

JDiff: A differencing technique and tool for object-oriented programs

Taweessup Apiwattanapong · Alessandro Orso ·
Mary Jean Harrold

Published online: 29 December 2006
© Springer Science + Business Media, LLC 2006

Abstract During software evolution, information about changes between different versions of a program is useful for a number of software engineering tasks. For example, configuration-management systems can use change information to assess possible conflicts among updates from different users. For another example, in regression testing, knowledge about which parts of a program are unchanged can help in 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. Finally, we present the results of four empirical studies, performed on many versions of two medium-sized subjects, that show the efficiency and effectiveness of the technique when used on real programs.

Keywords Program differencing · Software evolution

An earlier version of this work has been presented at the 19th IEEE International Conference on Automated Software Engineering (ASE 2004) (Apiwattanapong et al., 2004).

T. Apiwattanapong (✉) · A. Orso · M. J. Harrold
Georgia Institute of Technology, Atlanta, Georgia, USA
e-mail: term@cc.gatech.edu

A. Orso
e-mail: orso@cc.gatech.edu

M. J. Harrold
e-mail: harrold@cc.gatech.edu

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 knowledge of the locations of changes between the two program versions. In addition, some tools also need the mapping of entities, such as statements 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 (also referred to as *stale profile propagation* (Wang et al., 2000)), the differencing results, along with the coverage or profile information from 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 (e.g., (Apiwattanapong et al., 2005; Bohner and Arnold, 1996; Law and Rothermel, 2003; Ren et al., 2004)) identifies the parts of a program that are affected by changes and, thus, requires knowledge of the locations of such changes. Many regression test selection techniques (e.g., (Leung and White, 1989; Orso et al., 2004; Rothermel and Harrold, 1996; Rothermel and Harrold, 1997)) use change information to select test cases to be rerun on modified versions of the software. For yet another example, in collaborative environments, differencing information is used for merging two modified versions of a software system into a new version that includes the changes from both earlier versions (Mens, 2002; Hunt and Tichy, 2002).

There are a number of existing techniques and tools for computing textual differences between files (e.g., the UNIX `diff` utility (Myers 1986)). However, these techniques are limited in their ability to detect changes in programs because they provide purely textual differences and do not consider changes in program behavior indirectly caused by textual modifications.

Consider, for example, the two partial Java programs in Fig. 1: the original program P and the modified version P' . If we were to inspect the output of `diff`, run on P and P' , we would see that method $B.m1$ has been added to 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.

Other existing differencing techniques are specifically targeted at comparing two versions of a program (e.g., (Horwitz, 1990; Jackson and Ladd, 1994; Wang et al., 2000)), but they are not suitable for object-oriented code. BMAT (Wang et al., 2000) cannot recognize, for instance, differences caused by changes in the exception hierarchy, and would fail to recognize that the exception thrown in $D.m3$ can be caught by different catch blocks in P and P' . Semantic diff (Jackson and Ladd, 1994)

Program P	Program P'
<pre> 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) {...} } } </pre>	<pre> 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) {...} } } </pre>

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

is defined to work only at the procedure level and cannot be straightforwardly extended to work on entire object-oriented programs. Horwitz's technique (Horwitz, 1990) 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)¹ to model behaviors caused by object-oriented features in the program. Using this graph, we identify changes in those behaviors and relate them to the points in the code where the different behavior occurs.

Our algorithm extends an existing differencing algorithm (Laski and Szermer, 1992) 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.

¹ A *control-flow graph* is a directed graph in which nodes represent statements and edges represent flow of control between statements.

- A tool, JDIFF, that implements our differencing algorithm for Java programs.
- A set of empirical studies, performed on two real, medium-sized programs that show the efficiency and precision of our algorithm.

2 Differencing algorithm

In this section, we first overview the algorithm. Then, we detail the levels at which the algorithm compares the original and modified versions of the program. Finally, we discuss the algorithm's complexity.

2.1 Overview

Our algorithm, `CalcDiff`, given in Fig. 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— LH and S —that are used in the node-level matching. Parameter LH is the maximum lookahead that `CalcDiff` uses when attempting to match nodes in methods. 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 (C), interfaces (I), and methods (M) in P and P' .

`CalcDiff` performs its comparison first at the 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`, using the code in Fig. 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 name. 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 Fig. 1, each class in P has a match in P' , and, thus, there is a pair in C for each class in P .

To improve the differencing results, our algorithm also accounts for the possibility of interacting with the user while matching classes and interfaces. After matching classes and interfaces in P with classes and interfaces in P' that have the same fully-qualified names, `CalcDiff` offers users the possibility of defining additional

Algorithm CalcDiff

Input: original program P
 modified program P'
 maximum lookahead LH
 hammock similarity threshold S

Output: set of $\langle \text{class}, \text{class} \rangle C$
 set of $\langle \text{interface}, \text{interface} \rangle I$
 set of $\langle \text{method}, \text{method} \rangle M$
 set of $\langle \langle \text{node}, \text{node} \rangle, \text{status} \rangle N$

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 $\langle c, c' \rangle$ in C or I **do**
- 4: compare methods; add matched method pairs to M
- 5: **for** each pair $\langle m, m' \rangle$ 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: $N = N \cup \text{HmMatch}(n, n', LH, S)$
- 10: **end for**
- 11: **end for**
- 12: **return** C, I, M, N

end CalcDiff

Fig. 2 Algorithm CalcDiff

matchings. Users can provide the algorithm with a match between unmatched classes (interfaces) in P and unmatched classes (interfaces) in P' . This additional feature accounts for cases in which the user renamed or changed the type of one or more classes and/or interfaces. The interaction with the user can be implemented efficiently because a match is required only for unmatched classes (rather than all classes) in P . Note that this part of the approach could be automated, at least partially, by matching unmatched classes and interfaces based on various similarity metrics (e.g., (Xing and Stroulia, 2005)).

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 only on the name. 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 increases the number of matches at the node level. Pairs of matching methods are added to M . Like the approach used for classes, methods in P that do not appear in set M are deleted methods, whereas methods in P' that do not appear in set M

are added methods. In the example (Fig. 1), there would be a pair in M for each method in P , but not for method $B.m1$ in P' (which would therefore be considered as added).

Analogous to the matching at the class level, the matching at the method level can also leverage user-provided information. Given two matching classes (or interfaces), c and c' , the user can provide matches between unmatched methods in c and c' . User-provided matchings can improve the differencing by accounting for cases in which either methods are renamed or their signature changes. Also in this case, this additional matching could be (partially) automated using similarity metrics at the method level (e.g., (Xing and Stroulia, 2005)).

2.4 Node level

`CalcDiff` uses the set of matched method pairs (M) to perform matching at the node level. First, the algorithm considers each pair of matched methods $\langle m, m' \rangle$ in M , and builds ECFGs G and G' for m and m' (lines 5–6). 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 C , I , M , and N (line 12).

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.² In the following, we illustrate how the ECFG represents various object-oriented features of the Java language.

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

² The ECFG is similar to the Java Interclass Graph (JIG) (Harrold et al., 2001) in terms of the handling of variable and object types and exception constructs. However, the JIG is an inter-procedural representation of a program whereas the ECFG is an intra-procedural representation of a method with summarized information from inter-procedural analyses. In addition, the ECFG also models synchronization constructs.

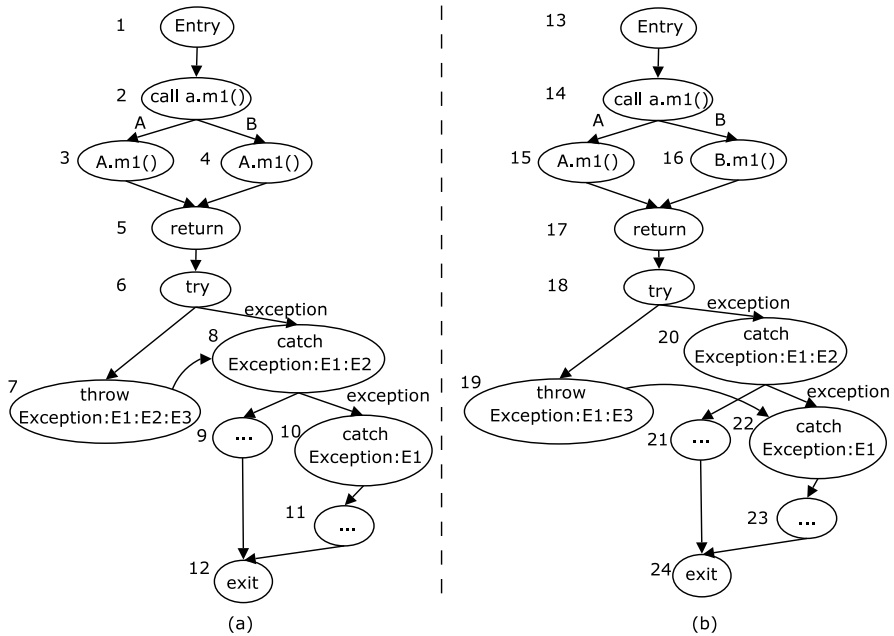


Fig. 3 ECFGs for $D.m3$ in P and P' (Fig. 1)

in any of the 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 *call* and *return* nodes. 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 (Fig. 1). The ECFG for $D.m3$ (Fig. 3(a)), contains two callee 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 Fig. 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 results 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 program's behavior.

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

Another change that may lead to subtle changes in program behavior is the modification of class and interface hierarchies (e.g., moving a class from one hierarchy to another, by changing the class that it extends). Hereafter, we will simply use “class hierarchies” to represent both class and interfaces hierarchies, unless otherwise stated. Effects of these changes that result in different bindings in P and P' are captured by our method-call representation. Other effects, however, must be specifically addressed. To this end, instead of explicitly representing class hierarchies, we encode the hierarchy information at points where a class is used as an argument to operator *instanceof*, as an argument to operator *cast*, as a type of a newly created exception, and as the declared type of a catch block. To encode the type information, we use globally-qualified names. A *globally-qualified 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 A *globally-qualified name* for an interface contains all the super interfaces 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 Fig. 3 show the different globally-qualified names for class $E3$ in P and P' , respectively.

Exception handling. As for dynamic binding, program modifications in the presence of exception-handling constructs can cause subtle side effects in parts of the code that have no obvious relation to the modifications. For example, a modification of an exception type or 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 Harrold et al. (2001). 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.

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

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} .

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, an edge connects 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 inter-procedural exception analysis (Sinha and Harrold, 2000), 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) the exit node of m 's ECFG, otherwise. Conversely, if an exception is thrown in method m from outside 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 (Fig. 1) and its ECFG (Fig. 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 $\langle 7, 8 \rangle$).

Consider now the second 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 $\langle 7, 8 \rangle$, which connects the throw node to the catch node for exception $E2$ in P , is replaced by edge $\langle 19, 22 \rangle$, which connects the throw node to 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. Again, a simple textual comparison would let us identify only the change in the type of $E3$, whereas identifying the side effects of such a change would require further analysis.

Synchronization. Java provides explicit support for threading and concurrency through the `synchronized` construct. Using such 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. For another example, a program may contain a predicate whose truth value depends on the number of fields in a given class; in such a case, the control flow in the program may be affected by the (apparently harmless) addition of unused fields to that class. Although some uses of reflection 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, for dynamic class loading, we assume that the classes that can be loaded (and instantiated) by name at a specific program point either can be inferred from the code (e.g., when a `cast` operator is used on the instance after the object creation) or are specified by the user. It is worth noting that, for the subjects used in our empirical studies (see Section 3), we provided such external information for all uses of reflection that could not be handled automatically by our analysis.

2.4.2 Hammocks

Our algorithm uses hammocks and hammock graphs for its comparison of two methods. Hammocks are single-entry, single-exit subgraphs (Ferrante et al., 1987) 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

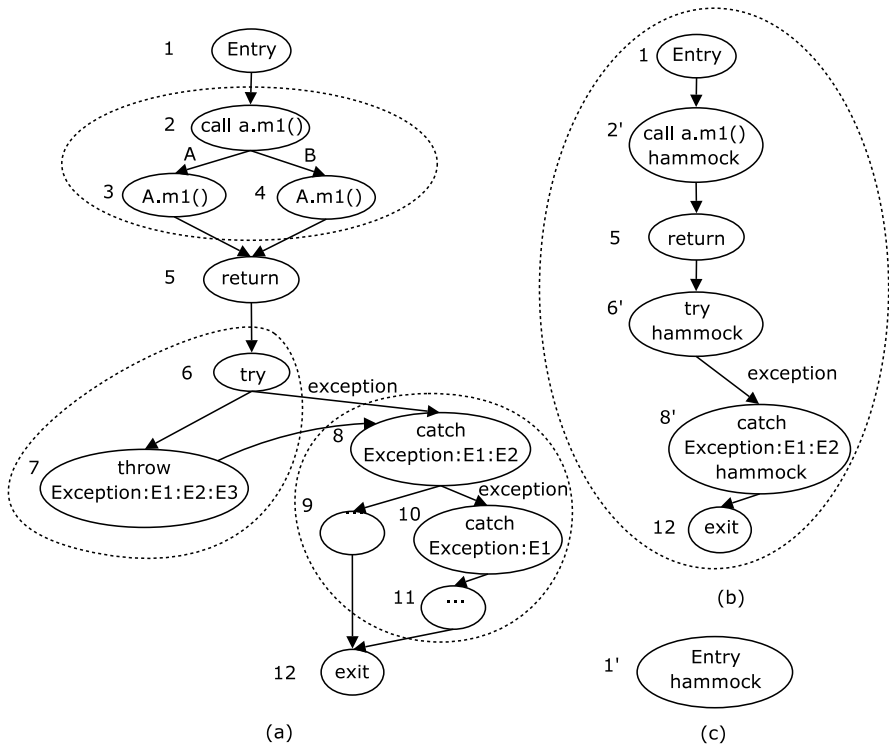


Fig. 4 ECFG for *D.m3* (a), intermediate hammock graph for *D.m3* (b), and resulting hammock node for *D.m3* (c)

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 Laski and Szermer (1992), once a hammock is identified, our algorithm reduces it to a *hammock node* in three steps. First, the nodes in the hammock are 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 how the ECFG for method *D.m3* in *P* is transformed into a single hammock node and the intermediate hammocks that are generated during the transformation. The regions inside the dotted lines in Fig. 4(a) (i.e., nodes 2 to 4, nodes 6 and 7, and nodes 8 to 11) represent the three hammocks that the algorithm identifies and replaces with hammock nodes 2', 6', and 8', respectively (Fig. 4(b)). Then, all nodes in Fig. 4(b) are identified as a hammock and, in turn, reduced to a single node (Fig. 4(c)). To identify hammocks, we use the algorithm described in Ferrante et al. (1987).

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 Fig. 5. The algorithm is based on Laski and Szermer’s algorithm for transforming two graphs into their respective isomorphic graphs (Laski and Szermer, 1992). `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 for 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 Fig. 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 in 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. Thus, at line 3, the pair of start nodes is added to stack ST , which the algorithm uses as a worklist. Each iteration over 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 first checks whether any node in the current pair is already matched (line 6). A matched node that has already been visited must not be considered again; in this 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 unchanged-matched nodes in the smaller *hammock* to the number of all nodes in that *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 later 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 the specified 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

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 $\langle \langle \text{node,node} \rangle, \text{label} \rangle N$

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 $\langle \text{node,node} \rangle ST$
 current depth d
 expanded graphs G_e and G'_e
 current nodes c and c'
 lookahead node sets L and L'
 pair $\langle \text{node,node} \rangle match$

Begin: HmMatch

```

1: expand  $n$  and  $n'$  one level to graphs  $G_e$  and  $G'_e$ 
2: add exit-node label pair  $\langle \langle x, x' \rangle, \text{"unchanged"} \rangle$  to  $N$ 
3: push start node pair  $\langle s, s' \rangle$  onto  $ST$ 
4: while  $ST$  is not empty do
5:   pop  $\langle c, c' \rangle$  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:     $match = \langle c, c' \rangle$ 
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  $match \neq null$  then
22:     push  $\langle match, \text{"unchanged"} \rangle$  onto  $N$ 
23:     set  $c$  and  $c'$  to the two nodes in  $match$ 
24:   else
25:     push  $\langle \langle c, c' \rangle, \text{"modified"} \rangle$  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

```

Fig. 5 Hammock matching algorithm

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 in 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 over 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 Fig. 1. In particular, we consider the execution from the point at 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.

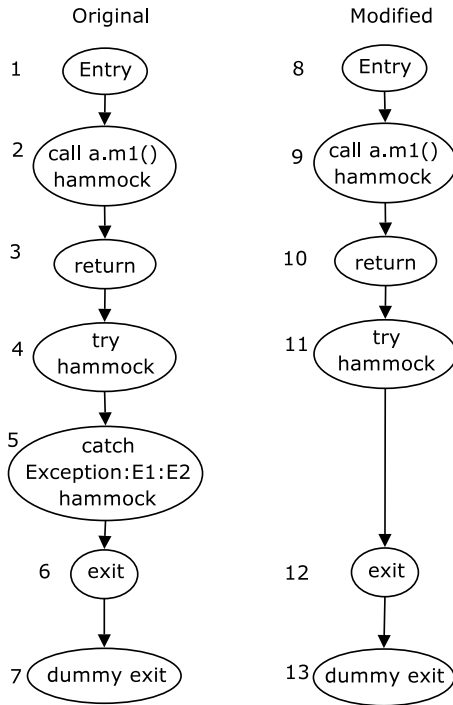
HmMatch first expands the hammock nodes, adds the dummy exit nodes, and suitably connects them (line 1). Figure 6 shows the resulting hammock graphs for the original and modified version of the program. Then, *HmMatch* adds dummy exit nodes 7 and 13 as a pair of matching nodes to set N (line 2) and pushes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

In the first iteration over the main *while* loop, the algorithm extracts node pair $\langle 1, 8 \rangle$ 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 $\langle 2, 9 \rangle$ is pushed onto the worklist.

In the second iteration over the main *while* loop, because nodes 2 and 9 are not already matched and are both hammock nodes, *comp* (line 9) calls *HmMatch* ($\langle 2, 9, 1, 0.5 \rangle$). *HmMatch* 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 $\langle 14, 18 \rangle$ onto ST_1 .⁴ This pair is then extracted from the stack and compared (lines 5–9). Because

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

Fig. 6 Hammock graphs for the original and modified version of *D.m3*



both nodes have the same label, they are matched, and the pair is added to N_1 with label “unchanged” (lines 22–23). *edgeMatching* is then called on the two nodes in the pair, 14 and 18; *edgeMatching* matches like-labeled edges and the two pairs of sink nodes $\langle 15,19 \rangle$ and $\langle 16,20 \rangle$ are pushed onto ST_1 .

In the next iteration over the main loop, the nodes in pair $\langle 15,19 \rangle$ are compared, matched, and pushed onto N_1 with label “unchanged,” and the pair of direct successors of nodes 15 and 19, $\langle 17,21 \rangle$, is then pushed onto ST_1 . At the following loop iteration, the nodes in pair $\langle 16,20 \rangle$ 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 $\langle 16,20 \rangle$ is added to N_1 with label “modified,” and the pair of successors $\langle 17,21 \rangle$ is then pushed onto ST_1 (so the pair appears twice on the stack). In the next two iterations, the pair $\langle 17,21 \rangle$ is processed: because both nodes

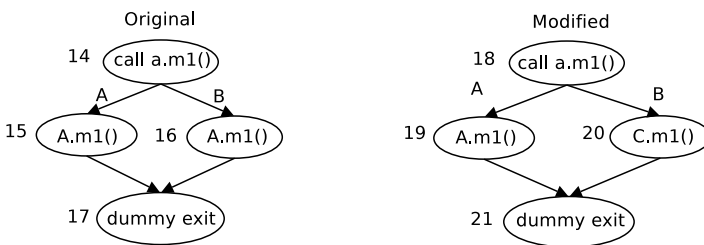


Fig. 7 Hammock graphs for hammock nodes 2 and 9 in Fig. 6

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 in the smaller hammock are unchanged-matched, and the similarity threshold is 0.5, `comp` classifies the two hammocks as matched. Therefore, the pairs in N_1 are added to N , `comp` returns `true` (line 9), pair $\langle 2,9 \rangle$ is added to N with label “unchanged” (line 22), and pair $\langle 3,10 \rangle$ is pushed onto ST (line 27).

Pair $\langle 3,10 \rangle$ is then matched and pair $\langle 4,11 \rangle$ is pushed onto ST and then compared in the next iteration. `HmMatch` continues comparing and matching the rest of the graphs in an analogous way. The part of the example shown so far already illustrates 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 `CalcDiff` 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 versions 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 $O(k \cdot \min(d, l) \cdot \min(m, n))$.

3 Studies

To evaluate our algorithm, we implemented a tool, `JDIFF`, and performed four studies on two real, medium-sized programs. 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. The differencing tool calls the graph-building tool to build an ECFG for each method. To build ECFGs, the tool leverages the capabilities of `JABA` (Java Architecture for Bytecode Analysis),⁵ a

⁵ <http://www.cc.gatech.edu/aristotle/Tools/jaba.html>

Java-analysis front-end developed within our research group. We are currently using JDIFF in our impact analysis and regression testing tools (Apiwattanapong et al., 2005; Orso et al., 2003).

We used two subjects for our studies: DAIKON and JABA. DAIKON is a tool for discovering likely program invariants developed by Ernst and colleagues (Ernst, 2000), whereas JABA is the analysis tool described above. To evaluate the effectiveness of our algorithm, we ran JDIFF on 39 pairs of consecutive versions of DAIKON ($\langle v1, v2 \rangle$ to $\langle v39, v40 \rangle$) and on seven pairs of consecutive versions of JABA ($\langle v1, v2 \rangle$ to $\langle v7, v8 \rangle$). All versions are real versions that we extracted from the CVS repositories for the two subjects.

The time interval between two versions of DAIKON is one week (except for weeks in which there were no changes in the software; in these cases, we extended the interval one week at a time until the new version included modifications). The versions of JABA can be divided into two groups: $v1$ – $v4$ and $v5$ – $v8$. Versions $v1$ to $v4$ correspond to a period in which JABA was undergoing a major restructuring, and therefore contain a greater number of changes than versions $v5$ to $v8$. Also, whereas the time interval between consecutive versions for $v1$ – $v4$ and $v5$ – $v8$ is about two weeks, the time interval between $v4$ and $v5$ is about six months.

The size of DAIKON varies widely between versions. DAIKON's size constantly increases from one version to the next, ranging from 357 classes, 2,878 methods, and approximately 48 KLOC to 755 classes, 7,112 methods, and approximately 123 KLOC. The size of JABA varies little between versions. All versions consist of about 550 classes, 2,800 methods, and 60 KLOC.

We run all studies on a Sun Fire v480 server equipped with four 900 MHz UltraSparc-III processors and 16 GB of memory. Each run used only one processor and a maximum heap size of 3.5 GB.

3.2 Study 1

One of the main advantages of using our technique, instead of traditional text-based differencing approaches, is that our technique can account for the effect of changes due to object-oriented features. (For simplicity, in the following, we refer to such changes as *object-oriented changes*.) The goal of Study 1 is to assess the usefulness of our technique by investigating how often object-oriented changes occur in practice. For the study, we used the 39 pairs of consecutive versions of DAIKON and the seven pairs of consecutive versions of JABA. First, we ran `Jdiff` on each pair of versions $\langle P_i, P_{i+1} \rangle$, where $1 \leq i \leq 39$ for DAIKON and $1 \leq i \leq 7$ for JABA. Then, we analyzed `Jdiff`'s output and determined how many changes were object-oriented changes.

Table 1 presents the results of this study. The upper part shows the number of object-oriented changes for DAIKON, and the lower part shows the number of object-oriented changes for JABA. The table presents, in separate columns and for each pair of versions, the number of occurrences of two types of object-oriented changes: changes in dynamic binding (Binding) and changes in types of local variables and fields (Type). (The other two types of object-oriented changes discussed in Section 2.4.1 occur rarely in our subject programs and versions: there are only three changes related to synchronized blocks and no changes in the exception class hierarchy.) A pair of

Table 1 Number of actual changes (AC) and number of statements indirectly affected (IA) for each kind of object oriented changes in each pair of versions of DAIKON and JABA

Pair	Binding		Type	
	AC	IA	AC	IA
DAIKON				
v_1, v_2	26	151	14	31
v_2, v_3				
v_3, v_4			17	54
v_4, v_5			1	2
v_5, v_6			2	8
v_6, v_7	4	4	19	34
v_7, v_8	3	48		
v_8, v_9	5	14		
v_9, v_{10}	2	2		
v_{10}, v_{11}	8	14		
v_{11}, v_{12}	4	5	6	53
v_{12}, v_{13}	2	3	4	10
v_{13}, v_{14}				
v_{14}, v_{15}	3	4		
v_{15}, v_{16}			1	1
v_{16}, v_{17}				
v_{17}, v_{18}	6	8		
v_{18}, v_{19}	21	90		
v_{19}, v_{20}	5	6		
v_{20}, v_{21}	17	19		
v_{21}, v_{22}	11	15		
v_{22}, v_{23}	29	90	3	4
v_{23}, v_{24}	7	45		
v_{24}, v_{25}	13	187	1	1
v_{25}, v_{26}	38	230	2	6
v_{26}, v_{27}				
v_{27}, v_{28}				
v_{28}, v_{29}				
v_{29}, v_{30}				
v_{30}, v_{31}				
v_{31}, v_{32}				
v_{32}, v_{33}				
v_{33}, v_{34}	19	130	10	16
v_{34}, v_{35}	8	14		
v_{35}, v_{36}				
v_{36}, v_{37}	6	14		
v_{37}, v_{38}	7	54		
v_{38}, v_{39}				
v_{39}, v_{40}	14	22	10	30
JABA				
v_1, v_2	2	2		
v_2, v_3	1	1		
v_3, v_4	7	12	1	1
v_4, v_5	12	313	73	289
v_5, v_6	14	21		
v_6, v_7				
v_7, v_8			1	4

columns under each type show the number of actual (i.e., syntactic) changes (AC) and the number of statements indirectly affected by such changes (IA).

For changes in dynamic binding, the actual changes are additions or deletions of methods, changes in method from non-static to static or vice versa, and changes of method access modifiers. The statements indirectly affected by such changes are all the method calls that may be bound to different methods as a consequence of the change. For changes in types, the actual changes consist of changes in declarations of local variables and fields, and the affected statements are all the statements in which such variables and fields are referenced. For example, the changes for pair $\langle v1, v2 \rangle$ of DAIKON consist of modifications to 26 methods, which may cause 151 method calls to behave differently, and modifications to the types of 14 local variables and fields, which may affect the behavior of 31 statements.

The indirectly-affected statements represent the additional information provided by our technique that other differencing approaches, such as `diff`, could not identify (without further analysis). The results of this study show that object-oriented changes occur in more than half of the cases considered. In a few cases, object-oriented changes may result in more than 100 indirectly-affected statements, that is, statements in the code that may behave differently and that traditional differencing approaches would not identify.

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 . Also in this case, we used as subjects of the study all versions of JABA and DAIKON. For each pair of versions, we ran JDIFF with different values for LH and S , and measured the running times. Because the goal of the study is to assess the efficiency of our differencing technique, the running times we consider do not include the time required by JABA to perform the underlying control- and data-flow analysis. (This time ranges from 150 to 586 s for DAIKON and from 279 to 495 for JABA.)

Figures 8 and 9 show the running time (in seconds) of JDIFF. For DAIKON, because there are 39 pairs of versions, we report the running time of JDIFF using box-and-whisker plots. Figures 8(a)–(c) show box-and-whisker plots of the running times obtained for thresholds of 0.0, 0.6, and 1.0, respectively. In the diagrams, the horizontal axis represents the value of LH , and the vertical axis represents the running time. In a box-and-whisker plot, the ends of the box are the upper and lower quartiles; the median is marked by the horizontal line inside the box; the whiskers are the two lines outside the box that extends to the highest and lowest observations that are not outliers; and the outliers are marked by star symbols above and below the whiskers. For example, when running JDIFF with $LH = 10$ and $S = 1.0$, the plot divides the 39 runs (one per pair of versions) into four groups of about 10 runs each. The first group, represented by the whisker below the box, shows that the 10 runs in this group took between 90 and 120 s to complete. The second group, represented by the lower half of the box, shows that the runs in this group took between 120 and 135 s. The line inside the box shows the median, which is about 135 s. The third group, represented by the upper half of the box, shows that the running time for the 10 runs in this group ranges between 135 and 190 s. Finally, the last group, represented by the whisker above the

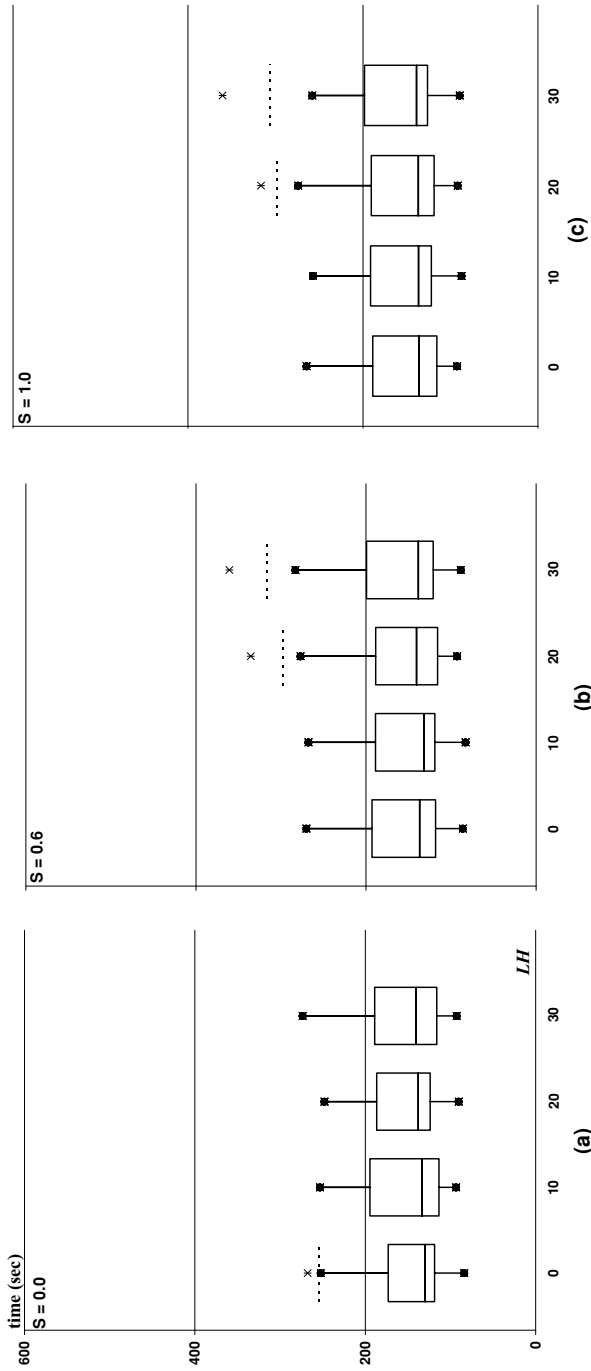


Fig. 8 Average time (sec) for various pairs of versions of DAIKON, lookaheads, and similarity thresholds

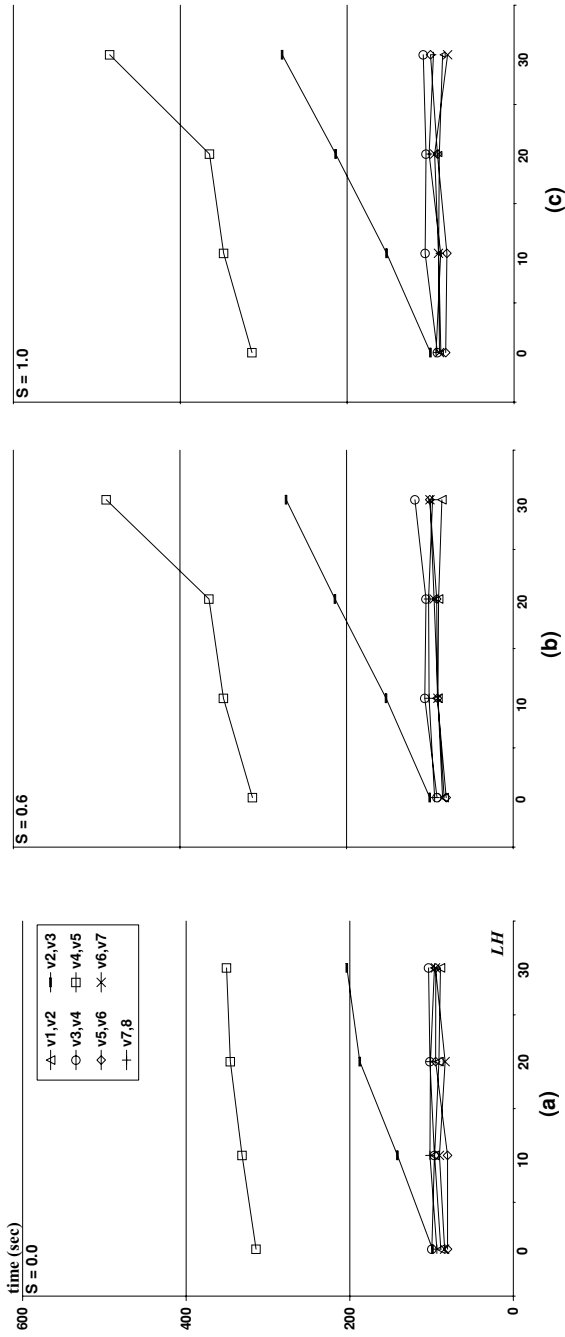


Fig. 9 Average time (sec) for various pairs of versions of JABA, lookaheads, and similarity thresholds

box, shows that the running time for the 10 runs in this group ranges between 190 and 250 s. For JABA, there are only seven pairs of versions; therefore, in Fig. 9(a)–(c), we show the running time of JDIFF on each individual pair for similarity thresholds of 0.0, 0.6, and 1.0, respectively. For example, JDIFF took about 200 s to compute the differences between two versions of JABA with $S = 0$ and $LH = 30$ (see Fig. 9(a)).

Our results show that, when LH is kept constant, the value of S affects only slightly the performance of JDIFF. 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 $S = 0.6$ and $S = 1.0$, where a hammock may be compared more than once. For $S = 0.6$ and $S = 1.0$, 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 increase when the lookahead increases. In all cases considered, JDIFF took less than six minutes to compute the differences between a pair of versions (less than 15 min if we also consider JABA's analysis). Note that our tool is still a prototype, so we expect even better performance on an optimized version of the tool.

3.4 Study 3

The goal of Study 3 is to evaluate the effectiveness of our algorithm compared to Laski and Szermer's algorithm (Laski and Szermer, 1992). To this end, 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. Note that in their paper (Laski and Szermer, 1992), the handling of some specific cases is not discussed. 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 all versions of DAIKON and JABA and counted the number of nodes in the sets of added, deleted, modified, and unchanged nodes. We ran our algorithm several times, using different values of lookahead LH and similarity threshold S . Note that, in the study, we considered only nodes in modified methods because added, deleted, and unchanged methods do not show differences in matching capability between the two algorithms.

To compare the effectiveness of the two algorithms, we compute the number of nodes that are matched by JDIFF but that are not matched by LS. We express this number as a percentage over the total number of nodes identified as unmatched by LS. Intuitively, this percentage represents the relative improvement, in terms of matching, achieved by our algorithm over LS. In our preliminary studies, we found

that the number of nodes identified as unmatched by LS varies between 745 and 46,745 for DAIKON and between 116 and 40,024 for JABA.

Figures 10 and 11 present the results of this study. Figures 10(a)–(c) present the results for DAIKON, summarized across all 39 pairs of versions, for similarity thresholds of 0.0, 0.6, and 1.0, respectively. Figures 11(a)–(c) present the results for each considered pair of versions of JABA for similarity thresholds of 0.0, 0.6, and 1.0, respectively. The horizontal axis of each graph represents the value of LH , and the vertical axis of each graph shows the percentage described earlier. For example, for $S = 1.0$ and $LH = 20$, our algorithm matches on average about 10% more nodes than LS, when both algorithms run on the same pair of versions of DAIKON (see Fig. 10(c)). For another example, for $S = 0$, and $LH = 20$, our algorithm matches about 45% more nodes than LS for JABA's pair of versions $\langle v5, v6 \rangle$. In this case, our algorithm matches 1,319 additional nodes out of 3,233 nodes that LS identifies as unmatched nodes.

The results of this study show that our algorithms performs better than LS in almost all cases in which LH is not 0. For $LH = 0$, the fact that LS performs better than our algorithm confirms our intuition. Without lookahead, our algorithm does not search further to find a match when two hammocks do not match, whereas LS matches hammocks and nodes by searching the graphs until the hammock exit node is found. It is worth noting that the LS approach has a higher cost than our algorithm. In fact, as we verified in our studies, LS always takes longer than our algorithm to compare two versions.

We further analyzed the data and found that the few cases in which our algorithm, with $LH \geq 10$, performs worse than LS are special cases in which: (1) there are very few changes between the considered versions; and (2) the headers of the hammocks that contain changes are the same, which favors LS's hammock-matching strategy.

In all other cases considered (i.e., $LH \neq 0$ and all pairs of versions except JABA $\langle v2, v3 \rangle$), our algorithm matches more nodes than LS, and we obtain improvements in matching up to 10% on average for DAIKON and up to 90% for JABA. The results also show that the number of matched nodes increases when LH increases, which confirms our intuition. Finally, the results show that, for $LH > 10$, the number of matched nodes is slightly greater in the case of $S = 0.6$ and $S = 1.0$ than in the case of $S = 0$.

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 in the second category only. In this sense, the percent improvement that we measured in our study is a lower bound for the actual improvement (i.e., it underestimates the actual improvement).

3.5 Study 4

The goal of Study 4 is to assess the effectiveness of our algorithm for a particular task—coverage estimation, a specific case of program-profile estimation. As stated in the Introduction, program-profile estimation can be used to approximate coverage or profile information when this information cannot be reproduced (e.g., in the case of coverage or profile information collected in the field). Coverage estimation could

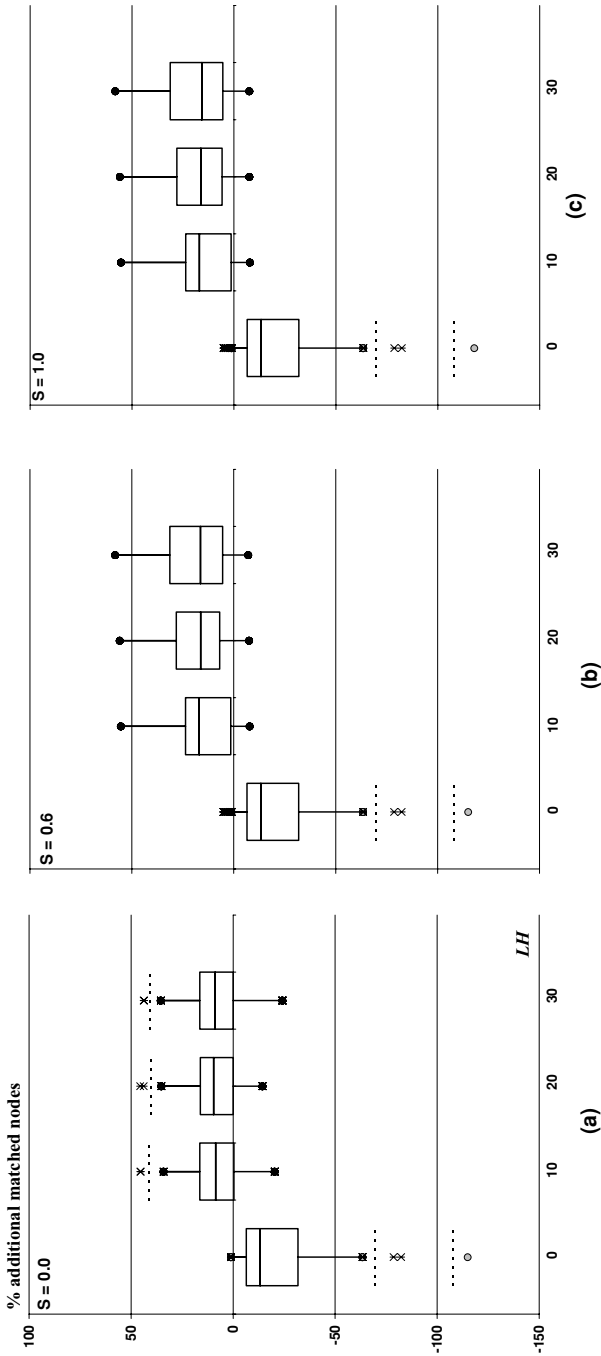


Fig. 10 Percentage of the number of nodes identified as matched by JDIFF but as unmatched by LS in DAIKON

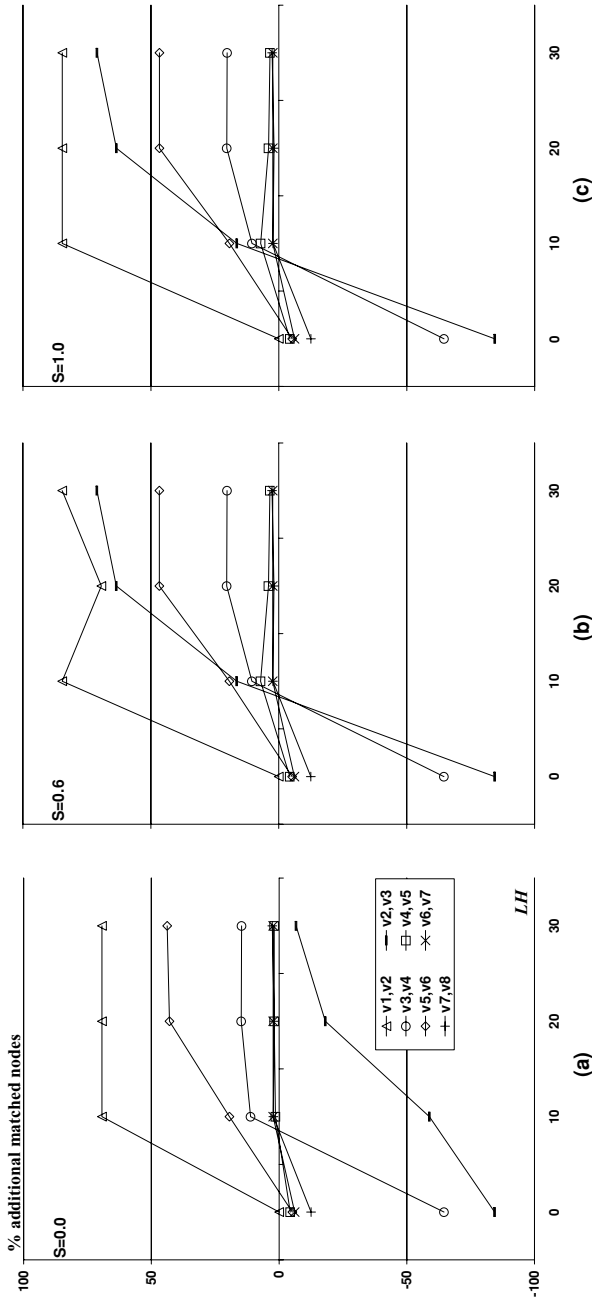


Fig. 11 Percentage of the number of nodes identified as matched by JDIFF but as unmatched by LS in JABA

also be used to migrate coverage or profiling information between versions in cases in which re-running the entire test suite takes a long time.

For the discussion, let P_i represent version i of the program and T_i represent a test suite used for testing P_i . The coverage information for P_i , C_{P_i} , consists of the entities exercised by each test case $t \in T_i$. Coverage estimation estimates $C_{P_{i+1}}$ from C_{P_i} and the correspondences between entities in P_i and P_{i+1} . In this study, we consider a node in the ECFG as an entity in the program. Therefore, the coverage information is expressed as a set of nodes that each test case t exercises. Also, to investigate how the size and number of changes affect the estimation results, we estimated the coverage for each P_i , with $i > 1$, using C_{P_1} (instead of $C_{P_{i-1}}$). In this way, the number of changes is constantly growing while i increases.

For each program P , we proceeded as follows. We first ran the test suite T_1 on version P_1 to collect C_{P_1} . Second, for each $\langle P_1, P_i \rangle$ pair, where $i > 1$, 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 $\langle P_1, P_i \rangle$ 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 corresponding 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 as q).

Finally, we compared the estimated and the actual coverage, C_{P_i} , where $i > 1$, and recorded the number of nodes whose coverage is correctly estimated.

For this study, we used our two subjects, DAIKON and JABA, but we used different pairs of versions than the ones used in the other studies. We used different pairs of versions mostly because we needed to have the same set of test cases for both versions in a pair, to be able to compare the coverage. Therefore, versions that have different test suites from the preceding and following versions were removed.

For DAIKON, we considered 35 of the 40 versions considered for the other studies. We used the Junit test cases extracted from DAIKON’s CVS repository as our test suite. The number of test cases in the test suite varies from 40 to 62 test cases and covers approximately 18% of the program. (DAIKON also has a set of system test cases, which cover a larger part of the program, but we were not successful in running such test cases on many of the versions considered.)

To be able to compare the actual and estimated coverage, we grouped together versions with the same test suite and obtained six groups. As explained above, to investigate how the size and number of changes affect the estimation results, we paired each version with the same base version instead of pairing consecutive versions, as we did for the other studies. Therefore, for each group, we paired each version with the first version of the group. Using this method, we produced 29 pairs of versions. The base versions for the six groups are v2, v7, v12, v15, v21, and v34. The size of the change sets between versions varies from 49 to 207 methods. Due to the low coverage of the test suite, we included in the results only parts of the program that

Table 2 Coverage-estimation results for the six sets of versions of DAIKON

Pair	Number of nodes	Covered	%Corr.Est./ Test suite	Pair	Number of nodes	Covered	%Corr.est./ Test suite
v2,v3	10,406	9,810	93.97	v21,v26	21,853	14,564	65.05
v2,v4	10,519	9,935	93.00	v21,v27	21,877	14,588	65.00
v2,v5	10,547	9,961	92.52	v21,v28	21,882	14,588	64.98
v7,v8	10,775	10,773	97.90	v21,v29	21,882	14,588	64.98
v7,v9	10,319	10,304	97.23	v21,v30	21,882	14,588	64.98
v12,v13	12,601	12,599	99.70	v21,v31	21,875	14,585	64.98
v12,v14	12,601	12,599	99.70	v21,v32	21,875	14,585	64.98
v15,v16	13,301	13,298	95.24	v21,v33	21,885	14,591	65.01
v15,v17	13,304	13,295	95.00	v34,v35	14,786	14,786	100.00
v15,v18	13,418	13,290	93.35	v34,v36	14,786	14,786	100.00
v15,v19	13,462	13,322	91.51	v34,v37	14,790	14,790	100.00
v21,v22	21,644	14,468	66.45	v34,v38	14,806	14,805	99.93
v21,v23	21,666	14,359	65.67	v34,v39	14,806	14,805	99.93
v21,v24	21,639	14,360	65.71	v34,v40	14,815	14,811	99.82
v21,v25	21,790	14,527	65.52				

are actually covered or estimated to be covered. Considering parts of the program that have never been exercised, and that will therefore always be estimated correctly, would considerably skew the results in our favor.

Table 2 shows, for each pair of versions, the number of nodes in the parts of the modified version that we consider (Number of Nodes), the number of nodes that are actually covered by the test suite (Covered), and the percentage of the number of nodes whose coverage of the test suite is correctly estimated (%Corr.Est./Test Suite).

The results show that the coverage estimation performs well in most cases and, as expected, slightly degrades as the change set becomes larger. For the group of versions v21–v33, for which the coverage estimate is below 70%, we further investigated the causes of the lower accuracy in the estimate. We found that one test case that succeeds in v21 fails immediately in all other versions. Because this test case covers a large part of DAIKON (relatively to the coverage of the whole test suite), its failure for versions v22–v33 changes considerably the coverage between those versions and version v21. For all other groups, the coverage estimate is always over 90%, and it is almost always perfect (100%) for versions v34–v40.

For JABA, we performed the study in two parts, by considering versions v1–v4 and v5–v8 separately. In this way, we were able to investigate the effectiveness of our approach, when used to support coverage estimation, for sets of changes of different sizes (see Section 3.1). We first considered versions v5–v8, which contain smaller sets of changes than v1–v4. Also in this case, we paired each version with the same base version. The size of the change sets between versions in the pairs ranges from a few to approximately 20 modified methods. As a test suite for these versions of JABA, we used its actual regression test suite T , which was developed over the years. The test suite contains 707 test cases and covers approximately 60% of the program.

Table 3 shows, for each pair, 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 (Avg. Covered), the percentage of number of nodes whose coverage of each test

Table 3 Coverage-estimation results for the first set of JABA

Pair	Number of nodes	Avg. covered	%Avg.Corr.Est./ Test case	%Corr.Est./ Test suite
v5,v6	22,026	12,592	98.57	97.17
v5,v7	22,063	12,621	98.46	97.17
v5,v8	22,315	12,661	98.03	96.20

Table 4 Coverage-estimation results for the second set of JABA

Pair	Number of nodes	Avg. covered	%Avg.Corr.Est./ Test case	%Corr.Est./ Test suite
v1,v2	19,639	12,200	96.25	95.30
v1,v3	20,076	12,344	86.08	80.97
v1,v4	20,258	12,316	84.70	77.81

case $t \in T$ is correctly estimated, averaged over all test cases (%Avg.Corr.Est./Test Case), and the percentage of the number of nodes whose coverage of the entire test suite is correctly estimated (%Corr.Est./Test Suite). For example, $v7$ contains 22,063 nodes, and a single test case exercised about 12,621 nodes on average. For this set of nodes, on average, 98.46% of the nodes' coverage for a single test case and 97.17% of the nodes' coverage for the entire test suite are correctly estimated. The results show that estimated coverage is high for the pairs we studied and, as for DAIKON, slightly degrades as the change set becomes larger (i.e., when considering pairs $\langle v1, v3 \rangle$ and $\langle v1, v4 \rangle$).

To investigate the effectiveness of our algorithm, when used for coverage estimation, in the presence of a larger number of changes, we then considered versions $v1$ – $v4$. (As stated in Section 3.1, those versions correspond to a period of major restructuring of JABA's code.) Also in this case, we constructed pairs of versions using the first version, $v1$, as the first element of each pair. The sizes of the change sets in these pairs are 15 methods for $\langle v1, v2 \rangle$, 100 methods for $\langle v1, v3 \rangle$, and 150 methods for $\langle v1, v4 \rangle$. The test suite for these versions is an earlier version of the one used for versions $v5$ – $v8$; it consists of 155 test cases and covers approximately 45% of the program.

Table 4 shows the results of the study for this set of pairs of versions, in the same format as Table 3. The results show that, also for this set of versions, the quality of the estimated coverage degrades when the change sets become larger. However, for the largest change set, our algorithm still allows for computing 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.

Overall, the results of Study 4 show that our algorithm provides information that allows for an effective coverage estimation. Although the estimate degrades when the set of changes increases, the results are in most cases over 80% correct (and, in many cases, over 90% correct). Moreover, degradation of results is not a major concern because we expect that the coverage information could be easily “resync-ed” when free cycles are available. Our results are consistent with the findings in Elbaum et al. (2001).

3.6 Threats to validity

There are several threats to the validity of our studies. An external threat exists because we conducted the studies using only two subject programs. Thus, we cannot claim generality for our results. However, the subject programs are real programs, that are used on a regular basis by several research groups, and the changes considered are real changes that include bug fixes and feature enhancements. Another external threat to validity exists for the fourth study: we used only one test suite for each subject 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.

4 Related work

There are a number of existing techniques for computing differences between two versions of a program that are related to ours. These techniques vary in terms of the levels of granularity at which they operate, the kinds of program representation they use, the differencing algorithms they utilize, and the applications for which the differencing results are intended.

One of the most general differencing tools is the UNIX `diff` utility (Myers, 1986). As discussed in the Introduction, `diff` compares two text files line by line and outputs differences in the files. Because `diff` computes purely-textual differences, it cannot identify changes in program behavior indirectly caused by textual modifications. Moreover, `diff` may report changes that have no effects on programs' behavior, such as reordering of class members and changes in comments. Because our differencing technique suitably models the Java programming-language constructs, it does not suffer from these limitations.

Maletic and Collard's approach (Maletic and Collard, 2004) transforms C/C++ source files into a format called srcML and leverages `diff` to compare the srcML representations for the old and new versions of the source code. (srcML is an XML-based format that represents the source code annotated with syntactic information.) The results of the comparison are then post-processed to create a new XML document, also in srcML format, with the additional XML tags that indicate the common, inserted, and deleted XML elements. Their approach takes advantage of available XML tools to ease the process of extracting change-related information. However, it is limited by the fact that it still relies on line-based differencing information obtained from `diff`.

Several modern integrated development environments, such as Eclipse (the Eclipse Foundation, 2001), incorporate a parser for the programming languages they support. Therefore, they can compare different versions of a program more effectively than tools based on simple textual comparison. However, the comparison capabilities of these tools are still limited, in that they only recognize changes at the purely-syntactic level. For example, they cannot identify indirect differences at the statement level due to object-oriented changes, as our approach does.

Laski and Szermer present an algorithm that computes program differences by analyzing the control-flow graphs of the original and modified versions of a program (Laski and Szermer, 1992). Their algorithm localizes program changes into clusters, which are single entry, single exit program fragments. Clusters are reduced to single nodes in the two graphs and are then recursively expanded and matched. As we discussed in Section 2.4.3, our algorithm is based on Laski and Szermer's algorithm. However, we made several modifications to the original algorithm to improve its matching capability (e.g., we match hammocks at different nesting levels and use hammock's similarity metrics and thresholds when comparing hammocks). In Study 3, we show how our algorithm outperforms, for the cases considered, Laski and Szermer's approach in terms of effectiveness in matching two programs.

BMAT (Binary MATching tool) (Wang et al., 2000) performs matching on both code and data blocks between two versions of a program in binary format and is similar in spirit to our approach. 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 (Jackson and Ladd, 1994) compares two versions of a program procedure by procedure, computes a set of input-output dependencies for each procedure, and identifies the differences between two sets computed for the same procedure in the original and modified programs. *Semantic diff* is performed only at the procedure level and may miss changes that, although not affecting input-output dependencies, may have inter-procedural side effects. Moreover, input-output dependencies are typically expensive to compute, so the approach is likely to have scalability issues when applied to medium and large programs.

Horwitz's approach (Horwitz, 1990) computes both syntactic and semantic differences between two programs using a partitioning algorithm. The approach models programs using a Program Representation Graph (PRG), a representation defined only for programs written in a language with scalar variables, assignment statements, conditional statements, while loops, and output statements. Because of this limitation in its underlying representation, Horwitz's approach cannot be applied to programs written using traditional languages, such as C, C++, or Java. In particular, the approach cannot be applied to object-oriented programs.

The approaches that we discuss next are related to our technique because they compute differencing information by comparing two versions of a program. However, they compare programs for a specific goal and, thus, produce differencing information that is targeted to such goal. Therefore, they are not directly comparable with our technique. A detailed comparison, although potentially interesting, is outside the scope of this paper.

Binkley's approach (Binkley, 1992) computes semantic differences between two program versions. The approach first identifies, on a System Dependence Graph (SDG), unmatched nodes and nodes with different incoming data- or control-flow edges in the two versions of the program considered. Then, it performs forward slicing starting from such nodes to identify all affected nodes in the graph. The goal of Binkley's approach is to reduce the cost of regression testing, so it produces different information than our technique.

Raghavan and colleagues present a differencing tool called DEX (Raghavan et al., 2004). DEX compares two abstract semantic graphs (i.e., abstract syntax trees augmented with extra edges that encode type information) that represent two versions of a program. At the end of the comparison, DEX produces an edit script that contains the transformations necessary to transform the semantic graph for the old program into the semantic graph for new one. The tool also collects change-related statistics to reveal some facts about the nature of bug fixes in software projects.

Ren and colleagues present CHIANTI, an impact analysis tool that uses a differencing engine to identify atomic changes between two versions of a Java program and dependences among these changes (Ren et al., 2004). (The authors also present an interesting study of the type of changes that occur in object-oriented systems during maintenance. Their results are similar to the ones we discuss in Section 3.2 and confirm that changes in such systems often include changes that involve object-oriented constructs and whose subtle effects must be suitably identified.) Rangarajan also uses the notion of atomic changes at the class and method levels and presents a tool called JEVOLVE (Rangarajan, 2001) that analyzes Java programs and identifies modified classes that must be regression tested. There are two main differences between the CHIANTI and JEVOLVE tools and JDIFF. First, their differencing operates at the method level, whereas we provide differencing information at the statement level. Second, the goal of their techniques is to provide differencing information to identify which test cases are affected by the changes between two versions. (CHIANTI also identifies which changes affect which test cases.)

Recently, Xing and Stroulia presented UMLDIFF (Xing and Stroulia, 2005), an algorithm for automatically detecting structural changes between the designs of subsequent versions of an object-oriented program. UMLDIFF compares the class diagrams of two program versions and uses structural information to identify the differences between the two versions. Differences are identified in terms of addition, removal, moving, and renaming of packages, classes, interfaces, fields, and methods. Unlike JDIFF, UMLDIFF works at the class and method levels and does not compare statements in matched method pairs. Furthermore, UMLDIFF is mostly targeted at identifying design-level changes.

Finally, there are three areas of research that, although not directly related to this work, rely on program differencing: software merging, clone detection, and data mining of software repositories. We briefly summarize the three areas and their relation with the work presented in this paper.

Because differencing approaches are used in the context of software merging (Mens, 2002), they often incorporate structure-based differencing techniques, such as LTDIFF (used in ELAM (Hunt and Tichy, 2002)), CDIFF (Grass, 1992), and Yang's approach (Yang, 1991). These approaches are based on variants of tree-differencing algorithms that operate on programs' parse trees and have some advantages over purely text-based approaches, such as `diff`. However, these approaches are still limited when used on object-oriented code (e.g., they cannot identify the differences in behavior caused by dynamic binding). According to the categorization presented in a recent survey on program-differencing approaches (Mens, 2002), JDIFF is a semantic, state-based approach because it captures the changes in program behavior and uses only the information in the original and modified versions of the program.

Approaches for data mining of software repositories (e.g., Fischer et al., 2003; German, 2004)) aim to find differences between several versions of a program. These approaches usually work at the file level and use textual differencing to quantify differences in terms of the number of lines changed between files. They typically collect and aggregate information on many versions of the source code extracted from a software repository, including annotations submitted by the authors when the program changes were committed. This data may allow to infer relevant information about a software project, such as error-proneness of different areas of the code and coupling between files.

Clone detection techniques, just like program differencing techniques, need source-code comparison capabilities. However, these two families of techniques differ in their goals. Program differencing compares entities in one version with corresponding entities in another version of the program to identify differences in behavior between them. Conversely, clone detection compares a fragment of code in one method with fragments of code in other methods in the same version to find similarities and duplications.

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 Java language. Given two programs, our algorithm identifies matching classes and methods, builds our representation for each pair of matching methods, and compares the representations for the two methods to identify similarities and differences between them. The results of our differencing can be used for various development and maintenance tasks, such as impact analysis and regression testing. Although our technique was defined for Java, it should be generally applicable or adaptable to any object-oriented language that has the same features of Java, such as C#.

We also presented a tool that implements our technique, JDIFF, and a set of studies that show the effectiveness and efficiency of the technique. Study 1 provides evidence that the types of changes targeted by our technique do occur in practice and can affect many statements in a program. Study 2 illustrates the efficiency of the technique for different execution parameters. Study 3 compares our technique to the most closely related program-differencing approach and shows that our technique achieves improvements from 17% to over 65% in terms of matching unchanged parts of the code. Study 4 shows how our differencing technique can successfully support coverage estimation.

Based on the empirical studies and our experience with the tool, we believe that JDIFF provides more useful information on the changes between two versions of an object-oriented program than purely-textual differencing and other previously-presented approaches. For example, although `diff` provides an efficient way of getting an overall idea of the changes between two program versions, often its results cannot be directly used to support software maintenance tasks. Conversely, JDIFF computes change information by considering both the program structure and the semantics of the programming-languages constructs. By doing so, JDIFF can provide information that accurately reflects the effects of code changes on the program at the statement level. Such information can support maintenance tasks and be used directly

by other tools. In addition, JDIFF seems to be efficient enough to be used in realistic settings and on programs of at least medium size. (Because we used an unoptimized implementation of the technique for our studies, we believe that the approach could scale to large subjects as well.)

In future work, we will investigate additional heuristics to further improve our 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 systems to assess whether the differencing algorithm could leverage some knowledge of common change patterns to improve its performance. Finally, we will investigate the use of our differencing algorithm to support test suite augmentation—selecting new test cases for a modified system based on the types of changes performed on the system.

Acknowledgments This work was supported in part by Tata Consultancy Services, National Science Foundation awards CCR-0306372, CCR-0205422, CCR-9988294, CCR-0209322, and SBE-0123532 to Georgia Tech, and by the State of Georgia to Georgia Tech under the Yamacraw Mission. Kathy Repine implemented a preliminary version of JDIFF.

References

- Apiwattanapong, T., Orso, A., Harrold, M.J.: A differencing algorithm for object-oriented programs. In: Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE 2004), pp. 2–13 (2004)
- Apiwattanapong, T., Orso, A., Harrold, M.J.: Efficient and precise dynamic impact analysis using execute-after sequences. In: Proceedings of the 27th IEEE and ACM SIGSOFT International Conference on Software Engineering (ICSE 2005), pp. 432–441 (2005)
- Binkley, D.: Using semantic differencing to reduce the cost of regression testing. In: Proceedings of IEEE Conference on Software Maintenance, pp. 41–50 (1992)
- Bohner, S.A., Arnold, R.S.: Software Change Impact Analysis. IEEE Press, Los Alamitos, CA, USA (1996)
- Elbaum, S., Gable, D., Rothermel, G.: The impact of software evolution on code coverage information. In: Proceedings of the International Conference on Software Maintenance, pp. 169–179 (2001)
- Ernst, M.D.: Dynamically discovering likely program invariants. Ph.D., University of Washington Department of Computer Science and Engineering, Seattle, Washington (Aug. 2000)
- Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. 9(3), 319–349 (1987)
- Fischer, M., Pinzger, M., Gall, H.: Populating a release history database from version control and bug tracking systems. In: Proceedings of the 19th IEEE International Conference on Software Maintenance (ICSM 2003), pp. 23–32 (2003)
- German, D.M.: Mining CVS repositories, the softChange experience. In: Proceedings of the First International Workshop on Mining Software Repositories, pp. 17–21 (2004)
- Grass, J.E.: Cdiff: A syntax directed differencer for C++ programs. In: Proceedings of the USENIX C++ Conference, pp. 181–193 (1992)
- Harrold, M.J., Jones, J.A., Li, T., Liang, D., Orso, A., Pennings, M., Sinha, S., Spoon, S.A., Gujarathi, A.: Regression test selection for java software. In: Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 312–326 (2001)
- Horwitz, S.: 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, pp. 234–246 (1990)
- Hunt, J.J., Tichy, W.F.: Extensible language-aware merging. In: Proceedings of the International Conference on Software Maintenance, pp. 511–520 (2002)
- Jackson, D., Ladd, D.A.: Semantic diff: A tool for summarizing the effects of modifications. In: Proceedings of the International Conference on Software Maintenance, pp. 243–252 (1994)

- Laski, J., Szermer, W.: Identification of program modifications and its applications in software maintenance. In: Proceedings of IEEE Conference on Software Maintenance, pp. 282–290 (1992)
- Law, J., Rothermel, G.: Whole program path-based dynamic impact analysis. In: Proceedings of the 25th International Conference on Software Engineering, pp. 308–318 (2003)
- Leung, H.K.N., White, L.: Insights into regression testing. In: Proceedings of the Conference on Software Maintenance, pp. 60–69. IEEE Computer Society Press (1989)
- Maletic, J.I., Collard, M.L.: Supporting source code difference analysis. In: Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM 2004), pp. 210–219 (2004)
- Mens, T.: A state-of-the-art survey on software merging. *IEEE Trans. Softw. Eng.* **28**(5), 449–462 (2002)
- Myers, E.W.: An O(ND) Difference algorithm an its variations. *Algorithmica* **1**(2), 251–266 (1986)
- Orso, A., Apiwattanapong, T., Harrold, M.J.: 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 In (2003)
- Orso, A., Shi, N., Harrold, M.J.: Scaling regression testing to large software systems. In: Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering In (2004)
- Raghavan, S., Rohana, R., Loen, D., Podgurski, A., Augustine, V.: Dex: A semantic-graph differencing tool for studying changes in large code bases. In: Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM 2004), pp. 188–197 (2004)
- Rangarajan, K.: Automatic analysis of java program evolution and its relevance to regression testing. <http://www.mmsindia.com/IEvolve.html> (2001)
- Ren, X., Shah, F., Tip, F., Ryder, B.G., Chesley, O.: Chianti: A tool for change impact analysis of java programs. In: Proceedings of the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 432–448 (2004)
- Rothermel, G., Harrold, M.J.: Analyzing regression test selection techniques. *IEEE Trans. Softw. Eng.* **22**(8), 529–551 (1996)
- Rothermel, G., Harrold, M.J.: A safe, efficient regressing test selection technique. *ACM Trans. Softw. Eng. Methodol.* **6**(2), 173–210 (1997)
- Sinha, S., Harrold, M.J.: Analysis and testing of programs with exception handling constructs. *IEEE Trans. Softw. Eng.* **26**(9), 849–871 (2000)
- the Eclipse Foundation. <http://www.eclipse.org> (2001)
- Wang, Z., Pierce, K., McFarling, S.: BMAT–A binary matching tool for stale profile propagation. *J. Instruction-Level Parallelism* **2** (2000)
- Xing, Z., Stroulia, E.: UMLDiff: An algorithm for object-oriented design differencing. In: Proceedings of the 20th IEEE/ACM Conference on Automated Software Engineering, pp. 54–65 (2005)
- Yang, W.: Identifying syntactic differences between two programs. *Software–Practice Exp.* **7**, 739–755 (1991)