Using Coverage Criteria on RepOK to Reduce Bounded-Exhaustive Test Suites

Valeria Bengolea[1,4], Nazareno Aguirre[1,4],
Darko Marinov[2], and Marcelo F. Frias[3,4]

[1] Department of Computer Science, FCEFQyN,
Universidad Nacional de Río Cuarto, Argentina
{vbengolea,naguirre}@dc.exa.unrc.edu.ar
[2] Department of Computer Science, University of Illinois at Urbana-Champaign, USA
marinov@illinois.edu
[3] Department of Software Engineering,
Buenos Aires Institute of Technology, Argentina
mfrias@itba.edu.ar
[4] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina

Abstract. Bounded-exhaustive exploration of test case candidates is a commonly employed approach for test generation in some contexts. Even when small bounds are used for test generation, executing the obtained tests may become prohibitive, despite the time for test generation not being prohibitive. In this paper, we propose a technique for reducing the size of bounded-exhaustive test suites. This technique is based on the application of coverage criteria on the representation invariant of the structure for which the suite was produced. More precisely, the representation invariant (which is often implemented as a `repOK` routine) is executed to determine how its code is exercised by (valid) test inputs. Different valid test inputs are deemed equivalent if they exercise the `repOK` code in a similar way according to a white-box testing criterion. These equivalences between test cases are exploited for reducing test suites by removing from the suite those tests that are equivalent to some test already present in the suite.

We present case studies that evaluate the effectiveness of our technique. The results show that by reducing the size of bounded-exhaustive test suites up to two orders of magnitude, we obtain test suites whose efficacy measured as their mutant-killing ability is comparable to that of bounded-exhaustive test suites.

1 Introduction

Testing is the primary approach to detect bugs in software. It consists of executing a piece of software under assessment for a variety of test cases. These cases often correspond to instantiating parameters of the software with different inputs. Moreover, in order to increase the chances of detecting bugs, one typically seeks these inputs to be as many and as varying as possible [19].

A.D. Brucker and J. Julliand (Eds.): TAP 2012, LNCS 7305, pp. 19–34, 2012.
© Springer-Verlag Berlin Heidelberg 2012

An essential task in testing is test-input generation. It is a difficult task because one has to come up with inputs exercising the software in many different ways, and it has been typically done manually. In the last few years, various approaches and tools have been developed to perform *automated* test-input generation. A particularly challenging task is generating test inputs for code that manipulates complex data structures, e.g., directed graphs or AVL trees, because these inputs need to satisfy complex constraints to be valid. In this and other related contexts, the *bounded-exhaustive* exploration of possible inputs is an approach that has been quite successful [2, 7, 10, 13, 17]. This technique consists of generating all the inputs that satisfy the constraints corresponding to the well-formedness of the generated structures, within certain prescribed bounds. Tools following this approach usually involve some form of constraint-solving process, e.g., based on search, model checking, or combinations of these.

The rationale behind bounded-exhaustive testing dwells on the *small-scope hypothesis* [8], which conjectures that (in some contexts) if a program has bugs, then most of these bugs can be reproduced using small inputs. However, the exploration of all possible structures within the given bounds is a costly task that, even for small scopes, may produce very large test suites. Moreover, the time required to execute the obtained test suite may be many times prohibitive. For instance, for testing a merge routine on binomial heaps, the bounded exhaustive test-suite bounded by 6 nodes for each binomial heap has 57,790,404 tests. Also, there are situations where larger scopes are necessary to achieve coverage and detect bugs, e.g., some insertion/deletion processes in balanced trees require structures of larger sizes to force rotations or enable other rebalancing mechanisms.

In this paper we propose a technique for reducing the size of bounded-exhaustive test suites. This technique is based on the application of coverage criteria on the representation invariant of the structure for which the suite was produced. More precisely, the representation invariant, i.e., the constraint indicating whether a structure is well-formed or not, is employed to define an equivalence relation between valid test inputs. The technique requires the representation invariant to be provided as a `repOK` routine [11], and consists of analysing how the code of this routine is exercised by different test inputs. Different valid test inputs will be considered equivalent if they exercise the `repOK` code in a similar way, according to some white-box testing criterion. These equivalences between test cases are exploited for filtering tests, leaving out of the suite those tests that are equivalent to some test already present in the suite.

Essentially, our proposal involves the definition of a *black-box testing criterion with respect to the code under test*, defined in terms of *white-box testing criteria with respect to the representation invariant for the inputs of the code under test*. Namely, our criterion specifies when two different inputs are to be considered equivalent disregarding the structure of the code under test (hence, black-box), by considering only the structure of `repOK` routine (hence, white-box). We present a particular application of this criterion to the reduction of bounded-exhaustive test suites for *imperative/executable* representation invariants. However, the approach presented in this paper can also be adapted to *declarative*

representation invariants, which are becoming popular in various object-oriented languages, e.g., invariants as specified in Eiffel or via contract languages such as JML [3] and Code Contracts [4]; the adaptation is straightforward when these invariants are involved in run-time contract-checking environments, where they are made "executable" and the code corresponding to their run-time evaluation would correspond to an imperative `repOK` routine.

To assess the effectiveness of the reduced test suites produced using our approach, we present some case studies comparing bounded-exhaustive suites with suites whose size is reduced employing a variety of white-box testing criteria on `repOK`, for various data structures. We find that the reduction of up to two orders of magnitude still largely preserves the mutant-killing capability of test suites for various operations on these data structures.

2 Preliminaries

Test Coverage Criteria. A test coverage criterion is a means for measuring how well a test suite exercises a program under test. Coverage criteria are mainly classified into *black-box* and *white-box* [6,19]; the former disregard the structure of the program under test, while the latter may pay special attention to the structure of the program under test. Black-box coverage criteria "see" the code under test as a black box, taking into consideration only the specification of the program. An example of a known black-box criterion is equivalence partitioning coverage, which consists of partitioning the space of program inputs into equivalence classes, defined in terms of the specification of the expected inputs for the program under test. White-box coverage criteria analyse the program under test, and how the tests in the test suite exercise it, in order to measure coverage. A simple well-known white-box coverage criterion is decision coverage, which, in order to be satisfied, requires each decision point in the program under test (conditions in if-then-else statements, loops, etc.) to evaluate to true and false when different tests in the suite are exercised.

Test-Input Generation for Complex Structures. In the context of test-input generation for complex structures, two approaches can be distinguished, the *generative* approach and the *filtering* approach [7]. The former works by generating instances of the input structure by calling a *generator* routine, that combines calls to constructors and insertion routines on the structure. The latter builds candidate structures using only its structural definition, and then employs a predicate that characterises valid structures, known as a representation or class invariant, in order to filter out the invalid candidates. The representation invariant can be defined declaratively, e.g., using some contract-specification language such as JML [3], or operationally, i.e., via a routine that, when applied to a candidate, returns true if and only if the candidate is a valid one. The latter are typically called `repOK` routines [11]. As put forward in [11], developers should equip their complex structures implementations with `repOK` routines, since these routines will greatly help in debugging the implementations.

Bounded-Exhaustive Testing. Bounded-exhaustive testing is a testing technique that has proved useful in certain testing contexts, in particular, testing code that manipulates complex data structures. Examples of such code include libraries of data structures such as AVL trees, graphs, linked lists, etc., and programs that manipulate source code (where source code can be viewed as data with a complex structure) such as compilers, type checkers, refactoring engines, etc.

Bounded-exhaustive testing produces, for a given program under test and a user-provided bound k on the size of inputs, all valid inputs whose size is bounded by k, and then tests the program using the produced test suite. The rationale behind the approach is that many bugs in programs manipulating complex structures can be reproduced using small instances of the structure. Thus, by testing the program on all possible structures bounded in size by some relatively small scope one would be able to exhibit many bugs.

3 Reducing Bounded-Exhaustive Test Suites

In this section, we present an approach to help in reducing bounded-exhaustive test suites. The approach assumes that we have an imperative implementation of the representation invariant of the structure for which the bounded-exhaustive suite was produced; thus, it fits better with filtering approaches to test generation (for which such a representation invariant is often a requirement). The reduction process works by defining a family of coverage criteria and employing the `repOK` routine (i.e., the imperative implementation of the representation invariant) to define an equivalence between inputs. Then, according to some reduction rate on the bounded-exhaustive suite, test cases are discarded if they are "equivalent" to some test cases remaining in the suite.

To describe how the technique works, let us first describe how we define coverage criteria using `repOK`. Let C be a class, and let `repOK` be a parameterless boolean imperative routine, characterising the representation invariant of C. The representation invariant is the property that distinguishes well-formed instances from ill-formed ones. A property expected of C is that its constructors must establish `repOK` after their execution, and public methods of C must preserve it. As an example, let us consider the following Java classes, implementing binary trees of integers:

```
public class BinaryTree {
    private Node root;
    private int size;

    ...
}
```

```
public class Node {
   private int key;
   private Node left;
   private Node right;

   ...
   // setters and getters
   // of the above fields
   ...
}
```

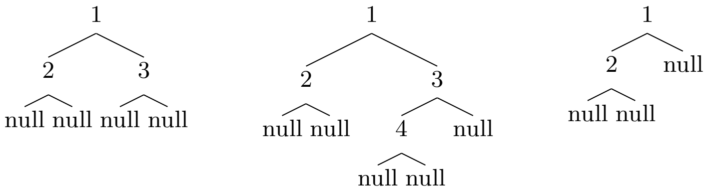
The representation invariant for this class should check that the linked structure starting with `root` is indeed a tree, i.e., that it is acyclic and with a single parent

for every reachable node except the root, and that the value of `size` agrees with the number of nodes in the structure. Checking that this property holds for a binary tree object can be implemented as in the following method from class `BinaryTree` (taken from the examples distributed with the Korat tool [2]):

```
public boolean repOK() {
    if (root == null) return size == 0;
    Set visited = new HashSet();
    visited.add(root);
    LinkedList workList = new LinkedList();
    workList.add(root);
    while (!workList.isEmpty()) {
        Node current = (Node) workList.removeFirst();
        if (current.getLeft() != null) {
            if (!visited.add(current.getLeft())) return false;
            workList.add(current.getLeft());
        }
        if (current.getRight() != null) {
            if (!visited.add(current.getRight())) return false;
            workList.add(current.getRight());
        }
    }
    return (visited.size() == size);
}
```

Now suppose that one needs to test a routine that receives as a parameter a binary tree, e.g., binary tree traversal routine. Notice that, as a (black-box) criterion for testing the traversal routine, we can define a partition of all possible binary tree structures according to the way the different structures "exercise" the `repOK` routine. The motivation is basically that tests that exercise the code of `repOK` in the same way can be considered as similar, and therefore can be thought of as corresponding to the same class.

We still have to define what we mean by "exercise in a similar way". This can be done, in principle, by choosing any white-box coverage criterion, to be applied to `repOK`. For instance, we can consider *decision coverage* on `repOK`; in this case, two inputs to the traversing routine (the code under test) would be considered equivalent if they make the decision points in `repOK` to evaluate to the same values. Thus, for instance, of the following three trees:



the first and the second would be considered equivalent, but none of these would be equivalent to the third one (notice that, as opposed to these other two, predicate `current.getRight() != null` never evaluates to true in this case).

In general, notice that any white-box testing criterion *Crit* gives rise to a *partition* of the input space of the program under test, with each class in the partition usually capturing some path or branch condition expressed as a constraint on the inputs. Given a program under test P, a criterion *Crit*, and an input c, we will denote by $[\![c]\!]^{P}_{Crit}$ the partition c belongs to, i.e., the set of all inputs that exercise the code of P in the same way c does, according to *Crit*. Our technique works by defining an equivalence between inputs. Let C be a `repOK`-equipped class, and let *Crit* be a selected white-box coverage criterion. Given two valid objects c_1 and c_2 of C, i.e., two objects satisfying C's representation invariant, we will say that c_1 is equivalent to c_2 (according to `repOK` under *Crit*), if and only if $[\![c_1]\!]^{\mathtt{repOK}}_{Crit} = [\![c_2]\!]^{\mathtt{repOK}}_{Crit}$.

In the above example we picked one of the simplest white-box coverage criteria to be applied to `repOK`; of course, choosing more sophisticated coverage criteria (e.g., path coverage, condition coverage, MCDC, etc.) would yield finer grained equivalence relations on the state space of the input data type.

Once one has decided the white-box criterion to be applied to `repOK`, one can use it to reduce bounded-exhaustive suites. The approach we followed for doing so is the following. Suppose that you have used some mechanism for generating a bounded-exhaustive test suite, to be used for testing, with N tests in it. Moreover, you have realised that you will not have enough resources to analyse the program under test for all these cases. Instead, you have resources to test your system for a fraction of this suite, let us say $N/10$. In this case, we do as follows:

- Determine the number of possible equivalence classes of inputs (depends both on the white-box criterion chosen on `repOK` and the complexity of `repOK`'s code).
- Set a maximum max_q for the number of tests for every single equivalence class q. For instance, divide the size of the test suite to be built (in the example $N/10$) by the number of equivalence classes, and set this as a maximum.
- Process the bounded-exhaustive test suite, leaving at most max_q tests for each equivalence class q of inputs.

As we mentioned, the result of applying the above process strongly depends on the selected white-box criterion. Moreover, this process strongly depends on the structure of the `repOK` routine too. For instance, an if-then-else with a composite condition could alternatively be written as nested if-then-else statements with atomic conditions; such structurally different but behaviourally equivalent programs may have very different equivalence classes, for the same white-box criterion, and therefore our approach may result in different reduced suites.

4 On the Effectiveness of Reduced Test Suites

In this section we evaluate the effectiveness of test suites reduced using the approach presented in the previous section. The evaluation is based on several case studies, corresponding to analyses of various routines on selected heap-allocated data structures, namely *binomial heaps*, *binary search trees*, *doubly linked lists*,

and *red black trees*. We have used the implementation of these structures provided in the Roops benchmark [15]. We are not dealing in this paper with bounded-exhaustive generation, so the approach would work with any generation tool. It is worth mentioning however that we generated the bounded-exhaustive suites on which reductions are applied, using Korat [2]. Also, we experimented with different coverage criteria on `repOK`, in order to perform the reductions. We selected three coverage criteria: *decision coverage*, *path coverage* and a variant of decision coverage, that we call *counting decision coverage*. Notice that, since we are comparing with bounded-exhaustive suites, we are able to determine precisely which are the coverable equivalence classes for each criterion (e.g., we are able to determine precisely which `repOK` paths the bounded-exhaustive suites cover), which is necessary for the reduction process. Of course, this requires executing `repOK` for *all* tests in the bounded-exhaustive suite, a task which would anyway be done at test generation time, prior to suite reduction and the testing of the program under test.

We also used counting decision coverage (CDC). This criterion takes into account the number of times each decision in the program evaluates to true and false. More precisely, given a program under test P and two inputs c_1 and c_2 for P, c_1 and c_2 are equivalent according to P under CDC if and only if, for every decision point *cond* in P, the number of times *cond* evaluates to true (resp. false) when P is executed for c_1 equals the number of times *cond* evaluates to true (resp. false) when P is executed for c_2. We believe CDC to be useful in our context since, in general, there is a relationship between the size of a structure and the number of times a particular decision point in the corresponding `repOK` evaluates to true or false (think of conditions inside loops). As a consequence, as the size of a structure increases, the number of equivalence classes will also increase, and hence the variety of cases in the reduced suite. For instance, while decision coverage considers as equivalent the first two trees in the example of the previous section, CDC will distinguish them.

Structure of the Experiments. We took the `repOK` code for each of the above mentioned structures, and we automatically instrumented it to obtain, from a `repOK` call on a given valid structure, the equivalence class the structure belongs to, for each of the selected criteria. We ran the instrumented `repOK` methods on tests of the bounded-exhaustive test suite to collect their equivalence class information. We then built reduced test suites that select from a bounded-exhaustive test suite some test cases for each (coverable) equivalence class corresponding to the criterion. In particular, we reduced the bounded-exhaustive test suites by one and two orders of magnitude, i.e., 10% and 1% of the starting test suite size. The test cases selected for the reduced test suite are the *first* generated/encountered test cases for each of the coverable equivalence classes. Note that other selections could be possible, e.g., randomly selecting an appropriate number of test cases for each equivalence class. The selection has been made taking at most N_r/M test cases for each equivalence class, where N_r is the size of the reduced test suite (e.g., 10% of the bounded-exhaustive suite) and M is the number of equivalence classes. In both cases (10% reduction and 1% reduction), when the

bounded-exhaustive test suite was too small to reduce it to 10% (or 1%) of its original size, we have taken at least one test case for each covered equivalence class.

To measure the effectiveness of the approach, we took some sample routines manipulating the data structures selected for analysis. These routines were `merge`, `insert`, `delete` and `find` for binomial heaps, `isPalindromic` for doubly linked lists, `insert`, `delete` and `search` for search trees, and `add`, `remove` and `contains` on red-black trees. We generated mutants of these routines, and measured the effectiveness of the different suites, bounded-exhaustive and reduced, in mutant killing. We also included in this assessment the "one per class" suites, consisting of exactly one test per coverable equivalence class (i.e., a minimal suite with the same coverage as the corresponding bounded-exhaustive suite). We used muJava [14] to generate mutants. The mutants we got are those obtained by the application of 12 different method-level mutation operators [12], including arithmetic, logical and relational operator replacement, when these ones were applicable to the selected routines.

We have tried to foresee potential threats to the validity of our experimental results. The case studies represent, in our opinion, typical testing situations in the context of the implementation of complex, heap allocated data structures (a main target for bounded-exhaustive testing). We chose case studies of varying complexities, including data structures with simple, intermediate, and complex constraints (e.g., linked lists, search trees and binomial heaps, respectively). Since the approach depends on the structure of `repOK`, we took implementations of these routines as provided in Korat, instead of providing our own. Also, for the evaluation we selected coverage criteria of varying complexities: the rather simple decision coverage, the more thorough path coverage, and an intermediate one, counting decision coverage.

4.1 Case Studies

Binomial Heaps (`merge`). This case study involves testing `merge`, a routine manipulating binomial heaps. This routine takes as parameters a pair of binomial heaps, and produces a binomial heap corresponding to the union of the two parameters. This is an example of a case in which the bounded-exhaustive suites quickly become too large, making bounded-exhaustive testing impractical. Figure 1 shows, for various scopes, the sizes of bounded-exhaustive (BE) suites and suites with `repOK`-based reductions to 10% and 1%, for the three mentioned white-box coverage criteria applied to `repOK`. For each criterion, it is also indicated the number of equivalence classes of inputs that have been covered (CC, for covered classes). The scope in this case specifies the maximum number of elements for both heaps, and the range for nodes' keys, from zero to the specified value. Since the bounded-exhaustive suites have been generated using Korat, these exclude symmetric cases on reference fields (Korat provides a symmetry-breaking mechanism as part of its generation process).

The `merge` routine was mutated, obtaining a total of 117 mutants. Then, the ability to kill mutants of the bounded-exhaustive, the reduced test suites

Scope	BE	Decision Cov.			Count. Decision Cov.			Path Cov.		
		10%	1%	CC	10%	1%	CC	10%	1%	CC
2,2	36	3	3	3	9	9	9	9	9	9
3,3	784	76	4	4	59	16	16	59	16	16
4,4	14,400	1200	144	4	1060	119	25	1060	119	25
5,5	876,096	49,420	7506	4	42,500	6460	36	42,500	6460	36
6,6	57,790,404	2,455,826	342,166	4	1,993,860	315,698	49	1,993,860	315,698	49

Fig. 1. Sizes of bounded-exhaustive and suites with `repOK`-based reductions, for testing binomial heap's `merge`

Scope	BE	Decision Cov.			Count. Decision Cov.			Path Cov.		
		10%	1%	OPC	10%	1%	OPC	10%	1%	OPC
2,2	38	100	100	100	42	42	42	42	42	42
3,3	8	11	86	86	8	14	14	8	14	14
4,4	7	7	11	86	7	7	12	7	7	12
5,5	7	7	7	86	7	7	12	7	7	12
6,6	7	7	7	86	7	7	12	7	7	12

Fig. 2. Measurement of effectiveness of bounded-exhaustive and reduced test suites, on mutant killing for `merge` (table reports mutants remaining live)

and the minimal "one per equivalence class", was assessed. Figure 2 reports the results indicating the remaining live mutants, and highlighting the cases in which the mutation score of the reduced suites matched that of the corresponding bounded-exhaustive suite. Out of the 7 mutants that remained live with the largest bounded-exhaustive suite, 4 are equivalent to the original program. Notice that in this case, the reduced test suites for all the coverage criteria analysed were in most cases as effective as the bounded-exhaustive suites, for mutant killing, even with suites of 1% the size of the bounded-exhaustive ones.

Binomial Heaps (`insert`, `delete` and `find`). Our second case study corresponds to routines manipulating a single binomial heap, namely *insert*, *delete* and *find*. Figure 3 shows, for various scopes, the sizes of the various suites. The scopes in this case simply indicate the sizes of the corresponding binomial heaps.

Routines `insert`, `delete` and `find` were mutated (the number of mutants obtained were 99, 184 and 28, respectively), and the effectiveness of the different suites on mutant killing was assessed. Figure 4 reports the results of the analysis for this case study. Out of the 25 and 31 mutants that remained live with the largest bounded-exhaustive suite for `insert` and `delete`, 2 and 14 are equivalent to the respective original program. In this case, the reduced suites were not as effective as the previous case study, especially for the `delete` routine. However, notice that the results are still very good, taking into account the reduction in size of the suites. For instance, for scope 8 and counting decision coverage, the 10%-reduced suite only misses one mutant (32 vs. 31 out of 184) compared to the bounded-exhaustive suite.

Scope	BE	Decision Cov.			Count. Decision Cov.			Path Cov.		
		10%	1%	CC	10%	1%	CC	10%	1%	CC
2	12	3	3	3	3	3	3	3	3	3
3	84	8	4	4	8	4	4	8	4	4
4	480	40	4	4	40	5	5	40	5	5
5	4680	264	38	4	339	40	6	339	40	6
6	45,612	1938	270	4	2772	367	7	2772	367	7
7	751,912	37,650	3814	4	33,052	4947	8	33,052	4947	8
8	4,829,952	241,568	24,220	4	217,662	29,494	9	217,662	29,494	9

Fig. 3. Sizes of bounded-exhaustive and suites with `repOK`-based reductions, for testing binomial heap's operations `insert`, `delete` and `search`

Doubly Linked Lists (`isPalindromic`). Our next case study corresponds to the routine `isPalindromic`, which checks whether a given sequence of integers (implemented over a doubly linked list) is a palindrome. Figure 5 shows, for various scopes, the sizes of the various suites and the number of equivalence classes covered. The scopes in this case correspond to the number of entries in the list, the range for the size of the list, and the number of integer values allowed in the list. The routine `isPalindromic` was mutated, obtaining 23 mutants. Figure 6 reports the results of the analysis for this case study. Out of the 13 mutants that remained live with the largest bounded-exhaustive suite, 2 are equivalent to the original program. In this case study, reduced test suites are again as effective as the bounded-exhaustive ones, in most of the cases, even reduced to 1% of the size of the bounded-exhaustive ones.

Search Trees (`insert`, `delete` and `search`). Our next case study regards the data structure search trees, and the main routines for insertion, deletion and search. Figure 7 shows, for various scopes, the sizes of the various suites, and the number of covered classes. The scopes indicate the maximum number of nodes in the tree, the range for the size field of the tree, and the number of keys allowed in the tree.

Routines `insert`, `delete` and `search` were mutated (the number of mutants obtained were 9, 24 and 4, respectively). Table 8 reports the results obtained for the analysis. In this case study, reduced test suites are again as effective as the bounded-exhaustive ones, in most of the cases, with less effectiveness in the `delete` routine. Notice however that the mutant-killing score is still very good for `delete` in the reduced suites, with counting decision coverage at a 10% almost matching the bounded-exhaustive suite in scope 6,0,6,9 (2 vs. 0 out of 24 mutants).

Red-Black Trees (`remove`, `add` and `contains`) The last case study we present involves routines manipulating red-black trees. There routines are `remove`, `add` and `contains`. Figure 9 shows, for various scopes, the sizes of the corresponding suites and the number of equivalence classes covered in each case. The scopes

Scope	Oper.(#Mutants)	BE	Decision Cov.			Count. Decision Cov.			Path Cov.		
			10%	1%	OPC	10%	1%	OPC	10%	1%	OPC
2	insert(99)	36	44	44	44	44	44	44	44	44	44
	delete(184)	124	152	152	152	152	152	152	152	152	152
	find(28)	0	12	12	12	12	12	12	12	12	12
3	insert(99)	25	26	34	34	26	34	34	26	34	34
	delete(184)	81	106	149	149	106	149	149	106	149	149
	find(28)	0	8	12	12	8	12	12	8	12	12
4	insert(99)	25	25	34	34	25	34	34	25	34	34
	delete(184)	77	99	149	149	101	149	149	101	149	149
	find(28)	0	6	12	12	6	12	12	6	12	12
5	insert(99)	25	25	25	34	25	25	34	25	25	34
	delete(184)	61	80	101	149	65	85	149	65	85	149
	find(28)	0	2	6	12	2	6	12	2	6	12
6	insert(99)	25	25	25	34	25	25	34	25	25	34
	delete(184)	33	65	99	149	48	53	115	48	53	115
	find(28)	0	2	6	12	0	3	12	0	3	12
7	insert(99)	25	25	25	34	25	25	34	25	25	34
	delete(184)	31	37	82	149	35	49	115	35	49	115
	find(28)	0	0	5	12	0	0	12	0	0	12
8	insert(99)	25	25	25	34	25	25	34	25	25	34
	delete(184)	31	54	70	149	32	48	115	32	48	115
	find(28)	0	2	5	12	0	0	12	0	0	12

Fig. 4. Measurement of effectiveness of bounded-exhaustive and reduced test suites, on mutant-killing for `insert`, `delete` and `search` for binomial heaps (table reports mutants remaining live)

indicate the maximum number of nodes in the tree, the range for the size field of the tree, and number of keys allowed in the tree. In this case study, paths and sizes were for some scopes too large to enable us to perform the analysis. Thus, we considered in this case study a *bounded* version of path coverage, namely path coverage without taking into account repetitions of edges (known as *simple path coverage* [19]).

Routines `remove`, `add` and `contains` were mutated (the number of mutants obtained were 142, 126 and 36, respectively), and the results of the analysis are reported in Figure 10. Out of the 41, 36 and 6 mutants that remained live with the largest bounded-exhaustive suite for `remove`, `add` and `contains`, respectively, 4, 11 and 4 are equivalent to the respective original program. In this case study, reduced test suites showed better effectiveness for the `contains` routine, matching in many cases the mutant-killing score of the bounded-exhaustive suites. For the other two routines it was not the same case, although they achieved a very good mutant-killing score in many cases (e.g., counting decision coverage for `add` in scope 7,0,7,7 missed only 4 out of 126 compared to the bounded-exhaustive suite).

Scope	BE	Decision Cov.			Count. Decision Cov.			Path Cov.		
		10%	1%	CC	10%	1%	CC	10%	1%	CC
4,0,4,4	156	8	2	2	10	4	4	10	4	4
4,0,4,8	820	42	5	2	50	7	4	50	7	4
5,0,5,5	1555	78	8	2	100	13	5	100	13	5
5,0,5,10	16,105	806	81	2	777	108	5	777	108	5
6,0,6,6	19,608	981	99	2	1035	136	6	1035	136	6
6,0,6,12	402,234	20,112	2,012	2	15,786	2,193	6	15,786	2,193	6
7,0,7,7	299,593	14,980	1,498	2	13,239	1,781	7	13,239	1,781	7
7,0,7,14	12,204,241	610,213	61,022	2	402,933	55,918	7	402,933	55,918	7

Fig. 5. Sizes of bounded-exhaustive suites and suites with `repOK`-based reductions, for testing `isPalindromic` operation for doubly linked lists

Scope	BE	Decision Cov.			Count. Decision Cov.			Path Cov.		
		10%	1%	OPC	10%	1%	OPC	10%	1%	OPC
4,0,4,4	15	15	23	23	15	22	22	15	22	22
4,0,4,8	15	15	23	23	15	15	22	15	15	22
5,0,5,5	13	15	22	23	13	15	22	13	15	22
5,0,5,10	13	15	15	23	13	13	22	13	13	22
6,0,6,6	13	13	15	23	13	13	22	13	13	22
6,0,6,12	13	13	15	23	13	13	22	13	13	22
7,0,7,7	13	13	13	23	13	13	22	13	13	22
7,0,7,14	13	13	13	23	13	13	22	13	13	22

Fig. 6. Measurement of effectiveness of bounded-exhaustive and reduced test suites, on mutant-killing for `isPalindromic` for doubly linked lists (table reports mutants remaining live)

5 Related Work

There exist some approaches that are related to the work presented in this paper. With respect to the reduction of bounded-exhaustive test suites, the work of some of the authors of this paper [1] is strongly related to the work presented in this paper, especially because both approaches are based on the use of coverage criteria. However, the previous approach [1] differs from the work of this paper in two aspects. First, it requires the user to provide the coverage criterion to perform the suite reduction, as opposed to our work here, where the coverage criterion is a standard one applied to the representation invariant. Second, the previous approach targets the improvement in the *test generation process*, whereas our work in this paper concerns the reduction of bounded-exhaustive test suites to reduce the time for testing. Another work related to ours is the one presented in [9]. In [9], the authors present various techniques for reducing the costs of bounded-exhaustive testing. These techniques are sparse test generation, which attempts to reduce the time to the first failing test (but not the

Scope	BE	Decision Cov.			Count. Decision Cov.			Path Cov.		
		10%	1%	CC	10%	1%	CC	10%	1%	CC
3,0,3,3	45	5	5	5	7	7	7	9	9	9
3,0,3,4	148	10	5	5	14	7	7	9	9	9
3,0,3,6	822	70	5	5	72	7	7	78	9	9
3,0,3,8	2,760	228	25	5	242	21	7	248	27	9
5,0,5,8	29,416	1836	240	5	2634	278	16	2888	260	65
6,0,6,9	167,814	10,158	1095	5	14,430	1605	22	16,665	1576	197

Fig. 7. Sizes of bounded-exhaustive suites and suites with `repOK`-based reductions, for testing `delete`, `insert` and `search` operations of search trees

Scope	Oper.(#Mutants)	BE	Decision Cov.			Count. Decision Cov.			Path Cov.		
			10%	1%	OPC	10%	1%	OPC	10%	1%	OPC
3,0,3,3	delete(24)	2	12	12	12	12	12	12	12	12	12
	insert(9)	0	0	0	0	0	0	0	0	0	0
	search(4)	0	0	0	0	0	0	0	0	0	0
3,0,3,4	delete(24)	2	12	12	12	12	12	12	12	12	12
	insert(9)	0	0	0	0	0	0	0	0	0	0
	search(4)	0	0	0	0	0	0	0	0	0	0
3,0,3,6	delete(24)	2	12	12	12	12	12	12	12	12	12
	insert(9)	0	0	0	0	0	0	0	0	0	0
	search(4)	0	0	0	0	0	0	0	0	0	0
3,0,3,8	delete(24)	2	9	12	12	9	12	12	9	12	12
	insert(9)	0	0	0	0	0	0	0	0	0	0
	search(4)	0	0	0	0	0	0	0	0	0	0
5,0,5,8	delete(24)	0	9	16	16	9	12	12	9	12	12
	insert(9)	0	0	0	0	0	0	0	0	0	0
	search(4)	0	0	0	0	0	0	0	0	0	0
6,0,6,9	delete(24)	0	9	16	16	2	9	12	0	12	12
	insert(9)	0	0	0	0	0	0	0	0	0	0
	search(4)	0	0	0	0	0	0	0	0	0	0

Fig. 8. Measurement of effectiveness of bounded-exhaustive and reduced test suites, on mutant-killing for `insert`, `delete` and `search` for search trees (table reports mutants remaining live)

suite); oracle-based test clustering, which groups together failing tests to reduce the time for inspection of failing tests; and structural test merging, whose purpose is to generate smaller suites of larger tests by merging together smaller test inputs. Of these three, the latter is related to our work, since it has as a purpose to reduce the size of the test suite. However, the approach is rather different, since bounded exhaustiveness is preserved in structural test merging (although sets of small inputs are encoded as a single large input), whereas in our case we drop bounded exhaustiveness by selecting only some tests. The same differences apply to other works based on test granularity [16].

Scope	BE	Decision Cov.			Count. Decision Cov.			Simple Path Cov.		
		10%	1%	CC	10%	1%	CC	10%	1%	CC
4,0,4,4	164	14	7	7	16	16	16	108	108	108
4,0,4,8	6408	500	62	7	608	64	16	169	157	157
5,0,5,5	575	53	7	7	30	30	30	97	97	97
5,0,5,10	56,790	2732	496	7	5313	532	30	245	165	157
6,0,6,6	1962	174	14	7	184	16	46	113	113	113
6,0,6,12	412,140	10,411	2,652	7	38,579	4,017	46	505	229	157
7,0,7,7	6377	469	61	7	570	66	66	154	142	142
7,0,7,14	3,045,266	89,960	11,654	7	284,408	29,449	66	2211	465	157

Fig. 9. Sizes of bounded-exhaustive suites and suites with `repOK`-based reductions, for testing `remove`, `add` and `contains` operations of red-black trees

Scope	Oper.(#Mutants)	BE	Decision Cov.			Count. Decision Cov.			Simple Path Cov.		
			10%	1%	OPC	10%	1%	OPC	10%	1%	OPC
4,0,4,4	remove(142)	47	82	82	81	70	70	70	47	47	47
	add(126)	38	58	90	90	90	90	90	44	44	44
	contains(36)	6	8	9	9	9	9	9	6	6	6
4,0,4,8	remove(142)	47	66	81	81	65	70	70	82	82	82
	add(126)	38	40	43	90	40	58	90	53	62	62
	contains(36)	6	6	7	9	6	8	9	8	8	8
5,0,5,5	remove(142)	43	81	82	81	68	68	68	68	68	68
	add(126)	38	43	90	90	90	90	90	56	56	56
	contains(36)	6	7	9	9	9	9	9	8	8	8
5,0,5,10	remove(142)	43	55	77	81	52	67	68	82	82	82
	add(126)	36	40	42	90	39	43	90	49	58	78
	contains(36)	6	6	6	9	6	7	9	8	8	8
6,0,6,6	remove(142)	41	66	82	81	46	66	66	71	71	71
	add(126)	36	42	58	90	55	90	90	62	62	62
	contains(36)	6	6	8	9	7	9	9	8	8	8
6,0,6,12	remove(142)	41	53	60	81	50	61	66	82	82	82
	add(126)	36	40	40	90	37	40	90	47	49	78
	contains(36)	6	6	6	9	6	6	9	8	8	8
7,0,7,7	remove(142)	41	60	81	81	41	66	66	76	76	76
	add(126)	36	40	43	90	40	90	90	53	62	62
	contains(36)	6	6	7	9	6	9	9	8	8	8
7,0,7,14	remove(142)	41	50	55	81	44	50	66	76	82	82
	add(126)	36	42	58	90	55	90	90	47	49	79
	contains(36)	6	6	6	9	6	6	9	8	8	8

Fig. 10. Measurement of effectiveness of bounded-exhaustive and reduced test suites, on mutant-killing for `add`, `remove` and `contains` for red-black tree (table reports mutants remaining live)

Other researchers have studied the effects of reducing test suites in finding bugs, e.g., the work in [18]. Our work is related, but we propose a specific approach for

test-suite reduction (as opposed to studying the effects of test-suite reductions in general), and we target specifically bounded-exhaustive test suites.

6 Conclusions and Further Work

Bounded-exhaustive test suites are popular in some testing contexts, such as that of testing complex heap allocated data structures. However, in many cases bounded-exhaustive test suites become too large as the bound for the generated suites increases, thus making their (exhaustive) use impractical. We have presented an approach for reducing bounded-exhaustive test suites, and consequently also the time spent in testing using these suites, for cases in which an imperative representation invariant routine is available for the inputs for which the suites were generated. The approach works by defining black-box criteria for the program under test, based on the definition of equivalence relations of inputs, defined in terms of white-box criteria on the imperative representation invariant; basically, the rationale for this is that, if two inputs exercise the representation invariant code in the same way, according to a white-box criterion, these inputs may be considered similar, i.e., considered to belong to the same equivalence class of inputs. These equivalence classes are then employed in order to filter out of the exhaustive suites some tests that are equivalent to some others already present in the suite.

Although our motivation is the reduction of bounded-exhaustive test suites, the idea of using white-box criteria on the representation invariant is indeed the definition of a new black-box coverage criterion, for programs whose inputs count on a representation invariant. This idea can also be adapted to declarative representation invariants, which are becoming popular, e.g., invariants as specified in Eiffel or via contract languages such as JML and Code Contracts; these invariants are typically involved in run-time contract-checking environments, so they are "executable", and the code corresponding to their run-time evaluation would correspond to what we referred to as `repOK` in this paper.

We presented some case studies showing the performance of suites reduced using the above approach, compared to bounded-exhaustive suites. As the experiments show, for some white-box coverage criteria on the representation invariant, we obtain a performance in mutant killing that is comparable to that of bounded-exhaustive suites. In particular, we used a variant of decision coverage, called *counting decision coverage*, which takes into account the number of times each decision point in the program under test becomes true and false. This criterion, applied to the representation invariant, is useful in our context, since in general we observe that there is a relationship between the size of the structure and the number of times a particular decision point in the corresponding representation invariant evaluates to true or false.

As work in progress, we are currently examining the approach proposed in this paper for several additional case studies, based on more complex data structures. We also plan to assess the approach in the context of testing applications manipulating source code, such as compilers or, more particularly, refactoring engines, as is done using ASTGen [5].

References

1. Aguirre, N., Bengolea, V., Frias, M., Galeotti, J.: Incorporating Coverage Criteria in Bounded Exhaustive Black Box Test Generation of Structural Inputs. In: Gogolla, M., Wolff, B. (eds.) TAP 2011. LNCS, vol. 6706, pp. 15–32. Springer, Heidelberg (2011)
2. Boyapati, C., Khurshid, S., Marinov, D.: Korat: Automated Testing based on Java Predicates. In: Proc. of Intl. Symposium on Software Testing and Analysis ISSTA 2002. ACM Press (2002)
3. Chalin, P., Kiniry, J.R., Leavens, G.T., Poll, E.: Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 342–363. Springer, Heidelberg (2006)
4. Code Contracts, `http://research.microsoft.com/en-us/projects/contracts/`
5. Daniel, B., Dig, D., García, K., Marinov, D.: Automated Testing of Refactoring Engines. In: Proc. of European Software Engineering Conference and Intl. Symposium on Foundations of Software Engineering ESEC/FSE 2007. ACM Press (2007)
6. Myers, G.J.: The Art of Sofware Testing, 2nd edn. John Wiley & Sons, Inc. (2004)
7. Gligoric, M., Gvero, T., Jagannath, V., Khurshid, S., Kuncak, V., Marinov, D.: Test Generation through Programming in UDITA. In: Proc. of Intl. Conference on Software Engineering ICSE 2010. ACM Press (2010)
8. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2006)
9. Jagannath, V., Lee, Y.Y., Daniel, B., Marinov, D.: Reducing the Costs of Bounded-Exhaustive Testing. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 171–185. Springer, Heidelberg (2009)
10. Khurshid, S., Marinov, D.: TestEra: Specification-Based Testing of Java Programs Using SAT. Automated Software Engineering 11(4) (2004)
11. Liskov, B., Guttag, J.: Program Development in Java: Abstraction, Specification and Object-Oriented Design. Addison-Wesley (2000)
12. Ma, Y.-S., Offutt, J., Kwon, Y.-R.: MuJava: An Automated Class Mutation System. Journal of Software Testing, Verification and Reliability 15(2) (2005)
13. Milicevic, A., Misailovic, S., Marinov, D., Khurshid, S.: Korat: A Tool for Generating Structurally Complex Test Inputs. In: Proc. of Intl. Conference on Software Engineering ICSE 2007. IEEE Press (2007)
14. MuJava, `http://www.cs.gmu.edu/~offutt/mujava/`
15. Roops, `http://code.google.com/p/roops/`
16. Rothermel, G., Elbaum, S., Malishevsky, A., Kallakuri, P., Davia, B.: The Impact of Test Suite Granularity on the Cost-Effectiveness of Regression Testing. In: Proc. of Intl. Conference on Software Engineering ICSE 2002. ACM Press (2002)
17. Sullivan, K., Yang, J., Coppit, D., Khurshud, S., Jackson, D.: Software Assurance by Bounded Exhaustive Testing. In: Proc. of Intl. Symposium on Software Testing and Analysis ISSTA 2004. ACM Press (2004)
18. Yu, Y., Jones, J., Harrold, M.: An Empirical Study of the Effects of Test-Suite Reduction on Fault Localization. In: Proc. of Intl. Conference on Software Engineering ICSE 2008. ACM Press (2008)
19. Zhu, H., Hall, P., May, J.: Software Unit Test Coverage and Adequacy. ACM Computing Surveys 29(4) (1997)