Identifying and Summarizing Systematic Code Changes via Rule Inference

Miryung Kim, *Member, IEEE,* David Notkin, *Fellow, IEEE,* Dan Grossman, Gary Wilson Jr.

Abstract—Programmers often need to reason about how a program evolved between two or more program versions. Reasoning about program changes is challenging as there is a significant gap between how programmers think about changes and how existing program differencing tools represent such changes. For example, even though modification of a locking protocol is conceptually simple and systematic at a code level, *diff* extracts scattered text additions and deletions per file. To enable programmers to reason about program differences at a high-level, this article proposes a rule-based program differencing approach that automatically discovers and represents systematic changes as logic rules. To demonstrate the viability of this approach, we instantiated this approach at two different abstraction levels in Java: first, at the level of application programming interface (API) names and signatures; and second, at the level of code elements (e.g., types, methods, and fields) and structural dependences (e.g., method-calls, field-accesses, and subtyping relationships). The benefit of this approach is demonstrated through its application to several open source projects as well as a focus group study with professional software engineers from a large E-commerce company.

Index Terms—Software evolution, program differencing, rule learning, logic-based program representation

1 Introduction

As software evolves, developers often inspect program differences between two versions. For example, a team lead reviews modifications done by her team members to check whether the intended change is implemented correctly. Questions developers ask about code changes are often of the following style [1], [2]: "What changed?" "Is anything missing in that change?" and "Why did this set of code fragments change together?"

To enable developers to reason about program differences at a high level and to help answer these kinds of high-level questions about program modifications, this article proposes a novel *rule-based program differencing* approach that automatically discovers and summarizes systematic code changes as logic rules. This rule inference approach is based on the observation that high-level changes such as refactorings, feature additions, and updates to code clones are often systematic edits—a group of related edits is required in multiple places to ensure consistency and completeness of the high-level change. Consider an example where a programmer reorganizes a chart-drawing program by the type of a rendered object, moving axis-drawing classes from the package `chart` to the package `chart.axis`. Then, to allow toggling of tool tips by the user, she appends a `boolean` parameter to a set of chart-creation interfaces. Even though the goals of these transformations can be stated concisely in natural language, existing program differencing tools, such as *diff*, would report modified lines per file, enumerating moved methods and modified interfaces.

- *M. Kim and G. Wilson Jr. are with the Department of Electrical and Computer Engineering at the University of Texas at Austin.*
- *D. Notkin and D. Grossman are with the Department of Computer Science & Engineering at the University of Washington.*

To demonstrate the viability of our rule-based program differencing approach, we instantiated *change-rules* at two different abstraction levels in Java. The first level of change-rules describes changes to method-header names and signatures. For example, the API-level changes in the preceding scenario are represented as the following change-rules:

for all x in chart.*Axis*.*(*)   packageRename(x, chart, chart.axis)
Interpretation: All methods with a name "chart.*Axis*.*(*)" moved from package chart to chart.axis.
for all x in chart.Factory.create*Chart(*Data)   argAppend(x, boolean)
Interpretation: All methods with a name "chart.Factory-.create*Chart(*Data)" added a new input parameter with the boolean type.

The second level of change-rules captures changes to code elements (packages, types, methods, and fields) and structural dependences (method-calls, field-accesses, overriding, subtyping, and containment). For each level of change-rules, we developed a rule-inference algorithm that explores the space of candidate change-rules. We refer to the tool implementation of the second rule inference algorithm as Logical Structural Diff (LSdiff). During the rule-inference process, to prevent almost-correct rules from being invalidated by a few missing or inconsistent change-facts, our approach identifies these rules but explicitly notes where the few exceptions occurred. Noting anomalies can help users focus their attention on possible errors [3]. In our domain, reporting anomalies can help developers avoid inconsistent or incomplete modifications that could lead to bugs.

We applied our rule-based program differencing approach to the histories of six software projects. We also conducted a focus group study with professional developers in a large E-commerce company to understand the target users' perspectives on our approach. Our evaluation shows the following results:

- Most changes to method-header names and signatures are explained by a small portion of change-rules. This confirms our hypothesis that leveraging a systematic change structure witnessed by multiple low-level transformations is a good approach for concisely representing high-level program differences.
- By inferring change-rules instead of finding individual changes, our technique significantly reduces the size of change descriptions at the method-header level by a factor of 3.5, 2.5, and 1.7 times in JFreeChart, JHotDraw, and jEdit respectively (median). Our technique also reduces the size of change descriptions at the level of code elements and structural dependences by a factor of 5.8, 4.8, and 9.8 times in Carol, Dnsjava, and LSdiff respectively (median).
- The focus group study participants indicated that our rule-based program differencing approach can complement existing uses of *diff* in code review tasks by providing a high-level overview and by helping them to focus their attention to potentially inconsistent or missing updates.
- Manual inspection of rules with exceptions shows that the rule exceptions often indicate either bugs caused by inconsistent edits or benign, yet suspicious, changes that are worthwhile to note.

The objective of our rule-based program differencing approach is to effectively find a high-level structure of changes between a program and a modified version of the program. Meanwhile, other API-matching and refactoring reconstruction techniques [4]–[9] do not focus on discovering a high-level structure among program changes; thus, a head-to-head comparison of our approach against others is not possible. Nevertheless, we compared the precision and recall of method-level matches with six other approaches to show that our approach provides a good number of method-header level matches as a starting point for inferring higher level structural changes. The comparison demonstrates that our technique is roughly on par in terms of method matches compared to other techniques and that it produces more concise output by inferring higher level change-rules from these method matches (see Appendix).

The rest of this article is organized as follows. Section 2 describes related work. Section 3 describes the syntax and semantics of the change-rules. Section 4 describes a rule-inference algorithm for each kind of change-rule. Sections 5 and 6 describe the evaluation of our rule-based program differencing approach. Section 7 discusses the limitations of our approach, and Section 8 concludes.

2 Related Work

The novelty of our approach is best seen in the context of existing approaches that can be used to reason about software changes.

Program Differencing and Refactoring Reconstruction. Existing program differencing techniques use similarities in names and structure to match code elements at a particular granularity: (1) lines and tokens [10], (2) abstract syntax tree nodes [11]–[14], (3) control flow graph nodes [15], (4) program dependence graph nodes [16], [17], etc. For example, the ubiquitous tool *diff* computes line-level differences per file using the longest common subsequence algorithm [10]. As another example, JDiff computes CFG-node level matches between two program versions based on similarity in node labels and nested hammock structures [15]. While the objective of these differencing tools is to accurately identify individual additions and deletions at a particular granularity, our rule-based approach focuses on recognizing a systematic structure among individual program differences. Some approaches attempt to address a similar problem by grouping program differences by physical locations (directories and files) [10], by logical locations (packages, classes, and methods) [4], by structural dependences (define-use and overriding) [18], or by similarity of names.

The API-matching problem addressed in our article is related to the problem of inferring refactorings from two program versions. Demeyer et al. first inferred refactorings from two program versions using a set of ten characteristic metrics, such as LOC and the number of method calls within a method [19]. Zou and Godfrey first coined the term origin analysis, which serves as a basis for refactoring reconstruction by matching code elements using multiple criteria (e.g., names, signatures, metric values, callers, and callees) [20]. Their approach infers merge, split, and rename refactorings.

Van Rysselberghe and Demeyer used a clone detector to detect moved methods [21]. Antoniol et al. identified class-level refactorings using a vector space information retrieval approach [22]. Dig et al.'s approach identifies refactorings in two stages. First, it finds a list of code element pairs using *shingles* (a metric-based fingerprint) and performs a semantic analysis based on reference relationships (calls, instantiations, uses of types, import statements) [9]. Second, it uses an iterative, fix point algorithm to find refactorings in a top-down order. Xing et al.'s approach [4] extracts class models from two versions of a program, traverses the two models, and identifies corresponding entities based on their name similarity and structure similarity (i.e., similarity in type declaration and uses, field accesses, and method calls). It then reports additions and removals of these entities, as well as inferred refactorings. Weißgerber and

Diehl's approach [5] extracts added and deleted entities (fields, methods, and classes) by parsing deltas from a version control system and then compares these entities based on their name similarity. When it cannot disambiguate all refactoring candidates, it uses a clone detector (CCFinder [23]) to rank these candidates. S. Kim et al.'s approach [8] considers various information (such as calling relationships, clone detection results, and name similarity) to match method-headers. Wu et al.'s approach [6] is a hybrid approach that combines the strengths of call-graph matching and name-similarity based matching. Nguyen et al.'s approach [7] identifies refactorings in libraries to support adaptation of the client applications that use those libraries. Similar to Xing et al.'s approach, the algorithm matches code elements top-down based on method name similarity and method body contents. Fluri et al.'s approach [11] compares two versions of abstract syntax trees, computes tree-edit operations, and maps each tree-edit to atomic AST-level change types (e.g., parameter ordering change). Prete et al.'s approach extends our rule-based program differencing approach, and was developed by the first author and her students to identify complex refactorings [24]. It encodes 63 out of 72 refactoring types in Fowler's catalog as template logic rules, and uses a logic-query approach to infer concrete refactoring instances [24]. While our article focuses on automatically inferring rules from program differences, Prete et al. use pre-defined logic rules to detect structural differences that fit known refactoring types.

The objective of our rule-based approach is to report structural differences in a concise manner, while many refactoring reconstruction techniques have different objectives (e.g., automatically updating API clients using reconstructed refactorings [7], [9]). This poses different requirements for the conciseness, precision, and recall of results that each tool reports.

Source Transformation Languages and Tools. Source-transformation tools let developers encode systematic changes in a formal syntax to automate repetitive and error-prone program updates [25], [26]. For example, iXj enables developers to easily perform systematic code transformations by providing a visual language and a tool [27]. Coccinelle lets developers apply systematic updates to Linux device drivers [28]. This approach is appropriate in situations where developers are willing to plan changes in advance and to learn a transformation language. While these tools focus on *applying systematic changes* to a program, our work focuses on *recovering systematic changes* from two program versions.

Identification of Related Changes. Several approaches use change history to identify code elements that tend to change together [29]. However, they do not explicitly group systematic changes nor report their common structural characteristics, leaving it to developers to figure out why some code fragments change together. Crisp [18], a part of the Chianti change impact analysis tool [30], computes AST-level structural differences and groups related differences using four predefined rules, such as "identify all method additions that refer to a new field." Instead of using four pre-defined rules, Logical Structural Diff (LSdiff) infers change-rules to describe related changes with similar dependence characteristics such as "accessing the same field in the classes with the same name." Furthermore, while Crisp's goal is to create a compilable intermediate version for fault localization, our approach focuses on recovering a latent systematic structure in program differences. JUnitMX, an extension of Chianti, groups syntactic changes at the type, method, and field levels based on their effect on the regression test outcomes and coverage information [31]. Unlike JUnitMX, our approach currently does not consider any dynamic information, and it remains as a future work to extend our approach to summarize runtime behavior differences.

Systematic Code Changes. Our rule-based program differencing approach is based on the insight that high-level changes are often systematic—consisting of related transformations at a code level. The same insight arises from numerous other research efforts, primarily within the domain of refactorings and crosscutting concerns. Refactoring [32]–[34] often consists of one or more elementary transformations, such as "moving the `print` method in ***each*** `Document` subclass to its superclass" or "introduce ***three*** abstract `visit*` methods." *Crosscutting concerns* represent secondary design decisions—e.g., performance, error handling, and synchronization—that are generally scattered throughout a program [35], [36]. Modifications to these design decisions involve similar changes to every occurrence of the design decision. To cope with evolution of crosscutting concerns, AspectJ provides language constructs that allow these concerns to be updated in a modular fashion [35]. Several techniques locate and document crosscutting concerns based on similarities in a program's dependence structure, naming conventions, formatting styles, and ordering of code in a file [37], [38]. Our approach focuses on summarizing scattered, but related *program changes*, and thus is complementary to existing approaches for evolving crosscutting concerns.

3 Rule-based Change Representations

Change-rules are inferred from two program versions and represent systematic code changes from one version to another. This section describes our change-rule representations, including their syntax and semantics. A change-rule consists of a scope, exceptions, and a transformation.

```
for all x: code element in (scope)
  except (exceptions)
  transformation(x)
```

The *scope* defines a subset of code elements in the first program version, the *exceptions* remove a subset of these elements, and the *transformation* describes how the pertinent elements in the scope changed between the two versions. Noting exceptions to the rules prevents almost-correct rules from being invalidated by a few missing or inconsistent change-facts during rule inference. Section 3.1 describes the syntax of change-rules at a method-header level, and Section 3.2 describes the syntax of change-rules at the level of a program's dependence structure.

3.1 Definition of API Change-Rule at a Method-Header Level

The unit of code elements in the scope of an API change-rule is a method-header, which is defined as a tuple: (`package:String, class:String, procedure:String, input_argument_list:[String], return_type:String`).

To represent the scope of a change-rule, we summarize a group of similarly named method-headers using a wild card pattern-matching operator. For example, *.*Plot.get*Range() describes methods with any package name, any class name that ends with `Plot`, any procedure name that starts with `get` and ends with `Range`, and an empty argument list. This use of a wild card pattern is based on the observation that developers tend to name code elements similarly when they belong to the same concern [38]. A scope can have disjunctive scope expressions.

To represent transformations at a method-header level, we define nine types of transformations that describe changes to method-header names and signatures (see Table 1). These transformations can describe both rename and move refactorings depending on a scope expression. For example, 'for all x:chart.*Chart.*(*), packageRename(x, chart, chart.plot)' means that all classes with the name `*Chart` were moved from package `chart` to package `chart.plot`, while 'for all x:chart.*.draw(*), methodRename(x, draw, render)' means that all `draw` methods in package `chart` were renamed to `render`. As another example, the following rule means that all classes in package `chart` whose name ends with `Plot` moved to package `chart.plot`.

```
for all x:method-header in chart.*Plot.*(*)
  packageRename(x, chart, chart.plot)
```

A method-header-level matching between two program versions can be described by a set of change-rules. Method headers that are identical in both versions are excluded. After rule inference, the methods that are not matched by any rules are either deleted or added methods. For example, the five API change-rules in Table 2 (b) explain seven method-header matches in Table 2 (a).

3.2 Definition of Logical Structural Diff Rules

This section describes change-rules at the level of a program's dependence structure. In Logical Structural Diff (LSdiff), code elements and their structural dependences are modeled using the thirteen logic predicates shown in Table 1.[1] In a change-rule, we prefix each predicate with past_ or current_ to denote code elements and structural dependences in the old or new version, respectively.

To represent the scope of a change-rule (i.e., a subset of code elements), a literal is created by binding a predicate's argument to universally quantified variables or constants. To further refine the scope of a transformation, one or more literals can be combined using a conjunction. For example, past_method(m, "draw", t) $\wedge$ past_extends("Plot", t) represents "all methods m in `Plot`'s subclasses in the old version."

Transformations are represented as deleted facts from the old version or added facts in the new version. For example, deleted_accesses(m, "Shape.dotted") means that method m deleted accesses to field `Shape.dotted`.

Change-rules follow the syntax of horn clauses, where the conjunction of one or more literals in the antecedent implies a single literal in the conclusion. In a change-rule, all variables are universally quantified and variables do not appear in the conclusion, unless they are bound in the antecedent. In addition, the antecedent of a rule cannot have predicates with different prefixes. By using only deleted_* or added_* in a rule's consequent, change-rules describe *differences between* two versions as opposed to the structural property of *a single program version*. For instance, the following rule states that all methods with a name `draw` in `Plot`'s subclasses removed accesses to `Shape`'s `dotted` field.

```
∀ m, t, past_method(m, "draw", t) ∧ past_extends(t, "Plot")
⇒ deleted_accesses (m, "Shape.dotted")
```

3.3 Change-Rule Relationships

Table 1 summarizes the syntax of both kinds of change-rules. While the first kind of change-rules model transformations as replacements of strings within method signatures, the second kind of change-rules describe deletions and additions of code elements and associated structural dependences. While it is possible to define *add* and *delete* transformations at the level of method-headers, we decided to focus on replacement transformations because our initial goal was to infer renaming and moving at or above the level of method-headers rather than finding systematic change patterns among a group of deleted or added methods.

In our work, we used the results of API-level change-rules to filter fact-level differences within method bodies caused by method rename, move, and change signature refactorings. Table 2 illustrates this process. After inferring a set of API change-rules shown in Table 2 (b), these results are used to filter the original fact-level differences in Table 2 (c). From the remaining set of fact-level differences in Table 2 (d), two LSdiff change-rules

1. Their detailed definitions, including argument types, are summarized in the first author's dissertation, which can be found at http://users.ece.utexas.edu/~miryung/Publications/uw08-dissertation-mkim.pdf

TABLE 1
The syntax of change-rules

API Rule	
Abstraction	method-header
Scope	a subset of method-headers expressed using a wild-card operator
Transformation	1. packageRename(x:Method, f:String, t:String): change x's package name from f to t 2. classRename(x:Method, f:String, t:String): change x's class name from f to t 3. procedureRename(x:Method, f:String, t:String): change x's procedure name from f to t 4. returnReplace(x:Method, f:String, t:String): change x's return type from f to t 5. inputSignatureReplace(x:Method,f:List[String], t:List[String]): change x's input argument list from f to t 6. argReplace(x:Method, f:String, t:String): change argument type f to t in x's input argument list 7. argAppend(x:Method, t:List[String]): append all of the argument types in t to the end of x's input argument list 8. argDelete(x:Method, t:String): delete every occurrence of type t in the x's input argument list 9. typeReplace(x:Method, f:String, t:String): change every occurrence of type f to t in x
LSdiff Change-Rule	
Abstraction	package, type, method, field
Scope	a subset of code elements, expressed in a conjunctive logic literal
Transformation	addition and deletion of code elements and structural dependences represented by the following predicates. 1. package (packageFullName) 2. type (typeFullName, typeShortName, packageFullName) 3. method (methodFullName, methodShortName, typeFullName) 4. field (fieldFullName, fieldShortName, typeFullName) 5. return (methodFullName, returnTypeFullName) 6. fieldoftype (fieldFullName, declaredTypeFullName) 7. typeintype (innerTypeFullName, outerTypeFullName) 8. accesses (fieldFullName, accessorMethodFullName) 9. calls (callerMethodFullName, calleeMethodFullName) 10. extends (superTypeFullName, subTypeFullName) 11. implements (superTypeFullName, subTypeFullName) 12. inheritedfield (fieldShortName, superTypeFullName, subTypeFullName) 13. inheritedmethod (methodShortName, superTypeFullName, subTypeFullName)

in Table 2 (e) are inferred. Any remaining change-facts that are not explained by any of the inferred rules are shown as is.

4 Rule Inference Algorithm

Section 4.1 describes an inference algorithm for API change-rules and Section 4.2 describes an algorithm for LSdiff change-rules. Though we have two separate inference algorithms, the two algorithms share common characteristics. First, they both compute individual differences at a chosen abstraction level. Second, they systematically generate candidate rules, each of which represents a group of related differences. They then evaluate the accuracy of each candidate rule with respect to the two input program versions and select a subset of the candidate rules. For both kinds of change-rules, we implemented a set of optimization heuristics to reduce the search space of candidate rules.

4.1 API Change-Rule Inference

The API change-rule inference algorithm first finds seed matches. Based on these seeds, the algorithm then generates candidate rules and iteratively selects the best rule among the candidate rules. We first describe a naïve version of our algorithm, followed by our optimization heuristics.

4.1.1 *Identification of Seed Matches*

Given the two program versions (P_1, P_2), we extract two sets of method headers O and N from the old version P_1 and the new version P_2 respectively. Then, for each method header x in $O - N$, we find the closest method header y in $N - O$ in terms of the token-level name similarity. This seed match generation is purely based on method name similarity and we do not use other types of information such as inheritance or call relationships. We call the resulting set of matches as *seed matches* because they are used to derive initial hypotheses about systematic change patterns.

The token-level similarity measure involves separating out the package name, class name, method name, signature, and return type from each header. Each of these strings is then broken into a list of tokens by splitting on capital and non-alphabet characters, based on a camel case naming convention.[2] An overall similarity measure is then calculated using a weighted sum of the similarities of each part, which is based on the longest common subsequence (LCS) algorithm [10]. A LCS-based similarity between two strings A and B is defined as:

$$S = \frac{len(LCS(A,B))}{max(len(A), len(B))} \quad (1)$$

If the name similarity is over a threshold γ (default γ=0.7), the pair is added to the initial set of seed matches. The seeds need not all be correct matches, as our rule selection algorithm rejects bad seeds and leverages good seeds. Section 5.3 presents six additional similarity measures that we implemented to characterize the impact

2. http://msdn.microsoft.com/en-us/library/x2dbyw72(VS.71).aspx,http://en.wikipedia.org/wiki/CamelCase

TABLE 2
Rule-based program differencing example

(a) Textual differences between the old version P_1 and the new version P_2

```
package chart .plot ;
public class VerticalPlot extends Plot {
 void draw (Graph g~~, Shape s~~){
  ~~if (s.dotted)~~ {...} } }

package chart;
public class VerticalRenderer {
 void draw (Graph g, Shape s) {...} }

package chart .plot ;
public class HorizontalPlot extends Plot {
 void draw (Graph g~~, Shape s~~) {
  ~~s.dotted = true;~~ ...} }
```

```
package chart .axis ;
public class HorizontalAxis {
 public int ~~height()~~ getHeight() {...} }
package chart .axis ;
public class VerticalAxis {
 public int ~~height()~~ getHeight() {...} }

package chart;
public class ChartFactory { ...
 public Chart createAreaChart(Data d, boolean b ) { ...
  c.setToolTip(b); ...}
 public Chart createPieChart(PieData p, boolean b ) { ...
  c.setToolTip(b); ...} }
```

(b) Inferred API change-rules

for all x:chart.*Plot.*(*), packageRename(x, chart, chart.plot)

for all x:chart.*Axis.*(*), packageRename(x, chart, chart.axis)

for all x:chart.ChartFactory.create*Chart.(*), argAppend(x, {boolean})

for all x:*.*.*(Graph, Shape) except VerticalRenderer.draw(Graph, Shape), argDelete(x, Shape)

for all x:chart.*Axis.height() procedureRename(x, height, getHeight)

(c) $\triangle$FB, the original fact-level differences between P_1 and P_2.
The deleted and added facts are marked with - and + for presentation purposes.

-type(”chart.VerticalPlot”, ”VerticalPlot”, ”chart”)

+type(”chart.plot.VerticalPlot”, ”VerticalPlot”, ”chart.plot”)

-extends(”chart.Plot”, ”chart.VerticalPlot”)

+extends(”chart.plot.Plot”, ”chart.plot.VerticalPlot”)

-method(”chart.VerticalPlot.draw(Graph, Shape)”, ”draw”, ”chart.VerticalPlot”)

-accesses(”Shape.dotted”, ”chart.VerticalPlot.draw(Graph,Shape)”)

+method(”chart.plot.VerticalPlot.draw(Graph)”, ”draw”, ”chart.plot.VerticalPlot”) -method(”chart.HorizontalPlot.draw(Graph, Shape)”, ”draw”, ”chart.HorizontalPlot”)

-accesses(”Shape.dotted”, ”chart.HorizontalPlot.draw(Graph,Shape)”)

+method(”chart.plot.HorizontalPlot.draw(Graph)”, ”draw”, ”chart.plot.HorizontalPlot”)

-type(”chart.HorizontalAxis”, ”HorizontalAxis”, ”chart”)

+type(”chart.axis.HorizontalAxis”, ”HorizontalAxis”, ”chart.axis”)

-method(”chart.HorizontalAxis.height()”, ”height”, ”chart.HorizontalAxis”)

+method(”chart.HorizontalAxis.getHeight()”, ”getHeight”, ”chart.axis.HorizontalAxis”)

-method(”chart.ChartFactory.createAreaChart(Data)”, ”createAreaChart”, ”chart.ChartFactory”)

+method(”chart.ChartFactory.createAreaChart(Data, boolean)”, ”createAreaChart”, ”chart.ChartFactory”)

+calls(”chart.ChartFactory.createAreaChart(Data, boolean)”, ”chart.Chart.setToolTip(boolean)”)

-method(”chart.ChartFactory.createPieChart(PieData)”, ”createPieChart”, ”chart.ChartFactory”)

+method(”chart.ChartFactory.createPieChart(PieData, boolean)”, ”createPieChart”, ”chart.ChartFactory”)

+calls(”chart.ChartFactory.createPieChart(PieData, boolean)”, ”chart.Chart.setToolTip(boolean)”)

(d) $\triangle$FB′ after removing fact-level differences caused by rename refactorings

-accesses(”Shape.dotted”, ”chart.VerticalPlot.draw(Graph, Shape)”)

-accesses(”Shape.dotted”, ”chart.HorizontalPlot.draw(Graph, Shape)”)

+calls(”chart.Chart.createAreaChart(Data, boolean)”, ”chart.Chart.setToolTip(boolean)”

+calls(”chart.Chart.createPieChart(PieData, boolean)”, ”chart.Chart.setToolTip(boolean)”

(e) Inferred LSdiff change-rules

$\forall$ m, t, past_method(m, ”draw”, t) $\wedge$ past_extends(t, ”Plot”) $\Rightarrow$ deleted_accesses (m, ”Shape.dotted”)

$\forall$ m, n, past_method(m, n, ”ChartFactory”) $\Rightarrow$ added_calls(m, ”Chart.setToolTip()”

of the seed generation algorithm on the rule-inference process.

4.1.2 Rule Inference

Generating Candidate Rules. For each seed match $[x, y]$, we build a set of *candidate rules* in three steps. A candidate rule may include one or more transformations $t_1, \ldots, t_i$ such that $y = t_i(\ldots t_1(x))$. This representation allows our algorithm to find a match $[x, y]$ where x undergoes multiple transformations to become y.

First, we compare x and y to find a set of transformations $T = \{t_1, t_2, \ldots, t_i\}$ such that $t_i(\ldots t_2(t_1(x))) = y$. For example, a seed [`chart.VerticalAxis.height()`, `chart.plot.VerticalAxis.getHeight()`] produces the power set of packageRename(x, chart, chart.plot) and procedureRename(x, height, getHeight).

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

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

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

The inputs are a set of candidate rules (CR), a domain ($D = O - N$), a codomain ($C = N$), and an exception threshold ($0 \leq \epsilon < 1$, default ϵ=0.34). The outputs are a set of selected candidate rules (R), and a set of found matches (M). For a candidate rule r, "for all x in scope, $t_1(x) \wedge ... \wedge t_i(x)$":

1) r has a **match** $[a, b]$ if $a \in$ scope, $t_1, ..., t_i$ are applicable to a, and $t_i(...t_1(a)) = b$.
2) a match $[a, b]$ **conflicts** with a match $[a', b']$ if $a = a'$ and $b \neq b'$
3) r has a **positive** match $[a, b]$, given D, C, and M, if $[a, b]$ is a match for r, $[a, b] \in D \times C$, and none of the matches in M conflict with $[a, b]$
4) r has a **negative** match (an exception) $[a, b]$, if it is a match for r but not a positive match for r.
5) r is a **valid** rule if the number of its positive matches is at least $(1 - \epsilon)$ times the number of its matches.

Our algorithm greedily selects one candidate rule at each iteration such that the selected rule maximally increases the total number of matches. Initially, we set both R and M to the empty set. In each iteration, for every candidate rule $r \in CR$, we compute r's matches and check whether r is valid. Then, we select a valid candidate rule s that maximizes $|M \cup P|$, where P is s's positive matches. After selecting s, we update $CR := CR - \{s\}$, $M := M \cup P$, and $R := R \cup \{(s, P, E)\}$, where P and E are s's positive and negative matches, respectively. After updating, we continue to the next iteration. The iteration terminates when no remaining candidate rules can explain any additional matches. Because the candidate rule investigation order is set by the algorithm, the output R for given CR, D, and C is deterministic. The naïve version of this greedy algorithm has $O(|CR|^2 \times |D|)$ time complexity. This means that generating more seeds leads to more candidate rules, and thus a longer running time; however, it can also increase the accuracy of found matches (see Section 5).

Optimization Heuristics. We implemented an optimized version of this algorithm based on two observations. First, if a candidate rule r can add n additional matches to M at the i^{th} iteration, r cannot add more than n matches on any later iteration. By storing n, we can skip evaluating r on any iteration where we have already found a better rule s that can add more matches than r. Second, candidate rules have a subsumption structure because the scopes can be subsets of other scopes (e.g., *.*.*(*Axis) $\subset$ *.*.*(*)). The pseudo code of our optimized algorithm is described in Algorithm 1. It starts with the most general candidate rule for each set of transformations and generates more candidate rules on demand. It has the same worst case complexity as the naïve algorithm: $O(|CR|^2 \times |D|)$. However, its empirical performance is much better. This can be seen by an approximation of the common case, in which only one rule needs be expanded to investigate its children's rules at each level of the subsumption lattice: $O(|log_n(CR)|^2 \times |D|)$, where n is the number of tokens in $seed.left$. The optimized algorithm remains a heuristic and may not find the smallest number of rules.

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

4.2 LSdiff Change-Rule Inference

This section describes a change-rule inference algorithm for the second kind of rules described in Section 3.2. Section 4.2.1 describes how facts about code elements and structural dependences are extracted from each program version in order to compute structural differences. This section also discusses how the initial set of differences are pruned using the results of inferred renamings in order to compute change facts. Section 4.2.2 describes how the algorithm systematically enumerates and evaluates candidate rules.

Algorithm 1: API rule inference algorithm

```
     Input: S, /* a set of seed matches */
1.1  $\epsilon$, /* an exception threshold */
1.2  D, /* domain: extractMethodHeaders($P_1$) − extractMethodHeaders($P_2$) */
1.3  C /* codomain: extractMethodHeaders($P_2$) */
     Output: R, /* a set of selected rules */
1.4  M /* a set of found matches */
1.5  R := $\emptyset$, M := $\emptyset$, CR := $\emptyset$;
     /* Create an initial set of rules */
1.6  foreach seed $\in$ S do
1.7      $2^T$ := extractTransformations (seed);
1.8      foreach trans $\in$ $2^T$ do
1.9          scope:= findTheMostGeneralScope (seed.left, trans);
1.10         rule:= createNewRule (scope, trans);
1.11         CR := CR $\cup$ {rule};
1.12     end
1.13 end
1.14 cont := true;
1.15 while cont do
1.16     n := |M |;
         /* select the best rule */
1.17     N := 0, s := null;
1.18     foreach $r_k \in$ CR do
1.19         if (numRemainingPositive ($r_k$) $>$ N) $\wedge$ (isValid ($r_k$, D, C, M, $\epsilon$)) then
1.20             N = numRemainingPositive ($r_k$);
1.21             s = $r_k$;
1.22         end
1.23     end
         /* If an invalid rule $r_k$ in CR can find more than N matches, expand its children rules. */
1.24     toBeRemoved := $\emptyset$; toBeAdded := $\emptyset$;
1.25     foreach $r_k \in$ CR do
1.26         if numRemainingPositive ($r_k$)=0 then
1.27             toBeRemoved := toBeRemoved $\cup$ {$r_k$};
1.28         end
1.29         else if (numRemainingPositive ($r_k$) $>$ N) $\wedge$ (isValid ($r_k$, D, C, M, $\epsilon$))= false then
1.30             toBeRemoved := toBeRemoved $\cup$ {$r_k$};
                 children = createChildrenRules ($r_k$,N);
1.31             foreach c $\in$ children do
1.32                 if (isValid (c, D, C, M, $\epsilon$)) $\wedge$ (numRemainingPositive (c) $>$ N) then
1.33                     N := numRemainingPositive (c); s := c;
1.34                 end
1.35             end
1.36             toBeAdded := toBeAdded $\cup$ children;
1.37         end
1.38     end
         /* Add toBeAdded to CR and remove toBeRemoved from CR. */
1.39     CR := CR $\cup$ toBeAdded;
1.40     CR := CR $-$ toBeRemoved;
1.41     R := R $\cup$ {s };
1.42     CR := CR $-$ {s };
1.43     M := M $\cup$ s.positive;
1.44     if (|M|=n) then
1.45         cont := false;
1.46     end
1.47 end
```

4.2.1 Identification of Structural Differences (ΔFB)

This section describes how structural differences are computed from two input program versions P_1 and P_2. In our original prototype published in 2009 [39], we used JQuery [40] to compute FB_1 and FB_2 (fact-base representations of P_1 and P_2). We then applied a set differencing operator between them to compute ΔFB, assuming that code elements can be mapped by their fully qualified names between FB_1 and FB_2.

To avoid re-processing of unmodified files, we developed our own incremental fact-extraction analysis based on the Eclipse Java Development Toolkit (JDT)'s AST analysis. It uses the knowledge of change-sets at a file level and parses only modified and added files in the new program version. For each deleted file, it simply retrieves the associated facts and marks them deleted. Some code elements may produce change facts, even if their source files are not modified. For example, suppose that class `C` extends class `B`, which extends class `A`, and class `A` declares a method `m`. When a method `m` is added to `B`, `C`'s lookup for `m` changes from `A.m` to `B.m`, even though the source file containing class `C` has not been modified. We derive such inheritance-related differences from both the subtyping and member declaration facts in the old program version and in the initially computed ΔFB. We also handle a few other rare cases caused by moving or renaming code elements. For instance, when class `C` is moved from `packageA` to `packageB`, all classes that instantiate `C` and import both packages will change their structural dependences, even though they have no textual modifications. The fact extractor detects such cases via pattern matching on ΔFB and re-processes the affected files.

Resolving fact-level differences caused by code renaming. The fact extractor uses the results of API change-rule inference to prune out spurious fact-level differences caused by code renaming and moving. Consider the example in Table 2. Recall that our API-matching tool in Section 4.1 identifies method-header level matches between two versions by leveraging similarity in names. Moving classes `HorizontalAxis` and `VerticalAxis` from package `chart` to `chart.axis` causes several fact-level additions and deletions. Based on the inferred rule: for all x:chart.*Axis.*(*), packageRename(x, chart, chart.axis), we derive that class `chart.HorizontalAxis` maps to `chart.axis.HorizontalAxis` and that class `chart.VerticalAxis` maps to `chart.axis.VerticalAxis`. Based on these matches, we filter out fact-level additions and deletions such as deleted_type("chart.HorizontalAxis") and added_type("chart.axis.HorizaontalAxis"). Table 2 (d) shows ΔFB after filtering out the facts in Table 2 (c) using the renaming-rules shown in Table 2 (b).

4.2.2 Rule Inference

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 (FB_o, FB_n, and ΔFB) and outputs inferred rules and remaining unmatched facts in ΔFB. The remaining unmatched facts in ΔFB are shown to developers as is, since the algorithm could not discover systematic change patterns that could explain those facts.

Three input parameters define which rules are considered for 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 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 dependence 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 dependences involving the method. To prevent learning such rules, we have written 30 *default winnowing rules* by hand—they are described elsewhere [41]. Using these rules, we 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. The inference algorithm is summarized in Algorithm 2.

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) the rule 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.

Algorithm 2: LSdiff rule inference algorithm

```
Input: $FB_o$, /* a fact-base of an old program
    version                                        */
$\Delta FB$, /* fact-level differences between $FB_o$
    and $FB_n$                                     */
$m$, /* the minimum number of facts a rule
    must match to be selected                      */
$a$, /* the minimum accuracy of a rule             */
$k$, /* the maximum number of literals in a
    rule's antecedent                              */
$\beta$ /* beam search window size                 */
Output: L /* a set of valid learned rules          */
R := $\emptyset$, L := $\emptyset$, U := ΔFB;
U := reduceDefaultWinnowingRules ($\Delta FB$, $FB_o$) ;
foreach $i = 0 \ldots k$ do
    if ($i = 0$) then
        R := $\emptyset$;
        foreach $p \in$ DELTA_PREDICATES do
            l:= createLiteral(p, freshvariables());
            r:= new Rule();
            r.setConsequent(l);
            if $|r.matches| \geq m$ then
                R := R $\cup$ {r};
            end
        end
    end
    else
        NR := $\emptyset$;
        foreach $r \in$ R do
            foreach $p \in$ ANTECEDENT_PREDICATES do
                bindings :=
                enumerateBindingsForPredicate (r, p)
                foreach $b \in$ bindings do
                    r:= new Rule(r);
                    r.addAntecedentLiteral(l);
                    if $|r.matches| \geq m \wedge !(r \in NR)$ then
                        NR := NR $\cup$ {r};
                    end
                end
            end
            R := NR;
        end
    end
    foreach $r \in$ R do
        NR:= $\emptyset$;
        S = new Stack();
        S.push(r);
        while !S.isEmpty() do
            pr = S.pop();
            foreach variable $\in$ pr.remainingVariables() do
                constants := getReplacementConstants(pr,
                variable);
                foreach constant $\in$ constants do
                    n = substitute(pr, variable, constant) if
                    $|$n.matches$| \geq m \wedge$ accuracy(n) $\geq a$ then
                        NR := NR $\cup$ {n };
                    end
                    if n.remainingVariables.size() $> 0$ then
                        S.push(n)
                    end
                end
            end
        end
        G := NR;
        foreach $g$ in G do
            if isValid ($g$) then
                L :=L $\cup$ {g};
                U :=U $-$ {g.matches};
            end
        end
    end
    R :=selectRules (R, $\beta$);
end
```

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 on the fact-bases using Tyruba, a Prolog-like logic programming engine [42] 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.

Rules may still have overlapping matches. To avoid outputting rules that cover the same set of facts in the ΔFB, we select a subset of the rules using the greedy version of the SET-COVER algorithm [43]. 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.

Optimization Heuristics. We implemented two optimization options that limit the size of rule search space and input fact-bases to improve the performance of rule inference:

1) Beam search. To tame the exponential growth of rule search space, we save only the best β number of ungrounded rules and pass them to the next iteration [44].
2) Reducing the scope of contextual facts. Since distant contextual facts are unlikely to contribute to finding shared structural characteristics of modified code, we prune contextual facts beyond a hop distance of k.

In addition, we allow the user to select the level of abstraction for facts in ΔFB. Facts lower than the selected level of granularity are aggregated into modified_* facts. For instance, when operating at a *type* granularity, all added_method and deleted_method facts from the same class are aggregated into a single modified_type fact.

5 Evaluation of API Change-Rules

We applied our API change-rule inference approach to three open source Java projects, which have release archives on *sourceforge.net*. *JFreeChart* is a library for drawing different types of charts, *JHotDraw* is a GUI framework for technical and structured graphics, and *jEdit* is a cross-platform text editor. On average, releases are separated by a two-month gap in *JFreeChart* and a nine-month gap in *JHotDraw* and *jEdit*. To demonstrate our tool's effectiveness in cases where existing approaches produce overwhelmingly large results, we purposefully use version pairs at a release granularity.

The main purpose of our evaluation is to measure how concisely a set of rules explain method-header matches. To measure **conciseness** improvement, we measure a M/R ratio = $\frac{|M|}{|Rules|}$, where $Rules$ are API change-rules identified by our approach and M represents a set of method matches explained by these change rules. A high M/R ratio means that using rules instead of plain matches significantly reduces the size of results.

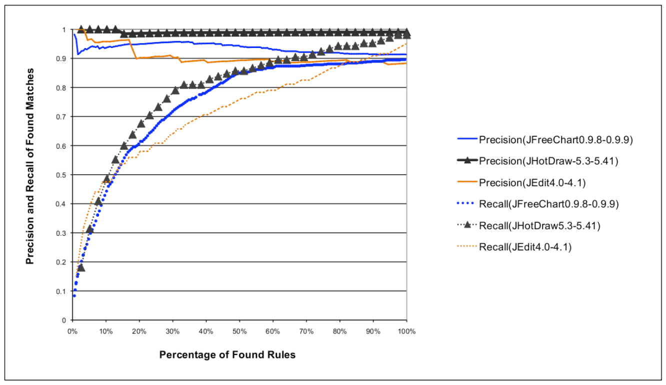


Fig. 1. Recall and precision vs. percentage of found matches

To measure the **accuracy** of found method matches, we need the ground truth—a set of correct matches. It is impossible to achieve the correct set of "ground truth" without asking the original developers of the subject programs, who are unavailable. We approximated the ground truth by constructing an evaluation data set (E) in two steps. First, we used our algorithm on each version pair in both directions. While our approach finds only n-to-1 matches in a forward direction, it can find 1-to-n matches when it is run backwards. Thus, this process could find additional matches, such as method-header matches representing API deprecation and replacement (e.g. `Foo.foo()`$\mapsto$ *deprecated* `Foo.foo()` and `Foo.foo()`$\mapsto$`Foo.bar()`. For the *JFreeChart* data set, we also added the matches found by UMLDiff to E [4]. Thus, E could include method-level matches whose names are not similar but have similar reference relationships (e.g., type usages, method-calls, field-accesses, etc.) [4]. Next, we manually inspected all matches in E to remove incorrect ones, by consulting the content of the corresponding method bodies as needed. Using the resulting data set, E, we measured **precision**, the percentage of our matches that are correct ($\frac{|E \cap M|}{|M|}$), and **recall**, the percentage of correct matches that our tool finds ($\frac{|M \cap E|}{|E|}$). The higher the precision measure is, the lower the number of false positives. The higher the recall measure is, the lower the number of false negatives.

Section 5.1 presents the evaluation of API change-rules in terms of conciseness and accuracy. In Section 5.2, we describe the impact of varying a similarity threshold (γ) and an exception threshold (ϵ). Then, in Section 5.3, we describe the evaluation of seven similarity measures for seed generation.

5.1 API-Matching Conciseness and Usefulness

Table 3 summarizes results for the three projects. We use default thresholds (γ=0.7 and ϵ=0.34) for all experiments. $|O|$ and $|N|$ are the number of methods in an old version and a new version respectively. $|O \cap N|$ is the number of methods whose name and signature did not change.

TABLE 3
Rule-based matching results

Versions	$\|O\|$	$\|N\|$	$\|O \cap N\|$	**Rule**	**Match**	**Prec.**	**Recall**	**M/R**	**Time (min.)**
JFreeChart (www.jfree.org/JFreeChart) The actual release numbers are prefixed with 0.9.									
4→5	2925	3549	1486	178	1198	0.92	0.92	6.73	21.01
5→6	3549	3580	3540	5	6	1.00	1.00	1.20	<0.01
6→7	3580	4078	3058	23	465	1.00	0.99	20.22	1.04
7→8	4078	4141	0	30	4057	1.00	0.99	135.23	43.06
8→9	4141	4478	3347	187	659	0.91	0.90	3.52	22.84
9→10	4478	4495	4133	88	207	0.99	0.93	2.35	0.96
10→11	4495	4744	4481	5	14	0.79	0.79	2.80	<0.01
11→12	4744	5191	4559	61	113	0.78	0.79	1.85	0.40
12→13	5191	5355	5044	10	145	1.00	0.99	14.50	0.11
13→14	5355	5688	5164	41	134	0.94	0.86	3.27	0.43
14→15	5688	5828	5662	9	21	0.90	0.70	2.33	0.01
15→16	5828	5890	5667	17	77	0.97	0.86	4.53	0.32
16→17	5890	6675	5503	102	285	0.91	0.86	2.79	1.30
17→18	6675	6878	6590	10	61	0.90	1.00	6.10	0.08
18→19	6878	7140	6530	98	324	0.93	0.95	3.31	1.67
19→20	7140	7222	7124	4	14	1.00	1.00	3.50	<0.01
20→21	7222	6596	4454	71	1853	0.99	0.98	26.10	62.99
MED						**0.94**	**0.93**	**3.50**	**0.43**
MIN						**0.78**	**0.70**	**1.20**	**0.00**
MAX						**1.00**	**1.00**	**135.23**	**62.99**
JHotDraw (www.jhotdraw.org)									
5.2→5.3	1478	2241	1374	34	82	0.99	0.83	2.41	0.11
5.3→5.41	2241	5250	2063	39	104	0.99	0.98	2.67	0.71
5.41→5.42	5250	5205	5040	17	17	0.82	1.00	1.00	0.07
5.42→6.01	5205	5205	0	19	4641	1.00	1.00	244.26	27.07
MED						**0.99**	**0.99**	**2.54**	**0.41**
MIN						**0.82**	**0.83**	**1.00**	**0.07**
MAX						**1.00**	**1.00**	**244.26**	**27.07**
jEdit (www.jedit.org)									
3.0→3.1	3033	3134	2873	41	63	0.87	1.00	1.54	0.13
3.1→3.2	3134	3523	2398	97	232	0.93	0.98	2.39	1.51
3.2→4.0	3523	4064	3214	102	125	0.95	1.00	1.23	0.61
4.0→4.1	4064	4533	3798	89	154	0.88	0.95	1.73	0.90
4.1→4.2	4533	5418	3799	188	334	0.93	0.84	1.78	4.46
MED						**0.93**	**0.98**	**1.73**	**1.21**
MIN						**0.87**	**0.84**	**1.23**	**0.61**
MAX						**0.95**	**1.00**	**2.39**	**4.46**

Running time is measured on a Quad-Core Intel Xeon processor running on a Mac Pro, and is described in minutes. The precision of our tool is generally high, in the range of 0.78 to 1.00. Recall is in the range 0.70 to 1.00, with median values higher than 0.90 for all three subjects.

The M/R ratio shows significant variance across different release pairs in the three subjects. The low end of the range is at or just over 1 for each subject, representing cases where each rule represents roughly a single match. The high end of the range varies from 2.39 (for *jEdit*) to nearly 244.26 (for *JHotDraw*). We observed that most matches are actually found by a small portion of rules (recall our algorithm finds rules in descending order of the number of matches). Figure 1 plots the cumulative distribution of matches for the version pairs with the median M/R ratio from each of the three projects. The x axis represents the percentage of rules found after each iteration, and the y axis represents the recall and precision of matches found up to each iteration.

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

Our tool handled the major refactorings in the subject programs quite well. For example, from release 0.9.4 to 0.9.5 of *JFreeChart*, when nearly half of the methods cannot be matched by name, our tool finds many package-level splits and low-level API changes. For example, the following change pattern is not found by UMLDiff [4], as UMLDiff simply enumerates individual return type refactorings one by one.

```
for all x:int renderer.*.draw*(*, Graph, Rect)
 returnReplace(x, int, AxisState)
```

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

The M/R ratio was extremely high from release 0.9.7 to 0.9.8 of *JFreeChart* and from release 5.42 to 6.01 of *JHotDraw* due to domain renamings: *JFreeChart*'s domain was renamed from `com.jrefinery` to `org.jfree` and *JHotDraw*'s domain was renamed from `ch.ifa` to `org.jhotdraw`.

Our approach also found rules that summarize multiple related refactorings, which no existing refactoring reconstruction tools summarize as a single change. For example, for all x: chart.*Axis.height() procedureRename(x, height, getHeight) represents multiple method renamings. These changes cannot be summarized as a single refactoring by existing refactoring reconstruction techniques, because the scope of change is *the classes that end with the name* `Axis` and not the entire program.

Evaluation of Rule Exceptions. In order to assess whether the identified rule exceptions can help developers prevent inconsistent edits, we manually inspected all rules with exceptions in *JFreeChart*. In 16 version pairs of *JFreeChart*, there were 49 rules with exceptions in total. From the inspection, we categorized them into three categories: (1) inconsistent edits which lead to bugs, (2) inconsistent edits that are benign yet suspicious and noteworthy, and (3) false positive inconsistent edits.

Out of 49 rules, 14 were inconsistent edits that led to bugs—they were corrected by developers in later versions, or they were obvious errors such as a subclass' method no longer overriding a renamed abstract method. For example, in *JFreeChart* 0.9.4→0.9.5, a developer renamed `addTitle` to `addSubtitle` in class `JFreechart` and subsequently renamed `addTitle` in two of its subclasses; however, the developer misspelled one to `addSubitle` in the `JThermometer` class. This bug was corrected in a later version. 22 out of 49 rules were benign, inconsistent edits, yet suspicious and worthwhile to note. For example, 17 methods changed their input signatures from `AxisLocation` to `RectangleEdge`, but one method did not change similarly. As another example, all classes with the name `*Category*Dataset` moved from package `org.jfree.data` to sub-package `org.jfree.data.category` except `CategoryTableXYDataset`. 13 out of 49 rules were false positive identifications of inconsistent edits. For example, we identified a wrong rule, in turn accidentally classifying a correct match (`setVerticalLabel` $\mapsto$ `setVerticalTickLabels`), as an exception.

This evaluation shows that the rule exceptions can indeed help developers find bugs and prevent inconsistent edits. It can also help developers raise appropriate design rationale questions about component organization during peer code reviews, such as *"Why wasn't* `CategoryTableXYDataset` *moved when all other* `CategoryDataset` *classes moved to the* `data.category` *subpackage?"*

Example of Unmatched Methods. Since our approach relies heavily on method name similarity and systematic renaming patterns, it cannot find matches that do not bear any textual similarity. For example, our approach cannot detect that `TriangleFigure.polygon` is renamed to `Triangle.getPolygon` since the token-level name-similarity between the two methods is very low. Furthermore, our approach cannot find 1-to-n matches, which are useful for describing replacement of a deprecated API. For example, `Drawing.orphanAll(Vector)` should have been mapped to both `Drawing.orphanAll(Vector)` (deprecated) and `Drawing.orphanAll(FigureEnumeration)` in *JHotDraw* 5.2→5.3.

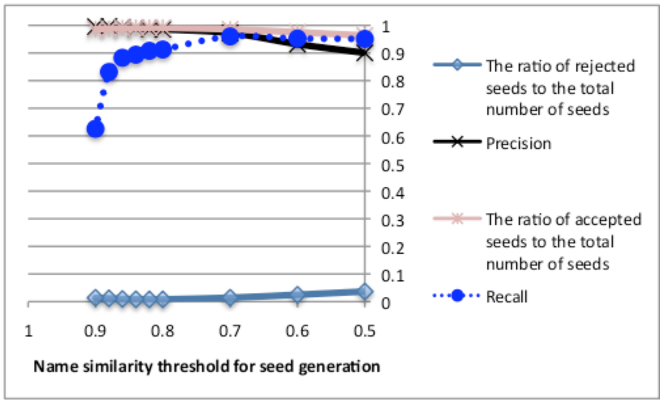


Fig. 2. Impact of seed threshold γ

5.2 Impact of Thresholds

Seed Threshold (γ). Our rule-based API-matching results, in part, depend on the quantity and quality of seeds. The seed threshold specifies the similarity required for a match to be considered in an initial set of seed matches.

Figure 2 shows how our algorithm behaves when we change the seed threshold γ for all version pairs in *JFreeChart*. The precision and recall measures are weighted-average measures. We varied γ from 0.9 to 0.5 and measured recall of seeds, precision, recall, and the ratio of rejected seeds to the total number of seeds. When γ is set high, in the range of 0.9 to 0.8, the name matching technique finds a relatively small number of seeds, but the seeds tend to be all good seeds. So, our algorithm rejects very few seeds and leverages the good seeds to quickly reach a recall of 0.65 to 0.85. However, the recall is still below 0.85 as the seeds do not contain enough transformations. As γ decreases, more seeds are produced and a higher percentage of them are bad seeds that our algorithm later rejects. Using a low threshold (<0.6) generally leads to higher recall (above 0.9). However, it lowers precision and increases the running time, as there are more candidate rules based on bad seeds. For the results in Figure 2, we observed a roughly linear increase in running time, from a total of 9 minutes (γ=0.9) to 115 minutes (γ=0.5), for all version pairs.

Exception Threshold (ϵ). We experimented with three exception thresholds: 0.25, 0.34, and 0.5 on all version pairs of *JFreeChart*. Using a low threshold increases running time from 7.54 minutes to 10.44 minutes, on average, and slightly decreases the M/R ratio from 14.33 to 14.12, on average. Surprisingly, we found that changing exception thresholds does not affect precision and recall much—the recall measure remained at 0.91 and the precision slightly decreased from 0.94 to 0.93. We suspect that this is because there are few rules with exceptions.

TABLE 4
Performance of similarity measures for seed generation (varying γ from 0.6 to 0.95 in steps of 0.05)

Similarity Measure	F measure		
	Min	Max	Med
Method-Header Level Similarity Measures			
(original): weighted token LCS	0.095	0.870	0.395
(1) bi-gram	0.075	0.779	0.287
(1′) tri-gram	0.120	0.613	0.362
(2) weighted LCS and ISC	0.004	0.169	0.059
(3) Levenshtein distance	0.051	0.581	0.228
Method-Body Level Similarity Measures			
(4) method body size	0.001	0.007	0.002
(5) method body content	0.067	0.371	0.194
(6) combined method signature and body	0.252	0.823	0.481

5.3 Various Textual Similarity Measures for Seed Generation

Since our algorithm heavily depends on the quality and quantity of seed matches, we have implemented six additional textual similarity measures for comparison against the similarity measure from our initial publication [45]:

1) N-gram based Dice's Coefficient [46]: the number of common n-grams within two strings, used by Fluri et al. [11] and Xing and Stroulia [4].
2) Weighted LCS and ISC: a weighted longest common subsequence count and intersection set count at the character level, used by S. Kim et al. [8].
3) Levenshtein Distance: a normalized edit distance in terms of delete, insert, and substitute operations, used by Fluri et al. [11].
4) Method Body Size: a normalized measure of size difference between two method bodies, used by Demeyer et al. [19].
5) Method Body Content: Dice's coefficient and Levenshtein distance-based similarity at the method body level [46].
6) Combined Method Signature and Body Similarity: an equal-weighted combination of the weighted n-gram method-header similarity and Dice's coefficient method body similarity.

To understand *how well these similarity measures alone match entities across program versions,* we evaluated the performance of seven similarity measures against the set of manually labeled matches. Using the *JFreeChart* data set, we varied the threshold (γ) from 0.6 to 0.95 in steps of 0.05 and calculated the F-measure of the precision and recall values for each measure. The F-measure is a measure of accuracy: $F = \frac{2 \times precision \times recall}{precision + recall}$.

Table 4 presents the minimum, maximum, and median F-measure of the different similarity measures. Without considering method body content, the original weighted token similarity measure (original) had the best overall performance, with the weighted bi-gram Dice's Coefficient measure (2) ranking the second. When considering method body content, the combined method signature and body similarity (6) takes the top spot.

6 Evaluation of LSdiff Change-Rules

Section 6.1 discusses a focus group study and Section 6.2 describes comparisons between LSdiff and an existing approach.

6.1 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 study is typically carried out in an early stage of product development to gather target users' opinions on new products, concepts, or messages.

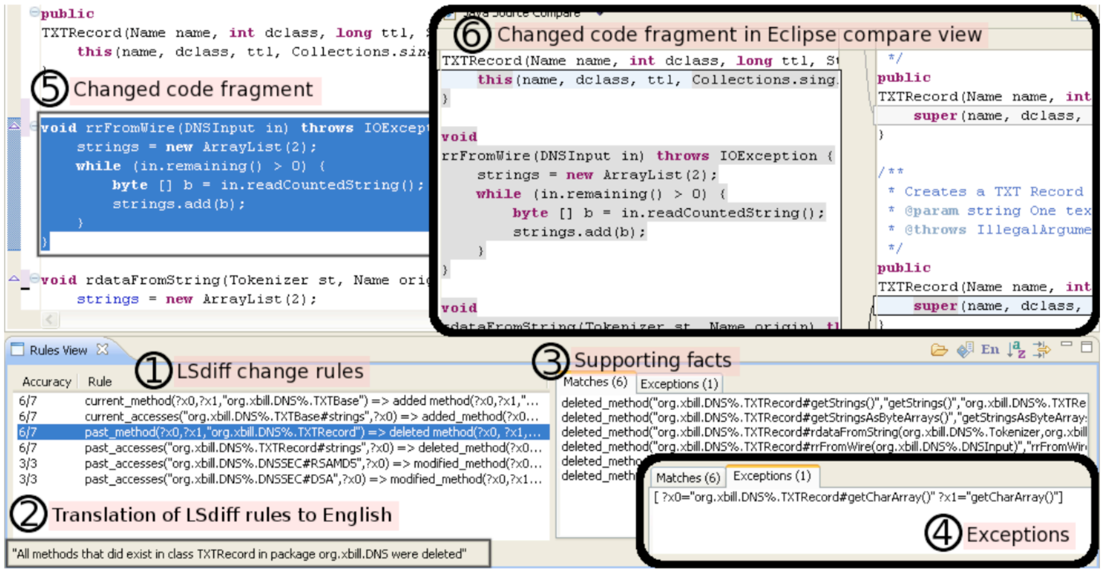
The goal of the focus group was to answer: (1) In which task contexts do programmers need to understand code changes? (2) What are difficulties of using program differencing tools such as *diff*? and (3) How can LSdiff complement existing uses of program differencing tools?

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

The hands-on trial in the focus group used a sample LSdiff output on the *CAROL* project, revision 430. We chose this change because it is a conceptually simple change based on dispersed textual modifications of 723 lines across 9 files. LSdiff identified the systematic nature of the change, inferring 16 rules and 11 facts. The LSdiff output is presented in an HTML format, which is similar to what is shown in the screen snapshot of the LSdiff Eclipse plug-in [47] (see Figure 3).

During the focus group, the first author worked as the moderator of the focus group discussion. We audio-taped the discussion and had a note-taker transcribe the conversation. The appendices I, J, and K in the first author's dissertation [41] describe the complete screener questionnaire, discussion guide, and the full transcript of the focus-group discussions, respectively.

The study found that 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. If they could develop an ideal program differencing tool, they would like to see program-wide, explicit, semantic relationships between different changed files. Many complained that



```
public
TXTRecord(Name name, int dclass, long ttl,
    this(name, dclass, ttl, Collections.sin
}

void rrFromWire(DNSInput in) throws IOExcept
    strings = new ArrayList(2);
    while (in.remaining() > 0) {
        byte [] b = in.readCountedString();
        strings.add(b);
    }
}

void rdataFromString(Tokenizer st, Name orig
    strings = new ArrayList(2);
```

```
TXTRecord(Name name, int dclass, long ttl, St
    this(name, dclass, ttl, Collections.sing.
}

void
rrFromWire(DNSInput in) throws IOException {
    strings = new ArrayList(2);
    while (in.remaining() > 0) {
        byte [] b = in.readCountedString();
        strings.add(b);
    }
}

void
rdataFromString(Tokenizer st, Name origin) tl
```

```
 */
public
TXTRecord(Name name, int
    super(name, dclass,
}

/**
 * Creates a TXT Record
 * @param string One tex
 * @throws IllegalArgume
 */
public
TXTRecord(Name name, int
    super(name, dclass,
}
```

Accuracy	Rule
6/7	current_method(?x0,?x1,"org.xbill.DNS%.TXTBase") => added method(?x0,?x1,"...
6/7	current_accesses("org.xbill.DNS%.TXTBase#strings",?x0) => added_method(?x0...
6/7	past_method(?x0,?x1,"org.xbill.DNS%.TXTRecord") => deleted method(?x0, ?x1,...
6/7	past_accesses("org.xbill.DNS%.TXTRecord#strings",?x0) => deleted_method(?x0...
3/3	past_accesses("org.xbill.DNS%.DNSSEC#RSAMD5",?x0) => modified_method(?x0...
3/3	past_accesses("org.xbill.DNS%.DNSSEC#DSA",?x0) => modified_method(?x0,?x1...

"All methods that did exist in class TXTRecord in package org.xbill.DNS were deleted"

```
deleted_method("org.xbill.DNS%.TXTRecord#getStrings()","getStrings()","org.xbill.DNS%.TXTRe
deleted_method("org.xbill.DNS%.TXTRecord#getStringsAsByteArrays()","getStringsAsByteArray
deleted_method("org.xbill.DNS%.TXTRecord#rdataFromString(org.xbill.DNS%.Tokenizer,org.xbill
deleted_method("org.xbill.DNS%.TXTRecord#rrFromWire(org.xbill.DNS%.DNSInput)","rrFromWir
deleted_metho
deleted_meth
```

```
[ ?x0="org.xbill.DNS%.TXTRecord#getCharArray()" ?x1="getCharArray()"]
```

Fig. 3. Overview based on *LSdiff* rules

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

The participants believed that LSdiff can be used in the situations where they are already using *diff*, such as during code reviews and, in particular, when there is a large amount of changes. One testing engineer said he would like to use LSdiff to understand the evolution of the component that he is writing test cases for [48], [49]. The participants believed that LSdiff's ability to discover exceptions can help programmers find missing updates and better understand design decisions.

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

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 reviewing.

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

The participants were concerned that LSdiff does not identify cross-language systematic changes such as changing a Java program and subsequently changing XML configuration files. Some were concerned that LSdiff would not provide much additional benefits for non-systematic, random, or small changes and that LSdiff may find uninteresting systematic changes.

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

6.2 Empirical Assessment of Change-Rules

We applied LSdiff to two open source projects, *CAROL* and *dnsjava*, and to LSdiff itself. We selected these programs because their medium code size (up to 30 KLOC) allowed us to manually analyze changes in these programs in detail. We did not use the same subject program pairs described in Section 5.1 because the API change-rule evaluation purposely targets a large number of method deletions and additions, while the evaluation of LSdiff targets small programs to limit the size of the rule search space. *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 indicated non-trivial changes. Its size ranged from 10,800 to 29,050 LOC and from 90 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 5,080 to 14,500 LOC and from 40 to 83 files. We also selected LSdiff's 10 versions pairs—revisions that were at least 8 hours apart and committed by different authors. Its program size ranged from 15,651 to 16,897 LOC and from 93 to 101 files.

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

(1) How often do individual changes form systematic change patterns? LSdiff is based on the observation that high-level changes are often systematic at a code level. To understand how often this observation holds true in

practice, we measured *coverage*, the percentage of facts in ΔFB explained by inferred rules: # of facts matched by rules / ΔFB. For example, when 10 rules explain 90 out of 100 facts in ΔFB, the coverage of rules is 90%.

(2) How concisely does LSdiff describe structural differences by inferring rules in comparison to an existing differencing approach that computes differences without any structure? We measured *conciseness* improvement: ΔFB / (# rules + # facts). For example, when 4 rules and 16 remaining facts explain all 100 facts in ΔFB, LD 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 developer's burden of examining the code surrounding 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 that are mentioned by the rules but are not contained in ΔFB.

Table 5 shows the results for the three data sets. These results are generated using default parameter settings: m=3, a=0.75, and k=2 because these default thresholds tend to produce good results according to our evaluation. Rule shows the number of inferred rules, and Fact shows the number of remaining facts in ΔFB not explained by any inferred rules. On average, 75% of facts in ΔFB are covered by inferred rules; this implies that 75% of structural differences form higher-level, systematic change patterns. Inferring rules improves the conciseness measure by a factor of 9.3 on average. LSdiff finds an average of 9.7 more contextual facts than ΔFB.

Restriction of Rule Styles. While our algorithm systematically enumerates all possible candidate rules, certain styles of rules have a stronger meaning than others. For example, inheritance, call, and access based dependence relationships have a stronger meaning than containment-based relationships: "all `setPort` methods that called `foo(int)` in the old version were deleted" is more informative than "all `setPort` methods were deleted." As another example, rules with constants are more specific and thus easier to understand: "all `int` type fields were deleted" and "all `port` fields with type `int` were deleted" may explain the same set of change facts, but the latter is easier to understand than the former.

To understand how the rule style restriction affects the end results, we have devised three different schemes: Option A—use our default algorithm described in Section 4.2, Option B—restrict at least one of the antecedent predicates to inheritance, method call, or field access predicates, and Option C—further restrict Option B by imposing at least one constant to appear in each predicate. Our case study on the *CAROL* version pair 429-430 shows the following results.

Using Option A, 16 rules were inferred and 11 change facts are not covered by any of the inferred rules. Using Option B, 16 rules were inferred and 12 change facts are

TABLE 5
Comparison with ΔFB

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

not covered by any of the inferred rules. We found that this restriction reduces the rule search space significantly, from 358 ungrounded rules to 180 ungrounded rules, yet the total number of rules in the final result does not change much. Many rules cover overlapping sets of change facts, and thus are filtered by our SET-COVER post processing step. However, in terms of quality, the rules produced with Option B tend to be more interesting because they describe dependence similarity among changed code rather than containment similarity. When using Option C, our algorithm finds 18 rules and the remaining 33 change facts are not covered by any of the inferred rules. We found that abandoning rules with no constants in each predicate forced our algorithm to infer specific rules and reduce its coverage.

Varying Threshold 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 in increments of 0.125, and k from 1 to 2. Table 6 shows the results of varying these parameters 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. In the case of a=0.5, the trend is slightly opposite because the low accuracy criteria enabled our algorithm to find more rules.

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.

TABLE 6
Impact of varying input parameters a, m, and k

		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

(default parameters: a=0.75, m=3, k=2)

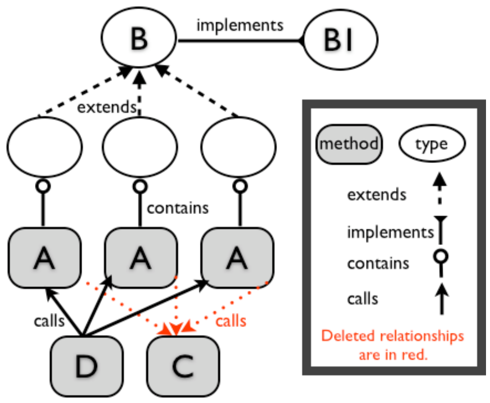


Fig. 4. Overlapping rules

7 Limitations

Our change rule inference is limited to the abstraction level of code elements and structural dependences in Java, described in Table 1. It does not model features such as exceptions thrown by a Java method nor its access modifiers. As opposed to AST differencing, the goal of LSdiff rule inference is to discover systematic change patterns at the abstraction level described in Table 1, but not to produce syntactic differences between two full ASTs. Similarly, our API rule inference is limited to method headers, and thus cannot capture renaming at a field level. Though our fact extraction analysis handles inner types, it does not handle anonymous classes with type names. To extract change facts, we analyzed Java source code using the Eclipse JDT AST syntax analysis toolkit. In addition, when finding method_calls facts, it finds static method calls but does not account for dynamic dispatching, as determination of a run-time object type requires precise alias analysis.

Our API change-rule inference heavily relies on seed generation, which uses textual similarity to find seed matches. Thus, it is prone to miss changes that involve drastic renamings such as rename `getStart` to `getFirstMillisecond`. A seed generation method that looks up synonyms in a dictionary may find more seed matches, overcoming this limitation [50].

Our API change-rules describe functional mappings (n-to-1) from the method headers deleted in the old version to the method headers added in the new version. Thus, when a method `foo` is renamed to `bar` but is kept as a deprecated method, our API-matching cannot find a mapping from `foo` to `bar`.

While it is possible to construct a golden-standard set by recording refactorings or requesting developers to manually label matches, we inspected the results in-house as the recorded refactoring scripts are unavailable and it is prohibitively expensive for developers to manually label all matches in the version history.

The current LSdiff algorithm has three limitations. First, top-down rule-learning is inefficient in that it searches a very large number of rules and then discards most of them—for example, in one of the worst cases, it reported 36 rules after generating over 1 million and finding about 4400 of them valid. This limitation comes from the fact that the same phenomena can be explained by a very large number of slightly different, yet equivalent, patterns.

Consider the example in Figure 4. Suppose that there exists a rule, "all `A` methods in `B`'s subclasses deleted calls to `C`." When `B` implements the `B1` interface and `A` methods were called by `D`, the algorithm may find another rule that explains the same method-call deletions, "all methods that were called by `D` in `B1`'s subtypes deleted calls to `C`." This first limitation leads to the second limitation: LSdiff's output is often unstable because small changes in the input programs or the order of rule-selection leads to very different final outputs. Third, it does not group related rules and facts to identify complex refactorings or changes to design patterns. The first author's follow-up work, Ref-Finder, overcomes this limitation by defining the structural constraints of each refactoring type as template logic rules and uses a logic query approach to infer concrete refactoring instances [24].

As described in Section 4, our approach takes input parameters (thresholds), which must be tuned by developers. Though the use of threshold parameters is common in programming differencing and refactoring reconstruction, this has a practical implication because the thresholds need to be tuned.

8 Conclusion

To help developers reason about software changes at a high level, this article introduced a rule-based program differencing approach that extracts high-level change descriptions as logic rules. This rule-based approach is instantiated at two abstraction levels: first at the method-header level and then at the level of code elements and their structural dependences.

This rule-based change inference approach has been assessed both quantitatively and qualitatively through its application to multiple open source projects and through a focus group study with professional developers from a large E-commerce company. The participants' comments show that our approach is promising, both as a complement to *diff*'s file-based approach and as a way to help developers discover potential bugs by identifying exceptions to inferred systematic changes. The quantitative assessments show that our rule-based approach produces concise results compared to other refactoring reconstruction tools.

ACKNOWLEDGMENTS

The authors thank Wei Wu and Hoan Nguyen for their help with the evaluation of the API-matching tool. This work was in part supported by National Science Foundation under the grants CCF-1043810, CCF-1117902, and CCF-1149391 and by a Microsoft SEIF award.

REFERENCES

[1] A. J. Ko, R. DeLine, and G. Venolia, "Information needs in collocated software development teams," in *International Conference on Software Engineering*, 2007, pp. 344–353.
[2] J. Sillito, G. C. Murphy, and K. D. Volder, "Asking and answering questions during a programming change task," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 434–451, 2008.
[3] R. C. Miller and B. A. Myers, "Outlier finding: focusing user attention on possible errors," in *UIST '01: Proceedings of the 14th annual ACM Symposium on User Interface Software and Technology*. New York, NY, USA: ACM, 2001, pp. 81–90.
[4] Z. Xing and E. Stroulia, "UMLDiff: an algorithm for object-oriented design differencing," in *ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2005, pp. 54–65.
[5] P. Weißgerber and S. Diehl, "Identifying refactorings from source-code changes," in *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 231–240.
[6] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, "AURA: a hybrid approach to identify framework evolutions," in *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. New York, NY, USA: ACM, 2010, pp. 325–334.
[7] H. A. Nguyen, T. T. Nguyen, G. Wilson, A. T. Nguyen, M. Kim, and T. Nguyen, "A graph-based approach to API evolution," in *OOPSLA '10: Proceedings of the 2010 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications*. New York, NY, USA: ACM, 2010, p. 10.
[8] S. Kim, K. Pan, and J. E. James Whitehead, "When functions change their names: Automatic detection of origin relationships," in *WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 143–152.
[9] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated detection of refactorings in evolving components," in *ECOOP '06: Proceedings of the European Conference on Object-Oriented Programming*. Springer, 2006, pp. 404–428.
[10] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," *Communications of the ACM*, vol. 20, no. 5, pp. 350–353, 1977.
[11] 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*, vol. 33, no. 11, p. 18, November 2007.
[12] I. Neamtiu, J. S. Foster, and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," in *MSR'05: the Workshop on Mining Software Repositories*, 2005, pp. 2–6.
[13] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine, "Dex: A semantic-graph differencing tool for studying changes in large code bases," in *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 188–197.
[14] W. Yang, "Identifying syntactic differences between two programs," *Software – Practice & Experience*, vol. 21, no. 7, pp. 739–755, 1991.
[15] T. Apiwattanapong, A. Orso, and M. J. Harrold, "A differencing algorithm for object-oriented programs," in *ASE '04: Proceedings of the 19th IEEE International Conference on Automated Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 2–13.
[16] S. Horwitz, "Identifying the semantic and textual differences between two versions of a program," in *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*. New York, NY, USA: ACM, 1990, pp. 234–245.
[17] D. Jackson and D. A. Ladd, "Semantic diff: A tool for summarizing the effects of modifications," in *ICSM '94: Proceedings of the International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 1994, pp. 243–252.
[18] O. C. Chesley, X. Ren, and B. G. Ryder, "Crisp: A debugging tool for java programs," in *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 401–410.
[19] S. Demeyer, S. Ducasse, and O. Nierstrasz, "Finding refactorings via change metrics," in *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. New York, NY, USA: ACM, 2000, pp. 166–177.
[20] L. Zou and M. W. Godfrey, "Using origin analysis to detect merging and splitting of source code entities," *IEEE Transactions on Software Engineering*, vol. 31, no. 2, pp. 166–181, 2005.
[21] F. V. Rysselberghe and S. Demeyer, "Reconstruction of successful software evolution using clone detection," in *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*. Washington, DC, USA: IEEE Computer Society, 2003, p. 126.
[22] G. Antoniol, M. D. Penta, and E. Merlo, "An automatic approach to identify class evolution discontinuities," in *IWPSE '04: Proceedings of the Principles of Software Evolution, 7th International Workshop*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 31–40.
[23] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code." *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.
[24] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *ICSM 2010: 2010 IEEE International Conference on Software Maintenance*, 2010, pp. 1 –10.
[25] E. Visser, "Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9," *Domain-Specific Program Generation*, vol. 3016, pp. 216–238, 2004.
[26] J. R. Cordy, "The TXL source transformation languages," *Science of Computer Programming*, vol. 61, no. 3, pp. 190–210, 2006.
[27] M. Boshernitsan, S. L. Graham, and M. A. Hearst, "Aligning development tools with the way programmers think about code changes," in *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*. New York, NY, USA: ACM, 2007, pp. 567–576.
[28] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller, "Documenting and automating collateral evolutions in Linux device drivers," in *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*. New York, NY, USA: ACM, 2008, pp. 247–260.
[29] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 563–572.
[30] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, "Chianti: a tool for change impact analysis of Java programs," in *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. New York, NY, USA: ACM, 2004, pp. 432–448.
[31] J. Wloka, B. G. Ryder, and F. Tip, "JUnitMX - a change-aware unit testing tool," in *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 567–570.
[32] W. F. Opdyke and R. E. Johnson, "Refactoring: An aid in designing application frameworks and evolving object-oriented systems," in *SOOPPA2000: Proceedings of the Symposium on Object Oriented Programming Emphasizing Practical Applications*, 2000.
[33] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 2000.

[34] T. Mens and T. Tourwe, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[35] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP'97: the 11th European Conference on Object-oriented Programming*, vol. 1241. Lecture Notes in Computer Science 1241, 1997, pp. 220–242.

[36] P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton, "N degrees of separation: multi-dimensional separation of concerns," in *ICSE '99: Proceedings of the 21st International Conference on Software Engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1999, pp. 107–119.

[37] S. Breu and T. Zimmermann, "Mining aspects from version history," in *ICSE'06: Proceedings of the 28th International Conference on Automated Software Engineering*, 2006, pp. 221–230.

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

[39] M. Kim and D. Notkin, "Discovering and representing systematic code changes," in *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 309–319.

[40] D. Janzen and K. D. Volder, "Navigating and querying code without getting lost," in *AOSD'03: Proceedings of the International Conference on Aspect Oriented Software Development*, 2003, pp. 178–187.

[41] M. Kim, "Analyzing and inferring the structure of code changes," Ph.D. dissertation, University of Washington, 2008.

[42] K. D. Volder, "Type-oriented logic meta programming," Ph.D. dissertation, Vrije Universiteit Brussel, 1998.

[43] E. Balas and M. W. Padberg, "Set Partitioning: A Survey," *SIAM Review*, vol. 18, pp. 710–760, 1976.

[44] S. Kok and P. Domingos, "Learning the structure of markov logic networks," in *ICML '05: Proceedings of the 22nd international conference on Machine learning*. New York, NY, USA: ACM, 2005, pp. 441–448.

[45] M. Kim, D. Notkin, and D. Grossman, "Automatic inference of structural changes for matching across program versions," in *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 333–343.

[46] L. R. Dice, "Measures of the amount of ecologic association between species," *Ecology*, vol. 26, no. 3, pp. 297–302, 1945.

[47] A. Loh and M. Kim, "Lsdiff: a program differencing tool to identify systematic structural differences," in *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. New York, NY, USA: ACM, 2010, pp. 263–266.

[48] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," in *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*. New York, NY, USA: ACM, 2008, pp. 481–490.

[49] T. Schäfer, J. Jonas, and M. Mezini, "Mining framework usage changes from instantiation code," in *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*. New York, NY, USA: ACM, 2008, pp. 471–480.

[50] K. Taneja, D. Dig, and T. Xie, "Automated detection of API refactorings in libraries," in *ASE'07: Proceedings of the 22nd IEEE/ACM international conference on Automate Software Engineering*, ser. ASE 2007. New York, NY, USA: ACM, 2007, pp. 377–380.

Miryung Kim received a B.S. in Computer Science from Korea Advanced Institute of Science and Technology in 2001. She received a Ph.D. in Computer Science & Engineering from the University of Washington in 2008 and then joined the faculty at the University of Texas at Austin where she is currently an assistant professor. Her research interests lie in the area of software engineering, with a focus on improving developer productivity in evolving software systems.

David Notkin received his Sc.B. in computer science from Brown University in 1977 and his Ph.D. in computer science from Carnegie Mellon University in 1984. In 1984 he joined the Computer Science & Engineering faculty at the University of Washington. He served as department chair from 2001-06 and now serves as Professor and Bradley Chair. His research and educational interests are in software engineering in general, and software evolution in particular. He is a Fellow of the IEEE and of the ACM, and he currently serves as editor-in-chief of ACM Transactions on Software Engineering and Methodology.

Dan Grossman received a B.A. in Computer Science and a B.S. in Electrical Engineering from Rice University in 1997. He received a Ph.D. in Computer Science from Cornell University in 2003 and then joined the faculty at the University of Washington where he is currently an associate professor. His research interests lie in the area of programming languages, ranging from theory to design to implementation, with a focus on improving software quality.

Gary Wilson Jr. received his B.S. in Electrical Engineering from The University of Texas in 2003. He is currently a lead software engineer at The University of Texas, where he is also pursuing a master's degree in software engineering. He is a core contributor to the Django open source project.

APPENDIX

It is not possible to conduct a head to head comparison of our approach with other refactoring reconstruction techniques, because they are developed for different purposes, such as updating client applications broken due to API refactoring. Nevertheless, we compare a set of method headers explained by our API change-rules against method matches found by other tools to show that our matches are roughly on par with other approaches and that our approach, in turn, produces more concise outputs in terms of change-rules based on the matches.

In particular, our comparison focused on the following six approaches:

1) Xing and Stroulia's approach (XS) [4].
2) Weißgerber and Diehl's approach (WD) [5].
3) S. Kim et al.'s approach (KPW) [8].
4) Wu et al.'s approach (WG) [6].
5) Nguyen et al.'s approach (NN) [7].
6) Dig et al.'s approach (DC) [9].

While our approach primarily leverages name similarity to match corresponding method headers, the other approaches use various content, name, and structure similarity criteria to map methods between two program versions. For other approaches, we used the results produced with default thresholds set by the authors. For example, we found XS's results on their web site.[3] Similarly, we found DC's results on their web site.[4] When comparing with DC's results, we excluded the matches derived from return type change refactorings in our results, because such changes are not detected by DC. WD and KPW results were sent to us by the respective authors. The first author participated in developing the WG and NN approaches as a co-author. In the case of WD, because it finds redundant refactoring events for a single match, we compared our results with both (1) all refactoring candidates (RC_{all}) and (2) only the top-ranked refactoring candidates (RC_{best}). We also developed a tool that deduces method matches from the refactorings inferred by XS and WD, because these approaches find refactorings instead of method-level matches.

When comparing the size of inferred changes, in our approach we measured the number of rules. In XS, WD, and DC, we measured the number of relevant refactorings that explain method-level matches. In KPW, WG, and NN, we measured the number of method matches, as they do not output their results in terms of refactorings. Our comparative evaluation was done all at the method level and we did not consider the types of refactorings that do not contribute to finding method matches. For example, XS detects changes to method visibility and *encapsulate field* refactorings, but we did not include those refactorings in our comparison.

For each common version pair, we categorized the set of method-header level matches into three sets: (A) matches found by both approaches, (B) matches found by only our approach, and (C) matches found by only their approach. We randomly sampled at least 50 matches from each set and manually inspected them. Note that this random sampling was done for *each version pair*, and thus the total number of inspected matches is far greater than 150 matches per each subject program. We then estimated the overall precision as a weighted average of precision measures. Suppose that, in *JFreeChart*: NN and our approach found 4249 matches in common with an estimated precision of 1.00; 395 matches are found by our tool only with an estimated precision 0.32; and 264 matches are found by NN only with an estimated precision of 0.59. We estimated that our tool has an overall precision of $0.94 = (1.00 \times 4249 + 395 \times 0.32)/(4249 + 395)$, while NN has an overall 0.98 precision. We also estimated an overall recall similarly; we estimated our tool has a recall of $0.97 = (1.00 \times 4249 + 395 \times 0.32)/(4249 \times 1.00 + 395 \times 0.32 + 264 \times 0.59)$.

The comparison results are summarized in Table 7. Our approach improves the conciseness of matching results significantly without much sacrifice in precision and recall. The improvement in *conciseness* must be carefully interpreted because other approaches were developed for different purposes, such as migrating a client application when libraries evolve, and thus their primary goal is not about presenting refactoring results in a concise manner. The following summarizes the highlights of our detailed comparison:

- XS: Matches missed by XS often involve both rename and move refactorings. For example, it cannot handle combinations of move and rename refactorings such as "move `CrosshairInfo` class from `chart` to `chart.plot` package and rename it to `CrosshairState`." Matches missed by our tool often had a very low name similarity.
- WD: WD missed many matches when compound transformations were applied. Our tool missed some matches because using γ=0.65 did not generate enough seeds to find them.
- KPW: KPW missed many matches because it cannot accept correct matches when their overall similarity score is lower than a certain threshold. It also cannot easily prune incorrect matches once their overall similarity score is over a certain threshold and is the highest among potential matches. On the other hand, our algorithm tends to reject matches whose transformation is an isolated incident, even if the similarity score is high. Our tool's incorrect matches usually come from bad seeds that, coincidentally, have similar names.
- WG: WG finds more matches than ours. The boost in recall for WG comes from finding many-to-one

3. http://webdocs.cs.ualberta.ca/~stroulia/Zhenchang_Xing_Old_Home/jdevan.html
4. https://netfiles.uiuc.edu/dig/RefactoringCrawler/evaluations.html

TABLE 7
Comparison: number of matches, precision, recall, and size of result

Program	Our Approach				Other Approaches				Improvement	
	Matches	Rules	Precision	Recall	Matches	Refactorings	Precision	Recall	Matches	Size
					Xing and Stroulia (XS)					
JFreeChart (0.9.4-0.9.21)	9633	939	0.97	0.98	8883	4004	0.99	0.92	8%	77%
					Weißgerber and Diehl (WD), RCAll					
jEdit (2715 check-ins)	1488	906	0.98	0.93	1333	2133	0.86	0.73	12%	58%
Tomcat (5096 check-ins)	2984	1033	0.92	0.87	3608	3722	0.75	0.86	(17%)	72%
					Weißgerber and Diehl (WD), RCBest					
jEdit (2715 check-ins)	1488	906	0.98	0.96	1172	1218	0.93	0.72	27%	26%
Tomcat (5096 check-ins)	2984	1033	0.93	0.89	2907	2700	0.89	0.82	3%	62%
					S. Kim et al. (KPW)					
jEdit (1189 check-ins)	2009	1119	0.96	0.96	1430	N/A	0.98	0.70	40%	22%
ArgoUML (4683 check-ins)	4612	2127	0.95	0.95	3819	N/A	0.98	0.82	21%	44%
					Nguyen et al. (NN)					
JFreeChart (0.9.5-0.9.19)	4644	686	0.94	0.97	4433	N/A	0.98	0.97	4%	85%
JHotDraw (5.2-6.0b1)	3398	109	0.99	0.99	3400	N/A	0.99	0.99	(0.05%)	96%
					Wu et al. (WG)					
JFreeChart (0.9.11-0.9.12)	113	61	0.78	0.79	115	N/A	0.83	0.99	(2%)	66%
jEdit (4.1-4.2)	334	188	0.93	0.84	318	N/A	0.81	0.99	(36%)	40%
JHotDraw (5.2-5.3)	82	34	0.99	0.83	100	N/A	0.97	0.99	(22%)	47%
					Dig et al. (DC)					
JHotDraw (5.2-5.3)	78	34	0.99	0.83	24	24	1.00	0.26	71%	(29%)

matches or simply deleted target methods, which our approach does not model explicitly. Furthermore, by considering similarity in method-call relationships, it can find matches when the method headers do not bear any textual name similarity.

- NN: NN finds about the same number of matches as ours, yet our approach significantly reduces the size of the results using rule-based representations. NN is similar to XS, sharing similar strengths and limitations.
- DC: DC finds 1-to-n matches to account for deprecated APIs, thus it can map `orphanAll(Vector)` to both `orphanAll(Vector)` (deprecated) and `orphanAll(FigureEnumeration)`. Since its results are intended to migrate clients broken due to API refactoring, it produces results with high precision yet its recall was lower than our approach.