Change-Based Random Testing with Static Program Slices

Marcelo d'Amorim Juliano Iyoda

Informatics Center
Federal University of Pernambuco, Brazil
{damorim,jmi}@cin.ufpe.br

ABSTRACT

State-space exploration is a popular approach for the automation of test generation. One key issue of state-space exploration is dealing with the size of the state-space. We present *Change-Based Random Testing* (CB), a technique that uses the changes programmers make between two program versions to reduce the search space for random sequence generation. The context of use of CB is that of testing the program incrementally to find regression errors introduced with changes. CB takes as input a program under test and a set of change descriptions and produces a set of test sequences as output. CB builds on a program slicer to automatically produce a list of methods and constructors. A test sequence generator uses this list to build calling contexts for changed methods and constructors. We conducted two kinds of experiments on four open-source subjects of varying sizes: nanoxml 1.5KLOC, health-watcher 3.0KLOC, jmeter 19.9KLOC, and poi 46.1KLOC. We compare CB with *all-at-once* generation (AO), a scenario where the generator takes a list with *all* public methods and constructors as input. We evaluate the difference in coverage and mutation scores between CB and AO. The results indicate that, for larger programs, the suite generated with the slicer covered more basic blocks consistently: the average coverage difference (i.e., coverage of changed members obtained with the CB suite minus that obtained with AO's.) was -5.94 for nanoxml, -0.23 for health-watcher, 20.28 for jmeter, and 29.21 for poi. A positive (resp., negative) value indicates that the suite generated with the slicer covered more (resp., less). The results on mutation scores show that either suite kill mutants that the other could not kill, however the suite from the slicer could kill more mutants consistently.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Experimentation, Verification

Keywords

random testing, dynamic analysis, static analysis, slicing

1. INTRODUCTION

Testing consists of two activities: (i) generating inputs, and (ii) generating classifiers (i.e., oracles) to determine whether the execution of one test input passes or fails. Testing is the dominant approach in industry to assure software quality. Software testing is also *very expensive*. Beizer [11] estimates that total labor resources spent on testing range from 30 to 90%. In a more recent study, Santhanam and Hailpern [33] report that from 50 to 75% of the total cost of a project involves testing and debugging. In 2002 the National Institute of Standards and Technology (NIST) estimated that the cost of the inadequate infrastructure to the American economy was 59.2 billion dollars [28]. Automation can help reduce this cost. This paper proposes a technique to improve test automation.

A test input consists of a sequence of operations. More specifically, we consider in the context of this paper sequential object-oriented programs whose inputs consist of sequences of method calls. A typical white-box test sequence generator takes as input a program under test and a *list of its classes and methods* and generates as output a set of test sequences. The tests these generators produce are *complementary* to manually-written functional tests. They are particularly useful to explore corner-case scenarios (which requirements-driven generation typically fail to capture) and to keep tests in sync with constantly evolving programs (as interface changes often lead to test changes). Recently, several automated test sequence generation techniques have been proposed. They have been effective for crashing programs [12, 14, 20], violating language-specific invariants [31], and exposing application-specific logical errors [22, 30].

We propose *Change-Based Random Testing* (CB), a technique that uses the changes programmers make between two program versions to reduce the search space for random sequence generation. CB takes as input a program under test and a set of change identifiers and produces as output a set of test sequences (i.e., test inputs) to exercise the changes. A change identifier is an identifier of a method or constructor that programmers changed between a pair of program versions. The operational goal of the generator is to construct sequences that exercise the changed parts of the program. The expected scenario of use for change-based testing is that of a tester who wants to construct test sequences that exercise methods and constructors she has changed during some period of work. We are particularly interested in testing changes on parts of the program that interact with others,

i.e., on the *automation of integration testing*.

Important to note is that current test sequence generation techniques explore the application *all-at-once* (AO). In this setting, the user provides the same list of classes and methods across different runs of a test sequence generator. One generation does *not* use information from previous runs. Two sources of inefficiency are possible in this context: (i) the generator explores large portions of the program that have not changed (and must not have changed behavior since last generation), and (ii) the generator may neglect parts that *have changed* and are more likely to contain errors [27]. Conceptually all-at-once exploration works well for small search spaces or when the user constantly updates the input list of methods and classes to only explore changes. For large search spaces all-at-once exploration may *not scale* as the sequence generator considers of equal importance the members from the input list to explore. Typically, such a list includes all public methods and constructors of the program.

CB uses program slicing to improve test sequence generation. Program slicing [38, 29, 9, 37] is a technique – with many applications in software engineering – for simplifying a program with respect to some (slicing) criteria. The reduced program is typically called a slice. The slicer we propose uses a criteria based on program changes. It takes a list of changed members as input and produces a list of members – the slice – as output. We use the term (class) member to denote a method or constructor. One expects the slice to include the changed members plus members that build calling contexts to the changed ones. For example, if an instance method m is part of the slice so should be the public constructors of its declaring class, the ancestors of m in the call graph (i.e., methods that can transitively call m), and methods that assign to a field that m reads. Section 2 discusses the slicer that CB uses in detail. This paper uses the term *test goal* to denote a member whose body has changed across a pair of versions.

Intuitively, one factor that determines the effectiveness of CB is the size of the slice computed from program changes. The greater the difference between the size of the slice and the total number of public members the greater the potential impact of CB: the potential space reduction increases with the increase of this difference. We discuss next the results of a feasibility study to evaluate the magnitude of this difference.

Note on distribution representation. We use boxplot notation to represent a distribution of slice sizes for one group of randomly selected changes. In this notation, the lower hinge of one box indicates the upper bound of the first distribution quartile (i.e., 25% of the population) while the upper hinge is the upper bound of the third quartile. The line across the box defines the second quartile (i.e., the median value). The line below (resp., above) the box indicates the first (resp., fourth) quartile. The circles outside the limits of these vertical lines indicate outliers, when they occur.

Figure 1 illustrates how the size of the slice increases with the number of test goals for a selection of subjects of varying sizes (1.5, 3.0, 19.9, and 46.1 KLOC). For a number n of test goals, varying from 2 to 60 with a 2-units step, we run the slicer 10 times with a different selection of n goals and measure the size of the resulting slice as the fraction of public concrete members in the slice out of the total number of these public members.

This figure shows that CB could produce more gains (i.e., reduce more space) for bigger programs (e.g., jmeter and poi): the average median values of slice size across the distributions are 43.75% for nanoxml, 38.27% for health-watcher, 34.9% for jmeter, and 26.08% for poi. The plot also shows a more gradual increase in the median values for bigger programs: the slopes of the linear regression lines (which do not appear in the plots) built from the median values of each distribution are 0.75 for nanoxml, 0.63 for health-watcher, 0.21 for jmeter, and 0.40 for poi.

Note that the slice size grows slower for jmeter than for poi even though poi is a bigger program. Note also that even for small programs with similar sizes (e.g., nanoxml and health-watcher) the rate of increase varies significantly. These observations highlight that the size of the program is an important factor to determine the size of the slice but there are certainly others. For example, note that jmeter has a sharp increase in the size of the slice with the selection of more than 4 goals and that the variance in the size of the slice is minor, compared to other subjects, from 4 up to 60 input goals.

In summary, this feasibility study shows that the size of the slice computed from changes does not grow too fast for large programs. However, this metric is not sufficient to demonstrate the potential impact of CB for space reduction in general. We noted from our experiments that other factors also affect the slice size in particular and CB. For example, the number of modifications (goals), the timeout (which determines when generation stops), and the program organization (e.g., the number of incoming paths in the call graph to a goal). Section 3 discusses the evaluation of CB in more detail.

The contributions of this paper are as follows:

- The proposal and implementation of a test sequence generation technique based on changes.
- The evaluation of this technique with respect to basic block coverage and mutation scores.

Summary of evaluation

We use basic block coverage and mutation scores to evaluate the impact of CB for search space reduction. To that end we use four open-source subjects of different sizes. We run CB with several random selections of test goals using a fixed timeout value. The results indicate that for bigger programs CB covered more basic blocks from the test goals than AO on average: the average difference of coverage value for nanoxml was -5.94, for health-watcher was -0.23, for jmeter was 20.28, and for poi was 29.21. A positive (resp., negative) value indicates that CB (resp., AO) covered more. We also compare coverage of CB and AO on

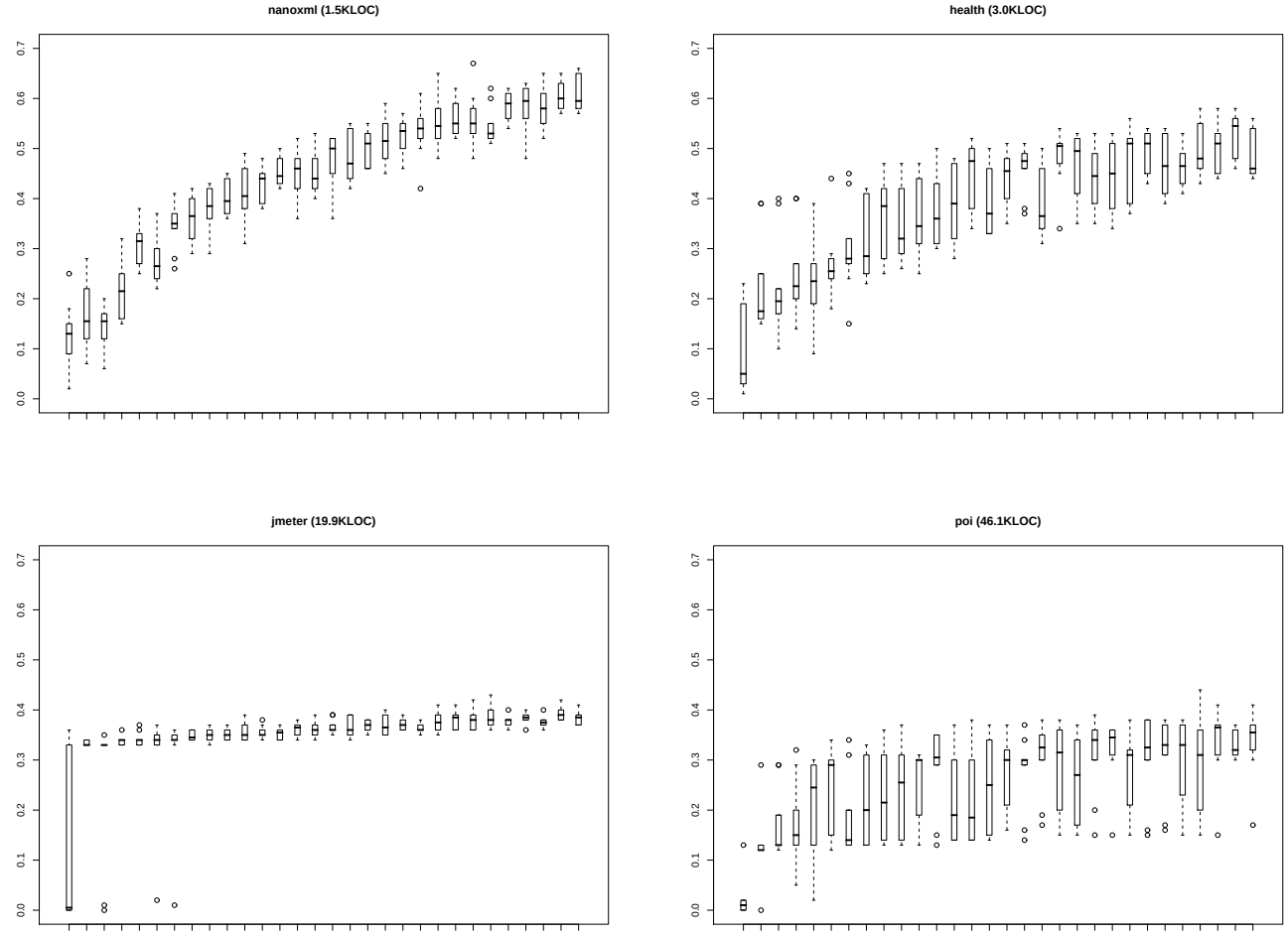

Figure 1: Slice size varies with the number of goals. Horizontal and vertical axis indicate respectively the number of goals and the size of the slice.

4 different random selection of test goals for increasing timeout values that we provide to the sequence generators. The results show that AO can achieve higher coverage than CB for some time bounds. This indicates that the slicer does not conservatively add to the slice all members that can build context to calling test goals. We also compare CB and AO with respect to their coverage on 13 faulty members from the jmeter subject obtained from the SIR repository. The results show that for 7 members CB covers more than AO, and for 6 it covers the same. Finally, we measure the scores CB and AO obtain for killing several mutants manually created. The results show that either CB or AO kill mutants that the other could not kill, however CB could consistently kill more mutants. This indicates that AO and CB are complementary with respect to the capability of finding these injected errors.

2. TECHNIQUE

This section describes *Change-based Random Testing* (CB), a technique for generating random test sequences based on the information of which parts of the program have changed.

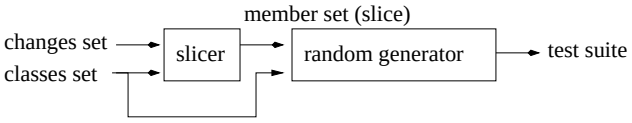

Figure 2: CB inputs and output. CB automates the generation of the member set, input to the random generator.

Slicing + Random generation. A random sequence generator takes as input a set of classes and a set of public members (methods and constructors) and generates random sequences of method calls as output. AO and CB differ in the elements of the member set they provide to the random generator: AO includes in this set all public methods and constructors, while CB attempts to include only members that can build context to calling a method or constructor that has changed. Figure 2 shows the inputs and output of CB. The slicer, appearing to the left of the figure, automates the generation of the member set. One can obtain AO by

replacing the slicer with a component that provides to the generator all public members from the program under test.

We next describe the two components of the technique – the slicer and the random generator, and discuss the main limitations associated to CB.

2.1 Slicing on changes

This section describes the program slicer that CB uses to *automate the generation of its input member set*. Figure 2 shows the inputs and outputs of the slicer.

Ideally, one wants the slicer to generate a minimal member set including all members of the program that can contribute to building input to the test goals. This would maximize reduction without hurting the quality of the random sequence exploration. Unfortunately, finding such ideal set is neither practical (as the slice can grow too fast) nor possible (as we cannot prove in general that one member is useful). We discuss next the main design decisions for the slicer to produce an approximation of this idealized set.

The slicer takes as input the set *Mod* of modified members and computes the set *Sl* denoting the program slice. Any member satisfying the following criteria should participate on the slice *Sl*. The following conditions determine the selection of a member in the slice (consider $s \in Sl$ and $m \in Mod$):

- any public method that can transitively call m
- any public factory for a type that is assignable to one of the formal parameters of s (receiver is also a parameter)
- any public method that can write to a field that s reads

The first condition results in the addition of ancestors of m to the slice. A method is ancestor of another if there is a path connecting one to the other in the call graph. A node in this graph corresponds to a method (resp., constructor) identifier and an edge corresponds to an invocation (resp., allocation). The rationale for this condition is that ancestors can build calling context "for free" to m. It is important to note that the execution of a test including an ancestor of m does not assure a dynamic call to m. More precisely, because of the infeasibility of some program paths [32] the slicer can add methods to the slice that will not necessarily contribute to calling m. For example, consider the void methods `a(int x){ b(x*x);}`, and `b(int x){if (x > 0) c(); else d();}`. Note that despite the fact that a is an ancestor of d, there cannot be a call to d from a since b does not call d on positive values. Note that x^2 is the argument passed to the call to b. This condition results in an over-approximation of the set of ancestors that can actually call m.

We use the term factory to denote any non-void method or constructor of the program under test. In this context, a factory is a member that can build a fresh object. This condition results in the addition to the slice of factories for objects that can be passed as arguments to methods and constructors in the slice: the slicer adds to the slice *all* factories of t', a subtype of t, when t appears as a formal parameter of some slice member. The slicer does not consider primitive types in this case. The rationale for this condition is that we cannot build a sequence without input data to the elements in this sequence. Again, note that the slicer may include members which are not indeed factories. For example, members that only mutate fields and even pure functions. This condition results in an over-approximation of the set of factories.

The third condition results in the addition to the slice of methods that can write to object fields that methods in the slice reads. The rationale for this condition is that the behavior of members in the slice depends on the state they can read. Conceptually, this condition enables the sequence generator to create tests that mutate the state that members in the slice, including the test goals, can read.

It is important to note that the slicer is not able to detect general mutation of state. For example, the slicer identifies that the methods *push* and *pop* from the stack class in Figure 3 should appear together in the slice because they read and write to the same integer field denoting the number of elements in the stack. However, the slicer currently does not detect general data-dependencies on the heap [41]. The slicer applied this constraint to control the size of the slice. The use of this condition results in an under-approximation of the set of state mutators.

```
public class Stack<T> {
  int num; ...
  public void push(T t) { num++; ... }
  public T pop() { num--; ... }
  public T peek() { ... }
}
```

Figure 3: Stack example

```
1: Set<Member> slice(Set<Member> goals) :
2:   Set<Member> result = ∅
3:   Set<Member> wset = {m | m ∈ goals}
4:   while wset ≠ ∅ do
5:    Member entry = wset.remove(0)
6:    if entry ∈ result then continue fi
7:    if entry is-public then
8:      // add entry to slice
9:      result.add(entry)
10:     // generate input directly. add all factories
11:     foreach Type t in entry.paramTypes() do
12:       foreach Member m ∈
13:         {f | f ∈ factories(v) ∧ (v,t) ∈ SUB} do
14:         wset.add(m) done
15:       done
16:     // m writes to a field that entry reads
17:     foreach Member m ∈
18:       {w | w ∈ assigns(f) ∧ f ∈ reads(entry)} do
19:       wset.add(m) done
20:   fi
21:   // generate input indirectly from callers.
22:   foreach Method m ∈
23:     {x | (x,entry) ∈ CG} do
24:     wset.add(m) done
25:  done
26:  return result
```

Figure 4: Chaotic iteration algorithm for building program slice.

2.1.1 Implementation

The slicer of CB uses the ByteCode Engineering Library (BCEL) [15] to build a (i) call-graph (CG), (ii) subtype-relation (SUB), and (iii) field read-write relation. The slicer uses the call graph CG to identify ancestors, the subtype relation SUB to identify useful object factories, and the field read-write relation to identify state mutators (i.e., methods that can mutate state that a member in the slice can read during execution). We use the notation $(a, b) \in G$ to indicate that the graph G contains the vertices a and b and that there is a directed edge connecting them. The pair of functions $reads : Member \rightarrow Field$ and $assigns : Field \rightarrow Member$ model together the field read-write relation. The function *reads* indicates which (possibly non-local) fields a method or constructor can read and function *assigns* indicates which methods and constructors can write to a given field.

Figure 4 shows the pseudo-code of the slicer. It takes a set of goal members as input and produces a set of members (i.e., the slice) as output. Variables *result* and *wset* (lines 2 and 3) denote respectively the state of the slice and a working set of members to process. The slicer incrementally adds new test goals to process and the working set avoids a recursive definition of the function *slice*. Initially, the set *result* is empty and the working set contains only the elements in the input set of members (initial goals). The while loop processes the elements in the working set as a queue and ignores the member when it has already been visited (line 6). Note that as the number of members is finite and each member is processed only once (assuming the containment check is correct), the procedure *slice* must terminate.

The slicer makes the following decisions when the member $entry{=}m(t_1, ..., t_n)$ is public: (i) adds *entry* to the slice (line 9), (ii) adds to *wset* all factories of every subtype of t_i, for each t_i in $t_1, ..., t_n$ (lines 11-15), and (iii) adds to *wset* all members that can assign to a field that *entry* can read (lines 17-19). Regardless the access modifier of *entry*, the slicer also adds all ancestors that can transitively call it (lines 22-24). Conceptually, we want to transfer the obligation from building input to *entry* to building input to one of its ancestors.

2.2 Random generation

CB builds on the feedback-directed random sequence generation algorithm (FD) [31] for generating test sequences. Figure 5 summarizes the FD algorithm. The algorithm takes as input the set *memset* of methods and constructors and reports as output the suite *pass* of passing tests (for regression) and the suite *fail* of failing tests (for debugging).

The pseudo-code *gen* attempts to generate a new sequence in each loop iteration by randomly selecting one constructor or method from *memset*. The call to the external method *choose* randomly selects a member from the set *memset*. Previously generated sequences provide input data to new sequences in the following manner. The method *genSeq* takes as input a constructor or method m and a suite of passing test sequences *pass* and builds a new sequence rooted in m if it can find in *pass* inputs to all parameters of m. The external method *findInput* searches in *pass* for sequences whose executions build value assignable to a variable of type *type*.

```
Set<TestSeq> pass = ∅; Set<TestSeq> fail = ∅
gen(Set<Member> memset) :
  while !stop() do
     TestSeq seq = genSeq(choose(memset), pass)
     if seq == EMPTY then continue fi
     if seq.run() == PASS then pass.add(seq)
     else fail.add(seq) fi
  done
genSeq(Member m, Set<TestSeq> pass) :
  List<TestSeqs> seqArgs
  foreach type in m.types() do
    TestSeq tseq = findInput(pass, type)
    if tseq == EMPTY then return tseq fi
    seqArgs.add(tseq)
  done
  return createSequence(m, seqArgs)
```

Figure 5: Feedback-directed random sequence generation.

The external method *createSequence* creates a new sequence with the construction of a method call to m passing the expressions available at *seqArgs* as parameters.

Note that *gen* effectively implements a test driver with the systematic construction and execution of test sequences. The Randoop [31] implementation of this algorithm combines this driver with programming contractsof the Java language to find general kinds of errors. For example, Randoop uses the contract that the execution of a method should not raise a null pointer exception if the arguments of the method call are non-null. We disable these contract checks in our experiments.

One obtains all-at-once generation by calling method *gen*, from Figure 5, passing as argument to the *memset* parameter a set with *all* public constructors and methods of one subject application.

2.3 Limitations

We list below the main limitations associated to CB.

- **Scope.** In principle, CB will not perform well for testing units of a program or the entire system. The search space associated with a program unit is typically small; CB would not reduce sufficient space to improve exploration. For system testing, the slicer is certainly not as useful too. For example, there are no callers to the main functions of the system within the code (except for existing system tests) that the slicer could use to build calling contexts. For those reasons, CB's scope of application is integration testing.

- **Input Data.** The implementation of CB we described inherits the limitations associated with random testing. CB will likely have difficulties to construct tests that expose bugs which only manifest on very specific input data. For example, a bug in a xml parser that only manifests with a specific contrived xml input, or a bug that only manifests under specific relationships of primitive values passed as arguments to the method calls in a test sequence [20, 16].

```
public class Foo {
  Stack<?> s;
  public Stack<?> getStack() { return s; }
  public void foo() {...getStack().peek();...}
  public void bar(int k) {...getStack().push(k);...}
}
```

Figure 6: Method foo accesses a Stack object.

- **Missing members.** It is possible that the slicer leaves out of the slice members that could help building a calling context to a changed member. Consider, for example, the method `foo` declared in class `Foo` from Figure 6 is in the slice. The slicer does not identify that the call to `peek`, a method that reads the top element from the stack, reads state reachable from the reference to `this`. As such, the slicer is unable to add to the slice the method `bar` that could mutate the state of the stack object they access. This condition results in an under-approximation of the set of state mutators. Section 5 elaborates our plans to evaluate the impact of using contextual def-use associations [35, 17] on the quality of the slices.

3. EVALUATION

This section describes an empirical evaluation of the impact of CB's reduced search space on coverage and mutation scores. We conducted two sets of experiments; one using basic block (statement) coverage and the other using mutation scores. For all experiments, we used an Intel Core2 Duo L7700 machine with 3GB of memory running Ubuntu 8.04. Section 3.1 describes the subjects we used, Section 3.2 the results for coverage, Section 3.3 the results for mutation scores, and Section 3.4 discusses main conclusions.

3.1 Subjects

We used four open-source subjects in our experiments. The list below describes each one. We used JavaNCSS [2] to count non-comment non-blank lines of Java source.

- **nanoxml** [5] is a 1.54 KLOC XML parser for Java. It is part of the Software-artifact Infrastructure Repository (SIR) [7], a repository of software-related artifacts built to support rigorous controlled experimentation with program analysis and software testing techniques.
- **health-watcher** [1] is 2.99 KLOC application for registering public service complaints (e.g., food poisoning) in a local community.
- **jmeter** [3] is a 19.88 KLOC Java application designed for load and performance testing. It is maintained by the Jakarta subproject of the open source Apache project. This subject is also part of SIR.
- **poi** [6] is a 46.10 KLOC pure Java API, from the Apache project, for manipulating various file formats based on Microsoft's format and Office OpenXML format.

3.2 Coverage

We compare CB and AO with respect to the basic-block coverage they obtain, i.e., the fraction of basic blocks from the control-flow graph of a goal method (or constructor) that a test suite visits [32]. We conducted three different experiments. The first one measures the difference of coverage between CB and AO for an increasing number of test goals, using a fixed timeout value. The second measures the difference of coverage for an increasing timeout value, using a fixed number of goals; and finally the third measures the coverage each technique obtains for goals informed from SIR [7].

Impact of number of goals on CB

This section discusses the impact that the increase in the number of goals have in coverage. Intuitively, one expects that minimum (resp., maximum) number of changes (goals) results in maximum (resp., minimum) gain.

The horizontal axis from Figure 7 determine the number of test goals and the vertical axis determine the distribution of coverage difference between CB and AO. A positive (resp., negative) value indicates that CB covers more (resp., less) than AO. For each test goal, ranging from 4 to 60 with a 4-units step, we run each technique for 10 times with different random seeds and compute the difference in the coverage they obtain. It is important to note that we use the same set of seeds to run the experiments reported on figures 1 and 7. Note from Figure 1 that the size of the slice can vary significantly for a different selection of goals (but with same number of them).

The coverage of one execution (with CB or AO) consists of the ratio of the sums $\sum_i c(i) / \sum_i b(i)$, where the index i ranges over the set of goals, $c(i)$ denotes the number of basic blocks of i covered and $b(i)$ the total number of basic blocks of i. Note that we only consider coverage for the test goals. Figure 7 reports a distribution of coverage differences for each pair of number of goals used and experimental subject. For example, the plot to the bottom right of Figure 7 indicates that the range of coverage difference for poi with 32 different selections of goals is $[-3.9, 59.4]$ with a median value above 20 (i.e., CB covered at least 20% more basic blocks in 5 out 10 cases).

The list below describes key observations:

- The coverage difference was negative in some cases, i.e., AO achieves more coverage than CB in some cases. The average of the median coverage for nanoxml was -5.94, for health-watcher was -0.23, for jmeter was 20.28, and for poi 29.21. This happens because the slicer does not add important members to the slice (i.e., members that can build calling context to exercise the test goals). This result indicates that missing members have a bigger impact in smaller subjects and for a small number of goals. For larger programs (or more goals) we noted that additional members compensate for those missing.
- The distributions show that the median coverage difference reduces with the increase in the number of goals. For smaller programs, CB improves coverage

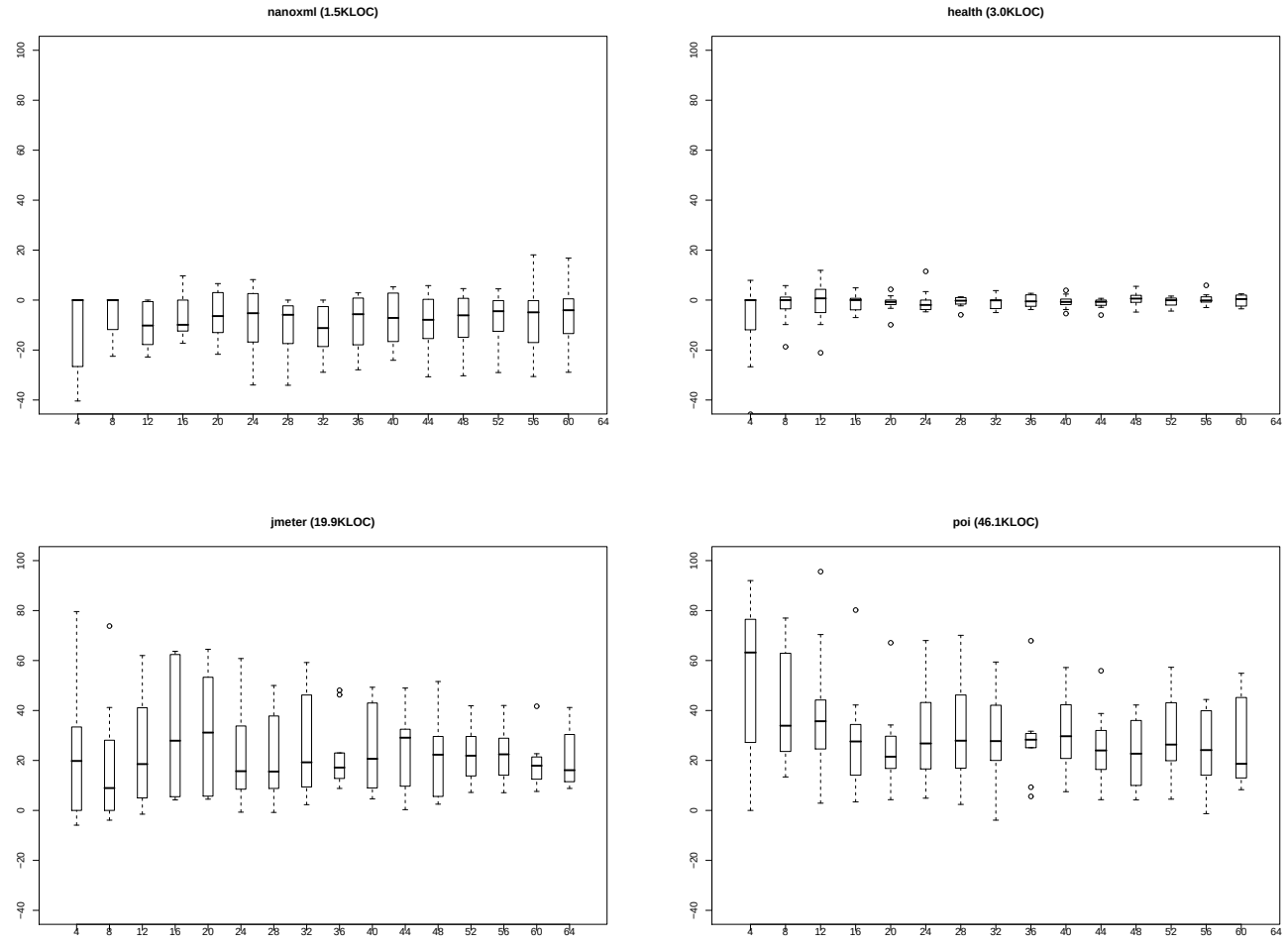

Figure 7: Impact of CB on coverage. A datapoint corresponds to the difference in coverage when running Randoop with a 10s timeout with and without CB. A point below (above) 0 indicates loss (gain) in coverage.

relative to AO for an increased number of test goals. The opposite happens for bigger programs. This result confirm our expectations that AO and CB should produce similar results for a sufficiently large number of goals.

Impact of time on CB

This section discusses the impact that the increased values of timeout yields on coverage. Intuitively, one expects that the gain of CB reduces with the increase of timeout.

Figure 8 shows how the coverage difference (between CB and AO) evolves with the increased time alloted for the generation of random sequences. In this experiment we run CB and AO on the jmeter and poi subjects using 4 different random selections of 32 goals. For each triple consisting of seed (used to control both the selection of goals and the random generator), timeout, and subject, we run CB and AO and compute the difference in coverage they obtain. We use timeouts from 10 to 40 seconds, with a 5 second step. Figure 8 plots for each subject the coverage difference as a function of time. It is important to note that Randoop raised a stack overflow error for AO on poi after 25 seconds for seeds 43 and 65465 and after 30s for seed 91. For these cases we could not plot the data-points.

The list below describes key observations:

- The pattern of the plots are more similar for jmeter than for poi. This suggests that the variance in the size of the slice correlates positively with the similarity of plots. Note from Figure 1 that the variance in slice sizes for jmeter is small. In contrast, the similarity of plots and the variance in slice sizes are higher for poi. This result suggests that the variance in slice sizes, which one can compute statically, can help to estimate the performance of CB from previous runs.
- The difference of coverage did not decrease sharply with time for the parameters we used in this experiment. Note also that for seeds 91 and 6 of poi the difference increased and decreased rapidly before returning to the measurement at 20s. The plots show

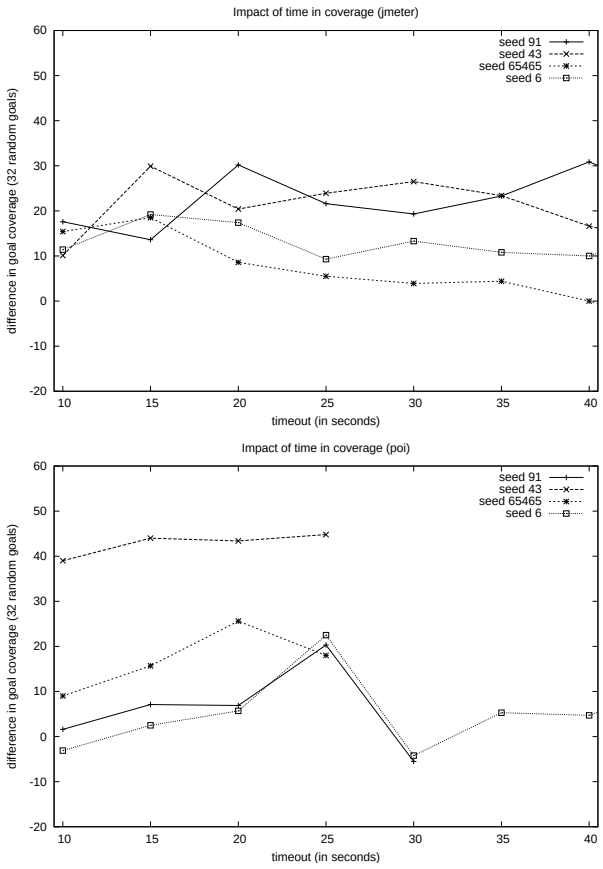

Figure 8: Impact of time in AO and CB. Each point corresponds to the difference in coverage (CB - AO).

that coverage varies for both techniques at different rates but CB still showed gain in most samples. For the jmeter subject, for example, the measurement of 3 out 4 seeds keeps above the 10-points difference, i.e., CB cover 10% more basic blocks than AO.

Seeded changes

Figure 9 shows the difference in coverage between CB and AO when providing CB with a selection of changes from SIR. We use a line to indicate AO coverage and a box to indicate CB's. The SIR infrastructure provides sequential correct versions of each subject and scripts to inject faults to each correct version. We provide as input to CB a set of 13 changes corresponding to all changes from faulty versions 1 and 2 of jmeter that modify method bodies that exist in correct version 0. The results show that for 7 changed methods CB covers more than AO, for 6 it covers the same, and in no case CB covers less. Note that in 3 of the 6 cases both CB and AO cover 0% of the test goal. This suggests that the random generation – not the techniques themselves – could not create valid inputs.

3.3 Mutation scores

We evaluate CB and AO on the jmeter and poi subjects with respect to their capability for killing mutants we constructed manually. For each of these subjects, we selected 101 members of the program and added the state-

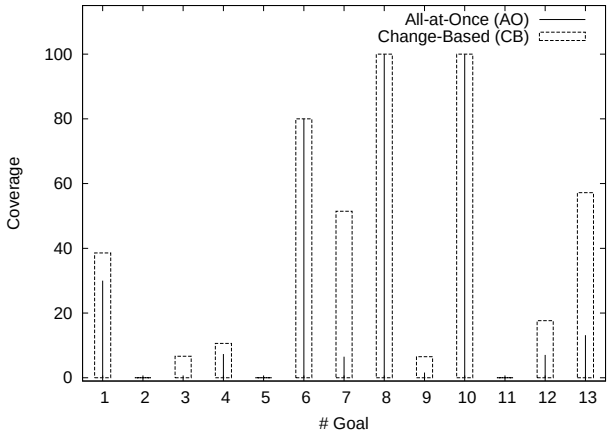

Figure 9: Basic-block coverage for a selection of jmeter methods that SIR seeds with faults.

ment `System.err.println("MUTATION-REGRESSION");` within the member at an arbitrary location. We calculate mutation scores by comparing the output logs produced with the execution of the suites that CB and AO generates. We consider a technique to kill a mutant if the execution of a suite prints `MUTATION_REGRESSION` on the output. We count the number of mutants that both techniques kill, the number of mutants that only one of them kills, and the number that neither kill.

Table 1 shows mutation scores for 5 runs of CB and AO using different seeds. Column "exp" shows the subject under test, column "#mut." shows the number of mutants constructed, column "seed" shows the seed identifier, column "cb+ao" shows the number of mutants both techniques kill, column "cb-ao" shows the number of mutants that only CB kills, column "ao-cb" shows the number of mutants that only AO kills, and "none" shows the number of mutants that neither kills. Line "avg." shows averages for each column as the ratio of mutants killed out of the total. Line "avg. relative to randoop" shows the ratio of mutants killed out of the total that Randoop can find, i.e., it ignores all mutants from column "none". For example, for column "cb+ao" from jmeter, the relative average corresponds to the ratio obtained from the sum of values from this column (namely 129) out of the sum of cells from columns "cb+ao", "cb-ao" and "ao-cb" (namely 249). The relative average helps to better identify the cases when Randoop could not generate sequences for killing a mutant.

The results show that CB killed consistently more mutants than AO for jmeter. For poi, CB and AO killed roughly the same number of mutants on average. We noted that the slicer misses members relevant to exercise the changes. This suggests that one should run the techniques in parallel when possible. It is important to note that Randoop could create more tests that exercise the changes on jmeter than on poi: column "none" shows that Randoop (i.e., neither CB nor AO) could not kill the mutant on 51% of the cases in jmeter and 74% in poi.

exp	#mut.	seed	killed			
			cb+ao	cb-ao	ao-cb	none
jmeter	101	1	26	18	7	50
		2	33	12	5	51
		3	25	16	10	50
		4	26	14	7	54
		5	19	17	14	51
avg.			0.26	0.15	0.09	0.51
avg. relative to randoop			0.52	0.31	0.17	-
poi	101	1	16	4	7	74
		2	10	10	8	73
		3	14	5	5	77
		4	18	3	5	75
		5	15	10	3	73
avg.			0.14	0.06	0.06	0.74
avg. relative to randoop			0.55	0.24	0.21	-

Table 1: Mutation scores for jmeter and poi.

3.4 Discussion

The results confirmed that the slice that CB uses is unsafe with respect to changes. It can leave important members out of the slice and that can affect the quality of the suite Randoop generates. Intuitively, the slicer employs the typical static analyzes trade between precision and scalability: the size of the slicer grows faster when it uses less restrictive slicing criteria which, in principle, would permit inclusion of members relevant to exercise changes. Note, however, that in many cases the results show that CB improved over AO on coverage and mutation scores. This suggests that CB and AO are complementary. Section 5 elaborates on our plan to evaluate the impact of alternation between AO and CB to improve change-based generation.

The effectiveness of CB depends on a number of factors including the number of changes, the time alloted for sequence generation, and also the structure of the program. For members that are reachable from many different paths in the call graph, for example utility functions, one expects that AO will have better chances (compared to other members) to generate sequences that cover those members.

4. RELATED WORK

Incremental model checking. Traditional model checking explores the state space of a transition system to find violations of an input property. A particular problem in *software* model checking is time-inefficiency: the increase in the amount of information associated to each state demands more time for the model checker to execute each atomic transition. Lauterburg et al. [24] propose Incremental State Space Exploration (ISSE), a time reduction technique that reuses parts of the results from previous runs of a software model checker. The first time the model checker runs (say, over program version n), it stores the state space of that exploration. Once the program evolves to version $n+1$, the ISSE model checker takes advantage of the previous run in order to avoid re-running unnecessary transitions. Exploration ignores the execution of a transition if it happens over an unmodified event which has been already explored. Its goal is to improve time efficiency. Our goal is to build test sequences that exercise changes. For example, ISSE and CB differ in the drivers they use to construct the state space. ISSE uses a driver the user informs and tries to improve its exploration time; but may fail to exercise some change not reachable with the driver. CB builds a driver dynamically: it attempts to build calling contexts from the changes. CB can also fail to exercise a change for the reasons we explain on Section 2.3. There have been other previous works on incremental model checking [13, 26, 34]. To a great extent they share similar differences (and orthogonal nature) existent between ISSE and our change-based testing.

El-Kafik et al. [18] propose an incremental testing for systems modeled as a finite state machines (FSMs) — typically, reactive systems or communication protocols. In this domain, both specification and implementation are modeled as FSMs. The classical model-driven approach derive test suites for a given FSM specification, provided that an implementation can be modeled with the same number of states as the specification. Such implementation will only pass the tests if and only if it conforms to the specification (i.e. the implementation contains no output or transition errors). The incremental test derivation proposed by El-Kafik et al. [18] generates new tests whenever the specification changes. The generated test suite tests only what has changed. Their approach is orthogonal to ours as it applies to more abstract system models.

Data flow testing. Data flow information is useful to capture data dependencies between different parts of the program. The use of *contextual def-use association*, in particular, can assist integration testing. A contextual def-use association for a variable v is the tuple (d,u,cd,cu), where d is the location of an assignment to v, u is the location of an usage of v, and cd and cu are contexts of d and u, respectively (a context is a sequence of nested method calls which leads to d or to u). For instance, consider the tuple (14, 34, `Account::setBalance()`, `Customer::printBalance()` → `Account::getBalance()`) as a contextual def-use association of the variable `Account::balance`. This tuple reports that `balance` is defined at line 14 and is used at line 34. The context of d is constructed with a call to `setBalance()` only and the context of u with the call to `printBalance()` which, in turn, calls `getBalance()`. Souter and Pollock [35] present contextual def-use association with partial aliasing analysis. They keep track of pointers by using *points-to escape graphs*, a structure proposed by Whaley and Rinard [39] and originally applied to compiler optimizations. As this structure may grow exponentially depending on the call chain length, a partial analysis is performed. We plan to investigate the impact of using a similar approach to improve the slicer quality. Similar works are also additional sources of inspiration [23, 36]. The research on static computation of def-use associations complements ours w.r.t. the slicer: in principle, we can leverage on this data for adding data-dependent members to the slicer.

Denaro et al. [17] propose a more efficient but less precise implementation of the contextual def-use associations algorithm proposed by Souter and Pollock [35]. The goal is the same as well: evaluate the quality of test suites for integration. The input of their tool is a set of classes and the output are the contextual associations of all variables of any input class (note that each variable can have more than one association). They evaluate their technique with DaTeC (Data flow Testing of Classes), a tool that measures the number of associations a user-provided test suite covers. Currently,

DaTeC provides limited supported to arrays, pointers, exception and dynamic binding. The main differences between their work and ours are: (goal) their work evaluate quality of integration tests while ours propose a sequence generation technique for integration testing, and (technique) their work propose contextual def-use chains to capture data dependencies while we use read-write inter-class field dependencies. Nevertheless, it is clear that we should evaluate the impact of more precise data-flow analysis with CB.

Impact analysis. Impact analysis tries to determine which parts of the software are affected by a change [10, 25]. The analysis can be done either statically or dynamically. Orso et al. [29] report that a conservative static analysis can produce slices as large as the entire software. However, in practice, only a subset of the slice are actually impacted by the changes. Dynamic impact analysis [9] produces slices based on the actual program execution, i.e. it always produces relevant (and smaller) slices. The disadvantage of dynamic analysis concerns safety: an important impacted code might not be selected. Both alternatives can improve our slicer with respect to the granularity of the changes and precision. We plan to evaluate the impact of a simultaneous construction of the slice and generation can yield on CB.

Random testing. Andrews et al. [8] conducted an empirical study with random testing to evaluate the typical length of fault-revealing test sequences. They investigate whether there is a correlation between the length of a test sequence and its capability of failure detection. One possible application of this study is test generation: the knowledge of optimal sequence lengths can improve automation as one can make a conscious decision for setting the random exploration depth. The generator can use this information to prune the search space. Their subjects were affected in different ways: some showed linear and non-linear correlation. Although Andrews et al. did not provide a method for selecting an optimal length, any other random sequence generation could benefit from these results by allowing an optimal length to be given as input.

5. CONCLUSIONS

We propose Change-Based Random Testing (CB), a technique that uses the changes programmers make to reduce the search space for random sequence generation. The expected scenario of use for change-based testing is that of a tester who wants to construct test sequences that exercise the integration of the program members she has changed during some period of work. One can automate the identification of these changed parts with a tool similar to Mylin [4] that keeps track of programmer's activity. (We did *not* implement such a tool nor evaluate CB on the field.)

We conducted experiments with 4 open-source subjects of varying sizes to compare the impact of CB and AO with respect to coverage and mutation scores. (AO refers to the use of random sequence generation with an input list of all public methods and constructors.) We also evaluate the impact that the number of changes and that time yields on coverage, and show the coverage difference for a set of changes on the jmeter [3] subject collected from the SIR [7] repository. The results show that CB could improve coverage consistently. The results, especially on mutation scores, confirm that CB can sometimes miss the generation of a sequence that could potentially lead to a bug. Overall, the results show that the techniques are complementary suggesting that one should use CB and AO in parallel when possible.

We plan to extend this work in several ways. First and foremost we plan to quantify the quality of the slices we generate. In particular, we want to identify the fraction of the slice that actually contribute to exercising changes in a test suite. Second, we plan to investigate the impact of alternation as means to alleviate the lack of important members in the slice. Note that we have observed that the execution of AO creates important sequences that CB could build on. Third, we plan to investigate whether a tighter integration with a random sequence generation could improve CB. More specifically, we plan to compute dynamic slices simultaneously with the random sequence generation to improve quality of the slices and of the test suite. For example, we could avoid adding to the suite *pass* from Figure 5 tests rooted on goal ancestors whose executions do not cover test goals. We plan to generate sequences in a bottom-up approach: starting with the test goals and adding sequences to the *pass* only when it helps to build calling context. We also plan to integrate sequence generation with oracle generation techniques in a tool. In particular, we plan to compare the spectra of executions [40, 21, 19] to assert correctness of the tests that CB generates.

It is important to mention that we present a technique that leverages on changes to improve testing. We evaluate this technique with random testing. The results we showed are subject to the limitations of random sequence generation in general and the Randoop's implementation in particular.

6. ACKNOWLEDGMENTS

We sincerely thank our colleagues Carlos Pacheco for the support on Randoop and with comments on the manuscript, Filipe Cesar for discussing early versions of CB and implementing one early prototype, and Glaucia Peres, Diego Cavalcanti and Rohit Gheyi for revising the manuscript. This work was partially supported by FACEPE grants APQ-0093-1.03/07 and APQ-0074-1.03/07.

7. REFERENCES

[1] HealthWatcher webpage. `http://www.comp.lancs.ac.uk/~greenwop/tao`.
[2] JavaNCSS webpage. `http://www.kclee.de/clemens/java/javancss`.
[3] JMeter webpage. `http://jakarta.apache.org/jmeter/`.
[4] Mylin web page. `http://www.eclipse.org/mylyn/`.
[5] NanoXML webpage. `http://nanoxml.sourceforge.net`.
[6] POI webpage. `http://poi.apache.org/`.
[7] SIR webpage. `http://sir.unl.edu`.
[8] J. H. Andrews, A. Groce, M. Weston, and R.-G. Xu. Random test run length and effectiveness. In *Proc. of Automated Software Engineering (ASE)*, pages 19–28, 2008.
[9] T. Apiwattanapong, A. Orso, and M. J. Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proc. of International Conference on Software Engineering (ICSE)*, pages 432–441, 2005.

[10] R. S. Arnold and S. A. Bohner. Impact analysis - towards a framework for comparison. In *Proc. of International Conference on Software Maintenance (ICSM)*, pages 292–301, 1993.

[11] B. Beizer. *Software Testing Techniques*. International Thomson Computer Press, 1990.

[12] C. Cadar and D. R. Engler. Execution generated test cases: How to make systems code crash itself. In *Proc. of the International SPIN Workshop on Model Checking of Software (SPIN)*, pages 2–23, 2005.

[13] C. L. Conway, K. S. Namjoshi, D. Dams, and S. A. Edwards. Incremental algorithms for inter-procedural analysis of safety properties. In *Proc. of Computer Aided Verification (CAV)*, pages 449–461, 2005.

[14] C. Csallner and Y. Smaragdakis. Check'n Crash: Combining static checking and testing. In *Proc. of International Conference on Software Engineering (ICSE)*, pages 422–431, 2005.

[15] M. Dahm and J. van Zyl. Byte Code Engineering Library, April 2003. `http://jakarta.apache.org/bcel/`.

[16] M. d'Amorim, C. Pacheco, T. Xie, D. Marinov, and M. D. Ernst. An empirical comparison of automated generation and classification techniques for object-oriented unit testing. In *Proc. of Conference on Automated Software Engineering (ASE)*, pages 59–68, 2006.

[17] G. Denaro, A. Gorla, and M. Pezzè. Contextual integration testing of classes. In *Proc. of Fundamental Approaches to Software Engineering (FASE)*, pages 246–260, 2008.

[18] K. El-Fakih, N. Yevtushenko, and G. von Bochmann. FSM-based incremental conformance testing methods. *IEEE Transactions on Software Engineering (TSE)*, 30(7):425–436, 2004.

[19] R. B. Evans and A. Savoia. Differential testing: a new approach to change detection. In *Proc. of the European Software Engineering Conference and the Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 549–552, 2007.

[20] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proc. of the Programming Language Design and Implementation (PLDI)*, pages 213–223, 2005.

[21] A. Groce, G. Holzmann, and R. Joshi. Randomized differential testing as a prelude to formal verification. In *Proc. of International Conference on Software Engineering (ICSE)*, pages 621–631, 2007.

[22] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proc. of International Conference on Software Engineering (ICSE)*, pages 291–301, 2002.

[23] M. J. Harrold and M. L. Soffa. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(2):175–204, 1994.

[24] S. Lauterburg, A. Sobeih, D. Marinov, and M. Viswanathan. Incremental state-space exploration for programs with dynamically allocated data. In *Proc. of International Conference on Software Engineering (ICSE)*, pages 291–300, 2008.

[25] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *Proc. of International Conference on Software Engineering (ICSE)*, pages 308–318, 2003.

[26] Makowsky and Ravve. Incremental model checking for decomposable structures. In *Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 540–551, 1995.

[27] G. J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., 1979.

[28] National Institute of Standards and Technology. The economic impacts of inadequate infrastructure for software testing. Planning Report 02-3, May 2002.

[29] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *Proc. of the Foundations of Software Engineering (FSE)*, pages 128–137, 2003.

[30] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *Proc. of the European Conference Object-Oriented Programming (ECOOP)*, pages 504–527, 2005.

[31] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proc. of International Conference on Software Engineering (ICSE)*, pages 75–84, 2007.

[32] M. Pezze and M. Young. *Software Testing and Analysis: Process, Principles, and Techniques*. John Wiley & Sons, 2008.

[33] P. Santhanam and B. Hailpern. Software debugging, testing, and verification. *IBM Systems Journal*, 41:4–12, 2002.

[34] Sokolsky and O. S. A. Smolka. Incremental model checking in the modal mu-calculus. In *Proc. of the Computer Aided Verification (CAV)*, pages 351–363, 1994.

[35] A. Souter and L. Pollock. The construction of contextual def-use associations for object-oriented systems. *IEEE Transactions on Software Engineering (TSE)*, 29(11):1005–1018, 2003.

[36] A. L. Souter, L. L. Pollock, and D. Hisley. Inter-class def-use analysis with partial class representations. In *Proc. of the Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 47–56, 1999.

[37] T. Wang and A. Roychoudhury. Dynamic slicing on Java bytecode traces. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(2):1–49, 2008.

[38] M. Weiser. Program slicing. In *ASE*, pages 439–449, 1981.

[39] J. Whaley and M. C. Rinard. Compositional pointer and escape analysis for java programs. In *Proc. of the Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 187–206, 1999.

[40] T. Xie and D. Notkin. Checking inside the black box: Regression testing by comparing value spectra. *IEEE Transactions on Software Engineering (TSE)*, 31(10):869–883, October 2005.

[41] X. Yuan and A. M. Memon. Using GUI run-time state as feedback to generate test cases. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 396–405, 2007.