Discovering and Representing Logical Structure in Code Change

Miryung Kim, Jonathan Beall, David Notkin
Computer Science & Engineering
University of Washington
Seattle WA, USA
{miryung, jibb, notkin}@cs.washington.edu

ABSTRACT

There is a significant gap between how programmers think about code change and how change is represented in most software engineering tools. Programmers often think about code change in terms of structure: which code elements changed and how their structural dependencies are affected by the change. By reasoning about structural information within and around changed code, they recognize high-level systematic changes such as refactorings and crosscutting changes. Yet, most software engineering tools are based on a textual representation of code change. To bridge this gap, we propose a novel rule-based delta representation that explicitly and concisely captures systematic changes to a program's structure, along with an engine that automatically infers such rules. Our logical structural delta (LSD) can complement existing uses of textual deltas: e.g., understanding another programmer's modification, reviewing a patch before submission, and writing change documentation. We believe that LSD may serve as a basis for many software engineering tools that can benefit from explicit logical structure in code change: a bug finding tool, a refactoring reconstruction tool, a dependency removal checker, etc.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*restructuring, reverse engineering, and reengineering*

Keywords

code change, delta, systematic change, software evolution

1. INTRODUCTION

Programmers often inspect differences between program versions. For example, a team lead may review the work done by her team members by examining a program delta rather than the entire program. For such change-centric tasks, programmers generally use software engineering tools such as diff, CVS, and Unix patch that are based on line-based textual program deltas.

However, in some situations, textual deltas are not very effective in helping programmers understand code change. Suppose a team lead is reviewing a patch that involves hundreds of lines across multiple files. The team lead may find it difficult to figure out whether the intended change is implemented completely, why a certain group of files changed together, or whether a new dependency was unexpectedly introduced by the patch.

Several limitations of textual deltas seem to contribute to these difficulties. First, textual deltas do not explicitly capture structural information that programmers often look for when they understand code changes: which code elements (types, methods, and fields) changed and how their structural dependencies (subtyping, overriding, data access, invocation, and containment) are affected by the change. Second, textual deltas are verbose because they treat code changes as line-level differences even if the individual changes form a single high-level change. Finally, they do not contain contextual information that can help programmers understand logical structure in code change; for example, when all added methods override type t's method m, a textual delta reports the methods individually, leaving it to the programmer to discover their common structural characteristic.

To overcome these limitations, we propose a novel program delta that represents structural information using logic rules and facts, along with an algorithm that automatically infers these rules. Each of our inferred rules describes a group of atomic changes that share similar structural characteristics and thus corresponds to a high-level systematic change. To discover contextual information, our inference algorithm examines not only changed code fragments but also unmodified code fragments connected to them by structural dependencies. Using logic rules makes the delta more concise because a single rule can summarize many related facts at once. In addition, our approach detects and explicitly represents anomalies that signal incomplete and inconsistent change by allowing exceptions to a general rule.

For example, imagine a crosscutting change made by a programmer to prevent SQL injection attacks: removing all calls to `DB.exec` from the old version and replacing them with calls to `SafeSQL.exec`. Although this change is conceptually simple, its corresponding textual delta would likely involve changes scattered across many files. In our logical structural delta (LSD), this change is represented concisely as the following two rules—the first rule states

that all methods that called `DB.exec` in the old version no longer call `DB.exec` and the second rule states that these methods now call `SafeSQL.exec`. The details on the syntax and semantics of rules appear in Section 3.

past_calls(m, "DB.exec") ⇒ deleted_calls(m, "DB.exec")
past_calls(m, "DB.exec") ⇒ added_calls(m, "SafeSQL.exec")

As another example, imagine a pull-up method refactoring that moves `getHost` methods to their superclass, `NameSvc`. The following two rules show that `getHost` methods were pulled up to `NameSvc` from all of its subclasses except for `LmiSvc`, indicating that a programmer may have forgotten to finish the refactoring.

current_inheritedmethod("getHost", "NameSvc", t)
⇒ added_inheritedmethod("getHost", "NameSvc", t)
past_subtype("NameSvc", t) ∧ past_method(m, "getHost", t)
⇒ deleted_method(m, "getHost", t), except t = "LmiSvc"

We applied our LSD tool to two open source projects as well as our own project. While an average textual delta consists of 997 lines of change scattered across 16 files, our LSD represents its logical structure using an average of 7 rules and 27 facts. Our qualitative assessment shows that LSDs complement textual deltas by providing an overview of systematic changes and complement change descriptions by providing concrete structural information that can be traced to code. Compared to structural deltas without rules, LSDs are 9.3 times more concise and they include 10 additional structural dependencies on average.

This paper makes the following three contributions:

- Our LSD explicitly represents logical structure in code change.
- Our rule inference approach improves conciseness and also discovers useful contextual information from the code fragments related to changed code.
- Our empirical results show promise that our LSD representation and tool can serve as a basis for many tools that focus on code change.

The rest of this paper is organized as follows. Section 2 describes several scenarios that motivate our LSD. Section 3 and Section 4 present the formal definition of LSD and our LSD inference algorithm. Section 5 compares LSDs with textual deltas, change descriptions provided by a programmer, and structural deltas without rules. Section 6 discusses potential applications of LSD based on the motivating examples found in our study. Section 7 discusses related work, and Section 8 concludes with future work.

2. MOTIVATING SCENARIOS

In this section, we list several scenarios that illustrate difficulties that programmers face when investigating code change using textual deltas.[1]

Understand 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 truly 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.

Review 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.

Write change documentation. To write a check-in comment, Alice ran a 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.

[1]The scenarios are motivated by the observation carried out by Ko et al. [20] as well as the examples found in our study.

3. LOGICAL STRUCTURAL DELTA

We hypothesize that many difficulties of investigating code change using textual deltas originate from a lack of structural information about code change: which code elements (types, methods, and fields) changed and how their structural dependencies (subtyping, overriding, data access, invocation, and containment) were affected by the change.

To augment a program delta with such structural information, we represent each program version as a set of logic facts that describe code elements and their structural dependencies to other code elements. The difference between two versions is then represented as a set of facts deleted from the old fact-base (FB_o) and a set of facts added to the new fact-base (FB_n). For example, Table 1 shows the fact-base representation of two program versions and the delta between the two fact-bases (ΔFB). Even though the change is conceptually simple, "remove all accesses to `Key.on`, and invoke `Key.chk` from the `start` methods in `Car`'s subtypes," ΔFB lists the three accesses facts separately and also does not capture contextual information such as subtype("Car", "BMW") and subtype("Car", "GM").

Although ΔFB is a program delta with structural information, it has two major weaknesses. The first weakness is poor conciseness; because ΔFB is a set of facts without any high level structure, it is time-consuming to read and understand when it contains a large number of facts. The second weakness is that ΔFB describes only the structural dependencies of changed code fragments but not those of their surrounding code. For example, when several fields are removed from many different classes, it is useful to know that the fields of t_1 type were removed from t_2's subclasses even if t_2 does not appear in ΔFB. To overcome these two problems, our approach infers logic rules from the union of all three fact-bases, FB_o, FB_n, and ΔFB.

This approach has two advantages. First, our rule-based delta is often very concise because a single logic rule can imply a number of related facts. Second, by inferring rules from not only ΔFB but also from FB_o and FB_n, our approach finds useful structural information that cannot be found in ΔFB itself: for example, one can answer whether a systematic change evidenced by ΔFB is true for the entire old or new version, and one can discover structural characteristics shared by changed code fragments even if

Table 1: A Fact-Base Representation of Two Program Versions and their Difference

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

`class BMW implements Car`
`void start (Key c) {`
`...`
	subtype("Car","BMW"), ...
	method("BMW.start", "start", BMW)
	...
		`class BMW implements Car`
		`void start () {`
		`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) { ....`
`...`
	subtype("Car","GM"), ...
	method("GM.start", "start", "GM")
	accesses("Key.on", "GM.start")
	...
		`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; ....`
	subtype("Car","Kia"), ...
	method("Kia,start", "start", "Kia")
	accesses("Key.on", "Kia.start"), ...
		`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;} }`
	type("Bus")
	method("Bus,start", "start", Bus)
	accesses("Key.on", "Bus.start")
		`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) { ...`
	type ("Key")
	field("Key.on", "on", "Key")
	method ("Key.chk", "chk", "Key")
		`class Key {`
		`boolean on = false;`
		`static void chk (Key c) {`
			type ("Key")
			field("Key.on", "on", "Key")
			method ("Key.chk", "chk", "Key")	

* The deleted and added facts in ΔFB are noted with + and − sign respectively.

Table 2: LSD 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")

they appear in unmodified parts of a program.

The intuition behind our rule-based approach is that there are many situations in which apparently independent changes implement a higher-level, more systematic change together. By inferring rules that correspond to such high-level systematic changes, our approach concisely summarizes structural information within and around changed code. For instance, changing an API and subsequently changing all invocations of the API is an example of such systematic change. A crosscutting change that removes all dependencies to a particular module is another familiar example.

LSD Predicate. Our prototype currently models structural dependencies in a Java program at the type, field, and method level using the following twelve predicates. The first seven predicates describe code elements and their containment relationships. For example, type("org.foo.Bar", "Bar", "org.foo") means that there is either a class or an interface with the name `Bar` in `org.foo` package, and its fully qualified name is `org.foo.Bar`. The next five predicates describe field access, method invocation, subtyping, and overriding dependencies. For example, inheritedmethod("foo", "Boo", "Bob") means that `Bob` inherits `foo` method of `Boo` class.

1. package (p:Package).
2. type (t:Type, tn:TypeName, p:Package).
3. method (m:Method, mn:MethodName, t:Type).
4. field (f:Field, fn:FieldName, t:Type).
5. return (m:Method, returnType:Type).
6. fieldoftype (f:Field, declaredType:Type).
7. typeintype (inner:Type, outer:Type).
8. accesses (f:Field, accessor:Method).
9. calls (caller:Method, callee:Method).
10. subtype (super:Type, sub:Type).
11. inheritedfield (fn:FieldName, super:Type, sub:Type).
12. inheritedmethod (mn:MethodName, super:Type, sub:Type).

To distinguish which fact-base each fact belongs to, we prefix past_ and current_ to the facts in FB_o and FB_n respectively. To distinguish which facts were deleted from the old version and added to the new version, we prefix deleted_ and added_ to the corresponding facts in ΔFB.

Currently LSD predicates do not model access modifiers, local variable accesses, control logic, and temporal logic.

LSD Rule. A logic rule describes the relationship among groups of related logic facts. An LSD rule describes a high-level systematic change 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, our LSD 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) ∧ B(x,y)... ∧ C(x,z) ⇒ D(x,z). In LSD rules, all variables are universally quantified and variables do not appear in the conclusion unless they are bound in the antecedent. LSD 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)⇒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.

Table 3 shows the rule styles and an example rule for each style. These rule styles can express high-level systematic changes such as dependency removal and addition, feature addition and deletion, consistent maintenance, replacement of API usage or related code change.

Example. Suppose that a programmer intended to remove all accesses to a field `Key.on` and call `Key.chk` from the `start` methods in the classes implementing `Car`. Table 1 presents the fact-bases and Table 2 shows the rule inference and ΔFB reduction process. Based on the fact that all accesses to `Key.on` are removed from the old version, ΔFB is reduced to $\Delta FB'$ by replacing the three deleted_accesses facts with

Table 3: LSD 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) ∧ 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.

the following rule:
past_accesses ("Key.on", m) ⇒ deleted_accesses ("Key.on", m)
Based on the fact that `start` methods that call `Key.chk` are contained in `Car`'s subtypes, $\Delta FB'$ is reduced to $\Delta FB''$ by winnowing out the two added_calls facts using the following inferred rule. This rule also signals inconsistency that `Kia` did not change similarly.
past_method (m, "start", t) ∧ past_subtype ("Car", t)
⇒added_call (m, "Key.chk"), except t=Kia.

4. 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 use JQuery to extract logic facts—whose predicates are the types described in Section 3—from a Java program [14]. JQuery is a query-based browser that represents a Java program in terms of logical relations and provides an interface to execute a logic query. It analyzes a Java program and extracts code elements and their structural dependencies using the Eclipse JDT Parser; thus, its precision depends on the Eclipse's static analysis capability. Using JQuery, we create fact-bases, FB_o and FB_n, from the old and new version respectively and compute ΔFB using a set-difference operator. When a programmer renames some code elements, ΔFB includes pairs of deleted_ and added_ facts even though the corresponding code did not change. For example, if `foo` package is renamed to `bar`, ΔFB will contain deleted_package("foo") and added_package("bar"). We identify code element matches using our previous work [18], allowing our delta algorithm to consider these as renamings rather than deleted_ and added_ fact pairs.

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 1.

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.

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.

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 [21]. 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 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.

Algorithm 1 LSD Rule Inference Algorithm

```
1: R := ∅ // a set of ungrounded rules
2: L := ∅ // a set of valid learned rules
   // U is a set of facts in ΔFB that are not covered by L.
3: U := reduced ΔFB using default winnowing rules
4: for each antecedent size, i = 0 ... k do
5:   if (i = 0) then
6:     R := ungrounded rules with an empty antecedent by
       enumerating all possible conclusions.
7:   else
8:     R := extend all ungrounded rules in R by adding all
       possible literals to their antecedent.
9:   end if
10:  for each ungrounded rule r do
11:    G := try all possible constant substitutions for r's
       variables
12:    for each partially grounded rule g in G do
13:      if g is valid then
14:        L:=L ∪ g
15:      end if
16:    end for
17:  end for
18:  R := select the best β rules in R.
19:  U := U − {facts covered by L}
20: end for
```

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 SET-COVER algorithm [2]. 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.

5. EVALUATION

To investigate if and when LSD can be useful for describing code change, we performed a set of quantitative and qualitative assessments. We compared LSDs with textual deltas (TDs) and change descriptions written by programmers. We also compared LSD with ΔFB, a simple structural delta between two versions.

Subject Programs. We applied our LSD tool to two open source projects, *carol* and *dnsjava*, and to our LSD tool 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 our LSD tool'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.

Table 4: Comparison with Textual Delta

	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**
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**
LSD tool									
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**
MED	**0**	**0**	**6**	**9**	**344**	**31**	**8%**	**2**	**17**
AVG	**3**	**2**	**11**	**16**	**997**	**54**	**19%**	**7**	**27**

Comparison with Textual Delta. The goal of a comparison with TD is to investigate (1) how LSD and TD differ in terms of conciseness and (2) which types of changes LSD describes more effectively than TD. We computed TDs using *diff* and LSDs using our tool with default input parameter settings (m=3, a=0.75, k=2). We then investigated individual TDs and LSDs and studied associated change contents. To assist in this investigation, we built a viewer that visualizes each rule along with the facts explained by the rule. When a user clicks on a fact, it shows the corresponding code snippet in both old and new version.

Table 4 shows quantitative comparison results. *CLOC*

Table 5: Extracted Rules and Associated Change Descriptions

Source	Rules and Their Interpretation	Excerpt from Change Description
carol		
62-63	current_field(x,y,"CarolConfiguration") ∧ current_accesses(x,"CarolConfiguration.loadCarolConfiguration()") ⇒ added_field(x,y,"CarolConfiguration") Addition of related fields: all fields accessed from `loadCarolConfiguration` are newly added constants.	A new simplified configuration mechanism. (with bug id references)
	past_field(x,y,"CarolDefaultValues") ∧ past_fieldoftype(x,"Properties") ⇒ deleted_field(x,y,"CarolDefaultValues") Deletion of related fields: all fields with type `Properties` in `CarolDefaultValues` are deleted.	
128-129	current_method(x,"getPort()",z) ⇒ added_method(x,"getPort()",z) Feature addition: many `getPort` methods are added	Port number trace problem. (with bug id references)
421-422	current_calls(x,"NamingExceptionHelper.create(Exception)") ⇒ added_calls(x,"NamingExceptionHelper.create(Exception)") past_calls(x,"JNDIRemoteResource.getResource()") ⇒ deleted_calls(x,"Throwable.printStackTrace()") Changes in exception handling: all calls to `NamingExceptionHelper` are newly added ones, and all methods that called `getResource` no longer calls `printStackTrace`.	Refactoring of the spi package... (247 words long)
	current_inheritedmethod(x,"AbsContext",y) ⇒ added_inheritedmethod(x,"AbsContext",y) past_method(x,y,"JRMPContext") => deleted_method(x,y,"JRMPContext")... Extract superclass refactoring: create `AbsContext` by extracting common methods from `Context` classes.	
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" Extract superclass refactoring: `host` related fields and methods are pulled from `NameSvc`'s subclasses to `AbsRegistry` class, except from `LmiRegistry`.	Common methods go in an abstract class, easier to extend/maintain/fix.
480-481	past_calls(x,"CarolCurrentConfiguration.setRMI(String)") ⇒ deleted_calls(x,"Enumeration.nextElement())" current_accesses("MultiContext.contextsOfConfigurations",x) ⇒added_calls(x,"Iterator.next()") Library usage change: All methods that call `setRMI` no longer use `Enumeration` and use `Iterator` instead.	Change the configuration process of Carol as discussed... (139 words long.)
dnsjava		
0.6-0.7	current_method(x,y,"RRset") ∧ current_calls("Cache.addRRset(RRset,byte,Object)",x) ⇒ added_method(x,y,"RRset") Related change: all newly added methods in `RRSet` call `Cache.addRRSet`.	DNS.dns uses Cache
1.0.2-1.1	past_method(x,"sendAsync()",w) ⇒ added_return(x,"Object") past_method(x,"sendAsync()",w) ⇒ deleted_return(x,"int") API change: `Resolver.sendAsync` returns `Object` instead of `int`.	Resolver.sendAsync returns an Object instead of an int.
1.1.4-1.1.5	past_calls(x,".update.parseRR(Tokenizer,short,int)") ⇒ deleted_calls(x,"String.equals(Object)") past_calls(x,".update.parseSet(Tokenizer,short)") ⇒ deleted_calls(x,".update.parseRR(Tokenizer,short,int)") past_calls(x,".update.parseSet(Tokenizer,short)") ⇒ added_calls(x,"String.startsWith(String)") Consistent change of code clones.	update client syntax enhancement (add/delete/require/prohibit/glue) no longer require -r, -s, or -n.
LSD tool		
20-21	past_type(x,y,"edu.uw.cs.lsd") ∧ past_type(z,y,"edu.uw.cs.lsd.jquery") ⇒ deleted_type(x,y,"edu.uw.cs.lsd") Removal of code clones: remove the same types from `lsd` package	*no corresponding comment*

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 5). 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. We conjecture that LSDs and TDs can complement each other since LSDs provide an overview of systematic changes and TDs provide change details at a line level.

Comparison with Change Description. 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 *LSD tool*) 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 5 shows the comparison results. (It includes only several versions due to limited space.)

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.

Comparison with a Fact-Level Difference (ΔFB). As we discussed in Section 3, although ΔFB represents a structural difference between two versions, it is verbose and it does not contain contextual information. Based on the following three metrics, we measure the benefits of inferring rules on all three fact-bases instead of using ΔFB.

- *Coverage*: the percentage of facts in ΔFB explained by rules, represented as (# 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%.

- *Conciseness*: the measure of how concisely LSD explains ΔFB, represented as (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.

- *Additional Information*: the measure of how much additional structural information was extracted from outside of changed code fragments, represented as (# 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 2 refers to two additional facts that are not in ΔFB, subtype("Car","BMW") and subtype("Car", "GM").

Table 6: Comparison with ΔFB

	FB_o	FB_n	ΔFB	Rule	Fact	Cvrg.	Csc.	Ad'l.
Carol								
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
dnsjava								
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
LSD tool								
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
Med	6650	6712	132	2	17	89%	7.3	0.0
Avg	6632	6732	302	7	27	75%	9.3	9.7

Table 6 shows the results for the three data sets (m=3, a=0.75, k=2). On average, the inferred rules cover 75% of facts in ΔFB and also improve the conciseness measure by a factor of 9.3. They contain an average of 9.7 additional facts that are in FB_o or FB_n but not in ΔFB.

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 affect 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 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 [21]—to find rules more efficiently.

Threats to Validity. Although our evaluation provides a valuable illustration of how LSD can complement existing uses of textual deltas and change descriptions, our findings may not generalize to other data sets. We need further investigations into how LSD results are affected by other

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

factors such as the size of a program and the gap between program versions. In terms of internal validity, when comparing LSDs with textual deltas and change descriptions, the investigator's familiarity with LSD rules may have influenced qualitative assessments. In addition, the rules found by our algorithm depend on both input parameter settings and the rule styles supported by our algorithm. We plan to carry out further investigations to understand what kinds of systematic changes are frequent yet not captured by our current algorithm.

6. APPLICATIONS OF LOGICAL STRUCTURAL DELTA

Based on example LSDs found in our study, we believe that LSD can serve as a basis for many tools that can benefit from explicit logical structure in code change.

Dependency Creation or Removal Checker. 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 LSD rules that 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(m1, "NamingHelper()") ⇒ added_calls(m1, "NamingHelper()")

past_accesses("JNI.URL", m)⇒deleted_accesses("JNI.URL", m)

In addition to examining inferred rules, team leads can manually write and check rules using our tool, as it provides a rule vocabulary to state a high-level systematic change and a checker to evaluate a rule.

Identifying Related Changes. Programmers often need to sort out mixed logical changes because some programmers commit unrelated changes together. LSDs can help identify related changes by showing structural dependencies and further identifying their common characteristics. 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 `Cache.addRRSet` calls them.

current_calls("Cache.addRRSet", m)⇒added_method(m,"RRset")

Incomplete Change Detection. We believe that LSD rules can help programmers identify incomplete change by noting exceptions to systematic changes. For example, the following rule found in dnsjava 0.4-0.5 can help programmers raise a suspicion about why the three `rrToWire` methods did not change similarly.

past_method(m, "rrToWire", t)⇒deleted_calls(m, "toArray")
(12 matches, 3 exceptions)

In addition to these tools, many rules found in our evaluation suggest promises of using LSD to locate crosscutting concerns and to identify high-level refactorings such as *pull-up method*, *collapse hierarchy*, *extract superclass*, etc. Furthermore, LSD can be also used for mining software repository research that focuses on code change by complementing textual deltas and change descriptions.

7. RELATED WORK

Canonical Systematic Change. Several kinds of canonical systematic changes are well understood and studied in software engineering community, and many have built tools that automatically identify such systematic changes.

Refactorings are systematic changes that are intended to preserve program semantics [7]. There are several tools that automatically infer refactorings by comparing two program versions. Many of these tools are summarized elsewhere [17, 18]. While most tools as well as our previous work [18] focused on simple refactorings such as renaming, moving, and API signature change, LSD can help identify high-level refactorings such as *extract superclass* by considering structural dependencies. Using LSD for inferring refactorings has two strengths: (1) Our approach does not require pre-defined refactoring patterns, which makes refactoring inference both more flexible and easier, and (2) it is robust to the situations when refactoring is incomplete or when it is mixed with other changes.

Crosscutting concerns represent secondary design decisions—for example, performance, error handling, and synchronization—that are generally scattered throughout a program [16, 27]. Aspect-oriented programming languages provide language constructs that allow concerns to be updated in a modular fashion [15]. 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 techniques use naming conventions, formatting styles, and ordering of code in a file to provide indications about code that should change together [10]. Dagenais et al. [6] automatically infer structural patterns among the participants of the same concern and represent such concern using a rule syntax. The inferred rules were used to trace concerns over program versions. Breu et al. [3] mine aspects from version history by grouping method calls that are added together.

Code clones—code snippets that are syntactically or semantically similar—often change similarly; consistent maintenance of code clones is another kind of systematic change. Simultaneous editing [24] and linked editing [28] provide a programmer with mechanisms to characterize similar code fragments and to edit them with a single stream of editing commands. In our clone genealogy analysis [19], we built a tool that automatically identifies several types of systematic changes on clones (e.g., consistent update, inconsistent update) and studied clone evolution.

Code Change Analysis. Several approaches represent the difference between two versions as a set of atomic changes to allow for more semantic analysis on code change. Change

Distiller [8] compares two versions of abstract syntax trees to compute tree-edit operations and then maps each tree-edit to an atomic AST-level change type (e.g., parameter ordering change). Crisp [4] models code change using several different types of coarse grained changes and their syntactic inter-dependencies such as def-use relationships. Robbes' approach [26] captures AST-level changes from IDE and groups them to a higher-level change such as a refactoring recorded in IDE. None of these automatically infer systematic changes from a set of atomic changes.

Delta Representation. Representing deltas between entities—files, programs, databases, video sequences—is a long-standing and rich research area. Most of the work focuses on the time and or space efficiency of these representations: Hunt et al. performed an empirical analysis of a set of algorithms for computing deltas [13]. Conradi and Westfechtel [5] surveyed delta representations pertinent to software configuration management. Horwitz used program dependence graphs to compute program deltas and classified changes as either semantic or textual [12]. SmPL is a program delta description language designed to ease API evolution [25]. Mehra et al. [22] presented a collaborative and visual approach to differencing diagrams as opposed to text. Westfechtel earlier discussed generalized support for merging arbitrary structure-oriented documents [29]. To the best of our knowledge, neither these nor other related approaches compute rule-based deltas.

Analogous to finding contextual information from FB_o and FB_n, several different enhancements to diff provide additional contextual information. The most basic enhancement is diff's '-c' flag, which shows a fixed number of unchanged lines around each hunk of changed text. Other tools post-process diff's output and provide side-by-side visualization of differences or markup differences to aid in understanding textual deltas.

Rule-based Change Representation. In our previous work [18], we developed a rule-based change representation that expresses a high-level semantics-preserving transformation (e.g., moving a group of related classes and renaming a set of related APIs) and built an algorithm that automatically infers high-level refactoring rules. In both efforts, we represent systematic changes using logic rules and infer such rules using machine learning techniques. LSD differs from our previous work in several ways. First, the goal of our previous work was to match code elements across program versions to enable program analysis over multiple versions, while LSD aims to explicitly represent logical structure in code change. Second, while our previous work focused on semantics-preserving changes above the level of method-headers, LSD focuses on changes in structural dependencies. Finally, from a machine learning technique perspective, our previous algorithm learns rules in an open system because there is no ground truth for a code matching problem. On the other hand, our current algorithm learns rules in a closed system—the three fact-bases—by enumerating all rules within the rule search space set by the input parameters.

Mining Association Rules From Version History. Association rule learning discovers rules that relate elements that co-occur frequently within a data set [1]. This technique has been applied to version history to discover which code elements frequently change together. While previous research on co-change reports only what changed together [9, 31, 32], LSD can help answer what, how, and why questions by hinting at the common structural characteristics of co-changed code and their common transformation, e.g., all methods that access c's fields deleted a call to m.

Logic-based Program Representations. In software engineering community, there has been a long tradition of representing a program as a logic-base (or database). JQuery [14] and CodeQuest [11] allow programmers investigate a program's structure by formulating a logic query in a language like Prolog. Wuyts et al.'s approach [30] uses logic rules to describe software architecture and design patterns and checks their conformance on a fact-base. Mens et al. [23] allow programmers to specify a group of related code fragments that address the same concern using logic rules (e.g., all methods that access the same variable). Our work differs from these—and the many other logic-based representations of programs—in focusing on explicitly representing the difference between two programs.

8. CONCLUSIONS AND FUTURE WORK

Our approach is the first to represent the difference between two program versions using logic rules and facts by automatically inferring rules. Each rule in a logical structural delta concisely explains a group of atomic changes that share similar structural characteristics; thus, LSD can complement textual deltas by providing structural information and explicitly presenting its systematic nature. We believe LSD can serve as a basis for many software engineering tools that focus on code change—mining aspects based on change history, automatic identification of refactorings, checking dependency removal and creation, detection of incomplete or inconsistent changes, etc.

As future work, we plan to investigate effective clustering algorithms for grouping LSD rules because often several rules together form a higher-order change pattern. Furthermore, we believe that visualization of textual deltas can be improved by filtering out or regrouping line-level differences using the structure found by LSD. To assist programmers in interpreting LSD rules, we plan to build an automatic rule translator since it is fairly mechanical to translate first order logic rules to English sentences.

Acknowledgment.

We especially thank Kris De Volder for providing JQuery and Rob DeLine for sharing motivating examples from his observational study of programmers. We thank Marius Nita, Noah Snavely, Vibha Sazawal, and Alan Ho for their comments on our draft.

9. REFERENCES

[1] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. In *SIGMOD '93*, pages 207–216, 1993.

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

[3] S. Breu and T. Zimmermann. Mining aspects from version history. In *International Conference on Automated Software Engineering*, pages 221–230, 2006.

[4] O. Chesley, X. Ren, and B. Ryder. Crisp: A Debugging Tool for Java Programs. *International Conference on Software Maintenance*, pages 401–410, 2005.

[5] R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Computing Surveys (CSUR)*, 30(2):232–282, 1998.

[6] B. Dagenais, S. Breu, F. W. Warr, and M. P. Robillard. Inferring structural patterns for concern traceability in evolving software. In *International Conference on Automated Software Engineering*, 2007.

[7] M. Flowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[8] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall. Change distilling—tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):18, November 2007.

[9] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *International Conference on Software Maintenance*, page 190, 1998.

[10] W. Griswold. Coping with crosscutting software changes using information transparency. *Reflection 2001: The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 250–265, 2001.

[11] E. Hajiyev, M. Verbaere, and O. de Moor. Codequest: Scalable source code queries with datalog. In D. Thomas, editor, *European Conference on Object-oriented Programming*, volume 4067, pages 2–27, 2006.

[12] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Programming Language Design and Implementation*, pages 234–245, 1990.

[13] J. J. Hunt, K.-P. Vo, and W. F. Tichy. Delta algorithms: an empirical analysis. *ACM Trans. Softw. Eng. Methodol.*, 7(2):192–214, 1998.

[14] D. Janzen and K. D. Volder. Navigating and querying code without getting lost. In *International Conference on Aspect Oriented Software Development*, pages 178–187, 2003.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072(327-355):110, 2001.

[16] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-oriented Programming*, volume 1241, pages 220–242. 1997.

[17] M. Kim and D. Notkin. Program element matching for multi-version program analyses. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 58–64, 2006.

[18] M. Kim, D. Notkin, and D. Grossman. Automatic inference of structural changes for matching across program versions. In *International Conference on Software Engineering*, pages 333–343, 2007.

[19] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy. An empirical study of code clone genealogies. In *ESEC/SIGSOFT FSE '05*, pages 187–196, 2005.

[20] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *International Conference on Software Engineering*, pages 344–353, 2007.

[21] S. Kok and P. Domingos. Learning the structure of Markov logic networks. *Proceedings of 22nd International Conference on Machine Learning*, pages 441–448, 2005.

[22] A. Mehra, J. Grundy, and J. Hosking. A generic approach to supporting diagram differencing and merging for collaborative design. In *International Conference on Automated Software Engineering*, pages 204–213, 2005.

[23] K. Mens, T. Mens, and M. Wermelinger. Maintaining software through intentional source-code views. *International Conference oln Software Engineering and Knowledge Engineering*, pages 289–296, 2002.

[24] R. C. Miller and B. A. Myers. Interactive simultaneous editing of multiple text regions. In *USENIX Annual Technical Conference, General Track*, pages 161–174, 2001.

[25] Y. Padioleau, J. Lawall, and G. Muller. SmPL: A Domain-Specific Language for Specifying Collateral Evolutions in Linux Device Drivers. *Electronic Notes in Theoretical Computer Science*, 166:47–62, 2007.

[26] R. Robbes. Mining a change-based software repository. In *International Workshop on Mining Software Repositories*, page 15, 2007.

[27] P. Tarr, H. Ossher, W. Harrison, and S. Sutton Jr. N degrees of separation: multi-dimensional separation of concerns. pages 107–119, 1999.

[28] M. Toomim, A. Begel, and S. L. Graham. Managing duplicated code with linked editing. In *Visual Languages and Human Centric Computing*, pages 173–180, 2004.

[29] B. Westfechtel. Structure-oriented merging of revisions of software documents. In *Proceedings of the 3rd international workshop on Software configuration management*, pages 68–79, 1991.

[30] R. 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.

[31] A. Ying, G. Murphy, R. Ng, and M. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, 2004.

[32] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *International Conference on Software Engineering*, pages 563–572, 2004.