7):739–755, 1991.

[305] Wuu Yang, Susan Horwitz, and Thomas Reps. Detecting program components with equivalent behaviors. Technical Report CS-TR-1989-840, University of Wisconsin, Madison, 1989.

[306] Andrew Y. Yao. Cvssearch: Searching through source code using cvs comments. In *ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance*, page 364, Washington, DC, USA, 2001. IEEE Computer Society.

[307] Cemal Yilmaz, Arvind S. Krishna, Atif Memon, Adam Porter, Douglas C. Schmidt, Aniruddha Gokhale, and Balachandran Natarajan. Main effects screening: a distributed continuous quality assurance process for monitoring performance degradation in evolving software systems. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 293–302, New York, NY, USA, 2005. ACM.

[308] Annie T. T. Ying, Gail C. Murphy, Raymond Ng, and Mark Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, 2004.

[309] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *ESEC/FSE-7: Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 253–267, London, UK, 1999. Springer-Verlag.

[310] Xiangyu Zhang and Rajiv Gupta. Whole execution traces. In *MICRO 37: Proceedings of the 37th annual International Symposium on Microarchitecture*, pages 105–116, Washington, DC, USA, 2004. IEEE Computer Society.

[311] Xiangyu Zhang and Rajiv Gupta. Matching execution histories of program versions. In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 197–206, New York, NY, USA, 2005. ACM.

[312] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.

[313] Thomas Zimmermann and Peter Weißgerber. Preprocessing CVS data for fine-grained analysis. In *MSR '04: Proceedings of 2004 International Workshop on Mining Software Repositories*, pages 2–6, 2004.

[314] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.

[315] Lijie Zou and Michael W. Godfrey. Detecting merging and splitting using origin analysis. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*, page 146, Washington, DC, USA, 2003. IEEE Computer Society.

[316] Lijie Zou and Michael W. Godfrey. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, 2005.

Appendix A

COPY AND PASTE STUDY: EDIT LOG FORMAT

This chapter describes an edit log file format used by the logger and the replayer, which we developed for studying copy and paste programming practices (Chapter 3). The logger creates an XML file when an Eclipse IDE starts. It creates an *initial_document* node for each active editor in the Eclipse workbench and records its content snapshot. For each editing command, it creates a *command* node with (1) its type, COPY, CUT, PASTE, UNDO, REDO, or DELETE; (2) the file name of an edited document; (3) the range of selected text; (4) the length and offset of text entry; (5) the content of inserted text and selected text; and (6) a time stamp. For each keystroke, it creates a *typing* node with (1) its type, TYPING or BACKSPACE; (2) the file name of an edited document; and (3) the length, offset, and content of inserted text. When a programmer closes the Eclipse IDE, the logger creates a *final_document* node to save the editor's final snapshot. A sample log file is included below.

```
<editlog>
   <initial_document file="/PolyGlot/com/xj/XJImpl.java">
   XJImpl.java content.</initial_document>
   <command file="/PolyGlot/ext/xj/ast/GeneratedMethodInstance.java" type="COPY">
      <selection endline="33" length="119" offset="893" startline="32">
      // public void serialize(PrintStream stream) throws IOException;
      public static GeneratedMethodInstance SERIALIZE;</selection>
      <timestamp>Mon Aug 18 10:55:48 EDT 2003</timestamp>
   </command>
   <command file="/PolyGlot/ext/xj/ast/GeneratedMethodInstance.java"
   length="0" offset="1012" textLength="119" type="PASTE">
      <selection endline="33" length="0" offset="1012" startline="34"></selection>
      <timestamp>Mon Aug 18 10:55:49 EDT 2003</timestamp>
   //  public void serialize(PrintStream stream) throws IOException;
   public static GeneratedMethodInstance SERIALIZE;
```

```
    </command>
    <command file="/PolyGlot/polyglot/ext/xj/ast/GeneratedMethodInstance.java"
    length="19" offset="1060" textLength="0" type="DELETE">
       <selection endline="34" length="19" offset="1060" startline="34">
       throws IOException</selection>
       <timestamp>Mon Aug 18 10:56:13 EDT 2003</timestamp>
    </command>
    <typing file="/PolyGlot/ext/xj/ast/GeneratedMethodInstance.java"
       length="0" offset="1058" textlength="1" type="TYPING">t</typing>
    <typing file="/PolyGlot/polyglot/ext/xj/ast/GeneratedMethodInstance.java"
       length="1" offset="1045" textlength="0" type="BACKSPACE"></typing>
    <final_document file="/PolyGlot/polyglot/ext/jl/types/TypeSystemc.java">
    ... TypeSystemc.java content
    </final_document>
</editlog>
```

Appendix B

COPY AND PASTE STUDY: CODING SESSION ANALYSIS NOTE

This appendix contains an experimenter's note excerpt for one of the coding sessions.

Date: Mon, August 18th

Time : Mon Aug 18 10:55:48 EDT 2003 Mon Aug 18 10:58:36 EDT 2003

Participant: **** ******

Task and Objective: Adding the functionality called INSERT_AT_END.

Direct Observation: No

Instrumented Observation: Yes

Statistics: Available

Intention: A programmer copied A and pasted as B as A and B are semantically parallel concerns. Inside ****.java file, there are three functions. The system had two existing functions LOAD LOCAL, SERIALIZE that are similar to INSERT AT END.

At Event 1: Inside public class GeneratedMethodInstance

```
//  public static XMLNode load(String localFilename) throws IOException;
public static GeneratedMethodInstance LOAD_LOCAL;
//  public void serialize(PrintStream stream) throws IOException;
public static GeneratedMethodInstance SERIALIZE;
//  public void insertAtEnd(Element newElement);
public static GeneratedMethodInstance INSERT_AT_END;
```

At Event 53: Inside public static void initialize(XJTypeSystem ts)

```
assert(LOAD_LOCAL == null);
Flags publicStatic = Flags.PUBLIC.Static();
GeneratedClassType root = GeneratedClassType.getRoot();
LOAD_LOCAL =
        new GeneratedMethodInstance(
                ts,
```

```
                root,
                publicStatic,
                root,   //return type
                "load", //name
                Collections.singletonList(ts.String()), //arguments
                Collections.singletonList(ts.IOException())); //Exceptions
        SERIALIZE =
                new GeneratedMethodInstance(
                ts,
                root,
                Flags.PUBLIC,
                        ts.Void(),   //return type
                        "serialize", //name
                        Collections.singletonList(ts.PrintStream()), //arguments
                        Collections.singletonList(ts.IOException()); //Exceptions
        INSERT_AT_END =
                new GeneratedMethodInstance(
                        ts,
                        root,
                        Flags.PUBLIC,
                        ts.Void(),   //return type
                        "insertAtEnd", //name
                        Collections.singletonList(root), //arguments
                        Collections.singletonList(EMPTY_LIST); //Exceptions
}
```

At Event 83: Inside GeneratedClassType.java: public static void initStatic(XJTypeSystem_c ts, Package gp)

```
assert(ts != null && gp != null);
assert(s_root == null);
s_ts = ts;
s_package = gp;
//s_allElemClasses = new ArrayList(15);
s_root = new ElementClass(Position.COMPILER_GENERATED);
s_root.name = "XMLNode";
s_rootnode =
```

```
        ts.lang.nodeFactory().CanonicalTypeNode(
                Position.COMPILER_GENERATED,
                s_root);
GeneratedMethodInstance.initialize(ts);
s_root.addMethod(GeneratedMethodInstance.LOAD_LOCAL);
s_root.addMethod(GeneratedMethodInstance.SERIALIZE);
s_root.addMethod(GeneratedMethodInstance.INSERT_AT_END);
}
```

Table B.1: Copy and paste statistics

Type	SubType	Count	Avg Text Length
Operation		11	
	Copy	3	144.33
	Cut	0	
	Delete	3	17
	Paste	5	149.8
	Redo	0	
	Undo	0	
Typing		79	1.3544
	SingleType	71	1
	MutipleType	3	12
	Backspace	5	

Appendix C

COPY AND PASTE STUDY: AFFINITY DIAGRAMS

This chapter contains detailed sub parts of the affinity diagram in Figure 3.1. In these diagrams, light-yellow post-its represent C&P instance notes. Green post-its represent selected illustrating examples. Red post-its describe potential maintenance problems associated with the particular C&P patterns. Purple, blue, and yellow post-its respectively describe the first, second, and third level header-cards.

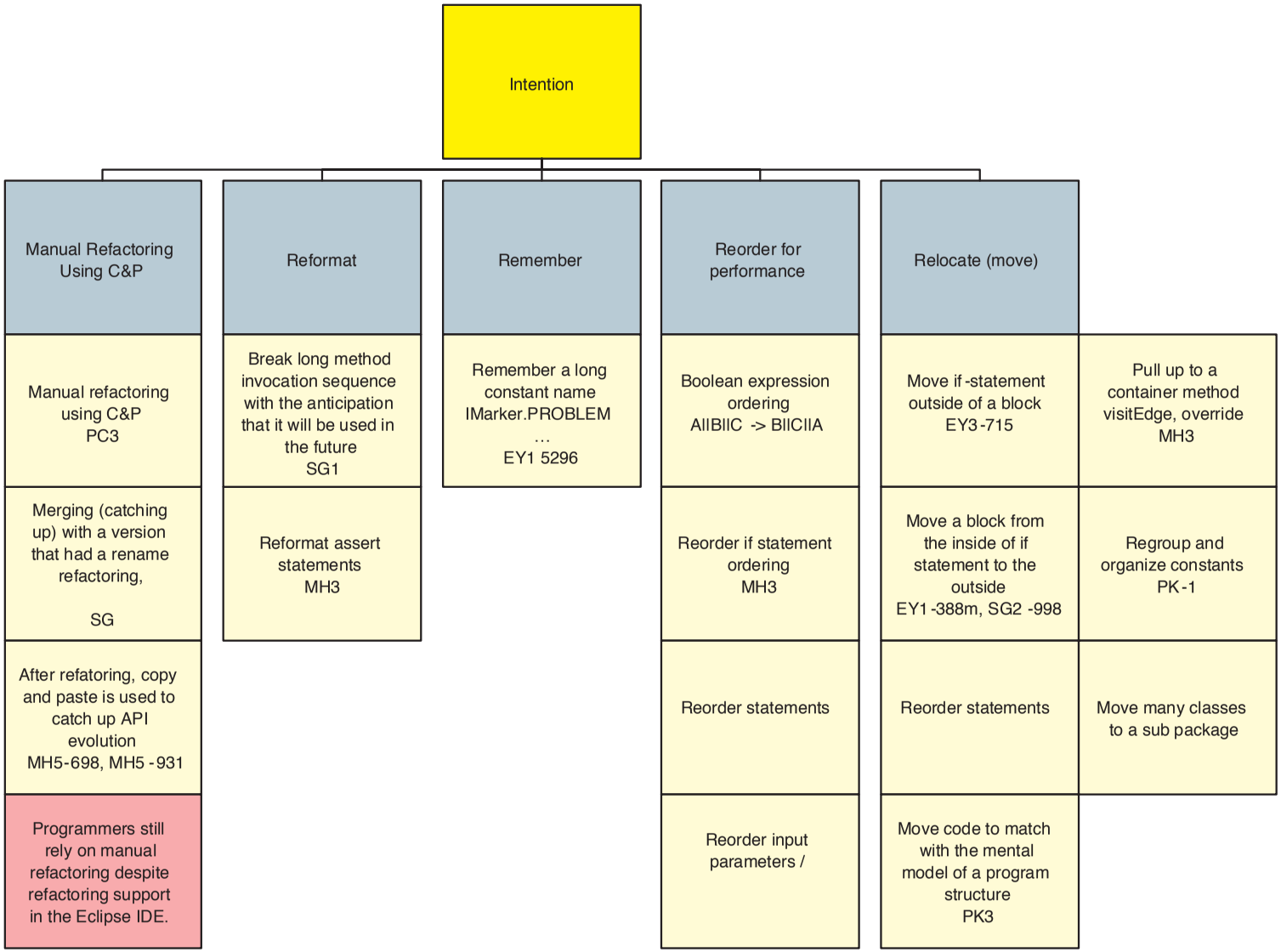

Figure C.1: Affinity diagram part 1: programmers' intentions associated with C&P

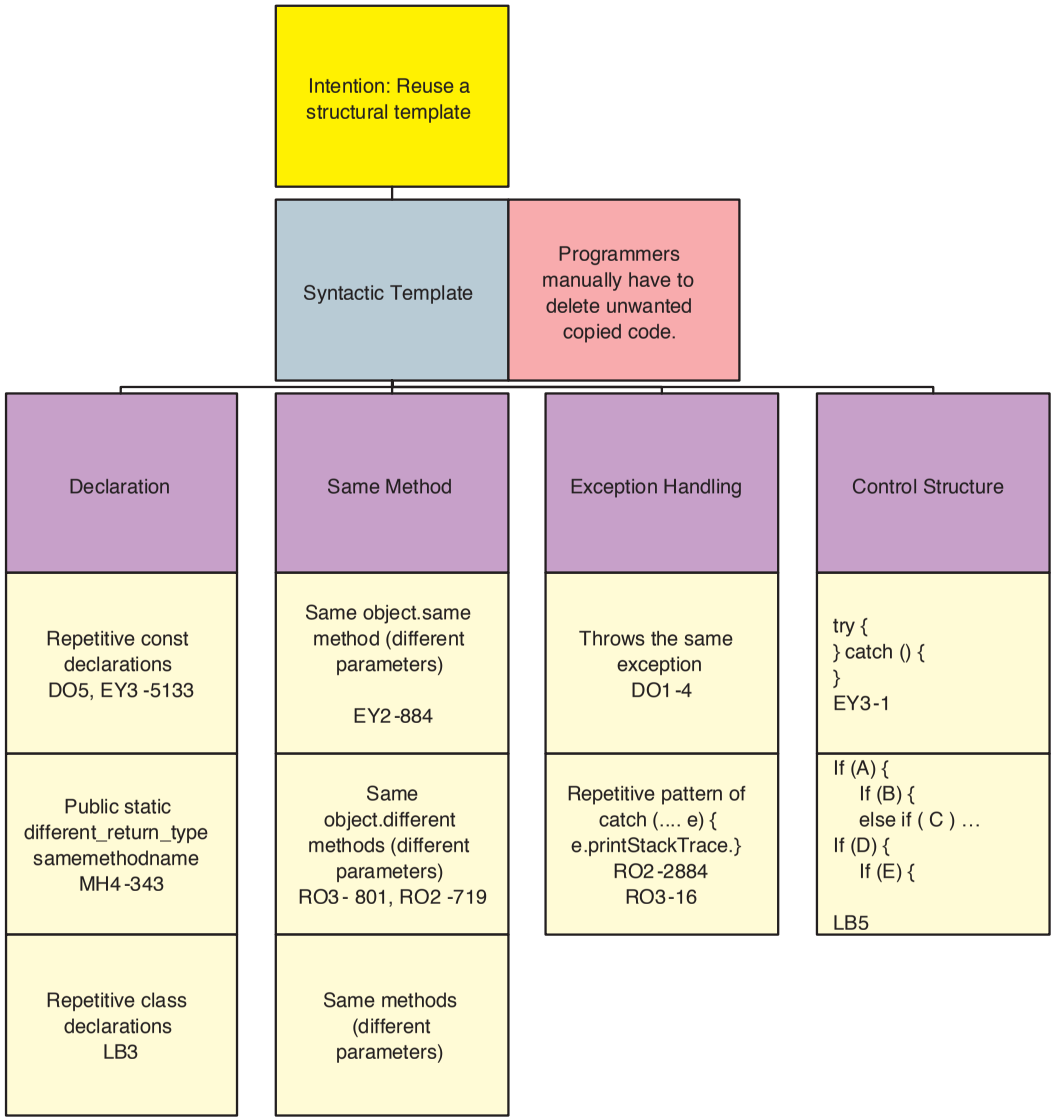

Figure C.2: Affinity diagram part 2: using copied code as a syntactic template

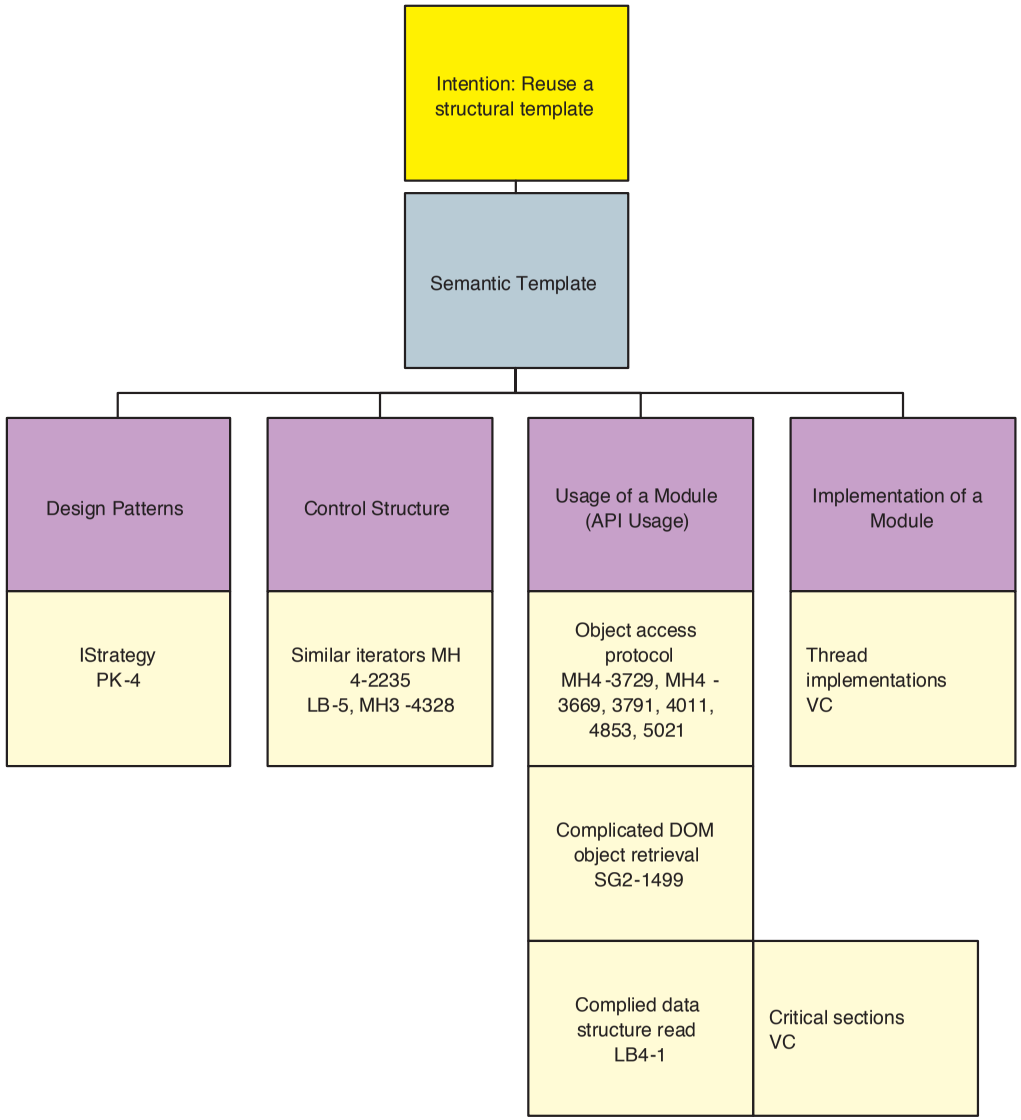

Figure C.3: Affinity diagram part 3: using copied code as a semantic template

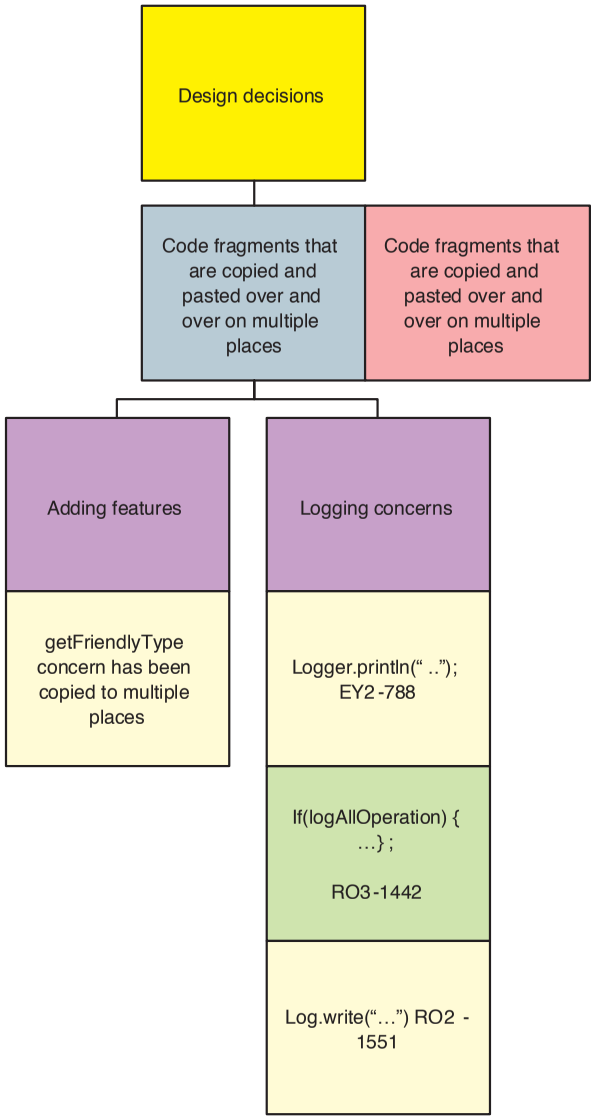

Figure C.4: Affinity diagram part 4: why is text copied and pasted repeatedly in multiple places?

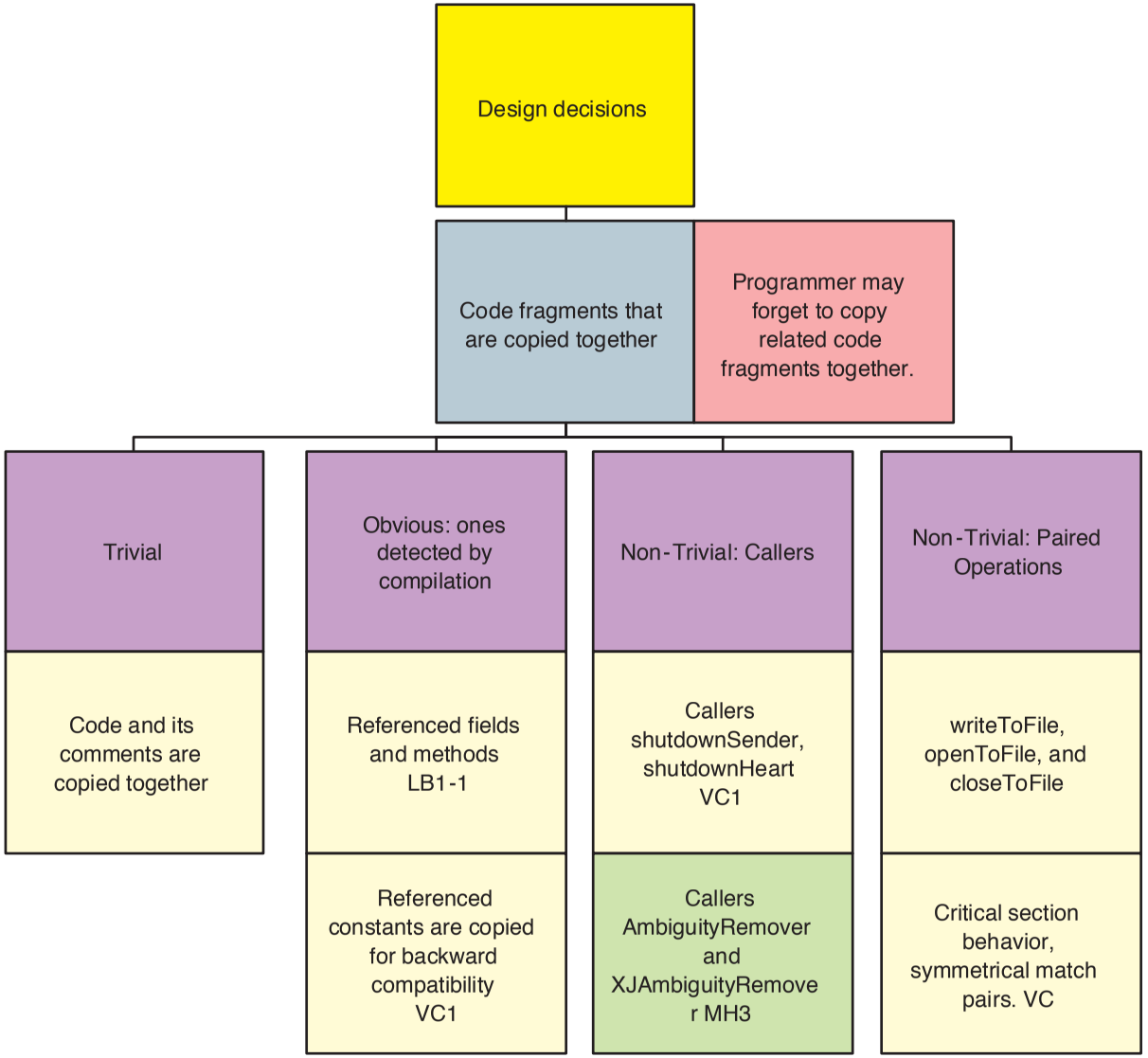

Figure C.5: Affinity diagram part 5: why are blocks of text copied together?

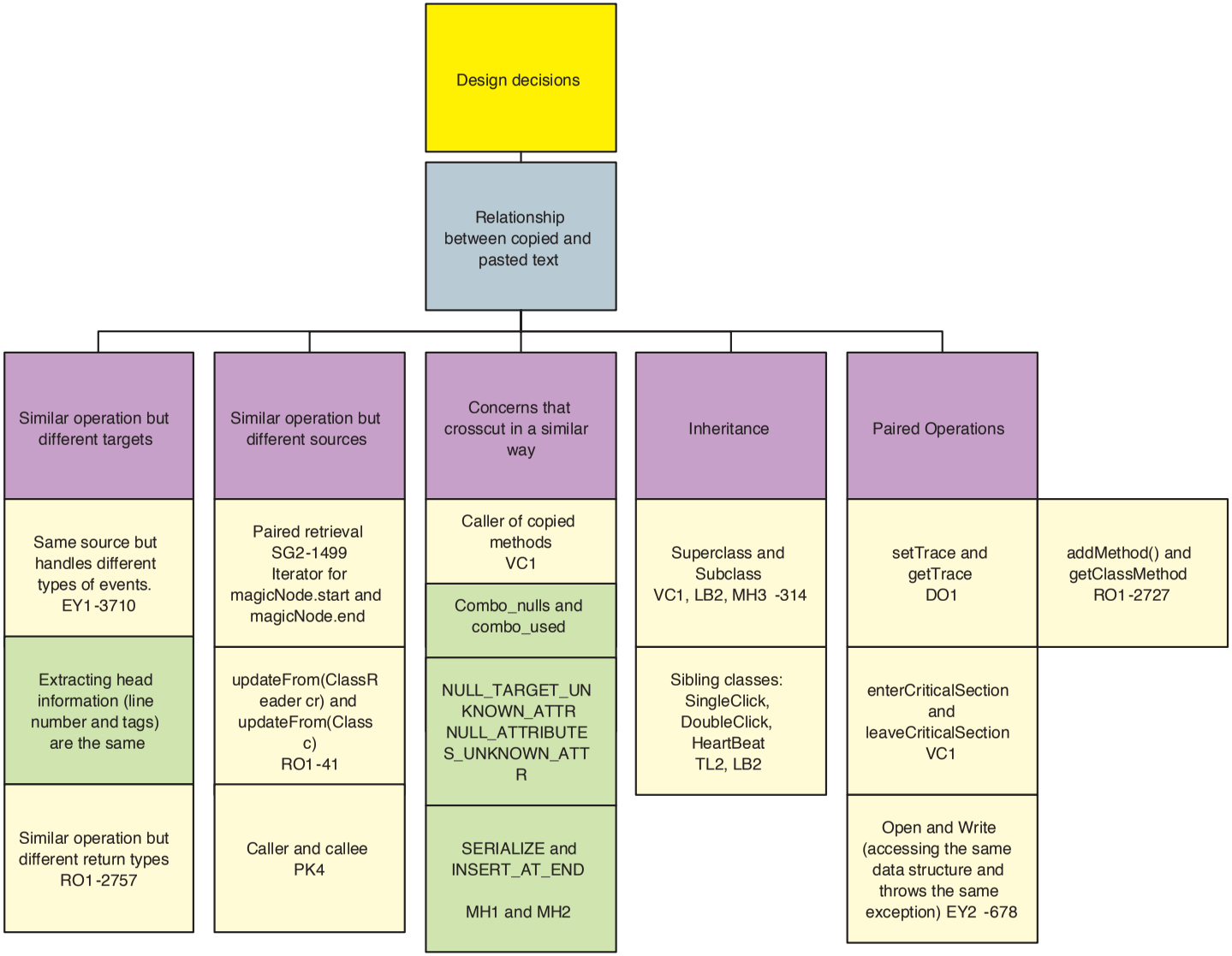

Figure C.6: Affinity diagram part 6: what is the relationship between copied and pasted text?

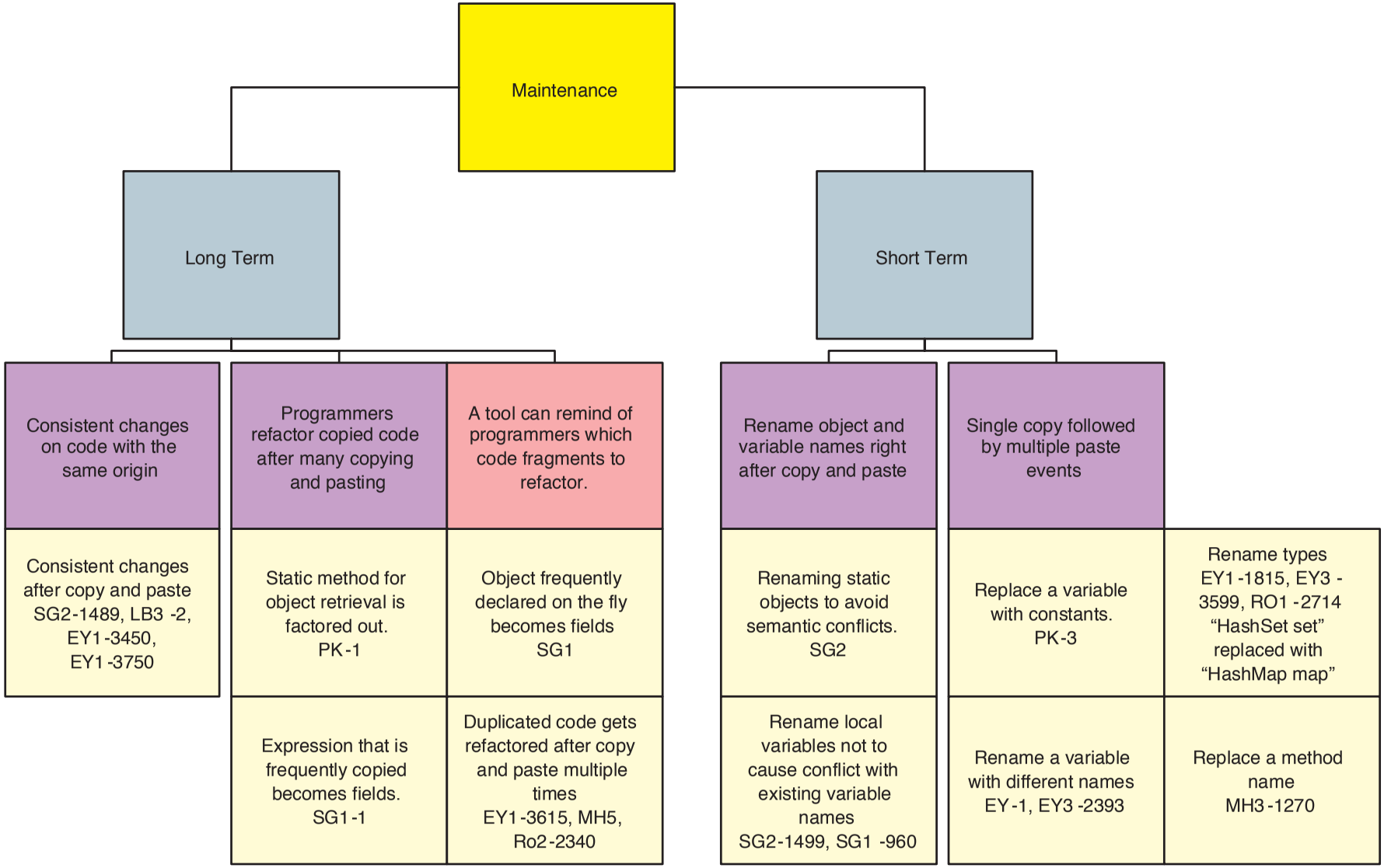

Figure C.7: Affinity diagram part 7: maintenance tasks for copied code

Appendix D

CLONE GENEALOGY STUDY: MODEL IN ALLOY CODE

```
module clonelineage
open std/ord

sig Text{}
fun similarhigh (t1:Text,t2:Text) {
 t1=t2 // exactly the same
}
fun similar (t1:Text,t2:Text) {
  #OrdPrevs(t1) = #OrdPrevs(t2) +1 || #OrdPrevs(t2) = #OrdPrevs(t1)+1 // similar
}
fun notsimilar (t1:Text, t2:Text) {
! similarhigh(t1,t2) && !similar(t1,t2) // not similar
}

//test functions
run similarhigh for 3
run similar for 3
run notsimilar for 3

sig Location{}
fun overlaphigh (o1:Location,o2:Location) {
  o1=o2 // exactly the same location
}
fun overlap (o1:Location,o2:Location) {
  #OrdPrevs(o1) = # OrdPrevs(o2) +1 || #OrdPrevs(o2) = #OrdPrevs(o1)+1 // partially overlap
}
fun notoverlap (o1:Location, o2:Location) {
  ! overlaphigh(o1,o2) &&
  !overlap(o1,o2) // does not overlap at all
}
```

```
// test functions
run overlaphigh for 3
run overlap for 3
run notoverlap for 3

// code is identified with its text and its location
sig Code{
  text: Text,
  location: Location
}
// clone group is a set of code with the same text within a clone group,
// every code has a unique location.
sig Group{
  group: set Code
}{
  all c1,c2:Code |  c1 in group && c2 in group => c1.text = c2.text
  # group > 1
 all disj c1,c2:Code | c1 in group && c2 in group => c1.location!=c2.location
}
// clone relationship is defined between one clone group in a old version and
// one clone groiup in a new version.

sig Relationship {
new : Group,
old : Group
}{
  new!=old
}

// clone genealogy is a graph which describes evolution of a code snippet. this graph
// is a direct graph where all nodes are connected by at least one edge if edges are present.
sig Genealogy {
nodes : set Group,
edges : set Relationship
}{
```

```
#nodes >0
#nodes>1 => (nodes = edges.new+edges.old)
}
fun testlineage () {
Genealogy= univ[Genealogy]
}

// evolution patterns
fun SAME (r:Relationship){
  similarhigh(r.new.group.text,r.old.group.text)
  all csn:Code | some cso:Code | csn in r.new.group =>
cso in r.old.group && overlaphigh(csn.location,cso.location)
  all cso:Code | some csn:Code | cso in r.old.group =>
csn in r.new.group && overlaphigh(csn.location,cso.location)
}
//run SAME for 5

fun SHIFT (r:Relationship) {
  similarhigh(r.new.group.text,r.old.group.text)
  some csn:Code | some cso:Code | csn in r.new.group
&& cso in r.old.group && overlap(csn.location,cso.location)
}
//run SHIFT for 5

fun ADD (r:Relationship) {
  (similarhigh(r.new.group.text, r.old.group.text) ||
   similar(r.new.group.text,r.old.group.text))
   some csn:Code | all cso:Code | cso in r.old.group =>
csn in r.new.group && notoverlap(csn.location,cso.location)
}
//run ADD for 5

fun SUBTRACT(r:Relationship) {
  (similarhigh(r.new.group.text, r.old.group.text) ||
   similar(r.new.group.text,r.old.group.text))
   some cso:Code | all csn:Code | csn in r.new.group =>
```

```
cso in r.old.group && notoverlap(csn.location,cso.location)
}
//run SUBTRACT for 5

fun CONSISTENT(r:Relationship) {
  similar(r.new.group.text,r.old.group.text)
  all cso:Code | some csn:Code | cso in r.old.group => csn in r.new.group &&
(overlap(csn.location,cso.location) || overlaphigh(csn.location,cso.location))
}
//run CONSISTENT for 5

fun INCONSISTENT(r:Relationship) {
  similar(r.new.group.text,r.old.group.text)
  some cso:Code | all csn:Code | csn in r.new.group =>
cso in r.old.group && notoverlap(csn.location,cso.location)
}
//run INCONSISTENT for 5

assert ALL_EXHAUSTIVE {
  all r:Relationship |
    !notsimilar(r.new.group.text,r.old.group.text) =>
ADD(r)|| SHIFT(r) || SAME(r) || SUBTRACT(r) || CONSISTENT(r) || INCONSISTENT(r)
}
//check ALL_EXHAUSTIVE for 5
//proved true

assert SAME_IN_SHIFT {
  all r:Relationship | SAME(r) => SHIFT(r)
}
//check SAME_IN_SHIFT for 5

assert SHIFT_IN_SAME {
  all r:Relationship | SHIFT(r) => SAME(r)
}
//check SHIFT_IN_SAME for 5
```

```
fun SHIFT_AND_SAME (r:Relationship) {
  SHIFT(r) && SAME(r)
}
//run SHIFT_AND_SAME for 5

assert INCONSISTENT_IN_SUBTRACT {
  all r:Relationship | INCONSISTENT(r)
  => SUBTRACT(r)
}
//check INCONSISTENT_IN_SUBTRACT for 5
```

Appendix E

CLONE GENEALOGY STUDY: GENEALOGY DATA FORMAT

```
<genealogy age="12" enddate="0206222147"
  falsePositiveReason="" falsepositive="false" startdate="0204221819">
  <sinknode c_chg="false" c_cpy="false" c_new="true" createdReason="">
    <locationmapping ng_members="2" og_members="1">
      <n_map from="0" index="0" score="100" />
      <n_map from="1" index="1" score="0" />
      <o_map from="0" index="0" score="100" />
    </locationmapping>
    <group id="10" versionString="0204221819">
        <codesnippet end_col="2" end_line="85"
      file="c:\dnsjava0204221819\org\xbill\DNS\CERTRecord.java"
      start_col="2" start_line="82" />
      <codesnippet end_col="2" end_line="79"
      file="c:\dnsjava0204221819\org\xbill\DNS\DSRecord.java"
      start_col="2" start_line="73" />
      </group>
    </sinknode>
  <evolutionpattern A="false" C="false" I="false" R="false" S="true" SH="false">
    <newgroup>
      <group id="10" versionString="0204302236">
          <codesnippet end_col="2" end_line="85"
        file="c:\dnsjava0204302236\org\xbill\DNS\CERTRecord.java"
        start_col="2" start_line="82" />
        <codesnippet end_col="2" end_line="79"
        file="c:\dnsjava0204302236\org\xbill\DNS\DSRecord.java"
        start_col="2" start_line="73" />
      </group>
    </newgroup>
    <oldgroup>
      <group id="10" versionString="0204221819">
```

```
          <codesnippet end_col="2" end_line="85"
          file="c:\dnsjava0204221819\org\xbill\DNS\CERTRecord.java"
          start_col="2" start_line="82" />
          <codesnippet end_col="2" end_line="79"
          file="c:\dnsjava0204221819\org\xbill\DNS\DSRecord.java"
          start_col="2" start_line="73" />
          </group>
        </oldgroup>
      <groupmapping locationscore="100" newgroupid="10"
      oldgroupid="10" textsimilarity="100">
        <locationmapping ng_members="2" og_members="2">
            <n_map from="0" index="0" score="100" />
            <n_map from="1" index="1" score="100" />
          <o_map from="0" index="0" score="100" />
          <o_map from="1" index="1" score="100" />
        </locationmapping>
      </groupmapping>
    </evolutionpattern>
    <sourcenode age="12" d_chg="true" d_old="false" d_rmv="false"
      disappearingReason="Diverged" factorReason="Header Prep" refactorable="false">
      <locationmapping ng_members="2" og_members="2">
        <n_map from="0" index="0" score="100" />
        <n_map from="1" index="1" score="100" />
        <o_map from="0" index="0" score="100" />
        <o_map from="1" index="1" score="100" />
      </locationmapping>
      <group id="9" versionString="0206222147">
        <codesnippet end_col="2" end_line="85"
        file="c:\dnsjava0206222147\org\xbill\DNS\CERTRecord.java"
        start_col="2" start_line="82" />
        <codesnippet end_col="2" end_line="79"
        file="c:\dnsjava0206222147\org\xbill\DNS\DSRecord.java"
        start_col="2" start_line="73" />
      </group>
    </sourcenode>
  </genealogy>
```

Appendix F

LSDIFF PREDICATES IN TYRUBA LANGUAGE

This chapter defines LSDiff predicates in Tyruba, typed logic programming languatge. SEMIDET, DET, NONDET, and MULTI mean that the expected number of query results is 0-n, 1, 0-n, and 1-n respectively.

```
package :: String
MODES
  (F) IS NONDET
END

type :: String, String, String
MODES
  (F,F,F) IS NONDET
END

field :: String, String, String
MODES
  (F,F,F) IS NONDET
END

method :: String, String, String
MODES
  (F,F,F) IS NONDET
END


return :: String, String
MODES
  (F,F) IS NONDET
  (B,F) IS SEMIDET
  (F,B) IS NONDET
```

```
END

subtype :: String, String
MODES
  (F,F) IS NONDET
  (B,F) IS NONDET
  (F,B) IS NONDET
END

accesses :: String, String
MODES
  (F,F) IS NONDET
  (B,F) IS NONDET
  (F,B) IS NONDET
END

calls :: String, String
MODES
  (F,F) IS NONDET
  (B,F) IS NONDET
  (F,B) IS NONDET
END

fieldoftype ::String, String
MODES
  (F,F) IS NONDET
  (B,F) IS SEMIDET
  (F,B) IS NONDET
END

typeintype :: String, String
MODES
  (F,F) IS NONDET
  (B,F) IS SEMIDET
  (F,B) IS NONDET
END
```

```
inheritedmethod :: String, String, String
MODES
  (F,F,F) IS NONDET
END

inheritedfield :: String, String, String
MODES
  (F,F,F) IS NONDET
END
```

Appendix G

JQUERY LOGIC QUERIES FOR GENERATING FACTBASES

1. package: "package(?X)"
2. type: "class(?X)" or "interface(?X)"
3. method: "method(?M)" or "constructor(?C)"
4. field: "field(?F)"
5. return: "returns(?C,?T)"
6. fieldoftype: "type(?F,?T)"
7. accesses: "accesses(?B,?F,?)"
8. calls: "calls(?B,?M,?)"
9. subtype: "subtype+(?Super,?Sub),class(?Sub)" or "subtype+(?Super,?Sub),interface(?Sub)"
10. inheritedfield: "inheritedField(?Sub,?SupF,?Sup)"
11. inheritedmethod: "inheritedMethod(?Sub,?SupM,?Sup)"

Appendix H

DEFAULT WINNOWING RULES

The following pre-defined rules winnow out some of the superfluous facts in ΔFB. These rules are tautologies; thus they should not be output to the user.

```
deleted_type(i,n,g) ^ past_typeintype(i,t)       => deleted_typeintype(i,t)
deleted_type(p,n,g) ^ past_subtype(p,s)          => deleted_subtype(p,s)
deleted_type(s,n,g) ^ past_subtype(p,s)          => deleted_subtype(p,s)
deleted_type(s,n,g) ^ past_inheritedfield(f,p,s) =>deleted_inheritedfield(f,p,s)
deleted_type(s,n,g) ^ past_inheritedmethod(f,p,s)=>deleted_inheritedmethod(f,p,s)
deleted_method(m,n,c)  ^ past_return(m,t)       => deleted_return(m,t)
deleted_method(m1,n,c) ^ past_calls(m1,m2)      => deleted_calls(m1,m2)
deleted_method(m2,n,c) ^ past_calls(m1,m2)      => deleted_calls(m1,m2)
deleted_field(f,n,c) ^ past_fieldoftype(f,t) => deleted_fieldoftype(f,t)
deleted_field(f,n,c) ^ past_accesses(f,m)      => deleted_accesses(f,m)
deleted_method(m,n,c) ^ past_accesses(f,m) => deleted_accesses(f,m)
deleted_package(p) ^ past_type(t,n,p)  => deleted_type(t,n,p)
deleted_type(t,n,g) ^ past_typeintype(i, t) ^ past_type(i, n1, g) => deleted_type(i, n1, g)
deleted_type(t,n,g)  ^ past_method(m,mn,t)   => deleted_method(m, mn, t)
deleted_type(t,n,g)  ^ past_field(f,fn,t)    => deleted_field(f, fn, t)

added_type(i,n,g) ^ current_typeintype(i,t)       => added_typeintype(i,t)
added_type(p,n,g) ^ current_subtype(p,s)          => added_subtype(p,s)
added_type(s,n,g) ^ current_subtype(p,s)          => added_subtype(p,s)
added_type(s,n,g) ^ current_inheritedfield(f,p,s) =>added_inheritedfield(f,p,s)
added_type(s,n,g) ^ current_inheritedmethod(f,p,s)=>added_inheritedmethod(f,p,s)
added_method(m,n,c)  ^ current_return(m,t)       => added_return(m,t)
added_method(m1,n,c) ^ current_calls(m1,m2)      => added_calls(m1,m2)
added_method(m2,n,c) ^ current_calls(m1,m2)      => added_calls(m1,m2)
added_field(f,n,c) ^ current_fieldoftype(f,t) => added_fieldoftype(f,t)
added_field(f,n,c) ^ current_accesses(f,m)      => added_accesses(f,m)
added_method(m,n,c) ^ current_accesses(f,m) => added_accesses(f,m)
```

```
added_package(p) ^ current_type(t,n,p)  => added_type(t,n,p)
added_type(t,n,g) ^ current_typeintype(i, t) ^ current_type(i, n1, g) => added_type(i, n1, g)
added_type(t,n,g)  ^ current_method(m,mn,t)   => added_method(m, mn, t)
added_type(t,n,g)  ^ current_field(f,fn,t)    => added_field(f, fn, t)
```

Appendix I

FOCUS GROUP SCREENER QUESTIONNAIRE

Hello, my name is Miryung Kim and I am a Ph.D student at the University of Washington, currently working on a new program differencing tool. I would like to understand current practices of using a program differencing tool and get comments and feedback on my new differencing tool that represents code changes semantically and structurally. All of your responses will be kept confidential.

1. How many years have you worked in software industry?

2. Do you have programming experience?

3. Are you familiar with the Java programming language?

4. If so, how many years of Java programming experience do you have?

5. Have you ever used *diff* (a program differencing tool that compares programs textually at line-level)?

6. Have you ever used version control systems (CVS, SVN, or SourceDepot, etc)?

7. If you answered YES on Question 5 or 6, how often do you use these tools?

   (a) more than 5 times a week

   (b) 1-5 times a week

   (c) less than once a week

   (d) less than once a month

8. Have you ever participated in code review (inspection) meetings?

9. How often do you examine program changes done by other software engineers?

    (a) more than 5 times a week

    (b) 1-5 times a week

    (c) less than once a week

    (d) less than once a month

10. What is the size of code bases that you regularly work with?

    (a) 10K-100K

    (b) 100K-500K

    (c) 500K+

11. Have you ever participated in focus groups?

The discussion will center on the current practices of code change reviews and your comments on our new program differencing tool. The session will last approximately 60 minutes, and refreshments will be served. In addition, you will receive a Starbucks gift card. Would you be interested in attending this group?

Appendix J

FOCUS GROUP DISCUSSION GUIDE

1. Introduction 12:00-12:05 (5 minutes)

   Greeting (Slide 1)

   Purpose two reasons: (1) to gather insights into the current practice and (2) to get your comments and feedback.

   Focus group format (Slide 2)

   Distribute name tags

   Reminder—Your active participation is a key to the success of this focus group.

   Agenda (Slide 3)

2. Discussion on the current practice 10 minutes, 12:05-12:15

   Diff (Slide 4)

   VCS is based on *diff* tools. (Slide 5)

   Side-by-side views. Eclipse IDE and SVN

   Most of you said that you have experience of using *diff*-based tools.

   Can you please tell me in which task contexts do you use diff?

   What do you like about diff?

   What do you not like about diff?

3. LSDiff Presentation 12:15 - 12:20 (5 minutes)

   Motivating scenarios (if necessary) (Slide 7)

Version control systems (Slide 8)

What we would like to have instead (Slide 9)

Algorithm overview (Slide 10)

Fact-base generation (Slide 11)

Compute the set-level difference (Slide 12)

Rule learning (Slide 13)

LSDiff output (Slide 14)

4. LSDiff Demo 12:20 - 12:25 (5 minutes) Pop up the Firefox browser. This looks like a regular *diff* result. In the bottom, there are a set of files, either modified, added or deleted. For each file, you can click on the file and see word-level differences. In this output, added text is yellow and deleted text is red strike-through. We manually augmented a HTML *diff* output using LSDiff output. LSDiff rules are complimentary to *diff* output. Now, let's come back to the top of the page. This is an overview generated by the LSDiff rules. We translated first order logic rules into English sentences over here. You can see statements like this.... Okay once you click on the rule 4. You can see more details about which structural differences are explained by this. For example. you can see that there many `host` fields are deleted. If you click on it, you can see the regular textual *diff* output. The main difference is that there is an annotation (a hyperlink back to the rule) which describes systematic changes. LSDiff currently targets Java, but it is potentially language independent. It accounts for renaming and moving. LSDiff is complementary to *diff*.

5. LSDiff Initial Evaluation (5 minutes) 12:25-12:30

   Initially what's your reaction?

   What do you like about LSDiff?

   What do you not like about LSDiff?

6. LSDiff Hands-on Trial (10 minutes) 12:30-12:40

   Go to website URL.

   You can ask questions.

7. LSDiff In-Depth Evaluation (15 minutes) 12:40-12:50

   What do you like about LSDiff?

   What do you dislike about LSDiff?

   What do you see as the potential benefits of LSDiff?

   In which context would you like to use this?

   In which context would you like not to use this?

Appendix K

FOCUS GROUP TRANSCRIPT

M. Mediator

L. Liaison

P1. SDE

P2. Senior SDE

P3. Senior Principal SDE

P4. SDET

P5. SDET

(12:02 Introduction)

M. Hi, my name is Miryung Kim. I am a Ph.D student at the University of Washington and I am doing research in software engineering. There are two things that I would like to focus in today's focus group. First, I would like to gather insights into how programmers understand code changes or in which kinds of situations programmers use program differencing tools such as *diff*. Second, I built a new program differencing tool called *Logical Structural Diff (LSDiff)*. This tool allows a new way of thinking about code changes and it is an automatic program differencing tool. So I would like to get your comments and feedback on this program differencing tool (*LSDiff*).

So let me just start by explaining the format of this focus group. Focus groups are often used in marketing research to test new concept or to get feedback on a new product in an early stage of product development. We are recording audio as you may be already aware of it. If you are uncomfortable with audio recording, you may leave the room. I will guarantee that your comments will be kept anonymous throughout this research and after we transcribe the audio tape, your identity will be kept confidential. Second, in a focus group, there is no right or wrong answer. Whatever opinion or feedback you have is valid. The way that we gather your feedback is through your quotes. So it is very important for

you to speak one at a time as clearly as possible.

(12:10 Finished Introduction)

M. First, we will have a discussion on the current practices of using *diff* or *diff*-based version control systems. Over here at X company, I was told that you use Perforce (P4). I am going to explain a little bit about how *LSDiff* works and then demonstrate some of its output. Next we will have a short discussion on *LSDiff*. There will be a chance for about 10 minutes that you can use your own web browser to browse a sample *LSDiff* output. I would like to finish this by leading a discussion on what you think about *LSDiff* and how it can change the picture of software development.

I believe that most of you are familiar with *diff*. If you compare two versions of program using *diff*, you get line-level textual differences per each file. For example, if you compare two versions of a program, for each file, you can see this kind of line-level differences. The point is not about how a tool represents the *diff* output either as side-by-side views, tree-views, or some other UIs. The core of *diff* is that it represents textual line-level differences. Because *diff* has been used as a basis for many version control systems such as CVS or Subversion, this output is probably what you see very often in your version control system. There might be some check-in comments, which describes changes at a high-level. It is often easy to identify high-level intent from these check-in comments, but they do not always map to code-level changes. Sometime like this case where the change consists of over 4000 lines of code changes across 9 different files, it is difficult for human beings to understand what programmers intended to do. I have several questions about the current practices of using *diff*, and you can chime in and have your say.

M. In which task contexts, do you examine code changes? Based on the survey, I understand that most of you look at code changes usually weekly and at minimum monthly basis, or even daily sometimes.

P1. The one that comes up the most frequently is a code review.

M. A code review?

P1. I will say that that's multiple times a day. After that, a lot of times when I'm looking and troubleshooting then and trying to figure out what's wrong with a piece of code, usually I'd like to know some context about when it changed, cause when something

broke on a certain day, it is nice to find out about what changed at that time.

M. Before the code review meetings, do people usually go over code changes or . . . ?

P1. For us it's not usually a review meeting. Someone makes changes and sends them out so that everybody can see it. So no meetings.

M. So again. . .

P2. I was about to say that you know we usually get code reviews via email. So if you don't know the code, you can see the history. *Diff* may help you understand how the code has changed.

P1. In troubleshooting context or what kinds of context?

P2. Yeah, usually in troubleshooting context.

P1. Yeah. It's hard to change something without knowing how it evolved and it is in the state that it is at.

M. Can someone tell me a little bit about what is troubleshooting? And what is the scenario?

P3. Basically what they are saying here is that you need to see generational changes, not just this file and that file, but how it has changed over time in the sense you know, as you went through a series of change motivations, how the code changed.

P1. Sometimes, I am looking certain lines, and I am like, I want to know who changed that line. In troubleshooting, what I'm talking about is you get an error and I think the code should be doing this, and as you are going through, in this particular context, this variable is not being set or they did not anticipate these situations. Is it that I got a bad input, or that they are not handling correctly, or what? And so in that context, you are kind of like, the only documentation you have is the code that you are staring at right there. So you wanna know how it got to the state that it is at.

P3. So *diff* tool doesn't do that, forcing source code to be annotated and to have comments or something like that. A lot of people put remarks at the end of the line with comments, or some sort of. This line got added because of that particular trouble ticket or something. It is annoying for a person to manually (do that). Most *diff* tools that I have experienced do not have a capacity like that.

M. Right.

P1. The reason that people put that kind of comments is like that, 'Yeah, this looks convoluted and stupid.' 'Don't change this because if you do, the same type of bugs will happen.'

M. So the common scenario is that you have some bugs, suddenly after the last two months, it seems like a bug popped up. You need to go over the past history to figure out which change might have led to this bug. Is that right?

P1. Yeah.

M. In this situation, what is very useful is an association to high-level change, right? A ticket or a bug fix that relates to that particular source lines?

P1. And what other changes were made at the same time.

P3. If you undo a bug fix, it will easily reintroduce the bug that was previously fixed. But without annotations, you don't discover that except by having the bug show up again. It's kinda painful. The *diff* tools that I use, they are all file-oriented.

P3. They don't have notions, which I think you are trying to address is that, they don't have semantic relationships between different files. I want to say, 'What did I change due to this problem in our company terminology, a trouble ticket?' It might have changed over 300 different files. I'd like to see not just one file but all 300 files that were included as part of that. It is scaling up for a single source file to into spacing in which changes—correlated change took place.

M. Actually I wanna dig a little deeper. You mentioned that, I think, there are two related problems. You mentioned that it's hard to understand what changed at the same time using *diff*, right? What are the files that changed together?

P1. Yeah. It's like one of the things that he said and I agree with is that, let's say that somebody refactored something and they took a big chunk of code and moved it from this file to that file, and looking at this file, you have no idea about its history and how it evolved. It evolved over here and then (it) got cut over here and pasted over here. So it's like you have no idea, and you have just lost all the contexts.

M. Yeah. . .

P3. Suppose that 2000 lines of code just disappeared, and you don't know where it went...

M. When refactoring happens, there's a discontinuation in the evolution history, because it is hard to trace code based on file names or using file granularity differencing tools.

M. I see.

M. Is there anything that you like about *diff*?

P3. Certainly, there are little changes that are hard for human beings to notice like changing a character in a line. You can't see that, but a tool can see that. That makes it very easy for you to notice a stuff like that.

P1. And it is also nice that it can filter out white space changes, too. Because again, you want to ignore those changes that have nothing to do with the context of the code. But *diff* is a great tool, though.

P2. *Diff* does a good job at figuring out what changed. It is amazing actually. It does a pretty good job of figuring out what you changed.

P1. P4 seems to be smart about language-level diffs. It's not like a typical *diff* where it is just a line-level. It can kind of see, especially when you are doing a merge, that's when diff really comes in handy in this respect. It figures out, 'This is a method encapsulated here, not just a collection of lines,' 'There's actually logical cohesion here.' It's not perfect about that and it would be nice if it does better.

M. It seems like having some sort of language semantics such as code elements—a class or a method or a function—having that information is useful for merging code or understanding code changes.

P2. Yeah, also suppose that you recognized a method also. Even despite IDEs, you may want to group similar methods together, etc. A chunk of lines moved here to here. *Diff* doesn't help with that and if a tool was aware of that (that would be great), not just simple methods added here and there.

(12:22 *LSDiff* Introduction)

M. Now let me go over and talk about my tool. The motivation of my tool is based on looking at *diff* or check-in comments, it is difficult for human beings to understand why a set of files changed together. It is especially difficult if the code changes involves renaming or moving of code. In general, if you are looking for a bug that might have happened because of inconsistent or incomplete changes, it is hard to identify missing changes by looking at

diff unless you read all of the differences. In the case of this refactoring example, to check whether this refactoring was complete or not, you have to read over 4000 lines of code. Most people do not have time to do that.

This is like what I would like to have. This kinds of high-level change descriptions. This is what developers would see; English translation of logic rules that our tool finds. So for example, by looking at this first line, I know that all `draw` methods take an additional `int` parameter. No matter how many lines have changed due to this API extension, I am going to represent such change as one rule. The other one is like there is some sort of refactoring happened, all `port` fields in the class of `ImplementationService` got deleted except `NameService` class. By generating hypotheses about systematic changes and evaluating the hypotheses, we can also find exceptions that violate its general change patterns. What our tool outputs is concise high-level change descriptions where you can easily note missed changes and, unlike English descriptions, it has explicit structure that code elements can relate back to the code changes.

P3. In this case where you see 'all', tell me if this tool does, it would be really useful if 9 out of 10 got changed, but one got forgotten. It doesn't know it got certainly forgotten, but with a high probability that this instance is kinda against the other ones. In fact, it is a missing change.

M. Yes, exactly. As you will see later in my tool demonstration, this tool does exactly what you said. My tool gives you the confidence about 9 out of 10 places changed this way; however, one place violated that systematic change patterns. My tool tells you explicitly where the exception is.

M. So now, I am going to briefly go over the algorithm. Basically, like any regular differencing tools, it takes two versions of a program, it automatically finds logic rules and facts that together explain structural differences between two versions of a program. We do this in three steps. We first use a program analysis tool to represent code as a set of facts in a database. For example, there is a class `GM` that extends `Car` class, that means (that) there is a type `GM`, there is `Car`, and there is an inheritance relationship between `Car` and `GM`. And suppose that there is a method `GM.run`. We also identify data accesses such as reading or writing fields or variables and method calls between different methods. Basically, we

represent a program as a set of code elements and their structural dependencies.

P3. You used the term 'program.' Are you defining this as a context of a program? What do you mean by a program here? A 'program' is ambiguous. Is it a temporary term you are using here? In this case, you are equating a program as a class?

M. Oh, no, no,, no. I was just showing you one class. What I mean by a program is... (Interrupted)

P3. A same class can be used by thirty different programs. What constitutes a program in a traditional sense?

M. So, in this case, if it is a Java program, then all the classes and all the packages. We analyze everything. Usually there's a unique identifier for each (class or package).

P3. You mean a program as a class with a `main` in it? Is it your definition of a program?

M. No every source code, except libraries. Basically we do a source code analysis. We reverse engineer ... (Interrupted)

P1. Basically what we call as modules or packages.

M. Right...

P3. You mean a set of directories that you get pointed to. Is that what you are saying?

P4. (Chimes in) I don't think it really needs a definition.

P3. I am just trying to understand the scope of this tool.

P1. It sounds like it is a common repository.

M. So for example, if you say a revision 200 and a revision 201, I am going to pull out all the source code from the source repository for those two versions.

P3. Oh, you walk through the entire repository to...

M. Yeah, it is not that difficult because you can analyze the differences incrementally. Suppose that you represent a program as a set of facts and (represent) the other version as a set of facts. We difference those facts using a set difference operator. So you can see what are the deleted methods and what are the dependencies between modules that were added or deleted. And in the step 3, I am not going to go into details, but the idea is that we generate hypotheses about high-level changes as logic rules. We systematically generate all of them, we evaluate them and accept the ones with high support and accuracy. So the set of fact-level differences is reduced to a set of logic rules that imply the (fact-level) differences.

So for example, what this example means is that in the old version, all the methods that called `SQL.exec` method now deleted calls to `DB.connect`. We learn this kinds of rules. This is a machine learning algorithm that automatically infers rules from a set of database facts. So suppose that the SQL library that you were using had a risk of SQL injection bug. So the management or somebody requested to remove all calls to this library and replace with the `SafeSQL`. This will be represented as a concise rule, 'all places that called `SQL.exec` now added calls to `SafeSQL.exec` except one.' So you can explicitly note where the exception happened. Or the other common case is that when you try to add a feature, it may involve scattered changes across a program. For example, when the change involves adding `setHost` methods to every subclass of `Service` class except `NameService` class. Again if we identify exceptions to general change patterns, you can spot inconsistent or incomplete changes more easily. Now I am going to show a sample output of my tool that was augmented with ... As you know, logic rules are not always easy to understand. But they can be always directly translated to English descriptions.

(12:30 Showing example output of *LSDiff*) There is a project called Carol. It has revisions. This is similar to what you may see in a typical version control system. It has a check-in message and it says, 'Common methods go into an abstract class, easy to maintain/fix.' What you see in the bottom is similar to what you will see in most version control systems. There are all the files that were either deleted or added, and if you click on one of the files, you can see line-level or word-level differences. Red strike-through means that they were deleted. Yellow-highlighted parts mean that they were added. This UI is not what I chose. I augmented an existing HTML *diff* output. Please don't criticize on the colors and particular choices of highlights. I just augmented my results on it. What is different from the existing diff output is that there is an overview about code changes. This is about over 4000 lines of code changes and it is very difficult to read. You can see the inferred rules. The first rule says, 'By this change six classes inherit many methods from the `AbstractRegistry` class.' Let's read another rule, the rule number four over here, 'All `host` fields in the classes that implement `NameService` class got deleted.'

P1. A lot of times, people will make corresponding changes. Does this handle configuration files?

M. It doesn't, but you're right.

P1. Especially when there are tight couplings between semantics and configuration.

P3. Can you *diff* those types of files?

M. Right right.

M. This example doesn't have config files, but you can still get basic *diff* results out of config files.

P3. These things are all predetermined? Are you hard-coding patterns? or Are you discovering them?

M. Discovering them. We systematically search and pick the rules that have high support.

P3. You could do this with pre-discovered rules...

P1. It does understand constructs like methods, parameters, etc., right?

M. Right.

(12:36 Discussion)

M. What do you like about the tool?

P4. This seems like a repository view. It helps with understanding differences. I was thinking more about individual file changes. Could you do this for class-level differences? Some changes are not that big, some here some there.

M. The example I showed you had lots of changes. If there were just a few changes, you might not be able to group them as a higher order pattern.

M. You can run this on any two versions, any version pair. It's more or less like *diff*.

P5. This looks great for big architectural changes but I'm wondering what it would give you if you had lots of random changes.

P3. It'll look for relationships that don't exist.

M. If there are no structural changes, you'll just see *diff*. This doesn't replace *diff*, it complements it.

P2. Does it make any sense to try to—does it use type information?

M. It uses type information.

P2. So the facts do contain types also.

M. Yes.

P3. Would you notice all `ints` being turned to `longs`?

M. Yes.

P3. There goes to my scoping question. All the `ints` go to `longs` in a particular class, or a method, or a package? These are different scopes. What type of scopes do you recognize?

M. All scopes that you mentioned.

P4. Who was your customer? Who did you think your customer would be?

M. I'm here to get that answer from you. Let's defer that a little bit.

(... missed some stuff here ...)

P1. I think, everyone with a large code base and individual developers who don't have time to go in and become intimately familiar with the code, and Different developers working on the same code base.

M. So you think it's more useful for looking at code you're not familiar with?

P1. Yeah, getting more context about the evolution of the code.

P3. It it lets you do things that would be so tough to do with *diff* that you don't even try. There is a big opportunity here that you don't cover—recognizing that refactoring should occur. All this code inherits from the same super-class. You should recognize that and suggest it.

P1. I actually disagree. I think this is the right scope. You want to recognize what went on, not to suggest what to do. One of the questions that came up earlier is that 'Did they forget this?' This tool isn't trying to infer... (Interrupted)

P3. You missed my point. My point isn't to say not to build this tool but you should build an additional tool which has the opposite of this functionality.

P1. I don't think that's a tractable problem though.

P4. I would definitely disagree.

P1. I don't think this is in the scope.

P3. She (the mediator) pruned the search tree to do this in a reasonable level. I think similar efforts are required in the other tool.

M. I understand what you are saying. There's a huge body of literature about program transformation or recommendation tools. I think that's not exactly the scope of our discussion today. I will be happy to talk more about those tools.

P3. This is a different tool.

M. Yeah, they are targeting different goals. They have different purposes.

M. So I wanna dig a little deeper. You said that this tool can be used for all cases that people use a regular *diff* for.

P1. I disagree with that.

P1. *Diff* is a specialized tool for what it does. There are sometimes that you wanna look at two files. I guess you could say that. You have to know something about what the file is saying. In this case, you've taken for example Java, and we know some of the syntax of Java, and we know how it fits, and what it is saying, etc. If you were looking at natural language descriptions, you cannot do that. But I am thinking text files and configuration files, there's a lot of other types of files where *diff* makes sense but this wouldn't make any sense at all.

P3. Is your intent to make *diff* obsolete? Nobody ever uses *diff* anymore? I don't know whether that's the goal here.

M. No. I am not trying to replace *diff*.

P3. You said that in the degenerate case you fall back to *diff*-like behavior. For example, if somebody gives you a new programming language that you did not know the context of, you basically fall down to *diff*, line-level comparison, is that true?

M. You mentioned about language dependency. This tool has two components. One is a language dependent component, which is a program analysis component that extracts facts. The later rule-inference component is language independent. You can potentially imagine using this tool for other languages like Perl, Python, you know C or any other language. I think that's a separate issue. I think what P1 was saying, you can correct me if I am wrong, that configuration files often do not have this kind of structure to extract.

P1. Yeah, well actually that's something else though. This has two things. It gives you the information about, it has some some awareness of, what the file has in it. A Java file and it is organized with these structures. You can't do that with configuration files. In fact I think this would be one of things that could be a great improvement. Most of the programs we do, it is not strictly in Java, but they are Java plus SQL plus XML. It would be beneficial if it could do that.

P1. We're talking about where this wouldn't be used.

M. Right.

P1. It wouldn't be used in case where if you were just working with one file. That won't make a lot of sense cause there again, unless the same change happens multiple times in the same file. In case where if you don't have rules about the structure of the file, it is not going to add anything that is helpful.

P2. This and *diff* has a very little overlap actually. Because this is a different level of abstraction, so this differencing is contextual. It is much more complementary to *diff*. So it gives you condensed information.

M. Can you tell me a little bit about in which cases that this complements *diff* well? So you may imagine yourself using it?

P2. I guess it is much a higher level of abstraction so it gives you context. You may start with the summary of changes and dive down to details using a tool like *diff*. *Diff* will print out details and this will give you overall things. It is complementary in different levels. I like the acronym *Logical Structural Diff*.

P5. I would like to use it with the XXXXXX SDK. I was writing tests for PR1 and now we are jumping to PR2. There were changes in the SDK. We don't know what exactly they are. If we could run that between PR1 and PR2, (Interrupted)

P5. The XXXXXX SDK ...

M. I am not fully understanding the context, but you are ...

P5. I write tests for the new XXXXXX SDK.

M. Is it a framework?

P5. Yeah, it's an E-commerce platform SDK. They released a PR1 and now a PR2. we wrote all our tests against PR1 and now we have to move them to PR2. How do we figure out those differences?

P1. That's a good usage case.

P5. I used to know their stuffs. But now their members of the classes are gone, so things are retyped or whatever, I need to find out all that.

M. So you are saying that, the component you depended on made a migration to the next milestone or release, and you want to understand what's the difference between them.

P1. Specifically with testing, this is where this can be really powerful. Because you can see and this is what you are saying that, you can see generic types of changes, you don't have to go by line by line, 'Oh, these are the types of changes that we made, and we can make tests for those types of changes.' This will make the tester's time much more efficient.

L. Particularly, the project that you were working on, a lot of changes are crosscutting; for example, we encrypted a market place ID for all our APIs for security reasons, those kinds of crosscutting concerns, if you know it, that will be great before you dive in and figure out how much work you actually need to do. So for scoping perspective, it is really useful, too.

P2. Maybe there's something simpler than this (*LSDiff*). Maybe just saying 'This field got deleted' would be useful... Even 'This field got deleted and that field got deleted' is still a useful summary. It does not need to know anything more than a structure. Even those things too with lots of changes, it will be still useful.

P1. Are you saying that you can do that without knowing the structure?

P2. I am saying that a part of this process is knowing systematic changes not just one change.

M. To answer your question, if there are systematic changes, we summarize it. If there were no systematic changes, we just output as is.

P2. I'm saying this might be useful on its own.

(12:52 Hands-on Demo)

M. If you have a browser, you can type this URL and then you can see the same output yourself.

P1. This is cool. I'd use it if we had one.

L. I didn't hear anyone say that they wouldn't use it

P5. This is a definitely winner tool.

P4. This is definitely good for code reviews. If you look at it, compare to looking at line changes and overall file changes, this gives you a lot more context behind the actual change. Oh, instead of looking at some guy replaced this variable and that variable...

P1. And then when you click through to drill down, you know what you're looking at. I am looking at who deleted this.

P2. You know what to expect. You can minimize the time that you are looking at code changes.

P1. This 'except' thing is great, because there's always a situation that you are thinking, 'Why is this one different?'

M. Where do you think the exception might be useful?

P1. I think a lot of times you're going to have 50 and 50. These 4 were changed and these 4 weren't. It just gives you more context. It just tells you that this thing is different for some reason. As I said before, you can't infer the intent of a programmer, but this is pretty close.

If I'm going to assume that this was a correct change, it might be interesting for me to look at the exceptions and contrast with those. Then I would understand better why these ones need to change. It's just more context.

M. Probably something to communicate. You might ask developers why they did not change this.

P1. Oh, you may see things in this package changed this way but things in this other package did not change this way. It may be related to business logic changes that the regular tools are not able to pick up.

P4. This looks very good for platform testing. Being able to see how people are using the platform. Suppose that I made a platform change and some guy is adapting to my platform change. I can definitely see easily whether other people are using my platform the way I want it.

P1. Third party situations, such as open source projects.

P1. If I'm following a code base, I'd like to read the change list that went through, or you can read something like this, to see how it's changed. This will give you more structure information.

M. Knowing what kinds of dependencies were added and deleted, and whether they are really invoking my service or not. That's sort of things that you are looking for?

P2. In services you deal with interfaces more, though I guess you can extend this to handle service definitions like add and remove arguments, etc.

M. So you're talking about API migration. The service has a new API.

P2. I don't think this will work with service definitions and migration. It's more decoupled... What I am saying is that this will work with library usage changes, but not service definitions or migrations.

M. What you're saying is that the service is not statically determined. It's dynamically loaded, and the service depends on the loaded data. So it's not easy to see the dependencies.

P2. Yeah, you have to process the interface definitions.

P1. Yeah, and it is not explicitly in the code base. It isn't brought in as a dependency at a compile time.

P4. I noticed one of the file had replaced a variable with a method. I didn't notice a statement about it.

P1. That's a standard idiom change.

M. The reason why it's not showing up is that there are rules that say if you intro a new method, all the places where you invoke the new method, we are currently suppressing that—everything implied by that new class is obvious.

P1. It is a standard idiom in Java.

M. You'll see it as a refactoring but I don't have a hyperlink in every line.

P4. It'd be good to see that I replaced `port` with `getPort()` in every class or whether there are exceptions.

P4. Yeah, I am thinking from QA perspectives.

VITA

Miryung Kim is from Seoul, Korea. She attended Seoul Science High School for gifted students and earned her bachelor's degree in Division of Computer Science, Department of Electrical Engineering and Computer Science at the Korea Advanced Institute of Science and Technology (KAIST) in 2001. She graduated as the top of all science and engineering undergraduate students in KAIST and received an award from the Secretary of Ministry of Science and Technology of Korea in 2001.

She will be working as an assistant professor at the University of Texas at Austin starting from January, 2009. Her research interests are software evolution, mining software repositories, and human aspects of software development.