LSdiff: A Program Differencing Tool to Identify Systematic Structural Differences

Alex Loh
Department of Computer Science
The University of Texas at Austin
alexloh@cs.utexas.edu

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

ABSTRACT

Program differencing tools such as GNU *diff* identify individual differences but do not determine how those differences are related to each other. For example, an *extract superclass* refactoring on several subclasses will be represented by *diff* as a scattered collection of line additions and deletions which must be manually pieced together. In our previous work, we developed LSdiff, a novel program differencing technique that automatically identifies systematic structural differences as logic rules. This paper presents an LSdiff Eclipse plug-in that provides a summary of systematic structural differences along with textual differences within an Eclipse integrated development environment. This plug-in provides several additional features to allow developers to interpret LSdiff rules easily, to select the abstraction level of program differencing analysis, and to reduce its running time through incremental program analysis.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments—*integrated environments*

General Terms

Design, Experimentation, and Measurement

Keywords

Software evolution, program differencing, code change

1. INTRODUCTION

Developers use program differencing tools to understand what changed between two versions while carrying out peer code reviews, resolving parallel edit conflicts, or isolating failure inducing changes.

The ubiquitous program differencing tool, *diff*, identifies individual textual differences at a line level, even though high-level software changes such as refactorings [6] and cross-cutting modifications [8] often consist of a group of changes that share similar structural characteristics. Thus, *diff* leaves it to developers to read through a large number of textual differences and mentally sort them into related logical groups. This problem is not exclusive to *diff* but also occurs in other program differencing techniques, even those that compute differences at different abstraction levels, such as abstract syntax trees [5], or control flow graph nodes [1].

In our previous work [9], we presented Logical Structural Diff (LSdiff), a novel program differencing technique that infers systematic structural differences as change-rules. LSdiff makes it easier for developers to understand code changes by grouping related differences as a single rule and by finding exceptions that indicate missed or inconsistent updates.

LSdiff extracts facts about code elements (packages, types, methods, and fields) and their structural dependencies (subtyping, overriding, method-calls, and field-accesses) from the old and new program versions into factbases FB_O and FB_N respectively. It then computes $\Delta\mathsf{FB}$, the set difference of FB_O and FB_N, and infers change-rules that encode systematic structural differences from these three factbases. Each *change-rule* is a Horn clause where the antecedents are facts from FB_O and FB_N and the consequent is a fact in $\Delta\mathsf{FB}$. For example, a rule past_subtype("Shape", c) ⇒ added_method(m, "calcArea", c) means each class c that was a subtype of `Shape` in the old version added a `calcArea` method in the new version. LSdiff also identifies *exceptions*, a set of facts that satisfy a rule's antecedents but not its consequent.

Our prior focus group [9] study showed that LSdiff can complement existing uses of *diff*. Thus, we developed an LSdiff Eclipse plug-in to allow developers to run LSdiff within their integrated development environment (IDE) and examine LSdiff output along with textual differences. Our tool also provides three new features that enhance its usability and performance. First, as developers are often not familiar with the Prolog-based syntax of LSdiff rules, we developed a translator that generates English language descriptions from LSdiff rules. Second, our plug-in avoids processing unmodified files that do not contribute to structural differences between two program versions by making fact extraction analysis incremental. Third, it enables developers to select the granularity of program differencing, discussed further in Section 3.1. Running the tool at a coarser granularity produces higher level change-rules by aggregating lower level facts.

Section 2 illustrates LSdiff's features using a code review scenario. Section 3 describes the tool implementation of LSdiff. Section 4 discusses related work. Section 5 concludes with a summary of LSdiff.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE '10, May 2-8 2010, Cape Town, South Africa
Copyright 2010 ACM 978-1-60558-719-6/10/05 ...$10.00.

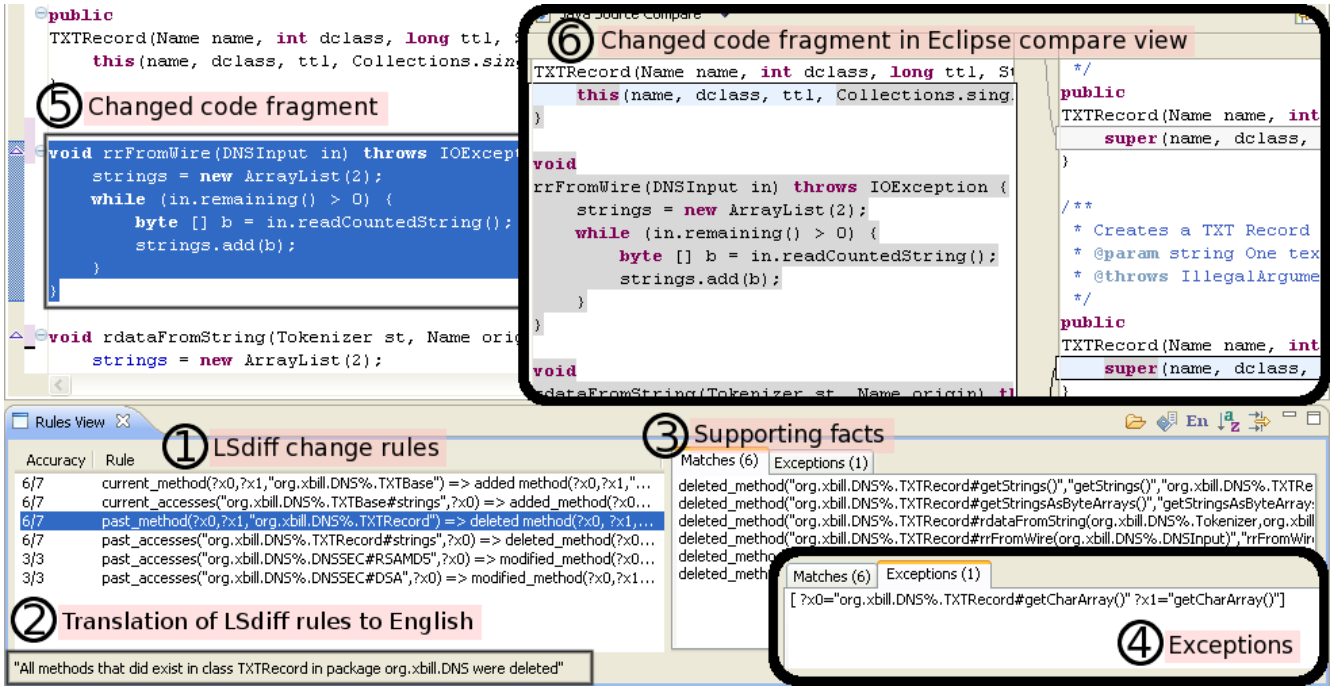

Figure 1: A screen snapshot of the LSdiff Eclipse plug-in and its features

2. LSDIFF FEATURES

A user begins by selecting two versions of a program from either the local workspace or a version control system. From the rules and facts presented in the LSdiff view, the user may expand a rule to examine the structural differences that are part of the identified change pattern or navigate to corresponding code fragments.

Suppose Cathy reviews the changes made by her co-worker Alan. Alan's check-in message reads "Performed an *extract superclass* refactoring." Using *diff*, Cathy found 49 hunk-level differences in 17 files where each hunk consists of contiguous modified lines.[1] Without manually browsing through them, Cathy cannot easily identify which differences are part of the refactorings or if Alan had missed anything. LSdiff is able to summarize Alan's changes into rules, reducing the number of items Cathy has to investigate. She may be able to guess the high-level change from these rules and to find potential missed updates from the rules' exceptions.

Figure 1 shows the results of running LSdiff. LSdiff summarized 49 hunk-level differences into 6 rules. For example, the third rule states that all methods in the org.xbill.DNS.TXTRecord class were deleted (see ① in Figure 1).

Being new to LSdiff, Cathy may be unable to interpret this rule if she cannot remember what each argument in the current_method predicate means. She clicks on the *translate to English* button to see the English translation displayed on the status bar (see ② in Figure 1).

Looking at these six rules, Cathy notices that the first and third rules appear related. The first rule, current_method(?x0, ?x1, "org.xbill.DNS%.TXTBase") ⇒ added_method(?x0, ?x1, "org.xbill.DNS%.TXTBase"), states that methods in the TXTBase class were added in the new version, while the third rule,

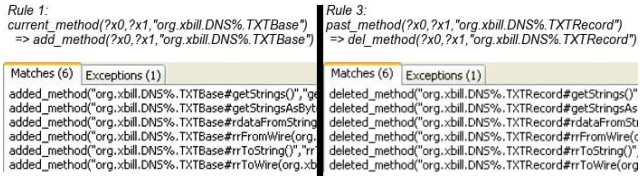

Figure 2: Supporting facts for the first and third rules show the deleted methods being added to another class

past_method(?x0, ?x1, "org.xbill.DNS%.TXTRecord") ⇒ deleted_method(?x0, ?x1, "org.xbill.DNS%.TXTRecord"), states that methods in the TXTRecord class were deleted. The names of these two classes look similar, although she remembers seeing only TXTRecord in the old version. Cathy clicks on the third rule to open Eclipse's editor on the TXTRecord class and finds that in the new version TXTRecord now extends TXTBase instead of Record. TXTBase in turn extends Record in the new version. She then decides to view the constituent structural differences by clicking on the *explain* button (see ③ in Figure 1). From the facts view in Figure 2, she realizes that for each deleted method, a method with the same name was added. She suspects that this is the *extract superclass* refactoring that Alan mentioned in his check-in message.

Next, Cathy notices that the accuracy for the third rule is not 100%. The accuracy of a rule indicates the number of times the rule was true out of the number of times it was applicable. She then switches to the *exceptions* tab to see where this deviation occurred (see ④ in Figure 1). Cathy may further inspect this exception in two ways. Left-clicking on a matched fact or an exception opens the editor to the corresponding code fragment. Right-clicking on a fact or an exception brings up the context menu with the option to use Eclipse's *compare* tool to view the corresponding *diff*

[1]This example was adapted from running LSDiff on DNSJava versions 2.0.2 and 2.0.3, http://www.xbill.org/dnsjava/

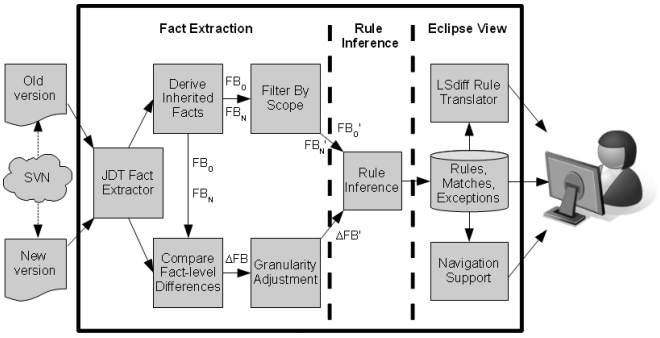

Figure 3: Architecture of LSdiff

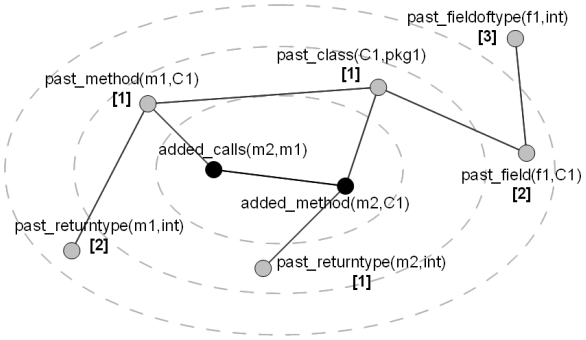

Figure 4: A dependency graph annotated with the hop distance of each fact

outputs (see ⑤ and ⑥ in Figure 1 respectively).

After examining the source code, Cathy decides that this method, `getCharArray()`, should have been updated but was missed by Alan. To find such missed updates using *diff* would have been very difficult. Cathy would have had to read through every difference to find the relevant ones, mentally compose them together to understand the refactoring, then browse through the old version's source code to find potentially incomplete or inconsistent updates.

3. IMPLEMENTATION

As shown in Figure 3, there are three parts to our plug-in: (1) a fact extractor that processes files to identify changes to a program's structure, (2) a rule inference engine that finds rules among those changes, and (3) a viewer that allows a user to investigate the inferred rules and facts interactively.

The previous implementation of LSdiff is a standalone tool which uses JQuery [7] to extract structural facts from two program versions and outputs a list of change rules. Our current implementation extracts facts using Eclipse's Java Development Tools (JDT) library and is compatible with any version of Eclipse after version 3.2. Repository retrieval is compatible with Subversion version 1.5. A fully working implementation is available for download.[2] We also changed the original fact extraction approach to an incremental approach and added features to adjust the granularity of program differencing and to manipulate the scope of rule inference by post-processing the factbases.

3.1 Extraction of Structural Differences

The fact extractor computes FB_O and FB_N from the old and new versions and computes $\Delta\mathsf{FB}$, the differences between them. It also post-processes these factbases to optimize rule-inference.

Factbase Construction. A user can select the program versions to be processed by the fact extractor from either the local workspace or a revision in an SVN repository. The fact extractor first parses all files in the old version using Eclipse's JDT library to obtain structural facts about code elements (packages, types, methods, and fields) and structural dependencies (containment, subtyping, method-calls, and field-accesses). Information about how a program's structure is represented as logic predicates is detailed in our previous paper on LSdiff [9]. The fact extractor then derives overriding-related facts—inherited_method and inherited_field—using the Tyruba logic programming system [4].

[2] `http://www.cs.utexas.edu/~alexloh/lsdiff/`

The resulting set of facts form FB_O.

Next, the fact extractor parses modified and added files in the new program version to compute $\Delta\mathsf{FB}$. This incremental approach improves on the original LSdiff by avoiding re-processing of unmodified files. However, some code elements may have structural changes even though their source files were 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` is changed from `A.m` to `B.m`, even though the source file containing class `C` was not modified. We use Tyruba to derive such inheritance-related differences from subtyping and member declaration facts in the old version and the initially computed $\Delta\mathsf{FB}$. We also handle a few other rare cases caused by moving or renaming code elements. For instance, when a class `packageA.C` is moved from `packageA` to `packageB`, all classes that instantiate `C` and import both packages will change their structural dependencies even though they have no textual modifications. The fact extractor detects such cases via pattern matching on $\Delta\mathsf{FB}$ and re-processes the affected files.

Filtering FB_O and FB_N for Rule-Inference. To improve the performance of rule inference, the fact extractor prunes facts beyond a certain hop distance. Figure 4 shows a fact dependency graph annotated with the hop distance of each contextual (gray) fact from the changed (black) facts about the newly added method `m2` and its call to `m1`. An edge indicates that two facts share the same code element constant. Hop distance is the number of edges to the nearest added, deleted, or modified fact in the graph. In Figure 4, if the user sets the threshold to 1, only past_class("C1", "pkg1"), past_method("m1", "C1"), and past_returntype("m2", "int") will be selected, whereas a threshold of 3 will select all facts in the graph. Distant contextual facts are unlikely to contribute to finding shared structural characteristics of modified code. Hence, removing them in FB_O and FB_N improves the efficiency of rule-inference while sacrificing only a small amount of contextual information in the inferred rules.

Adjustable Granularity. The fact extractor allows the user to select its granularity, the level of abstraction of facts in $\Delta\mathsf{FB}$. A coarser granularity not only improves running time by reducing the number of facts, but also provides an aggregated output by abstracting low-level details. There are six levels of granularity in LSdiff: *package*, *type*, *type hierarchy*, *field*, *method*, and *method body*. 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 will be aggregated into a single modified_type fact.

3.2 Rule Inference

Rule inference takes as input ΔFB, post-processed FB_O and FB_N, and outputs the list of change-rules. For each rule, LSdiff also returns the set of matches, facts that support the rule, and exceptions, facts which contradict the rule.

The rule inference algorithm takes as input four parameters: (1) m, the minimum number of facts the rule must match, (2) a, the minimum accuracy of the rule, where accuracy is number of matches divided by total number of matches and exceptions, (3) k, the maximum number of literals in the antecedent, and (4) β, a constant used to bound the search space . It then uses a bounded-depth search to find rules. In each iteration, the length of the antecedent is increased by one, and new candidate rules are created by trying all possible substitutions. Each candidate rule is then checked using the Tyruba logic query engine and selected for the next iteration only if it exceeds the specified accuracy and support thresholds. LSdiff's rule inference algorithm is detailed elsewhere [9].

3.3 User Interface

As seen in Figure 1, LSdiff displays structural differences as a set of rules and their supporting facts.

Navigating from LSdiff Rule to Source Code. Left-clicking on a fact opens the editor to view the associated code element; right-clicking brings up the context menu with the option to run Eclipse's *compare* tool on the relevant file. If the fact is associated with more than one code element—for instance a calls fact that is associated with multiple call sites—then the first one is chosen.

Translation of LSdiff Rule to English Description. LSdiff outputs its rules in a Prolog-like syntax which is both compact and precise. However, users who are unfamiliar with this notation may find it hard to read. Even experienced users may not always remember what each argument in a predicate means or may find the use of fully qualified names too verbose.

To improve LSdiff's accessibility, our tool translates its rules into English descriptions using templates. A subject is chosen from the constants bound to one of the antecedents, a verb is chosen from the consequent's predicate, and optionally an object is chosen from the consequent's constant. Our translation distinguishes facts in FB_O and FB_N using past and present tenses. A preliminary evaluation suggests that most human subjects prefer the generated English descriptions to the original Prolog-like syntax.

4. RELATED WORK

Several program differencing techniques improve on *diff* by organizing individual changes into related groups. Some techniques group by dependencies [2]; others groups by change history [11]. Refactoring reconstruction tools organize structural changes into refactorings. SemDiff [3] groups structural changes based on their call structure. UMLDiff [10] uses name similarity and relationship with surrounding code elements to match moved methods. UMLDiff and SemDiff are able to compose a pair of method addition and deletion into a method replacement. Crisp [2] organizes atomic changes, such as method additions or deletions, into a directed graph based on structural dependency and uses this graph to isolate failure-inducing changes. In our *extract superclass* example, Crisp will group each method addition with changes in its caller. On the other hand, LSdiff groups differences that share structural characteristics, such as the deletion of a method `m` from all classes that inherit the `C` class, and represents them as a logic rule.

5. SUMMARY

LSdiff assists developers in investigating program differences by discovering and summarizing systematic structural differences as change-rules. Our plug-in makes it easier for developers to examine both the change-rules and their constituent differences and to navigate to the corresponding code element in the Eclipse editor. To improve usability, LSdiff presents automatically-translated English descriptions. It also allows developers to manipulate the granularity and scope of structural program differencing.

Acknowledgments

We thank Marynia Demkowicz for designing and implementing the LSdiff rule translator. This work was supported in part by IBM corporation under 2009 IBM Jazz Innovation Award 'Assisting Code Review Tasks by Identifying and Summarizing Systematic Code Changes' and NSF Grant SHF-0910818.

6. REFERENCES

[1] T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In *ASE '04*, pages 2–13, 2004.

[2] O. C. Chesley, X. Ren, and B. G. Ryder. Crisp: a debugging tool for java programs. In *ICSM '05*, 2005.

[3] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *ICSE '08*, pages 481–490, New York, NY, USA, 2008. ACM.

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

[5] B. Fluri, M. Wüersch, M. Pinzger, and H. Gall. Change distilling:tree differencing for fine-grained source code change extraction. *TSE*, 33(11):725–743, November 2007.

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

[7] D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *AOSD '03*, pages 178–187, New York, NY, USA, 2003. ACM.

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP '97*. Springer-Verlag, 1997.

[9] M. Kim and D. Notkin. Discovering and representing systematic code changes. In *ICSE '09*, pages 309–319, Washington, DC, USA, 2009. IEEE Computer Society.

[10] Z. Xing and E. Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *ASE '05*, pages 54–65, New York, NY, USA, 2005. ACM.

[11] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *TSE*, 31(6):429–445, July 2005.