Discovering and Representing Systematic Code Changes

Miryung Kim
Electrical and Computer Engineering
The University of Texas at Austin
Austin TX, USA
miryung@ece.utexas.edu

David Notkin
Computer Science & Engineering
University of Washington
Seattle WA, USA
notkin@cs.washington.edu

Abstract

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 after modification. Program differencing tools that support these tasks are limited in their ability to group related code changes or to detect potential inconsistencies in those changes. To overcome these limitations and to complement existing approaches, we built Logical Structural Diff (LSdiff), a tool that infers systematic structural differences as logic rules. LSdiff notes anomalies from systematic changes as exceptions to the logic rules.

We conducted a focus group study with professional software engineers in a large E-commerce company; we also compared LSdiff's results with textual differences and with structural differences without rules. Our evaluation suggests that LSdiff complements existing differencing tools by grouping code changes that form systematic change patterns regardless of their distribution throughout the code, and its ability to discover anomalies shows promise in detecting inconsistent changes.

1 Introduction

Suppose Alan asks Cathy, his team lead, to review his most recent software change. Alan's check-in message, *"Common methods go in an abstract class. Easier to extend/maintain/fix,"* suggests some questions to Cathy: "Was it indeed an *extract superclass* refactoring?" "Did Alan miss any parts of the refactoring?" and "Did Alan make some other changes along the way?" Cathy 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 some surrounding unchanged code, or perhaps even the entire codebase to determine if potential updates have been missed.[1]

Similar questions arise with respect to code change in other situations such as when programmers write check-in comments or when they try to determine why a program behaves differently from expected behavior.

Existing program differencing approaches 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 the programmer applied a high-level operation such as a refactoring or a crosscutting modification; existing approaches identify the low-level changes but not systematic relationships created by the programmer's implementation of the high-level change.

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 with respect to the program's structure. Similarly other differencing tools that work at different levels of abstraction (e.g., abstract syntax trees [31] and control flow graphs [1]) report individual differences without structure. Some approaches attempt to mitigate this problem by grouping the differences by physical locations (directories and files), by logical locations (packages, classes, and methods), by structural dependencies (define-use and overriding), or by similarity of names. However, they generally do not capture systematic changes along other dimensions. For example, Eclipse *diff* and UMLDiff [30] organize differences by logical locations but do not group changes that are orthogonal to a program's containment hierarchy. In contrast, techniques that identify crosscutting concerns [16] do not find regularities within a program's containment

[1]This scenario is based on a real example found in our evaluation (*carol* revision 430) and Ko et al.'s study [22].

hierarchy such as adding similar fields in the same class.

Because existing approaches do not recognize regularities in code changes, the programmer is burdened with detecting inconsistencies that may lead to potential bugs.

To overcome these limitations and to complement existing differencing approaches, we designed Logical Structural Diff (LSdiff) to concisely infer systematic changes and report exceptions that deviate from these systematic changes. LSdiff abstracts a program as code elements (e.g., methods and fields) and their structural dependencies (e.g., field-accesses and overriding). This abstraction is used to identify systematic changes that are characterized by consistent changes to code elements that share common structural characteristics such as accessing the same field or implementing the same interface.

LSdiff infers logic rules to discover and represent systematic structural differences. Any differences not represented by the inferred rules are retained as logic facts. Consider how LSdiff can help Cathy better understand Alan's changes and find answers to her questions by reporting the following results:[2]

Fact 1. AbsRegistry is a new class.
Rule 1. All host fields in NameSvc's subtypes were deleted except LmiRegistry class.
Rule 2. All setHost methods in NameSvc's subtypes were deleted except LmiRegistry class.
Rule 3. All getHost methods in NameSvc's subtypes deleted calls to SQL.exec except LmiRegistry class.

The first fact and the first two rules show that Alan created the `AbsRegistry` class by pulling up `host` fields and `setHost` methods from the classes implementing the `NameSvc` interface except from the `LmiRegistry` class. The third rule shows that he deleted calls to the `SQL.exec` method again except from the `LmiRegistry` class. The result in this form makes it simple for Cathy to double check with Alan about why `LmiRegistry` was left out and why he deleted calls to the `SQL.exec` method.

This rule-based approach is motivated by our previous work that identifies systematic refactorings at a method-header level [20]. LSdiff extends both its rule representation and inference algorithm to describe changes within a method body as well as at a field level. Furthermore, a new rule syntax relies on the use of conjunctive logic literals instead of regular expressions to allow programmers to understand shared structural characteristics, not only a shared naming pattern, e.g., "all `setHost` methods in `Service`'s subclasses" instead of "all methods with the name `*Service.setHost(*)`."

[2]For presentation purposes, we described the inferred rules in English. The details on the syntax and semantics of rules appear in Section 3.

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's outputs contain an average of 9.7 additional facts that cannot be found in naïve structural differences by including contextual information such as `LmiRegistry`'s `host` field not being deleted.

The rest of this paper is organized as follows. Section 2 presents related work. Section 3 and Section 4 describe how LSdiff represents and identifies systematic changes. Section 5 discusses the focus group study. Section 6 describes quantitative and qualitative assessments of LSdiff. Section 7 discusses LSdiff's limitations and future work, and Section 8 concludes.

2 Related Work

Program Differencing Tools. These tools compute the delta between two program versions based on various underlying representations including text lines, abstract syntax trees, control flow graphs, program dependence graphs, program structures, etc. To the best of our knowledge, in contrast to LSdiff, these tools do not identify systematic changes nor note anomalies in them [19].

Systematic Changes. LSdiff relies on the observation that high-level changes are often systematic at a code level. This observation arises from our studies of code clones [21] as well as from numerous other research efforts, primarily within the domain of refactorings and crosscutting concerns.

Refactorings are usually systematic in the sense that *multiple* code elements with *similar structural characteristics* undergo *similar* transformations [11]. Refactoring reconstruction tools find a collection of rename and move refactorings that map one program version to a successive version. (Refactoring reconstruction tools are described and compared elsewhere [7, 19]). LSdiff differs from these tools by identifying a broader class of systematic changes that include behavior-changing modifications.

Crosscutting concerns [17, 28] can be seen as another type of systematic change. Tarr et al. [28] argued that dominant design decisions make secondary design decisions such as logging be inserted across a program. Many techniques automatically or semi-automatically locate crosscutting concerns in a program using program structure information, clone detection techniques, natural language processing, or program change history [16]. For example, Dagenais et al. [6] automatically infer structural patterns among the participants of the same concern and represent such concerns using a rule syntax for tracing concerns over program versions. LSdiff differs from these by inferring general kinds of systematic changes that may or may not be crosscutting concerns and by detecting anomalies from systematic changes.

Identification of Related Changes. Several approaches use change history to identify code elements that tend to change together [12, 32, 33]. Like LSdiff, 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 [33] may report that methods `foo` and `bar` frequently changed together in the past but does not report that both methods are called by the same method `fun`. Crisp [4], a part of Chianti change impact analysis tool [26], computes AST-level structural differences and groups related differences using four predefined rules. While Crisp's goal is to create a compilable intermediate version for fault localization, LSdiff's goal is to help programmers understand code changes by recovering a latent systematic structure in program differences.

Logic-based Program Representation. Representing a program's code elements and structural dependencies as a set of logic facts has been used for decades. Approaches such as JQuery [15] or CodeQuest [13] automatically evaluate logic queries specified by programmers to assist program investigation. Mens et al.'s intentional view [24] allows programmers to specify concerns or design patterns using logic rules. Eichberg et al. [9] use Datalog rules to continuously enforce constraints on structural dependencies as software evolves. LSdiff differs 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 [14] to each of two program versions and use a set-difference operator to compute fact-level differences. Section 6 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 LSdiff's condensed rule representation.

Source Transformation Languages and Tools. To automate repetitive and error-prone program update, several tools allow programmers to write scripts that systematically modify a program to create a new version. For example, TXL [5] allows programmers to write and apply systematic changes to a program. Boshernitsan et al.'s iXJ [3] provides a visual language and a tool for describing and prototyping source transformations. Erwig and Ren [10] designed a rule-based language to express systematic updates in Haskell. Coccinelle [25] allows programmers to safely apply crosscutting updates to Linux device drivers. These tools focus on applying systematic changes to a program, while LSdiff in contrast focuses on recovering systematic changes from a pair of versions.

Framework Evolution. Dagenais and Robillard's SemDiff [7] 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. Consistent with its focus on framework evolution, SemDiff carries out a partial program analysis to find changes in the callers of a particular deleted API. 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 [27]. 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 combination of different types of structural characteristics such as subtyping and method-calls.

3 Delta Representation

LSdiff represents each program version using a set of predicates that describe code elements, their containment relationships, and their structural dependencies:

1. package (packageFullName)
2. type (typeFullName, typeShortName, packageFullName)
3. method (methodFullName, methodShortName, typeFullName)
4. field (fieldFullName, fieldShortName, typeFullName)

Table 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.

5. return (methodFullName, returnTypeFullName)
6. fieldoftype (fieldFullName, declaredTypeFullName)
7. typeintype (innerTypeFullName, outerTypeFullName)
8. accesses (fieldFullName, accessorMethodFullName)
9. calls (callerMethodFullName, calleeMethodFullName)
10. subtype (superTypeFullName, subTypeFullName)
11. inheritedfield (fieldShortName, superTypeFullName, subTypeFullName)
12. inheritedmethod (methodShortName, superTypeFullName, subTypeFullName)

LSdiff captures only structural differences, which are used to characterize systematic changes. Other types of differences, such as differences in control and temporal logic, are omitted as they have a relatively small role in discovering systematic changes; omitting these differences makes it impossible to use LSdiff's representation to reconstruct a version from another version and a delta.

Table 1 shows the fact-base representation of an example program. It shows two program versions (P_o and P_n), the fact-base representation of the new version (FB_n) and the fact-level differences between two versions (ΔFB). Using ΔFB, which is computed by taking the set difference of FB_o and FB_n, has two weaknesses. First, as an unstructured list of a potentially large number of facts, it is likely to be time-consuming to read and understand in many cases. Second, although it includes dependencies in the changed code fragments, it does not capture the surrounding context of those changed fragments. 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` happened in `Car`'s subtypes' `start` methods.

Our approach overcomes these two weaknesses by inferring logic rules from the union of 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, for example subtype("Car", "Kia") that signals a missing +calls("Kia.start", "Key.chk").

Each rule describes a systematic change by relating groups of facts in the three fact-bases. Our rules follow a Datalog rule syntax—a horn clause where a conjunction of logic literals in the antecedent implies a single literal in the consequent and all variables in the rules are universally quantified.[3] LSdiff further restricts rules to a set of styles that identify regularities about changes between the two versions rather than regularities in either of the versions alone. These restrictions, exhibited in Table 2 include: only deleted_* or added_* can appear in the consequent of a rule; and the antecedent of a rule cannot have predicates with different prefixes. 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_* $\Rightarrow$ deleted_*),
• consistent updates to clones by stating that all code elements with similar characteristics in the old version added similar code (e.g., past_* $\Rightarrow$ added_*),

[3]We chose Datalog rules because the evaluation of Datalog rules ensures termination and is faster than full Prolog.

Table 2. LSdiff rule styles

Rule Styles Antecedent ⇒ Consequent	Example Rule and Its Interpretation
past_* ⇒ deleted_*	past_calls(m, "DB.exec") ⇒ deleted_calls(m, "DB.exec") All methods that called `DB.exec` in the old version deleted calls to `DB.exec`.
past_* ⇒ added_*	past_accesses("Log.on",m) ⇒ added_calls(m, "Log.trace") All methods that accessed `Log.on` in the old version added calls to `Log.trace`.
current_* ⇒ added_*	current_method(m, "getHost", t) ∧ current_subtype("Svc", t) ⇒ added_method(m, "getHost", t) All `getHost` methods in the `Svc`'s subtypes are newly added ones.
deleted_* ⇒ added_* added_* ⇒ deleted_*	deleted_method(m, "getHost", t) ⇒ added_inheritedfield("getHost", "Service", t) All types that deleted `getHost` method inherit `getHost` from `Service` instead.

Table 3. LSdiff rule inference example

ΔFB′ (after inferring the first rule)	ΔFB″ (after inferring the second rule)
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")

• 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.

A rule r has a **match** f in ΔFB if f is a fact created by grounding r's consequent with constants that satisfy r's antecedent. A rule r has an **exception** if there is no match in ΔFB implied by a true grounding of its antecedent. We explicitly encode exceptions as a part of a rule to note anomalies to a systematic change.

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 1 presents the fact-bases and Table 3 shows the ΔFB reduction process through rule inference (See Section 4). By inferring a rule, "all accesses to the Key.on field were deleted from the old version (#1 in ΔFB′)," three deleted_accesses facts are replaced by the rule, reducing ΔFB to ΔFB′. The second rule, "all Car's subtypes' start methods added calls to the Key.chk method (#2 in ΔFB″)," has two added_calls fact matches and one exception. The remaining added_calls("Bus.start", "log") is output as is, because it does not form a systematic change pattern.

4 Algorithm

Our algorithm accepts two program versions and outputs logic rules and facts. It 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 [15], a logic query-based program investigation tool. JQuery analyzes a Java program using the Eclipse JDT Parser. We first compute ΔFB using set-difference. Using inferred method-header level refactorings from our previous work [20], we then remove spurious added_ and deleted_ facts caused by code renaming. For example, ΔFB in Table 1 does not contain −method("Key.out", ...) and +method("Key.output", ...) by accounting for the renaming.

Part 2. Rule Inference. This step infers rules that imply a group of added_ and deleted_ facts and outputs remaining unmatched facts in ΔFB.

Four 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), (3) k, the maximum number of literals in a rule's antecedent, and (4) β, which as described below is used to prune the search space. 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 determined by k. Increasing k allows our algorithm to find more contextual information from FB_o and FB_n; evaluating 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. 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. To determine the validity of each enumerated rule, we use the Tyruba logic query engine to find constant bindings to the rule's variables [29]. After selecting valid rules in each iteration, we remove all facts matched by rules from U, the set of unmatched facts in ΔFB, and proceed to the next iteration.

Some rules do not provide useful information about

Algorithm 1: LSdiff Rule Inference Algorithm

```
Input: $FB_o$, $FB_n$, $\Delta FB$, $m$, $a$, $k$, and $\beta$
Output: L and U
/* Initialize R, a set of ungrounded rules; L,
   a set of learned rules; and U, a set of
   facts in ΔFB that are not covered by L.  */
R := ∅, L := ∅, U := ΔFB;
U := applyDefaultWinnowingRules ($\Delta FB$, $FB_o$,
$FB_n$);    /* reduce ΔFB with default winnowing
rules.  */
R := createInitialRules ($m$);    /* create rules
with an empty antecedent by enumerating all
possible consequents.  */
foreach $i = 1 \ldots k$ do
    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 ∪ {g};
                U :=U − {g.matches};
            end
        end
    end
    R :=selectRules (R, $\beta$) ; /* select the best $\beta$
    rules in R  */
end
```

code changes as they are always true regardless of change content. 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 LSdiff from learning such rules, we have written 30 **default winnowing rules** by hand ([18], pp. 228–229) and remove the facts implied by these rules from U before rule inference.

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 [23]. 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 rank rules by their number of matches. The first tie-breaker prefers rules with fewer number of exceptions, as these rules are worth refining further. The second tie-breaker prefers rules whose variables are more general in terms of Java containment hierarchy: package > type > field = method > name.

Our rule inference algorithm is summarized in Algorithm 1, and the pseudo code of its subroutines appears in [18], pp. 143–145.

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 SET-COVER algorithm [2] and output the selected rules and the remaining facts in ΔFB.

5 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. A focus group is typically carried out in early stages of product design to seek target users' feedback on new products, concepts, or messages [8]. We selected this study method as a low-cost way to assess LSdiff's potential benefits before investing further efforts in its development.

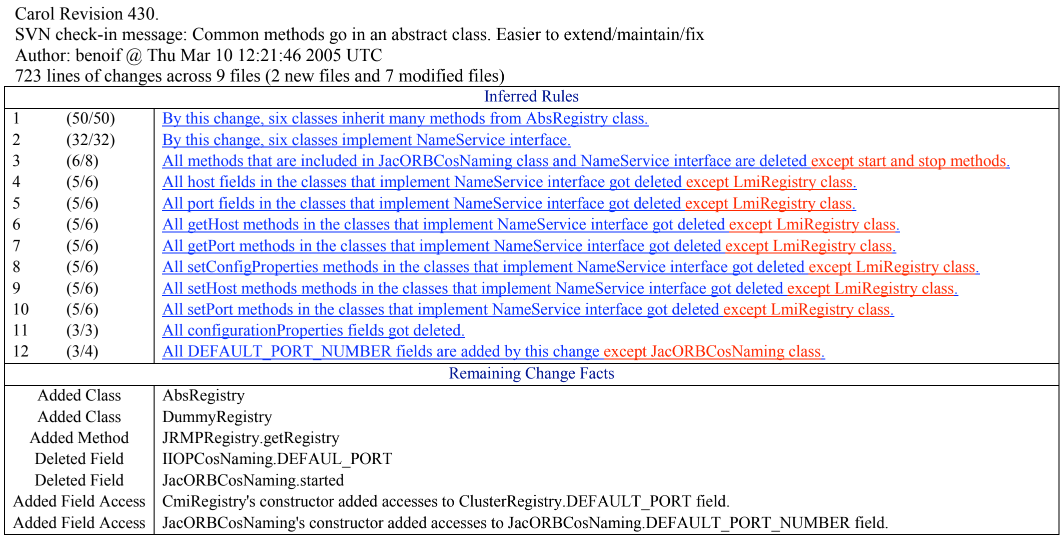
The goal of the focus group was to answer: (1) In which task contexts do programmers need to understand code changes? (2) What are difficulties of using program differencing tools such as *diff*? and (3) How can LSdiff complement existing uses of program differencing tools?

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 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. All sixteen participants responded to the questionnaire and five out of them attended the focus group: each had primary development responsibilities; each had industrial 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.

For one hour, the first author worked as the moderator and led the focus group through an introduction, a discussion on current practices for using *diff*, an overview, demonstration and brief evaluation of LSdiff, a hands-on trial of reviewing a sample LSdiff output, followed by an in-depth evaluation of LSdiff.

The hands-on trial used the output that LSdiff generated on *carol* project revision 430.[4] We chose

[4]http://users.ece.utexas.edu/~miryung/LSDiff/carol429-


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 1. Overview based on LSdiff rules and facts

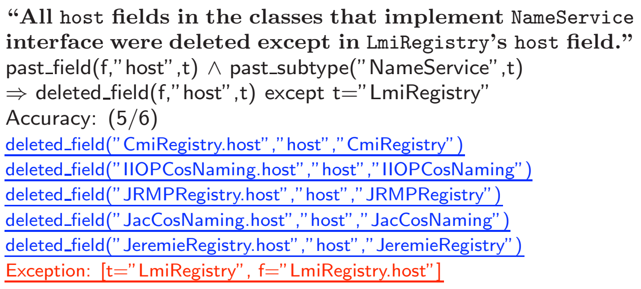

"All host fields in the classes that implement NameService interface were deleted except in LmiRegistry's host field."
past_field(f,"host",t) ∧ past_subtype("NameService",t)
⇒ deleted_field(f,"host",t) except t="LmiRegistry"
Accuracy: (5/6)
deleted_field("CmiRegistry.host","host","CmiRegistry")
deleted_field("IIOPCosNaming.host","host","IIOPCosNaming")
deleted_field("JRMPRegistry.host","host","JRMPRegistry")
deleted_field("JacCosNaming.host","host","JacCosNaming")
deleted_field("JeremieRegistry.host","host","JeremieRegistry")
Exception: [t="LmiRegistry", f="LmiRegistry.host"]

Figure 2. Rule description

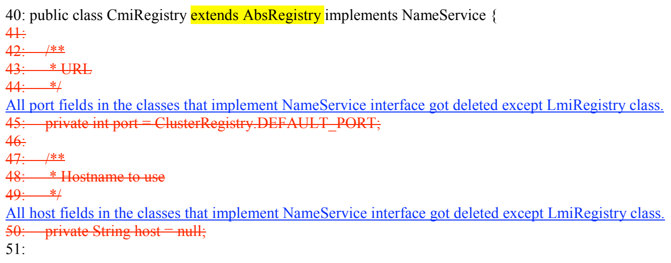

```
40: public class CmiRegistry extends AbsRegistry implements NameService {
~~41:~~
~~42:     /**~~
~~43:      * URL~~
~~44:      */~~
All port fields in the classes that implement NameService interface got deleted except LmiRegistry class.
~~45:     private int port = ClusterRegistry.DEFAULT_PORT;~~
~~46:~~
~~47:     /**~~
~~48:      * Hostname to use~~
~~49:      */~~
All host fields in the classes that implement NameService interface got deleted except LmiRegistry class.
~~50:     private String host = null;~~
51:
```

Figure 3. HTML diff augmented with rules

this revision because it is a conceptually simple change based on dispersed textual modifications of 723 lines across 9 files. By identifying the systematic nature of the change, LSdiff found 12 rules and 7 facts.

We used CSDiff[5] to prepare a regular word-level differencing result as a 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. Based on the rules found by LSdiff, we manually created an overview of systematic changes in the HTML document. Each rule was directly translated to an English sentence and presented as a hyperlink (See Figure 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 Figure 2).

In addition, by clicking each match, a programmer can navigate to corresponding word-level differences (See Figure 3). The corresponding rule description is inserted as a hyperlink in between the word-level differences so that a programmer can navigate to other related code changes (See lines 49–50 in Figure 3).

We audio-taped the discussion and had a note-taker transcribe the conversation. Our key findings are organized by the questions raised during the discussion.[6]

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."

". . . You need to see generational changes; not just this file

430.htm, computed using default parameters m=3, a=0.75, k=2

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

[6]The discussion guide, screener questionnaire results, and the focus group transcript are available in [18].

and that file but how they went through a series of change motivations..."

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

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 write test cases for [7, 27].

"I write tests for the new 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 wonder, 'why is this one different?' You can't infer the programmer's intent, but this is pretty close..."

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.

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 or small changes and that LSdiff may find uninteresting systematic changes. For example, "all types that declared `toString()` added constructors." is a valid systematic pattern but may be uninteresting to programmers.

Summary. 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. Though this study shows when and how LSdiff can complement existing use of differencing tools, it does not directly assess how much LSdiff helps programmers in the context of real tasks. Evaluating LSdiff's utility remains as future work.

6 Assessments

Comparison with Structural Deltas. Our goal is to answer the following questions by comparing LSdiff's result (LSD) with ΔFB, as ΔFB represents what an existing program differencing approach would produce at the same abstraction level.

(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 rules explain 90 facts out of 100 facts in ΔFB, the coverage of rules is 90%.

(2) How concisely does LSdiff describe structural differences 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 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 3 refers to three additional facts: subtype("Car","BMW"),

Table 4. Comparison with ΔFB

	FB_o	FB_n	ΔFB	Rule	Fact	Cvrg.	Csc.	Ad'l.
10 revision pairs in carol (carol.objectweb.org)								
Min	3080	3452	15	1	3	59%	2.3	0.0
Max	10746	10610	1812	36	71	98%	27.5	19.0
Med	9615	9635	97	5	16	87%	5.8	4.0
Avg	8913	8959	426	10	20	85%	9.9	5.5
29 release pairs in dnsjava (www.dnsjava.org)								
Min	3109	3159	4	0	2	0%	1.0	0.0
Max	7200	7204	1500	36	201	98%	36.1	91.0
Med	4817	5096	168	3	24	88%	4.8	0.0
Avg	5144	5287	340	8	37	73%	8.4	14.9
10 version pairs in LSdiff								
Min	8315	8500	2	0	2	0%	1.0	0.0
Max	9042	9042	396	6	54	97%	28.9	12.0
Med	8732	8756	142	1	11	91%	9.8	0.0
Avg	8712	8783	172	2	17	68%	11.2	2.3
three data sets above								
Med	6650	6712	132	2	17	89%	7.3	0.0
Avg	6632	6732	302	7	27	75%	9.3	9.7

(m=3, a=0.75, k=2)

Table 5. Comparison with textual delta

	Textual Delta				LSD	
Version	Files	CLOC	Hunk	% Touched	Rule	Fact
10 revision pairs in carol (carol.objectweb.org)						
Med	11	626	38	7%	5	16
Avg	13	1229	57	8%	10	20
29 release pairs in dnsjava (www.dnsjava.org)						
Med	9	354	40	17%	3	23
Avg	20	1159	65	28%	8	34
10 version pairs in LSdiff						
Med	6	227	15	6%	1	8
Avg	6	294	19	5%	1	13
three data sets above						
Med	9	344	31	8%	2	17
Avg	16	997	54	19%	7	27

(m=3, a=0.75, k=2)

subtype("Car", "GM"), and subtype("Car", "Kia").

For this comparison, we selected source projects *carol* and *dnsjava*, and our LSdiff itself as a subject program 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 supports different remote method invocation implementations; we selected 10 revisions with check-in comments that indicate non-trivial changes. Dnsjava is an implementation of domain name services in Java; we selected 30 releases. We also selected our LSdiff's first 10 version pairs—revisions that are at least 8 hours apart.

Table 4 shows the results for the three data sets with the default parameter settings m=3, a=0.75, k=2. 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 additional facts than ΔFB.

Comparison with Textual Deltas and Change Descriptions. 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 *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 Figures 1 and 3.

Table 5 shows the median and average size of TDs and LSDs for the subject programs. CLOC represents the number of added or deleted lines. Hunk represents the number of blocks with consecutive line changes, and % Touched represents the percentage of files that includes added or deleted text, computed as (# added files + # deleted files + 2 × # changed files) / (total # files in both versions). The more hunks there are, generally the harder it is to inspect a TD. Overall, when an average TD consists of 997 lines scattered across 16 files, LSdiff reports an average of 7 rules and 27 facts.

To give an idea about the quality of inferred LSdiff rules, we present representative rules along with the size of TD and check-in comments in Table 6.

The benefits of LSdiff appear to depend heavily on how systematic the change is. For example, *carol* 128-129, *"Bug fix, port number trace problem."* consists of 164 changed lines across 10 files. LSdiff finds 1 rule and 4 facts indicating that `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 TD, upon inspecting one corresponding class, she can probably skip five other classes.

When several different systematic changes are mixed with many non-systematic changes, LSdiff rules help programmers quickly understand the systematic changes and focus on other changes instead. For instance, programmers can discover that exception handling mechanism was modified to use `NamingExceptionHelper` by skimming 36 rules and 30 facts instead of over 4000 lines of changes.

7 Discussions

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

Table 6. Representative LSdiff rules and diff outputs

Source	File	CLOC	Rule	Fact	Excerpt from Change Description
Representative Rules and Their Interpretation					
carol 62-63	21	2151	12	71	A new simplified configuration mechanism (with bug id)
All fields in the `CarolConfiguration` class that are accessed from the `loadCarolConfiguration` method are newly added fields. All `Properties` type fields in the `CarolDefaultValues` class were deleted.					
carol 128-129	10	164	1	4	Port number trace problem. (with bug id references)
All `getPort` methods are newly added one.					
carol 421-422	14	4313	36	30	Refactoring of the spi package. (247 words long)
All calls to the `NamingExceptionHelper.create` method are newly added. All methods that called the `getResource` method no longer call the `printStackTrace` method.					
dnsjava 1.0.2-1.1	53	3362	29	174	Resolver.sendAsync returns an Object instead of an int.
All `sendAsync` methods return `Object` instead of `int`.					
LSdiff 20-21	11	637	6	19	(no check-in comment)
All classes in the `lsd` package that have the same name class in the `jquery` package were deleted.					

• For presentation purposes, we translated the inferred rules to English sentences.

Table 7. 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

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 k from 1 to 2. Table 7 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.

Threats to Validity. In terms of our focus group study, though it is common to commission an external, professional research vendor, the first author 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. We need to investigate 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. Vice versa, it may miss to identify systematic changes that programmers are expecting to see—in part due to selecting a subset of rules in Part 3 of our algorithm. Examining the precision and recall of LSdiff output is beyond the scope of our work-to-date largely because it is heavily task, project and programmer dependent. 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 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.

8 Conclusions

LSdiff discovers and represents systematic structural differences as logic rules. Each rule concisely describes a group of changes with 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.

Acknowledgment We thank Jonathan Beall for his contribution in implementing LSdiff and Marius Nita for his help in carrying out the focus group study. We thank Kris De Volder for providing JQuery, Rob DeLine for sharing motivating examples from his study, Alan Ho, Noah Snavely, Vibha Sazawal and Michael Bond for their comments on our draft, and Pedro Domingos, James Fogarty, Günter Kniesel, Stanley Kok, and Gail Murphy for discussions that helped us refine our ideas.

References

[1] T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In *ASE*, pages 2–13. IEEE, 2004.

[2] E. Balas and M. Padberg. Set Partitioning: A Survey. *SIAM Review*, 18:710–760, 1976.

[3] M. Boshernitsan, S. L. Graham, and M. A. Hearst. Aligning development tools with the way programmers think about code changes. In *CHI*, pages 567–576. ACM, 2007.

[4] O. C. Chesley, X. Ren, and B. G. Ryder. Crisp: A debugging tool for java programs. *ICSM*, 00:401–410, 2005.

[5] J. R. Cordy. The txl source transformation language. *Sci. Comput. Program.*, 61(3):190–210, 2006.

[6] B. Dagenais, S. Breu, F. W. Warr, and M. P. Robillard. Inferring structural patterns for concern traceability in evolving software. In *ASE*, pages 254–263. ACM, 2007.

[7] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *ICSE*, pages 481–490. ACM, 2008.

[8] H. Edmunds. *Focus Group Research Handbook*. McGraw-Hill, 2000.

[9] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini. Defining and continuous checking of structural program dependencies. In *ICSE*, pages 391–400. ACM, 2008.

[10] M. Erwig and D. Ren. A rule-based language for programming software updates. In *Proceedings of the ACM SIGPLAN workshop on Rule-based programming*, pages 67–78, New York, NY, USA, 2002. ACM.

[11] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 2000.

[12] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *ICSM*, pages 190–197, 1998.

[13] E. Hajiyev, M. Verbaere, and O. de Moor. Codequest: Scalable source code queries with datalog. In *ECOOP*, volume 4067, pages 2–27, 2006.

[14] R. C. Holt. Structural manipulations of software architecture using tarski relational algebra. In *WCRE*, page 210. IEEE, 1998.

[15] D. Janzen and K. D. Volder. Navigating and querying code without getting lost. In *AOSD*, pages 178–187, 2003.

[16] A. Kellens, K. Mens, and P. Tonella. A survey of automated code-level aspect mining techniques. *Transactions on Aspect-Oriented Software Development IV*, pages 143–162, 2007.

[17] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, volume 1241, pages 220–242. LNCS, 1997.

[18] M. Kim. *Analyzing and Inferring the Structure of Code Changes*. PhD thesis, University of Washington, 2008.

[19] M. Kim and D. Notkin. Program element matching for multi-version program analyses. In *MSR*, pages 58–64, 2006.

[20] M. Kim, D. Notkin, and D. Grossman. Automatic inference of structural changes for matching across program versions. In *ICSE*, pages 333–343, 2007.

[21] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy. An empirical study of code clone genealogies. In *ESEC/SIGSOFT FSE*, pages 187–196, 2005.

[22] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *ICSE*, pages 344–353, 2007.

[23] S. Kok and P. Domingos. Learning the structure of Markov logic networks. *ICML*, pages 441–448, 2005.

[24] K. Mens, T. Mens, and M. Wermelinger. Maintaining software through intentional source-code views. *SEKE*, pages 289–296, 2002.

[25] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in linux device drivers. In *EUROSYS*, pages 247–260. ACM, 2008.

[26] X. Ren, F. Shah, F. Tip, B. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of Java programs. In *OOPSLA*, pages 432–448, Vancouver, BC, Canada, October 26–28, 2004.

[27] T. Schäfer, J. Jonas, and M. Mezini. Mining framework usage changes from instantiation code. In *ICSE*, pages 471–480. ACM, 2008.

[28] P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *ICSE*, pages 107–119. IEEE, 1999.

[29] K. D. Volder. *Type-Oriented Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, 1998.

[30] Z. Xing and E. Stroulia. UMLDiff: an algorithm for object-oriented design differencing. In *ASE*, pages 54–65, 2005.

[31] W. Yang. Identifying syntactic differences between two programs. *Software - Practice and Experience*, 21(7):739–755, 1991.

[32] A. T. T. Ying, G. C. Murphy, R. Ng, and M. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Softw. Eng.*, 30(9):574–586, 2004.

[33] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *ICSE*, pages 563–572, 2004.