A Differencing Algorithm for Object-Oriented Programs

Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold
Georgia Institute of Technology
Atlanta, Georgia
{term|orso|harrold}@cc.gatech.edu

ABSTRACT

During software evolution, information about changes between different versions of a program is useful for a number of software engineering tasks. For example, in regression testing, knowing which parts of a program are unchanged can help identifying test cases that need not be rerun. For many of these tasks, a purely syntactic differencing may not provide enough information for the task to be performed effectively. This problem is especially relevant in the case of object-oriented software, for which a syntactic change can have subtle and unforeseen effects. In this paper, we present a technique for comparing object-oriented programs that identifies both differences and correspondences between two versions of a program. The technique is based on a representation that handles object-oriented features and, thus, can capture the behavior of object-oriented programs. We also present JDiff, a tool that implements the technique for Java programs, and empirical results that show the efficiency and effectiveness of the technique on a real program.

1. INTRODUCTION

Software maintenance tasks often involve analyses of two versions of a program: an *original* version and a *modified* version. Many program-analysis tools that automate these tasks require the knowledge of the locations of changes between the two program versions. In addition, some tools also need the mapping of entities, such as statements, basic blocks, and methods, from the original version to their counterparts in the modified version. Differencing algorithms can provide information about the locations of changes and can classify program entities as added, deleted, modified, or unchanged.

The results of these differencing algorithms are therefore useful for many software-maintenance tasks. For example, for program-profile estimation,[1] the differencing results, along with the coverage or profile information for the original version of the program, are used to estimate the coverage or profile information for the modified version. This approach eliminates the cost of rerunning the test suite on the modified version of the program to measure coverage or to obtain profiles. The approach also facilitates the estimation of coverage or profile information in cases in which this information cannot be reproduced (e.g., coverage/profiles from deployed software). For another example, the change information that is produced by differencing is also useful for impact analysis and regression testing. Impact analysis identifies the parts of a program that are affected by changes and, thus, requires knowledge of the location of such changes. Many regression test selection techniques (e.g., [3, 10]) use change information to select test cases to be rerun on modified versions of the software.

There are a number of existing techniques for computing differences between two versions of a program. However, these techniques are limited in their ability to detect differences in object-oriented programs. The UNIX `diff` utility [7] identifies textual differences between two files, and reports them in terms of changed lines. Therefore, it provides purely syntactic differences and does not consider changes in program behavior indirectly caused by syntactic modifications. Consider, for example, the two partial Java programs in Figure 1, the original program P and the modified version P'.

If we would inspect the output of `diff`, run on P and P', we would see that method $B.m1$ has been added in P' and that the exception-type hierarchy has changed. However, without additional analyses, it would not be straightforward to detect that, in P and P', the call to $a.m1$ in $D.m3$ can be bound to different methods, and the exception thrown in $D.m3$ can be caught by a different catch block.

There are other differencing techniques that are targeted at comparing versions of a program (e.g., [4, 5, 12]). However, these techniques are not suitable for object-oriented code. BMAT [12] cannot recognize, for example, differences caused by changes in the exception hierarchy, and would overlook that the exception thrown in $D.m3$ can be caught by different catch blocks in P and P'. Semantic diff [5] is defined to work only at the procedure level and cannot be straightforwardly extended to work on entire object-oriented programs. Horwitz's technique [4] is also not suitable for object-oriented programs, in that it is defined only for a simplified C-like language.

To overcome problems with existing approaches, and provide the differencing information required for tasks such as program-profile estimation, impact analysis, and regression testing, we defined a new graph representation and a differencing algorithm that uses the representation to identify and classify changes at the statement level between two versions of a program. Our representation augments a traditional control-flow graph (CFG)[2] to model behaviors in the program due to object-oriented features. Using this graph, we

[1] Program-profile estimation is also referred to as *stale profile propagation* in Reference [12].

[2] A *control-flow graph* is a directed graph in which nodes represent statements and edges (possibly labeled) represent flow of control between statements. A CFG has a unique entry node that has no predecessors and a unique exit node that has no successors.

```
         Program P

public class A {
      void m1() {...}
}

public class B extends A {

      void m2() {...}
}

public class E1 extends Exception {}
public class E2 extends E1 {}
public class E3 extends E2 {}

public class D {
      void m3(A a) {
            a.m1();
            try {
               throw new E3();
            }
            catch (E2 e) {...}
            catch (E1 e) {...}
      }
}
```

```
         Program P'

public class A {
      void m1() {...}
}

public class B extends A {
      void m1() {...}
      void m2() {...}
}

public class E1 extends Exception {}
public class E2 extends E1 {}
public class E3 extends E1 {}

public class D {
      void m3(A a) {
            a.m1()
            try {
                  throw new E3();
            }
            catch(E2 e) {...}
            catch(E1 e) {...}
      }
}
```

Figure 1: Partial code for an original program (P) and the corresponding modified version (P').

identify changes in those behaviors and relate them to the point of the code where the different behavior occurs.

Our algorithm extends an existing differencing algorithm [6] and consists of five steps. First, it matches classes, interfaces, and methods in the two versions. Second, it builds enhanced CFGs for all matched methods in the original and modified versions of the program. Third, it reduces all graphs to a series of nodes and single-entry, single-exit subgraphs called hammocks. Fourth, it compares, for each method in the original version and the corresponding method in the modified version, the reduced graphs, to identify corresponding hammocks. Finally, it recursively expands and compares the corresponding hammocks.

The main contributions of the paper are

- A new graph representation that models the behavior of object-oriented programs.
- A differencing algorithm that works on the graph representations and uses different heuristics to increase the precision of the results.
- A set of empirical studies that shows the efficiency and precision of an implementation of our algorithm.

2. DIFFERENCING ALGORITHM

In this section, first we give an overview of the algorithm and discuss its input and output. Then, we discuss in detail each of the levels at which the algorithm compares the original and modified versions of the program. Finally, we discuss the complexity of the algorithm.

2.1 Overview

Our algorithm, `CalcDiff`, given in Figure 2, takes as input an original version of a program (P) and a modified version of that program (P'). The algorithm also inputs two parameters that are used in the node-level matching. The first parameter (LH) is the maximum lookahead that `CalcDiff` uses when attempting to match nodes in methods. The second parameter (S) is used when determining the similarity of two hammocks. At completion, the algorithm outputs a set of pairs (N) in which the first element is a pair of nodes and the second element is the status—either "modified" or "unchanged." The algorithm also returns sets of pairs of matching classes, interfaces, and methods in P and P'.

`CalcDiff` performs its comparison first at class and interface levels, then at the method level, and finally at the node level. The algorithm first compares each class in P with the like-named class in P', and each interface in P with the like-named interface in P', and produces sets of class pairs (C) and interface pairs (I), respectively. For each pair of classes and interfaces, `CalcDiff` then matches methods in the class or interface in P with methods having the same signature in the class or interface in P'; the result is a set of method pairs (M). Finally, for each pair of concrete (i.e., not abstract) methods in M, the algorithm constructs enhanced CFGs (hereafter, ECFGs) for the two methods and match nodes in the two ECFGs.

The next sections give details of `CalcDiff`. Throughout the discussion, we use the code in Figure 1 as an example.

2.2 Class and Interface Levels

`CalcDiff` begins its comparison at the class and interface levels (lines 1–2). The algorithm matches classes (resp., interfaces) that have the same fully-qualified name; the fully-qualified name consists of the package name followed by the class or interface names. Matching classes (resp., interfaces) in P and P' are added to C (resp., I). Classes in P that do not appear in set C are deleted classes, whereas classes in P' that do not appear in set C are added classes. Analogous considerations hold for interfaces. In the example programs in Figure 1, each class in P has a match in P', and, thus, there is a pair in C for each class in P.

2.3 Method Level

After matching classes and interfaces, `CalcDiff` compares, for each pair of matched classes or interfaces, their methods (lines 3-4). The algorithm first matches each method in a class or interface with the method with the same signature in another class or interface. Then, if there are unmatched methods, the algorithm looks for a match based on the name only. This matching accounts for cases in which parameters are added to (or removed from) an existing method, which we found to occur in practice, and allows for increasing the number of matches at the node level. Pairs of methods in P

```
Algorithm CalcDiff
Input: original program $P$
       modified program $P'$
       maximum look ahead $LH$
       hammock similarity threshold $S$
Output: set of (class,class) $C$
       set of (interface,interface) $I$
       set of (method,method) $M$
       set of ((node,node),status) $N$
Declare: Node $n$, $n'$

Begin: CalcDiff
 1: compare classes in $P$ and $P'$; add matched class pairs to $C$
 2: compare interfaces in $P$ and $P'$; add matched interface pairs
    to $I$
 3: for each pair (c,c') in $C$ or $I$ do
 4:    compare methods; add matched method pairs to $M$
 5:    for each pair (m,m') in $M$ do
 6:       create ECFGs $G$ and $G'$ for methods $m$ and $m'$
 7:       identify, collapse hammocks in $G$ until one node $n$ left
 8:       identify, collapse hammocks in $G'$ until one node $n'$ left
 9:       $N = N \cup$ HmMatch($n$,$n'$,$LH$,$S$)
10:    end for
11: end for
12: return $C$, $I$, $M$, $N$
end CalcDiff
```

Figure 2: Algorithm `CalcDiff`

and P' that match are added to M. For example, method $m1$ in class A in P matches $m1$ in class A in P'. Thus, there is a pair in M for each method in P. Conversely, method $B.m1$ is added because it has no match in P.

2.4 Node Level

`CalcDiff` uses the sets of matched method pairs (M) to perform matching at the node level. First, the algorithm considers each pair of matched methods $< m, m' >$ in M, and builds ECFGs G and G' for m and m' (line 6). Then, the algorithm identifies all hammocks in G and G', and collapses G and G' to one node (lines 7-8); we call these nodes n and n', respectively. Next, `CalcDiff` calls procedure `HmMatch`, passing n, n', LH, and S as parameters. `HmMatch` identifies differences and correspondences between nodes in G and G' (line 9), and creates and returns N, the set of matched nodes and corresponding labels ("modified" or "unchanged"). Finally, `CalcDiff` returns N, C, I, and M.

In the next section, we discuss the ECFG, the representation we use to perform node matching. Then, we discuss hammocks and how we process them. Finally, we present and explain our hammock-matching algorithm, `HmMatch`.

2.4.1 Enhanced Control-Flow Graphs

When comparing two methods m and m', the goal of our algorithm is to find, for each statement in m, a matching (or corresponding) statement in m', based on the method structure. Thus, the algorithm requires a modeling of the two methods that (1) explicitly represents their structure, and (2) contains sufficient information to identify differences and similarities between them. Although CFGs can be used to represent the control structure of methods, traditional CFGs do not suitably model many object-oriented constructs. To suitably represent object-oriented constructs, and model their behavior, we define the ECFG. ECFGs extend traditional CFGs and are tailored to represent object-oriented programs. In the following, we illustrate how the ECFG represents some important Java features.

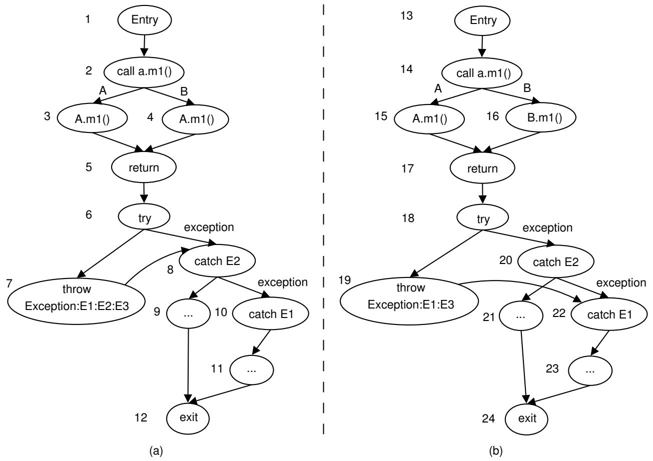

Figure 3: ECFGs for $D.m3$ in P and P' (Figure 1).

Dynamic Binding

Because of dynamic binding, an apparently harmless modification of a program may affect call statements in a different part of the program with respect to the change point. For example, class-hierarchy changes may affect calls to methods in any classes in the hierarchy, and adding a method to a class may affect calls to the methods with the same signature in its superclasses and subclasses.

We illustrate how we model a call site, in which a method m is called on an object o, to capture these modifications. First, we create a *call* and a *return* node. Then, for each dynamic type T that can be associated with o, we create a callee node. A *callee* node represents the method that is bound to the call when the type of o is T, and is labeled with the signature of that method. We also create (1) a *call edge* from the call node to each callee node, labeled with the type that causes such a binding, and (2) a *return edge* from each callee node to the return node. Note that if the call is static (i.e., not virtual), there is only one callee node.

To illustrate, consider method $D.m3$ in P (Figure 1). The ECFG for $D.m3$ is shown in Figure 3(a); it contains two callee nodes (nodes 3 and 4) because a's dynamic type can be either A or B. Both added nodes correspond to the same method, and thus have the same label, because B does not override method $m1$.

Consider now one of the two differences between P and P' in Figure 1: the redefinition of method $m1$ in B. Such a change causes a possibly different behavior in P and P' for the call to $a.m1$ in method $D.m3$: if the dynamic type of a is B, the call results in an invocation of method $A.m1$ in P, and in an invocation of method $B.m1$ in P'.

Figure 3(b) shows how the different binding, and the possibly different behavior, is reflected in the ECFG for method $D.m3$: the call edge labeled B from the call node for $a.m1$ (i.e., the call edge representing the binding when a's type is B) is now connected to a new callee node that represents method $B.m1$. This difference between the ECFGs for $D.m3$ in P and P' lets our analysis determine that this call to $a.m1$ may behave differently in P and P'. Note that a simple textual comparison would identify the addition of the method, but it would require a manual inspection of the code (or some further analysis) to identify the points in the code where such change can affect the behavior of the program.

Variable and object types

When modifying a program, changing the type of a variable may lead to changes in program behavior (e.g., if a *long* is changed to an *int*). To identify these kinds of changes, in our representation, we augment the name of scalar variables with type information. For example, a variable *a* of type *double* is identified as *a_double* in our representation. This method for representing scalar variables reflects any change in the variable type in the locations where that variable is referenced.

Another possible change that may lead to subtle changes in program behavior is the modification of class hierarchies (e.g., moving a class from one hierarchy to another, by changing the class that it extends). All effects of these changes that result in different bindings in P and P' are captured by our method-call representation. Other effects, however, need to be specifically addressed. To this end, instead of explicitly representing class hierarchies, we encode the hierarchy information at each point where a class is used as an argument to operator $instanceof$, as an argument to operator *cast*, and as a type of newly created exception. To encode the type information, we use globally-qualified class names. A *globally-qualified class name* for a class contains the entire inheritance chain from the root of the inheritance tree (i.e., from class $java.lang.Object$) to its actual type.[3] The interfaces that are implemented by the class are also included in globally-qualified names. If a class implements more than one interface, the names of the interfaces are inserted in alphabetical order. This method reflects changes in class hierarchies in the locations where the change may affect the program behavior. For example, nodes 7 and 19 in Figure 3 show the globally-qualified name for class $E3$ in P and P', respectively.

Exception Handling

As for dynamic binding, when the program is modified, the presence of exception-handling constructs can cause subtle side effects in parts of the code that have no obvious relation to the modified parts of the code. For example, a modification of the type of an exception, or of a catch block, can cause a previously caught exception to go uncaught in the modified program, thus changing the flow of control in unforeseen ways.

To identify these changes in the program, we explicitly model, in the ECFG, exception-handling constructs in Java code. We represent such constructs using an approach similar to that used in Reference [3]. For each try statement, we create a *try* node and an edge between the try node and the node that represents the first statement in the try block.

We then create a *catch* node and a CFG to represent each catch block of the try statement. Each catch node is labeled with the type of the exception that is caught by the corresponding catch block. An edge connects the catch node to the entry of the CFG for the catch block.

An edge, labeled "exception", connects the try node to the catch node for the first catch block of the try statement. That edge represents all control paths, from the entry node of the try block, along which an exception can be propagated to the try statement. An edge labeled "exception" connects also the catch node for a catch block b_i to the catch node for catch block b_{i+1} that follows b_i (if any). This edge represents all control paths, from the entry node of the try block, along which an exception is (1) raised, (2) propagated to the try statement, and (3) not handled by any of the catch blocks that precede b_{i+1} in the try statement.

Our representation models finally blocks by creating a CFG for each finally block, delimited by *finally entry* and *finally exit* nodes. An edge connects the last node in the corresponding try block to the finally entry node. The representation also contains one edge from the last node of each catch block related to the finally to the finally entry node. If there are exceptions that cannot be caught by any catch block of the try statement, and there is at least one catch block, we also connect the catch node for the last catch block to the finally entry node.

Because the information we use to build the exception-related part of the ECFG is computed through interprocedural exception analysis [11], we can represent both intra- and inter-procedural exception flow. If an exception is thrown in a try block for a method m, the node that represents the throw statement is connected to (1) the catch block in m that would catch the exception, if such a catch block exists, (2) the finally entry node, if no catch block can catch the exception and there is a finally block for the considered try block, or (3) to the exit node of m's ECFG, otherwise. Conversely, if an exception is thrown in method m not from within a try block, the node that represents the throw statement is always connected to the exit node of m's ECFG.

For example, consider again method $D.m3$ in P (Figure 1) and its ECFG (Figure 3(a)). The ECFG contains a try node for the try block (node 6) and catch nodes for the two catch blocks associated with the try block (nodes 8 and 10). The catch nodes are connected to the entry nodes of the CFGs that represent the corresponding catch blocks (nodes 9 and 11). Also, the node that represents the throw statement in the code is connected to the catch node whose type matches the type of the exception (edge $< 7, 8 >$).

Consider now the other difference between P and P': the modification in the type hierarchy that involves class $E3$. $E3$ is a subclass of $E2$ in P and a subclass of $E1$ in P'. Such a change causes a possibly different behavior in P and P' because the exception thrown in method $D.m3$ is caught by different catch blocks in P and P'. Because, in P', class $E3$ is no longer a subclass of $E2$, edge $< 7, 8 >$, which connects the throw node to the catch node for exception $E2$ in P, is replaced by edge $< 19, 22 >$, which connects the throw node with the catch node for exception $E1$ in P'.

Figure 3(b) shows how the possibly different behavior is reflected in our representation: the node that represents the throw statement is connected to two different catch nodes in the ECFGs for $D.m3$ in P and P'. In addition, the nodes that represent the throw statement in P and P' (nodes 7 and 19) differ because of the use of globally-qualified names for the exception types. These differences between the two ECFGs let our analysis determine that, if the throw statement is traversed, P and P' may behave differently. Note that this would occur also for an exception that propagates outside the method, due to the difference between the globally-qualified names in nodes 7 and 19. Also in this case, a simple textual comparison would only allow us to identify the change in the type of $E3$, whereas identifying the side effects of such a change would require further analysis.

[3] For efficiency, we actually exclude class *Object* from the name, except that for class *Object* itself.

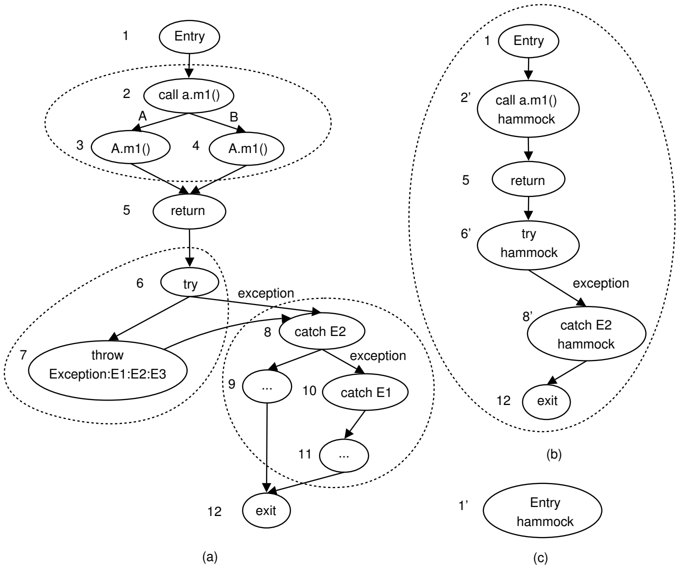

Figure 4: ECFG for $D.m3$ (a), intermediate hammock graph for $D.m3$ (b), and resulting hammock node for $D.m3$ (c)

Synchronization

Java provides explicit support for threading and concurrency through the `synchronized` construct. Using the `synchronized` construct, Java programmers can enforce mutual exclusion semaphores (mutexes) or define critical sections, that is, atomic blocks of code. Synchronized areas of code can be declared at the block, method, and class level.

In the ECFG, we account for synchronized areas of code by creating two special nodes: synchronize start and synchronize end. A *synchronize start* node is added before the node that represents the first statement of a synchronized area of code. Analogously, a *synchronize end* node is added after the node that represents the last statement of a synchronized area of code.

In a program that uses synchronized constructs, changes in behavior can occur because (1) an area of code that was not synchronized becomes synchronized, (2) an area of code that was synchronized is no longer synchronized, or (3) a synchronized area is expanded or contracted. In the ECFG, these cases are suitably captured by addition, removal, or replacement of synchronize start and synchronize end nodes.

Reflection

In Java, reflection provides runtime access to information about classes' fields and methods, and allows for using such fields and methods to operate on objects. In the presence of reflection, our representation can fail to capture some of the behaviors of the program. For example, using reflection, a method may be invoked on an object without performing a traditional method call on that object. Although some uses of reflections can be handled through analysis, others require additional, user-provided information. In our work, we assume that such information is available and can be leveraged for the analysis. In particular, in the case of dynamic class loading, we assume that the classes that can be loaded (and instantiated) by name at a specific program point are specified by the user.

2.4.2 Hammocks

Our algorithm uses hammock graphs (hereafter, hammocks) for its comparison of two methods. Hammocks are single-entry, single-exit subgraphs [2] and provide a way to impose a hierarchical structure on the ECFGs that facilitates the matching. Formally, if G is a graph, a *hammock* H is an induced subgraph of G with a distinguished node V in H called the *entry node* and a distinguished node W not in H called the *exit node* such that

1. All edges from (G - H) to H go to V.
2. All edges from H to (G - H) go to W.

Similar to the approach used in Reference [6], once a hammock is identified, our algorithm reduces it to a *hammock node* in three steps. First, the set of nodes inside the hammock is replaced by a new node. Second, all incoming edges to the hammock are redirected to the new node. Third, all edges leaving the hammock are replaced by an edge from the new node to the hammock exit. The resulting graph at each intermediate step is called a *hammock graph*.

Figure 4 illustrates the steps and hammock graphs in reducing ECFGs for P to a single node. The regions inside the dotted lines—nodes 2–4, nodes 6 and 7, and nodes 8–11—in Figure 4(a) represent the three hammocks that are identified and then replaced by hammock nodes 2', 6' and 8', respectively, as shown in Figure 4(b). Then, all nodes in Figure 4(b) are identified as a hammock and reduced to a single node, as shown in Figure 4(c). To identify all hammocks, we use an algorithm described in Reference [2].

A hammock H with start node s is minimal if there is no hammock H' that (1) has the same start node s, and (2) contains a smaller number of nodes. Hereafter, when we use the term hammock, we always refer to a minimal hammock, unless otherwise stated.

2.4.3 Hammock Matching Algorithm

Our hammock matching algorithm, `HmMatch`, is given in Figure 5. The algorithm is based on Laski and Szermer's algorithm for finding an isomorphism between two graphs [6]. `HmMatch` takes as input n and n', two hammock nodes, LH, an integer indicating the maximum lookahead, and S, a threshold for deciding whether two hammocks are similar enough to be considered a match. The algorithm outputs N, a set of pairs whose first element is, in turn, a pair of matching nodes, and whose second element is a label that indicates whether the two nodes are "unchanged" or "modified."

To increase the number of matches, we modified Laski and Szermer's algorithm to allow the matching of hammocks at different nesting levels. This modification accounts for some common changes that we encountered in our preliminary studies, such as the addition of a loop or of a conditional statement at the beginning of a code segment. In the following, we first describe algorithm `HmMatch` and then present an example of use of the algorithm on the code in Figure 1.

`HmMatch` expands the two input hammock nodes, n and n', into two hammock graphs, G_e and G'_e (line 1). In the expansion process, a dummy exit node is added and all edges from nodes inside the hammock to the actual exit node are redirected to the dummy node. At line 2, `HmMatch` adds the two dummy exit nodes as a pair of matching nodes to set N. Then, the algorithm starts matching nodes in the two graphs by performing a depth-first pairwise traversal of G_e and G'_e, starting from their start nodes. Therefore, at line 3, the pair of start nodes is added to stack ST, which functions as a worklist. Each iteration of the main *while* loop (lines 4–28) extracts one node pair from the stack and checks whether the two nodes match. The body of the loop

```
procedure HmMatch
Input: hammock node in original version $n$,
       hammock node in modified version $n'$
       maximum lookahead $LH$
       hammock similarity threshold $S$
Output: set of pair <{node,node},label> $N$
Use: $succs(A)$ returns set of successors of each node $a$ in $A$
     $comp(m, n, S, N)$ returns true if m and n are matched
     $edgeMatching(n, n')$ returns matched outgoing edge pairs
Declare: stack of $< node, node >$ $ST$
         current depth $d$
         expanded graphs $G_e$ and $G'_e$
         current nodes $c$ and $c'$
         lookahead node sets $L$ and $L'$
         pair $< node, node >$ $match$

Begin: HmMatch
 1: expand $n$ and $n'$ one level to graphs $G_e$ and $G'_e$
 2: add exit-node label pair $< \{x, x'\}$, "$unchanged$" $>$ to $N$
 3: push start node pair <s,s'> onto $ST$
 4: while $ST$ is not empty do
 5:   pop <c,c'> from $ST$
 6:   if $c$ or $c'$ is already matched then
 7:     continue
 8:   end if
 9:   if $comp(c, c', S, N)$ then
10:     $match = \{c, c'\}$
11:   else
12:     $match = null$; $L = \{c\}$; $L' = \{c'\}$
13:     for $(d = 0; d < LH; d + +)$ do
14:       $L = succs(L)$; $L' = succs(L')$
15:       if $\bigvee_{p' \in L'} comp(c, p', S, N) \vee \bigvee_{p \in L} comp(c', p, S, N)$ then
16:         set match to the first pair that matches
17:         break
18:       end if
19:     end for
20:   end if
21:   if match != null then
22:     push <match,"unchanged"> onto $N$
23:     set $c$ and $c'$ to the two nodes in $match$
24:   else
25:     push <{c,c'},"modified"> onto $N$
26:   end if
27:   push a pair of sink nodes for each edge pair returned from
      $edgeMatching(c, c')$ onto $ST$
28: end while
end HmMatch
```

Figure 5: A hammock matching algorithm

first checks whether any of the nodes in the current pair is already matched (line 6). A matched node that has already been visited must not be considered again; therefore, if this is the case, the algorithm continues by considering the next pair in the worklist (line 7).

To compare two nodes, HmMatch invokes $comp(c, c', S, N)$ (line 9), where c and c' are the two nodes to compare, S is the similarity threshold for matching hammocks, and N is the set of matching nodes. Unless c and c' are hammocks, *comp* returns *true* if the two nodes' labels are the same. If c and c' are hammocks, *comp* (1) recursively calls HmMatch, to obtain the set of matched and modified pairs, and (2) computes the ratio of unchange-matched pairs in the set to the number of nodes in the smaller hammock. If the ratio is greater than threshold S, *comp* returns *true* (i.e., the two hammocks are matched) and pushes all pairs in the set returned by HmMatch onto N. Otherwise, *comp* returns $false$.

If two nodes c and c' are matched (i.e., *comp* returns *true*), they are stored in variable *match* as a pair (line 10) and further added to the set of matched nodes with label "unchanged" (line 22). Otherwise, HmMatch tries to find a match for c (resp., c') by examining c''s (resp., c's) descendants, up to a given maximum lookahead (lines 12–19). First, *match* is initialized to null, and the lookahead sets L and L' are initialized to contain only the current nodes (line 12). The algorithm then executes the for loop until a match is found or depth d reaches the maximum lookahead LH (lines 13–19). At each iteration, the algorithm updates L and L' to the sets of successors of their members, obtained by calling procedure *succs* (line 14). $succs(L)$ returns, for each node l in L and each outgoing edge from l, the sink of such edge. If node l is a hammock node, *succs* returns a set that consists of the start node and the exit node of the hammock. In this way, a match can occur between nodes inside hammocks at different nesting levels. After computing the lookahead sets L and L', the algorithm compares each node in set L' with c and each node in set L with c' (line 15). If there is a match, the search stops, and the first matching pair found is stored in variable *match* (lines 16–17). The matching pair is then added to the set of matched nodes with label "unchanged" (line 22). After two nodes have been matched as unchanged, c and c' are set to be the two nodes in the matching pair (line 23).

If no matching is found, even after the lookahead, c and c' are added to the set of matched nodes with label "modified."

After processing nodes c and c', the outgoing edges from the two nodes are matched by calling $edgeMatching(c, c')$. *edgeMatching* matches outgoing edges from c and c' based on their labels. For each pair of matching edges, the corresponding sink nodes are pushed onto worklist ST (line 27). At this point, the algorithm continues iterating the main *while* loop until ST is empty.

When the algorithm terminates, all nodes in the old version that are not in any pair (i.e., that have not been matched to any other node) are considered deleted nodes. Similarly, all nodes in the new version that are not in any pair are considered added nodes.

To better illustrate HmMatch, we consider a partial run of CalcDiff on the example code in Figure 1. In particular, we consider the execution from the point in which the pair of methods $D.m3$ in P and P' is compared (line 5). At line 6 of CalcDiff, the ECFGs for the methods are created, and at lines 7 and 8 of the algorithm, hammocks in the ECFGs are identified and reduced to single hammock nodes. Then, at line 9, CalcDiff calls HmMatch, passing it the two hammock nodes. For the example, we assume that the lookahead threshold (LH) is 1, and that the hammock similarity threshold (S) is 0.5.

HmMatch first expands the hammock nodes, adds the dummy exit nodes, and suitably connects them (line 1). Fig. 6 shows the resulting hammock graphs for the original and modified version of the program. Then, HmMatch adds dummy exit nodes 7 and 13 as a pair of matching nodes to set N (line 2) and adds the pair of start nodes <1,8> to stack ST (line 3).

In the first iteration of the main *while* loop, the algorithm extracts node pair <1,8> from ST (line 5). Because neither node is already matched, the algorithm compares the two nodes by calling $comp(1, 8, 0.5, N)$ (line 9), which compares the nodes' labels and returns *true*. Therefore, the algorithm sets *match* to this pair (line 10), adds the pair of nodes to N with label "unchanged," and sets c and c' to be nodes 1 and 8 (lines 22-23), which in this case leaves c and c' unchanged. At this point, the outgoing edges from 1 and 8 are matched by calling $edgeMatching(1, 8)$. Each node in the entry pair has only one outgoing edge, and the two edges match, so the pair of sink nodes $< 2, 9 >$ is pushed onto the worklist.

In the second iteration of the main *while* loop, because nodes 2 and 9 are not already matched and are both hammock nodes, *comp* (line 9) calls HmMatch(2,9,1,0.5). HmMatch

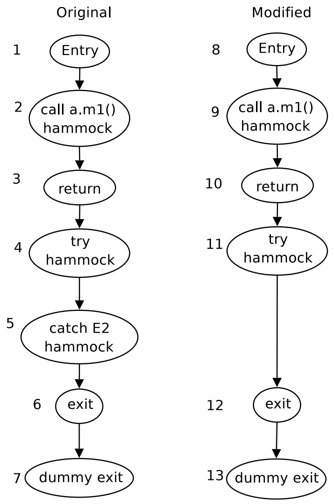

Figure 6: Hammock graphs for the original and modified version of $D.m3$.

expands nodes 2 and 9 to get the two graphs shown in Fig. 7, matches the dummy exit nodes 17 and 21, and pushes the pair of start nodes <14,18> onto ST_1.[4] This pair is then extracted from the stack and compared (lines 5–9). Because both nodes have the same label, they are matched, and the pair is pushed onto N_1 with label "unchanged" (lines 22–23). $edgeMatching$ is then called on the two nodes in the pair, 14 and 18; $edgeMatching$ matches likely labeled edges and the two pairs of sink nodes <15,19> and <16,20> are pushed onto ST_1.

In the next iteration of the main loop, the nodes in pair <15,19> are compared, matched, and pushed onto N' with label "unchanged," and the pair of direct successors of nodes 15 and 19, <17,21>, is then pushed onto ST_1. At the following loop iteration, the nodes in pair <16,20> are compared. Because they do not match, a lookahead is performed, trying to match nodes 16 and 21 and nodes 20 and 17. Because neither comparison is successful, the pair <16,20> is added to N_1 with label "modified," and the pair of successors <17,21>, is then pushed onto ST_1 (so the pair appears twice on the stack). In the next two iterations, the pair <17,21> is processed: because both nodes are already matched (being the dummy exits), the algorithm skips them. At this point, ST_1 is empty and HmMatch returns to the calling procedure, *comp*. Because 3 out of 4 nodes are unchange-matched, and the similarity threshold is 0.5, *comp* classifies the two hammocks as matched. Therefore, the pairs in N_1 are pushed onto N, *comp* returns *true* (line 9), pair <2,9> is pushed onto N with label "unchanged" (line 22), and pair <3,10> is pushed onto ST (line 27).

Pair <3,10> is then matched and pair <4,11> is pushed onto ST and then compared in the next iteration. For the sake of space, we do not show the comparison of these two hammocks and the rest of the execution of the algorithm. The part of the example shown so far already illustrated the main parts of the algorithm, including the matching of hammock nodes and the lookahead.

2.5 Worst-Case Time Complexity

The dominating cost of the algorithm is the matching at the node level. Let m and n be the number of nodes in all matched methods in the original and modified version of a program. Let k be the maximum number of nodes in a method, and let the maximum lookahead be greater than k. In the worst case, if no matching of hammocks at different nesting levels occurs, the algorithm compares each node in a method with all nodes in the matching method (at most, k), leading to a worst-case complexity of $O(k \cdot min(m, n))$. If matching of hammocks at different nesting levels occurs, the algorithm may compare a pair of nodes more than once. To decide whether two hammocks are matched, $HmMatch$ compares each node in one hammock with nodes in the other hammock and counts the number of matched nodes. If lookahead is performed, the same pairs of nodes are compared again in the context of the new pair of hammocks. The number of times the same pairs of nodes are compared depends on the maximum nesting depth of hammocks and on the maximum lookahead. If d is the maximum nesting level of hammocks and l is the maximum lookahead, the worst-case complexity of our algorithm is then $O(k \cdot min(d, l) \cdot min(m, n))$.

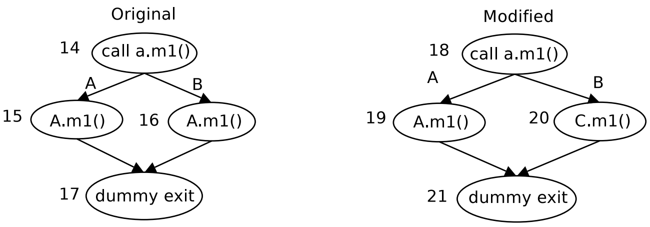

Figure 7: Hammock graphs for hammock nodes 2 and 9 in Fiugure 6.

3. STUDIES

To evaluate our algorithm, we implemented a tool, JDiff, and performed three studies on a real, medium-sized program. In this section, we describe our experimental setup, present the studies, and discuss their results.

3.1 Experimental Setup

JDiff is a Java tool that implements our differencing algorithm. The tool consists of two main components: a differencing tool and a graph-building tool. The differencing tool inputs the original and modified versions of a program, compares them, and outputs sets of pairs of matching classes, methods, and nodes. To build ECFGs, the tool leverages the capabilities of Jaba (Java Architecture for Bytecode Analysis),[5] a Java-analysis front-end developed within our research group. We are currently using JDiff in our Jaba-based impact analysis and regression testing tools [8].

As a subject for our studies, we use Jaba, the analysis tool described above. Jaba contains 550 classes, 2800 methods, and approximately 60KLOC. To evaluate the effectiveness of our algorithm, we ran JDiff on pairs of successive versions of Jaba extracted from its CVS repository.

For the timing experiments, we used two dedicated Pentium4 2.80GHz PCs, with 1 GB of memory, running GNU/Linux 2.4.22.

There are several threats to the validity of our studies. An external threat exists because we conducted the studies using only one subject program. Thus, we cannot claim generality for our results. However, the subject program is a real program, that is used constantly by students and researchers in and outside our group, and the changes considered are real changes that include bug fixes and feature enhancements.

[4] We use the subscript notation to distinguish variables in recursively called procedures.

[5] http://www.cc.gatech.edu/aristotle/Tools/jaba.html

Another external threat to validity exists for the first study: we used only one test suite throughout the study. Different test suites may generate different results.

Threats to internal validity mostly concern possible errors in our algorithm implementations and measurement tools that could affect outcomes. To control for these threats, we validated the implementations on known examples and performed several sanity checks.

3.2 Study 1

The goal of Study 1 is to assess the effectiveness of our algorithm for a particular task—coverage estimation.

For the discussion, let P represent the original version of the program, P' represent the modified version of P, and T represent a test suite used for testing P. The coverage information for P, C_P, records, for each test case $t \in T$, entities exercised by t. *Coverage estimation* estimates $C_{P'}$ from C_P and the correspondences between entities in P and P'. In this study, we consider a node in the ECFG as an entity in the program. Therefore, the coverage information is expressed in term of a set of nodes that each test case t exercises.

We used four versions of Jaba and its development test suite T, which contains 707 test cases and covers approximately 60% of the program. The size of the change sets between versions in the pairs ranges from a few modified methods to approximately 20 methods.

Let P_i represent version i of the Jaba program, where $1 \leq i \leq 4$. We first ran the test suite T on each version P_i to collect C_{P_i}. Second, for each $< P_1, P_i >$ pair, where $2 \leq i \leq 4$, we computed the node correspondences between P_1 and P_i using JDiff. Third, we computed the estimated coverage as follows:

- For each node n in the modified version (i.e., the second element of each $< P_1, P_i >$ pair), if there exists a correspondence between node n and some node n' in the original version (i.e., P_1), then we estimate that n is exercised by the test cases in T that exercised n'.
- If there is no such correspondence, we visit the predecessors of n in the ECFG until either the predecessor is a branch, or the predecessor is a node q that has a correspondent node q' in P_1. In the former case, n is designated as "not exercised." In the latter case, we estimate that n is exercised by all the test cases in T that exercised q' (because n is in the same control-dependence region of q).

Finally, we compared the estimated and the actual coverage, and recorded the number of nodes whose coverage is correctly estimated.

Pair	Number of Nodes	Average Covered	Average Correct/ Test Case	Correct/ Test Suite
v1,v2	22026	12591.75	98.57	97.17
v1,v3	22063	12620.75	98.46	97.17
v1,v4	22315	12661.23	98.03	96.20

Table 1: Coverage-estimation results for part 1 of Study 1.

Table 1 shows, for each pair of versions, the number of nodes in the parts of the modified version that we consider (*Number of Nodes*), the average number of nodes covered by a test case (*Average Covered*), the number of nodes whose coverage of each test case $t \in T$ is correctly estimated, averaged over all test cases (*Average Correct / Test Case*), and the number of nodes whose coverage of the entire test suite is correctly estimated (*Correct / Test Suite*). For example, there are 22,063 nodes in $v3$, and a single test case exercised about 12,620 nodes on average. For this set of nodes, on average, 98.46% of the nodes' coverage for a single test case is correctly estimated, and 97.17% of the nodes' coverage for the entire test suite is correctly estimated. The results show that estimated coverage is high for the pairs we studied but that, as expected, it slightly degrades as the change set becomes larger (i.e., when considering pairs $< v1, v3 >$ and $< v1, v4 >$).

To further evaluate the effectiveness of our algorithm when used for coverage estimation, we applied it on program versions with a large number of changes. To this end, we extracted four additional versions of Jaba from its CVS by selecting a period of time in which Jaba was undergoing a major restructuring. We call these versions *va*, ..., *vd*. We constructed pairs of versions using the first version *va* as the first element of each pair. The sizes of the change sets in these pairs are 15 methods for $< va, vb >$, 100 methods for $< va, vc >$, and 150 methods for $< va, vd >$. The test suite for these versions consists of 155 test cases, which covers approximately 45% of the program.

Pair	Number of Nodes	Average Covered	Average Correct/ Test Case	Correct/ Test Suite
va,vb	19639	12199.60	96.25	95.30
va,vc	20076	12343.54	86.08	80.97
va,vd	20258	12316.49	84.70	77.81

Table 2: Coverage-estimation results for part 2 of Study 1.

Table 2 gives the data for this set of pairs of versions in the same format as Table 1. The results show that, also for this set of versions, the quality of estimated coverage degrades when the change sets becomes larger. However, for the largest change set, our algorithm still computes estimated coverage that correctly matches the actual coverage for 84.70% of the nodes (on average) for the single test cases, and for 77.81% of the nodes for the entire test suite. Our results are consistent with the findings in Reference [1].

3.3 Study 2

The goal of Study 2 is to measure the efficiency of JDiff for various values of lookahead, LH, and hammock similarity threshold, S.

In this study, we used the first three pairs of versions of Jaba that we used in Study 1. We ran JDiff on each pair of versions $< P_1, P_i >$, where $2 \leq i \leq 4$, with different values for LH and S, and collected the running times. This running time includes the time for creating the ECFGs, as well as comparing and matching classes, interfaces, methods, and nodes in the two programs.

Figure 8 shows the running time (in seconds) of JDiff on a number of pairs of versions, lookaheads, and similarity thresholds. The x-axis represents the value of LH, and the y-axis represents the running time for each pair of versions and for two different values of S. For example, JDiff took about 375 seconds when run on versions $< v1, v2 >$ for $(S > 0)$ and LH of 10. We present the results for only two values of S $(S = 0$ and $S > 0)$ because we found that, for this subject,

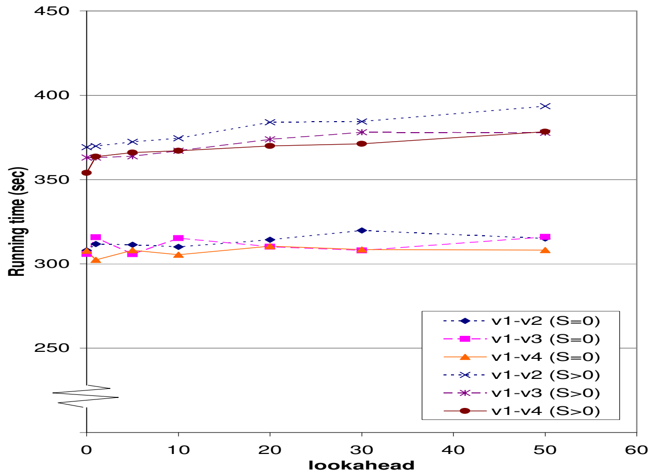

Figure 8: Average time (sec) for various pairs of versions, lookaheads, and similarity thresholds.

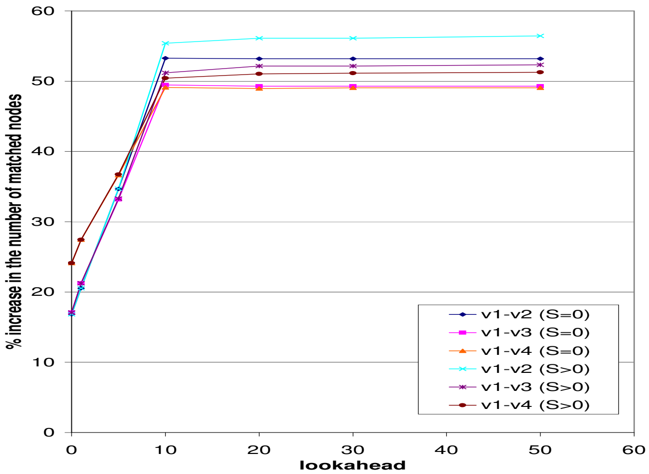

Figure 9: Percentage increase in matched nodes.

the running time is almost the same for $S = 0.2, 0.4, 0.6$, and 0.8. Our results show that, LH staying the same, the value of S affects the running time of the comparison. Intuitively, with $S = 0$, the algorithm matches a hammock in the original program's ECFG with the first hammock found in the modified version's ECFG. Thus, each hammock is compared at most once, and the running time is almost the same regardless of the value of LH. In addition, because each hammock is compared at most once, the running time for these cases is less than for cases where $S > 0$, where a hammock may be compared more than once. For $S > 0$, the number of times a hammock is compared depends on the lookahead and on the actual changes. As shown in the results, only in this case does the time slightly increase when the lookahead increases. Note that, in all cases, JDiff took less than seven minutes to compute the differences between a pair of versions.

3.4 Study 3

The goal of Study 3 is to evaluate the effectiveness of our algorithm compared to Laski and Szermer's algorithm [6].

To do this, we compared the number of nodes that each algorithm matches. For the study, we implemented Laski and Szermer's algorithm (LS) by modifying our tool. The paper that presents LS does not discuss how to handle some specific cases. For example, when two hammocks have the same label, they are expanded and compared, but the algorithm behavior is undefined in the case in which the expanded graphs cannot be made isomorphic by applying node renaming, node removing, and node collapsing. We handle those cases in the same way for both algorithms. There are three differences between the two algorithms: (1) LS does not use the lookahead but searches the graphs until the hammock exit node is found; (2) LS does not allow the matching of hammocks at different nesting levels; and (3) LS does not use the hammock similarity threshold but decides whether two hammocks are matched by comparing the hammocks' entry nodes only.

We ran both algorithms on the first three pairs of versions of Jaba used in Study 1, and counted the number of nodes in each group of added, deleted, modified, and unchanged nodes. We consider only nodes in modified methods because added, deleted, and unchanged methods do not show differences in matching capability between the two algorithms. In our preliminary studied, we found that LS identified, in the modified methods, about 5,000 nodes as added and about 5,000 nodes as deleted.

Figure 9 presents the results of this study. The x-axis represents the size of LH, and the y-axis shows, for each pair of versions and values of S, the percent increase in the number of matched nodes over the number of nodes identified as added by LS For example, our algorithm (for $S > 0$, and $LH = 20$) matches about 55% more nodes than LS when both algorithms run on the same pair $< v1, v2 >$. In this case, our algorithm matches an additional 2750 nodes over the number of nodes matched by LS. The results show that the increase achieved by our algorithm ranges from about 17% to over 65%. Note that added nodes identified by LS or `CalcDiff` can be classified as (1) code that is actually added, or (2) code that cannot be matched because of the limitations of the algorithms. Therefore, to measure the relative effectiveness of the two algorithms, the percentage should be computed using the number of nodes only in the second category. In this sense, the percent improvement that we measured in our study is an underestimate of the actual improvement.

The results also show that the number of matched nodes increases when LH increases, which is intuitively reasonable. Finally, the results show that the number of matched nodes is slightly higher in the case of $S > 0$ than in the case of $S = 0$, for $LH > 10$.

4. RELATED WORK

There are a number of existing techniques for computing differences between two versions of a program that are related to ours. The UNIX `diff` utility [7], as discussed in the Introduction, compares two text files line-by-line and outputs differences in the files. However, because `diff` compares files line-by-line, it does not recognize differences that can occur because of changes in object-oriented features such as changes related to polymorphism and exception handling. In contrast, our differencing technique does handle these object-oriented features, and thus, can provide information that can be used for various software engineering tasks.

BMAT [12] (binary matching tool) is similar to our algorithm. However, it performs matching on both code and data blocks between two versions of a program in binary format. BMAT uses a number of heuristics to find matches for as many blocks as possible. Being designed for the purpose of program profile estimation, BMAT does not provide information about differences between matched entities (unlike our algorithm). Moreover, BMAT does not compute information about changes related to object-oriented constructs.

`Semantic diff` [5], compares two versions of a program procedure-by-procedure. `Semantic diff` computes a set of input-output dependencies for each procedure and identifies the differences between two sets from the same procedure in the original and the modified versions. However, `semantic diff` is performed only at the procedure level and may miss some changes that do not affect the dependencies of variables (e.g., changing branch conditions) but may drastically change the behavior of the program. Furthermore, because there is no input-output dependency changes, it will fail to detect some kinds of changes (e.g., constant value changed) that may affect the program behavior. Conversely, our technique is able to identify these differences.

Horwitz's approach [4] computes both syntactic and semantic differences between two programs using a partitioning algorithm. Horwitz's technique is based on the program representation graph (PRG). Because PRGs are defined only for programs written in a language with scalar variables, assignment statements, conditional statements, while loops, and output statements only, the technique is limited and cannot be used in general. In particular, it cannot be applied to object-oriented programs.

Laski and Szermer present an algorithm that analyzes corresponding control-flow graphs of the original and modified versions of a program [6]. Their algorithm localizes program changes into clusters, which are single entry, single exit parts of code. Clusters are reduced to single nodes in the two graphs, and then these nodes are recursively expanded and matched. As we discussed in Section 2.4.3, our algorithm is based on this algorithm. However, we make several modifications to the algorithm to improve matching capability (e.g., matching hammocks at different nesting levels, hammock similarity metric, and threshold). In Study 3, we show how our algorithm outperforms, for the case considered, Laski and Szermer's approach in terms of effectiveness in matching two programs.

Although not directly related to this work, Reference [9] presents an interesting study of the type of changes that occur in object-oriented systems during maintenance. Their results confirm that changes in such systems often involve changes in the behavior of object-oriented constructs that must be suitably handled.

5. CONCLUSION

In this paper, we presented an algorithm for comparing two Java programs. The algorithm is based on a method-level representation that models the object-oriented features of the language. Given two programs, our algorithm identifies matching classes and methods, builds our representation for each pair of matching methods, and compares the representation for the two methods to identify similarities and differences. The results of our differencing can be used for various development and maintainance tasks, such as impact analysis and regression testing.

We also presented a tool that implements our technique (JDiff), and a set of studies that show the effectiveness and efficiency of the approach. Study 1 shows how our differencing technique can be successfully applied to perform coverage estimation. Study 2 illustrates the efficiency of the technique for different execution parameters. Study 3 compares our technique with the most closely related existent technique and shows that our technique achieves improvements from 17% to over 65% in terms of matching unchanged parts of the code.

In future work, we will investigate additional heuristics to further improve the matching results. Our initial experiences in this direction show that there are a number of tradeoffs, in terms of execution cost versus precision, that can be leveraged. We will also study typical changes in evolving system to assess whether the differencing algorithm could take advantage of known change patterns. Finally, we will investigate the use of our differencing algorithm to perform test suite augmentation, that is, selecting new test cases for a modified system based on the types of changes performed on the system.

6. REFERENCES

[1] S. Elbaum, D. Gable, and G. Rothermel. The impact of software evolution on code coverage information. In *Proceedings of the International Conference on Software Maintenance*, pages 169–179, November 2001.

[2] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[3] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for java software. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 312–326, November 2001.

[4] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 234–246, June 1990.

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

[6] J. Laski and W. Szermer. Identification of program nodifications and its applications in software maintenance. In *Proceedings of IEEE Conference on Software Maintenance*, pages 282–290, November 1992.

[7] E. W. Myers. An O(ND) difference algorithm an its variations. *Algorithmica*, 1(2):251–266, 1986.

[8] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *Proceedings of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, September 2003.

[9] X. Ren, F. Shah, F. Tip, B. G. Ryder, O. Chesley, and J. Dolby. Chianti: A prototype change impact analysis tool for java. Technical Report DCS-TR-533, Department of Computer Science, Rutgers University, September 2003.

[10] G. Rothermel and M. J. Harrold. A safe, efficient regressing test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, April 1997.

[11] S. Sinha and M. J. Harrold. Analysis and testing of programs with exception handling constructs. *IEEE Transaction on Software Engineering*, 26(9):849–871, September 2000.

[12] Z. Wang, K. Pierce, and S. McFarling. BMAT – a binary matching tool for stale profile propagation. *The Journal of Instruction-Level Parallelism*, 2, May 2000.