Detecting and Investigating the Source Code Changes using Logical rules

E.Kodhai
Associate Professor,
Dept. of Information and Technology,
Sri Manakula Vinayagar Engineering College,
Puducherry, India.
kodhaiej@yahoo.co.in

B.Dhivya
PG Student,
Dept. of Computer Science,
Sri Manakula Vinayagar Engineering College,
Puducherry, India.
dhivyamar@gmail.com

Abstract— **Software developers often need to examine program differences between two versions and reason about the changes. Analyzing the changes is the task. To facilitate the programmers to represent the high level source code changes, this proposed system introduces the rule-based program differencing approach to represent the changes as logical rules. This approach is instantiated with three levels: first level describes the changes in method header names and signature; second level captures change in the code level and structural dependences; and third level identifies the same set of function with different name. This approach concisely represents the systematic changes and helps the software engineers to recognize the program differences. This approach can be applied in open source project to examine the difference among program version**

Keywords— **source code changes, logical rules, open source**

I. INTRODUCTION

Software evolution plays an ever-increasing role in software development. Programmers constantly update existing software to provide new features to customers or to fix defects. During evolution, developers often need to inspect changes in the source code of the software. When inspecting program differences, programmers or researchers may ask the following kinds of high-level questions about code changes: "What data is being modified in this code?"; "What is changed in the existing software?" "Is anything missing in the change?" and "Why did the set of code fragments change together?"[1].

To enable programmer to recognize the source code level changes and help the developer to answer these kind of question, this rule based program differencing approach is introduced.

The uniqueness of this approach is applied in four existing approach. The first approach records what all are the edit operation made by the developer in the editor or Integrated Development Environment (IDE)[15]. Some editors or IDE capture and replay the key strokes, editing operations, and high-level update commands. With the help of recorded information difference among program version is identified. A key limitation of the first approach is that it made the programmers depend into a specific editor or an environment, which is not an acceptable transaction.

The second approach is a programming language-based approach [16]. In this approach, programmer specifies the high level changes using the syntax of programming languages and automatically update the program with the help of specified change script. Software changes are described in high level, this approach having high adoption cost and programmer need to plan software changes in advance and write the syntax for that changes.

The third approach use check in comment or change log that programmers manually write the changes in natural language. This approach is easy to understand but natural language descriptions are incomplete and may not write the actual code changes faithfully.

The fourth approach automatically identifies the program difference between two versions. This approach can be applied in open source project to achieve the version control. The drawback of existing program differencing approach makes the programmer difficult to reason about software changes at a high level.

For example, the ubiquitous program differencing tool diff computes differences per file, programmer read the changed line file by file, even when those cross-file changes were done systematically.

To facilitate the programmers to represent the high level source code changes, this article introduces the novel approach that extracts the program differences and represent them as change rules with the help of change rule inference algorithm.

The rest of this paper is organized as follows section 2 describe about the Related work section 3 describe about the Proposed work and rule based differencing section 4 describe the algorithm and technique to identify section 5 describe the conclusion and future work.

II. RELATED WORK

The uniqueness of the rule based approach is best seen in the context of existing approaches that can be used to reason software changes.

TABLE I
COMPARISION OF CODE MATCHING TECHNIQUE

Program representation	Citation	Granularity	Assumed correspondence	Application	Strength and weakness
String	Diff[2]	Line	File	Merging Clone genealogy And Fix inducing code	-Sensitive to File name changes
	Bdiff[6]	Line	File	Merging	+Can trace copied block
AST	Cdiff[7]	AST Node	Procedure	Merging	-Sensitive to nested level changes -Require procedure level mapping
	Neamtiu et.al[8]	Type and variable	-	Type change	-Partial AST Matching
	Hunt,Tichy [9]	Token	File	Merging	-require procedure level mapping +can identify procedure renaming
CFG	JDiff[10]	CFG Node	-	Regression Testing and Impact Analysis	+robust to signature changes -sensitive to control structure changes

A. Program Differencing and Refactoring Reconstruction

The existing program differencing techniques use similarities in names and structure to match code elements at a particular level of granularity such as (1) lines and tokens[2], (2) abstract syntax tree nodes[3], (3) control flow graph nodes[4], (4) program dependence graph nodes[5], etc. For example, the ubiquitous tool diff computes line-level differences per file using the longest common subsequence algorithm [2]. As another example, JDiff computes CFG-node level matches between two program versions based on similarity in node labels and nested hammock structures [4]. 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. The Comparison of code matching technique is shown in Table 1.

B. 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. For example, iXj enables developers to easily perform systematic code transformations by providing a visual language and a tool. Coccinelle lets developers apply systematic updates to Linux device drivers.

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.

C. Identification of Related Changes

Several approaches use change history to identify code elements that tend to change together. 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. 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.

III.PROPOSED WORK

The rule based approach focuses on helping programmers reason about software changes as opposed to reconstructing a new program version given the old version of the program. In the rule based approach, there are two similar but separate change-rule inference techniques, each of which captures a different kind of change (Figure 1).

The first kind of change-rules capture changes to API names and signatures. This change rule represents a group of similar transformations explicitly in a rule based representation at the level of a method-header (i.e., API level). API change-rules identify the program changes only at the method-headers. It should not recognize the changes in the coding level.

The second kind of change-rules capture changes to code elements and structural dependencies. Logical Structural Diff (LSDiff) inference algorithm is used in the second level. LSDiff assists developers in investigating the code level program differences by discovering and summarizing systematic structural differences as change-rules.

First level of 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 renamings such as rename executeUpdate to performUpdate. To overcome this problem, Synonyms matching technique is used in the API change-rule.

Synonyms matching technique is applied in the API Change rule level. This technique identify the two methods across the two versions are similar, they are in the same class yet they differ in name. Refactoring Crawler tool is employed to identify these kinds of changes. The idea 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 existing techniques do not focus on discovering a high-level structure among program changes.

A. Rule – Based Program Differencing Approach

Logical rules are identified from the changes between two program versions. The format of the logical rule is represented as follows

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

The format consists of three main entities: scope, exceptions and transformation. 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 relevant 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. This rule based approach is instantiated with two levels: first level describes the changes in method header names and signature; and second level captures change in the code level and structural dependences.

1) API-Change Rule

The scope of the API-Change Rule is method header. It identifies the changes among the following tuple: (package: String, class: String, procedure: String, input_argument_list:[String], return_type:String). To represent the scope of a change rule, wild card pattern-matching operator is used. It summarizes a group of similarly named method-headers. For example, *.*Draw. get*Input() identifies methods with any class name, and package name that ends with Draw , any method that begin with get and end with Input, 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. A scope can have disjunctive scope expressions. To represent the transformation of a change rule, nine types of transformations are defined. It describes about the changes to the method- header names and signature (as shown in Table 2).

For example, ‘for all x:graph.*Graph.*(*), packageRename(x, graph, graph.plot)’ means that all classes with the name *Graph were moved from package graph to package graph.plot, while ‘for all x:graph.*.draw(*), methodRename(x, draw, render)’ means that all draw methods in package graph were renamed to render.

```
For all x:method-header in graph.*Plot.*(*)
packageRename(x,graph,graph.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 procedure that should not satisfied by any rules are either deleted or added methods.

2) Logical Structural Diff (LSDiff) Rules

LSDiff discovers a latent structure in low-level changes. It abstracts a program as code elements (packages, types, methods, and fields) and their structural dependencies (method-calls, field-accesses, sub typing, overriding, and containment) based on two premises. First, programmers often look for structural information when inspecting program differences: which code elements changed and how their structural dependencies are affected by the change. Second, code elements that changes similarly together often share common structural characteristics such as using the same field or implementing the same interface. Finding such shared characteristics in changed code is useful for discovering systematic changes.

Logical Structural Dif (LSdiff), identifies the changes in the code elements and their structural dependences and modeled using the thirteen logic predicates as shown in (Table 2).

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. 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. Changerules 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.

```
For all m, past_method(m,”draw”,t) ^ past_extends(t,”Plot”)
=>deleted_accesses(m,”Shape.dotted”)
```

IV.RULE INFERRING ALGORITHM

API Change Rule and LSDiff Change Rule algorithm are used to identify the header and code level changes between two program versions.

TABLE II
THE FORMAT OF CHANGE RULE

API Rule	
Abstraction	method-header
Scope	a subset of method-headers
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
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)

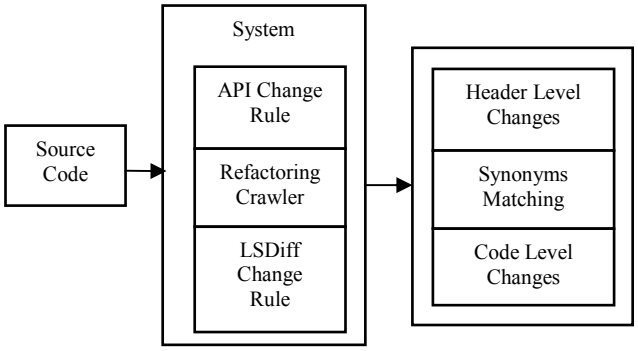

Fig 1. Block Diagram of Proposed Work

In the proposed system, two version of source code is fed as input to the system. System identifies the program difference between two versions. In the system, API Change rule identify the header level changes and Refactoring Crawler identify the same set of function with different name and LSDiff change rule recognize the code level changes.

A. API Change Rule Algorithm

API Change rule algorithm[11] identifies the header level changes. This algorithm first discovers the seed level matches. Based on the seed it generates the candidate rule then it iteratively select the best candidate rule (Figure 2).

1) Identification of Seed Level Matches

Seed matches are recognising based on the textual similarity of header level matches. Method headers are extracted from the two version of program with the help of longest common subsequence algorithm[2]. For example, P1 and P2 are the two program version; O and N are the extracted method headers from p1 and p2 respectively.

2) Generate Candidate Rule

For every seed matches, Candidate rules are generated. A candidate rule allows one or more transformation of seed matches.

B. Synonyms matching

API Change rule depend on the textual similarity to identify the changes between two versions. Some time it may misinterpret the changes [13]. For example, we use the method header name as performUpdate that should be changed as executeUpdate in the subsequent version means meaning of both the method is same.

API Change rule represent both are different method. This Limitation should be overcome with Synonyms matching technique. Synonyms matching technique is applied in the API Change rule level. It identify the two methods across the two versions are similar, but it differ in name. Refactoring Crawler tool is employed to identify these kinds of changes.

```
Input: S, /* a set of seed matches */
        ∈ , /* an exception threshold */
        D, /* domain: extractMethodHeaders(P1) −
extractMethodHeaders(P2) */
   C /* codomain: extractMethodHeaders(P2) */
  Output: R, /* a set of selected rules */
  M /* a set of found matches */

   R :=" Ø ", M :=" Ø ", CR :=" Ø ";
  /* Create an initial set of rules */
   foreach seed∈ S do
      2^T := extractTransformations (seed);
      foreach trans ∈2^T do
          scope:= findTheMostGeneralScope (seed.left,
           trans);
          rule:= createNewRule (scope, trans);
          CR := CR U {rule};
      end
  end
  cont := true;
  while cont do
     n := |M |;
          /* select the best rule */
     N := 0, s := null;
     foreach r_k ∈ CR do
        if (numRemainingPositive (r_k) > N)^
          (isValid(r_k,D,C,M, ∈))then
             N = numRemainingPositive (r_k);
             s = r_k;
        end
     end
/* If an invalid rule rk in CR can find
more than N matches, expand its
children rules. */
     toBeRemoved := Ø ; toBeAdded := Ø;
      foreach r_k ∈ CR do
       if numRemainingPositive (r_k)=0 then
            toBeRemoved := toBeRemoved U {r_k};
       end
       else if (numRemainingPositive (r_k) > N) ^
            (isValid(r_k,D,C,M, ∈))=false then
           toBeRemoved :=toBeRemoved U {r_k};
              children = createChildrenRules (r_k,N);
           foreach c∈ children do
              if (isValid (c, D, C, M, ∈))^
                   (numRemainingPositive(c)>N)then
                   N := numRemainingPositive (c); s := c;
              end
           end
          toBeAdded := toBeAdded U children;
        end
      end
/* Add toBeAdded to CR and remove
toBeRemoved from CR. */
    CR := CR U toBeAdded;
    CR := CR − toBeRemoved;
     R := R U {s };
    CR := CR − {s };
    M := M U s.positive;
     if (|M|=n) then
       cont := false;
     end
end
```

Fig 2. API change rule algorithm

```
Input: FBo, /* a fact-base of an old program version */
  ΔFB, /* fact-level differences between FBo and FBn */
   m, /* the minimum number of facts must match */
   a, /* the minimum accuracy of a rule */
   k, /* the maximum number of literals in a rule's antecedent
   β /* beam search window size */
Output: L /* a set of valid learned rules */
 R :=Ø , L :=Ø , U := ΔFB;
 U := reduceDefaultWinnowingRules (ΔFB, FBo) ;
 foreach i = 0 . . . k do
    if (i = 0) then
      R := Ø ;
      foreach p ∈DELTA PREDICATES do
          l:= createLiteral(p, freshvariables());
          r:= new Rule();
          r.setConsequent(l);
          if |r.matches| >=m then
              R := R U {r};
          end
       end
   end
   else
     NR :=Ø ;
     foreach r ∈ R do
        foreach p ∈ANTECEDENT_PREDICATES do
         bindings := enumerateBindingsForPredicate (r, p)
             foreach b ∈ bindings do
                r:= new Rule(r);
                r.addAntecedentLiteral(l);
                 if |r.matches| >= m^!(r ∈ NR) then
                  NR := NR U {r};
                end
            end
          end
         R := NR;
      end
     end
     foreach r ∈R do
       NR:=Ø ;
       S = new Stack();
       S.push(r);
       while !S.isEmpty() do
          pr = S.pop();
          foreach variable ∈ pr.remainingVariables() do
            constants := getReplacementConstants(pr, variable);
            foreach constant ∈ constants do
              n = substitute(pr, variable, constant) if
                   |n.matches| >= m ^accuracy(n) >= a then
              NR := NR U {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 U{g};
       U :=U − {g.matches};
     end
   end
 end
R :=selectRules (R, β);
end
```

Fig 3. LSdiff change rule algorithm

Refactoring Crawler [14] is an analysis tool that detects synonym that happened between two program versions. The strength of the tool lies is the combination of syntactic analysis and semantic analysis to recognize the changes. Thus, the syntactic analysis identifies those bodies of source text that are similar.

Syntactic analysis use **Shingles encoding** technique to find similar pairs of entities in the two program versions. Shingles identify the string even if the string changes slightly. The semantic analysis builds **reference graphs** for the two program version. If two method are called from the same place in two version means having same method bodies however they are differ in names Refactoring Crawler identifies the method is renamed in the successive versions.

C. LSDiff Change rule algorithm

LSDiff change rule algorithm [12] identifies the second level changes. This algorithm first identifies the structural difference among two version of code (Figure 3). Based on the structural difference, it generates the candidate rule.

1) Structural Difference Identification

Structural difference are identify among two program version P1 and P2.Those differences are represented as FB1 and FB2 (Fact Base of program 1 and program 2 respectively).Then identify the difference among the fact base with the help of fact extraction tool

2) Generate rule from Fact Base

LSDiff algorithm represents the code level changes. This step takes the three fact-bases and outputs inferred rules and remaining unmatched facts.

V CONCLUSION AND FUTURE WORK

To help programmers reason about high level software changes, this approach introduced a program differencing approach that automatically discover the change rule among two program versions. This approach is applied in three levels: first at a method-header level and second level at the level of code elements and their structural dependencies and third level identify same set of function with different name.

We propose the future work as change rule are extended to exception thrown by Java and enhance the LSDiff to identify the cross-language systematic changes such as changing a Java program and subsequently changing XML configuration files.

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. 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.

[3] B. Fluri, M. W ursch, 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

[4] 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.

[5] 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.

[6] Walter F. Tichy. The string-to-string correction problem with block moves. ACM Transactions on Computer Systems, 2 (4):309–321, 1984.

[7] Wuu Yang. Identifying syntactic differences between two programs. Software – Practice& Experience, 21(7):739–755, 1991.

[8] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. In MSR'05, pages 2–6, 2005.

[9] James J. Hunt and Walter F. Tichy. Extensible language-aware merging. In ICSM'02: Proceedings of the International Conference on Software Maintenance (ICSM'02),page 511, Washington, DC, USA, 2002. IEEE Computer Society.

[10] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. A differencing algorithm for object-oriented programs. In ASE '04: Proceedings of the 19th IEEE International Conference on Automated Software Engineering, pages 2–13, Washington, DC, USA, 2004. IEEE Computer Society.

[11] 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 SoftwareEngineering*. Washington, DC, USA: IEEE Computer Society, 2007,pp. 333–343.

[12] 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.

[13] 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.

[14] Danny Dig and Ralph Johnson. Automated detection of refactorings in evolving components. In ECOOP '06: Proceedings of European Conference on Object-Oriented Programming, pages 404–428. Springer, 2006.

[15] Romain Robbes. Mining a change-based software repository. In MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories, page 15, Washington, DC, USA, 2007. IEEE Computer Society.

[16] Marat Boshernitsan, Susan L. Graham, and Marti 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, pages 567–576, New York, NY, USA, 2007. ACM.