Identifying Static Analysis Techniques for Finding Non-fix Hunks in Fix Revisions[*]

Yungbum Jung, Hakjoo Oh, and Kwangkeun Yi
Seoul National University
{dreameye, pronto, kwang}@ropas.snu.ac.kr

ABSTRACT

Mining software repositories for bug detection requires accurate techniques of identifying bug-fix revisions. There have been many researches to find exact bug-fix revisions. However there are still noises, we call these noises non-fix hunks, even in exactly identified bug-fix revisions. Our goal is to remove these non-fix hunks automatically. First we inspected every 50 bug-fix revisions of three open source projects (Eclipse, Lucene, and Columba). Among total 2146 hunks we found 179 non-fix hunks. We classified these non-fix hunks into 11 patterns. For all patterns we enumerate enabling static analysis techniques.

Categories and Subject Descriptors

D2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; K6.3 [**Management of Computing and Information Systems**]: Software Management – Software maintenance

General Terms

Measurement, Reliability, Experimentation, Human Factors.

Keywords

Mining Software Repository, Static Analysis

1. INTRODUCTION

Identifying bug-fix revisions in software repositories is an important problem. Many applications such as automatic identification of bug-introducing changes [9], understanding of bug-fix patterns [13], and predicting faults from cached history [10] are based on identified bug-fix revisions and their hunks, atomic changed parts. As shown in Kim et al. [9], it is critical for those application to have precise bug-fix hunks information as well as bug-fix revisions to achieve precise and meaningful analysis.

Not every hunk in identified bug-fix revisions contributes to bug fixing. Some of them are related with refactoring and have nothing to do with bug fixing. Consider the following example which we took from the Eclipse project.

```
  110: int readOffset=8;
- 113: ... u2At(8) ...
+ 113: ... u2At(readOffset) ...
```

In this change, an integer constant `8` is replaced with a variable `readOffset` whose value is also 8. Therefore this change does not affect the semantics of the program. This hunk should not be regarded as a fix hunk even if it is in a bug-fix revision. However, previous techniques [10, 17] mainly focus on identifying bug-fix revisions while filtering out only few non-fix hunk patterns such as adding or removing comments/white spaces and simple format changes. Those patterns are easily recognized by syntactic analysis. However, they miss many non-fix patterns which require semantic analysis. For instance, they fail to recognize the above non-fix hunk example.

This paper addresses important questions, "which hunks are non-fix hunks?" and "how can we detect non-fix hunks?" We started by carefully observing hunks in bug-fix revisions of three open source projects. We identified about 8.3% of hunks are non-semantic changes thus they are non-fix hunks. We classified these non-fix hunks and found common patterns. We found that there are many patterns which require semantic analysis and therefore not recognizable by the previous approaches. Our final goal is to design a tool that automatically identifies common patterns of non-fix hunks we present in this paper. Hence, the results of this paper is the starting point of our research.

2. NON-FIX HUNKS

Usually developers commit multiple files at the same time. A file change contains a list of regions that differ in the two files; each region is called a hunk (H), as shown in Figure 1. As a result, a revision has multiple hunks usually in several files. In this section, we discuss which errors can occur in bug-fix identifications, why non-fix hunks are contained in a fix revision, and the manual inspection process for classifying non-fix hunk patterns.

2.1 Bug-fix Identification Errors

There are two potential errors in the fix identification process. First, it is possible to identify fix revisions incorrectly. For example, a change whose log includes "fix" or "bugs" does not necessarily a bug fix revision. Recovering links between changes and bug reports

[*] This work was supported by the Brain Korea 21 Project, School of Electrical Engineering and Computer Science, Seoul National University in 2009 and the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST)/Korea Science and Engineering Foundation(KOSEF). (R11-2008-007-01002-0)

DSMM'09, November 6, 2009, Hong Kong, China.
Copyright 2009 ACM 978-1-60558-810-0/09/11 ...$10.00.

```
         DOMPackage.java
H1  - 24: buffer...RATOR);
    + 24: buffer...RATOR).append(...SEPARATOR);

         CompilationUnit.java
H2  - 70: source=...RATOR;
    + 70: source=...RATOR+ Java...SEPARATOR;

         Buffer.java
H3  - 300:  i++;

H4  - 302:  lineStart = i + 1;
    + 301:  lineStart = i + 2;
```

Figure 1: The four hunks in one fix-change from Eclipse

are not perfect. It is possible the links are wrong or even if all links are correct, it is still possible linked bug reports are not about bugs. They can be feature enhancement. The other source of error is non-fix hunks mixed in bug-fix revisions. Suppose there is a real bug-fix revision. The revision consists of multiple hunks like the Figure 1. Not always all hunks in the revision are bug-fix hunks. In hunks H3 and H4, if variable i is no longer used after the line number 302 then there are no semantic changes on these hunks. Sometimes source code format changes and some other non-semantic changes are committed on bug-fix revision.

Our main concern is the latter case. Our long-term goal is to remove all non-fix hunks in the fix revision. Researchers propose techniques to increase the accuracy of fix revisions [4,5,11]. Whereas identifying non-fix hunks are neglected. We believe identifying non-fix hunks and removing them are important for other mining research which uses the bug-fix data to yield accurate and meaningful results.

2.2 Why do non-fix hunks break in?

Code developers often do different jobs (e.g. bug-fixing, refactoring, and development) at the same time. Bug-fix revisions committed by such developers are likely to contain other non-fix hunks. Suppose the following scenario. While a code developer is working on refactoring his code a certain bug-fix request arrives. The bug may be critical and a bottleneck to other developers. Then she must start fixing the bug before finishing the refactoring. As a result, when she commits the code after fixing the bug the remaining refactoring hunks would be mixed.

Distributed version control systems (DVCS) can mitigate the problem of mixed commits. Git, Darcs, Mercurial, and Bazaar belong to these systems. In these systems creating a branch and merging branches are lightweight: developers easily make their own branches according to their purposes. If they want to do refactoring then they can work on a new branch. If a certain bug-fix request arrived then they can make another branch for that bug-fix. This branch is totally independent of the previous branch created for refactoring. Hence they can commit the bug-fix hunks only. After then they come back to the refactoring branch and can keep working on the refactoring. However when two branches are merged or some series of commits are flattened to one commit, refactoring and bug-fix are likely to be merged [3]. Still there may exist mixed changes issue.

2.3 Manual Inspection

Previous technique [10,17] is used to identify fix revisions. Keywords in the change logs, links between changes and bug reports are used. Authors manually inspected each hunk in the first 50 fix revisions and determined if they are real fixes. All authors have multiple years of Java and other programming experience. We found 179 non-fix hunks from 2146 hunks. About 8.3% of identified hunks are non-fix hunks. It takes about 12 hours to inspect all these hunks for skillful developers. Even though these 50 revisions are just a tiny part among the whole software history. Hence if we make an automatic detector for non-fix hunks then it must be useful. In the next section, based on our findings, we discuss common non-fix change patterns.

3. NON-FIX HUNK PATTERNS

Based on the manual inspection, we identified common non-fix hunk patterns. Since our concern is identifying correct fix hunks, we identified all hunks as non-fix if they do not contribute any possible bug fixes. The numbers of each found pattern are shown on the Table 1.

We divide all the found patterns into two categories, syntactically detectable patterns and semantically detectable patterns. Naturally, syntactically detectable patterns entail semantically detectable patterns. But the opposite direction does not. Hence automatic identifying latter equivalence is more challenging than identifying the former equivalence. A parsing technique is sufficient to detect syntactically detectable patterns. While semantically detectable patterns require some static analysis techniques. For example in our observation the following techniques are required: class analysis [18], data flow analysis, constant propagation [1], and semantic equivalence checker.

3.1 Syntactically Detectable Non-fix Hunks

1. Unnecessary code addition/deletion (Eclipse #51, Lucene #3)
   It is possible that some unnecessary code segments exist in software such as dangling semicolons and braces. These unnecessary code structures can be removed or added during change processes. Braces for a single line block shown in Figure 2 is not necessary but are recommended to increase source code readability. Figure 3 shows a dangling semicolon change in Eclipse. Removing unnecessary semicolon does not affect the semantics of program. However, it is optional and developers may have various opinions about using these unnecessary structures. Changes that add or remove unnecessary structures are observed in our benchmarks. These changes never affect program behaviors. Thus these are not bug-fix hunks.

```
  77:   if (sEnd || pEnd)
+ 78:   {
  79:      break;
+ 80:   }
```

Figure 2: A hunk from Lucene in which unnecessary braces are added.

```
  215:  if (Character.isWhitespace( ...
  216:  {
- 218:   ;
+ 218:  }
```

Figure 3: A hunk from Eclipse in which a dangling semicolon is deleted.

	Eclipse	Columba	Lucene	Total
Unnecessary code addition/deletion	51	0	3	54
Import change	41	8	4	53
Class attributes change	2	0	0	2
Renaming	0	16	2	18
Method addition/deletion	21	1	9	31
Unnecessary `this`	1	0	1	2
Exchange between constants and variables	3	0	0	3
Initialization with declaration	1	1	0	2
Debugging printouts	4	0	1	5
Removing temporary variables	0	1	2	3
Deep semantics	0	0	6	6
Total	124	27	28	179

Table 1: Non-fix hunks from Eclipse, Columba, and Lucene

2. Import change (Eclipse #41, Columba #8, Lucene #4)
   A hunk caused by changing import package code is shown in Figure 4. Sometimes it is essential to change some imported packages to fix bugs. However just adding or deleting "import packages" itself does not affect the semantics of a program. Other hunks using this package can be bug-fix hunks. Import addition or deletion needs not to be captured as bug-fix hunks.

```
- 3:  import java.util.Hashtable;
+ 3:  import java.util.Collection;
```

Figure 4: Pattern import addition/deletion from Lucene

3. Class attributes change (Eclipse #2)
   Class attributes(e.g. interfaces and inheritances information on class) can change. For instance, a hunk caused by changing interface implementation is presented in Figure 5. A class implements an interface to provide the methods necessary to conform to the interface. Implementing means satisfying the existing interface contract by writing the prescribed method bodies. Hence this change is for restricting code behavior not for fixing bugs. This change itself can not change the behavior of the program, hence this is not a bug-fix.

```
- 7: public class ... implements ... IBinaryField {
+ 7: public class ... implements ... Comparable {
```

Figure 5: Pattern class attributes change from Eclipse

4. Renaming (Columba #16, Lucene #2)
   Packages, methods, and variables could be renamed to enhance the readability and maintenance of code. Even classes are able to be renamed. We found some renaming hunks in bug-fix revisions. In Figure 6, the method name `run` is renamed as `shutdown`. This alpha conversion is done consistently over the whole program. Alpha conversion does not affect the behaviors of the program.

```
-  3: public void run();
+  3: public void shutdown();
```

```
- 14: plugin.run();
+ 14: plugin.shutdown();
```

Figure 6: Pattern renaming methods from Columba

3.2 Semantically Detectable Non-fix Hunks

1. Method addition/deletion (Eclipse #21, Columba #1, Lucene #9)
   Code developers usually do refactoring code to maintain the simplicity and modularity of their code. Depending on refactoring strategies some methods could be added or deleted. In bug-fix revisions, adding or deleting some methods may be a crucial part for fixing bugs. However deleted methods can not be called. Moreover there exist certain methods that are added and never called on the same bug-fix revisions. These methods must not be related with the bug-fix.

   Virtual methods are frequently used in Java programs. So it is impossible to statically know the exact set of methods that are called. But class analysis enables us to find a super set of the exact set. With this information we soundly determine if newly added methods are called.

```
+ 39:   public void setStemmer( GermanStemmer ...
+ 40:      if (stemmer != null)
+ 41:      {
+ 42:         this.stemmer=stemmer;
+ 43:      }
+ 44:   }
```

Figure 7: Pattern method addition from Lucene

2. Unnecessary `this` (Eclipse #1, Lucene #1)
   In Java, keyword `this` is used in instance methods to refer to the object on which they are currently working. Some uses of `this` are unnecessary. In Figure 8, an unnecessary `this` is removed. Using "`this`.<name>" is necessary only if we use the member name <name> as a local variable. In this case <name> refers to the local variable. To point member variable we need to use "`this`" explicitly. If code developers know which object the methods are working on then developers may explicitly remove `this`. To find this pattern we need a class analysis to determine which class is pointed to by an instance.

```
- 17: this.exclusions=exclusiontable;
+ 17: exclusions=exclusiontable;
```

Figure 8: Pattern unnecessary this from Lucene

3. Exchange between constants and variables (Eclipse #3)
   Sometimes developers replace constant values with the corresponding constant variables to enhance the readability and maintainability of code. In Figure 9, constant 8 is replaced by variable `readOffset` which has the same value 8 and never changes in the loop. In order to find this pattern classic constant propagation analysis may be sufficient.

```
  110: int readOffset=8;
  111: boolean isSynthetic=false;
  112: for (int i=0; i < attributesCount; i++) {
- 113: ... constantPoolOffsets[u2At(8)] ...
+ 113: ... constantPoolOffsets[u2At(readOffset)] ...
```

Figure 9: Pattern Exchange between constants and variables from Eclipse

4. Initialization with declaration (Eclipse #1, Columba #1)
   It seems that some developers prefer separating initialization and declaration into two statements, initialization after declaration. While the developer that committed the code in Figure 10 does not. The declaration and initialization statements on variable `command` are merged. If there are some complex statements between declaration and initialization then any heuristic based detection would fail. To accurately recognize this pattern a symbolic executor is needed.

```
- 202: String command;
- 203: command = e.getActionCommand();
+ 174: String command = e,getActionCommand();
```

Figure 10: Pattern initialization with declaration from Columba

5. Debugging printouts (Eclipse #4, Columba #1)
   Some additional code like "`System.out.println`" can be inserted for the purpose of debugging. This code is not directly related with the behaviors of the program. When these debugging parts are not required anymore debugging message code can be removed. If there is a need to change place in which debugging message is logged then code can be changed 11. Changing debugging message printout code does not affect on the functional behaviors of the program. Besides a symbolic executor collecting library functions that are used to log some messages is required.

```
- 169: System.out.println("Messagebox header...
+ 171: ColumbaLogger.log.info("Messagebox header...
```

Figure 11: Pattern debugging printouts from Columba

```
-  9: URL url=DiskIO.getResourceURL(fileName);
- 10: setFile(new File(url.getFile()));
+  8: setURL(DiskIO.getResourceURL(fileName))
```

Figure 12: Pattern removing temporal variables from Columba

6. Removing temporary variables (Columba #1, Lucene #2)
   In Figure 12 the temporarily used variable url is removed. According to Apache Common Library, both tt setFile and `setURL` functions set the location of configuration. Function `setFile` takes `fileName` as an argument, function `setURL` takes `URL` as an argument. Since `fileName` could be `URL`, function `setURL` can replace function `setFile`.

   From the symbolic execution of the method that contains this hunk, symbolic memory state at the end of this method is acquired. Among the addresses in the state non-reachable from all visible environment (e.g. global variables, class members, ...) does not affect the semantics of program. So these temporary addresses are removable. If two memory states are same after removing all temporary addresses then we can say there are no semantic changes.

7. Deep semantics (Columba #6)
   Automatically detecting this pattern is decidedly challenging. This pattern claims understanding deep semantics of Java programs as well as semantic equivalence checker. Finding this pattern automatically itself can be a fabulous contribution to static analysis group.

```
- 54: return new Boolean(value).booleanValue();
+ 54: return Boolean.valueOf(value).booleanValue();
```

Figure 13: Pattern deep semantics using static boolean from Columba

   In the Figure 13 both of `new Boolean(value)` and `Boolean.valueOf(value)` return a `Boolean` class instance whose boolean value is of `value`. Therefore this change does not affect the semantics of program and this is a non-fix hunk. On the deleted code in Figure 14, `JPanel` is constructed with default constructor and the layout is set later right next line. In the added code, `JPanel` is constructed with a constructor taking layout as an argument. Therefore this change occurs no difference in the semantics.

```
- 78: JPanel list1Panel=new JPanel();
- 79: list1Panel.setLayout(new BorderLayout());
- 80: JPanel list1TopPanel=new JPanel();
- 81: list1TopPanel.setLayout(new BorderLayout());
+ 57: JPanel list1Panel=new JPanel(new BorderLayout());
+ 58: JPanel list1TopPanel=new JPanel(new BorderLayout());
```

Figure 14: Pattern deep semantics from Columba

4. RELATED WORK

Kim et al. identify some non-fix hunks in fix revisions [9] but their approach is limited to a few patterns. In finding fix changes, they ignore some non-fix patterns such as comments, blank lines, and format changes and show that ignoring them is important to identify bug-introducing changes. However, they cannot handle some non-fix patterns that are prevalent in Java. In addition, it is unclear whether their technique can be applicable to a general setting, since they do not mention their non-fix pattern identification algorithm. Williams et al. uses a Java syntax-aware tool to identify non-fix hunks [19] , but it is also limited to a small subset of real non-fix patterns. They use a Java syntax-aware diff tool, DiffJ, to identify non-fix hunks in fix revisions. The tool helps them identify more formatting changes than those found by Kim et al. [9]. However, this approach is not accurate enough to cope with some common non-fix patterns such as addition/deletion of temporary variables presented in this paper. Moreover, their approach has a possibility to mistakenly classify some fixes into non-fix hunks, because they depend on just change types, reported by the DiffJ tool, for classification.

Many tools can identify semantics-preserving transformations [2, 7, 8, 12, 14], but always with limitations for our purposes. Semantic-diff [8] analyzes two versions of programs and identifies changes with respect to dependencies between variables. Though it can be used to find non-fix hunks that do not affect variable dependencies, it cannot identify non-fix hunks that modify the dependencies such as constant propagation or variable deletion/addition, which are found to be common patterns in Java. Horwitz's technique [7] is powerful but not suitable for Java programs, since it targets on a subset of C. Tools that compare two dimensional structures such as syntax trees [6, 12] or flow graphs [2, 14] also have unacceptable limitations: they often miss non-fix hunks that modify the structures that they use. All these tools use some kind of (lightweight) static analysis to identify changes that actually affect program behaviors. Because there always exists at least one program that fools their static analyses, they miss some non-fix hunks or accept fix as non-fix. Hence, we need a tool that is designed to be complete in the sense that we never classifies fix hunks as non-fix. This property is particularly important in mining fix changes: we do not want to miss any fix hunks in our databases. This is our direction.

5. FUTURE WORK

Based on classified patterns authors are developing a Java symbolic executor to detect non-fix hunks. As a first step we implemented Jimple [15] OCaml parser. So our symbolic executor can handle all Java programs that can be translated into Jimple. First syntactically detectable patterns will be implemented. After then we will enhance our checker's ability to find semantically detectable patterns. Finding complicated non-fix hunk patterns (e.g. removing temporary variables and deep semantics) consequently requires checking semantic equivalences between two different programs. If our tool is tuned enough to find such patterns then the tool can be used as a general equivalence checker. General semantic equivalences checker can be useful to a wide spectrum of programming language and software engineering areas. For instance 1) commutativity analysis [16] requires a semantic equivalence checker as a crucial part; 2) correctness of translated programs can be proven; 3) maliciously copied programs look different can be detected.

ACKNOWLEDGMENT We thank Sunghun Kim for his contributions to give us bug-fix revision, their hunks, and many useful comments. We also thank Soonho Kong for his precious comments.

6. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 2–13, 2004.

[3] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hmilton, D. M. German, and P. Devanbu. The promises and perils of mining git. In *MSR 2009: 6th IEEE Working Conference on Mining Software Repositories*, May 2009.

[4] D. Cubranic and G. C. Murphy. Hipikat: Recommending pertinent software development artifacts. page 408, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[5] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 23, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[6] B. Fluri, M. Wuersch, M. PInzger, and H. Gall. Change distilling:tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007.

[7] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 234–245, 1990.

[8] D. Jackson and D. A. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *Proceedings of the International Conference on Software Maintenance*, pages 243–252, 1994.

[9] S. Kim, T. Zimmermann, K. Pan, and E. J. W. Jr. Automatic identification of bug introducing changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 81–90, September 2006.

[10] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. Predicting faults from cached history. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 489–498, Washington, DC, USA, 2007. IEEE Computer Society.

[11] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *ICSM '00: Proceedings of the International Conference on Software Maintenance*, page 120, Washington, DC, USA, 2000. IEEE Computer Society.

[12] I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, 2005.

[13] K. Pan, S. Kim, and E. J. Whitehead, Jr. Toward an understanding of bug fix patterns. *Empirical Softw. Engg.*, 14(3):286–315, 2009.

[14] 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 *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 188–197, Washington, DC, USA, 2004.

[15] R. V. Rai and L. J. Hendren. Jimple: Simplifying java bytecode for analyses and transformations. Technical report, Sable Research Group, McGill University, Montreal, Quebec, Canada, 1998.

[16] M. C. Rinard and P. C. Diniz. Commutativity analysis: A new analysis framework for parallelizing compilers. In *In Proceedings of the SIGPLAN '96 Conference on Program Language Design and Implementation*, pages 54–67. ACM, 1996.

[17] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM.

[18] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the Conference on Programming Language Design and Implementation*. ACM Press, June 2004.

[19] C. Williams and J. Spacco. Szz revisited: verifying when changes induce fixes. In *Proceedings of the 2008 workshop on Defects in large software systems*, pages 32–36, July 2008.